Скъпи читатели,

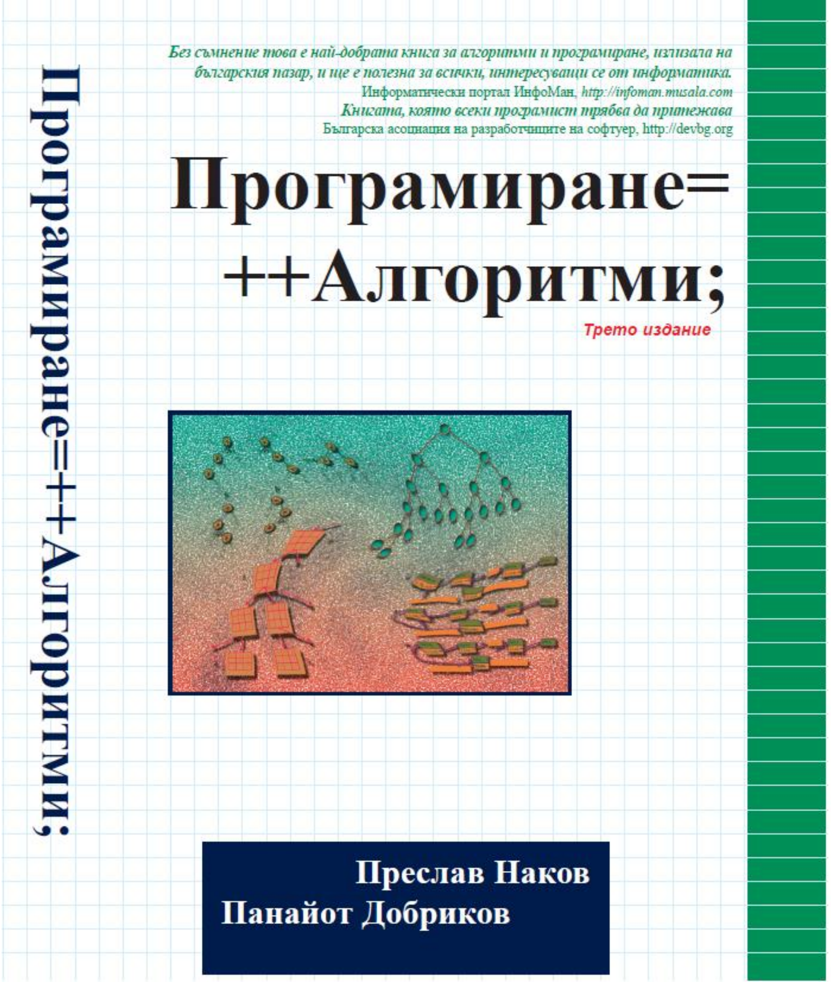
Поради постоянното изчерпване на книгата и трудностите с намирането ѝ в мрежата за разпространение решихме да ви направим малък „подарък“, като публикуваме книгата безплатно в електронен вид (лицензът за рапространение се намира на страница 2).

С пожелание за много успехи,
Панайот и Преслав

10 февруари 2013

За контакти с нас (авторите):

- Уеб сайт на книгата: http://www.programirane.org
- Facebook група: http://www.facebook.com/groups/168112146541301



©Автори: Преслав Наков и Панайот Добриков, 2012.

Всички запазени марки, използвани в тази книга, са собственост на техните притежатели.
Всички права запазени. Никоя част от съдържанието на тази книга не може да бъде репродуцирана или предавана под каквато и да е форма или по какъвто и да е повод без писменото съгласие на авторите.
Авторите на книгата „Програмиране = ++Алгоритми;“ Преслав Наков и Панайот Добриков ви предоставят заедно с настоящия pdf файл еднократен, безсрочен, персонален, нетрансферуем лиценз за ползване на книгата за лични и некомерсиални цели. В допълнение настоящият лиценз дава право на преподаватели в български училища и университети да използват същата в процеса на обучение, например като учебник или за подготовка на учебни материали. Настоящият лиценз не позволява книгата да се разпространява допълнително под каквато и да е форма, както и да се променя съдържанието на този pdf файл.

Програмиране=++Алгоритми;

Книгата е оригинално българско творение, неотстъпващо от световното ниво в разглежданата бързо развиваща се съвременна област на компютърната информатика.
Емил Келеведжиев, Институт по математика и информатика, Българска академия на науките

За създаването на качествени програми не е достатъчно перфектното владеене на един език за програмиране. Без сериозни познания в областта на алгоритмите не е възможно да се направи ефективна програма.
доц. Красимир Манев, Факултет по Математика и Информатика, СУ "Св. Климент Охридски", Американски университет - Благоевград

Книгата е оригинално българско творение, неотстъпващо от световното ниво в разглежданата бързо развиваща се съвременна област на компютърната информатика. Тя съчетава програмистка практика с теория, изградена върху математически методи, което допринася за по-добро разбиране и прилагане на многобройните алгоритми, съдържащи се в нея. Предназначена е за читатели, които биха оценили този синтез — ученици и техните учители, студенти и техните преподаватели, професионални програмисти и техните ръководители и разбира се, тя е за всички любители.

Книгата може да служи и като превъзходен университетски курс за въведение в алгоритмите и структурите от данни. Може да се каже, че научното и педагогическото ѝ ниво е значително. Всъщност тя е експериментирана от авторите ѝ в курса "Проектиране и анализ на компютърни алгоритми" в СУ "Св. Климент Охридски".

Забележимата разлика с повечето университетски учебници и ръководства по алгоритми и структури от данни е, че авторите използват подход "отдолу-нагоре", тръгвайки от самото програмиране, за да стигнат до теорията. Това обяснява и значителното присъствие в книгата на цялостно завършени, елегантно оформени програми с изходен текст на езика Си.

За начинаещия читател то може да служи като пътеводител в една обширна област, а за запознатия с тази област опитен програмист то има качества на справочник.

Емил Келеведжиев, Институт по математика и информатика, Българска академия на науките

За създаването на качествени програми не е достатъчно перфектното владеене на един език за програмиране. Без сериозни познания в областта на алгоритмите не е възможно да се направи ефективна програма, особено когато се работи с големи обеми от данни. За съжаление, в достъпната за българския читател литература съществува празнота в това отношение. Наличните текстове са малко и не покриват достатъчно добре темата.

Книгата, която държите в ръцете си, е следващо, по-мъдро и по-систематично, усилие след двутомното издание на единия от авторите - Преслав Наков - добре познато на всички, които в последните години са участвали в състезания по програмиране. И Преслав Наков, и Панайот Добриков са дългогодишни участници в състезания по програмиране. Отначало за ученици, а по-късно и за студенти. И двамата имат огромен опит в решаване на програмистки задачи с определено алгоритмичен характер. А многообразието от алгоритми и алгоритмични техники, много решени примери и много задачи за самостоятелна работа, са основни характеристики на настоящата книга.

Разбира се, никое издание, колкото и обемисто да е то, не може да обхване абсолютно всичко, което е направено от човечеството в областта на алгоритмите, но работата, която двамата автори са свършили, е чудесна.

доц. Красимир Манев, Факултет по Математика и Информатика, СУ "Св. Климент Охридски", Американски университет - Благоевград

Неслучайно авторите на книгата са поставили за нейно заглавие "Програмиране = ++Алгоритми;". Макар такова опростяване все повече да губи своята актуалност в съвременното програмиране, изборът на алгоритми (и структури от данни) продължава да оказва значително влияние върху ефективността на почти всяка програма.

Основният акцент в настоящата книга е поставен върху проектирането и анализа на компютърни алгоритми, независимо от конкретния език за програмиране, макар за реализацията им да е избран Си. Предложено е задълбочено и изчерпателно изложение, което се възприема лесно от читателя; едно наистина рядко съчетание.

Не може да не се подчертае, че авторите са сред най-изявените състезатели по програмиране у нас и, макар и млади, имат значителен практически опит във водещи софтуерни фирми. С тази книга на читателя се дава възможност да се запознае с важни и фини програмистки техники.

доц. Асен Рахнев, Пловдивски Университет

Съдържание

СЪДЪРЖАНИЕ **3**
ПРЕДГОВОР ОТ НАУЧНИЯ РЕДАКТОР **13**
ПРЕДГОВОР КЪМ ТРЕТОТО ИЗДАНИЕ **15**
***ГЛАВА 0* КОМПЮТЪРНА ИНФОРМАТИКА, АЛГОРИТМИ, ПРОГРАМИРАНЕ** **17**
0.1. ПЕРСПЕКТИВНИ НАПРАВЛЕНИЯ В КОМПЮТЪРНАТА ИНФОРМАТИКА 18
0.1.1. Връзка компютър-потребител 18
0.1.2. Компютърни комуникации и сигурност 19
0.1.3. Новаторски компютърни архитектури 20
0.1.4. Моделиране на сложни феномени 20
0.1.5. Изкуствен интелект 21
0.1.6. Нетрадиционни методи за изчисление 21
0.1.7. Фундаментална компютърна информатика 22
0.2. ПОНЯТИЕТО ЛЕЙМЪР 23
0.3. РАЗВИТИЕ НА АЛГОРИТМИТЕ 24
0.3.1. Произход на думата "алгоритъм" 24
0.3.2. Алгоритмите през вековете 25
0.3.3. Анализ на алгоритмите 25
0.3.4. Изчислимост 26
0.4. ПРОГРАМИРАНЕ = ++АЛГОРИТМИ 26
0.4.1. За кого е предназначена книгата 26
0.4.2. Кратко представяне на съдържанието 27
0.4.3. Защо Си? 28
0.4.4. Конвенция на изходните текстове 29
0.5. "СЛЕДГОВОР" 29
Посвещение 29
0.5.1. Благодарности 29
0.5.2. Търсене на грешки 30
0.5.3. Коментари и въпроси 30
ГЛАВА 1 ВЪВЕДЕНИЕ В АЛГОРИТМИТЕ **33**
1.1. ОСНОВНИ МАТЕМАТИЧЕСКИ ПОНЯТИЯ И АЛГОРИТМИ 33
1.1.1. Основни математически понятия 33
- Множества 33
- Числа 35
- Целочислено деление и остатък. Брой цифри на естествено число 37
- Сума и произведение 38
- Степен, логаритъм, n-ти корен 39
- Факториел. Рекурентни функции 40
- Матрици 41
1.1.2. Намиране броя на цифрите на произведение 43
1.1.3. Прости числа 44
- Проверка дали дадено число е просто 45
- Решето на Ератостен. Търсене на прости числа в интервал 47
- Разлагане на число на прости делители 50
- Намиране на броя на нулите, на които завършва произведение 51

1.1.4. Мерсенови и съвършени числа.....52
- Мерсенови числа.....52
- Съвършени числа. "Големи" числа.....54
1.1.5. Биномни коефициенти, триъгълник на Паскал. Факторизация.....56
1.1.6. Бройни системи и преобразуване.....59
- Преминаване от десетична в р-ична бройна система.....61
- Преминаване от р-ична в десетична бройна система. Формула на Хорнер.....63
1.1.7. Римски цифри.....65
- Представяне на десетично число с римски цифри.....65
- Преобразуване на римско число в десетично.....66
1.2. РЕКУРСИЯ И ИТЕРАЦИЯ.....67
1.2.1. Факториел.....68
1.2.2. Редица на Фибоначи.....69
1.2.3. Най-голям общ делител. Алгоритъм на Евклид.....74
1.2.4. Най-малко общо кратно.....76
1.2.5. Връщане от рекурсията и използване на променливите.....77
1.3. ОСНОВНИ КОМБИНАТОРНИ АЛГОРИТМИ.....80
1.3.1. Пермутации.....80
- Генериране.....81
- Кодиране и декодиране.....84
- Пермутации с повторение.....86
1.3.2. Вариации.....86
- Видове, генериране.....86
- Сума нула.....88
1.3.3. Комбинации.....90
1.3.4. Разбиване на числа.....92
- Генериране на всички разбивания на число като сума от дадени числа.....92
- Генериране на всички разбивания на число като произведение от числа.....93
- Генериране на всички разбивания на число като сума от дадени числа.....94
1.3.5. Разбиване на множества.....96
- Числа на Бел и Стирлинг.....96
1.4. ОЦЕНКА И СЛОЖНОСТ НА АЛГОРИТМИ.....97
1.4.1. Размер на входните данни.....99
1.4.2. Асимптотична нотация.....99
1.4.3. O(F): Свойства и примери.....100
1.4.4. Ω(F): Свойства и примери.....102
1.4.5. Θ(F): Свойства и примери.....103
1.4.6. Асимптотични функции и реални числа.....105
1.4.7. Нарастване на основните функции.....106
1.4.8. Определяне на сложност на алгоритъм.....107
- Елементарна операция.....107
- Последователност от оператори.....107
- Композиция на оператори.....107
- if-конструкции.....107
- Цикли.....108
- Вложени цикли.....108
- Още примери от вложени цикли.....109
- Логаритмична сложност.....110
- Рекурсия.....110
1.4.9. Характеристични уравнения.....112
- Линейни хомогенни уравнения с прости корени.....112

- Линейни хомогенни уравнения с кратни корени 114
- Линейни нехомогенни уравнения 114
1.4.10. Специални техники за анализ на алгоритми 117
- Използване на барометър 117
- Амортизационен анализ 118
- Основна теорема 118
1.4.11. Проблеми на асимптотичната нотация 120
1.5. Въпроси и задачи 120
1.5.1. Задачи от текста 120
1.5.2. Други задачи 133
- Задачи от числа, редици, функции 133
- Задачи от матрици и общи задачи 137
- Комбинаторни задачи 138
ГЛАВА 2 ВЪВЕДЕНИЕ В СТРУКТУРИТЕ ОТ ДАННИ 143
2.1. Списък, стек, опашка 144
- Стек 145
- Опашка 146
- Дек 147
2.1.1 Последователна (статична) реализация 147
- Стек 147
- Опашка 149
2.1.2 Свързана (динамична) реализация 151
- Включване на елемент 152
- Обхождане на списък 153
- Включване след елемент, сочен от даден указател 153
- Включване пред елемент, сочен от даден указател 154
- Изтриване по зададен ключ и указател към начало на списъка 154
- Изтриване на елемент, сочен от даден указател 155
2.2. Двоични дървета 158
- Търсене по даден ключ 163
- Включване на нов връх 164
- Изключване на връх по даден ключ 164
- Обхождане 168
2.3. Балансирани дървета 169
2.3.1. Ротация. Червено-черни дървета 172
2.3.2. B-дървета 174
2.4. Хеш-таблици 176
- Хеш-функция 176
- Колизии 177
2.4.1. Класически хеш-функции 178
- Остатък при деление с размера на таблицата 178
- Мултипликативно хеширане 178
- Хеш-функции върху части от ключа 179
- Сравнение на някои от разгледаните хеш-функции 179
- Хеш-функции върху символни низове 180
2.4.2. Справяне с колизии 184
- Затворено хеширане 184
- Линейно пробване 184
- Квадратично пробване 185
- Двойно хеширане 185
- Отворено хеширане 185

- Допълнителна памет за колизии.....185
- Списък на препълванията.....186
2.4.3. Реализации на хеш-таблица.....186
2.5. Въпроси и задачи.....192
2.5.1. Задачи от текста.....192
2.5.2. Други задачи.....196
ГЛАВА 3 СОРТИРАНЕ.....199
3.1. Сортиране чрез сравнение.....200
3.1.1. Дърво на сравненията.....200
3.1.2. Сортиране чрез вмъкване.....201
3.1.3. Сортиране чрез вмъкване с намаляваща стъпка. Алгоритъм на Шел.....203
3.1.4. Метод на мехурчето.....206
3.1.5. Сортиране чрез клатене.....207
3.1.6. Бързо сортиране на Хоор.....208
3.1.7. Метод на "зайците и костенурките".....213
3.1.8. Сортиране чрез пряк избор.....215
3.1.9. Пирамидално сортиране на Уилямс.....216
3.1.10. Минимална времева сложност на сортирането чрез сравнение.....219
3.2. Сортиране чрез трансформация.....221
3.2.1. Сортиране чрез множество.....221
3.2.2. Сортиране чрез броене.....223
3.2.3. Побитово сортиране.....226
3.2.4. Метод на бройните системи.....229
3.2.5. Сортиране чрез пермутация.....232
3.3. Паралелно сортиране.....233
3.3.1. Принцип на нулите и единиците.....235
3.3.2. Битонични последователности.....236
3.3.3. "Изчисти наполовина".....237
3.3.4. Сортиране на битонична последователност.....237
3.3.5. Сливаща схема.....237
3.3.6. Сортираща схема.....238
3.3.7. Транспозиционна сортираща схема.....238
3.3.8. Четно-нечетна сливаща схема на Бетчер.....239
3.3.9. Четно-нечетна сортираща схема.....239
3.3.10. Пермутационна схема.....239
3.4. Въпроси и задачи.....240
3.4.1. Задачи от текста.....240
3.4.2. Други задачи.....243
ГЛАВА 4 *ТЪРСЕНЕ*.....245
4.1. Последователно търсене.....246
4.1.1. Последователно търсене в сортиран списък.....248
4.1.2. Последователно търсене с преподреждане.....249
4.2. Търсене със стъпка. Квадратично търсене.....251
4.3. Двоично търсене.....252
4.4. Фибоначиево търсене.....256
4.5. Интерполационно търсене.....258
4.6. Въпроси и задачи.....260
4.6.1. Задачи от текста.....260

4.6.2. Други задачи ... *261*
ГЛАВА 5 ТЕОРИЯ НА ГРАФИТЕ ... 263
5.1. ОСНОВНИ ПОНЯТИЯ ... 263
5.2. ПРЕДСТАВЯНЕ И ПРОСТИ ОПЕРАЦИИ С ГРАФ ... 267
5.2.1. Списък на ребрата ... *267*
5.2.2. Матрица на съседство, матрица на теглата ... *268*
5.2.3. Списък на наследниците (списък на инцидентност) ... *268*
5.2.4. Матрица на инцидентност между върхове и ребра ... *269*
5.2.5. Компоненти на свързаност ... *269*
5.2.6. Построяване и прости операции с графи ... *270*
5.3. ОБХОЖДАНЕ НА ГРАФ ... 271
5.3.1. Обхождане в ширина ... *271*
5.3.2. Обхождане в дълбочина ... *274*
5.4. ОПТИМАЛНИ ПЪТИЩА, ЦИКЛИ И ПОТОЦИ В ГРАФ ... 276
5.4.1. Директни приложения на алгоритмите за обхождане ... *277*
- най-кратък път между два върха по брой на върховете ... 277
- проверка дали граф е цикличен ... 279
- намиране на всички прости пътища между два върха ... 281
5.4.2. Екстремален път в граф ... *283*
- неравенство на триъгълника ... 284
- алгоритъм на Форд-Белман ... 284
- алгоритъм на Флойд ... 285
- обобщен алгоритъм на Флойд ... 288
- алгоритъм на Дейкстра ... 290
- повдигане в степен на матрицата на съседство ... 293
- алгоритъм на Уоршал и матрица на достижимост ... 293
- най-дълъг път в ацикличен граф ... 294
- най-дълъг прост път между два върха в произволен граф ... 297
5.4.3. Цикли ... *297*
- намиране на фундаментално множество от цикли ... 297
- минимален цикъл през връх ... 300
5.4.4. Хамилтонови цикли. Задача за търговския пътник ... *300*
5.4.5. Ойлерови цикли ... *303*
5.4.6. Потоци ... *306*
- Максимален поток ... 307
- повече от един източник и консуматор ... 311
- капацитет на върховете ... 312
5.5.ТРАНЗИТИВНОСТ И ПОСТРОЯВАНЕ. ТОПОЛОГИЧНО СОРТИРАНЕ ... 312
5.5.1. Транзитивно затваряне. Алгоритъм на Уоршал ... *313*
5.5.2. Транзитивно ориентиране ... *314*
5.5.3. Транзитивна редукция ... *317*
5.5.4. Контрол на компании ... *318*
5.5.5. Топологично сортиране ... *320*
5.5.6. Пълно топологично сортиране ... *322*
5.5.7. Допълване на ацикличен граф до слабо свързан ... *324*
5.5.8. Построяване на граф по дадени степени на върховете ... *325*
5.6. ДОСТИЖИМОСТ И СВЪРЗАНОСТ ... 326
5.6.1. Компоненти на свързаност ... *326*
5.6.2. Компоненти на силна свързаност в ориентиран граф ... *328*
5.6.3. Разделящи точки в неориентиран граф. Двусвързаност ... *330*

5.6.4. k-свързаност на неориентиран граф......333
5.7.ОПТИМАЛНИ ПОДМНОЖЕСТВА И ЦЕНТРОВЕ......334
5.7.1. Минимално покриващо дърво......334
- алгоритъм на Крускал......334
- алгоритъм на Прим......338
- частично минимално покриващо дърво......340
5.7.2. Независими множества......340
- максимални независими множества......341
5.7.3. Доминиращи множества......343
5.7.4. База......346
5.7.5. Център, радиус и диаметър......348
- p-център и p-радиус......350
5.7.6. Двойкосъчетание. Максимално двойкосъчетание......353
5.8. ОЦВЕТЯВАНИЯ И ПЛАНАРНОСТ......354
5.8.1. Оцветяване на граф. Хроматично число......354
- ограничаване отдолу на хроматичното число......355
- намиране на върховото хроматичното число......355
5.8.2. Планарност на графи......355
5.9. ВЪПРОСИ И ЗАДАЧИ......357
5.9.1. Задачи от текста......357
5.9.2. Други задачи......363
ГЛАВА 6 ТЪРСЕНЕ С ВРЪЩАНЕ. NP-ПЪЛНИ ЗАДАЧИ......369
6.1. КЛАСИФИКАЦИЯ НА ЗАДАЧИТЕ......369
6.1.1. Сложност по време......369
6.1.2. Сложност по памет......370
6.1.3. Нерешими задачи......370
6.1.4. Примери......370
6.2. NP-ПЪЛНИ ЗАДАЧИ......373
6.3. ТЪРСЕНЕ С ВРЪЩАНЕ......374
6.3.1. Удовлетворимост на булева функция......376
6.3.2. Оцветяване на граф......380
6.3.3. Най-дълъг прост път в цикличен граф......383
6.3.4. Разходка на коня......385
6.3.5. Задача за осемте царици......388
6.3.6. Разписание на училищна програма......391
6.3.7. Побуквено превеждане......395
6.4. МЕТОД НА РАЗКЛОНЕНИЯТА И ГРАНИЦИТЕ......398
6.4.1. Задача за раницата (оптимална селекция)......398
6.5. ОПТИМАЛНИ СТРАТЕГИИ ПРИ ИГРИ......401
6.5.1. Игра "X"-чета и "O"......402
6.5.2. Принцип на минимума и максимума......405
6.5.3. Алфа-бета отсичане......406
6.5.4. Алфа-бета изследване до определена дълбочина......408
6.6. ВЪПРОСИ И ЗАДАЧИ......409
6.6.1. Задачи от текста......409
6.6.2. Други задачи......413
- NP-пълни задачи......413
- Изчерпващи задачи......425
ГЛАВА 7 РАЗДЕЛЯЙ И ВЛАДЕЙ......427

7.1. НАМИРАНЕ НА К-ИЯ ПО ГОЛЕМИНА ЕЛЕМЕНТ....................427
7.2. МАЖОРАНТ....................434
7.3. СЛИВАНЕ НА СОРТИРАНИ МАСИВИ....................445
7.4. СОРТИРАНЕ ЧРЕЗ СЛИВАНЕ....................450
7.5. БЪРЗО ПОВДИГАНЕ В СТЕПЕН....................456
7.6. АЛГОРИТЪМ НА ЩРАСЕН ЗА БЪРЗО УМНОЖЕНИЕ НА МАТРИЦИ....................458
7.7. БЪРЗО УМНОЖЕНИЕ НА ДЪЛГИ ЧИСЛА....................461
7.8. ЗАДАЧА ЗА ХАНОЙСКИТЕ КУЛИ....................464
7.9. ОРГАНИЗИРАНЕ НА ПЪРВЕНСТВА....................466
7.10. ЦИКЛИЧНО ИЗМЕСТВАНЕ НА ЕЛЕМЕНТИТЕ НА МАСИВ....................471
7.11. ПОКРИВАНЕ С ШАБЛОН....................474
7.12. ВЪПРОСИ И ЗАДАЧИ....................475
7.12.1. Задачи от текста....................475
7.12.2. Други задачи....................478
ГЛАВА 8 ДИНАМИЧНО ОПТИМИРАНЕ....................481
8.1.ВЪВЕДЕНИЕ....................481
8.2.КЛАСИЧЕСКИ ОПТИМИЗАЦИОННИ ЗАДАЧИ....................484
8.2.1. Задача за раницата....................484
8.2.2. Братска подялба....................495
8.2.3. Умножение на матрици....................497
8.2.4. Триангулация на многоъгълник. Числа на Каталан....................504
8.2.5. Оптимално двоично дърво за претърсване....................508
8.2.6. Най-дълга обща подредица....................513
8.2.7. Най-дълга ненамаляваща подредица....................517
8.2.8. Сравнение на символни низове....................522
8.2.9. Задача за разделянето....................526
8.3.НЕОПТИМИЗАЦИОННИ ЗАДАЧИ....................528
8.3.1. Числа на Фибоначи....................528
8.3.2. Биномни коефициенти....................532
8.3.3. Спортни срещи....................533
8.3.4. Представяне на сума с неограничен брой монети....................537
8.3.5. Представяне на сума с ограничен брой монети....................539
8.3.6. Разбиване на естествено число....................540
8.3.7. Числа без две съседни нули....................542
8.3.8. Разпознаване на контекстно свободен език....................543
8.3.9. Хедонийски език....................548
8.3.10. Символно умножение....................549
8.4. ДРУГИ ИНТЕРЕСНИ ОПТИМИЗАЦИОННИ ЗАДАЧИ....................551
8.4.1. Такси компания....................552
8.4.2. Билети за влак....................553
8.4.3. Числов триъгълник....................555
8.4.4. Представяне на сума с минимален брой монети....................559
8.4.5. Опроводяване на платка....................561
8.4.6. На опашка за билети....................564
8.4.7. Разпределение на ресурси....................565
8.4.8. Семинарна зала....................568
8.4.9. Крайпътни дървета....................571
8.4.10. Разрязване на материали....................574

8.4.11. Зациклен израз......576
8.4.12. Домино-редица......579
8.4.13. Трионообразна редица......582
8.5. Въпроси и задачи......584
8.5.1. Задачи от текста......584
8.5.2. Други задачи......591
ГЛАВА 9 ЕВРИСТИЧНИ И ВЕРОЯТНОСТНИ АЛГОРИТМИ......595
9.1. Алчни алгоритми......595
9.1.1. Египетски дроби......596
9.1.2. Максимално съчетание на дейности......598
9.1.3. Минимално оцветяване на граф и дърво......601
9.1.4. Алгоритми на Прим и Крускал......604
9.1.5. Дробна задача за раницата......605
9.1.6. Задача за магнитната лента......607
9.1.7. Процесорно разписание......608
9.1.8. Разходката на коня. Хиперкуб. Код на Грей......610
9.2. Търсене с налучкване......616
9.2.1. Алгоритми Монте Карло и Лас Вегас......617
- проверка дали число е просто......618
9.2.2. Числени алгоритми с приближение......621
9.2.3. Генетични алгоритми......622
9.3. Достигане на фиксирано приближение......627
9.3.1. Върхово покритие на граф......628
9.4. Въпроси и задачи......629
9.4.1. Задачи от текста......629
9.4.2. Други задачи......633
ГЛАВА 10 КОМПРЕСИРАНЕ......637
10.1. Кодиране......637
10.2. Обща класификация......638
10.3. Кодиране на последователности......640
10.3.1. Премахване на нулите......640
10.3.2. Компресиране с битови карти......641
10.3.3. Полубайтово пакетиране......643
10.3.4. Съвместно използване на битови карти и полубайтово пакетиране......646
10.3.5. Двуатомно кодиране......646
10.3.6. Замяна на шаблони......648
10.3.7. Относително кодиране......649
10.3.8. Математическо очакване. Кодиране с линейно предсказване......650
10.3.9. Кодиране на последователности......653
10.3.10. Един представителен пример: PackBits......655
10.4. Статистически методи......656
10.4.1. Алгоритъм на Шенън-Фано......656
10.4.2. Алгоритъм на Хъфман......660
10.4.3. Обобщен алгоритъм на Хъфман......671
10.4.4. Код с разделители......672
10.4.5. Аритметично кодиране......673
10.5. Адаптивно компресиране......681
10.5.1. Адаптивно компресиране по Хъфман......683

10.5.2. Модели на Марков......*686*
10.5.3. Един представителен пример: MNP-5......*688*
10.6. Речниково кодиране 690
10.6.1. Ентропия *691*
10.6.2. Малко история *695*
10.6.3. Стандарти и патенти *696*
10.6.4. Статични срещу адаптивни методи *697*
10.6.5. LZ77. Компресиране с плъзгащ се прозорец......*698*
10.6.6. LZSS. Едно подобрение......*698*
10.6.7. FLZ. Друг вариант на LZ77 ......*699*
10.6.8. LZW. Модификацията на Уелч......*700*
10.6.9. GIF. Гледната точка на CompuServe......*706*
10.6.10. Оптимални срещу алчни алгоритми *707*
10.6.11. Компресиране в реално време......*708*
10.6.12. LZW срещу Марков *709*
10.7. Компресиране със загуба......710
10.7.1. Изрязване и квантифициране......*710*
10.7.2. JPEG......*711*
10.7.3. Компресиране на видеоизображение. MPEG......*713*
10.7.4. Уейвлети *715*
10.7.5. Компресиране с фрактали *716*
10.8. Въпроси и задачи......717
10.8.1. Задачи от текста......*717*
10.8.2. Други задачи......*725*
ЛИТЕРАТУРА 727
ПРЕДМЕТЕН УКАЗАТЕЛ......733

Предговор от научния редактор

С настоящата книга се продължава уникалното явление в нашата учебна и научна литература (а, може би и в световната) — студенти да пишат оригинални учебници за компютърната информатика на изключително високо ниво, новаторски съчетавайки практика и теория. Първата книга, започнала този стил, беше "Основи на компютърните алгоритми" на Преслав Наков. Сега авторът ѝ заедно с Панайот Добриков правят следваща стъпка, като предлагат на читателя значително разширен и видоизменен труд, който по качества се доближава до такава основна книга за съставители на алгоритми, каквато е "библията" *Introduction to Algorithms* от Cormen, Leiserson и Rivest, но притежаващ и много свои специфични черти.

Забележимата разлика с повечето университетски учебници и ръководства по алгоритми и структури от данни е, че авторите използват подход "отдолу-нагоре", тръгвайки от самото програмиране, за да стигнат до теорията. И това е естествено за тях, защото и двамата са навлезли в компютърната информатика чрез състезанията по програмиране и са преминали успешно през многобройните кръгове на националната и международната олимпиади по информатика. Това обяснява и значителното присъствие в книгата на цялостно завършени, елегантно оформени програми с изходен текст на езика Си. Те са написани от авторите и носят отпечатъка им на елитни алгоритмисти и програмисти-практици.

Оттук следва и основното предназначение на книгата — да бъде учебник за състезатели — ученици и студенти. Разбира се, това никак не изключва възможността тя да бъде ползвана от професионални програмисти, на които е възложено или им се е наложило да програмират нещо, което надхвърля леймърското ниво (за "леймър" — виж въведението на авторите) — т.е. нещо, за което се изисква да се помисли предварително и да се потърси теория, преди да се започне с програмирането.

Книгата може да служи и като превъзходен университетски курс за въведение в алгоритмите и структурите от данни. Може да се каже, че научното и педагогическото ѝ ниво е значително. Всъщност тя е експериментирана от авторите ѝ именно за такъв курс, защото са я ползвали, когато е била в още незавършения си вид, за да преподават по нея. Така предложеният учебник има достойнствата, че споделя едновременно опита на доскорошни активни състезатели по информатика с по-новия им опит да преподават на студенти основите на компютърната информатика. Авторите са водили курса си "Проектиране и анализ на компютърни алгоритми", когато самите те са били студенти и по всеобщо признание този курс, който е тясно свързан с настоящата книга, е оценен много високо.

В книгата са застъпени няколкото основни линии, по които могат да се разглеждат и преподават алгоритми. Въпреки, че е трудно да се правят методологически класификации, може да приемем най-общо, че двете възможни гледни точки са — от позицията на предмета и от позицията на метода. Традиционно към предмета, върху който работят алгоритмите, се причисляват обекти като числа, множества и по-общи структури от данни, графи и т.н., а към предмета на самите алгоритми могат да бъдат отнесени темите за сортиране и търсене, компресиране, получаване на комбинаторни конфигурации и т.н., докато методите за съставяне на алгоритми са рекурсия и итерация, търсене с връщане, разделяй и владей, динамично оптимиране, евристични алгоритми и др.

Подробното съдържание на книгата дава много добра представа за включения материал. За начинаещия читател то може да служи като пътеводител в една обширна област, а за запознатия с тази област опитен програмист то има качества на справочник.

Книгата е оригинално българско творение, неотстъпващо от световното ниво в разглежданата бързо развиваща се съвременна област на компютърната информатика. Тя съчетава програмистка практика с теория, изградена върху математически методи, което допринася за по-добро разбиране и прилагане на многобройните алгоритми, съдържащи се в нея. Предназначена е за читатели, които биха оценили този синтез — ученици и техните учители, студенти и техните преподаватели, професионални програмисти и техните ръководители и разбира се, тя е за всички любители.

1 Август 2002 Емил Келеведжиев

Институт по математика и информатика
Българска академия на науките

Предговор към третото издание

Скъпи читатели,

Поради големия интерес и изчерпване на предишните два тиража се наложи поредното трето издание на настоящата книга. Разбира се, всяко преиздаване ни дава възможност за:

- подобрения и промени, продиктувани от развитието в различни алгоритмични области;
- корекции на грешки, които открихме сами или получихме по e-mail и на Интернет страницата на книгата.

Бихме искали да благодарим на всички читатели, които се свързаха с нас (или използваха форума на Интернет страницата) – както за топлите и приятелски писма, така и за препоръките и откритите *грешки* и пропуски.

За някои от най-активните сме приготвили специален подарък, с който искаме да изразим своята признателност за показаната съпричастност – "Авторско издание" с различна (и твърда) корица и автограф от авторите.

Благодарим Ви!

Имаме удоволствието да ви съобщим, че в периода от първото издание досега, няколко големи университета и средни училища избраха книгата като основен учебник или допълнителна литература в курсове по алгоритми и структури от данни. Бихме се радвали да помогнем, например с:

- възможност за отстъпки при закупуване на книгата от студентите от курса;
- осигуряване на контакт с други преподаватели, водещи подобни курсове;
- помощ при подбор и организиране на подходящ учебен материал;
- други.

Пишете ни! Вашите идеи, препоръки и коментари са изключително ценни за нас!

С пожелание за много успехи,
Панайот и Преслав

3 януари 2005 г.

Глава 0
Компютърна информатика, алгоритми, програмиране

"Think of all the psychic energy expended in seeking a fundamental distinction between 'algorithm' and 'program'".

~ Epigrams in Programming

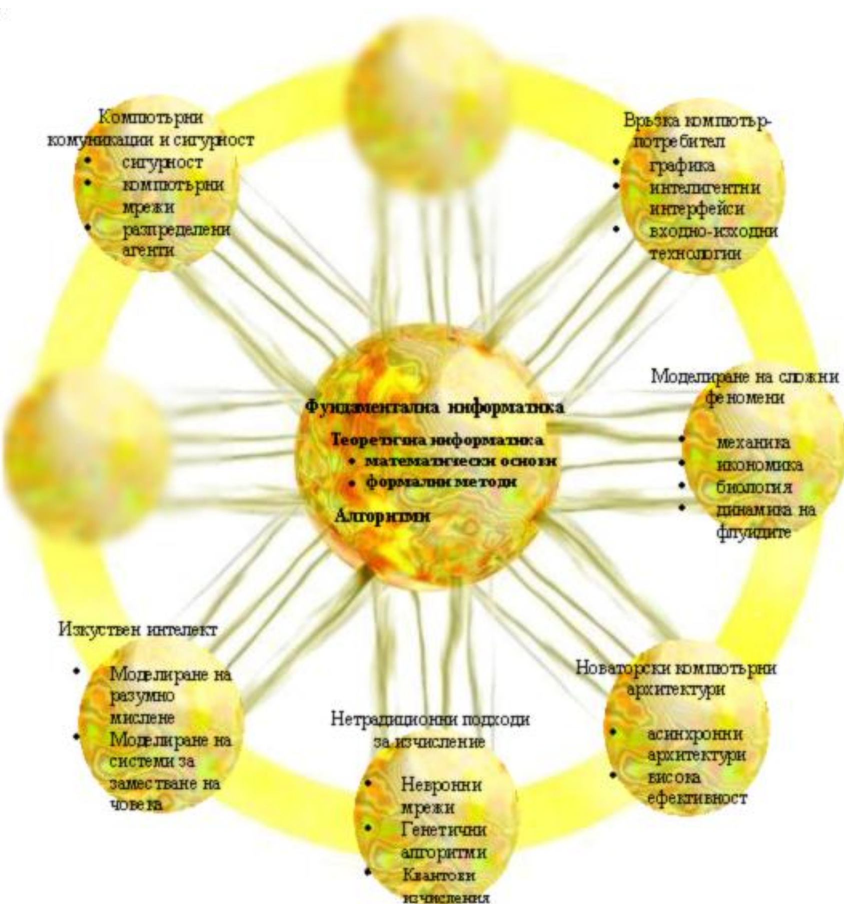
Според една добре известна теория [*Adams-1980*] планетата Земя представлява един огромен компютър, построен с единствената цел да разкрие въпроса за "смисъла на живота, Вселената и всичко останало". Тази щура мисъл, родила се в главата на Дъглас Адамс още през 1975 (докато се е излежавал в тръстиката), може би не е най-точното описание на състоянието на нещата, но, както и други негови "хипотези", проектира една обща допирателна между минало, настояще и бъдеще. Ще отбележим, че построяването на обща допирателна през повече от два, и то не толкова закръглени, обекта, каквито са състоянията на времето, е една доста трудна и често нерешима задача.

Трудно е да се избегнат клишетата, когато трябва да се описва ефектът, който компютърните системи и технологии оказват върху съвременното общество. Забележителните им темпове на развитие оставят дълбока следа в организационните и оперативните процеси в индустрията, бизнеса, управлението и обучението. Целта на настоящата уводна глава не е разсъждения върху това как след 10 години компютрите ще се хранят и [*цензурирано*, *ред.*] вместо нас, а да запознае накратко читателя с аспектите на съвременните компютърни технологии, както и да го подготви за света на компютърните алгоритми, който ще изследваме на страниците на тази книга.

В основата на развитието на компютърните технологии стои *компютърната информатика* — термин, който многократно ще използваме по-нататък. Ще отбележим, че в английски език (особено в *американски* английски) се предпочита използването на термина *computer science*, който се превежда на български буквално като *компютърна наука*. В Европа има по-различно разбиране, например: в бившия социалистически блок (руски и български: *информатика*), във Франция (френски: *informatique*), Германия (немски: *informatik*) и др. Тук свободно циркулира и английското название *informatics* — вероятно първоначално появило се като буквален превод някъде извън Великобритания, но с тенденцията да се наложи. Въпреки това, *informatics* е силно дискриминирано: липсва в повечето речници на съвременния английски език, университетската специалност във Великобритания най-често се нарича *computer science* (макар напоследък да се използва и *информатика*), дори вградената в *Microsoft Word* стандартна програма за правописна корекция в момента систематично подчертава с червено тази дума. Въпреки това понятието постепенно се налага и на международно ниво: така например международната олимпиада е по *информатика* (англ. *International Olympiad in Informatics*), така е и с Балканиадата (англ. *Balkan Olympiad in Informatics*). В САЩ обаче продължават категорично да отричат правото му на съществуване... В основата на понятието *информатика* стои осъзнатият факт, че всъщност става въпрос за по-широкия процес на обработване на *информация*, докато от американска гледна точка е важна не *информацията*, а конкретното средство за обработката ѝ — *компютърът*.

0.1. Перспективни направления в компютърната информатика

Областите, в които има поле за по-нататъшно развитие и научно-изследователска дейност през следващото десетилетие, могат условно да бъдат разграничени както на *фигура 0.1*. (Защо са оставени празни полета? Какво би могло да стои там?)



Фигура 0.1. Основни области на компютърната информатика.

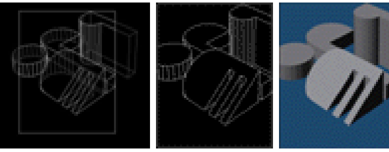
0.1.1. Връзка компютър-потребител

Когато се използва добре известното клише *"компютрите са навсякъде"*, в най-голяма степен се има предвид *връзката между компютър и потребител*. Макар за момента да не съществува общоприета дефиниция какво обхваща, това е една от най-важните области на компютърната информатика. Жизнено необходимо е по-нататъшното ѝ развитие, тъй като настоящите ограничения при това взаимодействие се чувстват осезателно. Съвсем естествено е желанието за много по-интерактивно взаимодействие с компютъра: да може да разбира печатен и ръкописен текст, човешка реч и жестове, както и да изразява свои собствени идеи в общуването си с нас. В

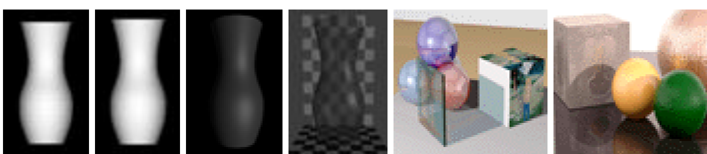
момента възможностите за подобно “общуване” са тромави, неудобни или най-малкото бавни. Въпреки някои опити за подобрения в тази насока засега практиката определя компютърните системи като “слепи” и “глухи”.

Компютърна графика

Може би най-разпространеното схващане за *компютърна графика* е: "почти всичко, свързано с компютрите, което не е текст и звук". На практика става въпрос за сериозна научна област, свързана с редица традиционни математически науки като: линейна алгебра, аналитична геометрия, диференциална геометрия, диференциални уравнения, (комплексен) анализ и други, както и с множество фундаментални компютърни алгоритми. Целите, поставени в тази насока, включват развитието и експлоатацията на графични хардуерни и софтуерни системи, както и съставянето на нови методи за *моделиране*, *рендериране*, *симулиране*, *визуализиране* и др. (*виж фигури 0.1.1а.*, *0.1.1б.* и *0.1.1в.*).



Фигура 0.1.1а. *Рендериране*: визуализиране на модели на обекти така, че да изглеждат триизмерни.



Фигура 0.1.1б. Някои от най-често използваните техники за засилване на реализма на компютърни графични обекти: прозрачност, отражение и сянка.

Фигура 0.1.1в. Проследяване на лъча, светлоусещане и светлоотражение.

0.1.2. Компютърни комуникации и сигурност

Компютърната комуникация е процес, при който се създават, разменят и получават данни и информация, като се използват компютърни системи, свързани в мрежа. Например взаимо-

действията между различни “локални” изчислителни процеси, компютърни конференции, електронни бюлетини, глобалната мрежа Интернет (във всички аспекти, включително прости неща като електронната поща). Основните проблеми тук са начинът на кодиране, предаване, транспортиране и декодиране на данните, изпращани по мрежата, абстракцията на предаваните съобщения (протоколи) и др. Разпределянето и организирането на информацията са обект на усилени изследвания, а едно от челните места заема въпросът за сигурността на предаваните данни.

Компютърна сигурност

В ранните дни на “компютърната ера”, поради ограничения достъп до компютърните системи, както и поради малкия им брой, проблемите, свързани със сигурността, почти не се разглеждат. Първите “пробиви” датират от средата на века, когато се поставя въпросът за поверителността и личната неприкосновеност на информацията. Малко по-късно Масачузетският технологичен институт, наред с множеството постижения в областта на компютърните системи и технологии, става известен и с първия *хакер*. Ще отбележим, че понятията *хакер* (от англ. *hacker*, *hacking*) и *кракер* (от англ. *cracker*, *cracking)* често се бъркат. Днес в значението на последното се влага смисъл на извършване на *незаконни* действия, докато за първото това не е задължително. От тази гледна точка (поверителност и лична неприкосновеност), под *сигурност на информацията* се разбират, грижите, полагани за *“опазването”* ѝ.

В настоящия момент защитата на компютърната информация е от първостепенно значение както по отношение на операционните системи, така и на целия компютърен софтуер.

Предпоставките за разрастване на проблема със сигурността са много — като се започне от използването на общи ресурси и се стигне до широкото разпространение и използване на компютърни системи в най-разнообразни сфери на живота. Вече става въпрос не само за пробиви в неприкосновеността на информацията, а за извършването на икономически, политически и други *престъпления* по компютърен път, дори за компютърен тероризъм.

0.1.3. Новаторски компютърни архитектури

През последните 10 години важно място в изследванията в областта на компютърната информатика заемат *компютърните архитектури* и в частност т.нар. *асинхронни архитектури*. Става въпрос за част от по-общия модел на Фон-Неймановата архитектура на *паралелните системи* (името на Джон Фон-Нойман, както ще стане дума по-нататък в текста, се свързва с първото формално описание на структурата на изчислителна система).

При паралелните системи определен, често доста голям, брой компютри или само процесори се свързват заедно и оперират като една голяма изчислителна машина. Съществува широк клас от задачи, свързани с този аспект на компютърната информатика: разработване на техники за програмиране в асинхронни среди, анализиране на ефективността на паралелни решения и др.

Други новаторски компютърни архитектури ще бъдат разгледани в последния параграф, отнасящ се за изкуственият интелект.

0.1.4. Моделиране на сложни феномени

Изследването на явления и структури от различни области на науката и живота, както и симулацията на реални функции, поведение и процеси от природата са изключително важна част от компютърната информатика. Компютърни игри като "Сим"-серията на *Maxis* и много други комерсиални продукти допринасят значително за популяризирането на областта. Напредъкът и развитието в областта на компютърното моделиране на процеси в химията, биологията, медицината и други науки неминуемо ще доведе до множество важни открития. Така например, в биологията специфични техники от компютърната информатика могат да се използват за изследване на преноса на информацията в клетките, централната нервна система и други. В социално-икономически аспект компютърното изследване и разпознаване на образци в сложни финансови системи

може да се използва за ориентиране в икономическа ситуация и предвиждане на бъдещото ѝ развитие.

0.1.5. Изкуствен интелект

Изкуственият интелект е добре известна материя, но спорове за това какво обхваща тя и какво точно представлява не липсват и днес. Въпросът в каква степен "интелектът" е свързан с компютърния потенциал е основен и в момента доминират главно две направления, които могат да бъдат обобщени накратко така:

- *Силно направление:* Задачата на изкуствения интелект е да изследва, разбира и моделира процеса на разумно мислене.
- *Слабо направление:* Задачата на изкуствения интелект е да моделира системи, в които определени действия могат да бъдат изпълнявани със същата успеваемост, както ако се изпълняват от човека.

Границата между двете направления е условна и теоретически всяка разглеждана задача и всеки получен резултат може да се интерпретира от двете гледни точки: силна (доколко точно начинът на работа на системата *моделира* мисленето и действията на човека върху същата задача) и слабо (доколко *успешна* е машината). Съществено различие е, че силното направление изисква машината да работи и мисли така, както се предполага, че го прави човекът, докато слабото не се интересува от *начина*, а от *резултата*. В частност при слабото направление е напълно допустимо и дори силно желателно машината да се справя по-добре от човека, ако това е възможно, докато силното направление не толерира това: ако човекът често прави определен вид логическа грешка в разсъжденията си, машината също трябва да я прави.

Силното направление се занимава главно с проблеми като разбиране и интерпретиране на естествените езици, както и философски въпроси като: "Какво да разбираме под интелигентна машина?" В допълнение, съществува връзка с логическото мислене, когнитивната наука, психологията, лингвистика и др.

При слабото направление засега е постигнат доста по-сериозен напредък. Един конкретен успешен пример в това отношение са *експертните системи*. Най-общо, те разглеждат въпроса за *разпространение* на знания и опит на специалисти в определена тясно специализирана област. Съществуват редица успешни реализации на експертни системи за търсене на (ценни) суровинни находища, медицински системи за диагностика на определени заболявания, системи за помощ при интерпретиране и прилагане на закона и други.

Като цяло идеите и на двата "противоположни лагера" заемат централно място в изследванията на съвременната компютърна информатика и пред тях непрекъснато се откриват нови възможности за разширение и бъдещо развитие.

Следващата тема е тясно свързана с някои вече разгледани аспекти на компютърната информатика, в това число и изкуствения интелект.

0.1.6. Нетрадиционни методи за изчисление

Невронни мрежи

Проблеми като разпознаване по образец, разпознаване на говор, разпознаване на образи и др. са някои задачи, при които се счита, че *невронните мрежи* могат да се прилагат успешно. Без да се задълбочаваме в тази насока, ще скицираме съвсем накратко идеите, върху които се основава една невронна мрежа: това е техника за обработване на данни, вдъхновена от начина, по който това се извършва в човешкия мозък. *Изкуствената невронна мрежа* е математически модел, съставен от набор отделни елементи, които симулират някои наблюдавани свойства на биологичните невронни системи (както и процесите на адаптивно биологично усвояване на нови знания и умения). Съвременният модел на невронна мрежа е композиция на добре взаимодействащи си елементи (аналог на *невроните*) и свързващите ги канали (аналог на

синапсите). Основно свойство на невронните мрежи е самомодификацията (самообучението) дотогава, докато бъде постигнат определен желан резултат.

Генетични алгоритми

Генетичните алгоритми също черпят своите идеи от природата. Като цяло те се основават на принципа на естествения подбор (оцеляване на най-доброто) при последователно генериране на "поколения" до решаване на поставената задача. Техниката наподобява процесите на генетично ниво при живите организми и все по-успешно се използва за решаването на трудни изчислителни задачи. В настоящия момент изследванията в тази насока са свързани с разбирането и поставянето на теоретична основа за генетичните алгоритми, както и разширяване на приложението им.

Квантови изчисления

Квантовите изчисления се основават на идеи от физиката. Анализира се потенциалното им приложение в нетрадиционна компютърна архитектура, която може да се окаже значително по-ефективна от сега съществуващите. Изследванията в тази област на този етап са строго теоретични и често се ограничават до дискутиране на връзката между съвременната теоретична физика и компютърната информатика.

0.1.7. Фундаментална компютърна информатика

След краткия увод за периферията на компютърните технологии и информатика ще се насочим към основния предмет на настоящата книга: "ядрото" на компютърните технологии — *фундаменталната компютърна информатика*. Образно казано, "плътността" на това ядро е съсредоточена в *компютърните алгоритми* — главната движеща сила и най-важният механизъм за успешното функциониране на целия компютърен свят.

Съществуват два основни термина при разглеждането на "поведението" на компютъра: *представяне* и *трансформиране* на информацията. *Информация* е всичко, което носи някакъв смисъл (числа, думи, имена, понятия и др.). Компютърът от своя страна работи със символно представяне на информацията под формата на последователност от двоични цифри, което е прието да се нарича *данни*. Следва да се отбележи, че данните са *носител* на информация, но сами по себе си *не са* информация. Така например, за компютъра, както и за всеки незапознат, числото 25 не означава нищо конкретно. Едва когато в него се вложи някакъв смисъл, например цена на чаша кафе в стотинки, може да се говори за информация.

Представянето на информацията се състои в символното кодиране на смисъла, заложен в нея. Трансформирането — това са стъпките на алгоритъма за достигане на определена цел. Получаваме следната неформална дефиниция на *компютърен алгоритъм*:

Компютърният алгоритъм е схема на последователност от компютърни изчисления за решаване на определена задача (достигане на определена цел).

Преди да преминем към кратък преглед на историята на възникване на алгоритмите, както и към материала, разглеждан в книгата по същество, ще направим кратко лирично отклонение: есе, чиято цел е да представи по малко по-нетрадиционен начин (както читателят сам ще се убеди) ползата от изучаването и разбирането на компютърните алгоритми.

0.2. Понятието леймър

Често са ни задавали въпроса какво означава *леймър* (от англ. *lamer*), когато се използва като определение по адрес на някой програмист. Отново, без да даваме формална дефиниция, ще обясним понятието, като използваме въведение *ала Карл Май* [*Май-1989*].

На първо място, най-важната характеристика за един леймър е, че той не подозира, че е такъв.

По-нататък, едно от най-разпространените определения за леймър, е програмист, който пише програми в колонка и който, като чуе за рекурсия, получава леко разстройство.

Леймър е програмист, който знае няколко езика за програмиране, но няма никакво понятие от алгоритми.

Леймър е програмист, който ако изобщо слага коментари в програмите си, те са главно от следния вид:

```
i++; /* увеличаваме i с единица */
```

Леймър е програмист, който не знае, че зад простичката игра *Minesweeper* за *Windows* се крие една от най-сериозните задачи в информатиката, за разрешаването на която университетът Станфорд е определил награда от 1 милион долара [*Prize-2000*].

Леймър е програмист, който може да напише *бързо сортиране* единствено, ако го научи наизуст. Нещо повече, истинските програмисти рядко могат да го запомнят и напишат бързо, но за сметка на това за 10 минути могат сами да си го "измислят". Нашата цел е читателят да достигне това ниво, при което решаването на сложни задачи се свежда до просто упражнение на интуицията му.

Леймър е програмист, който не знае, че "събирачът на боклук" (от англ. *garbage collector*) не е измислен, специално за да събира "труповете" на "починалите" обекти на *Java*, а зад него стои мощна и красива теория, представена още в средата на *XX* век.

Леймър е програмист, който не знае, че най-ефективната търсеща машина в Интернет *Google* дава толкова релевантни резултати, защото се основава на новаторски методи от теорията на графите.

Леймър е програмист, който не знае, че формулата за ефективната реализация на произволен компютърен проект е малко спецификации + много и ефективни алгоритми.

Леймър е програмист, който не може да проумее защо една програма, за която "някой бил казал, че има сложност $\Theta(n^2)$", е по-бърза от друга със сложност $\Theta(n^3)$, след като $n^3 > n^2$.

Леймър е програмист, който може да програмира на Си и който може да намери най-малко общо кратно на две числа на лист, но не може да си напише програма за това.

Леймър е програмист, който не може да си обясни защо като се компресира един файл, и после се компресира още веднъж, той не намалява още повече.

Леймър е програмист, който знае, че това

е дърво, но не подозира, че това също е дърво, както и това . И, че всъщност той самият, като програмист, също е едно младо и зелено дърво.

Леймър е програмист, който за 10 минути може да направи дадена програма 10 пъти по-бавна, без да се притесни, че някой преди него е загубил 10 часа в оптимизиране, за да я направи с 10% по-бърза.

Леймър е програмист, който *brute-force*-ва *ftp* (от англ. *File Transfer Protocol* — протокол за пренос на файлове в компютърна мрежа) пароли, без да съобрази, че при сегашната скорост на Интернет трудно би могъл да *hack*-не парола, по-дълга от 5 символа (а за това е необходимо едно съвсем просто комбинаторно изчисление), освен ако не извади луд късмет.

Леймър е програмист, който трудно може да проумее асоциация на хубава вечеря с хубава програма (а тяхното качество зависи предимно от рецептата/алгоритъма, който се използва).

Леймър е програмист, който не знае, че най-краткото разстояние между две точки не винаги е правата линия (понякога, то се намира по алгоритъма на Дейкстра).

Леймър е програмист, за когото *двоично търсене* означава *търсене* в *двоични* файлове.

Леймър е програмист, който се възхищава на програмни фрагменти като този:

```
0x000000FF & (i >> 24)
```

често, просто защото си няма понятие какво точно означават и панически преминава на следващата страница, когато види "йероглиф" от вида:

$$\chi^2 = \frac{n}{m}\sum_{i=1}^{n}\left(f_i - \frac{m}{n}\right)^2$$

Леймър е програмист, който не е чувал за динамично оптимиране. Всъщност, един от наистина добрите начини да разбиете в спор някой леймър, е да му заявите: "Хей, ами тази задача е добре известна и се решава лесно, елегантно и ефективно с динамично оптимиране".

В началото, като всички начинаещи програмисти, и ние не бяхме нищо повече от едни млади и зелени леймъри. С течение на времето с хубави книги и статии, и най-вече с упорит труд по решаване на алгоритмични проблеми, започнахме да надрастваме това ниво. Днес сме готови да помогнем и на теб, драги читателю, да избягаш от стерилитета на занаятчийското програмиране.

Ще завършим настоящия параграф с една от прекрасните мисли на друг знаменит автор (*Марк Твен*):

"Когато искам да прочета нещо хубаво, сядам и си го написвам".

Целта на настоящата книга е да доведе теб, читателю, до положение, при което да имаш самочувствието да заявиш:

"Когато искам да видя хубава програма, сядам и си я написвам".

0.3. Развитие на алгоритмите

На пръв поглед изглежда, че не може да става дума за алгоритми преди раждането на самия компютър. Това може би е интуитивно ясно, но всъщност съвсем не е вярно. Както всяко нещо в природата, така и *алгоритмите* не са се появили за един ден, а дълго време са търсели своето място под слънцето.

0.3.1. Произход на думата "алгоритъм"

Думата *алгоритъм* идва от името на арабския математик *al-Khowarizmi*, който през VIII век описва откриването на десетичната бройна система и представя редица важни концепции в книгата си *Al-jabr wa'l muqabala* (заглавие, от което по-късно е образувана още една добре известна дума – *алгебра*). Това е най-разпространената хипотеза за появата на думата алгоритъм, въпреки че съществуват и други: по

името на арабския математик *Al-Khashi*, който през XV век предлага метод за изчисляване на константата π с точност 16 знака след десетичната запетая.

Истината, както винаги, е някъде по средата. Това не означава, че в средата на XII-ти век трети арабски математик *Al-** е предложил някаква нова и зашеметяващата за времето си схема за изчисляване на “нещо” — просто, когато се поставя въпросът за произхода на думата алгоритъм, не съществува друг отговор, освен множество хипотези, някои от които се налагат повече от други [*Knuth-1/1968*].

“Историческото” развитие на компютърните алгоритми, доколкото може да се говори за такова, условно може да се раздели на три етапа:

- намиране на начин за символно представяне на информацията под формата на данни;
- формулиране на алгоритми, използващи данните за решаване на изчислителни задачи;
- създаване на механични изчислителни машини, които могат да изпълняват алгоритмите ефективно.

Първият въпрос (за представянето на данните) най-добре може да се илюстрира, като се разгледат различните начини за представянето на числата (някои, от които датират от най-дълбока Древност). Най-простото представяне е като редица от идентични символи, например чертички. Така всеки уважаващ себе си затворник отбелязва върху стената на килията броя на годините, които е прекарал лишен от свобода. Следващ добре известен пример е представянето на числа с *римски цифри*. По-нататък, наред с начините за представяне на числата започват да се появяват и правила за манипулиране с тях — аритметични операции, както и първите форми на прости изчислителни машини — сметало и др.

Следващата кратка хронология показва някои проекти и събития, имащи пряка връзка с компютърните алгоритми, като се започне от Древността и се стигне до средата на XX век.

0.3.2. Алгоритмите през вековете

300 г. пр. н. е. Евклид предлага алгоритъм, който намира най-големия общ делител на две естествени числа m и n:

1) Ако стойността на m е по-малка от тази на n, разменяме стойностите им.
2) На m даваме нова стойност: остатъка от делението на m с n.
3) Ако m е различно от 0, преминаваме към стъпка 1), като използваме новите стойности на m и n.
4) Резултатът (най-големият общ делител на двете числа) е равен на n.

250 г. пр. н. е. Простите числа, както и методите за тяхното намиране, са в основата на най-старите изчислителни задачи. Алгоритъмът на Ератостен, известен като *решетото на Ератостен*, датира още от 250 г. пр. н. е. Оттогава, до днес (с малки изключения), в методите за търсене на прости числа не е постигнат съществен напредък. Наскоро обаче беше направен пробив при сходна задача: намерен беше ефективен алгоритъм за проверка дали дадено число е просто.

780-850 г. Както вече споменахме, това е периодът, през който *Abu Ja'far Mohammed Ben Musa al-Khwarizmi* публикува "*Hisab al-jabr w'al-muqabala*".

1424 г. Годината, през която *Ghiyath al-Din Jamshid Mas'ud al-Kashi* изчислява π с точност 16 знака след десетичната запетая.

0.3.3. Анализ на алгоритмите

1845 г. Гейбриел Лейм (*Gabriel Lame* — Името му няма нищо общо с понятието леймър, въпреки че, ако можеше да прочете описанието от предишния параграф, прочутият математик вероятно би се обърнал в гроба) показва, че алгоритъмът на Евклид извършва брой деления, не повече от 5 пъти броя на десетичните цифри на по-малкото число.

1910 г. Поклингтън дефинира *сложност* на алгоритъм като полином, зависещ от броя на цифрите в двоичното кодиране на обработваните данни.

0.3.4. Изчислимост

1900 г. Дейвид Хилберт поставя въпроса за намирането на процедура за решаване на *диофантови уравнения* (полиноми с цели коефициенти) и по този начин полага основите на по-късно появилите се формални дефиниции за изчислимост.

1920-30 г. Пост предлага прост унарен модел за изчисления, известен като *машина на Пост*.

1930 г. Алонсо Чърч представя *ламбда-смятането*.

1936 г. Алън Тюринг публикува статия, където представя *машината на Тюринг* — формален модел за изчислимост, който освен това е физически осъществим. (Всъщност *почти* осъществим, тъй като формалният модел изисква *неограничен* размер на паметта.)

Последната дата се смята за рождена на съвременния компютър.

От средата на XX век (по обясними причини) теоретичната компютърна информатика започва да се развива с изключително бързи темпове, за да достигне днешния си вид. Изхождайки от "неизвестността на бъдещето", никой не може да бъде сигурен, че тя ще затвърди своя блестящ старт с нови успехи. Въпреки това, авторите на тази книга биха се обзаложили, че през настоящия XXI век това, което ще определя и променя облика на планетата ще бъдат именно компютърните и информационните технологии. И без още да знаем каква ще бъде следващата фундаментална научна теория, която ще преобърне представите ни за света, ние ще се постараем да ти осигурим, драги ни читателю, летящ старт в надбягването с времето.

0.4. програмиране = ++алгоритми

Настоящата книга разглежда компютърните алгоритми и прилагането им за решаване на изчислителни задачи, болшинството от които породени от известни практически проблеми. Този параграф има за цел да изясни:

- за кого е предназначена книгата;
- какво е естеството на представяния материал;
- защо избрахме езика Си за реализациите на алгоритмите, както и няколко думи по конвенцията на предложените изходни текстове;
- нашите благодарности, както и адресите за връзка с нас.

Забележка: Ти, любознателни читателю, вероятно вече си се замислил над заглавието на книгата? Ако не си, ти предлагаме да се опиташ да си отговориш на следните въпроси:

1. Защо заглавието е "програмиране = ++алгоритми", а не "програмиране = алгоритми++".

2. Замисли се отново и над понятието *леймър*.

3. Опитай се да сравниш заглавието на книгата с името на езика Си++: защо не е ++Си и каква е връзката с нашия случай.

0.4.1. За кого е предназначена книгата

Целта на настоящата книга е да запознае читателите с най-разпространените техники на програмиране. Наред с представянето на широкоизвестни методи за решаване на алгоритмични задачи (и анализ на техните свойства, приложения, предимства и недостатъци), се разглеждат и стотици конкретни алгоритмични проблеми, обръща се внимание на анализа на алгоритмичната сложност на предложените решения, прави се сравнение между различни подходи и други. В

книгата се засяга широк спектър от теми, както в теоретичен, така и в чисто приложен аспект. Материалът е ориентиран по-скоро към приложната страна и реализацията на разглежданите алгоритми за сметка на чисто теоретични изследвания и доказателства за коректност (като все пак целта е да се обхванат в някаква степен най-актуалните съвременни научни резултати в съответната област).

Книгата е подходяща за всички, които са избрали програмирането за своя професия и по някакъв начин то е свързано със Си, Си++, *Java*. Тъй като разгледаните техники са фундаментални и обикновено не разчитат на определени характеристики на езика (и средата) за програмиране, то съответните алгоритми са приложими във всеки език за програмиране от високо ниво.

Книгата представлява полезно ръководство за участниците в състезанията по програмиране, ученици и студенти. Като дългогодишни участници в "подобен род мероприятия" ние се постарахме да систематизираме и подредим материала по такъв начин, че той да бъде от максимална полза тогава, когато се налага да се решават алгоритмични задачи бързо и без "право на грешки".

За пълното разбиране на материала предполагаме, че е необходимо читателят да познава езика Си като синтаксис и да има поне малко опит с прилагането му на практика. Почти всичко, необходимо като математическа основа, е включено в уводната глава.

Съдържанието на книгата покрива почти целия материал, необходим за провеждането на съответен университетски курс по алгоритми и/или структури от данни, като е наблегнато основно на алгоритмите. Освен най-разпространените и полезни алгоритмични техники на редица места се засяга по-специфична тематика и, като цяло, материалът трудно може да се преподаде в рамките на един учебен семестър. Такива курсове по алгоритми се четат в няколко университета ("Програмиране и анализ на компютърни алгоритми" в Софийски университет "Св. Климент Охридски", "Структури от данни и програмиране", "Програмиране за напреднали" в Нов Български университет и др.). Съдържанието на курсовете както и по-подробна (и актуална) информация за тях може да бъде намерена на адреса на книгата в Интернет (*виж коментари и въпроси* по-долу).

0.4.2. Кратко представяне на съдържанието

Глава 1. "Въведение в алгоритмите" разглежда някои по-важни математически понятия, функции и техни свойства, както и основни програмистки техники — рекурсия и итерация, генериране на комбинаторни конфигурации и др.

Глава 2. "Въведение в структурите от данни" запознава читателите с най-важните структури от данни или, казано с други думи, със "скелета" на всяка една програма.

Глава 3. "Сортиране" и

Глава 4. "Търсене" са важни теми от "всекидневието" на програмиста. Засегнати са както широко известни, така и някои екзотични (но често изключително полезни) техники на манипулация с "подредени" данни.

Следващите няколко глави са ориентирани към техники и алгоритми за решаване на широка гама от алгоритмични задачи. Те откриват пътя към истински ефективното мислене при проектиране и решаване на алгоритмични проблеми, както и при преценяване и сравняване между различни алтернативи. Техниките са разгледани както в практически, така и в теоретичен аспект. Наред с дефинициите, строго формулираните твърдения и свойства се разглеждат множество примери, задачи и почти винаги е приложена съответна програмна реализация.

глава 5. "Теория на графите"
глава 6. "Търсене с връщане. NP-пълни задачи"
глава 7. "Разделяй и владей"
глава 8. "Динамично оптимиране"
глава 9. "Алчни алгоритми"
глава 10. "Компресиране"

0.4.3. Защо Си?

Изборът на език за програмиране за реализацията на разглежданите алгоритми се предопределя от няколко основни фактора.

Езикът Си (и неговият "*upgrade*" Си++) е традиционен, както в обучението, така и в средите, където се разработва приложен софтуер. Не може да се подмине и все по-нарастващата (и незатихваща) популярност на "по-малкия брат" на Си — *Java*. В основата си двата езика са много близки и, тъй като в настоящата книга се разглеждат и реализират конкретни алгоритмични проблеми, без да се навлиза в специфични "приложни" детайли, програмните фрагменти без особени проблеми могат да се използват в средата на *Java* (Вероятно понякога с някои дребни модификации).

Изборът на език е съобразен и с целите на книгата: освен да изведе обикновения програмист до висините на изкуството на ефективното алгоритмично мислене, да представлява и своеобразна "енциклопедия" по алгоритми за ученици и студенти, участващи в състезания по програмиране (където езикът Си е доминиращ). Не на последно място, отскорошна практика във висшите учебни заведения (особено за *информатика* и сродните ѝ специалности) е да се изучава Си като *основен* език за програмиране. Впрочем, подобна тенденция се наблюдава и в средното образование, но там този процес е по-бавен поради липсата на достатъчно подготвени педагогически кадри, владеещи Си — вероятно тепърва трябва да бъдат произведени от висшите учебни заведения.

Си е ефективен и гъвкав, позволява почти всичко, което е необходимо на един програмист, като често е изключително полезен и при теоретични изследвания — ефективен е при емпирични експерименти и др.

И все пак, защо не използвахме Си++? Ами защото "*upgrade*"-ът на Си няма какво повече да ни предложи, освен може би синтактични удобства в някои ограничени случаи. При това, използването на Си++ определено би ни тласнало в посока на *обектноориентирано програмиране*, което би било не само изкуствено, а просто вредно, когато става дума за *алгоритми*. Наистина, смисъл от въвеждане на обекти може да има при някои *структури от данни*: дървета, списъци, хеш-таблици и други, но едва ли при *алгоритми*. Така например, Робърт Седжуик, автор на превъзходната книга "*Algorithms in Pascal*", по-късно издаде варианти на Си, Си++ и *Java*. На неубедения във вредата от обектите читател предлагаме да сравни вариантите на книгата на Си и Си++.

И все пак, защо не Паскал? За съжаление, Паскал е вече почти мъртъв език. И двамата автори са силно пристрастни към него: той е по-прост от Си, крие по-добре подробностите по вътрешната реализация на определени операции, има вградена обработка на грешки, поради което е по-толерантен към неопитния програмист, и най-важното: по-удобен е за обяснение на алгоритми. Никлаус Уирт, неговият създател, го е замислил именно като *език за обучение* и тук е основната му сила. Тук е обаче и основната му слабост: езикът се оказа недостатъчно гъвкав и изразителен (например по отношение на Си), когато става въпрос за реално програмиране, поради което постепенно беше изоставен преди повече от 10 години. Днес Паскал е в основата на средата за програмиране *Delphi*. Езикът на системата обаче все повече се отдалечава от първоначалния вариант на Уирт и все повече се утвърждава като самостоятелен, различен от Паскал. Подобно на понятието *informatics*, днес Паскал е обект на предразсъдъци и сред програмистките среди има репутацията на "несериозен" език. От *Borland International* съзнаваха това добре, но дълго време се опитваха да се борят за репутацията на езика. Докато накрая и те се предадоха и се принудиха да сменят името му на *Delphi* по чисто търговски съображения. Днес влиянието на Паскал се ограничава до *Macintosh* (системата е разработена на този език, но той постепенно е изтласкван и оттам) и до отделни групи ентусиасти под *Linux*, *Unix* и ДОС.

Не е така със Си: нито Си++, нито *Java*, успяха да го "детронират" и вече няколко десетилетия той е един от основните езици за програмиране. Впрочем, фактът че дори най-съвременните специализирани езици за програмиране в Интернет като *Java*, *Javascript*, *Perl*, *PHP* и други имат Си-подобен синтаксис говори сам по себе си за влиянието на езика в програмисткия свят: разчита се на привличане главно на програмисти, владеещи Си.

0.4.4. Конвенция на изходните текстове

Стилът на изходните текстове (подравняване, отместване, именуване на променливи, функции и т.н.), използван при реализация на алгоритмите в настоящата книга, не се отличава съществено от широко възприетите и за читателя едва ли ще представлява трудност "четенето" на предложените програми.

Всички програми са компилирани и тествани както под *DOS* (с *Borland C*++ 3.1), така и под *Windows* (с *Visual C*++ 6.0). Записани са с разширение на програми на Си и не се отклоняват от стандартите на *ANSI* за езика. Възможно е обаче при някои *входни данни* дадена програма да работи коректно под *Windows*, но не и под *DOS*, поради ограниченията на размерите на променливите и/или на наличната адресируема памет.

0.5. "Следговор"

Посвещение

Панайот посвещава тази книга на Силвия, Никол, Марко и Мария.

Преслав я посвещава на Петя и Слав.

0.5.1. Благодарности

На първо място благодарим на научния си редактор Емил Келеведжиев от БАН за подкрепата, която ни оказа във всички етапи от разработването на книгата, за неколкократното задълбочено изчитане на материала, съпътствано с множество ценни забележки, предложения и корекции.

Сърдечни благодарности на Райко Чалков, който "прецака жестоко всички скатаващи се тихомълком" сгрешени индекси (и не само). Убедени сме, че някой ден ще бъдем изключително горди, че подобен научен талант е участвал в редакцията на нашата книга.

Тук е мястото да изразим и признателността си към нашите колеги от *SAP Labs* и *Rila Solutions* за оказаната подкрепа и прекрасните условия за творческа дейност. Съчетаването на дейности като четене на спецификации, писане на код (на килограм), оптимизиране и дебъгване ни най-малко не си пречеше с "литературните" ни занимания, дори напротив — двете неща се допълваха по особено очарователен начин (освен в периодите, когато наближава *deadline* на проект. Тогава времето от 22:20 до 01:55 е особено ценно.).

Благодарим на доц. Красимир Манев от СУ "Климент Оридски" — гръбнакът на българската компютърна информатика през последните няколко години (в съвременният световен смисъл на думата). Курсът (и учебникът) по дискретна математика, който се чете във Факултета по математика и информатика на Софийски университет, е едно от нещата, заради които човек си заслужава да бъде студент в България (Останалите са общоизвестни — повечето са производни на атмосферата в Студентски град.).

Специални благодарности на доц. Асен Рахнев от ПУ "Паисий Хилендарски", който се запозна отблизо с книгата в нейния почти финален вариант, даде своите ценни препоръки и любезно се съгласи да напише рецензия.

Редакциите и дискусиите с всички наши колеги и приятели беше изключително полезна и безценна, за да придобие материалът финален вид. Още един път изказваме нашата сърдечна благодарност на Станислав Овчаров, Иво Маринчев, Светлин Наков, Павлин Добрев, Васил Поповски, Петко Минков, Петър Петров, Деян Ламбов, Кирил Арабаджийски, Иван Пейков и Валентин Бакоев.

Специална благодарност на всички студенти от курса по ПрАнКА, воден по работен вариант на настоящата книга, и изобщо на всички, които съзнателно или несъзнателно дадоха своя принос.

Много хора ни помогнаха без ни най-малко да подозират (Последната фраза е изплагиатствана, без авторът ѝ да подозира. Благодарности и на него). Това са много от приятелите ни (както и приятелките ни), които ни припомняха къде се намираме в моментите на хронична шизофрения.

Параграфът няма да бъде пълен, ако пропуснем благодарностите за нашите семейства. Не на последно място, и без да правим изявления в стил "негър при връчване на Оскар" като изреждаме всичките си роднини, ще отбележим с **bold специфичната подкрепа**, която получихме от най-близките си хора.

0.5.2. Търсене на грешки

Специални благодарности на всички, които до момента използваха Интернет страницата за публикуване на намерени грешки в книгата http://www.nakov.com/algoplus-bugs/submit-bug.php:

Моника Алексиева (20), Симеон Манчев (15), Георги Пенчев (12), Светлин Наков (12), Стефан Кирязов (6), Цветелина Атанасова (5), Валентин Младенов (3), Виолета Сиракова (3), Захари Караджов (3), Иван Матеев (3), Николай Киров (3), Ивелина Николова (2), Йосиф Бехар (2), Красимир Добрев (2), Николай Недев (2), Николай Станевски (2), Павел Коцев (2), Петър Пенчев (2), Петър Пировски (2), Петър Събев (2), Ростислав Кандиларов (2), Станимир Марков (2), Цветко Д Трендафилов (2), Катя Тодорова (1), Даниела Радева (1), Диана Миленкова (1), Драгомир Кръстев (1), Емил Симеонов (1), Иван Георгиев (1), Катя Тодорова (1), Маргарита Начева (1), Маргарита Стоева (1), Николай Василев (1), Николай Калайджиев (1), Радослав Ненчев Колев (1), Славомир Къслев (1), Стефан Иванов (1) и Тодор Тодоров (1) .

0.5.3. Коментари и въпроси

Материалът в настоящата книга е препрочитан и коригиран внимателно в продължение на няколко години. Въпреки това, както във всеки сериозен труд, допускането на грешки е задължително и авторите не са и помисляли да отнемат удоволствието на внимателния читател от намирането на сгрешени индекси, неточности и двусмислия. Разбира се, надяваме се и ти, драги читателю, да не ни лишиш (нас, както и останалите читатели) от възможността да разберем за някои допуснати грешки (и да ги поправим за следващото издание). За целта можеш да използваш Фейсбук страницата на книгата:

http://www.facebook.com/groups/168112146541301

Можеш и да ни пишеш директно на някой от следните адреси:

panayot.dobrikov@sap.com; dobrikov@gmail.com
nakov@cs.berkeley.edu; preslav.nakov@gmail.com

На изброените по-горе адреси можеш да отправяш запитвания по материала, както и да инициираш дискусии по интересни алгоритмични проблеми.

Най-актуалният вариант на списъка с корекциите може да бъде намерен на страницата на книгата в Интернет на адрес:

http://www.programirane.org

На посочения адрес може да бъде намерена много допълнителна информация: решения на задачите за упражнение и изходните кодове на всички програми; части от книгата в електронен вид, свободни за разпространение; списък с корекции; форум; различни промоции и конкурси; новини; както и препратки към най-разнообразни алгоритмични ресурси.

Глава 1
Въведение в Алгоритмите

"Съществуват три типа математици:
Такива, които могат да броят, и такива, които не могат."

~ Математически фолклор

1.1. Основни математически понятия и алгоритми

В тази част ще обърнем внимание на някои фундаментални математически понятия, без които би било невъзможно разглеждането на който и да е алгоритъм в материала по-нататък. Ще се спрем също и на някои основни математически функции и техни свойства, които ще помогнат за по-доброто разбиране на алгоритмичните проблеми, разглеждани в тази книга.

1.1.1. Основни математически понятия

Материалът в настоящия параграф, има за цел да запознае читателя с използваните в книгата означения, термини, понятия и основни свойства на някои математически обекти. Нашата препоръка, дори и за тези, които вече имат богат опит с компютърните алгоритми, е да прочетат внимателно тази уводна глава така, че неизяснените неща от нея да бъдат максимално малко. Това е необходимо условие за пълното разбиране на материала по-нататък. Разбира се, не е необходимо да се помнят всички термини и дефиниции. Достатъчно е да се прояви само “малко разбиране” към строгия и абстрактен език на математиката.

- *Множества*

Понятието *множество* е първично и не се дефинира строго с други математически понятия. То се възприема интуитивно и най-често се изяснява чрез примери.

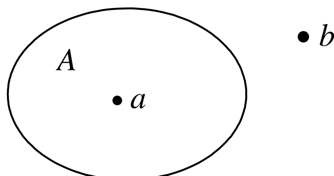
Приема се, че е зададено множество, когато е дадена система (съвкупност) от обекти, най-често обединени на базата на някакъв общ признак, свойство или изобщо по някакво правило. Така например, множество образуват пчелите в един кошер, върховете на многоъгълник, правите, минаващи през дадена точка, корените на уравнението $ax^2 + bx + c = 0$ и др.

Като означение за множество най-често се използват главните букви от латинската азбука: A, B, C, ... Обектите, които участват в дадено множество, се наричат *елементи на множеството* и се бележат най-често с малките латински букви (или чрез подходящо индексиране). Ще избегнем строгата дефиниция на това какво представлява индексно множество, но често за елементи на множеството A ще използваме индексираната редица $a_1, a_2, ..., a_n$ и ще го означаваме така:

$$A = \{a_1, a_2, ..., a_n\}$$

Фактът, че елементът a_i, $i = 1,2,...,n$, принадлежи на множеството A се обозначава с $a_i \in A$, а че не принадлежи — с $a_i \notin A$. Броят на елементите на множество се нарича *мощност* на множеството (за множеството A по-горе този брой е равен на n), като понякога за означение на мощност ще използваме $|A|$. При $n = 0$, множеството се нарича *празно* и се бележи с $\varnothing$. Най-често, множествата се изобразяват графично чрез *диаграми на Вен* (*виж фигура 1.1.1а.*): Всяко

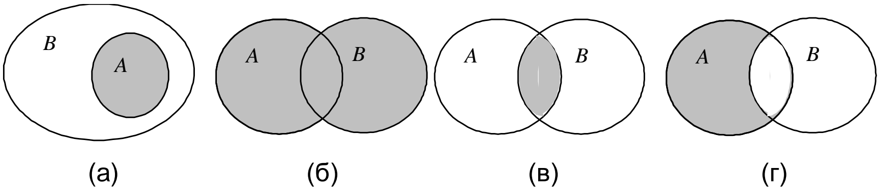
множество A се изобразява като затворена област, а елементите (a, b, …) — като точки, които се намират в, или извън нея в зависимост от това дали принадлежат на множеството, или — не.



Фигура 1.1.1а. Диаграми на Вен.

Съществуват много понятия, свойства и операции, обединени в добре построена *теория на множествата* [*Дилова,Стоянов-1973*]. Ще споменем накратко още някои от тях, които ще бъдат необходими в материала по-нататък.

Дефиниция 1.1. Ако всички елементи на дадено множество A са елементи на друго множество B, то A се нарича *подмножество* на B. Това се означава с $A \subseteq B$ (*виж фигура 1.1.1б.*). Когато допълнително е известно, че B не съвпада с A, множеството A се нарича *собствено* (*същинско*) *подмножество* на B. В този случай ще използваме означението $A \subset B$.



Фигура 1.1.1б. *Подмножество* (а), *обединение* (б), *сечение* (в) и *разлика* (г) на множества.

Дефиниция 1.2. Множеството C се нарича *обединение* на множествата A и B, ако се състои от всички елементи a такива, че $a \in A$ или $a \in B$. Записва се $C = A \cup B$.

Дефиниция 1.3. *Сечение* $C = A \cap B$ на две множества A и B се нарича множество C, състоящо се от всички елементи, които принадлежат едновременно на A и B.

Дефиниция 1.4. *Разлика* $C = A \backslash B$ на множествата A и B се нарича множество C, състоящо се от всички елементи, които принадлежат на A, но не принадлежат на B.

Дефиниция 1.5. Едно множество се нарича *крайно*, ако съдържа краен брой елементи. Иначе се нарича *безкрайно*.

Преди да преминем по-нататък, ще приведем още няколко важни дефиниции. Когато се разглежда множество, не е от значение колко пъти се среща даден елемент, както и наредбата на елементите му. Така следните множества са еквивалентни (съвпадат):

$$\{a, a, b\} \equiv \{a, b, b\} \equiv \{a, b, a\} \equiv \{a, b\} \equiv \{b, a\}$$

Дефиниция 1.6. Множество, в което се допуска повторение на елементите, се нарича *мултимножество*.

Дефиниция 1.7. Ако в n-елементно множество е въведена наредба на елементите, полученият обект се нарича *наредена n-торка* (*списък*).

За обозначаване на наредено n-орки ще използваме кръгли, вместо с къдрави скоби. Така например, наредената тройка (a, b, c) се различава от наредената тройка (a, c, b) и т.н.

Задачи за упражнение:

1. Дадени са множествата $A = \{1,2,4,5,7\}$ и $B = \{2,3,4,5,6\}$. Да се определят множествата: $A \cup B$, $A \cap B$, $A \setminus B$, $B \setminus A$.

2. Дадени са две множества A и B и е известно, че $A \cap B = A$. Какво можете да кажете за множеството B?

3. Операцията *симетрична разлика* $\oplus$ на A и B се дефинира така: $A \oplus B = (A \cup B) \setminus (A \cap B)$. Да се определи симетричната разлика на множествата $A = \{1,2,4,5,7\}$ и $B = \{2,3,4,5,6\}$.

4. Една операция $\bullet$ се нарича *симетрична*, ако $A \bullet B = B \bullet A$. Кои от изброените операции са симетрични: $A \cup B$, $A \cap B$, $A \setminus B$, $B \setminus A$, $A \oplus B$?

- Числа

Числата са основа за всякакъв вид математически изчисления така, както алгоритмите стоят в основата на компютърната информатика. Хронологично погледнато (в самото начало са били използвани само 3 "числа" за броене: *едно*, *две* и *много*), те се считат за първата форма на абстрактно мислене и много хора ги възприемат като нещо магическо и особено.

Числата участват в почти всеки алгоритъм и програма. Ето защо въпросите, касаещи тяхното компютърно представяне, съхранение и използване, са от първостепенна важност.

Дефиниция 1.8. Множеството от *естествените* числа (ще го бележим с N) съдържа числата, с които броим — 0, 1, 2, 3, ...

Съществува спор дали *нулата* трябва да се разглежда като естествено число, или — не. То е по-абстрактно от останалите естествени числа и липсва в много от старите бройни системи: например, в римската бройна система представянето на числата започва от едно. От друга страна, в дискретната математика въвеждането на N най-често се прави, като се започва от нула. Това е така, тъй като при работа с n-елементни множества често се налага в дефиницията да се включи и празното множество.

Дефиниция 1.9. Множеството от *целите* числа Z е съставено от:

..., –3, –2, –1 (цели *отрицателни* числа), 0 (*нула*), 1, 2, 3, ... (цели *положителни* числа).

В паметта на компютъра те се представят като последователност от битове — в *прав*, *обратен* или *допълнителен* код [*Шишков-1995*]. Днес най-често се използва допълнителен код, тъй като при него изваждането лесно се свежда до събиране, което е удобно за компютърна реализация.

При различните компилатори на Си типовете и тяхната дефиниционна област могат да варират. Стандартът *ANSI C* (*American National Standards Institute* [*ANSIC*]) не дефинира конкретни диапазони (в *таблица 1.1.1а.* като пример са показани тези за *Borland C* за *DOS*), а само отношения, например:

`|short| ≤ |int| ≤ |long|`

По стандарт размерът на `int` е една машинна дума (това обяснява, защо под *DOS* типът е с размер 2 байта, а под *Windows* — 4).

Един елегантен начин да се получи максималната стойност на тип, например `unsigned`, е, като се изхожда от вътрешното представяне на числата в допълнителен код: `(unsigned)(-1)`.

Тип	Стойност	Размер
char	-128, ..., 127	8 бита със знак
unsigned char	0, ..., 255	8 бита без знак
short int	-32768, ..., 32767	16 бита със знак
int	-32768, ..., 32767	16 бита със знак
long int	-2147483648, .., 2147483647	32 бита със знак
unsigned short int	0, ..., 65535	16 бита без знак
unsigned long int	0, ..., 4294967295	32 бита без знак

Таблица 1.1.1а. Целочислени типове в *Borland C* за *DOS*.

В средната колона на *таблица 1.1.1а.* е даден диапазонът от стойности, който може да приема променлива от съответния тип. Всеки тип се характеризира с максимално и минимално цяло число, което може да съхранява. Всяко присвояване, надхвърлящо максималната (или минималната) стойност на съответния тип, се нарича *препълване* (от англ. *overflow*) и може да доведе до катастрофални за всеки алгоритъм и програма резултати.

Дефиниция 1.10. *Рационални числа* са тези от вида p/q, където p и q са цели и q е положително. Множеството на рационалните числа се бележи с Q.

Дефиниция 1.11. *Реални* са тези числа, които могат да се запишат във вида:

$$x = n + 0,d_1d_2d_3\ldots ,$$

където n е цяло число, а d_i са десетични цифри от 0 до 9. Числото $0,d_1d_2d_3\ldots$ се нарича *дробна част*. Представяне в горния вид означава, че неравенството

$$n+\frac{d_1}{10}+\frac{d_2}{100}+\cdots+\frac{d_k}{10^k}\le x\le n+\frac{d_1}{10}+\frac{d_2}{100}+\cdots+\frac{d_k}{10^k}+\frac{1}{10^k}$$

е изпълнено за всяко $k\in N$, $k \geq 0$.

Забележете, че последователността на цифрите d_i може да бъде безкрайна. Това може да се случи както за рационални така и за *ирационални* (т.е. които не са рационални) числа. Така например числото 1/3 = 0,333333..... Тук цифрата 3 се повтаря до безкрайност и се нарича *период* на дробта. Записва се още 1/3 = 0,(3) и се чете "нула цяло и три в период". Периодът може да бъде по-дълъг от една цифра, например 1/7 = 0,(142857).

Един пример за реално число, което е ирационално и не е периодично, т.е. не се представя точно във вида p/q ($p,q\in N$, $q > 0$), е отношението на обиколката на окръжност към диаметъра ѝ — константата π:

$$\pi = 3,1415926535 \ldots.$$

Както всяко реално число, така и π може да се апроксимира с рационално число — например 355/113, което ще го определи с известна точност след десетичната запетая, но след определен брой цифри (за избрания пример това са 6 цифри след десетичната запетая) тази точност се губи. Друго полезно приближение е 22/7.

В паметта на компютъра реалните числа теоретично могат да се представят по три начина: с *фиксирана*, *естествена* и *плаваща* запетая [*Шишков-1995*]. На практика, най-често се използват числа с плаваща запетая по стандарт на *IEEE* (*Institute of Electrical & Electronics Engineers*). При реалните типове важат същите забележки, както и при целочислените. В *таблица 1.1.1б.* са показани диапазоните на реалните типове в *Borland C* за *DOS*.

Тип	Стойности	Размер
float	$3,4.10^{-38}$, ..., $3,4.10^{38}$	32 бита
double	$1,7.10^{-308}$, ..., $1,7.10^{308}$	64 бита
long double	$3,4.10^{-4932}$, ..., $1,1.10^{4932}$	80 бита

Таблица 1.1.1б. Реалните типове в *Borland C*.

При работа с реални числа трябва да се внимава както за препълване, така и за загуба на точността след десетичната запетая (на англ. *underflow*): При представяне с фиксиран брой битове точност се губи при закръгляне (в резултат на това, че можем да използваме само ограничен брой битове). При представяне на реално число като рационална дроб точност се губи от апроксимацията при представянето: 1/3 е *безкрайна* десетична дроб и не може да се представи точно с крайно реално число.

Така за всеки тип данни, представящ реално число, съществува най-малко ε ($\varepsilon > 0$) такова, че всяко число (или резултат от аритметична операция), по-малко от ε по абсолютна стойност, се закръгля до 0.

Задача за упражнение:

Да се покаже, че реалните числа, които могат да се представят в стандартните реални типове числа с плаваща запетая всъщност са крайно подмножество на рационалните числа.

- Целочислено деление и остатък. Брой цифри на естествено число

Нека m и n са цели числа, $m \neq 0$. Тогава съществуват единствени цели числа q и r ($0 \leq r < m$) такива, че $n = q.m + r$. Числото q се нарича *частно* от целочисленото деление n/m, а r се нарича остатък. Ако остатъкът r от целочисленото деление е нула, се казва, че m дели n (n е кратно на m) и се записва $m|n$. В Си делението с частно и остатък (при работа с цели числа) става с помощта на операциите / и %. За да не настава "паника", ще използваме тези две означения и в математически текстове:

$$q = n\ /\ m;$$
$$r = n\ \%\ m;$$

Дефиниция 1.12. Когато $(n - m)\ \%\ z = 0$, се казва, че n е *сравнимо* с m по *модул* z и се записва $n \equiv m \pmod{z}$.

Делението с частно и остатък може да се използва за намиране на броя на цифрите на дадено естествено число n. Алгоритъмът е следният: Последователно делим (докато е възможно) целочислено n на 10. Очевидно, при всяко такова деление цифрите на n намаляват с една. Така, броят на цифрите на n се определя от броя на деленията, които извършваме, докато n стане равно на нула:

```
#include <stdio.h>
unsigned n = 4242;

int main(void) {
  unsigned digits;
  for (digits = 0; n > 0; n /= 10, digits++);
  printf("Броят на цифрите на %u е %u\n", n, digits);
  return 0;
}
```
🖫 digits.c

Задачи за упражнение:

1. Да се намерят частното и остатъка от делението на m на n, ако (m,n) е: (7,3), (–7,3), (7,–3), (–7,–3), (3, 7), (–3,7), (3,–7), (–3,–7).

2. При дадени цели m и n ($m \neq 0$) се опитайте да обосновете съществуването и единствеността на представянето $n = q.m + r$, $0 \leq r < m$, (q, r цели). [*Сидеров-1995*]

3. Да се предложи алгоритъм за намиране броя на цифрите на реално число. Какви проблеми възникват?

- Сума и произведение

Дадени са числата $a_1, a_2, \ldots, a_n$. За означаване на сумата им $S = a_1 + a_2 + \ldots + a_n$ най-често се използва някоя от следните три нотации:

$$S = \sum_{i=1}^{n} a_i, \qquad S = \sum_{1 \le i \le n} a_i, \qquad \text{или} \qquad S = \sum_{i=1..n} a_i$$

Възможно е да бъде дадена функция $R(x)$, пораждаща стойностите на i — тогава сумата се записва във вида:

$$S = \sum_{i:R(x)} a_i$$

Следната кратка функция на Си намира сумата S_n на първите n естествени числа:

```
unsigned sum(unsigned n)
{ unsigned i, s = 0;
  for (i = 1; i <= n; i++) s += i;
  return s;
}
```

🖫 sum1.c

Аналогично може да се намери сумата от елементите на даден масив `a[]` с n елемента:

```
int sum(int a[], unsigned n)
{ unsigned i;
  int s = 0;
  for (i = 0; i < n; i++) s += a[i];
  return s;
}
```

🖫 sum2.c

В първия пример цикълът е излишен — сумата можем да намерим директно по формулата за *аритметична прогресия*:

$$S_n = \frac{n.(n+1)}{2}$$

Във втория — подобна директна формула не може да съществува, тъй като елементите на масива могат да бъдат произволни цели числа.

Краен брой суми могат да се влагат една в друга:

$$(1) \quad \sum_{j=1}^{n}\sum_{i=1}^{m} a_j.b_i = a_1.b_1 + a_1.b_2 + \ldots + a_1.b_m + \ldots + a_n.b_1 + \ldots + a_n.b_m =$$

$$= a_1.(b_1 + \ldots + b_m) + a_2.(b_1 + \ldots + b_m) + a_n.(b_1 + .. + b_m) = (a_1 + \ldots + a_n).(b_1 + \ldots + b_m) =$$

$$= \sum_{j=1}^{n} a_j . \sum_{i=1}^{m} b_i$$

Такива вложени суми могат да се пресмятат, като се използват вложени цикли:

```
unsigned i, j;
int result = 0;
for (j = 1; j <= n; j++)
  for (i = 1; i <= m; i++)
    result += a[i] * b[j];
```

Други две интересни свойства, валидни за сумата, са:

$$(2) \sum_{i:R(x)} \sum_{j:S(x)} a_{ij} = \sum_{j:S(x)} \sum_{i:R(x)} a_{ij}$$

$$(3) \sum_{R(x)} a_i + \sum_{S(x)} a_i = \sum_{S(x)\|R(x)} a_i + \sum_{S(x)\&\&R(x)} a_i$$

Тук с $S(x)\|R(x)$ и $S(x)\&\&R(x)$ сме означили съответно обединението и сечението на стойностите, породени от функциите S и R, които могат да приемат параметрите i и j.

При произведение на числа $P = a_1.a_2.a_3.....a_n$ математическата нотация, която се използва, е:

$$P = \prod_{i=1}^{n} a_i \text{ , } P = \prod_{1 \le i \le n} a_i \text{ или } P = \prod_{i=1..n} a_i$$

Функция на Си за намиране на произведението от елементите на даден масив `a[]`, с n елемента, може да има следния вид:

```
int mult(int a[], unsigned n)
{ unsigned i;
  int s = 1;
  for (i = 0; i < n; i++) s *= a[i];
  return s;
}
```

🖫 mult.c

Задачи за упражнение:

1. Да се изведе формулата за сума на аритметична прогресия.
2. Да се докажат свойства (1), (2) и (3) на сумата.
3. Да се формулират и докажат за произведение свойства, аналогични на (1), (2) и (3).
4. Като се използват свойства (1), (2) и (3), да се провери валидността на равенството:

$$\sum_{i=1..n} a_i + \sum_{i=n..m} a_i = \sum_{i=1..m} a_i + a_n \quad , 1 \le n \le m$$

- Степен, логаритъм, n-ти корен

Дефиниция 1.13. Ако x е реално число, а y — естествено, то *степен* се дефинира по следния начин:

$$x^y = x.x. \dots .x$$
(y пъти)

Когато $y < 0$, то $x^y = 1 / x^{-y}$.

Валидни са свойствата ($x \ne 0$):

$$x^y = x^{y-1}.x$$
$$x^y = x^{y+1}/x$$
$$x^{y_1+y_2} = x^{y_1}.x^{y_2}$$
$$x^{y_1.y_2} = (x^{y_1})^{y_2}$$

Тривиалната реализация за пресмятане на x^y е чрез извършване на y последователни умножения:

```
double power(double x, unsigned y)
{ double res = x;
  unsigned i;
  for (i = 1; i < y; i++) res *= x;
  return res;
```

```
}
```
🖫 power.c

По-късно (*виж 7.5.*) ще разгледаме още някои, значително по-интересни и ефективни начини за намиране на x^y.

Ако $x^n = y$ (n е естествено число, $n > 1$), то x се нарича *корен n-ти* от y и се записва $x = \sqrt[n]{y}$. Операцията, при която по дадено y се получава корен n-ти от y се нарича *коренуване*. При $n = 2$, вместо *корен втори* от y се казва *корен квадратен* или само *корен* от y и се записва $\sqrt{y}$. Като се използва операцията коренуване, може да се дефинира и повдигане на число в рационална степен:

$$x^{\frac{p}{q}} = \sqrt[q]{x^p}$$

Най-интересен е случаят, когато x и y са реални числа ($x > 1$). Нека y се представя във вида $y = d,d_1d_2d_3$ ... и да разгледаме следното ограничение:

$$x^{d+\frac{d_1}{10}+\frac{d_2}{100}+\dots+\frac{d_k}{10^k}} \le x^y \le x^{d+\frac{d_1}{10}+\frac{d_2}{100}+\dots+\frac{d_k}{10^k}+\frac{1}{10^k}}$$

Очевидно, то дефинира x^y като уникално реално число — можем да получим произволен брой негови цифри след десетичната запетая при условие, че сме избрали достатъчно голямо k.

Ще разгледаме още една важна функция, която ще бъде необходима в материала понататък, най-вече при анализ на сложността на даден алгоритъм. Уравнението $y = b^x$ за $b \neq 1$, $b > 0$, $y > 0$ има единствено решение относно x. Това решение се нарича *логаритъм от y при основа b* и се бележи с $\log_b y$. Ето някои от свойствата на функцията логаритъм ($x > 0$, $y > 0$, $b > 0$, $b \neq 1$, $c > 0$, $c \neq 1$):

$$x = b^{\log_b x} = \log_b b^x \qquad (1)$$

$$\log_b(xy) = \log_b x + \log_b y \qquad (2)$$

$$\log_b x^y = y.\log_b x \qquad (3)$$

$$\log_b x = \frac{\log_c x}{\log_c b} \qquad (4)$$

По-нататък в книгата често ще изпускаме основата на логаритъма, когато тя е 2 и ще пишем просто $\log x$ вместо $\log_2 x$. Ще използваме още нотацията $\ln x$ за обозначаване на натурален логаритъм $\log_e x$: с основа Неперовото число $e = 2{,}71828...$ (виж 1.1.6.)

Задачи за упражнение:

1. Да се докажат цитираните по-горе свойства на степента, като се изходи от дефиницията.
2. Да се докажат свойства (1), (2) и (3) на логаритъма, като се изходи от дефиницията.

- *Факториел. Рекурентни функции*

Факториел от n, $n \in N$ (записва се $n!$) е произведението на естествените числа от 1 до n:

$$n! = 1.2...n = \prod_{i=1}^{n} i \ ,$$

като по дефиниция $0! = 1$.

Реализацията на Си за пресмятане на $n!$ не представлява особена трудност:

```
unsigned long factoriel(unsigned n)
{ unsigned i;
  unsigned long r = 1;
```

```
  for (i = 2; i <= n; i++) r *= i;
  return r;
}
```
🖫 fact.c

При някои математически функции и редици много по-удобна и нагледна е съответната *рекурентна* дефиниция. Това означава дефиниране на функцията чрез самата нея или изчисляване на всеки пореден член на редицата от стойности на функцията, като се използват стойности на предхождащи го членове. В нашия случай рекурентната дефиниция за факториел изглежда така:

$$n! = \begin{cases} 1, n = 0 \\ n.(n-1)!, n > 0 \end{cases}$$

Аналогично на факториел за сумата на първите *n* естествени числа S_n получаваме рекурентната дефиниция:

$$S_n = \begin{cases} 0, n = 0 \\ S_{n-1} + n, n > 0 \end{cases}$$

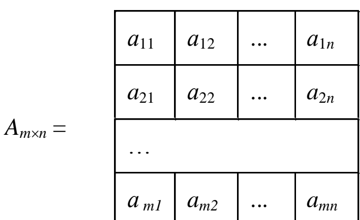
По-късно ще видим (*виж 1.2.*), че за пресмятането на всяка *рекурентна* математическа функция може да бъде написана съответна *рекурсивна* функция на Си.
Задачи за упражнение:

1. Да се предложи рекурентна формула за повдигане на *x* в степен *y* ($x \in R$, $y \in N$).
2. Да се предложи рекурентна формула за намиране на най-големия общ делител на две естествени числа.

- Матрици

Матрицата е правоъгълна таблица от числа (в общия случай правоъгълна таблица от произволни обекти).

Числата *m* и *n* определят *размерността* ($m \times n$) на матрицата: т.е. показват, че тя се състои от *m реда* и *n стълба* ($n \geq 1$, $m \geq 1$). Когато $m = n$ матрицата се нарича *квадратна*. Всеки елемент a_{ij} на матрицата се характеризира с двоен индекс: първото число в него определя номера на реда, а второто — номера на стълба, където е разположен (*виж фигура 1.1.1в.*).



$A_{m \times n} =$

a_{11}	a_{12}	...	a_{1n}
a_{21}	a_{22}	...	a_{2n}
…			
a_{m1}	a_{m2}	...	a_{mn}

Фигура 1.1.1в. Матрица $m \times n$.

Матриците намират широко приложение в алгебрата, при решаване на системи от уравнения, в компютърната анимация, където матрици с размерност 2×2, 3×3, 4×4 се използват за транслация и ротация на обекти, за постигане на различни графични ефекти (като изглед в перспектива в тримерен терен) и др. Използването на матрици в тези и други случаи води до значително по-ефективни алгоритми и съответно до по-висока производителност [*Ayres-1962*].

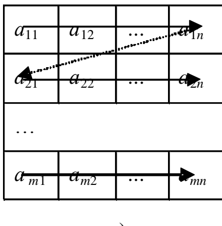
В Си правоъгълна таблица от числа може да се представи чрез двумерен масив:
```
int A[m][n];
```

Възможен е и по-общият случай, когато елементи на матрицата са произволни обекти от някакъв предварително дефиниран тип `struct data`, например:

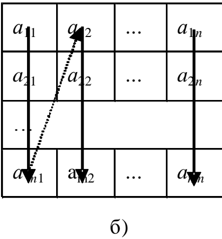
```
struct data {
  int a;
  int b;
  ...
} A[m][n];
```

Въвеждане и отпечатване на матрица

Елементите на матрица могат да бъдат прочетени/отпечатани по най-разнообразни начини. Двата най-прости са *матрица по редове* или *по стълбове* (*фигура 1.1.1г.*).



а)



б)

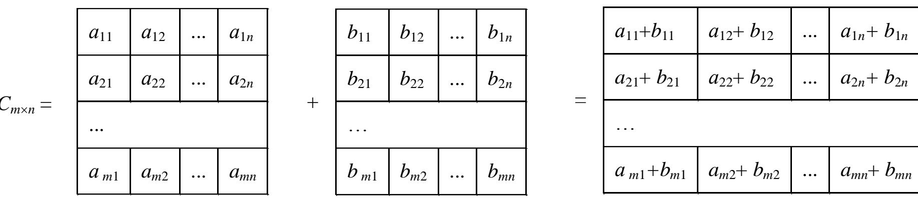
Фигура 1.1.1г. Обхождане на елементите на матрица: (*а*) по редове и (*б*) по стълбове.

```
/* Въвеждане по редове */
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    scanf("%d", &A[i][j]);
/* Въвеждане по стълбове */
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    scanf("%d", &A[j][i]);

/* Отпечатване на матрица по редове */
for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++)
    printf("%.3d", A[i][j]);
  printf("\n");
}
```

🖫 matrix.c

Събиране на матрици

Сума на две матрици $A_{m\times n}$ и $B_{m\times n}$ е трета матрица $C_{m\times n}$ такава, че $c_{ij} = a_{ij} + b_{ij}$ (за $i = 1, 2,...,$ m, $j = 1, 2, ..., .n$), *виж фигура 1.1.1д*.



Фигура 1.1.1д. Сума на матрици.

```
for (i = 0; i < m; i++)
```

```
    for (j = 0; j < n; j++)
      C[i][j] = A[i][j] + B[i][j];
```

🖫 summat.c

Умножение на матрици

Произведение на две матрици $A_{m\times n}$ и $B_{n\times p}$ е трета матрица $C_{m\times p}$, за която е в сила:

$$c_{ij} = \sum_{k=1}^{n}\left(a_{ik}.b_{kj}\right),\text{ за всяко } i = 1,2,\ldots,m \text{ и } j = 1,2,\ldots,p.$$

Реализация по-долу, директно прилага формулата от дефиницията и извършва $m.p.n$ прости умножения. Ако $n > m$ и $n > p$, то броят на простите умножения е ограничен от n^3.

```
for (i = 0; i < m; i++)
  for (j = 0; j < p; j++) {
    C[i][j] = 0;
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
  }
```

🖫 multmat.c

Произведението на матрици намира много приложения — при решаване на системи от уравнения, в статистиката, в теорията на графите и др. Съществува метод за умножение на числови матрици, който извършва $n^{\log 7}$ ($\cong n^{2,81}$) прости умножения, като за сметка на това извършва допълнителни събирания. Методът, известен като *метод на Щрасен* за бързо умножение на матрици, ще разгледаме в *7.6.*, като демонстрация на техниката *разделяй и владей*.

Задачи за упражнение:

1. Дадена е матрица `unsigned a[MAX][MAX]`. Да се напише функция

`void fillMatrix(unsigned a[][MAX], unsigned n),`

която запълва с числа елементите на `a[][]` по следния начин:

0	20	19	17	14
1	0	18	16	13
2	5	0	15	12
3	6	8	0	11
4	7	9	10	0

2. Дадена е матрица `unsigned a[MAX][MAX]`. Да се напише функция, която извежда елементите ѝ по спирала, например за $n = 5$ имаме:

1	16	15	14	13
2	17	24	23	12
3	18	25	22	11
4	19	20	21	10
5	6	7	8	9

1.1.2. Намиране броя на цифрите на произведение

Задача: Дадени са целите числа a_1, a_2, ..., a_n. Да се намери броят на цифрите на произведението $P = a_1.a_2.\ldots.a_n$.
В частност, ако $a_i = i$, за $i = 1,2,\ldots,n$, то се търси броят на цифрите на $P = 1.2 \ldots n = n!$.

Едно очевидно решение е да се извърши умножението и, като се използва алгоритъмът от *1.1.1.*, да се намери броят на цифрите на резултата. За избрания пример това означава да се пресметне $n!$. Функцията $n!$ обаче расте прекалено бързо, докато броят на цифрите се увеличава многократно по-бавно. 10! е 3628800, а цифрите му са едва седем. При 20! резултатът надхвърля максималната допустима стойност за целочислен тип в Си.

Ще приложим по-ефективен алгоритъм, основаващ се на зависимост между числата, които умножаваме, и броя на цифрите в полученото произведение:

Броят на цифрите на P е равен на $\left[1+\sum_{i=1}^{n}\log_{10} a_i\right]$, където с $[x]$ се означава най-голямото цяло число, по-малко или равно на x.

За да обясним валидността на горната формула, ще обърнем внимание на факта, че броят на цифрите на всяко цяло число P е равен на $[1+\log_{10}(P)]$, както и на свойството на логаритмичната функция:

$$\log P = \log(a_1.a_2. \ldots a_n) = \log a_1 + \log a_2 + \ldots + \log a_n.$$

Следва реализация на Си за намиране цифрите на $n!$

```
#include <stdio.h>
#include <math.h>
const unsigned long n = 123;
int main(void)
{
  double digits = 0;
  unsigned i;
  for (i = 1; i <= n; i++)
    digits += log10(i);

  /* [x] се реализира, като се използва превръщане до unsigned long */
  printf("Броят на цифрите на %lu! е %lu\n", n, (unsigned long)digits+1);
  return 0;
}
```

🖫 digitsnf.c

Задача за упражнение:

Да се докаже, че за всяко естествено число P броят на десетичните му цифри е равен на $[1+\log_{10}(P)]$.

1.1.3. Прости числа

Простите числа се разглеждат в математиката още от дълбока древност и са свързани с много интересни задачи далеч извън нейните предели. В информатиката те намират приложение в криптирането, архивирането и т.н.

Дефиниция 1.14. Едно естествено число се нарича *просто*, ако няма други делители освен 1 и себе си, като числото 1 не се счита за просто. Ако не е просто, то се нарича *съставно*.

Редицата на простите числа започва така:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, …

Доказано е (*Евклид* – 300 г. пр.н.е.), че простите числа са безброй много.

Както вече споменахме, едно от приложенията на простите числа е за криптиране на данни, които се предават през "несигурна" мрежа (например финансови транзакции в Интернет). Някои алгоритми за кодиране на предаваната информация (например *RSA* [*RSA-78*]) използват произведение на големи прости числа. За да бъде "разбит" такъв канал за предаване на информация, трябва да бъдат известни *самите* прости числа, а не само тяхното *произведение*. Ако разгледаме числото 55, веднага можем да се досетим как се разлага то — като произведение на 5 и 11. За числото 4853 трудно бихме възстановили на ум простите му делители (211 и 23), но

бихме могли да съставим програма, която ги намира. В настоящия момент са известни много големи прости числа — ако умножим например две произволни 100 цифрени прости числа и разполагаме само с полученото произведение, то възстановяването на числата, по който и да е алгоритъм, ще отнеме неприемливо дълго време (например, многократно повече от времето, необходимо за приключването на една банкова транзакция).

Когато разглеждаме редицата на простите числа, възникват някои въпроси:

- Колко прости числа има в даден интервал $[a, b]$?
- Каква част от безкрайността представляват простите числа?
- Съществува ли формула за намиране на n-тото поред просто число?

Ако с $\pi(x)$ означим броя на всички прости числа, ненадминаващи дадено естествено число x, то намирането на точна формула за изчисляването на $\pi(x)$ ще даде отговор на горните три въпроса. За съжаление такава формула все още няма (и е малко вероятно да се намери — *виж 6.2.*), но съществуват някои формули за апроксимиране на $\pi(x)$, например (нека да си припомним, че $\ln x \equiv \log_e x$):

Теорема (*за простите числа*) $\pi(x) \cong x / ln(x - a)$, където a е произволна положителна константа, по-малка от x.

Най-добро приближение се постига при $a = 1$.

Следствие. n-тото просто число е приблизително $[n.ln(n)]$. По-добро приближение се постига с $[n(ln(n) + ln(ln\ n - 1))]$.

Следствие. Вероятността едно число x да бъде просто е приблизително $1/ln(x)$.

Преди да преминем към алгоритмите за проверка дали едно число е просто, ще споменем още няколко интересни свойства и теореми за простите числа [*Primes-1*][*Primes-2*]:

Хипотези на Голдбах

1. Всяко цяло четно число $n > 2$ може да се представи като сума на две прости числа.
2. Всяко цяло число $n > 17$ може да се представи като сума на три различни прости числа.
3. За всяко цяло число съществува представяне като сума на най-много шест прости числа.
4. Всяко цяло нечетно число $n > 5$ може да се представи като сума на три прости числа.
5. Всяко четно число може да се представи като разлика на две прости числа.

Теорема. Съществуват безброй много прости числа от вида n^2+m^2 и n^2+m^2+1.

Хипотеза. Съществуват безброй много прости числа от вида n^2+1.

Теорема. (*Оперман*) Винаги съществува просто число между n^2 и $(n+1)^2$.

Задачи за упражнение:

1. Защо намирането на точна формула за $\pi(x)$ ще даде отговор и на трите въпроса за простите числа, формулирани по-горе?
2. Да се напишат програми за проверка на хипотезите на Голдбах.
3. Да се напише програма за проверка на теоремата на Оперман.

- Проверка дали дадено число е просто

Един очевиден алгоритъм, пряко следствие от дефиницията, е следният: проверяваме дали всяко число от интервала $[2, \frac{p}{2}-1]$ дели p и, ако намерим такова, следва, че p е съставно.

Съществуват някои “формули” за проверка дали число е просто, но на практика те се оказват неприложими, тъй като реализацията им изисква много повече изчислителни ресурси, отколкото току-що описаният алгоритъм. Един пример е следната

Теорема (*Уилсън*). Числото p е просто тогава и само тогава, когато $(p-1)! \equiv -1 \pmod{p}$.

При нея трябва да се пресметне $(p-1)!$, което е доста по-трудно като реализация и предполага повече изчисления от извършването на $\frac{p}{2}-1$ деления в горния алгоритъм.

Лесно може да се съобрази, че е излишно да изследваме всички числа до $\frac{p}{2}-1$: достатъчно е да проверим за делимост само до $\sqrt{p}$ (включително). Това е така, тъй като винаги, когато p има делител x, $x > \sqrt{p}$, то следва, че p се представя във вида $p = x.y$, $y < \sqrt{p}$, т.е. има и делител по-малък от $\sqrt{p}$. Следва реализация на този алгоритъм:

```
#include <stdio.h>
#include <math.h>
const unsigned n = 23;

char isPrime(unsigned n) /* връща 1, ако е просто, и 0 - при съставно */
{ unsigned i = 2;
  if (n == 2) return 1;
  while (i <= sqrt(n)) {
    if (n % i == 0) return 0;
    i++;
  }
  return 1;
}

int main(void) {
  if (isPrime(n))
    printf("Числото %u е просто.\n", n);
  else
    printf("Числото %u е съставно.\n", n);
  return 0;
}
```

🖫 prime.c

Можем да разширим още последния резултат: за да установим, че p е просто, е достатъчно да сме сигурни, че не се дели на нито едно друго *просто* число от интервала $[2, \sqrt{p}\,]$ (Оставяме като упражнение за читателя да покаже, че последното заключение е коректно). Така, ако разполагаме с първите k прости числа (да ги означим с P_i, за $i = 1,2,\ldots,k$), ще можем да проверяваме дали произволно число от интервала $[2, (P_k)^2]$ е просто. Следва реализация на Си:

```
#include <stdio.h>
const unsigned n = 23;

/* брой прости числа, с които разполагаме - изчислени предварително */
#define K 25
unsigned prime[K] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
                      47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};

/* проверяваме дали число е просто, като проверяваме дали има делители *
   сред числата от масива prime[] */
char checkprime(unsigned n)
```

```
{ unsigned i = 0;
  while (i < K && prime[i] * prime[i] <= n) {
    if (n % prime[i] == 0) return 0;
    i++;
  }
  return 1;
}

int main(void)
{ if (checkprime(n))
    printf("Числото %u е просто. \n", n);
  else
    printf("Числото %u е съставно. \n", n);
  return 0;
}
```

🖫 preproc.c

Задачи за упражнение:

1. Да се напише програма за проверка дали едно число е просто, като се ползва теоремата на Уилсън. Необходимо ли е да се пресмята $(p-1)!$, като се има предвид, че ни интересува само остатъкът му по модул p?

2. Да се докаже, че за да докажем, че p е просто, е достатъчно да сме сигурни, че не се дели на нито едно друго *просто* число от интервала $[2, \sqrt{p}\,]$.

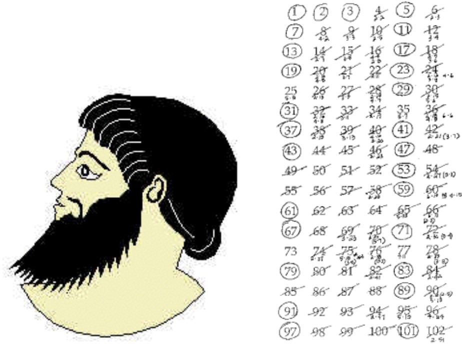
- Решето на Ератостен. Търсене на прости числа в интервал

Задача: Да се намерят всички прости числа, по-малки или равни на дадено естествено число n.

Очевиден подход за решаване на задачата е да се проверява дали всяко естествено число по-малко от n е просто. Така се извършват n проверки, като за всяко число k ще бъдат необходими най-много $\sqrt{k}$ на брой проверки за делимост.

При условие, че разполагаме с достатъчно памет, можем да приложим по-бърз метод за намиране на простите числа в интервал, наречен "*решето на Ератостен*". Методът носи името на своя създател – Ератостен от Сирен (275-195 пр. н. е.) — първият, предположил точно диаметъра на Земята. Известен е още с това, че дълги години е работил в Александрийската библиотека.

Както подсказва името, "*решетото*" е метод за програмиране, при който се изключват всички елементи от крайно множество, които не ни интересуват [*Рахнев, Гъров, Гаврилов-1995*] [*Рейнгольд, Нивергельт, Део-1980*]. Може да се илюстрира с цедка за спагети – водата, от която искаме да се "отървем", изтича, а спагетите остават. По подобен начин решето на Ератостен "изцежда" съставните числа, оставяйки простите.



Фигура 1.1.3. Ератостен и неговото решето.

В случая идеята на този подход идва от предишния параграф: едно число е просто, ако няма други прости делители (освен себе си).

Решето на Ератостен:

Записваме числата от 2 до n в редица:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, ... , n

Намираме първото незачертано и немаркирано число — това е 2. Маркираме го, след което задраскваме всяко второ число след него:

(2), 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ... , n

По-нататък, отново намираме първото незачертано и немаркирано число: това е числото 3. Маркираме го и задраскваме всички числа в редицата, кратни на 3:

(2), (3), ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ... , n

След това, на ред е числото 5 — маркираме го и задраскваме всяко 5-то:

(2), (3), ~~4~~, (5), ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ... , n

Така всички съставни числа се "отсяват" и винаги сме сигурни, че минималното незадраскано и немаркирано i е просто. Процесът продължава, докато не остане нито едно незадраскано или немаркирано число — тогава всички маркирани числа са прости, а всички задраскани — съставни. Ето и по-подробна схема за реализация на отсяване по метода на решетото на Ератостен:

Алгоритъм на решетото на Ератостен

1) Инициализираме масив `sieve[]` с нули. По-късно, когато задраскаме някое число, на съответната позиция в масива ще записваме 1. Въвеждаме променлива-брояч i, която показва кое число разглеждаме в момента (т.е. ще сочи първото незадраскано или немаркирано число). Инициализираме $i = 2$.
2) Увеличаваме i, докато `sieve[i]` стане 0. Тогава числото i е просто и го отпечатваме.
3) Маркираме с 1 всички стойности `sieve[k]`, за $k = i, 2i, 3i, \ldots, (n/i).i$ (т.е. всички кратни на i стойности).
4) Ако $i \leq n$, то се връщаме на стъпка 2, иначе приключваме.

```
#include <stdio.h>
#define MAXN 30000

/* Намира и отпечатва простите числа до n */
```

```
const unsigned n = 200;
char sieve[MAXN];
void eratosten(unsigned n)
{ unsigned j, i = 2;
  while (i <= n) {
    if (sieve[i] == 0) {
      printf("%5u", i);
      j = i * i;
      while (j <= n) {
        sieve[j] = 1;
        j += i;
      }
    }
    i++;
  }
}

int main(void) {
  unsigned i;
  for (i = 0; i < n; i++) sieve[i] = 0;
  eratosten(n);
  return 0;
}
```

🖫 sieve.c

Резултат от изпълнението на програмата:

```
  2    3    5    7   11   13   17   19   23   29   31   37   41   43   47   53
 59   61   67   71   73   79   83   89   97  101  103  107  109  113  127  131
137  139  149  151  157  163  167  173  179  181  191  193  197  199
```

Съществува и още по-ефективен алгоритъм за търсене на всички прости числа в интервал. При него не е необходима памет (масив `sieve[N]`) с големината на интервала, както не е необходимо и на всяка стъпка да се обхожда целият интервал.

При реализацията 🖫 preproc.c от предишния параграф въведохме предварително масив `prime[]`, съдържащ първите k прости числа, и с негова помощ проверявахме дали едно число от интервала $[2, (P_k)^2]$ е просто. Сега ще приложим следната модифицирана схема: ще започнем от празен списък, който ще попълваме последователно. Например, поставяйки в него първото просто число 2 можем да намерим всички прости числа в интервала $[3, 2^2]$ – такова е 3 и го добавяме в списъка. По-нататък, разполагайки вече с простите числа до 3, можем да намерим всички прости числа в интервала $[4, 3^2]$ — това са 5 и 7, които също добавяме в списъка. Продължаваме този процес, докато в `prime[]` се добавят достатъчно прости числа, за да покрият проверката дали всяко число от интервала $[2, n]$ е просто. Следва примерна реализация:

```
#include <stdio.h>

#define MAXN 10000
/* Намира простите числа до n */
const unsigned n = 500;

unsigned primes[MAXN], pN = 0;

char isPrime(unsigned n)
{ unsigned i = 0;
  while (i < pN && primes[i] * primes[i] <= n) {
    if (n % primes[i] == 0) return 0;
    i++;
  }
  return 1;
```

```
}

void findPrimes(unsigned n)
{ unsigned i = 2;
  while (i < n) {
    if (isPrime(i)) {
      primes[pN] = i;
      pN++;
      printf("%5u", i);
    }
    i++;
  }
}

int main(void) {
  findPrimes(n);
  printf("\n");
  return 0;
}
```

🖫 rproc.c

В случаите, когато търсим първите n прости числа, методът на решетото на Ератостен дава много добри резултати. Когато обаче търсим простите числа в интервала $[a,b]$ за достатъчно големи a (a>>1), е по-добре да се използва друг алгоритъм — подобрен вариант на непосредствената проверка.

Първите три прости числа са 2, 3 и 5. Така всяко естествено число може да се представи във вида:

$$n = 30.q+r,\ r\in[0..29] \text{ или } r\in[0, \pm1, \pm2, ..., \pm14, 15].$$
$$30.q = 2.3.5.q$$

Тогава от всеки 30 последователни числа *евентуално* прости са само 8. Т. е. трябва да проверим само $4k/15$ на брой числа, а именно:

$$30.q\pm1,\ 30.q\pm7,\ 30.q\pm11,\ 30.q\pm13$$

Предложеният метод води до двукратно ускорение в сравнение с метода на непосредствената проверка на всички числа от зададения интервал.

Задачи за упражнение:

1. Алгоритъмът може да се подобри: на стъпка 3) търсенето започва от $k = i^2$, като нарастването със стойност i се запазва. Да се обоснове този резултат и да се модифицира по подходящ начин реализацията. Да се сравни с изходната.
2. Да се обоснове алгоритъмът на решетото.
3. Да подобри алгоритъмът за търсене на прости числа в интервал, като за целта се разгледат първите 4 прости числа.

- *Разлагане на число на прости делители*

Теорема (*Основна теорема на аритметиката*) Всяко естествено число P (P>1) може да се представи (*факторизира*) по единствен начин във вида $P_1^{q_1}. P_2^{q_2}. ... .P_n^{q_n}$, където $P_1 < P_2 < ... < P_n$ и P са прости числа, а q_i — цели положителни числа. [*Сидеров-1995*].

Следват няколко примера:

$$520 = 2^3.5^1.13^1$$
$$64 = 2^6$$
$$2345 = 5^1.7^1.67^1$$

Алгоритъмът за получаване на такова разлагане, необходим при някои изчислителни задачи (например *Алгоритъм 2 от 1.1.5.*), е следният:

Нека разлагаме числото P.

1) Полагаме $i = 2$.
2) Полагаме $k = 0$. Докато P се дели на i, извършваме делението и увеличаваме k с единица. Преминаваме към 3).
3) Ако $k > 0$, то сме получили поредния член от разлагането — той е i^k. Преминаваме към 4)
4) Ако $P > 1$, увеличаваме i с единица и се връщаме на 2).

Предоставяме на читателя да докаже, че алгоритъмът работи правилно. Следва пълна реализация на Си:

```
#include <stdio.h>
unsigned n = 435; /* Число, което ще се разлага */

int main(void) {
  unsigned how, i, j;
  printf("%u = ", n);
  i = 1;
  while (n != 1) {
    i++;
    how = 0;
    while (0 == n % i) {
      how++;
      n = n / i;
    }
    for (j = 0; j < how; j++)
      printf("%u ", i);
  }
  printf("\n");
  return 0;
}
```

🖫 numdev.c

Задача за упражнение:

Да се обоснове предложеният алгоритъм за факторизация.

- Намиране на броя на нулите, на които завършва произведение

Задача: Дадена е редица от цели числа $a_1, a_2, ..., a_n$. Търсим броя на нулите, на които завършва произведението $P = a_1.a_2. ... .a_n$.

Както в *1.1.2.*, така и тук извършването на умножението е нежелателно и не винаги ще доведе до получаване на търсения резултат. За да решим задачата, ще обърнем внимание на следния факт: Единствените числа, чието произведение завършва на нула, са 2 и 5, или произведение на число, кратно на 2, с число, кратно на 5.

Алгоритъмът, който решава задачата без извършване на умножението, има следния вид:

1) За всяко a_i ($i = 1, 2,..., n$) представяме a_i във вида $a_i = 2^{Mi}.5^{Ni}.b_i$, като $b_i \% 2 \neq 0$, $b_i \% 5 \neq 0$.
2) Резултатът от произведението ще бъде $P = 2^{\sum_{i=1..n} M_i}.5^{\sum_{i=1..n} N_i}.c$, ($c$ е константа), а броят на нулите в края на произведението ще бъде минималното от $\sum_{i=1}^{n} Mi$ и $\sum_{i=1}^{n} Ni$.

Например, за редицата:

25, 4, 20, 11, 13, 15

след разлагането ще получим:

$$2^0.5^2.1,\ 2^2.5^0.1,\ 2^2.5^1.1,\ 2^0.5^0.11,\ 2^0.5^0.13,\ 2^0.5^1.3,$$

което дава 4 нули. Правилността на получения резултат можем да проверим директно: 25.4.20.11.13.15 = 4290000.

Реализацията на току-що описания алгоритъм ще оставим като упражнение за читателя. Тя ще бъде лека модификация на вече разгледания алгоритъм за разлагане на число на прости делители от предишния параграф.

Ще завършим с формула за намиране на броя на нулите в края на n!. Този брой е равен на $\sum_{k=1}^{[\log_5 n]} \left[\frac{n}{5^k} \right]$. Формулата се извежда като следствие от по-общата схема, след като се пресметне колко пъти цифрите 2 и 5 участват като делители на първите n естествени числа.

```
#include <stdio.h>
const unsigned n = 10;

int main(void) {
  unsigned zeroes = 0, p = 5;
  while (n >= p) {
    zeroes += n / p;
    p *= 5;
  }
  printf("Броят на нулите в края на %u! е %u\n", n, zeroes);
  return 0;
}
```

🖫 factzero.c

Задачи за упражнение:

1. Да се обоснове предложеният алгоритъм за намиране броя на нулите, на които завършва произведение.
2. Да се реализира предложеният алгоритъм за намиране броя на нулите, на които завършва произведение.
3. Да се обоснове формулата за намиране броя на нулите, на които завършва n!

1.1.4. Мерсенови и съвършени числа

- Мерсенови числа

Дефиниция 1.15. Едно просто число се нарича *Мерсеново просто число*, ако може да се представи във вида 2^p-1, където p е просто.

За момента за известни 39 стойности на p, за които числата 2^p-1 са Мерсенови:

2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917

Първите 37 стойности на p числа съответстват на степенните показатели на първите 37 Мерсенови числа. Последните две числа са Мерсенови (открити съответно през 1999 и 2001 г.), но не са непременно 38-ото и 39-тото, тъй като са намерени с компютър, без да са проверявани *всички* по-малки. стойности на p. Т.е. възможно е да има по-малко Мерсеново число.

Мерсеновите числа намират редица приложения: за търсене на *съвършени* числа, при търсенето на много големи прости числа и други. Те имат както практическа, така и символична стойност. Когато 23-тото Мерсеново просто число е било намерено през 1963 г., Математическият факултет на университета в Илинойс толкова се гордеел с откритието си, че

всички писма са били изпращани с допълнителна марка с надписа "$2^{11213}–1$ е просто" (*виж фигура 1.1.4.*).

Едни от най-големите прости числа, намерени до момента, са Мерсенови:

$2^{13466917}–1$ с 8107892 цифри (??39-то Мерсеново просто число)
$2^{6972593}–1$ с 2098960 цифри (??38-то Мерсеново просто число)
$2^{3021377}–1$ с 909526 цифри (37-то Мерсеново просто число)
$2^{2976221}–1$ с 895932 цифри (36-то Мерсеново число)
$2^{1398269}–1$ с 420921 цифри (35-то Мерсеново число)

Все по-често срещано явление е за сериозни научни или емпирични резултати, да се обявяват големи парични награди. Така например, съществуват общи проекти в Интернет за търсене на следващото голямо просто число (а то с голяма вероятност ще се окаже Мерсеново просто — *виж по-долу*). Подобни проекти са финансирани от известни изследователски организации и участването в такъв проект прилича по-скоро на лотария: късметлията, имал щастието да "уцели" 38-то Мерсеново просто число, е спечелил $50000. Наградите за следващите такива числа започват от $250000 [*Primes-3*].

Фигура 1.1.4. 2^{11213}-1 е просто.

Очевиден алгоритъм за търсене на Мерсенови числа е следният: последователно за всяко просто число p = 2, 3, 5, 7, 11, … проверяваме дали $M = 2^p – 1$ е просто. Този алгоритъм обаче е крайно неефективен и на практика не дава задоволителни резултати, приложен за големи числа. Ключът за намирането на такива големи числа е теоремата на *Лукас* от *1870* година, модифицирана по-късно от *Лемер*, като за нейното приложение е необходима и бърза програма за умножаване на числа (такъв метод за умножаване на числа е намерен по-късно и се основава на *бързото преобразуване на Фурие* [*Guinier-1991*]).

Методът на Лукас и Лемер (*Lucas–Lehmer Test*, 1930) се състои в следното:

Дефинира се рекурентната редица:

$$E_1 = 4$$
$$E_{n+1} = (E_n)^2 – 2$$

Първите няколко члена на тази редица са:

4, 14, 194, 37634, ….

Теорема (*Лукас-Лемер*). Естествено число $m = 2^p – 1$ е Мерсеново просто (за p нечетно) тогава и само тогава, когато:

$$(E_{p–1}) \% (2^p – 1) = 0$$

Предоставяме на читателя да реализира програма за намиране на първите няколко Мерсенови прости числа. В следващия параграф ще разгледаме едно от приложенията на Мерсеновите прости числа — за търсене на съвършени числа.

Задачи за упражнение:

1. Да се напише програма за намиране на първите n Мерсенови числа, без да се използва теоремата на Лукас-Лемер.
2. Да се напише програма за намиране на първите n Мерсенови числа, като се използва теоремата на Лукас-Лемер. Да се сравни по ефективност с предходната задача.
3. Може ли $2^n–1$ да бъде просто, ако n не е просто?

- Съвършени числа. “Големи” числа

Дефиниция 1.16. Едно естествено число n се нарича *съвършено*, ако е равно на сумата от всички свои делители (самото n не се счита за делител).

Първите 3 съвършени числа са:

$$6 = 1 + 2 + 3,$$
$$28 = 1 + 2 + 4 + 7 + 14,$$
$$496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248.$$

Редицата от съвършените числа нараства изключително бързо:

8128, 33550336, 8589869056, …

Така, ако се опитаме да ги търсим с директен алгоритъм (последователно за всяко $n = 1$, 2, 3,… да проверяваме дали е съвършено), то няма да можем да намерим повече от 5-тото съвършено число за разумно изчислително време. На помощ ни идва следната:

Теорема. (*Ойлер*) Ако 2^p-1 е просто, то $2^{p-1}.(2^p-1)$ е съвършено.

По горната теорема може директно да се състави програма за намиране на първите n съвършени числа: достатъчно е да знаем степенните показатели p_i на първите n Мерсенови числа (първите 10 степени са 2, 3, 5, 7, 13, 17, 19, 31, 61, 89). Проблем възниква, тъй като дори при малки стойности на n, търсените съвършените числа надхвърлят максималната допустима стойност за целочислен тип в Си. Поради това няма да използваме стандартни типове, а ще реализираме необходимите операции с “големи” числа: умножение на число по 2 и увеличаване с единица. “Голямото” число ще пазим в едномерен масив `number[]`: всеки негов елемент представя по една десетична цифра. За удобство цифрите на числото се разполагат в обратен ред в масива: първата цифра е записана в `number[k-1]` (ако числото има k цифри), втората в `number[k-2]` и т.н. Последната k-та цифра е записана в `number[0]`. Алгоритъмът за увеличаване с единица на число, записано в `number[]`, е следният:

```
i = 0; number[i]++;
while (10 == number[i]) { number[i] = 0; number[++i]++; }
if (i == k) k++;
```

Тъй като при търсенето на съвършени числа се извършва единствено умножение по 2, при което резултатът ще бъде степен на двойката, т.е. последната цифра не може да бъде 9, то последните два реда, започвайки от проверката `while(10 == number[i])`, са излишни.

Аналогично, реализираме умножение на голямо число по две:

```
unsigned i, carry = 0, temp;
for (i = 0; i < k; i++) {
  temp = number[i] * 2 + carry;
  number[i] = temp % 10;
  carry = temp / 10;
}
if (carry > 0) number[k++] = carry;
```

Следва пълна реализация, намираща първите 10 съвършени числа, като степените на първите десет Мерсенови прости числа са зададени като константи в масива `mPrimes[]`:

```
#include <stdio.h>
#define MN 10

unsigned mPrimes[MN] = { 2, 3, 5, 7, 13, 17, 19, 31, 61, 89 };
unsigned k, number[200];

void doubleN(void)
{ unsigned i, carry = 0, temp;
```

```
  for (i = 0; i < k; i++) {
    temp = number[i] * 2 + carry;
    number[i] = temp % 10;
    carry = temp / 10;
  }
  if (carry > 0)
    number[k++] = carry;
}

void print(void)
{
  unsigned i;
  for (i = k; i > 0; i--)
    printf("%u", number[i-1]);
  printf("\n");
}

void perfect(unsigned s, unsigned m)
{
  unsigned i;
  k = 1; number[0] = 1;
  for (i = 0; i < m; i++)
    doubleN();  /* това са делители от вида 2^i */
  number[0]--;  /* последната цифра със сигурност е измежду {2,4,8,6} */
  for (i = 0; i < m - 1; i++)
    doubleN();
  printf("%2u-то съвършено число е = ", s);
  print();      /* отпечатва поредното число */
}

int main(void) {
  unsigned i;
  for (i = 1; i <= MN; i++)
    perfect(i, mPrimes[i - 1]);
  return 0;
}
```

🖫 perfect.c

Резултат от изпълнението на програмата:

```
 1-то съвършено число е = 6
 2-то съвършено число е = 28
 3-то съвършено число е = 496
 4-то съвършено число е = 8128
 5-то съвършено число е = 33550336
 6-то съвършено число е = 8589869056
 7-то съвършено число е = 137438691328
 8-то съвършено число е = 2305843008139952128
 9-то съвършено число е = 2658455991569831744654692615953842176
10-то съвършено число е = 191561942608236107294793378084303638130997321548169216
```

Забележка: Очевидно теоремата по-горени дава само четни съвършени числа (всъщност, *всички* четни съвършени числа). Въпросът, дали съществуват нечетни съвършени числа, е все още без отговор (най-вероятно няма такива). Доказано е, че ако има такива, то те би трябвало да имат поне 300 цифри и множество делители.

Задачи за упражнение:

1. Да се докаже, че сумата от реципрочните стойности на делителите (включително 1 и самото число) на всяко съвършено число е 2. Например, за 6 имаме: 1/1 + 1/2 + 1/3 + 1/6 = 2.

2. Да се напише програма, която намира всички числа, за които сумата от реципрочните стойности на делителите (включително 1 и самото число) е 2. Съвършени ли са?

1.1.5. Биномни коефициенти, триъгълник на Паскал. Факторизация

Дефиниция 1.17. Нека е дадено n-елементно множество. Броят на всички възможни негови k-елементни подмножества се нарича *биномен коефициент* и се означава и пресмята по следния начин:

$$\binom{n}{k} = C_n^k = \frac{n.(n-1)...(n-k+1)}{k(k-1)...1} \qquad (*)$$

Ще отбележим, че означенията C_n^k и $\binom{n}{k}$ не са съвсем еквивалентни и второто е по-общо от първото, тъй като по принцип допуска реални стойности на n, при което няма комбинаторен смисъл. За нашите цели обаче тези нотации са еквивалентни и ние ще ги използваме паралелно. Ако n е естествено число, можем да запишем (*) като:

$$\binom{n}{k} = C_n^k = \frac{n!}{k!(n-k)!} \qquad (**)$$

Ще отбележим, че това е точно формулата, даваща броя на комбинациите без повторение на n елемента от клас k (*виж 1.3.3.*) и това съвсем не е случайно (*Защо?*).

Биномните коефициенти имат редица приложения, например в развитието на $(a+b)^n$, откъдето идва и името им:

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + ... + \binom{n}{n}a^0 b^n = \sum_{i=0}^{n}\binom{n}{i}a^{n-i}b^i$$

Съществуват редица свойства и формули, касаещи биномните коефициенти [*Knuth-1/1968*]. Някои от тях ще разгледаме в настоящия параграф.

В *таблица 1.1.5.* са показани стойностите на C_n^k, за $0 \le k \le n < 10$.

n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$	$\binom{n}{8}$	$\binom{n}{9}$
0	1	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
2	1	2	1	0	0	0	0	0	0	0
3	1	3	3	1	0	0	0	0	0	0
4	1	4	6	4	1	0	0	0	0	0
5	1	5	10	10	5	1	0	0	0	0
6	1	6	15	20	15	6	1	0	0	0
7	1	7	21	35	35	21	7	1	0	0
8	1	8	28	56	70	56	28	8	1	0
9	1	9	36	84	126	126	84	36	9	1

Таблица 1.1.5. Биномни коефициенти.

Някои от свойствата на биномните коефициенти проличават, ако анализираме по-внимателно *таблица 1.1.5.*, например:

$$\binom{n}{0} = \binom{n}{n} = 1 \qquad (1)$$

$$\binom{n}{k} = \binom{n}{n-k} \qquad (2)$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \qquad (3)$$

ако премахнем нулевите елементи от *таблица 1.1.5.* по-горе ще получим триъгълник, наречен *триъгълник на Паскал*:

$n = 0$						1					
$n = 1$					1		1				
$n = 2$				1		2		1			
$n = 3$			1		3		3		1		
$n = 4$		1		4		6		4		1	
$n = 5$	1		5		10		10		5		1

...

Фигура 1.1.5. Триъгълник на Паскал.

Първото и последното число във всеки ред е 1 (следва от свойства (1) и (2)), а всяко друго се получава като сума от двете числа над него (свойство (3)). Така коефициентите от n-тия ред можем да намерим, ако разполагаме с (n–1)-вия ред. Ако търсим $C_n^k = \binom{n}{k}$ и започнем от първия ред, можем да намерим последователно всички редове до n-тия и стойността в k-тата позиция ще бъде точно търсеното C_n^k. Забележете, че при пресмятане на стойностите в ред i ползваме само тези от ред i–1, но не и предишните.

Следва директна реализация на Си: последния пресметнат ред ще пазим в масив `lastLine[]`, а параметрите n и k за търсеното C_n^k са дефинирани в началото на програмата като константи.

```
#include <stdio.h>

/* Максимален размер на триъгълника */
#define MAXN 1000
const unsigned n = 7;
const unsigned k = 3;
unsigned long lastLine[MAXN + 1];

int main(void) {
  unsigned i, j;
  lastLine[0] = 1;
  for (i = 1; i <= n; i++) {
    lastLine[i] = 1;
    for (j = i - 1; j >= 1; j--)
      lastLine[j] += lastLine[j - 1];
  }
  printf("C(%u,%u) = %lu\n", n, k, lastLine[k]);
  return 0;
}
```

🖫 pascalt.c

Ще предложим друг алгоритъм, който се основава на формулата (*) и демонстрира полезна схема за съкращаване на рационални дроби с голям числител и знаменател. Ако опитаме директно да пресметнем по (**) $n!$ и $k!(n-k)!$, в много случаи се получават големи междинни резултати, докато крайният резултат съвсем не е толкова голямо число: например за $n = 100$ и $k = 2$ ще трябва да пресметнем 100! и 2!.98! (числа с над 150 цифри), а резултатът C_{100}^{2} е едва 4950.

Алгоритъм 2 за намирането на C_n^k *(с факторизация)*

1) Като пример ще разгледаме C_7^3. Ще представим числителя от формулата (*) като произведение от числа, всяко от които ще разложим на прости множители. Сега можем да получим и разлагането на цялото произведение: за избрания пример $7! = 1.2.3...7 = 1.2.3.2^2.5.(2.3).7 = 2^4.3^2.5^1.7^1$.
2) Аналогично разлагаме знаменателя: $3!.(7-3)! = 1.2.3.\ 1.2.3.2^2 = 2^4.3^2$.
3) Съкращаваме числителя и знаменателя: $\dfrac{2^4.3^2.5^1.7^1}{2^4.3^2} = \dfrac{5^1.7^1}{1}$.
4) След съкращаването извършваме останалите умноженията в числителя и получаваме търсения резултат: $5^1.7^1 = 35$.

Следва реализация на *алгоритъм 2*.

```
#include <stdio.h>
#define MAXN 500

unsigned long n = 7;
unsigned long k = 3;
unsigned long pN, primes[MAXN], counts[MAXN];

void modify(unsigned long x, unsigned how)
{ unsigned i;
  for (i = 0; i < pN; i++)
    if (x == primes[i]) {
      counts[i] += how;
      return;
    }
  counts[pN] = how;
  primes[pN++] = x;
}

void solve(unsigned long start, unsigned long end, unsigned long inc)
{ unsigned long prime, mul, how, i;
  for (i = start; i <= end; i++) {
    mul = i;
    prime = 2;
    while (mul != 1) {
      for (how = 0; mul % prime == 0; mul /= prime, how++);
      if (how > 0)
        modify(prime, inc * how);
      prime++;
    }
  }
}

unsigned long calc(void)
{ unsigned i, j;
```

```
  unsigned long result = 1;
  for (i = 0; i < pN; i++)
    for (j = 0; j < counts[i]; j++)
      result *= primes[i];
  return result;
}

int main(void) {
  printf("C(%lu,%lu)= ", n, k);
  pN = 0;
  if (n - k < k) k = n - k;
  solve(n - k + 1, n, 1); /* Факторизира числителя (n-k+1),...,n */
  solve(1, k, -1);        /* Факторизира знаменателя 1,...,k */
  printf("%lu\n", calc());
  return 0;
}
```

🖫 cnk.c

Подобен алгоритмичен подход (*пресмятане с факторизация*, т.е. извършване на действия с числа в разложен на прости множители вид) се прилага успешно не само когато се търси C_n^k, а винаги, когато се получават големи междинни стойности в числител и знаменател, докато самият резултат не е толкова голям.

Забележка: В горната програма при нужда прилагаме формулата за "обръщане" на входните данни:

$$\binom{n}{k} = \binom{n}{n-k}$$

При това, първо се извършват очевидните съкращения и едва *след това* — разлагането.

Задачи за упражнение:

1. Да се докажат свойства (1), (2) и (3), като се използва дефиницията на биномен коефициент. Например, доказателството на (1) може да се извърши по следния начин:

$$\binom{n}{0} = C_n^0 = \frac{n!}{0!(n-0)!} = \frac{n!}{0!n!} = 1$$

$$\binom{n}{n} = C_n^n = \frac{n!}{n!(n-n)!} = \frac{n!}{n!0!} = 1$$

2. Да се предложат комбинаторни доказателства на свойства (1), (2) и (3).

3. При предложения *Алгоритъм 1* (🖫pascalt.c) за намиране на C_n^k е необходим масив `lastLine[]` с n+1 елемента и, при по-големи стойности на n, заделянето на толкова памет ще бъде невъзможно. Възможно е да се модифицира по следния начин: вътрешният цикъл, попълващ поредния ред от триъгълника, да се изпълнява не от 1 до i, а от 1 до k, тъй като отрязъкът от триъгълника по-нататък не ни интересува. Да се обърне внимание, че когато k е близо до n, може да се приложи свойство (2) и да се търси C_n^{n-k}, вместо C_n^k.

1.1.6. Бройни системи и преобразуване

Най-общо *бройната система* бихме могли да определим като съвкупност от графични знакове и правила за представяне на числата. През различните епохи на историческото развитие на човечеството са използвани множество различни бройни системи, като най-широко разпрос-

транение днес има *арабската*. При нея за изобразяване на естествените числа се използват един или повече от графичните знакове 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9, наричани *цифри* (Следва да се отбележи, че съвременното *графично изображение* на цифрите в арабския свят се отличава от посочените по-горе десет графични знака, но става въпрос за същата бройна система.). Броят на цифрите, използвани за записване на числата, ще наричаме *основа* на системата. Така например арабската бройна система е *десетична* (използва 10 цифри). Съществуват и обобщения на понятието *основа*, като се допуска тя да бъде произволно (по модул различно от 0 и 1) реално (или дори комплексно) число със знак. В случай, че за основата p на бройната система имаме $p < 0$, отрицателните числа ще се записват без явно използване на знака "–".

За компютърната аритметика особена роля играят други системи — двоична, шестнадесетична и отчасти осмична. При двоичната бройна система за записване на числата се използват цифрите 0 и 1. Така например числото $11_{(10)}$ в двоична бройна система ще има вида $1011_{(2)}$ (Ще използваме скоби и умален шрифт, за да указваме основата на бройната система.). За записването на числата в шестнадесетична бройна система са необходими 16 цифри. Ето защо, освен цифрите от 0 до 9 от десетичната бройна система, се използват и главните латински букви A, B, C, D, E и F, със стойности съответно 10, 11, 12, 13, 14 и 15. Така например числото $254_{(10)}$ се представя в шестнадесетичен вид като $FE_{(16)}$.

Чрез средствата на математическия анализ може да се покаже, че оптималната бройна система е тази с основа e = 2,718281828... (*Неперовото число*. Въвежда се като граница на редицата $a_n = \left(1+\frac{1}{n}\right)^n$ при $n \to \infty$). Тук под *оптимална* се има предвид даваща най-добро съотношение между дължина на записа на число в съответната система и брой използвани цифри от системата.

Тъй като числото e е ирационално (не може да се представи като отношение на две цели числа), работата с тази система е неудобна и в този смисъл бихме могли да считаме за оптимални бройните системи с основа 3 и 2. От математическа гледна точка троичната бройна система е за предпочитане, тъй като числото 3 е по-близо до e. От чисто техническа гледна точка обаче се предпочита двоичната бройна система. Двоичната бройна система е изключително удобна за техническа реализация — на *0* се съпоставят напрежение 0 V, а на *1* — 5 V. Практически всички съвременни компютри работят в двоична бройна система. В миналото в СССР са правени и реализации на машини, работещи с троична симетрична бройна система, например Сетунь, създадена през 1960 г. в Москва, използваща цифрите –1, 0 и 1.

Значението на шестнадесетичната система се определя от това, че 16 е степен на 2 ($16=2^4$). За кодиране на числата от 0 до 15 са необходими 4 бита (двоични цифри — 0 и 1). С цел по-лесната им обработка тези двоични разряди се групират в байтове — групи от по 8 бита. Лесно се вижда, че в един байт могат да се запишат числата от 0 до 255, т.е. 256 на брой различни числа. Тъй като $256 = 16^2$, то всеки байт може да се представи с две последователни шестнадесетични цифри. Преминаването от двоична в шестнадесетична бройна система става просто — двоичното число се разбива отдясно-наляво на *тетради* (4 последователни бита), като при нужда отляво се допълва с нули, след което всяка тетрада се преобразува поотделно в съответна шестнадесетична цифра. Ще илюстрираме току-що изложения метод, като преобразуваме числото $10101000110111_{(2)}$ в шестнадесетична бройна система. Разделяме го на тетради:

```
10|1010|0011|0111
```

и го допълваме с нули:

```
0010|1010|0011|0111
```

Сега кодираме всяка тетрада със съответната ѝ шестнадесетична цифра и получаваме $2A37_{(16)}$.

Аналогично стоят нещата при преминаване от двоична в осмична бройна система, като в този случай числото се разбива на *триади* (тройки последователни двоични разряди). По-долу ще разгледаме по-общ метод за преминаване между бройни системи.

Забележете, че от записа на числото не може еднозначно да се определи в каква бройна система е записано. Така например, за числото 153 можем да твърдим единствено, че това е *поне*

6. Всъщност, абстрахирайки се от значението на цифрите 1, 5 и 3, можем да кажем, че числото е записано в система с основа *поне 3*, тъй като са налице 3 *различни* цифри, които можем да интерпретираме като 0, 1 и 2. С цел избягване на подобни недоразумения е прието в скоби след числото с по-ситен шрифт, както по-горе, да се указва основата на използваната система.

Арабската бройна система е позиционна, т.е. стойността на цифрите не е строго определена и се изменя в зависимост от позицията, на която се намира съответната цифра. Така например в числото 123 цифрата 3 има стойност 3, докато в числото 34 има стойност 30. Съществуват и *непозиционни* бройни системи (*виж 1.1.7.*), при които стойността на всеки знак е строго фиксирана и не зависи от позицията му.

Методите за записване на число в дадена бройна система са най-общо два: *цифров* и *полиномиален*. Обикновено се предпочита цифровият запис като по-кратък. При него съответните цифри на числото се записват долепени една до друга, като стойността им нараства отдясно-наляво в геометрична прогресия с частно p, където p е основа на използваната бройна система. Цифровият запис има вида $a_n a_{n-1} \ldots a_{0\,(p)}$, където a_i $(1 \leq i \leq n)$ е цифра. В полиномиален запис числото има вида:

$$A = a_n p^n + a_{n-1} p^{n-1} + \ldots\ a_1 p + a_0$$

Полиномиалният запис е полезен при преминаването от една бройна система в друга, както ще видим по-долу.

Задачи за упражнение:

1. Защо основата на бройната система трябва да бъде по модул различна от 0 и 1?
2. Да се представят числата 17 и –17 в бройна система с основа: 2; 8; 16.
3. Да се представят числата 17 и –17 в бройна система с основа –2; –8; –16.
4. Без да се преминава през десетична система, да се преобразуват в шестнадесетична бройна система двоичните числа: 111, 110100, 1110100101, 10010101, 10101010101 и 10111110101.
5. Без да се преминава през десетична система, да се преобразуват в осмична бройна система двоичните числа: 11, 11001, 1010101, 111111, 1010101000, 10101101000 и 11010111000.
6. Как изглежда полиномиалният запис на число в симетрична бройна система?
7. Да се представят числата 17 и –17 в троична симетрична бройна система.

- Преминаване от десетична в p-ична бройна система

Изхождайки от представянето на число в полиномиален вид, можем да формулираме следния алгоритъм за преминаване от десетична в p-ична бройна система: делим числото A на p с частно и остатък, докато A не стане 0, след което записваме остатъците в ред, обратен на тяхното получаване (*Защо?*). Така например при преобразуване на числото 29 в двоична бройна система получаваме (с курсив под всяко деление е записан съответният остатък):

```
29:2=14:2=7:2=3:2=1:2=0
  1    0   1   1   1
```

Вземаме получените остатъци в обратен ред и получаваме: $29_{(10)} = 11101_{(2)}$.

Приведената функция `convert()` извършва съответно преобразуване и връща резултата като символен низ:

```
char getChar(char n) /* Връща символа, съответстващ на n */
{ return (n < 10) ? n + '0' : n + 'A' - 10; }

void reverse(char *pch)
{ char *pEnd;
  for (pEnd = pch + strlen(pch) - 1; pch < pEnd; pch++, pEnd--) {
    char c = *pch;
    *pch = *pEnd;
    *pEnd = c;
```

```
  }
}

void convert(char *rslt, unsigned long n, unsigned char base)
/* Преобразува цялото десетично число n (n >= 0) */
/* в бройна система с основа base */
{ char *saveRslt = rslt;
  while (n > 0) {
    *rslt++ = getChar((char)(n % base));
    n /= base;
  }
  *rslt = '\0';
  reverse(saveRslt);
}
```

🖫 base.c

Ако числото е по-малко от 1, за него отново съществува записване в полиномиален вид, като в този случай ще се получи полином на *отрицателните* степени на *p*. При преобразуването на *A* в *p*-ична бройна система нещата стоят точно обратно на случая на естествено число: *Умножаваме A* по *p*, отделяме *целите части* и ги записваме в реда на получаването им. Преобразуването на 0,125 в двоична бройна система например изглежда така:

0,125.2=***0***,25.2=***0***,5.2=***1***,0

Получаваме $0{,}125_{(10)} = 0{,}001_{(2)}$. Тук възниква един допълнителен проблем — записът на *A* в *p*-ична бройна система може да се окаже *безкрайна* (не)периодична дроб. Ето защо би било разумно да въведем някаква точност — максимален брой на цифрите след десетичната запетая. Функцията `convertLessThan1()` извършва преобразуване на числото *A* ($0 \leq A < 1$) с точност `cnt` на брой цифри след десетичната запетая, като следващите цифри се игнорират.

```
void convertLessThan1(char *rslt,
                      double n,
                      unsigned char base,
                      unsigned char cnt)
/* Преобразува числото 0 <= n < 1 в бройна система с основа base
   с не повече от cnt на брой цифри след десетичната запетая */
{
  while (cnt--) {
    /* Дали не сме получили 0? */
    if (fabs(n) < EPS) break;
    /* Получаване на следващата цифра */
    n *= base;
    *rslt++ = getChar((char)(int)floor(n));
    n -= floor(n);
  }
  *rslt = '\0';
}
```

🖫 base.c

След като реализирахме поотделно функции за преобразуване за случаите $A > 1$ и $0 \leq A < 1$, остава да ги обединим по естествен начин, позволявайки преобразуването на произволно реално число, включително и отрицателно. Получаваме функцията `convertReal()`:

```
void convertReal(char *rslt, double n,
                 unsigned char base, unsigned char cnt)
/* Преобразува реалното число n в бройна система с основа base */
{ double integer, fraction;
```

```
    /* Намиране на знака */
    if (n < 0) {
      *rslt++ = '-';
      n = -n;
    }

    /* Разбиване на цяла и дробна част */
    fraction = modf(n, &integer);

    /* Конвертиране на цялата част */
    convert(rslt, (unsigned long)integer, base);

    /* Поставяне на десетична точка (по принцип запетая...) */
    if ('\0' == *rslt) *rslt++ = '0';
    else rslt += strlen(rslt);
    *rslt++ = '.';

    /* Конвертиране на дробната част */
    convertLessThan1(rslt, fraction, base, cnt);
    if ('\0' == *rslt) {
      *rslt++ = '0';
      *rslt = '\0';
    }
}
```

🖫 base.c

Задачи за упражнение:

1. Да се запише числото 157 в бройна система с основа: 3;5;7;14.
2. Да се запише числото 0,321 в бройна система с основа: 3;5;7;14.
3. Да се запише числото 157,321 в бройна система с основа: 3;5;7;14.
4. Да се обоснове предложеният алгоритъм за преобразуване на естествено число от десетична в p-ична бройна система.
5. Да се обоснове предложеният алгоритъм за преобразуване на дробната част на десетична дроб число от десетична в p-ична бройна система.

- Преминаване от p-ична в десетична бройна система. Формула на Хорнер

Преминаването от p-ична в десетична бройна система се свежда до пресмятане стойността на полинома от полиномиалния запис на A, като всички аритметични действия се извършват в десетична бройна система. В случай, че A е естествено число, например $12734_{(8)}$, алгоритъмът изглежда така:

$$12734_{(8)} = 1.8^4 + 2.8^3 + 7.8^2 + 3.8 + 4 = 5596_{(10)}$$

Забележете, че при пресмятането ни бяха необходими 10 умножения. Бихме могли значително да намалим този брой, ако *пазим* вече пресметнатите степени на 8, вместо да ги *пресмятаме* всеки път отново. Съществува обаче друг по-общ метод, който ни позволява пресмятане стойността на полином от n-та степен с не повече от n умножения — *формула на Хорнер*. Идеята е вместо вида (1) при изчислението да се използва (2).

(1) $P_n(x) = a_0x^n+a_1x^{n-1}+\ldots+a_{n-1}x+a_n$

(2) $P_n(x) = a_n+x(a_{n-1}+x(a_{n-2}+\ldots+x(a_2+x(a_1+xa_0))\ldots))$ — формула на Хорнер

Примерна реализация на този подход е функцията `calculate()`:

```
char getValue(char c) /* Връща стойността на символа c */
{  return (c >= '0' && c <= '9') ? c - '0' : c - 'A' + 10; }

unsigned long calculate(const char *numb, unsigned char base)
/* Намира десетичната стойност на числото numb, зададено в бройна система
   с основа base, numb >= 0 */
{ unsigned long result;
  for (result = 0; '\0' != *numb; numb++)
    result = result*base + getValue(*numb);
  return result;
}
```

🖫 base.c

По аналогичен начин можем да реализираме и преобразуване на неотрицателно число, по-малко от 1. Този случай не се различава съществено от случая на преобразуване на естествено число. Единствената разлика е, че тук полиномът е от *отрицателни* степени на *p*.

```
double calculateLessThan1(const char *numb, unsigned char base)
/* Намира десетичната стойност на числото numb (0 < numb < 1),
   зададено в бройна система с основа base */
{ const char *end;
  double result;
  for (end = numb + strlen(numb) - 1, result = 0.0; end >= numb; end--)
    result = (result + getValue(*end)) / base;
  return result;
}
```

🖫 base.c

Накрая, остава да обединим двата случая, за да получим функция за преобразуване на произволно реално число със знак:

```
double calculateReal(char *numb, unsigned char base)
/* Намира десетичната стойност на реалното число numb, зададено
   в бройна система с основа base */
{ char *pointPos;
  char minus;
  double result;

  /* Проверка за минус */
  if ('-' == *numb) {
    minus = -1;
    numb++;
  }
  else
    minus = 1;

  if (NULL == (pointPos = strchr(numb, '.')))
    return calculate(numb, base); /* Няма дробна част */

  /* Пресмятане на цялата част */
  *pointPos = '\0';
  result = calculate(numb, base);
  *pointPos = '.';

  /* Прибавяне на дробната част */
  result += calculateLessThan1(pointPos+1, base);

  return minus*result;
```

```
}
```
🖫 base.c

Задачи за упражнение:

1. Да се намери десетичният запис на числото $126_{(8)}$; $10101_{(2)}$; $3F2B_{(16)}$; $3CB_{(14)}$.
2. Да се намери десетичният запис на числото $0{,}233_{(8)}$; $0{,}01_{(2)}$; $0{,}34_{(16)}$; $0{,}2A_{(14)}$.
3. Да се намери десетичният запис на числото $126{,}233_{(8)}$; $10101{,}01_{(2)}$; $3F2B{,}34_{(16)}$; $3CB{,}2A_{(14)}$.
4. Да се обоснове предложеният алгоритъм за преобразуване от p-ична в десетична бройна система.

1.1.7. Римски цифри

Разгледаните по-горе бройни системи, въпреки че се различаваха по основата си, принадлежаха към един и същ клас: *позиционни*. При тях една и съща цифра имаше различни стойности в зависимост от *позицията*, на която се намира. Съществуват обаче и други бройни системи, наречени *непозиционни*, при които стойността на отделните цифри е фиксирана и не зависи от позицията, на която се намира. Класически пример в това отношение е *римската* бройна система. При нея за представяне на естествените числа в интервала [1;3999] се използват 7 главни латински букви (в скоби са посочени съответните им стойности): I(1), V(5), X(10), L(50), C(100), D(500) и M(1000). Така например числото 1989 се записва като MCMLXXXIX.

- Представяне на десетично число с римски цифри

Няма да се спираме в детайли върху свойствата на римската бройна система. Ще отбележим само, че когато по-малка по стойност цифра стои пред по-голяма, то стойността ѝ се вади от тази на голямата. В противен случай се прибавя:

$$XI = 10 + 1 = 11$$
$$IX = 10 - 1 = 9$$

Всяка цифра може да бъде следвана от произволна по-малка от нея. Съществуват обаче ограничения за броя на последователните цифри с еднаква стойност, както и по отношение на това коя по-малка цифра коя по-голяма може да предхожда. Това се вижда добре от *таблица 1.1.7*.

1(I)	2(II)	3(III)	4(IV)	5(V)	6(VI)	7(VII)	8(VIII)	9(IX)
10(X)	20(XX)	30(XXX)	40(XL)	50(L)	60(LX)	70(LXX)	80(LXXX)	90(XC)
100(C)	200(CC)	300(CCC)	400(CD)	500(D)	600(DC)	700(DCC)	800(DCCC)	900(CM)

Таблица 1.1.7. Запис на някои числа с римски цифри.

Таблица 1.1.7. съдържа представянията на някои десетични числа с римски цифри. Забелязваме, че някои от числата се записват по сходен начин, като разликата е единствено в използваните цифри. При това подобно сходство наблюдаваме не при случайни числа, без връзка помежду им, а например при 3(III), 30(XXX) и 300(CCC). Същото е налице и при 7(VII), 70(LXX) и 700(DCC). Изобщо числата, принадлежащи на една и съща колона на таблицата, притежават такова свойство. Последното означава, че римската бройна система все пак не е чак толкова различна и по някакъв особен начин (за наше щастие) е свързана с именно с десетичната бройна система — оказва се, че всяко римско число можем да "сглобим" от римските числа, представящи цифрите на единиците, десетиците, стотиците и хилядите на разглежданото десетично число. За целта е достатъчно да се разгледат отдясно-наляво цифрите на десетичното число и да се преобразуват една по една в римски цифри, след което да се долепят. Такъв именно е и подходът, възприет в приведената по-долу функция `decimal2Roman()`.

При преобразуването на число от и в римска бройна система ще използваме следните декларации:

```
    const char *roman1_9[]={"", "A", "AA", "AAA", "AB", "B", "BA", "BAA", "BAAA",
"AC"};
    const char *romanDigits[] = {"IVX", "XLC", "CDM", "M" };
```

Последователностите от символи *A*, *B* и *C* представляват своеобразен *модел* на числата от съответния стълб на *таблица 1.1.7.*, върху който ще се налагат истинските римски цифри.

```
void getRomanDigit(char *rslt, char x, unsigned char power)
{ const char *pch;
  for (pch = roman1_9[x]; '\0' != *pch; pch++)
      *rslt++ = romanDigits[power][*pch - 'A'];
  *rslt = '\0';
}

char *decimal2Roman(char *rslt, unsigned x)
{ unsigned char power;
  char buf[10];
  char oldRslt[MAX_ROMAN_LEN];
  for (*rslt = '\0', power = 0; x > 0; power++, x /= 10) {
      getRomanDigit(buf, (char)(x % 10), power);
      strcpy(oldRslt, rslt);
      strcpy(rslt,buf);
      strcat(rslt,oldRslt);
  }
  return rslt;
}
```
🖫 rom2dec.c

Задачи за упражнение:

1. Да се запишат в римска бройна система числата: 10; 19; 159; 763; 1991; 1979; 1997; 2002.
2. Да се запишат в римска бройна система числата: 0; –10; 0,28; 3,14; 1/7.
3. Да се обоснове предложеният алгоритъм за преобразуване на десетично число в римска бройна система.

- Преобразуване на римско число в десетично

Тук нещата стоят по-просто. Този път ще разглеждаме цифрите отляво-надясно, ще пресмятаме десетичната им стойност, след което ще я натрупваме в някаква сума. Ако се окаже, че предходната цифра има по-малка десетична стойност от текущата, това ще означава, че предходната цифра е трябвало не да се прибави, а да се *извади*. Тъй като в такъв случай ние вече ще сме я прибавили веднъж, то следва я извадим 2 пъти:

```
if (value > old) rslt -= 2*old;
```

Например при числото 19(XIX) цифрата *I* трябва да се извади от сумата, докато при 21(XXI) — да се прибави.

Възниква още един проблем: как да проверим дали подадената като параметър на `roman2Decimal()` последователност от римски цифри е *коректно* римско число? Тази проверка е сложна, но бихме могли значително да опростим нещата. За целта е достатъчно да извършим обратното преобразуване и да сравним получения резултат с изходното число. Подробностите могат да се видят от предложената реализация.

```
unsigned roman2Decimal(const char *roman, char *error)
{ unsigned rslt, value, old;
```

```
  const char *saveRoman = roman;
  char buf[MAX_ROMAN_LEN];

  old = 1000; rslt = 0;
  while ('\0' != *roman) {
    switch (*roman++) {
      case 'I': value =    1; break;
      case 'V': value =    5; break;
      case 'X': value =   10; break;
      case 'L': value =   50; break;
      case 'C': value =  100; break;
      case 'D': value =  500; break;
      case 'M': value = 1000; break;
      default:
        *error = 1;
        return (unsigned)(-1);
    }
    rslt += value;
    if (value > old)
      rslt -= 2*old;
    old = value;
  }
  return (*error = strcmp(saveRoman,decimal2Roman(buf,rslt)))
        ? (unsigned)(-1) : rslt;
}
```

🖫 dec2rom.c

Задачи за упражнение:

1. Да се запишат в десетична бройна система римските числа: DCLXXXIV, DCCLXIV, LX, LXX, LXXX, XL, XXL, XXXL.
2. Да се обоснове предложеният алгоритъм за преобразуване на римско число в десетична бройна система.
3. Да се напише програма за проверка дали дадена последователност от римски цифри е коректно записано римско число, без да се преминава през десетична бройна система. За целта наблюдавайте внимателно закономерностите в *таблица 1.1.7*.

1.2. Рекурсия и итерация

Добре известна сентенция от програмисткия фолклор е следната: "За да дефинираме понятието *рекурсия*, първо трябва да дефинираме понятието *рекурсия*". В света на *UNIX* съществуват множество подобни "дефиниции" (Например *GNU* е съкращение от *GNU is Not UNIX*, *WINE* — от *WINE Is Not an Emulator*, и др.).

Дефиниция 1.18. Един обект се нарича *рекурсивен*, ако се съдържа в себе си, или е дефиниран чрез себе си.

В компютърната информатика рекурсията е една от най-мощните техники на програмиране: чрез нея елегантно се дефинират алгоритми, съставят се удобни и гъвкави структури от данни.

В много случаи използването на рекурсия може да доведе до неефективни алгоритми. Целта на настоящия параграф, освен запознаване с рекурсията като фундаментален алгоритмичен подход, ще бъде и изследване на въпроса кога прилагането ѝ е полезно на практика.

Средство за рекурсивно изразяване в една програма на Си са функциите. Ако в тялото на дадена функция *P* се извършва обръщение към себе си, казваме, че тя е *пряко рекурсивна*. Възмо-

жен е и “по-сложен” сценарий: функцията P_1 да се обръща към функцията P_2, P_2 към P_3, … , P_n към P_1. В такъв случай казваме, че P_1, както и P_2, P_3, ..., P_n, са *непряко* (*косвено*) рекурсивни функции.

При програмирането на рекурсивни функции трябва да се обърне внимание на няколко важни условия:

- Задачата, която разглеждаме, трябва да се разбива на подзадачи, за които (рекурсивно) може да се приложи същият алгоритъм. Комбинирането на решенията на всички подзадачи трябва да води до решение на *изходната* задача.
- Реализирайки рекурсивен алгоритъм, трябва да бъдем сигурни, че след краен брой стъпки ще достигнем до *конкретен* резултат, т.е. трябва да са налице *краен* брой *прости случаи* (поне един), чието решение може да бъде намерено директно. Прието е, те да се наричат *дъно на рекурсията*.
- Всички подзадачи на задачата трябва да се "стремят" към някой от тези прости случаи, т.е. след краен брой рекурсивни извиквания да се достига дъното на рекурсията (Аналогично, при рекурентни формули, както и при индуктивни разсъждения, е необходимо да има база.).

Първите няколко примера за рекурсивни функции на Си, които ще приведем, ще демонстрират пресмятане на рекурентно дефинирани математически функции.

1.2.1. Факториел

Дефиницията на факториел, както и итеративно пресмятане на функцията, разгледахме в *1.1.1*. За да реализираме рекурсивна функция, пресмятаща факториел, трябва да бъдем сигурни, че са изпълнени двете ключови условия:

- Дъното на рекурсията е простият случай $n = 0$ — тогава стойността на функцията е 1.
- Във всички останали случаи решаваме подзадачата за $n-1$ и умножаваме резултата по n. Така рекурсивният параметър намалява монотонно и дъното на рекурсията ще бъде достигнато след краен брой стъпки (между n и 0 има краен брой естествени числа).

Следва функция на Си, пресмятаща рекурсивно n!:

```
#include <stdio.h>

const unsigned n = 6;

unsigned long fact(unsigned i)
{ if (i < 2) return 1;
  return i * fact(i - 1);
}

int main(void) {
  printf("%u! = %lu \n", n, fact(n));
  return 0;
}
```

🖫 factrec.c

По аналогичен начин можем да пресметнем рекурсивно сумата на първите n естествени числа (директно по рекурентната дефиниция от *1.1.1.*):

```
unsigned long sum(unsigned n) {
  if (0 == n) return 0;
  else return n + sum(n - 1);
}
```

В изложените реализации могат да се забележат няколко неща: когато е дадена рекурентно дефинирана математическа функция, реализирането на съответна рекурсивна функция на Си не представлява трудност, докато *итеративен* (последователен) алгоритъм за решение в някои случаи, особено когато функцията е по-сложна, не е толкова очевиден. (*виж 1.2.2.*)

От друга страна, изразходваната памет е повече: при всяко рекурсивно извикване в стека се заделя нова памет за аргументите и за локалните променливи, както и за връщаните от функцията резултати. В примерите по-горе ще бъдат необходими *n*.`sizeof(unsigned)` байта памет за *n* рекурсивни извиквания. За сравнение: програмата за намиране на *n*! от *1.1.1.* използваше единствена променлива, в която се натрупваше резултатът, т.е. многократно по-малко памет. В тези примери обаче паметта не е критичен фактор, тъй като поради бързото нарастване на *n*! много по-скоро ще получим препълване на целочисления тип `unsigned long`, отколкото недостиг на памет.

1.2.2. Редица на Фибоначи

Числата на Фибоначи са един от любимите примери на редица книги и ръководства по алгоритми за елегантността на рекурсията като програмистка техника. Поради тази причина, на моменти няма да устояваме на изкушението да се впускаме в допълнителни подробности относно тяхната история, природа и свойства, което, вярваме, няма да отегчи читателя.

Фигура 1.2.2а. Числата на Фибоначи в архитектурата и изкуството.

Всеки, занимавал се поне малко с програмиране, непременно се е сблъсквал с числата на Фибоначи. Защо ли? Съществува широк клас от алгоритми и задачи по програмиране, на пръв

поглед несвързани по никакъв начин едни с други, в чиято основа стоят именно тези числа. Едва ли ще бъде преувеличено, ако кажем, че присъствието и ролята, която играят те в информатиката, математиката и въобще в природата, е наистина потресаваща. Стига да знае как да гледа, човек може да открие числата на Фибоначи навсякъде около себе си: започвайки от начина, по който се подреждат ятата птици при полет, през шишарките и поемите, слънчогледовите пити и симфониите, античното изкуство (*виж фигура 1.2.2а.*) и съвременните компютри, та чак до Слънчевата система и борсите.

Какво всъщност представляват тези прочути числа? Това са членовете на следната числова редица:

`0,1,1,2,3,5,8,13,21,34,55,89,...,`

Всеки член на редицата се получава като сума от предходните два, като първите два члена по дефиниция са съответно 0 и 1, т.е. в сила са формулите:

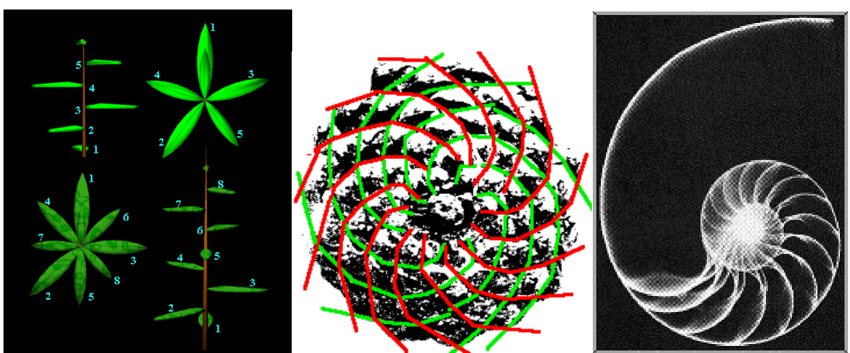
$F_0 = 0,$
$F_1 = 1,$
$F_i = F_{i-1} + F_{i-2}$

Фигура 1.2.2б. Леонардо Фибоначи.

Пръв изследовател на горната последователност е Леонардо Пизано (Леонардо от Пиза), известен повече като Леонардо Фибоначи (*Filius Bonaccii*, т.е. син на Боначо), който през 1202 г. в книгата си *Liber Abacci* (Книга за броенето) предлага прочутата си *задача за зайците*. Задачата се състои в намиране броя на зайците, които ще се получат от една двойка за една година при следните условия:

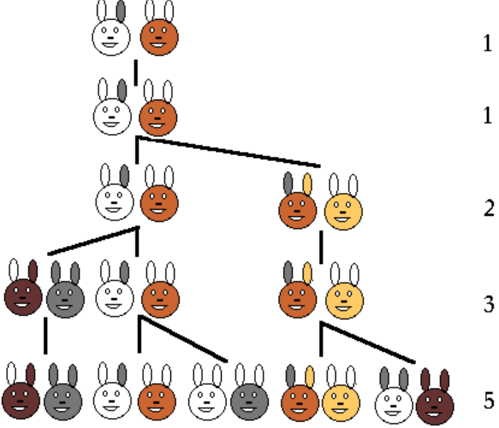
- всяка двойка плодоносни зайци дава прираст два заека на месец;
- новите зайци стават плодоносни на едномесечна възраст;
- зайците не умират никога.

И така, след месец ще имаме две двойки зайци, след още един — 3 двойки, на следващия месец ще имаме вече 5 двойки (една нова двойка от първоначалните зайци и една — от двойката, получена през първия месец), на следващия месец ще имаме общо 8 двойки и т.н. Ясно е, че описаният процес може да продължи до безкрайност. (*виж фигура 1.2.2г.*).



Фигура 1.2.2в. Числата на Фибоначи в природата.

По-късно през 1611 г. Кеплер, непознавайки изследванията на Фибоначи, преоткрива редицата във връзка със своите изследвания върху начина на разположение на листата на някои растения върху стъблото (*фигура 1.2.2в.*). Изобщо, както по-горе споменахме, числата на Фибоначи се срещат изключително често в природата, най-вероятно в резултат на процеси, подобни на тези от задачата на Фибоначи за зайците.



Фигура 1.2.2г. Задачата за зайците.

Каква е връзката между числата на Фибоначи и алгоритмите? Първата връзка е самият Леонардо Фибоначи, който старателно е изучавал трудовете на ал-Хорезми (от чието име произхожда думата алгоритъм, както споменахме в уводната глава). По-късно, през 1845 г. Ламе използва последователността на Фибоначи в процеса на изследване на широко известния алгоритъм на Евклид (*виж 1.2.3.*) за намиране на най-голям общ делител. Оттук нататък числата на Фибоначи постепенно започват да заемат подобаващото им се място при разработването и изследването на различни алгоритми.

Интересна е връзката между редицата на Фибоначи и числото

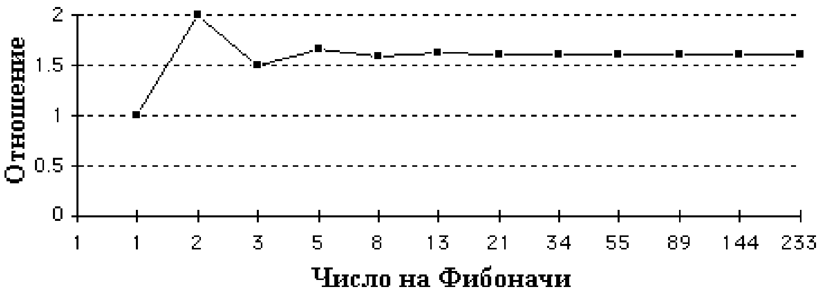
$$\phi = \frac{1+\sqrt{5}}{2} = 1{,}61803$$

Числото ϕ има много интересна история и е било известно на хората от най-дълбока древност. Евклид го нарича *отношение на крайното към средното* и го дефинира като отношението на двойка числа A и B, за които е в сила пропорцията:

$$\frac{A}{B} = \frac{A+B}{A} (= \phi)$$

По-късно в епохата на Ренесанса числото ϕ е било считано за *божествена пропорция*, а през $XIX^{-и}$ век получава окончателното си днешно наименование: *златно сечение*. Възприетото обозначение ϕ произхожда от първата буква от името на древногръцкия скулптор Фидий, който често е използвал тази пропорция в скулптурите си. Не е трудно да се забележи сходството между предложените по-горе дефиниции на числата на Фибоначи и на златното сечение. Оказва се, че отношението на две последователни числа на Фибоначи клони към златното сечение ϕ, като понякога именно така се дефинира — като граница на отношението на две последователни числа на Фибоначи. (*виж фигура 1.2.2д.*)

Забележка: Преди да преминем към разглеждането на алгоритмите за намиране числата на Фибоначи, следва да отбележим, че понякога се среща и друга дефиниция, съгласно която нулевият член на редицата на Фибоначи е 1, а не 0. Разбира се, последното не е толкова съществено и читателят при нужда би могъл без особени усилия да промени приведените по-долу програмни реализации в съответствие с предпочитаната дефиниция.



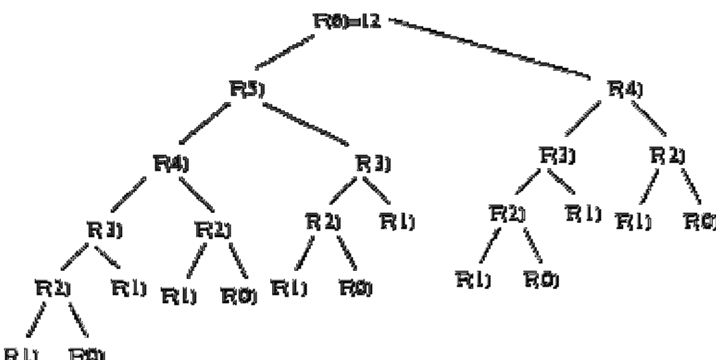
Фигура 1.2.2д. Отношение на две последователни числа на Фибоначи.

Може би най-естествената реализация на функция, намираща n-тото число на Фибоначи, получаваме, изхождайки директно от приведената по-горе рекурентна дефиниция:

```
unsigned long fib(unsigned n)
{ if (n < 2) return 1;
  else return fib(n - 1) + fib(n - 2);
}
```

🖫 fibrec.c

Макар и достатъчно проста и на пръв поглед естествена, подобна реализация е изключително неефективна. Читателят би могъл да изпробва горната функция, например за $n = 40$. Всяко рекурсивно обръщение на функцията, извън тривиалните (при $n = 0$ и 1), води след себе си още две. Така дървото на подзадачите расте експоненциално. При това абсолютно безсмислено, тъй като през по-голямата част от времето се пресмятат вече пресметнати стойности на функцията (*виж фигура 1.2.2е.*).



Фигура 1.2.2е. Дърво на извикванията на рекурсивната функция.

Числата на Фибоначи са класически пример, при който използването на рекурсия е неудачно. Проблемът произтича от това, че се правят напълно излишни изчисления, т.е. много от членовете на редицата се пресмятат неколкократно. Така например, при изчисляването на F_{10} ще пресметнем F_8 два пъти — веднъж при изчислението на $F_{10} = F_9 + F_8$ и после — при $F_9 = F_8 + F_7$. На по-дълбоките нива на рекурсията положението е още по-лошо. Тук рекурсията би могла да даде значително по-добри резултати, разбира се, за сметка на необходима допълнителна памет. Методът (наречен *memoization*) е разгледан в *глава 8 — Динамично оптимиране*.

Не е трудно да се забележи, че бихме могли и да не използваме рекурсия, пресмятайки числата на Фибоначи последователно: започвайки от първото пазим само последните два пресметнати члена на редицата, тъй като само те ще ни бъдат необходими за получаването на следващия член. Така, получаваме следния итеративен алгоритъм:

```
unsigned long fibIter(unsigned n)
{ unsigned long fn = 1, fn_1 = 0, fn_2;
  while (n--) {
    fn_2 = fn_1;
    fn_1 = fn;
    fn = fn_1 + fn_2;
  }
  return fn_1;
}
```
🖫 fib2.c

Не е трудно да се забележи, че всъщност нямаме нужда от три променливи и нещата могат да се решат само с две така:

```
unsigned long fibIter2(unsigned n)
{ unsigned long f1 = 0, f2 = 1;
  while (n--) {
    f2 = f1 + f2;
    f1 = f2 - f1;
  }
  return f1;
}
```
🖫 fib2.c

В случаите, когато търсим първите *n* числа на Фибоначи, най-добре е да се използва последният вариант с извеждане на междинните резултати. Как обаче стоят нещата, ако търсим *n*-тото число на Фибоначи?

Съществува нерекурентна формула за общия член на редицата на Фибоначи, известна като формула на Моавър (*виж 1.4.8.*):

$$F_n = \frac{1}{\sqrt{5}}.\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

Формулата на Моавър на пръв поглед изглежда по-ефективна от итеративния вариант — тук n-тият член на редицата като че ли може да се намери директно. За съжаление използването ѝ е свързано с работа с реални числа, изисква коренуване и не по-малко тежката операция степенуване, което променя нещата. Все пак за достатъчно големи n формулата на Моавър е за предпочитане, особено ако при повдигането в степен се използва бърз метод за степенуване (*виж 7.3.*, [*Наков-1998c*]). Ще отбележим, че коренуването на 5 може да се извърши предварително и резултатът да се зададе като константа в програмата. По този начин ще се избегне пресмятането му в хода на изпълнение, което е бавен итеративен процес, тъй като става с редове.

Разглеждането на алгоритмите за намиране на n-тото число на Фибоначи ще завършим в *глава 8* (*виж 8.1.*), където освен приложението на специфичните за техниката *Динамично оптимиране* методи ще видим и как те се комбинират с формулата на Моавър и бързото степенуване.

Задачи за упражнение:

1. Да се покаже, че отношението на две последователни числа на Фибоначи клони към златното сечение.
2. Да се докаже еквивалентността на двете итеративни реализации на функция за намиране на n-тото число на Фибоначи: с три и с две променливи.

1.2.3. Най-голям общ делител. Алгоритъм на Евклид

Най-големият общ делител (*НОД*) на две числа е друг добър пример за илюстриране на рекурсията като програмистка техника.

Дефиниция 1.19. Дадени са две естествени числа a и b. Казваме, че d е *най-голям общ делител* на a и b, ако е максималното естествено число, което дели едновременно a и b. Записано формално, последното означава да бъдат изпълнени следните две условия:

1) $d|a$, $d|b$
2) Ако $d_1|a$ и $d_1|b$, то следва че $d_1|d$.

В дефиницията сме използвали нотация за делимост: $d|a$ означава, че d дели целочислено a без остатък. Двете условия на дефиницията могат да бъдат интерпретирани и по следния начин: Всеки общ делител d_1 на a и b дели и d.

Ще бележим най-големия общ делител на a и b с $НОД(a,b)$ или просто (a,b).

Примери: $(12, 8) = (8, 12) = (8, 4) = 4$, $(1, 10) = (7, 10) = 1$.

Дефиниция 1.20. Ако *НОД* на две цели положителни числа е равен на 1, то числата се наричат *взаимно прости*.

За най-голям общ делител на повече от две числа е валидна следната формула:

$$НОД(a_1, a_2, ..., a_n) = НОД(НОД(a_1, a_2, ..., a_{n-1}), a_n)$$

Ще разгледаме два алгоритъма за намиране на $НОД(a,b)$, носещи името на Евклид. Първият от тях се нарича *алгоритъм на Евклид с изваждане*:

1) Ако $a > b$, се изпълнява 4), в противен случай се изпълнява 2).
2) Ако $a = b$ следва, че сме намерили $НОД(a,b)$ — това е стойността на b и приключваме. Ако $a \neq b$ изпълняваме 3).

3) Присвояваме $b = b - a$ и се връщаме на стъпка 1).
4) Присвояваме $a = a - b$ и се връщаме на стъпка 1).

Няма да се спираме по-подробно върху този алгоритъм, тъй като следващият, който ще разгледаме, е доста по-ефективен. Нарича се *алгоритъмът на Евклид с деление* (за него вече стана дума и в увода на книгата):

Алгоритъм на Евклид с деление

Разделяме целочислено a на b, при което получаваме частно q_1 и остатък r_1. По-нататък делим b на получения остатък, след което делим r_1 на остатъка r_2 и т.н., докато остатъкът стане равен на нула:

$$a = q_1.b + r_1$$
$$b = q_2.r_1 + r_2$$
$$r_1 = q_3.r_2 + r_3$$
$$\ldots$$
$$r_{k-1} = q_{k+1}.r_k + r_{k+1}, \text{ като } r_{k+1} = 0$$

Последният ненулев остатък r_k ще бъде търсеният *НОД*. Алгоритъмът има и съответен рекурсивен вариант, основаващ се на следното свойство на *НОД*:

$$НОД(a, b) = НОД\ (b, a\ \%\ b)$$

Дъното на рекурсията е при $b = 0$ (тъй като $НОД(a, 0) = a$) и ще бъде достигнато след краен брой стъпки, тъй като вторият аргумент на функцията намалява строго и монотонно, а естествените числа между b и 0 очевидно са краен брой.

Следват две реализации на Си (итеративна и рекурсивна) на току-що описания алгоритъм:

```
unsigned gcd(unsigned a, unsigned b)
{ unsigned swap;
  while (b > 0) {
    swap = b;
    b = a % b;
    a = swap;
  }
  return a;
}
```
🖫 gcditer.c

```
unsigned gcd(unsigned a, unsigned b)
{ return (0 == b) ? a : gcd(b, a % b);
}
```
🖫 gcdrec.c

Вижда се, че рекурсивната функция е много по-кратка и стегната. Изразходваната при нея памет обаче е повече: за всяко рекурсивно извикване в стека се заделя памет за формални параметри, както и за връщания от функцията резултат.

Разширен алгоритъм на Евклид

Понякога, например при модулна аритметика, алгоритъмът на Евклид се използва за намирането на два допълнителни цели множителя x и y ($x, y \in Z$), такива, че:

$$d = НОД(a,b) = ax + by$$

Задачи за упражнение:

1. Да се намери *НОД* на: (10,5); (5,10); (15,25); (25,15); (7,8,9); (3,6,9); (158,128,256); (64,28,72,18).

2. Да се напише програма за съкращаване на обикновени дроби. Така например при вход 10/15 да дава 2/3.

3. Да се докаже, че $(a_1, a_2, ..., a_n) = ((a_1, a_2, ..., a_{n-1}), a_n)$.

4. Да се реализира алгоритъмът на Евклид с изваждане.

5. Да се докаже, че $(a, b) = (b, a \% b)$.

6. Как трябва да се модифицират алгоритъмът и програмата, така че по дадени цели числа a и b, освен *НОД* да намира два цели множителя x и y ($x, y \in Z$), за които $(a,b) = ax + by$?

7. Да се предложи и реализира друг алгоритъм за намиране на *НОД*, като се изхожда от основната теорема на аритметиката. Да се сравни ефективността му с тази на предложената тук реализация при 2; 5; 100; 1000 числа.

8. (*Задача на Поасон за трите съда*) Дадени са три съда с вместимост a, b и c литра, a, b и c са естествени числа, $c > a > b$, $c \geq a + b - 1$, най-големият от които е пълен, а другите два са празни. Да се отмерят d литра, където d е естествено число и $0 < d \leq a$. Допустими са само такива преливания от един съд в друг, при които става максимално напълване и/или максимално изпразване на някой от съдовете.

1.2.4. Най-малко общо кратно

Дефиниция 1.21. Дадени са две цели числа a и b. Минималното естествено число d ($d > 0$) такова, че $a|d$ и $b|d$ се нарича *най-малко общо кратно* (*НОК*) на a и b.

Ще бележим най-малкото общо кратно на a и b с $НОК(a,b)$ или просто $[a,b]$.

Примери: [6, 15] = [15, 6] = 30, [1, 10] = 10, [5,10] = 10, [5, 12] = 60.

Когато търсим *НОК* на повече от две числа е налице аналогична зависимост, както и при *НОД*:

$$НОК(a_1, a_2, ..., a_n) = НОК(НОК(a_1, a_2, ..., a_{n-1}), a_n)$$

Най-малкото общо кратно може да се намери, като се използва съществуваща зависимост между него и *НОД*, а именно:

$$НОК(a,b) = \frac{a.b}{НОД(a,b)}$$

На базата на приведените по-горе свойства ще реализираме *НОК* на n числа. Числата са зададени в масив `A[]` и се подават, заедно с техния брой, като параметър на рекурсивната функция `lcm()`:

```
#include <stdio.h>

const unsigned n = 4;
const unsigned A[] = { 10, 8, 5, 9 };

unsigned gcd(unsigned a, unsigned b)
{ return (0 == b) ? a : gcd(b, a % b);
}

unsigned lcm(unsigned a[], unsigned n)
{ unsigned b;
  if (2 == n)
    return(a[0] * a[1]) / (gcd(a[0], a[1]));
  else {
    b = lcm(a, n - 1);
    return(a[n - 1] * b) / (gcd(a[n - 1], b));
  }
}
```

```
int main(void)
{ printf("%u\n", lcm(A, n));
  return 0;
}
```
🖫 lcm.c

Резултат от изпълнението на програмата:
```
360
```

Задачи за упражнение:

1. Да се намери: [10,15]; [15,10]; [7,8,9]; [3,6,9]; [158,128,256]; [64,28,72,18].
2. Да се докаже, че $[a_1, a_2, ..., a_n] = [[a_1, a_2, ..., a_{n-1}], a_n]$.
3. Да се докаже, че $[a,b] = ab / (a,b)$.
4. Да се предложи и реализира друг алгоритъм за намиране на *НОК*, като се изхожда от основната теорема на аритметиката. Да се сравни ефективността му с тази на предложената тук реализация при 2; 5; 100; 1000 числа.

1.2.5. Връщане от рекурсията и използване на променливите

Във всички рекурсивни функции на Си, показани досега, изпълнявахме операции само *преди* рекурсивното извикване. В практиката обаче често се налага да се изпълняват операции *след* връщане от рекурсията.

Задача: Да се отпечатат цифрите на дадено естествено число n, зададено в десетична бройна система, последователно от първата до последната.

Например, за $n = 7892$ ще трябва да се отпечата: 7, 8, 9, 2.

Проблемът в тази задача се състои в определянето на първата цифра на числото. От друга страна, намирането на последната цифра и елиминирането ѝ е възможно само с две операции: $n\%10$ и $n/10$. Предлагаме следния алгоритъм: на всяка стъпка намираме последната цифра и я записваме в целочислен масив. След записването на всички цифри в масива, го извеждаме в обратен ред:

```
#include <stdio.h>
unsigned n = 7892;
int main(void)
{ unsigned dig[20], i, k = 0;
  while (n > 0) {
    dig[k] = n % 10;
    n = n / 10;
    k++;
  }
  for (i = k; i > 0; i--) printf("%u ", dig[i-1]);
  printf("\n");
  return 0;
}
```
🖫 print.c

Използването на рекурсия при тази задача отново ще доведе до доста по-елегантно решение. Ще използваме следното свойство: за да отпечатаме 7892, трябва да отпечатаме цифрите на 789 и след него цифрата 2. Изобщо, за да отпечатаме числото n, трябва да отпечатаме (рекурсия!) числото $n/10$ и след него последната му цифра $n\%10$. Така, използвайки стека на рекурсията, се

освобождаваме от нуждата от явно въвеждане на масив. На базата на последното заключение ще съставим и съответната функция на Си:

```
void printN(unsigned n)
{ if (n >= 10) printN(n / 10);
  printf("%u ", n % 10);
}
```
🖫 printrec.c

Дъното на рекурсията, когато спираме да правим рекурсивни извиквания, е $n \le 9$, т.е. когато n стане едноцифрено число.

Важни при рекурсията са операциите с променливите. Досега в тялото на една рекурсивна функция се променяха само променливите-параметри и евентуално временни локални променливи. Операциите с глобални променливи, преди и след връщането от рекурсията, са важни при проектирането на рекурсивни алгоритми. Заместването на променливи-аргументи с глобални променливи ще илюстрираме с няколко примера: Намирането на $n!$ може да се извърши по следния начин, като се замени аргументът на функцията с глобална променлива i (Предимството на тази рекурсивна реализация пред разгледаните досега е, че не се използва никаква памет от стека):

```
#include <stdio.h>
const unsigned n = 6;
unsigned i;
unsigned long fact(void)
{ if (1 == i) return 1;
  return --i * fact();
}

int main(void) {
  i = n + 1;
  printf("%u! = %lu \n", n, fact());
  return 0;
}
```
🖫 factrec.c

Забележка: Понякога "трикове" като `return --i * fact();` са опасни: Програмата 🖫factrec.c работи правилно в средата на *Borland C* за *DOS*, но не и в средата на *Microsoft Visual C++* за *Windows*.

Ще разгледаме друг прост пример: За дадено естествено число n да се отпечатат в нарастващ и намаляващ ред числата 10^k ($1 \le k \le n$). Например, за $n = 5$ трябва се отпечата редицата:

10, 100, 1000, 10000, 100000, 100000, 10000, 1000, 100, 10

Ще решим задачата рекурсивно по три различни начина. С тях ще илюстрираме взаимната заменяемост на глобалните променливи и променливите параметри:

Вариант 1. Всички променливи (освен входът n) са параметри на рекурсивната функция:

```
#include <stdio.h>
const unsigned n = 5;

void printRed(unsigned k, unsigned long result)
{ printf("%lu ", result);
  if (k < n) printRed(k + 1, result * 10);
  printf("%lu ", result);
}
```

```
int main(void) {
  printRed(1, 10);
  printf("\n");
  return 0;
}
```
🖫 print1.c

Вариант 2. Параметърът-брояч *k* може да бъде изнесен като глобална променлива:

```
#include <stdio.h>
const unsigned n = 5;
unsigned k = 0;

void printRed(unsigned long result)
{ k++;
  printf("%lu ", result);
  if (k < n) printRed(result * 10);
  printf("%lu ", result);
}

int main(void) {
  printRed(10);
  printf("\n");
  return 0;
}
```
🖫 print2.c

Вариант 3. Ако модифицираме подходящо резултата `result` преди и след рекурсивното извикване, той също може да се изнесе като глобална променлива:

```
#include <stdio.h>
const unsigned n = 5;
unsigned long result = 1;
unsigned k = 0;

void printRed(void)
{ k++;
  result *= 10;
  printf("%lu ", result);
  if (k < n) printRed();
  printf("%lu ", result);
  result /= 10;
}

int main(void) {
  printRed();
  printf("\n");
  return 0;
}
```
🖫 print3.c

В следващия параграф, при разглеждането на основните комбинаторни алгоритми (както и изобщо в материала по-нататък), ще продължим да използваме рекурсията във всичките ѝ "екзотични" разновидности.

Задачи за упражнение:

1. Да се направи предположение за възможна причина за различното изпълнение на 🖫factrec.c под *Borland C* за *DOS* и *Microsoft Visual C++* за *Windows*. Можете ли да предложите "безопасен" вариант?

2. Да се направи предположение за резултата от изпълнението на фрагмента по-долу. Очаквате ли разлики под *Borland C* за *DOS* и *Microsoft Visual C++* за *Windows*? Отговаря ли предположението Ви на действителния резултат?

```
unsigned i = 1;
printf("%u %u", ++i, i);
```

А какво мислите за фрагмента:

```
unsigned i = 1;
printf("%u %u", i, ++i);
```

3. Да се направи предположение за стойността на променливата `x` след изпълнение на програмния фрагмент по-долу. Очаквате ли разлики под *Borland C* за *DOS* и *Microsoft Visual C++* за *Windows*? Отговаря ли предположението Ви на действителния резултат?

```
unsigned x, a = 3, b = 5;
x = a+++b;
```

4. Въз основа на резултатите от предходните задачи да се направят препоръки за писане на максимално преносим и недвусмислен код.

1.3. Основни комбинаторни алгоритми

Голяма част от задачите, които ще разгледаме в тази книга, предполагат намиране на *оптимално* (*екстремално*) решение. В подобни ситуации често прилаган подход е да се изследват *всички* възможни допустими решения на задачата и от тях да се избере оптималното — едно или няколко. Генерирането на комбинаторни обекти (с помощта на комбинаторни алгоритми) отговаря точно на тази схема.

В задачите от настоящия параграф обаче няма да се интересуваме от екстремалното решение, а единствено от *генерирането* на всички решения: всички възможни начини за нареждане на обекти, всички възможни начини за избор от обекти и т.н. Така например, ако е дадена колода с 52 карти, може да се интересуваме от всички възможни начини за избор на 5 карти така, че да бъдат поредни и от една и съща боя (на езика на покера: *чиста кента*). Друг пример е да пресметнем броя или да генерираме всички възможни шест-цифрени телефонни номера, с допълнителното условие в тях да не участва цифрата 9.

Ще разгледаме най-известните комбинаторни конфигурации: както алгоритми за генерирането им, така и формули за намиране на броя им. Въпреки че на пръв поглед разглежданите алгоритми не решават директно някаква практическа задача освен чисто комбинаторни проблеми, те (включително и в комбинация с други алгоритмични техники) се оказват изключително полезни при решаване на редица оптимизационни задачи.

1.3.1. Пермутации

Дефиниция 1.22. Разглеждаме n-елементно множество $A = \{a_1, a_2, ..., a_n\}$. Всяка наредена n-торка, получена с елементите от A, като всеки елемент от A участва *точно веднъж*, се нарича *пермутация*.

Множеството от всички възможни пермутации се бележи с P_n, а броят им — с $|P_n|$. Не е трудно да покаже, че $|P_n| = n!$

Например, ако е дадено множеството $\{a, b, c\}$ с 3 елемента, то всички възможни пермутации (т.е. всички възможни наредени n-торки) са:

(a, b, c)
(a, c, b)
(b, a, c)
(b, c, a)
(c, a, b)
(c, b, a)

До края на тази част елементите на множествата, с които работим, ще бъдат целите числа от 1 до n. Това не ограничава общността на разглежданите алгоритми и те ще бъдат валидни за произволни множества (произволно n-елементно множество е изоморфно на множеството от първите n естествени числа).

- Генериране

За компютърното генериране на пермутации ще използваме рекурсивен алгоритъм:
Алгоритъм 1:

Всеки от елементите поставяме на първа позиция в линейното нареждане, след което на останалите $n–1$ позиции разполагаме последователно всички възможни пермутации на останалите $n–1$ елемента. Така получаваме следната рекурсивна схема за генериране на пермутации:

```
/* Поставяне на елемент на позиция i */
void permute(i) {
  if (i >= n) { /* Намерили сме едно линейно нареждане
                   на елементите и го печатаме */
    printPerm();
  } else
  for (k = 0; k < n; k++)
   if (!used[k]) {     /* ако елементът k не е използван до момента */
     used[k] = 1;      /* маркираме елемента k като използван */
     position[i] = k;  /* елементът на позиция i е k */
     permute(i+1);
     used[k] = 0;      /* размаркираме елемента k като използван */
   }
}
```

Функцията `permute()` има единствен параметър: позицията, на която ще "поставяме" елемент. С вътрешния цикъл последователно на i-та позиция поставяме всички елементи, неучастващи до момента в нареждането, и продължаваме рекурсивно за $(i+1)$-вата позиция. Ще въведем масив `used[]`, за да можем да определяме дали даден елемент е използван: ако елементът вече участва в пермутацията, то `used[k] == 1`, и `used[k] == 0`, в противен случай (т.е. предстои му "поставяне"). Дъното на рекурсията е при `i == n` — тогава всички елементи са "поставени" и разполагаме с една пермутация, която отпечатваме.

Ето и пълната реализация, отпечатваща всичките $n!$ пермутации на множество с n елемента (n се задава в началото като константа):

```
#include <stdio.h>
#define MAXN 100
const unsigned n = 3;
char used[MAXN];
unsigned mp[MAXN];

void print(void)
{ unsigned i;
  for (i = 0; i < n; i++) printf("%u ", mp[i] + 1);
  printf("\n");
}
```

```
void permute(unsigned i)
{ unsigned k;
  if (i >= n) { print(); return; }
  for (k = 0; k < n; k++) {
    if (!used[k]) {
      used[k] = 1; mp[i] = k;
      permute(i+1); /* if (ако има смисъл да продължава генерирането)
                          { permute(i+1); } */
      used[k] = 0;
    }
  }
}

int main(void) {
  unsigned i;
  for (i = 0; i < n; i++) used[i] = 0;
  permute(0);
  return 0;
}
```

🖫 permute.c

Резултат от изпълнението на програмата:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Обърнете внимание на коментара по-горе, съдържащ `if`. Ако е дадена оптимизационна задача, чието решаване се свежда до генериране на пермутации, то е възможно да прекъсваме текущото генериране, ако е сигурно, че то няма да доведе до оптимално решение. В материала понататък (*виж глава 6*) ще разгледаме обстойно този въпрос, като засега ще се ограничим до един конкретен пример за такова прекъсване.

Задача: Да се намерят всички десетцифрени телефонни номера, в които всяка цифра участва точно по веднъж и такива, че: ако k_i е i-тата цифра на телефония номер, то $\sum_{\substack{1 \le i \le 10 \\ i \ne k_i}} k_i = 20$.

Очевидно, всички телефонни номера могат да се генерират чрез пермутации. Така, ако при поставянето на първите няколко цифри в линейното нареждане забележим, че сумата надвишава 20, то няма смисъл да продължаваме с по-нататъшното генериране, тъй като със сигурност няма да получим желаната сума.

Програмата по-горе генерира пермутациите в *лексикографски ред*. Това означава, че за всеки две последователни пермутации (i_1, i_2, …, i_n) и (j_1, j_2, …,j_n) съществува номер k ($1 \le k < n$) такъв, че $i_p = j_p$, $p = 1, 2, \ldots, k-1$ и $i_k < j_k$. За да стане по-ясно, можем да разглеждаме пермутациите като числа в съответна бройна система: тогава една пермутация е лексикографски по-голяма от друга, ако съответното число е по-голямо.

Възможно е да извършваме генерирането и по друг начин: пермутациите с размер k+1 да получаваме от пермутациите с размер k, като се използваме механизма на рекурсията. По този начин не е необходима и допълнителната памет за масива `used[]`. Това може да се осъществи, например със следния рекурсивен алгоритъм [*Наков-1998*]:

Алгоритъм 2:

1) При $n = 1$ генерираме единствена пермутация: (1).

2) Нека разглеждаме пермутация (p_1, p_2,…, p_k), състояща се от k ($1 \le k < n$) елемента. Поставяме (k+1)–вия елемент съответно на 1,2, … , (k+1)-ва позиция:

$$(p_{k+1}, p_1, p_2, \ldots, p_k)$$
$$(p_1, p_{k+1}, p_2, \ldots, p_k)$$
$$\ldots$$
$$(p_1, p_2, \ldots, p_k, p_{k+1})$$

Така, повтаряйки стъпка 2) за всички пермутации на k елемента, ще получим всички пермутации на k+1 елемента.

Предложената по-долу реализация е модификация на *алгоритъм 2*. Оставяме като упражнение на читателя да съобрази връзката между тях, която на пръв поглед не изглежда толкова директна.

```
#include <stdio.h>
#define MAXN 100
const unsigned n = 3;
unsigned a[MAXN];
void print(void)
{ unsigned i;
  for (i = 0; i < n; i++) printf("%u ", a[i] + 1);
  printf("\n");
}

void permut(unsigned k)
{ unsigned i, swap;
  if (k == 0) print();
  else {
    permut(k - 1);
    for (i = 0; i < k - 1; i++) {
      swap = a[i]; a[i] = a[k-1]; a[k-1] = swap;
      permut(k - 1);
      swap = a[i]; a[i] = a[k-1]; a[k-1] = swap;
    }
  }
}

int main(void) {
  unsigned i;
  for (i = 0; i < n; i++) a[i] = i;
  permut(n);
  return 0;
}
```
🖫 permswap.c

Резултат от изпълнението на програмата:

```
1 2 3
2 1 3
3 2 1
2 3 1
1 3 2
3 1 2
```

Задачи за упражнение:

1. Да се докаже, че броят на пермутациите на n-елементно множество е $n!$
2. Да се намерят ръчно пермутациите на елементите на $\{a,b,c,d\}$, като се използва:
   а) алгоритъм 1

б) алгоритъм 2

Да се сравнят резултатите.

3. Да се обосноват алгоритъм 1 и алгоритъм 2.

4. Да се покаже, че 🖫 permswap.c действително е реализация на *алгоритъм 2*.

5. Да се реализира алгоритъм за итеративно генериране на пермутации.

- Кодиране и декодиране

Понякога, например при генерирането на произволни пермутации с *евристични алгоритми* (*глава 9*) или при *memoization* (*глава 8*) се налага еднозначното кодиране и декодиране на пермутация. Това означава на всяка различна пермутация да се съпостави уникално естествено число, така че по-късно да може да бъде възстановена еднозначно от него. Например, нека разглеждаме лексикографски наредените пермутации на три елемента:

(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1).

На всяка от тях можем да съпоставим уникално естествено число между 0 и 5. Изобщо, на всяка пермутация на дадено n-елементно множество съответства биективно число между 0 и $n!-1$. Алгоритъм за определянето му по дадена пермутация (*кодиране на пермутация*) и обратната операция (*декодиране*) могат да се съставят по следните схеми [*Шишков-1995*]:

Кодиране

1) Нека `perm[]` е масив с дължина `n`, съдържащ елементите на пермутацията, която трябва да кодираме.
2) Нека `pos = result = 0`, `p[]` е масив с `n` елемента, `p[i] = i+1`, $i = 0,1, ..., n-1$.
3) Ако `pos < n`, то изключваме елемента, заемащ `r`-та позиция в `p`, а на `r` присвояваме `i`, където `i` е индекс, за който `perm[pos] == p[i]`. В противен случай алгоритъмът приключва и `result` е търсеното.
4) Присвояваме `result = result*(n - pos) + r`.
5) `pos++` и преход към 3).

Декодиране

1) Дадено е число `num` и от него трябва да възстановим пермутацията.
   Нека `k = n-1`, `p[]` е `n`-елементен масив, такъв че `p[i-1] = i`, за $i = 1, 2, ..., n$.
2) Докато `k ≥ 0` повтаряме:
```
m = n - k;
perm[k] = num % m;
if (k > 0) num /= m;
k--;
```
3) Присвояваме `k = 0`. Докато `k < n` повтаряме:
```
m = perm[k];
perm[k] = p[m];
if (k < n)
  for (i = m+1; i < n; i++) p[i-1] = p[i];
k++;
```

Следва пълният изходен код на програмата, реализираща горните операции:

```
#include <stdio.h>

#define MAXN 100

const unsigned n = 6;
const unsigned perm[MAXN] = { 5, 3, 6, 4, 2, 1 };
const unsigned long code = 551;
```

```
unsigned long codePerm(unsigned n, const unsigned perm[])
{ unsigned p[MAXN], i, pos;
  unsigned long r, result;
  result = 0;
  for (i = 0; i < n; i++) p[i] = i + 1;
  for (pos = 0; pos < n; pos++) {
    r = 0;
    while (perm[pos] != p[r]) r++;
    result = result * (n - pos) + r;
    for (i = r + 1; i < n; i++) p[i - 1] = p[i];
  }
  return result;
}

void decodePerm(unsigned long num, unsigned n, unsigned perm[])
{ unsigned long m, k;
  unsigned i, p[MAXN];
  for (i = 0; i < n; i++) p[i] = i + 1;
  k = n;
  do {
    m = n - k + 1;
    perm[k - 1] = num % m;
    if (k > 1) num /= m;
  } while (--k > 0);
  k = 0;
  do {
    m = perm[k]; perm[k] = p[m];
    if (k < n)
      for (i = m + 1; i < n; i++) p[i - 1] = p[i];
  } while (++k < n);
}
int main(void) {
  unsigned i, dperm[MAXN];
  printf("Дадената пермутация се кодира като %lu \n", codePerm(n, perm));
  printf("Декодираме пермутацията отговаряща на числото %lu: ", code);
  decodePerm(code, n, dperm);
  for (i = 0; i < n; i++) printf("%u ", dperm[i]);
  printf("\n");
  return 0;
}
```

🖫 codeperm.c

Резултат от изпълнението на програмата:

```
Дадената пермутация се кодира като 551
Декодираме пермутацията, отговаряща на числото 551: 5 3 6 4 2 1
```

Задачи за упражнение:

1. Като се използва алгоритъмът по-горе, да се кодират ръчно пермутациите: (2,3,1,4), (5,3,2,4,1), (3,6,4,1,5,2).

2. Като се използва алгоритъмът по-горе, да се намерят ръчно пермутациите на 5 елемента, съответстващи на код: 3, 13, 27, 87, 119.

- *Пермутации с повторение*

Нека сега вместо множество A разглеждаме мултимножество (т.е. в A е възможно да участват повтарящи се елементи). Да вземем например мултимножеството $A = \{1, 1, 2, 3\}$ от 4 елемента. Интересуваме се от генерирането на всички *пермутации с повторение*. Така отново търсим всички различни наредени n-торки, образувани от елементите на A. Обърнете внимание, че когато местата си сменят еднакви елементи — например първите два в пермутацията (1, 1, 2, 3), не се получава нова пермутация. Изобщо, две пермутации се считат за различни, когато съществува позиция, на която са поставени различни елементи.

Множеството от всички пермутации с повторение бележим с $\tilde{P}_n^{s_1,s_2,\ldots,s_k}$, където s_i е броят на елементите от вид i ($i = 1,2,\ldots,k$). Броят на всички пермутации с повторение може да се намери по формулата:

$$|\tilde{P}_n^{s_1,s_2,\ldots,s_n}| = \frac{n!}{s_1!s_2!\ldots s_k!}, \quad n = \sum_{i=1}^{k} s_i \quad (*)$$

За $s_1 = 2$, $s_2 = 1$, $s_3 = 1$ търсеният брой $|\tilde{P}_4^{2,1,1}|$ ще бъде равен на $\frac{4!}{2!.1!.1!} = 12$.

Задачи за упражнение:

1. Да се изпишат пермутациите на елементите на мултимножеството {1,1,2,3}.

2. Да се напише програма за намиране на всички пермутации с повторение. Може ли да се използват като основа реализацията на пермутации без повторение? Ако *да*, какви промени се налагат. Ако *не* — защо?

3. Да се напише програма за пресмятане на броя на пермутациите с повторение при зададени съответни параметри. Да се използва факторизация.

4. Да се реализират алгоритми за кодиране и декодиране на пермутации с повторение.

5. Да се докаже формулата (*).

1.3.2. Вариации

- *Видове, генериране*

Следващите алгоритми, които ще разгледаме, са за генериране на *вариации с и без повторение*. На всеки програмист се налага да пише вложени цикли — когато те са два, три или предварително известен брой, това става лесно. Когато обаче броят им не е известен предварително, например задачата, която решаваме, изисква n вложени цикъла, тогава стандартният подход е неприложим. Да разгледаме следния фрагмент:

пример 1.

```
for (a1 = 1; a1 <= k; a1++)
  for (a2 = 1; a2 <= k; a2++)
    for (a3 = 1; a3 <= k; a3++)
      ...
      for (an = 1; an <= k; an++)
        printf("  %u %u %u … %u;", a1, a2, a3, … , an);
```

Резултатът от изпълнението за $n = 2$ и $k = 3$, което е еквивалентно на 2 вложени цикъла за a_i от 1 до 3, ще бъде:

1 1; 1 2; 1 3; 2 1; 2 2; 2 3; 3 1; 3 2; 3 3;

пример 2.
```
for (a1 = 1; a1 <= k; a1++)
  for (a2 = 1; a2 <= k; a2++) if (a2 != a1)
    for (a3 = 1; a3 <= k; a3++) if ((a3 != a1) && (a3 != a2))
      …
      for (an = 1; an <= k; an++)
        if ((an!=a1)&&(an!=a2)&&(an!=a3)&&…&&(an!=an-1))
          printf("  %u %u %u … %u;", a1, a2, a3, … ,an);
```

Резултатът за $n = 2$ и $k = 3$ сега ще бъде:

1 2; 1 3; 2 1; 2 3; 3 1; 3 2;

Разликата в двата варианта на генериране е, че при втория няма *повтарящи се* числа.

Резултатът от първия фрагмент представлява генериране на всички *вариации с повторение* на n елемента от k-ти клас, а от втория фрагмент — *вариации без повторение* на n елемента от k-ти клас. Ще дефинираме понятията по-точно:

Нека е дадено множество A с n елемента.

Дефиниция 1.23. *Вариация с повторение на n елемента от k-ти клас* се нарича k-елементен мултисписък, образуван от елементи на A (не непременно различни).

Множеството от всички вариации с повторение се бележи с $\tilde{V}_n^k$ и броят на елементите му е n^k. Оставяме доказателството на последното на читателя като леко упражнение.

Дефиниция 1.24. *Вариация без повторение на n-елемента от k-ти клас* се нарича всеки нареден k-елементен списък, образуван от елементи на A.

Броят на различните вариации без повторение е $|V_n^k| = \frac{n!}{(n-k)!}$.

Непосредствено се вижда, че при $k = n$ множеството на вариациите без повторение съвпада с множеството на пермутациите без повторение.

Ще реализираме генериране на всички вариации с повторение по дадени n и k, за множеството A, съставено от естествените числа от 1 до n. Това, разбира се, не може да стане с вложени цикли, както в двата фрагмента по-горе. Ще се спрем на подхода, който използвахме при генерирането на пермутации. Разликата ще бъде, че на всяка позиция ще поставяме *всеки* от елементите, а не само неизползван до момента такъв (тъй като се допускат повторения). По този начин отпада нуждата от масив `used[]` за маркиране на използваните елементи и лесно достигаме до аналогична на 🖫 permute.c програма:

```
#include <stdio.h>
#define MAXN 100

const unsigned n = 4;
const unsigned k = 2;
int taken[MAXN];

void print(unsigned i)
{ unsigned l;
  printf("( ");
  for (l = 0; l <= i - 1; l++) printf("%u ", taken[l] + 1);
  printf(")\n");
}

void variate(unsigned i) /* рекурсия */
```

```
{ unsigned j;
  /* без if (i>=k) и return; тук (а само print(i); ако искаме всички
   * генерирания с дължина 1,2, …, k, а не само вариациите с дължина k */
  if (i >= k) { print(i); return; }
  for (j = 0; j < n; j++) {
    /* if (allowed(i)) { */
    taken[i] = j;
    variate(i + 1);
  }
}

int main(void) {
  variate(0);
  return 0;
}
```

🖫 variate.c

Резултат от изпълнението на програмата:

```
( 1 1 )
( 1 2 )
( 1 3 )
( 1 4 )
( 2 1 )
( 2 2 )
( 2 3 )
( 2 4 )
( 3 1 )
( 3 2 )
( 3 3 )
( 3 4 )
( 4 1 )
( 4 2 )
( 4 3 )
( 4 4 )
```

Задачи за упражнение:

1. Да се модифицира горната програма така, че да генерира вариации *без* повторение.
2. Да се изпишат 3-елементните вариации на елементите на множеството $\{a,b,c,d,e\}$:
   - с повторение
   - без повторение
3. Да се докаже, че броят на k-елементни вариации с повторение над n-елементно множество е n^k.
4. Да се докаже, че броят на k-елементни вариации без повторение над n-елементно множество е $n!/(n-k)!$
5. Да се реализира алгоритъм за итеративно генериране на вариации с/без повторение.
6. Да се реализират алгоритми за кодиране и декодиране на вариации с и без повторение.

- Сума нула

Нека са дадени числата a_1, a_2, ..., a_n. Да се поставят операции "+" и "–" между числата a_i и a_{i+1}, за $i = 1,2, \ldots, n-1$ така, че резултатът след пресмятане на получения израз да бъде равен на 0.

Например, ако са дадени естествените числа от 1 до 8, то няколко възможни решения на задачата са:

$$1 + 2 + 3 + 4 - 5 - 6 - 7 + 8 = 0$$
$$1 + 2 + 3 - 4 + 5 - 6 + 7 - 8 = 0$$
$$1 + 2 - 3 + 4 + 5 + 6 - 7 - 8 = 0$$

$$1 + 2 - 3 - 4 - 5 - 6 + 7 + 8 = 0$$

Решение: Ще генерираме всевъзможните вариации с повторение на n–1 елемента от втори клас, т.е. всевъзможните наредени (n–1)-орки, съставени от 0 и 1 (което отговаря на положителен и отрицателен знак пред съответното число). За всяка такава (n–1)-орка ще проверяваме дали е решение на задачата, като за целта ще пресмятаме стойността на съответния израз. Следва пълна реализация на това решение, като входните данни се задават в началото като константи — n е броят на числата, а масивът `a[]` съдържа самите числа:

```
#include <stdio.h>
#include <math.h>

/* Брой числа в редицата */
const unsigned n = 8;
/* Редица */
int a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
/* Търсена сума */
int sum = 0;

void checkSol(void)
{ unsigned i;
  int tempSum = 0;
  for (i = 0; i < n; i++) tempSum += a[i];
  if (tempSum == sum) { /* намерено е решение => отпечатваме го */
    for (i = 0; i < n; i++)
      if (a[i] > 0) printf("+%d ", a[i]);
      else printf("%d ", a[i]);
    printf(" = %d\n", tempSum);
  }
}

void variate(unsigned i)
{ if (i >= n) {
    checkSol();
    return;
  }
  a[i] = abs(a[i]);  variate(i + 1);
  a[i] = -abs(a[i]); variate(i + 1);
}

int main(void) {
  variate(0);
  return 0;
}
```
sumzero.c

Резултат от изпълнението на програмата:

```
+1 +2 +3 +4 -5 -6 -7 +8  = 0
+1 +2 +3 -4 +5 -6 +7 -8  = 0
+1 +2 -3 +4 +5 +6 -7 -8  = 0
+1 +2 -3 -4 -5 -6 +7 +8  = 0
+1 -2 +3 -4 -5 +6 -7 +8  = 0
+1 -2 -3 +4 +5 -6 -7 +8  = 0
+1 -2 -3 +4 -5 +6 +7 -8  = 0
-1 +2 +3 -4 +5 -6 -7 +8  = 0
-1 +2 +3 -4 -5 +6 +7 -8  = 0
-1 +2 -3 +4 +5 -6 +7 -8  = 0
```

```
-1 -2 +3 +4 +5 +6 -7 -8  = 0
-1 -2 +3 -4 -5 -6 +7 +8  = 0
-1 -2 -3 +4 -5 +6 -7 +8  = 0
-1 -2 -3 -4 +5 +6 +7 -8  = 0
```

Задачи за упражнение:

1. Да се реши задачата, ако освен събиране и изваждане, е възможно да се поставя и знак за умножение.
2. Да се предложи начин за спестяване на извършването на част от изчисленията. Например, кога генерирането може да бъде прекъсвано след k-та позиция, т.е. какъв критерий може да гарантира, че няма да се получи сума 0, независимо от това какви аритметични операции ще бъдат поставени на останалите n–k позиции?

1.3.3. Комбинации

Ако досега наредбата на елементите имаше значение, в този параграф тя вече няма да ни интересува. Нека е дадено множество A с n елемента.

Дефиниция 1.25. Комбинация без повторение на n елемента от k-ти клас се нарича k-елементно подмножество на A.

Дефиниция 1.26. *Комбинация с повторение на n елемента от k-ти клас* се нарича k-елементно мултимножество, съдържащо елементи от A.

Например за $n = 4$, $k = 2$ от четириелементното множество $\{a,b,c,d\}$ комбинациите без повторение от втори клас са $\{a,b\}$, $\{a,c\}$, $\{a,d\}$, $\{b,c\}$, $\{b,d\}$, $\{c,d\}$, а комбинациите с повторение са: $\{a,a\}$, $\{a,b\}$, $\{a,c\}$, $\{a,d\}$, $\{b,b\}$, $\{b,c\}$, $\{b,d\}$, $\{c,c\}$, $\{c,d\}$, $\{d,d\}$.

Комбинациите намират широко приложение в оптимизационните задачи. По-нататък многократно ще се сблъскваме с тях, затова алгоритмите за тяхното генериране са важни. Някои задачи, свързани с комбинации, са дадени в раздел *1.5*.

Броят на комбинациите без повторение, които вече видяхме как можем да пресметнем чрез факторизация в *1.1.5.*, е:

$$\binom{n}{k} = C_n^k = \frac{n.(n-1)...(n-k+1)}{k(k-1)...1} = \frac{n!}{k!(n-k)!},$$

а на комбинациите с повторение:

$$\tilde{C}_n^k = \frac{(n+k-1)!}{k!(n-1)!}$$

Генерирането на комбинации без повторение ще извършим по схемата на алгоритъма за генериране на пермутации. Придържаме се към нея, тъй като при генериране в лексикографски ред тя е изключително удобна, ако се налага “отрязване” на част от генерираните решения. Както вече стана дума, такова отрязване възниква при решаването на реални задачи, при които може да се окаже безсмислено изследването на голям клон от възможните комбинаторни конфигурации.

```
#include <stdio.h>
#define MAXN 20

/* Намира всички комбинации на n елемента от k-ти клас */
const unsigned n = 5;
const unsigned k = 3;

unsigned mp[MAXN];
```

```
void print(unsigned length)
{ unsigned i;
  for (i = 0; i < length; i++) printf("%u ", mp[i]);
  printf("\n");
}

void comb(unsigned i, unsigned after)
{ unsigned j;
  if (i > k) return;
  for (j = after + 1; j <= n; j++) {
    mp[i - 1] = j;
    if (i == k) print(i);
    comb(i + 1, j);
  }
}
int main(void) {
  printf("C(%u,%u): \n", n, k);
  comb(1, 0);
  return 0;
}
```

🖫 comb.c

Резултат от изпълнението на програмата:

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

Задачи за упражнение:

1. Да се модифицира горната програма така, че да генерира комбинации с повторение.
2. Да се изпишат 3-елементните комбинации на елементите на множеството $\{a,b,c,d,e\}$:
   - с повторение
   - без повторение
3. Да се обясни връзката между броя на комбинациите без повторение на n елемента от клас k и биномните коефициенти.
4. Да се докаже, че броят на k-елементни комбинации без повторение над n-елементно множество е $n! / (k!(n-k)!)$.
5. Да се докаже, че броят на k-елементни комбинации с повторение над n-елементно множество е $(n+k-1)! / (k!(n-1)!)$. Да се приведат две доказателства: едно алгебрично и едно комбинаторно.
6. Да се реализира алгоритъм за итеративно генериране на комбинации с/без повторение.
7. Да се реализират алгоритми за кодиране и декодиране на комбинации с/без повторение.

1.3.4. Разбиване на числа

- Генериране на всички разбивания на число като сума от дадени числа

Задача: По дадено естествено число n да се намерят всички възможни ненаредени представяния (*разбивания*) на n като сума от естествени числа (не непременно различни). Така например, числото 5 може да се разбие по следните 7 начина:

$5 = 5$
$5 = 4 + 1$
$5 = 3 + 2$
$5 = 3 + 1 + 1$
$5 = 2 + 2 + 1$
$5 = 2 + 1 + 1 + 1$
$5 = 1 + 1 + 1 + 1 + 1$

Алгоритъмът, по който ще реализираме разлагането, е рекурсивен:

разбиване$(0) = \{\}$
разбиване$(n) = \{k\}$ + *разбиване*$(n-k)$, $k = n, n-1,\ldots,1$.

Трябва да внимаваме и да избегнем генериране на повтарящи се разбивания, като:

$5 = 3 + 2$
$5 = 2 + 3$

Последното се осъществява лесно: ще поискаме всяко следващо събираемо да бъде по-малко или равно на предходното. Рекурсивната функция, извършваща разбиването, има два аргумента: `n` (число за разбиване) и `pos` — променлива, показваща колко пъти досега е било разбивано числото:

```
#include <stdio.h>

#define MAXN 100

const unsigned n = 7;
unsigned mp[MAXN + 1];

void print(unsigned length)
{ unsigned i;
  for (i = 1; i < length; i++)
    printf("%u+", mp[i]);
  printf("%u\n", mp[length]);
}

void devNum(unsigned n, unsigned pos)
{
  if (0 == n)
    print(pos-1);
  else {
    unsigned k;
    for (k = n; k >= 1; k--) {
      mp[pos] = k;
      if (mp[pos] <= mp[pos-1])
         devNum(n-k, pos+1);
    }
```

```
    }
}

int main(void) {
  mp[0] = n+1;
  devNum(n, 1);
  return 0;
}
```

🖫 devnum.c

Резултат от изпълнението на програмата:

```
7
6+1
5+2
5+1+1
4+3
4+2+1
4+1+1+1
3+3+1
3+2+2
3+2+1+1
3+1+1+1+1
2+2+2+1
2+2+1+1+1
2+1+1+1+1+1
1+1+1+1+1+1+1
```

Ако в задачата се търси само *броят* на различните разбивания, то (както и при другите комбинаторни конфигурации) не е нужно тяхното пълно генериране и преброяване. Подобен подход би бил крайно неефективен. Съществува рекурентна формула, позволяваща директното пресмятане на този брой. Всъщност, нещата с тази формула не са толкова прости и се налага използване на динамично оптимиране, поради което ще я разгледаме по-късно, в *8.3.6*.

Задачи за упражнение:

1. Да се реализира алгоритъм за генериране на разбиванията в лексикографски ред.
2. Да се реализира итеративна версия на алгоритъма за генериране на разбиванията.

- Генериране на всички разбивания на число като произведение от числа

Разликата с предходния параграф, както в алгоритъма, така и в реализацията, е минимална. Вместо `devNum(n-k,cnt+1)` ще извикваме рекурсивно `devNum(n/k,cnt+1)`, при това не за всяко k, а само за тези, за които `n % k == 0`. Условието за продължаване на разбиването (цикъла `for`) ще бъде $k > 1$, а не $k \geq 1$, т.е. дъното на рекурсията ще бъде k `==` 1, а не k `==` 0 (последното се обяснява лесно: 0 и 1 са именно идентитетите на операциите събиране и умножение). Следва модифицираната реализация:

```
#include <stdio.h>
#define MAXLN 20 /* Множители: най-много log2n (минималният е 2) */
const unsigned n = 50; /* Число, което ще разбиваме */
unsigned mp[MAXLN];

void print(unsigned length)
{ unsigned i;
  for (i = 1; i < length; i++) printf("%u * ", mp[i]);
  printf("%d\n", mp[length]);
```

```
}

void devNum(unsigned n, unsigned pos) {
  if (1 == n)
    print(pos-1);
  else {
    unsigned k;
    for (k = n; k > 1; k--) {
      mp[pos] = k;
      if (mp[pos] <= mp[pos-1] && n % k == 0)
        devNum(n / k, pos+1);
    }
  }
}

int main(void) {
  mp[0] = n + 1;
  devNum(n, 1);
  return 0;
}
```

🖫 devnum2.c

Резултат от изпълнението на програмата:

```
50
25 * 2
10 * 5
5 * 5 * 2
```

Задачи за упражнение:

1. Да се реализира алгоритъм за генериране на разбиванията в лексикографски ред.
2. Да се реализира итеративна версия на алгоритъма за генериране на разбиванията.

- Генериране на всички разбивания на число като сума от дадени числа

По интересен е случаят, когато трябва да разбием естествено число като сума от дадени естествени числа. Например, разполагаме с пощенски марки от 2, 5 и 10 лева и трябва да изпратим колет на стойност 20 лева. Всички възможности (общо 6 на брой) за образуване на тази сума са:

```
      20 = 10 + 10
20 = 10 + 5 + 5
20 = 10 + 2 + 2 + 2 + 2 + 2
20 =  5 + 5 + 5 + 5
20 =  5 + 5 + 2 + 2 + 2 + 2 + 2
      20 =  2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2
```

Алгоритъмът за решаването на тази задача е подобен на този за разбиване на число като сума от естествени числа. Нека числата, които можем да ползваме при разбиването, се намират в масива `given[gN]`. Ще изпълним цикъл за `p = 0,1,...,gN-1`, като при рекурсивното извикване вместо с `p`, ще намаляваме *n* със съответната стойност на `given[p]`:

```
#include <stdio.h>
#define MAX_ADDS 100

/* Сума, която ще разбиваме */
const unsigned n = 15;
/* Брой различни стойности на монетите */
```

```
const unsigned gN = 3;

/* Стойности на монетите */
const unsigned given[] = { 2, 3, 5 };

unsigned mp[MAX_ADDS];

void print(unsigned length)
{ unsigned i;
  for (i = 1; i < length; i++)
    printf("%u + ", mp[i]);
  printf("%d\n", mp[length]);
}
void devNum(unsigned n, unsigned pos)
{ unsigned k, p;
  for (p = gN; p > 0; p--) {
    k = given[p - 1];
    if (n > k) {
      mp[pos] = k;
      if (mp[pos] <= mp[pos - 1])
        devNum(n - k, pos + 1);
    }
    else if (n == k) {
      mp[pos] = k;
      if (mp[pos] <= mp[pos - 1])
        print(pos);
    }
  }
}

int main(void) {
  mp[0] = n + 1;
  devNum(n, 1);
  return 0;
}
```
🖫 devnum3.c

Резултат от изпълнението на програмата:

```
5 + 5 + 5
5 + 5 + 3 + 2
5 + 3 + 3 + 2 + 2
5 + 2 + 2 + 2 + 2 + 2
3 + 3 + 3 + 3 + 3
3 + 3 + 3 + 2 + 2 + 2
3 + 2 + 2 + 2 + 2 + 2 + 2
```

Задачи за упражнение:

1. Възможно ли е алгоритъмът да се обобщи и за реални числа? Ако *не*: защо? Ако *да*: какви промени се налагат?
2. Да се реализира алгоритъм за генериране на разбиванията в лексикографски ред.
3. Да се реализира итеративен вариант на алгоритъма за генериране на разбиванията.
4. Да се предложи и реализира алгоритъм за разбиване на число като *произведение* от дадени числа.

1.3.5. Разбиване на множества

Задачата, която ще разгледаме, е за намиране на броя на всевъзможните *разбивания на множество*, т.е. представяне на множеството като обединение от непресичащи се негови непразни подмножества. Така например, за множество $A = \{1, 2, 3\}$ разбиванията ще бъдат:

{1,2,3}
{1,2}, {3}
{1,3}, {2}
{1}, {2,3}
{1}, {2}, {3}

- Числа на Бел и Стирлинг

Докато при разгледаните по-горе комбинаторни конфигурации за броя съществуваше някаква директна формула, тук нещата са малко по-сложни.

Броят на разбиванията за множество с n елемента е равен на n-тото число на *Бел* $B(n)$. Числата на Бел се дефинират по следния начин:

$$B(n) = \sum_{k=0}^{n} St(n,k),$$

а $St(n,k)$ са числата на Стрилинг от втори род, които се дефинират рекурсивно така:

$$St(n,k) = \begin{cases} St(n-1,k-1) + k.St(n-1,k), \text{за } k = 2,3,\ldots,n \\ 1, \text{за } n > 0, k = 1 \\ 0, \text{за } k = 0 \\ 0, \text{за } n = 0 \end{cases}$$

Числата на Стирлинг представляват броя на представянията на n-елементно множество като обединение на *точно* k непресичащи се негови непразни подмножества. Тях можем да намерим по няколко начина. Единият е да използваме директно рекурентната дефиниция и да реализираме съответна рекурсивна функция. Така обаче някои стойности ще се изчисляват многократно, което ще доведе до неефективно решение. С подобен проблем вече се сблъскахме, когато пресмятахме редицата на Фибоначи. Числата на Стирлинг и Бел ще намерим итеративно по следния начин: Първо ще намерим и запазим числата на Стирлинг `M[i] = St(n,i)`, $i = 0,1,\ldots,n$. След това n-тото число на Бел ще намерим като сума на съответните числа на Стирлинг.

Числата на Стирлинг определят триъгълник, подобен на този на Паскал — триъгълник на Стирлинг. Това наблюдение директно ни навежда на мисълта за реализация на алгоритъм, аналогичен на този, който използвахме при биномните коефициенти.

```
                    1
                 1     1
              1     3     1
           1     7     6     1
        1    15    25    10     1
     1    31    90    65    15     1
  .........................................
```

Фигура 1.3.5. Триъгълник на Стирлинг.

```
#include <stdio.h>
#define MAXN 100
const unsigned long n = 10;

unsigned long M[MAXN+1];
void stirling(unsigned n)
```

```
{ unsigned i, j;
  if (0 == n) M[0] = 1;
    else M[0] = 0;
  for (i = 1; i <= n; i++) {
    M[i] = 1;
    for (j = i - 1; j >= 1; j--) M[j] = j * M[j] + M[j - 1];
  }
}

unsigned long bell(unsigned n)
{ unsigned i;
  unsigned long result = 0;
  for (i = 0; i <= n; i++) result += M[i];
  return result;
}

int main(void) {
  stirling(n);
  printf("bell(%lu) = %lu\n", n, bell(n));
  return 0;
}
```

🖫 bell.c

Резултат от изпълнението на програмата:

```
bell(10) = 115975
```

Задачи за упражнение:

1. Да се намери ръчно петото число на Стирлинг.
2. Да се реализира рекурсивен вариант на алгоритъма за намиране на числата на Стирлинг.
3. Да изведат формулите, отразяващи зависимостите между числата в триъгълника на Стирлинг. Да се сравнят с тези от триъгълника на Паскал от *1.1.5*.
4. Да се напише програма за намиране всички разбивания на дадено множество като обединение на непресичащи се непразни негови подмножества.

1.4. Оценка и сложност на алгоритми

Ще завършим настоящата уводна глава с кратко въведение в математическия апарат, необходим за пълноценното изследване на всеки компютърен алгоритъм. Темата за оценка и сложност на алгоритмите е важна и не бива да се подминава. Означенията и свойствата, които ще представим, ще се срещат почти непрекъснато в материала по-нататък.

Когато разглеждаме даден компютърен алгоритъм, се интересуваме най-общо от три негови свойства:

- простота (и елегантност)
- коректност
- бързодействие

Докато първото от тях всеки може да "премери" интуитивно (и донякъде субективно), то за последните две е необходим много по-задълбочен анализ. Уменията, които ще придобием в настоящия параграф, ще ни позволяват лесно и точно да определяме ефективността на даден алгоритъм, да сравняваме алгоритми, да повишаваме бързодействието им чрез премерени и точни промени.

Нека разгледаме следния програмен фрагмент:

```
1)    n = 100;
2)    sum = 0;
3)    for (i = 0; i < n; i++)
4)      for (j = 0; j < n; j++)
5)        sum++;
```

Интересува ни колко бързо ще работи горната програма. Това, което можем да направим експериментално, е да проверим за какво време ще завърши работата си. За да изследваме по-общо нейното поведение, можем да я изпълним за други стойности на *n*. *Таблица 1.4.* показва зависимостта между размера на входните данни и скорост на изпълнение.

Размер на входа	Време за изпълнение
10	0,000001 сек.
100	0,0001 сек.
1000	0,01 сек.
10000	1,071 сек.
100000	106,543 сек.
1000000	10663,6 сек.

Таблица 1.4. Зависимост между размер на входните данни и скорост на изпълнение.

От *таблица 1.4.* се вижда, че когато увеличаваме n десет пъти, времето за изпълнение на програмата се увеличава 100 пъти.

Нека изследваме по-задълбочено горния фрагмент. На редове 1) и 2) има статична инициализация, която отнема константно време. Да го означим с a. За операциите $i = 0$ и i++, както и за проверката $i < n$, отново е необходимо константно време (всяка от тях представлява константен брой инструкции на процесора), ще го означим съответно с b, c, d. На ред 4) времената, необходими за операциите $j = 0$, $j < n$ и j++, означаваме с e, f, g. Последно, операцията на ред 5) също изисква константно време: нека бъде h.

С така въведените означения не е трудно да се пресметне общото време на работа на програмата за произволна стойност на n:

$$a + b + n.c + n.d + n.(e + n.f + n.g + n.h) =$$
$$= a + b + n.c + n.d + n.e + n.n.f + n.n.g + n.n.h =$$
$$= n^2.(f + g + h) + n.(c + d + e) + a + b$$

Припомняме, че a, b, c, d, e, f, g, h са константи. Нека означим:

$$i = f + g + h$$
$$j = c + d + e$$
$$k = a + b$$

(тук i и j нямат общо с използваните във фрагмента по-горе променливи). Така алгоритъмът се изпълнява за време:

$$i.n^2 + j.n + k$$

Константите i, j и k са *важни* за бързодействието на алгоритъма, но не са *определящи*. На практика когато изследваме ефективността на даден алгоритъм, ние не се интересуваме от тях. Тези константи зависят предимно от машинното представяне на нашата програма, както и от бързината на машината, на която тя се изпълнява.

Нещо повече, когато изследваме поведението на нашия алгоритъм, можем да игнорираме дори и едночлените $j.n$ и k и да оставим единствено този, в който n участва в най-висока степен. Целта на това "опростяване" е да оставим само *най-значимата* характеристика за даден алгоритъм, т.е. функцията, от която в най-голяма степен зависи времето на изпълнение, т.е. която нараства най-бързо, когато се увеличава размерът на входните данни.

Нека разгледаме две функции, които показват времето за изпълнение на два дадени алгоритъма A_1 и A_2 в зависимост от размера n на входните данни: $f = 2.n^2$ и $g = 200.n$. Лесно се забе-

лязва, че въпреки че коефициентът на g е много по-голям от този на f, когато n премине някаква фиксирана стойност (в случая $n > 100$), алгоритъмът A_2 ще решава задачата по-бързо от A_1. Нещо повече, колкото повече се увеличава n, толкова повече отношението между времето за изпълнение на двата алгоритъма се увеличава в полза на A_2. Асимптотично алгоритъмът A_2 е по-бърз и неговата сложност е *линейна*, докато тази на A_1 е *квадратична*.

В изложението по-долу ще поставим върху по-здрави основи последните наблюдения.

Задачи за упражнение:

1. Дадени са три алгоритъма със сложности съответно $5n^2 - 7n + 13$, $3n^2 + 15n + 100$ и $1000n$. Кой от тях следва да се използва при входни данни с размер до: 100; 1000; 10000; 1000000?

2. Възможно ли е полиномът $5n^3 - 5n^2 + 5$ да задава сложност на алгоритъм? А полиномът $5n^3 - 5n^2 - 5$? Ами $-5n^3 - 5n^2 + 5$?

1.4.1. Размер на входните данни

Нека е дадена задача, в която размерът на входните данни е определен от дадено цяло число n. Почти всички задачи, които ще разглеждаме, притежават това свойство. Ще поясним последното, като разгледаме няколко примера:

Пример 1. Даден е масив с n елемента и искаме да го сортираме. Размерът на входните данни се определя от броя n на елементите на масива.

Пример 2. Дадени са две естествени числа a и b и търсим най-големия им общ делител.

Тук размерът на входните данни се определя от броя на двоичните цифри (битовете) на по-голямото от числата a и b: т.е. от $\lceil \log_2(\max(a,b)) \rceil$. Когато по-долу въведем апарата за изследване на ефективността на алгоритмите, ще видим защо избрахме именно *максималното* от двете числа.

*Пример 3.*Даден е граф и търсим негово покриващо дърво (*виж 5.1.*).

В този случай е удобно да характеризираме размера на входа с два параметъра: *брой на върховете* и *брой на ребрата* на графа.

1.4.2. Асимптотична нотация

Когато се интересуваме от сложността на алгоритъм, най-често се интересуваме как се държи при достатъчно голям размер n на входните данни. При формалното оценяване на сложността на алгоритмите ще се интересуваме от поведението им при n, клонящо към безкрайност. Сложността на даден алгоритъм ще описваме с функции от вида $f : N \to N$. (Ще припомним, че с N означаваме множеството на естествените числа: 0,1,2,...). Понякога ще работим с функции, дефинирани върху *подмножество* на N, например валидни само за четни n или пък от дадена стойност нататък. Ще използваме и реални функции, например $\log n$, като по принцип ще подразбираме тяхно ограничение върху N. Последните случаи не са особено "чисти" от теоретична гледна точка, но спестяват много трудности.

Дефиниция 1.27. $O(F(n)) = \{f(n) \mid \exists c\ (c > 0), \exists n_0(c): \forall n > n_0 : 0 \leq f(n) \leq c.F(n)\}$

Т.е. $O(F(n))$ е множеството от функции f, за които съществува константа c ($c > 0$) такава, че $f(n) \leq c.F(n)$, за всички *достатъчно големи* стойности на n, т.е. съществува константа n_0 (евентуално зависеща от c), за която горното неравенство е изпълнено за всяко $n > n_0$. Така $O(F)$ определя множеството на всички функции, които *нарастват* ***не*** *по-бързо* от F.

Когато се разглежда сложност на алгоритмите, се използват още две основни означения: $\Theta(F)$ и $\Omega(F)$.

Дефиниция 1.28. $\Omega(F(n)) = \{f(n) \mid \exists c\ (c > 0), \exists n_0\ (c): \forall n > n_0 : f(n) \geq c.F(n) \geq 0\}$

Т.е. $\Omega(F)$ е множеството от функции $f(n)$, за които $f(n) \geq c.F(n)$ за всяко $n > n_0$. Така $\Omega(F)$ включва всички функции, които *нарастват* ***не*** *по-бавно* от F.

Дефиниция 1.29.

$\Theta(F(n)) = \{f(n) \mid \exists c_1 (c_1 > 0), \exists c_2(c_2 > 0), \exists n_0(c_1,c_2): \forall n \geq n_0 : 0 \leq c_1.F(n) \leq f(n) \leq c_2.F(n)\}$

Непосредствено от дефиницията следва, че $\Theta(F) = O(F) \cap \Omega(F)$. Т.е. $\Theta(F)$ съдържа всички функции, които нарастват *толкова бързо*, *колкото* и F (с точност до константен множител).

По-рядко се използват следните нотации (обърнете внимание на *строгите* неравенства в дефинициите.):

Дефиниция 1.30. $o(F(n)) = \{f(n) \mid \forall c\ (c > 0), \exists n_0(c): n > n_0 : 0 \leq f(n) < c.F(n)\ \}$

Непосредствено от дефиницията следва, че $o(F) = O(F) \setminus \Theta(F)$. Т.е. $o(F)$ съдържа всички функции, които нарастват *по-бавно* от F (с точност до константен множител).

Забележете, че докато $3n^2 \in O(n^2)$ и $3n^2 \in O(n^3)$, то $3n^2 \in o(n^3)$, но $3n^2 \notin o(n^2)$.

Дефиниция 1.31. $\omega(F(n)) = \{f(n) \mid \forall c\ (c > 0), \exists n_0\ (c): n > n_0 : 0 \leq c.F(n) < f(n)\ \}$

Непосредствено от дефиницията следва, че $\omega(F) = \Omega(F) \setminus \Theta(F)$. Т.е. $\omega(F)$ съдържа всички функции, които нарастват *по-бързо* от F (с точност до константен множител).

В следващите точки ще разгледаме полезни свойства и примери как на практика можем да оценяваме сложност на алгоритъм.

По-долу ще използваме стандартния символ за означаване на принадлежност към множество "$\in$", за да покажем, че $f \in \zeta(F)$, където ζ е някоя от горните асимптотични функции. Когато е удобно обаче, ще го заменяме с равенство. Макар на пръв поглед това да изглежда странно, то ни дава предимството да можем да пишем рекурентни зависимости за асимптотичната функция $T(n)$ от вида:

$$T(n) = T(n-1) + O(n)$$

Задачи за упражнение:

1. Да се докаже, че $\Theta(F) = O(F) \cap \Omega(F)$.
2. Да се докаже, че $o(F) = O(F) \setminus \Theta(F)$.
3. Да се докаже, че $\omega(F) = \Omega(F) \setminus \Theta(F)$.

1.4.3. O(F): Свойства и примери

Нотацията $O(F)$ доскоро беше най-често използваната при оценка на сложност на алгоритми и програми. Сравнително отскоро се предпочита по-точната оценка $\Theta(F)$, но това изисква допълнителни усилия при анализа на алгоритмите и все още не е широко възприет подход. В следващите глави най-често ще ползваме нотацията $\Theta(...)$ и само в специални случаи — други нотации.

Някои свойства на $O(F)$:

- *рефлексивност*: $f \in O(f)$
- *транзитивност*: $f \in O(g),\ g \in O(h) \Rightarrow f \in O(h)$
- *транспонирана симетрия:* $f \in \Omega(g) \Leftrightarrow g \in O(f)$
- Константите могат да бъдат игнорирани:
  За всяко $k > 0$, $k.F \in O(F)$.

- n, повдигнато в по-висока степен, нараства по-бързо:
  $n^r \in O(n^s)$, за $0 \le r \le s$.
- Нарастването на сума от функции се определя от асимптотично най-бързо нарастващата (това е означено с max):

$$f + g \in O(\max(f,g)),$$

  което може да се запише и така:

$$f \in O(g) \Rightarrow f+g \in O(g)$$

  или така:

$$O(c_1 f + c_2 g) \in O(\max(f,g)), \text{ за } c_1, c_2 > 0$$

- Ако $f(n)$ е полином от степен d или по-ниска, то $f \in O(n^d)$.
- Произведение на функции:

$$f \in O(F) \text{ и } g \in O(G) \Rightarrow f.g \in O(F.G)$$

Проверка (съгласно дефиницията):

За да твърдим, че функцията f принадлежи на $O(g)$, е необходимо и достатъчно да намерим естествено число n_0 и положителна константа c такива, че за всяко естествено n ($n > n_0$) да бъде в сила неравенството $f(n) \le c.g(n)$.

Асимптотични свойства (предполагаме, че границите съществуват)*:*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow f(n) \in O(g(n)), g(n) \in O(f(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), g(n) \notin O(f(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \notin O(g(n)), g(n) \in O(f(n))$$

Тези свойства могат да бъдат използвани като алтернативен начин за проверка на принадлежността на една функция към клас функции от тип $O(\ldots)$.

Пример: Да разгледаме функциите $\log n$ и $\sqrt{n}$. Като използваме правилото на Лопитал, получаваме:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{1/n}{1/\left(2\sqrt{n}\right)} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$$

От последното равенство непосредствено следва, че $\log n \in O(\sqrt{n})$, но $\sqrt{n} \notin O(\log n)$.

Положителни примери

$10n \in O(n)$,
$10n \in O(n^2)$,
$10n \in O(n^4)$
$10n \in O(3n^4 - 10n^2 + 7)$
$10n+3 \in O(n)$
$4n^2 - 5n + 2 \in O(n^2)$
$4n^3 + 5n^2 + 5 \in O(n^3)$
$\sqrt{n} \in O(n)$
$\log n \in O(\sqrt{n})$

Отрицателни примери

$2n \notin O(1)$
$4n^2 - 5n + 2 \notin O(n)$
$5n + 1 \notin O(\sqrt{n})$
$\sqrt{n^3} \notin O(n)$

Задачи за упражнение:

1. Да се провери съгласно дефиницията, че $5n^3 - 5n^2 + 5 \in O(n^3)$.
2. Да се докаже, че $\sqrt{n} \in O(n)$.
3. Да се докажат приведените по-горе общи свойства на $O(F)$.
4. Да се докажат приведените по-горе асимптотични свойства на $O(F)$.
5. Да се даде пример за функции $f(n)$ и $g(n)$, за които границата $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ не съществува.

1.4.4. Ω(F): Свойства и примери

Това е рядко използвана нотация, тъй като долната граница на сложността на даден алгоритъм рядко представлява интерес, освен в някои специални случаи, главно като инструмент за доказателство за принадлежност към $\Theta(n)$. Припомняме, че $\Theta(F) = O(F) \cap \Omega(F)$.

Някои свойства на $\Omega(F)$:

- *рефлексивност*: $f \in \Omega(f)$
- *транзитивност*: $f \in \Omega(g)$, $g \in \Omega(h) \Rightarrow f \in \Omega(h)$
- *транспонирана симетрия*: $f \in \Omega(g) \Leftrightarrow g \in O(f)$
- Константите могат да бъдат игнорирани:
  За всяко $k > 0$, $k.F \in \Omega(F)$.
- Ако $f(n)$ е полином от степен d или по-висока, то $f \in \Omega(n^d)$.

Проверка (съгласно дефиницията):

За да твърдим, че функцията f принадлежи на $\Omega(g)$, е необходимо и достатъчно да намерим естествено число n_0 и положителна константа c такива, че за всяко естествено n ($n > n_0$) да бъде в сила неравенството $f(n) \geq c.g(n)$.

Асимптотични свойства (предполагаме, че границите съществуват):

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow f(n) \in \Omega(g(n)), g(n) \in \Omega(f(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \notin \Omega(g(n)), g(n) \in \Omega(f(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \Omega(g(n)), g(n) \notin \Omega(f(n))$$

Тези свойства могат да бъдат използвани като алтернативен начин за проверка на принадлежността на дадена функция към клас функции от тип $\Omega(\ldots)$.

Положителни примери:

$10n \in \Omega(n)$
$10n+3 \in \Omega(1)$
$4n^2 - 5n + 2 \in \Omega(n^2)$
$4n^2 - 5n + 2 \in \Omega(n)$
$4n^3 + 5n^2 + 5 \in \Omega(n^3)$
$4n^3 + 5n^2 + 5 \in \Omega(n^2)$
$4n^3 + 5n^2 + 5 \in \Omega(7n - 10)$
$n \in \Omega(\sqrt{n})$
$\sqrt{n} \in \Omega(\log n)$
Отрицателни примери:
$4n^2 - 5n + 2 \notin \Omega(n^3)$
$\sqrt{n} \notin \Omega(n)$
$\log n \notin \Omega(\sqrt{n})$

Задачи за упражнение:

1. Да се провери съгласно дефиницията, че $5n^3 - 5n^2 + 5 \in \Omega(n^3)$.
2. Да се докаже, че $\sqrt{n} \in \Omega(n^{0,48})$.
3. Да се докажат приведените по-горе общи свойства на $\Omega(F)$.
4. Да се докажат приведените по-горе асимптотични свойства на $\Omega(F)$.

1.4.5. Θ(F): Свойства и примери

Някои свойства на $\Theta(F)$*:*

- *рефлексивност*: $f \in \Theta(f)$
- *транзитивност*: $f \in \Theta(g),\ g \in \Theta(h) \Rightarrow f \in \Theta(h)$
- *симетричност:* $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$
- *Константите могат да бъдат игнорирани*:
  За всяко $k > 0$, $k.F \in \Theta(F)$.
- Нарастването на сума от функции се определя от най-бързо нарастващата от тях:
  $$f + g \in \max(\Theta(f(n)), \Theta(g(n))),$$
  което може да се запише и така:
  $$f \in \Theta(g) \Rightarrow f+g \in \Theta(g) \text{ и}$$
  или така:
  $$\Theta(c_1f(n) + c_2g(n)) \in \max(\Theta(f(n)), \Theta(g(n))), \text{ за } c_1,c_2 > 0$$
- Ако $f(n)$ е полином от степен *точно* d, то $f \in \Theta(n^d)$.

Дефиниция 1.32. Релация, притежаваща едновременно свойствата *рефлексивност*, *симетричност* и *транзитивност*, е релация на еквивалентност.

Да дефинираме релацията R = "принадлежи към Θ(...)", задавана като $(f,g) \in R$ тогава и само тогава, когато $f \in \Theta(g)$. От свойствата на Θ(...) и от *дефиниция 1.3.2.* следва, че R е релация на еквивалентност.

Проверка (съгласно дефиницията):

За да твърдим, че функцията f принадлежи на $\Theta(g)$, е необходимо и достатъчно да намерим естествено число n_0 и две положителни константи c_1 и c_2 такива, че за всяко естествено n ($n > n_0$) да бъде в сила неравенството $c_1.g(n) \leq f(n) \leq c_2.g(n)$.

Пример 1:

Да вземем например функцията $n^2/2 - 3n$. Ще покажем, че тя принадлежи на множеството от функции $\Theta(n^2)$. Търсим такива константи c_1, c_2 и n_0, че $c_1.n^2 \leq n^2/2 - 3n \leq c_2.n^2$ ($n > n_0$). Разделяме на n^2 (при допълнителното ограничение $n > 0$) и получаваме: $c_1 \leq 1/2 - 3/n \leq c_2$. Един възможен избор е $c_1 = 0$, $c_2 = 1/2$ и $n_0 = 5$. Оставяме на читателя да извърши съответната проверка.

Ще отбележим, че тези стойности на c_1, c_2 и n_0 съвсем не са единствени. Така например, можехме да изберем $c_1 = 1/20$, $c_2 = 1$ и $n_0 = 100$. При това са силно зависими от коефициентите на разглеждания полином и за друг полином в общия случай ще се различават значително.

Пример 2:

Да разгледаме функцията $2n^3$. Ще покажем, че тя не принадлежи на множеството от функции $\Theta(n^2)$. Наистина, ако допуснем обратното, то би трябвало да съществува константа c_2 такава, че $6n^3 \leq c_2.n^2$. Но ако разделим на n^2 (при допълнителното ограничение $n > 0$) двете страни на неравенството, получаваме неравенството $6n \leq c_2$, а оттук: $n \leq c_2/6$. Така се оказва, че n е по-малко от някаква *константа*. Очевидно не може да се намери такова n_0, че за всяко n, $n > n_0$ да бъде изпълнено $n \leq c_2/6$: от лявата страна на неравенството имаме неограничено отгоре число, а отдясно — константа.

Асимптотични свойства:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} \in R^+ \Rightarrow f(n) \in \Theta(g(n)), g(n) \in \Theta(f(n))$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), f(n) \notin \Theta(g(n)), \text{т.е. } f(n) \in o(g(n))$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \Omega(g(n)), f(n) \notin \Theta(g(n)), \text{т.е. } f(n) \in \omega(g(n))$$

Тези свойства могат да бъдат използвани като алтернативен начин за проверка на принадлежността на една функция към определен клас функции.

Положителни примери:

$10n \in \Theta(n)$
$4n^2 - 5n + 2 \in \Theta(n^2)$
$4n^2 - 5n + 2 \in \Theta(6n^2 - n + 1)$
$\log_{10} n \in \Theta(\log_2 n)$

Отрицателни примери:

$10n \notin \Theta(1)$
$10n \notin \Theta(n^2)$
$4n^2 - 5n + 2 \notin \Theta(n^3)$
$4n^2 - 5n + 2 \notin \Theta(n + 1)$

Дефиниция 1.33. Сложност $\Theta(1)$ се нарича *константна*, $\Theta(\log n)$ — *логаритмична*, $\Theta(n)$ — *линейна*, $\Theta(n^2)$ — *квадратична*, $\Theta(c^n)$ — *експоненциална*. Когато функцията f е полином, сложността $\Theta(f)$ се нарича *полиномиална*.

Задачи за упражнение:

1. Да се провери съгласно дефиницията, че $5n^3 - 5n^2 + 5 \in \Theta(n^3)$.
2. Да се докаже, че $\sqrt{n} \notin \Theta(n^{0,48})$.
3. Да се докажат приведените по-горе общи свойства на $\Theta(F)$.
4. Да се докажат приведените по-горе асимптотични свойства на $\Theta(F)$.
5. Да се провери, че релацията R = "принадлежи към $\Theta(...)$" (задавана като $(f,g) \in R$ тогава и само тогава, когато $f \in \Theta(g)$) е релация на еквивалентност.
6. Да се посочат верните твърдения:
   а) $4n^3 + 5n^2 + 5 \in \Omega(n^4)$
   б) $log_2 n \notin \Omega(\sqrt{n})$
   в) $4n^3 + 5n^2 + 5 \in \Omega(7n^4 - 10)$
   г) $n \in \Omega(\sqrt{n})$
   д) $log_2 n \in O(\sqrt{n})$
   е) $10n+3 \in O(n)$
   ж) $10n \in O(3n^4 - 10n^2 + 7)$
   з) $5n + 1 \in O(\sqrt{n})$
7. Да се посочат верните твърдения:
   а) $\Theta(F) = O(F) \cap \Omega(F)$
   б) $\omega(F) = \Omega(F) \setminus \Theta(F)$
   в) $o(F) = O(F) \setminus \Theta(F)$
   г) $O(c_1 f + c_2 g) = O(\max(f,g))$
   д) $f \in \Omega(g) \Rightarrow g \in \Theta(f)$
   е) $f \in O(g) \Rightarrow g \in \Omega(f)$
   ж) $f \in \Omega(g) \Leftrightarrow g \notin O(f)$
8. Да се посочат верните твърдения:
   а) $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow f(n) \in O(g(n)), g(n) \in O(f(n))$
   б) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), g(n) \notin O(f(n))$
   в) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f(n) \notin O(g(n)), g(n) \in O(f(n))$
   г) $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow f(n) \in \Theta(g(n))$
   д) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in \Omega(g(n)), f(n) \notin \Theta(g(n))$

1.4.6. Асимптотични функции и реални числа

Разгледаните по-горе релации между асимптотичните функции притежават свойства, близки до тези на класическите релации над реалните числа. Връзките се дават от *таблица 1.4.6*.

Връзката обаче е само частична, тъй като асимптотичните функции не притежават едно важно свойство на реалните числа: *трихотомичност*, т.е. за всеки две реални числа a и b е в сила *точно* една измежду релациите: $a < b$, $a = b$ и $a > b$. Така например, за функциите n и $n^{1+\cos n}$ не е

в сила нито една измежду горните релации. Доказателството на последното оставяме на читателя като леко упражнение.

асимптотична релация	*релация над реални числа*
$f \in O(g)$	$a \leq b$
$f \in \Omega(g)$	$a \geq b$
$f \in \Theta(g)$	$a = b$
$f \in o(g)$	$a < b$
$f \in \omega(g)$	$a > b$

Таблица 1.4.6. Връзка между релации над асимптотични функции и реални числа.

Задачи за упражнение:

1. Да се даде друг пример за двойка асимптотично несравними функции.
2. Необходимо ли е едната от функциите $f(n)$ и $g(n)$ да бъде композиция на две функции, една от които да бъде периодична, за да бъдат f и g асимптотично несравними?

1.4.7. Нарастване на основните функции

При оценка на сложността на алгоритми най-често се използват следните функции: c, $\log n$, n, $n \log n$, n^2, n^3, n^k, 2^n, $n!$, n^n. Тук сме ги подредили по-скорост на нарастване. За да придобие читателят по-добра представа за скоростта им на нарастване, прилагаме *таблица 1.4.7.*, показваща стойностите на функциите при различни стойности на аргумента им n.

Функция	***Стойност***				
	$n = 1$	$n = 2$	$n = 10$	$n = 100$	$n = 1000$
5	5	5	5	5	5
$\log n$	0	1	3,32	6,64	9,96
n	1	2	10	100	1000
$n \log n$	0	2	33,2	664	9966
n^2	1	4	100	10000	10^6
n^3	1	8	1000	10^6	10^9
2^n	2	4	1024	10^{30}	10^{300}
$n!$	1	2	3628800	10^{157}	10^{2567}
n^n	1	4	10^{10}	10^{200}	10^{3000}

Таблица 1.4.7. Нарастване на някои по-често използвани функции.

Непосредствено от *таблица 1.4.7.* се вижда, че:

- Експоненциалната функция нараства по-бързо от степенната:
  $n^k \in O(b^n)$, за всяко $b > 1$, $k \geq 0$, n – естествено.
- Логаритмичната функция нараства по-бавно от степенната:
  $\log_b n \in O(n^k)$, за всяко $b > 1$, $k > 0$, n – естествено.

Задачи за упражнение:

1. Да се докаже, че $n!$ расте по-бързо от c^n, за всяко $c > 0$.
2. Да се разположи функцията c^n, $c > 0$ в редицата на нарастване на най-често използваните функции (виж *таблица 1.4.7.*).

1.4.8. Определяне на сложност на алгоритъм

Ще разгледаме някои основни свойства, с помощта на които ще можем да определяме сложност на алгоритъм по дадена негова реализация на Си. Ще използваме T(**код**) за означаване сложността както на отделна операция, така и на програмен фрагмент. Понякога, когато кодът се подразбира, ще използваме означението $T(\boldsymbol{n})$, за да покажем изрично, че сложността е функция на променливата n. Ще работим изключително с нотацията $O(...)$. Оставяме на читателя да съобрази къде $O(...)$ може да се замени с по-точната оценка $\Theta(...)$.

- Елементарна операция

Сложността на елементарна операция е константа, т.е. $O(1)$.

Не е лесно да се дефинира какво е елементарна операция. При различни обстоятелства ще си позволяваме да променяме дефиницията. По принцип елементарна операция е такава, която се изпълнява за константно време, независещо от размера на обработваните данни. Елементарни операции в общия случай са например присвояването, събирането, умножението и др. Когато обаче работим със стоцифрени числа, едва ли е добре да приемем умножението за елементарна операция. Не е добре да приемаме за елементарни операции тригонометричните функции (синус, косинус и др.), експонентата, логаритъмът и други библиотечни функции в Си, които се пресмятат с *редове*. Обръщението към такава функция предизвиква цикъл за пресмятане на търсената стойност. От друга страна тези редове се пресмятат до достигането на *определена точност*, като броят на итерациите за пресмятане на търсената стойност може да се разглеждат като *независещи* от n. Дали и какво влияние оказва това на оценката на сложността на алгоритъма зависи от случая и следва да се разглежда конкретно.

- Последователност от оператори

Времевата сложност на последователност от оператори се определя от сложността на по-бавния от тях. Формално, ако операторът s_1 със сложност F_1 е следван от оператора s_2 със сложност F_2, можем да запишем:

$$T(s_1) \in O(F_1),\ T(s_2) \in O(F_2) \Rightarrow T(s_1; s_2) \in O(\max(F_1, F_2))$$

Това е еквивалентно на правилото:

$$f_1 + f_2 \in (\max(f_1, f_2))$$

- Композиция на оператори

При влагане на оператор в областта на действие на друг оператор сложността се пресмята като произведение от сложностите им, т.е.

$$T(s_1) \in O(F_1),\ T(s_2) \in O(F_2) \Rightarrow T(\mathrm{s}_1\{\mathrm{s}_2\}) \in O(F_1.F_2))$$

Това е еквивалентно на правилото:

$$f_1.f_2 \in O(f_1.f_2)$$

- if-конструкции

```
if (p)
  s1;
else
  s2;
```

Ако сложностите на p, s_1 и s_2 са $O(P)$, $O(F_1)$, $O(F_2)$, то сложността на показания фрагмент е $\max(O(P), O(F_1), O(F_2))$, т.е. сложността на най-бързо нарастващата функция измежду P, F_1 и F_2.

Тук предполагаме, че условието p не е булева константа, т.е. в зависимост от входните данни е възможно да бъде както истина, така и лъжа.

$$T(p) \in O(P),\ T(s_1) \in O(F_1)\ ,\ T(s_2) \in O(F_2) \Rightarrow T(\texttt{if}\ (p)\ s_1;\ \texttt{else}\ s_2) \in \max(O(P),\ O(F_1),\ O(F_2))$$

Ще се опитаме да докажем това свойство. Да предположим, че условието p е вярно. Тогава *if*-конструкцията е еквивалентна на последователността $\boldsymbol{p;\ s_1}$ (Забележете, че условието е било проверено.). По правилото за последователността сложността в този случай е $\max(O(P),O(F_1))$. По същия начин получаваме, че сложността в случай на невярно p е $\max(O(P),O(F_2))$. Тъй като предварително не е известно коя от двете последователности ще се изпълни, можем да ограничим отгоре сложността на *if*-конструкцията, предполагайки по-тежкия случай. Така получаваме:

$$T(\text{if}\ (p)\ s_1;\ \text{else}\ s_2) \in \max(\max(O(P),O(F_1)),\ \max(O(P),O(F_2))) = \max(O(P),\ O(F_1),\ O(F_2))$$

По подобен начин се извежда сложност на `switch` конструкции.

- Цикли

Да разгледаме цикъла:

```
fact = 1;
 for (i = 1; i <= n; i++)
   fact *= i;
```

Можем да считаме, че тялото на цикъла отнема константно време c, независещо от n. Сложността на оператора за цикъл *for* е $O(n)$. Тогава по правилото за композицията за сложност-та на целия цикъл получаваме $O(c.n)$, т.е. $O(n)$. Тук трябва да прибавим и сложността на началната инициализация преди цикъла (която има сложност $O(1)$), при което, по правилото за последователността, получаваме: $O(1+n)$. В крайна сметка сложността се оказва $O(n)$.

- Вложени цикли

```
sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    sum++;
```

Сложността при два или повече вложени цикли с взаимно независими броячи може да се изведе лесно. В случая на два вложени цикъла от фрагмента по-горе тя е $f \in n.O(g)$, където g е сложността на вътрешния цикъл. Но $g \in O(n)$, тогава $f \in O(n.n)$, т.е. $f \in O(n^2)$. Тук и по-долу често ще инициализираме специална променлива `sum` с 0, а после ще използваме `sum++` като най-вътрешен оператор в циклите. Това дава възможност на любознателния читател да провери на практика нашите теоретични разсъждения. За целта е достатъчно да наблюдава стойността на `sum` след изпълнение на фрагмента за различни стойности на n.

```
sum = 0;
for (i = 0; i < n-1; i++)
  for (j = i+1; j < n; j++)
    sum++;
```

В този пример на първата стъпка на външния цикъл, i=0, вътрешният ще се изпълни n–1 пъти. На втората стъпка, за i=1, вътрешният цикъл ще се изпълни n–2 пъти, след това n–3 пъти и т.н., като накрая, за i=n–2, ще се изпълни само веднъж. Имаме аритметична прогресия с пръв елемент 1, последен елемент n–1 и стъпка 1. Така `sum++` ще се изпълни $\frac{n.(n-1)}{2}$ пъти, т.е. сложността на фрагмента ще бъде $O(n^2)$. Тук приемаме, че изпълнението на `sum++` има константна сложност $O(1)$.

- Още примери от вложени цикли

Пример 1.

```
sum = 0;
for (i = 0; i < n*n; i++)
  sum++;
```

Сложността е $O(n^2)$. (*Защо?*)

Пример 2.

```
sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i == j)
      for (k = 0; k < n; k++)
        sum++;
```

Операторът *if* ще се изпълни n^2 пъти, но само n пъти резултатът от проверката `i == j` ще бъде истина. Тъй като сложността на най-вътрешния цикъл е линейна, получаваме обща сложност $O(n^2)$. Разбира се, оценката $O(n^3)$ също е вярна, но е по-неточна. На практика сложността тук е $\Theta(n^2)$.

Пример 3.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i == j)
      break;
```

Как може да се покаже, че сложността в горния фрагмент отново е $O(n^2)$, а не по-малко? Обърнете внимание, че, за разлика от предходния пример, тук липсва допълнителен линеен цикъл!

Пример 4.

```
sum = 0;
for (i = 1; i <= n; i++)
  for (j = 1; j <= i*i; j++)
    sum++;
```

Вътрешният цикъл ще се изпълни n пъти. От свойството:

$$\sum_{i=1}^{n} i^2 = \frac{n.(n+1)(2n+1)}{6}$$

получаваме, че сложността на фрагмента е $O(n^3)$. Всъщност бихме могли да решим, че сложността на външния цикъл е $O(n)$, а на вътрешния — $O(n^2)$, защото максималната стойност на i е n. Така при `i == n` имаме `i*i ==` n^2 и пак достигаме до сложност $O(n^3)$.

Пример 5.

```
sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < i*i; j++)
    for (k = 0; k < j*j; k++)
      sum++;
```

Отново е възможно да пресметнем сложността, като направим анализ на броя операции в най-вътрешния цикъл. На практика обаче е възможно да достигнем до същия резултат и далеч по-лесно, като използваме единствено елементарните свойства на нотацията $O(...)$: В предходния пример вече пресметнахме сложността на външните два вложени цикъла — $O(n^3)$. Последният цикъл сам по себе си е със сложност $O(n^4)$ — това се вижда лесно, ако разгледаме случая $i = n$, тогава горната граница за k е $j^2 = i^4 = n^4$. (*Каква е общата сложност на фрагмента и защо?*)

- Логаритмична сложност

Да разгледаме програмния фрагмент:

```
for(sum = 0, h = 1; h < n; h *= 2)
  sum++;
```

Тук h приема стойности $1,2,4,...,2^k,...$ докато достигне n. Така `sum++` се изпълнява $\lfloor \log_2 n \rfloor$ пъти и сложността на алгоритъма е $O(\log_2 n)$. Забележете, че сложността се задава с логаритъм, което е реална функция. Както по-горе отбелязахме, това не е проблем и ние ще разбираме подходящо целочислено приближение/ограничение.

Такава алгоритмична сложност има и *двоичното търсене*, при което на всяка стъпка интервалът на търсене се дели на две (почти) равни части. Няма да се спираме на този алгоритъм, тъй като по-нататък в книгата ще го разгледаме по-подробно (*виж 4.3.*).

По-долу вместо $O(\log_2 n)$, ще записваме $O(\log n)$. От една страна, изпускането на основата при двоичен логаритъм е стандартна конвенция. От друга, тя не е важна при асимптотична нотация, поради свойство (4) от *1.1.1. степен, логаритъм, n-ти корен.* Ако предпочитаме да работим с друга основа c ($c > 0$, $c \neq 1$) , нищо не се променя, защото $\log_2 x = \log_c x / \log_c 2$, но $\log_c 2$ е константа и се изпуска в асимптотичната нотация. Така $O(\log_2 n) \equiv O(\log_c n)$.

- Рекурсия

Анализирането на рекурсията в общия случай не е тривиално. Обикновено за сложността на алгоритъма се получава зависимост от вида $T(n) = f(T(n-1))$. За да намерим явния вид на сложността, се налага решаване на рекурентната зависимост, което в общия случай е тежко. За щастие в повечето интересни практически случаи това не е толкова трудно и понякога може да стане с други средства.

- Факториел

Да разгледаме функцията:

```
unsigned fact(unsigned n)
{ if (n < 2)
    return 1;
  return n*fact(n-1);
}
```

В този случай рекурсията е еквивалентна на единствен цикъл от тип *for*, откъдето за сложността лесно получаваме $O(n)$.

- Числа на Фибоначи

Не винаги обаче нещата са толкова прости. Да вземем за пример числата на Фибоначи, за които показахме в *1.2.2.*, че могат да бъдат намерени крайно неефективно с рекурсия:

```
unsigned fib(unsigned n)
{
  if (n < 2)
    return 1;
  return fib(n-1) + fib(n-2);
}
```

При $n = 0$ и $n = 1$ имаме константна времева сложност: една елементарна проверка и връщане на резултат. В другия случай имаме отново същата проверка, но този път последвана от две рекурсивни обръщения. Изобщо, в сила са формулите:

$$T(0) = T(1) = O(1)$$
$$T(n) = T(n-1) + T(n-2) + O(1),\ n \geq 2$$

Горните зависимости силно напомнят тези от дефиницията на числа на Фибоначи:

$$f_0 = f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Оттук непосредствено следва, че $T(n) \geq f_n$. За числата на Фибоначи обаче са в сила неравенствата: $\left(\frac{3}{2}\right)^{n-1} \leq f_n \leq 2^n,\ n \geq 1$ (Доказва се леко по индукция.). Така се оказва, че $T(n)$ расте експоненциално.

Ако обаче бъдем достатъчно съобразителни, за да не пресмятаме повторно нещо, което вече сме смятали, можем да намалим сложността на алгоритъма до $O(n)$:

```
unsigned long f[MAX] = {0,0,0,...};
unsigned long fib(unsigned n)
{
  if (0 == f[n])
    if (n < 2)
      f[n] = 1;
    else
      f[n] = fib(n-1) + fib(n-2);
   return f[n];
}
```

Току-що демонстрираната техника (*memoization*) е разгледана обстойно в *глава 8*.

Задачи за упражнение:

1. Да припомним, че за конструкцията `if-then-else` имахме

$$T(p) \in O(P),\ T(s_1) \in O(F_1)\ ,\ T(s_2) \in O(F_2) \Rightarrow T(\texttt{if}\ (p)\ s_1;\ \texttt{else}\ s_2) \in \max(O(P), O(F_1), O(F_2))$$

Да се намери израз от този тип за конструкцията `switch`.

2. Да се докаже, че сложността на програмния фрагмент от пример 1 по-горе е $\Theta(n^2)$.
3. Да се докаже, че сложността на програмния фрагмент от пример 3 по-горе е $\Theta(n^2)$.
4. Да се даде оценка от тип $\Theta(...)$ за сложността на програмния фрагмент от пример 5.
5. Безопасно ли е да се замени $O(...)$ с $\Theta(...)$ в цялата точка *1.4.8.*?
6. Да се докаже, че за числата на Фибоначи е в сила равенството: $\left(\frac{3}{2}\right)^{n-1} \leq f_n \leq 2^n,\ n \geq 1$.
7. Да се определи сложността $\Theta(...)$ на следния фрагмент:

```
for (i = 0; i < 2*n; i++)
  for (j = 0; j < 2*n; j++)
    if (i<j) for (k = 0; k < 2*n; k++) break;
```

8. Да се определи сложността $\Theta(...)$ на следния фрагмент:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
        for (j = i+1; j < i*i; j++) sum++;
```

9. Да се определи сложността $\Theta(...)$ на следния фрагмент:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i==j)
      for (k = 0; k < n; k++) break;
```

10. Да се определи сложността Θ(...) на следния фрагмент:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < i*i; j++) sum++;
```

11. Да се определи сложността Θ(...) на функцията `trib()`:

```
unsigned trib(unsigned n)
{
    if (n < 3)
      return 1;
    if ((n % 2) == 1)
      return trib(n-1) + trib(n-2) + trib(n-3);
    else
      return trib(n / 3) + trib(n / 2);
}
```

12. Да се определи сложността Θ(...) на функцията `streep()`:

```
unsigned fib(unsigned n)
{
   if (n < 2)
     return 1;
   return fib(n-1) + fib(n-2);
}
void streep(unsigned n) {
  fib(fib(n));
}
```

13. Да се определи сложността Θ(...) на следния фрагмент:

```
int n = 10;
int i;
for (i = 0; i < n; (n = i)++);
```

1.4.9. Характеристични уравнения

Да се върнем отново на числата на Фибоначи. Този път ще се съсредоточим не върху сложността на наивната рекурсивна реализация от предходния параграф, а по-скоро върху решаването на рекурентната зависимост, зададена от дефиницията на числата на Фибоначи. В резултат ще получим известната формула на Моавър. Но преди това — малко теория...

- Линейни хомогенни уравнения с прости корени

В процеса на анализ на сложността на компютърни алгоритми често се получават рекурентни зависимости от вида:

$$a_0T(n) + a_1T(n-1) + ... + a_kT(n-k) = 0 \qquad (1)$$

Тук $T(n)$ е неизвестна функция. За да можем да оценим сложността на алгоритъма асимптотично, се налага да намерим изцяло или поне частично явния вид на $T(n)$, т.е. да решим рекурентната зависимост. Стандартият подход е свеждане на задачата до решаване на съответно хомогенно уравнение. Да положим $T(n) = x^n$, при което получаваме:

$$a_0x^n + a_1x^{n-1} + \ldots + a_k\, x^{n-k} = 0 \qquad (2)$$

Едно очевидно решение на горното уравнение, както впрочем и на всяко хомогенно уравнение, е $x = 0$, което не представлява интерес за нас. За да намерим останалите решения, делим на x^{n-k} (за $x \neq 0$) и получаваме полином от степен k — характеристичен полином на изходната рекурентна зависимост:

$$a_0x^k + a_1x^{k-1} + \ldots + a_k = 0 \qquad (3)$$

Полученото уравнение е полином от степен k. Съгласно основната теорема на алгебрата, то има точно k корена (не непременно различни и не непременно реални, т.е. евентуално комплексни). Да ги означим с α_i ($1 \leq i \leq k$). Тогава имаме:

$$0 = a_0x^k + a_1x^{k-1} + \ldots + a_k = a_0(x-\alpha_1)(x-\alpha_2)\ldots(x-\alpha_k) \qquad (4)$$

Ако умножим уравнение (4) обратно по x^{n-k}, получаваме, че числата α_i ($1 \leq i \leq k$) са корени не само на уравнението (3), но и на (2). Връщайки се към полагането $T(n) = x^n$, получаваме, че α_i^n са корени на изходната рекурентна зависимост (1).

Уравнението (1) е хомогенно и има безброй много решения. В случай, че числата α_i са различни, то множеството от числа α_i^n представлява фундаментална система решения на (1), т.е. всички останали решения се получават като линейна комбинация на α_i^n. Така общото решение на (1) има вида:

$$T(n) = c_1\alpha_1^n + c_2\alpha_2^n + \ldots + c_k\alpha_k^n \qquad (5)$$

Тук коефициентите c_1, c_2, ..., c_n са константи, които еднозначно се определят от граничните условия. По-долу ще видим как става това на практика.

Пример:

Да разгледаме като пример числата на Фибоначи, които удовлетворяват рекурентната зависимост:

$$T(n) = T(n-1) + T(n-2)$$

Съответното характеристично уравнение от вида (4) е:

$$x^2 = x + 1$$

а корените му са:

$$\alpha_{1,2} = \frac{1 \pm \sqrt{5}}{2}$$

Търсим решение от вида (5), т.е.:

$$T(n) = c_1\alpha_1^n + c_2\alpha_2^n \qquad (6)$$

Константите c_1 и c_2 определяме от граничните условията $T(0) = 0$ и $T(1) = 1$, замествайки в (6), при което получаваме системата:

$$T(0) = 0 = c_1 + c_2$$
$$T(1) = 1 = c_1\alpha_1 + c_2\alpha_2$$

откъдето получаваме:

$$c_1 = \frac{1}{\sqrt{5}}, c_2 = -\frac{1}{\sqrt{5}}$$

Сега заместваме в (6) и получаваме явния вид на $T(n)$. Получената формула е известна като формула на Моавър.

$$T(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

Горната формула показва явно, че числата на Фибоначи растат експоненциално с нарастването на n. На пръв поглед формулата изглежда странно — съдържа ирационални числа. Оставяме на читателя да се увери, че те се съкращават и за всяко естествено n се получават съответните числа на Фибоначи.

- Линейни хомогенни уравнения с кратни корени

В случай на кратни корени на уравнението (4) общото решение има малко по-различен вид от (5). Ако α е двукратен корен, то корен ще бъде и $n\alpha^n$. Забележете, че този корен не може да се получи като линейна комбинация на останалите, т.е. той е съществено нов и трябва да се включи във фундаменталната система решения. Ако коренът е трикратен, освен $n\alpha^n$ следва да добавим и $n^2\alpha^n$. Изобщо, в случай на p-кратен корен α корени ще бъдат всички едночлени $n^s\alpha^n$, $0 \le s \le p$.

Да преименуваме корените α_i ($1 \le i \le k$) така, че да премахнем кратните корени. Нека *различните* корени (3) са $\beta_1, \beta_2, ..., \beta_l$, $1 \le l \le k$. Нека q_j е кратността на корена β_j, $1 \le j \le l$. Тогава общото решение на (1) ще има вида:

$$T(n) = \sum_{j=1}^{l} \sum_{r=0}^{q_j-1} c_{j,r+1} n^r \beta_j^n \qquad (7)$$

Числата $c_{j,r+1}$ са коефициенти, които се определят еднозначно, както и по-горе, от граничните условия.

Пример:

$$T(n) = \begin{cases} 3 & n = 0,1 \\ 5 & n = 2 \\ 4T(n-1) - 5T(n-2) + 2T(n-3) & n \ge 3 \end{cases}$$

Съответното характеристично уравнение има вида:

$$x^3 = 4x^2 - 5x + 2$$

Или, ако прехвърлим всичко отляво:

$$x^3 - 4x^2 + 5x - 2 = 0$$

Решавайки го, получаваме:

$$x_1 = x_2 = 1,\ x_3 = 2$$

Търсим явния вид на $T(n)$ във вида (7):

$$T(n) = c_{1,1}.1^n + c_{1,2}.n.1^n + c_{2,1}.2^n \qquad (8)$$

Константите $c_{1,1}$, $c_{1,2}$, и $c_{2,1}$ определяме от граничните условия за $n = 0$, 1 и 2. Получаваме системата:

$$\begin{aligned} T(0) &= 3 = c_{1,1} + c_{2,1} \\ T(1) &= 3 = c_{1,1} + c_{1,2} + 2c_{2,1} \\ T(2) &= 5 = c_{1,1} + 2c_{1,2} + 4c_{2,1} \end{aligned}$$

откъдето:

$$c_{1,1} = 1,\ c_{1,2} = -2,\ c_{2,1} = 2$$

Сега заместваме в (8) и получаваме:

$$T(n) = 1 - 2n + 2^{n+1}$$

- Линейни нехомогенни уравнения

В процеса на анализ на алгоритми често се налага анализиране на *нехомогенни* уравнения. Да разгледаме следната задача:

Задача: Дадена е функцията на Си:

```
float P(unsigned i, unsigned j)
{ if (0 == i)
    return 1.0;
  else if (0 == j)
    return 0.0;
  else
    return p * P(i - 1, j) + (1 - p) * P(i, j - 1);
}
```

Искаме да оценим времевата сложност на обръщението `P(n,n)` като функция на n.

Решение:

Ще решим по-общата задача за оценка на сложността на функцията относно параметрите i и j, след което ще видим какво се получава при $i = j = n$.

Лесно се вижда, че горната функция `P()` пресмята стойността на функцията $P(i,j)$, зададена с рекурентните равенства:

$$P(0,j) = 1,\ j = 1,2,\ldots,n$$
$$P(i,0) = 0,\ i = 1,2,\ldots,n$$
$$P(i,j) = p.P(i-1,j) + (1-p).P(i,j-1)\ ,\ i > 0\ ,\ j > 0$$

Тук сложността зависи от два параметъра, което затруднява пресмятането й. Лесно обаче се забелязва, че нещата са симетрични по отношение на i и j. Така, полагайки $k = i + j$, получаваме:

$$T(1) = c$$
$$T(k) = 2T(k-1) + d,\ k > 1$$

Забележете, че тук се опитваме не да намерим *явния* вид на функцията $P(i,j)$, а да намерим *асимптотична оценка* на програмната реализация на функцията `P()`. Това ни дава право да не се задълбочаваме над стойностите на константите c и d, освен ако наистина не се налага. По-долу ще видим, че в нашия случай конкретните им *стойности* не са от значение.

Нека първо разгледаме прост начин за получаване на характеристичното уравнение в нашия случай, където в рекурентната връзка участва константа. Записваме рекурентанта зависимост за два последователни члена на редицата:

$$T(k) = 2T(k-1) + d$$
$$T(k+1) = 2T(k) + d$$

Вадим ги почленно, при което константата се съкращава, и получаваме:

$$T(k+1) - T(k) = 2.[T(k) - T(k-1)]$$

което е еквивалентно на:

$$T(k+1) - 3T(k) + 2T(k-1) = 0$$

Съответното характеристично уравнение е:

$$x^2 - 3x + 2 = 0$$

или

$$(x-1)(x-2) = 0,$$

откъдето директно се вижда, че корените на уравнението са $\alpha_1 = 2$ и $\alpha_2 = 1$. Т.е. търсим явния вид на $T(k)$ във вида:

$$T(k) = c_1\alpha_1^k + c_2\alpha_2^k$$

Или след заместване:

$$T(k) = c_1 2^k + c_2$$

Коефициентите c_1 и c_2 можем да намерим, използвайки стойностите на $T(1)$ и $T(2)$. Получаваме системата:

$$T(1) = c = 2c_1 + c_2$$
$$T(2) = 2c + d = 4c_1 + c_2$$

откъдето:

$$c_1 = (c + d) / 2$$
$$c_2 = -d$$

От $c_1 \neq 0$ веднага следва, че $T(k) \in \Theta(2^k)$. (*Защо?*) Сега да се върнем към полагането $k = i + j$, откъдето получаваме $\Theta(2^{i+j})$. Да си припомним, че искахме да оценим сложността на програмния фрагмент при обръщението `P(n,n)`, т.е. за $i = j = n$. Така получаваме $\Theta(2^{2n})$ или окончателно: $\Theta(4^n)$.

Теория:

Да разгледаме нехомогенното уравнение:

$$a_0T(n) + a_1T(n-1) + \ldots a_kT(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + \ldots + b_s^n p_s(n) \quad (9)$$

Тук $b_1, b_2, \ldots, b_s$ са различни реални константи, а $p_1(n), p_2(n), .., p_s(n)$ са полиноми на n. Нека степените на полиномите са съответно $d_1, d_2, \ldots, d_s$. Тогава характеристичното уравнение на рекурентната зависимост (9) има вида:

$$(a_0x^k + a_1x^{k-1} + \ldots + a_k)(x-b_1)^{d_1+1}(x-b_2)^{d_2+1}\ldots(x-b_s)^{d_s+1} = 0 \quad (10)$$

Полученото уравнение (10) по-нататък се решава, както в хомогенния случай.

Пример:

Да разгледаме рекурентната зависимост:

$$T(n) = \begin{cases} 0 & n = 0 \\ 2T(n-1) + n + 2^n & n \geq 1 \end{cases}$$

Очевидно това е нехомогенна рекурентна зависимост:

$$T(n) - 2T(n-1) = n + 2^n, \quad (11)$$

като $b_1 = 1$, $b_2 = 2$, $p_1(n) = n$, $p_2(n) = 1$, и оттук: $d_1 = 1$, $d_2 = 0$. Тогава съответното характеристично уравнение има вида:

$$(x-2)(x-1)^2(x-2) = 0.$$

Корените на уравнението са $x_1 = x_2 = 1$, $x_3 = x_4 = 2$. Търсим явния вид на $T(n)$:

$$T(n) = c_{1,1}.1^n + c_{1,2}.n.1^n + c_{2,1}.2^n + c_{2,2}.n.2^n \quad (12)$$

За да намерим константите, този път граничното условие няма да бъде достатъчно. Налага се пресмятане на стойността на $T(n)$ за следващите три стойности. Така получаваме системата:

$$T(0) = 0 = c_{1,1} + c_{2,1}$$
$$T(1) = 3 = c_{1,1} + c_{1,2} + 2c_{2,1} + 2c_{2,2}$$
$$T(2) = 12 = c_{1,1} + 2c_{1,2} + 4c_{2,1} + 8c_{2,2}$$
$$T(3) = 35 = c_{1,1} + 3c_{1,2} + 8c_{2,1} + 24c_{2,2}$$

Решаваме я и получаваме $c_{1,1} = -2$, $c_{1,2} = -1$, $c_{2,1} = 2$ и $c_{2,2} = 1$. Сега заместваме в (12) и за $T(n)$ окончателно получаваме:

$$T(n) = -2 - n + 2^{n+1} + n.2^n \quad (13)$$

Горната система има четири неизвестни, поради което решаването ѝ е неприятно (макар и не толкова тежко). Бихме могли да намерим коефициентите по друг начин — като заместим (12) в (11), при което получаваме:

$$c_{2,2}2^n - c_{1,2}n + (2c_{1,2} - c_{1,1}) = 2^n + n \qquad (14)$$

Приравнявайки коефициентите пред 2^n от двете страни на равенството, получаваме $c_{2,2} = 1$. Аналогично, приравняването на коефициентите пред n ни дава $c_{1,2} = -1$. Сега, знаейки $c_{1,2}$, от приравняването на свободните членове получаваме $c_{1,1} = -2$. Константата $c_{1,2}$ не можем да получим по този начин, но пък това става директно, ако използваме граничното условие.

Да предположим, че уравнението от последния пример е получено в резултат на анализ на някакъв алгоритъм. Това, от което в такъв случай се интересуваме, не е толкова *явният вид* на $T(n)$, колкото *асимптотичната му оценка*. Разбира се, знаейки явния вид (13) на $T(n)$, веднага получаваме, че $T(n) \in \Theta(n2^n)$. Бихме могли обаче да установим това с по-малко усилия. Наистина, още от (12) става ясно, че $T(n) \in O(n2^n)$. В случай, че този резултат ни удовлетворява, бихме могли да спрем дотук. Ако обаче държим да получим оценка от вида $\Theta(...)$, това не е достатъчно, тъй като коефициентът $c_{2,2}$ пред най-бързо нарастващата функция $n2^n$ би могъл да се окаже 0. Да си припомним, че от (14) директно следва, че $c_{2,2} = 1$, т.е. $c_{2,2} \neq 0$ и оттук заключаваме, че $T(n) \in \Theta(n2^n)$. В случай, че $c_{2,2}$ се беше оказало 0, щяхме да проверим коефициента $c_{2,1}$ пред следващата най-бързо нарастваща функция 2^n и т.н.

Следва да се отбележат две неща:

1) **Не е непременно необходимо да се намерят *всички* коефициенти.** В нашия случай беше достатъчно да покажем, че $c_{2,2} \neq 0$.

2) **Граничните условия не са непременно от значение.** Никъде не сме ползвали условието $T(0) = 0$, което означава, че резултатът $T(n) \in \Theta(n2^n)$ ще бъде валиден при *произволна* стойност на $T(0)$. Разбира се, не можем да очакваме, че това ще бъде така за *всяка* рекурентна зависимост от вида (9). Да си припомним, че $c_{2,1}$ не можеше да се получи с просто изравняване на коефициентите в (13) и следователно зависи от $T(0)$. А $c_{2,1}$ можеше да бъде коефициентът пред най-бързо растящата функция, при което $T(0)$ щеше да има значение.

Задача за упражнение:

Да се определи сложността на алгоритъм, зададена с рекурентната зависимост:

а) $T(1) = 1, T(n) = 4T(n-1) - 2^n, n \geq 2$
б) $T(1) = 0, T(n) = 2T(n-1) + n + 2^n, n \geq 2$
в) $T(0) = 1, T(n) = 2T(n-1) + n, n \geq 1$
г) $T(1) = 1, T(n) = 2T(n-1) + 3^n, n \geq 2$
д) $T(1) = 2, T(2) = 3, T(n) = 2T(n-1) - T(n-2), n \geq 3$

1.4.10. Специални техники за анализ на алгоритми

- Използване на барометър

Да разгледаме отново следния програмен фрагмент:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < i*i; j++)
    sum++;
```

По-горе определихме сложността му, използвайки свойства на сумите. Можем да подходим по друг начин: избираме подходяща инструкция (барометър) и следим колко пъти се изпъл-

нява. Това ни освобождава от грижата за анализа на всички останали инструкции, които нямат отношение към избраната. Как да избираме барометъра? Това трябва да бъде такава инструкция, която се изпълнява *поне толкова често*, колкото *всяка друга* инструкция в програмата. В горния програмен фрагмент подходящ кандидат за това е `sum++`. Впрочем, стойността на променливата `sum` след изпълнение на фрагмента ще ни даде броя на изпълненията на инструкцията `sum++`.

- Амортизационен анализ

При анализ на компютърни алгоритми най-често изследваме поведението им в най-лошия или в средния случай и почти никога не се интересуваме как се държат в най-добрия случай. Често използвана техника при анализ на алгоритъм с цел определяне на сложността му в най-лошия случай е да се предполага, че на всяка негова стъпка настъпва най-лошото възможно стечение на обстоятелствата. Това ни дава коректна сложност $O(...)$, но не винаги ни дава вярна оценка за $\Theta(...)$: резултатите често са доста песимистични и на практика алгоритъмът работи доста по-бързо дори в най-лошия случай. Причината е, че най-тежкият случай не винаги може да възниква на *всяка* стъпка. Често за това е необходимо да са се случвали няколко пъти по-леки случаи. Така общата сложност може да се окаже значително по-добра от предвидената.

Да разгледаме като пример следната функция:

```
void addOne(char c[], unsigned m)
{ int i;
  for (i = 0; 1 != c[i] && i < m; i++)
    c[i] = 1 - c[i];
}
```

Функцията получава като параметър масив с m елемента, съдържащ нули и единици, и го разглежда като двоичен запис на естествено число, като най-младшата цифра е в елемент 0. Тя прибавя единица, при което в масива се получава двоичният запис на увеличеното число. В случай на препълване се получава 0. Каква е сложността на алгоритъма? Очевидно, най-тежкият случай е когато числото съдържа m единици (т.е. когато настъпва препълването), защото тогава се извършва преминаване през *всички* елементи на масива. Сложността в този случай очевидно е $O(m)$ и съответно $\Theta(m)$. Да предположим, че ползваме `addOne()` като брояч и сме го извикали n пъти, като $n = 2^m-1$. Каква е общата времева сложност? Предполагайки на всяка стъпка най-лошия случай, достигаме до сложност $O(n.m)$, т.е. $O(n.\log_2 n)$. Това е вярна оценка, но при по-внимателен анализ се вижда, че не можем всеки път да достигаме до толкова тежък случай. Ще отбележим, че в процеса на броене получаваме всяко двоично число в интервала $[0; 2^m-1]$ точно веднъж. Половината от тези числа са четни и за тях ще имаме само едно завъртане по цикъла. Една четвърт от останалите (нечетните) имат предпоследен бит, установен в 0, и за тях ще имаме две завъртания по цикъла. Една осма от останалите ще имат 0 в предпредпоследния бит (и 1 — в последните два бита) и ще изискват три завъртания по цикъла и т.н. И само в един-единствен случай ще имаме 1 във всички позиции, т.е. колкото по-тежък е случаят, толкова по-рядко се случва. Оставяме на читателя да покаже, че сумарният брой завъртания цикъла е $2n-1$. Друга възможно разсъждение, което отново ни води до същия асимптотичен резултат, е, че можем да броим до n за време $\Theta(n)$, което е доста по-точна оценка.

- Основна теорема

При анализ на алгоритми от типа *разделяй и владей* (*виж глава 7*) често възникват рекурентни зависимости от вида:

$$T(n) = a.T(n/b) + c.n^k$$

Теорема 1. Нека е дадена рекурентната зависимост $T(n) = a.T(n/b) + c.n^k$, $n > n_0$, $a \geq 1$, $b > 1$, $k \geq 0$, $c > 0$, $n_0 \geq 1$ и a, b, k, c, n_0 са цели числа. Решението ѝ се дава от формулата:

$$T(n) = \begin{cases} \Theta(n^k) & a < b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^{\log_b a}) & a > b^k \end{cases}$$

Пример 1:
Да разгледаме рекурентната зависимост:

$$T(1) = 1$$
$$T(n) = 3T(n/2) + n,\ n \geq 2$$

Като използваме теорема 1, получаваме: $a = 3$, $b = 2$, $c = 1$, $k = 1$. Имаме: $3 = a > b^k = 2$. Оттук попадаме в третия случай и директно получаваме сложност $\Theta(n^{\log 3})$.

Пример 2:
Да разгледаме рекурентната зависимост:

$$T(1) = d$$
$$T(n) = 4T(n/2) + n^2,\ n \geq 2$$

Като използваме теорема 1, получаваме: $a = 4$, $b = 2$, $c = 1$, $k = 2$. Имаме: $4 = a = b^k = 2^2$. Оттук попадаме във втория случай и директно получаваме сложност $\Theta(n^2.\log_2 n)$. *Забележете, че няма значение каква е стойността на d.*

Пример 3:
Да разгледаме рекурентната зависимост:

$$T(1) = d$$
$$T(n) = 2T([\sqrt{n}]) + \log_2 n,\ n \geq 2$$

Полагаме $m = \log_2 n$ и получаваме:

$$T(2^m) = 2T(2^{m/2}) + m$$

Сега полагаме $S(m) = T(2^m)$:

$$S(m) = 2S(m/2) + m$$

Сега вече можем да използваме теорема 1: $a = 2$, $b = 2$, $c = 1$, $k = 1$. Имаме: $2 = a = b^k = 2^1$. Попадаме във втория случай и получаваме $\Theta(m.\log_2 m)$. Връщаме се към полаганията:

$$T(n) = T(2^m) = S(m) = \Theta(m.\log_2 m) = \Theta(\log_2 n.\log_2(\log_2 n))$$

Теорема 1 е частен случай на по-общата теорема 2:

Теорема 2. Нека е дадена рекурентната зависимост $T(n) = a.T(n/b) + f(n)$, $n > n_0$, $a \geq 1$, $b > 1$, $n_0 \geq 1$ и a,b,n_0 са цели числа. Тогава:

1) ако $f(n) \in O(n^{\log_b a-\varepsilon})$, за всяко $\varepsilon > 0$, то $T(n) \in \Theta(n^{\log_b a})$.

2) ако $f(n) \in \Theta(n^{\log_b a})$, за всяко $\varepsilon > 0$, то $T(n) \in \Theta(n^{\log_b a} \log_2 n)$.

3) ако $f(n) \in \Omega(n^{\log_b a+\varepsilon})$, за всяко $\varepsilon > 0$, $a.f(n/b) \leq c.f(n)$, за $c < 1$ и достатъчно големи n, то $T(n) \in \Theta(f(n))$.

Задача за упражнение:

Да се определи сложността на алгоритъм, ако е зададена с рекурентната зависимост:

а) $T(1) = 2$, $T(n) = 4T(n/3) + n^{\log_3 4}$, $n \geq 2$.

б) $T(1) = 0$ и $T(2n+1) = T(2n) = T(n) + \log_2 n$, $n \geq 2$.
в) $T(1) = 4$ и $T(n) = 2T([\sqrt{n}\,]) + \log_2 n$, $n \geq 2$.

1.4.11. Проблеми на асимптотичната нотация

Въпреки че ни дават възможност да поставим на здрава теоретична основа оценяването на сложността на алгоритмите и програмите, следва да се отнасяме с известно недоверие към асимптотичните оценки. Основна тяхна характеристика е, че се интересуват от поведението на алгоритъма при *неограничено* нарастване на n. *Безкрайно големи числа* в реалния *компютърен* свят обаче няма. Това означава, че оценката, която ни дава асимптотичната функция, може да се окаже неадекватна: например, защото крие константите, или пък защото не се интересуваме от толкова големи стойности на n. Да предположим, че искаме да пресметнем стойността на функция, но само за аргументи от интервала [0,65535], и разполагаме с два алгоритъма: линеен ($200000n$) и квадратичен ($2n^2$). Погледнато асимптотично, линейният алгоритъм би трябвало да бъде по-бърз от квадратичния. Това е така, но от определено място нататък. За наше нещастие това е $n = 100000 > 65535$. Така, за всички практически аргументи на функцията квадратичният алгоритъм ще бъде по-бърз.

По-нататък ще се сблъскваме често с този проблем и ще се научим да го използваме. Така например, бързото сортиране (*виж 3.1.6.*) в средния случай има сложност $\Theta(n.\log_2 n)$, а сортирането чрез вмъкване (*виж 3.1.2.*) — $\Theta(n^2)$. За достатъчно големи стойности на броя сортирани елементи n бързото сортиране е много по-бързо, но при 10-20 елемента сортирането чрез вмъкване е за предпочитане. На пръв поглед това едва ли е от значение, тъй като при такъв малък брой елементи бързото сортиране си остава приемливо бързо и едва ли си струва да се ползва сортиране чрез вмъкване. При по-внимателен поглед обаче става ясно, че бързото сортиране *многократно* разглежда дялове с малък размер, като разликата се натрупва. Ето защо обикновено двата алгоритъма се комбинират, при което се постига значително подобрение.

Задачи за упражнение:

1. Дадени са три алгоритъма със сложности съответно: $5n^2 - 7n + 13$, $3n^2 + 15n + 100$ и $1000n$. За всеки алгоритъм да се определи интервал от стойности n, за които той е по-бърз от другите два.
2. Основателна ли е критиката, отправена към асимптотичната нотация?

1.5. Въпроси и задачи

1.5.1. Задачи от текста

Задача 1.1.

Дадени са множествата $A = \{1,2,4,5,7\}$ и $B = \{2,3,4,5,6\}$. Да се определят множествата: $A \cup B$, $A \cap B$, $A \setminus B$, $B \setminus A$. (*виж 1.1.1.*)

Задача 1.2.

Дадени са две множества A и B и е известно, че $A \cap B = A$. Какво можете да кажете за множеството B? (*виж 1.1.1.*)

Задача 1.3.

Операцията *симетрична разлика* $\oplus$ на A и B се дефинира така: $A \oplus B = (A \cup B) \setminus (A \cap B)$. Да се определи симетричната разлика на множествата $A = \{1,2,4,5,7\}$ и $B = \{2,3,4,5,6\}$. (*виж 1.1.1.*)

Задача 1.4.

Една операция • се нарича *симетрична*, ако $A \bullet B = B \bullet A$. Кои от изброените операции са симетрични: $A \cup B$, $A \cap B$, $A \backslash B$, $B \backslash A$, $A \oplus B$? (*виж 1.1.1.*)

Задача 1.5.

Да се покаже, че реалните числа, които могат да се представят в стандартните реални типове числа с плаваща запетая всъщност са крайно подмножество на рационалните числа (*виж 1.1.1.*).

Задача 1.6.

Да се намерят частното и остатъка от делението на m на n, ако (m,n) е: (7,3), (–7,3), (7,–3), (–7,–3), (3, 7), (–3,7), (3,–7), (–3,–7). (*виж 1.1.1.*)

Задача 1.7.

При дадени цели m и n ($m \neq 0$) се опитайте да обосновете съществуването и единствеността на представянето $n = q.m + r$, $0 \leq r < m$, (q, r цели). (*виж 1.1.1.*)

Задача 1.8.

Да се предложи алгоритъм за намиране броя на цифрите на реално число. Какви проблеми възникват? (*виж 1.1.1.*)

Задача 1.9.

Да се изведе формулата за сума на аритметична прогресия. (*виж 1.1.1.*)

Задача 1.10.

Да се докажат свойства (1), (2) и (3) от *1.1.1.* на сумата.

Задача 1.11.

Да се формулират и докажат за произведение свойства, аналогични на (1), (2) и (3) от *1.1.1.*

Задача 1.12.

Като се използват свойства (1), (2) и (3) от *1.1.1.* на сумата, да се провери валидността на равенството:

$$\sum_{i=1..n} a_i + \sum_{i=n..m} a_i = \sum_{i=1..m} a_i + a_n \quad , 1 \leq n \leq m$$

Задача 1.13.

Да се докажат свойства на степенната функция, като се изходи от *дефиниция 1.13*.

Задача 1.14.

Да се докажат свойствата (1), (2) и (3) на логаритъма, като се изходи от дефиницията на логаритъм.

Задача 1.15.

Да се предложи рекурентна формула за повдигане на x в степен y ($x \in R$, $y \in N$).

Задача 1.16.

Да се предложи рекурентна формула за намиране на най-големия общ делител на две естествени числа.

Задача 1.17.

Дадена е матрица `unsigned a[MAX][MAX]`. Да се напише функция

```
void fillMatrix(unsigned a[][MAX], unsigned n),
```

която запълва с числа елементите на `a[][]` по следния начин:

0	20	19	17	14
1	0	18	16	13
2	5	0	15	12
3	6	8	0	11
4	7	9	10	0

Задача 1.18.

Дадена е матрица `unsigned a[MAX][MAX]`. Да се напише функция, която извежда елементите ѝ по спирала, например за $n = 5$ имаме:

1	16	15	14	13
2	17	24	23	12
3	18	25	22	11
4	19	20	21	10
5	6	7	8	9

Задача 1.19.

Да се докаже, че за всяко естествено число P броят на десетичните му цифри е равен на $[1+\log_{10}(P)]$. (*виж 1.1.2.*)

Задача 1.20.

Защо намирането на точна формула за $\pi(x)$ ще даде отговор и на трите въпроса за простите числа, формулирани по-горе? (*виж 1.1.3.*)

Задача 1.21.

Да се напишат програми за проверка на хипотезите на Голдбах. (*виж 1.1.3.*)

Задача 1.22.

Да се напише програма за проверка на теоремата на Оперман. (*виж 1.1.3.*)

Задача 1.23.

Да се напише програма за проверка дали едно число е просто, като се ползва теоремата на Уилсън. Необходимо ли е да се пресмята $(p-1)!$, като се има предвид, че ни интересува само остатъкът му по модул p? (*виж 1.1.3.*)

Задача 1.24.

Да се докаже, че за да докажем, че p е просто, е достатъчно да сме сигурни, че не се дели на нито едно друго *просто* число от интервала $[2, \sqrt{p}\,]$. (*виж 1.1.3.*)

Задача 1.25.

Алгоритъмът за намиране на прости числа в интервал (Решето на Ератостен — (*виж 1.1.3.*) може да се подобри: на стъпка 3) търсенето започва от $k = i^2$, като нарастването със стойност i се запазва. Да се обоснове този резултат и да се модифицира по подходящ начин реализацията. Да се сравни с изходната.

Задача 1.26.

Да се обоснове алгоритъмът на решетото на Ератостен. (*виж 1.1.3.*)

Задача 1.27.

Да подобри алгоритъмът за търсене на прости числа в интервал, като за целта се разгледат първите 4 прости числа. (*виж 1.1.3.*)

Задача 1.28.

Да се обоснове предложеният в *1.1.3.* алгоритъм за факторизация.

Задача 1.29.

Да се обоснове предложеният в *1.1.3.* алгоритъм за намиране броя на нулите, на които завършва произведение.

Задача 1.30.

Да се обоснове формулата от *1.1.3.* за намиране броя на нулите, на които завършва $n!$

Задача 1.31.

Да се напише програма за намиране на първите n Мерсенови числа, без да се използва теоремата на Лукас-Лемер. (*виж 1.1.4.*)

Задача 1.32.

Да се напише програма за намиране на първите n Мерсенови числа, като се използва теоремата на Лукас-Лемер. Да се сравни по ефективност с предходната задача. (*виж 1.1.4.*)

Задача 1.33.

Може ли 2^n-1 да бъде просто, ако n не е просто? (*виж 1.1.4.*)

Задача 1.34.

Да се докаже, че сумата от реципрочните стойности на делителите (включително 1 и самото число) на всяко съвършено число е 2. Например, за 6 имаме: 1/1 + 1/2 + 1/3 + 1/6 = 2 (*виж 1.1.4.*).

Задача 1.35.

Да се напише програма, която намира всички числа, за които сумата от реципрочните стойности на делителите (включително 1 и самото число) е 2. Съвършени ли са? (*виж 1.1.4.*)

Задача 1.36.

Да се докажат свойства (1), (2) и (3) от *1.1.5.*, като се използва дефиницията на биномен коефициент. Например, доказателството на (1) може да се извърши по следния начин:

$$\binom{n}{0} = C_n^0 = \frac{n!}{0!(n-0)!} = \frac{n!}{0!n!} = 1$$

$$\binom{n}{n} = C_n^n = \frac{n!}{n!(n-n)!} = \frac{n!}{n!0!} = 1$$

Задача 1.37.

Да се предложат комбинаторни доказателства на свойства (1), (2) и (3) от *1.1.5.*

Задача 1.38.

При предложения в *1.1.5. Алгоритъм 1* (🖫pascalt.c) за намиране на C_n^k е необходим масив `lastLine[]` с n+1 елемента и, при по-големи стойности на n, заделянето на толкова памет ще бъде невъзможно. Възможно е да се модифицира по следния начин: вътрешният цикъл, попълващ поредния ред от триъгълника, да се изпълнява не от 1 до i, а от 1 до k, тъй като отрязъкът от триъгълника по-нататък не ни интересува. Да се обърне внимание, че когато k е близо до n, може да се приложи свойство (2) и да се търси C_n^{n-k}, вместо C_n^k.

Задача 1.39.

Защо основата на бройната система трябва да бъде по модул различна от 0 и 1? (*виж 1.1.6.*)

Задача 1.40.

Да се представят числата 17 и –17 в бройна система с основа: 2; 8; 16. (*виж 1.1.6.*)

Задача 1.41.
Да се представят числата 17 и –17 в бройна система с основа –2; –8; –16. (*виж 1.1.6.*)

Задача 1.42.
Без да се преминава през десетична система, да се преобразуват в шестнадесетична бройна система двоичните числа: 111, 110100, 1110100101, 10010101, 10101010101 и 10111110101. (*виж 1.1.6.*)

Задача 1.43.
Без да се преминава през десетична система, да се преобразуват в осмична бройна система двоичните числа: 11, 11001, 1010101, 111111, 1010101000, 10101101000 и 11010111000. (*виж 1.1.6.*)

Задача 1.44.
Как изглежда полиномиалният запис на число в симетрична бройна система? (*виж 1.1.6.*)

Задача 1.45.
Да се представят числата 17 и –17 в троична симетрична бройна система. (*виж 1.1.6.*)

Задача 1.46.
Да се запише числото 157 в бройна система с основа: 3;5;7;14. (*виж 1.1.6.*)

Задача 1.47.
Да се запише числото 0,321 в бройна система с основа: 3;5;7;14. (*виж 1.1.6.*)

Задача 1.48.
Да се запише числото 157,321 в бройна система с основа: 3;5;7;14. (*виж 1.1.6.*)

Задача 1.49.
Да се обоснове предложеният в *1.1.6.* алгоритъм за преобразуване на естествено число от десетична в p-ична бройна система.

Задача 1.50.
Да се обоснове предложеният в *1.1.6.* алгоритъм за преобразуване на дробната част на десетична дроб число от десетична в p-ична бройна система.

Задача 1.51.
Да се намери десетичният запис на числото $126_{(8)}$; $10101_{(2)}$; $3F2B_{(16)}$; $3CB_{(14)}$. (*виж 1.1.6.*)

Задача 1.52.
Да се намери десетичният запис на числото $0{,}233_{(8)}$; $0{,}01_{(2)}$; $0{,}34_{(16)}$; $0{,}2A_{(14)}$. (*виж 1.1.6.*)

Задача 1.53.
Да се намери десетичният запис на числото $126{,}233_{(8)}$; $10101{,}01_{(2)}$; $3F2B{,}34_{(16)}$; $3CB{,}2A_{(14)}$. (*виж 1.1.6.*)

Задача 1.54.
Да се обоснове предложеният в *1.1.6.* алгоритъм за преобразуване от p-ична в десетична бройна система.

Задача 1.55.
Да се запишат в римска бройна система числата: 10; 19; 159; 763; 1991; 1979; 1997; 2002. (*виж 1.1.7.*)

Задача 1.56.
Да се запишат в римска бройна система числата: 0; –10; 0,28; 3,14; 1/7. (*виж 1.1.7.*)

Задача 1.57.

Да се обоснове предложеният в *1.1.7.* алгоритъм за преобразуване на десетично число в римска бройна система.

Задача 1.58.

Да се запишат в десетична бройна система римските числа: DCLXXXIV, DCCLXIV, LX, LXX, LXXX, XL, XXL, XXXL (*виж 1.1.7.*).

Задача 1.59.

Да се обоснове предложеният в *1.1.7.* алгоритъм за преобразуване на римско число в десетична бройна система.

Задача 1.60.

Да се напише програма за проверка дали дадена последователност от римски цифри е коректно записано римско число, без да се преминава през десетична бройна система. За целта наблюдавайте внимателно закономерностите в *таблица 1.1.7.*

Задача 1.61.

Да се покаже, че отношението на две последователни числа на Фибоначи клони към златното сечение (*виж 1.2.2.*).

Задача 1.62.

Да се докаже еквивалентността на двете итеративни реализации на функция за намиране на n-тото число на Фибоначи: с три и с две променливи. (*виж 1.2.2.*)

Задача 1.63.

Да се намери *НОД* на: (10,5); (5,10); (15,25); (25,15); (7,8,9); (3,6,9); (158,128,256); (64,28,72,18) (*виж 1.2.3.*).

Задача 1.64.

Да се напише програма за съкращаване на обикновени дроби. Така например при вход 10/15 да дава 2/3. (*виж 1.2.3.*)

Задача 1.65.

Да се докаже, че $(a_1, a_2, ..., a_n) = ((a_1, a_2, ..., a_{n-1}), a_n)$. (*виж 1.2.3.*)

Задача 1.66.

Да се реализира алгоритъмът на Евклид с изваждане. (*виж 1.2.3.*)

Задача 1.67.

Да се докаже, че $(a, b) = (b, a \% b)$. (*виж 1.2.3.*)

Задача 1.68.

Как трябва да се модифицират алгоритъмът и програмата от *1.2.3.*, така че по дадени цели числа a и b, освен *НОД* да намира два цели множителя x и y ($x, y \in Z$), за които $(a,b) = ax + by$?

Задача 1.69.

Да се предложи и реализира друг алгоритъм за намиране на *НОД*, като се изхожда от основната теорема на аритметиката. Да се сравни ефективността му с тази на предложената тук реализация при 2; 5; 100; 1000 числа. (*виж 1.2.3.*)

Задача 1.70. *Задача на Поасон за трите съда*

Дадени са три съда с вместимост a, b и c литра ($c > a > b$, $c \geq a + b - 1$, a, b и c са естествени числа) най-големият от които е пълен, а другите два са празни. Да се отмерят d литра, където d е естествено число и $0 < d \leq a$. Допустими са само такива преливания от един съд в друг, при които става максимално напълване и/или максимално изпразване на някой от съдовете. (*виж 1.2.3.*)

Задача 1.71.
Да се намери: [10,15]; [15,10]; [7,8,9]; [3,6,9]; [158,128,256]; [64,28,72,18]. (*виж 1.2.4.*)

Задача 1.72.
Да се докаже, че $[a_1, a_2, ..., a_n] = [[a_1, a_2, ..., a_{n-1}], a_n]$. (*виж 1.2.4.*)

Задача 1.73.
Да се докаже, че $[a,b] = ab / (a,b)$. (*виж 1.2.4.*)

Задача 1.74.
Да се предложи и реализира друг алгоритъм за намиране на *НОК*, като се изхожда от основната теорема на аритметиката. Да се сравни ефективността му с тази на предложената тук реализация при 2; 5; 100; 1000 числа. (*виж 1.2.4.*)

Задача 1.75.
Да се направи предположение за възможна причина за различното изпълнение на 🖫factrec.c под *Borland C* за *DOS* и *Microsoft Visual C++* за *Windows*. Можете ли да предложите "безопасен" вариант? (*виж 1.2.5.*)

Задача 1.76.
Да се направи предположение за резултата от изпълнението на фрагмента по-долу. Очаквате ли разлики под *Borland C* за *DOS* и *Microsoft Visual C++* за *Windows*? Отговаря ли предположението Ви на действителния резултат?

```
unsigned i = 1;
printf("%u %u", ++i, i);
```

А какво мислите за фрагмента:

```
unsigned i = 1;
printf("%u %u", i, ++i);
```

Задача 1.77.
Да се направи предположение за стойността на променливата `x` след изпълнение на програмния фрагмент по-долу. Очаквате ли разлики под *Borland C* за *DOS* и *Microsoft Visual C++* за *Windows*? Отговаря ли предположението Ви на действителния резултат?

```
unsigned x, a = 3, b = 5;
x = a+++b;
```

Задача 1.78.
Въз основа на резултатите от предходните три задачи да се направят препоръки за писане на максимално преносим и недвусмислен код.

Задача 1.79.
Да се докаже, че броят на пермутациите на n-елементно множество е $n!$ (*виж 1.3.1.*)

Задача 1.80.
Да се намерят ръчно пермутациите на елементите на $\{a,b,c,d\}$, като се използва:
а) *алгоритъм 1* от *1.3.1.*
б) *алгоритъм 2* от *1.3.1.*
Да се сравнят резултатите.

Задача 1.81.
Да се обосноват алгоритъм 1 и алгоритъм 2 от *1.3.1.*

Задача 1.82.
Да се покаже, че 🖫 permswap.c действително е реализация на *алгоритъм 2* от *1.3.1.*

Задача 1.83.
Да се реализира алгоритъм за итеративно генериране на пермутации. (*виж 1.3.1.*)

Задача 1.84.
Като се използва алгоритъмът за кодиране на пермутации (*виж 1.3.1.*), да се кодират ръчно пермутациите: (2,3,1,4), (5,3,2,4,1), (3,6,4,1,5,2).

Задача 1.85.
Като се използва алгоритъмът за декодиране на пермутации (*виж 1.3.1.*), да се намерят ръчно пермутациите на 5 елемента, съответстващи на код: 3, 13, 27, 87, 119.

Задача 1.86.
Да се изпишат пермутациите на елементите на мултимножеството {1,1,2,3}. (*виж 1.3.1.*)

Задача 1.87.
Да се напише програма за намиране на всички пермутации с повторение. Може ли да се използват като основа реализацията на пермутации без повторение? Ако *да*, какви промени се налагат. Ако *не* — защо? (*виж 1.3.1.*)

Задача 1.88.
Да се напише програма за пресмятане на броя на пермутациите с повторение при зададени съответни параметри. Да се използва факторизация. (*виж 1.3.1.*)

Задача 1.89.
Да се реализират алгоритми за кодиране и декодиране на пермутации с повторение. (*виж 1.3.1.*)

Задача 1.90.
Да се докаже формулата (*) от *1.3.1*.

Задача 1.91.
Да се модифицира горната програма така, че да генерира вариации *без* повторение.

Задача 1.92.
Да се изпишат 3-елементните вариации на елементите на множеството $\{a,b,c,d,e\}$:

- с повторение
- без повторение

(*виж 1.3.2.*)

Задача 1.93.
Да се докаже, че броят на k-елементни вариации с повторение над n-елементно множество е n^k. (*виж 1.3.2.*)

Задача 1.94.
Да се докаже, че броят на k-елементни вариации без повторение над n-елементно множество е $n!/(n-k)!$ (*виж 1.3.2.*)

Задача 1.95.
Да се реализира алгоритъм за итеративно генериране на вариации с/без повторение. (*виж 1.3.2.*)

Задача 1.96.
Да се реализират алгоритми за кодиране и декодиране на вариации с и без повторение. (*виж 1.3.2.*)

Задача 1.97.
Да се реши задачата "сума нула" от *1.3.2.*, ако освен събиране и изваждане, е възможно да се поставя и знак за умножение.

Задача 1.98.

Да се предложи начин за спестяване на извършването на част от изчисленията в задачата "сума нула" от *1.3.2*. Например, кога генерирането може да бъде прекъсвано след k-та позиция, т.е. какъв критерий може да гарантира, че няма да се получи сума 0, независимо от това какви аритметични операции ще бъдат поставени на останалите $n–k$ позиции?

Задача 1.99.

Да се модифицира програмата от *1.3.3.* така, че да генерира комбинации с повторение.

Задача 1.100.

Да се изпишат 3-елементните комбинации на елементите на множеството $\{a,b,c,d,e\}$:

- с повторение
- без повторение

(*виж 1.3.3.*)

Задача 1.101.

Да се обясни връзката между броя на комбинациите без повторение на n елемента от клас k и биномните коефициенти. (*виж 1.3.3.*)

Задача 1.102.

Да се докаже, че броят на k-елементни комбинации без повторение над n-елементно множество е $n! / (k!(n–k)!)$. (*виж 1.3.3.*)

Задача 1.103.

Да се докаже, че броят на k-елементни комбинации с повторение над n-елементно множество е $(n+k–1)! / (k!(n–1)!)$. Да се приведат две доказателства: едно алгебраично и едно комбинаторно. (*виж 1.3.3.*)

Задача 1.104.

Да се реализира алгоритъм за итеративно генериране на комбинации с/без повторение. (*виж 1.3.3.*)

Задача 1.105.

Да се реализират алгоритми за кодиране и декодиране на комбинации с и без повторение. (*виж 1.3.3.*)

Задача 1.106.

Да се реализира алгоритъм за генериране на разбивания на число (като сума, произведение, сума и произведение от дадени числа) в лексикографски ред. (*виж 1.3.4.*)

Задача 1.107.

Да се реализира итеративна версия на алгоритъма за генериране на разбиванията на число (като сума, произведение, сума и произведение от дадени числа) (*виж 1.3.4.*).

Задача 1.108.

Възможно ли е алгоритмите от *1.3.4.* да се обобщи и за реални числа? Ако *не*: защо? Ако *да*: какви промени се налагат?

Задача 1.109.

Да се предложи и реализира алгоритъм за разбиване на число като *произведение* от дадени числа. (*виж 1.3.4.*)

Задача 1.110.

Да се намери ръчно петото число на Стирлинг. (*виж 1.3.5.*)

Задача 1.111.

Да се реализира рекурсивен вариант на алгоритъма за намиране на числата на Стирлинг. (*виж 1.3.5.*)

Задача 1.112.

Да изведат формулите, отразяващи зависимостите между числата в триъгълника на Стирлинг. Да се сравнят с тези от триъгълника на Паскал от *1.1.5.* (*виж 1.3.5.*)

Задача 1.113.

Да се напише програма за намиране всички разбивания на дадено множество като обединение на непресичащи се непразни негови подмножества (*виж 1.3.5.*).

Задача 1.114.

Дадени са три алгоритъма със сложности съответно $5n^2 - 7n + 13$, $3n^2 + 15n + 100$ и $1000n$. Кой от тях следва да се използва при входни данни с размер до: 100; 1000; 10000; 1000000? (*виж 1.4.*).

Задача 1.115.

Възможно ли е полиномът $5n^3 - 5n^2 + 5$ да задава сложност на алгоритъм? А $5n^3 - 5n^2 - 5$? А $-5n^3 - 5n^2 + 5$? (*виж 1.4.*)

Задача 1.116.

Да се докаже, че $\Theta(F) = O(F) \cap \Omega(F)$. (*виж 1.4.2.*)

Задача 1.117.

Да се докаже, че $o(F) = O(F) \setminus \Theta(F)$. (*виж 1.4.2.*)

Задача 1.118.

Да се докаже, че $\omega(F) = \Omega(F) \setminus \Theta(F)$. (*виж 1.4.2.*)

Задача 1.119.

Да се провери съгласно дефиницията на $O(f)$, че $5n^3 - 5n^2 + 5 \in O(n^3)$. (*виж 1.4.3.*)

Задача 1.120.

Да се докаже, че $\sqrt{n} \in O(n)$. (*виж 1.4.3.*)

Задача 1.121.

Да се докажат приведените в *1.4.3.* общи свойства на $O(F)$.

Задача 1.122.

Да се докажат приведените в *1.4.3.* асимптотични свойства на $O(F)$.

Задача 1.123.

Да се даде пример за функции $f(n)$ и $g(n)$, за които границата $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ не съществува. в (*виж 1.4.3.*)

Задача 1.124.

Да се провери съгласно дефиницията на $\Omega(F)$, че $5n^3 - 5n^2 + 5 \in \Omega(n^3)$. (*виж 1.4.4.*)

Задача 1.125.

Да се докаже, че $\sqrt{n} \in \Omega(n^{0,48})$. (*виж 1.4.4.*)

Задача 1.126.

Да се докажат приведените в *1.4.4.* общи свойства на $\Omega(F)$.

Задача 1.127.

Да се докажат приведените в *1.4.4.* асимптотични свойства на $\Omega(F)$.

Задача 1.128.

Да се провери съгласно дефиницията на $\Theta(F)$, че $5n^3 - 5n^2 + 5 \in \Theta(n^3)$. (*виж 1.4.5.*)

Задача 1.129.

Да се докаже, че $\sqrt{n} \notin \Theta(n^{0{,}48})$. (*виж 1.4.5.*)

Задача 1.130.

Да се докажат приведените в *1.4.5.* общи свойства на $\Theta(F)$.

Задача 1.131.

Да се докажат приведените в *1.4.5.* асимптотични свойства на $\Theta(F)$.

Задача 1.132.

Да се провери, че релацията R = "принадлежи към $\Theta(...)$" (задавана като $(f,g) \in R$ тогава и само тогава, когато $f \in \Theta(g)$) е релация на еквивалентност. (*виж 1.4.5.*)

Задача 1.133.

Да се посочат верните твърдения:

а) $4n^3 + 5n^2 + 5 \in \Omega(n^4)$
б) $log_2 n \notin \Omega(\sqrt{n})$
в) $4n^3 + 5n^2 + 5 \in \Omega(7n^4 - 10)$
г) $n \in \Omega(\sqrt{n})$
д) $log_2 n \in O(\sqrt{n})$
е) $10n+3 \in O(n)$
ж) $10n \in O(3n^4 - 10n^2 + 7)$
з) $5n + 1 \in O(\sqrt{n})$

(*виж 1.4.2-1.4.5.*)

Задача 1.134.

Да се посочат верните твърдения:

а) $\Theta(F) = O(F) \cap \Omega(F)$
б) $\omega(F) = \Omega(F) \setminus \Theta(F)$
в) $o(F) = O(F) \setminus \Theta(F)$
г) $O(c_1 f(n) + c_2 g(n)) = \max(O(f(n)), O(g(n)))$
д) $f \in \Omega(g) \Rightarrow g \in \Theta(f)$
е) $f \in O(g) \Rightarrow g \in \Omega(f)$
ж) $f \in \Omega(g) \Leftrightarrow g \notin O(f)$

(*виж 1.4.2-1.4.5.*)

Задача 1.135.

Да се посочат верните твърдения:

а) $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow f(n) \in O(g(n)), g(n) \in O(f(n))$

б) $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), g(n) \notin O(f(n))$

в) $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f(n) \notin O(g(n)), g(n) \in O(f(n))$

г) $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow f(n) \in \Theta(g(n))$

д) $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in \Omega(g(n)), f(n) \notin \Theta(g(n))$

(*виж 1.4.2-1.4.5.*)

Задача 1.136.

Да се даде друг пример за двойка асимптотично несравними функции. (*виж 1.4.6.*)

Задача 1.137.

Необходимо ли е едната от функциите $f(n)$ и $g(n)$ да бъде композиция на две функции, една от които да бъде периодична, за да бъдат f и g асимптотично несравними? (*виж 1.4.6.*)

Задача 1.138.

Да се докаже, че $n!$ расте по-бързо от c^n, за всяко $c > 0$. (*виж 1.4.7.*)

Задача 1.139.

Да се разположи функцията c^n, $c > 0$ в редицата на нарастване на най-често използваните функции (виж *таблица 1.4.7.*).

Задача 1.140.

Да припомним, че за конструкцията `if-then-else` имахме

$T(p) \in O(P), T(s_1) \in O(F_1)\ , T(s_2) \in O(F_2) \Rightarrow T(\texttt{if}\ (p)\ s_1;\ \texttt{else}\ s_2) \in \max(O(P), O(F_1), O(F_2))$

Да се намери израз от този тип за конструкцията `switch`. (*виж 1.4.8.*)

Задача 1.141.

Да се докаже, че сложността на програмния фрагмент от пример 1 от *1.4.8.* е $\Theta\ (n^2)$.

Задача 1.142.

Да се докаже, че сложността на програмния фрагмент от пример 3 от *1.4.8.* е $\Theta(n^2)$.

Задача 1.143.

Да се даде оценка от тип $\Theta(...)$ за сложността на програмния фрагмент от пример 5 от *1.4.8.*

Задача 1.144.

Безопасно ли е да се замени $O(...)$ с $\Theta(...)$ в цялата точка *1.4.8.*?

Задача 1.145.

Да се докаже, че за числата на Фибоначи е в сила равенството: $\left(\frac{3}{2}\right)^{n-1} \le f_n \le 2^n,\ n \ge 1$.

(*виж 1.4.8.*)

Задача 1.146.

Да се определи сложността $\Theta(...)$ на следния фрагмент:

```
for (i = 0; i < 2*n; i++)
  for (j = 0; j < 2*n; j++)
    if (i<j) for (k = 0; k < 2*n; k++) break;
```

(*виж 1.4.8.*)

Задача 1.147.

Да се определи сложността $\Theta(...)$ на следния фрагмент:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
  for (j = i+1; j < i*i; j++) sum++;
```

(*виж 1.4.8.*)

Задача 1.148.

Да се определи сложността $\Theta(...)$ на следния фрагмент:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
```

```
    if (i==j)
      for (k = 0; k < n; k++) break;
```
(*виж 1.4.8.*)

Задача 1.149.
Да се определи сложността Θ(...) на следния фрагмент:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < i*i; j++) sum++;
```
(*виж 1.4.8.*)

Задача 1.150.
Да се определи сложността Θ(...) на функцията trib(n):

```
unsigned trib(unsigned n)
{
    if (n < 3)
      return 1;
    if ((n % 2) == 1)
      return trib(n-1) + trib(n-2) + trib(n-3);
    else
      return trib(n / 3) + trib(n / 2);
}
```
(*виж 1.4.8.*)

Задача 1.151.
Да се определи сложността Θ(...) на функцията streep(n):

```
unsigned fib(unsigned n)
{
   if (n < 2)
     return 1;
   return fib(n-1) + fib(n-2);
}
void streep(unsigned n) {
  fib(fib(n));
}
```
(*виж 1.4.8.*)

Задача 1.152.
Да се определи сложността Θ(...) на следния фрагмент:

```
int n = 10;
for (int i = 0; i<n; (n=i)++);
```
(*виж 1.4.8.*)

Задача 1.153.
Да се определи сложността на алгоритъм, зададена с рекурентната зависимост:

а) $T(1) = 1$, $T(n) = 4T(n-1) - 2^n$, $n \geq 2$
б) $T(1) = 0$, $T(n) = 2T(n-1) + n + 2^n$, $n \geq 2$
в) $T(0) = 1$, $T(n) = 2T(n-1) + n$, $n \geq 1$
г) $T(1) = 1$, $T(n) = 2T(n-1) + 3^n$, $n \geq 2$
д) $T(1) = 2$, $T(2) = 3$, $T(n) = 2T(n-1) - T(n-2)$, $n \geq 3$

(*виж 1.4.9.*)

Задача 1.154.
Да се определи сложността на алгоритъм, зададена с рекурентната зависимост:

а) $T(1) = 2$, $T(n) = 4T(n/3) + n^{\log_3 4}$, $n \geq 2$.
б) $T(1) = 0$ и $T(2n+1) = T(2n) = T(n) + \log_2 n$, $n \geq 2$.
в) $T(1) = 4$ и $T(n) = 2T([\sqrt{n}\,]) + \log_2 n$, $n \geq 2$.
(*виж 1.4.10.*)

Задача 1.155.
Дадени са три алгоритъма със сложности съответно: $5n^2 - 7n + 13$, $3n^2 + 15n + 100$ и $1000n$. За всеки алгоритъм да се определи интервал от стойности n, за които той е по-бърз от другите два. (*виж 1.4.11.*)

Задача 1.156.
Основателна ли е критиката, отправена към асимптотичната нотация? (*виж 1.4.11.*)

1.5.2. Други задачи

- Задачи от числа, редици, функции

Използвайте задачите, изложени по-долу, освен за да решите конкретния математически проблем и за да експериментирате с различни схеми на реализация (итеративна, рекурсивна). Определете сложността на алгоритъма си и я използвайте, за да правите предвиждания, колко бързо би работила евентуалната реализация за различни входни данни.

Задача 1.157.
Да се напише програма, намираща всички числа, равни на сумата от факториелите на своите цифри.
Упътване: Важно е да се стесни диапазонът (да се намери горна граница), до която да се търсят такива числа. Например няма смисъл да се проверяват числа с 9 и повече цифри, тъй като 9.9! = 3265920, което очевидно е по-малко от произволно 9-цифрено число.

Задача 1.158. *Прости числа – близнаци*

Дефиниция 1.34. Две прости числа p и q се наричат *"близнаци"*, ако $p = q + 2$. Доказано е, че съществуват безброй много двойки прости числа "близнаци". Първите 4 от тях са {3,5} {5,7} {11,13} {17,19}.

Задача: Да се намерят първите n двойки прости числа близнаци.

Дефиниция 1.35. Сумата на редицата от реципрочните стойности на числата близнаци

$$S = 1/3 + 1/5 + 1/5 + 1/7 + 1/11 + 1/13 + 1/17 + 1/19 + \ldots$$

е константа, известна като *константата на Бран*, и е приблизително равна на 1,902160578.

Задача: Намерете сумата от реципрочните числа на първите n прости числа-близнаци.

Задача 1.159. *Диофантово уравнение на 2 променливи*
Дадено е уравнението $ax + by = c$, където a, b и c са дадени цели числа. Да се намерят x и y, ако е известно, че са цели числа.
Упътване: Нека с d означим най-големия общ делител на a и b. Ако d не дели c, то не съществува решение на задачата. Използвайте разширен алгоритъм на Евклид за намиране на x и y. Можете ли да намерите още решения, ако разполагате с едно решение на задачата (x_0, y_0) [*Рахнев, Гъров, Гаврилов-1995*]?

Задача 1.160. x % y
Да се реализират операциите *целочислено деление* и *остатък от целочислено деление*, като се използват само стандартните математически операции +,–,*,/ върху реални числа.

Задача 1.161. *Произведение на две прости числа*

Дадено е естествено число n. Да се намери броят на естествените числа, по-малки от n, които могат да се представят като произведение на две прости числа.

Задача 1.162. *Сума на цифрите*

За дадено естествено число n да се намери минималното естествено число m, $m > n$, което има сума от цифрите, равна на сумата от цифрите на n. Цифрите на n се задават като елементи на масив и могат да бъдат до 2000.

Задача 1.163. *Невъзможна сума*

Дадени са n цели числа. Да се намери такова измежду тях, което не може да бъде представено като сума на някои от останалите.

Задача 1.164. $n + 2 = 2m$

Да се намери най-малкото съставно число n такова, че $n + 2 = 2m$, където m е дадено нечетно число.

Задача 1.165. *Щастливи числа*

От списъка 1, 2, 3,... последователно се изключва всяко второ число. Така остават 1, 3, 5, 7,... . След това от новополучения списък се изключва всяко трето (остават 1, 3, 7, 9, 13) и т.н. Да се отпечата резултатът след k на брой стъпки.

Упътване: Използвайте модификация на *метода на решетото*.

Задача 1.166. *Взаимно прости двойки*

Дадени са n естествени числа. Да се намерят максимален брой двойки числа измежду дадените така, че числата във всяка двойка да бъдат взаимно прости.

Задача 1.167. *Полиноми на Ермит*

Да се напише функция за намиране стойността на полинома на Ермит $H_n(x)$:

$$H_0(x) = 1$$
$$H_1(x) = 2x$$
$$H_n(x) = 2x.H_{n-1}(x) - 2(n-1).H_{n-2}(x),\ n > 1$$

Задача 1.168. *Числа на "Трибоначи"*

Числата на Трибоначи се дефинират със следната рекурентна формула:

$$F(n) = F(n-1) + F(n-2) + F(n-3), \text{ като } F(1) = F(2) = F(3) = 1$$

Началото на редицата на "Трибоначи" изглежда така:

1, 1, 1, 3, 5, 9, 17, ...

Да се състави програма, която по зададено n намира n-тото число на "Трибоначи". Да се напише съответно рекурсивна и итеративна реализация. Коя от двете е по-ефективна и защо? Каква е сложността на рекурсивната реализация? Да се потърси формула за намиране на n-тото число на "Трибоначи".

Задача 1.169. *Числа на Фибоначи от ред p*

Числата на Фибоначи от ред p се дефинират като:

$$f_{i+1}^{(p)} = f_i^{(p)} + f_{i-1}^{(p)} + \ldots + f_{i-p}^{(p)}, i \geq p$$
$$f_p^{(p)} = 1$$
$$f_i^{(p)} = 0,\quad 0 \leq i \leq p$$

Така обикновените числа на Фибоначи се дефинират като числа на Фибоначи от ред 1. Да се напише програма за намиране първите k числа на Фибоначи от ред p, p и k са естествени числа.

Задача 1.170*. Сума от корени*

Да се напише програма, която по дадено цяло положително число n пресмята сумата $S = \sum_{i=1}^{n} \frac{1}{\sqrt{i}}$. Възможно ли е да бъдат спестени част от изчисленията, които извършва тривиалната реализация?

Задача 1.171*. Анти-първо число*

Анти-първо число се нарича естествено число, което има повече делители от всяко друго естествено число преди него. Първите 6 числа от този вид са:

1: (има единствен делител 1)
2: (1, 2)
4: Има 3 делителя (1, 2, 4)
6: Има 4 делителя (1, 2, 3, 6)
12: Има 6 делителя (1, 2, 3, 4, 6, 12)
24: Има 8 делителя (1, 2, 3, 4, 6, 8, 12, 24)

По дадено естествено число n да се намерят първите n такива числа. Можете ли да откриете някаква зависимост между тях?

Задача 1.172*. $4k + 3$*

Дадено е естествено число n. Да се намерят първите n прости числа от вида $4k + 3$, $k \in N$.

Задача 1.173*. Формула за n!*

Известно е, че съществува формула за намиране на сумата:

$$S = 1 + 2 + 3 + ... + n$$

Аналогично, разглеждаме произведението на първите n числа:

$$P = 1.2.3. \ldots .n = n!$$

Да се намери формула за намиране на $n!$, която не извършва всичките n умножения.

Упътване: Ако не успеете да намерите такава формула до няколко часа, по-добре се откажете: Всъщност досега никой не успял. Съществуват формули, например формулата на Стирлинг, които намират $n!$ с доста добро *приближение*:

$$n! \approx n^n e^{-n} \sqrt{2.\pi.n}$$

Ползата от тях обаче е по-скоро теоретична, отколкото практическа.

Задача 1.174*. Интервал със сума n*

За дадено естествено число n да се намери интервал $[a, b]$ такъв, че сумата от числата

$$a + (a+1) + (a+2) + \ldots + b$$

да бъде равна на n. Например, за $n = 1986$ някои интервали, удовлетворяващи условието, са:

[160,171], [495,498] и [661,663].

Задача 1.175*. $\varphi(n)$*

Нека n е естествено число. С $\varphi(n)$ означаваме броя на числата, по-малки от n, които са взаимно прости с n. Да се намери $\varphi(n)$ за дадено цяло положително число n. Какво е Вашето обяснение на факта, че за $n > 2$ $\varphi(n)$ е винаги четно число? Опитайте се да намерите точна формула за $\varphi(n)$.

Задача 1.176*. Питагорови тройки*

Дадено е естествено число n. Да се намерят всички Питагорови тройки числа a, b, c (естествени числа), $c < n$, за които $a^2 + b^2 = c^2$.

Задача 1.177. *Палиндром-степен*

Нека е дадено естествено число n. Палиндром-степен $P(n)$ на n се дефинира така:

1) Ако n е палиндром, то $P(n) = 1$.
2) Ако n не е палиндром, то $P(n) = 1 + P(s)$, където s е сумата на n с огледалния му образ, т.е. с n, записано отзад-напред.

Например, палиндром-степен на 36 е 2. За две стъпки получаваме палиндром:

$36 + 63 = 99$

Аналогично $P(48) = 3$:

$48 + 84 = 132$
$132 + 231 = 363$

Да се намери $P(n)$ за произволно n. Изпробвайте програмата си за всички естествени числа от 1 до 250. Какво намира тя за специалния случай $P(196)$?

Задача 1.178. *Питагорови четворки*

Дадено е естествено число n. Да се намерят n (n е дадено естествено число) различни четворки цели a, b, c, d, такива че:

$$a^2 + b^2 + c^2 = d^2$$

Пример: (1,2,2,3).

Задача 1.179. *Тройки естествени числа*

Дадено е естествено число n. Да се намерят n (n е дадено цяло число) различни тройки естествени числа a, b, c такива, че $\sqrt{a^2+b^2}$, $\sqrt{a^2+c^2}$, $\sqrt{b^2+c^2}$ също да бъдат естествени числа.

Задача 1.180. *Най-кратка сума от кубове*

Дадено е естествено число n. Да се намери представяне на n като сума от кубове, като се използват минимален брой събираеми. Например:

$$567 = 8^3 + 3^3 + 3^3 + 1^3 \text{ (с дължина 4),}$$

но съществува и по кратко решение:

$$567 = 7^3 + 6^3 + 2^3$$

Задача 1.181. 2^n

Да се намери минималното n, за което първите девет цифри на числото 2^n са 123454321.

Задача 1.182. *Числова пирамида*

Да се напише функция, която по подадено естествено число n извежда на екрана пирамида с височина n от вида 1 или 2 (*виж фигура 1.5.2а.*).

```
Вид 1:
                9
              8 9 8
            7 8 9 8 7
          6 7 8 9 8 7 6
        5 6 7 8 9 8 7 6 5
      4 5 6 7 8 9 8 7 6 5 4
    3 4 5 6 7 8 9 8 7 6 5 4 3
  2 3 4 5 6 7 8 9 8 7 6 5 4 3 2
1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
```

```
Вид 2:
                8 8
              7 8 8 7
            6 7 8 8 7 6
          5 6 7 8 8 7 6 5
        4 5 6 7 8 8 7 6 5 4
      3 4 5 6 7 8 8 7 6 5 4 3
    2 3 4 5 6 7 8 8 7 6 5 4 3 2
  1 2 3 4 5 6 7 8 8 7 6 5 4 3 2 1
```

Фигура 1.5.2а. Пирамида.

Задача 1.183. *Каре*

Да се напише функция, която по подадено естествено число n извежда на екрана каре с височина n (*виж фигура 1.5.2б.*).

```
                1
            1   2   1
        1   2   3   2   1
    1   2   3   4   3   2   1
1   2   3   4   5   4   3   2   1
```

```
            1   2   3   4   5   6   5   4   3   2   1
        1   2   3   4   5   6   7   6   5   4   3   2   1
    1   2   3   4   5   6   7   8   7   6   5   4   3   2   1
1   2   3   4   5   6   7   8   9   8   7   6   5   4   3   2   1
    1   2   3   4   5   6   7   8   7   6   5   4   3   2   1
        1   2   3   4   5   6   7   6   5   4   3   2   1
            1   2   3   4   5   6   5   4   3   2   1
                1   2   3   4   5   4   3   2   1
                    1   2   3   4   3   2   1
                        1   2   3   2   1
                            1   2   1
                                1
```

Фигура 1.5.2б. Каре.

- *Задачи от матрици и общи задачи*

Задача 1.184. *Магически квадрат*

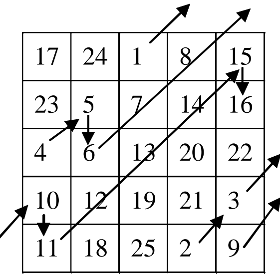
Магически квадрат от ред n е таблица с размери $n \times n$, във всяка клетка от която е записано естествено число от 1 до n^2 така, че сумата от числата, записани във всеки хоризонтал, вертикал, или главен диагонал, е една и съща и е равна на $n(n^2+1)/2$. Например, за $n = 5$ един възможен магически квадрат е даден на *фигура 1.5.2в*.

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Фигура 1.5.2в. Магически квадрат.

Упътване: За нечетни n ($n \geq 3$) един възможен алгоритъм за попълване е следният:

1) Започваме от средата на първия ред и там записваме числото 1.
2) На всяка стъпка се придвижваме по диагонала нагоре и надясно и записваме следващото число. Ако "излезем" извън таблицата отгоре, се пренасяме в последния ѝ ред, а ако "излезем" отдясно, се пренасяме в първия ѝ стълб. Ако при придвижването се озовем в клетка, която вече е попълнена, то "се спускаме" в клетката под нея. *Фигура 1.5.2г.* показва как протича част от попълването, започвайки от 1:



Фигура 1.5.2г. Построяване на магически квадрат.

За четни n също съществуват алгоритми за получаване на магически квадрат, но те са малко по сложни [*Magic-1*].

Задача 1.185. *Шахматни дами*

Да се определи максималният брой шахматни дами, които могат да се поставят върху шахматна дъска с размер $m \times n$ (n, m — дадени естествени числа) така, че никои две да не са под удар. Да се реши същата задача, ако фигурата, която трябва да се постави, е: цар; топ; кон; офицер.

Задача 1.186. *Четирибуквени думи*

Дадени са две четирибуквени думи A_0 и A_n и речник с четирибуквени думи A_i, за $i = 1, 2, \ldots, n-1$. Възможно ли е да бъде получена думата A_0 от думата A_n чрез последователност от думи от речника:

$$A_0 - A_1 - A_2 - \ldots - A_n,$$

като всеки две последователни думи A_i и A_{i+1}, за $i = 0, 1, \ldots, n-1$, имат точно една различна буква. Проверете своето решение с думите "муха" и "слон" и подходящ речник.

Задача 1.187. *Транспонирана матрица*

Дадена е целочислена матрица $A_{n\times n}$ с елементи a_{ij}. Да се намери транспонираната ѝ матрица $A' = \{a'_{ij}\}$ на дадената, $a'_{ij} = a_{ji}$, за $1 \le i, j \le n$.

- Комбинаторни задачи

В някои от задачите, които следват, се иска да се генерират обекти с определени свойства, например (*виж задача 1.43.*) валидни номера на *ЕГН*. В такива задачи е за предпочитане не да се генерира по-обща комбинаторна структура (вариации с повторение за избрания пример) и после да се изключват невалидните обекти, а да се състави схема, която генерира *само валидни* обекти.

Задача 1.188. *Тот-2*

Да се напише програма, симулираща тираж на *Тото-2* 6/49. Колко различни комбинации трябва да попълним, за да бъдем сигурни, че ще спечелим Джакпота?

Задача 1.189. *Телефонни номера*

През 1930 г. изследователска група от лабораториите *Bell* е възложена задачата да предложат как да бъдат проектирани телефонните номера на Съединените щати. Било поставено изискването да бъдат валидни поне до края на XX-и век. След приблизителна оценка колко телефонни поста ще има тогава, изследователите се спират на следната схема за телефонен номер:

$(a_1a_2a_3)–b_1b_2b_3–c_1c_2c_3c_4$, където
$a_1 \in \{2,3,...,9\}$
$a_2 \in \{0,1\}$
$a_3 \in \{0,1,...,9\}$
$b_1, b_2 \in \{2,3,...,9\}$
$b_3, c_1, c_2, c_3, c_4 \in \{0,1,...,9\}$

Да се намери броят на различните телефонни номера, които могат да се получат по горната схема.

Задача 1.190. *Латински квадрат*

Латински квадрат от ред n е таблица с размери $n \times n$ такава, че:

- във всяка клетка от таблицата има едно естествено число между 1 и n.
- числата от всеки ред и всеки стълб са пермутация на числата от 1 до n.

Например, за $n = 4$ квадратът от *фигура 1.5.2д.* е латински.

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

Фигура 1.5.2д. Латински квадрат.

Да се генерира латински квадрат по зададено естествено число n.

Задача 1.191. *Константно произведение*

Дадени са $2n$ естествени числа. Известно е, че има точно m двойки числа (a_i, b_i) сред тях такива, че произведението $a_i.b_i$ е равно на едно и също число (за $i = 1,2,...,m$). Да се намерят и отпечатат всичките m такива двойки числа.

Задача 1.192. *Вечеря у принцеса Анна*

Принцеса Анна планира да покани 15 свои приятелки на вечеря. В продължение на 35 дни те ще ѝ гостуват точно по 3 на ден. Възможно ли е така да бъдат избрани тройките, че през всеки от 35-те дни Анна да вечеря с различна тройка свои гости? Генерирайте едно възможно "разписание".

Задача 1.193. *Пресечни точки на диагоналите в правилен n-гълник*

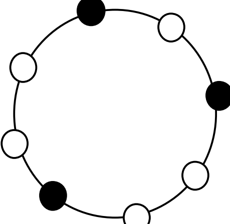
Дадено е естествено число n, $n > 2$. Да се намери броят на пресечните точки на диагоналите в правилен n-гълник. Например, в триъгълник няма диагонали и този брой е нула. Квадратът има два диагонала, и съответно една пресечна точка, а за правилен петоъгълник този брой е 5.

Задача 1.194. *Топки в кутии*

Дадени са $2t + 1$ топки и трябва да се разпределят в три кутии така, че сумата от броя на топките в произволни две кутии да бъде по-голяма от броя на топките в третата кутия. Да се напише програма, която генерира всички възможни разпределения на топките така, че да бъде изпълнено горното условие. Да се намери формула за броя на тези разпределения.

Задача 1.195. *Огърлица*

Огърлица се състои от бели и черни мъниста така, че няма две съседни черни мъниста (*виж фигура 1.5.2е.*).



Фигура 1.5.2е. Мъниста.

Ако n е броят на черните мъниста, а k — на белите, какъв е броят на различните огърлици, които могат да бъдат образувани от мънистата? Напишете програма за генериране на t различни огърлици, по дадено естествено число t.

Задача 1.196. *Бридж*

Да се напише програма, която раздава карти за бридж: 4 играча по 13 карти за всеки. Абстрахирайки се от реалността, напишете програмата си по такъв начин, че при t последователни

раздавания всеки път да се генерира уникално раздаване — никой играч да не получава два пъти един и същи карти.

Задача 1.197. *Квадрати в мрежа*

Дадена е квадратна мрежа с размери $n \times n$. Какъв е броят на квадратите в мрежата като функция на n: т.е. сумата от броя на различните квадрати с големина 1×1, 2×2, ..., $n \times n$?

Задача 1.198. *Променливи в Си*

Какъв е броят на валидните имена на променливи в Си с дължина по-малка от k символа, k е естествено число?

Задача 1.199. *n-цифрени числа*

Да се намери формула за броя на n-цифрените числа ($n < 11$), които се делят на k, ако не се допуска повторение на цифри. Да се предложи алгоритъм за лексикографски наредено генериране на всички такива числа.

Задача 1.200. *Разстояние между шахматни фигури*

Да се намери начин за поставяне на 9 фигури върху шахматна дъска с размери 8×8 така, че разстоянието между всеки две фигури да бъде различно, или да се покаже, че това не е възможно. Под "разстояние между две фигури" се има предвид броят на полетата по най-краткия път между тях (без диагонални преходи).

Задача 1.201. *MJ-нощни клубове*

Дадени са n съпружески двойки. Колко са възможните начини за разпределяне на хората в групи по k (k-четно число, k дели n) така, че във всяка група да има точно $k/2$ жени и $k/2$ мъже, като никой да не бъде в една група със собствения си съпруг/съпруга?

Задача 1.202. *Млад хакер*

Да се напише програма, която генерира валидни номера за кредитни карти (от даден тип кредитна карта — *VISA* или *MasterCard*). Да се използва следното описание за валидни номера на кредитни карти:

Номерата на кредитните карти са 16-цифрени в групи по четири. Да проверим дали следната кредитна карта е валидна:

4 2 0 4 – 5 8 7 6 – 9 0 1 2 – 5 2 3 4

1) Обръщаме цифрите на номера и получаваме:
   4 3 2 5 2 1 0 9 6 7 8 5 4 0 2 4
2) Удвояваме цифрите, стоящи на четна позиция, и получаваме:
   4 6 2 10 2 2 0 18 6 14 8 10 4 0 2 8
3) Ако получим събираеми, по-големи от 9, ги разделяме на две отделни цифри. Събираме така получените цифри:
   4 + 6 + 2 + 1 + 0 + 2 + 2 + 0 + 1 + 8 + 6 + 1 + 4 + 8 + 1 + 0 + 4 + 0 + 2 + 8 = 60
4) Ако сборът се дели на 10 без остатък, то кредитната карта е валидна. Ако номерът започва с 4, то типът ѝ е *VISA*, а ако има префикс 51–55, тя е *MasterCard*.

Задача 1.203. *ЕГН*

Да се напише програма за генериране на валидни ЕГН номера на жени.

Упътване: Проверката за валидност на ЕГН се извършва по следната схема: на всяка цифра от първите девет се съпоставят следните тегла: 2, 4, 8, 5, 10, 9, 7, 3, 6. Всяка от цифрите на ЕГН се умножава по съответното тегло и получените произведения се сумират. Остатъкът при целочислено деление на сумата с 11 е контролно число и трябва да съвпада с десетата цифра от ЕГН. При остатък 0 или 10 контролната цифра трябва да бъде 0.

В допълнение, тъй като първите 6 цифри представляват рождената дата, те трябва да бъдат валидна дата от годината. Например, ЕГН, започващо с 810229, е невалидно, тъй като 1981 година не е високосна.

Полът се определя от деветата цифра: ако е четно число, то ЕГН е на мъж.

Задача 1.204. *Палиндроми*

Какъв е броят на симетричните думи (палиндроми) с дължина n символа, съставени от малки латински букви? Да се напише програма за генериране на всички палиндроми с дължина n.

Глава 2
Въведение в структурите от данни

"A program without a loop and a structured variable isn't worth writing."

~ Epigrams in programming

Най-важното за ефективното решаване на всяка една задача е изборът на подходящ алгоритъм. От този избор зависи дали задачата ще бъде решена точно или приближено, частично или пълно, какво изчислително време и ресурси ще са необходими за решаването ѝ и т.н. От своя страна, ефективността на всеки конкретен алгоритъм зависи допълнително от начина, по който ще бъде реализиран. Неговата сложност е тясно свързана с начина, по който ще се представят и организират обектите в задачата, т. е. с избора на подходящите *структури от данни*. За да разберем защо е така, достатъчно е да се замислим по-дълбоко в какво се състои работата на една компютърна програма — освен за избрания алгоритъм, голяма част от изчислителното време се отделя за процеси, свързани с достъпа, съхранението и обработката на данните.

Един от най-простите примери за структури от данни са *променливите* от прост тип (цели/реални числа, символи). Те се поддържат от почти всички езици за програмиране и се използват в почти всяка програма. Тяхната структура е постоянна — могат да приемат само стойности, допустими за типа им, и заемат константно количество памет (т. е. не е възможно да се променя в процеса на работа на програмата).

Друга основна структура от данни е *масивът*: фиксиран брой променливи от един и същ тип, съхранени последователно в паметта на компютъра. Достъпът до всяка една от тези променливи (т. е. до елементите на масива) се извършва посредством комбинация от името на масива и индекса на съответната променлива. Основно предимство при работата с масиви е, че достъпът до k-тия елемент е директен (все пак, за разлика от простите променливи, е налице косвена адресация: пресмята се адрес на началото плюс отместване). От друга страна, размерът на един масив е фиксиран — това означава, че първо, паметта, заделена при декларирането на масива, остава *заета* през цялото време на работа на програмата, и второ, ако големината на масива се окаже недостатъчна за нуждите на програмата, увеличаването може да стане единствено чрез копиране на елементите в голям масив.

Променливите от прост тип и масивите се определят като *статични структури* от данни. Повечето езици за програмиране предоставят възможност и за *динамично* заделяне на памет. При него паметта се резервира тогава, когато е необходима, и се освобождава, когато престане да бъде нужна. Резервирането и освобождаването става в процеса на работа на програмата, което води до гъвкаво и ефективно използване на паметта, както и до възможност за моделиране на по-сложни структури от данни, които имат собствена логика и свой вътрешен алгоритмичен "живот".

Изобщо, представянето и моделирането на реални обекти в компютърна програма, както и операциите, които се извършват над тях, могат условно да се разделят на две подзадачи:

1) *Дефиниране на абстрактни структури от данни* (*АСД*): това е начинът, по който реалните обекти ще бъдат моделирани като математически обекти, както и определяне на множеството от допустимите операции над тях.

2) *Реализация на абстрактните структури от данни*: Това е начинът, по който дефинираните математически обекти ще бъдат представени в паметта на компютъра (чрез прости типове, или като комбинация от налични реализации на други АСД), както и начинът, по който ще се реализират операциите с тях.

С други думи, целта на дефинирането на абстрактни структури от данни е да се определи, *какво* може да се прави с обектите, а на реализацията — *как* да се прави.

Ще илюстрираме казаното дотук с конкретен пример. Нека е дадена колода от 52 карти. За представянето ѝ можем да използваме абстрактна структура от данни — *множество*, чиито елементи са картите. Разглеждаме следните операции:

- *изключване* на произволен елемент от множеството (изваждане на произволна карта от колодата).
- *включване* на елемент в множеството (добавяне на карта).
- *проверка* дали елемент принадлежи на множеството (проверка дали дадена карта се намира в колодата).

Реализацията на това множество също не е еднозначна. Например всеки негов елемент (карта) можем да представим, като използваме два символа — за вида на картата ('2', ..., '9', '*T*', '*J*', '*Q*', '*K*', '*A*') и за боята ('*S*', '*C*', '*D*', '*H*'). Възможно е и друго представяне — на всяка карта да съпоставим уникално естествено число между 0 и 51 (броят на различните карти е 52). Освен това трябва да изберем представяне на множеството и да реализираме позволените операции над него. Ще покажем една възможна реализация (приемаме, че картите са означени с целите числа от 0 до 51):

Ще използваме масив `int cards[52]`. Стойността на `cards[i]` ще бъде 1, ако картата с номер i е в колодата. В противен случай стойността на `cards[i]` ще бъде 0. Така, за да добавим карта с номер k, е достатъчно да извършим присвояването `cards[k] = 1;`. Операцията изключване на произволен елемент се осъществява по следния начин: намираме някое k, за което `cards[k] == 1` и присвояваме `cards[k] = 0`. Проверката дали карта с номер k е в колодата, се състои от единствен тест на стойността `cards[k]`.

Въпросът за реализацията е от първостепенно значение при решаване на конкретната задача. В примера с колодата карти реализация на множество чрез използване на масив `cards[52]` е удачна, тъй като картите са само 52. Възможно е обаче елементите на множеството да бъдат много повече. Например, нека номерираме студентите в един университет — на всеки студент се съпоставя уникално естествено число. Ако за да представим множеството от студентите в една административна група (приемаме, че студентите в университета са не-повече от 30000) използваме реализация, подобна на тази в примера с картите, ще има голямо разхищение на памет: цял масив `int students[30000]`, докато студентите в една група реално не са повече от 30. Можем да подходим по друг начин: да въведем целочислена променлива `unsigned n`, означаваща броя на студентите в групата, и масив `unsigned students[n]` такъв, че на i-та позиция ($0 \le i < n$) да бъде записан номер на студент.

Изобщо, от избора на реализация зависи не само количеството памет, което ще се използва, а и изчислителната сложност на всяка една операция. В тази глава ще се запознаем с основните абстрактни структури от данни, както и с два важни подхода за тяхната реализация — *последователен* (*статичен*, какъвто приложихме при изброените по-горе примери) и *свързан* (*динамичен*).

2.1. Списък, стек, опашка

Дефиниция 2.1. *Линеен списък* (или само *списък*) се нарича последователност от n ($n \ge 0$) елемента $x_1, x_2, ..., x_n$, подредени последователно (линейно). При $n = 0$ списъкът се нарича *празен*. Ако $n > 0$, то x_1 се нарича *първи* елемент в списъка, а x_n — *последен*.

Единствената връзка между елементите в тази структура се определят от условията: За всяко i ($1 \le i \le n$) i-тият елемент x_i е предшестван от елемента x_{i-1} и е следван от елемента x_{i+1}.

Всъщност думата "линеен" е излишна: нелинейните списъци имат специални имена — графи, дървета и др., и не се разглеждат като списъци. Ние я използваме с цел придържане към общоприетата, но за съжаление погрешна, терминология.

На практика под "елемент" в списък разбираме произволна структура от данни (*полета*). Най-често елементите на списъка са прости типове данни и имат идентична структура.

Да означим типа на данните, които съдържа всеки елемент, с `data` и да разгледаме следния пример: Искаме да построим списък на студентите в един университет. Всеки елемент от списъка може да съдържа следната информация:

```
struct data {
  short age;       /* възраст на студента (по-малка от 128 години ;) */
  char sex;        /* пол на студента — 'm' за мъж и 'f' за жена */
  unsigned fn;     /* факултетен номер на студента */
  char *name;      /* име на студента */
};
```

Да разгледаме основните операции, които могат да се извършват върху линеен списък:

- Получаване на достъп до k-тия елемент в списък (и евентуална *промяна* на стойността на някое от полетата му).
- *Включване* (*вмъкване*) на елемент в списъка (преди или след даден елемент, както и в празен списък)
- *Изключване* (*изтриване*) на k-тия елемент от списък.
- *Намиране* на всички елементи в списъка, съдържащи дадена стойност (в зададено поле или набор от полета).

Могат да се дефинират още много операции — сортиране, сливане, обръщане на списъци и т.н. Тези операции са повече алгоритмични и за тях ще стане дума по-нататък в книгата, като в тази глава ще се ограничим с разглеждането само на основните принципи при дефинирането и реализацията на структурите от данни.

Изобщо, операцията *търсене* може да се извършва по различен критерий: например, в дефинирания по-горе списък от студенти, може да се интересуваме от всички *студентки* на възраст до 22 години, или пък от всички студенти, чийто факултетен номер е просто число.

Оттук нататък, когато говорим за елементи на някоя структура, за удобство ще приемем, че наред с данните, които съдържа, всеки елемент ще има задължително едно поле, което се нарича *ключ* на елемента. Търсенето на елемент в списъка ще се извършва *само* по този ключ.

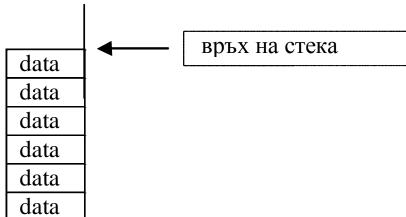
Операциите, дефинирани по-горе, не могат да се реализират така, че всичките едновременно да бъдат ефективни (например всички да имат константна сложност). Освен това случаите $k = 1$ и $k = n$ са по-специални — възможно е достъпът до тях да се извършва с по-малка сложност, отколкото достъпът до останалите елементи в списъка. Нещо повече, често се налага в списъка да се извършват операции само върху първия и/или последния елемент, което води до дефиниране на по-специфични структури от данни:

Задача за упражнение:

Защо не е възможна реализация, при която и четирите основни операции да бъдат еднакво ефективни, например да имат константна сложност?

- Стек

Стекът е линеен списък, в който всички операции (включване, изключване и т.н.) се извършват *само* в единия му край, т. е. само върху първия или само върху последния елемент. Прието е елементът, върху който се извършват операциите, да се нарича *връх* на стека (*виж фигура 2.1а.*).



Фигура 2.1а. Стек.

Класическите абстрактни операции върху стек *S* са следните:

```
void init(S);          /* инициализира празен стек */
void push(S, data x); /* добавя нов елемент на върха на стека */
data pop(S);           /* извлича елемента от върха на стека */
int isEmpty(S);        /* проверява дали стек е празен */
```

Често се реализират и допълнителни операции върху стек: извличане на елемент от върха на стек, без изключването му; добавяне/изключване на *k* елемента и др.

Могат да бъдат посочени много примери за стек. Думата стек (от англ. *stack*) се превежда на български като *купчина*. Аналогията е очевидна — ако разгледаме купчина от вестници (или книги) в кашон, лесно можем да вземем вестника от върха на купчината или да добавим нов най-отгоре. Казва се още, че стекът се подчинява на дисциплината *LIFO* (*Last-In-First-Out* — *последен влязъл*, *първи излязъл*).

- *Опашка*

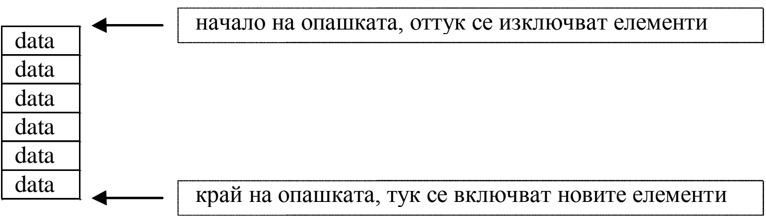
Опашката е линеен списък, в който операцията включване може да се извършва само в единия край на списъка, а операцията изключване — в другия (*фигура 2.1б.*).

Класическите операции върху опашка *Q* са следните:

```
void init(Q);          /* инициализира празна опашка */
void put(Q, data x);  /* добавя елемент (в края на опашката) */
data get(Q);           /* извлича първия елемент на опашката (връща първия
                          елемент, след което го изтрива) */
int isEmpty(Q);        /* проверява дали опашка е празна */
```

Така елементите на опашката се подчиняват на дисциплината *FIFO* (*First-In-First-Out* — *първи влязъл*, *първи излязъл*) — обратно на стека. И тук имаме аналогия с използваното в практиката значение на думата *опашка* — например, на опашка за билети (освен, ако не сме достатъчно нахални) се нареждаме най-отзад и си чакаме реда, а когато стигнем най-отпред, биваме обслужвани и напускаме опашката.

Съществуват някои интересни и важни разновидности на опашка — например приоритетна опашка (*виж задача 2.6.*) и др..



Фигура 2.1б. Опашка.

Стекът и опашката имат редица приложения. В почти всички езици за програмиране, неявно за програмиста част от паметта се използва за програмен стек, изчисляването на изрази се извършва лесно чрез стек и т.н. Инструкциите, които чакат да се изпълнят от процесора, образуват опашка; опашка има при документите подадени на принтера за отпечатване; системи с единствен процесор, при които се симулира паралелно изпълнение на задачи, използват опашка и др.

- Дек

Съществува още една, по-рядко използвана, разновидност на списък, която се явява обобщение на стек и опашка едновременно — това е *дек* (*DEQue*, съкращение от англ. *Double-Ended-Queue* — опашка с два края). Както подсказва името ѝ, при тази структура елементи могат да се включват и изключват в *двата* ѝ края. От своя страна декът също има някои разновидности — възможно е включването да бъде позволено само от едната страна, а изключването — от двете (*дек с ограничен вход*) или обратно: включване от двете страни и изключване — от едната (*дек с ограничен изход*). Като цяло дековете не намират широко приложение, въпреки че в някои специални случаи (например при многопроцесорните системи) е удобно да се използва дек за различни задачи (пресмятане стойността на изрази и др.) [*Шишков-1995*].

В следващите две точки ще се спрем на реализацията на трите основни абстрактни структури от данни — списък, стек и опашка.

2.1.1 Последователна (статична) реализация

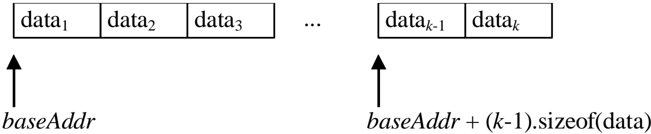
Най-простият начин за реализация на списък е елементите му да се съхраняват *последователно* в паметта на компютъра (*фигура 2.1.1а.*). В този случай адресът на k-тия елемент $addr(k)$ се определя по формулата

$$addr(k) = addr(k-1) + sizeof(data),$$

където $sizeof(data)$ е паметта, необходима за съхранение на един елемент от списъка. При този вид реализация достъпът до k-тия елемент е *пряк*. Тъй като адресът на началния елемент е известен (това е базовият адрес *baseAddr*), по формула може да се определи адресът на k-тия елемент, независимо от адреса на предхождащия го:

$$addr(k) = baseAddr + (k-1).sizeof(data)$$

Удобството от директния достъп до елементите обаче се заплаща с неефективно включване и изключване: за да включим елемент след k-тия, трябва да изместим всички елементи от $k+1$ до n с една позиция надясно. Аналогично, за да изключим k-тия елемент, трябва да изместим всички елементи от $k+1$ до n с една позиция наляво. Търсенето по ключ също е бавна операция: извършват се средно $n/2$ сравнения (същото е валидно и за вмъкването и изтриването), т. е. средната сложност отново е $\Theta(n)$.



Фигура 2.1.1а. Линеен списък при статично заделяне на памет.

- Стек

Ще реализираме структурите стек и опашка статично чрез масив. В случая със стек ще въведем единствен индекс, който ще сочи адреса на върха на стека. Така при операцията включване

елементът ще се записва на адреса, сочен от индекса, след което индексът ще се увеличава с единица. Съответно, при изключване първо индексът ще се намалява с единица, след което ще се извлича елементът. Ето как изглежда реализацията на Си, при използване на масив `stack[]` от тип `data` и променлива `top`, сочеща върха на стека:

```
#include <stdio.h>

typedef int data;
data stack[10];
int top;

void init(void) { top = 0; }
void push(data i) { stack[top++] = i; }
data pop(void) { return stack[--top]; }
int isEmpty(void) { return(0 == top); }

int main(void) {
  /* ... */
  return 0;
}
```
🖫 stack1.c

Недостатъците на горната реализация са няколко. В началото декларираме масив с 10 елемента. Ако в стека вече има 10 елемента (индексът `top` съответно е равен на 10) и искаме да включим нов елемент, ще настъпи препълване, а програмата няма да го регистрира. Това записване по "чужда" памет може да доведе до неочаквани (и често катастрофални) резултати. Аналогично, при опит за изключване на елемент от празен стек, индексът `top` ще стане отрицателно число и функцията `pop()` ще върне произволна стойност (всъщност ще върне стойността на адреса в паметта, намиращ се непосредствено преди първия елемент на масива `stack`). Освен това, известно неудобство създава константата 10, използвана в програмата. Ако решим да променим максималния брой елементи на 20, например, ще трябва да прегледаме целия код на програмата, което от една страна изисква време, а от друга — е несигурна операция, тъй като лесно можем да пропуснем нещо (например евентуално използване на числото 9, дошло от 10–1 и др.).

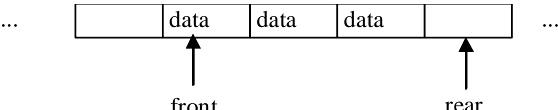
За да се справим с тези недостатъци ще направим следните модификации — ще дефинираме макрос `MAX`, който задава максималния брой елементи в стека. При включване на нов елемент ще се проверява дали `top` не е станало по-голямо от `MAX` и ако е така, значи има препълване. В програмата по-долу е направена още една промяна — преди всеки опит за изключване на елемент се извършва проверка дали стекът не е празен.

```
#include <stdio.h>
#define MAX 10

typedef int data;
data stack[MAX];
int top;

void init(void) { top = 0; }

void push(data i)
{ if (MAX == top)
    fprintf(stderr, "Препълване на стека! \n");
  else
    stack[top++] = i;
}

data pop(void)
```

```
{ if (0 == top) {
    fprintf(stderr, "Стекът е празен! \n");
    return 0;
  }
  else
    return stack[--top];
}

int isEmpty(void) { return (0 == top); }

int main(void) {
  data p;
  init();

  /* Четат се цели числа и се включват в стека до постъпване на 0 */
  scanf("%d", &p);
  while (0 != p) {
    push(p);
    scanf("%d", &p);
  };

  /* Изключват се последователно всички елементи от стека и се печатат.
   * Това ще доведе до отпечатване на първоначално въведената
   * последователност в обратен ред
   */
  while (!isEmpty()) printf("%d ", pop());
  printf("\n");
  return 0;
}
```
🖫 stack2.c

- *Опашка*

Реализацията на опашка е малко по-сложна. За последователното резервиране на памет отново ще използваме масив, но индексите ще бъдат два: `front`, който сочи началото на опашката, и `rear`, сочещ позицията след края ѝ (*фигура 2.1.1б.*).



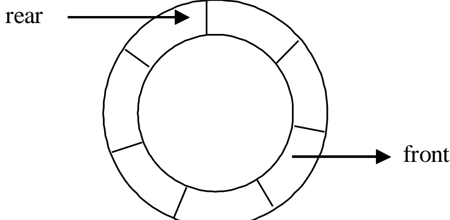
Фигура 2.1.1б. Статична реализация на опашка.

Включването и изключването на елементи се осъществява по следния начин:

- *Включване на елемент i*: Записваме *i* на позицията, сочена от индекса `rear`, и го увеличаваме с едно.
- *Изключване на елемент*: Връщаме елемента на позиция `front`, след което увеличаваме индекса `front` с едно.

При реализацията на включването и изключването трябва да се справим с няколко проблема. Основният е свързан с това, че опашката "се движи" в паметта: Ако многократно включваме и изключваме елементи, то индексите `front` и `rear` ще се увеличават непрекъснато и бързо ще надхвърлят първоначално заделената за опашката памет. Така, ако използваме масив `queue[MAX]` (с фиксиран брой елементи `MAX`), размерът му бързо ще се окаже недостатъчен, въпреки че на практика опашката ще съдържа по-малко от `MAX` елемента. Затова, ако някой от

двата индекса `front` или `rear` стане равен на `MAX`, ще му се присвоява стойност 0. Така постигаме "цикличност" на масива `queue[]` (*виж фигура 2.1.1в.*).



Фигура 2.1.1в. Опашка, представена чрез “цикличен” масив.

В началото индексите `front` и `rear` сочат нулевия елемент. По натататък, ако в даден момент те отново се окажат равни (сочат една и съща клетка), това означава едно от двете:

- Ако равенството се е получило след *изключване* на елемент (индексът `front` е застигнал `rear`), то опашката е останала *празна* след извършване на операцията.
- Ако равенството се е получило след *включване* на елемент (`rear` е застигнал `front`), то опашката е препълнена — съдържа `MAX` елемента и повече не могат да се добавят.

За статуса на опашката (дали тя е пълна или празна) ще използваме допълнителна променлива `empty`, равна на 1, когато опашката е празна, и на 0 — в противен случай. Променливата `empty` се инициализира с 1 (опашката в началото е празна) и тя приема стойност 0 веднага след първото включване на елемент. По-нататък тази променлива се модифицира единствено в двата по-особени случая, изброени по-горе. Следва пълната реализация на Си:

```
#include <stdio.h>
#define MAX 10

typedef int data;
data queue[MAX];
int front, rear, empty;

void init(void) { front = rear = 0; empty = 1; }

void put(data i)
{ if (front == rear && !empty) { /* Проверка за препълване */
    /* препълване - индексите са равни, а опашката не е празна */
    fprintf(stderr, "Препълване! \n");
    return;
  }
  queue[rear++] = i;
  if (rear >= MAX) rear = 0;
  empty = 0;
}

data get(void)
{ data x;
  if (empty) { /* Проверка за празна опашка */
    fprintf(stderr, "Опашката е празна! \n");
    return 0;
  }
  x = queue[front++];
  if (front >= MAX) front = 0;
  if (front == rear) empty = 1;
  return x;
```

```
}

int main(void) {
  data p;
  int i;
  init();
  for (i = 0; i < 2 * MAX; i++) {
    put(i);
    p = get();
    printf("%d ", p);
  }

  printf("\n");

  /* Ще причини препълване при последното включване */
  for (i = 0; i < MAX + 1; i++) put(i);

  /* Ще причини грешка при последното изключване:опашката е празна */
  for (i = 0; i < MAX + 1; i++) get();
  return 0;
}
```

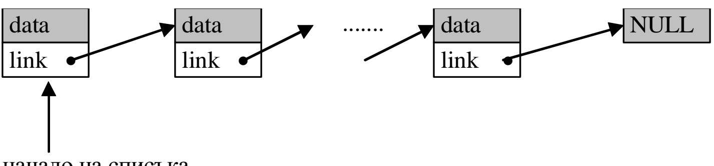
🖫 queue.c

Задачи за упражнение:

1. Да се преработят горните програми така, че функциите да не печатат съобщения за грешка, а да връщат булева стойност дали са успели.
2. Да се реализира “полустатична” версия на горните програми: ако стекът/опашката се напълни, да се извършва автоматично разширяване на масива.
3. Да се реализират основните функции за работа с дек.

2.1.2 Свързана (динамична) реализация

При динамичните структури от данни не е задължително елементите да се намират в последователни адреси от паметта. В общия случай разполагаме единствено с указател към началото (първия елемент) на списъка, а адресите на всички останали елементи не са достъпни директно. Как тогава се осъществява достъпът до тях? Към всеки елемент се добавя още едно поле (освен полетата, съдържащи данните): *указател към следващ елемент* (*фигура 2.1.2а.*).



Фигура 2.1.2а. Динамичен свързан списък.

Този вид линеен списък се нарича *линеен едносвързан списък*: едносвързан, защото имаме указател единствено към следващ елемент.

За да достигнем до k-тия елемент от списъка, трябва да преминем последователно през всичките k–1 предшестващи го елемента. Структурата на елементите е следната:

```
typedef int data;
typedef int keyType;

struct list {
```

```
  keyType key;
  data info;
  struct list *next;
};
```

Всеки елемент има идентификатор: *ключ* (полето `keyType key`). Полето `info` съдържа допълнителните данни за всеки елемент. Указателят към следващия елемент е указател към същата структура `struct list *next;` — езикът Си позволява такова рекурсивно дефиниране на структури (но само ако използваме *указател* към структура; статична декларация от вида `struct list next;` в тялото на `list` е недопустима). Последният елемент на динамичния свързан списък има стойност `NULL`: присвояване на такава стойност на указателя `*next` означава, че няма повече елементи в списъка.

Съществуват и други възможни реализации на линеен списък [*Наков-1998*][*Уирт-1980*]. Така например, при *двусвързан списък* за всеки елемент се пазят по два указателя: към предшестващия го и към следващия го елементи. Ако пък в едносвързан списък има допълнителен указател от последния към първия елемент, то списъкът се нарича *цикличен* и др.

За да използваме свързан списък, декларираме:

```
struct list *L;
```

и инициализираме `L = NULL`. Първата операция, която ще разгледаме, е включване на елемент в началото (преди първия елемент) на списък.

- Включване на елемент

Извършват се последователно три прости операции (*виж фигура 2.1.2б.*):

1) Заделя се памет за нов елемент от тип `struct list`:

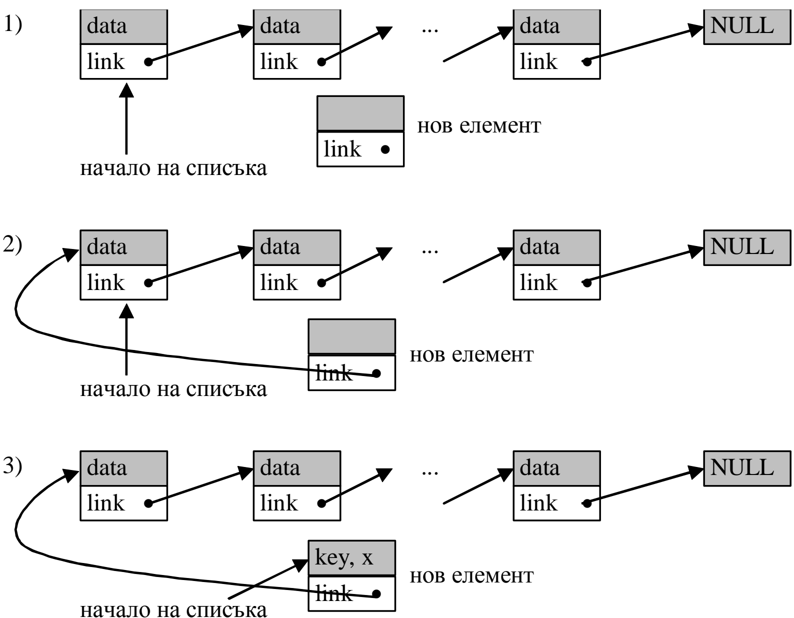
```
struct list *temp;
temp = (struct list *) malloc(sizeof(*temp));
```

2) Новият елемент `temp` сочи началото на списъка:

```
temp->next = L;
```

3) Началото на списъка се установява в новия елемент и му се присвояват зададените ключ и поле с данни:

```
L = temp;
L->key = key;
L->info = x;
```



Фигура 2.1.2б. Включване в началото: създаване на празен елемент и преместване на началото

- Обхождане на списък

С помощта на тази операция е възможно да се построи цял списък. Той може да се обходи по следния начин: започваме от указателя към началото на списъка `L`, като ще отпечатваме ключовете на елементите при обхождането:

```
while (L != NULL) {
  printf("%d ", L->key);
  L = L->next; /* преминаваме към следващия елемент */
}
```

Подобно обхождане се използва и когато се търси елемент по зададен ключ `key`:

```
struct list* search(struct list *L, keyType key)
{ while (L != NULL) {
    if (L->key == key) return L;
    L = L->next;
  }
  return NULL;
}
```

Характерно при включването в началото на списък е, че редът на елементите в него е обратен на реда на постъпването им. За да можем да работим пълноценно със списъци, ще ни бъдат необходими още три операции, всяка от които ще разгледаме поотделно:

- Включване след елемент, сочен от даден указател

Функцията за включване дефинираме като `void insertAfter(struct list **L, keyType key, data x)`. Двойният указател `L` сочи елемент от списъка (в частност това може да бъде първият елемент), след който ще се вмъкне елемент с ключ `key` и поле с данни `data`. Алгоритъмът за вмъкване е следният:

1) Създава се празен елемент и му се присвояват даденият ключ `key` и стойността `x`:

```
struct list *temp;
temp = (struct list *) malloc(sizeof(*temp));
temp->key = key;
temp->info = x;
```

2) Пренасочват се указателите: Новият елемент ще сочи към този след `L`, а `L` ще сочи новия елемент (*фигура 2.1.2в.*):

```
temp->next = (*L)->next;
(*L)->next = temp;
```


Фигура 2.1.2в. Включване след елемент, сочен от даден указател.

- *Включване пред елемент, сочен от даден указател*

Функцията за включване е `void insertBefore(struct list **L, keyType key, data x)`. Включването преди даден елемент, сочен от `L`, се усложнява от факта, че нямаме пряк достъп до предшестващия го елемент (за да променим указателя му `next`). За решаване на проблема се прилага следният трик: новият елемент се вмъква *след* `L` по-начина, описан по-горе, след което съответните стойности на полетата му се разменят с тези на `L`.

```
struct list *temp;
temp = (struct list *) malloc(sizeof(*temp));
*temp = **L;
(*L)->next = temp;
(*L)->key = key;
(*L)->info = x;
```

- *Изтриване по зададен ключ и указател към начало на списъка*

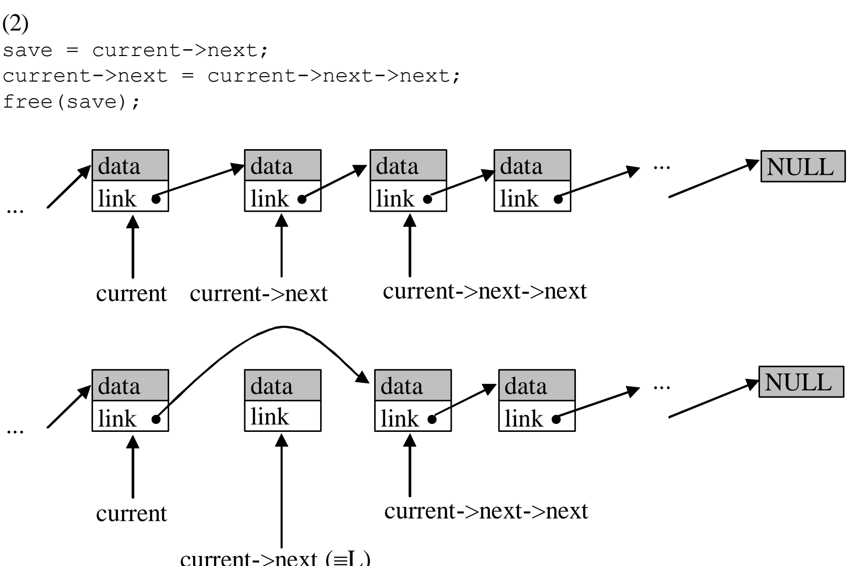
Функцията за изтриване дефинираме като `void deleteNode(struct list **L, keyType key)`. Първата стъпка е да бъде открит елементът с ключ `key`. Търсенето се извършва по такъв начин, че в момента, в който намерим елемента за изтриване, да разполагаме и с указател към неговия *предшественик*:

(1)

```
struct list *current = *L;
while (current->next != NULL && current->next->key != key)
  current = current->next;
```

Така елементът за изтриване е соченият от `current->next`. Изтриването се извършва по-следния начин: На предшественика на елемента за изтриване се задава да сочи към наследника му, т. е. елементът се "прескача", след което паметта, която заема, се освобождава — *фигура 2.1.2г.*:

(2)
```
save = current->next;
current->next = current->next->next;
free(save);
```



Фигура 2.1.2г. Изключване от списък.

При изключване на елемент от списък трябва да се обърне внимание на още нещо — при търсенето (1), ако се достигне до края на списъка, то следва, че в него няма елемент с дадения ключ `key` и се извежда съобщение за грешка.

- *Изтриване на елемент, сочен от даден указател*

Операцията изтриване на елемент, сочен от даден указател `L`, е аналогична на операцията изтриване по ключ — необходимо е да се намери предшественикът на елемента, сочен от `L`, и неговият указател към следващ елемент да се пренасочи по подходящ начин. Така сложността на изтриването по ключ, както и по указател към елемент, е линейна. За да се постигне константна сложност при изтриване по указател, е необходимо да се използва двусвързан линеен списък (*виж задача 2.4.*). Следва пълна динамична реализация на свързан списък:

```
#include <stdio.h>
#include <stdlib.h>

typedef int data;
typedef int keyType;

struct list {
  keyType key;
  data info;
  struct list *next;
};

/* включва елемент в началото на свързан списък */
void insertBegin(struct list **L, keyType key, data x)
{
```

```
  struct list *temp;
  temp = (struct list *) malloc(sizeof(*temp));
  if (NULL == temp) {
    fprintf(stderr, "Няма достатъчно памет за нов елемент!\n");
    return;
  }

  temp->next = *L;
  (*L) = temp;
  (*L)->key = key;
  (*L)->info = x;
}

/* включва след елемент */
void insertAfter(struct list **L, keyType key, data x)
{ struct list *temp;

  if (NULL == *L) {      /* ако списъкът е празен => специален случай */
    insertBegin(L, key, x);
    return;
  }

  temp = (struct list *) malloc(sizeof(*temp));
  /* създаване на новия елемент */
  if (NULL == temp) {
    fprintf(stderr, "Няма достатъчно памет за новия елемент!\n");
    return;
  }

  temp->key = key;
  temp->info = x;
  temp->next = (*L)->next;
  (*L)->next = temp;
}

/* включва преди елемент */
void insertBefore(struct list **L, keyType key, data x)
{ struct list *temp;

  if (NULL == *L) {      /* елементът трябва да се вмъкне преди първия */
    insertBegin(L, key, x);
    return;
  }

  temp = (struct list *) malloc(sizeof(*temp));
  /* създаване на новия елемент */
  if (NULL == temp) {
    fprintf(stderr, "Няма достатъчно памет за новия елемент!\n");
    return;
  }

  *temp = **L;
  (*L)->next = temp;
  (*L)->key = key;
  (*L)->info = x;
}

/* изтрива елемент от списъка */
void deleteNode(struct list **L, keyType key)
```

```
{ struct list *current = *L;
  struct list *save;
  if ((*L)->key == key) {  /* трябва да се изтрие първият елемент */
    current = (*L)->next;
    free(*L);
    (*L) = current;
    return;
  }

  /* намира елемента, който ще се трие */
  while (current->next != NULL && current->next->key != key) {
    current = current->next;
  }

  if (NULL == current->next) {
    fprintf(stderr, "Грешка: Елементът за изтриване не е намерен! \n");
    return;
  }
  else {
    save = current->next;
    current->next = current->next->next;
    free(save);
  }
}

/* отпечатва елементите на свързан списък */
void print(struct list *L)
{ while (NULL != L) {
    printf("%d(%d) ", L->key, L->info);
    L = L->next;
  }
  printf("\n");
}

/* търси по ключ елемент в свързан списък */
struct list* search(struct list *L, keyType key)
{ while (L != NULL) {
    if (L->key == key) return L;
    L = L->next;
  }
  return NULL;
}

int main(void) {
  struct list *L = NULL;
  int i, edata;
  insertBegin(&L, 0, 42);
  for (i = 1; i < 6; i++) {
    edata = rand() % 100;
    printf("Вмъкване преди: %d(%d)\n", i, edata);
    insertBefore(&L, i, edata);
  }

  for (i = 6; i < 10; i++) {
    edata = rand() % 100;
    printf("Вмъкване след: %d(%d)\n", i, edata);
    insertAfter(&L, i, edata);
  }
```

```
  print(L);
  deleteNode(&L, 9); print(L);
  deleteNode(&L, 0); print(L);
  deleteNode(&L, 3); print(L);
  deleteNode(&L, 5); print(L);
  deleteNode(&L, 5);
  return 0;
}
```

🖫 list.c

Резултат от изпълнението на програмата:

```
Вмъкване преди: 1(46)
Вмъкване преди: 2(30)
Вмъкване преди: 3(82)
Вмъкване преди: 4(90)
Вмъкване преди: 5(56)
Вмъкване след: 6(17)
Вмъкване след: 7(95)
Вмъкване след: 8(15)
Вмъкване след: 9(48)
5(56) 9(48) 8(15) 7(95) 6(17) 4(90) 3(82) 2(30) 1(46) 0(42)
5(56) 8(15) 7(95) 6(17) 4(90) 3(82) 2(30) 1(46) 0(42)
5(56) 8(15) 7(95) 6(17) 4(90) 3(82) 2(30) 1(46)
5(56) 8(15) 7(95) 6(17) 4(90) 2(30) 1(46)
8(15) 7(95) 6(17) 4(90) 2(30) 1(46)
Грешка: Елементът за изтриване не е намерен!
```

Задачи за упражнение:

1. Да се преработят горните програми така, че функциите да не печатат съобщения за грешка, а да връщат булева стойност дали са успели.
2. Структурите от данни стек, опашка, дек и т.н. могат да се разглеждат като специални линейни списъци и също могат да бъдат реализирани динамично. Предоставяме на читателя, като използва горната реализация на линеен едносвързан списък, да се опита да я модифицира до динамична реализация на стек и опашка. Важно е сложността на основните операции (включване, изключване) да запазят константната си сложност.
3. Да се реализира комбинирана стратегия за представяне на стек в паметта, при която се използва *списък от масиви*. В началото се започва с единствен масив и, когато той се напълни, се заделя нов блок, който се свързва с указател към предишния. Когато последно включеният масив се изпразни, той се изключва от списъка. Какви са предимствата и недостатъците на предложения подход? Бихте ли го препоръчали и за опашка? А за дек?

2.2. Двоични дървета

Използването на линеен списък е крайно неефективно при задачи, в които операцията *търсене* се изпълнява много по-често от останалите, отнасящи се до промяна на списъка: включване, изтриване, пренареждане и др.

Както вече видяхме, при динамична реализация на линеен списък включването и изключването на елемент са бързи операции (тяхната сложност е константа — $\Theta(1)$), за разлика от търсенето по ключ, което има сложност $\Theta(n)$. При статична реализация сложността на включването е $\Theta(1)$, а на изключването, както и на търсенето — $\Theta(n)$. Възможно е елементите да бъдат съхранявани в списъка, сортирани по-своя ключ (статична реализация). Това ще намали ефективността на операцията включване (елементите ще трябва да се пренареждат), но ще позволи прилагане на класическо *двоично търсене* (разгледано в *глава 4 — виж 4.3.*) и сложността на търсенето ще на-

малее до $\Theta(\log_2 n)$ (структурата се нарича *приоритетна опашка*, *виж задача 2.6.*). Прилагането на двоично търсене при динамична реализация е невъзможно, тъй като няма директен достъп до k-тия елемент от списъка.

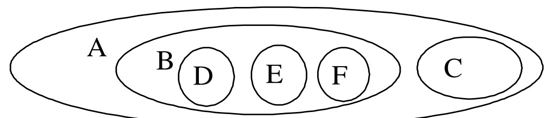
Ще разгледаме *дървовидна* структура от данни, която комбинира преимуществата на динамичната реализация с ефективността на търсене при статичната. При тази структура, трите операции включване, изключване и търсене по ключ в средния случай имат логаритмична сложност.

Дефиниция 2.2. Множеството T е *кореново дърво* (по-нататък ще казваме само *дърво*), ако е празното множество, или ако са едновременно изпълнени следните условия:

1) Съдържа единствен елемент t, наречен *корен* на дървото, който ще означаваме с $root(T)$.
2) Останалите елементи (с изключение на корена) са разделени на m ($m \geq 0$) непресичащи се непразни множества $T_1, T_2, ..., T_m$, всяко от които е *дърво*.

Множествата $T_1, T_2, ..., T_m$ се наричат *поддървета* на T. Корените t_1 на T_1, t_2 на T_2, ..., t_m на T_m се наричат *преки наследници* на t. Всички останали върхове от $T_1, T_2, ..., T_m$ са *непреки наследници* на t.

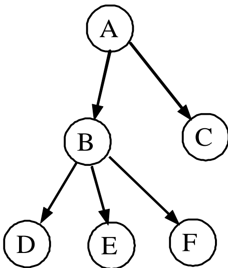
Като следствие всеки елемент (по-нататък, елементите на T ще наричаме още *върхове на дървото*) е корен на някакво поддърво. Броят на преките наследници на даден връх е неговата *степен*. Ако степента на връх е 0, той се нарича *листо*.



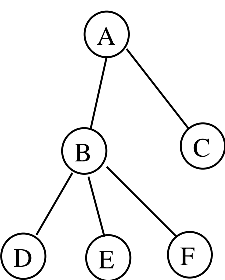
Фигура 2.2а. Нагледно представяне чрез вложени множества.

(**A** (**B**(**D**)(**E**)(**F**)) (**C**))

Фигура 2.2б. Представяне чрез вложени скоби.



Фигура 2.2в. Представяне чрез ориентиран ацикличен свързан граф (*виж глава 5*).



Фигура 2.2г. Представяне чрез неориентиран ацикличен свързан граф (*виж глава 5*), като един от върховете е избран за корен: в случая връх A.

Фигура 2.2д. Представяне чрез стъпаловидно отместване.

На *фигури 2.2а.*, *2.2б.*, *2.2в.*, *2.2г.* и *2.2д.* са показани различни начини за нагледно представяне. Да разгледаме дървото $T = \{A, B, C, D, E, F\}$, показано на тези фигури: тук върхът A е корен, а $T_1 = \{C\}$ и $T_2 = \{B, D, E, F\}$ са негови поддървета (върхът C е листо). В дървото $T_2 = \{B, D, E, F\}$ корен е върхът B, а поддървета са $\{D\}$,$\{E\}$,$\{F\}$ (D, E и F са листа).

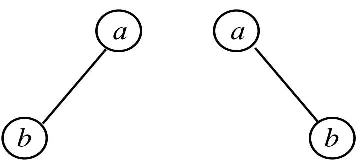
Дефиниция 2.3. *Път* в дърво е последователност от върхове t_1, t_2, ..., t_k без повторение, като за всеки два последователни върха t_{i-1} и t_i ($2 \le i \le k$) е изпълнено точно едно от двете: t_i е наследник на t_{i-1}, или t_{i-1} е наследник на t_i. Числото k–1 се нарича *дължина* на пътя.

Твърдение. Между всеки два върха в дърво съществува единствен път.

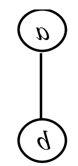
Дефиниция 2.4. *Ниво* на връх се нарича дължината на пътя от корена до върха. *Височина на дърво* се нарича максималното *ниво* на връх в него.

Дефиниция 2.5. Ако степента на всеки връх от дърво е по-малка или равна на две, то дървото се нарича *двоично дърво*. Тъй като поддърветата на всеки връх са най-много две, най-често се въвежда наредба на наследниците (и съответно на поддърветата): *ляв* и *десен*.

Следва да се отбележи, че, след като сме въвели наредба, в структурно отношение двоичните дървета качествено се отличават от дърветата и съвсем не са тяхно подмножество. Така например двата обекта от *фигура 2.2е.* съвсем не изобразяват едно и също *двоично* дърво, макар като дървета да са неразличими и да представляват дървото от *фигура 2.2ж.*



Фигура 2.2е. Двоични дървета.



Фигура 2.2ж. Дърво.

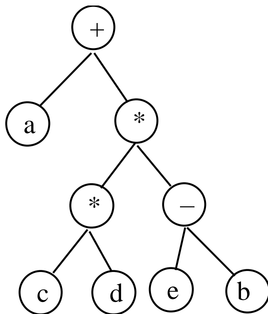
До края на този параграф ще разглеждаме двоични дървета. Следват някои примери:

- Схема за турнир по тенис, където се играе с пряко елиминиране.
- Представяне на аритметичен израз с двуаргументни операции (*виж фигура 2.2и.*): всяка операция е връх с наследници двата ѝ операнда.
- Родословните дървета, при които *наследниците* на всеки връх са неговите родители. Ще отбележим, че примерът от *фигура 2.2з.* не е двоично дърво в този си вид.

Например, в случай на двама братя, е нарушено условието за всеки връх t_1 да съществува единствен връх t_2, такъв че t_1 е наследник на t_2. Каква част от "дървото" трябва да се отсече, така че примерът да отговаря на *дефиниция 2.2.*?

Двоичните дървета намират широко приложение при компилаторите, базите от данни и др.

Фигура 2.2з. Родословно дърво на Симеон Сакс-Кобурготски. [*Симеон-II*].

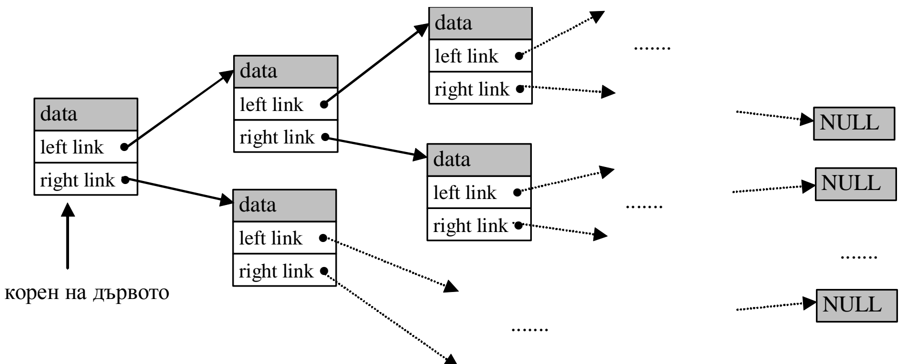


Фигура 2.2и. Представяне на израза $a+((c*d)*(e–b))$.

Нека е дадено двоично дърво T, като във всеки негов връх t се пази следната информация (*фигура 2.2й.*):

- уникален ключ `key(t)` на върха.
- поле `data(t)` с допълнителни данни за върха.
- два указателя `left(t)` и `right(t)`: към лявото и към дясното поддърво на t съответно

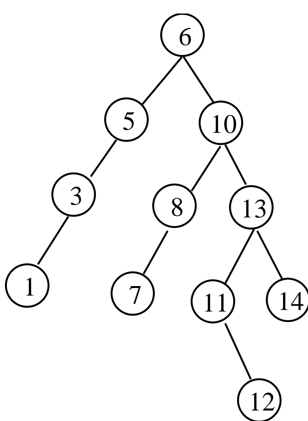
```
typedef char *data;
typedef int keyType;

struct tree {
  keyType key;
  data info;
  struct tree *left;
  struct tree *right;
};
```



Фигура 2.2й. Динамично представяне на двоично дърво в паметта.

Ще дефинираме три основни операции върху двоични дървета:

- Добавяне на връх
`void insertKey(keyType key, data x, struct tree **T)`
- Изтриване на връх по зададен ключ
`void deleteKey(keyType key, struct tree **T);`
- Търсене на връх (и извличане на данните от него) по ключ
`struct tree *search(keyType key, struct tree *T);`



Фигура 2.2к. Двоично дърво за претърсване.

Разновидност на двоичните дървета са *двоичните дървета за претърсване*. С тяхна помощ ще реализираме току-що изброените операции така, че сложността им да бъде логаритмична.

Дефиниция 2.6. *Двоичното дърво за претърсване* (още *наредено двоично дърво*) е двоично дърво, за всеки връх на което е изпълнено: ключът му е по-голям от ключовете на всички елементи от лявото му поддърво, и по-малък (или равен, в случай, че в дървото се допуска добавяне на елементи с еднакви ключове) от ключовете на всички елементи от дясното.

Фигура 2.2к. показва двоично дърво за претърсване (ключовете на елементите са цели числа). Вижда се, че ключът на всеки връх е с по-голяма стойност от ключовете, намиращи се в лявото му поддърво, и с по-малка стойност от ключовете от дясното му поддърво. В частност последното условие е изпълнено и за наследниците на всеки връх (това има значение при реализацията на операциите в дървото).

Търсенето по ключ в двоично дърво се извършва по-следния начин:

- *Търсене по даден ключ*

Започваме търсенето от корена, т. е. `t = root(T);`

`search(key, t):`

1) Ако `key < key(t)`, продължаваме търсенето в лявото поддърво на `t`, т. е. изпълняваме рекурсивно `search(left(t), key);`
2) Ако `key > key(t)`, продължаваме търсенето в дясното поддърво на `t`, т. е. изпълняваме рекурсивно `search(right(t), key);`
3) Ако `key == key(t)`, сме открили търсения връх.

В горния алгоритъм сме приели, че ключът, по който търсим, се намира в дървото. Ако се търси по ключ, който не съществува, то на някоя стъпка ще се окаже, че дървото, което подлежи на претърсване, е празно (т. е. съответният указател на родителя сочи `NULL`). Например, ако в дървото от *фигура 2.2к.* търсим елемент с ключ 9, алгоритъмът ще извърши последователно следните проверки:

$9 > 6$ — преминаваме към корена на дясното поддърво — върха с ключ 10.

$9 < 10$ — преминаваме към корена на лявото поддърво — върха с ключ 8.

$9 > 8$ — трябва да преминем към корена на дясното поддърво, но той сочи `NULL`, следователно търсеният елемент липсва.

- *Включване на нов връх*

Включването се извършва аналогично на търсенето, като целта тук е да се стигне до празно дърво, на което да се добави нов връх.

insert(t, p):

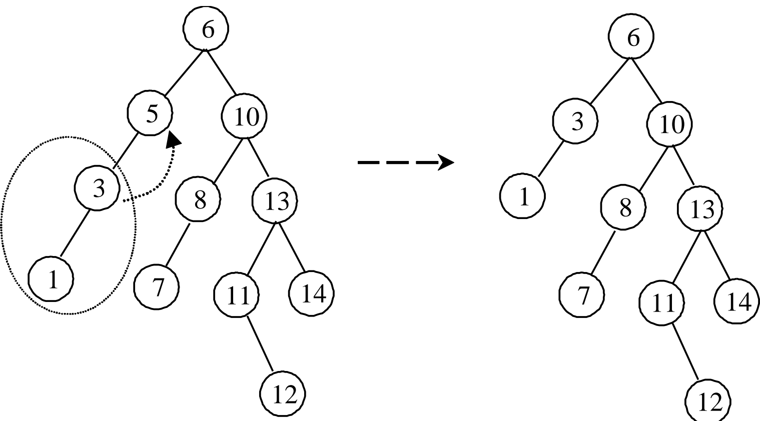
1) Ако `t == NULL` следва, че сме намерили къде трябва да включим новия връх *p* и го включваме: `t = p;`
2) Ако `key(p) < key(t) insert(left(t), key );`
3) Ако `key(p) > key(t) insert(right(t), key );`
4) Ако `key(p) == key(t)` следва, че елементът, който ще се вмъква, е вече в дървото. В този случай имаме две възможности: може да не предприемаме нищо, или да изведем съобщение за грешка (т. е. не допускаме дублиране на ключове).

- *Изключване на връх по даден ключ*

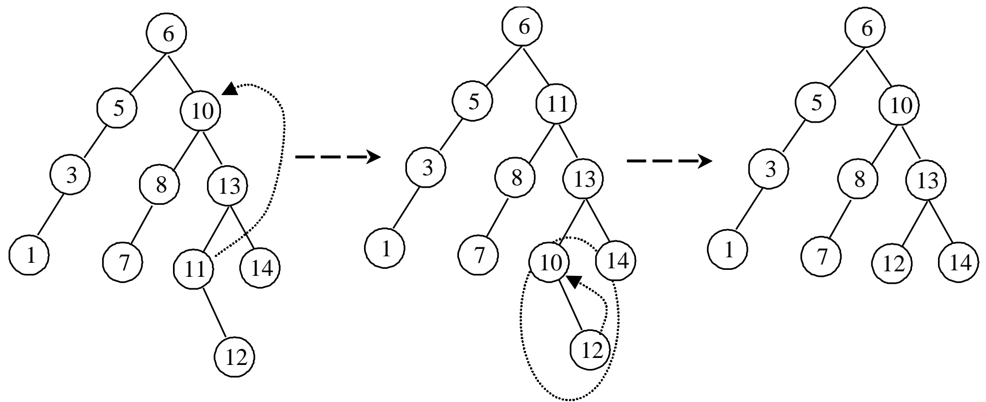
Тази операция е малко по-сложна. Ясно е, че преди да изтрием върха с ключ *k* от дървото, трябва да го намерим. Това става по вече описания алгоритъм за търсене на връх. По-нататък са възможни 3 ситуации:

- Ако върхът е листо — освобождава се заетата от него памет и се променя указателят на върха, сочещ към него (присвоява му се стойност `NULL`).
- Ако върхът има само ляво или само дясно поддърво, се замества с корена на това поддърво.
- Най-сложният случай е, когато върхът за изтриване *p* има едновременно ляво и дясно поддърво. Тогава се прилага следното: Намира се върхът с най-малък ключ в дясното поддърво (най-левият в дясното поддърво) и този връх се разменя с *p*. След размяната *p* ще има най-много едно поддърво и се изключва по някое от горните две правила. (Разбира се, бихме могли да разменим ляво с дясно и обратно: да намерим елемента с най-голям ключ в лявото поддърво, да го разменим с *p* и т.н.)

Последните два случая сме илюстрирали на *фигура 2.2л.* и *фигура 2.2м.*



Фигура 2.2л. Изключване на върха с ключ 5.



Фигура 2.2м. Изключване на върха с ключ 10.

Следва пълна реализация на описаните по-горе операции:

```
#include <stdio.h>
#include <stdlib.h>

typedef char *data;
typedef int keyType;
struct tree {
  keyType key;
  data info;
  struct tree *left;
  struct tree *right;
};
/* Търсене в двоично дърво */
struct tree *search(keyType key, struct tree *T)
{ if (NULL == T)
    return NULL;
  else if (key < T->key)
    return search(key, T->left);
  else if (key > T->key)
    return search(key, T->right);
  else
    return T;
}

/* Включване в двоично дърво */
void insertKey(keyType key, data x, struct tree **T)
{ if (NULL == *T) {
    *T = (struct tree *) malloc(sizeof(**T));
    (*T)->key = key;
    (*T)->info = x;
    (*T)->left = NULL;
    (*T)->right = NULL;
  }
  else if (key < (*T)->key)
    insertKey(key, x, &(*T)->left);
  else if (key > (*T)->key)
    insertKey(key, x, &(*T)->right);
  else
    fprintf(stderr, "Елементът е вече в дървото!\n");
```

```
}

/* Изключване от двоично дърво */

/* Намиране на минималния елемент в дърво */
struct tree *findMin(struct tree *T)
{ while (NULL != T->left) T = T->left;
  return T;
}

void deleteKey(keyType key, struct tree **T)
{ if (NULL == *T) {
    fprintf(stderr,"Върхът,който трябва да се изключи, липсва!\n");
  } else {
    if (key < (*T)->key)
      deleteKey(key, &(*T)->left);
    else if (key > (*T)->key)
      deleteKey(key, &(*T)->right);
    else  /* елементът за изключване е намерен */
    if ((*T)->left && (*T)->right) {/* върхът има два наследника */
      /* намира се върхът за размяна */
      struct tree *replace = findMin((*T)->right);
      (*T)->key = replace->key;
      (*T)->info = replace->info;
      deleteKey((*T)->key, &(*T)->right);  /* върхът се изключва */
    }
    else {  /* елементът има нула или едно поддървета */
      struct tree *temp = *T;
      if ((*T)->left)
        *T = (*T)->left;
      else
        *T = (*T)->right;
      free(temp);
    }
  }
}

void printTree(struct tree *T)
{ if (NULL == T) return;
  printf("%d ", T->key);
  printTree(T->left);
  printTree(T->right);
}

int main(void) {
  struct tree *T = NULL, *result;
  int i;

  /* включва 10 върха с произволни ключове */
  for (i = 0; i < 10; i++) {
    int ikey = (rand() % 20) + 1;
    printf("Вмъква се елемент с ключ %d \n", ikey);
    insertKey(ikey, "someinfo", &T);
  }

  printf("Дърво: ");

  printTree(T);
  printf("\n");
```

```
  /* претърсва за елемента с ключ 5 */
  result = search(5, T);
  printf("Намерен е: %s\n", result->info);

  /* изтрива произволни 10 върха от дървото */
  for (i = 0; i < 10; i++) {
    int ikey = (rand() % 20) + 1;
    printf("Изтрива се елемента с ключ %d \n", ikey);
    deleteKey(ikey, &T);
  }

  printf("Дърво: ");

  printTree(T);
  printf("\n");
  return 0;
}
```

bintree.c

Резултат от изпълнението на програмата:

```
Вмъква се елемент с ключ 1
Вмъква се елемент с ключ 1
Елементът е вече в дървото!
Вмъква се елемент с ключ 7
Вмъква се елемент с ключ 1
Елементът е вече в дървото!
Вмъква се елемент с ключ 8
Вмъква се елемент с ключ 5
Вмъква се елемент с ключ 11
Вмъква се елемент с ключ 4
Вмъква се елемент с ключ 15
Вмъква се елемент с ключ 19
Дърво: 1 7 5 4 8 11 15 19
Намерен е: someinfo
Изтрива се елемента с ключ 6
Върхът, който трябва да се изключи, липсва!
Изтрива се елемента с ключ 9
Върхът, който трябва да се изключи, липсва!
Изтрива се елемента с ключ 3
Върхът, който трябва да се изключи, липсва!
Изтрива се елемента с ключ 14
Върхът, който трябва да се изключи, липсва!
Изтрива се елемента с ключ 12
Върхът, който трябва да се изключи, липсва!
Изтрива се елемента с ключ 1
Изтрива се елемента с ключ 4
Изтрива се елемента с ключ 17
Върхът, който трябва да се изключи, липсва!
Изтрива се елемента с ключ 14
Върхът, който трябва да се изключи, липсва!
Изтрива се елемента с ключ 16
Върхът, който трябва да се изключи, липсва!
Дърво: 7 5 8 11 15 19
```

- *Обхождане*

Вижда се, че в програмата по-горе сме използвали и функция за отпечатване на всички върхове на двоично дърво `printTree(struct tree *T)`. Функцията е рекурсивна и се основава на следния алгоритъм: Отпечатва се коренът на дървото, след което се отпечатва (рекурсия!) лявото, а след това и дясното поддърво на *T*.

Без да даваме формална дефиниция (такава ще има при разглеждане на графи в *глава 5*), под *обхождане на дърво* ще разбираме последователното разглеждане на всеки негов връх, започвайки от корена.

Горната функция за отпечатване на дървото е на практика обхождане. При нея първо се разглежда коренът и едва след това върховете от лявото и дясното поддърво. Това обхождане носи названието *обхождане корен-ляво-дясно* или *КЛД* (на англ. *preorder*). Очевидно, като се използва същият принцип, дървото може да се обходи по още два съществено различни начина:

- *ЛКД* (англ. *inorder*): Разглежда се в този ред: ляво поддърво, корен и дясно поддърво.
- *ЛДК* (англ. *postorder*): Първо се разглеждат върховете на лявото и дясното поддърво и едва след това се разглежда коренът.

Всеки един от трите вида обхождания намира приложение при различни задачи и интерпретации на двоичното дърво. Например, ако приложим *inorder* в двоично дърво за претърсване и отпечатваме ключа на всеки връх, който посещаваме, ще получим ключовете, сортирани във възходящ ред.

Да разгледаме двоичното дърво от *фигура 2.2и.* Нека го обходим и да видим какво се получава:

- *КЛД*: *+a**cd–eb*
- *ЛКД*: *a+c*d*e–b*
- *ЛДК*: *acd*eb–*+*

Всеки един от трите вида обхождания има и своята "огледална" разновидност — разменяме реда на обхождане на лявото и дясното поддърво:

- *КДЛ*: *+*–be*dca*
- *ДКЛ*: *b–e*d*c+a*
- *ДЛК*: *be–dc**a+*

Така видовете обхождания стават общо 6: толкова, колкото са пермутациите на буквите *Л*(яво), *Д*(ясно) и *К*(орен). Вижда се, че при обхождане от тип *ЛКД* се получава стандартният алгебричен запис на израза (който обаче не запазва приоритетите при пресмятането и, за да може да се използва на практика, трябва да се модифицира по подходящ начин — как?), а при *КЛД* и *ЛДК* се получават съответно *прав* и *обратен полски* запис. Правият и обратен полски запис са полезни, тъй като по тях лесно може да се пресметне числената стойност, при това без използване на скоби за промяна на приоритета на операциите (при съпоставяне на стойности на променливите от израза, в случая това са *a*, *c*, *d*, *e* и *f*) [*Уирт-1980*][*Наков-1998*].

Задачи за упражнение:

1. Да се построи наредено двоично дърво за търсене чрез последователно добавяне на следните върхове:

   а) 7, 14, 28, 35, 65, 12, 18, 42, 81, 64, 61, 4, 13
   б) 12, 7, 14, 81, 42, 18, 61, 4, 64, 35, 13, 28, 65
   в) 4, 7, 12, 13, 14, 18, 28, 35, 42, 61, 64, 65, 81
   г) 81, 65, 64, 61, 42, 35, 28, 18, 14, 13, 12, 7, 4
   д) 28, 64, 13, 42, 7, 81, 61, 4, 12, 65, 35, 18, 14

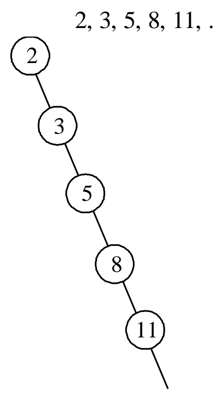
Да се сравнят получените дървета. Какви изводи могат да се направят?

2. Да се изтрият в този ред върховете 8, 13, 5 и 6 от дървото на *фигура 2.2к*.

3. Даден е изразът `(A+B–C)*(D/F)–G*H`. Да се намери неговият обратен полски запис.
4. Да се докаже, че между всеки два върха в дърво съществува единствен път.
5. Каква най-малка част от "дървото" на *фигура 2.2з.* трябва да се отсече така, че примерът да отговаря на *дефиниция 2.2.*?
6. Да се напише итеративен вариант на функцията за обхождане на двоично дърво.
7. Да се напише вариант на функцията за обхождане на двоично дърво, която да го отпечатва така, че да се вижда структурата му.
8. Да се докаже коректността на описания алгоритъм за изтриване на връх от наредено двоично дърво.
9. Описаният по-горе алгоритъм за изтриване на връх от наредено двоично дърво в случай на два наследника търсеше *най-левия наследник на дясното поддърво*. Да се реализира вариант, който търси най-десния наследник на лявото поддърво. Съществуват ли други възможности?
10. Да се предложи и реализира статично представяне на двоично дърво.
11. По-горе видяхме, че стандартен начин за представяне на израз в двоично дърво е в листата да се записват променливи и константи, а във върховете — операции. Забележете, че позволявахме само *двуаргументни* операции (като +, –, *, /), при което левият наследник съдържа първия аргумент, а десният — втория. Възможно ли е да позволим и унарни (едноаргументни) операции като: '+' и '–'? Какво се променя?

2.3. Балансирани дървета

При по-внимателен анализ на операциите с двоично наредено дърво се забелязва, че съществуват случаи, при които полученото дърво е много слабо разклонено и прилича на списък по структура, а оттам и по ефективност. Нека, започвайки от празно дърво, да изпълним последователност от n включвания на елементи. Полученият резултат може да бъде крайно "неприятен" — например, ако редицата от ключовете на включваните елементи е строго растяща (*фигура 2.3а.*):



Фигура 2.3а. Най-лош случай при двоично наредено дърво.

Полученото дърво се "изражда" в свързан списък, търсенето в който, както знаем, става с *линейна* сложност. Често в практиката е необходимо да разполагаме с бързо търсене, независимо от вида на входните данни, и в такива случаи линейна сложност в най-лошия случай е неприемливо решение.

Ще разгледаме структура от данни, която "запазва" дървото в състояние, при което основните операции (включване, изключване, търсене) имат логаритмична сложност дори и в най-лошия случай.

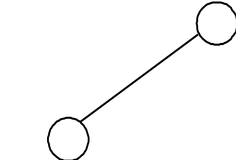
Дефиниция 2.7. Едно двоично дърво се нарича *балансирано*, ако разликата във височините на лявото и дясното поддърво на всеки връх е най-много единица.

Дефиниция 2.8. Едно двоично дърво с n върха се нарича *идеално балансирано*, ако разликата в броя на върховете на лявото и дясното поддърво на всеки връх е най-много единица.

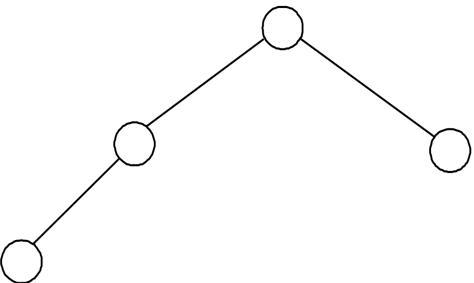
Дефиниция 2.9. *Дърво на Фибоначи* T_k *от ред* k, се нарича двоично дърво, за което:

- T_0 е празното дърво — с височина 0;
- T_1 е дърво, съдържащо единствен възел — с височина 1;
- За $k \geq 2$ дървото се състои от корен, дърво на Фибоначи T_{k-1} от ред k–1 (ляво поддърво) и дърво на Фибоначи T_{k-2} от ред k–2 (дясно поддърво).

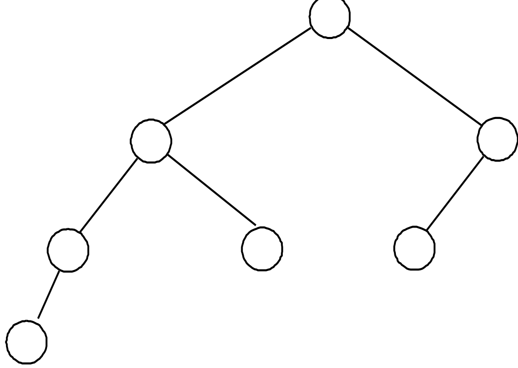
Фигура 2.3б. Дърво на Фибоначи от ред 0.



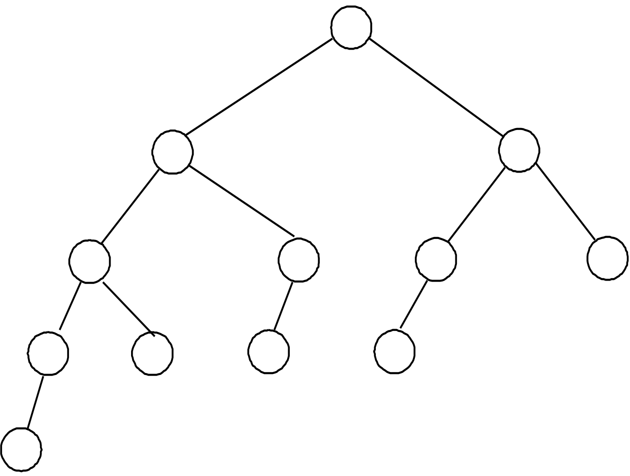
Фигура 2.3в. Дърво на Фибоначи от ред 1.



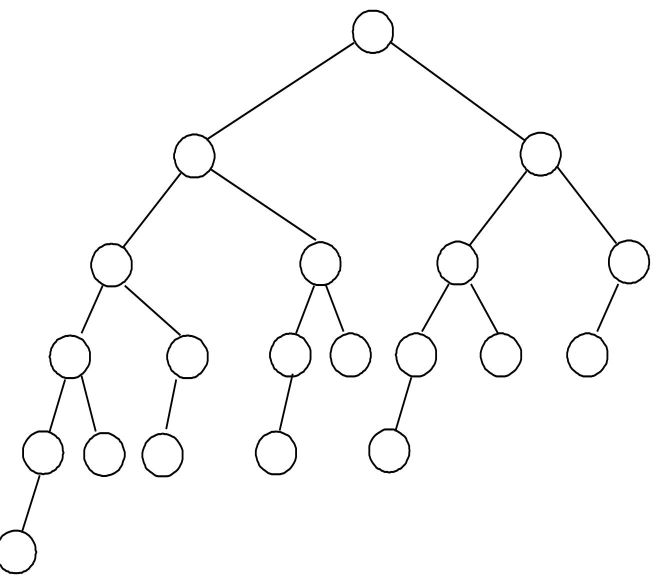
Фигура 2.3г. Дърво на Фибоначи от ред 2.



Фигура 2.3д. Дърво на Фибоначи от ред 3.



Фигура 2.3е. Дърво на Фибоначи от ред 4.



Фигура 2.3ж. Дърво на Фибоначи от ред 5.

Първите няколко дървета на Фибоначи са показани на *фигури 2.3б, 2.3в., 2.3г., 2.3д.*, *2.3е.* и *2.3ж.* Лесно е забелязва, че дървото на Фибоначи е най-лошият случай, който можем да получим за двоично балансирано дърво. Този факт се използва и при следващата теорема:

Теорема. (*Аделсън*, *Велски и Ландис*) Нека е дадено двоично дърво T с *n вътрешни* върха. Нека h е височината на T. Тогава:

$$\log_2(n+1) < h < 1{,}4404.\log_2(n+2) - 0{,}3277$$

Как се получава последния резултат? Очевидно, двоично дърво с височина h не може да има повече от 2^h вътрешни върха, т. е. $n + 1 \leq 2^h$, или $h \geq \lceil \log_2(n+1) \rceil$.

За да ограничим h отгоре, нека означим с T_h балансирано дърво, със следното свойство: T_h да бъде с височина h и с минимален брой върхове. Тъй като T_h е балансирано, то (без

ограничение на общността) лявото поддърво на T_h е с височина $h-1$, а дясното: с височина $h-1$ или $h-2$. Тъй като T_h е дърво с минимален брой върхове, то лявото поддърво на корена на T_h ще бъде T_{h-1}, а дясното: T_{h-2}. При индуктивно продължение на построяването, ще получим че T_h ще бъде именно дървото на Фибоначи от ред $h+1$ (*Защо?*). Така $n \geq F_h + 2$, и от свойството [*Knuth-3/1968*]:

$$F_{h+2}-1 > \varphi^{h+2} / \sqrt{5} - 2$$

се получава другата част от неравенствата.

Нека сега отново се върнем на двоичните дървета за претърсване. При последователно добавяне на нови елементи съществуват основно два подхода за запазване на дърво за претърсване в "балансиран" вид:

- преструктуриране на двоичното дърво при изграждане.
- намаляване на нивото чрез използване на дървета от по-висок ред — k-ични.

Тези два подхода, както и някои специфични разновидности на балансирани дървета, са разгледани в следващите два параграфа.

Задачи за упражнение:

1. Да се построи *идеално балансирано* двоично дърво за търсене за следното множество от върхове {4, 7, 12, 13, 14, 18, 28, 35, 42, 61, 64, 65, 81}.
2. *Балансирано* ли е дървото от:
   а) *фигура 2.2г.*
   б) *фигура 2.2и.*
   в) *фигура 2.2к.*
3. *Идеално балансирано* ли е дървото от:
   а) *фигура 2.2г.*
   б) *фигура 2.2и.*
   в) *фигура 2.2к.*

2.3.1. Ротация. Червено-черни дървета

За да се запази логаритмична височината на двоично дърво за претърсване, се дефинират специални преструктурирания (ротации), които се прилагат винаги, когато се получи "дисбаланс". Такива са лява и дясна ротация, двойна ротация и др. (*фигура 2.3.1а.*).

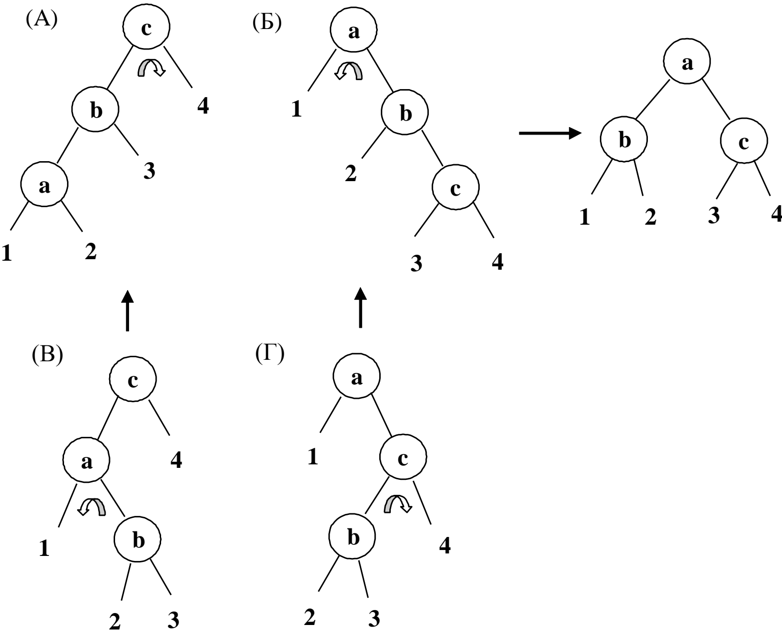
Важни свойства на показаните ротации са:

- реализацията им е едновременно проста и ефективна;
- запазват резултата от обхождане на дървото тип ЛКД;
- гарантират балансираност (но не и идеална балансираност) на дървото.

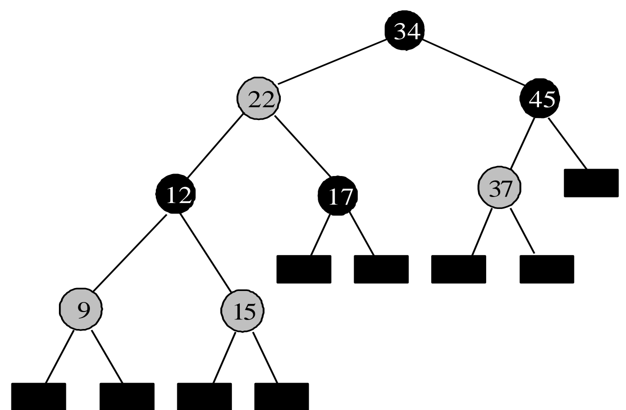
Изборът, какви ротации да се изпълняват върху различните върхове, подлежи на допълнително уточняване, свързано с конкретни алгоритми. Като пример ще разгледаме интересна и широко разпространена структура от данни — т.нар. *червено-черно дърво* (от англ. *Red-Black Tree*).

Дефиниция 2.10. *Червено-черно дърво* се нарича двоично дърво за претърсване, в което всеки връх е означен или като *червен*, или като *черен* и допълнително са изпълнени следните свойства:

- Всички листа (върховете със стойност NULL, те реално *не* присъстват) са черни.
- Ако даден връх е червен, то двата му наследника са черни.
- Всички пътища от произволен връх t до произволно листо от поддървото с корен t съдържат еднакъв брой черни върхове.



Фигура 2.3.1а. Дясна (А), лява (Б), двойна ляво-дясна (В) и двойна дясно-лява (Г) ротация.



Фигура 2.3.1б. Пример за червено-черно дърво.

Пряко следствие от *дефиниция 2.10.* е следното полезно свойство: височината на червено-черно дърво с n върха е най-много $2.\log_2(n+1)$, т. е. червено-черните дървета са приблизително балансирани двоични дървета за претърсване. Те обаче не са балансирани съгласно нашата дефиниция. Да си припомним, че балансирано е дърво, за което разликата във височините на лявото и дясното поддърво на всеки връх е най-много 1. При червено-черните тази разликата е най-много два пъти. За червено-черните дървета съществуват относително прости алгоритми за реализиране на основните операции (включване, търсене, изключване) със сложност $\Theta(\log_2 n)$, поради което те са най-масово използваните в практиката балансирани дървета [*Шишков-1995*] [*Cormen, Leiserson, Rivest-1997*].

Съществуват други класически видове балансирани двоични дървета за претърсване — например *AVL-дървета* (от имената на *Adelson-Velskii и Landis*), които също гарантират логарит-

мична сложност за посочените по-горе операции [*Уирт-1980*][*Knuth-3/1968*]. Алгоритмите за работа с *AVL*-дървета обаче са по-сложни и ефективността им на практика често е по-лоша от тази на червено-черните, поради което се използват по-рядко.

Задачи за упражнение:

1. Като се използва *дефиниция 2.10.*, да се докаже, че всяко нелисто в червено-черно дърво има точно два наследника.
2. Да се помисли как биха могли да се реализират основните операции за работа с червено-черни дървета при даденото описание на структурата им.

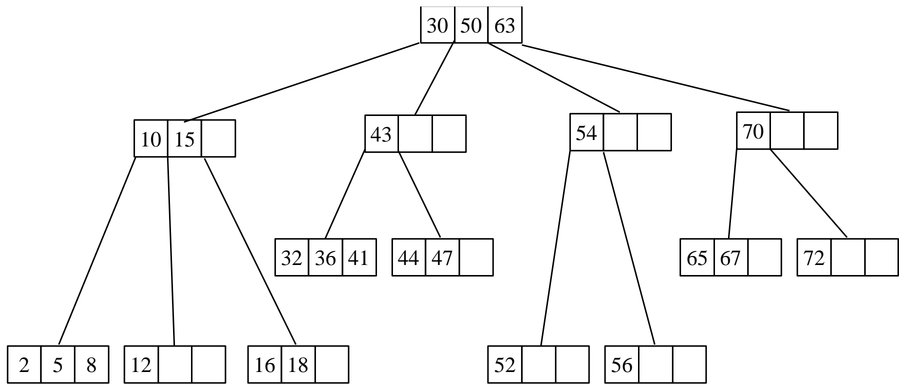
2.3.2. B-дървета

Идеята за построяване на дърво за претърсване (в материала досега разглеждахме само двоични дървета) може да се разшири и до дървета от по-висока степен.

Дефиниция 2.11. *k-връх* в дърво наричаме връх, който съдържа информация за $k-1$ елемента (тук под елемент се разбира ключ + допълнителните данни) с ключове $t_1 < t_2 < ... < t_{k-1}$ и с k наследника – поддърветата T_1, T_2 ,..., T_k, където за всеки ключ t_i, i = 1,2,..., $k-1$, е изпълнено следното условие: всички ключове в поддървото T_i са по-малки от t_i и всички ключове от поддървото T_{i+1} са по-големи от t_i.

Дефиниция 2.12. *2-3-4 дърво* наричаме дърво T със следните свойства:

- T е идеално балансирано дърво.
- Всеки вътрешен връх (т. е. различен от листо) е или 2-връх, или 3-връх, или 4-връх.
- Всички пътища от корена до произволно листо имат еднаква дължина.



Фигура 2.3.2a. 2-3-4 дърво.

Съществуват ефективни алгоритми за работа с 2-3-4 дървета, които гарантират сложност на основните операции (добавяне, търсене, изтриване) от порядъка на $\Theta(\log_2 n)$. Важно е още да се отбележи, че съществува начин за преобразуване на 2-3-4 дърветата в еквивалентни на тях двоични дървета (а именно разгледаните по-горе червено-черни дървета), което позволява да се представят по-просто в паметта и за които съществуват по-прости алгоритми, поради което в практиката вместо 2-3-4 дървета често се използват червено-черни дървета [*Flamig-1993*].

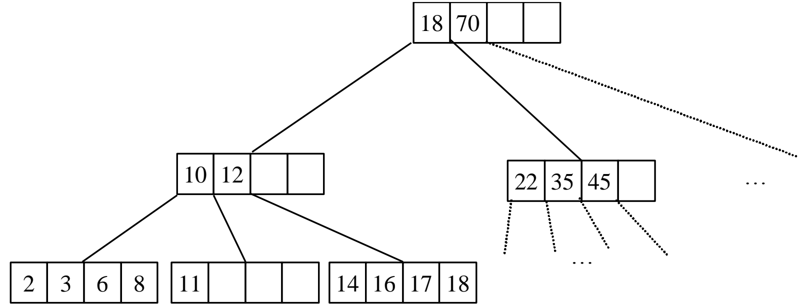
Ще разгледаме важно обобщение на 2-3-4 дървета.

Дефиниция 2.13. *B-дърво* от ред m наричаме дърво за претърсване със следните свойства:

- Всеки връх, освен корена и листата, е k-връх, където k е число между $[m/2]$ и m.

- Всички пътища от корена до произволно листо имат еднаква дължина.

Като следствие от *дефиниция 2.13.* B-дърветата се оказват идеално балансирани дървета за претърсване. В частност, при $m = 4$ получаваме 2-3-4-дърво.



Фигура 2.3.2б. B-дърво от ред 5.

Ще разгледаме някои по-важни свойства и приложения на B-дървета. За любознателния читател съществуват множество източници с допълнителна информация за алгоритми и реализация на основните операции върху B-дърво [*Уирт-1980*][*Шишков-1995*][*Knuth-3/1968*].

B-дърветата се използват широко в практиката, особено при системи за управление на голям обем информация (например релационни бази от данни). Известно е, че достъпът до външната памет в компютърните системи обикновено става на блокове. Да предположим, че при някои модели твърди дискове един физически сектор съдържа точно 4096 байта. Тогава четенето и записването ще стават на блокове именно с такъв размер. Достъпът до блоковете на твърдия диск обаче, е значително по-бавен от достъпа до оперативната памет, поради което за оптимизиране на системата е необходимо да се минимизира броят сектори, които се прехвърлят между оперативната памет и твърдия диск. Да предположим, че трябва да организираме голямо количество информация, което не се побира в оперативната памет, и е необходимо да се търси елемент от него по зададен ключ. Функцията за търсене извършва следното: прочита някакъв блок и търси в него ключа, който ѝ трябва. Ако го намери — приключва работа, в противен случай прочита друг блок и пак търси. Процесът приключва, ако елементът бъде намерен или се установи, че не съществува. Следователно блоковете, които се прочитат, трябва да носят такава информация, че да бъде необходимо прочитането на минимален брой допълнителни блокове, за да се установи дали търсеният ключ съществува

Ефективното прилагане на B-дърветата при такива системи произтича от следната важна тяхна характеристика: всеки k-връх на B-дърво съдържа между $[m/2]-1$ и $m-1$ елемента. При подходящо избрана стойност на m се получава така, че един връх заема точно един блок памет (в нашия случай един физически сектор). Понеже B-дърветата са идеално балансирани, то височината им е най-много $h = \log_{[m/2]}((n+1)/2)$, което при по-големи стойности на m е достатъчно малко число, дори и при огромно n. Алгоритмите за търсене, добавяне и изтриване на елемент имат сложност от порядъка на $\Theta(h)$ и дори нещо повече — те осъществяват достъп до *не много повече* от h на брой върха от B-дървото.

Трябва да отбележим, че блоковият достъп до паметта е типичен не само за устройствата за съхранение на данни върху външен носител. Така е и при системите с виртуална памет. Системата за управление на релационни бази данни *Oracle* използва модификация на класическите B-дървета за организиране и съхранение на индексите си. Операционната система *Windows* на *Microsoft* също използва модифицирани B-дървета, включително и за организация на файловата система.

Задачи за упражнение:

1. Да се докаже, че B-дърветата са идеално балансирани дървета за претърсване.

2. Да се сравнят *B*-дърветата, червено-черните дървета и дърветата на Фибоначи.

3. Да се помисли как биха могли да се реализират основните операции за работа с *B*-дърво при даденото описание на структурата му.

2.4. Хеш-таблици

Хеш-таблицата е структура от данни, за която е характерен директен достъп до елементите, независимо от типа им. Сложността на елементарните операции по ключ (търсене, вмъкване, изтриване и актуализиране) в общия случай е константна, което я прави изключително полезна в много ситуации. При дефинирането на хеш-таблица като абстрактна структура ще се ограничим до следните три класически операции:

- `void put(key, data);` — включване на елемент
- `data get(key);` — търсене на елемент
- `void remove(key);` — изключване на елемент

Както при разглеждането на списъци и дървета, така и сега, всеки елемент на хеш-таблицата се характеризира с две полета: ключ `key` и данни `data`. Ключът представлява уникален идентификатор: ако два елемента имат един и същ ключ, те се считат за идентични.

Като частен случай на хеш-таблицата може да се разглежда n-елементен масив, в който индексът i ($i = 0,1,...,n-1$) е ключ на i-тия елемент. Вижда се, че два елемента с еднакъв ключ (индекс) ще попадат в един и същи елемент на масива. При това, операциите включване, изтриване и търсене имат константна сложност (поради директния достъп до елементите на масива). На практика, при реализацията на хеш-таблица често се използва масив (т. е. структура с директен достъп до елементите).

Процесът, при който по зададен ключ на елемент се определя неговият адрес с константна сложност, се нарича *хеширане*.

- Хеш-функция

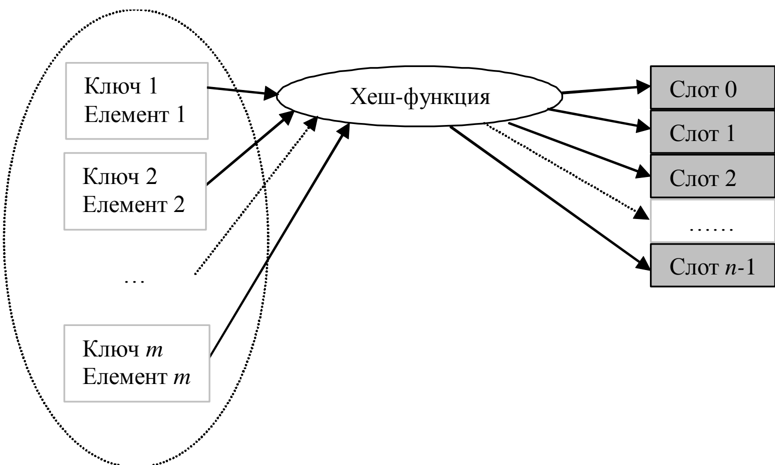
Ще подчертаем още веднъж, че ключът на елемента може да бъде произволна структура, идентифицираща го уникално. Така например, служителите в една компания могат да се идентифицират със своя ЕГН (10-цифрено естествено число, евентуално с водеща нула. Всъщност скорошно проучване показа редица грешки при издаване на ЕГН, включително дублиране, което ги прави ненадежден ключ.) или име (символен низ, но само при допълнителното условие, че няма двама души с едни и същи имена). Ще се спрем по-подробно на първия пример: приемаме, че ключът е 10-цифрено число. Ще използваме масив `a[]` с n елемента (числото n се нарича *капацитет* на хеш-таблицата). Ясно е, че ако броят на служителите е по-малък или равен на n, ще разполагаме с достатъчно памет за записването им в масива `a[]`. За да се достигне обаче константна сложност на включването, трябва да разполагаме с функция (*хеш-функция*), която на ключа на всеки елемент да съпоставя *еднозначно* адрес от 0 до $n-1$. Ако разглеждаме полетата на масива като *слотове* (*виж фигура 1*), номерирани от 0 до $n-1$, в които може да се поставя единствен елемент, то е необходима функция, която да "определя" еднозначно по даден ключ на елемент съответния му слот.

Дефиниция 1. Разглеждаме хеш-таблица с размер n и множество U от допустими стойности за ключовете (*универсум*). Тогава под *хеш-функция* ще разбираме изображението:

$$h : U \rightarrow \{0,1,...,n-1\}$$

Така например, за произволно 10-цифрено число k (ЕГН) може да се получи цяло число от 0 до $n-1$ с помощта на операцията k % n. Остатъкът, получен при делението на k с капацитета на хеш-таблицата, задава една възможна хеш-функция. В случая се забелязва и второто важна

особеност при хеш-таблиците: можем да попаднем в ситуация, при която на елементи с *различен* ключ се съпоставя *един и същ* адрес от масива. Например, при k_1 = 8004104369, k_2 = 8004102469 и n = 100 имаме: k_1 % n = k_2 % n = 69, т. е. елементи с два различни ключа получават един и същ хеш-адрес. Възникването на подобна ситуация се нарича *колизия* и е основен проблем при хеш-таблиците.



Фигура 2.4а. Хеш-функция, съпоставяща на всеки ключ адрес в хеш-таблицата (*хеш-адрес*).

- *Колизии*

Дефиниция 2. Ако за хеш-функция h и два ключа k_1, $k_2 \in U$ е изпълнено $h(k_1) = h(k_2)$, то казваме, че елементите k_1 и k_2 са в *колизия*. Ще казваме още, че са *синоними*.

Не е трудно да се види, че хеш-функцията задава *релация на еквивалентност*: множеството от синоними от един и същ клас, и само те, образуват един *клас на еквивалентност*.

Принцип (*на Дирихле*, *принцип на чекмеджетата*): Дадени са n чекмеджета. Ако сложим в тях n+1 предмета, то в поне една чекмедже ще има поне два предмета.

Колизиите са неизбежни — дори да съставим "перфектна" хеш-функция, колизия гарантирано ще настъпи най-късно в момента, в който броят на включените елементите надхвърли капацитета на хеш-таблицата (съгласно принципа на Дирихле). На практика колизия ще настъпи доста по-рано (като че ли по Закона на Мърфи, но всъщност статистически очаквано). Показателен пример за последното е "примерът с рождените дати": Каква е вероятността измежду 23 души да има двама, родени на една и съща дата (т. е. със съвпадащи ден и месец на раждане)? Оказва се, че тази вероятност е число малко над $\frac{1}{2}$: пресмята се като $1-\prod_{i=2}^{23}\frac{366-i}{365}$ = 0,5063. Как се получава горното равенство? Нека означим с A събитието "***няма*** *двама души с еднаква рождена дата*". Тогава търсената вероятност ще бъде равна на $1-P(A)$. За да пресметнем $P(A)$, ще използваме класическото определение за вероятност: отношение на броя на благоприятните събития към общия брой елементарни събития. В нашия случай благоприятни (по отношение на A) са събитията, при които няма двама души (измежду 23) с еднаква рождена дата, а те са 365.364.363...343 на брой. Същевременно, всевъзможните конфигурации от рождени дати се дават от числото 365^{23}. Оттук лесно следва горната формула. Така, шансовете за печалба ще бъдат на наша страна, ако се обзаложим за последното на някой купон с достатъчен брой гости. Тук сме приели, че годината има 365 дни, т.е. не е високосна.

Така, освен стремежът да се избере добра хеш-функция, т.е. водеща до минимален брой колизии, от първостепенно значение е и въпросът за справяне с тях. В следващите две точки ще разгледаме най-често използваните хеш-функции, както и най-популярните подходи за справяне с колизии. В *2.5.3.* ще приведем и две примерни реализации на хеш-таблица.

2.4.1. Класически хеш-функции

Дали даден избор на хеш-функция е удачен се определя от две неща: хеш-функцията да не отнема много изчислително време и да разпределя елементите относително равномерно в различните хеш-адреси. Ще приемем, че ключовете на елементите са цели числа. Ако не са, винаги можем да им съпоставим такива по някакво правило: Например, ако ключовете са символни низове, можем да сумираме съответните *ASCII*-кодове. Така, за низа "*hello*" ще получим 104+101+108+108+111 = 532. Ако сме сигурни, че низът се състои само от малки латински букви, то можем да изваждаме от *ASCII* кода на всяка буква *ASCII* кода на буквата '*a*'. Така ще избегнем препълване в случая, когато низът е прекалено дълъг. В някои случаи, описаният начин за превръщане от низов ключ в целочислен води до много лоши резултати. Така например, за горната хеш-функция произволна пермутация на символите ще дава един и същ хеш-код; при това кратките символни низове ще попадат в началото на хеш-таблицата. Затова, понякога е удачно при превръщането полученото число да зависи и от *позицията* на символа в низа. Без ограничение на общността можем да считаме, че работим с целочислени ключове: всеки символен низ $a_1a_2\ldots a_n$ можем да разглеждаме като число, записано в b-ична бройна система (където b е броят на различните допустими символи).

Нека е дадена хеш-таблица с капацитет n и елемент с ключ k. По-долу ще приведем най-често срещаните в практиката хеш-функции (върху целочислен ключ).

- Остатък при деление с размера на таблицата

По-горе споменахме този прост и същевременно доста ефективен начин за хеширане — ключът k се разделя целочислено на n и се взема остатъкът от делението. При този подход не е удачно за капацитет на хеш-таблицата да се избира число n, степен на двойката: ако $n = 2^p$, то хеш-кодът на даден ключ k ще бъдат младшите p бита на k. Например, нека $n = 2^4 = 16$ и $k = 173$. Двоичното представяне на 173 е 10101101. Остатъкът 173 % 16 е 13, последното, записано в двоична бройна система, е 1101 — т.е. точно младшите 4 бита на 173. (*виж 1.1.6.*)

Това хеширане ще се държи лошо, ако разпределението на ключовете е такова, че има голям брой числа със съвпадащи младши битове. Изобщо, добре е полученият хеш-код да зависи от *всички* битове на ключа. За тази цел често за капацитет на хеш-таблицата най-често се избира просто число.

- Мултипликативно хеширане

Друг широко разпространен метод е *мултипликативното хеширане*. Избира се реална константа a, $0 < a < 1$. За даден ключ k хеш-функцията има вида:

$$h(k) = \lfloor n.\ \{k.a\} \rfloor$$

Тук с $\{k.a\}$ сме означили дробната част на реалното число, т. е. $k.a - \lfloor k.a \rfloor$.

Въпреки че изборът на константата a (при ограничението $0 < a < 1$) е произволен, за някои стойности практическите резултати са по-добри. Кнут [*Knuth-3/1968*] предлага за a да се използва *златното сечение* (*виж 1.2.2.*):

$$a = \frac{\left(\sqrt{5} - 1\right)}{2} = 0{,}6180339887....$$

- *Хеш-функции върху части от ключа*

- *Извличане на цифри*

При тази схема от ключа се извличат само цифри, намиращи се на определени позиции (например, можем да вземаме първата, третата и петата цифра от числото). Така за ключовете 123569, 425435, 546754, 676576 ще получим хеш-адреси съответно 136, 453, 565, 667. Вижда се, че за конкретния случай n трябва да бъде 1000 (тъй като при извличане на 1, 3 и 5-тата цифра можем да получим число в интервала [0, 999]). Методът има добра успеваемост, когато числата не съдържат много повтарящи се цифри.

- *Сгъване*

Този метод се прилага най-често, когато ключовете са много големи числа. Възможни са разновидности, но като цяло всички се основават на разделяне на ключа на части и извършване на някакви аритметични действия върху получените части. Например, числото може да се раздели на две (или три и повече части) и сумата от получените числа да определя хеш-адреса.

- *Повдигане на средата в квадрат*

Тази схема се основава на извличане на средните p цифри на ключа и повдигането им в квадрат. Например, за ключа 125657134280980 средните три цифри са 134. Повдигаме ги в квадрат: 134.134 = 17956. Ако резултатът надхвърля n, се премахват първите няколко значещи цифри: например, ако $n = 10001$, то от 17956 се премахва първата цифра и полученият хеш-адрес е 7956. Забележете, че последната операция не е еквивалента на намиране на остатъка при деление на n.

- *Сравнение на някои от разгледаните хеш-функции*

За сравнение на изложените методи ще приложим резултатите от един практически тест: Разглеждаме хеш-таблица с големина n и m произволни номера на ЕГН, които ще използваме като ключове (генерирани по такъв начин, че да бъдат валидни и рождената дата да бъде от последното столетие). Целта ни ще бъде да сравним разпределението на хеш-кодовете при използването на всяка една от разгледаните по-горе хеш-функции. Резултатите от експеримента (при $m = 1031000$, $n = 1031$) са показани в *таблица 2.4.1.*, където $\chi^2 = \frac{n}{m}\sum_{i=1}^{n}\left(f_i - \frac{m}{n}\right)^2$, а f_i е броят на ключовете с хеш-код, равен на i.

Хеш-функция	χ^2
Остатък при деление с 1031	729
Извличане на цифри (числото, образувано от последните 3 цифри на ЕГН)	352
Сгъване (разделяне на ЕГН във вида 1-3-3-3 и сума на получените числа)	735
Повдигане в квадрат на числото, образувано от средните три цифри на ЕГН.	233

Таблица 2.4.1. χ^2 сравнение на хеш-функции върху ЕГН, $n = 1031$, $m = 1031000$.

За статистиката χ^2 е известно, че ако хеш-функцията е случайна (т. е. очаква се да работи еднакво "добре" върху произволен набор от ключове, а не само за числа, които са валидни ЕГН) и $m > c.n$, то χ^2 трябва да бъде равна на $n \pm \sqrt{n}$ с вероятност $1-1/c$.

Ясно се вижда, че за избрания пример (ключ ЕГН) и стойности $n = 1031$ и $m = 1031000$ третият и първият метод са най-ефективни, а вторият и четвъртият водят до значително по-неравномерно хеширане.

- Хеш-функции върху символни низове

Символните низове са най-често хешираният тип данни. Същевременно намирането на добра хеш-функция е сериозен проблем. По-долу ще приведем някои от най-често използваните хеш-функции за символни низове, без да претендираме за изчерпателност и без да отегчаваме читателя с излишни детайли. Действието на функциите ще обясним синтезирано на основата на конкретни програмни фрагменти. Преди това ще приведем най-обща схема:

```
result = инициализиране();
while (c = следващ_символ()) {
  result = комбиниране(result, c);
  result = вътрешно_модифициране(result);
}
result = допълнително_модифициране(result);
```

- Адитивно хеширане

Това е най-простото, най-често разпространеното (*класическо*), но за съжаление най-неефективното хеширане. Сумират се *ASCII* кодовете на символите и сумата се взема по модул размера на масива.

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = 0;
  while (*key)
    result += (unsigned char) *key++;
  return result % size;
}
```

Обикновено дължината на низа се включва в сумата, за да може изрично да влияе на хеш-кода:

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = strlen(key);
  while (*key)
    result += (unsigned char) *key++;
  return result % size;
}
```

Ще отбележим, че дължината на символния низ би могла да се подава като параметър, което да спести скъпото обръщение към функцията `strlen()`. Или просто да се получи като разлика между текущата позиция преди и след обхождането на низа (това обаче не винаги ще бъде възможно при следващите хеш-функции, тъй като извършват по-сложни манипулации по `result` в тялото на цикъла):

```
unsigned long hashFunction(const char *key, unsigned long size)
{ const char *saveKey = key;
  unsigned long result = 0;
  while (*key)
    result += (unsigned char) *key++;
  result += saveKey - key;
  return result % size;
}
```

Размерът на масива най-често се избира да бъде просто число. Понякога обаче се използва масив с размер степен на 2, при което последният ред може да се опрости до:

```
    return result & (size - 1);
```

или още по-добре:

```
    hash = (hash ^ (hash>>10) ^ (hash>>20)) & mask;
```

Тази техника е валидна и за функциите по-долу, които съдържат такъв последен ред.

- Ротирано хеширане

Тук няма събиране, а само поразредни операции. Размерът на масива трябва да бъде просто число.

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = strlen(key);
  while (*key)
    result = (result << 4) ^ (result >> 8) ^ ((unsigned char) *key++);
  return result % size;
}
```

- Хеширане едно по едно

Функцията е вариация на горната но с повече манипулации върху променливата `result`, при това извършвани поотделно.

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = 0;
  while (*key) {
    result += (unsigned char) *key++;
    result += result << 10;
    result ^= result >> 6;
  }
  result += result << 3;
  result ^= result >> 11;
  result += result << 15;
  return result % size;
}
```

- Хеширане по Пиърсън

Тук се използва допълнителен масив `tab[]`, съдържащ пермутация на числата от 0 до 255. Забележете, че полученият хеш-код е еднобайтов: от 0 до 255. По-голям код може да се получи, ако функцията се извика няколко пъти с различни масиви `tab[]`, при което всяко извикване ще дава един байт от резултата.

```
unsigned char hashFunction(const char *key, unsigned long size,
                           const unsigned char tab[])
{ unsigned long result = strlen(key);
  while (*key)
    result = tab[result ^ ((unsigned char) *key++)];
  return result;
}
```

- Хеширане по CRC

Тук се изисква стойностите в масива да бъдат генерирани от *Linear Feedback Shift Register*. Без да навлизаме в излишни детайли, ще споменем, че последното е устройство, което генерира псевдослучайна двоична последователност, удовлетворяваща определени условия.

```
unsigned long hashFunction(const char *key, unsigned long size,
```

```
                            const unsigned long tab[])
{ unsigned long result = strlen(key);
  while (*key)
    result = (result<<8) ^ tab[(result>>24) ^ ((unsigned char) *key++)];
  return result % size;
}
```

- Обобщено хеширане по CRC

Същото като предходното, но `tab[]` може да съдържа *произволни* стойности.

- Универсално хеширане

Каквато и хеш-функция да изберем, все едно, в най-лошия случай тя ще предизвиква $\Theta(n)$, колизии където n е размерът на хеш-таблицата. Разбира се, това ще се компенсира при многократно извикване на функцията, тъй като лошите случаи ще се случват рядко.

И все пак какво можем да направим, за да се предпазим от по-чести колизии? Отговорът е: да си гарантираме *равномерно* разпределение на множеството от допустимите стойности на ключовете върху множеството от индексите на масива. Тогава вероятността за съпоставяне на един и същ индекс в масива за два различни низа, т. е. за възникване на колизия, ще бъде $1/n$.

Но как да го постигнем? Една възможност е да комбинираме няколко хеш-функции, които разпределят елементите равномерно. Добре, но колко такива функции да изберем? Нека броят на различните допустими стойности на ключа да бъде m ($m = |U|$), а броят на нужните ни функции — f. Тогава ще искаме $f/m = 1/n$. Т.е. ще са ни нужни m/n на брой различни функции, всяка от които разпределя ключовете равномерно върху индексите на масива.

Дефиниция 2.16. Едно множество $H = \cup_a \{h_a\}$ от хеш-функции се нарича *универсална колекция от хеш-функции*, ако:

- $|H| = m/n$ — съдържа точно m/n на брой функции h_a
- $P(h_a(x) = h_a(y)) = 1/n$ — вероятността за колизия е $1/n$

Един възможен начин да получим универсална колекция хеш-функции е да изберем m/n функции, дефинирани като (a е параметър, различен за всяка функция):

$$h_a(k) = \left(\sum_{i=0}^{r} a_i k_i \right) \bmod n$$

където:

$k = (k_0, k_1, .., k_r)$ е разбиване на ключа k на r+1 части (например байтове)

$a = (a_0, a_1, .., a_r)$, вектор с компоненти, избрани случайно измежду $\{0, 1, .., n-1\}$.

Следва примерна реализация. Масивът `tab[]` съдържа толкова елемента, колкото е максималният брой битове във входния низ, избрани случайно измежду $\{0, 1, .., n-1\}$. Подробностите оставяме на читателя.

```
unsigned long hashFunction(const char *key, unsigned long size,
                           const unsigned long tab[MAXBITS])
{ unsigned char k;
  unsigned i;
  unsigned long result;
  unsigned long l3 = (result = strlen(key)) << 3;
  for (i = 0; i < l3; i += 8)
  {
    k = (unsigned char) key[i >> 3];
    if (k&0x01) result ^= tab[i+0];
    if (k&0x02) result ^= tab[i+1];
    if (k&0x04) result ^= tab[i+2];
    if (k&0x08) result ^= tab[i+3];
```

```
    if (k&0x10) result ^= tab[i+4];
    if (k&0x20) result ^= tab[i+5];
    if (k&0x40) result ^= tab[i+6];
    if (k&0x80) result ^= tab[i+7];
  }
  return result % size;
}
```

- Хеш-функция на Зобрист

Тук таблицата `tab[][]` е двумерна, като стойностите отново са избрани случайно измежду $\{0, 1, .., n-1\}$. При внимателен подбор може да се получи универсално хеширане.

```
unsigned long hashFunction(const char *key, unsigned long size,
                           const unsigned long tab[MAXBYTES][256])
{ unsigned i;
  unsigned long result = strlen(key);
  for (i = 0; i < len; i++)
    result ^= tab[i][*key++]
  return result % size;
}
```

Задачи за упражнение:

1. Да се сравнят разгледаните по-горе хеширащи функции по отношение на:
   - изчислителна сложност
   - равномерност на разпределението на ключовете
2. От какви проблеми страдат хеш-функциите върху части от ключа?
3. Какви предимства и недостатъци виждате за всяка от разгледаните хеш-функции за символни низове?
4. Винаги ли хешираща функция, която разпределя елементите по-равномерно, е по-добра?
5. Да се сравнят универсалното хеширане и хеш-функцията на Зобрист.
6. Дадена е колекция от m/n на брой хеш-функции, дефинирани като (a е параметър, различен за всяка функция):

$$h_a(k) = \left(\sum_{i=0}^{r} a_i k_i \right) \bmod n$$

където:

$k = (k_0, k_1, .., k_r)$ е разбиване на ключа x на r+1 части (например байта)

$a = (a_0, a_1, .., a_r)$, вектор с компоненти, избрани случайно измежду $\{0, 1, .., n-1\}$.

n — размер на хеш-таблицата

m — мощност на множеството на допустимите стойности на ключа

Като се изхожда от *дефиниция 2.16.*, да се докаже, че това е универсална колекция хеш-функции.

7. Да се допълни *таблица 2.4.1.* с мултипликативно хеширане.
8. Да се помисли каква комбинация (или модификация) на разгледаните методи е възможно да се приложи, за да се получат още по-добри резултати (върху произволно избран набор от ЕГН, n и m)? Може ли да се очаква χ^2 да бъде $n \pm \sqrt{n}$, ако се състави хеш-функция, работеща оптимално за конкретната задача с хеширане на ЕГН?
9. Да се повтори описаният експеримент за различни стойности на n и m. Потвърждават ли се горните резултати? При генериране на тестови данни може да се използва алгоритъмът за проверка за коректност на номера на ЕГН от *задача 1.203*.
10. Съгласни ли сте с начина на набавяне на данни за теста за попълване на данните от *таблица 2.4.1.* (генериране с функция за псевдослучайни числа, но с какво разпределение...)? Какво влияние оказва това върху крайния резултат? Можете ли да предложите решение на проблема?

2.4.2. Справяне с колизии

Различните схеми за справяне с колизиите ще илюстрираме с конкретен пример. Нека е дадена хеш-таблица с капацитет $n = 10$. Ще използваме проста хеш-функция, която се основава на *извличане на цифри* — първата цифра на ключа ще бъде нашият хеш-адрес. В примера, елементите, които ще искаме да включим, са с ключове 234, 235, 567, 123, 534 и 647.

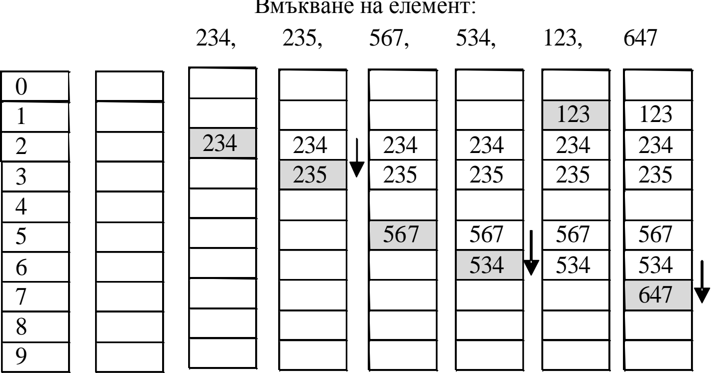
- Затворено хеширане

- Линейно пробване

Линейното пробване, *квадратичното пробване* и *двойното хеширане* са част от по-обща схема за разрешаване на колизиите, наречена *затворено хеширане*. При нея разполагаме с едно основно място, където се съхраняват данните (n "слота", за разлика от реализациите с допълнителна част за колизиите). Когато настъпи колизия, пробваме последователно да променяме получения хеш-адрес, докато достигнем до свободен "слот". Промяната се прави по-някаква предварително дефинирана схема.

Например, възможно е да увеличаваме адреса с някакво естествено число s ($0 < s < n$). Ако адресът стане равен на n, продължаваме търсенето в началото на хеш-таблицата, като вземаме остатъка по модул n. За да може при подобно увеличаване да се обходят *всички* адреси на хеш-таблицата, трябва n и s да бъдат взаимно прости, т. е. $НОД(n, s) = 1$.

Очевидно, при този подход не могат да се включат повече елементи, отколкото е капацитетът на хеш-таблицата (*виж фигура 2.4.2а.*) и когато тя се препълни, е необходимо разширяване (заделяне на памет за по-голям масив и освобождаване на заетата от текущия масив памет). Как се реализира това на практика ще видим при втората реализация на хеш-таблица по-долу (*виж 2.5.3*).



Фигура 2.4.2а. Последователно включване на елемент. Разрешаване на колизиите с линейно пробване със стъпка 1.

При търсенето на елемент по ключ се процедира по аналогичен начин — първо се изчислява хеш-адресът с помощта на хеш-функцията и в случай, че не се попадне на търсения ключ, се прави линейно пробване (променяне на адреса), докато се достигне търсеният елемент. Последното предполага, че ключът е в хеш-таблицата.

Широко разпространен е случаят $s = 1$: при колизия се преминава към следващия адрес и т.н. със стъпка 1. В този случай, както и в по-общия, е възможно да се попадне на ключове, при които хеширането да се окаже неефективно. Например, когато елементите образуват групи с близки хеш-кодове (т. нар. "клъстери"). Тогава в процеса на разрешаване на колизия възникват

серия *нови (вторични) колизии* (*виж фигура 2.4.2а.*). По-долу ще разгледаме две други важни разновидности на затвореното хеширане.

- Квадратично пробване

При *квадратичното пробване*, както подсказва и самото име, се използва стъпка от вида $c_1i + c_2i^2$, $c_2 \neq 0$. Отместването зависи квадратично от поредния номер на пробата i. Този метод работи значително по-ефективно от линейното пробване, въпреки, че и при него се забелязват негативните ефекти в случаите на клъстер от ключове (вторична струпване на ключове). Основният му проблем обаче е, че ***не*** гарантира обхождане на цялата таблица, т.е. възможно е да не намери свободно място, макар да има такова. [*Cormen, Leiserson, Rivest-1997*].

- Двойно хеширане

Двойното хеширане, е подобно на разгледаните два метода, като тук се използват две хеш-функции:

$$h(k, i) = h_1(k) + i.h_2(k)$$

Тук i е номерът на пробата след първата колизия. Втората хеш-функция се използва само в случаите, когато резултатът от първата води до колизия. Като ключ при повторното хеширане може да се използва както оригиналният ключ на елемента, така и хеш-адресът, получен при прилагане на първата хеш-функция.

- Отворено хеширане

- Допълнителна памет за колизии

Както подсказва самото име, при тази схема се отделя *допълнителна памет* за обработване на колизии. Всеки елемент, попаднал в колизия, се разполага на първото свободно място в допълнително отделената памет (*виж фигура 2.4.2б.*).

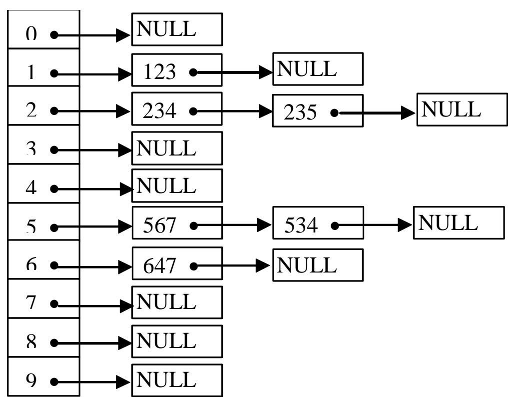
Вмъкване на елемент:

	234,	235,	567,	534,	123,	647
0						
1					123	123
2	234	234	234	234	234	234
3						
4						
5			567	567	567	567
6						647
7						
8						
9						
10		235	235	235	235	235
11				534	534	534
…	…	…	…	…	…	…

Фигура 2.4.2б. Разрешаване на колизиите чрез заделяне на допълнителна памет.

При търсене в хеш-таблицата първо се проверява адресът, получен от хеш-функцията, и, ако елементът с търсения ключ не е там, се претърсва *цялата* допълнителна част.

- Списък на препълванията



Фигура 2.4.2в. Динамично свързване: хеш-таблицата, след включване на елементите 234, 235, 567, 123, 534, 647.

Възможно е да се използва и външна *динамично заделяна* памет, когато хеш-таблицата се запълни: Елементите със съвпадащи хеш-адреси се разполагат в някаква стандартна структура от данни. Най-често използваната, но не непременно най-ефективната за тази цел, структура е свързан списък. Затова този подход за справяне с колизии често се нарича *списък на препълванията*: всеки слот на хеш-таблицата е указател към динамичен свързан списък, съдържащ само членове на съответния клас синоними. При включване на нов елемент той се добавя в началото на списъка, намиращ се на слота, определен от хеш-функцията (*виж фигура 2.4.2в.*). Така по естествен начин отпада грижата за колизиите. Аналогично, при търсене се преглежда списъкът, определен от хеш-функцията. Сложността на операцията търсене тогава зависи от броя на елементите, с които даденият е в колизия, т. е. от дължината на съответния динамичен свързан списък.

Елементите в колизия могат да се пазят в структура, в която търсенето е с по-малка алгоритмична сложност от тази на линейния списък — например двоично дърво за търсене. Така всяко поле от хеш-таблицата ще бъде указател към корена на отделно двоично дърво.

Задачи за упражнение:

1. Да се сравнят отвореното и затвореното хеширане: предимства и недостатъци. Кога следва да се предпочита всяко от тях?
2. Защо квадратичното пробване не може да гарантира пълно обхождане на масива? Вярно ли е това за всички квадратни функции?

2.4.3. Реализации на хеш-таблица

Ще разгледаме две различни реализации на хеш-таблица. В първата ключът на хеш-таблиците е целочислен (това е важно при реализацията на хеш-функцията, което ще бъде на принципа на деление с остатък). С колизиите ще се справяме, като използваме динамичен свързан списък. Така ще илюстрираме как бързо и лесно може да се построи ефективна структура от данни (обърнете внимание, че като изключим реализацията на свързан списък, допълнителният код за хеш-таблицата е съвсем кратък).

Капацитетът на хеш-таблицата `N` се задава в началото на програмата като макрос. Подробностите по реализацията могат да се намерят като коментари в изходния код.

```
#include <stdio.h>
```

```
#include <stdlib.h>

#define N 211

typedef int data;
typedef long keyType;

#define NOT_EXIST (-1) /* връща се от get(), когато елементът липсва */

struct list {
  keyType key;
  data info;
  struct list *next;
};

struct list *hashTable[N];

/* включва елемент в началото на свързан списък */
void insertBegin(struct list **L, keyType key, data x)
{
  struct list *temp;
  temp = (struct list *) malloc(sizeof(*temp));
  if (NULL == temp) {
    fprintf(stderr, "Няма достатъчно памет за нов елемент!\n");
    return;
  }

  temp->next = *L;
  (*L) = temp;
  (*L)->key = key;
  (*L)->info = x;
}

/* изтрива елемент от списъка */
void deleteNode(struct list **L, keyType key)
{ struct list *current = *L;
  struct list *save;
  if ((*L)->key == key) {  /* трябва да се изтрие първият елемент */
    current = (*L)->next;
    free(*L);
    (*L) = current;
    return;
  }

  /* намира елемента, който ще се трие */
  while (current->next != NULL && current->next->key != key)
    current = current->next;
  if (NULL == current->next) {
    fprintf(stderr, "Грешка: Елементът за изтриване не е намерен! \n");
    return;
  }
  else {
    save = current->next;
    current->next = current->next->next;
    free(save);
  }
}

/* търси по ключ елемент в свързан списък */
```

```
struct list* search(struct list *L, keyType key)
{ while (L != NULL) {
    if (L->key == key) return L;
    L = L->next;
  }
  return NULL;
}

unsigned hashFunction(keyType key)
{ return(key % N); }

void initHashTable(void)
{ unsigned i;
  for (i = 0; i < N; i++) hashTable[i] = NULL;
}

void put(keyType key, data x)
{ int place = hashFunction(key);
  insertBegin(&hashTable[place], key, x);
}

data get(keyType key)
{ int place = hashFunction(key);
  struct list *l = search(hashTable[place], key);
  return (NULL != l) ? l->info : NOT_EXIST;
}

int main(void) {
  initHashTable();
  put(1234, 100);        /* -> в слот 179 */
  put(1774, 120);        /* -> в слот 86 */
  put(86, 180);          /* -> в слот 86 -> колизия */
  printf("Отпечатва данните на елемента с ключ   86: %d \n", get(86));
  printf("Отпечатва данните на елемента с ключ 1234: %d \n", get(1234));
  printf("Отпечатва данните на елемента с ключ 1774: %d \n", get(1774));
  printf("Отпечатва данните на елемента с ключ 1773: %d \n", get(1773));
  return 0;
}
```

🖫 hash.c

Резултат от изпълнението на програмата:

```
Отпечатва данните за елемента с ключ   86: 180
Отпечатва данните за елемента с ключ 1234: 100
Отпечатва данните за елемента с ключ 1774: 120
Отпечатва данните за елемента с ключ 1773: -1
```

Като втори пример ще разгледаме една конкретна практическа задача, която се решава ефективно, като се използва хеш-таблица. При тази реализация ще се спрем на още някои специфични проблеми: хеширане на символен низ, разширяване на хеш-таблицата, когато предварително заделеният масив се окаже недостатъчен и др.

Задача: Даден е текст (списък от думи), разделени с един интервал. Да се намери честотата на срещане на всяка дума от текста.

Описаната задача възниква често като част от по-сложни практически проблеми. Така например, анализирането на документ от търсеща машина (напр. *Google*), преди каквато и да е понататъшна обработка, почти задължително се състои именно в намиране на броя на срещанията на всяка дума от документа.

За да бъде решена задачата ефективно с хеш-таблица, следва да се обърне внимание на няколко основни неща:

- Ключът на хеш-таблицата ще бъде *символен низ* — това е дума от документа. Допълнително ще се пази честотата на срещане на думата. При някои задачи е възможно да няма никакви допълнителни данни. Тогава структурата се нарича *хеш-множество*.
- Колизиите ще разрешаваме с линейно пробване, като размерът на хеш-таблицата n ще бъде степен на 2. Забележете, че в предложената реализация позицията на символа има значение, т. е. *abc* и *bca* ще имат различен хеш-код. При разглеждането на хеш-функция, основана на остатък при деление с просто число в *2.5.1.*, показахме с конкретен пример, че това не е ефективен подход, тъй като хеш-кодът не зависи от всички битове на ключа. В случая това обаче не е сериозен проблем, тъй като ключовете на практика са символни низове и се очаква превръщането им до число, преди операцията остатък по модул, да осигури по-равномерното разпределение в младшите битове. Предимството при използване на хеш-таблица с такъв размер е възможността за използване на по-бързата операция — побитово "и", вместо целочислено деление с остатък. Стъпката s ($s > 2$), с която ще извършваме пробването, ще бъде просто число: така гарантираме, че е изпълнено свойството n и s да бъдат взаимно прости.
- Ако предварително определената големина за хеш-таблицата се окаже недостатъчна, ще заделяме памет за още елементи: ще удвояваме големината ѝ и по този начин ще запазваме свойството на капацитета ѝ да остава степен на 2.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define S 107              /* стъпка на увеличаване при колизия */
#define MAX_FILL_LEVEL 0.8 /* Максимално ниво на запълване */
unsigned long n = 32;      /* начален размер на хеш-таблицата */

struct singleWord {
  char *word;              /* ключ - символен низ */
  unsigned long freq;      /* честота на срещане на думата */
} *ht;

unsigned long count;

/* Хеш-функция за символен низ */
unsigned long hashFunction(const char *key)
{ unsigned long result = 0;
  while (*key)
    result += result + (unsigned char) *key++;
  return result & (n - 1);
}

/* Инициализиране на хеш-таблицата */
void initHashtable(void)
{ unsigned long i;
  count = 0;
  ht = (struct singleWord *) malloc(sizeof(*ht)*n);
  for (i = 0; i < n; i++)
    ht[i].word = NULL;
}

/* Търсене в хеш-таблицата: връща 1 при успех, и 0 - иначе */
/* При успех: *ind съдържа индекса на намерения елемент */
```

```
/* При неуспех: свободна позиция, където може да бъде вмъкнат */
char get(const char *key, unsigned long *ind)
{ unsigned long k;
  *ind = hashFunction(key);
  k = *ind;
  do {
    if (NULL == ht[*ind].word) return 0;
    if (0 == strcmp(key, ht[*ind].word)) return 1;
    *ind = (*ind + S) & (n - 1);
  } while (*ind != k);
  return 0;
}

/* Разширяване на хеш-таблицата */
void resize(void)
{ unsigned long ind, hashInd;
  struct singleWord *oldHashTable;

  /* 1. Запазване на указател към хеш-таблицата */
  oldHashTable = ht;

  /* 2. Двойно разширяване */
  n <<= 1;

  /* 3. Заделяне на памет за новата хеш-таблица */
  ht = (struct singleWord *) malloc(sizeof(*ht)*n);
  for (ind = 0; ind < n; ind++)
    ht[ind].word = NULL;

  /* 4. Преместване на записите в новата хеш-таблица */
  for (ind = 0; ind < (n >> 1); ind++) {
    if (oldHashTable[ind].word != NULL) {
      /* Премества записа на новото място */
      if (!get(oldHashTable[ind].word, &hashInd))
        ht[hashInd] = oldHashTable[ind];
      else
        printf("Грешка при разширяване на хеш-таблицата!\n");
    }
  }

  /* 5. Освобождаване на старата памет */
  free(oldHashTable);
}

/* Добавяне на елемент в хеш-таблицата */
void put(char *key)
{ unsigned long ind;
  if (!get(key, &ind)) { /* Думата не е в хеш-таблицата */
    ht[ind].word = strdup(key);
    ht[ind].freq = 1;
    if (++count > ((unsigned long) n * MAX_FILL_LEVEL)) resize();
  }
  else
    ht[ind].freq++;
}

/* Отпечатване на хеш-таблица */
void printAll(void)
{ unsigned long ind;
```

```
  for (ind = 0; ind < n; ind++)
    if (ht[ind].word != NULL)
        printf("%s %ld \n", ht[ind].word, ht[ind].freq);
}

/* Унищожаване на хеш-таблица */
void destroy(void)
{ unsigned long ind;
  for (ind = 0; ind < n; ind++)
    if (ht[ind].word != NULL) free(ht[ind].word);
  free(ht);
}

int main(void) {
  unsigned long find;
  initHashtable();

  put("reload");
  put("crush tour");
  put("room service");
  put("load");
  put("reload");
  put("reload");

  printAll();

  if (get("reload", &find))
    printf("Честота на думата 'reload': %d \n", ht[find].freq);
  else
        printf("Думата 'reload' липсва!");

  if (get("download", &find))
    printf("Честота на думата 'download': %d \n", ht[find].freq);
  else
        printf("Думата 'download' липсва!");

  destroy();
  return 0;
}
```
🖫 hashset.c

Резултат от изпълнението на програмата:

```
load 1
reload 3
crush tour 1
room service 1
Честота на думата 'reload': 3
Думата 'download' липсва!
```

Задачи за упражнение:

1. Да се реализира функция за изключване на елемент от хеш-таблица по даден ключ. Сложността трябва да бъде константна. Да се обърне внимание, че не е достатъчно единствено да се намери елементът със зададения ключ и да се изтрие, както при реализацията със списък на препълване: това би довело до *скъсване на веригата* от елементи с еднакъв хеш-код, т.е. на *веригата от синоними*.

2. Възможно ли е да се модифицира алгоритъмът за разширяване на хеш-таблица от *2.4.3.* (реализацията на принципа на линейно пробване) така, че да не бъде необходимо преместване на всеки един от елементите (рехеширане), а да се извършва просто копиране на памет?

3. Да се реализира метод за "свиване" на хеш-таблица: намаляване на размера ѝ, когато броят на елементите ѝ намалее. Какъв би могъл да бъде критерият за намаляване размера на хеш-таблицата? Ускоряване или забавяне следва да се очаква след сгъстяването? Защо?

4. В предложената по-горе програма не се допуска запълване на хеш-таблицата над 80%: при достигане на това ниво тя автоматично се разширява. Изследванията показват, че при такова ниво на запълване и подходящо избрана хеш-функция средният брой проби е по-малко от 2. Да се оцени експериментално зависимостта между средния брой проби и степента на запълване на хеш-таблицата. Какъв е средният брой проби при запълване: 85%, 90%, 93%, 95%, 97%, 98%, 99%, 100%?

5. В предложената по-горе реализация стойността на хеш-функцията отчита позицията на символите:

```
result += result + (unsigned char) *key++;
```

Да се направи сравнение с класическата хеш-функция за символни низове:

```
result = result + (unsigned char) *key++;
```

Да се оцени средният брой проби, необходими за откриване на елемент с даден ключ или установяване, че липсва при всяка от двете хеш-функции.

6. В предложената по-горе реализация използвахме затворено хеширане с линейни проби. Да се експериментира с други стандартни стратегии за справяне с колизии като квадратични проби и двойно хеширане при подходящо избрани функции. Намалява или се увеличава средният брой проби? Зависи ли съотношението между средния брой проби при различните стратегии за справяне с колизии от степента на запълване на хеш-таблицата? А от размера й? Добър избор ли е степен на 2?

7. Да се тества предложената по-горе реализация върху различни входни данни. Променя ли се средният брой проби при търсене?

2.5. Въпроси и задачи

2.5.1. Задачи от текста

Задача 2.1.

Защо не е възможна реализация, при която и четирите основни операции върху линеен списък да бъдат еднакво ефективни, например да имат константна сложност? (*виж 2.1.*)

Задача 2.2.

Да се преработят програмите от *2.1.1.* така, че функциите да не печатат съобщения за грешка, а да връщат булева стойност дали са успели.

Задача 2.3.

Да се реализира “полустатична” версия на програмите от *2.1.1.*: ако стекът/опашката се напълни, да се извършва автоматично разширяване на масива.

Задача 2.4.

Да се реализират основните функции за работа с дек. (*виж 2.1.1.*)

Задача 2.5.

Да се преработят програмите от *2.1.2.* така, че функциите да не печатат съобщения за грешка, а да връщат булева стойност дали са успели.

Задача 2.6.

Структурите от данни стек, опашка, дек и т.н. могат да се разглеждат като специални линейни списъци и също могат да бъдат реализирани динамично. Предоставяме на читателя, като използва реализацията на линеен едносвързан списък от *2.1.2.*, да се опита да я модифицира до динамична реализация на стек и опашка. Важно е сложността на основните операции (включване, изключване) да запазят константната си сложност.

Задача 2.7.

Да се реализира комбинирана стратегия за представяне на стек в паметта, при която се използва *списък от масиви*. В началото се започва с единствен масив и, когато той се напълни, се заделя нов блок, който се свързва с указател към предишния. Когато последно включеният масив се изпразни, той се изключва от списъка. Какви са предимствата и недостатъците на предложения подход? Бихте ли го препоръчали и за опашка? А за дек? (*виж 2.1.2.*)

Задача 2.8.

Да се построи наредено двоично дърво за търсене чрез последователно добавяне на следните върхове:

а) 7, 14, 28, 35, 65, 12, 18, 42, 81, 64, 61, 4, 13
б) 12, 7, 14, 81, 42, 18, 61, 4, 64, 35, 13, 28, 65
в) 4, 7, 12, 13, 14, 18, 28, 35, 42, 61, 64, 65, 81
г) 81, 65, 64, 61, 42, 35, 28, 18, 14, 13, 12, 7, 4
д) 28, 64, 13, 42, 7, 81, 61, 4, 12, 65, 35, 18, 14

Да се сравнят получените дървета. Какви изводи могат да се направят? (*виж 2.2.*)

Задача 2.9.

Да се изтрият в този ред върховете 8, 13, 5 и 6 от дървото на *фигура 2.2к*.

Задача 2.10.

Даден е изразът `(A+B-C)*(D/F)-G*H`. Да се намери неговият обратен полски запис. (*виж 2.2.*)

Задача 2.11.

Да се докаже, че между всеки два върха в дърво съществува единствен път. (*виж 2.2.*)

Задача 2.12.

Каква най-малка част от "дървото" на *фигура 2.2з.* трябва да се отсече така, че примерът да отговаря на *дефиниция 2.2.*?

Задача 2.13.

Да се напише итеративен вариант на функцията за обхождане на двоично дърво. (*виж 2.2.*)

Задача 2.14.

Да се напише вариант на функцията за обхождане на двоично дърво, която да го отпечатва така, че да се вижда *структурата* му. (*виж 2.2.*)

Задача 2.15.

Да се докаже коректността на описания в *2.2.* алгоритъм за изтриване на връх от наредено двоично дърво.

Задача 2.16.

Описаният в *2.2.* алгоритъм за изтриване на връх от наредено двоично дърво в случай на два наследника търси *най-левия наследник на дясното поддърво*. Да се реализира вариант, който търси най-десния наследник на лявото поддърво. Съществуват ли други възможности?

Задача 2.17.

Да се предложи и реализира *статично* представяне на двоично дърво. (*виж 2.2.*)

Задача 2.18.

В *2.2.* видяхме, че стандартен начин за представяне на израз в двоично дърво е в листата да се записват променливи и константи, а във върховете — операции. Забележете, че позволявахме само *двуаргументни* операции (като +, –, *, /), при което левият наследник съдържа първия аргумент, а десният — втория. Възможно ли е да позволим и унарни (едноаргументни) операции като: ‘+’ и ‘–’? Какво се променя?

Задача 2.19.

Да се построи *идеално балансирано* двоично дърво за търсене за следното множество от върхове {4, 7, 12, 13, 14, 18, 28, 35, 42, 61, 64, 65, 81}. (*виж 2.3.*)

Задача 2.20.

Балансирано ли е дървото от (*виж 2.3.*):

а) *фигура 2.2г.*
б) *фигура 2.2и.*
в) *фигура 2.2к.*

Задача 2.21.

Идеално балансирано ли е дървото от (*виж 2.3.*):

а) *фигура 2.2г.*
б) *фигура 2.2и.*
в) *фигура 2.2к.*

Задача 2.22.

Като се използва *дефиниция 2.10.*, да се докаже, че всяко нелисто в червено-черно дърво има точно два наследника. (*виж 2.3.1.*)

Задача 2.23.

Да се помисли как биха могли да се реализират основните операции за работа с червено-черни дървета при даденото описание на структурата им. (*виж 2.3.1.*)

Задача 2.24.

Да се докаже, че *B*-дърветата са идеално балансирани дървета за претърсване. (*виж 2.3.2.*)

Задача 2.25.

Да се сравнят *B*-дърветата, червено-черните дървета и дърветата на Фибоначи. (*виж 2.3.*)

Задача 2.26.

Да се помисли как биха могли да се реализират основните операции за работа с *B*-дърво при даденото описание на структурата му. (*виж 2.3.2.*)

Задача 2.27.

Да се сравнят разгледаните в *2.4.1.* хеширащи функции по отношение на:

- изчислителна сложност
- равномерност на разпределението на ключовете

Задача 2.28.

От какви проблеми страдат хеш-функциите върху части от ключа? (*виж 2.4.1.*)

Задача 2.29.

Какви предимства и недостатъци виждате за всяка от разгледаните в *2.4.1.* хеш-функции за символни низове?

Задача 2.30.

Винаги ли хешираща функция, която разпределя елементите по-равномерно, е по-добра? (*виж 2.4.1.*)

Задача 2.31.

Да се сравнят универсалното хеширане и хеш-функцията на Зобрист (*виж 2.4.1.*).

Задача 2.32.

Дадена е колекция от m/n на брой хеш-функции, дефинирани като (a е параметър, различен за всяка функция):

$$h_a(k) = \left(\sum_{i=0}^{r} a_i x_i \right) \bmod n$$

където:

$k = (k_0, k_1, .., k_r)$ е разбиване на ключа x на r+1 части (например байта)

$a = (a_0, a_1, .., a_r)$, вектор с компоненти, избрани случайно измежду $\{0, 1, .., n-1\}$.

n — размер на хеш-таблицата

m — мощност на множеството на допустимите стойности на ключа

Като се изхожда от *дефиниция 2.16.*, да се докаже, че това е универсална колекция хеш-функции.

Задача 2.33.

Да се допълни *таблица 2.4.1.* с мултипликативно хеширане.

Задача 2.34.

Да се помисли каква комбинация (или модификация) на разгледаните в *2.4.1.* методи е възможно да се приложи, за да се получат още по-добри резултати върху произволно избран набор от ЕГН, n и m? Може ли да се очаква χ^2 да бъде $n \pm \sqrt{n}$, ако се състави хеш-функция, работеща оптимално за конкретната задача с хеширане на ЕГН номера?

Задача 2.35.

Да се повтори описаният в *2.4.1.* експеримент за различни стойности на n и m. Потвърждават ли се резултатите? При генериране на тестови данни може да се използва алгоритъмът за проверка за коректност на номера на ЕГН от *задача 1.203.*

Задача 2.36.

Съгласни ли сте с начина на набавяне на данни за теста за попълване на данните от *таблица 2.4.1.* (генериране с функция за псевдослучайни числа, но с какво разпределение...)? Какво влияние оказва това върху крайния резултат? Можете ли да предложите решение на проблема?

Задача 2.37.

Да се сравнят отвореното и затвореното хеширане: предимства и недостатъци. Кога следва да се предпочита всяко от тях? (*виж 2.4.2.*)

Задача 2.38.

Защо квадратичното пробване не може да гарантира пълно обхождане на масива? Вярно ли е това за всички квадратни функции? (*виж 2.4.2.*)

Задача 2.39.

Да се реализира функция за изключване на елемент от хеш-таблица по даден ключ. Сложността трябва да бъде константна. Да се обърне внимание, че не е достатъчно единствено да се намери елементът със зададения ключ и да се изтрие, както при реализацията със списък на препълване: това би довело до *скъсване на веригата* от елементи с еднакъв хеш-код, т.е. на *веригата от синоними*. (*виж 2.4.*)

Задача 2.40.

Възможно ли е да се модифицира алгоритъмът за разширяване на хеш-таблица от *2.4.3.* (реализацията на принципа на линейно пробване) така, че да не бъде необходимо преместване на всеки един от елементите (рехеширане), а да се извършва просто копиране на памет?

Задача 2.41.

Да се реализира метод за "свиване" на хеш-таблица: намаляване на размера ѝ, когато броят на елементите ѝ намалее. Какъв би могъл да бъде критерият за намаляване размера на хеш-таблицата? Ускоряване или забавяне очаквате след сгъстяването? Защо? (*виж 2.4.3.*)

Задача 2.42.

В предложената в *2.4.3.* програма не се допуска запълване на хеш-таблицата над 80%: при достигане на това ниво тя автоматично се разширява. Изследванията показват, че при такова ниво на запълване и подходящо избрана хеш-функция средният брой проби е по-малко от 2. Да се оцени експериментално зависимостта между средния брой проби и степента на запълване на хеш-таблицата. Какъв е средният брой проби при запълване: 85%, 90%, 93%, 95%, 97%, 98%, 99%, 100%?

Задача 2.43.

В предложената в *2.4.3.* реализация стойността на хеш-функцията отчита позицията на символите:

```
result += result + (unsigned char) *key++;
```

Да се направи сравнение с класическата хеш-функция за символни низове:

```
result = result + (unsigned char) *key++;
```

Да се оцени средният брой проби, необходими за откриване на елемент с даден ключ или установяване, че липсва, при всяка от двете хеш-функции.

Задача 2.44.

В предложената в *2.4.3.* реализация използвахме затворено хеширане с линейни проби. Да се експериментира с други стандартни стратегии за справяне с колизии като квадратични проби и двойно хеширане при подходящо избрани функции. Намалява или се увеличава средният брой проби? Зависи ли съотношението между средния брой проби при различните стратегии за справяне с колизии от степента на запълване на хеш-таблицата? А от размера й? Добър избор ли е степен на 2?

Задача 2.45.

Да се тества предложената в *2.4.3.* реализация върху различни входни данни. Променя ли се средният брой проби при търсене?

2.5.2. Други задачи

Задача 2.46. *Двусвързан списък*

Да се състави динамична реализация на *двусвързан линеен списък*. Двусвързан линеен списък се дефинира рекурсивно по следния начин:

```
typedef int data;
typedef int keyType;

struct list {
  keyType key;
  data info;
  struct list *prev;
  struct list *next;
};
```

Всеки елемент в двусвързан списък има два указателя — към следващ елемент (както при едносвързан списък) и към предхождащ елемент (указателят `prev`).

Да се реализират основните операции за работа с линеен списък — включване на елемент (преди и след елемент, сочен от даден указател), както и да се реализира (с константна сложност) операцията изтриване на елемент, сочен от даден указател.

Задача 2.47. *Цикличен списък*

Цикличен едносвързан списък се нарича линеен едносвързан списък, в който указателят към следващ елемент `next` на последния елемент сочи първия елемент в списъка.

Да се състави динамична реализация на цикличен едносвързан списък и да се реализират основните операции за работа с него.

Задача 2.48. *Приоритетна опашка*

Приоритетната опашка намира много приложения (кеширане и др.). Всеки елемент, освен данните, които съдържа, се характеризира и с едно цяло число — неговият "приоритет". Когато се включва елемент в опашката, това става не в края ѝ, а веднага след последния елемент с приоритет, по-голям от неговия (ако елементите са равни, наредбата е по реда на постъпване). Така всички елементи с по-голям приоритет се намират преди него, а елементите с по-малък приоритет — след него в опашката. Изключването от опашката става от началото ѝ — там винаги се намира елементът с най-висок приоритет.

Да се напише програма, която симулира приоритетна опашка и реализира основните операции включване и изключване със сложност $\Theta(\log_2 n)$ [*Наков-1998*][*Уирт-1980*].

Забележка: Структурата от данни *пирамида*, е разгледана в *3.1.9. — Пирамидално сортиране на Уилямс*. Освен за ефективна реализация на приоритетна опашка, пирамидата се прилага успешно за понижаване на сложността в много други алгоритми — алгоритъм на Дейкстра (*виж 5.4.2.*), и др.

Задача 2.49. *"Отборна"опашка*

Една малко известна разновидност на опашка, която обаче често се среща във всекидневието, е т.нар. "*отборна*" *опашка*. Така например, опашката, която се формира пред студентския стол е "отборна" опашка. Когато нов елемент се присъединява към опашката, той първо претърсва за свои "познати" в нея, и, ако намери, се включва точно след тях. Ако не намери, новият елемент е нямал късмет и застава накрая на опашката. Изключването на елемент от опашката става по стандартния начин — само от началото на опашката. Да се напише програма, която по дадени групи с елементи (елементите, принадлежащи на една група считаме за "познати"), симулира "отборна" опашка.

Задача 2.50. *Обратен полски запис*

Да се напише програма, която пресмята стойността на израз, записан в обратен полски запис. За решаването на задачата да се използва стек [*Наков-1998*].

Задача 2.51. *Възстановяване на израз*

Да се напише програма, която от зададен аритметичен израз, записан в обратен полски запис, възстановява класическия алгебричен запис на израза със скоби. Да се реши задачата по два начина: с и без използване на двоично дърво [*Наков-1998*].

Задача 2.52. *Бърз синтактичен анализ*

Да се напише програма, която прочита изходния код на програма на Си и проверява дали двойките `(` и `)`; `{` и `}`; `[` и `]`; `/*` и `*/` участват симетрично в него.

Упътване: Да се използва стек.

Задача 2.53. *Симулация на рекурсия*

Когато се изпълняват рекурсивни извиквания, локалните данни за всяко поредно извикване се разполагат в паметта на принципа на стека (на ниво компилатор/операционна система реализацията на този процес е точно такава: съществува част от паметта, наречена *програмен стек*, и *регистър-указател* към върха на стека).

Всяка рекурсивна функция може да бъде заменена от еквивалентен итеративен вариант. Това може да стане по някой от следните начини:

- Всички параметри на рекурсивната функция се изнасят като глобални променливи и се модифицират по подходящ начин. Този подход е свързан с модифициране на логиката на програмния код и не винаги е възможен. (Защо?)

- Симулира се организация на програмния стек: Използва се собствен стек, в който се записват последните стойности на всички участващи във функцията променливи. Функцията от своя страна работи с последния набор от променливи, записани в стека.

За да упражните втората техника, решете, като използвате стек, следните задачи:

- Повдигане на реално число в степен.
- Намиране на n-тото число на Фибоначи. Да се запази логиката и неефективността на директната рекурсивна реализация, *виж 1.2.2.*
- Задачата за Ханойските кули (*виж 7.8.*).

Задача 2.54. *Полиноми*

Да се напише програма за работа с полиноми с реални коефициенти и корени от вида:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

представени, чрез свързан списък. Модулът за работа с полиноми трябва да включва операциите събиране, изваждане, умножение, деление на полиноми, както и намиране на стойността на полином за дадена стойност на неизвестната променлива x [*Наков-1998*].

Задача 2.55. *Сортиране чрез дърво*

В *2.3.* показахме как могат да се получат ключовете на двоично дърво за претърсване, сортирани във възходящ ред. Какво трябва да се промени при обхождането, така че редицата от ключове да бъде сортирана низходящо?

Задача 2.56. *Сума*

Да се напише програма, която намира сумата от нивата по всички върхове в двоично дърво.

Задача 2.57. *Почти-листа*

Да се напише програма, която отпечатва всички върхове на двоично дърво, които имат за наследници само листа.

Задача 2.58. *Възстановяване на дърво*

Даден е резултатът от *две* обхождания на двоично дърво измежду {*ЛКД, ЛДК и КЛД*}. Да се намери третото обхождане [*Наков-1998*].

Задача 2.59. *Построяване на идеално балансирано дърво*

Да се напише програма, която по дадено дърво построява *идеално* балансирано дърво, съдържащо същите ключове.

Упътване: Може да се построи линеен списък, в който елементите участват във възходящ ред, и след това да се използва за построяването на идеално балансирано дърво [*Наков-1998*].

Задача 2.60. *Проверка за идеално балансирано дърво*

Да се напише функция, която проверява дали дадено двоично дърво за претърсване е идеално балансирано.

Задача 2.61. *Търсене в интервал*

Дадени са двоично дърво за претърсване и две цели числа t_1 и t_2. Да се намерят всички върхове x от дървото с ключове, намиращи се в интервала $[t_1, t_2]$, т. е. $t_1 \leq$ `key[x]` $\leq t_2$.

Глава 3
Сортиране

"... there are many best methods, depending on what is to be sorted, on what machine, for what purpose."

~ D.E.Knuth

Често при работа с големи еднотипни данни се налага въвеждането на някаква наредба с цел по-лесната им обработка. Както ще видим в *глава 4*, подредбата на елементите би могла да ни даде значително по-ефективен алгоритъм за търсене в сравнение със случая, когато данните не са подредени.

Прието е процесът на пренареждане (пермутиране по подходящ начин) на елементите на някакво множество от обекти в определен ред да се нарича *сортиране*. Сортирането е основна дейност с широка сфера на приложение: речници, телефонни указатели, справочни индекси и въобще навсякъде, където се налага бързо търсене и намиране на различни обекти. Сортирането е неделима част от нашето ежедневие — зад всяко подреждане, от бюрото на което седим, до гардероба, чантата ни и т.н. се крие някакъв вид сортиране.

Сортирането е изключително широко понятие и в зависимост от типа на сортираните данни може да се реализира по най-разнообразни начини. Разнообразието от алгоритми е толкова голямо, че Кнут посвещава на сортирането (и търсенето) целия трети том (над 800 страници) на знаменитата си монография "Изкуството на компютърното програмиране" (*виж* [*Knuth-3/1968*]). Немалко внимание им отделя и Никлаус Уирт, създателят на езика Паскал, в "Алгоритми + структури от данни = програми" (*виж* [*Уирт-1980*]).

Съществуват различни класификации на алгоритмите за сортиране. Може би най-популярната е в зависимост от местонахождението на данните. Изхождайки от този критерий, различаваме *вътрешно* (данните се намират в оперативната памет на компютъра и най-често е възможен директен достъп до произволен елемент на множеството) и *външно* (данните са във външната памет на компютъра и достъпът до тях става най-често строго последователно, като се започва от първия елемент). В зависимост от операцията, извършвана над елементите, различаваме сортиране чрез *сравнение* (най-често с помощта на <, > и =) и чрез *трансформация* (с помощта на *аритметични операции*, без пряко сравнение на двойки елементи). Други важни класификации се основават на определени свойства на алгоритмите за сортиране. Например *устойчиви* и *неустойчиви*. Един метод се нарича *устойчив*, ако относителният ред на елементите с равни ключове остава непроменен в процеса на сортиране. Устойчивите методи на сортиране се предпочитат тогава, когато елементите на множеството вече са сортирани съгласно някакъв друг критерий. Например, нека искаме да сортираме файловата си система по име и по разширение: файловете се нареждат по име, като тези с еднакво име се нареждат по разширение. Това може да стане на две стъпки: 1) сортиране по разширение, и 2) сортиране по име. Докато при първото сортиране (по разширение) устойчивостта на алгоритъма е без значение, при повторното сортиране (по име) използването на устойчив метод вече ще бъде задължително.

Процесът на сортиране на елементите от дадено множество, както вече по-горе споменахме, се свежда най-общо до тяхното пренареждане. Ще дефинираме понятията по-строго. Нека е дадено множеството M с елементи

$$a_1, a_2, \ldots, a_n,$$

и функция f, дефинирана върху тях.

Под *сортиране* на елементите на M ще разбираме пермутирането им в подходящ ред

$$a_{i_1}, a_{i_2}, \ldots, a_{i_n}$$

така, че да бъде изпълнено:

$$f(a_{i_1}) \leq f(a_{i_2}) \leq \ldots \leq f(a_{i_n}).$$

Функцията *f* се нарича *функция на нареждане* на множеството. Ще считаме, че тя предварително е изчислена за всеки елемент и е запомнена явно като част (поле, *ключ*) от него. Елементите на множеството, подлежащо на сортиране, най-често се представят като записи, едно от полетата на които е ключът:

```
#define MAX 100
struct  CElem {
  int key;
  /* .............
     Някакви данни
     ............. */
} m[MAX];
```

При универсалните методи на сортиране, и там, където е възможно, ще следваме горната декларация за тип на елементите на множеството. Множеството ще представяме най-често като масив, но понякога ще ползваме и динамично представяне като свързан списък.

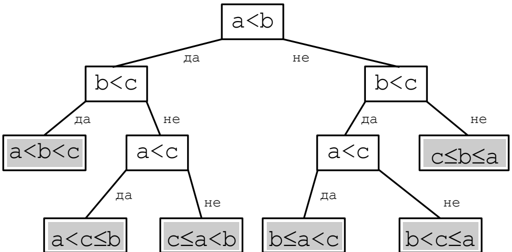
Основно изискване към алгоритмите за сортиране е минималният разход на *допълнителна памет*. Друго важно изискване е за минимален брой извършвани сравнения и размени на елементи. Обикновено сортирането се извършва чрез проста размяна на място на два елемента на масива. В приложените по-долу програми за размяна стойностите на две променливи ще използваме следната функция:

```
/* Разменя стойностите на *x1 и *x2 */
void swap(struct CElem *x1, struct CElem *x2)
{ struct CElem tmp = *x1; *x1 = *x2; *x2 = tmp; }
```

3.1. Сортиране чрез сравнение

Всички алгоритми в настоящия параграф, принадлежат към класа на алгоритмите за *сортиране чрез сравнение*. Това са класически алгоритми, при които единствената позволена операция е сравнение между двойки елементи с помощта на операциите < (≤), > (≥) и = (≠).

3.1.1. Дърво на сравненията



Фигура 3.1.1. Дърво на сравненията за множеството {*a*, *b*, *c*}.

Това е може би най-елементарният метод за сортиране чрез сравнение. Основава се на поредица от сравнения, всяко от които ни носи допълнителна информация. Процесът продължава до пълно сортиране на множеството. В зависимост от резултата от сравнението получаваме различни съотношения между елементите и оттук — различно продължение на процеса. Можем да считаме, че всяко сравнение от вида $x < y$ има два изхода: *да*, ако *x* е по-малко от *y*, и *не* — в противен случай. Процесът на сравнение би могъл да се изобрази графично като двоично дърво, в чиито листа се намират сортираните последователности, а в нелистата — сравнения между

двойки елементи на множеството. *Фигура 3.1.1.* илюстрира метода върху триелементното множество $\{a, b, c\}$.

Дървото ни дава следния алгоритъм за сортиране на множеството:

```
if (a < b)
   if (b < c) printf("a,b,c");
   else
      if (a < c) printf("a,c,b");
      else printf("c,a,b");
else
   if (b < c)
      if (a < c) printf("b,a,c");
      else printf("b,c,a");
   else printf("c,b,a");
```

Горният алгоритъм е красив и полезен дотолкова, доколкото 1) ни гарантира минимален брой сравнения (*Защо?*), и 2) дава *явно* дървото на сравненията (последното се използва за доказване на важен резултат: минимална времева сложност на *произволен* алгоритъм за сортиране чрез сравнение — *виж 3.1.10.*). Предимствата му, за съжаление, свършват дотук. Не е трудно да се забележи, че броят на възможните изходи (листата на дървото) е $n!$, колкото са всевъзможните пермутации на елементите на изходното множество, което, дори и при достатъчно малък брой елементи, е доста голямо число, непозволяващо написване на съответна програма.

Налага се да се търсят други по-ефективни методи за сортиране.

Задача за упражнение:

Да се докаже, че дървото на сравненията гарантира минимален брой сравнения в най-лошия случай.

3.1.2. Сортиране чрез вмъкване

Съществуват три класически елементарни универсални методи за сортиране чрез сравнение: чрез вмъкване, чрез избор и по метода на мехурчето. В основата на всеки един от тях стои проста идея, позволяваща бърза и ясна реализация. Елементарните методи за сортиране са ефективни при сравнително малък брой елементи (около 20) и често се използват на практика. За съжаление, при по-голям брой елементи скоростта им рязко пада, поради което се налага използване на други методи. Действително, и трите елементарни метода се характеризират с алгоритмична сложност $\Theta(n^2)$, което е доста по-бавно в сравнение със сложността $\Theta(n.\log_2 n)$, характерна за съвременните методи за сортиране като пирамидалното (*виж 3.1.9.*) или бързото сортиране (*виж 3.1.6.*). Въпреки това, елементарните методи имат своето място, тъй като за достатъчно малки последователности те са по-ефективни и, както ще видим по-долу, често се използват в хибридни варианти с цел повишаване на бързодействието. Така например, широко използван подход при бързото сортиране е, при достигане до дял с достатъчно малко елементи да се използва по-прост метод като сортиране чрез вмъкване.

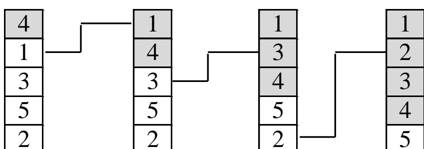
Сортирането чрез вмъкване е добре познатият на картоиграчите метод за нареждане, при който играчът, държейки в лявата си ръка картите, ги изважда една по една в дясната си ръка и ги поставя на правилната им позиция. За вмъкване на картата на правилното място се налага последователното ѝ сравнение (с поглед) с вече наредените карти до откриване на вярната позиция.

Нека се върнем към нашата терминология. Имаме масив от елементи от тип `struct CElem` с ключ `key`, които искаме да сортираме. Масивът се разделя на сортирана и несортирана област. Сортираната област обикновено се разполага в началото на масива, като първоначално обхваща само първия му елемент. Сортирането протича на $n-1$ стъпки. На i-тата стъпка сортираната област се разширява с един елемент отдясно, като за целта $(i+1)$-тият поред елемент (нека го

означим с x) се вмъква на подходящо място в сортираната последователност, т. е. сред елементите от 1 до i.

Как става вмъкването? Един очевиден алгоритъм е последователното сравнение и евентуална размяна на x със стоящия вляво от него елемент. Процесът продължава до възникване на една от следните ситуации (*виж фигура 3.1.2.*):

1) достигане на елемент с ключ, по-малък или равен от този на x;
2) достигане на първия елемент на масива.



Фигура 3.1.2. Сортиране чрез вмъкване.

Следва примерна реализация. Забележете, че *първият* елемент на масива всъщност е `m[0]`.

```
void straightInsertion(struct CElem m[], unsigned n)
{ struct CElem x;
  unsigned i;
  int j;
  for (i = 0; i < n; i++) {
    x = m[i];
    j = i-1;
    while (j >= 0 && x.key < m[j].key)
      m[j+1] = m[j--];
    m[j+1] = x;
  }
}
```

🖫 insert_s.c

Двете проверки могат да се обединят в една чрез добавяне на елемент с ключ $-\infty$ като нулев елемент на масива. Този подход е известен като *метод на ограничителя*. Тъй като стойността $-\infty$ се представя трудно в компютърните системи, по-просто е да използваме като ограничител някакъв ключ от множеството на допустимите ключове. Не е трудно да се съобрази, че стойността на ключа на x би ни свършила чудесна работа. (*Защо?*) Получаваме следната програмна реализация на алгоритъма за пряко вмъкване (Този път елементите са разположени на позиции 1,2,...,n):

```
void straightInsertion(struct CElem m[], unsigned n)
{ unsigned i, j;
  struct CElem x;
  for (i = 1; i <= n; i++) {
    x = m[i]; m[0].key = x.key;
    for (j = i - 1; x.key < m[j].key; j--)
      m[j+1] = m[j];
    m[j+1] = x;
  }
}
```

🖫 inserts2.c

вмъкване	Минимален брой	Среден брой	Максимален брой
сравнения	$n-1$	$(n^2+n-2)/4$	$(n^2+n)/2-1$

размени	$2(n-1)$	$(n^2+9n-10)/4$	$(n^2+3n-4)/2$

Таблица 3.1.2. Сравнения и размени при сортирането чрез вмъкване.

Не е трудно да се установи, че средният брой на извършваните сравнения, както и този на необходимите размени (забележете, че в горната програма *преки размени* на практика липсват, като вместо това се използват еднопосочни *присвоявания*), е от порядъка на n^2. Това се вижда и от приложената *таблица 3.1.2*. Любознателният читател би могъл да намери подробна обосновка на горните оценки в [*Knuth-3/1968*][*Уирт-1980*].

Дали бихме могли да повишим ефективността на сортирането чрез вмъкване? И да, и не. Наистина, използвайки двоично търсене (*виж 4.3.*), бихме могли да намалим броя на *сравненията*, необходими за намиране позицията на вмъкване на x от i-тата стъпка, до $\log_2 i$:

```
void binaryInsertion(struct CElem m[], unsigned n)
{ struct CElem x;
  unsigned i, med, r;
  int j, l;
  for (i = 1; i < n; i++) {
    x = m[i];
    l = 0;
    r = i - 1;
    /* Двоично търсене */
    while (l <= (int)r) {
      med = (l + r) / 2;
      if (x.key < m[med].key)
        r = med - 1;
      else
        l = med + 1;
    }
    /* Мястото е намерено. Следва вмъкване и изместване надясно */
    for (j = i - 1; j >= l; j--)
      m[j + 1] = m[j];
    m[l] = x;
  }
}
```
🖫 insert_b.c

За съжаление, последното няма да повлияе върху броя на *размените*, тъй като, макар и да знаем конкретната позиция на вмъкване, се налага да преминем последователно през елементите на масива до достигането ѝ. Това неудобство може да се заобиколи чрез използване на динамичен свързан списък, но тогава не бихме могли да приложим двоично търсене.

Задачи за упражнение:

1. Да се докаже, че ключът на елемента, който ще се вмъква, е добър ограничител при използване на метода на ограничителя за реализация на сортиране чрез вмъкване.
2. Да се сравнят различните варианти на сортирането чрез вмъкване.
3. Да се изведат формулите от *таблица 3.1.2*.

3.1.3. Сортиране чрез вмъкване с намаляваща стъпка. Алгоритъм на Шел

През 1959 г. Шел предлага едно нелошо подобрение на метода за сортиране чрез вмъкване. Идеята е да се извършва многократно сортиране чрез пряко вмъкване на част от елементите на масива със стъпка δ, която постепенно намалява и накрая достига 1.

Процеса на сортиране чрез пряко вмъкване само на подмножествата от вида x_k, $x_{k+\delta}$, $x_{k+2\delta}$ ... ($1 \leq k \leq \delta$) ще наричаме *δ-сортиране*. Нека е дадена последователността от стъпки δ_s, δ_{s-1}, ...,δ_1, удовлетворяваща условието

$$\delta_s > \delta_{s-1} > ... > \delta_1 = 1.$$

Очевидно в резултат на последователното прилагане на δ_s-сортиране, δ_{s-1}-сортиране, ..., δ_1-сортиране изходното множество ще бъде сортирано. Не е трудно да се забележи, че ако една последователност е била δ-сортирана, то след прилагане на δ′-сортиране ($\delta' < \delta$), тя продължава да бъде δ-сортирана. (*Защо?*) Така, всяко следващо δ_i-сортиране извлича полза от предишното, като в крайна сметка сортирането със стъпка 1 довършва останалата работа.

Всяко δ-сортиране се свежда до δ на брой сортирания чрез пряко вмъкване, всяко от които има нужда от собствен ограничител. Добавянето на нулев елемент в началото на масива се оказва недостатъчно. Необходими са ни общо δ_s на брой ограничителя, което налага разширяване на масива с δ_s на брой елемента. Ще работим с единствен масив, като първите му елементи ще използваме като ограничители. За да можем да се обръщаме към елементи с отрицателен индекс, ще използваме трик: ще извикаме функцията shellSort() не с shellSort(m, MAX), а с обръщението shellSort(m + steps0 + 1, MAX).

```
#define MAX 100
#define STEPS_CNT 4
#define steps0 40
const unsigned steps[STEPS_CNT] = { steps0, 13, 4, 1 };

struct  CElem {
  int key;
  /* .............
     Някакви данни
     ............. */
};

..................

void shellSort(struct CElem m[], unsigned n)
{ int i,j,k,s;
  unsigned stepInd;
  struct CElem x;

  for (stepInd = 0; stepInd < STEPS_CNT; stepInd++) {
    s = -(k = steps[stepInd]); /* Ограничител */
    for (i = k + 1; i <= (int)n; i++) {
      x = m[i];
      j = i - k;
      if (0 == s)
        s = -k;
      m[++s] = x;
      while (x.key < m[j].key) {
         m[j + k] = m[j];
         j -= k;
      }
      m[j + k] = x;
    }
  }
}

..................
```

```
int main(void) {
  struct CElem m[MAX + steps0 + 2];
  .................
  shellSort(m + steps0 + 1, MAX);
  .................
  return 0;
}
```

🖫 shell.c

Идеята за ограничителя беше добра при сортирането с вмъкване, но тук възниква нужда от *няколко* ограничителя, което очевидно е неефективно и води до неудобства при адресирането на масива, които по-горе решихме с "програмисти хватки". Може би е ефектно и поучително, но дали беше нужно? Ако се замислим малко, лесно стигаме до извода, че всъщност можем и без ограничители! В резултат на което получаваме далеч по-кратък и по-ясен код. Обръщаме внимание обаче, че за това "си плащаме" с ефективност, тъй като нашият нов алгоритъм (виж по-долу) извършва два пъти повече сравнения във вътрешния си *while*-цикъл. Тук използваме друга ефективна 16-елементна редица (сортираме елементи от `l+1` до `r`).

```
void shellSort(struct CElem m[], unsigned l, unsigned r)
{ static unsigned incs[16] = { 1391376,463792,198768,86961,33936,
                               13776,4592,1968,861,336,112,48,
                               21,7,3,1 };
  unsigned i, j, k, h;
  struct CElem v;

  for (k = 0; k < 16; k++)
    for (h = incs[k], i = l+h; i <= r; i++) {
      v = m[i]; j = i;
      while (j > h && m[j-h].key > v.key) {
        m[j] = m[j-h];
        j -= h;
      }
    m[j] = v;
  }
}
```

🖫 shell2.c

Как да изберем нужните ни стъпки? Този въпрос е изключително сложен и математически все още не е решен. За да се извлече максимална полза от последователното прилагане на δ-сортиранията, е необходимо те да си взаимодействат в максимална степен. Това ни навежда на идеята, че стъпките трябва да бъдат кратни една на друга, например последователни степени на двойката.

Папернов и Стасевич показват, че ако се използва редицата

$$1, 3, 7, 15, 31, 63, 127, ..., 2^k - 1, ...$$

алгоритъмът изисква $\Theta(n\sqrt{n})$ стъпки [*Papernov*][*Stasevic-1975*].

Прат предлага

$$1, 2, 3, 4, 6, 8, 9, 12, 16, ..., 2^p3^q, ... ,$$

за която доказва, че са нужни $\Theta(n(\log_2 n)^2)$ стъпки. Асимптотично погледнато, редицата на Прат е най-добра, но, за съжаление расте бавно и е прекалено дълга (*виж* [*Pratt-1979*]). Кнут (*виж* [*Knuth-3/1968*]) предлага редицата

$$1, 4, 13, 40, 121, ...,$$

зададена с уравненията $\delta_{k-1} = 3\delta_k + 1$, $\delta_s = 1$, $s = [\log_3 n] - 1$, записана в обратен ред. Друга добра редица, предложена също от Кнут, е:

1, 3, 7, 15, 31, ...

зададена с уравненията $\delta_{k-1} = 2\delta_k + 1$, $\delta_s = 1$, $s = [\log_2 n] - 1$, отново записана в обратен ред. В този случай броят на сравненията е от порядъка на $n^{1,2}$. Добри резултати се получават при редици, доближаващи се максимално до намаляваща геометрична прогресия с частно 1,7. В този случай сложността на алгоритъма е $\Theta(n(\log_2 n)^2)$.

Задача за упражнение:

Да се докаже, че ако една последователност е била δ-сортирана, то след прилагане на δ′-сортиране, $\delta' < \delta$ тя продължава да бъде δ-сортирана.

3.1.4. Метод на мехурчето

Методът на мехурчето несъмнено е най-популярният сред програмистите, особено при достатъчно малък брой елементи за сортиране. Както личи от приложената по-долу *таблица 3.1.4.* обаче, популярността му едва ли се дължи на ефективността му (Това е най-лошият алгоритъм измежду всички разгледани в настоящата глава!). "Чарът" му би могъл да се търси по-скоро в лекотата, с която се реализира, в простотата на идеята му и може би донякъде в екзотичното му име. Обикновено първият алгоритъм за сортиране, с който се запознава начинаещият програмист, е именно методът на мехурчето. Не бива да се пренебрегва и фактът, че този метод стои в основата на бързото сортиране на Хоор — най-бързият *универсален* алгоритъм за сортиране, известен до момента.

Идеята на алгоритъма е последователно да се преглеждат елементите на масива и, ако за някоя двойка съседни елементи x_{i-1} и x_i се окаже, че $x_{i-1} > x_i$, те си разменят местата (предполагаме сортиране във възходящ ред). В процеса на сортиране най-леките елементи, подобно на мехурчета, изплуват към "повърхността", т. е. към левия край на масива. Оттук произлиза и името на метода.

```
void bubbleSort1(struct CElem m[], unsigned n)
{ unsigned i, j;
  for (i = 1; i < n; i++)
   for (j = n-1; j >= i; j--)
     if (m[j-1].key > m[j].key)
       swap(m+j-1, m+j);
}
```

🖫 bubsort1.c

мехурче	Минимален брой	Среден брой	Максимален брой
сравнения	$n(n-1)/2$	$n(n-1)/2$	$n(n-1)/2$
размени	0	$3.(n^2-n)/4$	$3.(n^2-n)/2$

Таблица 3.1.4. Сравнения и размени при сортирането по метода на мехурчето.

Таблица 3.1.4. дава информация за основните характеристики на сортирането по метода на мехурчето. Както се вижда, броят на сравненията, извършвани от алгоритъма, винаги е един и същ, а именно $n(n-1)/2$. Това ни навежда на мисълта, че са възможни някои подобрения. Бихме могли например да добавим флаг, указващ дали на текущата итерация е била извършена размяна. Процесът на сортиране ще продължава дотогава, докато при преминаванията се извършват размени. Реалните тестове обаче показват, че ползата от въвеждането на флаг е почти незабележима, като при обратно подреден масив алгоритъмът дори се забавя.

Друго по-съществено подобрение можем да получим, изхождайки от разсъждението, че е безсмислено да преглеждаме елементите, за които със сигурност се знае, че са на окончателните си места. Този път сортирането ще извършваме в обратна посока. Последното, разбира се, е не-

съществено, но дава нов поглед върху алгоритъма: започваме от "дъното" в края на масива и най-тежките елементи потъват към него. Ще поддържаме променлива *i*, указваща максималния индекс, при който е била извършена размяна на предходната итерация. Очевидно елементите, разположени вдясно от последната размяна, са на окончателните си места, което ни дава основание да не ги разглеждаме повече.

```
void bubbleSort2(struct CElem m[], unsigned n)
{ unsigned i, j, k;
  for (i = n; i > 0; i = k)
    for (k = j = 0; j < i; j++)
      if (m[j].key > m[j+1].key) {
        swap(m+j, m+j+1);
        k = j;
      }
}
```

🖫 bubsort2.c

Задача за упражнение:

Да се изведат формулите от *таблица 3.1.4*.

3.1.5. Сортиране чрез клатене

Въпреки ниската си ефективност методът на мехурчето дава възможност за редица подобрения. По-горе вече посочихме две от тях — добавяне на флаг и поддържане на променлива, указваща максималния индекс на размяна от предходната стъпка. Съществува още една особеност: отделен лек елемент в края на масива ще изплува за една-единствена итерация, докато "самотен" тежък елемент, разположен близо до повърхността, би потъвал на всяка стъпка на алгоритъма само едно ниво надолу към дъното. Ако обърнем посоката на циклите, нещата ще стоят точно обратно, но асиметрията очевидно ще се запази. Това ни навежда на мисълта за обръщане посоката на сортиране *на всяка стъпка*. Така получаваме трето подобрение на алгоритъма. Прилагайки трите подобрения едновременно, получаваме нов алгоритъм — сортиране чрез клатене:

```
void shakerSort(struct CElem m[], unsigned n)
{ unsigned k = n, r = n-1;
  unsigned l = 1, j;
  do {
    for (j = r; j >= l; j--)
      if (m[j-1].key > m[j].key) {
        swap(m+j-1,m+j);
        k = j;
      }

    l = k + 1;
    for (j = l; j <= r; j++)
      if (m[j-1].key > m[j].key) {
        swap(m+j-1,m+j);
        k = j;
      }

    r = k - 1;
  } while (l <= r);
}
```

🖫 shaker.c

В най-добрия случай горният алгоритъм извършва $n-1$ сравнения, а в средния случай — отново $\Theta(n^2)$. Макар в това отношение да се забелязва известно подобрение в сравнение с класическия метод на мехурчето, общата ефективност на алгоритъма не се подобрява особено, тъй като сортирането чрез клатене не намалява броя на извършваните *размени*. А размяната по-принцип е неколкократно по-тежка операция от сравнението.

Задача за упражнение:

Да се сравнят различните варианти на сортирането по метода на мехурчето:
а) мехурче без флаг
б) мехурче с флаг
в) сортиране чрез клатене.

3.1.6. Бързо сортиране на Хоор

Разгледаните по-горе алгоритми, основаващи се на размяна на елементи, имат доста скромна ефективност, като класическият метод на мехурчето има дори най-лоши параметри в сравнение с всички разгледани до момента алгоритми за сортиране. Въпреки положените немалко усилия от наша страна (при сортирането чрез клатене бяха направени цели три подобрения!), тези характеристики не се подобриха особено. Като че ли сортирането чрез размяна не води до ефективни алгоритми... Дали това наистина е така?

При клатенето, въпреки направените подобрения, не постигнахме задоволителни резултати, тъй като, намалявайки броя на *сравненията*, не успяхме да повлияем на броя на *размените* на елементи. Сега ще подходим обратно: ще се стремим главно към минимизиране на броя на размените. Тъй като размяната на два елемента е неколкократно по-скъпа операция от сравнението, би могло да се очаква сериозно подобрение. Как обаче да редуцираме броя на размените? Нека си припомним третото подобрение на метода на мехурчето (*виж 3.1.4.*), което направихме при реализацията на клатенето (*виж 3.1.5.*): обръщане на посоката на обхождане на елементите на всяка стъпка на алгоритъма. Причина за това беше нашето наблюдение, че отделен тежък елемент, разположен близо до повърхността, ще потъва на всяка стъпка само с една позиция към дъното. Същевременно, отделен лек елемент, разположен в края на масива, ще изплува до крайната си позиция за една единствена итерация. Забележете обаче, че дори и в този случай лекият елемент ще трябва последователно да се разменя с *всички* елементи по пътя към крайната си позиция, а тази операция е скъпа! Как да намалим броя на размените? В случай на обратно подреден масив бихме могли да разменим първия елемент с последния, втория — с предпоследния и т.н., сортирайки целия масив за $[n/2]$ размени. От само себе си се налага изводът — колкото на по-големи разстояния се извършват размените, толкова по-голяма е тяхната ефективност.

Идеята на алгоритъма, предложен от Хоор, е да изберем някакъв елемент x и да разделим масива на два дяла: ляв, в който елементите са по-малки от x, и десен, в който са по-големи. Прилагаме същия алгоритъм за лявата и дясната част, намалявайки постепенно лявата и дясната граница на разглежданите подмасиви, докато не достигнем до интервали, съдържащи единствен елемент. След приключване работата на алгоритъма, масивът ще бъде сортиран. (*Защо?*)

Да означим с q индекса на x в масива, т.е. `x = m[q]`. Нека сортираме подмасива `m[l,l+1,...,r]` и нека `partition()` го разделя на две части: лява (`m[l,l+1,...,q]`) и дясна (`m[q+1,q+2,...,r]`), и връща q като резултат. Забележете, че разделянето почти сигурно е свързано с размени на елементи, т.е. `partition()` ***не търси*** елемент x в масива, ами го ***избира*** и извършва разделяне на две области относно стойността му. По-долу ще видим как може да стане това.

Бързото сортиране би могло да се запише най-общо така (Масивът `m[]` не се подава като параметър, а се разглежда като глобална променлива: така се пести място в стека и се печели скорост, тъй като функцията е рекурсивна.):

```
void quickSort(int l, int r)
{ int q;
  if (l < r) {
    q = partition(l,r);
    quickSort(l,q);
    quickSort(q+1,r);
  }
}
```

Нека предположим, че въз основа на някаква формула вече сме пресметнали q. (На първо време ще избираме най-левия или най-десния елемент на масива преди разделянето.) Ще отбележим, че `m[q]` съдържа стойността на x *преди* разделянето. След разделянето q ще бъде граничен индекс: вляво от него (включително) ще имаме елементи, чийто ключ не надвишава стойността на x, а вдясно — ключове, по-големи от x.

Възниква следващият съществен въпрос: Как да извършваме разделянето? Съществуват най-общо два различни подхода. Следва първият:

```
unsigned partition(int l, int r)
{ int q, j, x;
  q = l - 1; x = m[r].key;
  for (j = l; j <= r; j++)
    if (m[j].key <= x) {
      q++;
      swap(m+q,m+j);
    }

  if (q == r) /* Всички елементи са <= x. Областта намалява с 1. */
    q--;
  return q;
}
```

Как работи предложеният метод? Избира се елемент `x`, по който да се извършва разделянето. В лявата част на масива трябва да останат елементи, по-малки или равни на `x`, а в дясната — строго по-големи от `x`. Разглежданият дял от масива `m[l,l+1,...,r]` се преглежда отляво-надясно, като при това в лявата му част се изгражда постоянно разширяваща се област от елементи, по-малки или равни на `x`. Десният край на областта се определя от `q`. Когато `j` достигне края `r` на дяла, `q` ще сочи границата между двете области. (*Забележка:* При реална програма е добре обръщението към функцията `swap()` да се замени с нейния код с цел по-голяма ефективност.)

Друг възможен метод е да се използват два индекса `i` и `j`, показващи границите на *две* непрекъснато разширяващи се области от *двата* края към центъра. На всяка стъпка на алгоритъма се прави опит за разширяване на лявата област надясно, докато това е възможно, т. е. докато вдясно от нея стои елемент, по-малък или равен на x. Същото се извършва и за дясната област (двата *while*-цикъла). След това се разменят местата на двата елемента-прегради за лявата и дясната области, които са спрели разширяването им, и процесът се повтаря отначало. Приключва при "срещане" на двете области, т. е. когато двете им граници се разминат. Индексът `i` сочи десния край на лявата област, а `j` — левия край на дясната:

```
unsigned partition(int l, int r)
{ unsigned i, j, x;
  i = l; j = r; x = m[l].key;
  do {
    while (x > m[i].key) i++;
    while (x < m[j].key) j--;
    if (i <= j) {
      swap(m+i,m+j);
      i++;
```

```
      j--;
    }
  while (i <= j);
  return j;
}
```

След обединяване на функциите `partition()` и `quickSort()` в едно, получаваме (за първия вариант):

```
void quickSort(int l, int r)
{ int i,j,x;
  i = l-1; x = m[r].key;
  for (j = l; j <= r; j++)
    if (m[j].key <= x) {
      i++;
      swap(m+i,m+j);
    }
  if (i == r) /* Всички елементи са <= x. Областта намалява с 1. */
    i--;
  if (l < i)
    quickSort(l,i);
  if ((i+1) < r)      /***/
    quickSort(i+1,r); /***/
}
```

За втория вариант имаме:

```
void quickSort(int l, int r)
{ int i, j, x;
  i = l;
  j = r;
  x = m[r].key;
  do {
    while (x > m[i].key) i++;
    while (x < m[j].key) j--;
    if (i <= j) {
      swap(m+i, m+j);
      i++;
      j--;
    }
  } while (j >= i);
  if (j > l)
    quickSort(l, j);
  if (i < r)         /***/
    quickSort(i, r); /***/
}
```

🖫 qsort.c

За сортиране на двете части на масива се извършват две последователни рекурсивни обръщения. Ясно е, че те не могат да се извършат едновременно и второто винаги се отлага. То се оказва "излишно" в смисъл, че няма нужда от нова рекурсия — вече можем да се справим и итеративно. Някои компилатори успяват автоматично да разпознаят и премахнат втората рекурсия. Разбира се, с минимални усилия бихме могли и сами да се справим, получавайки *полуитеративен* вариант. За целта е достатъчно да заменим редовете, отбелязани с `/***/` с присвояването `l = i+1;` (за първия вариант) и `l = i;` (за втория вариант), след което да повторим още веднъж действията на функцията. За целта, ще трябва да я "облечем" в още един цикъл.

При желание бихме могли изобщо да се освободим от рекурсията. За целта, ще използваме стандартния начин за премахването ѝ — с използване на стек. Ще отбележим, че рекурсия и итерация при разработката на алгоритми са абсолютно взаимозаменяеми Действително, всеки цикъл може да се симулира с рекурсия. В обратната посока нещата са по-сложни. Ще отбележим само, че компютрите са итеративни машини и следователно всяка рекурсия, която пишем, в крайна сметка се свежда до итерация. От друга страна, рекурсията се моделира от компютъра чрез поставяне на определени данни в системния стек. Тогава защо да не използваме свой собствен стек в явен вид? За целта, на всяка стъпка лявата част на масива ще се обработва директно, докато дясната ще се поставя в стека, като обработката ѝ ще се отлага за момента, когато се приключи окончателно с лявата част и с всички свързани с нея възникнали впоследствие подзадачи (тук x избираме като елемента в средата на разглеждания подмасив):

```
void quickSort(void)
{ int i,j,l,r,s,x;
  struct { int l, r; } stack[MAX];

  stack[s = 0].l = 0;
  stack[0].r = n-1;
  for (;;) {
    l = stack[s].l;
    r = stack[s].r;
    if (0 == s--)
      break;
    do {
      i = l; j = r; x = m[(l+r)/2].key;
      do {
        while (m[i].key < x) i++;
        while (m[j].key > x) j--;
        if (i <= j) {
          swap(m+i,m+j);
          i++;
          j--;
        }
      } while (i<=j);
      if (i < r) {          /***/
        stack[++s].l = i; /***/
        stack[s].r := r;  /***/
      }                     /***/
      r = j;                /***/
    } while (l < r);
   }
 }
```

Нека се опитаме да оценим необходимия програмен стек при работата на предложеното итеративно решение. Ясно е, че в оптималния случай всеки път ще разполовяваме дяла, получавайки в резултат $\Theta(\log_2 n)$ рекурсивни обръщения, които се стекуват. В най-лошия случай обаче, при неподходящ избор на елемент, размерът на разглеждания дял всеки път ще намалява само с 1, при което ще имаме $\Theta(n)$ стекувания. При рекурсивния вариант нещата стоят още по-лошо, тъй като там в стека постъпват още повече данни: адрес за връщане, параметри и локални променливи. Един възможен изход е да се стекуват по-дългите дялове, а по-късите да се разглеждат незабавно. Не е трудно да се забележи, че при този подход необходимият размер стек ще бъде винаги $\Theta(\log_2 n)$. За целта редовете, отбелязани с `/***/` следва да се заменят със следния програмен фрагмент (*Защо?*):

```
/* Поставя се по-големият дял в стека */
if (j - l < r - i) {
```

```
      /* Поставя се десният дял в стека */
      if (i < r) {
        stack[++s].l = i;
        stack[s].r = r;
      }
      r = j;
    }
    else {
      /* Поставя се левият дял в стека */
      if (l < j)
        stack[++s].l = l;
        stack[s].r = j;
      }
      l = i;
    }
```

Ясно е, че колкото по-равномерно се разпределят елементите в лявата и дясната част на масива в резултат на разделянето, толкова по-малко стъпки ще бъдат необходими. В най-добрия случай, ако всеки път избираме *средния по големина* елемент, за цялостното сортиране на масива ще ни бъдат достатъчни $\log_2 n$ разделяния. За общия брой сравнения в оптималния случай получаваме $n.\log_2 n$. Очакваният брой размени при всяко разделяне е от порядъка на $n/6$. Оттук броят на размените в оптималния случай е $(n/6).\log_2 n$. Макар вероятността да улучим средния елемент да е $1/n$, средната ефективност на бързото сортиране не зависи от n, като при това е по-лоша от посочената средно само с 2.ln 2 [*Knuth-3/1968*].

Да се върнем на въпроса: Как да избираме елемента x, относно който разделяме масива? Един възможен подход е да вземаме винаги фиксиран елемент от масива, например първия, последния или средния (както постъпвахме по-горе). Изборът на разделящ елемент е изключително важен за скоростта на алгоритъма: колкото по-близък е ключът на избрания елемент до ключа на средния *по големина* елемент за дяла, толкова по-равномерно става разбиването на масива на две части, а оттук и толкова по-малка е дълбочината на дървото на рекурсията, т. е. по-ефективен е алгоритъмът. (*Защо?*) Ето защо понякога за разделящ елемент се избира медианата на два или повече елемента. Това обаче е свързано с допълнителни операции, а оттук води до забавяне на избора, което не винаги се компенсира с извлечените ползи и не води задължително до подобрение. Съветваме читателя да се върне на *бързото сортиране* по-късно, след като се запознае с рекурсивния вариант на *побитовото сортиране* от *3.2.3.*, и да сравни начина на избор на разделящ елемент.

Бързото сортиране крие редица изненади. Както вече по-горе споменахме, това е най-бързият известен *универсален* алгоритъм за сортиране (в *3.2.* ще видим, че в някои важни частни случаи при допълнителни ограничения съществуват линейни алгоритми!). Средната му алгоритмична сложност е $\Theta(n.\log_2 n)$, каквато е сложността и на главния му конкурент — пирамидалното сортиране (*виж 3.1.9.*), което ще бъде разгледано по-късно. Практиката обаче показва, че бързото сортиране "бие" пирамидалното средно от два до три пъти. Същевременно сортирането по Хоор е доста по-непостоянно и, при неподходящ избор на x, ефективността му би могла да падне значително. В най-лошия случай, когато всеки път размерът на разглеждания дял намалява само с 1, сложността му е пропорционална на n^2. [*Knuth-3/1968*][*Уирт-1980*]

Силата на бързото сортиране е в неговата *хазартност*. Колкото по-разбъркан е масивът, толкова по-голяма е ефективността на метода. Авторът Хоор препоръчва изборът на x да се прави по случаен начин или дори като средно аритметично на три или повече *случайно избрани* елемента. На практика, подобна стратегия почти не влияе на алгоритъма в *общия*, като същевременно значително повишава ефективността му в *най-лошия* случай. Следва да се отбележи, че подобно на повечето бързи съвременни методи, силата на бързото сортиране също не се проявява при малък брой елементи. Един добър подход, елиминиращ посочения недостатък, е да се прилага бързо сортиране за дялове, с поне k (например $k = 20$) елемента, като за по-малките дялове се използва по-прост алгоритъм като мехурче (*виж 3.1.4.*), пряк избор (*виж 3.1.8.*) или вмъкване (*виж 3.1.2.*). Още по-хитра стратегия е при достигане на дял с по-

малко от k елемента да *не се прави нищо*. След приключване работата на така модифицираното бързо сортиране, масивът не е нареден, но е разделен на множество малки групи от елементи с близки стойности, като всеки елемент от коя да е група е по-голям от елементите на всички групи, вляво от разглежданата. По този начин масивът се оказва почти сортиран и е добър вход за сортирането чрез вмъкване: вмъкванията ще се извършват в рамките на съответната област, т. е. гарантирано на малки разстояния.

Задачи за упражнение:

1. Да се докаже, че бързото сортиране на Хоор сортира правилно всяка входна последователност.
2. Да се докаже, че бързото сортиране на Хоор изисква стек от порядъка на $\Theta(\log_2 n)$.
3. Да се тестват предложените програмни реализации на бързото сортиране на Хоор за всички възможни входни последователности с до 32 елемента:
   а) за последователност от случайни числа;
   б) за всички пермутации на числата от 1 до n;
   в) за всички пермутации на числата от дадено мултимножество (с повтарящи се на елементи);
   г) като се ползва принципът на нулите и единиците.
4. Да се направят емпирични тестове за всяка от подточките от горната задача и да се сравнят получените резултати с теоретичните.
5. Да се реализира итеративен вариант на бързото сортиране на Хоор.
6. При какъв брой елементи в сортирания дял при бързото сортиране следва да се използва по-прост алгоритъм като сортиране чрез вмъкване?
7. Да се докаже, че колкото по-близо до медианата е елементът, относно който става разделянето на два дяла при бързото сортиране на Хоор, толкова по-ефективен е алгоритъмът.
8. Да се определи теоретично най-добрият начин за избор на разделящ елемент при бързото сортиране на Хоор (*виж 3.1.6.*):
   а) първи;
   б) последен;
   в) среден;
   г) средно аритметично на два елемента;
   д) средно аритметично на три елемента;
   е) медиана: среден *по големина* елемент за дяла (*виж 7.1.*);
   ж) изкуствен елемент, например както при побитовото сортиране (*виж 3.2.3.*).
9. Да се направят емпирични тестове за всяка от подточките от горната задача и да се сравнят получените резултати с теоретичните.
10. Да се предложи модификация на бързото сортиране на Хоор, при която ще се гарантира времева сложност $\Theta(n.\log_2 n)$ в *най-лошия случай*.
11. Да се промени типът на формалните параметрите на функцията `quickSort()` от `int` на `unsigned`.. Какви проблеми възникват и защо? Какви решения предлагате?
12. За всяко естествено n да се намери входна последователност, при която бързото сортиране има сложност:
   а) $\Theta(n.\log_2 n)$
   б) $\Theta(n^2)$

3.1.7. Метод на "зайците и костенурките"

Ще се опитаме да подобрим скромната ефективност на метода на мехурчето (*виж 3.1.4.*), тръгвайки по друг път. Както неколкократно отбелязахме по-горе (*виж 3.1.5.* и *3.1.6.*), основен недостатък на алгоритъма е асиметрията в поведението му — лек елемент от края на масива "изплува" веднага, докато отделен тежък елемент на всяка стъпка "потъва" едно-единствено ниво надолу. Леките елементи, т. е. тези, които се придвижват бързо, ще наричаме *зайци*, а бавните —

костенурки. Наблюденията показват, че почти всяка голяма последователност от елементи съдържа костенурка, а всяка костенурка води до максимално забавяне.

Как да намалим вредата от костенурките? Промяната на посоката на обхождане, която приложихме при алгоритъма за сортиране чрез клатене (*виж 3.1.5.*), просто разменя ролите на зайците и костенурките. През 1991 г. в две последователни статии в *сп. Byte* Лейси и Бокс предлагат друг радикален подход — елиминиране на костенурките, като им се позволява да "скачат" към крайната си позиция, вместо да "пълзят". За целта се извършват сравнения между *отдалечени* един от друг елементи, вместо между *съседни*, както е при класическото мехурче. Разстоянието между сравняваните елементи се определя от постепенно намаляваща стъпка.

Как да изберем стъпката? След около 200 000 опита Лейси и Бокс достигат чисто емпирично до извода, че оптималната стъпка на намаляване (свиващ фактор) е 1,3. При достигане до стъпка 1 алгоритъмът вече е премахнал костенурките и се държи като обикновено мехурче. При стойности на стъпката, по-малки от 1,3, алгоритъмът се забавя, в резултат на нарастване броя на излишните сравнения, а при по-големи — поради улучване на малко костенурки. Двамата опитно установяват още, че разделянето на масива на области, всяка от които се обработва с различна стъпка, както и използването на експоненциално намаляващ или динамично изменящ се свиващ фактор, не водят до подобрение.

Все пак те успяват да направят известно подобрение, което формулират като *"Правило 11"*. Идеята е вместо стъпки със стойност 9 и 10 да се използва 11. При свиващ фактор, близък до оптималния 1,3, стъпката може да намалява към 1 по следните три начина:

9 6 4 3 2 1
10 7 5 3 2 1
11 8 6 4 3 2 1

При третия начин малките костенурки изчезват още преди стъпката да стане 1. В същото време при първите два с вероятност 8% списъците продължават да съдържат малки костенурки, в резултат на което алгоритъмът се забавя с 15-20%.

Така описаният алгоритъм поразително напомня сортирането по Шел (*виж 3.1.3.*) — и двата използват свиващ фактор. Не е трудно да се убедим обаче, че приликата е само външна. Алгоритъмът на Шел представлява *вмъкване* (*виж 3.1.2.*) с намаляваща стъпка, докато методът на зайците и костенурките се основава на идеите на *мехурчето* (*виж 3.1.4.*). Освен това при Шел на всяка стъпка се извършва *цялостно сортиране* на съответните подсписъци. Оттук произтичат и някои други различия. Докато при Шел оптималният свиващ фактор беше 1,7, при "зайците и костенурките" той е 1,3. И накрая, алгоритмичната сложност на Шел е $\Theta(n.(\log_2 n)^2)$, докато при зайците и костенурките е $\Theta(n.\log_2 n)$ както в средния, така и в най-лошия случай. Това нарежда предложения от Лейси и Бокс метод сред най-бързите съвременни методи за сортиране. Следва примерна реализация:

```
void combSort(struct CElem m[], unsigned n)
{ unsigned s, i, j, gap = n;
  do {
    s = 0;
    gap = (unsigned) (gap/1.3);
    if (gap < 1)
      gap = 1;
    for (i = 0; i < n-gap; i++) {
      j = i + gap;
      if (m[i].key > m[j].key) {
        swap(m+i, m+j);
            s++;
      }
    }
  } while (s != 0 || gap > 1);
}
```

🖫 combsort.c

Като известен недостатък на сортирането по метода на зайците и костенурките можем да посочим работата с числа с плаваща запетая и използването на тежката операция деление. Делението лесно може да се избегне, като вместо това се използва умножение. За целта редът:

```
gap = (long)(gap/1.3);
```

може да се замени с:

```
gap = (long)(gap*0.76923076923);
```

Преминаването към целочислени операции пък става с:

```
gap = gap*8 / 11;
```

Бихме могли да премахнем делението, отдалечавайки се още повече от оптималната редица, но ускорявайки значително пресмятанията:

```
gap = gap*6 >> 3.
```

След публикуването на статията са правени редица опити за нейното подобрение, най-важното от които е на Джим Вийл. Той предлага използване на константата 1,279604943109628 вместо 1,3. Всъщност, той не я използва директно, а предлага стъпките да са предварително фиксирани. Следва неговата редица (използва се в намаляващ ред) за масиви с до 4 милиарда елемента:

11, 13, 17, 23, 29, 37, 47, 61, 79, 103, 131, 167, 216, 277, 353, 449, 577, 739, 947, 1213, 1553, 1987, 2543, 3259, 4166, 5333, 6829, 8741, 11177, 14310, 18313, 23431, 29989, 38371, 49103, 62827, 80407, 102881, 131648, 168463, 215573, 275840, 352973, 451669, 577957, 739560, 946346, 1210949, 1549547, 1982809, 2537202, 3246624, 2147483647

Задачи за упражнение:

1. Да се докаже, че методът на зайците и костенурките има сложност $\Theta(n.\log_2 n)$ както в средния, така и в най-лошия случай.
2. Да се сравнят теоретично и емпирично алгоритъмът на Шел (*виж 3.1.3.*) и методът на зайците и костенурките.

3.1.8. Сортиране чрез пряк избор

Друг основен метод за сортиране със сложност $\Theta(n^2)$ е методът на *прекия избор*, известен още като метод на *пряката селекция*. Масивът се разделя на сортирана и несортирана част, като на всяка стъпка на алгоритъма сортираната област се разширява отдясно с един елемент. На първата стъпка се намира минималният елемент на масива и се разменя с първия. На втората стъпка минималният измежду останалите елементи се разменя с втория елемент на масива и т.н. На всяка следваща стъпка минималният елемент от несортираната част на масива си разменя мястото с първия елемент от несортираната част (забележете, че той е по-голям или равен на всеки елемент от несортираната част, *защо?*), разширявайки по този начин сортираната част, докато тя не обхване целия масив.

```
void straightSelection(struct CElem m[], unsigned n)
{ unsigned i, j;
  for (i = 0; i < n-1; i++)
    for (j = i+1; j <= n; j++)
      if (m[i].key > m[j].key)
        swap(m+i, m+j);
}
```

🖫 selsort.c

Широко разпространен е и леко модифициран вариант на горната реализация: В програмата, която следва, разликата е, че във вътрешния цикъл е спестено адресирането на *j*-тия елемент като `m[j]` , като за целта е въведена променливата `x`.

```
void straightSelection(struct CElem m[], unsigned n)
{ unsigned i, j, ind;
  struct CElem x;
  for (i = 0; i < n - 1; i++)
    for (x = m[ind = i], j = i + 1; j < n; j++)
      if (m[j].key < x.key) {
        x = m[ind = j];
        m[ind] = m[i];
        m[i] = x;
      }
}
```

🖫 selsort2.c

Пряк избор	Минимален брой	Среден брой	Максимален брой
сравнения	$n(n-1)/2$	$n(n-1)/2$	$n(n-1)/2$
размени	$3(n-1)$	$n\ln n$	$[n^2/4]+3(n-1)$

Таблица. 3.1.8. Сравнения и размени при сортирането по метода на прекия избор.

Подобно на сортирането по метода на мехурчето алгоритъмът на прекия избор винаги извършва $n(n-1)/2$ сравнения, както се вижда от *таблица 3.1.8.* По принцип алгоритъмът на прекия избор е за предпочитане пред метода на мехурчето, макар в частните случаи на предварително сортиран или почти сортиран масив мехурчето да дава по-добри резултати.

Задачи за упражнение:

1. Да се докаже, че при сортиране с пряк избор всеки елемент от несортираната част е по-голям или равен от всеки елемент път сортираната.
2. Да се сравнят теоретични и емпирично двата варианта на сортирането с пряк избор.
3. Да се изведат формулите от *таблица 3.1.8.*

3.1.9. Пирамидално сортиране на Уилямс

Сортирането чрез пряк избор (*виж 3.1.8.*) се нарежда сред най-неефективните, подобно на *мехурчето* (*виж 3.1.4.*) и *прякото вмъкване* (*виж 3.1.2.*). Основен негов недостатък е, че на *i*-тата стъпка винаги се извършват точно $n - i$ сравнения, независимо от входните данни.

Очевидно произволен алгоритъм за намиране на минималния (максималния) елемент на *n*-елементен масив изисква точно $n-1$ сравнения. Действително, при всяко сравнение се отхвърля точно един кандидат за оптималност, а за да се определи минималният елемент, е необходимо да бъдат отхвърлени точно $n-1$ кандидата.

Така, на първата стъпка на произволен алгоритъм, основаващ се на избор, ще се извършват $n-1$ сравнения и този резултат, съгласно направените по-горе разсъждения, не би могъл да се подобри. На пръв поглед изглежда, че на втората стъпка ще бъдат необходими още $n-2$ сравнения. Разсъждавайки индуктивно, получаваме че на *i*-тата стъпка ще ни трябват точно $n-i$ сравнения. Не бихме ли могли да подобрим тази оценка? Внимателният читател сигурно е забелязал, че първата и втората стъпка на алгоритъма не са съвсем равнопоставени. Действително, в резултат на прилагането на първата стъпка получаваме не само минималния елемент на масива, но и редица съотношения между двойки измежду останалите елементи, които можем да съхраним и да използваме на втората стъпка. Ясно е, че колкото повече допълнителни съотно-

шения получим, толкова по-малко сравнения ще ни бъдат необходими на следващите стъпки, т. е. колкото по-ниско и по-разклонено е двоичното дърво на сравненията (*виж 3.1.1.*), толкова повече информация ще ни носи то.

Едно нелошо дърво на сравненията можем да получим, използвайки механизма на *турнира с елиминиране* (*виж* [*Knuth-3/1968*], [*Рейнгольд*, *Нивергельт*, *Део-1980*]), построяващ дървото на сравненията от листата към корена. В първия кръг на турнира се играят срещите $x_1 : x_2$, $x_3 : x_4$, ..., $x_{n-1} : x_n$. Във втория кръг играят двойки победители от предишния кръг и т.н. На финала се срещат двама участници и определят шампиона на турнира. В случай че в *i*-ия кръг на турнира броят на участниците е нечетен (а това ще се случи поне веднъж, ако n не е степен на 2, *защо?*), то един от играчите почива, като автоматично се класира за следващия кръг. Използвайки посочената стратегия за определяне на минималния елемент на масив, получаваме: В първия кръг на турнира с $n/2$ сравнения се определя минималният елемент за всяка двойка, а с още $n/4$ сравнения се определя минималният елемент измежду две двойки (т. е. 4 елемента) и т.н.

Очевидно изложеният алгоритъм построява дървото на турнира, т. е. определя минималния елемент на масива с точно $n-1$ сравнения. Сега, заменяйки минималния елемент (шампиона) с $-\infty$ в съответното му листо и преизчислявайки съдържанието на върховете по пътя до върха, във върха на дървото получаваме следващия по големина елемент. Тъй като дървото има височина $\lceil \log_2 n \rceil$, то описаният процес на определяне на “вицешампиона", т. е. на втория по големина елемент на масива, изисква $\lceil \log_2 n \rceil - 1$ сравнения вместо $n-2$, както беше при алгоритъма на прекия избор. Така процесът на цялостно сортиране на масива изисква не повече от $n-1+(n-1)(\lceil \log_2 n \rceil - 1)$ сравнения.

Макар изложеният алгоритъм да представлява едно наистина сериозно подобрение по отношение на алгоритъма на прекия избор, все още има какво да се желае. На първо място следва да се определи ефективен метод, минимизиращ броя на извършваните размени. Да си припомним, че размяната е многократно по-бавна операция от сравнението и проблемът не бива да се подценява: именно прекомерният брой размени не позволи на сортирането чрез клатене (*виж 3.1.5.*) да постигне по-добра асимптотична ефективност по отношение на сортирането по метода на мехурчето (*виж 3.1.4.*), въпреки силно редуцирания брой на извършваните сравнения. Друг източник на неприятности са стойностите $-\infty$, които водят до излишни сравнения, като накрая запълват цялото дърво. Освен това, алгоритъмът на турнира с елиминиране, макар и да строи *идеално балансирано дърво* (*виж 2.4.*), не строи онова дърво, при което теоретично биха могли да се очакват оптимални резултати, тъй като позволява листата да се появяват на *всички нива*. И накрая, но не на последно място, участниците в турнира са разположени в листата, като върховете на дървото на сравненията само ги дублират. Естествено е желанието да минимизираме размера на необходимата памет, която при алгоритъма на турнира с елиминиране е от порядъка на $2n-1$. За целта, ще ни бъде необходим ефикасен механизъм за линеаризация на дървото, който да позволи да пазим *само листата*, без да губим известните съотношения между тях. Дж. Уилямс предлага ефективно решение на всички изложени проблеми чрез структурата *пирамида*.

Дефиниция 3.1. *Пирамида* ще наричаме идеално балансирано двоично дърво с височина h, едновременно удовлетворяващо следните условия:

1) всички листа са на нива h и $h-1$;
2) всички наследници на даден връх са по-малки от него;
3) всички наследници от ниво h са максимално изместени вляво.

Интересното тук е, че пирамидата позволява ефективна реализация не само чрез дърво, но и чрез n-елементен масив, като наследствените връзки между елементите се определят като проста линейна функция на позициите им. Следващото твърдение ни дава конкретен механизъм за реализация на описаното представяне:

Твърдение. *Пирамидата* е еквивалентна на редицата от ключове $h_l, h_{l+1}, \ldots, h_r$ $(1 \le l \le r \le n)$, за които $h_i \ge h_{2i}$ и $h_i \ge h_{2i+1}$, $i = l, l+1 \ldots, r/2$.

Ясно е, че върхът h_1 на пирамидата $h_1, h_2, ..., h_n$ съдържа най-големия ѝ елемент. (*Защо?*) Всъщност, Твърдението умишлено е дадено в по-общ вид, по-долу ще стане ясно защо. В частност, при $l = 1$ еквивалентността е очевидна. Ще отбележим само, че h_{2i} и h_{2i+1} са съответно левият и десният наследник на h_i.

Нека предположим, че разполагаме с ефективни алгоритми за изграждане на пирамида `buildHeap()`, както и за възстановяване на пирамида от k елемента `restoreHeap(k)`, след като върхът ѝ е бил заменен с произволен елемент. Тогава, след като сме изградили пирамида с n елемента, можем да разменим върха ѝ h_1 (най-големият ѝ елемент) с последния ѝ елемент h_n, при което старият връх заема окончателната си позиция: последен в сортирания масив. Тъй като тази операция разрушава условието на Твърдението, се налага отсяване на новия връх надолу по пирамидата. В резултат получаваме нова пирамида $h_1, h_2, ..., h_{n-1}$, съдържаща един елемент по-малко. Следва нова размяна на върха на пирамидата с последния ѝ $(n-1)$-ви елемент, последвана от ново отсяване. Описаният процес се повтаря $n-1$ пъти (няма нужда от n-та стъпка, защото накрая n-ият елемент си е на място, *защо?*). Получаваме следния алгоритъм за пирамидално сортиране на масив:

```
buildHeap();
for (i = n; i >= 2; i--) {
  swap(m+1, m+i);
  restoreHeap(i-1)
}
```

Как да изградим пирамидата? Нека за момента приемем, че n е четно. Тогава съгласно Твърдение за елементите $h_{n/2+1}, h_{n/2+2}, ..., h_n$ не се изисква никакво съотношение на наредба, тъй като за никои два елемента i и j не е изпълнено $j = 2i$ или $j = 2i + 1$. Действително, посочените елементи (и само те!) са листата на дървото (разположени са на последно или евентуално предпоследно ниво) и за тях не се изисква да удовлетворяват някакви съотношения. Нека разширим пирамидата отляво с един елемент. Новодобавеният елемент x ще бъде от предпоследното ниво и ще има поне един наследник. Това налага извършването на проверка дали не са нарушени съотношенията от Твърдението, последвана евентуално от размяна на x с един от двата му наследника. Сега елементите $h_{n/2}, h_{n/2+1}, ..., h_n$ отново образуват пирамида. Следва ново разширяване на пирамидата отляво, като процесът продължава до включване на всички елементи. При нечетно n листата са $h_{n/2+1}, h_{n/2+2}, ..., h_n$. На практика, в реализацията на Си няма нужда от разграничаване на двата случая, тъй като операцията / е целочислена.

Ще отбележим, че отсяването на новодобавения елемент x надолу по пирамидата може да протича на няколко етапа, т. е. след размяна на x с някой от наследниците му може отново да се окаже, че x не удовлетворява условията на Твърдението и т.н. Така x се отсява надолу по пирамидата, като на всяка стъпка потъва едно ниво надолу, докато не заеме мястото си. Тук може да се направи едно очевидно подобрение, ускоряващо процеса, а именно x да се разменя с *по-малкия* от двата си наследника, макар това не винаги да носи полза. Освен това размяната на x с наследниците му може да се отлага дотогава, докато не бъде намерено точното му място, като вместо това в процеса на търсене на крайната му позиция се извършват еднопосочни присвоявания.

Забележете, че както построяването, така и възстановяването изискват отсяване на върха *надолу* по пирамидата. Това ни навежда на мисълта за реализиране на обща функция за отсяване `sift()`:

```
/* Отсяване на елемента от върха надолу по пирамидата */
void sift(struct CElem m[], unsigned l, unsigned r)
{ unsigned i = l,
           j = i + i;
  struct CElem x = m[i];
  while (j <= r) {
      if (j < r && m[j].key < m[j+1].key)
        j++;
      if (x.key >= m[j].key)
```

```
          break;
        m[i] = m[j];
      i = j;
      j <<= 1; /* еквивалентно на j *= 2; */
    }
    m[i] = x;
}
```

🖫 heapsort.c

Сега построяването на пирамидата (функция `buildHeap()`) може да се извърши с помощта на следния програмен фрагмент:

```
for (k = n/2 + 1; k > 1; k--)
  sift(m,k-1,n);
```

За сортирането — многократна размяна на върха на пирамидата с последния ѝ елемент, последвана от отсяване (функция `restoreHeap()`), получаваме:

```
for (k = n; k > 1; k--) {
  swap(m+1,m+k);
  sift(m,1,k-1);
}
```

Окончателно за пирамидалното сортиране получаваме:

```
void heapSort(struct CElem m[], unsigned n) /* Пирамидално сортиране */
{ unsigned k;

  /* 1. Построяване на пирамидата */
  for (k = n/2 + 1; k > 1; k--)
    sift(m,k-1,n);

  /* 2. Построяване на сортирана последователност */
  for (k = n; k > 1; k--) {
    swap(m+1,m+k);
    sift(m,1,k-1);
  }
}
```

🖫 heapsort.c

Задачи за упражнение:

1. Да се докаже, че при турнир по тенис с пряко елиминиране, ако броят на участниците не е степен на 2, поне в един кръг ще трябва да има почиващ играч.
2. Да се докаже, че върхът на пирамидата съдържа най-големия ѝ елемент.
3. Да се разработи метод за сортиране, използващ *тернарна пирамида*. Тернарната пирамида е обобщение на класическа пирамида и се основава на пълни тернарни (троични) дървета. Следва ли да се очаква ускорение в сравнение с класическата пирамида?
4. Да се разработи алгоритъм за бързо обединение на две пирамиди.

3.1.10. Минимална времева сложност на сортирането чрез сравнение

Всички разгледани досега алгоритми принадлежат към класа на алгоритмите за *сортиране чрез сравнение* (т. е. такива, при които единствената позволена операция е сравнението между двойки елементи с помощта на операциите > (≥), < (≤) и = (≠). По-простите от тях (например вмъкване, мехурче или пряк избор, *виж 3.1.2., 3.1.4 и 3.1.8.*) извършват $\Theta(n^2)$ на брой

сравнения, докато най-добрите имат сложност $\Theta(n.\log_2 n)$ в средния (бързо сортиране, *виж 3.1.6.*) или дори в най-лошия случай (пирамидално сортиране: *виж 3.1.9.*, метод на "зайците и костенурките": *виж 3.1.7.* и сортиране чрез сливане, ще бъде разгледано в *7.3.*). При това, за всеки такъв алгоритъм може да се намери входна последователност, за която тези оценки *да се достигат*. Т. е. най-добрите познати ни до момента алгоритми за сортиране имаха алгоритмична сложност $\Omega(n.\log_2 n)$. Дали тази оценка не би могла да се подобри? Отговор на този въпрос ни дава следното:

Твърдение. Всеки алгоритъм за сортиране чрез сравнение извършва брой сравнения от порядъка на $\Omega(n.\log_2 n)$.

Доказателство. На всеки алгоритъм за сортиране чрез сравнение може еднозначно да се съпостави съответно двоично дърво на сравненията, подобно на това от *фигура 3.1.1*. Листата са $n!$ на брой и всяко от тях съдържа точно една от пермутациите на входната последователност. Нелистата от своя страна съдържат сравнения между двойка елементи. Без ограничение на общността можем да считаме, че всички сравнения са от вида <. Действително за всяка от релациите =, ≠, ≤, <, > и ≥ имаме, че тя или е в сила, или не е, т. е. всяко такова сравнение има два изхода, на които съответстват два наследника в дървото. Изпълнението на алгоритъма за сортиране се свежда до търсене на път в двоично дърво. Търсенето започва от корена и завършва при достигане на листо. Пермутацията, съдържаща се в листото, определя търсената наредба на входната последователност.

Както по-горе беше посочено, сложността на даден алгоритъм за сортиране в най-лошия случай се определя от максималния брой сравнения, извършвани от него, т. е. от дължината на максималния път от корена до някое от листата. За да сортира правилно *всяка* входна последователност от n елемента, алгоритъмът трябва да може да може да дава като резултат всяка една от възможните $n!$ пермутации, т. е. всяка пермутация трябва да се появява като листо в дървото на сравненията му *поне веднъж*.

Да означим с h височината на дървото (т. е. дължината на максималния път от корена до някое от листата му). В сила е неравенството:

$$n! \leq 2^h$$

Оттук след логаритмуване получаваме:

$h \geq \log_2 (n!)$

За да логаритмуваме $n!$, ще използваме апроксимацията на Стирлинг $n! > (n/e)^n$, при което получаваме:

$$h \geq \log_2 (n!) \geq \log_2 (n/e)^n = n.\log_2 n - n.\log_2 e,$$

откъдето директно следва, че $h \in \Omega(n.\log_2 n)$. Т. е. височината на дървото, а оттук и минималният брой необходими сравнения, е $\Omega(n.\log_2 n)$. □

От горното твърдение веднага следва, че пирамидалното сортиране, методът на зайците и костенурките и сортирането чрез сливане са асимптотично оптимални. Тук ще направим уговорката, че при някои по-силни ограничителни условия (отнасящи се главно до разпределението на стойностите на ключовете и/или вътрешното им представяне) има и по-ефективни алгоритми, някои от които ще разгледаме в следващия *параграф 3.2*.

Задача за упражнение:

Да се докаже, че всяко подмножество на множество с линейна наредба също е линейно наредено.

3.2. Сортиране чрез трансформация

Съществува и друг широк клас алгоритми, основаващи се не на сравнение, а на извършването на операции (главно аритметични) над елементите на множеството — *сортиране чрез трансформация*. Разликата между двата подхода е сравнима с разликата между търсене в *двоично дърво* (*виж 2.3.*) и *хеширане* (*виж 2.5.*). Обикновено алгоритмите за сортиране чрез трансформация са приложими само в някои частни случаи при допълнителни ограничителни условия по отношение на типа и/или множеството от допустими стойности и/или броя на срещанията на ключовете на сортираните елементи.

3.2.1. Сортиране чрез множество

Първият представител на класа на алгоритмите за сортиране чрез трансформация, който ще разгледаме, е алгоритъмът за *сортиране чрез множество*. Приложим е върху множество, чиято функция на нареждане f удовлетворява едновременно следните условия (Обозначаваме множеството с M, а броя на елементите му — с n):

1) стойностите на f са естествени числа от даден интервал $[a, b]$, съдържащ $m = b - a + 1$ цели числа. Не е задължително всяко цяло число от $[a, b]$ да бъде стойност на f, но всички стойности на f трябва да са от $[a, b]$.
2) f е *инективна*, т. е. за $x_1, x_2 \in M$ и $x_1 \neq x_2$ имаме $f(x_1) \neq f(x_2)$ (т. е. няма повторения)

С цел опростяване на алгоритъма няма да работим с елементи от тип `struct CElem`, а ще използваме тип `unsigned`, като ще предполагаме, че стойността на максималния елемент не надвишава някаква разумна константа `MAX_VALUE`. (Всъщност, `MAX_VALUE` е програмно обозначение на m) Така ще работим с интервала [0, `MAX_VALUE`). Сортирането става линейно посредством едно преминаване през елементите на масива, последвано от обхождане на числата от интервала. В началото инициализираме с 0 елементите на множество `set[]` (от тип `char`, който ще третираме като булев), след което преминаваме през масива и за всеки негов елемент установяваме в 1 съответния елемент на множеството. На втората стъпка извършваме проверка последователно за всички числа от интервала дали принадлежат на `set[]`. В началото на втория пас считаме, че `m[]` не съдържа нито един елемент и го изграждаме наново, този път вече като сортирана последователност. Подробностите се виждат от приложения програмен фрагмент.

```
void setSort(unsigned m[], unsigned n)
{ char set[MAX_VALUE];
  unsigned i,j;
  /* 0. Инициализиране на множеството */
  for (i = 0; i < MAX_VALUE; i++)
    set[i] = 0;

  /* 1. Формиране на множеството */
  for (j = 0; j < n; j++) {
    assert(m[j] >= 0 && m[j] < MAX_VALUE);
    assert(0 == set[m[j]]);
    set[m[j]] = 1;
  }
  /* 2. Генериране на сортирана последователност */
  for (i = j = 0; i < MAX_VALUE; i++)
    if (set[i])
      m[j++] = i;

  assert(j == n);
}
```

🖫 set_sort.c

Сложността на горния алгоритъм е $\Theta(m+n)$ и при стойности на n, близки до m, би могла да се разглежда като линейна относно n. Следва обаче да отбележим, че това не е "безплатно": От една страна имаме тежко допълнително предположение за стойностите и типа на ключовете на сортираните елементи (така не можем да сортираме реални числа, например), а от друга — ползваме допълнителна памет от порядъка на $\Theta(m)$. Ще отбележим, че повечето разгледани по-горе алгоритми използваха *константна* допълнителна памет. За някои рекурсивни алгоритми, например бързо сортиране (*виж 3.1.6.*), беше необходима логаритмична памет, а в най-лошия случай дори линейна (това е скрито зад механизма на рекурсията). Тук обаче е налице още по-голямо изискване: не $\Theta(n)$, а $\Theta(m)$, като $m > n$ и е възможно $m >> n$ (т.е. m е *много по-голямо* от n).

Бихме могли да обобщим алгоритъма, разширявайки сферата му на приложение. Ще "се отървем" от неприятното ограничение да работим само с цели числа. Този път ще използваме елементи от тип `struct CElem`, за които ключът удовлетворява горните две условия. Сложността на този алгоритъм отново ще бъде $\Theta(m+n)$, където m е броят цели числа в интервала $[a, b]$.

Този път при организирането на множеството ще се наложи да пазим информация не само за това дали дадено число от $[a, b]$ се среща като ключ на елемент от `m[]`, но и *стойността* на този елемент. Съществуват най-общо два подхода в това отношение. При първия подход в множеството се пази индексът на съответния елемент в `m[]`:

```
struct CSetEl {
  char found;
  unsigned index;
} m[MAX];
```

Полето `found` може да се спести, ако се приеме че нулевият елемент на `m[]` не се използва. Нулата ще означава отсъствието на елемент с такъв ключ, а всяка друга ненулева стойност ще бъде индекс в `m[]` на елемент със съответен ключ:

```
unsigned m[MAX];
```

Впрочем, от гледна точка на езика Си, е по-добре да ползваме за служебна стойност не 0, а напротив — най-голямата стойност на типа `unsigned`. Това е различна стойност за различните платформи (например под *DOS* е 65 535, а под *Windows* — 4 294 967 295). Ще използваме споменатия в *глава 1* подход за получаването ѝ — изхождайки от вътрешното представяне на числата в допълнителен код: `(unsigned)(-1)`.

Предимство на горните две декларации е пестеливостта, доколкото това е възможно, по отношение на използваната памет — пазят се не елементите на `m[]`, които могат да бъдат големи по размер записи, а само техните индекси. Забележете двата `assert()`-а, пазещи ни както от повторения, така и от непозволени стойности на ключа.

```
#define NO_INDEX (unsigned)(-1)
/* Сортира масив с използване на множество */
void setSort(struct CElem m[], unsigned n)
{ unsigned indSet[MAX_VALUE]; /* Индексно множество */
  unsigned i,j;
  /* 0. Инициализиране на множеството */
  for (i = 0; i < MAX_VALUE; i++)
    indSet[i] = NO_INDEX;
  /* 1. Формиране на множеството */
  for (j = 0; j < n; j++) {
    assert(m[j].key >= 0 && m[j].key < MAX_VALUE);
    assert(NO_INDEX == indSet[m[j].key]);
```

```
    indSet[m[j].key] = j;
  }
  /* 2. Генериране на сортирана последователност */
  for (i = j = 0; i < MAX_VALUE; i++)
    if (NO_INDEX != indSet[i])
      do4Elem(m[indSet[i]]);
}
```

🖫 setsort2.c

Приложената програма решава частично задачата. Тя получава сортираната последователност на елементите, но не в масива `m[]`. Вместо това извиква функцията `do4Elem()` за всеки следващ елемент от сортираната последователност. Това е приемливо, ако еднократно искаме да извършим някаква операция върху елементите. Например, да ги изведем в сортиран вид.

Запазването на сортираната последователност би довела до разрушаване на `m[]`, а оттук и на съответните индекси. Един възможен изход е предварително запазване на копие на `m[]`. Бихме могли да си спестим проблемите по поддържането на копие, запазвайки в множеството съответните елементи на `m[]`. Последното, разбира се, е свързано с допълнително разхищение на памет — по един запис от тип `struct CElem` за всяко число от интервала $[a, b]$. Вземайки предвид, че размерът на сортираните записи е фиксиран, получаваме, че размерът необходима допълнителна памет отново е $\Theta(m)$, като този път зад нотацията $\Theta(...)$ се крие по-голяма константа.

Задача за упражнение:

Да се модифицира предложената реализация на сортиране чрез множество така, че сортираната последователност да се получава в изходния масив.

3.2.2. Сортиране чрез броене

Да се опитаме да доразвием идеята на сортирането чрез множество. Основен недостатък на метода беше изискването за уникалност на ключа, т. е. не се допускаха повторения. Да предположим, че искаме да сортираме елементите на множество от естествени числа от даден интервал, всяко от които може да се среща не повече от k на брой пъти. Тук елементите на масива не са от тип запис и ролята на ключ се поема от идентитета, подобно на случая на сортиране на цели числа с множество. Дефинираме масив `cnt[]`, като `cnt[i]` съдържа броя на срещанията на числото `i`. Започваме с нулиране на елементите на `cnt[]`, след което преминаваме през елементите на `m[]` и намираме броя срещания за всяко цяло число от $[a, b]$. На втората стъпка преминаваме през елементите на `cnt[]`, включвайки в сортираната последователност `cnt[x]` на брой пъти числото x, за всяко цяло $x \in [a, b]$.

```
void countSort(unsigned m[], unsigned n) /* Сортира чрез броене */
{ unsigned char cnt[MAX_VALUE];
  unsigned i,j;

  /* 0. Инициализиране на множеството */
  for (i = 0; i < MAX_VALUE; i++)
    cnt[i] = 0;

  /* 1. Формиране на множеството */
  for (j = 0; j < n; j++) {
    assert(m[j] >= 0 && m[j] < MAX_VALUE);
    cnt[m[j]]++;
  }

  /* 2. Генериране на сортирана последователност */
  for (i = j = 0; i < MAX_VALUE; i++)
```

```
    while (cnt[i]--)
      m[j++] = i;
  assert(j == n);
}
```

🖫 count_s.c

Горният алгоритъм работи добре при предположението, че всяко число се среща не повече от k пъти, за някаква разумна предварително дефинирана константа k. Теоретично областта на приложение на горния алгоритъм остава ограничена от следните изисквания:

- елементите да бъдат от интервала $[a, b]$, като не е задължително всяко число от $[a, b]$ да бъде стойност на f, но всички стойности на f трябва да бъдат от $[a, b]$.
- всяко число от $[a, b]$ да се среща не повече от k пъти;
- да се сортират само числа, а не произволни записи.

Първите две ограничения при горния алгоритъм са естествени. Впрочем подобни ограничения са характерни за почти всички алгоритми за сортиране чрез трансформация. Последното изискване обаче не е толкова фундаментално и лесно бихме могли да се освободим от него. Основният проблем е необходимостта да се пази информация не само за това дали дадено цяло число x от $[a, b]$ се среща като ключ на елемент от M, но и стойността на този елемент. Проблема решаваме сравнително леко: ще пазим информация за това дали ключът x се среща и отделно от това — съответния елемент с ключ x.

При сортирането чрез броене за всяка допустима стойност на ключа може да има няколко съответни елемента, което усложнява нещата: налага се да се организира списък. Тъй като броят на срещанията на даден ключ не надвишава k, то всеки такъв списък може да съдържа най-много k елемента. Заделянето на статична памет с размер k за всеки ключ p обаче води до излишно разхищение.

В приложената по-долу програма с всеки ключ е асоцииран динамичен списък, съдържащ съответните елементи на M. Поддържането на единствен указател — към началото на списъка, може да бъде потенциален източник на проблеми, тъй като прави сортирането *неустойчиво* (списъкът е едносвързан). Така елементите с еднакви ключове попадат в изходната последователност в ред, обратен на реда, в който са били в M. Последното лесно би могло да се поправи, например чрез добавяне на допълнителен указател към края на списъка (или пък да се отчете по-късно, при сглобяване на сортираната последователност: да се обърнат всички списъци). Сложността на алгоритъма отново е $\Theta(m+n)$.

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

#define MAX 100
#define FACTOR 5
#define MAX_VALUE (MAX*FACTOR)
#define TEST_LOOP_CNT 100

struct CElem {
  int key;
  /* Някакви данни */
};

struct CList {
  struct CElem data;
  struct CList *next;
};
void init(struct CElem m[], unsigned n)
```

```
{ unsigned i;
  srand(time(NULL));
  for (i = 0; i < n; i++)
    m[i].key = rand() % MAX_VALUE;
}

void countSort(struct CElem m[], unsigned n)
{ /* Сортира масив по метода на броенето. Внимание - неустойчив метод! */
  unsigned i,j;
  struct CList *lst[MAX_VALUE], *p;

  /* 0. Начално инициализиране */
  for (i = 0; i < MAX_VALUE; i++)
    lst[i] = NULL;

  /* 1. Разпределяне на елементите по списъци */
  for (j = 0; j < n; j++) {
    /* 1.1. Проверка за ключа */
    assert(m[j].key >= 0 && m[j].key < MAX_VALUE);
    /* 1.2. Добавяне на елемента в началото на списъка */
    p = (struct CList *) malloc(sizeof(struct CList));
    p->data = m[j];
    p->next = lst[m[j].key];
    lst[m[j].key] = p;
  }

  /* 2. Извеждане на ключовете на сортираната последователност */
  for (i = j = 0; i < MAX_VALUE; i++)
    while (NULL != (p = lst[i])) {
      m[j++] = lst[i]->data;
          lst[i] = lst[i]->next;
          free(p);
    }
}

void print(struct CElem m[], unsigned n)
{ unsigned i;
  for (i = 0; i < n; i++)
    printf("%8d", m[i].key);
}

int main(void) {
  struct CElem m[MAX];
  init(m, MAX);
  printf("Масивът преди сортирането:\n");
  print(m,MAX);
  countSort(m, MAX);
  printf("Масивът след сортирането:\n");
  print(m,MAX);
  return 0;
}
```

🖫 count_s2.c

Задача за упражнение:

Да се реализира устойчив вариант на сортирането чрез броене.

3.2.3. Побитово сортиране

Въпреки направените подобрения, сортирането чрез броене (*виж 3.2.2.*) наследява повечето недостатъци на сортирането чрез множество (*виж 3.2.1.*) и едва ли би могло да претендира поне отчасти за универсалност. Основен проблем на алгоритъма е необходимостта от поддържане на допълнителен масив с размер, пропорционален на броя на възможните стойности на ключовете m. Ясно е, че m може да бъде достатъчно голямо число и да не позволява заделяне на необходимия размер памет. На практика голяма част от възможните ключове могат да не се срещат, докато за други ключове може да се получи значително струпване на елементи, което води до неефективно използване на паметта.

Сортирането чрез броене преминава веднъж през елементите на множеството, което изисква време $\Theta(n)$, и веднъж — през множеството от допустими стойности за ключовете, за време $\Theta(m)$. Общата алгоритмична сложност става $\Theta(m+n)$. Полученият резултат показва, че алгоритъмът е линеен едновременно по m и n. Доколко е линеен тогава? Ясно е, че при $m > n$ алгоритъмът за сортиране чрез броене е линеен по броя на възможните стойности на ключовете m. За нас обаче е по-важно, каква е сложността му по отношение на броя на елементите n. Не е трудно да се забележи, че при $m = n^3$, сложността на алгоритъма става $\Theta(n^3)$, т. е. той се оказва по-лош дори от сортирането по метода на мехурчето! А ако $m = n^7$? Получаваме отчайваща неефективност…

Макар това да не личи на пръв поглед, идеята на сортирането чрез броене е добра и може да ни доведе до истински линеен алгоритъм за сортиране. (Разбира се, отново при някои ограничения…) В началото ще се спрем на алгоритъма за *побитово сортиране*, след което ще разгледаме и неговото естествено обобщение — *методът на бройните системи* (*виж 3.2.4.*).

Идеята на *побитовото сортиране* до голяма степен се основава на двоичното вътрешно представяне на числата в компютъра. Нека е зададено множество от цели числа без знак, чиито елементи ще сортираме. Разделяме числата в два списъка в зависимост от стойността на най-младшия им двоичен разряд. Четните числа попадат в първия списък, а нечетните — във втория. Следва добавяне на втория списък към края на първия, при което получаваме общ списък. За разлика от сортирането чрез броене тук новият елемент попада задължително *в края* на съответния списък. Смисълът на това уточнение ще стане ясен по-нататък. Повтаряме операцията с предпоследния бит, след това с предпредпоследния и т.н. Процесът приключва след извършване на операцията с най-старшия бит. Не е трудно да се забележи, че така описаният процес действително води до получаване на сортирана последователност (*Защо?*). Сега вече става ясно и посоченото по-горе изискване. То ни гарантира, че всяка следваща стъпка ще извлича полза от предишните, като в случай на равни стойности на съответния бит ще запазва получената на предните стъпки наредба. Така всяка стъпка се оказва устойчива, а оттук и методът е устойчив.

Изхождайки от вътрешното представяне на данните в компютъра, не е трудно да се съобрази, че описаният метод е приложим, с минимални модификации, за сортиране на произволен тип данни — сортирането на по-сложни структури като масиви, записи и низове практически се свежда до сортиране на числа. Все пак следва да се отбележи, че това не винаги е толкова просто. Така например, съществуват известни проблеми при сортиране на отрицателни числа (вътрешно се представят в допълнителен код), както и на числа с плаваща запетая.

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 100

struct CElem {
  int key;
  /* .............
     Някакви данни
     ............. */
};
```

```
struct CList {
  struct CElem data;
  struct CList *next;
};


struct CList* init(unsigned n) /* Запълва масива със случайни числа */
{ struct CList *head, *p;
  unsigned i;
  srand(time(NULL));
  for (head = NULL, i = 0; i < n; i++) {
    p = (struct CList *) malloc(sizeof(struct CList));
    p->data.key = rand();
    assert(p->data.key);
    p->next = head;
    head = p;
  }
  return head;
}

struct CList *bitSort(struct CList *head)
{ struct CList *zeroEnd, *oneEnd, *zero, *one;
  unsigned maxBit, bitPow2;

  /* 0. Определяне на максималната битова маска */
  maxBit = 1 << (8*sizeof(head->data.key)-1);

  /* 1. Фиктивен елемент в началото на списъците */
  zero = (struct CList *) malloc(sizeof(struct CList));
  one  = (struct CList *) malloc(sizeof(struct CList));

  /* 2. Сортиране */
  for (bitPow2 = 1; bitPow2 < maxBit; bitPow2 <<= 1) {

    /* 2.1. Разпределяне по списъци */
    for (zeroEnd = zero, oneEnd = one; NULL != head; head = head->next)
      if (!(head->data.key & bitPow2)) {
        zeroEnd->next = head;
          zeroEnd = zeroEnd->next;
      }
      else {
        oneEnd->next = head;
        oneEnd = oneEnd->next;
      }

    /* 2.2. Обединение на списъците */
    oneEnd->next = NULL;
    zeroEnd->next = one->next;
    head = zero->next;
  }

  /* 3. Освобождаване на паметта */
  free(zero);
  free(one);

  return head;
}
```

```
void print(struct CList *head)
{ for (; NULL != head; head = head->next)
    printf("%8d", head->data.key);
  printf("\n");
}

void check(struct CList *head)
{ if (NULL == head)
    return;
  for (; NULL != head->next; head = head->next)
    assert(head->data.key <= head->next->data.key);
}

void clear(struct CList *head)
{ struct CList *p = head;
  while (NULL != head) {
    head = (p = head)->next;
    free(p);
  }
}

int main(void) {
  struct CList *head;
  head = init(MAX);
  printf("Масивът преди сортирането:\n");
  print(head);
  head = bitSort(head);
  printf("Масивът след сортирането:\n");
  print(head);
  check(head);
  clear(head);
  return 0;
}
```

🖫 bitsort.c

При горната реализация данните са организирани в динамични списъци. По-добра ефективност би могла да се получи при статична реализация. За целта е достатъчен единствен допълнителен n-елементен масив, в който двата списъка нарастват един срещу друг. Това изисква модификация на процеса на обхождане на елементите на масива, тъй като, макар и да се сливат, двата списъка остават *разделени*. На следващата стъпка списъците ще растат от двата края на първоначалния масив и т.н. Накрая ще бъде необходима допълнителна стъпка по обръщане на втория списък, което ще изисква не повече от $n/2$ размени (в най-лошия случай вторият списък съдържа всички елементи). [*Шишков-1995*] [*ComputerNews-1994a*] [*TopTeam-1997*]

Не е трудно да се забележи, че побитовото сортиране е линейно и има сложност от порядъка на $C.n$, където C е броят на разредите на сортираните числа. В общия случай, при динамична реализация допълнителната необходима памет е константна.

Алгоритъмът може да се обърне и да се застави да преглежда битовете от най-старшия към най-младшия. Идеята е, подобно на бързото сортиране на Хоор, масивът да се раздели на два дяла: ляв — с най-старши бит 0, и десен — с най-старши бит 1, след което същата операция да се приложи върху всеки от дяловете, като този път разделянето става по следващия по старшинство бит. Рекурсивното разделяне приключва след разглеждане на най-младшия бит. За разлика от класическото бързо сортиране на Хоор тук за разделител се използва не елемент на масива, а числото 2^b, $b = 0,1,2,...$. Тъй като 2^b не се съдържа непременно в изходното множество, няма гаранция, че на текущата итерация някой елемент ще отиде в крайната си позиция. Така, при $i = j$ може да се получи допълнителна излишна размяна на елементи. (*Защо?*) Следва примерна реализация.

```
struct CElem m[MAX];

................

void bitSort2(int l, int r, unsigned bitMask)
{ int i,j;
  if (r > l && bitMask > 0) {
    i = l; j = r;
    while (j != i) {
      while (!(m[i].key & bitMask) && i < j) i++;
      while ((m[j].key & bitMask) && j > i) j--;
      swap(&m[i], &m[j]);
    }
    if (!(m[r].key & bitMask)) j++;
    bitSort2(l,j-1,bitMask >> 1);
    bitSort2(j,r,bitMask >> 1);
  }
}

................

int main(void) {
  ................
  bitSort2(0,MAX-1,1 << 8*sizeof(m[0].key) - 1);
  ................
  return 0;
}
```

🖫 bitsort2.c

Рекурсивният вариант на побитовото сортиране има интересно поведение: колкото е по-разбъркан масивът, толкова по-добре работи, като при силно разбъркан масив дори превъзхожда по скорост бързото сортиране на Хоор.

Горните реализации са примерни и предполагат, че стойностите на ключовете са относително равномерно разпределени в интервала. На практика това не винаги е така. Да предположим, че допускаме стойности на ключа от интервала 0÷65535, а всъщност те са в 0÷1000. Така, при прилагане на побитово сортиране, последните 6 стъпки на алгоритъма се оказват излишни, тъй като не променят по никакъв начин наредбата на елементите: всички числа имат 0 в съответния бит.

Задачи за упражнение:

1. Да се докаже, че побитовото сортиране сортира правилно всяка входна последователност.
2. Да се реализира статичен вариант на побитовото сортиране, при който се използва единствен допълнителен масив, в който двата списъка, изграждани при итеративната реализация, нарастват един срещу друг.
3. Да се сравнят итеративният и рекурсивният вариант на побитовото сортиране.

3.2.4. Метод на бройните системи

Изложеният по-горе алгоритъм дава отлични резултати. За пръв път реализирахме истински линеен алгоритъм за сортиране, редуцирайки доколкото е възможно допълнителните изисквания. И все пак… Какво би станало, ако използваме не два, а три списъка? А защо не 10? Или пък 16? Или пък дори 256? Очевидно, с увеличаване броя на списъците ще намалява константата *C*, т. е. броят на преминаванията през елементите на множеството.

Идеята на метода на бройните системи е аналогична на тази на побитовото сортиране. Нека имаме s на брой списъка. На първата стъпка елементите се разпределят по списъците в зависимост от остатъка, който дават при деление на s. При това остава в сила изискването добавянето да става *само в края* на списъците. Следва конкатенация на списъците по реда на нарастване на остатъците. Процесът се повтаря до изчерпване цифрите на числата.

Докато при побитовото сортиране разглеждахме *двоичните цифри* на числата, при изложения метод работим със стойностите на цифрите в *s-ичното им представяне*, т. е. разглеждаме числата като записани в *s-ична бройна система*. Оттук и името на алгоритъма — метод на бройните системи. В частност, при $s = 2$ получаваме алгоритъма на побитовото сортиране.

По принцип няма ограничения за броя на списъците. Все пак трябва да се внимава, тъй като по-големият брой списъци s ще влоши скоростта на сортиране за достатъчно малки n (например за $n < s$). За предпочитане е този брой да бъде степен на 2 с цел по-ефективно намиране на съответните остатъци.

Приложената програма работи с 16 списъка и числа с не повече от 8 шестнадесетични цифри. Началата и краищата на списъците са обединени в 16-елементен масив, като конкатенацията им се извършва със същата лекота, с която се извършваше и при побитовото сортиране. И тук, както при побитовото сортиране, можем да се откажем от използването на динамични списъци, заествайки ги с масиви. Конкатенацията отново би могла да бъде изключително проста. За целта, всеки масив се снабдява с указател към следващия го. Все пак това си има своята цена — необходима е допълнителна статична памет.

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX     100
#define BASE   16 /* Основа на бройната система */
#define POW2    4 /* 16 = 1 << 4 */
#define DIG_CNT 8 /* Брой цифри */
struct CElem {
  int key;
  /* Някакви данни */
};

struct CList {
  struct CElem data;
  struct CList *next;
};

struct CList *init(unsigned n) /* Запълва масива със случайни числа */
{ struct CList *head, *p;
  unsigned i;
  srand(time(NULL));
  for (head = NULL, i = 0; i < n; i++) {
    p = (struct CList *) malloc(sizeof(struct CList));
    p->data.key = rand();
    assert(p->data.key);
    p->next = head;
    head = p;
  }
  return head;
}

struct CList *radixSort(struct CList *head)
{ struct { struct CList *st, *en; } mod[BASE];
  unsigned i, dig, mask, shrM;
```

```
  /* 1. Инициализиране */
  for (i = 0; i < BASE; i++)
    mod[i].st = (struct CList *) malloc(sizeof(struct CList));

  /* 2. Сортиране */
  mask = BASE-1; shrM = 0;
  for (dig = 1; dig <= DIG_CNT; dig++) {

    /* 2.1. Инициализиране */
    for (i = 0; i < BASE; i++)
      mod[i].en = mod[i].st;

    /* 2.2. Разпределяне на елементите по списъци */
    while (NULL != head) {
      /* 2.2.1. Намиране на i-та цифра в BASE-ичния запис на числото */
      i = (head->data.key & mask) >> shrM;
      /* 2.2.2. Включване числото в съответния списък */
      mod[i].en->next = head;
      mod[i].en = mod[i].en->next;

      head = head->next;
    }

    /* 2.3. Обединяване на списъците */
    mod[BASE-1].en->next = NULL;
    for (i = BASE - 1; i > 0; i--)
      mod[i-1].en->next = mod[i].st->next;
    head = mod[0].st->next;

    /* 2.4. Изчисляване на новите маски */
    shrM += POW2; mask <<= POW2;
  }

  /* 3. Освобождаване на паметта */
  for (i = 0; i < BASE; i++)
    free(mod[i].st);
  return head;
}
void print(struct CList *head)
{ for (; NULL != head; head = head->next)
    printf("%8d", head->data.key);
  printf("\n");
}

void clear(struct CList *head)
{ struct CList *p = head;
  while (NULL != head) {
    head = (p = head) ->next;
    free(p);
  }
}

int main(void) {
  struct CList *head;
  head = init(MAX);
  printf("Масивът преди сортирането:\n");
  print(head);
  head = radixSort(head);
```

```
  printf("Масивът след сортирането:\n");
  print(head);
  clear(head);
  return 0;
}
```

🖫 radsort.c

Методът на бройните системи би могъл да се обобщи допълнително за числа, записани в произволна бройна система. Така например, записите могат да се разглеждат като съставни елементи, всеки от които е записан в своя собствена бройна система. Целият запис се оказва записан в *смесена* бройна система. Нека разгледаме например начина, по който измерваме времето: секунди, минути, часове, дни, седмици, години. Това е смесена бройна система с основи съответно $d_1 = 60$, $d_2 = 60$, $d_3 = 24$, $d_4 = 7$, $d_5 = 52$. Бихме могли да гледаме на тези пет елемента като на съставен ключ-запис, всеки от елементите на който е записан в своя собствена бройна система. Тогава на първата стъпка на алгоритъма ще разпределим елементите в 60 списъка в зависимост от броя на секундите, на втората — отново в 60 списъка, определени от броя на минутите, на третата — в 24 списъка, съответстващи на часовете и т.н.

Задача за упражнение:

Да се сравнят побитовото сортиране и методът на бройните системи.

3.2.5. Сортиране чрез пермутация

Друг интересен представител на методите за сортиране чрез трансформация е сортирането *чрез пермутация*. Сортирането чрез пермутация е приложимо при по-силни ограничителни условия от сортирането чрез броене (*виж 3.2.2.*) или чрез множество (*виж 3.2.1.*). При сортирането чрез множество за функцията на нареждане изисквахме да бъдат едновременно изпълнени следните две условия (Обозначаваме множеството с M, а броят на елементите му — с n):

1) стойностите на f са естествени числа от даден затворен интервал $[a, b]$, съдържащ m цели числа, $m = b - a + 1$. Не е задължително всяко цяло число от $[a, b]$ да бъде стойност на f, но всички стойности на f трябва да са от $[a, b]$.

2) f е *инективна*, т. е. за $x_1, x_2 \in M$, $x_1 \neq x_2$ имаме $f(x_1) \neq f(x_2)$ (т. е. няма повторения)

Тук налагаме още едно допълнително изискване:

3) $f: M \rightarrow M$ е *сюрективна*, т. е. за всяко $s \in M$ съществува $x \in M$, такова че $f(x) = s$

Нека броят на елементите на M е n, т.е. $|M| = n$. Означаваме с S множеството $\{1, 2, ..., n\}$. Тогава горните три изисквания биха могли да се формулират още така:

1) $f : S \rightarrow S$
2) f е *биекция* (т. е. инективна и сюрективна едновременно)

Съгласно тези условия функцията на нареждане f действа върху елементите на множеството M като пермутация. Това ни дава ефективен линеен алгоритъм, позволяващ сортиране елементите на M на място без използване на допълнителен масив, какъвто ни беше необходим при сортирането чрез множество или чрез броене.

В процеса на сортиране не се извършва пряко сравнение на ключовете на различните елементи. Единствената проверка е дали елементът с ключ i е на i-та позиция. В случай, че на позиция i стои елемент с ключ j, $j =$ `m[i]`, $i \neq j$, ще разменим i-тия и j-тия елементи, при което елементът с ключ j ще отиде на j-та позиция. Ако след размяната на i-тата позиция стои елемент с ключ i, то преминаваме към разглеждане на следващия (i+1)-ви елемент на `m[]`. В противен случай извършваме нова размяна: с елемента с индекс `m[i]`. Лесно се вижда, че този процес е краен и в даден момент елементът с ключ i ще дойде на i-та позиция. (*Защо?*)

Ще илюстрираме описания метод върху пермутацията 4375612. Започваме с $i = 1$ и последователно разменяме първия елемент `m[1]` с `m[m[1]]` докато на позиция 1 не дойде елементът с ключ 1 (Ще изписваме само ключовете, като ще подчертаваме елементите, които разменяме.):

437**5**612
5374**6**12
63745**1**2
1374562

Сега числото 1 е на мястото си. Пристъпваме към 2:

1**37**4562
1**7**3456**2**
1234567

Числото 2 също вече си е на мястото. Остава да преминем през останалите елементи и да се уверим, че също са си по местата. Описаният процес силно напомня алгоритъма за намиране циклите на пермутация [*Наков-1998*]. В действителност алгоритъмът обхожда един по един *циклите на пермутацията*, като при всяка размяна един или два елемента отиват на окончателните си позиции. Оттук е ясно, че броят на размените не надвишава n. Броят на сравненията от своя страна не надвишава $2n$. Действително, всеки елемент на пермутацията се преглежда най-много два пъти: веднъж при преминаване по съответния му цикъл и втори път при преминаване през крайната му позиция. В случай, че даден елемент е на мястото си в изходната пермутация, той ще участва само в едно сравнение.

```
void permSort(struct CElem m[], unsigned n)
{ unsigned i;
  for (i = 0; i < n; i++)
    while (m[i].key != (int)i)
      swap(&m[i],&m[m[i].key]);
}
```

🖫 permsort.c

Задача за упражнение:

Да се докаже, че за всяко i ($1 \leq i \leq n$) и при всяка пермутация на елементите на масива при последователно разменяне на елемента `m[i]` с `m[m[i]]` в един момент `m[i]` ще съдържа i.

3.3. Паралелно сортиране

Характерна особеност на по-широкоизвестните алгоритми за сортиране е, че не позволяват извършването на повече от една операция (сравнение) едновременно. На пръв поглед подобно ограничение изглежда напълно естествено отражение на архитектурните особености на разпространените днес компютри. Действително, с малки изключения, всички те са серийни машини, при които изчислителните процеси протичат строго последователно. Напоследък обаче нещата започнаха да се променят. Усъвършенстването на технологиите за производство, понижаването на разходите и свързаното с него понижение на цените на процесорите направи възможно все по-увереното настъпление на *многопроцесорните системи* и *клъстери*. Доскоро само екзотика, днес компютърни системи с два и повече процесора започнаха да се възприемат като нещо напълно нормално. Масовото им навлизане, съчетано с евентуално нарастване броя на процесорите, неминуемо ще доведе със себе си и повишен интерес към алгоритмите за паралелни изчисления. Същевременно, усъвършенстването на мрежовите технологии и относителното поевтиняване на компютърните системи и сървъри ще води до все по-силно утвърждаване на така наречените *разпределени изчисления*, където работата по решаване на даден проблем се поделя между няколко свързани в мрежа компютърни системи. Нещо повече: днес

паралелизацията е успяла да се настани на още по-ниско ниво — вътре в самия процесор. Действително, налице е вътрешно хеширане и изпълняване на повече от една операции едновременно, като изпълнението на всяка следваща операция започва веднага, след като се освободят съответните ресурси в процесора, без да се чака окончателното завършване на предходната. Изобщо, днес *паралелизацията* е в настъпление по всички фронтове ;))

Въпреки, че масовото ѝ навлизане е сравнително ново явление, би било грешно да се счита, че тя е плод на съвременните технологии. Действително, паралелизацията присъства в компютрите от самата им поява. Така например, фундаменталното понятие *байт* е свързано с обработката на осем двоични цифри (разряда, бита) *едновременно*. Паралелизация може да се открие при предаване на данни по шината или мрежата, при работата на многопотребителски системи (мейнфрейми), при извършването на сложни изчисления върху многопроцесорни системи и много други.

Горните примери биха могли да се продължат още. За нас обаче е важен не самият процес на паралелизация, а отражението му върху разработването и използването на алгоритми. За пълното разгръщане възможностите на паралелния *хардуер* са нужни *софтуерни* средства за управление процеса на декомпозиция (разбиване) на задачата и разпределение на отделните подзадачи между паралелните хардуерни устройства (най-често процесори) за тяхната обработка. Т. е. възниква нужда от *паралелни алгоритми*.

Паралелните методи за обработка на данни не са нещо ново и исторически възникват много преди очертаващия се днес паралелен бум. Всъщност, дори много преди появата на първото изчислително устройство. От най-дълбока древност хората са забелязвали, че много от извършваните дейности могат значително да се ускорят, ако се паралелизират. Така например, в строителната (например при строежа на пирамидите) и селскостопанската дейност паралелизацията води до значително ускорение и се прилага широко. Друг пример за успешно паралелизиране е устройството на човешкия мозък, който е изключително паралелен: 10 милиарда неврона са свързани в мрежа, като всеки има връзка с около 10000 други. И, макар предаването на сигнал между два свързани неврона да е милиони пъти по-бавно отколкото при съвременните компютри, съществуват редица задачи, с които човешкият мозък се справя значително по-бързо от най-мощния съвременен компютър.

Разбира се, не всеки процес се поддава на паралелизация и често се оказва, че някой процес задължително трябва да предхожда друг. Така например, преди да сме построили покрива на една сграда, следва да сме построили всичко останало, започвайки от основите.

Съществуват редица алгоритми, които лесно се паралелизират, докато други се поддават трудно или пък изобщо не се поддават. Нещо повече: универсален алгоритъм за паралелизиране не съществува. Въпреки че съществуват някои общовалидни схеми, които улесняват процеса, по принцип във всеки конкретен случай следва да се подхожда по различен начин. Проблемът за паралелизирането е изключително сложен и разглеждането му излиза извън рамките на настоящето изложение.

По-долу ще насочим усилията си към паралелизиране на алгоритмите за сортиране и по-специално към разработването на паралелен вариант на *сортирането чрез сливане* (класическият непаралелен вариант ще бъде разгледан в *7.4.*). При разработването на алгоритъма ще се ограничим до абстрактна машина, изградена от множество идентични елементи с два входа и два номерирани изхода, които ще наричаме *компаратори*. Когато на входа на компаратор постъпят две числа x и y, на първия изход излиза $\min(x,y)$, а на втория — $\max(x,y)$. Ще считаме, че разполагаме с достатъчен брой компаратори, както и че произволен брой от тях могат да работят едновременно, т. е. паралелно. Тъй като разполагаме с единствена операция, извършвана от компаратора, множеството от алгоритмите за сортиране, които можем да построим, ще бъде ограничено. Така например, няма да можем да разглеждаме алгоритми, основаващи се на броене или използващи особености на вътрешното двоично представяне на числата и други.

Ще се опитваме да съставяме схеми от компаратори и свързващи ги линии (кабели). Ще искаме да съставим схема с n входа и n изхода, която при всеки вход $<a_1, a_2, \ldots, a_n>$ дава като резултат n-орката $<b_1, b_2, \ldots, b_n>$, която е пермутация на елементите, постъпили на входа, т. е. може да се получи от изходната n-орка само с размени на двойки елементи. Ще искаме $a_1, a_2, \ldots,$

a_n да бъдат елементи на някакво множество A, в което е въведена линейна наредба, т. е. всеки два елемента x и y от A са сравними и при това е в сила точно едно от трите съотношения: $x < y$, $x = y$ или $x > y$ (добре известното свойство *трихотомичност*). Интуитивно е ясно, че елементите на A могат да се подредят в нарастващ ред в една редица. По-долу за простота ще считаме, че елементите на A са числа. Ще припомним, че множества с линейна наредба образуват естествените числа, целите числа, реалните числа, символните низове и др. При това, не е трудно да се съобрази, че всяко подмножество на множество с линейна наредба също е линейно наредено. (*Защо?*)

Схемите ще чертаем като n паралелни хоризонтални линии, на места свързани по двойки с вертикални линии — компараторите. Ще считаме, че след преминаването на двойка линии през компаратор, горната ще съдържа по-малкото от числата, предавани по двете линии, а долната — по-голямото. Ще считаме, че времето за работа на всеки компаратор е константно, например 1. При това всеки компаратор ще генерира изход, едва след като и по двете линии на входа му са постъпили числа. Така само някои групи от компаратори в схемата наистина ще работят паралелно, а останалите или ще са приключили, или ще чакат числа и на двата си входа.

Нека се опитаме да дефинираме *време за работа* на схема от компаратори. Предполагайки, че всеки компаратор извършва работата си за време 1, а предаването на данни по линиите отнема пренебрежимо малко време, което можем да считаме за 0, можем да дефинираме времето за работа на схемата като времето, за което всяка от изходните линии получава своята стойност. Ясно е, че времето за работа на схемата съвпада с максималния брой компаратори, през които минава елемент от входната пермутация, преди да достигне изхода.

По аналогичен начин бихме могли да приведем рекурсивна дефиниция на *дълбочина на линия в дадена точка*. Всички линии на входа имат дълбочина 0. Ако на входа на компаратор постъпят линии с дълбочина $d(x)$ и $d(y)$ съответно, то линиите на изхода му имат дълбочина $\max(d(x), d(y)) + 1$. *Дълбочина на компаратор* дефинираме като дълбочината на излизащите от него линии, а *дълбочина на схема* — като максималната дълбочина на компаратор от схемата. Една схема ще наричаме *сортираща*, ако за *всяка* входна последователност $<a_1, a_2, \ldots, a_n>$ елементите на изхода монотонно нарастват: $b_1 \le b_2 \le \ldots \le b_n$.

Преди да продължим, ще формулираме един особено полезен принцип, улесняващ верификацията на сортиращи алгоритми и схеми.

3.3.1. Принцип на нулите и единиците

Теорема. (*Принцип на нулите и единиците*) Ако даден алгоритъм (схема) сортира вярно всяка входна последователност от 0 и 1, той ще сортира вярно и всяка входна последователност с елементи, принадлежащи на множество с линейна наредба.

Горната теорема е известна още като "Принцип на нулите и единиците.". Ползата от него очевидно е голяма, тъй като с негова помощ силно се улеснява формалното доказателство (а и емпиричното тестване) на коректността на даден алгоритъм за сортиране. Достатъчно е да покажем, че той работи правилно върху всяка от 2^n-те на брой входни последователности от 0 и 1 с дължина n, за всяко естествено n, за да докажем, че той работи правилно. Изобщо, принципът би могъл да бъде полезен и при тестването на програма, реализираща даден алгоритъм за сортиране: Тъй като числото 2^n расте относително бавно, разработчикът спокойно би могъл да тества коректността на програмата си за всички входни последователности с не повече от например 30 елемента. Преди да пристъпим към доказателството на принципа, ще формулираме и докажем две леми.

Лема 1. Нека имаме монотонно растяща функция f и схема с единствен компаратор, на чийто вход постъпват $f(x)$ и $f(y)$. Тогава на горния и долния му изход се получават $f(\min(x,y))$ и $f(\max(x,y))$ съответно.

***Доказателство*:** Действително, от монотонността на f следва, че $\min(f(x),f(y)) = f(\min(x,y))$ и $\max(f(x),f(y)) = f(\max(x,y))$, откъдето директно следва твърдението на лемата.

Лема 2. Нека е дадена монотонна функция f. Нека е дадена още схема от компаратори, която при вход $a = <a_1, a_2, \ldots, a_n>$ дава като резултат $b = <b_1, b_2, \ldots, b_n>$. Тогава ако на входа на схемата се подаде $f(a) = <f(a_1), f(a_2), \ldots, f(a_n)>$ на изхода ще се получи $f(b) = <f(b_1), f(b_2), \ldots, f(b_n)>$.

***Доказателство*:** Нека за определеност считаме, че функцията f е монотонно растяща, т. е. за всеки два елемента $x < y$ имаме $f(x) \leq f(y)$. Сега, използвайки индукция по дълбочината на всяка от линиите, можем да докажем дори по-строго твърдение от това на лемата, откъдето тя ще следва директно, а именно: Ако дадена линия получава стойност y при вход x, то тя ще получи стойност $f(y)$ при вход $f(x)$. Ще докажем това твърдение с индукция по дълбочината d на линията:

1) *База*: $d = 0$. Нека при x на входа имаме y на изхода на линията. Тогава, очевидно при $f(x)$ на входа ще имаме $f(y)$ на изхода.
2) *Индукционно предположение:* Нека предположим, че за всяка линия с дълбочина, строго по-малка от d ($d > 0$), ако при вход x имаме изход y, то при вход $f(x)$ ще имаме изход $f(y)$.
3) *Индукционна стъпка:* Нека на входа на компаратор с дълбочина d постъпват линии със стойности x_1 и x_2 съответно. Съгласно определението за компаратор на двата му изхода ще получим $y_1 = \min(x_1,x_2)$ и $y_2 = \max(x_1,x_2)$ съответно. От друга страна, съгласно индукционното предположение за двете входящи линии, при вход $f(x_1)$, $f(x_2)$ преди влизане в компаратора те ще получат стойности $f(y_1)$ и $f(y_2)$ съответно. Но f е монотонно растяща функция. Тогава, съгласно Лема 1 на изхода на компаратора ще получим $\min(f(y_1), f(y_2))$ и $\max(f(y_1), f(y_2))$ съответно.

Случаят на монотонно *намаляваща* функция f се разглежда аналогично. Лема 2 е доказана. Сега вече сме готови да пристъпим към доказателството на принципа на нулите и единиците.

Доказателство (Принцип на нулите и единиците): По-горе при доказателството на Лема 2 използвахме метода на *пълната математическа индукция*. Сега ще използваме друг универсален метод за доказване на твърдения: *допускане на противното*.

Нека е дадена схема от компаратори, сортираща правилно всички последователности от 0 и 1 и да допуснем, че съществува последователност $<a_1, a_2, \ldots, a_n>$ от елементи на някакво линейно наредено множество A, за което тя дава несортирана последователност. Тогава ще съществува поне една двойка елементи a_i и a_j ($a_i < a_j$) такава, че на изхода на схемата a_j е някъде преди a_i. Нека сега си дефинираме функция f по следния начин:

$$f(x) = \begin{cases} 0, x \leq a_i \\ 1, x > a_i \end{cases}$$

Не е трудно да се забележи, че така дефинираната функция е монотонно растяща. Тогава от това че a_j е преди a_i по Лема 2 получаваме, че $f(a_j)$ е преди $f(a_i)$. От друга страна, съгласно дефиницията на f имаме $f(a_j) = 1$ и $f(a_i) = 0$. Но тогава разглежданата схема не сортира правилно последователността $<f(a_1), f(a_2), \ldots, f(a_n)>$, която е последователност от 0 и 1. Получаваме противоречие с допускането, че схемата сортира правилно *всички* последователности от 0 и 1.

3.3.2. Битонични последователности

Доказаният току-що принцип ще ни бъде от особена полза при по-нататъшното изложение. По-долу ще се опитаме да разработим ефективен паралелен алгоритъм за сортиране, основаващ се на схеми от компаратори. За целта, първо ще разгледаме алгоритъм за сортиране на *битонични последователности*. Под битонична последователност ще разбираме последователност, която монотонно расте, след което монотонно намалява или обратно: монотонно намалява, след което монотонно расте. Примери: <1, 7, 9, 11, 27, 5, 4, 3, 3, 1> и <37, 20, 17, 16, 10, 10, 10, 18, 25, 47>. В частност, всяка сортирана, в нарастващ или намаляващ редпоследователност също е битонична. Битоничните последователности, съставени от 0 и 1, имат вида $0^i1^j0^k$ или $1^i0^j1^k$, i, j, k

≥ 0. Възползвайки се от принципа на нулите и единиците, по-долу ще се ограничим до разработването на схеми за сортиране на битонични последователности само от 0 и 1.

3.3.3. "Изчисти наполовина"

С цел опростяване на по-нататъшните разсъждения по-долу ще работим с последователности с четен брой елементи. При тези предположения ще разработим схема от компаратори, която действа в известен смисъл подобно на бързото сортиране (*виж 3.1.6.*), и която ще наречем *"изчисти наполовина"*. При постъпване на битонична последователност с дължина n на входа на изхода ще се получават две битонични последователности, всяка с дължина $n/2$. При това по-големите стойности ще бъдат във втората последователност, т. е. всеки елемент от първата последователност ще бъде по-малък или равен на всеки елемент на втората последователност. Оттук е ясно, че поне една от двете последователности е съставена само от 0 (горната) или само от 1 (долната), откъдето идва и името "изчисти наполовина". Разглежданата схема има дълбочина 1 и извършва паралелно всички сравнения от вида $(i,n/2+i)$ за $i = 1, 2, \ldots, n/2$.

3.3.4. Сортиране на битонична последователност

Нека предположим, че $n/2$ също е четно. Тогава бихме могли да приложим "изчисти наполовина" за всяка от двете новополучени последователности. Ако $n/4$ отново се окаже четно, то можем да продължим процеса върху всяка от четирите нови последователности и т.н. Ако n е степен на 2, описаният процес би могъл да се продължи рекурсивно до достигане на последователности с дължина 1. Не е трудно да се забележи, че така може да се сортира произволна битонична последователност. (*Защо?*)

Нека опитаме да оценим сложността на описания процес. Тъй като всяка стъпка изисква време 1 (времето за работа на "изчисти наполовина"), компараторите действат паралелно (съгласно дефиницията на схема от компаратори), а дължините на разглежданите редици всеки път намаляват наполовина, то общото необходимо време е $\log_2 n$. Получихме алгоритъм за сортиране на *битонични последователности от 0 и 1* със сложност $\Theta(\log_2 n)$. Сега от принципа на нулите и единиците следва, че предложената схема работи за произволна битонична последователност, т. е. не само от 0 и 1.

3.3.5. Сливаща схема

Вече разполагаме с ефективен паралелен алгоритъм за сортиране на *битонични* последователности. Ще се опитаме да го обобщим така, че да получим ефективен алгоритъм за сортиране на *произволни* последователности, без да изискваме непременно да бъдат битонични. За целта, първо ще разработим алгоритъм за сливане на две сортирани последователности в обща сортирана последователност по начина, описан от Бетчер.

Ако обърнем втората последователност, ще получим битонична последователност, която вече знаем как да сортираме. Ясно е, че обръщането може да стане за време 1 при използване на достатъчен брой компаратори. След това можем да приложим разработената по-горе схема за сортиране на битонична последователност. Общото необходимо време за сливане на двете последователности ще бъде $1 + \log_2 n$. Ако разгледаме по-внимателно първите две стъпки на получения алгоритъм, можем да забележим, че те биха могли сравнително лесно да се обединят в една, т. е. да се извърши едновременно обръщане на втората последователност и "изчисти наполовина". За целта, вместо да сравняваме двойките $(i,n/2+i)$, следва да сравним $(i,n-i+1)$, за $i = 1, 2, \ldots, n/2$. В резултат получаваме две битонични последователности, като при това всеки елемент на първата е по-малък или равен на всеки елемент от втората. Тогава можем да продължим да прилагаме битоничното сортиране направо от втората стъпка. В резултат получаваме алгоритъм, за сливане на два сортирани последователности за време $\log_2 n$.

3.3.6. Сортираща схема

След като реализирахме ефективна схема от компаратори за сливане на сортирани последователности, вече сме готови да пристъпим към същинската си цел: разработване на ефективен алгоритъм за сортиране на *произволна* входна последователност. Ще насочим усилията си към реализиране на паралелна версия на *сортирането чрез сливане* (*виж 7.4.*) на базата на горната сливаща схема.

Нека с цел улесняване на по-нататъшните разсъждения предположим, че n е степен на 2. Ще разсъждаваме индуктивно. Нека предположим, че имаме две сортирани последователности, всяка с дължина $n/2$. Тогава бихме могли да ги слеем в обща сортирана последователност за време $\log_2 n$, използвайки сливащата схема. Как обаче да получим двете сортирани последователности? Нека предположим, че разполагаме с 4 сортирани последователности l_1, l_2, l_3, l_4 с дължина $n/4$ всяка. Използвайки сливащата схема, бихме могли да слеем l_1 и l_2 в обща сортирана последователност с дължина $n/2$ за време $\log_2 (n/2)$. Бихме могли същевременно да слеем l_3 и l_4, използвайки второ копие на сливащата схема, без допълнителен разход на време. В резултат ще получим исканите две сортирани последователности с дължина $n/2$. А как да получим последователностите l_1, l_2, l_3, l_4? Нещата стоят аналогично. Отново предполагаме, че имаме осем сортирани последователности с дължина $n/8$, които комплектоваме и сливаме по двойки за време $\log_2 (n/8)$ и т.н. Продължаваме рекурсивно да намаляваме размера на задачата до получаване на едноелементни последователности, всяка от които можем да считаме за сортирана. Сега на обратния ход на рекурсията получаваме сортирана последователност.

Нека се опитаме да оценим дълбочината на така конструираната сортираща схема. Дълбочината $D(n)$ на схема, сортираща n-елементна последователност, може да се пресметне като дълбочината $D(n/2)$ на схема, сортираща $(n/2)$-елементна последователност (всъщност, имаме две такива схеми, но те действат паралелно), плюс дълбочината $\log_2 n$ на сливащата схема. Получаваме следната рекурентна зависимост:

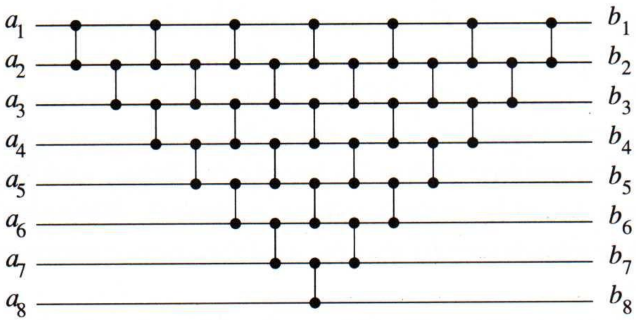
$$D(n) = \begin{cases} 0 & n = 1 \\ D\left(\dfrac{n}{2}\right) + \log_2 n & n = 2^k, k \geq 1 \end{cases}$$

Оттук по Основната теорема (*виж 1.4.10.*) получаваме:

$$D(n) \in \Theta\left(\log_2^2 n\right)$$

3.3.7. Транспозиционна сортираща схема

Предложената по-горе схема, разбира се, съвсем не е единствената възможна сортираща схема. Практически всеки универсален алгоритъм за сортиране, извършващ само размени на елементи, поражда съответна сортираща схема от компаратори. *Фигура 3.3.7.* показва схемата, съответстваща на сортирането *чрез вмъкване* (*виж 3.1.2.*).



Фигура 3.3.7. Сортираща схема за сортирането чрез вмъкване.

Тъй като при сортирането чрез вмъкване се извършва сравнение само между съседни елементи, компараторите от горната схема свързват само съседни линии. Подобна е ситуацията и при метода на мехурчето. Схеми, при които компараторите свързват само двойки съседни линии, се наричат *транспозиционни схеми*. Разбира се, не всички транспозиционни схеми са сортиращи.

3.3.8. Четно-нечетна сливаща схема на Бетчер

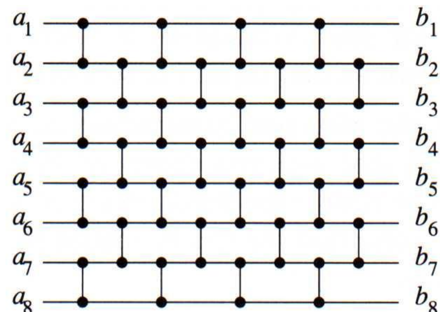
Четно-нечетната сливаща схема е разработена от Бетчер през 1960 г. Идеята е следната: Нека са дадени две сортирани n-елементни последователности $<a_1, a_2, \ldots, a_n>$ и $<a_{n+1}, a_{n+2}, \ldots, a_{2n}>$, които искаме да слеем в обща сортирана последователност (n е степен на 2). Ще разсъждаваме индуктивно. Да слеем поотделно нечетните ($<a_1, a_3, \ldots, a_{n-1}>$ и $<a_{n+1}, a_{n+3}, \ldots, a_{2n-1}>$) и четните ($<a_2, a_4, \ldots, a_n>$ и $<a_{n+2}, a_{n+4}, \ldots, a_{2n}>$) елементи на двете последователности. Сега можем да обединим четните и нечетните последователности, с помощта на $n/2$ компаратора, поставени между a_{2i-1} и a_{2i} за $i = 1, 2, \ldots, n$. В резултат получаваме схема с дълбочина $\Theta(\log_2 n)$.

3.3.9. Четно-нечетна сортираща схема

Доразвивайки идеята на четно-нечетната сливаща схема, Бетчер успява да получи сортираща схема с дълбочина $\Theta\left(\log_2^2 n\right)$. Тук компараторите са на n нива и разположението им се пресмята по проста формула. Нека обозначим с i номера на линията ($i = 2, 3, \ldots, n-1$), а с d — дълбочината на компаратора ($d = 1,2, \ldots,n$). Тогава линия i се свързва с линия $j = i+(-1)^{i+d}$, $1 \leq j \leq n$ на дълбочина d. Т. е. схемата би могла да се построи по следния алгоритъм: (*виж фигура 3.3.9.*)

За $d = 1, 2, \ldots, n$ повтаряме:

1) Ако d е четно, свързваме с компаратор линии $2i-1$ и $2i$, за $i = 1, 2, \ldots, [n/2]$.
2) Ако d е нечетно, свързваме с компаратор линии $2i$ и $2i+1$, за $i = 1, 2, \ldots, [(n-1)/2]$.



Фигура 3.3.9. Четно-нечетна сортираща схема.

3.3.10. Пермутационна схема

Да разгледаме схема, в която вместо компаратори се използват *превключватели* с две състояния, които могат да се включват и изключват. Подобно на компараторите превключвателите също имат два входа и два изхода. При изключено състояние на изхода на превключвателя се подават стойностите, получени на входа, без изменение, а при включено — се разменят (кръстосват). Нека е дадена схема с n линии и подадем на входа ѝ $<1, 2, \ldots, n>$. Ясно е, че, настройвайки превключвателите, можем да получаваме различни пермутации на първите n естествени числа. В случай, че можем да получим всичките $n!$ пермутации, схемата ще наричаме *пермутационна*. Може да се покаже, че при замяна на всички компаратори с превключватели в произволна сортираща схема се получава пермутационна схема.

Повечето източници сочат като първи изследователи на сортиращите схеми Армстронг, Нелсън и О'Конър, публикували своите изследвания през 1954 г. По-късно през 1960 г. Бетчер разработва своята четно-нечетна сливаща, а заедно с това и сортираща схема. На него дължим и сортирането, основаващо се на битонични последователности. За достатъчно дълго време това са най-добрите известни алгоритми — с алгоритмична сложност $\Theta\left(\log_2^2 n\right)$.

През 1983 г. Айтай, Комлос и Земереди успяват значително да подобрят този резултат, разработвайки сортираща схема с дълбочина $\Theta\left(\log_2 n\right)$, използваща $\Theta\left(n \log_2 n\right)$ компаратора. Въпреки логаритмичната си сложност, алгоритъмът се оказва трудно приложим на практика, заради огромната константа, която се крие зад Θ(...).

Задачи за упражнение:

1. Да се докаже, че алгоритъмът от 3.3.4. сортира правилно всяка битонична последователност (*виж 3.3.4.*).

2. Да се докаже, че всяка сортираща транспозиционна схема с n входа съдържа $\Omega(n^2)$ компаратора (*виж 3.3.7.*).

3. Да се определи дълбочината и да се намери броят компаратори в транспозиционна схема с n входа (*виж 3.3.7.*), съответстваща на:

   а) сортиране чрез вмъкване (*виж 3.1.2.*);

   б) сортиране по метода на мехурчето (*виж 3.1.4.*).

4. Като се използва индукция по броя на линиите, да се докаже, че една транспозиционна схема с n линии е сортираща тогава и само тогава, когато сортира последователността $(n,n-1,\ldots,1)$.

5. Да се докаже, че за $n > 2$ за всяка пермутационна схема с n входа ще съществува поне една пермутация, която се получава поне по два различни начина (*виж 3.3.10.*).

3.4. Въпроси и задачи

3.4.1. Задачи от текста

Задача 3.1.

Да се докаже, че дървото на сравненията гарантира минимален брой сравнения в най-лошия случай (*виж 3.1.1.*).

Задача 3.2.

Да се докаже, че ключът на елемента, който ще се вмъква, е добър ограничител при използване на метода на ограничителя за реализация на сортиране чрез вмъкване (*виж 3.1.2.*).

Задача 3.3.

Да се сравнят различните варианти на сортирането чрез вмъкване (*виж 3.1.2.*).

Задача 3.4.

Да се изведат формулите от *таблица 3.1.2.*

Задача 3.5.

Да се докаже, че ако една последователност е била δ-сортирана, то след прилагане на δ′-сортиране, δ′ < δ тя продължава да бъде δ-сортирана (*виж 3.1.3.*).

Задача 3.6.

Да се изведат формулите от *таблица 3.1.4.*

Задача 3.7.

Да се сравнят различните варианти на сортирането по метода на мехурчето (*виж 3.1.4.*):

а) мехурче без флаг
б) мехурче с флаг
в) сортиране чрез клатене (*виж 3.1.5.*).

Задача 3.8.

Да се докаже, че бързото сортиране на Хоор (*виж 3.1.6.*) сортира правилно всяка входна последователност.

Задача 3.9.

Да се докаже, че бързото сортиране на Хоор (*виж 3.1.6.*) изисква стек от порядъка на $\Theta(\log_2 n)$.

Задача 3.10.

Да се тестват предложените програмни реализации на бързото сортиране на Хоор (*виж 3.1.6.*) за всички възможни входни последователности с до 32 елемента:

а) за последователност от случайни числа;
б) за всички пермутации на числата от 1 до n;
в) за всички пермутации на числата от дадено мултимножество (с повтарящи се на елементи);
г) като се ползва принципът на нулите и единиците.

Задача 3.11.

Да се определи теоретично най-добрият начин за избор на разделящ елемент при бързото сортиране на Хоор (*виж 3.1.6.*):

а) първи;
б) последен;
в) среден;
г) средно аритметично на два елемента;
д) средно аритметично на три елемента;
е) медиана: среден *по големина* елемент за дяла (*виж 7.1.*);
ж) изкуствен елемент, например както при побитовото сортиране (*виж 3.2.3.*).

Задача 3.12.

Да се направят емпирични тестове за всяка от подточките от горната задача и да се сравнят получените резултати с теоретичните.

Задача 3.13.

Да се реализира итеративен вариант на бързото сортиране на Хоор (*виж 3.1.6.*).

Задача 3.14.

При какъв брой елементи в сортирания дял при бързото сортиране (*виж 3.1.6.*) следва да се използва по-прост алгоритъм като сортиране чрез вмъкване?

Задача 3.15.

Да се докаже, че колкото по-близо до медианата е елементът, относно който става разделянето на два дяла при бързото сортиране на Хоор (*виж 3.1.6.*), толкова по-ефективен е алгоритъмът.

Задача 3.16.

Да се предложи модификация на бързото сортиране на Хоор (*виж 3.1.6.*), при която ще се гарантира времева сложност $\Theta(n.\log_2 n)$ в *най-лошия случай*.

Задача 3.17.

Да се промени типът на формалните параметрите на функцията `quickSort()` от `int` на `unsigned` в *3.1.6.*. Какви проблеми възникват и защо? Какви решения предлагате?

Задача 3.18.

За всяко естествено n да се намери входна последователност, при която бързото сортиране (*виж 3.1.6.*) има сложност:

а) $\Theta(n.\log_2 n)$
б) $\Theta(n^2)$

Задача 3.19.

Да се докаже, че методът на зайците и костенурките (*виж 3.1.7.*) има сложност $\Theta(n.\log_2 n)$ както в средния, така и в най-лошия случай.

Задача 3.20.

Да се сравнят теоретично и емпирично алгоритъмът на Шел (*виж 3.1.3.*) и методът на зайците и костенурките (*виж 3.1.7.*).

Задача 3.21.

Да се докаже, че при сортиране с пряк избор всеки елемент от несортираната част е по-голям или равен от всеки елемент път сортираната (*виж 3.1.8.*).

Задача 3.22.

Да се сравнят теоретични и емпирично двата варианта на сортиране с пряк избор (*виж 3.1.8.*).

Задача 3.23.

Да се изведат формулите от *таблица 3.1.8.*

Задача 3.24.

Да се докаже, че при турнир по тенис с пряко елиминиране, ако броят на участниците не е степен на 2, поне в един кръг ще трябва да има почиващ играч (*виж 3.1.9.*).

Задача 3.25.

Да се докаже, че върхът на пирамидата съдържа най-големия ѝ елемент (*виж 3.1.9.*).

Задача 3.26.

Да се разработи метод за сортиране, използващ *тернарна пирамида*. Тернарната пирамида е обобщение на класическа пирамида и се основава на пълни тернарни (троични) дървета. Следва ли да се очаква ускорение в сравнение с класическата пирамида (*виж 3.1.9.*)?

Задача 3.27.

Да се разработи алгоритъм за бързо обединение на две пирамиди (*виж 3.1.9.*).

Задача 3.28.

Да се докаже, че всяко подмножество на множество с линейна наредба също е линейно наредено (*виж 3.2.*).

Задача 3.29.

Да се модифицира предложената реализация на сортиране чрез множество (*виж 3.2.1.*) така, че сортираната последователност да се получава в изходния масив.

Задача 3.30.

Да се реализира устойчив вариант на сортирането чрез броене (*виж 3.2.2.*).

Задача 3.31.

Да се докаже, че за всяко i ($1 \le i \le n$) и при всяка пермутация на елементите на масива при последователно разменяне на елемента `m[i]` с `m[m[i]]` в един момент `m[i]` ще съдържа i (*виж 3.2.5.*).

Задача 3.32.

Да се докаже, че алгоритъмът от *3.3.4.* сортира правилно всяка битонична последователност (*виж 3.3.4.*).

Задача 3.33.

Да се докаже, че побитовото сортиране (*виж 3.2.3.*) сортира правилно всяка входна последователност.

Задача 3.34.

Да се реализира статичен вариант на побитовото сортиране (*виж 3.2.3.*), при който се използва единствен допълнителен масив, в който двата списъка, изграждани при итеративната реализация, нарастват един срещу друг.

Задача 3.35.

Да се сравнят итеративният и рекурсивният вариант на побитовото сортиране (*виж 3.2.3.*).

Задача 3.36.

Да се сравнят побитовото сортиране (*виж 3.2.3.*) и методът на бройните системи (*виж 3.2.4.*).

Задача 3.37.

Да се докаже, че всяка сортираща транспозиционна схема с n входа съдържа $\Omega(n^2)$ компаратора (*виж 3.3.7.*).

Задача 3.38.

Да се определи дълбочината и да се намери броят компаратори в транспозиционна схема с n входа (*виж 3.3.7.*), съответстваща на:

а) сортиране чрез вмъкване (*виж 3.1.2.*);
б) сортиране по метода на мехурчето (*виж 3.1.4.*).

Задача 3.39.

Като се използва индукция по броя на линиите, да се докаже, че една транспозиционна схема с n линии е сортираща тогава и само тогава, когато сортира последователността $(n,n-1,\ldots,1)$.

Задача 3.40.

Да се докаже, че за $n > 2$ за всяка пермутационна схема с n входа ще съществува поне една пермутация, която се получава поне по два различни начина (*виж 3.3.10.*).

3.4.2. Други задачи

Задача 3.41. *Сравнение на елементарни методи за сортиране.*

Да се сравнят на практика елементарните методи за сортиране: вмъкване (*виж 3.1.2.*), мехурче (*виж 3.1.4.*) или пряк избор (*виж 3.1.8.*). Да се направят теоретични разсъждения и емпирични тестове за различни:

а) брой елементи: 10; 20; 50; 100; 1000; 10000;
б) наредба на масива: подреден; обратно подреден; слабо разбъркан; силно разбъркан.

Задача 3.42. *Достатъчен брой итерации.*

Да се докаже, че при вмъкване, пряк избор, мехурче и пирамида $n-1$ итерации на външния случай са достатъчни.

Задача 3.43. *Сравнение на бързите методи за сортиране.*

Да се сравнят теоретично и емпирично бързото сортиране (*виж 3.1.6.*), методът на зайците и костенурките (*виж 3.1.7.*) и пирамидалното сортиране (*виж 3.1.9.*).

Задача 3.44. *Комбиниране на бързо сортиране с елементарен метод.*

Да се определи за кой от елементарните методи за сортиране (вмъкване, мехурче или пряк избор: *виж 3.1.2., 3.1.4. и 3.1.8.*) е теоретично най-добре да се комбинира с бързото сортиране (*виж 3.1.6.*). Да се съставят съответни програмни реализации и да се направят съответни тестове. Потвърждават ли се предварителните теоретични предположения на практика?

Задача 3.45.*Най-лош и най-добър случай.*

За всеки от алгоритмите за сортиране чрез сравнение да се определи най-лошият и най-добрият му случай.

Задача 3.46. *Устойчивост.*

Кои от алгоритмите за сортиране, разгледани в настоящата глава, са устойчиви?

Задача 3.47. *Последователен достъп.*

Кои от разгледаните алгоритми са приложими за сортиране на файлове и линейни списъци без възможност за пряк достъп до елементите?

Задача 3.48. *Гарантиран минимален брой сравнения и размени.*

Кои от алгоритмите за сортиране, разгледани в настоящата глава, гарантират в най-лошия случай минимален брой:

а) сравнения;
б) размени.

Задача 3.49. *Минимален брой размени.*

Даден е масив от цели числа, несъдържащ равни елементи, и съотношения между някои от елементите му. Да се състави алгоритъм за сортиране на масива, извършващ минимален брой размени.

Задача 3.50. *Двойно сортиране.*

Дадена е матрица $A = (a_{ij})$. Всеки неин ред поотделно се сортира в нарастващ ред, след което поотделно се сортират и стълбовете ѝ. Ще останат ли редовете сортирани в нарастващ ред?

Глава 4
Търсене

“I have an existentialist map of the world.
It has "You are here" written all over it.”

~ Unix Fortune

Процесът на сортиране, разгледан в *глава 3*, е тясно свързан с процеса на търсене. Действително, частичното или цялостно сортиране на дадено множество от елементи може силно да ускори процеса на търсене. Разбира се, двете операции могат да се разглеждат и като напълно независими. Така например, имената на служителите в дадено предприятие могат да се сортират по азбучен ред, само за да бъде и разпечатан на хартия съответен списък. Същевременно, по-голямата част от алгоритмите за търсене не предполагат и не изискват предварително сортиране.

Търсенето е фундаментална дейност, извършвана всекидневно от всеки от нас по най-разнообразни поводи. Дали го съзнаваме или не, все едно ние прекарваме в търсене значителна част от времето си: търсим телефонен номер, търсим си изгубените ключове, търсим нещо за ядене в хладилника, търсим Замунда на картата на Африка, търсим си работа, търсим интересно предаване по телевизията, търсим новия брой на "Жълт труд", търсим си белята и др. Спокойно би могло да се каже, че търсенето е добре позната дейност за всеки от нас. Въпреки това, замислилите се върху използваните методи са единици. Същевременно, разнообразието от видове търсения е почти необятно, което силно затруднява класифицирането им. В процеса на търсене повечето от нас изхождат главно от натрупания опит, а в непозната ситуация се справят *по метода на пробите и грешките*. Съществуват достатъчно мощни методи за търсене, които могат съществено да се различават един от друг в зависимост от конкретните условия. Така например, търсенето би могло да бъде некоректно зададено (търсене на теле под вола), целта би могла да бъде неясно дефинирана (търсене на интересно предаване по телевизията), целта би могла да се движи (търсене на вражеска подводница), търсенето би могло да бъде ограничено по време (търсене на бомба с часовников механизъм), пробите биха могли да имат различна цена (търсене на игла в купа сено), множеството би могло да бъде безкрайно (търсене на петрол) и други. В частност търсенето би могло да притежава всички изброени по-горе свойства, както и евентуално някои други (търсене на смисъл в живота).

Ясно е, че обхващането на всички изброени по-горе случаи е почти невъзможно, поради което се налага някои от тях да бъдат фиксирани. По-долу ще работим с опростен и напълно определен модел, притежаващ следните свойства:

- целта е фиксирана и ясно дефинирана;
- времето е неограничено;
- множеството, в което търсим, е крайно и статично, т. е. не се изменя в процеса на търсене;
- всички проби (сравнения) имат еднаква цена;
- никога не грешим при пробата;
- цялата информация от предходните сравнения се запазва.

Търсенето рядко е изолирана операция. Горният модел е полезен, но, за съжаление, не винаги е налице. Често се случва множеството, в което се търси, да се изменя динамично: включват се и се изключват елементи, пренареждат се и др. Ето защо е полезно алгоритмите за търсене да се разглеждат не като нещо отделно и независимо, а като част от цялостен пакет фундаментални операции над елементите на дадено множество. По този начин, разглеждайки всички операции в съвкупност, ще можем по-правилно да оценим взаимното им влияние и да се предпазим от неправилни оценки на ефективността на търсенето. Така например, двоичното търсене, което ще

бъде разгледано по-долу, има сложност $\Theta(\log_2 n)$ както в средния, така и в най-лошия случай. Същевременно обаче, то изисква елементите на множеството да бъдат сортирани. В случай че това не е така, предварителното им сортиране преди всяко търсене ще изисква време от порядъка на $\Theta(n.\log_2 n)$. Така действителната му сложност ще се окаже $\Theta(n.\log_2 n)$. Бихме могли, разбира се, да поддържаме множеството винаги в сортиран вид. Това обаче ще доведе до значително усложняване на операциите по включване и изключване на елементи. Удобно е търсенето да се разглежда като елемент на следния комплект операции:

- инициализиране
- търсене
- вмъкване
- изтриване
- обединяване на множества
- сортиране

Между някои от горните операции съществува непосредствена връзка и те често се изпълняват едновременно. Така например, преди вмъкване на елемент често се извършва търсене на съответната позиция. Както при сортирането (и структурите от данни, разгледани в *глава 2*), ще считаме, че елементите на разглежданото множество представляват записи от тип `struct CElem`, едно от чиито полета е избрано за *ключ*. Понякога ключовите полета са уникални в рамките на елементите на множеството, друг път повтарянето им се оказва допустимо. По-долу ще видим, че възможността за повтаряне или неповтаряне на елементите на множеството се оказва важен елемент на реализацията на горния комплект операции. Да предположим, че за горните операции вече сме реализирали съответни алгоритми, изрично недопускащи дублиране на ключовите полета. Бихме ли могли достатъчно лесно да преработим съответния програмен код, така че да се допуска дублиране? Един възможен подход за решаване на проблема е добавяне на допълнителен указател в записа, представляващ елементите на множеството. Този указател ще сочи към списък на елементите с еднакви ключове. Основно предимство на този подход е, че едно-единствено търсене се оказва достатъчно, за да се намерят *всички* ключове със зададена стойност. Друга възможна реализация е да се допуска дублиране на елементите, като при търсене се връща *произволен елемент*, с посочения ключ. Основен проблем тук ще бъде намирането на *всички* елементи с този ключ, за което в случай на нужда следва да се предвиди специален механизъм. Бихме могли, разбира се, да въведем и *втори служебен ключ*, за който можем да гарантираме, че е уникален в рамките на цялото множество. Тогава бихме могли да реализираме търсене както по само по първия, така и по двата ключа едновременно. Може би най-гъвкавият вариант е реализирането на специална функция на сравнение, към която да се обръща процедурата за търсене. По този начин ще бъде възможно в процеса на търсене да се проверява дали разглежданият в момента елемент с търсения ключ отговаря на произволен набор допълнителни условия.

4.1. Последователно търсене

Най-елементарният и най-очевиден алгоритъм за търсене е последователното търсене. Нека предположим, че елементите на множеството се съдържат в едномерен масив. Търсенето се извършва посредством последователно преглеждане на елементите на масива до откриване на търсения елемент или до преглеждане на всички елементи. Последното означава, че елемент с такъв ключ не съществува. При постъпване на нов елемент ще го вмъкваме в края на масива `seqSearch()`. Следва една възможна реализация на основните функции, основаваща се на последователното търсене:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
```

```
#define DataType int
struct CElem {
  int key;
  DataType data;
  /* ... */
} m[MAX + 1]; /* Масив от записи */
unsigned n;   /* Брой елементи в масива */

void seqInit(void) { n = 0; }          /* Инициализиране */

unsigned seqSearch(int key)            /* Последователно търсене */
{ unsigned x;
  m[0].key = key; /* Ограничител */
  for (x = n + 1; key != m[--x].key; ) ;
  return x;
}

void seqInsert(int key, DataType data) /* Добавя нов елемент */
{ m[++n].key = key;
  m[n].data = data;
}

void seqPrint(void)                    /* Извежда списъка на екрана */
{ unsigned i;
  char buf[9];
  for (i = 1; i <= n; i++) {
    sprintf(buf, "%d|%d", m[i].key, m[i].data);
    printf("%8s", buf);
  }
}

void performSearchTest(void)
{ unsigned ind, elem2Search;

  for (elem2Search = 0; elem2Search < 2*MAX; elem2Search++) {
    printf("Търсим елемент с ключ %u.\n", elem2Search);
    if (0 == (ind = seqSearch(elem2Search)))
      printf("%s","Елемент с такъв ключ не съществува!\n");
    else
      printf("%Елементът е намерен! Инф. част: %d\n", m[ind].data);
  }
}

int main(void) {
  unsigned ind;
  seqInit();
  for (ind = 0; ind < MAX; ind++)
    seqInsert(rand() % (MAX*2), ind);
  printf("Списъкът съдържа следните елементи: \n");
  seqPrint();
  printf("\nТестване:\n");
  performSearchTest();
  return 0;
}
```

🖫 seq_arr.c

Също както при сортирането, по-горе сме приели, че същинската информационна част на елементите на масива съдържа единствено поле `data`. Разбира се, при нужда читателят би могъл

без особени усилия да модифицира структурата на тип `struct CElem` съобразно конкретните си нужди и изисквания, което ще изисква и съответните минимални промени в горния код.

Търсенето се извършва последователно по посока на намаляване на индексите. Забележете, че нулевият елемент на масива не съдържа елемент на множеството, а служебна информация. Той е използван като *елемент-ограничител*, позволяващ опростяване на цикъла: проверява се само едно условие, тъй като сме осигурили намирането на елемент със зададения ключ (в краен случай нулевият). В случай на дублиране на ключове горният алгоритъм очевидно връща последния постъпил елемент с този ключ.

Каква е сложността на алгоритъма в средния и в най-лошия случай съответно? Ясно е, че най-лошият случай е когато търсеният елемент липсва. Тогава се преглеждат всичките n+1 елемента на масива, включително нулевият. В общия случай, когато търсенето е успешно, средният брой на извършваните сравнения е (n+1)/2 (*Защо?*). Същевременно, за намиране на всички елементи с даден ключ е необходимо *цялостно* преглеждане на масива.

Задача за упражнение:

Да се докаже, че средният брой сравнения при последователно търсене е (n+1)/2.

4.1.1. Последователно търсене в сортиран списък

Последователното търсене, разгледано по-горе, е най-простият и най-неефективният алгоритъм за търсене. Бихме могли да се опитаме да подобрим ефективността му, поддържайки елементите в сортиран вид. Това ще ни се удаде сравнително лесно, ако използваме свързан списък вместо масив. Всичко, което трябва да направим, е да вмъкваме всеки новопостъпил елемент на съответното му място, така че списъкът да остава сортиран. Така, преди всяко вмъкване ще се изисква извършване на съответно търсене, за да се открие съответната позиция.

Операцията по вмъкване се усложнява значително. По-горе я извършвахме за константно време, без нито едно сравнение, а сега това ще изисква предварително търсене. Какво все пак постигнахме? Обстоятелството, че елементите са подредени в нарастващ ред, ще ни позволи да прекратяваме по-нататъшното търсене при достигане на елемент с ключ, по-голям от търсения. Броят на необходимите сравнения в случай на неуспешно търсене намалява до (n+1)/2, в сравнение с n+1 при несортирания случай. На практика успешното и неуспешното търсене се оказват равнопоставени, тъй като тази модификация очевидно не влияе на успешното търсене. В най-лошия случай отново трябва да се прегледат всички елементи, т. е. да се извършат n+1 сравнения. Оказва се, че в замяна на това, че сме приравнили сложността на вмъкването, която по-рано беше константна, към сложността на търсенето, получаваме само равнопоставеност на успешно и неуспешно търсене. Полученият резултат е изключително поучителен и показва, че преди да се пристъпи към някаква "очевидна" оптимизация, следва много внимателно да се прецени възможната полза и вреда, които тя ще принесе.

Следва примерна реализация на последователно търсене в сортиран списък, заедно със съответна версия на функции за вмъкване и инициализиране:

```
struct CElem {
  int key;
  DataType data;
  struct CElem *next;
  /* ... */
} *head;

void listInit(void)                    /* Инициализиране */
{ head = (struct CElem *) malloc(sizeof *head);
  head->next = NULL;
}

void listInsert(int key, DataType data) /* Добавя нов елемент */
```

```
{ struct CElem *p, *q, *r;
  q = (struct CElem *) malloc(sizeof *head),
  r = (p = head)->next;
  while (r != NULL && r->key < key) {
    p = r;
    r = r->next;
  }
  q->key = key;
  q->data = data;
  q->next = r;
  p->next = q;
}

struct CElem *listSearch(int key)  /* Последователно търсене */
{ struct CElem *q;
  for (q = head->next; q != NULL && q->key < key; q = q->next) ;
  if (NULL == q || key != q->key)
    return NULL;
  else
    return q;
  }

void listPrint(void)               /* Извежда списъка на екрана */
{ struct CElem *q;
  char buf[9];
  for (q = head->next; q != NULL; q = q->next) {
    sprintf(buf, "%d|%d", q->key, q->data);
    printf("%8s", buf);
  }
}
```

seq_list

Задача за упражнение:

Да се определи средният брой сравнения при последователно търсене в сортиран списък. Да се сравни с класическото последователно търсене.

4.1.2. Последователно търсене с преподреждане

В случай, че разполагаме с предварителна информация относно вероятността за достъп до всеки от елементите, бихме могли да ги подредим така, че най-често търсеният да бъде на върха на списъка, следващият — непосредствено след него и т. н. Използването на подобна стратегия се оказва изключително ефективно, особено при силно неравномерно разпределение, при което малък брой елементи се търсят с много голяма вероятност.

В случай, че не разполагаме с такава предварителна информация, бихме могли сами да си я получим с помощта на прости статистически наблюдения: достатъчно е към всеки елемент да прикрепим брояч на достъпа до него. При всяко ново търсене ще актуализираме брояча на търсения елемент. Ясно е, че след актуализацията той евентуално ще може да мине по-напред в списъка. Ето защо следва да се направи сравнение с предходния елемент в списъка, при което евентуално двата да се разменят. Тъй като в списъка може да има многократно дублиране на честотите на достъп, в случай на размяна се налага съответно сравнение и със следващия предходен елемент и т. н. до достигане на точната позиция, съответстваща на новата честота на елемента. Реорганизацията става за време, пропорционално на n, като в най-лошия случай може да се случи последният елемент на списъка да стане първи, разменяйки се последователно с всички преди него [*Уирт-1980*].

Предложената по-горе схема с поддържане на специални броячи обаче се оказва тромава, изисква допълнителна памет от порядъка на *n* и би могла да се оптимизира. За пореден път се оказва, че по-простото е по-добро. Действително, в този случай се оказва изключително ефективно прилагането на по-проста стратегия за реорганизация. Не се поддържат никакви броячи и не се води никаква статистика. Вместо това, при всяко успешно търсене на елемент той се поставя в началото на списъка. Разбира се, тази стратегия не ни гарантира оптимално подреждане на елементите съгласно честотата им на достъп, но пък е лесна за поддържане и достатъчно ефективна. Действително, за разлика от варианта с броячите, тук реорганизацията става за време константа. От друга страна, колкото по-често е осъществяван достъп до даден елемент, с толкова по-голяма вероятност той ще се намира сред първите елементи на списъка. При това така по-добре се отчитат локалните особености: ако някой глобално погледнато рядко търсен елемент в даден момент се търси често, тази стратегия ще ни позволи да се възползваме от това. (Сравнете с адаптивните методи за компресиране от *10.5.*)

```
void listInsert(int key, DataType data) /* Добавя нов елемент */
{ struct CElem *q = (struct CElem *) malloc(sizeof *head);
  q->key = key; q->data = data;
  q->next = head; head = q;
}

/* Последователно търсене с преподреждане */
struct CElem *listSearch(int key)
{ struct CElem *q, *p = head;
  if (NULL == head)
    return NULL;
  if (head->key == key) return head;
  for (q = head->next; q != NULL; )
    if (q->key != key) {
      p = q;
      q = q->next;
    }
    else {
      p->next = q->next;
      q->next = head;
      return (head = q);
    }
  return NULL;
}
```

🖫 reorder.c

Задачи за упражнение:

1. Да се реализира претърсване на списък с преподреждане, базирано на брояч на достъпа. Да се сравни ефективността му с реализацията без броячи по-горе.
2. Да се сравни основната идея на последователното търсене с преподреждане без броячи с тази на адаптивните методи за компресиране (*виж 10.5.*).
3. Да се определи средният брой сравнения при последователно търсене *с преподреждане*. Да се сравни с класическото последователно търсене.
4. Да се сравнят в най-добрия, средния и най-лошия случай трите разгледани метода за последователно търсене:

   а) в неподреден масив (*виж 4.1.*)
   б) в сортиран списък (*виж 4.1.1.*)
   в) в несортиран списък с преподреждане (*виж 4.1.2.*)

4.2. Търсене със стъпка. Квадратично търсене

Нека отново разгледаме случая на наредено множество. Разгледаният по-горе метод на търсене в сортиран списък се отличаваше незначително от стандартното последователно търсене и не използваше достатъчно пълноценно наредбата на елементите. Ще се опитаме да поправим тази грешка, като за целта ще възприемем коренно различен подход. Нека изберем някаква стъпка k и последователно извършваме проверката дали ключът на търсения елемент е по-голям от първия елемент, от (k+1)-ия елемент, от (2k+1)-ия елемент, от (3k+1)-ия елемент,... Т. е. сравняваме го с `m[1].key`, `m[k+1].key`, `m[2k+1].key`,... Процесът приключва при достигане до елемент, по-голям или равен на x, или на края на масива.

Да разгледаме по-внимателно горната схема (ще я наричаме *търсене със стъпка*). Нека предположим, че, прилагайки я, сме достигнали до елемент, по-голям от x. Сега можем да използваме последователно търсене в интервала, определен от последните две проби. В случай, че сме били излезли извън масива, можем да използваме последователно търсене от предишната проба до края на масива. Очевидно подобен подход би могъл да доведе до силно съкращаване на броя на елементите, които се преглеждат от последователното търсене. Освен това лесно се вижда, че изложеният метод е обобщение на линейното търсене. Действително, последното се получава при $k = 1$.

Съгласно приведеното по-горе описание, предложеният метод винаги започва с `m[1].key`. Доколко подобно начало е удачно? Не е трудно да се види, че това всъщност е лош избор, при това по същите причини, по които е лош и изборът `m[n].key`: носи ни минимално количество информация. Оказва се, че е много по-удачно да се започне направо с `m[k].key`. (*Защо?*) Лесно се вижда че и в този случай при $k = 1$ се получава линейно търсене.

Каква е ефективността на описания метод при фиксирано k и кой е най-лошият случай? Ясно е, че линейното търсене има еднаква цена за всички интервали от вида [`m[i*k+1].key`; `m[(i+1)*k].key`], тъй като се извършва върху еднакъв брой елементи. Изключение може да прави единствено последният интервал, който евентуално може да съдържа по-малко от k елемента. В най-лошия случай търсеният ключ е в последния интервал, което означава, че ще ни бъдат необходими $[n/k]$ сравнения, за да определим нужния ни интервал, в който да приложим линейното търсене. Към тях следва да прибавим дължината на интервала, която при n, кратно на k, е $k–1$. Получаваме, че в най-лошия случай при търсене със стъпка k се извършват не повече от $[n/k] + k – 1$ сравнения. Следва примерна реализация на описания алгоритъм:

```
unsigned seqSearch(unsigned l, unsigned r, int key)
{ while (l <= r)
    if (m[l++].key == key)
      return l-1;
  return NOT_FOUND;
}

unsigned jmpSearch(int key, unsigned step)
{ unsigned ind;
  for (ind = 0; ind < n && m[ind].key < key; ind += step) ;
  return seqSearch(ind + 1 < step ? 0 : ind + 1 - step,
                                       n < ind ? n : ind, key);
}
```

🖫 jumpsear.c

Възниква важният въпрос: Как, при зададено n да избираме k така, че да си осигурим максимална ефективност? *Таблица 4.2.* показва максималния брой сравнения, извършвани от вмъкването със стъпка при различни стойности на n и k:

$n\backslash k$	1	2	3	4	5	6	7	8
1	1							
2	2	2						
3	3	2	3					
4	4	3	3	4				
5	5	3	3	4	5			
6	6	4	4	4	5	6		
7	7	4	4	4	5	6	7	
8	8	5	4	5	5	6	7	8

Таблица 4.2. Максимален брой сравнения при различни стойности на n и k.

Вижда се, че най-добрите стойности на k са близки до $n/2$, т. е. разположени са в средата на съответния ред на таблицата или непосредствено вляво от нея. За да определим по-прецизно кои стойности на k са най-добри и как зависят от n, следва да определим при каква зависимост между n и k функцията $f(k)$ приема минимална стойност:

$$f(k) = [n/k] + k - 1.$$

Не е трудно да се покаже (например с изследване на знака на втората производна), че това е стойността $k = \sqrt{n}$. Тогава $f(k) \leq 2\sqrt{n} + 1$. В този случай търсенето се нарича *квадратично*.

И така, постигнахме едно наистина добро подобрение: от n до $\sqrt{n}$. Бихме ли могли да постигнем повече? Както следва да се очаква, отговорът на този въпрос е положителен. Действително, по-горе минимизирахме $f(k)$, предполагайки, че след определяне на интервала ще извършим последователно търсене. И точно тук идва идеята: А какво ще стане, ако в този момент приложим още веднъж търсене с някаква нова стъпка l $(1 < l < k)$ и едва след това последователно търсене? Действително, дължината на интервала след първото търсене със стъпка k е $\sqrt{n}$, което може да бъде достатъчно голямо число, така че подобно подобрение би било разумно. Сега на първата стъпка ще имаме не повече от k сравнения за определяне на първия интервал, след това не повече от l — за определяне на втория, и накрая не повече от $n/(k.l)$ — за последователното търсене. Можем да образуваме нова функция, която да минимизираме, като в този случай лесно се получава, че минимумът се достига за $k = l = n/(k.l)$, откъдето $n = k^3$. В този случай алгоритъмът ще извършва не повече от $3\sqrt[3]{n}$ сравнения. За $n \geq 12$ се получава, че $3\sqrt[3]{n} < 2\sqrt{n}$, т. е. постигнахме подобрение [*Gregory*,*Rawlins-1997*].

А какво би станало, ако приложим алгоритъма и трети, четвърти и т. н. път? Разсъждения, аналогични на горните, ни водят до граничната *логаритмична сложност*. Въпреки значението на описаното обобщение (постигнахме логаритмична сложност!) ние няма да се спираме повече на него, тъй като съществува по-прост начин за постигане на такава сложност.

Задачи за упражнение:

1. Да се реализира двустепенно търсене със стъпки k и l. Да се намерят експериментално оптималните стойности на k и l като функции на n.
2. Да се докаже, че най-добрата стъпка при квадратично търсене в нареден масив с n елемента е $\sqrt{n}$.

4.3. Двоично търсене

В случай на множество с голям брой елементи и голям брой търсения е удобно използването на *двоично търсене* след предварително сортиране на елементите. Двоичното търсене е до-

бър пример за приложение на римския принцип *разделяй и владей*, който ще бъде подробно разгледан в *глава 7*. Основната идея е сложната задача да се разбие на няколко по-прости, те от своя страна също да се разделят на по-прости и т. н. Процесът продължава до достигане на достатъчно проста задача с тривиално решение.

Как стоят конкретно нещата при двоичното търсене? Предполагаме, че имаме сортиран масив. Идеята е масивът да се раздели на два подмасива и да се определи в кой от тях търсеният елемент със сигурност не би могъл да се намира. Въпросният подмасив се изключва от по-нататъшни разглеждания, а по-нататъшните усилия се насочват към по-перспективния. Би могло да се възрази, че двоичното търсене не представлява класически пример на приложение на стратегията *разделяй и владей*, тъй като тя предполага разглеждане и на *двата* подмасива. От горното изложение обаче се вижда, че тя не забранява ранно отрязване на безперспективните подмножества.

Как става разделянето? Нека означим търсения елемент с x и предположим, че масивът е сортиран. Да сравним x със средния му елемент (При четен брой елементи има два средни елемента и няма значение кой ще изберем.). В случай на равенство търсенето приключва успешно. Ако x се окаже по-малък от средния елемент, веднага можем да определим безперспективния подмасив. Действително, това са елементите вдясно от средата: тъй като масивът е сортиран, те всичките са по-големи от средния, а той пък от своя страна вече се е оказал по-голям от x. Аналогично се подхожда в случая, когато x се окаже по-голям от средния елемент, като тогава се отхвърля левият подмасив. След това същият процес се прилага към неотхвърления подмасив. На всяка стъпка разделяме перспективния масив на два подмасива с приблизително равен брой елементи. Процесът приключва успешно при намиране на търсения елемент, или достигане до празния масив. Не е трудно да се види, че процесът винаги завършва. Действително, на всяка стъпка разглежданият масив намалява поне наполовина. *Поне*, тъй като винаги се отхвърля средният елемент на масива, а това при нечетен брой елементи означава, че се отхвърлят с един елемент повече отколкото остават. Впрочем това, че винаги се отхвърля поне един елемент, а именно средният, е съществено, тъй като някое от двете множества (лявото или дясното) може да се окаже празно. Въпреки това, и в този случай броят на елементите в разглеждания подмасив ще намалее строго (поне с 1).

Следвайки директно горните предписания, получаваме следната очевидна рекурсивна реализация на двоичното търсене (извиква се с `binSearch(ключ_който_търсим,0,n-1)`):

```
unsigned binSearch(int key, int l, int r)
{ int mid;
  if (l > r)
    return NOT_FOUND;
  mid = (l + r) / 2;
  if (key < m[mid].key)
    return binSearch(key,l,mid-1);
  else if (key > m[mid].key)
    return binSearch(key,mid+1,r);
  else
    return mid;
}
```

🖫 binsear0.c

Описаният по-горе процес по принцип е рекурсивен. На практика обаче на всяка стъпка се разглежда *точно едно* от двете подмножества. Вземайки предвид това, не е трудно да реализираме съответно итеративно решение:

```
unsigned binSearch(int key)
{ int l = 0, r = n-1, mid;
  while (l <= r) {
    mid = (l + r) / 2;
    if (key < m[mid].key)
      r = mid - 1;
```

```
    else if (key > m[mid].key)
      l = mid + 1;
    else
      return mid;
  }
  return NOT_FOUND;
}
```

🖫 binsear1.c

Горната функция използва два индекса `l` и `r`, указващи съответно лявата и дясната граница на разглежданата област. Разделянето на всяка стъпка става по средата `mid` на областта. Функцията връща позицията, на която се намира елементът с ключ `value` или –1, в случай на неуспех. Тъй като на всяка стъпка на алгоритъма разглежданият подмасив намалява поне наполовина, то горният алгоритъм извършва не повече от $\lceil \log_2 n \rceil + 1$ сравнения. Това е горната граница както за успешно, така и за неуспешно търсене [*Уирт-1980*].

Да се опитаме да подобрим горния вариант на програмата. Очевидно основната идея на двоичното търсене твърдо "закова" горната граница на $\lceil \log_2 n \rceil + 1$ сравнения. Въпреки това, място за подобрения има. Ще започнем с проста модификация на предложената програмна реализация и по-специално на начина на задаване на границите на разглежданата област. По-горе това ставаше с помощта на двойката индекси `l` и `r`, указващи съответно левия и десния ѝ край. Сега ще преминем към ново представяне, при което ще използваме `offset = r-l` вместо `r`. Ще искаме още `offset` винаги да бъде степен на двойката. В случай, че размерът на първоначалната област е степен на двойката, това свойство очевидно ще се поддържа от само себе си без допълнителни усилия от наша страна. В противен случай ще се наложи да си го осигуряваме. Най-простият начин за справяне с проблема е на първата стъпка да разделим масива на две области, които непременно да имат размер, степен на двойката. Това не е трудно да се извърши, макар че ще доведе до пресичане на областите от първата стъпка. Така например, ако размерът на масива е 1000, едно добро решение е на първата стъпка да го разделим на следните области: (1, 2, ..., 512) и (489, 490, ..., 1000).

Двете области имат еднакъв размер, а именно 512, което е максималната степен на 2, ненадвишаваща размера на масива $n = 1000$. Елементите 489, 490, …, 512 принадлежат и на двете области. За сметка на това обаче на всяка следваща стъпка размерът ще остава степен на двойката, при което общата ефективност на алгоритъма се очаква да се подобри. Тук тежката операция деление е заменена с поразредно изместване надясно, което води до допълнително подобрение. (Всъщност, това е малко спорно при днешните процесори.)

```
unsigned getMaxPower2(unsigned k)
{ unsigned pow2;
  for (pow2 = 1; pow2 <= k; pow2 <<= 1) ;
  return pow2 >> 1;
}

unsigned binSearch(int key)
{ unsigned i, l, ind;
  i = getMaxPower2(n);
  l = m[i].key >= key ? 0 : n - i + 1;
  while (i > 0) {
    i = i >> 1;
    ind = l + i;
    if (m[ind].key == key)
      return ind;
    else if (m[ind].key < key)
      l = ind;
  }
  return NOT_FOUND;
}
```

🖫 binsear2.c

Бихме могли да се освободим от едното сравнение във вътрешността на цикъла. За сметка на това обаче вече няма да можем да прекратяваме преждевременно работата на алгоритъма при откриване на търсения елемент преди последната стъпка. Така промяната се оказва в известен смисъл спорна:

```
unsigned binSearch(int key)
{ unsigned i, l;
  i = getMaxPower2(n);
  l = m[i].key >= key ? 0 : n - i + 1;
  while (i > 1) {
    i = i >> 1;
    if (m[l+i].key < key)
      l += i;
  }
  return (l < MAX && m[++l].key == key ? l : NOT_FOUND);
}
```

🖫 binsear3.c

Нека разгледаме по-внимателно идеята на алгоритъма. Не е трудно да се забележи, че, който и от горните варианти да вземем, все едно за всяка стойност на размера *n* на областта, редът на последователните сравнения за всяка стъпка е предварително определен. Тогава бихме могли да ги разпишем изрично, премахвайки необходимостта от деление и от цикъл въобще. Получената програма се оказва изключително ефективна. Бентли съобщава за наблюдавано подобрение от около 4,5 пъти спрямо първоначалния класически вариант (*виж* [*Bentley-1986*]). Разбира се, следва да се отбележи очевидното ограничение, че за разписването на цикъла в последователни условни конструкции, е необходимо размерът на областта да бъде поне приблизително известен. Освен това външният вид на програмата е способен да убие всеки ентусиазъм:

```
unsigned binSearch(int key)
{ unsigned l = 0;
  if (m[512].key   < key) l = 1000-512+1;
  if (m[l+256].key < key) l += 256;
  if (m[l+128].key < key) l += 128;
  if (m[l+ 64].key < key) l +=  64;
  if (m[l+ 32].key < key) l +=  32;
  if (m[l+ 16].key < key) l +=  16;
  if (m[l+  8].key < key) l +=   8;
  if (m[l+  4].key < key) l +=   4;
  if (m[l+  2].key < key) l +=   2;
  if (m[l+  1].key < key) l +=   1;
  return (l < 1000 && m[++l].key == key ? l : NOT_FOUND);
}
```

🖫 binsear4.c

Макар двоичното търсене да е изключително ефективно, то следва да се използва предпазливо. Основният му недостатък е, че непременно изисква пряк достъп до елементите на множеството, което ни ограничава до използването на масив и не позволява никакви динамични структури. Изискването за наредба на елементите на множеството пък от своя страна създава още по-големи проблеми пред вмъкването. Действително, ако вмъкваме новите елементи в края на масива, ще трябва да го сортираме преди всяко търсене, следващо вмъкване, което очевидно е крайно неразумно. Налага се вмъкване, при което масивът да остане сортиран. Тази операция е много неефективна, особено при по-малка стойност на ключа на вмъквания $(n+1)$-ви елемент, тъй като това води до изместване на всички по-големи от него елементи с една позиция надясно. Това

изисква изместване средно на $n/2$, а в най-лошия случай — на n елемента (n е броят на елементите *преди* вмъкването).

По-горе считахме изрично, че множеството не допуска повтаряне на ключове. Не е трудно обаче да се забележи, че дори и да допуснем повтаряне, двоичното търсене отново ще работи. Действително, то отново ще намира точно един елемент, в случай че такъв изобщо съществува. Сега остава да се съобрази, че масивът е сортиран. Тогава останалите елементи с този ключ следва да се намират в едната или в двете посоки, непосредствено до намерения елемент. Същата идея може да се използва за решаване на по-общата задача за намиране на всички елементи, попадащи в даден интервал.

Проблеми, свързани с необходимостта от преместване на голям брой елементи, възникват и при изтриването. Всъщност, при изтриване на малък брой елементи, е разумно те просто да се *маркират* като изтрити, без преместване. При търсене те следва да се прескачат по подходящ начин. Най-добре е ключовете им все пак да се използват при търсенето, и, едва в случай на успешно завършване, да се проверява дали намереният елемент не е изтрит. След това да се извършва евентуално търсене сред съседните му елементи в двете посоки.

Задачи за упражнение:

1. При какви стойности на n двоичното търсене ще бъде съответно 10, 100, 1000 пъти по-бързо в сравнение с класическото последователно търсене (*виж 4.1.*)?
2. Да се сравнят теоретично и емпирично различните варианти на двоичното търсене.

4.4. Фибоначиево търсене

Макар на пръв поглед да изглежда странно, понякога бихме могли да постигнем по-висока скорост на търсене, ако се откажем от разделянето на две *равни* части. По-долу ще разгледаме две модификации на двоичното търсене, всяка от които се основава на специфично съображение. Действително, основен недостатък на двоичното търсене е тежката операция деление, извършвана на всяка стъпка от изпълнението на алгоритъма. За съжаление, делението на две е тежка операция и може да забави алгоритъма. Ето защо е препоръчително при разработването на практически приложения редът

```
mid = (l + r) / 2;
```

да се замени с

```
mid = (l + r) >> 1;
```

Предложената модификация съществено използва вътрешното представяне на числата в двоична бройна система и факта, че делението на 2 може да се замени с отхвърляне на последната цифра на делимото, т. е. с изместване на една позиция надясно. Практически почти всички съвременни процесори разполагат с такава машинна инструкция. Ще припомним, че за произволна бройна система с основа p е в сила: Частното при целочислено деление на p с частно и остатък може да се намери чрез отхвърляне на последната цифра на делимото (остатъка от делението).

Макар и много малко вероятно, може да се окаже, че използваната от Вас машина не разполага с подходяща инструкция за изместване надясно с един бит (на авторите не е известна такава машина). В такъв случай е удачно да се използва така нареченото Фибоначиево търсене. Ще припомним, че числата на Фибоначи се задават с рекурентната формула (*виж 1.2.2.*):

$$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2},\ n \geq 2$$

Фибоначиевото търсене силно напомня двоичното — масивът се разделя на две части, като едната се изключва от по-нататъшни разглеждания. Единствената разлика е начинът на избор на елемента, с който ще сравняваме. Нека предположим, че търсенето става в n-елементен масив, като $n = F_k - 1$. На първата стъпка сравняваме търсения ключ x с `m[`F_{k-1}`].key`. В случай на равенство търсенето приключва успешно. В случай че $x <$ `m[`F_{k-1}`].key` на следващата стъпка

разглеждаме елементите 1, 2, ..., F_{k-1}–1. Ако пък x > `m[`F_{k-1}`].key` на следващата стъпка остават за разглеждане елементите F_{k-1}+1, F_{k-1}+2, ..., $n = F_k$. Директно се пресмята, че в първия случай за следващата стъпка получаваме последователност с дължина F_{k-1}–1, а във втория — F_{k-2}–1. Процесът се повтаря върху неотхвърлената част до намиране на търсения елемент или достигане до празната последователност (*виж* [*Horowitz-1977*]). Една възможна реализация на Фибоначиевото търсене изглежда така ($l \geq 0$ и е такова, че $F_k + l = n + 1$ и $F_{k+1} > n$+1):

```
unsigned fib[MAX]; /* Числата на Фибоначи, ненадвишаващи n */

unsigned findFib(unsigned n)
{ unsigned k;
  fib[0] = 0;
  fib[1] = 1;
  for (k = 2; ; k++)
    if ((fib[k] = fib[k-1] + fib[k-2]) > n)
      return k-1;
  return 0;
}

unsigned fibSearch(int key)
{ int p,q,r,k;
  k = findFib(n);
  p = fib[k-1];
  q = fib[k-2];
  r = fib[k-3];
  if (key > m[p].key)
    p += n - fib[k] + 1;
  while (p > 0)
    if (key == m[p].key)
      return p;
    else
      if (key < m[p].key)
        if (0 == r)
          p = 0;
        else {
          int t;
          p -= r;
          t = q;
          q = r;
          r = t-r;
        }
      else
        if (1 == q)
          p = 0;
        else {
          p += r;
          q -= r;
          r -= q;
        }
  return NOT_FOUND;
}
```
🖫 fibsear.c

Каква е ефективността на Фибоначиевото търсене? Оказва се, че ползата от премахването на делението не е голяма. Подобно на двоичното търсене, редът на извършване на сравненията е предварително определен за всяка стойност на n. На практика Фибоначиевото търсене строи дърво. Лесно се забелязва, че разликата във височините на поддърветата за всеки връх е най-

много 1, т. е. дървото е *балансирано*. Този тип дървета са известни в литературата като *дървета на Фибоначи* и вече бяха разгледани в параграф *2.4.*, където показахме, че те са най-лошият случай на балансирани дървета, като са по-високи от съответното идеално балансирано дърво с 45%. Така, в най-лошия случай Фибоначиевото търсене изследва дърво, с 45% по-високо от това на двоичното търсене (*виж 4.3.*), поради което се оказва по-неефективно от него.

Задача за упражнение:

Да се сравнят теоретично и емпирично двоичното и Фибоначиевото търсене.

4.5. Интерполационно търсене

Когато човек търси "Бонев" в телефонния указател, той гледа в началото на указателя, а когато търси "Янев" — в края. Не бихме ли могли по подобен начин да оптимизираме двоичното търсене (*виж 4.3.*)?

Нека го разгледаме по-внимателно. При двоичното търсене избирахме средния елемент `mid` по формулата `mid = (l+r)/2`. Да го запишем по друг начин:

```
mid = l + (r-l)/2      (*)
```

Получената формула показва, че следващата позиция на разделяне на интервала се получава, като към началото му добавим половината на дължината му. В случая на телефонния указател, човек не работи по тази формула. Вместо в средата той търси съответно в началото или в края на указателя. Това може да се изрази чрез формула от типа на (*), като за целта 1/2 се замени с друго подходящо число. Как да изберем това число? Нека се върнем отново към телефонния указател. Там човек се опитва да определи приблизителната позиция на търсеното име, изхождайки от първата му буква, знаейки първата и последната букви от азбуката. По аналогичен начин бихме могли да опитаме да намерим приблизителната позиция на произволен елемент `x`, знаейки стойността `x`, както и стойностите на началото и края на разглеждания интервал. Ясно е, че тази относителна позиция може да се зададе с интерполационната формула:

```
mid = l + k*(r-l),
```

където

```
k = (x - m[l].key) / (m[r].key - m[l].key)
```

Коефициентът k, който замества константата 1/2 всъщност ни дава приблизителната позиция на търсения елемент в разглеждания интервал. При реализацията на интерполационното търсене следва да се отчетат някои специфични особености, свързани с пресмятането стойността на k. На първо място, следва да се предпазим от деление на 0, в случай, че стойностите на всички ключове от разглеждания интервал [`l`,`r`] съвпадат. На второ място, следва да гарантираме, че $k \in [0,1]$. Действително, в противен случай стойността на `mid` би излязла извън границите на разглеждания интервал [`l`,`r`]. Когато k излезе извън границите [0,1] спокойно можем да считаме, че търсеният елемент липсва.

Каква е ефективността на разгледаното подобрение? В случай на равномерно разпределение интерполационното търсене дава наистина забележителни резултати! Тогава горното приближение, зададено от k, се оказва много близо до истинската позиция на търсения елемент и той бива открит значително по-бързо в сравнение с двоичното търсене. Доказва се, че тогава интерполационното търсене извършва по-малко от $\log_2(\log_2 n) + 1$ сравнения както при успешно, така и при неуспешно търсене. За да може читателят да оцени ефективността на изложения метод, ще посочим, че функцията $\log_2(\log_2 n)$ расте изключително бавно. Например $\log_2(\log_2 1000000000) < 5$.

```
unsigned interpolSearch(int key)
{ unsigned l, r, mid;
  float k;
```

```
  l = 0; r = n - 1;
  while (l <= r) {
    if (m[r].key == m[l].key)
      if (m[l].key == key)
        return l;
      else
        return NOT_FOUND;
    k = (float) (key - m[l].key) / (m[r].key - m[l].key);
    if (k < 0 || k > 1)
      return NOT_FOUND;
    mid = (unsigned)(l + k*(r-l) + 0.5);
    if (key < m[mid].key)
      r = mid - 1;
    else if (key > m[mid].key)
      l = mid + 1;
    else
      return mid;
  }
  return NOT_FOUND;
}
```

🖫 interpol.c

Въпреки безспорните предимства на интерполационното търсене, то следва да се използва изключително внимателно. На първо място не трябва да се забравя, че то е подобрение (много добро) на двоичното търсене, поради което наследява основните и най-неприятните му недостатъци, свързани с изискването за пряк достъп и наредба на елементите на множеството. На второ място, следва да се отбележат допълнителните изисквания и недостатъци, които се налагат (всяко нещо си има цена!). От една страна се изисква ключовете да бъдат числа или да могат лесно да се интерпретират като такива, така че k да попада винаги в интервала [0,1]. В противен случай методът може да се окаже неприложим. На трето място, следва да се обърне специално внимание на факта, че тук се работи с числа с плаваща запетая и се използва тежката аритметична операция деление. За съжаление, тук не можем да се отървем от делението така лесно, както при двоичното търсене: от една страна делението вече не е на степен на 2, а от друга — не е целочислено.

Изключително съществено се оказва и изискването ключовете на елементите да са равномерно разпределени. В случай, че това не е така, то скоростта му може да падне чувствително под тази на двоичното търсене, приближавайки се до линейна. Не е трудно да се конструира такъв пример. Нека за целта се опитаме да търсим числото 2 в следното множество:

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 10000

Освен това при малки стойности на n числата $\log_2 n$ и $\log_2 (\log_2 n)$ не се различават съществено, поради което обикновено се счита, че използването на интерполационното търсене не си струва риска. От друга страна, при особено големи файлове, при големи ключове или при външно разположение на данните, когато сравненията са особено скъпи, следва твърдо да се предпочита интерполационното търсене.

Задача за упражнение:

Да се сравнят теоретично и емпирично двоичното и интерполационното търсене.

4.6. Въпроси и задачи

4.6.1. Задачи от текста

Задача 4.1.

Да се докаже, че средният брой сравнения при последователно търсене е $(n+1)/2$ (*виж 4.1.*).

Задача 4.2.

Да се определи средният брой сравнения при последователно търсене в сортиран списък (*виж 4.1.1.*). Да се сравни с класическото последователно търсене (*виж 4.1.*).

Задача 4.3.+

Да се реализира претърсване на списък с преподреждане, базирано на брояч на достъпа. Да се сравни ефективността му с тази на предложената в *4.1.2.* реализация без броячи.

Задача 4.4.

Да се сравни основната идея на последователното търсене с преподреждане без броячи (*виж 4.1.2.*) с тази на адаптивните методи за компресиране (*виж 10.5.*).

Задача 4.5.

Да се определи средният брой сравнения при последователно търсене *с преподреждане* (*виж 4.1.2.*). Да се сравни с класическото последователно търсене (*виж 4.1.*).

Задача 4.6.

Да се сравнят в най-добрия, средния и най-лошия случай трите разгледани метода за последователно търсене:

а) в неподреден масив (*виж 4.1.*)

б) в сортиран списък (*виж 4.1.1.*)

в) в несортиран списък с преподреждане (*виж 4.1.2.*)

Задача 4.7.

Да се реализира двустепенно търсене със стъпки k и l. Да се намерят експериментално оптималните стойности на k и l като функции на n (*виж 4.2.*).

Задача 4.8.

Да се докаже, че най-добрата стъпка при квадратично търсене в нареден масив с n елемента е $\sqrt{n}$ (*виж 4.2.*).

Задача 4.9.

При какви стойности на n двоичното търсене (*виж 4.3.*) ще бъде съответно 10, 100, 1000 пъти по-бързо в сравнение с класическото последователно търсене (*виж 4.1.*)?

Задача 4.10.

Да се сравнят теоретично и емпирично различните варианти на двоичното търсене, реализирани в *4.3.*

Задача 4.11.

Да се сравнят теоретично и емпирично двоичното (*виж 4.3.*) и Фибоначиевото търсене (*виж 4.4.*).

Задача 4.12.

Да се сравнят теоретично и емпирично двоичното (*виж 4.3.*) и интерполационното търсене (*виж 4.5.*).

4.6.2. Други задачи

Задача 4.13. *Намиране на всички елементи с даден ключ.*

Да се модифицират предложените алгоритми за търсене, така че да намират не само един, а *всички* елементи, които имат даден ключ.

Задача 4.14. *Търсене в матрица.*

Да се реализира функция за търсене на число в дадена матрица с размер $n \times m$.

Задача 4.15. *Търсене в символен низ.*

Да се реализира функция за търсене на символен низ в даден масив от символни низове.

Задача 4.16. *Радиус на достижимост.*

Даден е масив от естествени числа, елемент с ключ k (k — естествено число) и естествено число d. Да се предложи алгоритъм и да се реализира функция за намиране на всички елементи x в масива, за които $|k - \texttt{x.key}| \leq d$.

Глава 5
Теория на графите

"There are two possible outcomes:
If the result confirms the hypothesis,
then you've made a measurement.
If the result is contrary to the hypothesis,
then you've made a discovery."

~ E. Fermi

5.1. Основни понятия

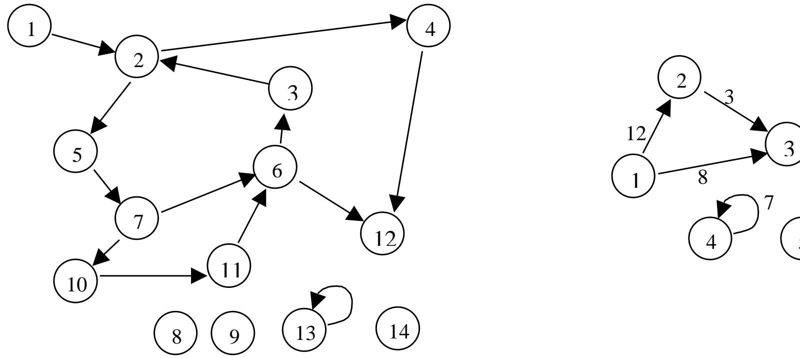
Теорията на графите е клон на съвременната математика, претърпял впечатляващо развитие през последните няколко десетилетия. Графите са едни от най-полезните абстрактни структури от данни и в информатиката. Много задачи от различни области на науката и практиката могат да бъдат моделирани с граф и решени с помощта на съответен алгоритъм върху него. Поради тази причина ще им отделим цяла глава, като до края на тази книга те ще присъстват в още много задачи. Преди да посочим някои примери, ще дефинираме строго понятието.

Дефиниция 5.1. *Краен ориентиран граф* се нарича наредената двойка (V, E), където:

- $V = \{v_1, v_2, ..., v_n\}$ е крайно множество от *върхове*
- $E = \{e_1, e_2, ..., e_m\}$ е крайно множество от *ориентирани ребра*. Всеки елемент $e_k \in E$ (k=1,2, ..., m) е *наредена* двойка (v_i, v_j), $v_i, v_j \in V$, $1 \leq i, j \leq n$.

Ако в допълнение е дадена числова функция f:$E \to R$, съпоставяща на всяко ребро e_k *тегло* $f(e_k)$, графът се нарича *претеглен*. Понякога, когато няма опасност от двусмислие, ще казваме само ребра вместо ориентирани ребра. Ще отбележим, че някои автори използват термина *дъга* за ориентирано ребро, а *ребро* — само за неориентирано ребро.

Най-често, ориентиран граф се представя графично в равнината чрез множество от точки (кръгчета), означаващи върховете му, и свързващи ги стрелки — ребрата на графа.



Фигура 5.1а. Ориентиран граф.

Фигура 5.1б. Ориентиран претеглен граф.

На *фигура 5.1а.* е показан граф с 14 върха и 13 ребра. Всеки връх $v_i \in V$ сме изобразили като кръгче и сме му съпоставили номер — уникално естествено число. Така, в случаите, когато

това няма да доведе до допълнителни недоразумения, вместо произволно множество V, ще приемем, че V е множеството от първите n естествени числа. Това не намалява общността на разглежданите алгоритми и същевременно води до редица удобства при реализацията — например, можем да използваме върховете като индекси на масив и др.

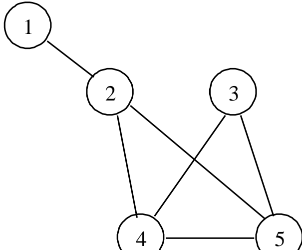
Всяко ребро $(i,j)\in E$ сме изобразили като стрелка, насочена от връх i към връх j. Забележете, че е допустимо ребро да излиза и влиза в същия връх: например връх 13. Такива ребра се наричат *примки*. Ако графът е претеглен, теглото на всяко ребро ще записваме до съответната стрелка, както на *фигура 5.1б.*

В редки случаи, при разглеждането на някои специфични задачи, е възможно да използваме и други начини за индексиране на върховете (например с малки латински букви).

Дефиниция 5.2. *Краен неориентиран граф* се нарича наредената двойка (V, E), където:

- $V = \{v_1, v_2, ..., v_n\}$ е крайно множество от *върхове*
- $E = \{e_1, e_2, ..., e_m\}$ е крайно множество от *неориентирани ребра*. Всеки елемент $e_k\in E$ $(k = 1,2, ..., m)$ е *ненаредена* двойка (v_i, v_j), $v_i,v_j\in V$, $1 \le i, j \le n$.

Ако освен това е зададена функция $f(i,j)$, съпоставяща целочислена стойност на всяко ребро $(i,j)\in E$, $f(i,j) = f(j,i)$, графът се нарича *претеглен неориентиран граф*.



Фигура 5.1в. Неориентиран граф.

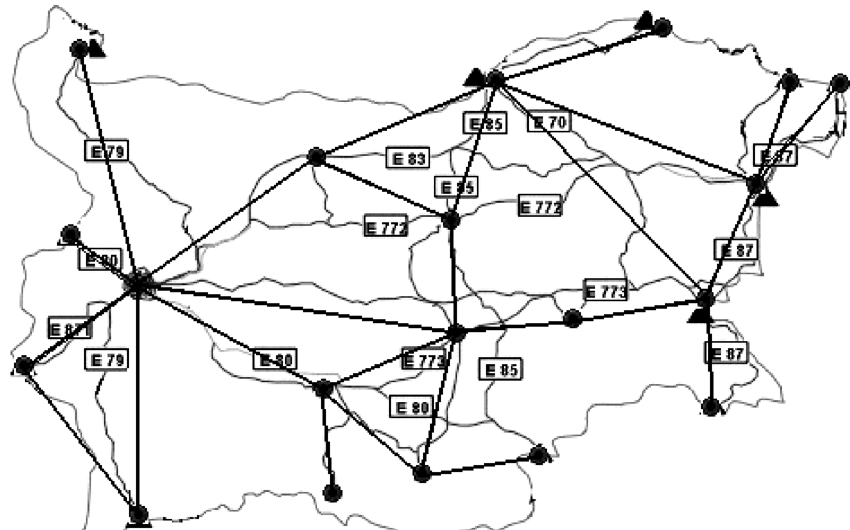
На *фигура 5.1в.* е показан неориентиран граф — върховете са изобразени с кръгчета, а всяко ребро (i,j) — с линия.

Понякога ще искаме да разглеждаме неориентирания граф $G(V,E)$ като ориентиран $G'(V,E')$. На всеки неориентиран граф може да се съпостави съответен ориентиран: на всяко ребро $(i,j)\in E$ се съпоставят ребрата $(i,j)\in E'$ и $(j,i)\in E'$ в G'. Подобен подход е полезен, когато искаме да приложим алгоритъм, валиден за ориентирани графи, а в постановката на задачата е даден неориентиран граф.

Почти всяка съвкупност от обекти с дефинирани връзки между тях може да бъде представена като граф. Ето няколко примера:

1) Да разгледаме транспортната карта на България и по-големите градове. Те могат да бъдат представени като върхове на граф, а преките пътища между тях — като ребра. Теглата на ребрата ще бъдат дължините на преките пътища. Илюстрация на примера е *фигура 5.1г.*, където графът е неориентиран, като е възможно да разглеждаме и ориентиран граф.
2) Компютърна мрежа може да бъде представена чрез неориентиран граф, в който компютрите са върхове, а всяко ребро между два върха показва, че съответните компютри са пряко свързани в мрежа.
3) Множеството от страниците в Интернет и връзките помежду им могат да бъдат представени като ориентиран граф.
4) Няколко химични съединения могат да бъдат представени като върхове на неориентиран граф: Всяко ребро ще показва дали съответните химични съединения могат да си взаимодействат. Аналогичен пример е върховете да представят видове декоративни риби, а ребрата да показват дали два вида риби могат да съжителстват заедно, или — не.

5) Етапите в изработването на едно изделие могат да се представят като върхове на ориентиран граф. Ребрата показват кой етап кого предхожда в процеса на изработване на изделието.



Фигура 5.1г. Транспортна карта на България.

Могат да бъдат посочени още много примери за графи, както и да се формулират задачи върху тях. Така например, може да се интересуваме от най-късия или най-евтиния път между два града в пример 1), или от минималното време за изработване на цялото изделие в пример 5).

За пълноценното разглеждане на материала по-нататък е необходимо да дефинираме още някои понятия от теория на графите. Не е нужно читателят да се опитва да запомни всички дефиниции сега. Достатъчно е при разглеждането на съответната част по-късно той да се върне тук и да си припомни, каквото му е необходимо.

Дефиниция 5.3. Даден е ориентиран (неориентиран) граф $G(V,E)$. Ако в множеството на ребрата му се допуска повторение (т.е. E е *мултимножество*), G се нарича *мултиграф.*

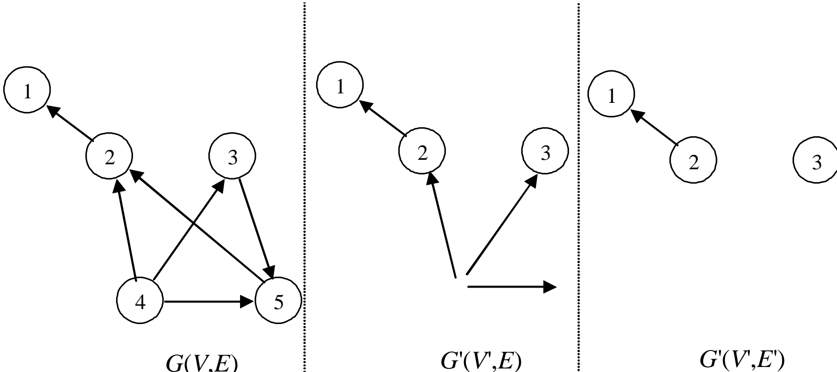
Забележете, че ако мултиграфът е претеглен, то на различните ребра (i,j) се съпоставят отделни (в общия случай различни) тегла $f^{(1)}(i,j)$, $f^{(2)}(i,j)$, ...

Дефиниция 5.4. Даден е ориентиран граф $G(V,E)$. Два върха i и j ($i,j \in V$) се наричат *съседни*, ако поне едно от ребрата (i,j) и (j,i) принадлежи на E. В такъв случай още казваме, че i и j са *краища* за ребрата (i,j) и (j,i). За всяко ребро (i,j) върхът i се нарича *предшественик* на j, а j — *наследник* на i. Всеки от върховете i и j се нарича *инцидентен* с реброто (i,j). Казваме, че две ребра са *инцидентни*, когато са инцидентни с един и същ връх. В случай на неориентиран граф, понятията *съседен* връх, *край* на ребро, *инцидентност* на върхове и ребра се дефинират аналогично.

Дефиниция 5.5. Даден е ориентиран граф $G(V,E)$. *Полустепен на изхода на връх i*, $i \in V$ се нарича броят на ребрата (i,j), $j \in V$. Аналогично, броят на ребрата (j,i), $j \in V$ се нарича *полустепен на входа на i.* Сборът от полустепента на входа и полустепента на изхода се нарича *степен* на върха i. *Изолиран* се нарича връх, чиято степен е 0. При неориентиран граф степен на връх i се нарича броят на ребрата (i,j), инцидентни с него.

На *фигура 5.1а.* степента на връх *2* е 4, а връх *14* е изолиран.

Дефиниция 5.6. Нека е даден ориентиран (неориентиран) граф $G(V, E)$ и $V' \subseteq V$. Ако от E се изключат всички ребра (i, j), такива че $i \notin V'$ или $j \notin V'$, а останалите се запазят, то казваме, че $G'(V',E')$ е *индуциран от V' подграф* (или само *подграф*) на G (*виж фигура 5.1д.*).

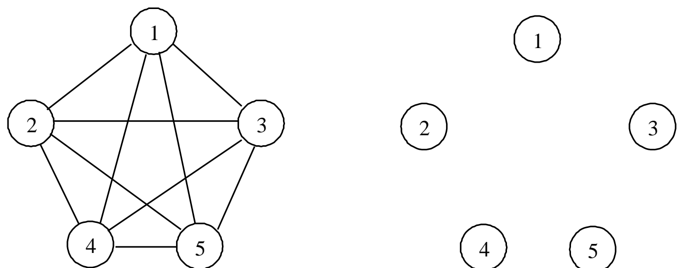


Фигура 5.1д. Индуциран от множеството $V' = \{1, 2, 3\}$ подграф G' на G.

Дефиниция 5.7. *Път* в ориентиран (неориентиран) граф $G(V,E)$ се нарича последователност от върхове $v_1, v_2,..., v_k$ такава, че за всяко $i = 1, 2, \ldots, k-1$ е изпълнено $(v_i, v_{i+1}) \in E$. Върховете v_1 и v_k се наричат *краища* на пътя. Ако $v_1 = v_k$, пътят се нарича *цикъл*. Ако за всяко $i \neq j$ $(1 \leq i, j \leq k)$ следва $v_i \neq v_j$, то пътят се нарича *прост*. Съответно, ако $v_1 = v_k$, и всички останали върхове са различни, цикълът се нарича *прост*. Когато граф съдържа поне един цикъл, той се нарича *цикличен*, а в противен случай — *ацикличен*.

Дефиниция 5.8. Нека са дадени графите $G(V, E)$ и $G'(V, E')$, такива че за всяко $i, j \in V$, $i \neq j$, наредената двойка (i, j) принадлежи на точно едно от множествата E' или E. Тогава G' се нарича *допълнение* на G. Граф, несъдържащ ребра, се нарича *празен* (*фигура 5.1е.*). Ако за даден граф е изпълнено $(i, j) \in E$ за всяко $i, j \in V$, $i \neq j$, графът се нарича *пълен*.

Забележете, че последната дефиниция е симетрична, т.е. ако G' е допълнение на G, то и G е допълнение на G'.



Фигура 5.1е. Пълен и празен неориентиран граф.

Дефиниция 5.9 Ако граф $G'(V', E')$ с t върха е индуциран от G, и G' е пълен, казваме че G' е *t-клика на G*. Максималното t, за което G има *t-клика* се нарича *кликово число* за G.

Дефиниция 5.10. Ориентиран граф се нарича *слабо свързан*, ако всеки два върха i и j са краища на поне един път (т.е. съществува поне един път от i до j или от j до i). Когато в ориентиран граф за всеки два върха i, j съществува път както от i до j, така и от j до i, то графът се нарича *силно свързан*. Неориентиран граф се нарича *свързан*, ако съществува път между всяка двойка негови върхове i, j.

Дефиниция 5.11. *Диаметър* на граф се нарича максималното число k, такова че най-краткия прост път (пътят, в който участват минимален брой върхове) между произволни два върха $i,j \in V$ съдържа поне k върха.

Така например, ако разглеждаме страниците в Интернет и връзките помежду им като ориентиран граф, то диаметърът му k е максималният брой страници, през които трябва да преминем, започвайки от произволна страница и следвайки хипервръзки, за да достигнем до произволна друга страница в мрежата. Проучванията показват, че "диаметърът на Интернет" е приблизително равен на 19 [*Web-d*].

Дефиниция 5.12. *Силно* (*слабо*) *свързана компонента* в ориентиран граф $G(V, E)$ се нарича всеки *подграф* $G'(V', E')$, за който е изпълнено:

1) G' е силно (слабо) свързан граф.
2) Не съществува друг силно (слабо) свързан собствен подграф $G''(V'',E'')$ на G, такъв, че G' да бъде подграф на G''.

Аналогично се дефинира *свързана компонента* в неориентиран граф.

Дефиниция 5.13. Неориентиран свързан граф без цикли се нарича *дърво*. Ако допълнително изберем някой връх от дървото за корен, то получената структура се нарича *кореново дърво*.

Дефиницията е алтернативна на разгледаната в *2.3.* дефиниция на кореново дърво).

Дефиниция 5.14. *Покриващо* (*обхващащо*) *дърво* в свързан неориентиран граф $G(V,E)$ се нарича всеки свързан ацикличен подграф $G'(V, E')$ на G.

Задача за упражнение:

Да се сравнят дефиниции 5.13. и 2.2.

5.2. Представяне и прости операции с граф

Досега използвахме два начина за представяне на граф:

- Директно следвайки дефиницията — чрез дадените множества V и E.
- Графично: чрез множество от точки (кръгчета) и връзки (стрелки) между тях.

Второто представяне (макар и нагледно) е неприложимо, а първото — често неудобно, за компютърно обработване. Затова съществуват други начини за представяне на граф — те ще бъдат предмет на настоящия параграф. Изборът на най-подходящото компютърно представяне зависи от конкретната задача — добре е да се използва такова, при което операциите, които се извършват най-често, да имат по-малка алгоритмична сложност (или да е необходимо по-малко памет).

5.2.1. Списък на ребрата

Най-близко до дефиницията на граф е представяне със списък от ребрата му. Ориентиран граф представяме като списък от наредени двойки (i,j). В случай на неориентиран граф, двойките са ненаредени. При претеглен (ориентиран) граф разглеждаме (наредени) тройки (i,j,k), където реалното число k е теглото $f(i,j)$ на съответното ребро. Последното бихме могли да реализираме на Си, като използваме масив `float A[3][m]`, където m е броят на ребрата на графа.

Очевидно, при това представяне необходимата памет ще бъде $\Theta(m)$, каквато е и сложността на проверката дали два върха са съседни, което се счита за една от най-често извършваните операции. В случай, че сортираме ребрата, ще можем прилагаме двоично търсене

(*виж 4.3.*) и така да намалим сложността на проверката до $\Theta(\log_2 m)$. Както ще видим по-долу, съществуват представяния, при които последната сложност е доста по-малка: функция на n, и дори константа. Така, това представяне е приложимо предимно за *разредени* графи, т.е. графи, които имат относително малък брой ребра: $m \in O(n)$.

5.2.2. Матрица на съседство, матрица на теглата

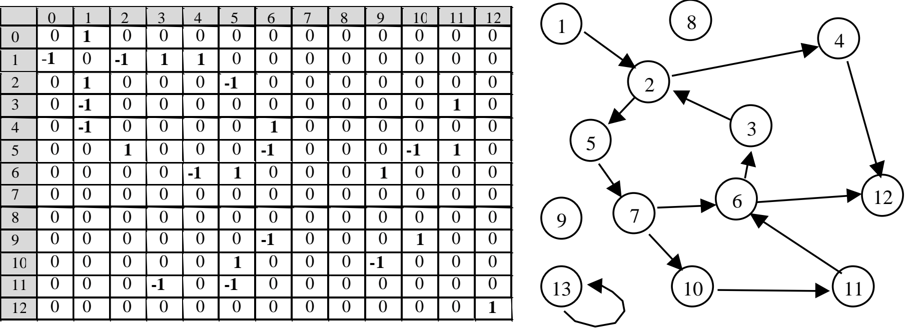
Матрицата на съседство е един от най-често използваните начини за представяне на граф в паметта. При него на ориентиран граф с n върха се съпоставя квадратна матрица `A[n][n]`. Стойността на `A[i][j]` е равна на 1, когато съществува реброто (i,j), и `A[i][j] = 0` — в противен случай. За удобство при реализирането на някои алгоритми, когато не съществува реброто (i, j), но съществува (j, i), на позиция `A[i][j]` в матрицата се записва стойност -1 (и съответно 1 — на `A[j][i]`).

Ако графът е неориентиран, то `A[i][j] = 1`, когато съществува реброто (i, j) или (j, i), и `A[i][j] = 0` в противен случай. При непретеглен мултиграф, в полето `A[i][j]` записва не 1, а броят на ребрата между върховете i и j. Когато графът е претеглен, на позиция `A[i][j]` се записва теглото $f(i, j)$, и матрицата `A[][]` се нарича *матрица на теглата*.

В някои задачи се дефинират и тегла за *върховете* на графа. Те могат да бъдат записани на позициите `A[i][i]` за всеки връх i от графа. Тези полета от матрицата няма да бъдат заети, единствено в случай, че графът не съдържа примки (ребра от вида (i,i)).

При неориентиран граф матрицата на съседство е *симетрична* относно главния си диагонал. В този случай може да се приложи по-специфично запазване в паметта (например линеаризиране, с цел да се спести дублиращата се част, която заема излишно $\frac{n(n-1)}{2}$ повече клетки памет [*Рахнев, Гъров, Гаврилов-1995*].

Важно преимущество на представянето чрез матрица на съседство е, че проверката за съществуване на ребро между два върха става с единствена операция. От друга страна намирането на всички *наследници* на даден връх има сложност $\Theta(n)$, независимо от това колко наследника има върхът (т.е. дори върхът да има единствен наследник, пак ще трябва да се извършат всичките n проверки). Пример за ориентиран граф и съответстващата му матрица на съседство е даден на *фигура 5.2.2*. Забележете, че докато графичната номерация на върховете започва от 1, то при реализация на Си индексирането на елементите започва от 0. Тази специфична особеност трябва да се има в предвид при разглеждането на всички реализации на алгоритми от настоящата глава.



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	**1**	0	0	0	0	0	0	0	0	0	0	0
1	**-1**	0	**-1**	**1**	**1**	0	0	0	0	0	0	0	0
2	0	**1**	0	0	0	**-1**	0	0	0	0	0	0	0
3	0	**-1**	0	0	0	0	0	0	0	0	0	**1**	0
4	0	**-1**	0	0	0	0	**1**	0	0	0	0	0	0
5	0	0	**1**	0	0	0	**-1**	0	0	0	**-1**	**1**	0
6	0	0	0	0	**-1**	**1**	0	0	0	**1**	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	**-1**	0	0	0	**1**	0	0
10	0	0	0	0	0	**1**	0	0	0	**-1**	0	0	0
11	0	0	0	**-1**	0	**-1**	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	**1**

Фигура 5.2.2. Матрица на съседство и съответният ѝ граф.

5.2.3. Списък на наследниците (списък на инцидентност)

При това представяне за всеки връх i от графа се пази списък с наследниците му. За графа от *фигура 5.2.2.* представянето ще има следният вид:

1 → {*2*}; *2* → {*4,5*}; *3* → {*2*}; *4* → {*12*}; *5* → {*7*}; *6* → {*3,12*}; *7* → {*6,10*}; *8* → {};
9 → {}; *10* → {*11*}; *11*→{*6*}; *12* → {}; *13* → {*13*}.

За компютърната реализация най-често се използват *динамични свързани списъци* (*виж 2.1.2.*). При тях сложността за намиране на всички наследници на даден връх е линейна по броя на наследниците: предимство пред матрицата на съседство, където намирането на наследниците е със сложност $\Theta(n)$. Недостатъкът обаче е, че проверката дали съществува ребро между два върха е също със сложност, линейно зависеща от броя на наследниците.

Възможна е и статична реализация на това представяне: едномерен масив с големина броя на наследниците за всеки един от върховете на графа. В този случай проверката дали има ребро между два върха се извършва значително по-ефективно — с логаритмична сложност. Това се постига, като се пазят наследниците в сортиран вид и се използва двоично търсене.

Ясно е, че както можем да поддържаме списък на наследниците, така можем да построим и *списък на предшествениците*, като в този случай сложностите на основните операции ще бъдат същите. На практика представяне на граф чрез списък на предшествениците се използва много рядко.

5.2.4. Матрица на инцидентност между върхове и ребра

При това представяне се използва матрица с размери $n \times m$. Ако разглеждаме неориентиран граф $G(V, E)$, за всяко ребро $e_k = (i, j)$ стълбът k ще съдържа две единици — на позиции i и j, а във всички останали елементи в стълба ще бъде записана 0. При ориентиран граф на i-та позиция се записва 1, а на j-та — –1, като всички останали елементи в стълба са нули. Ако имаме *примка* (i,i), тогава на i-тата позиция на стълба e_k се записва 2.

Това представяне намира приложение при някои специфични задачи (например намиране броя на покриващите дървета в граф, и др.), но като цяло се използва много рядко поради две причини. На първо място, нужната памет е $\Theta(m.n)$, като същевременно матрицата е силно разредена. Проверката, дали съществува ребро между два върха, също е неефективна и има сложност $\Theta(m)$.

5.2.5. Компоненти на свързаност

Последният начин за представяне, който ще разгледаме, е чрез компоненти на свързаност. Той се различава от предишните по това, че графът, веднъж представен в паметта, "се изгубва", и по-нататък е невъзможно еднозначното възстановяване на множествата на върховете и ребрата му, т.е. разглеждаме *представяне на определени свойства* на графа. Допълнително условие е графът да бъде неориентиран и непретеглен.

Графът се разделя на компоненти на свързаност, като на всяка компонента се съпоставя по едно множество. Практическата полза от това представяне е, че лесно може да се провери дали два върха са свързани с път — това е изпълнено тогава и само тогава, когато те принадлежат на едно и също множество.

За компютърната реализация може се използва едномерен масив `A[n]`, който приема стойности от 1 до n. `A[i]` има стойност k т.с.т.к i-тият връх от графа принадлежи на k-тата компонента на свързаност. Това представяне позволява да се запази огромен граф (необходимата памет е едва $\Theta(n)$), като за него може да провери дали два върха са свързани с път с едно-единствено сравнение: дали `A[i]` е равно на `A[j]`, или — не.

Матрицата на достижимост (която ще разгледаме отново в *5.4.2.*), е друг начин за представяне на *свойства* на граф чрез компоненти на свързаност. Той обаче е доста по-неефективен в сравнение с гореописания. При представяне на граф с матрица на достижимост се използва двумерен масив `A[n][n]`, в който стойността на `A[i][j]` е 1, ако има път между върховете i и j, и 0 — в противен случай.

5.2.6. Построяване и прости операции с графи

Основните операции, свързани с построяване и модифициране на граф, са следните:

- *създаване* на празен граф.
- *добавяне/премахване* на връх.
- *добавяне/премахване* на ребро.

По-нататък следват операциите, за които стана въпрос и в предходните точки:

- *проверка за съществуване на връх*
- *проверка за съществуване на ребро*
- *намиране на наследниците на даден връх*

Няма да уточняваме как се реализират тези операции при всеки един вид представяне в паметта. Като пример ще покажем реализация за граф, представен с *матрица на съседство* (*матрица на теглата*).

```
/* Максимален брой върхове в графа */
#define MAXN 200

/* Брой върхове в графа */
unsigned n;
/* Матрица на теглата на графа */
int A[MAXN][MAXN];

/* Модифициране на теглото на върха i */
A[i][i] = k;

/* Добавяне на ребро с тегло k, свързващо върховете i и j */
A[i][j] = k;
A[j][i] = k;              /* ако графът не е ориентиран */

/* Премахване на ребро, свързващо върховете i и j */
A[i][j] = 0;

/* Проверка дали има ребро между върховете i и j */
if (A[i][j] != 0) { /* има ребро */; } else { /* няма ребро */; }

/* Намиране на всички наследници на върха i */
for (k = 0;  k < n; k++) if (k != i) {
    if  (A[i][k] != 0) { /* върхът k е наследник на i */ ; }
  }
```

Преди да завършим, ще сравним сложностите на операциите при различните представяния. Във всяка задача ще се стремим към представяне, при което операциите, които се изпълняват най-често, имат най-малка сложност. Възможно е *едновременно* представяне на граф *по повече от един начин* в случай, че разполагаме с достатъчно памет, и когато това води до по-ефективна реализация на избраният алгоритъм.

Представяне / Операция	*Матрица на съседство*	*Списък на наследниците*	*Списък на ребрата*	*Матрица на инцидентност*	*Компоненти на свързаност*
Добавяне на ребро	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 m)$	$\Theta(1)$	$\Theta(n.\log_2 n)$
Изтриване на ребро	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 m)$	$\Theta(m)$	×

Проверка за съществуване на ребро	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 m)$	$\Theta(m)$	×
Намиране на наследниците на връх	$\Theta(n)$	$\Theta(d_i)$	$\Theta(n+\log_2 m)$	$\Theta(m)$	×
Проверка за свързаност на два върха с път	$\Theta(n^2)$	$\Theta(n+m)$	$\Theta(n.\log_2 m)$	$\Theta(m^2)$	$\Theta(1)$
Необходима памет	$\Theta(n^2)$	$\Theta(m)$	$\Theta(m)$	$\Theta(n.m)$	$\Theta(n)$

Таблица 5.2.6. Сравнение на сложностите на операциите при различни представяния.

В *таблица 5.2.6.* "×" означава, че графът не може да се възстанови еднозначно при избраното представяне и съответната операция не може да бъде реализирана.

Задача за упражнение:

Да се съобрази как могат да се постигнат сложностите в *таблица 5.2.6.*, за които не стана дума в настоящия параграф, както и при какви предположения: Например, за постигане на сложност $\Theta(\log_2 m)$ за проверка за съществуване на ребро при списък на ребрата, предполагаме, че използваме двоично търсене в статичен списък от ребра, сортирани по номер на връх.

5.3. Обхождане на граф

Под *обхождане* на неориентиран (ориентиран) граф ще разбираме последователно *посещаване* (разглеждане) на всеки връх от графа точно по веднъж. *Стратегията за обхождане* определя реда, в който ще се разглеждат върховете.

Съществуват две фундаментални стратегии за обхождане на граф (предмет на следващите две точки), които са известни като *обхождане в ширина* и *обхождане в дълбочина*. Изключителната важност на тези два алгоритъма се обуславя от:

- Широкия периметър от задачи, в които те намират приложение.
- Възможността да се решават сложни алгоритмични проблеми, с прилагане на малки модификации на основните алгоритми.
- Добрата алгоритмична сложност, която се постига в повечето случаи.

5.3.1. Обхождане в ширина

Обхождането в ширина от даден връх i ще наричаме следната стратегия за обхождане на граф: започваме от върха *i*, разглеждаме всички негови непосредствени *съседи*, и едва след това преминаваме към по-нататъшно обхождане (*обхождане в ширина от всеки един от съседите му*). По този начин се постига последователно обхождане "по нива", започвайки от стартовия връх, докато се обходят всички върхове на графа, достижими от *i*. *Обхождане в ширина на граф* наричаме преминаването през всички негови върхове по описания начин — т.е. последователно избиране на (произволен) стартов връх, докато всички върхове на графа не бъдат обходени.

От сега нататък, ще използваме съкращението *BFS*(*i*) (от англ. *Breadth-First-Search*), за да означаваме функцията за обхождане в ширина от връх *i*.

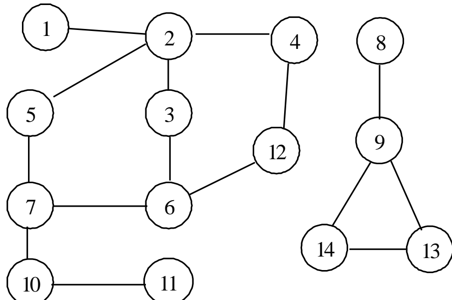
Като пример ще приложим описаната стратегия върху графа от *фигура 5.3.1*.

Ако за стартов връх изберем *1*, резултатът от обхождането ще изглежда по следния начин:
BFS(1):
ниво 1: *1*
ниво 2: *2*

ниво 3: *3*, *4*, *5*
ниво 4: *7*, *6*, *12*
ниво 5: *10*
ниво 6: *11*

Ако вместо *1* изберем *3* за начален връх, при изпълнението на *BFS*(*3*) получаваме:
BFS(3):
ниво 1: *3*
ниво 2: *2*, *6*
ниво 3: *1*, *4*, *5*, *7*, *12*
ниво 4: *10*
ниво 5: *11*

Веднага се забелязват някои възможни приложения на обхождането в ширина. Например, да се намери минималният по брой участващи върхове път между два върха или да се намерят *компонентите на свързаност* на граф. Тези и други примери ще разгледаме малко по-нататък.



Фигура 5.3.1. Неориентиран граф.

Обхождането в ширина намира косвено приложение и в други задачи, при които се търси минимално разстояние. Така например, *методът на вълната* (*виж задача 5.110.*), приложим при търсене на минимален път между две клетки на матрица и др., може да се разглежда и като обхождане на специален вид граф в ширина.

Реализация на обхождане в ширина от даден връх

Ще представим графа с матрица на съседство, като това представяне ще използваме при почти всички задачи от графи. То е достатъчно ефективно при повечето задачи и същевременно не нарушава четимостта на кода с динамично заделяне на памет, обемисти реализации на сложни структури от данни и др.

Разбира се, както при обхождането в ширина, така и при задачите, разглеждани по-нататък, ще се стараем да представяме на читателя винаги най-добрите (от алгоритмична гледна точка) известни до момента решения.

В процеса на обхождане в ширина се налага да намираме наследниците на даден връх i. От таблица *5.2.6.*, се вижда, че ако използваме матрица на съседство, сложността ще бъде $\Theta(n)$. Това е така, тъй като за всеки от n-те върха на графа се извършва проверка дали е наследник на i (за сравнение, при реализация със списък на наследниците можем да получим директно наследниците на i чрез обхождане на съответния списък). Така общата сложност на обхождането е $\Theta(m+n)$ — при представяне със списъци на съседство, и $\Theta(n^2)$ — при представяне с матрица на съседство.

Ще поддържаме опашка, в която първоначално се намира единствено стартовият връх. След това, докато в опашката има поне един връх, извършваме следното: изваждаме върха, намиращ се в началото на опашката, разглеждаме го и добавяме в опашката всички негови

непосетени до момента наследници. Върховете ще маркираме като посетени в момента, в който ги добавяме в опашката:

```
BFS(i)
{ Създаваме_празна_опашка_Queue;
  Добавяме_към_опашката_върха_i;
  for (k = 1,2,…,n) used[k] = 0;
  used[i] = 1;
  while (Опашката_не_е_празна) {
    p = Извличаме_елемент_от_началото_на_опашката;
    Анализираме_върха_p;
    for (за всеки наследник j на p)
      if (0 == used[j]) { /* ако върхът j не е обходен */
        Добавяме_към_опашката_върха_j;
        used[j] = 1; /* маркираме j като обходен */
      }
  }
}
```

Следва пълна реализация. Входните данни, използвани в реализацията по-долу, са зададени като константи в началото на програмата и са за графа, показан на *фигура 5.3.1*. Броят върхове на графа е `n`, а `A[MAXN][MAXN]` е матрицата му на съседство. Стартовият за обхождането връх се задава с константата `v`.

В показания пример графът е неориентиран, но програмата по-долу ще приложи правилно алгоритъма и в случаите на ориентиран граф.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 200
/* Брой върхове в графа */
const unsigned n = 14;
/* Обхождане в ширина с начало връх v */
const unsigned v = 5;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};

char used[MAXN];

/* Функция за обхождане в ширина от даден връх */
void BFS(unsigned i)
{ unsigned k, j, p, queue[MAXN], currentVert, levelVertex, queueEnd;
  for (k = 0; k < n; k++) queue[k] = 0;
  for (k = 0; k < n; k++) used[k] = 0;
  queue[0] = i;  used[i] = 1;
```

```
  currentVert = 0; levelVertex = 1; queueEnd = 1;
  while (currentVert < queueEnd) {  /* докато опашката не е празна */
    for (p = currentVert; p < levelVertex; p++) {
      /* p – вземаме поредния елемент от опашката */
      printf("%u ", queue[p]+1);
      currentVert++;

      /* за всеки необходен наследник j на queue[p] */
      for (j = 0; j < n; j++)
        if (A[queue[p]][j] && !used[j]) {
          queue[queueEnd++] = j;
          used[j] = 1;
        }
    }
    printf("\n");
    levelVertex = queueEnd;
  }
}
int main(void) {
  printf("Обхождане в ширина от връх %u: \n", v);
  BFS(v-1);
  return 0;
}
```

🖫 bfs.c

Резултат от изпълнението на програмата:

```
Обхождане в ширина от връх 5:
5
2 7
1 3 4 6 10
12 11
```

Задачи за упражнение:

1. Горната програма реализира алгоритъм за обхождане на граф в ширина от даден връх. В кои случаи, изпълнена за произволен връх, тя ще обходи *всички* върхове от графа?
2. Как трябва да се модифицира програмата така, че в случаите, когато останат необходени върхове, да се изпълнява ново обхождане в ширина върху тях?

5.3.2. Обхождане в дълбочина

Обхождането в дълбочина (ще съкращаваме с *DFS* — от англ. *Depth-First-Search*) наред с това, че представлява неразделна част от някои по-сложни алгоритми, е основна идея при един от фундаменталните методи за решаване на изчерпващи задачи — *търсене с връщане* (на англ. *backtracking*).

Докато обхождането в ширина разглежда върховете на графа *последователно* по нива, обхождането в дълбочина от даден връх се стреми да "се спусне" колкото се може "по-надълбоко" при обхождането. Алгоритъмът за обхождане на граф в дълбочина се описва най-лесно рекурсивно: започваме от избрания начален връх $i \in V$, маркираме го като посетен и продължаваме рекурсивно обхождането в дълбочина за всеки негов непосетен наследник, т.е. функцията за обхождане в дълбочина от връх i ($DFS(i)$) изглежда по следния начин:

1) Разглеждаме i.
2) Маркираме i като обходен.
3) За всяко инцидентно с i ребро $(i,j) \in E$ такова, че j е необходен връх от графа, изпълняваме рекурсивно $DFS(j)$.

Ще приложим *DFS* върху примера от *фигура 5.3.1*. В случаите, когато повече от едно ребро е инцидентно с разглеждан връх (стъпка 2), ще продължаваме последователно от върховете с по-малък към върховете с по-голям номер. Така резултатът от *DFS*(*1*) ще бъде: *1*, *2*, *3*, *6*, *7*, *5*, *10*, *11*, *12*, *4*, а от *DFS*(*3*): *3*, *2*, *1*, *4*, *12*, *6*, *7*, *5*, *10*, *11*.

Програмата, извършваща обхождането, ще реализираме директно по описания по-горе начин (чрез рекурсивната функция `DFS(i)`). В булев масив `used[]` ще маркираме обходените върхове: в началото го инициализираме с нули, а при посещаване на върха *i* извършваме присвояването `used[i] = 1`.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 200

/* Брой върхове в графа */
const unsigned n = 14;

/* Обхождане в дълбочина с начало връх v */
const unsigned v = 5;

/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};
char used[MAXN];

/* Обхождане в дълбочина от даден връх */
void DFS(unsigned i)
{ unsigned k;
  used[i] = 1;
  printf("%u ", i+1);
  for (k = 0; k < n; k++)
    if (A[i][k] && !used[k]) DFS(k);
}

int main(void) {
  unsigned k;
  for (k = 1; k < n; k++) used[k] = 0;
  printf("Обхождане в дълбочина от връх %u: \n", v);
  DFS(v-1);
  printf("\n");
  return 0;
}
```
🖫 dfs.c

Резултат от изпълнението на програмата:
```
Обхождане в дълбочина от връх 5:
5 2 1 3 6 7 10 11 12 4
```

Таблица 5.3.2. сравнява сложността на обхождане на граф при някои видове представяния (сложностите са еднакви при обхождане в ширина и дълбочина). Оставяме на читателя сам да съобрази как се получават.

	Матрица на съседство	*Списък на наследниците*	*Списък на ребрата*
Обхождане на граф DFS или BFS	$\Theta(n^2)$	$\Theta(n + m)$	$\Theta(n.m)$

Таблица 5.3.2. Сложност на обхождане на граф при различни представяния.

Преди да завършим настоящия параграф, ще разгледаме накратко как с помощта на търсене в дълбочина (или съответно в ширина) можем да намерим произволно покриващо дърво в граф (*виж дефиниция 5.14.*).

Нека $G(V, E)$ е свързан неориентиран граф. Покриващото дърво $T(V, D)$ ще построим по следния алгоритъм:

1) В началото T е граф, в който не участват ребра, т.е. инициализираме $D = \emptyset$.
2) Изпълняваме обхождане в дълбочина в G. При всеки рекурсивен преход по време на обхождане от връх i към негов *необходен* съсед j добавяме реброто (i,j) към D.

Лесно се доказва, че условията от дефиницията на покриващо дърво ще бъдат изпълнени: полученият подграф е свързан (тъй като G е свързан, обхождането в дълбочина/ширина ще "достигне" всички върхове на графа), и не е цикличен (при обхождането всеки връх се разглежда най-много по веднъж).

Задача за упражнение:

Да се обосноват резултатите от *таблица 5.3.2*.

5.4. Оптимални пътища, цикли и потоци в граф

Едни от най-често срещаните задачи, свързани с графи, са тези за търсене на *оптимален път*. Ако разгледаме отново пример 1) от началото на главата и интерпретираме върховете на графа като селища, а ребрата, които ги свързват — като улици (директни пътища), една практическа задача е да се намери път с *минимална дължина* между две селища. Обратно, ако посещаването на всяка улица (или селище) носи определена печалба, то нашата цел може да бъде да намерим *максималния път* (т.е. този, от който ще спечелим най-много) между две точки. Изобщо, условията за оптималност на маршрута, който ще търсим, могат да бъдат дефинирани по различен начин. Така например, е възможно да бъдат наложени допълнителни ограничения, с които трябва да се съобразим (например провизии); да е наложено условието да посетим задължително всяко селище точно по веднъж, или да преминем по всяка улица точно по веднъж и т.н. Независимо от това обаче трябва *точно* да са дефинирани две неща:

- *начин на оценяване* на маршрут (например — като сума от теглата на участващите в него ребра)
- *критерий за оптималност* на маршрута (например желаем да минимизираме горната сума)

Като естествено продължение на търсенето на оптимални пътища в граф ще разгледаме търсенето на (оптимални) *цикли* в граф (да припомним, че цикъл е път, при който началният и крайният връх съвпадат). Съществуват някои забележителни видове цикли, на които ще обърнем специално внимание.

Настоящия параграф ще завършим със задачата за намиране на максимален поток в граф — една основна задача, намираща приложение в голям брой практически проблеми.

5.4.1. Директни приложения на алгоритмите за обхождане

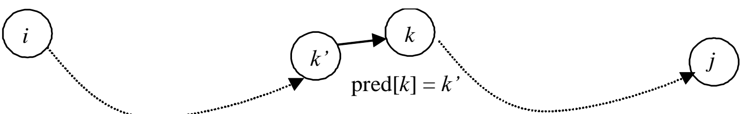
- най-кратък път между два върха по брой на върховете

Нека са дадени граф $G(V, E)$ и два негови върха i и j. Търсим път в G с начало върха i и край върха j, който има минимална дължина по брой на участващите в него върхове. След реализацията на обхождането в ширина лесно можем да съставим алгоритъм за намиране на един такъв минимален път: Изпълняваме $BFS(i)$ и, ако на някоя стъпка достигнем j, то веднага ще следва, че съществува път между двата върха, като наред с това ще разполагаме и с един конкретен минимален път.

Като пример ще разгледаме графа от *фигура 5.3.1*. Нека търсим път от връх *1* до връх *10*. Тогава резултатът от изпълнението на $BFS(1)$ ще бъде:

ниво 1: *1*
ниво 2: *2*
ниво 3: *3*, *4*, *5*
ниво 4: *7*, *6*, *12*
ниво 5: *10*

След като сме достигнали крайния връх j (в случая $j = 10$), очевидно дължината на минималния път е нивото, на което сме обходили j, минус единица. Как обаче да намерим и отпечатаме и междинните върхове, участващи в пътя? Проблемът се решава лесно, ако на всяко ниво разполагаме с предшественика на обхождания връх, т.е. върхът от предходната итерация на обхождането, от който сме добавили текущия връх като непосредствен съсед. За целта ще въведем масив `pred[]`: в него ще запазваме непосредствените предшественици на всеки обходен връх. За началния връх i, `pred[i]` ще инициализираме със стойност –1. Така `pred[k]` ще съдържа върха, от който сме преминали в `k`:



Фигура 5.4.1а. Запазване на път.

Възстановяването на междинните върхове, участващи в минималния път от i до j, се извършва по следната схема:

```
while (j != i) {
  печат(j);
  j = pred[j];
}
печат(j);
```

Горният псевдокод ще отпечата минималния път в *обратен ред* (т.е. от j към i). За да получим пътя правилно, можем да записваме върховете в масив, който после да отпечатаме в обратен ред. Така, неявно използваме стек — първоначално последователно добавяме всички елементи, а след това да изключваме и печатаме. Последното е класически начин за "обръщане на

резултата" и се реализира най-елегантно с рекурсия (виж реализацията по-долу – функцията `printPath`). Сложността на алгоритъма е $\Theta(n+m)$. Следва пълна реализация:

```
#include <stdio.h>
#define MAXN 200 /* Максимален брой върхове в графа */

/* Брой върхове в графа */
const unsigned n = 14;
const unsigned sv =  1; /* Начален връх */
const unsigned ev = 10; /* Краен връх */

/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};

int pred[MAXN];
char used[MAXN];

/* Обхождане в ширина от даден връх със запазване на предшественика */
void BFS(unsigned i)
{ unsigned queue[MAXN];
  unsigned currentVert, levelVertex, queueEnd, k, p, j;
  for (k = 0; k < n; k++) queue[k] = 0;
  queue[0] = i;  used[i] = 1;
  currentVert = 0; levelVertex = 1; queueEnd = 1;
  while (currentVert < queueEnd) {  /* докато опашката не е празна */
    for (p = currentVert; p < levelVertex; p++) {
      /* p - вземаме поредния елемент от опашката */
      currentVert++;

      /* за всеки необходен наследник j на queue[p] */
      for (j = 0; j < n; j++)
        if (A[queue[p]][j] && !used[j]) {
          queue[queueEnd++] = j;
          used[j] = 1;
          pred[j] = queue[p];
        }
    }
    levelVertex = queueEnd;
  }
}

/* Отпечатва върховете от минималния път и връща дължината му */
unsigned printPath(unsigned j)
```

```
{ unsigned count = 1;
  if (pred[j] > -1) count += printPath(pred[j]);
  printf("%u ", j + 1); /* Отпечатва поредния връх от намерения път */
  return count;
}

void solve(unsigned start, unsigned end)
{ unsigned k;
  for (k = 0; k < n; k++) { used[k] = 0; pred[k] = -1; }
  BFS(start);
  if (pred[end] > -1) {
    printf("Намереният път е: \n");
    printf("\nДължината на пътя е %u.\n", printPath(end));
  }
  else
    printf("Не съществува път между двата върха! \n");
}

int main(void) {
  solve(sv-1, ev-1);
  return 0;
}
```

🖫 bfsminw.c

Резултат от изпълнението на програмата:

```
Намереният път е:
1 2 5 7 10
Дължината на пътя е 5.
```

Задача за упражнение:

Да се модифицира функцията за обхождане `BFS()` така, че да прекратява работата си при достигане на крайния връх `ev`.

- проверка дали граф е цикличен

С помощта на алгоритъма за обхождане в дълбочина ще построим алгоритъм за проверка дали неориентиран граф е цикличен. Алгоритъмът за проверка на цикличност се състои в следното: Извършване обхождане в дълбочина. Ако на някоя стъпка от обхождането се окаже, че разглежданият връх i има съсед, който вече сме обходили и различен от предшественика на i, то следва, че графът съдържа цикъл.

Реализацията на алгоритъма ще извършим чрез модификация на функцията `DFS()` (като неин параметър ще добавим още `int parent` — родителя на върха, в който сме в момента). Ако от текущия връх i достигнем до връх, в който вече сме били, различен от `parent`, значи се е затворил цикъл. Посетените върхове ще маркираме както обикновено в масив `used[]`, а в главната функция ще извикваме `DFS(i,-1)` за всеки връх i, който не е бил разгледан до момента. Стойност `-1` за `parent` означава, че този връх е пръв в обхождането на съответната компонента на свързаност и съответно няма родител.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 200
/* Брой върхове в графа */
const unsigned n = 14;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};

char used[MAXN], cycl;

/* модифициран Depth-First-Search */
void DFS(unsigned i, int parent)
{ unsigned k;
  used[i] = 1;
  for (k = 0; k < n; k++) {
    if (cycl) return;
    if (A[i][k]) {
      if (used[k] && k != parent) {
        printf("Графът е цикличен! \n");
        cycl = 1;
        return;
      }
      else if (k != parent)
        DFS(k, i);
    }
  }
}


int main(void) {
  unsigned k, i;
  for (k = 0; k < n; k++) used[k] = 0;
  cycl = 0;
  for (i = 0; i < n; i++) {
    if (!used[i]) DFS(i, -1);
    if (cycl) break;
  }
  if (0 == cycl) printf(" Графът е дърво (не съдържа цикли)!\n");
  return 0;
}
```

🖫 formcycl.c

Резултат от изпълнението на програмата:

```
Графът е цикличен!
```

Сложността на алгоритъма е $\Theta(n+m)$, а при представяне на графа с матрица на съседство, както в предложената реализация, е $\Theta(n^2)$.

Задачи за упражнение:

1. Работи ли горната програма в случай на *ориентиран* граф?

2. Какви модификации можете да направите в случай на ориентиран граф? Опростява ли се или се усложнява кодът?

- намиране на всички прости пътища между два върха

В този параграф ще преразгледаме алгоритъма за обхождане в дълбочина. При *DFS* от *5.3.2.*, когато върхът, който разглеждаме в момента, има повече от един наследник, избирахме да продължим с този, който има най-малък номер. Обхождането ще бъде правилно и ако изберем да продължим с върха с най-голям номер, както и при произволно друго подреждане на върховете. След малко ще видим как това наблюдение ще ни помогне да построим алгоритъм, намиращ *всички възможни пътища* между два върха.

Когато търсим всички пътища от *i* до *j*, се налага да приложим *пълно изчерпване* (което може да се реализира чрез търсене с връщане — *виж глава 6*). Така сложността на алгоритъма, който следва, ще бъде експоненциална. Още веднъж ще отбележим, че търсим всички *прости* пътища, т.е. които не съдържат повтарящи се върхове. В противен случай, ако пътят съдържа повтарящ се връх, това означава, че съдържа цикъл — тогава ще могат да се намерят произволно много пътища, произлизащи от дадения (всеки следващ ще се получава, като се "завъртим" още един или повече пъти през цикъла).

Алгоритъм за намиране на всички прости пътища

Ако изпълним познатото ни обхождане в ширина или дълбочина, започвайки от ***i*****,** ще намерим *един* възможен прост път от *i* до *j*, тъй като задължително ще преминем през всички върхове от същата компонента (в това число и през върха *j*). Естествено, ако двата върха не принадлежат на една и съща компонента на свързаност, то път между тях не съществува и такъв няма да бъде намерен. Модификацията на алгоритъма за обхождане в дълбочина се състои в последователното рекурсивно изпълнение на *DFS*(*k*) за *всеки* връх *k*, съседен на текущо разглеждания връх *i*, а не само за съседа с минимален номер. Така основният цикъл в програмата няма да се промени:

```
for (k = 0; k < n; k++)
   if (A[i][k] && !used[k]) allDFS(k, j);
```

Разликата ще бъде, че в новата реализация ще извършим присвояването `used[i] = 0` след връщане от рекурсията, докато при обикновеното *DFS* това го нямаше. В примера от *фигура 5.3.1.* това означава следното: Нека в момента разглеждаме връх *2* — той има четири съседа: *1*, *4*, *3*, *5*. От връх *1* сме дошли и затова няма да тръгваме по него. Изпълнявайки `for`-цикъла, избираме връх *3* и стартираме *DFS*(*3*). Така рекурсивно ще продължим обхождането по-нататък от връх *3*, но след връщане от рекурсията в края на *DFS*(*3*) ще се изпълни присвояването `used[3] = 0`. По-нататък, когато обхождането продължи със следващите съседи — върховете *4* и *5*, при тези обхождания ще бъде възможно *отново* да се премине през връх *3*. Това не е така при обикновеното *DFS* — при него веднъж маркираме ли връх *3* с `used[3] = 1`, той вече не е достъпен на никоя следваща стъпка от обхождането.

Ще въведем и масив `path[]`, в който ще пазим върховете по реда на тяхното обхождане. Така, ако на някоя стъпка достигнем до крайния връх-цел *j*, можем да отпечатаме текущия път. Следва реализацията на функцията `allDFS(i,j)`. Тя се изпълнява с два параметъра: връх, от който тръгваме `i`, и връх, който трябва да достигнем `j`. Всички входни данни са зададени в началото на програмата като константи.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 200

/* Брой върхове в графа */
const unsigned n = 14;
```

```
const unsigned sv =  1; /* Начален връх */
const unsigned ev = 10; /* Краен връх */

/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};

char used[MAXN];
unsigned path[MAXN], count;

void printPath(void)
{ unsigned k;
  for (k = 0; k <= count; k++)
    printf("%u ", path[k] + 1);
  printf("\n");
}

/* Намира всички прости пътища между върховете i и j */
void allDFS(unsigned i, unsigned j)
{ unsigned k;
  if (i == j) {
    path[count] = j;
    printPath();
    return;
  }

  /* маркиране на посетения връх */
  used[i] = 1;
  path[count++] = i;
  for (k = 0; k < n; k++)  /* рекурсия за всички съседи на i */
    if (A[i][k] && !used[k]) allDFS(k, j);
  /* връщане: размаркиране на посетения връх */
  used[i] = 0; count--;
}
int main(void) {
  unsigned k;
  for (k = 0; k < n; k++) used[k] = 0;
  count = 0;
  printf("Прости пътища между %u и %u: \n", sv, ev);
  allDFS(sv-1, ev-1);
  return 0;
}
```

🖫 btdfs.c

Резултат от изпълнението на програмата:

```
Прости пътища между 1 и 10:
1 2 3 6 7 10
1 2 4 12 6 7 10
1 2 5 7 10
```

Задача за упражнение:

Да се реализира алгоритъм за намиране на всички прости пътища чрез обхождане в ширина (BFS). Опростява ли се или се усложнява кодът? Променя ли се сложността на алгоритъма?

5.4.2. Екстремален път в граф

Дефиниция 5.15. Нека е даден претеглен ориентиран граф $G(V, E)$ с тегла на ребрата дадени реални числа. *Дължина на път* в G се нарича *сумата* от теглата на участващите в него ребра.

В настоящия параграф ще съсредоточим вниманието си върху алгоритми за намиране на максимални и минимални по дължина пътища в граф. В *5.4.1.* разгледахме алгоритъм за намиране на *всички* пътища между два върха. Подобно решение с пълно изчерпване е единственият начин за решаване на някои задачи от намиране на оптимален път, но специално в случая, когато търсим максимален или минимален път като сума/произведение от теглата на съставящите го ребра, съществуват много по-ефективни алгоритми.

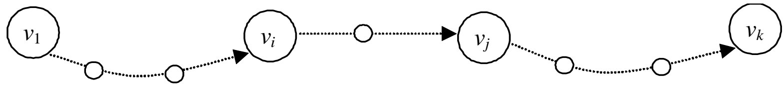
При някои задачи е възможно дължината на пътя да бъде дефинирана не като *сума*, а като някаква друга *функция* на теглата на участващите в пътя ребра (и дори върхове). За да останат валидни посочените алгоритми и в тези случаи, трябва да са налице някои специфични критерии за оптималност (те зависят от разглеждания конкретен алгоритъм).

Преди да преминем към същинската част, ще обърнем внимание на някои неизяснени подробности. Тъй като е възможно да няма ограничение пътят да бъде прост, то трябва да внимаваме в случаите, когато графът съдържа цикъл. Така например, ако търсим минимален път, а в него има *отрицателен цикъл* (цикъл с отрицателна дължина), ще можем да се “завъртим” по този цикъл произволен брой пъти, при което дължината на пътя (равна на сумата от ребрата в него) ще намалява произволно много към минус безкрайност. Аналогично, ако търсим максимален път и е налице *положителен цикъл*, то за всеки път, който го съдържа, ще можем да се “завъртим” по цикъла, при което да получим произволно голяма дължина.

Алгоритмите, с които ще започнем, ще бъдат за намиране на *минимален път* в граф. Ако търсим *максимален* път, бихме могли лесно да ги модифицираме. Едно възможно решение е да "обърнем" теглата на ребрата от графа: ще променяме теглото $f(i,j)$ на всяко ребро (i,j) на $-f(i,j)$. Ако след това изпълним алгоритъм за търсене на минимален път в новополучения граф, то намереният път ще бъде максимален за оригиналния. Този подход, обаче, не винаги е приложим и не е универсален за намиране на различни типове максимални пътища в различни типове графи.

Повечето алгоритми за намиране на екстремален път се основават на следния признак за оптималност:

Теорема. (*признак за оптималност*) Нека е даден граф, двойка негови върхове (v_1,v_k) и минимален път $p = (v_1, v_2, ..., v_k)$ между v_1 и v_k. Тогава, за $1 \leq i < j \leq k$ пътят $p' = (v_i, v_{i+1}, ..., v_j)$ е минималният път от върха v_i до v_j.



Фигура 5.4.2б. Оптимизиране на път.

Валидността на последната теорема може да се покаже лесно с допускане на противното (Оставяме го на читателя като упражнение).

За намиране на минималните пътища най-често се въвежда масив, стойностите на който означават горни граници на търсените минимални пътища. На всяка стъпка, някоя от горните граници се намалява (как става това, зависи от конкретния алгоритъм), докато се достигне търсеният минимум.

Разгледаната обща схема ще демонстрираме с някои от най-известните алгоритми за намиране на минимални пътища в граф.

Задача за упражнение:

Да се докаже признакът за оптималност.

- неравенство на триъгълника

Алгоритмите на Форд-Белман, Флойд и Дейкстра се основават на *неравенството на триъгълника*: сумата от дължините на кои да е две страни е по-голяма от дължината на третата му страна. Да означим с $d(i,j)$ разстоянието между върховете i и j. Тогава, в случай на ориентиран граф неравенствата на триъгълника за произволна тройка върхове i, j, k могат да се запишат така:

$$d(i,k) + d(k,j) \geq d(i,j)$$

$$d(i,j) + d(j,k) \geq d(i,k)$$

$$d(k,i) + d(i,j) \geq d(k,j)$$

$$d(j,k) + d(k,i) \geq d(j,i)$$

$$d(k,j) + d(j,i) \geq d(k,i)$$

$$d(j,i) + d(i,k) \geq d(j,k)$$

В случай на неориентиран граф имаме $d(x,y) = d(y,x)$ за всяка двойка върхове x,y и последните три неравенства стават излишни. Всеки от изброените алгоритми се основава на последователна проверка на (някои от) неравенствата на триъгълника и, в случай на нарушение, установяване в равенство.

Задача за упражнение:

Да се обоснове нуждата от спазване на неравенството на триъгълника като следствие от признака за оптималност.

- алгоритъм на Форд-Белман

Нека е даден ориентиран граф $G(V, E)$ и търсим най-кратките разстояния от даден връх s до всички останали върхове. Ще приемем, че работим с матрицата на теглата `A[i][j]` на графа G. В случаите, когато не съществува реброто (i, j), като стойност на `A[i][j]` ще бъде записано $+\infty$ (и тъй като работим с крайни структури от данни, това ще бъде максималната допустима стойност на типа, от който са елементите на `A`).

Алгоритъм на Форд-Белман:

1) Въвеждаме масив `D[]`, като след завършване на алгоритъма, `D[i]` ще съдържа дължината на минималния път от s до всеки друг връх i от графа. Инициализираме `D[i] = A[s][i]`, за всеки връх $i \in V$.

2) Опитваме да *оптимизираме* стойността на `D[i]`, за всяко $i \in V$ по-следния начин:

   За всяко $j \in V$, ако `D[i] > D[j] + A[j][i]`, присвояваме `D[i] = D[j] + A[j][i]`.

3) След повтаряне на стъпка 2) n–2 пъти, в масива `D[]` ще се съдържат търсените минимални пътища.

```
for (k = 1; k <= n - 2; k++)  /* повтаряме (n-2) пъти */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (D[i] > D[j] + A[j][i])
        D[i] = D[j] + A[j][i];
```
🖫 belman.c

Очевидно сложността на алгоритъма е $\Theta(n^3)$, тъй като стъпка 2) има сложност $\Theta(n^2)$ и се изпълнява n–2 пъти. Единственото, за което трябва да се внимава в случая от гледна точка на реализацията, е като сумираме `D[j] + A[j][i]` да не получим препълване (когато някое от `D[j]` или `A[j][i]` е инициализирано с максималната за типа стойност).

Алгоритъмът на Форд-Белман е валиден за графи с произволни тегла на ребрата: както положителни, така и отрицателни. Разбира се, при наличието на отрицателен цикъл, не можем да говорим за минимален път (виж по-горе). Полезно свойство на алгоритъма на Форд-Белман е, че позволява *откриването на отрицателни цикли*: Ако след приключването му, за някоя двойка от върхове (i, j) е изпълнено `D[i] > D[j] + A[j][i]`, то графът съдържа отрицателен цикъл (Защо?).

Задачи за упражнение:

1. Да се обоснове алгоритъмът на Форд-Белман.
2. Защо най-външният цикъл на горния алгоритъм се повтаря n-2 пъти, а не например n или n–1?
3. Да се докаже, че ако след приключването на алгоритъма на Форд-Белман за някоя двойка от върхове (i, j) е изпълнено `D[i] > D[j] + A[j][i]`, то графът съдържа отрицателен цикъл.
4. Вярно ли е и обратното твърдение, а именно: Ако графът съдържа отрицателен цикъл, то след приключване на алгоритъма на Форд-Белман ще съществува двойка върхове (i, j), за които да бъде изпълнено `D[i] > D[j] + A[j][i]`?

- алгоритъм на Флойд

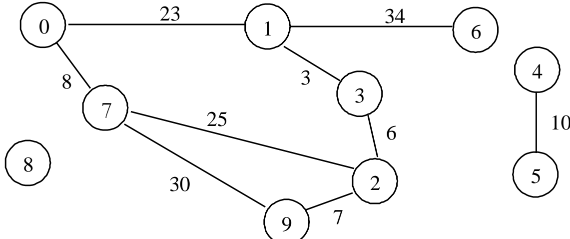
Алгоритъмът на Флойд е подобен на този на Форд-Белман. Съществената разлика е, че след като той приключи работа, разполагаме с дължините на минималните пътища между *всяка двойка* върхове от графа и то без да е необходима допълнителна памет (използва се матрицата на теглата му). Това означава, че ако в началото в `A[i][j]` е записано теглото на реброто (i, j), то след изпълнение на алгоритъма на Флойд върху матрицата `A[][]`, стойността на `A[i][j]` ще бъде дължината на минималния път между i и j.

Алгоритъмът се състои в следното: за всеки два върха $i, j \in V$, на `A[i][j]` се присвоява по-малкото от `A[i][j]` и `A[i][k]+A[k][j]`, за всеки връх $k \in V$. Смисълът на върха k, като *междинен* връх за оптимизиране на пътя, ще стане по-ясен в изложението на следващия параграф (*обобщена схема на Флойд*).

Сложността на алгоритъма на Флойд е $\Theta(n^3)$. Както и при Форд-Белман, наличието на отрицателни ребра не представлява проблем, стига графът да не съдържа отрицателни цикли.

Реализацията на горната проверка ще извършим с три вложени цикъла — по k, i и j. За удобство, при сравняването на основната стъпка, вместо 0 ще използваме стойност `MAX_VALUE`, с която ще означаваме липсата на ребро в матрицата на теглата (на практика е възможно нулата да бъде валидна стойност на тегло). За да работи програмата коректно, `MAX_VALUE` трябва да има стойност, по-голяма от $n.d_{max}$, където d_{max} е максималното тегло на ребро от графа (това изискване може да се отслаби: достатъчно е да се вземе сумата от положителните ребра). Трябва да се внимава и за препълване при сумирането: Тъй като имаме две събираеми, `MAX_VALUE` не

трябва да надвишава `MAXINT/2`, където `MAXINT` е максималната допустима стойност за типа `int` в съответния компилатор на Си (в повечето среди константата `MAXINT` е дефинирана в заглавния файл `<values.h>`).



Фигура 5.4.2в. Неориентиран претеглен граф.

Следва изходният код на програмата. Входните данни съответстват на неориентирания граф, показан на *фигура 5.4.2в.* (Разбира се, алгоритъмът на Флойд, както и програмата по-долу, ще работят коректно и при ориентирани графи). При задаването на входните данни като константи сме използвали стойност 0 за означаване на липсата на ребро в матрицата на теглата (с цел по-голяма нагледност в изходния код). По-късно (в началото на основната функция `floyd()`) на всички нулеви полета се присвоява стойност `MAX_VALUE`.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 150
#define MAX_VALUE 10000

/* Брой върхове в графа */
const unsigned n = 10;
/* Матрица на теглата на графа */
const unsigned A[MAXN][MAXN] = {
  { 0, 23,  0,  0,  0,  0,  0,  8,  0,  0 },
  {23,  0,  0,  3,  0,  0, 34,  0,  0,  0 },
  { 0,  0,  0,  6,  0,  0,  0, 25,  0,  7 },
  { 0,  3,  6,  0,  0,  0,  0,  0,  0,  0 },
  { 0,  0,  0,  0,  0, 10,  0,  0,  0,  0 },
  { 0,  0,  0,  0, 10,  0,  0,  0,  0,  0 },
  { 0, 34,  0,  0,  0,  0,  0,  0,  0,  0 },
  { 8,  0, 25,  0,  0,  0,  0,  0,  0, 30 },
  { 0,  0,  0,  0,  0,  0,  0,  0,  0,  0 },
  { 0,  0,  7,  0,  0,  0,  0, 30,  0,  0 }
};

/* Намира дължината на минималния път между всяка двойка върхове */
void floyd(void)
{ unsigned i, j, k;
  /* стойностите 0 се променят на MAX_VALUE */
  for (i = 0; i < n; i++) for (j = 0; j < n; j++)
    if (0 == A[i][j]) A[i][j] = MAX_VALUE;
  /* Алгоритъм на Флойд */
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        if (A[i][j] > (A[i][k] + A[k][j]))
          A[i][j] = A[i][k] + A[k][j];
  for (i = 0; i < n; i++) A[i][i] = 0;
```

```
}

void printMinPaths(void)
{ unsigned i, j;
  printf("Матрица на теглата след изпълнение на алгоритъма на Флойд:\n");
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
      printf("%3d ", (MAX_VALUE == A[i][j]) ? 0 : A[i][j]);
    printf("\n");
  }

}

int main(void) {
  floyd();
  printMinPaths();
  return 0;
}
```

🖫 floyd.c

Резултат от изпълнението на програмата:

```
Матрица на теглата след изпълнение на алгоритъма на Флойд:
  0  23  32  26   0   0  57   8   0  38
 23   0   9   3   0   0  34  31   0  16
 32   9   0   6   0   0  43  25   0   7
 26   3   6   0   0   0  37  31   0  13
  0   0   0   0   0  10   0   0   0   0
  0   0   0   0  10   0   0   0   0   0
 57  34  43  37   0   0   0  65   0  50
  8  31  25  31   0   0  65   0   0  30
  0   0   0   0   0   0   0   0   0   0
 38  16   7  13   0   0  50  30   0   0
```

Ще отбележим, че редът на подреждане на циклите е съществен: цикълът по k трябва задължително да бъде най-външният. Идеята е, че първо намираме най-късите пътища с междинен връх 1, след това тези, с междинни върхове 1 и 2, след това тези с междинни върхове 1, 2 и 3 и т.н. Ясно е, че взаимното положение на циклите по i и j не е съществено, т.е. и двете наредби k, i, j и k, j, i са верни. Но всяка друга наредба ни води до некоректен алгоритъм. Да разгледаме като пример матрицата (забележете, че тя не е симетрична, т.е. съответства на ориентиран граф):

```
const n = 4;
int A[n][n] =
{//   A   B   C   D
    { 0,  0,  6,  2},   // A
    { 0,  0,  0,  0},   // B
    { 0,  1,  0,  0},   // C
    { 0,  0,  2,  0}    // D
};
```

При правилна подредба, получаваме следната матрица на минималните пътища:

```
0 5 4 2
0 0 0 0
0 1 0 0
0 3 2 0
```

При наредба на циклите i, k, j обаче имаме:

```
0 7 4 2
0 0 0 0
0 1 0 0
0 3 2 0
```

В първия случай най-краткият път от A до B е (A, D, C, B) с дължина 5, а във втория – получаваме грешното (A, C, B) с дължина 7. Оставяме на читателя да съобрази защо се получава тази разлика.

Забележка: Възстановяването на конкретните минимални пътища, а не само намиране на тяхната дължина ще бъде разгледано по-долу, при разглеждане на алгоритъма на Дейкстра. Читателят би могъл директно да "пренесе" и приложи тази схема в алгоритъма на Флойд.

Задачи за упражнение:

1. По дадена двойка върхове (i,j) да се възстанови *един* конкретен минимален път.
2. По дадена двойка върхове (i,j) да се възстановят *всички* минимални пътища.

- обобщен алгоритъм на Флойд

Алгоритъмът на Флойд за намиране на минималните пътища между всички двойки върхове може да се обобщи до следната схема за намиране на *оптимални* пътища:

$$\varphi^{(k)}(i,j) = F\,\{\varphi^{(k-1)}(i,j),\ \varphi^{(k-1)}(i,k) \oplus \varphi^{(k-1)}(k,j)\ \},\ \text{където}\ \varphi^{(1)}(i,j) = f(i,j).$$

$\varphi^{(k)}$ е функция, представяща оптималния път между два върха, съдържащ неповече от k върха, която подлежи на оптимизация, а $\oplus$ е двуаргументна операция, изчисляваща общата стойност на пътя, получен от слепването (конкатенацията) на два непресичащи се пътя. Подобно рекурсивно представяне създава впечатлението, че работим с n числови матрици: като матрицата със стойности $\varphi^{(1)}(i,j)$ е матрицата на теглата на графа, а $\varphi^{(k)}(i,j)$ — матрицата с търсените оптимални пътища. Горната схема се реализира итеративно с квадратична памет, както при алгоритъма на Флойд от предходния параграф. Функцията F се използва за оптимиране на стойността, получена чрез операцията $\oplus$.

Сложността на обобщения алгоритъм на Флойд, ако приемем, че сложностите за извършване на операцията $\oplus$ и пресмятане на функцията F са константи, е същата както при стандартния: $\Theta(n^3)$.

Нека разгледаме как можем да приложим горната схема за намиране на пътищата с минимална дължина между всички двойки върхове. В този случай операцията $\oplus$ ще бъде сбор от дължините на два пътя. Тъй като търсим път с минимална дължина, F ще бъде функцията *минимум*.

Ще използваме единствена матрица `A[][]` за запазване на резултатите от k-тите стойности на функцията $\varphi^{(k)}(i,j)$. В началото инициализираме `A[][]` с дадените тегла на ребрата на графа (т.е. `A[][]` е матрицата на теглата). Когато F е *максимум* и реброто (i,j) не съществува, в матрицата `A[i][j]` се записва 0. Стойността на `A[i][i]` за примките ще бъде *+безкрайност* (`MAX_VALUE`). Обратно, когато F е *минимум* и реброто (i,j) не съществува, матрицата съдържа `MAX_VALUE`, а стойността на `A[i][i]` е 0.

Вместо `A[i][i] == MAX_VALUE` или `A[i][i] == 0` е по-удачно да се извършва следната проверка:

```
if ((i != j)&&(i != k)&&(k != j)) {...}
```

По този начин се избягва използването и инициализирането на диагоналните елементи `A[i][i]`.

Ще илюстрираме казаното дотук с няколко примера.

пример 1. Надеждност на път

Даден е претеглен граф, представящ компютърна мрежа, в който теглото на всяко ребро $f(i,j)$ показва *надеждността* на връзката между компютри i и j (число между 0 и 1). Тогава *надеждността на път* в графа се определя, като произведение от теглата на ребрата, които съдържа. Задачата е да се намери най-надеждният път за предаване на информацията между два дадени компютъра. Тук схемата на Флойд се прилага по следния начин:

$$\varphi^{(k)}(i, j) = max(\varphi^{(k-1)}(i, j),\ \varphi^{(k-1)}(i, k).\ \varphi^{(k-1)}(k, j)\ )$$

Матрицата `A[][]`, с която се реализира схемата, съдържа 0, ако между два върха няма ребро. Тъй като цената (надеждността) на всеки път се получава, като се умножат теглата на ребрата му, то операцията $\oplus$ е *умножение*. Тъй като се търси път с най-голяма надеждност, то F ще бъде *максимум*.

пример 2. Път с най-голяма пропускливост

В даден претеглен граф, представящ водоснабдителна система, теглото на всяко ребро $f(i,j)$ показва неговата *пропускливост*. Задачата е да се намери път от s до t с максимална *пропускливост*. *Пропускливостта на път* се определя като минималното тегло на ребро, свързващо два последователни върха от пътя. Обобщената схема на Флойд за тази задача ще бъде:

$$\varphi^{(k)}(i,j) = max(\varphi^{(k-1)}(i,j),\ min(\varphi^{(k-1)}(i,k),\ \varphi^{(k-1)}(k,j)\ ),$$

Матрицата `A[][]` съдържа 0, ако между два върха няма ребро. Тъй като цената (пропускливостта) на всеки път се определя от реброто с най-ниска пропускливост, то операцията $\oplus$ е *минимум*. Тъй като търсим пътя с най-голяма пропускливост, F ще бъде *максимум*.

пример 3. Намиране на първите t минимални пътя

Намирането на първите t минимални пътя между всеки два върха в граф е възможно да се извърши, като отново се използва обобщената схема на Флойд. Приложението ѝ в този случай не е толкова директно, но схемата се запазва: функцията $\varphi^{(k)}(i,j)$ се дефинира не като *число*, а като *вектор от числа*, имащи смисъла на дължините на минималните пътища. Аналогично, F и $\oplus$ се дефинират върху вектори: *минимум* на два вектора $(x_1, x_2, ..., x_t)$ и $(y_1, y_2, ..., y_t)$ е вектор, съставен от най-малките t числа измежду числата в двата вектора, а *сума* на два вектора: като вектор, съставен от минималните t измежду числата $x_1+ y_1, x_1+ y_2, ..., x_1+ y_t, x_2+y_1, x_2+y_2,..., x_2+y_t, ..., + x_t+y_1, x_t+y_2,..., x_t+y_t$.

Могат да се приведат още редица примери (със съответно практическо приложение), където обобщената схема на Флойд се прилага успешно: Оптимален път по сума от степените на участващите върхове, максимално нарастващ път (максимална сума на разликите между теглата на две поредни ребра от пътя) и др.

В случай, че търсим минималните разстояния от фиксиран връх s, до всички останали върхове, то съществува и по-добър алгоритъм от този на Флойд.

Задачи за упражнение:

1. Да се реализира програма за пресмятане на надеждността на път (виж пример 1).
2. Да се реализира програма за пресмятане на пропускливостта на път (виж пример 2).
3. Да се реализира програма за намиране на първите p минимални пътя (виж пример 3).

- алгоритъм на Дейкстра

Най-ефективния метод за намиране на минималните пътища от един конкретен връх до всички останали е алгоритъмът на *Дейкстра*.

Нека е даден претеглен ориентиран граф $G(V, E)$ с n върха. За да бъде приложен алгоритъмът, теглата на ребрата $f(i,j)$ трябва да бъдат *положителни* числа (виж забележката в края на параграфа). Търси се минималният път от фиксиран връх $s \in V$ до всички останали върхове на графа. С $\varphi(s,i)$ ще означим дължината на минималния път от s до i. Най-общо, алгоритъмът на Дейкстра се основава на следния принцип: За да намерим $\varphi(s,i)$, трябва да намерим минималното измежду $\varphi(s,j) + f(j,i)$, за всяко j, $j \neq i$. Така, на всеки връх i от графа се присвоява временна стойност `d[i]`, която представлява горна граница за $\varphi(s, i)$. В процеса на работа на алгоритъма тази стойност намалява, докато накрая `d[i]` стане точно равна на $\varphi(s, i)$:

Алгоритъм на Дейкстра

1) Инициализираме масив `d[]` по следния начин:
   `d[i] = A[s][i]` за всеки съсед $i \in V$ на s.
   `d[i] = MAX_VALUE`, за всеки връх i, несъседен на s.
   След приключване на алгоритъма `d[i] == MAX_VALUE` т.с.т.к. между s и i няма път.
2) Въвеждаме множество T, което в началото съдържа всички върхове на графа, без s:
   $T = V \setminus \{s\}$
3) Докато T съдържа поне един връх i, за който `d[i] < MAX_VALUE`:
   3.1) Избираме връх $j \in T$ такъв, че `d[j]` да бъде минимално.
   3.2) Изключваме j от T : $T = T \setminus \{j\}$
   3.3) За всяко $i \in T$ изпълняваме `d[i] = min(d[i], d[j] + A[j][i]);`

За да възстановим върховете, участващи в минимален път (а не само дължината на пътя), ще въведем допълнителен масив `pred[]`. Така в i-тата позиция на масива `pred[]` се записва върхът j, за който $\varphi(s, j) + f(j, i)$ е било минимално:

```
if (d[i] > d[j] + A[j][i]) {
  d[i] = d[j] + A[j][i];
  pred[i] = j;
}
```

По този масив, след приключване на алгоритъма, пътят може да се възстанови по-следния начин:

```
void printPath(unsigned s, unsigned j) {
  if (pred[j] != s)
    printPath(s, pred[j]);
  printf("%d ", j); /* отпечатването става след връщане от рекурсията */
}
```

Използваната техника за възстановяване и отпечатване на построения път е вече известна от *5.4.1.* — неявно се използва стек, в който се включва всеки пореден връх, а отпечатването се извършва *след* разглеждането на последния връх при обратния ход на рекурсията.

Примерните входни данни, използвани в реализацията по-долу, са записани като константи в началото на програмата и съответстват на графа от *фигура 5.4.2в*.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 150
```

```
#define MAX_VALUE 10000
#define NO_PARENT (unsigned)(-1)

/* Брой върхове в графа */
const unsigned n = 10;
const unsigned s = 1;
/* Матрица на теглата на графа */
const unsigned A[MAXN][MAXN] = {
  { 0, 23,  0,  0,  0,  0,  0,  8,  0,  0 },
  {23,  0,  0,  3,  0,  0, 34,  0,  0,  0 },
  { 0,  0,  0,  6,  0,  0,  0, 25,  0,  7 },
  { 0,  3,  6,  0,  0,  0,  0,  0,  0,  0 },
  { 0,  0,  0,  0,  0, 10,  0,  0,  0,  0 },
  { 0,  0,  0,  0, 10,  0,  0,  0,  0,  0 },
  { 0, 34,  0,  0,  0,  0,  0,  0,  0,  0 },
  { 8,  0, 25,  0,  0,  0,  0,  0,  0, 30 },
  { 0,  0,  0,  0,  0,  0,  0,  0,  0,  0 },
  { 0,  0,  7,  0,  0,  0,  0, 30,  0,  0 }
};

char T[MAXN];
unsigned d[MAXN];
int pred[MAXN];

/* Алгоритъм на Дейкстра - минимален път от s до останалите върхове */
void dijkstra(unsigned s)
{ unsigned i;
  for (i = 0; i < n; i++) /* инициализиране: d[i]=A[s][i], i∈V, i != s */
    if (0 == A[s][i]) {
      d[i] = MAX_VALUE;
      pred[i] = NO_PARENT;
    }
    else {
      d[i] = A[s][i];
      pred[i] = s;
    }
  for (i = 0; i < n; i++) T[i] = 1; /* T съдържа всички върхове */
  T[s] = 0;
  pred[s] = NO_PARENT;              /* от графа, с изключение на s */

  while (1) {           /* докато T съдържа i: d[i] < MAX_VALUE */
    /* избиране на върха j от T, за който d[j] е минимално */
    unsigned j = NO_PARENT;
    unsigned di = MAX_VALUE;

    for (i = 0; i < n; i++)
      if (T[i] && d[i] < di) {
        di = d[i];
        j = i;
      }
    if (NO_PARENT == j) break; /* d[i] = MAX_VALUE, за всички i: изход */
    T[j] = 0;           /* изключваме j от T */

    /* за всяко i от T изпълняваме D[i] = min (d[i], d[j]+A[j][i]) */
    for (i = 0; i < n; i++)
      if (T[i] && A[j][i] != 0)
        if (d[i] > d[j] + A[j][i]) {
          d[i] = d[j] + A[j][i];
          pred[i] = j;
```

```
      }
  }
}

void printPath(unsigned s, unsigned j)
{ if (pred[j] != s) printPath(s, pred[j]);
  printf("%u ", j+1);
}

/* Отпечатва намерените минимални пътища */
void printResult(unsigned s)
{ unsigned i;
  for (i = 0; i < n; i++) {
    if (i != s) {
      if (d[i] == MAX_VALUE)
        printf("Няма път между върховете %u и %u\n", s+1, i+1);
      else {
        printf("Минимален път от връх %u до %u:  %u ", s+1, i+1, s+1);
        printPath(s, i);
        printf(", дължина на пътя: %u\n", d[i]);
      }
    }
  }
}

int main(void) {
  dijkstra(s-1);
  printResult(s-1);
}
```

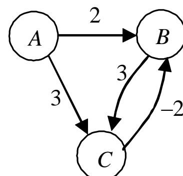
🖫 dijkstra.c

Резултат от изпълнението на програмата:

```
Минимален път от връх 1 до 2:  1 2 , дължина на пътя: 23
Минимален път от връх 1 до 3:  1 2 4 3 , дължина на пътя: 32
Минимален път от връх 1 до 4:  1 2 4 , дължина на пътя: 26
Няма път между върховете 1 и 5
Няма път между върховете 1 и 6
Минимален път от връх 1 до 7:  1 2 7 , дължина на пътя: 57
Минимален път от връх 1 до 8:  1 8 , дължина на пътя: 8
Няма път между върховете 1 и 9
Минимален път от връх 1 до 10:  1 8 10 , дължина на пътя: 38
```

Сложността на алгоритъма при горната реализация е $\Theta(n^2)$. При по-внимателно подбиране на структурите от данни (например, ако използваме пирамида — виж 3.1.9., [*Cormen, Leiserson, Rivest-1997*]) сложността може да се понижи до $\Theta(n.\log_2 n)$.

Забележка: Алгоритъмът на Дейкстра няма да работи правилно, ако в графа има ребра с отрицателни тегла. За да илюстрираме това, ще разгледаме един конкретен пример (*виж фигура 5.4.2г.*).



Фигура 5.4.2г. Претеглен граф с отрицателно ребро.

За да се намерят минималните пътища от A до B и C, ще се изпълни следното:

1) Инициализира се $d[A] = 0$, $d[B] = f(A,B) = 2$, $d[C] = f(A,C)=3$.

2) Инициализира се множество $T = \{A,B,C\}\backslash\{A\} = \{B,C\}$.

3) Избира се връх $i \in T$, за който $d[i]$ е минимално, това е върхът B:

Изключва се B от T. За всеки връх от T (а в T се намира единствено върхът C) се изпълнява: $d[C] = min(d[C], d[B] + f(B,C)) = min(3, 2+3) = 3$.

На следващата стъпка в T се намира само C, и след изключване на C множеството T остава празно и алгоритъмът приключва.

Така получихме $d[A] = 0$, $d[B] = 2$, $d[C] = 3$. Очевидно обаче, най-краткия път от A до B не е с дължина 2, а с дължина 1 (това е пътят A-C-B, защото $3+(-2) = 1$).

Задачи за упражнение:

1. Да се сравни алгоритъмът на Дейкстра с тези на Форд-Белман и Флойд.
2. Възможно ли е да се възстанови минималният път между дадена двойка върхове само от масива `d[]`, без да се използва допълнителен масив като `pred[]`?

- повдигане в степен на матрицата на съседство

Съществува интересна връзка между операцията повдигане на матрица в степен k и броя на пътищата с дължина k между два върха в неориентиран граф. Нека е даден неориентиран граф $G(V, E)$ с матрица на съседство `A[][]`. Нека `A'` е матрица, такава че `A'[i][j]` съдържа броя на пътищата с дължина k между върховете i и j. Тогава в матрицата `A" = A'.A` стойността на `A"[i][j]` ще бъде точно броят на пътищата с дължина k+1 между двата върха. Верността на това твърдение следва от правилото за умножение на матрици:

$$A''[i][j] = \sum_{t=1}^{n} A'[i][t].A[t][j]$$

Тъй като `A'[i][t]` е броят на пътищата с дължина k между i и t, то броят на пътищата с дължина k+1 между i и j ще се увеличи с `A'[i][t]` при условие, че реброто (t, j) е от графа (което е еквивалентно на `A[t][j] == 1`). Така достигнахме до следната

Теорема. Нека е даден неориентиран граф $G(V, E)$ с матрица на съседство `A`. Ако повдигнем `A` в степен k, то `C[i][j]` $=$ A^k`[i][j]` ще бъде броят на пътищата с дължина k, между върховете i и j. Ако `C[i][j] = 0`, следва, че не съществува път между i и j с дължина k.

Теоремата е валидна и в случай на ориентиран граф (и дори мултиграф), но само ако в матрицата на съседство не участват стойностите –1, с които се обозначава "обратно" ребро.

Полученият резултат е полезен не само при решаване на задачи за намиране на брой на пътища, но и техни "производни": цикли, различни видове компоненти и др. (алгоритъмът, описан в теоремата, се реализира тривиално и сглобяването в програма оставяме на читателя като леко упражнение). Нещо повече, поради връзката "граф – матрица на съседство" много класически задачи на алгебрата, отнасящи се до матрици, се прехвърлят като задачи върху графи, в това число все още нерешени.

Задача за упражнение:

Да се реализира програма на базата на горната теорема.

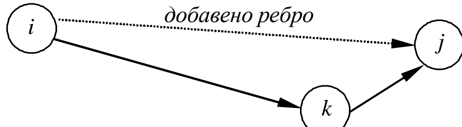
- алгоритъм на Уоршал и матрица на достижимост

Нека е даден ориентиран граф $G(V, E)$ с матрица на съседство `A[][]`. Търсим матрица `A'[][]`, такава че:

- `A'[i][j] == 1`, тогава и само тогава, когато съществува път (с каквато и да е дължина) между върховете i и j.
- `A'[i][j] == 0`, когато път между двата върха *не* съществува.

Алгоритъмът за решаването на тази задача носи името на Уоршал и резултатът от прилагането му — матрицата `A'[][]`, се нарича *матрица на достижимост* на графа. Графът $G'(V, E')$, съответен на матрицата `A'[][]` (ако разглеждаме `A'[][]` като матрица на съседство) се нарича *транзитивно затворен* граф на G (за него ще стане дума отново в *5.5.1.*).

Алгоритъмът на Уоршал е със сложност $\Theta(n^3)$ и процедира по следния начин: за всеки три върха k, i, $j \in V$, ако $(i, k) \in E$ и $(k, j) \in E$, то добавяме в множеството от ребра на графа и реброто (i, j) (*фигура 5.4.2д.*).



Фигура 5.4.2д. Стъпка от транзитивно затваряне.

Разглеждането на тройки върхове се извършва чрез три вложени цикъла по следния начин:

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++) {
    if (A[i][k])
      for (j = 0; j < n; j++)
    if (A[k][j]) A[i][j] = 1;
  }
```

След изпълнението на показания програмен фрагмент, матрицата `A[][]` ще бъде успешно модифицирана до търсената матрица на достижимост `A'[][]`. (*Защо?*) Внимателният читател вероятно е забелязал сходството между горния алгоритъма и този на Флойд. Това не е случайно: алгоритъмът на Уоршал е частен случай на този на Флойд.

Задачи за упражнение:

1. Да се реализира алгоритъмът на Уоршал.
2. Да се докаже, че алгоритъмът на Уоршал правилно намира матрицата на достижимост.

- най-дълъг път в ацикличен граф

Много практически задачи се решават, като се сведат до задача за намиране на най-дълъг път в *ацикличен граф*. Условието за ацикличност е съществено, тъй като единственият подход за решаване на задачата в общия случай е чрез пълно изчерпване. От друга страна, когато в графа няма цикли, са изпълнени някои принципи за оптималност (същите са необходими при всички задачи, които се решават с динамично оптимиране — *глава 8*) и задачата може да бъде решена значително по-ефективно.

Ще разгледаме една практическа постановка на задачата за намиране на максимален път.

Задача: Екип от програмисти разработва програмен продукт, който се състои от отделни задачи. Всяка задача има определена продължителност и свързва два етапа от разработването на продукта: начален и краен. Една задача не може да бъде започната, ако не е завършен началният ѝ етап. За да бъде завършен изцяло един етап, трябва да бъдат завършени всички задачи, за които той се явява краен. Да се определи минималното време (при условие, че разполагаме с неограничен брой програмисти), достатъчно за завършването на целия проект. Проектът се счита за завършен, когато са завършени всичките му етапи.

Ако за модел на задачата използваме претеглен ориентиран граф, в който върховете са етапите, а ребрата — задачите, то търсеното минимално време ще бъде равно на дължината на

максималния *по брой на върховете* път в графа. В литературата този път се среща още и под названието *критичен път*.

Алгоритъмът по-долу се основава на техниката динамично оптимиране (*виж глава 8*).

Алгоритъм:

Нека е даден ацикличен ориентиран граф $G(V, E)$ и `A[N][N]` е матрица на теглата на G.

1) Въвеждаме масив `maxDist[]`, в който ще пазим максималните разстояния от всеки връх в графа, т.е. `maxDist[i]` ще бъде равно на дължината на максималния път с начало върха i. В началото инициализираме всички елементи на `maxDist[]` с нули. Във втори масив `savePath[]` ще пазим намерения най-дълъг път, като масива инициализираме с –1.
2) Нека разглеждаме връх $i \in V$, от който не излизат ребра. Такъв връх задължително съществува, тъй като графът е ацикличен (нека читателят съобрази откъде следва това!). Премахваме този връх от графа и на всеки негов предшественик k присвояваме:

`maxDist[k] = max { maxDist[k], maxDist[i] + A[k][i] }`

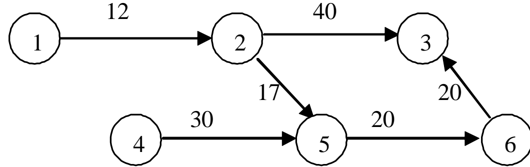
3) Изпълняваме стъпка 2), докато не остане нито един връх в графа. Тогава максималната стойност на `maxDist[i]`, за i=1,2,...n ще бъде дължината на търсения максимален път.

Алгоритъмът има сложност $\Theta(n+m)$, а при представяне на графа с матрица на съседство — $\Theta(n^2)$. Ще илюстрираме как се прилага върху един конкретен граф. *Фигура 5.4.2е.* показва реда, в който се отстраняват върховете от графа и как се променят стойностите в `maxDist[]`.

Реализация на алгоритъма:

Изложения по-горе алгоритъм ще реализираме рекурсивно. Ще използваме модификация на обхождане в дълбочина. При всяко *връщане* от рекурсивното извикване на $DFS(i)$, се озоваваме в някой *предшественик* j на върха i. Там ще запазваме максималното `maxDist[j]+A[j][i]`, за всеки наследник j на i и в края, преди изхода от функцията DFS, ще го присвояваме на `maxDist[i]`. Функцията $DFS(i)$ се изпълнява само, ако `maxDist[i]` още не е пресметнато, т.е. ако `maxDist[i] == 0` (като в началото инициализираме целия масив с нули). Функцията DFS се стартира последователно за всеки от върховете на графа.

За да запазим и отпечатаме всички върхове от търсения път, а не само дължината му, в масива `savePath[]` на позиция i ще записваме този наследник j, за който е намерено максималното `maxDist[i]+A[i][j]`.




```
1. Инициализираме maxDist[i]=0, за всяко i.
2. Изтриваме връх 3 =>
     maxDist[6] = max{ maxDist[6], maxDist[3]+20} = max{0, 20} =20;
     Аналогично, maxDist[2] = 40
3. Изтриваме връх 6 =>  maxDist[5] = 40;
4. Изтриваме връх 5 =>
     maxDist[4] = 70,  maxDist[2] = max{ 40, 40+17} = 57
5. Изтриваме връх 4
6. Изтриваме връх 2 =>  maxDist[1] = 69
7. Изтриваме връх 1

Най-дълъг път : 70 (maxDist[4])
```

Фигура 5.4.2е. Намиране на критичен път в граф.

Следва изходният код на програмата. Броят върхове и матрицата на теглата `A[][]` на графа се задават като константи.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 150

/* Брой върхове в графа */
const unsigned n = 6;

/* Матрица на теглата на графа */
const unsigned A[MAXN][MAXN] = {
  { 0, 12,  0, 0,  0,  0 },
  { 0,  0, 40, 0, 17,  0 },
  { 0,  0,  0, 0,  0,  0 },
  { 0,  0,  0, 0, 30,  0 },
  { 0,  0,  0, 0,  0, 20 },
  { 0,  0, 20, 0,  0,  0 }
};

int savePath[MAXN], maxDist[MAXN];

void DFS(unsigned i)
{ int max, d;
  unsigned j;
  if (maxDist[i] > 0) return;
  max = maxDist[i];
  for (j = 0; j < n; j++)
    if (A[i][j]) {
      DFS(j);
      d = maxDist[j] + A[i][j];
      if (d > max) {
        max = d;
        savePath[i] = j;
      }
    }

  maxDist[i] = max;
}

void solve(void)
{ unsigned i, maxi;

  /* инициализация */
  for (i = 0; i < n; i++) {
    maxDist[i] = 0;
    savePath[i] = -1;
  }

  for (i = 0; i < n; i++)
    if (maxDist[i] == 0) DFS(i);

  maxi = 0;
  for (i = 0; i < n; i++)
    if (maxDist[i] > maxDist[maxi])
      maxi = i;
```

```
    printf("Дължината на критичния път е %d\nПътят е: ", maxDist[maxi]);
    while (savePath[maxi] >= 0) {
      printf("%u ", maxi + 1);
      maxi = savePath[maxi];
    }

    printf("%d\n", maxi + 1);
}

int main(void) {
    solve();
    return 0;
}
```

🖫 longpath.c

Резултат от изпълнението на програмата:

```
Дължината на критичния път е 70
Пътят е: 4 5 6 3
```

Задачи за упражнение:

1. Да се докаже, че в ацикличен граф задължително съществуват:
   - поне един връх без предшественик
   - поне един връх без наследник

2. Да се реализира вариант на алгоритъма, при който на всяка стъпка се премахва връх без *предшественици*, вместо такъв без *наследници*.

- най-дълъг прост път между два върха в произволен граф

Още веднъж ще припомним, че търсенето на най-дълъг път в цикличен граф е *NP-пълна задача*, която ще разгледаме в *6.3.3*. За решаването ѝ се прилага *пълно изчерпване*.

Задачи за упражнение:

1. Да се даде пример за *ориентиран* граф и двойка негови върхове *i* и *j*, в който предходният алгоритъм *не* намира правилно най-дългия път между *i* и *j*.
2. Има ли случаи, когато алгоритъмът за намиране на най-дълъг път в *ацикличен* граф ще работи правилно за *някоя* двойка върхове в *цикличен* граф? Ако да — кога? Ако не — защо? Да се приведат съответни примери.

5.4.3. Цикли

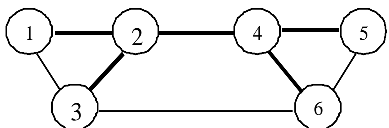
Логично продължение на темата от *5.4.2.* за пътищата е намирането и изследването на циклите на граф.

- намиране на фундаментално множество от цикли

Нека е даден неориентиран свързан граф $G(V, E)$ с n върха и m ребра. Нека $D(V, T)$ е произволно покриващо дърво на G. Добавянето към D на ребро, непринадлежащо на T, ще доведе до затваряне на *прост* цикъл. Ще казваме, че този цикъл принадлежи на *фундаменталното множество цикли* на графа G относно покриващото му дърво D.

Всяко покриващо дърво D има точно $q = n-1$ ребра. Ребрата на G, непринадлежащи на D, са общо $m-n+1$. Тъй като добавянето на всяко едно от тях към T води до получаването на точно един цикъл, то броят на циклите от фундаменталното множество е $m-n+1$.

Определяйки фундаменталното множество от цикли, ние определяме еднозначно цикличната структура на графа, тъй като всеки друг цикъл в G може да се представи чрез "слепване" на цикли, от някое фундаментално множество. Например, за показания на *фигура 5.4.3.* граф и избраното покриващо дърво (участващите в него ребра са показани с по-тъмен цвят) прости цикли са $A = (1, 2, 3)$, $B = (4, 5, 6)$ и $C = (2, 3, 4, 6)$. Всеки друг *прост* цикъл може да се построи чрез съединяване (слепване) на няколко цикъла A, B и C, които имат *поне едно* общо ребро (само общ връх/върхове не е достатъчно!). Под *съединяване на два цикъла* разбираме премахване на общите им ребра (и съответно изолираните върхове след премахването на ребрата) — така получаваме отново прости цикли. Например, съединявайки A и C, получаваме цикъла $(1, 2, 4, 6, 3)$.



Фигура 5.4.3. Фундаментално множество от цикли в граф.

Намирането на фундаментално множество от цикли ще извършим, като отново използваме модификация на функцията за обхождане в дълбочина *DFS*. На първата стъпка ще намерим едно произволно покриващо дърво на графа, а по-нататък всяко ребро, което не участва в построеното покриващо дърво, ще затваря цикъл и този цикъл ще намерим с още едно обхождане на графа, реализирано по-долу от функцията `DFS2()`. Общата сложност на алгоритъма е $\Theta(m.(m+n))$.

В изходния код, даден по-долу, графът е представен с матрица на съседство `A[][]`, като `A[i][j]` показва дали има ребро между i и j. Впоследствие, при изпълнението на програмата, `A[i][j] == 2`, ако реброто участва в построеното покриващо дърво.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 150
/* Брой върхове в графа */
const unsigned n = 10;

/* Графът, представен с матрица на съседство: 0 - няма ребро; 1 – има;
 * По-късно с 2 ще маркираме ребрата на покриващото дърво на графа.
 */
char A[MAXN][MAXN] = {
  { 0, 1, 1, 0, 0, 0 },
  { 1, 0, 1, 1, 0, 0 },
  { 1, 1, 0, 0, 0, 1 },
  { 0, 1, 0, 0, 1, 1 },
  { 0, 0, 0, 1, 0, 1 },
  { 0, 0, 1, 1, 1, 0 }
};

char used[MAXN];
unsigned cycle[MAXN], d;

/* Намира произволно покриващо дърво */
void DFS(unsigned v)
{ unsigned i;
  used[v] = 1;
  for (i = 0; i < n; i++)
    if (!used[i] && A[v][i]) {
```

```
      A[v][i] = 2;
      A[i][v] = 2;
      DFS(i);
    }
}

/* Отпечатва намерен цикъл */
void printCycle(void)
{ unsigned k;
  for (k = 0; k < d; k++)
    printf("%u ", cycle[k] + 1);
  printf("\n");
}

/* Намиране на един цикъл спрямо намереното покриващо дърво */
void DFS2(unsigned v, unsigned u)
{ unsigned i;
  if (v == u) { printCycle(); return; }
  used[v] = 1;
  for (i = 0; i < n; i++)
    if (!used[i] && 2==A[v][i]) {
      cycle[d++] = i;
      DFS2(i, u);
      d--;
    }
}

int main(void) {
  unsigned i, j, k;
  DFS(0);
  printf("Простите цикли в графа са: \n");
  for (i = 0; i < n - 1; i++)
    for (j = i + 1; j < n; j++)
      if (1 == A[i][j]) {
        for (k = 0; k < n; k++) used[k] = 0;
        d = 1;
        cycle[0] = i;
        DFS2(i, j);
      }
  return 0;
}
```

🖫 allcyc.c

Резултат от изпълнението на програмата:

```
Простите цикли в графа са:
1 2 3
2 3 6 4
5 4 6
```

Задачи за упражнение:

1. Работи ли описаният алгоритъм (с/без модификация) за неориентиран мултиграф?

2. Работи ли описаният алгоритъм (с/без модификация) за ориентиран граф? А за ориентиран мултиграф?

3. Разгледаният по-горе алгоритъм е със сложност $\Theta(m.(m+n))$, а реализацията, поради използването на матрица на съседство за представяне на графа — със сложност $\Theta(n^4)$. Възможно ли е да се подобри този резултат? Показахме, че броят на простите цикли е от порядъка на

$\Theta(m+n)$, а дължината на всеки цикъл зависи от височината h на построеното покриващо дърво, т.е. минималната сложност за намирането и отпечатването на циклите е $\Theta((m+n).h)$. Ясно е, че в най-лошия случай височината е равна на броя на върховете на графа, т.е. получава се сложност $\Theta((m+n).n)$.

Можете ли да съставите алгоритъм за намиране на циклите, който да работи с посочената минимална сложност $\Theta((m+n).h)$? Каква стратегия за построяване на покриващо дърво можете да предложите, така че то да бъде с минимална височина? Може ли да се твърди, че при добре избрана стратегия h ще принадлежи на $O(\log_2 n)$?

- минимален цикъл през връх

Задача: Даден е претеглен ориентиран граф $G(V, E)$ и връх $i \in V$. Да се намери цикъл (не непременно прост), съдържащ върха i, и за който сумата от теглата на участващите ребра да бъде минимална.

Решение: За всяко $j \in V$ намираме минималния път $\varphi(i,j)$ от i до j и минималния път $\varphi(j, i)$ от j до i. Нека $S_k = \varphi(i, k) + \varphi(k, i)$, $k \in V$. Тогава минималното S_k е дължината на минималния цикъл, който търсим.

Съществено е, че не е задължително намереният цикъл да бъде прост. Задачата, при която е наложено допълнително ограничение в цикъла да няма повтарящи се върхове, също е доста интересна. Ако разглеждаме граф без тегла на ребрата, задачата се решава с обхождане в ширина: лека модификация на алгоритъма за намиране на минимален по брой на върховете път между два върха. Решението в общия случай (при претеглен граф) е разгледано в *задача 5.5*.

Друга алгоритмично интересна задача от цикли е задачата за намиране на минимален прост цикъл, съдържащ поне k върха (*виж задача 5.93.*).

Неин "частен" случай, при $k = n$, е разгледан в следващия параграф.

Задачи за упражнение:

1. Да се реализира описаният алгоритъм и да се пресметне сложността му.
2. Даден е претеглен ориентиран граф и връх от него. Да се намери *прост* цикъл, съдържащ върха, за който сумата от теглата на участващите ребра да бъде минимална.

5.4.4. Хамилтонови цикли. Задача за търговския пътник

Дефиниция 5.16. *Хамилтонов цикъл* (*ХЦ*) в граф се нарича цикъл, съдържащ всеки връх от графа точно веднъж. Граф, съдържащ такъв цикъл, се нарича *Хамилтонов*.

Твърдение 1. В Хамилтонов граф съответният Хамилтонов цикъл може да се построи, като се започне от произволен начален връх.

Две класически задачи са проверката дали даден граф е Хамилтонов и *задачата за търговския пътник*, която се състои в това да се намери Хамилтонов цикъл с минимална дължина в претеглен граф. И двете задачи се определят като *NP-пълни* (*виж 6.2.*), което означава, че сложността в най-лошия случай е експоненциална. В настоящия параграф ще разгледаме и решим втората задача.

Алгоритъм за решаване на задачата за търговския пътник

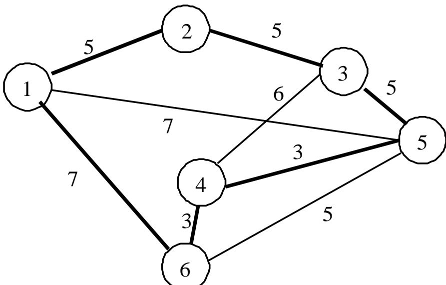
Ще решим задачата чрез пълно изчерпване. В *5.4.1.* вече разгледахме как да намерим всички *прости* пътища между два върха. Ще приложим същия подход, като ще искаме началния и крайния връх да съвпадат. Избираме произволен връх i и, тръгвайки от него, ще строим последователно всички възможни маршрути. Дължината `curSum` на маршрута, построен до момента, ще променяме в тялото на рекурсията. Когато намерим нов Хамилтонов цикъл, ще проверяваме

дали дължината му е по-малка от най-малката, намерена до момента (нея ще пазим в променлива `minSum`), и ако е по-малка — ще я запазваме. Ако теглата на всички ребра на графа са положителни, можем да направим нашия алгоритъм малко по-ефективен, прилагайки “рязане на рекурсията” (*виж 6.4.*). Техниката се състои в следното: ако на някоя стъпка дължината `curSum` на пътя, който строим в момента, стане по-голяма от `minSum`, ще прекъсваме строенето на текущия маршрут. Това се прави, тъй като дори и той да се разшири по-късно до Хамилтонов цикъл, неговата дължина задължително ще бъде по-голяма от най-малката. За да се възползваме от тази оптимизация, обаче, графът не трябва да съдържа отрицателни ребра, които да намалят `curSum` впоследствие! За да се убедим в това, нека разледаме следната примерна матрица на съседство:

```
int A[n][n] =
{   // A  B  C  D
    {  0, 1, 2, 0}, // A
    { -2, 0, 1, 0}, // B
    {  0, 0, 0, 1}, // C
    {  1, 2, 0, 0}  // D
};
```

Ако на някаква стъпка сме намерили като текущ минимален Хамилтонов цикъл (A,B,C,D,A) с дължина 1+1+1+1 = 4, то при достигане до върха B в Хамилтоновия цикъл (A,C,D,B,A) текущата сума ще бъде 2+1+2 = 5. И тъй като 5 > 4, а 4 е минималната до момента дължина, престава разглеждането на новия кандидат, който е минимален, защото при добавяне на реброто (B,A) с тегло –2, дължината на цикъла става 3, а 3 < 4.

В предложената реализация графът се представя с матрица на теглата `A[][]`, а входните данни са записани като константа в началото на програмата. Графът, използван като пример за входните данни в реализацията, е показан на *фигура 5.4.4*.



Фигура 5.4.4. Минимален Хамилтонов цикъл.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 150
#define MAX_VALUE 10000
/* Брой върхове в графа */
const unsigned n = 6;

/* Матрица на теглата на графа */
const int A[MAXN][MAXN] = {
  { 0, 5, 0, 0, 7, 7 },
  { 5, 0, 5, 0, 0, 0 },
  { 0, 5, 0, 6, 5, 0 },
  { 0, 0, 6, 0, 3, 3 },
  { 7, 0, 5, 3, 0, 5 },
  { 7, 0, 0, 3, 5, 0 }
};
```

```
char used[MAXN];
unsigned minCycle[MAXN], cycle[MAXN];
int curSum, minSum;

void printCycle(void)
{ unsigned i;
  printf("Минимален Хамилтонов цикъл: 1");
  for (i = 0; i < n - 1; i++) printf(" %u", minCycle[i] + 1);
  printf(" 1, дължина %d\n", minSum);
}

/* Намира минимален Хамилтонов цикъл */
void hamilton(unsigned i, unsigned level)
{ unsigned k;
  if ((0 == i) && (level > 0)) {
    if (level == n) {
      minSum = curSum;
      for (k = 0; k < n; k++) minCycle[k] = cycle[k];
    }
    return;
  }
  if (used[i]) return;
  used[i] = 1;
  for (k = 0; k < n; k++)
    if (A[i][k] && k != i) {
      cycle[level] = k;
      curSum += A[i][k];
      if (curSum < minSum) /* прекъсване на генерирането */
        hamilton(k, level + 1);
      curSum -= A[i][k];
    }
  used[i] = 0;
}
int main(void) {
  unsigned k;
  for (k = 0; k < n; k++) used[k] = 0;
  minSum = MAX_VALUE;
  curSum = 0;
  cycle[0] = 1;
  hamilton(0, 0);
  printCycle();
  return 0;
}
```

🖫 tsp.c

Резултат от изпълнението на програмата:

`Минимален Хамилтонов цикъл: 1 2 3 5 4 6 1, дължина 28.`

Изложеният алгоритъм и реализация са практически неприложими за големи графи: при сегашната мощ на изчислителната техника за графи с над 50 върха е възможно да се намерят примери, за които алгоритъмът ще работи неприемливо дълго време. Поради широкото си приложение, задачата е изследвана изключително подробно в литературата [*Reinelt-1994*]. За решението ѝ съществуват десетки (!) алгоритми, някои от които за *почти* всички графи (всички, с изключение на *логаритмично зависещ от n брой* графи) намират решението с полиномиална сложност.

Преди да преминем към следващия интересен вид цикли в граф, ще разгледаме още един въпрос, свързан с Хамилтоновите цикли и търсенето им. Става въпрос за това, как *вероятността да съществува* Хамилтонов цикъл в произволен граф се променя, в сравнение със

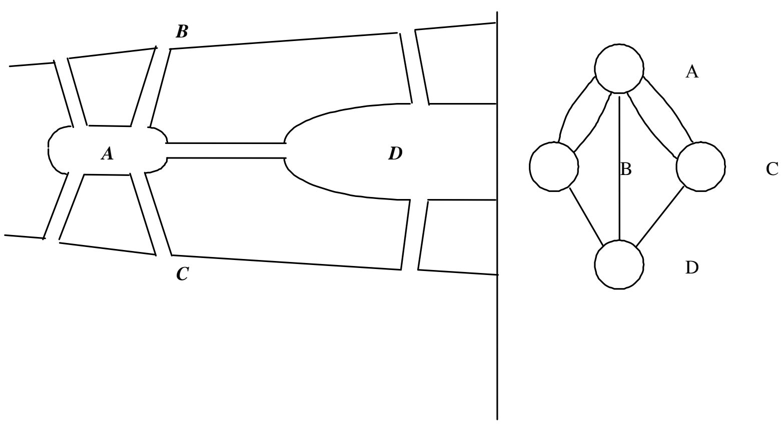
свързаността на графа. Ясно е, че един *пълен* граф (*виж дефиниция 5.8.*) е винаги Хамилтонов (каквато и подредба на върховете да вземем, те винаги ще образуват *ХЦ*). Когато графът е *почти пълен* (по-точно *почти напълно свързан*), вероятността за съществуване на *ХЦ* е много висока, както е голям и броят на Хамилтоновите цикли. Този брой рязко намалява с намаляване на броя на ребрата. При другата крайност, когато свързаността на графа е близо до 2, вероятността за съществуване на Хамилтонов цикъл е малка. В една *критична точка* между тези два екстремума вероятността за съществуване на Хамилтонов цикъл се разпределя равномерно между 0 и 1. Теорията показва, че това равномерно разпределяне на вероятността е изпълнено в случая, когато имаме *средна свързаност* на графа от порядъка на $\ln n + \ln \ln n$ [*Christofides-1975*].

Задача за упражнение:

Да се определят някои класове "специални" графи, за които намирането на Хамилтонов цикъл е "лесно" (става с полиномиална сложност). Примери за такива са разгледани в *9.1.8*.

5.4.5. Ойлерови цикли

За начало на теорията на графите се смята разглеждането на един специален вид цикли, наречени на името на техния пръв изследовател *Л.Ойлер*. Градът *Кьонигсберг* (днес *Калининград*) имал седем моста на река Прегел, построени, както е показано на *фигура 5.4.5а*.



Фигура 5.4.5а,б. Кьонигсбергските мостове и представянето им с неориентиран мултиграф.

Любознателният читател би могъл да намери картата на Калининград на следния адрес в Интернет: http://guide.kaliningrad.net/map.php3?language=eng.

Хората там често си задавали въпроса дали е възможна разходка през града, така че да се премине точно веднъж по всеки от мостовете и обиколката да приключи в изходната позиция. Ойлер представил мостовете с мултиграф (*фигура 5.4.5б.*) и показал, че това е невъзможно.

Дефиниция 5.17. Нека е даден свързан мултиграф. Цикъл, в който всяко ребро участва точно по веднъж, се нарича *Ойлеров цикъл*. Един мултиграф се нарича *Ойлеров*, ако в него съществува *Ойлеров* цикъл.

Аналогично се дефинира и *Ойлеров път* в граф.

Теорема. (*Ойлер*) (смятана за първата теорема в теорията на графите, изказана от Ойлер без доказателство). Свързан неориентиран мултиграф съдържа Ойлеров цикъл, тогава и само тогава, когато всички върхове на графа са от четна степен.

Следствие 1. Свързан неориентиран мултиграф съдържа Ойлеров *път* тогава и само тогава, когато има *точно два* върха от нечетна степен.

Теорема 2. Слабо свързан ориентиран мултиграф съдържа Ойлеров цикъл, тогава и само тогава, когато полустепента на входа $d^{+}(i)$ на всеки връх i е равна на полустепента на изхода $d^{-}(i)$, т.е.: $d^{+}(i) = d^{-}(i)$, за всяко $i \in V$.

Следствие 2. Слабо свързан ориентиран мултиграф съдържа Ойлеров път тогава и само тогава, когато има точно два върха i и j, такива че $d^{+}(i) = d^{-}(i)+1$, $d^{+}(j) = d^{-}(j)-1$ и $d^{+}(k) = d^{-}(k)$, за всяко $k \in V$, $k \neq i$, $k \neq j$.

Твърдение 1. В Ойлеров мултиграф Ойлеров цикъл може да се построи, като се започне от произволен начален връх.

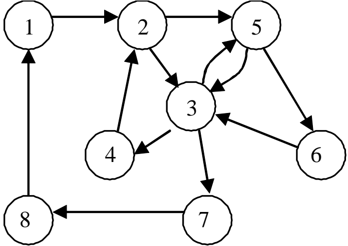
Следствие 3. Ако в слабо свързан ориентиран мултиграф има Ойлеров път, то тогава начален е този връх, чиято полустепен на изхода е по-голяма от полустепента на входа.

Пример за ориентиран Ойлеров граф е показан на *фигура 5.4.5в*. Един възможен Ойлеров цикъл в него е (*1*,*2*), (*2*,*3*), (*3*,*4*), (*4*,*2*), (*2*,*5*), (*5*,*3*), (*3*,*5*), (*5*,*6*), (*6*,*3*), (*3*,*7*), (*7*,*8*), (*8*,*1*).

Алгоритъм за намиране на Ойлеров цикъл

Започваме обхождане на графа от произволен връх i. Намираме инцидентно с него ребро (i,j) и го маркираме като посетено. Продължаваме с върха j: за него намираме непосетено ребро (j,k), маркираме го като посетено и преминаваме в k. Продължавайки по този начин, в даден момент ще се озовем в началния връх i, т.е. ще затворим цикъл (*Защо?*). Ако всички ребра на графа са вече маркирани, то този цикъл е Ойлеров. Ако са останали непосетени ребра, то намираме връх x, принадлежащ на току-що намерения цикъл, и инцидентен с поне едно *непосетено* ребро. От това, че всеки връх трябва да бъде от четна степен, следва, че x е инцидентен с четен брой (т.е. поне две) непосетени ребра. От x започваме да строим цикъл по вече описания начин, докато отново се върнем в него. Получаваме втори цикъл, който обединяваме с първия (общата им точка ще бъде върхът x). Така, след краен брой стъпки, всички ребра ще се включат в един общ цикъл — търсеният Ойлеров цикъл.

В случай, че търсим *Ойлеров път*, можем да приложим следната проста модификация на горния алгоритъм: съединяваме върховете от нечетна степен с ребро (от *следствие 1* следва, че съществуват точно два такива върха) и намираме Ойлеров цикъл по описания по-горе алгоритъм. След отстраняване на добавеното ребро получаваме търсения Ойлеров път.



Фигура 5.4.5в. Ойлеров граф.

Нека графът е представен с матрица на съседство `A[][]`. За реализацията ще използваме два стека: `stack[]`, за текущо построявания цикъл, и `cStack[]` — за обединението на всички

построени до момента цикли. И двата стека първоначално ще бъдат празни. Следният псевдокод описва по-подробно реализацията на алгоритъма:

```
Добавяме_произволен_връх_i_в_stack;
while (stack_не_е_празен) {
  Вземаме_върха_i_от_върха_на_stack_без_да_го_изключваме;
    if  (i_има_наследници) {
      Вземаме_произволен_наследник_j_на_i;
      Слагаме_j_в_stack;
      A[j][i] = 0; A[i][j] = 0;  /* изключваме реброто от графа */
    } else {
      Изключваме_i_от_stack;
      Включваме_i_в_cStack;
    }
}
```

Следва пълна реализация. Примерният граф е този от *фигура 5.4.5в.*

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 100
/* Брой върхове в графа */
const unsigned n = 8;
/* Матрица на съседство на графа */
char A[MAXN][MAXN] = {
  { 0, 1, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 1, 0, 1, 0, 0, 0 },
  { 0, 0, 0, 1, 1, 0, 1, 0 },
  { 0, 1, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 1, 0, 0, 1, 0, 0 },
  { 0, 0, 1, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 1 },
  { 1, 0, 0, 0, 0, 0, 0, 0 }
};

/* Проверява дали има Ойлеров цикъл (по теоремата на Ойлер) */
char isEulerGraph(void)
{ unsigned i, j;
  for (i = 0; i < n; i++) {
    int din = 0, dout = 0;
    for (j = 0; j < n; j++) {
      if (A[i][j]) din++;
      if (A[j][i]) dout++;
    }
    if (din != dout) return 0;
  }
  return 1;
}

/* Намира Ойлеров цикъл */
void findEuler(int i)
{ unsigned cStack[MAXN * MAXN], stack[MAXN * MAXN];
  unsigned k, j, cTop = 0, sTop = 1;
  stack[sTop] = i;
  while (sTop > 0) {
    i = stack[sTop];
    for (j = 0; j < n; j++)
```

```
      if (A[i][j]) {
        A[i][j] = 0; i = j;
        break;
      }
    if (j < n)
      stack[++sTop] = i;
    else
      cStack[++cTop] = stack[sTop--];
  }
  printf("Ойлеровият цикъл е: ");
  for (k = cTop; k > 0; k--) {
    printf("%u ", cStack[k] + 1);
  }
  printf("\n");
}

int main(void) {
  if (isEulerGraph()) findEuler(0);
  else
    printf("Графът не е Ойлеров!");
  return 0;
}
```

euler.c

Резултат от изпълнението на програмата:

```
Ойлеровият цикъл е: 1 2 3 4 2 5 3 5 6 3 7 8 1
```

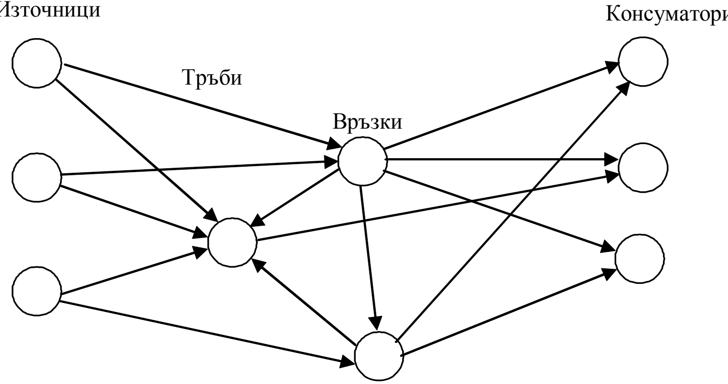
Задачи за упражнение:

1. Да се докаже теоремата на Ойлер и нейните следствия.
2. Да се сравни задачата за намиране на Хамилтонов цикъл с тази за намиране на Ойлеров. Каква е причината първата да бъде значително по-трудна от втората?

5.4.6. Потоци

Нека е даден ориентиран граф $G(V, E)$, в който върховете са разделени в три множества:

- множество от *източници* на някакъв материал;
- множество от *консуматори*;
- множество от междинни върхове — *връзки*, през които материалът може да бъде разпределян до други връзки или консуматори.



Фигура 5.4.6а. "Транспортен" граф.

На всеки източник е съпоставено число — максимално количество материал, което може да доставя, а за всеки консуматор е дадено максималното количество материал, което може да приема. Ориентираните ребра (i,j) от графа могат да бъдат интерпретирани като "тръби", свързващи върховете i и j, като за всяко ориентирано ребро дефинираме:

1) Функция c: $c(i,j)$ ограничава количеството материал, което може да преминава през реброто (i,j), $c(i,j) \geq 0$, за всички $(i, j) \in E$.
2) Функция t: $t(i,j)$ показва разхода (цената) за пренасяне на материала през реброто (i,j).

Общата задача за намиране на поток с минимална цена (от англ. *minimum cost network flow problem*) се състои в това да се намери схема за доставяне на материала от източниците до консуматорите така, че общият разход за пренасянето да бъде минимален. През 1960 г. Форд и Фулкерсон предлагат алгоритъм (наречен *out of kilter*) за решаването на тази задача. Задачата има множество различни интерпретации и полезни частни случаи, например:

Задача за оптимално назначение

Дадени са n машини, n детайли и квадратна матрица $A_{n\times n}$: a_{ij} показва разхода за изработване на детайл i от машина j. Да се намери схема за изработване на детайлите (пермутация p на елементите от 1 до n) такава, че общият разход да бъде минимален (т.е. да бъде минимална сумата $\sum_{i\in V} a_{i,p(i)}$).

Тази задача може да се сведе до задача за минимален поток по следния начин:

- Разглеждаме машините като източници, които доставят единица материал.
- Разглеждаме детайлите като консуматори, които изискват единица материал.
- Всяко ребро има капацитет единица и цена, равна на съответната стойност от таблицата, дадена в условието.
- Няма междинни върхове (връзки).

Възможен е и противоположният вариант на задачата: квадратната таблица показва приходите от изработката на детайлите (в зависимост от това, коя машина ще го изработи) и търсим максимална печалба.

Освен чрез свеждане до оптимален поток, последната задача може да се реши и с т. нар. *унгарски алгоритъм* [*Шишков-1995*].

- *Максимален поток*

Разновидност на горната задача е намирането на *максимален поток*. Задачи от този тип намират приложение в комуникационните мрежи, при планиране на мрежи за пренасяне на материал (газ, вода, горива) и др. Алгоритъмът, който я решава, е известен от 1950 г. и принадлежи отново на Форд и Фулкерсон.

Разглеждаме претеглен граф $G(V, E)$, в който на всяко ребро (i,j) е съпоставено неотрицателно число $c(i, j)$, имащо значение на капацитет (пропускателна способност). В графа са фиксирани два специални върха: *източник s* и *консуматор t*.

Дефиниция 5.18. *Поток* в граф се нарича функция $f:E\rightarrow Z$, съпоставяща на всяко ребро (i, j) число $f(i, j)$ и притежаваща следните три свойства:

1) Потокът не надвишава капацитета на "тръбата", т.е. за всяко ребро $(i, j)\in E$ е изпълнено $f(i, j) \leq c(i, j)$.
2) Ако потокът е отрицателен, това означава, че е налице "обратен поток". Изобщо, ако потокът от i до j е равен на $f(i, j)$, то задължително следва, че потока от j до i е $-f(i,j)$, за всяко $(i, j)\in E$.

3) Големината на входящия и изходящия поток са равни, т.е. за всеки връх $i \in V \setminus \{s, t\}$ е изпълнено $\sum_{j:(i,j)\in E} f(i,j) = 0$.

В задачата се търси стойността на максималния поток от s до t, т.е. такава функция f (изпълняваща горните три условия), че $\sum_{i:(i,t)\in E} f(i,t)$ да бъде максимална.

Алгоритъм на Форд-Фулкерсон

Основната идея на алгоритъма е да се увеличава непрекъснато потокът, докато това е възможно:

1) Започваме с нулев поток: $f(i, j) = 0$, за всяко $(i, j) \in E$.
2) Намираме *увеличаващ път* в графа. *Увеличаващ път* ($s = v_0, v_1, ..., v_k = t$) е такъв път от s до t, че за всеки два съседни върха v_i, v_{i+1} ($i = 0,1,..., k-1$) от пътя да бъде изпълнено $c(v_i, v_{i+1}) > 0$.
3) Ако не съществува такъв път, следва, че сме намерили максималния поток в графа.
4) В противен случай увеличаваме положителните и отрицателните потоци с максималното положително число p, което позволява намерения път, т.е.:

$$p = \min_{i=0,...k-1} \{c(v_i, v_{i+1})\}$$
$$f(v_i,v_{i+1}) = f(v_i,v_{i+1}) + p,\ i = 0,1,...,k-1$$
$$f(v_{i+1},v_i) = f(v_{i+1},v_i) - p,\ i = 0,1,...,k-1$$

Модифицираме теглата (пропускателната способност) на ребрата на графа по-следния начин:

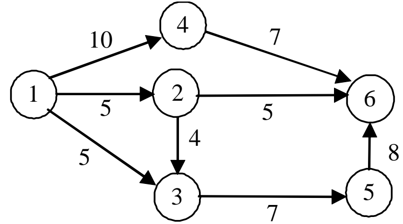
$$c(v_i,v_{i+1}) = c(v_i,v_{i+1}) - p,\ i = 0,1,...,k-1$$
$$c(v_{i+1},v_i) = c(v_{i+1},v_i) + p,\ i = 0,1,...,k-1$$

Връщаме се на стъпка 2, където търсим нов увеличаващ път и т.н.

Нека разгледаме графа от *фигура 5.4.6б.*, и изберем източник $s = 1$ и консуматор $t = 6$. Алгоритъмът ще намери последователно 5 увеличаващи пътя:

1. (*1*, *2*, *3*, *5*, *6*): минималното ребро от пътя е (*2*,*3*) и теглото му определя нарастването на общия поток: в случая с 4. Модифицираме теглата на ребрата — например, от връх 2 до връх 3 получаваме обратен поток с дължина 4, който (както се оказва на стъпка 3) се използва в строенето на следващ нарастващ път.
2. (*1*, *2*, *6*): минималното ребро от пътя е (*1*,*2*) с тегло 1 (тъй като сме намалили теглото му с 4 на стъпка 1).
3. (*1*, *3*, *2*, *6*): минималното ребро от пътя е (*3*,*2*) с тегло 4.
4. (*1*, *3*, *5*, *6*): минималното ребро от пътя е (*1*,*3*) с тегло 1.
5. (*1*, *4*, *6*): минималното ребро от пътя е (*4*,*6*) с тегло 7.

Така общата големина на максималния поток е 17.



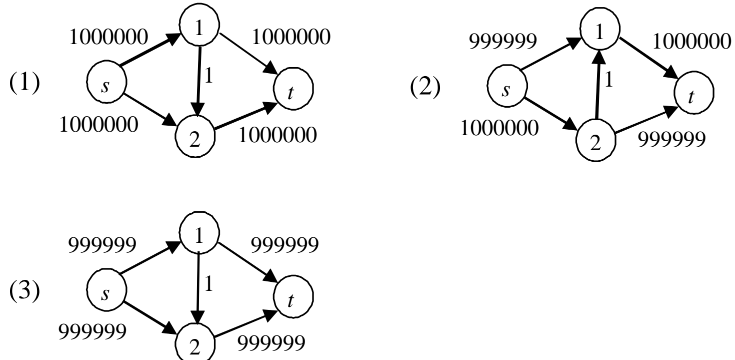
Фигура 5.4.6б. Максимален поток в граф.

Може да се докаже, че ако в графа няма увеличаващ път, то полученият поток е максимален [*Christofides-1975*]. Възможно е, обаче (ако в стъпка 2 от алгоритъма винаги се избира *произволен* увеличаващ път), сложността в най-лошия случай да достигне големината на максималния

поток, т.е. ще зависи от теглата на ребрата на графа. Така дори за съвсем прости графи (*виж фигура 5.4.6в.*) намирането на максималния поток може да отнеме неочаквано дълго време. За примера от фигурата сложността в най-лошия случай е равна на големината на максималния поток: На първата стъпка избираме нарастващ път *s*-1-2-*t* и увеличаваме потока с единица. Получава се графът (2), след което вземаме нарастващ път *s*-2-1-*t* — получава се графът (3), който е подобен на този, от който започнахме и т.н. Решението ще намери 1000000 увеличаващи пътища [*Cormen, Leiserson, Rivest-1997*].

За да се справим с последния проблем, ще използваме следната теорема [*Cormen, Leiserson, Rivest-1997*]:

Теорема. (*максимален поток*) Ако пътят, който се избира на стъпка 2), е винаги минималният по брой на участващи в него върхове, то сложността на алгоритъма в най-лошия случай е $\Theta(n^5)$, т.е. ще има най-много n^3 нарастващи пътища, всеки от които се намира със сложност $\Theta(n^2)$.



Фигура 5.4.6в. Пример за неефективност при произволно избиране на увеличаващ път.

В дадената по-долу реализация увеличаващите пътища се намират чрез обхождане в дълбочина, графът е представен с матрица на теглата `A[][]`, а потокът се пази в матрицата `F[][]`.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 100

#define MAX_VALUE 10000

const unsigned n = 10; /* Брой върхове в графа */
const unsigned s = 1;  /* Връх-източник */
const unsigned t = 6;  /* Връх-консуматор */

/* Матрица на теглата на графа */
int A[MAXN][MAXN] = {
  { 0, 5, 5, 10, 0, 0 },
  { 0, 0, 4, 0, 0, 5 },
  { 0, 0, 0, 0, 7, 0 },
  { 0, 0, 0, 0, 0, 7 },
  { 0, 0, 0, 0, 0, 8 },
  { 0, 0, 0, 0, 0, 0 }
};

int F[MAXN][MAXN];
```

```
unsigned path[MAXN];
char used[MAXN], found;

void updateFlow(unsigned pl)
{ int incFlow = MAX_VALUE;
  unsigned i;
  printf("Намерен увеличаващ път: ");
  for (i = 0; i < pl; i++) {
    unsigned p1 = path[i];
    unsigned p2 = path[i + 1];
    printf("%u, ", p1+1);
    if (incFlow > A[p1][p2]) incFlow = A[p1][p2];
  }
  printf("%u \n", path[pl]+1);
  for (i = 0; i < pl; i++) {
    unsigned p1 = path[i];
    unsigned p2 = path[i + 1];
    F[p1][p2] += incFlow;
    F[p2][p1] -= incFlow;
    A[p1][p2] -= incFlow;
    A[p2][p1] += incFlow;
  }
}

void DFS(unsigned i, unsigned level)
{ unsigned k;
  if (found) return;
  if (i == t-1) {
    found = 1;
    updateFlow(level - 1);
  }
  else
    for (k = 0; k < n; k++)
      if (!used[k] && A[i][k] > 0) {
        used[k] = 1;
        path[level] = k;
        DFS(k, level + 1);
        if (found) return;
      }
}

int main(void) {
  unsigned i, j;
  int flow;

  /* 1) инициализира се празен поток */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) F[i][j] = 0;

  /* 2) намира увеличаващ път, докато е възможно */
  do {
    for (i = 0; i < n; i++) used[i] = 0;
    found = 0;
    used[s-1] = 1;
    path[0] = s-1;
    DFS(s-1, 1);
  } while (found);

  /* Отпечатва потока */
```

```
  printf("Максимален поток през графа : \n");
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) printf("%4d", F[i][j]);
    printf("\n");
  }

  printf("\n");

  flow = 0;
  for (i = 0; i < n; i++) flow += F[i][t-1];
  printf("С големина : %d\n", flow);
  return 0;
}
```

🖫 fordfulk.c

Резултат от изпълнението на програмата:

```
Намерен увеличаващ път: 1, 2, 3, 5, 6
Намерен увеличаващ път: 1, 2, 6
Намерен увеличаващ път: 1, 3, 2, 6
Намерен увеличаващ път: 1, 3, 5, 6
Намерен увеличаващ път: 1, 4, 6
Максимален поток през графа:
   0   5   5   7   0   0   0   0   0   0
  -5   0   0   0   0   5   0   0   0   0
  -5   0   0   0   5   0   0   0   0   0
  -7   0   0   0   0   7   0   0   0   0
   0   0  -5   0   0   5   0   0   0   0
   0  -5   0  -7  -5   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0

С големина: 17
```

Оставяме на читателя задачата да модифицира горната програма така, че тя да намира увеличаващите пътища чрез обхождане в ширина, което ще гарантира минимален по-дължина път, тъй като ще изпълнява условието на теоремата.

В допълнение ще кажем, че съществуват алгоритми за намирането на максимален поток в граф със сложност $\Theta(n^4)$ и дори $\Theta(n^3)$ [*Cormen, Leiserson, Rivest-1997*].

Задача за упражнение:

Да се докаже, че ако в графа няма увеличаващ път, то полученият поток е максимален.

- повече от един източник и консуматор

Ако в задачата са дадени повече от един източник и консуматор, лесно можем да я сведем до току-що разгледаната. Нека източниците са $s_1, s_2,\ldots, s_p$, а консуматорите $t_1, t_2,\ldots, t_q$. Добавяме нов връх s в графа, наречен *суперизточник* и нов връх t — *суперконсуматор*. Освен тези два нови върха, добавяме и ребрата:

- (s, s_i), такива че $c(s, s_i) = +\infty$, за $i = 1, 2, \ldots, p$.
- (t_j, t), такива че $c(t_j, t) = +\infty$, за $j = 1, 2, \ldots, q$.

С помощта на алгоритъма на Форд-Фулкерсон от предходния параграф намираме максимален поток от s до t, който ще бъде равен на търсения в оригиналния вариант на задачата. (*Защо?*)

Задача за упражнение:

Да се докаже, че намереният от алгоритъма на Форд-Фулкерсон максимален поток върху модифицирания граф е максимален поток за изходния граф, който има няколко източника и няколко консуматора.

- капацитет на върховете

Възможно е да усложним допълнително задачата, като поставим ограничение за преминаващия през всеки връх поток. Нека е дадена функция $v(i)$, $v(i) \geq 0$, $i \in V$ такава че:

$$\sum_{j \in V} f(i, j) \leq v(i),\ i \in V$$

Търсим максимален поток от източника s до консуматора t. И тук задачата можем да решим, като я сведем до стандартната задача за търсене на максимален поток. За целта ще конструираме нов граф $G'(V',E')$, в който:

1) На всеки връх $i \in V$ ще отговарят два върха от V': i_1 и i_2, които ще бъдат свързани с ребро (i_1, i_2). Пропускливостта $c(i_1, i_2)$ на това ребро ще бъде равна на $v(i)$.
2) Всяко едно ребро (i, j) от G се пренася в G' като реброто (i_2, j_1).
3) В новия граф G' няма да има функция v, ограничаваща потока през върховете.

Както и в предходния параграф, намереният по алгоритъма на Форд-Фулкерсон максимален поток в G', ще бъде максимален и за графа G. (*Защо?*)

Други приложения на задачата за максимален поток ще бъдат разгледани в *5.6.4.* и *5.7.6*.
Задача за упражнение:

Да се докаже, че намереният от алгоритъма на Форд-Фулкерсон максимален поток върху модифицирания граф е максимален поток за изходния граф.

5.5. Транзитивност и построяване. Топологично сортиране

В началото на главата обърнахме внимание, че множества от обекти с дефинирани релации между тях могат да бъдат представени с граф.

Подобен прост пример е йерархичната организация в една корпорация. Всеки служител в нея е пряко подчинен на някой друг — такива отношения можем да представим чрез дърво. Те обаче, обикновено са по-сложни. Например, някой може да има повече от един пряк "началник", или пък да има цикличност. В подобни случаи се налага да се използва ориентиран граф. Възможни задачи са да се провери дали A е подчинен на B (пряко или косвено), или да бъдат подредени в списък хората съгласно йерархията, т.е. за всяка двойка (A, B), ако A е подчинен на B, то A трябва да бъде по-назад в списъка от B.

Като втори пример, нека е даден аквариум и желаем да купим *риби*. Известно е кои видове могат да съжителстват заедно и кои — не. Една възможна задача е да се намери максималният брой различни видове рибки, които могат да живеят заедно.

Задачите, които ще разгледаме в настоящия параграф, решават изброените примери, както и много други практически проблеми.

5.5.1. Транзитивно затваряне. Алгоритъм на Уоршал

В *5.4.2*. използвахме алгоритъма на *Уоршал*, когато искахме да намерим кои двойки върхове от графа са свързани с път. От матрицата на съседство `A` получихме матрица на достижимост `A'`. Казахме, че графът G' с матрица на съседство `A'` се нарича *транзитивно затваряне* на G.

По-горе разгледахме задача за подредба на служителите в една корпорация, според йерархията в нея. Нека построим съответен ориентиран (и евентуално цикличен) граф $G(V,E)$ по следния начин: реброто $(i,j)\in E$ съществува т.с.т.к. върхът i е пряк началник на j. Ако намерим транзитивното затваряне $G'(V',E')$ на G, то $(i,j)\in E'$ т.с.т.к i е (пряк или косвен) началник на j.

Могат да бъдат намерени още редица приложения на транзитивното затваряне. Като просто упражнение оставяме на читателя съобрази как то може да се приложи в следната задача: Дадено е множество от събития и релация, показваща кое събитие води директно до сбъдване на друго събитие. За дадено събитие t, искаме да намерим множеството от събития S, които не биха могли да се случат, преди да се случи t. Например, да намерим множеството от всички събития, които не могат да се случат преди да навършим пълнолетие.

Основна роля при разгледаните задачи играе цикличността в граф. В примери, които могат да бъдат представени с ацикличен граф, търсенето на “косвени” релации се решава значително по-ефективно (*виж 5.5.5 — Топологично сортиране*). Ще припомним алгоритъма на Уоршал от *5.4.2*:

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
    if (A[i][k])
      for (j = 0; j < n; j++)
        if (A[k][j])
          A[i][j] = 1;
```

Теорема. (*Уоршал*) Разглеждането на тройките върхове (i, j, k) чрез три `for` цикъла вложени по начина, показан по-горе, гарантира правилното намиране на транзитивно затваряне на граф [*Brassard,Bratley–1996*].

На пръв поглед, последната теорема не казва нищо ново. Тя представлява вариант “отгоре-надолу” (*виж глава 8*) на съответната интуитивна рекурсивна схема за намиране на транзитивно затваряне:

$$A_k[i][j] = \begin{cases} A[i][j], k = 0 \\ \min\{A_{k-1}[i][j], A_{k-1}[i][k] + A_{k-1}[k][j]\}, k > 0 \end{cases}$$

Важно е да се отбележи подредбата на циклите. Така например, ако вместо фрагмента по-горе, изпълним:

```
for (i = 0; i < n; i++)
  for (k = 0; k < n; k++) if (A[i][k])
    ...
```

ще получим алгоритъм, който в общия случай няма да намира правилно транзитивното затваряне на произволен граф. (*Защо?*)

Задачи за упражнение:

1. Да се докаже теоремата на Уоршал.
2. Да се докаже, че циклите не могат да се разменят. Да се посочи примерен граф, при който разместването води до проблем.

5.5.2. Транзитивно ориентиране

Дефиниция 5.19. Даден е неориентиран граф $G(V,E)$. *Ориентиране на ребрата* му се нарича въвеждането на наредба за всяко неориентирано ребро (i,j) от E, т.е. графът от неориентиран се превръща в ориентиран.

Дефиниция 5.20. Нека е даден ориентиран граф $G(V,E)$, в който за всеки три върха $i,j,k \in V$ е изпълнено: Ако $(i,k) \in E$ и $(k,j) \in E$, то следва, че реброто $(i,j) \in E$. Тогава G се нарича *транзитивен*.

Дефиниция 5.21. Неориентиран граф $G(V,E)$ се нарича *транзитивно ориентируем*, ако е възможно да се ориентират ребрата му по такъв начин, че новополученият граф да бъде *транзитивен*. В противен случай графът се нарича *транзитивно неориентируем*.

Ще разгледаме задачата за транзитивно ориентиране на граф.

Алгоритъм на Пнуели, Лемпел и Ивена

Ще приемем, че даденият неориентиран граф е свързан (ако не е, то алгоритъмът се прилага последователно за всяка негова компонента на свързаност).

С i-j ще означаваме факта, че върховете $i \in V$ и $j \in V$ са свързани с ребро, което не е ориентирано. С $i \rightarrow j$ ще означаваме ребро, което е ориентирано от i към j; с $i \leftarrow j$ ще означаваме ориентирано ребро от j към i, а с $i \neq j$ — липсата на ребро между i и j.

Ще използваме следните две правила (за $i, j, k \in V$):

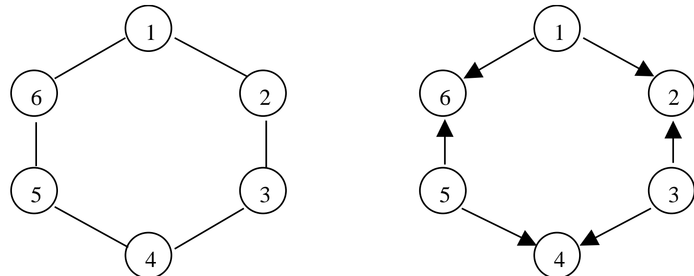
Правило R1. Ако са изпълнени условията $i \rightarrow j$, j–k, $i \neq k$, ориентираме реброто j–k: $k \rightarrow j$.

Правило R2. Ако са изпълнени условията $i \rightarrow j$, i–k, $j \neq k$, ориентираме реброто i–k: $i \rightarrow k$.

Алгоритъм

1) Избираме произволно неориентирано ребро и му задаваме някаква ориентация. Маркираме ориентираното ребро като разгледано така, че, ако втори път се върнем на тази стъпка, да не избираме същото ребро.
2) Прилагаме правилата *R1* и *R2*, докато е възможно, за всяко ориентирано ребро:
   Нека разглеждаме реброто $i \rightarrow j$. Тогава за всеки връх $k \in V$, съседен на j, преминаваме на случай 2.1), а за всеки връх k, съседен на i — на случай 2.2).
   2.1). Нека разглеждаме реброто (j,k)
      а.(по правило *R1*) Ако е изпълнено $i \neq k$ и j–k, то ориентираме (j,k) като $j \leftarrow k$
      б.(противоречие с правило *R1*) Ако е изпълнено $i \neq k$ и $j \rightarrow k$, то графът е *транзитивно неориентируем* и приключваме.
   2.2). Нека разглеждаме реброто (i,k)
      а.(по правило *R2*) Ако е изпълнено $j \neq k$ и i–k, то ориентираме (i,k) като $i \rightarrow k$.
      б.(противоречие с правило *R2*) Ако е изпълнено $j \neq k$ и $i \leftarrow k$ то графът е *транзитивно неориентируем* и приключваме.
3) Проверяваме дали всички ребра са ориентирани. Ако това е така, то сме получили транзитивното ориентиране на графа и приключваме. В противен случай, ако не са ориентирани всички ребра, премахваме ориентираните ребра от графа и се връщаме отново на стъпка 1).

Пример за транзитивно ориентиране е разгледан на *фигура 5.5.2*.



Фигура 5.5.2. Транзитивно ориентиране.

В реализацията, дадена по-долу, графът е представен с матрица на съседство `A[][]`. Ако неориентираното ребро (*i*,*j*) е от графа, то `A[i][j] == 1` и `A[j][i] == 1`. В противен случай стойностите на тези елементи са равни на 0. Когато ориентираме ребро *i*–*j* като *i*→*j*, в матрицата присвояваме `A[i][j] = 2` и `A[j][i] = -2`. Тъй като на стъпка 3) от алгоритъма се премахват всички ориентирани ребра, трябва да се погрижим да ги запазваме по някакъв начин, така че след завършване на алгоритъма да бъде възможно да отпечатаме транзитивно ориентирания граф. За тази цел ще използваме още две стойности: за всички елементи `A[i][j] == 2` (и съответно `A[j][i] == -2`) ще присвояваме `A[i][j] = -3` и `A[j][i] = -4`. Така проверката дали реброто (*i*, *j*) е от графа ще бъде `if (A[i][j] > 0)`.

Следва изходният код на програмата:

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 150

/* Брой върхове в графа */
const unsigned n = 6;

/* Матрица на съседство на графа */
int A[MAXN][MAXN] =
{
  { 0, 1, 0, 0, 0, 1 },
  { 1, 0, 1, 0, 0, 0 },
  { 0, 1, 0, 1, 0, 0 },
  { 0, 0, 1, 0, 1, 0 },
  { 0, 0, 0, 1, 0, 1 },
  { 1, 0, 0, 0, 1, 0 }
};

/* // Пример за транзитивно неориентируем граф
   const unsigned n = 5;
   int A[MAXN][MAXN] = {
     { 0, 1, 0, 0, 1},
     { 1, 0, 1, 0, 0},
     { 0, 1, 0, 1, 0},
     { 0, 0, 1, 0, 1},
     { 1, 0, 0, 1, 0}};
 */

char trOrient(void)
{ /* намира броя на ребрата в графа */
  unsigned i, j, k, r, tr = 0;
  char flag;
```

```
  for (i = 0; i < n - 1; i++)
    for (j = i + 1; j < n; j++)
      if (A[i][j]) tr++;

  r = 0;
  do {
    for (i = 0; i < n; i++) { /* стъпка 1 – ориентиране на ребро (i,j) */
      for (j = 0; j < n; j++)
        if (1 == A[i][j]) {
          A[i][j] = 2;
          A[j][i] = -2;
          break;
        }
      if (j<n) break;
    }

    /* прилагане на правила 1) и 2), докато е възможно */
    do {
      flag = 0;
      for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
          if (2 == A[i][j]) {
            for (k = 0; k < n; k++) {
              if (i != k && j != k) {
                if (0 == A[i][k] || A[i][k] < -2) { /* случай 2.1) */
                  /* a) -> графът е транзитивно неориентируем */
                  if (2 == A[j][k]) return 1;
                  /* b) -> ориентираме реброто (j,k) */
                  if (1 == A[j][k])
                    { A[k][j] = 2; A[j][k] = -2; flag = 1; }
                }
                if (0 == A[j][k] || A[j][k] < -2) { /* случай 2.2) */
                  /* a) -> графът е транзитивно неориентируем */
                  if (2 == A[k][i]) return 1;
                  /* b) -> ориентираме реброто (i,k) */
                  if (1 == A[i][k]) {
                    A[i][k] = 2;
                    A[k][i] = -2;
                    flag = 1;
                  }
                }
              }
            }
          }
        }
      }
    } while (flag);

    /* стъпка 3 – изключваме ориентираните ребра от графа */
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        if (2 == A[i][j]) {
          A[i][j] = -3; A[j][i] = -4; r++;
        }
  } while (r < tr);

  /* повтаря се, докато всички ребра от графа бъдат ориентирани */
  return 0;
}
```

```
void printGraph(void)
{ unsigned i, j;
  printf("Транзитивното ориентиране е: \n");
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
      if (-3 == A[i][j]) printf("  1");
        else (-4 == A[i][j]) ? printf(" -1") : printf("  0");
    printf("\n");
  }
}
int main(void) {
  if (trOrient())
    printf("Графът е транзитивно неориентируем! \n");
  else
    printGraph();
  return 0;
}
```

🖫 trans-or.c

Резултат от изпълнението на програмата:

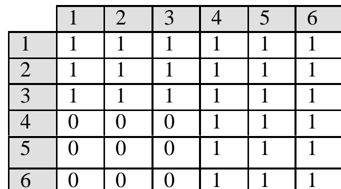
```
Транзитивното ориентиране е:
  0  1  0  0  0  1
 -1  0 -1  0  0  0
  0  1  0  1  0  0
  0  0 -1  0 -1  0
  0  0  0  1  0  1
 -1  0  0  0 -1  0
```

Задача за упражнение:

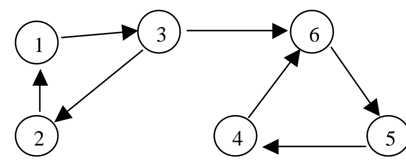
Да се докаже, че алгоритъмът на Пнуели, Лемпел и Ивена работи коректно.

5.5.3. Транзитивна редукция

Дефиниция 5.22. Даден е ориентиран граф $G(V, E)$ с матрица на достижимост `W`. Граф $G'(V,E')$, $E' \subseteq E$, с минимален брой ребра, чиято *матрица на достижимост* отново е `W`, се нарича *транзитивна редукция* на G.



	1	2	3	4	5	6
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	1	1	1	1	1
4	0	0	0	1	1	1
5	0	0	0	1	1	1
6	0	0	0	1	1	1



Фигура 5.5.3. Транзитивна редукция.

Например, показаният на *фигура 5.5.3.* (вдясно) граф е транзитивна редукция на всеки ориентиран граф, съдържащ дадения, и имащ за матрица на достижимост показаната вляво.

Ще разгледаме задачата за намиране на транзитивна редукция на даден ориентиран граф $G(V,E)$ с матрица на достижимост `W`.

Обратен алгоритъм на Уоршал

Инициализираме матрица `A[][]` със стойности, каквито са дадените в матрицата `W[][]`. Последователно премахваме ребра от `A[][]`, докато получим търсената транзитивна редукция (в

края на алгоритъма `A[][]` ще бъде матрицата на съседство за търсения граф с минимален брой ребра).

За всяко $k = 1, 2, ..., n$ намираме такива два върха i и j, за които:

- съществува ребро (i, k), т.е. `A[i][k] == 1`.
- съществува път от k до j, т.е. `W[k][j] == 1`.
- Ако са изпълнени горните две точки и реброто (i, j) съществува (`A[i][j] == 1`), то се премахва, защото е излишно.

Алгоритъмът се реализира с три вложени цикъла и както при алгоритъма на Уоршал, цикълът по k трябва да бъде най-външен:

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
    if (A[i][k])
      for (j = 0; j < n; j++)
        if (A[i][j] && W[k][j]) A[i][j] = 0;
```

Задача за упражнение:

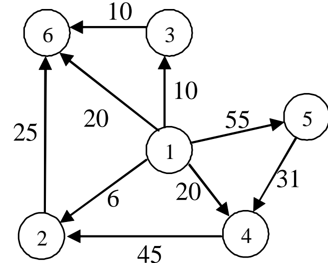
Какво ще се получи, ако се премахне условието $E' \subseteq E$ от дефиницията на транзитивна редукция, т.е. търси се граф с минимален брой ребра (не задължително подмножество на дадените) по дадена матрица на достижимост? Ще работи ли коректно и в този случай обратният алгоритъм на Уоршал?

5.5.4. Контрол на компании

Задача: Даден е претеглен ориентиран граф $G(V,E)$ с тегла на ребрата естествени числа между 0 и 100. Казваме, че върхът *i контролира j*, ако съществува реброто (i,j), $f(i,j) > 50$ или съществува множество от върхове $v_1, v_2,..., v_k$, контролирани от i, и съществуват ребрата (v_1,j), (v_2,j),..., (v_k,j) и $f(v_1, j) + f(v_2, j) + ... + f(v_k, j) > 50$. Задачата е по зададен връх m да се намерят всички контролирани от m върхове.

Задачата описва реална ситуация, в която върховете на графа представят компании, а всяко ребро (i, j) показва процента от акциите, които компанията i притежава от компанията j.

Примерни входни данни за задачата са показани на *фигура 5.5.4.*: на фигурата компания *1* контролира *2*, *4* и *5*.



Фигура 5.5.4. Ориентиран претеглен граф.

Задачата може да се реши чрез адаптация на алгоритъма на Уоршал за транзитивно затваряне. Ще въведем масив `control[]`, в който ще пазим притежаваните от компанията i проценти от другите компании. В началото инициализираме масива с теглата на ребрата от i към останалите върхове. Всеки път, когато се появи нова компания, контролирана от i, ще добавяме процентите, които тази компания контролира от останалите, към масива `control[]`. Нова компания, контро-

лирана от *i*, се появява тогава, когато процентите в някоя позиция на `control[]` станат над 50. Сложността на описания алгоритъм е $\Theta(n^3)$.

```
#include <stdio.h>

/* Максимален брой компании (върхове в графа) */
#define MAXN 150

/* Брой компании (върхове в графа) */
const unsigned n = 6;
const unsigned m = 1;     /* Търсим кои компании контролира компания 1 */

/* Матрица на съседство на графа */
const unsigned A[MAXN][MAXN] = {
  { 0,  6, 10, 20, 55, 20 },
  { 0,  0,  0,  0,  0, 25 },
  { 0,  0,  0,  0,  0, 10 },
  { 0, 45,  0,  0,  0,  0 },
  { 0,  0,  0, 31,  0,  0 },
  { 0,  0,  0,  0,  0,  0 }
};

unsigned control[MAXN];
char used[MAXN];

void addControls(void)
{ unsigned i, j;
  for (i = 0; i < n; i++)
    /* компанията i е контролирана, прибавяме акциите ѝ към тези на m */
    if (control[i] > 50 && !used[i]) {
      for (j = 0; j < n; j++) control[j] += A[i][j];
      used[i] = 1;
    }
}

void solve(void)
{ unsigned i;
  for (i = 0; i < n; i++) {
    control[i] = 0;
    used[i] = 0;
  }
  for (i = 0; i < n; i++) control[i] = A[m-1][i];
  for (i = 0; i < n; i++) addControls();
}

void printResult(void) {
  unsigned i;
  printf("Компания %u контролира следните компании: \n", m);
  for (i = 0; i < n; i++)
    if (control[i] > 50) printf("%u: %3u% \n", i, control[i]);
  printf("\n");
}

int main(void) {
  solve();
  printResult();
}
```

company.c

Резултат от изпълнението на програмата:
```
Компания 1 контролира следните компании:
1:  53%
3:  51%
4:  55%
```

Задача за упражнение:

Възможно ли е задачата да се реши със сложност, по-малка от $\Theta(n^3)$?

5.5.5. Топологично сортиране

Дефиниция 5.23. *Топологично сортиране* на ориентиран ацикличен граф $G(V, E)$ се нарича линейно нареден списък Z от върховете му, за който е изпълнено: За всеки два върха i, $j \in V$, ако съществува път от i до j, то върхът i трябва да предшества j в Z.

На практика, списъкът Z рядко може да се построи по единствен начин.

Дефиниция 5.24. Когато съществува повече от едно нареждане на върховете в Z, множеството от всички възможни списъци Z се нарича *пълно топологично сортиране*.

Твърдение 5.25. Даден е ориентиран ацикличен граф $G(V,E)$. Съществува единствен списък Z, който е топологично сортиране на G, т.с.т.к. G е слабо свързан.

Пример 1: В манастир в Тибет има строга йерархия: всеки монах има един или няколко преки началници (които отговарят за прегрешенията му), а за тези, които нямат шефове, се счита, че самият бог Шива им е началник. Всяка година се извършват ритуални жертвоприношения и трябва да се изберат q на брой монаси, които да се хвърлят в “бездната на безкрайността”. Винаги се избират за жертвоприношение монаси измежду тези, които стоят най-ниско в йерархията на манастира. Управителят на манастира обаче е зле с теорията на графите и е намерил програмист да определи тези q монаси, които трябва да се жертват. Очевидно задачата е да се намери топологичното сортиране на графа и да се принесат в жертва последните q монаси от получения списък Z.

Пример 2: Друг пример е учебник по математика. В доказателството на всяка теорема се използват други теореми, дефиниции, аксиоми. Учебникът трябва да бъде написан така, че да бъде последователен — "топологично сортиран" и всяка теорема да цитира само неща, които вече са били дефинирани/доказани по-рано.

Тъй като в разглежданите ориентирани графи не се допуска цикличност, то в тях винаги съществува поне един връх, който няма предшественици. (Защо?) Именно такъв връх (забележете, че могат да бъдат повече от един) ще заема челно място в наредбата. Като следствие от току-що направеното заключение, получаваме

Алгоритъм 1 за топологично сортиране

1) Инициализираме Z като празен списък.
2) Избираме връх i без предшественици и го добавяме в края на списъка Z. Изключваме i от графа, както и всички ребра, инцидентни с него.
3) Повтаряме стъпка 2), докато не остане нито един връх.

Забележки:

1. Ако на стъпка 2) има повече от един връх без предшественици, избираме произволен. (Защо?)

2. Ако на стъпка 2) няма нито един връх без предшественици, а в графа има още върхове, то следва, че графът е цикличен, което противоречи на условието и следователно не съществува топологично сортиране. Обратното също е вярно: ако графът е цикличен, то на някоя стъпка от алгоритъма задължително ще попаднем в описаната ситуация. (Защо?)

Реализацията на този алгоритъм се извършва най-ефективно, ако представим графа чрез списък на наследниците (или списък на предшествениците). Ще пазим броя num_i на предшествениците за всеки връх $i \in V$. Всички num_i могат да се намерят с обща сложност $\Theta(n+m)$ чрез едно обхождане на графа. Така, след като намерим наследниците i_1, i_2, ..., i_k на всеки връх i, увеличаваме с единица num_{i_1}, num_{i_2},..., num_{i_k}. По-нататък, на всяка стъпка избираме връх j, за който $num_j = 0$. Добавяме го в края на списъка Z, и изваждаме единица от num_j за всеки наследник j на избрания връх. Общият брой стъпки ще бъде n (при представяне с матрица на съседство). Така сложността на алгоритъма става $\Theta(n^2)$, а може да се сведе до $\Theta(m+n)$ по следния начин:

1) В началото поставяме в Z всички върхове, които имат брой на предшествениците, равен на нула. Ще въведем указател, който сочи последния неразгледан елемент на Z (в началото указателят сочи към първия елемент на Z).
2) Нека разглеждаме върха i, към който сочи указателят. За всеки негов наследник j намаляваме num_j с единица. Ако някое num_j стане равно на нула, добавяме j в края на Z. Преместваме указателя към следващия елемент от Z и повтаряме стъпка 2).

Ако графът не съдържа цикъл, след най-много n повторения на 2) всички върхове от графа ще се намират в Z и той ще бъде топологично сортиран.

Алгоритъм 2 за непълно топологично сортиране

Вторият алгоритъм, който ще разгледаме, има същата сложност $\Theta(m+n)$, както и *алгоритъм 1*. Той в известна степен се явява "обратен" на предишния: на всяка стъпка ще търсим връх *без наследници* и така ще получим топологичното сортиране на графа в обратен ред. Реализацията се извършва лесно и елегантно, като се използва обратният ход на рекурсията във функцията за обхождане в дълбочина:

```
void DFS(unsigned i)
{ unsigned k;
  used[i] = 1;
  for (k = 0; k < n; k++)
    if (A[i][k] && !used[k]) DFS(k);
  printf("%u ", i + 1);
}
```

Както се вижда, единствената модификация на стандартното *DFS* от *5.3.2.* е мястото на отпечатване на върха *i*. Това става след извършване на всички рекурсивни извиквания, с което си осигуряваме, че всички върхове, до които може да се достигне от текущия връх, са вече разгледани и отпечатани.

В `main` функцията ще стартираме *DFS*(*i*) за всеки връх *i*, който още не сме разгледали и който няма предшественици. Последното условие е наложителното за правилната работа на алгоритъма и ще го проверяваме, като използваме масива `used[]`, въведен за маркиране на посетените върхове. Следва изходният код на програмата:

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 200

/* Брой върхове в графа */
const unsigned n = 5;
```

```
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
  { 0, 1, 0, 0, 0 },
  { 0, 0, 1, 0, 1 },
  { 0, 0, 0, 1, 0 },
  { 0, 0, 0, 0, 0 },
  { 0, 0, 1, 0, 0 }
};

char used[MAXN];

/* модифицирано DFS */
void DFS(unsigned i)
{ unsigned k;
  used[i] = 1;
  for (k = 0; k < n; k++)
    if (A[i][k] && !used[k]) DFS(k);
  printf("%u ", i + 1);
}

int main(void) {
  unsigned i;
  /* инициализация */
  for (i = 0; i < n; i++) used[i] = 0;
  printf("Топологично сортиране (в обратен ред): \n");
  for (i = 0; i < n; i++)
    if (!used[i]) DFS(i);
  printf("\n");
  return 0;
}
```

🖫 topsort.c

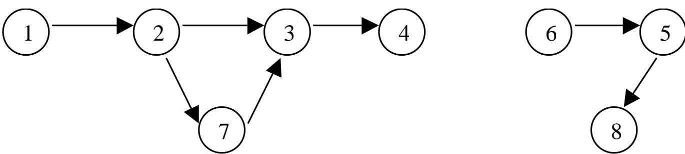
Резултат от изпълнението на програмата:

```
Топологично сортиране (в обратен ред):
4 3 5 2 1
```

Задачи за упражнение:

1. Да се докаже Твърдение 5.25.
2. Да се докаже, че ако на стъпка 2 на алгоритъм 1 има повече от един връх без предшественици, сме свободни да изберем произволен измежду тях.

5.5.6. Пълно топологично сортиране



Фигура 5.5.6. Пълно топологично сортиране.

Да разгледаме графа на *фигура 5.5.6*. Едно топологично сортиране за *G* би могло да бъде *Z*=(*1*, *2*, *7*, *3*, *4*, *6*, *5*, *8*) или *Z*=(*6*, *1*, *2*, *5*, *7*, *3*, *8*, *4*). Като се използва програмата за намиране на пълно топологично сортиране (която ще приведем по-късно) може да се пресметне, че броят на всички различни списъци *Z* е 56. (Защо?)

За момент ще се върнем отново на *алгоритъм 1* от *5.5.5*. Когато търсихме топологично сортиране, на всяко ниво избирахме връх без предшественици, а когато имаше няколко — избирахме произволен. От недетерминираността на избора следват и различните решения. За да намерим множеството от *всички* възможни списъци *Z*, ще приложим пълно изчерпване. Така, когато е налице повече от една възможност за избор на връх, ще изпробваме последователно всички (ясно е, че всеки избор води до решение). В реализацията по-долу това се извършва от функцията `fullTopSort()`. При разглеждане на връх ще го премахваме от графа и ще стартираме рекурсивно `fullTopSort()` с модифицирания граф. След връщане от рекурсията възстановяваме графа и повтаряме същото със следващия възможен връх. Дъното на рекурсията е, когато списъкът Z съдържа *n* върха, при което отпечатваме намереното решение. Следва схемата на основната рекурсивна функция:

```
fullTopSort(count) {
  if (count == n) {
    Намерили_сме_топологично_сортиране_=>_отпечатваме_го;
    return;
  }
  for (всеки връх Vi без_предшественици)
    премахваме_Vi_от_графа;
    fullTopSort(count + 1);  /* рекурсия и връщане */
    възстановяваме_Vi_във_графа;
  }
}
```

Следва пълната програма:

```
#include <stdio.h>

#define MAXN 200 /* Максимален брой върхове в графа */
/* Брой върхове в графа */
const unsigned n = 8;
/* Матрица на съседство на графа */
char A[MAXN][MAXN] = {
  { 0, 1, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 1, 0, 0, 0, 1, 0 },
  { 0, 0, 0, 1, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 1 },
  { 0, 0, 0, 0, 1, 0, 0, 0 },
  { 0, 0, 1, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0 }
};

char used[MAXN];
unsigned topsort[MAXN], total = 0;

void printSort(void)
{ unsigned i;
  printf("Топологично сортиране номер %u: ", ++total);
  for (i = 0; i < n; i++) printf("%u ", topsort[i] + 1);
  printf("\n");
}

void fullTopSort(unsigned count)
{ unsigned i, j, k, saved[MAXN];

  if (count == n) { printSort(); return; }
```

```
  /* намира всички върхове без предшественик */
  for (i = 0; i < n; i++) {
    if (!used[i]) {
      for (j = 0; j < n; j++)
        if (A[j][i]) break;
        if (j == n) {
          for (k = 0; k < n; k++) {
            saved[k] = A[i][k]; A[i][k] = 0;
          }
          used[i] = 1;
          topsort[count] = i;
          fullTopSort(count + 1); /* рекурсия */
          used[i] = 0;
          for (k = 0; k < n; k++) A[i][k] = saved[k];
        }
    }
  }
}
int main(void) {
  unsigned i;
  for (i = 0; i < n; i++) used[i] = 0;
  fullTopSort(0);
  return 0;
}
```

🖫 topsortf.c

Резултат от изпълнението на програмата:

```
Топологично сортиране номер 1: 1 2 6 5 7 3 4 8
Топологично сортиране номер 2: 1 2 6 5 7 3 8 4
Топологично сортиране номер 3: 1 2 6 5 7 8 3 4
Топологично сортиране номер 4: 1 2 6 5 8 7 3 4
Топологично сортиране номер 5: 1 2 6 7 3 4 5 8
…
Топологично сортиране номер 49: 6 1 5 2 8 7 3 4
Топологично сортиране номер 50: 6 1 5 8 2 7 3 4
Топологично сортиране номер 51: 6 5 1 2 7 3 4 8
Топологично сортиране номер 52: 6 5 1 2 7 3 8 4
Топологично сортиране номер 53: 6 5 1 2 7 8 3 4
Топологично сортиране номер 54: 6 5 1 2 8 7 3 4
Топологично сортиране номер 55: 6 5 1 8 2 7 3 4
Топологично сортиране номер 56: 6 5 8 1 2 7 3 4
```

Задача за упражнение:

Да се пресметне *аналитично* броят на различните топологични сортирания на графа от *фигура 5.5.6*.

5.5.7. Допълване на ацикличен граф до слабо свързан

Задача: Даден е ориентиран ацикличен граф $G(V, E)$. Търсим граф $G'(V, E')$, такъв че:

- $E \subseteq E'$
- Съществува *единствен* списък Z, който е топологично сортиране на G'.

Казано с други думи, търсим множество от ребра, които да добавим към G, така че да съществува единствен топологично сортиран списък Z на G'.

Задачата може да бъде зададена и така: Да се допълни множеството от ребра на ориентиран ацикличен граф, така че той да се превърне в слабо свързан, като остане ацикличен.

От предефинирането на задачата може да се направи следният извод: след извършване на транзитивно затваряне върху графа матрицата на съседство `A` трябва да има следния вид:

(*) За всяко $i \neq j$ *точно един* от двата елемента `A[i][j]` и `A[j][i]` да бъде равен на единица, а другият — на нула. (Защо?)

Алгоритъм 1

Първият алгоритъм, който ще предложим за решение на задачата, е следният:

1) Извършваме транзитивно затваряне и, ако има $i,j \in V$, $i \neq j$, за които `A[i][j] == 0` и `A[j][i] == 0`, то присвояваме `A[i][j] = 1`
2) Изпълняваме 1), докато (*) стане вярно за всяко i, j (т.е. броят на единиците в A стане $\left(\dfrac{n^2 - n}{2}\right)$.

Този интуитивен алгоритъм има сложност $\Theta(n^4)$ и е неприложим на практика. (Защо?)

Алгоритъм 2

Нека разгледаме едно топологично сортиране на графа $Z = (v_{i1}, v_{i2}, \ldots, v_{in})$. Допълването до G' се извършва по следната схема:

```
for (j = 0; j < n-1; j++)
  for (k = j+1; k < n; k++)
    if (не_съществува_реброто_(vij,vik)) { добавяме реброто (vij,vik); }
```

Този алгоритъм очевидно е по-ефективен от *Алгоритъм 1* и има сложност $\Theta(n^2)$. (Защо?)

Задачи за упражнение:

1. Да се изведе сложността на алгоритъм 1.
2. Нека е даден ацикличен граф $G(V, E)$. Търсим граф $G'(V, E')$, такъв че:
   - $E \subseteq E'$
   - Съществува *единствен* списък Z', който е топологично сортиране на G'.
   - E' съдържа минимален брой ребра.

   Каква е най-добрата сложност, която може да се постигне?

5.5.8. Построяване на граф по дадени степени на върховете

Задача: Дадени са степените на върховете на неориентиран граф. Да се построи графът (т.е. да се определи множеството от ребрата му). Възможно е задачата да няма решение, т.е. да не съществува граф, със зададените степени.

Решение: Не е особено трудно да се намери алгоритъм за решаване на задачата. Ще го построим въз основа на следното наблюдение: нека разгледаме един произволен връх от графа, да го означим с i, а степента му — с $d(i)$. Очевидно, за да има решение задачата, трябва да съществуват поне $d(i)$ върха със степен, по-голяма или равна на едно, които да са съседни на i. Можем да ги свържем и да намалим степента им с единица, а степента на върха i да установим в 0. Продължаваме така, докато е възможно.

Остана да уточним единствено как ще избираме върховете (това не може да става по произволен начин — защо?). Така, на всяка стъпка ще избираме този връх, който има най-висока степен и ще го свързваме отново с върховете с най-висока степен.

Реализацията на описания алгоритъм предоставяме на читателя като леко упражнение. Най-простата реализация би била със сложност $\Theta(n^2.\log_2 n)$: n стъпки, на всяка от които сортираме степените. Разбира се, сортирането не е необходимо, тъй като преподреждането, след

намаляване на степените с 1, може да се извърши с линейна сложност (как?), и така да получим обща сложност $\Theta(n^2)$.

Задачи за упражнение:

1. Ще работи ли коректно описаният алгоритъм, ако разглеждаме *мултиграф*?
2. Ще работи ли коректно описаният алгоритъм, ако в графа има *примки*?
3. Да се покаже, че редът, в който се разглеждат върховете, е от значение.
4. Да се предложи начин за преподреждане на върховете с линейна сложност.
5. Възможно ли е да се реши задачата със сложност $\Theta(m+n)$? А с по-малка?

5.6. Достижимост и свързаност

Задачите от достижимост и свързаност намират приложение в компютърните, комуникационните, пътните мрежи и др.

Пример: Дадена е компютърна мрежа, в която между някои от компютрите съществува директна връзка. Ако представим мрежата като ориентиран граф, интерес могат да представляват следните задачи:

- *Свързаност на мрежата*: Да се определи дали има връзка между *всеки* два компютъра, независимо дали е директна или индиректна, т.е. преминаваща през други компютри. За да се реши задачата, е необходимо да се провери дали графът е свързан.
- *Намиране на клъстер*: Да се определи най-голямото множество свързани компютри.
- *Стабилност на мрежата*: Ако всички компютри в системата са свързани, то следваща задача може да бъде да се намерят *разделящите компютри* или *разделящите връзки*, т.е. такива, че ако се премахнат, ще се появят двойки компютри, които вече не са свързани.

Тези и други задачи ще бъдат разгледани в настоящия параграф и решени с помощта на алгоритми от теория на графите.

5.6.1. Компоненти на свързаност

Задача: Даден е неориентиран граф. Да се провери дали е свързан, т.е. дали съществува път между всеки два негови върха. В случай, че не е свързан, да се намерят всички негови свързани компоненти, т.е. всички максимални свързани негови подграфи.

Алгоритъм

Задачата може да се реши чрез обхождане на графа, например в дълбочина:

1) Започваме от произволен връх и всички върхове, които разгледаме при обхождането, маркираме като принадлежащи на една свързана компонента. Ако тя съдържа всички върхове на графа, следва, че той е свързан.
2) Ако са останали необходени върхове, това означава, че графът не е свързан. Започваме ново обхождане от непосетен връх и строим втора свързана компонента. Повтаряме стъпка 2), докато не останат необходени върхове.

Сложността на алгоритъма е $\Theta(n+m)$.

Реализацията се състои в проста модификация на функцията за обхождане в дълбочина. Следва изходният код на програмата.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
```

```
#define MAXN 200
/* Брой върхове в графа */
const unsigned n = 6;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
  { 0, 1, 1, 0, 0, 0 },
  { 1, 0, 1, 0, 0, 0 },
  { 1, 1, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 1, 1 },
  { 0, 0, 0, 1, 0, 1 },
  { 0, 0, 0, 1, 1, 0 }
};

char used[MAXN];

/* модифицирано DFS */
void DFS(unsigned i)
{ unsigned k;
  used[i] = 1;
  printf("%u ", i + 1);
  for (k = 0; k < n; k++)
    if (A[i][k] && !used[k]) DFS(k);
}

int main(void) {
  unsigned i, comp;
  /* инициализация */
  for (i = 0; i < n; i++) used[i] = 0;
  printf("\nЕто всички компоненти на свързаност: \n");
  comp = 0;
  for (i = 0; i < n; i++)
    if (!used[i]) {
      comp++;
      printf("{ ");
      DFS(i);
      printf("}\n");
    }
  if (1 == comp)
    printf("Графът е свързан.\n");
  else
    printf("Брой на свързаните компоненти в графа: %d \n", comp);
  return 0;
}
```

🖫 strcon1.c

Резултат от изпълнението на програмата:

```
{ 1 2 3 }
{ 4 5 6 }
Брой на свързаните компоненти в графа: 2
```

Задача за упражнение:

Да се реализира алгоритъм за намиране на свързаните компоненти на граф с обхождане в *ширина*.

5.6.2. Компоненти на силна свързаност в ориентиран граф

Ще припомним *дефиниция 5.10.*: ориентираният граф $G(V,E)$ се нарича *силно свързан*, ако съществува път както от i до j, така и от j до i, за всяко $i \neq j$, i, $j \in V$. Ако графът не е силно свързан, се интересуваме от всички компоненти на силна свързаност (всички максимални *силно свързани* подграфи на G).

Алгоритмите, които ще разгледаме, са отново адаптация на обхождането в дълбочина, но тук нещата не са толкова тривиални, както при неориентиран граф.

Алгоритъм 1

1) Избираме произволен връх $i \in V$.
2) Изпълняваме $DFS(i)$ и намираме множеството от върхове R, достижими от i.
3) Образуваме "обърнат" граф $G'(V,E')$: посоките на всички ребра в който са "обърнати", т.е. реброто $(j,k) \in E'$ т.с.т.к. $(k,j) \in E$.
4) Изпълняваме обхождане в дълбочина от върха i в G' — така намираме множеството от върхове Q, достижими от i в G' (и съответно които *достигат* i в G).
5) Сечението на R с Q дава една силно свързана компонента.
6) Изключваме тази компонента от графа и, ако има още върхове, повтаряме стъпка 1).

Описаният алгоритъм има сложност $\Theta(n\ .\ (n + m))$. Вторият алгоритъм, който ще изложим, е със сложност $\Theta(m+n)$.

Алгоритъм 2

Първата модификация на функцията за обхождане в дълбочина, която ще направим, се състои в номерация на върховете: всеки връх ще получи номер, който трябва да бъде по-голям от този на наследниците му при обхождането. Тази стратегия за номериране (на английски се нарича *postnum*) и се реализира, като се добави следният ред в функцията `DFS(i)`, *след* рекурсивното извикване:

```
postnum[i] = count++;,
```

където `count` е броячът на номерацията (колко върха сме обходили до момента).

Следва същинската част от алгоритъма:

1) Изпълняваме обхождане в дълбочина в G, като номерираме върховете съгласно стратегията *postnum*, описана по-горе.
2) Образуваме граф $G'(V,E')$, подобен на G: разликата е, че посоките на всички ребра в него са "обърнати", т.е. реброто $(j,i) \in E'$ тогава и само тогава, когато $(i,j) \in E$.
3) Изпълняваме обхождане в дълбочина в графа G'. Започваме търсенето от този връх w, за който `postnum[w]` има най-голяма стойност. Всички върхове, достигнати при обхождането, принадлежат на една и съща силно свързана компонента. Ако търсенето не успее да обходи всички върхове, то графът не е силно свързан и избираме като втори начален връх този измежду непосетените върхове, за който стойността на `postnum` е най-голяма. Стъпката се повтаря, докато обходим всички върхове.

Следва изходният код на Си. Най-интересният момент в него е подходът, който използваме за реализация на стъпка 2) от алгоритъма. Тъй като би било крайно неефективно да използваме още една матрица на съседство за G', сме използвали втора функция за обхождане в дълбочина (`backDFS`), която гледа на ребрата от графа G в "обърнат" вид, все едно, че работи с G'. Забележете, че този подход не би бил възможен при всяко представяне на графа: например, при списък на наследниците.

```
#include <stdio.h>

const N = 10; /* Брой върхове на граф и матрица на съседство */
```

```
int A[N][N] = {
  { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 1, 1, 0, 0, 0, 0, 0, 0 },
  { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 1, 0, 1, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 },
  { 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
  { 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 }
};

int used[N];
int postnum[N], count = 0;

/* Обхождане в дълбочина със запазване на номерацията */
void DFS(int i)
{ int j;
  used[i] = 1;
  for (j = 0; j < N; j++)
    if (!used[j] && A[i][j]) DFS(j);
  postnum[i] = count++;
}

/* Обхождане в дълбочина на графа G’ */
void backDFS(int i)
{ int j;
  printf("%d ", i + 1);
  count++;
  used[i] = 1;
  for (j = 0; j < N; j++)
    if (!used[j] && A[j][i]) {
      backDFS(j);
    }
}

/* Намира силно свързаните компоненти на графа */
void strongComponents(void)
{ int i;
  for (i = 0; i < N; i++) used[i] = 0;
  while (count < N - 1) {
    for (i = 0; i < N; i++)
      if (!used[i]) DFS(i);
  }
  for (i = 0; i < N; i++) used[i] = 0;
  count = 0;
  while (count < N - 1) {
    int max = -1, maxv = -1;
    for (i = 0; i < N; i++)
      if (!used[i] && postnum[i] > max) {
        max = postnum[i];
        maxv = i;
      }
    printf("{ ");
    backDFS(maxv);
    printf("}\n");
  }
}
```

```
int main(void) {
  printf("Силно свързаните компоненти в графа са:\n");
  strongComponents();
}
```

🖫 strconn.c

Резултат от изпълнението на програмата:

```
Силно свързаните компоненти в графа са:
{ 1 3 2 6 5 4 }
{ 7 10 9 8 }
```

Задачи за упражнение:

1. Да се докаже, че сложността на алгоритъм 1 е $\Theta(n\ .\ (n + m))$.

Упътване: Да се докаже, че най-лошият случай за алгоритъма е ацикличен ориентиран граф с точно $m = n.(n-1)/2$ ребра. В този случай компонентите на силна свързаност са n, а обхождането в дълбочина за всяка компонента има сложност $\Theta(m + n)$.

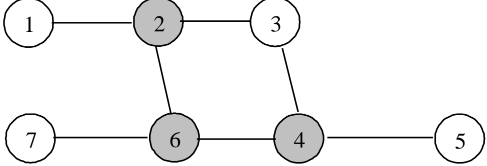
2. Какъв ще бъде алгоритъмът за проверка дали даден ориентиран граф е *слабо* свързан и за намиране на всички негови *слабо свързани компоненти*? Забележете, че за разлика от силната свързаност, която разбива графа на непресичащи се компоненти, в две различни слабо свързани компоненти могат да участват едни и същи върхове. Каква е сложността на Вашия алгоритъм?

5.6.3. Разделящи точки в неориентиран граф. Двусвързаност

Дефиниция 5.26. Нека е даден свързан неориентиран граф. *Разделяща точка* в графа се нарича връх, след чието премахване (както и на всички ребра, инцидентни с него) графът престава да бъде свързан.

Дефиниция 5.27. Свързан неориентиран граф се нарича *двусвързан*, ако няма разделяща точка. Това означава още, че той остава свързан след премахването на произволен *единствен* негов връх.

За свързания граф от *фигура 5.6.3.* разделящи върховете *2*, *4* и *6*. Така например, ако премахнем върха *6*, ще получим две компоненти на свързаност: {*1*, *2*, *3*, *4*, *5*} и {*7*}.



Фигура 5.6.3. Разделящи точки в граф.

Ако използваме директно дефиницията, можем да постигнем сложност $\Theta(n.(m+n))$: ще използваме алгоритъма за проверка дали неориентиран граф е свързан. Всеки връх, след чието премахване графът престава да бъде свързан, е разделяща точка.

Горният алгоритъм е лесен за реализация, но не е най-ефективният.

Твърдение 1. Върхът $k \in V$ е разделяща точка тогава и само тогава, когато съществуват такива два други върха i и j ($i, j \in V$), че *всеки* път между тях минава през k.

Твърдение 2. Нека $G(V, E)$ е свързан неориентиран граф, а $T(V, D)$ е покриващо дърво на G, построено при обхождане в дълбочина. Върхът $k \in V$ ще бъде разделяща точка тогава и само тогава, когато съществуват върхове $i, j \in V$ такива, че да са изпълнени едновременно условията:

1) $(k, i) \in D$
2) $j \neq k$
3) j не е наследник на i в T
4) `lowest[i]` ≥ `prenum[k]`, където:

• `prenum[k]` е номерът, който е получил върхът k, при номерация *prenum* на върховете (т.е. номерата на обхожданите върхове се получават в началото на функцията за обхождане, *преди* рекурсивното извикване)

• `lowest[i]` се получава като минимум от:
  а) `prenum[i]`
  б) `prenum[w]` за всеки връх w такъв, че $(i, w) \in E$ и $(i, w) \notin D$.
  в) `lowest[w]` за всеки наследник w на i в D.

На базата на *твърдение 2*, ще построим алгоритъм за намиране на разделящи точки в неориентиран граф:

Алгоритъм

1) Изпълняваме обхождане в дълбочина от произволен връх на G. Нека T е покриващо дърво, получено като резултат. За всеки връх i с `prenum[i]` сме означили *prenum* номера на върха, получен при обхождането.
2) Обхождаме дървото T в ЛДК (*виж 2.2.*). За всеки връх i, който разгледаме, изчисляваме `lowest[i]` по начина, описан в *твърдение 2*.
3) Разделящите точки се определят по следния начин:
   3.1) Коренът на дървото T, е разделяща точка тогава и само тогава, когато има повече от един наследник.
   3.2) Връх i, различен от корена на T, е разделяща точка тогава и само тогава, когато i има *пряк* наследник x, за който е изпълнено `lowest[x] ≥ prenum[i]`.

Сложността на този алгоритъм, при представяне на графа със списък на наследниците, е $\Theta(n+m)$. (*Защо?*) Нашата реализация обаче е $\Theta(n^2)$.

В предложената по-долу реализация сме използвали представяне на графа с матрица на съседство. За покриващото дърво T няма да използваме допълнителен масив: за да означим, че дадено ребро участва в T, ще използваме допълнителната стойност `+2` в матрицата на съседство `A[][]`.

```
#include <stdio.h>
#define MAXN 150 /* Максимален брой върхове в графа */
/* Брой върхове в графа */
const unsigned n = 7;
/* Матрица на съседство на графа */
char A[MAXN][MAXN] = {
  { 0, 1, 0, 0, 0, 0, 0 },
  { 1, 0, 1, 0, 0, 1, 0 },
  { 0, 1, 0, 1, 0, 1, 0 },
  { 0, 0, 1, 0, 1, 1, 0 },
  { 0, 0, 0, 1, 0, 0, 0 },
  { 0, 1, 1, 1, 0, 0, 1 },
  { 0, 0, 0, 0, 0, 1, 0 }
};

unsigned prenum[MAXN], lowest[MAXN], cN;
unsigned min(unsigned a, unsigned b) { return (a < b) ? a : b; }
```

```
void DFS(unsigned i)
{ unsigned j;
  prenum[i] = ++cN;
  for (j = 0; j < n; j++)
    if (A[i][j] && !prenum[j]) {
      A[i][j] = 2;  /* строим покриващо дърво T */
      DFS(j);
    }
}

/* Обхождане на дървото в postorder */
void postOrder(unsigned i)
{ unsigned j;
  for (j = 0; j < n; j++)
    if (2 == A[i][j]) postOrder(j);
  lowest[i] = prenum[i];
  for (j = 0; j < n; j++)
    if (1 == A[i][j]) lowest[i] = min(lowest[i], prenum[j]);
  for (j = 0; j < n; j++)
    if (2 == A[i][j]) lowest[i] = min(lowest[i], lowest[j]);
}

void findArticPoints(void)
{ unsigned artPoints[MAXN], i, j, count;
  for (i = 0; i < n; i++) {
    prenum[i] = 0; lowest[i] = 0; artPoints[i] = 0;
  }
  cN = 0;
  DFS(0);
  for (i = 0; i < n; i++)
    if (0 == prenum[i]) {
      printf("Графът не е свързан - \n");
      return;
    }
  postOrder(0);
  /* проверяваме 3.1) */
  count = 0;
  for (i = 0; i < n; i++)
    if (2 == A[0][i]) count++;
  if (count > 1) artPoints[0] = 1;
  /* прилагаме стъпка 3.2) */
  for (i = 1; i < n; i++) {
    for (j = 0; j < n; j++)
      if (2 == A[i][j] && lowest[j] >= prenum[i]) break;
    if (j < n) artPoints[i] = 1;
  }

  printf("Разделящите точки в графа са:\n");
  for (i = 0; i < n; i++)
    if (artPoints[i]) printf("%u ", i + 1);
  printf("\n");
}

int main(void) {
  findArticPoints();
  return 0;
}
```
🖫 artic.c

Резултат от изпълнението на програмата:

```
Разделящите точки в графа са:
2 4 6
```

Задача за упражнение:

Да се докажат твърдения 1 и 2.

5.6.4. *k*-свързаност на неориентиран граф

Дефиниция 5.28. Неориентиран граф се нарича *k-свързан по отношение на върховете*, ако е свързан и остава свързан след премахването на произволни k–1 негови върха.

Дефиниция 5.29. Неориентиран граф се нарича *k-свързан по отношение на ребрата*, ако е свързан и остава свързан след премахването на произволни k–1 ребра.

Нека е даден граф $G(V, E)$. Ще разгледаме следните задачи:

- Да се намери минималното k, за което G не е *k-свързан по отношение на върховете*.
- Да се намери минималното k, за което G не е *k-свързан по отношение на ребрата*.

Ще скицираме, как горните две задачи могат да се решат, като се използва задачата за максимален поток в граф.

k-свързаност по отношение на ребрата

Големината на максималния поток (при капацитет на всички ребра единица) между връх-източник i и консуматор j е равна точно на минималния брой ребра, които са необходими, за да разделят двата върха така, че да останат в различни компоненти на свързаност (Защо?). Така, минималното k е равно на минималната стойност от максималните потоци между произволен връх i и останалите n–1 върха от графа.

k-свързаност по отношение на върховете

Теорема. (*Менгер*) Граф е k-свързан по отношение на върховете тогава и само тогава, когато всяка двойка върхове е свързана чрез k непресичащи се (без съвпадащи междинни върхове) пътя.

Алгоритъм за проверка на k-свързаност по отношение на върховете:

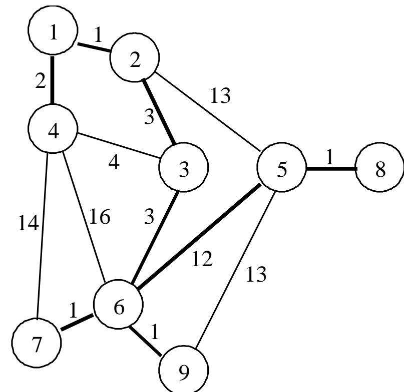
1) Построяваме граф $G'(V',E')$ със следното свойство: на всяко множество непресичащи се по отношение на върховете пътища в G да отговаря (биективно) множество от непресичащи се по отношение на ребрата пътища в G'.
   Построението се извършва по следния начин:
   - На всеки връх $i \in V$ ще отговарят два върха $i_1 \in V'$ и $i_2 \in V'$ и реброто $(i_1, i_2) \in E'$.
   - На всяко ребро $(i, j) \in E$ ще отговарят ребрата $(i_1, j_2) \in E'$ и $(i_2, j_1) \in E'$.
2) Чрез максимален поток в G' проверяваме дали съществуват k непресичащи се пътя (по отношение на участващите в тях ребра):
   а) Ако съществуват, следва че графът е k-свързан: свойството на непресичащите се пътища се прехвърля от ребра в G' във върхове в G. (*Защо?*) Прилагаме теоремата на Менгер за непресичащите се пътища върху G и получаваме търсения резултат.
   б) Ако не съществуват, то графът не е k-свързан.

Задачи за упражнение:

1. Да се докаже, че свойството на непресичащите се пътища се прехвърля от ребра в G' във върхове в G.

2. Да се модифицира алгоритъмът за проверка дали граф е k-свързан по отношение на върховете в такъв за намиране на максималната k-свързаност на граф.

5.7. Оптимални подмножества и центрове



Фигура 5.7. Минимално покриващо дърво в граф.

5.7.1. Минимално покриващо дърво

Да си представим група острови, които искаме да свържем с мостове по такъв начин, че да може да се стигне от всеки остров до всеки друг в групата. Тъй като строенето на мостовете е свързано с определени разходи, целта, която ще си поставим, ще бъде множеството от мостове, което строим, да има минимална цена. Ще приемем, че за всяка двойка острови знаем дали може да се свърже директно с мост и колко струва той.

Изказана в термините от теорията на графите, горната задача ще изглежда така: Даден е неориентиран граф $G(V,E)$, в който върховете от множеството V са островите, а ребро (i,j) съществува т.с.т.к е възможно да се построи мост между островите i и j. Теглото $f(i,j)$ определя евентуалната му цена. Търси се покриващо (обхващащо) дърво $T(V,D)$ на G с минимална сума от теглата на участващите в D ребра.

Покриващо дърво T с такова свойство се нарича *минимално покриващо дърво* (*МПД*). Ще разгледаме два известни *евристични* (*виж глава 9*) алгоритъма, които решават дадената задача.

- алгоритъм на Крускал

Разглеждаме претеглен неориентиран граф $G(V, E)$.

Алгоритъм на Крускал за намиране на МПД

1) Създаваме n множества, като в i-тото множество поставяме i-тия връх от графа.
2) Сортираме ребрата на графа във възходящ ред (по теглата им).
3) Създаваме празно дърво $T(V,\varnothing)$. След приключване на алгоритъма T ще бъде търсеното покриващо дърво.
4) Последователно (n–1 пъти) добавяме ребро $(i,j)\in E$ към T, така че да бъде изпълнено:
   - теглото $f(i, j)$ да бъде възможно най-малко (разполагаме със сортиран по теглата списък на ребрата от графа)
   - върховете i и j да се намират в различни множества

   След всяко добавено ребро (i,j) обединяваме множествата, в които се намират i и j.

Забележете, че всяко ребро се разглежда *най-много веднъж*. Наистина, ако на някоя стъпка i и j се намират в едно и също множество, те ще останат "заедно" докрай: алгоритъмът само *обединява*, но не разбива множества. Така, ако дадено ребро веднъж се окаже "непригодно", то ще бъде такова и на всяка следващата итерация от стъпка 4). Процесът на изграждане на дървото приключва при успешно включване на $n-1$ ребра (всяко следващо ще затвори цикъл, *защо?*).

Да разгледаме как се прилага алгоритъмът за примера на *фигура 5.7*. Първоначално деветте върха на графа се намират в девет различни множества (компоненти):

$$\{1\}\ \{2\}\ \{3\}\ \{4\}\ \{5\}\ \{6\}\ \{7\}\ \{8\}\ \{9\}$$

Сортираният списък от ребрата, получен на стъпка 2), е следният:

$$(1, 2) = 1; (5, 8) = 1; (7, 6) = 1; (6, 9) = 1; (1, 4) = 2; (2, 3) = 3; (3, 6) = 3;$$
$$(3, 4) = 4; (5, 6) = 12; (5, 9) = 13; (2, 5) = 13; (4, 7) = 14; (4, 6) = 14;$$

Избираме първото ребро (1, 2) и обединяваме множествата, в които се намират върховете 1 и 2. Получаваме $\{1, 2\}\ \{3\}\ \{4\}\ \{5\}\ \{6\}\ \{7\}\ \{8\}\ \{9\}$.

Избираме следващото по големина ребро (5, 8). Множествата са: $\{1, 2\}\ \{3\}\ \{4\}\ \{5, 8\}\ \{6\}\ \{7\}\ \{9\}$.

По-нататък към покриващото дърво последователно включваме (7, 6), (6, 9), (1, 4), (2, 3), (3, 6) и получаваме: $\{1, 2, 3, 4, 6, 7, 9\}\{5, 8\}$.

Пропускаме реброто (3, 4), тъй като върховете 3 и 4 са вече в едно и също множество. Вземаме следващото ребро (5, 6), при което всички върхове на графа попадат в едно общо множество. Минималното покриващо дърво е построено и се състои от ребрата (1, 2), (1, 4), (2, 3), (3, 6), (6, 7), (6, 9), (6, 5), (5, 8).

Да пресметнем сложността на алгоритъма. Стъпките, необходими за инициализация на множествата са $\Theta(n)$. По-нататък сложността се определя от две неща. Първото е, как ще извършим сортирането на стъпка 2). Ако използваме пирамидално сортиране (*виж 3.1.9.*), сложността на сортирането в най-лошия случай ще бъде $\Theta(m.\log_2 m)$. Вторият определящ сложността фактор е как ще бъде реализирана проверката дали два върха принадлежат на едно и също множество (и обединяването на множества при всеки избор на ребро на стъпка 4). Ще използваме компоненти на свързаност (за тях стана дума при разглеждането на начините за представяне на граф – *5.2.*). За всяка компонента на свързаност строим дърво с корен произволен неин връх. Когато добавяме ново ребро към графа, ще обединяваме двете дървета, съответстващи на двете компоненти на свързаност (например добавяне на ребро, при което коренът на едното дърво да стане наследник на корена на другото). Проверката дали два върха се намират в една и съща компонента на свързаност се извършва, като се сравнят корените на дърветата, които отговарят на тези компоненти (т.е. дали тези корени са един и същ връх или — не). Достигането на корена на дървото (тръгвайки от листо) има сложност $\Theta(\log_2 n)$, и тъй като извършваме $n-1$ стъпки, общата сложност на алгоритъма ще бъде: $\Theta(m.\log_2 m + n.\log_2 n)$.

В реализацията на Си графът е представен със списък на ребрата в масив `S[]` с `M` елемента (колкото е броят на ребрата в графа). `S[i].x`, `S[i].y`, `S[i].weight` показват че реброто (`S[i].x`, `S[i].y`) има тегло `S[i].weight`. С функцията `sort(A,0,M)` сортираме ребрата на графа по тяхното тегло във възходящ ред. Реализацията на операциите с компонентите на свързаност и съответстващите им дървета ще извършим по следния начин: Ще въведем масив `prev[]`, в който `prev[i]` е връх от същото множество, на което принадлежи i, и е негов предшественик (в това множество, не в D!). Ако върхът i е корен на съответното дърво, то `prev[i] == 0`. Обединяването на две множества става, като се изпълни `prev[r2] = r1`, където `r1` и `r2` са корените на дърветата, които обединяваме. Определянето на корена на дърво, в което се намира връх i, става по следния начин:

```
root = i;
while (NO_PARENT != prev[root]) root = prev[root];
```

Като допълнение, след всяко подобно търсене, ще извършваме *свиване на пътя*. То се прави с цел намаляване на общата дълбочина на дървото за сметка на едно допълнително обхождане от i до корена. След всяко търсене, елементите по пътя от i до корена ще стават наследници на корена:

```
while (i != root) {
  savei = i;
  i = prev[i];
  prev[savei] = root;
}
```

При по-голям брой елементи бихме могли да получим още по-голямо подобрение при съвместно използване на балансиране на дървото и свиване на пътя. Балансирането обаче изисква допълнителен обем памет от порядъка на общия брой n на елементите на множествата и при по-малки стойности на n не дава особени преимущества пред свиването на пътя без балансиране. Ето защо при реализацията на алгоритъма на Крускал ще използваме горната функция за свиване на пътя, без балансиране. Очевидно броят на операциите на търсене и обединение е пропорционален на общия брой n на елементите на множествата. Сложността при различните подходи в този случай може да се види от *таблица 5.7.1.*, където с $\log^* n$ сме означили броя на прилаганията на функцията логаритъм към n, необходими, за да се получи 0.

Брой елементарни операции на търсене и обединение	Балансиране на дървото	Свиване на пътя
n^2	НЕ	НЕ
$n.\log n$	ДА	НЕ
$n.\log^* n$	ДА	ДА

Таблица 5.7.1. Брой елементарни операции на търсене и обединение на дървета при n елемента при различни стратегии.

Примерните входни данни, използвани в изходния код по-долу, са за графа от *фигура 5.7*.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 200
#define NO_PARENT (unsigned)(-1)

/* Максимален брой ребра в графа */
#define MAXM 2000

const unsigned n = 9;  /* Брой върхове в графа */
const unsigned m = 14; /* Брой ребра в графа */
struct arc {
  unsigned i, j;
  int f;
};

struct arc S[MAXM] = {  /* Списък от ребрата на графа и техните тегла */
  { 1, 2,  1},
  { 1, 4,  2},
  { 2, 3,  3},
  { 2, 5, 13},
  { 3, 4,  4},
  { 3, 6,  3},
  { 4, 6, 16},
  { 4, 7, 14},
```

```
  { 5, 6, 12},
  { 5, 8,  1},
  { 5, 9, 13},
  { 6, 7,  1},
  { 6, 9,  1}
};

int prev[MAXN + 1];

int getRoot(int i)
{ int root, savei;
  /* намиране на корена на дървото */
  root = i;
  while (NO_PARENT != prev[root]) root = prev[root];
  /* свиване на пътя */
  while (i != root) {
    savei = i;
    i = prev[i];
    prev[savei] = root;
  }
  return root;
}

void kruskal(void)
{ int MST = 0;
  unsigned i, j;
  /* сортира списъка с ребрата в нарастващ ред по тяхното тегло */
  sort(S, 0, m);

  printf("Ребрата, които участват в минималното покриващо дърво:\n");
  for (i = 0; i < m; i++) {
    int r1 = getRoot(S[i].i);
    int r2 = getRoot(S[i].j);
    if (r1 != r2) {
      printf("(%u,%u) ", S[i].i, S[i].j);
      MST += S[i].f;
      prev[r2] = r1;
    }
  }
  printf("\nЦената на минималното покриващо дърво е %d.\n", MST);
}

int main(void) {
  unsigned i;
  for (i = 0; i < n + 1; i++) prev[i] = NO_PARENT;
  kruskal();
  return 0;
}
```

🖫 kruskal.c

Резултат от изпълнението на програмата:

```
Ребрата, които участват в минималното покриващо дърво:
(1,2) (5,8) (6,7) (6,9) (1,4) (2,3) (3,6) (5,6)
Цената на минималното покриващо дърво е 24.
```

Задачи за упражнение:

1. Да се докаже, че алгоритъмът на Крускал работи правилно.

2. Работи ли алгоритъмът на Крускал за *несвързан* граф? Ако да, какъв е критерият за прекратяване добавянето на нови ребра?

3. Работи ли алгоритъмът на Крускал за *ориентиран* граф?

- алгоритъм на Прим

Алтернатива на алгоритъма на Крускал за намиране на *МПД* е алгоритъмът на *Прим*.

Алгоритъм на Прим

1) Започваме строенето на *МПД* от произволен връх s: в началото дървото ще бъде $T(H,D)$, $H = \{s\}$, $D = \varnothing$.
2) Повтаряме $n-1$ пъти:
   2.1) Избираме реброто $(i, j)\in E$, такова че:
   - $i\in H$, $j\in V\backslash H$;
   - $f(i, j)$ е минимално.

   2.2) Добавяме върха j към H и реброто (i, j) към D.

Ето как се прилага алгоритъмът за примера от *фигура 5.7.*: Избираме произволен начален връх, например *1*. Минималното ребро, което го свързва с връх, неучастващ в H, е (*1*, *2*). Добавяме върха *2* към H, а реброто (1,2) — към D. Следващото минимално ребро, което свързва връх от H с връх, непринадлежащ на H, е (*1*, *4*). По нататък избраните ребра ще бъдат (*2*, *3*), (*3*, *6*), (*6*, *7*), (*6*, *9*), (*6*, *5*) и (*5*, *8*), с което минималното покриващо дърво е построено.

Нека реализираме описания алгоритъм. Стъпката, която се нуждае от допълнително конкретизиране, е 2.1). Ако приложим директен подход, при който последователно проверяваме и запазваме минималното разстояние между двойките от върхове, принадлежащи и непринадлежащи на H, ще получим алгоритъм със сложност $\Theta(n^3)$. За да избегнем n-пъти повече проверки, ще въведем допълнителен масив `T[]` — в него ще пазим най-кратките разстояния до неразгледани (непринадлежащи на H) върхове, т.е. разстоянието относно дървото до всеки връх извън дървото:

- Ако приемем, че стартовият връх е s, в началото инициализираме:
  - `T[j] = A[s][j]`, $j = 1,2,..., n.$, за всяко $(s, j)\in E$
  - `T[j] = MAX_VALUE`, в противен случай.
- На всяка стъпка:
  - Намираме това j, за което `T[j]` има минимална стойност.
  - За всяко $k\notin H$ изпълняваме `T[k] = min(T[k], A[j][k])`.

Следва изходният код на програмата, реализираща описания алгоритъм.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 150
#define MAX_VALUE 10000

/* Брой върхове в графа */
const unsigned n = 9;
/* Матрица на теглата на графа */
const int A[MAXN][MAXN] = {
  { 0,  1, 0,  2,  0,  0,  0, 0,  0 },
  { 1,  0, 3,  0, 13,  0,  0, 0,  0 },
  { 0,  3, 0,  4,  0,  3,  0, 0,  0 },
  { 2,  0, 4,  0,  0, 16, 14, 0,  0 },
  { 0, 13, 0,  0,  0, 12,  0, 1, 13 },
  { 0,  0, 3, 16, 12,  0,  1, 0,  1 },
```

```
  { 0,  0, 0, 14,  0,  1,  0, 0,  0 },
  { 0,  0, 0,  0,  1,  0,  0, 0,  0 },
  { 0,  0, 0,  0, 13,  1,  0, 0,  0 }
};

char used[MAXN];
unsigned prev[MAXN];
int T[MAXN];

void prim(void)
{ int MST = 0;  /* тук трупаме цената на минималното покриващо дърво */
  unsigned i, k;
  /* инициализация */
  for (i = 0; i < n; i++) { used[i] = 0; prev[i] = 0; }
  used[0] = 1;  /* избираме произволен начален връх */
  for (i = 0; i < n; i++)
    T[i] = (A[0][i]) ? A[0][i] : MAX_VALUE;
  for (k = 0; k < n - 1; k++) {
    /* търсене на следващо минимално ребро */
    int minDist = MAX_VALUE, j = -1;
    for (i = 0; i < n; i++)
      if (!used[i])
        if (T[i] < minDist) {
          minDist = T[i];
          j = i;
        }
    used[j] = 1;
    printf("(%u,%u) ", prev[j] + 1, j + 1);
    MST += minDist;
    for (i = 0; i < n; i++)
      if (!used[i] && A[j][i]) {
        if (T[i] > A[j][i]) {
          T[i] = A[j][i];
          /* запазване на предшественика
             за евентуално отпечатване на следващо минимално ребро */
          prev[i] = j;
        }
      }
  }

  printf("\nЦената на минималното покриващо дърво е %d.\n", MST);
  printf("\n");
}

int main(void) {
  prim();
  return 0;
}
```

🖫 prim.c

Резултат от изпълнението на програмата:

```
(1,2) (1,4) (2,3) (3,6) (6,7) (6,9) (6,5) (5,8)
Цената на минималното покриващо дърво е 24.
```

Лесно се забелязва, че сложността на така реализирания алгоритъм на Прим е квадратична по броя на върховете на графа (*Защо?*).

При по-внимателно подбиране на структурите от данни (например, ако се използва пирамидална структура [*Cormen, Leiserson, Rivest-1997*]), сложността на алгоритъма на Прим може да достигне до $\Theta(m + n.\log_2 n)$.

За специалния случай на разредени графи, т.е. когато $m \in O(n)$, съществува още по-ефективна модификация на алгоритъма на Прим, със сложност $\Theta(m.\beta(m, n))$ и дори $\Theta(m.\log_2(\beta(m,n)))$, където ([*Cormen, Leiserson, Rivest-1997*][*Fredman,Tarjan-1987*]):

$$\beta(m,n) = \min i, \textit{такова че}: \underbrace{\log(\log(\ldots\log(n)\ldots))}_{i \text{ пъти}} < \frac{m}{n}$$

Задачи за упражнение:

1. Да се докаже, че алгоритъмът на Прим работи правилно.
2. Работи ли алгоритъмът на Прим за *несвързан* граф?
3. Работи ли алгоритъмът на Прим за *ориентиран* граф?
4. Какви оптимизации могат да се приложат върху алгоритмите на Прим и Крускал за случая, когато теглата на ребрата на графа са в интервала $[1, n]$?

- частично минимално покриващо дърво

Ще разгледаме модификация на задачата за построяване на *МПД*.

Дефиниция. 5.30. Даден е граф $G(V, E)$. За дадено k, $k \leq n$ *частично* минимално покриващо дърво $T_k(V_k, D)$ се нарича дърво, съдържащо *точно* k върха, такова че:

- $V_k \subseteq V$.
- $D \subseteq E$.
- сумата от теглата на ребрата, участващи в D, е минимална.

Алгоритъм за намиране на частично МПД:

За всеки връх от графа ще опитваме да построим съответно минимално покриващо дърво по алгоритъма на Прим, но съдържащо точно k върха. Ще прекъсваме алгоритъма в момента, в който вече са добавени k–1 ребра (т.е. са изпълнени k–1 стъпки). По този начин ще получим n дървета (по едно за всеки връх), измежду които ще изберем това с най-малка цена.

Реализацията на адаптацията на алгоритъма на Прим предоставяме на читателя.

Задачи за упражнение:

1. Да се обоснове предложеният алгоритъм за построяване на частично минимално покриващо дърво на базата на алгоритъма на Прим.
2. Да се реализира предложената модификация на алгоритъма на Прим.

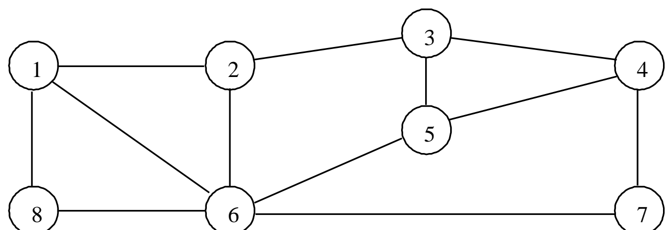
5.7.2. Независими множества

Ще припомним две дефиниции от началото на главата: Един граф $G(V, E)$ се нарича *пълен*, ако съществува ребро (i, j), за всички $i, j \in V$. *Кликово число* се нарича броят върхове в максималния пълен подграф $G'(V', E')$ на G. Обратното понятие на пълен граф е *празен* граф — несъдържащ нито едно ребро. Ще разгледаме две задачи и ще дефинираме понятие, обратно на кликово число.

Задача 1: Дадена е група от n души, в която всеки двама или са, или не са *приятели*. Да се изберат максимален брой така, че *всички* в избраната група да бъдат приятели.

Задача 2: Дадена е група от n души, в която всеки двама или са, или не са врагове. Да се изберат максимален брой хора, така че *никои* двама души от избраната група да не бъдат врагове.

Ще решим втората задача. Първата може да се реши с помощта на втората (оставяме като упражнение на читателя да съобрази как, след като се запознае с материала по-нататък).



Фигура 5.7.2. Неориентиран граф.

Дефиниция 5.30. Даден е неориентиран граф $G(V, E)$. Множеството H, $H \subseteq V$ се нарича *независимо множество от върхове* (или *антиклика*), ако не съдържа съседни върхове.

Дефиниция 5.31. Броят на елементите на независимото множество с максимален брой върхове се нарича *число на независимост* на G.

Казано с други думи, числото на независимост на неориентиран граф е броят върхове на максималния му по брой върхове празен подграф. Например, за графа от *фигура 5.7.2*. независими множества са $\{7, 8, 2\}$, $\{3, 1\}$, $\{7, 8, 2, 5\}$. Числото на независимост на графа е 4.

- максимални независими множества

Дефиниция 5.32. Едно независимо множество H в неориентиран граф се нарича *максимално*, ако не съществува друго независимо множество H', което да съдържа H като собствено подмножество.

В графа от *фигура 5.7.2* едно максимално независимо множество е $H_1 = \{7, 8, 2, 5\}$, докато $H_2 = \{7, 8, 2\}$ не е, тъй като се съдържа в H_1. Максимални независимите множества са още $\{1, 3, 7\}$ и $\{4, 6\}$.

Алгоритъм за намиране на всички максимални независими множества

Нека разглеждаме граф $G(V, E)$. Нека първо разгледаме алгоритъм за намиране на *едно* максимално независимо множество T:

1) Нека S и T в началото са празни множества.
2) Избираме *произволен* връх от графа $i \in V$, $i \notin S \cup T$.
3) Добавяме i в T. В S добавяме всички върхове, съседни на i.
4) Повтаряме стъпки 2) и 3), докато достигнем положение, при което всеки връх от графа е включен или в S, или в T, т.е. $S \cup T = V$. Тогава T е максимално независимо множество.

За да намерим *всички* независими множества, ще използваме рекурсия. На стъпка 2) от алгоритъма няма да избираме *произволен* връх, а ще изследваме какво ще се случи при добавяне на *всеки* свободен връх от графа, т.е. непринадлежащ на S и T. Това ще осъществим чрез рекурсивната функция `maxSubSet(last)`:

```
maxSubSet(last) {
  if (S∪T = V) {
    Отпечатваме_намереното_максимално_независимо_множество;
    return;
  }
  for (всеки_връх_i | i∈V, i∉S, i∉T, i > last) {
    Добавяме_i_в_T;
    Добавяме_ в_S_съседните_на_i_върхове;
    maxSubSet(i);
```

```
      Изключваме_i_от_T;
      Възстановяваме_S_от_преди_рекурсивното_извикване;
    }
  }
```

В горния фрагмент `last` е последният добавен връх в максималното независимото множество, което строим. Избираме само тези върхове *i*, за които *i* > `last`. Това се налага, за да се избегне конструирането на съвпадащи независими множества (от вида {1, 3, 7} и {1, 7, 3} и т.н.).

Следва изходният код на програмата. Графът е представен с матрица на съседство, и входните данни са зададени като константи.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 200
/* Брой върхове в графа */
const unsigned n = 8;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
  { 0, 1, 0, 0, 0, 1, 0, 1 },
  { 1, 0, 1, 0, 0, 1, 0, 0 },
  { 0, 1, 0, 1, 1, 0, 0, 0 },
  { 0, 0, 1, 0, 1, 0, 1, 0 },
  { 0, 0, 1, 1, 0, 1, 0, 0 },
  { 1, 1, 0, 0, 1, 0, 1, 1 },
  { 0, 0, 0, 1, 0, 1, 0, 0 },
  { 1, 0, 0, 0, 0, 1, 0, 0 } };
unsigned S[MAXN], T[MAXN], sN, tN;

void print(void)
{ unsigned i;
  printf("{ ");
  for (i = 0; i < n; i++)
    if (T[i]) printf("%u ", i + 1);
  printf("} \n");
}

void maxSubSet(unsigned last)
{ unsigned i, j;
  if (sN + tN == n) {
    print();
    return;
  }
  for (i = last; i < n; i++) {
    if (!S[i] && !T[i]) {
      for (j = 0; j < n; j++)
        if (A[i][j] && !S[j]) {
          S[j] = last+1; sN++;
        }
      T[i] = 1; tN++;
      maxSubSet(i+1);     /* рекурсия */
      T[i] = 0; tN--;
      for (j = 0; j < n; j++)
        if (S[j] == last+1) { S[j] = 0; sN--; }
    }
  }
}

int main(void) {
```

```
    unsigned i;
    printf("Всички максимални независими множества в графа са:\n");
    sN = tN = 0;
    for (i = 0; i < n; i++) S[i] = T[i] = 0;
    maxSubSet(0);
    return 0;
}
```

🖫 maxindep.c

Резултат от изпълнението на програмата:

```
Всички максимални независими множества в графа са:
{ 1 3 7 }
{ 1 4 }
{ 1 5 7 }
{ 2 4 8 }
{ 2 5 7 8 }
{ 3 6 }
{ 3 7 8 }
{ 4 6 }
```

Задачи за упражнение:

1. Да се докаже, че предложеният алгоритъм за намиране на всички максимални независими множества работи правилно.
2. Да се предложи начин за решаване на задача 1.

5.7.3. Доминиращи множества

Дефиниция 5.33. Даден е ориентиран граф $G(V, E)$. *Доминиращо множество от върхове* се нарича множество S, такова че:

- $S \subseteq V$
- Всеки връх, който не принадлежи на S, е пряк наследник на някой връх от S.

Дефиниция 5.34. Едно доминиращо множество се нарича *минимално*, ако не съдържа в себе си друго доминиращо множество като собствено подмножество.

Нека разгледаме един пример, който илюстрира въведената терминология.

Задача: Дадени са няколко преводача, владеещи различни езици, които ще превеждат от български на съответните езици. В *таблица 5.7.3.* преводачите сме означили с буквите A, B, C, D, E, а за езиците, които владеят, сме поставили 1 на съответната позиция.

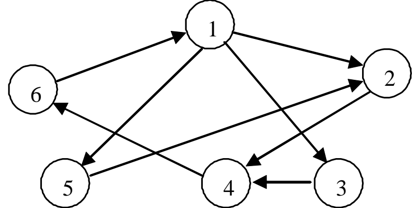
	A	*B*	*C*	*D*	*E*
Английски	1	0	1	1	0
Френски	1	1	0	0	0
Немски	0	1	0	0	0
Италиански	1	0	0	1	0
Руски	0	1	1	0	1
Китайски	0	0	0	1	1
Финикийски	0	0	1	0	0

Таблица 5.7.3. Преводачи.

Да се изберат минимален брой преводачи, така че да разполагаме с преводач за всеки от седемте езици, изброени по горе. Например, ако изберем B, C и D, исканото условие ще бъде изпълнено.

Тази и други задачи, които ще разгледаме по-късно, се решават, чрез намиране на минималните доминиращи множества в граф (В разгледания пример графът е двуделен. За такива графи съществуват ефективни алгоритми, които намират дори минималното по брой елементи множество с полиномиална сложност [*Манев-1996*].).

Доминиращите множества могат да бъдат повече от едно и с различен брой елементи. Ако означим с $p_1, p_2, ..., p_k$ броя на елементите на всяко от минималните доминиращи множества за даден граф, то $p = min\{p_1, p_2, ..., p_k\}$ се нарича *число на доминиране* за графа.



Фигура 5.7.3. Доминиращи множества в граф.

За графа на *фигура 5.7.3.* доминиращи множества са $\{1, 4\}$, $\{1, 4, 6\}$, $\{3, 5, 6\}$ и др., като множествата $\{1, 4\}$ и $\{3, 5, 6\}$ са минимални доминиращи множества. $\{1, 4, 6\}$ е доминиращо, но не е минимално (съдържа $\{1, 4\}$ в себе си). Числото на доминиране на графа е 2.

Дефиниция 5.35. Даден е ориентиран граф $G(V, E)$ и множество T, $T \subseteq V$. *Покритие* на множеството T се нарича множеството $P(T)$, съставено от всички върхове на T и всички върхове, съседни на някой връх от T.

Алгоритъм за намиране на всички минимални доминиращи множества

Нека T бъде празното множество. На всяка стъпка ще добавяме нов връх към T, така че след добавянето да остава изпълнено следното условие:

- да не съществува връх $i \in T$, такъв че $P(T \setminus \{i\}) \equiv P(T)$.

Ако на някоя стъпка $P(T) \equiv V$, то сме намерили едно минимално доминиращо множество.

За да намерим *всички* минимални доминиращи множества, ще строим T по *всички* възможни начини чрез пълно изчерпване.

В показаната по-долу реализация, функцията `ok()` проверява дали е изпълнено горното условие за някое множество T. Конструирането на T се осъществява по следната рекурсивна схема:

```
void findMinDom(int last) {
  if (P(T) == V) {
    Отпечатване_на_намереното_минимално_доминиращо_множество;
    return;
  }
  for (всеки_връх_i_от_графа, i >= last) {
    T = T∪{i};
    if (ok()) findMinDom(i+1);
    T = T\{i};
  }
}
```

Следва пълната реализация:

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 200
```

```
/* Брой върхове в графа */
const unsigned n = 6;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
  { 0, 1, 1, 0, 1, 0 },
  { 0, 0, 0, 1, 0, 0 },
  { 0, 0, 0, 1, 0, 0 },
  { 0, 0, 0, 0, 0, 1 },
  { 0, 1, 0, 0, 0, 0 },
  { 1, 0, 0, 0, 0, 0 }
};
unsigned cover[MAXN];
char T[MAXN];

void printSet(void)
{ unsigned i;
  printf("{ ");
  for (i = 0; i < n; i++)
    if (T[i])
      printf("%u ", i + 1);
  printf("}\n");
}

char ok(void)
{ unsigned i, j;
  for (i = 0; i < n; i++) if (T[i]) {
    /* проверка дали покритието се запазва при премахване на върха i */
    if (0 == cover[i]) continue;
    for (j = 0; j < n; j++) if (cover[j])
      if (!(cover[j] - A[i][j]) && !T[j])
        break; /* остава непокрит връх */
    if (j == n) return 0;
  }
  return 1;
}

void findMinDom(unsigned last)
{ unsigned i, j;
  /* проверява се дали е намерено решение */
  for (i = 0; i < n; i++)
    if (!cover[i] && !T[i]) break;
  if (i == n) { printSet(); return; }
  /* не - продължаваме да конструираме доминиращото множество */
  for (i = last; i < n; i++) {
    T[i] = 1;
    for (j = 0; j < n; j++) if (A[i][j]) cover[j]++;
    if (ok()) findMinDom(i + 1);
    for (j = 0; j < n; j++) if (A[i][j]) cover[j]--;
    T[i] = 0;
  }
}

int main(void) {
  unsigned i;
  printf("Минималните доминиращи множества в графа са: \n");
  for (i = 0; i < n; i++) { cover[i] = 0; T[i] = 0; }
  findMinDom(0);
  return 0;
```

```
}
```
🖫 mindom.c

Резултат от изпълнението на програмата:
```
Минималните доминиращи множества в графа са:
{ 1 2 6 }
{ 1 3 6 }
{ 1 4 }
{ 3 5 6 }
```

Задача за упражнение:

Да се докаже, че предложеният алгоритъм за намиране на всички минимални доминиращи множества работи правилно.

5.7.4. База

Задача: Дадена е мрежа от *n* компютъра и са известни каналите за връзка между тях. Да се определят минимален брой компютри (*база*) такива, че когато някаква информация трябва да се разпространи да може от базата да достигне всички компютри в мрежата.

На *фигура 5.7.4*. компютрите са представени като върхове на ориентиран граф. Една възможна база в графа е множеството {*4*, *7*, *9*}. Тя не е единствена: такива са например още {*5*, *7*, *9*}, {*7*, *8*, *9*} и др.

Дефиниция 5.36. Даден е ориентиран граф *G*(*V*, *E*). *База от върхове* (или само *база*) се нарича множество $T \subseteq V$ такова, че:

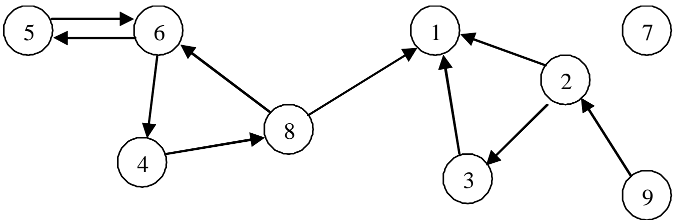
- всеки връх е достижим от някой връх от *T*
- *T* е *минимално* в смисъл, че не съществува собствено подмножество на *T*, отговарящо на първото условие за достижимост.

Твърдение 1. *T* е *база* тогава и само тогава, когато едновременно е изпълнено:

1) Всеки връх, неучастващ в *T*, е достижим от някой връх от *T*.
2) Не съществува връх $i \in T$, който да е достижим от друг връх от *T*.

Твърдение 2. В ориентиран ацикличен граф съществува единствена база *T* и тя се състои от всички върхове, чиято полустепен на входа е равна на нула (т.е. нямат предшественик).

От дефиницията (както и от твърдения 1 и 2) следва, че базите в граф може да са няколко, но те задължително имат равен брой върхове.



Фигура 5.7.4. Ориентиран граф.

Алгоритъм за намиране на база

Базови ще наричаме тези върхове, които участват в база. Ще въведем масив `base[]`, като `base[i] == 1`, когато *i* е базов, и `base[i] == 0`, в противен случай. В началото всички върхове маркираме като базови. Обхождаме в дълбочина от всеки връх *k*, за който `base[k] == 1`. При обхождането всички върхове *i* ($i \neq k$), през които преминем, ще маркираме с `base[i] = 0`. Така върховете, които останат маркирани като базови, ще образуват база в графа. Сложността на

описания алгоритъм е $\Theta(n+m)$, тъй като през всеки връх и всяко ребро се преминава най-много по веднъж (*Защо?*). Следва пълна реализация:

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 200
/* Брой върхове в графа */
const unsigned n = 9;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
  { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 1, 0, 1, 0, 0, 0, 0, 0, 0 },
  { 1, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 1, 0 },
  { 0, 0, 0, 0, 0, 1, 0, 0, 0 },
  { 0, 0, 0, 1, 1, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 1, 0, 0, 0, 0, 1, 0, 0, 0 },
  { 0, 1, 0, 0, 0, 0, 0, 0, 0 }
};

char used[MAXN], base[MAXN];

void DFS(unsigned i)
{ unsigned k;
  used[i] = 1;
  for (k = 0; k < n; k++)
    if (A[i][k] && !used[k]) {
      base[k] = 0;
      DFS(k);
    }
}

void solve(void) {
  unsigned i, j;
  for (i = 0; i < n; i++) base[i] = 1;
  for (i = 0; i < n; i++)
    if (base[i]) {
      for (j = 0; j < n; j++) used[j] = 0;
      DFS(i);
    }
}
void print() {
  unsigned i, count = 0;
  printf("Върховете, образуващи база в графа, са:\n");
  for (i = 0; i < n; i++)
    if (base[i]) { printf("%u ", i + 1); count++; }
  printf("\nБрой на върховете в базата: %u\n", count);
}

int main(void) {
  solve();
  print();
  return 0;
}
```

🖫 v-base.c

Резултат от изпълнението на програмата:

```
Върховете, образуващи база в графа, са:
4 7 9
Брой на върховете в базата: 3
```

Задачи за упражнение:

1. Да се докажат твърдения 1, 2.
2. Да се докаже, че предложеният алгоритъм за намиране на база работи правилно.

5.7.5. Център, радиус и диаметър

Широко приложение намират задачите за избор на един или фиксиран брой върхове на граф такива, че да бъде изпълнен определен критерий за оптималност. Например, нека гледаме на върховете на граф като на квартали, а свързващите ги ребра интерпретираме като директни пътища между тях. Търсим връх от графа, който да служи за аварийна служба (или пожарна, полиция, телефонна централа и т.н.) такъв, че разстоянието до най-отдалечения от него квартал да бъде минимално. Тази задача се нарича *оптимизация на най-лошия вариант*.

Възможно е да се търси и множество от p върха (няколко аварийни служби), като в този случай целта ще бъде да се минимизира най-дългото разстояние $S(i)$ от някой връх i до най-близката до него аварийна служба.

Дефиниция 5.37. Даден е свързан ориентиран граф $G(V, E)$. *Число на външно разделяне* $S_o(i)$ за даден връх $i \in V$ се нарича максималното $d(i, j)$, за всяко $j \in V$, където $d(i, j)$ е дължината на минималния път от i до j. Т.е. това число е най-краткият път до най-отдалечения връх от i. Аналогично, *число на вътрешно разделяне* $S_t(i)$ за върха i се нарича максимумът на минималното разстояние от някакъв връх $j \in V$ до върха i.

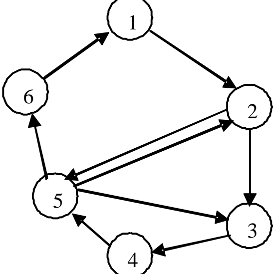
Дефиниция 5.38. Измежду всички върхове ще изберем такъв връх $i \in V$, за който $p_o = S_o(i)$ да бъде минимално. Върхът i се нарича *външен център* за графа G, а числото p_o се нарича *външен радиус* за графа. Аналогично, ще изберем такъв връх j, че $p_t = S_t(j)$ да бъде минимално. Този връх j се нарича *вътрешен център* за G, а p_t се нарича *вътрешен радиус* на графа.

Дефиниция 5.39. Да означим с $S_{ot}(i)$ минимума на $d(i, j) + d(j, i)$, пресметнат по всяко $j \in V$. Нека v е връх, за който $S_{ot}(v)$ е минимално. Тогава v се нарича *външно-вътрешен център* (или само *център*), а съответната стойност $S_{ot}(v)$ — *външно-вътрешен радиус* (или само *радиус*) на графа.

Забележете, че и в трите дефиниции въпросният връх може *да **не** бъде единствен*.

Пример за център в граф: Искаме да поставим полицейска служба по такъв начин, че ако S е минималното разстояние от нея до кой да е квартал плюс минималното разстоянието по обратния път, то S до най-отдалечения квартал да бъде минимално.

Преди да обобщим задачата (p-център и p-радиус в граф), ще покажем как се намира център в граф. Да разгледаме един конкретен пример (*виж фигура 5.7.5а.*).



	1	2	3	4	5	6
1	0	1	2	3	2	3
2	3	0	1	2	1	2
3	4	3	0	1	2	3
4	3	2	2	0	1	2
5	2	1	1	2	0	1
6	1	2	3	4	3	0

Център за графа: връх 2

$$\textit{Радиус} = \max_{i=1,2,\dots,n} \{d(2,i) + d(i,2) \mid i \in V\} = 4$$

Фигура 5.7.5а. Център и радиус в граф (всички ребра имат тегло 1).

Най-кратките пътища в графа (показани вдясно на *фигура 5.7.5а.*) сме намерили по алгоритъма на Флойд. След тази основна стъпка центърът и радиусът се определят, като се следва директно дефиницията по-горе. Сложността на алгоритъма е $\Theta(n^3)$. (*Защо?*)

В реализацията по-долу ориентирания граф сме представили чрез матрицата на теглата `A[][]`, като броят на върховете му е n. Входните данни се задават в началото на програмата като константи.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 150
#define MAX_VALUE 10000

/* Брой върхове в графа */
const unsigned n = 6;
/* Матрица на теглата на графа */
int A[MAXN][MAXN] = {
  { 0, 1, 0, 0, 0, 0 },
  { 0, 0, 1, 0, 1, 0 },
  { 0, 0, 0, 1, 0, 0 },
  { 0, 0, 0, 0, 1, 0 },
  { 0, 1, 0, 0, 0, 1 },
  { 1, 0, 0, 0, 0, 0 }
};
/* Намира дължината на минималния път между всяка двойка върхове */
void floyd(void)
{ unsigned i, j, k;

  /* стойностите 0 се променят на MAX_VALUE */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (A[i][j] == 0)
        A[i][j] = MAX_VALUE;

  /* Алгоритъм на Флойд */
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        if (A[i][j] > (A[i][k] + A[k][j]))
          A[i][j] = A[i][k] + A[k][j];

  for (i = 0; i < n; i++)
    A[i][i] = 0;
}

void findCenter(void)
{ unsigned i, j, center;
  int max, min;
  min = MAX_VALUE;
  /* Sot(Xi) = max { Vj [d(Xi, Xj)+d[Xj,Xi])] }, центърът е върхът X* за
   * който Sot(X*) е минимално
   */
  for (i = 0; i < n; i++) {
    max = A[i][0] + A[0][i];
    for (j = 0; j < n; j++)
      if ((i != j) && ((A[i][j] + A[j][i]) > max))
        max = (A[i][j] + A[j][i]);
```

```
        if (max < min) {
          min = max; center = i;
        }
      }
    printf("Центърът на графа е връх %u\n", center + 1);
    printf("Радиусът на графа е %u\n", min);
  }

  int main(void) {
    floyd();
    findCenter();
    return 0;
  }
```

a-center.c

Резултат от изпълнението на програмата:

```
Центърът на графа е връх 2
Радиусът на графа е 4
```

Задачи за упражнение:

1. Да се даде пример за граф с няколко възможни центъра.
2. Да се модифицира горната програма така, че да намира *всички* възможни центрове на графа.

- p-център и p-радиус

Обобщение на горната задача е да се изберат едновременно p на брой върха — центрове на графа. Искаме да минимизираме най-дългото разстояние от произволен връх i до най-близкия до него сред избраните. Да означим с $d(i,j)$ дължината на минималния път между върховете i и j.

Дефиниция 5.40. Даден е *ориентиран* граф $G(V,E)$ и множество $Q = \{v_{i1}, v_{i2}, \ldots, v_{ip} \mid v_{ik} \in V\}$. Въвеждаме означенията:

$$S(v,Q) = min\ \{\ d(v_k, v) + d(v, v_k) \mid v_k \in Q\},\ v \in V$$
$$S(Q) = max\ \{\ S(v,Q) \mid v \in V\ \}$$

Минималното $S(Q)$, пресметнато по всевъзможните p-елементни подмножества Q на V, се нарича *p-център* на графа.

Ето и дефиницията в по-простия случай, когато графът е неориентиран:

Дефиниция 5.41. Даден е *неориентиран* граф $G(V,E)$ и множество $Q = \{v_{i1}, v_{i2}, \ldots, v_{ip} \mid v_{ik} \in V\}$. Въвеждаме означенията:

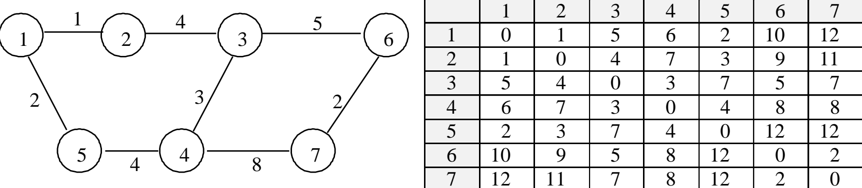
$$S(v,Q) = min\ \{\ d(v_k, v) \mid v_k \in Q\},\ v \in V$$
$$S(Q) = max\ \{\ S(v,Q) \mid v \in V\ \}$$

Минималното $S(Q)$, пресметнато по всевъзможните p-елементни подмножества Q на V, се нарича *p-център* на графа.

Алгоритъм за намиране на p-център

Ще разгледаме случая на *неориентиран* граф. Ако графът е ориентиран, промените са незначителни. Решението се състои в пълно изчерпване на възможните варианти: за всяко p-елементно подмножество Q, $Q \subseteq V$, ще пресмятаме разстоянието S до най-отдалечения връх. От всички възможности (общо C_n^p на брой) ще запазим тези, които дават минималната стойност на S.

В началото изчисляваме дължините на минималните пътища между всеки два върха, като използваме алгоритъма на Флойд. След като намерим матрицата с минималните пътища `A[][]`, ще генерираме всички възможни комбинации (всички възможни p-елементни подмножества Q на V) и за всяко ще изчисляваме разстоянието от него до най-отдалечения връх. Това не е особено трудно, след като имаме на разположение матрицата на Флойд. Един конкретен пример е показан на *фигура 5.7.5б.*



	1	2	3	4	5	6	7
1	0	1	5	6	2	10	12
2	1	0	4	7	3	9	11
3	5	4	0	3	7	5	7
4	6	7	3	0	4	8	8
5	2	3	7	4	0	12	12
6	10	9	5	8	12	0	2
7	12	11	7	8	12	2	0

Фигура 5.7.5б. Намиране на p-център.

Нека търсим 3-център, т.е. $p = 3$ в графа, показан на *фигура 5.7.5б*. Намерили сме матрицата на Флойд и разглеждаме подмножеството от върхове $\{2, 3, 6\}$:

$S = max$ { min { `A[i][2]`, `A[i][3]`, `A[i][6]` $| i = 1,2,...,7\} = max$ { 1, 0, 0, 3, 3, 0, 2 } = 3.

По-нататък, проверявайки всички останали възможности, ще се убедим, че $\{2, 3, 6\}$ е множеството, минимизиращо S и съответно то е търсеният 3-център в графа. Той обаче не е единствен! Например за множеството $\{1, 3, 6\}$ получаваме отново 3-радиус $S = 3$.

Следва реализацията на Си, която намира *само едно* решение. Предлагаме на читателя да я модифицира по подходящ начин така, че да намира всички решения.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 150
#define MAX_VALUE 10000
/* Брой върхове в графа */
const unsigned n = 7;
const unsigned p = 3;  /* p-център */

/* Матрица на теглата на графа */
int A[MAXN][MAXN] = {
  { 0, 1, 0, 0, 2, 0, 0 },
  { 1, 0, 4, 0, 0, 0, 0 },
  { 0, 4, 0, 3, 0, 5, 0 },
  { 0, 0, 3, 0, 4, 0, 8 },
  { 2, 0, 0, 4, 0, 0, 0 },
  { 0, 0, 5, 0, 0, 0, 2 },
  { 0, 0, 0, 8, 0, 2, 0 }
};

unsigned center[MAXN], pCenter[MAXN], pRadius;

/* Намира дължината на минималния път между всяка двойка върхове */
void floyd(void)
{ unsigned i, j, k;

  /* стойностите 0 се променят на MAX_VALUE */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
```

```
      if (A[i][j] == 0) A[i][j] = MAX_VALUE;

  /* Алгоритъм на Флойд */
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        if (A[i][j] > (A[i][k] + A[k][j]))
          A[i][j] = A[i][k] + A[k][j];

  for (i = 0; i < n; i++)
    A[i][i] = 0;
}

/* изчисляваме S за текущо генерираното подмножество */
void checkSol(void)
{ unsigned i, j, cRadius = 0;

  for (j = 0; j < n; j++) {
    int bT = MAX_VALUE;
    for (i = 0; i < p; i++)
      if (A[center[i]][j] < bT) bT = A[center[i]][j];
    if (cRadius < bT) cRadius = bT;
  }

  if (cRadius < pRadius) {
    pRadius = cRadius;
    for (i = 0; i < p; i++) pCenter[i] = center[i];
  }
}

/* комбинации C(n,p) - генерираме всички p-елементни подмножества на G */
void generate(unsigned k, unsigned last)
{ unsigned i;
  for (i = last; i < n - p + k; i++) {
    center[k] = i;
    if (k == p)
      checkSol();
    else
      generate(k + 1, i + 1);
  }
}

/* печатаме p-центъра и p-радиуса */
void printPCenter(void)
{ unsigned i;
  printf("%u-центърът на графа е следното множество от върхове: {", p);
  for (i = 0; i < p; i++) printf("%d ", pCenter[i] + 1);
  printf("}\n");
  printf("%u-радиусът на графа е %u\n", p, pRadius);
}
int main(void) {
  floyd();
  pRadius = MAX_VALUE;
  generate(0, 0);
  printPCenter();
  return 0;
}
```

🖫 p-center.c

Резултат от изпълнението на програмата:

```
3-центърът на графа е следното множество от върхове: {1 3 6 }
3-радиусът на графа е 3
```

Задачи за упражнение:

1. Да се провери, че множеството $\{2,3,6\}$ е p-център на графа от *фигура 5.7.5б*.
2. Да се даде пример за граф с няколко възможни p-центъра.
3. Да се модифицира горната програма така, че да намира *всички* възможни p-центрове.
4. Работи ли предложеният по-горе алгоритъм в случай на *ориентиран* граф? Ако да — налагат ли се някакви промени (ако пак да: какви?), ако не — защо (да се даде конкретен пример)?

5.7.6. Двойкосъчетание. Максимално двойкосъчетание

Намирането на *максимално* двойкосъчетание в граф (на англ. *matching*) е свързано с някои задачи от потоци в граф, които вече разгледахме.

Дефиниция 5.42. *Двойкосъчетание* в граф $G(V,E)$ се нарича такова подмножество $E'\subseteq E$, в което всеки връх от графа е край на не повече от едно ребро от E'. Върховете, инцидентни с ребро от E', се наричат *покрити*, а тези, които не са краища на ребра от двойкосъчетанието — *свободни*. Аналогично дефинираме *покрити* и *свободни ребра* в зависимост от това дали принадлежат на двойкосъчетанието, или — не.

Дефиниция 5.43. Едно двойкосъчетание E' от k ребра се нарича *максимално*, ако за всяко друго двойкосъчетание E'' от l ребра е изпълнено $k \geq l$.

Дефиниция 5.44. Нека $M\subseteq E$ е произволно двойкосъчетание в G. Един път в G се нарича *алтерниращ относно* M, ако от всеки две последователни ребра от пътя, точно едното участва в M. Един алтерниращ относно съчетанието M път се нарича *нарастващ*, ако началния и крайният му връх са *свободни* относно M.

Алгоритъм за намиране на максимално двойкосъчетание

Започваме с произволно двойкосъчетание (може и съставено от единствено ребро). На всяка стъпка намираме нарастващ път относно последното двойкосъчетание. Ако такъв път не съществува, то е максимално. (*Защо?*) В противен случай строим ново двойкосъчетание, което има повече ребра от първото по следния начин: Нека M е двойкосъчетанието, а P е нарастващият път. Тогава новото двойкосъчетание ще съдържа всички ребра от M и P, които не са едновременно в M и в P. Предоставяме на читателя една наистина сложна задача за упражнение: да състави схема за намиране на нарастващ път, необходима за реализацията на алгоритъма.

В случай на двуделен граф нещата значително се опростяват:

Дефиниция 5.45. Един неориентиран граф $G(V, E)$ се нарича *двуделен*, ако съществува разбиване $V=V'\cup V''$, $V'\cap V''=\varnothing$, такова че за всяко ребро $(i, j)\in E$ е изпълнено $i\in V'$ и $j\in V''$.

Задача: Да се намери максимално двойкосъчетание в *двуделен* граф.

Алгоритъм 1

Ще сведем задачата до задача за намиране на максимален поток в граф.

Нека е даден двуделен граф $G(V, E)$ с разбиване на множеството от върховете $V=V'\cup V''$, $V'\cap V''=\varnothing$. Ще решим задачата, като намерим максимален поток в графа, за който:

- върховете от множеството V' са източници.
- върховете от множеството V'' са консуматори.
- всички ребра от графа имат пропускливост $c(i, j) = 1$.

Алгоритъм 2

Общият алгоритъм (описан по-горе) за намиране на максимално двойкосъчетание в произволен граф може да се реализира лесно в случай на двуделен граф. В този случай броят на нарастващите пътища е ограничен от $\Theta(n)$, а нарастващ път може да се намери лесно с обхождане в ширина (*Как?*). Така общата сложност е $\Theta(n.(m+1)) = \Theta(n.m)$. За намиране на максимално двойкосъчетание в двуделен граф съществуват и по-ефективни алгоритми, със сложност $\Theta(\sqrt{n}\,.m)$ [*Hopcroft-Karp-1973*], обобщени по-късно и за произволен граф [*Micali,Vazirani-1980*].

Задачи за упражнение:

1. Да докаже, че ако не съществува нарастващ път относно дадено съчетание в граф, то съчетанието е максимално.
2. Да се състави схема за намиране на нарастващ път, необходима за реализацията на общия алгоритъм за намиране на максимално съчетание.
3. Да се състави схема за намиране на нарастващ път в двуделен път с обхождане в ширина.
4. Да се реализират алгоритми 1 и 2.

5.8. Оцветявания и планарност

5.8.1. Оцветяване на граф. Хроматично число

Много задачи, например при планиране на производствени процеси, съставяне на графици, съхранение и транспорт на товари, могат да се решат лесно, ако се сведат до съответна *задача за оцветяване на граф*.

Дефиниция 5.46. *r-оцветяване на върховете* на граф се нарича съпоставяне на точно един елемент от дадено множество $\{c_1, c_2, \ldots, c_r\}$ на всеки от върховете му. Аналогично се дефинира и *r-оцветяване на ребрата* на граф.

Дефиниция 5.47. Един неориентиран граф $G(V,E)$ се нарича *r-хроматичен*, ако върховете му могат да се *r-оцветят* така, че всеки два *съседни* да бъдат оцветени с различен цвят. Най-малкото число r, за което графът е r-хроматичен, се нарича *върхово хроматично число* на G (или само *хроматично число* на G) и се бележи с $\gamma_V(G)$. Аналогично, най-малкото число r такова, че ребрата на графа да могат да се r-оцветят така, че да няма инцидентни ребра, оцветени с еднакъв цвят, се нарича *ребрено хроматично число* на G и се бележи с $\gamma_E(G)$.

Следват два примера:

Пример 1: Трябва да подредим докладите на един конгрес по такъв начин, че никой участник да не бъде принуден да пропусне доклад, на който би желал да присъства. Каква минимална продължителност трябва да има програмата, ако разполагаме с достатъчен брой лекционни зали, позволяващи да се изнасят едновременно произволен брой доклади? В този пример можем да построим граф G, чиито върхове да са докладите. Два върха ще бъдат свързани с ребро т.с.т.к. има участник, който иска да ги посети и двата. Решението на тази задача се състои в намирането на $\gamma_V(G)$, което дава минималната продължителност на програмата.

Пример 2: Всеки от n бизнесмена иска да проведе срещи "на четири очи" с *някои* от останалите. Търсим какво е минималното време, за което ще се извършат срещите, ако всяка е с продължителност един ден и в нея участват точно двама души. Задачата може да се реши чрез намиране на ребреното хроматично число на граф $\gamma_E(G)$, което ни дава търсения брой дни.

- ограничаване отдолу на хроматичното число

В следващия параграф ще разгледаме алгоритми за намиране на хроматично число на граф. Тогава ще видим, че едно евентуално ограничаване отдолу на $\gamma_V(G)$ по някакъв критерий би спестило доста изчислително време. Такива критерии дават следните твърдения (някои от тях следват очевидно, докато валидността на други се проверява не толкова тривиално):

- $\gamma_V(G) \geq 2 \Leftrightarrow G$ съдържа поне едно ребро.
- $\gamma_V(G) \geq 3 \Leftrightarrow G$ съдържа цикъл с нечетна дължина.
- $\gamma_E(G) \geq d_m(G)$, където $d_m(G)$ е максималната степен на връх на G.
- $\gamma_V(G) \geq n / \gamma_V(G')$, където n е броят на върховете на G, а G' е допълнението на G.
- $\gamma_V(G) \geq n^2 / (n^2 - 2m)$, където n е броя на върховете, а m е броя на ребрата на G.

- намиране на върховото хроматичното число

Алгоритъм за намиране на $\gamma_V(G)$

1) С помощта на разгледаните в предходния параграф твърдения ограничаваме отдолу хроматичното число $\gamma_V(G)$ с някое $\gamma_{V\min}$.
2)

```
for (r = γVmin ; r < n; r++) {
if (G e r-хроматичен) { преминаваме_на_стъпка_3 }
```

3) r е хроматичното число на графа.

Очевидно задачата се свежда до това да проверим дали даден граф е r-хроматичен. Последното е *NP-пълна* задача и ще я решим в *6.3.2.*: Последователно се изпробват всички възможни оцветявания на всеки връх, като не се продължава с оцветяването, ако до момента вече има два съседни върха, оцветени с един и същ цвят. Този изчерпващ подход има добра успеваемост за графи с малък брой върхове, за разредени графи, както и за почти пълни графи. Съществуват и други изчерпващи алгоритми за намиране на върховото хроматично число. Така например, могат да се намерят максималните независими множества на графа: върховете, принадлежащи на едно независимо множество, могат да се оцветят с еднакъв цвят. За *произволни* графи това е един от най-добрите алгоритми [*Christofides-1975*][*Korman-1979*].

В общия случай, за да бъдем сигурни, че ще намерим правилно върховото хроматично число, трябва да приложим алгоритъм, основан на пълно изчерпване. На практика обаче, често се интересуваме не точно от минималното r-оцветяване (истинското хроматично число), а от някакво достатъчно добро негово *приближение*. За такива случаи съществуват алгоритми, които решават задачата бързо, но не винаги точно. За тях ще стане дума отново в *глава 9*.

Задача за упражнение:

Опитайте се да докажете критериите за ограничаване отдолу на $\gamma_V(G)$.

5.8.2. Планарност на графи

Дефиниция 5.48. Един граф се нарича *планарен* т.с.т.к може да се изобрази върху плоскост (начертае графично с точки и линии) така, че никои две негови ребра да не се пресичат.

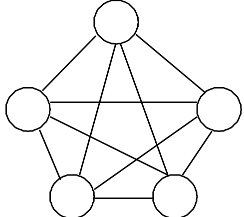
Например граф, описващ коридорите на една галерия, е планарен. Политическата карта на света също може да се представи с помощта на планарен граф (върховете съответстват на държавите, а всяко ребро показва обща сухоземна граница между две държави).

Известно е, че всеки планарен граф може да бъде нарисуван в равнината така, че не само никои две негови ребра да не се пресичат, но и ребрата да бъдат *прави отсечки*.

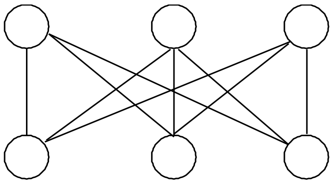
През 1933 Куратовски за пръв път дефинира строго понятието *планарност* на граф и дава съответно необходимо и достатъчно условие за планарност.

Ще припомним *дефиниция 5.8.* и ще приведем още няколко дефиниции, необходими за основната теорема за планарност на граф:

Нека са дадени неориентираните графи $G(V, E)$ и $G'(V, E')$, като за всяко $i, j \in V$ реброто (i, j) принадлежи на точно един от двете множества E' и E. Тогава G' се нарича *допълнение* на G. Ако за даден граф е изпълнено $(i, j) \in E$ за всяко $i, j \in V$, графът се нарича *пълен*. Пълен граф с n върха ще означаваме с K_n (K_5 е показан на *фигура 5.8.2а.*).



Фигура 5.8.2а. K_5.



Фигура 5.8.2б. $K_{3,3}$.

Дефиниция 5.49. Ако в двуделен граф $G(V, E)$ с разбиване $V = V' \cup V''$, $V' \cap V'' = \emptyset$, за всяко $i \in V'$ и $j \in V''$ следва $(i, j) \in E$, то казваме, че $G(V = V' \cup V'', E)$ е *пълен двуделен граф.*

Ако броят на елементите на V' е m, а броят на елементите на V'' е n, то графът се бележи с $K_{m,n}$. На *фигура 5.8.2б.* е показан $K_{3,3}$.

Дефиниция 5.50. *Свиване* на граф $G(V, E)$ се нарича графът $G'(V', E')$, който се получава от G чрез премахване на всички върхове от степен 2 и заместването на двете ребра през всеки такъв връх с едно ребро.

Дефиниция 5.51. Два графа G' и G'' се наричат *изоморфни* (записва се $G' \cong G''$), ако съществува биективно изображение на върховете на G' във върховете на G'', което запазва съседството.

Теорема. (*Куратовски*) Един граф е планарен т.с.т.к не съществува негов подграф, свиването на който е изоморфно на K_5 или на $K_{3,3}$.

Теорема. (*за четирите цвята*) Хроматичното число на планарен граф е по-малко или равно на 4.

Последната теорема означава, че винаги можем да оцветим политическа карта с четири цвята така, че да няма едноцветни съседни държави. Тя е била представена като хипотеза през 1850. По-късно са правени много безуспешни опити за доказателството ѝ, докато чак през 1976 г. К. Апел и В. Хакен успяват. Доказателството съдържа анализ на 2000 различни случая и е направено, с помощта на над 1000 часа компютърно време. В днешни дни съществуват много нейни доказателства, като повечето от тях отново използват компютърни алгоритми и изчисления.

Съществуват и полиномиални алгоритми за проверка [*Chiba,Nishizeki,Abe,Ozawa-1985*], като в случай на планарен граф, те намират и съответното планарно представяне. Тези въпроси обаче обхващат достатъчно обширен материал и няма да ги разглеждаме тук.

Задача за упражнение:

Вярна ли е *обратната* теорема за четирите цвята, т.е. ако хроматичното число на граф е строго по-малко от 5, то следва ли, че графът е планарен?

5.9. Въпроси и задачи

5.9.1. Задачи от текста

Задача 5.1.
Да се сравнят дефиниции *5.13.* и *2.2.*

Задача 5.2.
Да се съобрази как могат да се постигнат сложностите в *таблица 5.2.6.*

Задача 5.3.
Програмата от *5.3.1.* реализира алгоритъм за обхождане на граф в ширина от даден връх. В кои случаи, изпълнена за произволен връх, тя ще обходи *всички* върхове от графа?

Задача 5.4.
Как трябва да се модифицира програмата от *5.3.1.* така, че в случаите, когато останат необходени върхове, да се изпълнява ново обхождане в ширина върху тях?

Задача 5.5.
Да се обосноват резултатите от *таблица 5.3.2*.

Задача 5.6.
Да се модифицира функцията за обхождане `BFS()` от програмата за намиране на най-кратък път между два върха от *5.4.1.* така, че да прекратява работата си при достигане на крайния връх `ev`.

Задача 5.7.
Работи ли програмата за проверка дали граф е цикличен в случай на *ориентиран* граф (*виж 5.4.1.*)?

Задача 5.8.
Какви модификации можете да направите в програмата за проверка дали граф е цикличен в случай на ориентиран граф (*виж 5.4.1.*)? Опростява ли се или се усложнява кодът?

Задача 5.9.
Да се реализира алгоритъм за намиране на всички прости пътища на чрез обхождане в ширина (*BFS*), *виж 5.4.1.* Опростява ли се или се усложнява кодът? Променя ли се сложността на алгоритъма?

Задача 5.10.
Да се докаже признакът за оптималност от *5.4.2.*

Задача 5.11.
Да се обоснове нуждата от спазване на неравенството на триъгълника като следствие от признака за оптималност от *5.4.2.*

Задача 5.12.
Да се обоснове алгоритъмът на Форд-Белман от *5.4.2.*

Задача 5.13.
Защо най-външният цикъл в алгоритъма на Форд-Белман (*виж 5.4.2.*) се повтаря n-2 пъти, а не например n или n–1?

Задача 5.14.

Да се докаже, че ако след приключването на алгоритъма на Форд-Белман (*виж 5.4.2.*) за някоя двойка от върхове (*i*, *j*) е изпълнено `D[i] > D[j] + A[j][i]`, то графът съдържа отрицателен цикъл.

Задача 5.15.

Вярно ли е и обратното твърдение, а именно: Ако графът съдържа отрицателен цикъл, то след приключване на алгоритъма на Форд-Белман ще съществува двойка върхове (*i*, *j*), за които да бъде изпълнено `D[i] > D[j] + A[j][i]`? (*виж 5.4.2.*)

Задача 5.16.

По дадена двойка върхове (*i*,*j*) да се възстанови *един* конкретен минимален път, намерен по алгоритъма на Флойд (*виж 5.4.2.*).

Задача 5.17.

По дадена двойка върхове (*i*,*j*) да се възстановят *всички* минимални пътища, намерени по алгоритъма на Флойд (*виж 5.4.2.*).

Задача 5.18.

Да се реализира програма за пресмятане на надеждността на път (виж пример 1 за обобщения алгоритъм на Флойд, *5.4.2.*).

Задача 5.19.

Да се реализира програма за пресмятане на пропускливостта на път (виж пример 2 за обобщения алгоритъм на Флойд, *5.4.2.*).

Задача 5.20.

Да се реализира програма за намиране на първите *p* минимални пътя (виж пример 3 за обобщения алгоритъм на Флойд, *5.4.2.*).

Задача 5.21.

Да се сравни алгоритъмът на Дейкстра с тези на Форд-Фулкерсон и Флойд (*виж 5.4.2.*).

Задача 5.22.

Възможно ли е да се възстанови минималният път между дадена двойка върхове само от масива `d[]`, без да се използва допълнителен масив като `pred[]` (виж алгоритъма на Флойд от *5.4.2.*)?

Задача 5.23.

Да се реализира програма на базата на теоремата за повдигане в степен матрицата на съседство от *5.4.2.*

Задача 5.24.

Да се реализира алгоритъмът на Уоршал (*виж 5.2.4.*).

Задача 5.25.

Да се докаже, че алгоритъмът на Уоршал (*виж 5.2.4.*) правилно намира матрицата на достижимост на граф.

Задача 5.26.

Да докаже, че в ацикличен граф задължително съществуват:

- поне един връх без предшественик
- поне един връх без наследник

Задача 5.27.

Да се реализира вариант на алгоритъма най-дълъг път в ацикличен граф (*виж 5.2.4.*), при който на всяка стъпка се премахва връх без *предшественици*, вместо такъв без *наследници*.

Задача 5.28.

Да се даде пример за *ориентиран* граф и двойка негови върхове i и j, в който алгоритъмът за намиране на най-дълъг път между два върха в произволен граф от *5.2.4.* ***не*** намира правилно най-дългия път между i и j.

Задача 5.29.

Има ли случаи, когато алгоритъмът за намиране на най-дълъг път между два върха в *ацикличен* граф (*виж 5.2.4.*) ще работи правилно за *някоя* двойка върхове в *цикличен* граф? Ако да — при какво условие? Ако не — защо? Да се приведат съответни примери.

Задача 5.30.

Работи ли разгледаният в *5.4.3.* алгоритъм за намиране на всички цикли (с/без модификация) за неориентиран мултиграф?

Задача 5.31.

Работи ли разгледаният в *5.4.3.* алгоритъм за намиране на всички цикли (с/без модификация) за ориентиран граф? А за ориентиран мултиграф?

Задача 5.32.

Разгледаният в *5.4.3.* алгоритъм за намиране на всички цикли е със сложност $\Theta(m.(m+n))$, а реализацията, поради използването на матрица на съседство за представяне на графа — със сложност $\Theta(n^4)$. Възможно ли е да се подобри този резултат? Показахме, че броят на простите цикли е от порядъка на $\Theta(m+n)$, а дължината на всеки цикъл зависи от височината h на построеното покриващо дърво, т.е. минималната сложност за намирането и отпечатването на циклите е $\Theta((m+n).h)$. Ясно е, че в най-лошия случай височината е равна на броя на върховете на графа, т.е. получава се сложност $\Theta((m+n).n)$.

Можете ли да съставите алгоритъм за намиране на циклите, който да работи с посочената минимална сложност $\Theta((m+n).h)$? Каква стратегия за построяване на покриващо дърво можете да предложите, така че то да бъде с минимална височина? Може ли да се твърди, че при добре избрана стратегия h ще принадлежи на $O(\log_2 n)$?

Задача 5.33.

Да се реализира алгоритъмът за намиране на минимален цикъл през връх и да се пресметне сложността му (*виж 5.4.3.*).

Задача 5.34.

Даден е претеглен ориентиран граф и връх от него. Да се намери *прост* цикъл, съдържащ върха, за който сумата от теглата на участващите ребра да бъде минимална (*виж 5.4.3.*).

Задача 5.35.

Да се определят някои класове “специални” графи, за които намирането на Хамилтонов цикъл (*виж 5.4.4.*) е “лесно” (става с полиномиална сложност).

Задача 5.36.

Да се докаже теоремата на Ойлер и/или нейните следствия (*виж 5.4.5.*).

Задача 5.37.

Да се сравни задачата за намиране на Хамилтонов цикъл (*виж 5.4.4.*) с тази за намиране на Ойлеров (*виж 5.4.5.*). Каква е причината първата да бъде значително по-трудна от втората?

Задача 5.38.

Да се докаже, че ако в графа няма увеличаващ път, то полученият поток е максимален. (*виж 5.4.6.*)

Задача 5.39.

Да се докаже, че намереният от алгоритъма на Форд-Фулкерсон максимален поток върху модифицирания граф е максимален поток за изходния граф, който има няколко източника и няколко консуматора (виж *5.4.6.*)

Задача 5.40.

Да се докаже, че намереният от алгоритъма на Форд-Фулкерсон максимален поток върху модифицирания граф е максимален поток за изходния граф, който има капацитети на върховете (виж *5.4.6.*)

Задача 5.41.

Да се докаже теоремата на Уоршал (*виж 5.5.1.*).

Задача 5.42.

Да се докаже, че циклите от алгоритъма на Уоршал ***не*** могат да се разменят (*виж 5.5.1.*). Да се посочи примерен граф, при който разместването води до проблем.

Задача 5.43.

Да се докаже, че алгоритъмът на Пнуели, Лемпел и Ивена работи коректно (*виж 5.5.2.*).

Задача 5.44.

Какво ще се получи, ако се премахне условието $E' \subseteq E$ от дефиницията на транзитивна редукция, т.е. търси се граф с минимален брой ребра (не задължително подмножество на дадените) по дадена матрица на достижимост? Ще работи ли коректно и в този случай обратният алгоритъм на Уоршал (*виж 5.5.3.*)?

Задача 5.45.

Възможно ли е задачата за контрола на компании (*виж 5.5.4.*) да се реши със сложност, по-малка от $\Theta(n^3)$?

Задача 5.46.

Да се докаже Твърдение 5.25.

Задача 5.47.

Да се докаже, че ако на стъпка 2 на алгоритъм 1 от *5.5.5.* има повече от един връх без предшественици, сме свободни да изберем произволен измежду тях.

Задача 5.48.

Да се пресметне *аналитично* броят на различните топологични сортирания на графа от *фигура 5.5.6.*

Задача 5.49.

Да се изведе сложността на алгоритъм 1 от *5.5.7.*

Задача 5.50.

Нека е даден ацикличен граф $G(V, E)$. Търси се граф $G'(V, E')$, такъв че:

- $E \subseteq E'$
- Съществува *единствен* списък Z', който е топологично сортиране на G'.
- E' съдържа минимален брой ребра.

Каква е най-добрата сложност, която може да се постигне?

Задача 5.51.

Ще работи ли коректно алгоритъмът от *5.5.8.*, ако се разглежда *мултиграф*?

Задача 5.52.

Ще работи ли коректно алгоритъмът от *5.5.8.*, ако в графа има *примки*?

Задача 5.53.

Да се покаже, че редът на разглеждане на върховете в алгоритъма от *5.5.8.* е от значение.

Задача 5.54.

Да се предложи начин за преподреждане на върховете в алгоритъма от *5.5.8.* с линейна сложност.

Задача 5.55.

Възможно ли е да се реши задачата за построяване на граф по дадени степени на върховете (виж *5.5.8.*) със сложност $\Theta(m+n)$? А с по-малка?

Задача 5.56.

Да се реализира алгоритъм за намиране на свързаните компоненти на граф с обхождане в *ширина* (*виж 5.6.1.*).

Задача 5.57.

Да се докаже, че сложността на *алгоритъм 1* от *5.6.2.* е $\Theta(n\ .\ (n + m))$.

Упътване: Да се докаже, че най-лошият случай за алгоритъма е ацикличен ориентиран граф с точно $m = n.(n-1)/2$ ребра. В този случай компонентите на силна свързаност са n, а обхождането в дълбочина за всяка компонента има сложност $\Theta(m + n)$.

Задача 5.58.

Какъв ще бъде алгоритъмът за проверка дали даден ориентиран граф е *слабо* свързан и за намиране на всички негови *слабо свързани компоненти* (*виж 5.6.2.*)? Забележете, че за разлика от силната свързаност, която разбива графа на непресичащи се компоненти, в две различни слабо свързани компоненти могат да участват едни и същи върхове. Каква е сложността на Вашия алгоритъм?

Задача 5.59.

Да се докажат твърдения 1 и 2 от *5.6.3.*

Задача 5.60.

Да се докаже, че свойството на непресичащите се пътища се прехвърля от ребра в G' във върхове в G (*виж 5.6.4.*).

Задача 5.61.

Да се модифицира алгоритъмът за проверка дали граф е k-свързан по отношение на върховете в такъв за намиране на максималната k-свързаност на граф (*виж 5.6.4.*).

Задача 5.62.

Да се докаже, че алгоритъмът на Крускал работи правилно (*виж 5.7.1.*).

Задача 5.63.

Работи ли алгоритъмът на Крускал за *несвързан* граф (*виж 5.7.1.*)? Ако да, какъв е критерият за прекратяване добавянето на нови ребра?

Задача 5.64.

Работи ли алгоритъмът на Крускал за *ориентиран* граф (*виж 5.7.1.*)?

Задача 5.65.

Да се докаже, че алгоритъмът на Прим работи правилно (*виж 5.7.1.*).

Задача 5.66.

Работи ли алгоритъмът на Прим за *несвързан* граф (*виж 5.7.1.*)?

Задача 5.67.

Работи ли алгоритъмът на Прим за *ориентиран* граф (*виж 5.7.1.*)?

Задача 5.68.

4. Какви оптимизации могат да се приложат върху алгоритмите на Прим и Крускал (*виж 5.7.1.*) за случая, когато теглата на ребрата на графа са в интервала [1, n]?

Задача 5.69.

Да се обоснове предложеният алгоритъм за построяване на частично минимално покриващо дърво на базата на алгоритъма на Прим (*виж 5.7.1.*).

Задача 5.70.

Да се реализира предложената модификация на алгоритъма на Прим за построяване на частично минимално покриващо дърво (*виж 5.7.1.*).

Задача 5.71.

Да се докаже, че предложеният в *5.7.2.* алгоритъм за намиране на всички максимални независими множества работи правилно.

Задача 5.72.

Да се предложи начин за решаване на *задача 1*, дефинирана в началото на *5.7.2.*

Задача 5.73.

Да се докаже, че предложеният в *5.7.3.* алгоритъм за намиране на всички минимални доминиращи множества работи правилно.

Задача 5.74.

Да се докажат твърдения 1, 2 и 3 от *5.7.4.*

Задача 5.75.

Да се докаже, че предложеният в *5.7.4.* алгоритъм за намиране на база работи правилно.

Задача 5.76.

Да се даде пример за граф с няколко възможни центъра (*виж 5.7.5.*).

Задача 5.77.

Да се модифицира програмата от *5.7.5.* така, че да намира *всички* възможни центрове на графа.

Задача 5.78.

Да се провери, че множеството {*2*,*3*,*6*} е p-център на графа от *фигура 5.7.5б.*

Задача 5.79.

Да се даде пример за граф с няколко възможни p-центъра (*виж 5.7.5.*).

Задача 5.80.

Да се модифицира програмата от *5.7.5.* така, че да намира *всички* възможни p-центрове.

Задача 5.81.

Работи ли предложеният в *5.7.5.* алгоритъм в случай на *ориентиран* граф? Ако да — налагат ли се някакви промени (ако пак да: какви?), ако не — защо (да се даде конкретен пример)?

Задача 5.82.

Да докаже, че ако не съществува нарастващ път относно дадено съчетание в граф, то съчетанието е максимално (*виж 5.7.6.*).

Задача 5.83.

Да се състави схема за намиране на нарастващ път, необходима за реализацията на общия алгоритъм за намиране на максимално съчетание (*виж 5.7.6.*).

Задача 5.84.

Да се състави схема за намиране на нарастващ път в двуделен път с обхождане в ширина (*виж 5.7.6.*).

Задача 5.85.

Да се реализират алгоритми 1 и 2 от *5.7.6*.

Задача 5.86.

Опитайте се да докажете критериите за ограничаване отдолу на $\gamma_V(G)$ от *5.8.1*.

Задача 5.87.

Вярна ли е *обратната* теорема за четирите цвята (виж *5.8.2.*), т.е. ако хроматичното число на граф е строго по-малко от 5, то следва ли, че графът е планарен?

5.9.2. Други задачи

В заключителната част са приложени редица интересни теоретични задачи, както и такива, давани на български и международни ученически и студентски състезания по програмиране.

Задача 5.88. *Задача за Китайския пощальон, Duke programming contest, 1992, problem f.*

Тази задача (от англ. *Chinese Postman Problem*) е наречена на името на китайския математик Мей Го Гуан).

Да се намери цикъл с *минимална дължина* в претеглен граф, минаващ през всяко ребро *поне* веднъж. Графът не е задължително Ойлеров (Ако е Ойлеров, то произволен Ойлеров цикъл удовлетворява условието).

Задача 5.89. *Duke programming contest, 1993, problem g.*

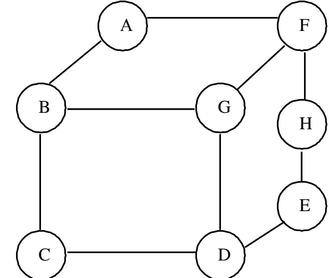
Даден е списък от релации между променливи от вида $i < j$. Да се напише програма, която отпечатва всички възможни подредби на променливите, съвместими с дадените релации. Например, за релациите $x < y$ и $x < z$ има две възможни подредби: $x\ y\ z$ и $x\ z\ y$.

Задача 5.90. *Duke programming contest, 1993, problem h.*

Дадени са площади, свързани с еднопосочни улици. Да се напише програма, която намира броя на пътищата между два дадени площада. Възможно е между два площада да има безброй много пътища (например, ако се минава през цикъл). В такъв случай програмата трябва да отпечата –1.

Задача 5.91. *ACM Regional contest, New Zealand, 1991, problem A*

Даден е неориентиран граф $G(V, E)$. Ако е дадена наредба на върховете му, то *B-степен* за върха $v \in V$ се дефинира като най-дългото разстояние в наредбата между v и някой негов съседен връх. *B-степен* на графа се нарича максимумът от индивидуалните *B-степени* за всеки връх.



Фигура 5.9.2а. *B*-степен на граф.

Например за графа от *фиг. 5.9.2а.* и за две примерни наредби (*A*, *B*, *C*, *D*, *E*, *H*, *F*, *G*) и (*A*, *B*, *C*, *D*, *G*, *F*, *H*, *E*) *B-степените* на върховете са: 6, 6, 1, 4, 1, 1, 6, 6 за първата наредба и 5, 3, 1, 4, 3, 5, 1, 4 — за втората, което дава *B-степен* за графа 6 в първия, и 5 — във втория случай.

Да се напише програма, която намира наредба на върховете, за която *B-степента* на графа да бъде минимална.

Задача 5.92*. Минимален прост цикъл*

Даден е неориентиран претеглен мултиграф. Да се намери минимален по дължина прост цикъл в графа, в който участват поне 3 върха.

Решение: Това, че е даден мултиграф не е съществено в случая. Ако между два върха i и j има повече от едно ребро, то оставяме само минималното от тях (в *прост* цикъл всеки връх може да участва най-много по веднъж — така е невъзможно в него да участва повече от едно ребро между два върха. Запазваме само минималното ребро, тъй като искаме да намерим минимален цикъл). По нататък решението е следното:

Алгоритъм 1. За всяко ребро (i, j) от графа изпълняваме:

- изключваме (i, j) от графа
- намираме минималния път между i и j. Заедно с изключеното ребро се получава цикъл с дължина L_{ij}.

Минималното L_{ij} е дължината на търсения минимален цикъл.

Сложността на алгоритъма е $\Theta(m.n.\log_2 n)$.

Алгоритъм 2. Нека:

- $d(i, j)$ е дължината на минималния път между върховете i и j.
- $f(i, j)$ е теглото на реброто (i, j).

Тогава дължината на минималния прост цикъл е минимумът на сумата $d(i,k) + f(k,j) + d(j,i)$, за всяко i, j и k, за които минималният път между i и k няма общи точки с минималния път между j и i.

Обяснете как се получава посочената сложност на *алгоритъм 1*. Пресметнете сложността на *алгоритъм 2*.

Задача 5.93. *Минимален цикъл през k върха*

Дадено е фиксирано естествено число k. Да се предложи алгоритъм, който за произволен неориентиран претеглен мултиграф $G(V,E)$ намира минимален по дължина прост цикъл в G, съдържащ поне k върха.

Очевидно, при $k = n$ се получава задачата за Търговския пътник и въпреки това, задачата не е *NP*-пълна (*виж глава 6*), тъй като сложността ѝ се определя от n, а k е фиксирано.

Каква най-добра сложност като полином на n можете да постигнете за решаване на задачата, като се има предвид, че k е произволна, но *фиксирана* константа?

Задача 5.94. *Числова база в граф*

Дефиниция 52. Даден е ориентиран граф $G(V, E)$ и функция $v : V \to R^+$, задаваща тегла на върховете на графа. *Числова база от върхове* се нарича множество $T \subseteq V$ такова, че:

- всеки връх от V е достижим от някой връх от T
- сумата от теглата на върховете от T е минимална.

Да се намери числовата база от върхове на графа.

Следват две задачи, които на пръв поглед нямат нищо общо с графи, но на практика се решават лесно, ако ги представим с граф и търсим най-дълъг път в него (*виж 5.4.2*).

Задача 5.95. *Максимална ненамаляваща редица*

Да се намери максимална ненамаляваща подредица на дадена числова редица. Или по-строго: Дадена е редица от цели числа $a_1, a_2, \ldots, a_n$. Да се намери такава нейна подредица $a_{i1}, a_{i2}, \ldots, a_{ik}$ ($i_1 < i_2 < \ldots < i_k$), че $a_{ij} \le a_{i(j+1)}$ за $j = 1, 2, \ldots, k-1$, и сумата $S = \sum_{j=1,2,\ldots k} a_{ij}$ да бъде максимална.

Задача 5.96. *Тримерно влагане*

Дадени са n кутии, определени с координатите на върховете си (x_i, y_i, z_i). Да се намерят максимален брой кутии такива, че да могат да се подредят последователно една в друга.

Упътване: При тази задача можем да построим граф, в който върховете ще представят кутиите, а ориентираните ребра ще показват дали една кутия може да се побере в друга.

Задача 5.97. *Минимален път в ацикличен граф*

Може ли да се подобри сложността от алгоритъма на Дейкстра, ако се търси минимален път *само между два* върха в *ацикличен* граф?

Задача 5.98. *Обратен корен*

Даден е ориентиран граф. Да се състави алгоритъм за проверка дали в графа съществува връх с полустепен на входа $n-1$ и полустепен на изхода 0.

Задача 5.99. *Проверка дали граф е двуделен*

Даден е граф с n върха и m ребра. Да се провери със сложност $\Theta(m+n)$ дали даденият граф е двуделен.

Упътване: Използвайте модификация на функцията за обхождане в дълбочина.

Задача 5.100. *Проверка за изоморфизъм на графи*

Каква най-добра сложност можете постигнете за проверката дали два дадени графа са изоморфни?

Задача 5.101. *Допълнение на планарен граф*

Даден е планарен граф $G(V, E)$ с n ($n \ge 11$) върха. Да се провери дали допълнението на G е планарен граф.

Упътване: Да се покаже, че допълнението на G не може да бъде планарен граф, независимо от вида на G.

Задача 5.102. *Списание Компютър-5/1994*

Даден е неориентиран граф. За някои от върховете от графа е дадено реално число — *тегло* на върха. Да се провери може ли да се съпоставят реални числа и на останалите върхове, така че теглото на всеки връх да е равно на средноаритметичното от теглата на съседите му.

Упътване: Графът се разделя на компоненти на свързаност. Всички върхове в една компонента трябва да имат едно и също тегло, за да бъде изпълнено условието на задачата.

Задача 5.103. *d-компресия*

Дадена е квадратна матрица `A[][]` с размер $n \times n$, съставена само от нули и единици. Известен е броят на единиците във всеки ред и всяка колона на матрицата. Да се възстанови матрицата.

Задача 5.104. *Балканска олимпиада по програмиране (Варна 1995, Ден 1, Задача 1)*

Система за петрол представлява n ($2 \le n \le 200$) резервоара с обем от 1 до 100 и $n-1$ тръби, свързващи резервоарите в затворена система (без цикли). Системата се пълни от един от резервоарите S (източник), който винаги е празен и се използва като помпа. За единица време по всяка от тръбите, свързани с източника S, преминава единица петрол. Процесът продължава, докато се напълнят всички резервоари.

При дадена свързана система от резервоари да се намери оптимален източник S, от който процесът на пълнене да изисква минимално време.

Решение [*Infoman*]: Даден е свързан ацикличен неориентиран граф $G(V, E)$ с тегла на върховете. След отстраняване на източника S графът се разделя на k ($2 \leq k \leq n-1$) компоненти на свързаност. Нека сумата от теглата на върховете в i-тата компонента на свързаност е T_i. Тогава, източникът S трябва да бъде избран така, че $T = \max\{T_i\}$, за $i = 1,2,...,k$ да бъде минимално.

Каква е най-добрата сложност която може да се постигне за решението на задачата, така че използваната памет да бъде $\Theta(n)$?

Задача 5.105. *Разделяне на върхове (ребра)*

Даден е неориентиран граф, в който са избрани два върха s и t. Да се намери броят елементи на минималното множество H от върхове (ребра) от графа такова, че ако се изключат от графа участващите в H върхове (ребра), то s и t остават в различни компоненти на свързаност.

Задача 5.106. *Поток в граф с пропускливост на ребрата реални числа*

Как ще се промени задачата за намиране на максимален поток в граф, ако дадените тегла (пропускливост) на ребрата са реални числа?

Упътване: Сложността на алгоритъма в този случай ще зависи от теглата на ребрата и от големината на потока. Алгоритъм със сложност, зависеща единствено от броя на върховете n на графа, не е известен.

Задача 5.107. *Допълване до силно свързан граф*

Даден е ориентиран слабо свързан граф. Да се добавят минимален брой ребра към графа така, че той да стане силно свързан.

Задача 5.108. *2^n*

Дадено е цяло число n. Да се намери низ с минимална дължина такъв, че в него да участват като поднизове двоичните представяния на всички числа от 1 до n.

Упътване: Задачата се решава, като се търси Ойлеров цикъл. Съществуват и някои по-прости алгоритми, чиято коректност се доказва с разглеждане на Ойлерови цикли.

Задача 5.109. *Диаметър на граф*

Да се напише програма, която намира диаметъра на граф.

Можете ли да съставите алгоритъм, който да има сложност, по-малка от $\Theta(n^3)$?

Задача 5.110. *Метод на вълната*

Дадена е квадратна матрица с елементи 0 и X и координатите на две клетки от нея: начална (s_x,s_y) и крайна (t_x,t_y). Да се намери път между тях, т.е. последователност от клетки такава, че:

- Пътят да се състои само от клетки със стойност 0.
- Всеки две поредни клетки от пътя да бъдат "съседни", т.е. да се различават с единица точно в една от двете си координати.
- Общата дължина на пътя да бъде минимална.

Решение: Задачата се решава ефективно с т.нар. *метод на вълната* (специфична интерпретация на техниката обхождане в ширина). Методът се състои в следното:

1) На стартовата позиция записваме числото 1. Полагаме $i = 1$.
2) Намираме всички клетки, в които е записано числото i. Във всички техни свободни съседи (т.е. клетки със стойност 0) записваме числото i+1.
3) Ако някой от съседите от стъпка 2 съвпада с крайната позиция, алгоритъмът приключва (намерен е минималния път и дължината му е i+1). В противен случай увеличаваме i и повтаряме стъпка 2).

X	X	X	X	X	X	X	X
X	**1**	**2**	**3**	**4**	**5**	**6**	X
X	**2**	X	X	X	**6**	**7**	X
X	**3**	**4**	**5**	X	**8**	X	X
X	**4**	X	**6**	**7**	**8**	X	X
X	**5**	**6**	**7**	X	X	X	X
X	**6**	**7**	**8**	**9**	X	X	X
X	X	X	X	X	X	X	X

Фигура 5.9.2б. Метод на вълната.

Методът носи името *метод на вълната*, защото търсенето на пътя с последователно напредване във всички възможни посоки прилича на начина, по който се разпространява вода в затворено пространство, с източник стартовата позиция. Ефективността на реализацията на горния алгоритъм се определя от начина на търсене на съседите от всяка поредна стъпка.

Глава 6
Търсене с връщане. NP-пълни задачи

"… не се предаваме и не отстъпваме,
ние просто напредваме в друга посока …"

~ Генерал Мак-Кларк, изтегляйки атакуващата войска.

Общоприето е една задача да се счита за "добре решена", ако сложността на съответния алгоритъм е линейна или е полином (от ниска степен). Често обаче се сблъскаме с важни задачи, за които не съществува "бърз" алгоритъм. Тези задачи ще бъдат предмет на настоящата глава, като различни техники за тяхното решаване ще се разглеждат не само тук, но и в следващите глави.

6.1. Класификация на задачите

Когато се говори за *класове* от задачи, се разбира групиране на задачите, според сложността им — задачи с близка сложност попадат в един и същ клас. Съществуват различни критерии за класифициране на задачите в зависимост от "трудността" им. Ще разгледаме класификация на задачите относно два от най-важните критерии за оценка: *време* и *памет*.

6.1.1. Сложност по време

Засега ще се ограничим с разглеждане единствено на *задачи за проверимост* (т.е. такива, при които се търси отговор с *да* или *не*).

NP-задачи

NP (от англ. *non-deterministic polynomial time*) е класът на *полиномиално проверимите* задачи. Една задача принадлежи на класа *NP*, ако е възможно с полиномиална сложност да се провери дали даден кандидат действително е решение, без да се интересуваме колко време ще отнеме намирането на такъв кандидат. Казано с прости думи, *NP*-задачите са такива задачи, за които, ако веднъж успеем да "налучкаме" правилен кандидат за решение, то лесно можем да го проверим (*виж пример 2* от *6.1.4.*).

P-задачи

Това са задачи за проверимост, за които съществува решение с полиномиална сложност (съкращението *P* идва от англ. *polynomial*). От своя страна този клас (който е подклас на класа *NP*) е доста обширен: когато се разглеждат практически проблеми, степента на полиномиалната функция е от изключително значение. Този клас съдържа и алгоритми, чиито функции имат асимптотично нарастване, което е по-ниско от това на линейната функция. Такива са логаритмичната функция (често тези задачи се разглеждат в отделен подклас — log-*задачи*), обратната функция на Акерман и др. (*виж пример 1* от *6.1.4.*).

Експоненциални задачи

Това са задачи, които могат да се решат с експоненциална сложност. Този клас съдържа предишните два, но не е всеобхватен — съществуват задачи, за които най-добрите алгоритми имат сложност, по-висока от експоненциалната.

6.1.2. Сложност по памет

При тази класификация критична е паметта, която се използва, като функция от големината на входните данни. При тях не се интересуваме от сложността на алгоритъма за решаването им. Този клас може да се разпадне на няколко подкласа: *P-space* (задачи, за които е необходима полиномиална памет), *exp-space* (експоненциална памет) и др. (*виж пример 3* от *6.1.4.*).

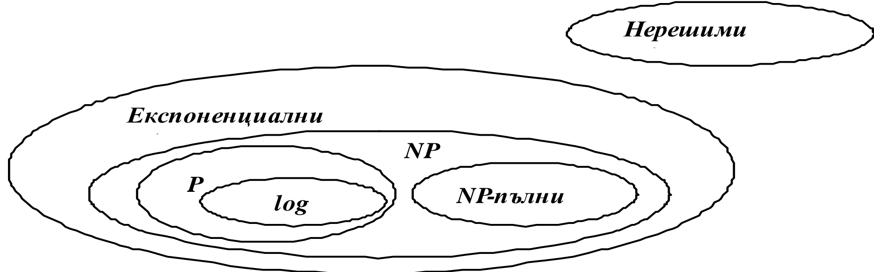
Ясно е, че изчислителната сложност на алгоритъма е винаги поне толкова, колкото е сложността по памет. Наистина, не е възможно да се заеме памет с размер $\Theta(n)$ за време, по-малко от $\Theta(n)$ [*Манев-1996*].

6.1.3. Нерешими задачи

Съществуват алгоритмични задачи, за които е възможно да се докаже [*Манев-1996*], че не могат да бъдат решени, независимо от това с колко време и памет разполагаме. По-нататък ще посочим конкретни примери (*виж пример 4 от 6.1.4.*).

6.1.4. Примери

Някои от изброените класове от задачи (и връзките между тях) са показани схематично на *фигура 6.1.4*. На фигурата е показан и класът на *NP-пълните* задачи, който ще разгледаме по-подробно в *6.2*. Фигурата показва *примерно* разположение на задачите, защото точното им положение не е определено. Един от големите въпроси в теоретичната информатика е дали $P \subset NP$ или $P \equiv NP$. Приема се, че по-вероятна е първата хипотеза и *фигура 6.1.4.* отразява това.



Фигура 6.1.4. Класификация на задачите (по време).

Ще разгледаме няколко примера за задачи, принадлежащи към различни класове.

***Пример 1**. Полиномиално разрешими задачи*

Могат да бъдат изброени редица примери за полиномиално решими задачи, доста от които бяха вече разгледани в материала дотук: алгоритми от теорията на графите (за обхождане на граф и модификациите им, алгоритми на Дейкстра, Флойд, Прим, Крускал и много други — разбира се, трябва да ги дефинираме като задачи за *проверимост*), за търсене (например двоично търсене — със сложност $\Theta(\log_2 n)$), почти всички алгоритми за сортиране, за пресмятане на различни математически функции и много други.

***Пример 2**. NP-задачи*

Следният прост пример илюстрира добре смисъла на класа на *NP*-задачите: Дадено е естествено число n. Да се провери дали съществува делител на дадено естествено число n, по-малък от дадено естествено число q $(1 < q < n)$.

Ако разполагаме с кандидат за решение q', лесно можем да проверим дали съществува p' такова, че $n = p'.q'$, но за решаването на горната задача необходимите изчислителни ресурси са значително повече.

Ще разгледаме и втори пример:
Дадени са граф $G(V,E)$, два негови върха $i,j \in V$ и естествено число k.

Задача: Да се провери дали съществува прост път (без повторение на върхове) между i и j с дължина поне k.

Тази задача принадлежи на класа *NP*: ако вече сме получили някакъв път, можем лесно (с линейна сложност) да проверим условието дали пътят е прост и дали е с дължина поне k. Алгоритъм с полиномиална сложност за намиране на такъв път обаче още не е известен, така че не можем да бъдем сигурни дали задачата принадлежи на класа *P*.

По-късно ще забележим, че тази задача принадлежи на нов клас от задачи, за който стана дума в увода и който ще бъде основен предмет на тази глава — класът на *NP-пълните* задачи. За тях се *предполага*, че не съществува полиномиално решение.

Пример 3. *Класификация по памет*

В почти всички видове "интелектуални" спортове (например шах, различни игри на карти и т.н.) от доста време се организира и участие на компютри. В началото, при първите подобни сблъсъци, компютърните "състезатели" нямаха почти никакъв шанс и победите на човека над машината служеха за утвърждаване на тезата, че естественият интелект трудно може да бъде конкуриран от изкуствени изчислителни процеси. Голяма част от поддръжниците на това мнение обаче "изпиха чаша студена вода", когато през 1997 *Deep Blue*, компютър на *IBM*, в официален мач победи за пръв път световния шампион по шах Гари Каспаров. През 1996 г. *Deep Blue* спечелва една партия срещу Каспаров, но губи срещата (общо 6 партии). По-късно, през 1997 г. обаче успява да го победи с 3,5:2,5 точки (2 победи, 1 загуба и 3 равни), за което получава учредената 17 години по-рано награда от $100,000 за първата компютърна програма, която победи световния шампион по шах. Ще отбележим обаче, че възможностите на съвременния компютър все още не позволяват на компютърна програма да стане световен шампион по шах, а ако е възможно, това все още не е доказано на практика: *Deep Blue* е спечелил само част от партиите, при това, след като е бил специално програмиран да играе добре именно срещу Каспаров. [*Russell,Norvig-1995*][*Loviglio*-1997]

При по-прости игри като *дама* обаче, машината е доказала вече своето превъзходство над човека. През 1992 г. компютърната програма *Chinook*, създадена от Джонатан Шефер и негови колеги, спечелва шампионата *US Open*, но не успява да стане световен шампион, тъй като Марион Тинсли успява да я победи с 21,5:18,5. Тинсли е световен шампион в продължение на 40 години (!) и през това време губи само 3 официални мача. При сблъсъка си с *Chinook* обаче записва своите четвърта и пета загуби. По-късно, през 1994 г. *Chinook* става световен шампион, след като официалният ѝ мач срещу Тинсли е прекъснат, поради оттегляне на последния по здравословни причини.

Да се върнем обаче на шахмата. В какво се състои програмирането на шахматни програми? Поредицата от ходове е възможно да се интерпретира като дърво: пред играча, започващ играта, са възможни 20 различни хода, за всеки от тях съществуват множество различни ответни ходове и т.н. Шахматните програми обхождат това дърво и проверяват какви ще бъдат възможните последствия от евентуалното извършване на всеки възможен ход.

Задачата за перфектно играеща шахматна програма е от клас *exp-space*, ако разглеждаме размера на дъската като размер на входните данни. Какъв е смисълът от класификация по памет? В шахмата размерът на дъската е константа — 8×8. Така запазването на всички възможни конфигурации и най-добрия ход за всяка от тях изисква константна памет, въпреки че тази константа е огромна (ако допуснем, че всяка шахматна конфигурация е възможна, то броят на различните шахматни конфигурации е 65^32 – защо?). От друга страна, височината на шахматното дърво не е голяма: рядко се играе партия по-дълга от 150 хода. Така, ако извършим претърсване на дървото при всеки ход, отпада проблемът, свързан с недостига на памет, но се появява друг: неприемливо дълго (експоненциално) време за изчисление *на всеки ход*.

Съществуват методи за обхождане, известни като алфа-бета отсичане (*виж 6.5.3.*), които ограничават изследването на дървото на решенията, но дори и те не могат да се справят с огромния брой върхове за обхождане. Поради тази причина шахматните програми реално извършват претърсване в по-малка дълбочина (например 8 хода напред) и от събраната информация, както и въз основата на някои "тактически" критерии (например колко и какви фигури са останали на дъската и др., *виж 6.5.4.*) определят най-добрия възможен ход.

***Пример 4**. Нерешими задачи*

Следната задача е нерешима: Дадени са два низа x и y и множество от правила за заместване от вида $v \to u$. Да се провери дали низът x може да се трансформира в низа y, чрез поредица от прилагания на заместващите правила. Правилата за заместване се прилагат по следния начин: всеки път, когато във временния низ (последната получена трансформация) участва поднизът v, той може да се замени с u (за дадено правило $v \to u$).

Повечето задачи, отнасящи се до поведението на дадена програма, също са нерешими. За целта ще разгледаме следния пример: Когато една програма работи продължително време, без да даде резултат, е трудно да се прецени дали тя, казано на "програмистки" език, е "зациклила" или просто алгоритъмът се нуждае от още време, за да приключи. Наистина, щеше да бъде изключително удобно, ако съществуваше автоматичен метод за подобна проверка (например компилаторът сам да може да прецени това). Тази задача обаче е от класа на нерешимите: не съществува алгоритъм, който може винаги и без грешка да определи дали дадена програма се намира в безкраен цикъл, или — не. Следва пример, илюстриращ казаното до тук:

```
int main() {
  unsigned a, b, c, i, x;
  for (x = 3;;) {
    for (a = 1; a <= x; a++)
      for (b = 1; b <= x; b++)
        for (c = 1; c <= x; c++)
          for (i = 3; i< = x; i++)
            if (pow(a, i) + pow(b, i) == pow(c, i))
              exit(0);
    x++;
  }
  return 0;
}
```

Показаният фрагмент проверява дали съществува решение на уравнението $a^n + b^n = c^n$, за a, b, c, n естествени числа и $n > 2$. Съгласно доказаната наскоро *Голяма теорема на Ферма*, такава четворка естествени числа не съществува. За човек (и за компилатор), който не знае, че теоремата вече е доказана ще бъде трудно да прецени дали горната програма ще намери решение и ще завърши, или — не. Нещо повече, съществуват много подобни задачи, за които съответен теоретичен резултат не е известен и изобщо не се знае дали съответстващите програми ще зациклят.

Задачата за определяне дали дадена програма зацикля се нарича *стоп-проблем*. Теоретичното ѝ разглеждане при някой от съществуващите изчислителни формализми (например машини на Тюринг — виж [*Манев-1996*]) позволява строго доказателство на този факт.

***Пример 5**. Неопределени задачи*

За някои задачи все още не е определено дали принадлежат на класа на нерешимите задачи: подобен пример е *задачата на Сколем* [*MathWorld-a*]:

Дадена е матрица $M_{3\times3}$. Да се провери дали съществува естествено число n такова, че ако M се повдигне в степен n, се получава матрица, чийто горен десен ъгъл съдържа 0.

Задачи за упражнение:

1. Да се определи броят на различните шахматни конфигурации.
2. Да се дадат примери за задачи от теорията на графите, които имат логаритмична, полиномиална или експоненциална сложност.
3. На един и същи клас ли принадлежат следните две задачи:
   - Да се провери дали дадено естествено число n е просто.
   - Да се провери дали съществува делител на дадено естествено число n, по-малък от дадено естествено число q ($q > 1$, $q < n$).

6.2. NP-пълни задачи

От теоретична гледна точка, един от най-интересните въпроси е дали класът *P* съвпада с *NP*. Казано с други думи, ако винаги с полиномиална сложност е възможно да се провери даден кандидат за решение, то трябва ли да може и да се намери цялото решение с такава сложност?

Очакванията са, че отговорът на горния въпрос е *отрицателен*. Доказателство, подкрепящо тази теза обаче, все още не съществува. Т.е. не можем да бъдем сигурни, че дадена *NP* задача не е в *P*. Типът задачи, които наричаме *NP-пълни*, внасят допълнителна яснота при търсенето на решение на този проблем. *NP-пълни* е клас от *NP*-задачи, които могат да се свеждат една към друга с полиномиална сложност (това ще дефинираме строго след малко), и всяка задача от *NP* може да се сведе полиномиално до някоя *NP*-пълна задача. Ако само за една *NP*-пълна бъде доказано, че принадлежи или не принадлежи към класа *P*, то това ще следва и за всички останали задачи от класа *NP* (Забележете от *целия* клас *NP*, а не само от *NP*-пълни.). Откриването на полиномиален алгоритъм за *NP*-пълна задача е обобщение на “фикс-идеята” на всеки начинаещ математик — откриване на проста формула за генериране на последователни прости числа (Предоставяме на читателя да съобрази това, след като се запознае с материала от настоящата и предходната точка).

Дефиниция 6.1. Казваме, че задачата A е *сводима към* задачата B и пишем $A \langle B$, ако е възможно да се намери алгоритъм за решаването на A, използващ полиномиален брой обръщения към програмата, решаваща B, и всички изчисления извън тези обръщения са с полиномиална сложност.

Тогава, ако дадена задача B е от класа P и $A \langle B$, то следва, че A също е от класа P. Да илюстрираме казаното дотук с един пример: намиране на Хамилтонов цикъл в граф (*виж 5.4.4.*) — т. е. цикъл, в който всеки връх участва точно по веднъж. Задачата може да се реши, с помощта на *пример 2*:

```
for (всяко_ребро_(i,j)_от_графа) {
  if (съществува_прост_път_с_дължина_n-1_върха_от_i_до_j) {
    return 1; /* Има Хамилтонов цикъл: намереният път + реброто (i,j)*/
  }
}
return 0;
```

Да означим броя на ребрата в графа с m. Алгоритъмът ще извърши m обръщения към задачата за намиране на най-дълъг път в граф и сложността извън тази задача ще бъде $\Theta(m)$. Ако приложим дефиницията по-горе, това означава, че “*Хамилтонов цикъл” 〈”Най-дълъг прост път”*. Тъй като обаче не е известен полиномиален алгоритъм за решаването на *“най-дълъг прост път”*, то не можем да твърдим, че Хамилтоновият цикъл е полиномиално решима задача.

Дефиниция 6.2. Дадена *NP*-задача A се нарича *NP-пълна* (англ. *NP-complete*), тогава и само тогава, когато за всяка друга задача B от класа *NP* следва, че $B \langle A$.

Теорема. (*Кук*) Съществува поне една задача, принадлежаща на класа *NP*-пълни.

Доказателството на последната теорема се извършва, чрез намиране на такава *NP*-пълна задача [*Манев-1996*] — например *задачата за удовлетворимост на булева функция* (*виж 6.3.1.*).

Прилагането на горната дефиниция и теоремата на Кук при много задачи е довело до доказателството, че те са *NP*-пълни. В последния параграф на главата (*виж 6.6.*) са дадени условията на над 70 *NP*-пълни задачи. Съществуват още стотици задачи, принадлежащи към този клас. Ако само за една от тях бъде доказано, че може или че не може да бъде решена с полиномиален алгоритъм, то това ще следва и за всички останали: не само за тези от *NP*-пълни, но и от *NP* изобщо. А оттук директно ще следва, че $P \equiv NP$.
В теорията на алгоритмичната изчислимост и сложност класовете, които разглеждахме досега (*P*, *NP*, *NP*-пълни) бяха задачи за *проверимост*, т.е. отговарящи на въпрос с *да* или *не*. Следват някои примери (включително вече разгледани):

1. Да се провери дали даден граф е Хамилтонов.

2. Да се провери дали съществува Хамилтонов цикъл в граф с дължина, по-малка от k.

Съществуват обаче редица задачи, които не са задачи за проверимост, например:

3. Да се намери Хамилтонов цикъл с минимална дължина.

В тези случаи се налага теорията да се разшири и за целта се въвежда още един клас задачи: *NP*-трудни.

Дефиниция 6.3. Една задача *A* (не задължително задача за проверимост) принадлежи на класа *NP-трудни*, тогава и само тогава, когато за всяка задача *B* от класа *NP* следва, че $B \langle A$.

Като следствие от дефиницията, лесно се доказва, че класът на *NP*-пълните задачи е сечение на класовете *NP* и *NP*-трудни. Така връзката между *NP*-пълни и *NP*-трудни задачи може да се изкаже като: *NP*-пълни са тези *NP*-трудни задачи, които са задачи за проверимост.
За всяка *NP*-пълна задача може да се дефинира съответна “не по-лесна” *NP*-трудна. Така например, за решаването на *задача 2* по-горе не съществува друго решение (в общия случай на произволен граф), различно от намиране на минималния Хамилтонов цикъл (което е точно *задача 3*).

Задачи за упражнение:

1. Като се използва само *дефиниция 6.2.* да се докаже, че ако дадена *NP*-пълна задача *A* е сводима към друга задача *B*, то *B* също е *NP*-пълна.
2. Да се дадат примери за задачи от теорията на графите, които принадлежат на класа *NP*-пълни задачи, посредством свеждане до търсене на Хамилтонов цикъл в граф (допускаме, че *NP*-пълнотата на задачата за търсене на Хамилтонов цикъл е доказана).
3. Задачата "Да се провери дали съществува делител на дадено естествено число n, помалък от дадено естествено число q ($q > 1$, $q < n$)" принадлежи към класа *NP*. А принадлежи ли към класа *NP*-пълни?

6.3. Търсене с връщане

Ако приемем хипотезата $P \neq NP$ за вярна (по-точен запис би бил $P \subset NP$, тъй като очевидно $P \subseteq NP$, но $P \neq NP$ се използва по-често в литературата), то единственият подход, който гарантира, че ще намираме винаги правилно решението на *NP*-пълна (или *NP*-трудна) задача, е пълното изчерпване. Възможен начин за реализация на пълно изчерпване е методът *търсене с връщане*.

Търсенето с връщане (от англ. *backtracking*) е похват, при който решението на задачата се конструира последователно (постъпково). На всяка стъпка се прави опит за *разширяване* на текущото *частично* (непълно) решение с всевъзможните налични продължения. Ако нито едно от тези разширения не доведе по-късно до *пълно* решение, случаят се обявява за безперспективен и алгоритъмът се връща една стъпка назад. Гранични случаи за алгоритъма настъпват, когато бъде намерено цялостно решение на задачата, или когато частичното решение не може да се разшири по никакъв начин. При решаване на задачи чрез търсене с връщане често се прилага следната рекурсивна схема:

```
void опит(стъпка i)
{ if (i > n) проверка_дали_сме_получили_решение;
   else
   /* разширяване на частичното решение по всички възможни начини */
   for (k = 1; k <= n; k++)
     if (k-тият_кандидат_е_приемлив) {
       регистрация_на_кандидата;
       опит(i+1);
       премахване_на_регистрацията;
     }
}
```

В предишните глави този метод намери многократно приложение. Така например, в класическата задача за намиране на Хамилтонов цикъл в граф, използвахме следния алгоритъм:

1) Започваме конструирането на Хамилтонов цикъл от произволен връх $s \in V$ и го маркираме като разгледан.
2) На всяка стъпка опитваме да преминем в нов, неразгледан връх v, инцидентен с текущия. С извършването на подобно преминаване правим *стъпка напред* — маркираме като разгледан новия връх и продължаваме по същия начин. Ако не съществува връх, в който да преминем, са възможни два случая:

   2.1) Ако всички върхове от графа са маркирани и v е инцидентен с s, то следва, че сме обходили графа и сме намерили Хамилтонов цикъл.

   2.2) Ако съществуват още необходени върхове (и понеже не можем да преминем в никой от тях), обявяваме текущото частично решение за безперспективно: то не може да бъде разширено по никакъв начин от текущия връх. Размаркираме върха, в който се намираме, и се връщаме стъпка назад към върха, от който сме дошли. Продължаваме да изпробваме останалите немаркирани, инцидентни с него върхове, в които не сме опитвали да преминем до момента.

Ако в 2.1) при намирането на един Хамилтонов цикъл не прекъснем алгоритъма, а се връщаме назад, както в 2.2), пълното изчерпване ще продължи и ще намери *всички* възможни Хамилтонови цикли в графа.

Методът *търсене с връщане* и неговите особености ще останат предмет на нашето внимание и в следващите няколко точки, където ще разгледаме още няколко *NP*-пълни и изчерпващи задачи.

Задача за упражнение:

Къде в горната схема е подходящо да се направи проверка за оптималност (например, ако се търси Хамилтонов цикъл *с минимална дължина*)?

6.3.1. Удовлетворимост на булева функция

Задачата за удовлетворимост на булева функция е в известен смисъл "първата" *NP*-пълна задача. Тя често се използва за доказателство на теоремата на Кук от *6.2.*, като чрез свеждане към нея се доказва *NP*-пълнотата на много други задачи. Ще разгледаме тази задача и ще я решим чрез търсене с връщане.

Нека са дадени булевите променливи $X_1,X_2,...,X_n$. С $\overline{X}_1, \overline{X}_2,..., \overline{X}_n$ ще означаваме отрицанията им (т. е. X_i е "истина" тогава и само тогава, когато $\overline{X}_i$ е "лъжа"). Ще използваме стандартните операции за конструиране на булеви изрази $\wedge$ и $\vee$, означаващи съответно логическо "и" (конюнкция) и логическо "или" (дизюнкция), както и стойностите 0 и 1 за означаване съответно на "лъжа" и "истина". Ще позволяваме използване на черта на отрицание върху цели изрази (виж законите на Де Морган по-долу). Понякога отрицанието ще обозначаваме със знака $\neg$, поставен пред израз, например: $\neg (A \vee B)$. Конюнкцията $A \wedge B$ често се записва като $A.B$ или дори AB.

В *таблица 6.3.1.* са показани възможните резултати от логическите операции $\wedge$ и $\vee$ върху два булеви израза.

x	y	$x \wedge y$	$x \vee y$
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Таблица 6.3.1. Таблица на истинност.

Дефиниция 6.4. Един булев израз на променливите $X_1,X_2,...,X_n$ се нарича *удовлетворим*, ако съществува набор от стойности "истина" или "лъжа" за $X_1,X_2,...,X_n$, за който стойността на израза да бъде "истина".

Задача: Да се провери дали даден израз е удовлетворим.

Така например, изразът $(\overline{X}_2 \vee X_2 \wedge X_1) \wedge \overline{X}_3$ е удовлетворим за $X_1 = 1$, $X_2 = 0$, $X_3 = 0$ или за $X_1 = 1$, $X_2 = 1$, $X_3 = 0$. Проверката, дали даден израз е удовлетворим, е *NP*-задача: ако е дадено някакво присвояване на стойности на променливите, можем, както ще се види след малко, с полиномиална сложност да проверим дали то е решение, но за самото *намиране* на това присвояване не е известен полиномиален алгоритъм.

Преди да преминем към решението на задачата, ще припомним някои елементи от математическата логика.

Дефиниция 6.5. Константите 0 и 1, както променливите и техните отрицания се наричат *литерали*.

Дефиниция 6.6. *Дизюнкт* се нарича булев израз, в който участва само операцията "или".

Дефиниция 6.7. Булев израз, в който участват само дизюнкти, обединени с операцията "и", се нарича израз в *конюнктивна нормална форма*.

С многократно прилагане на тъждеството

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

произволен булев израз може да се представи в конюнктивна нормална форма.

Други полезни правила за преобразуване на израз се дават от законите на Де Морган:

$$\overline{A \vee B} = \overline{A} \wedge \overline{B}$$
$$\overline{A \wedge B} = \overline{A} \vee \overline{B}$$

Нека разгледаме един примерен израз, записан в конюнктивна нормална форма:

$$(X_1 \vee X_4) \wedge (\overline{X}_1 \vee X_2) \wedge (X_1 \vee \overline{X}_3) \wedge (\overline{X}_2 \vee X_3 \vee \overline{X}_4) \wedge (\overline{X}_1 \vee \overline{X}_2 \vee \overline{X}_3)$$

За да проверим дали изразът е удовлетворим, можем да приложим следния комбинаторен подход: изпробваме възможните 2^n на брой присвоявания на стойности на променливите и за всяко присвояване проверяваме дали изразът е удовлетворен, със сложност $\Theta(2^n)$. Така, алгоритъмът за търсене с връщане добива следния вид:

Инициализираме $i = 1$.

1) Присвояваме на i-тата променлива стойност "истина" и преминаваме към (i+1)-вата променлива (правим стъпка напред). Ако това присвояване по-късно не доведе до решение, се връщаме (стъпка назад), присвояваме на променливата стойност "лъжа" и отново опитваме да продължим.
2) Граничен случай е налице, когато на всички променливи е присвоена някаква стойност — тогава се пресмята стойността на целия израз и се проверява дали тя е "истина".

В реализацията на описания алгоритъм булевият израз, приведен в конюнктивна нормална форма, ще представим с двумерен масив — на всеки дизюнкт се съпоставя един ред от масива. Булевите променливите от един ред (дизюнкт) ще представяме с техните индекси — целите числа от 1 до n, а за отрицание на променливата X_i ще използваме числото $-i$. В масива `values[]` се намират стойностите (0 или 1), които сме присвоили на всяка една променлива. Функцията `true()` пресмята стойността на булевия израз при направените във `values[]` присвоявания. Тази проверка се извършва лесно: За да има целият израз стойност "истина", трябва *всеки един* от дизюнктите да има стойност "истина" (тъй като операцията между дизюнктите е "и"). За да има един дизюнкт стойност "истина", *поне една* от променливите в него трябва да има стойност "истина" или отрицание на променлива да има стойност "лъжа" (тъй като всички литерали в един дизюнкт са обединени с операцията "или").

Присвояване на стойностите на променливите в алгоритъма по-горе, извършва функцията `assign(unsigned i)`, като в граничния случай ($i = n$) се извиква функцията `true()`, за да провери дали изразът е удовлетворен (т. е. да провери дали присвояването е решение на задачата). Следва пълна реализация:

```
#include <stdio.h>

#define MAXN 100 /* Максимален брой булеви променливи */
#define MAXK 100 /* Максимален брой дизюнкти */

const unsigned n = 4;  /* Брой на булевите променливи */
const unsigned k = 5;  /* Брой на дизюнктите */

const int expr[][MAXK] = {
  {  1,  4 },
  { -1,  2 },
  {  1, -3 },
  { -2,  3, -4 },
  { -1, -2, -3 }
};

int values[MAXN];

void printAssignment(void)
{ unsigned i;
  printf("Изразът е удовлетворим за: ");
  for (i = 0; i < n; i++) printf("X%u=%u ", i+1, values[i]);
  printf("\n");
}
```

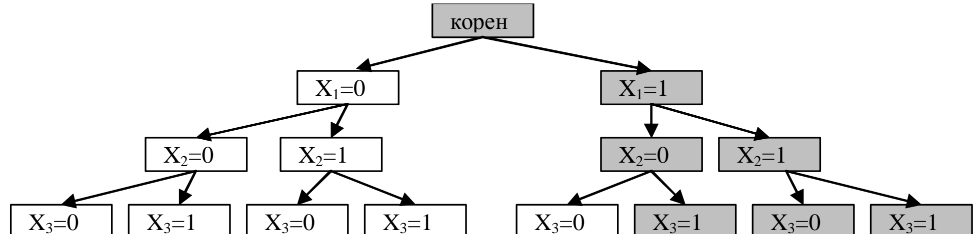
```
/* поне един литерал трябва да има стойност "истина" във всеки дизюнкт */
int true(void)
{ unsigned i;
  for (i = 0; i < k; i++) {
    unsigned j = 0;
    char ok = 0;
    while (expr[i][j] != 0) {
      int p = expr[i][j];
      if ((p > 0) && (1 == values[p-1])) { ok = 1; break; }
      if ((p < 0) && (0 == values[-p-1])) { ok = 1; break; }
      j++;
    }
    if (!ok) return 0;
  }
  return 1;
}

/* присвоява стойности на променливите */
void assign(unsigned i)
{ if (i == n) {
    if (true()) printAssignment();
    return;
  }
  values[i] = 1;  assign(i + 1);
  values[i] = 0;  assign(i + 1);
}

int main(void) {
  assign(0);
  return 0;
}
```

bool.c

Резултат от изпълнението на програмата:

```
Изразът е удовлетворим за: X1=1 X2=1 X3=0 X4=0
Изразът е удовлетворим за: X1=0 X2=0 X3=0 X4=1
```

Ще илюстрираме как се извършва присвояването в пример с 3 булеви променливи: *фигура 6.3.1.* — с по-тъмно са отбелязани върховете, които водят до удовлетворяване на израза $X_1 \wedge (X_2 \vee X_3)$.



Фигура 6.3.1. Удовлетворимост на булев израз.

При *NP*-пълните задачи (и в частност при задачата за удовлетворимост), кандидатите за решение по неявен начин формират дърво (или граф), което алгоритъмът за търсене с връщане обхожда. Често обаче изследването на голяма част от това дърво се оказва излишно. Например,

за израза $X_1 \wedge (X_2 \vee X_3)$ при присвояване $X_1 = 0$ веднага се забелязва, че няма да достигнем до решение, независимо от стойностите, които ще получат останалите две променливи X_2 и X_3. Когато променливите са повече и булевият израз е по-сложен, такова отсичане на изследваното дърво може значително да повиши бързодействието на програмата.

Отсичането ще извършим с помощта на следните модификации в горната програма:

1) На функцията `true()` се добавя нов аргумент `t`, означаващ, че са присвоени стойности само на първите `t` променливи. Така, ако даден дизюнкт е съставен само от променливи с номера, по-малки или равни на `t`, и нито една от тях няма стойност "истина", то целият булев израз ще бъде "лъжа" и няма смисъл да се присвояват стойности на останалите `n-t` променливи. Ако в дизюнкта има променливи, на които още не е присвоена стойност, то неговата стойност все още не може да се определи и процесът на присвояване продължава.
2) При генерирането се извиква функцията `true()` в началото на всяка стъпка. Така, понататъшното генериране се прекъсва веднага, щом стане безсмислено — стойността на булевия израз ще бъде "лъжа", независимо от това, как ще се извършат останалите присвоявания.

```
#include <stdio.h>

/* Максимален брой булеви променливи */
#define MAXN 100

/* Максимален брой дизюнкти */
#define MAXK 100

const unsigned n = 4;  /* Брой на булевите променливи */
const unsigned k = 5;  /* Брой на дизюнктите */

const int expr[][MAXK] = {
  {  1,  4 },
  { -1,  2 },
  {  1, -3 },
  { -2,  3, -4 },
  { -1, -2, -3 }
};

int values[MAXN];

void printAssign(void)
{ unsigned i;
  printf("Изразът е удовлетворим за: ");
  for (i = 0; i < n; i++) printf("X%d=%d ", i + 1, values[i]);
  printf("\n");
}

/* поне един литерал трябва да има стойност "истина" във всеки дизюнкт */
int true(int t)
{ unsigned i;
  for (i = 0; i < k; i++) {
    unsigned j = 0;
    char ok = 0;
    while (expr[i][j] != 0) {
      int p = expr[i][j];
      if ((p > t) || (-p > t)) { ok = 1; break; }
      if ((p > 0) && (1 == values[p-1])) { ok = 1; break; }
      if ((p < 0) && (0 == values[-p-1])) { ok = 1; break; }
```

```
        j++;
      }
      if (!ok) return 0;
    }
    return 1;
  }

  /* присвоява стойности на променливите */
  void assign(unsigned i)
  { if (!true(i)) return;
    if (i == n) {
      printAssign();
      return;
    }
    values[i] = 1; assign(i + 1);
    values[i] = 0; assign(i + 1);
  }

  int main(void) {
    assign(0);
    return 0;
  }
```

🖫 boolcut.c

Задачи за упражнение:

1. Да се предложат други критерии за отсичане на дървото на кандидатите за решение.
2. Удовлетворим ли е изразът $(\overline{X}_2 \vee X_3 \wedge X_1) \wedge X_2 \wedge (\overline{X}_1 \vee \overline{X}_3)$?
3. Да се докаже тъждеството $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$, като се използват таблиците на истинност.
4. Да се докажат законите на Де Морган.
5. Като се използват законите на Де Морган да се докаже равенството $A \vee B = \overline{\overline{A} \wedge \overline{B}}$.
6. Равенството от предходната задача изразява дизюнкцията чрез конюнкция и отрицание. Да се предложи и докаже формула, която изразява *конюнкцията* чрез дизюнкция и отрицание.
7. Като се използват предходните две задачи, да се докаже, че множествата $\{\neg, \wedge\}$ и $\{\neg, \vee\}$ имат същата изразителност като $\{\neg, \wedge, \vee\}$.
8. Вярно ли е, че множеството $\{\wedge, \vee\}$ има същата изразителност като $\{\neg, \wedge, \vee\}$?

6.3.2. Оцветяване на граф

В *5.8.1.* дефинирахме задачата за минимално оцветяване на граф. Тя принадлежи на класа *NP*-трудни задачи и в настоящия параграф ще я решим, като използваме метода на търсене с връщане. Връзката между *NP*-трудната задача за минималното оцветяване и *NP*-пълната за проверка дали граф може да се *r*-оцвети се вижда лесно:

Нека означим броя на върховете в графа с n. За всяко r ($1 \leq r \leq n$) ще извършваме проверка дали графът може да се оцвети с r цвята. Така решението на задачата за минимално оцветяване се свежда полиномиално до *r*-оцветяване на граф.

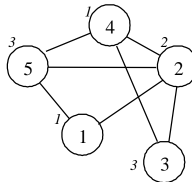
Да разгледаме следния алгоритъм, за проверка за *r*-оцветяване:

- Започваме оцветяването от първия връх на графа: $i = 1$.
- Стъпка i: Оцветяваме i-тия връх на графа с първия цвят и преминаваме към оцветяване на $(i+1)$-вия връх, след това оцветяваме i-тия връх на графа с втория цвят и преминаваме към оцветяване на $(i+1)$-вия и т.н.

- Ако всички върхове са оцветени (т. е. $i = n$), то попадаме в граничен случай. Проверяваме дали има съседни върхове с еднакъв цвят и, ако няма, то сме намерили решение.

Лесно се забелязва, че горният алгоритъм е крайно неефективен: сложността му е $\Theta\left(\sum_{r=1}^{n} r^n\right) = \Theta(n^n)$. Проблемът е същият, както в задачата за удовлетворимост и се състои в следното: ако още в началото оцветим два съседни върха с еднакъв цвят, ще продължим оцветяването, въпреки че то със сигурност няма да доведе до решение. Този недостатък на алгоритъма може да се преодолее, ако при оцветяването на i-тия връх проверяваме дали някой от съседните му вече не е оцветен със същия цвят. Ако това е така, то, вместо да преминем към (i+1)-вия връх, ще пробваме да оцветим i-тия с друг цвят. Ако не успеем да намерим оцветяване за i-тия връх измежду r-те цвята, се връщаме стъпка назад. При това връщане, ако $i = 1$ следва, че r-оцветяване на графа не е възможно (налага се да увеличим броят на цветовете).

Следва изходният код на програмата. Графът е представен с матрица на съседство `A[][]`, оцветяването на върховете се записва в масива `col[]`, а рекурсивната функция, опитваща да оцвети i-тия връх на графа, е `nextCol(unsigned i)`. Примерните входни данни са показани на *фигура 6.3.2*.



Фигура 6.3.2. Минимално r-оцветяване на граф.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 200

/* Брой върхове в графа */
const unsigned n = 5;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
  { 0, 1, 0, 0, 1 },
  { 1, 0, 1, 1, 1 },
  { 0, 1, 0, 1, 0 },
  { 0, 1, 1, 0, 1 },
  { 1, 1, 0, 1, 0 }};

unsigned col[MAXN], maxCol, count = 0;
char foundSol = 0;

void showSol(void)
{ unsigned i;
  count++;
  printf("Минимално оцветяване на графа: \n");
  for (i = 0; i < n; i++)
    printf("Връх %u - с цвят %u \n", i + 1, col[i]);
}

void nextCol(unsigned i)
{ unsigned k, j, success;
```

```
  if (i == n) { showSol(); return; }
  for (k = 1; k <= maxCol; k++) {
    col[i] = k;
    success = 1;
    for (j = 0; j < n; j++)
      if (1 == A[i][j] && col[j] == col[i]) {
        success = 0;
        break;
      }
    if (success) nextCol(i + 1);
    col[i] = 0;
  }
}

int main(void) {
  unsigned i;
  for (maxCol = 1; maxCol <= n; maxCol++) {
    for (i = 0; i < n; i++) col[i] = 0;
    nextCol(0);
    if (count) break;
  }
  printf("Общ брой намерени оцветявания с %u цвята: %u \n",maxCol,count);
  return 0;
}
```

🖫 colorm2.c

Резултат от изпълнението на програмата:

```
Минимално оцветяване на графа:
Връх 1 - с цвят 1
Връх 2 - с цвят 2
Връх 3 - с цвят 3
Връх 4 - с цвят 1
Връх 5 - с цвят 3
Минимално оцветяване на графа:
Връх 1 - с цвят 1
Връх 2 - с цвят 3
Връх 3 - с цвят 2
Връх 4 - с цвят 1
Връх 5 - с цвят 2
Минимално оцветяване на графа:
Връх 1 - с цвят 2
Връх 2 - с цвят 1
Връх 3 - с цвят 3
Връх 4 - с цвят 2
Връх 5 - с цвят 3
Минимално оцветяване на графа:
Връх 1 - с цвят 2
Връх 2 - с цвят 3
Връх 3 - с цвят 1
Връх 4 - с цвят 2
Връх 5 - с цвят 1
Минимално оцветяване на графа:
Връх 1 - с цвят 3
Връх 2 - с цвят 1
Връх 3 - с цвят 2
Връх 4 - с цвят 3
Връх 5 - с цвят 2
```

```
Минимално оцветяване на графа:
Връх 1 - с цвят 3
Връх 2 - с цвят 2
Връх 3 - с цвят 1
Връх 4 - с цвят 3
Връх 5 - с цвят 1
Общ брой намерени оцветявания с 3 цвята: 6
```

Задачи за упражнение:

1. Да се предложи алгоритъм за минимално оцветяване на върховете на граф, който намира минималния брой цветове *директно*, т.е. без да проверява последователно за всяко r ($1 \leq r \leq n$) дали графът може да се оцвети с r цвята.
2. Да се предложат още критерии за отсичане на дървото от кандидатите за решение.
3. Да се предложи и реализира алгоритъм за минимално оцветяване на *ребрата* на граф.

6.3.3. Най-дълъг прост път в цикличен граф

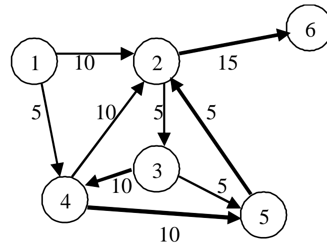
Досега неколкократно разглеждахме интерпретации на задачата за намиране на най-дълъг прост път в ориентиран претеглен цикличен граф (дължината на пътя се пресмята като сума от ребрата, които съдържа). По-долу ще предложим алгоритъм на принципа търсене с връщане.

На всяка стъпка опитваме да разширим частично построения път с още един връх. Нека j е крайният връх на пътя, построен до момента. Към пътя последователно добавяме всеки съседен на j и неучастващ в пътя връх. Ако в даден момент не можем да намерим такъв връх, то следва, че сме попаднали в граничен случай. Тогава пресмятаме дължината на пътя и, ако тя се окаже по-голяма от максималната, намерена до момента, го запазваме:

```
void добавянеНаВръх(i) {
   if (сме_намерили_по_дълъг_път) { запазваме_го и прекратяваме; }
   for (всеки_наследник_k_на_i)
     if ((пътят_не_съдържа_k) &&
         (k_е_съсед_на_последния_връх_в_пътя))
     {
        добавяме_върха_k_към_пътя;
        прибавяме_дължината_на_новото_ребро_към_дължината_на_пътя;
        добавянеНаВръх(i+1);
        /* връщане от рекурсията */
        размаркираме_k_като_участващ_в_пътя;
        премахваме_върха_k_от_пътя;
        изваждаме_от_дължината_на_пътя_дължината_на_новодобавеното_ребро;
     }
   }
}
```

Реализацията следва горната схема. В нея обаче не е уточнено кои са стартовите върхове. Ясно, е че не може да се избира за стартов само връх без предшественици (когато са налице отрицателни ребра е възможно да бъде пропуснато оптимално решение). Т.е. търсенето трябва да започва от всеки възможен връх. В реализацията по-долу ще извикваме `addVertex()` последователно за всеки от върховете на графа.

Графът е представен с матрица на съседство `A[][]`, а примерните входни данни са показани на *фигура 6.3.3*.



Фигура 6.3.3. Най-дълъг прост път в ориентиран граф.

```
#include <stdio.h>

/* Максимален брой върхове в графа */
#define MAXN 200
/* Брой върхове в графа */
const unsigned n = 6;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
  { 0, 10, 0,  5,  0,  0 },
  { 0,  0, 5,  0,  0, 15 },
  { 0,  0, 0, 10,  5,  0 },
  { 0, 10, 0,  0, 10,  0 },
  { 0,  5, 0,  0,  0,  0 },
  { 0,  0, 0,  0,  0,  0 }};

unsigned vertex[MAXN], savePath[MAXN];
char used[MAXN];
int maxLen, tempLen, si, ti;

void addVertex(unsigned i)
{ unsigned j, k;
  if (tempLen > maxLen) { /* намерили сме по-дълъг път => запазваме го */
    maxLen = tempLen;
    for (j = 0; j <= ti; j++) savePath[j] = vertex[j];
    si = ti;
  }
  for (k = 0; k < n; k++) {
    if (!used[k]) {       /* ако върхът k не участва в пътя до момента */
      /* ако върхът, който добавяме, е съседен на последния от пътя */
      if (A[i][k] > 0) {
        tempLen += A[i][k];
        used[k] = 1;    /* маркираме k като участващ в пътя */
        vertex[ti++] = k;  /* добавяме върха k към пътя */
        addVertex(k);
        used[k] = 0;    /* връщане от рекурсията */
        tempLen -= A[i][k]; ti--;
      }
    }
  }
}

int main(void) {
  unsigned i;
  maxLen = 0; tempLen = 0; si = 0; ti = 1;
  for (i = 0; i < n; i++) used[i] = 0;
  for (i = 0; i < n; i++) {
    used[i] = 1; vertex[0] = i;
```

```
    addVertex(i);
    used[i] = 0;
  }
  printf("Най-дългият път е: \n");
  for (i = 0; i < si; i++) printf("%u ", savePath[i] + 1);
  printf("\nс обща дължина %d\n", maxLen);
  return 0;
}
```

🖫 longpath.c

Резултат от изпълнението на програмата:

```
Най-дългият път е:
3 4 5 2 6
с обща дължина 40.
```

Задачи за упражнение:

1. Променя ли се принципно задачата за намиране на най-дълъг прост път в ориентиран претеглен цикличен граф, ако дължината на пътя се пресмята като *произведение* от теглата на ребрата, които съдържа. А ако дължината се определя от теглото на минималното (максималното) ребро, участващо в пътя?

2. Променя ли се принципно задачата за намиране на най-дълъг прост път в ориентиран цикличен граф, ако дължината на пътя се пресмята като сума от теглата на *върховете*, които съдържа (приемаме, че за всеки връх е зададено тегло — естествено число).

3. Да се модифицира горната програма така, че да отпечатва *всички* пътища с максимална дължина.

6.3.4. Разходка на коня

Една широко разпространена задача, илюстрираща добре метода търсене с връщане, е известна под името “*разходката на коня*”:

	•		•	
•				•
		К		
•				•
	•		•	

Фигура 6.3.4а. Допустими ходове на коня.

8	22	7	44	39	24	9	28	63
7	43	40	23	8	45	62	25	10
6	6	21	42	59	38	27	64	29
5	41	58	37	46	61	54	11	26
4	20	5	60	53	36	47	30	51
3	57	2	35	48	55	52	15	12
2	4	19	56	33	14	17	50	31
1	1	34	3	18	49	32	13	16
	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*

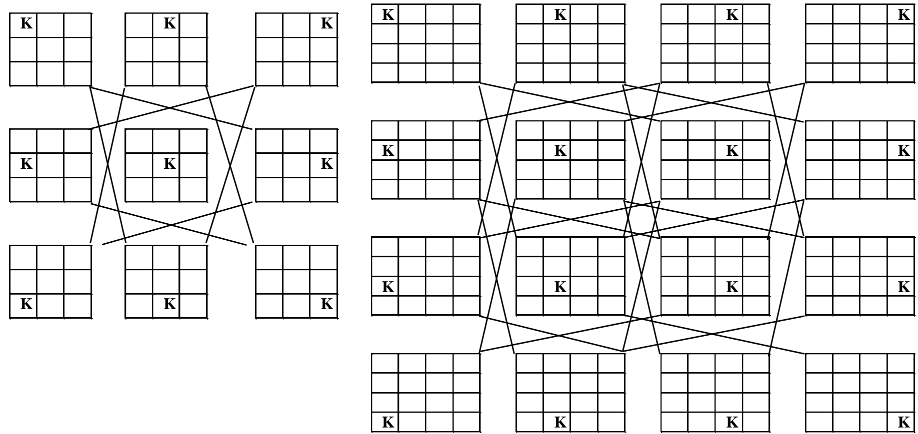
Фигура 6.3.4б. Решение на задачата за $n = 8$.

Задача: Дадено е естествено число n ($n > 4$) и обобщена шахматна дъска с размер $n \times n$ клетки. Да се намери обхождане на дъската с хода на коня. Всяка клетка от дъската трябва да

бъде посетена точно веднъж, като “обиколката” започва от клетката, намираща се в долния ляв ъгъл. Допустими ходове за коня са стандартните от шахмата, както е показано на *фигура 6.3.4а*.

Решение на задачата за $n = 8$ е показано на *фигура 6.3.4б.*

На всяка позиция на коня върху дъската (общо n^2 на брой) може да се съпостави един връх на неориентиран граф. Ребро в този граф ще свързва два върха, ако е възможно преминаването между съответните позиции с ход на коня. Ето как изглеждат подобни графи за $n = 3$ и $n = 4$ (*фигура 6.3.4в*):



Фигура 6.3.4в. Графи на задачата за $n = 3$ и $n = 4$.

За да намерим търсеното обхождане на дъската, трябва да намерим път в графа, посещаващ всеки връх точно веднъж, т. е. Хамилтонов път. Така получаваме, че

"Задачата за разходка на коня" 〈 *"Хамилтонов път"*

Нещо повече, задачата за разходката на коня може да се сведе до задача за намиране на Хамилтонов път в граф, в който степента на всеки връх d_i е ограничена, в случая $d_i \leq 8$ (*виж задачи 5, 7* и *8* в края на параграфа).

Разбира се, няма да решаваме задачата с граф, първо, защото съществува доста по-ясен алгоритъм, и второ, защото подобен граф с n^2 върха ще изисква приблизително $8.n^2$ памет (за представяне чрез списък на наследниците. При други представяния ще бъде необходима дори повече памет — *виж таблица 5.2а.*). Отново ще използваме търсене с връщане, което схематично се описва със следната рекурсивна функция:

```
/* board[][] е дъската с размери n×n */
void следващХод(x, y, i)
{ if (i == n*n) { /* Печатаме решението */ }
  board[x][y] = i;
  for (всеки_възможен_ход_(u,v)_на_коня_от_клетка_(x,y))
    if (0 == board[u][v]) /* ако полето е свободно */
      следващХод(u, v, i+1);
  board[x][y] = 0; /* връщане */
}
```

В реализацията, която следва, е разгледано обобщение на задачата. Първо, обиколката може да започне от произволна (предварително зададена с координатите `startX` и `startY`) клетка. В допълнение, могат да се предефинират допустимите ходове за коня, т.е. те може да не бъдат стандартните, показани на *фигура 6.3.4а.*, а произволни други. Валидните ходове за коня се задават в началото на програмата в константните масиви `diffX[]` и `diffY[]` — стойностите в

тях означават, че конят може да премине от клетката (x,y) във всяка друга клетка (x+`diffX[i]`, y+`diffY[i]`), за i = 1,2, ..., `maxDiff`:

```
#include <stdio.h>
#include <stdlib.h>

/* Максимален размер на дъската */
#define MAXN 10

/* Максимален брой правила за движение на коня */
#define MAXD 10

/* Размер на дъската */
const unsigned n = 6;

/* Стартова позиция */
const unsigned startX = 1;
const unsigned startY = 1;

/* Правила за движение на коня */
const unsigned maxDiff = 8;
const int diffX[MAXD] = { 1, 1, -1, -1, 2, -2, 2, -2 };
const int diffY[MAXD] = { 2, -2, 2, -2, 1, 1, -1, -1 };

unsigned board[MAXN][MAXN];
unsigned newX, newY;

void printBoard(void)
{ unsigned i, j;
  for (i = n; i > 0; i--) {
    for (j = 0; j < n; j++) printf("%3u", board[i-1][j]);
    printf("\n");
  }
  exit(0);  /* изход от програмата */
}

void nextMove(unsigned X, unsigned Y, unsigned i)
{ unsigned k;
  board[X][Y] = i;
  if (i == n * n) { printBoard(); return; }
  for (k = 0; k < maxDiff; k++) {
    newX = X + diffX[k]; newY = Y + diffY[k];
    if ((newX >= 0 && newX < n && newY >= 0 && newY < n) &&
        (0 == board[newX][newY]))
            nextMove(newX, newY, i + 1);
  }
  board[X][Y] = 0;
}
int main(void) {
  unsigned i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) board[i][j] = 0;
  nextMove(startX-1, startY-1, 1);
  printf("Задачата няма решение. \n");
  return 0;
}
```

knight.c

Резултат от изпълнението на програмата:
```
 10 27  6 19 16 25
  7 20  9 26  5 18
 28 11  4 17 24 15
 21  8 23 32  3 34
 12 29  2 35 14 31
  1 22 13 30 33 36
```

Забележка: Задачата за обхождане на шахматна дъска с ход на коня се разглежда често в литературата като класически пример за приложение на метода на търсене с връщане, въпреки, че не е *NP*-пълна, т.е. за нея съществува полиномиално решение (от това следва, че съществува полиномиално решение и за намиране на Хамилтонов път в специалния вид граф, показан на *фигура 6.3.4в.*). Този алгоритъм (с него ще се запознаем подробно в *глава 9* — Алчни алгоритми) найобщо се основава на следната схема: за всеки пореден ход на коня се избира тази клетка, от която след това ще има *най-малко възможности за следващ ход*.

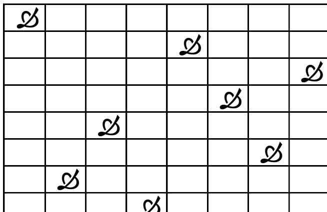
С малки модификации на задачата обаче, можем лесно да се озовем в ситуация, където изобщо не можем да бъдем сигурни дали задачата е полиномиална. Такива примери са следните задачи:

Задачи за упражнение:

1. Да се модифицира горната програма така, че да намира *всички* решения.
2. Да се модифицира горната програма така, че да намира *всички различни несиметрични* решения. За симетрични ще считаме решения, които могат да се получат едно от друго чрез завъртане или чрез симетрия относно хоризонталите, вертикалите или двата главни диагонала на дъската.
3. Да се намери броят на различните обхождания на дъската:
   а) всички обхождания
   б) всички различни несиметрични обхождания
4. Дадено е естествено число n ($n > 4$) и обобщена шахматна дъска с размер $n \times n$ клетки. Да се намери обхождане на дъската с хода на коня. Всяка клетка трябва да бъде посетена точно веднъж, като "обиколката" започва от *произволна* предварително зададена клетка от дъската.
5. Дадено е естествено число n ($n > 4$) и обобщена шахматна дъска с размер $n \times n$ клетки, някои от които са *забранени*. Да се намери обхождане на дъската с хода на коня. Всяка незабранена клетка от дъската трябва да бъде посетена точно веднъж, като "обиколката" започва от клетката, намираща се в долния ляв ъгъл.
6. Дадено е естествено число n ($n > 4$) и обобщена шахматна дъска с размер $n \times n$ клетки. Да се намери обхождане на дъската с хода на коня с минимално самопресичане на траекторията на коня.
7. Да се определи класът на задачата за намиране на Хамилтонов път в граф с ограничена степен на върховете (т.е. степента на всеки връх е по-малка от фиксирано естествено число k).
8. Възможно ли е *задача 7* да се сведе полиномиално до *задача 5*?

6.3.5. Задача за осемте царици

Проблемът за разполагане на царици върху шахматна дъска е класическа задача. При нея отново проличава колко важно за ефективното прилагане на пълно изчерпване е подходящото отсичане на дървото на рекурсията. Гаус разглежда тази задача още през 1850 г. и постига успех точно поради факта, че с аналитично изследване кандидатите за решение могат да намалеят значително. Затова целта ни ще бъде да стесним максимално диапазона за изследване и едва след това да извършваме пълно изчерпване.



Фигура 6.3.5. Решение за $n = 8$.

Задача: Да се разположат n царици върху обобщена шахматна дъска с размери $n \times n$ ($n \geq 2$) така, че никои две да не се намират под удар (да не се намират в един и същ хоризонтал, вертикал или диагонал).

Едно възможно решение на задачата за $n = 8$ е показано на *фигура 6.3.5*. Ще решим задачата чрез търсене с връщане по следната схема:

Инициализираме $i = 1$.

- Ако до момента сме разположили успешно i царици, опитваме да намерим "подходящо" поставяне на (i+1)-вата царица (стъпка напред).
- Ако сме поставили n царици (т. е. $i = n$) и те изпълняват условието на задачата, то следва, че сме намерили решение, отпечатваме го, след което евентуално прекратяваме пресмятането.
- Ако не съществува свободна позиция за i-тата царица, се връщаме стъпка назад и търсим друго разположение за (i–1)-вата.

Ще поясним какво се има предвид под "*подходящо*" поставяне по-горе. Едно неефективно решение на задачата е да се изпробват *всичките* възможни комбинации $C_{n^2}^{n}$ от поставяния на царици върху дъската и да се отхвърлят тези, които не са решение на задачата. Тъй като във всеки *ред* може да стои най-много една царица, то могат да се изпробват само различните n^n поставяния на царици, по една във всеки ред, и измежду тях да се търси решение. Ако се вземе предвид и, че в един *стълб* може да стои най-много една царица, то броят на възможностите за генериране намалява до n!. Ще въведем n-елементен масив, съдържащ номера на колони — на i-та позиция е записан номерът на колоната, в която е разположена царицата от i-тия ред. Всяка такава пермутация ще определя еднозначно разположение на цариците върху дъската. Конфигурациите за проверка могат да се намалят и още — при разположението на всяка поредна царица ще се избират само безопасни позиции, т. е. такива, които не се намират в един и същ хоризонтал, вертикал или диагонал с вече поставена царица. Ето как изглежда схематично описаният алгоритъм:

```
void поредна Царица(i) { /* поставяне на царица в i-тия ред */
  if (i > n) { Печатаме решението; }
  for (всеки_незает_от_царица_стълб_k)
    if (няма_царица_върху_диагонал_през_(i,k)) {
      /* позицията (i,k) е безопасна */
      Поставяме_i-тата_царица_на_позиция_(i,k);
      поредна Царица(i+1);
      Отстраняваме_i-тата_царица_от_позиция_(i,k);
    }
}
```

При реализацията за маркирането и проверката дали дадена позиция е безопасна, ще използваме четири едномерни масива:

1) `queens[N]`: `queens[i]` съдържа номера на стълба, в който е разположена царицата от i-я ред.

2) `col[N]`: `col[i]` има стойност 1, ако в стълба *i* няма поставена царица, и 0 — в противен случай.
3) `RD[2*N-1]` и `LD[2*N-1)]` показват дали на *k*-тия главен и съответно вторичен диагонал ($k = 1,2,...,2n-1$) има царица. При това:
   - `RD[i+k]` показва дали в главния диагонал, минаващ през клетката (*i*,*k*), има разположена царица.
   - `LD[N+i-k]` показва дали във вторичния диагонал, минаващ през клетката (*i*,*k*), има разположена царица.

Програмата по-долу завършва при намиране на едно решение. Предоставяме на читателя да се опита да я модифицира така, че да намира и извежда *всички* възможни решения за дадено *n*.

```
#include <stdio.h>
#include <stdlib.h>

/* Максимален размер на дъската */
#define MAXN 100
/* Размер на дъската */
const unsigned n = 13;

unsigned col[MAXN] , RD[2*MAXN - 1],
         LD[2*MAXN], queens [MAXN];

/* Отпечатва намереното разположение на цариците */
void printBoard()
{ unsigned i , j ;
  for (i = 0; i < n; i++) {
    printf("\n");
    for (j = 0; j < n; j++)
      if (queens[i] == j) printf("x ");
      else printf(". ");
  }
  printf("\n");
  exit(0);
}

/* Намира следваща позиция за поставяне на царица */
void generate(unsigned i)
{ unsigned j;
  if (i == n) printBoard();
  for (j = 0; j < n; j++)
    if (col[j] && RD[i + j] && LD[n + i - j]) {
      col[j] = 0; RD[i + j] = 0; LD[n + i - j] = 0; queens[i] = j;
      generate(i + 1);
      col[j] = 1; RD[i + j] = 1; LD[n + i - j] = 1;
    }
}

int main(void) {
  unsigned i;
  for (i = 0; i < n; i++) col[i] = 1;
  for (i = 0; i < (2*n - 1); i++) RD[i] = 1;
  for (i = 0; i < 2*n; i++) LD[i] = 1;
  generate(0);
  printf("Задачата няма решение! \n");
  return 0;
}
```

queens.c

Резултат от изпълнението на програмата:

```
x . . . . . . . . . . . .
. . x . . . . . . . . . .
. . . . x . . . . . . . .
. x . . . . . . . . . . .
. . . . . . . . x . . . .
. . . . . . . . . . . x .
. . . . . . . . . x . . .
. . . . . . . . . . . . x
. . . x . . . . . . . . .
. . . . . x . . . . . . .
. . . . . . . x . . . . .
. . . . . . . . . . x . .
. . . . . . x . . . . . .
```

Възможни са и допълнителни оптимизации на алгоритъма, които обаче няма да намалят сложността му $\Theta(n!)$ [*Наков-1998*][*Рейнгольд, Нивергельт, Део-1980*].

Задачи за упражнение:

1. Да се предложи и реализира алгоритъм за решаване на следната разновидност на задачата: Да се отпечатат всички възможни *несиметричните* разположения на цариците върху дъската. Две решения се смятат за симетрични, ако едното може да се получи от другото чрез завъртане на шахматната дъска или симетрия относно редовете, стълбовете, главния или вторичния диагонал.

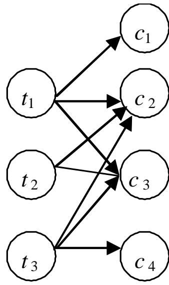
2. Има ли смисъл (и ако да — при какви допълнителни уточнения) задачата за обобщения случай на правоъгълна дъска?

6.3.6. Разписание на училищна програма

Съставянето на разписание е друг пример за задача, решавана по метода на търсене с връщане и пълно изчерпване. Без да разглеждаме в детайли този обширен въпрос (повече информация читателят би могъл да намери в [*Christofides-1975*]), ще се опитаме да го изясним най-общо с разглеждането на опростен вариант на задачата за съставяне на училищна програма:

Задача: Дадени са c класа и t учители (c, t — естествени числа). Известен е броят часове `cl[i][j]` (естествено число, $i = 1,2,...,c$, $j = 1,2,...t$), които учителят j трябва да изнесе пред класа i. Да се намери разписание за училищната програма, при което да бъдат изпълнени условията:

1) Във всеки учебен час един учител може да преподава най-много на един клас.
2) Общата продължителност на програмата да бъде минимална.



Фигура 6.3.6а. Разписание на училищна програма.

Да разгледаме следния пример: Дадени са 4 класа c_1, c_2, c_3 и c_4 и три учителя t_1, t_2 и t_3. Часовете, които трябва да се проведат, са:

Учител t_1 на клас c_1 — 5 часа, t_1 на c_2 — 5 часа, t_1 на c_3 — 5 часа,
Учител t_2 на клас c_2 — 5 часа, t_2 на c_3 — 5 часа,
Учител t_3 на клас c_2 — 5 часа, t_3 на c_3 — 5 часа, t_3 на c_4 — 5 часа.

Тази конфигурация сме представили схематично с ориентиран двуделен граф на *фигура 6.3.6а.*: Не сме посочили тегла на ребрата: за избрания пример всички тегла ще бъдат равни на 5. С $t_i \rightarrow c_j(x)$ ще означаваме, че учителят t_i преподава x часа на класа c_j. Така, за избрания пример са възможни следните две разписания:

Вариант 1.

1) $t_1 \rightarrow c_2(5)$, $t_3 \rightarrow c_3(5)$, t_2 почива, тъй като и двата класа c_2 и c_3 са заети.
2) $t_1 \rightarrow c_3(5)$, $t_3 \rightarrow c_2(5)$, t_2 почива.
3) $t_2 \rightarrow c_2(5)$, $t_1 \rightarrow c_1(5)$, $t_3 \rightarrow c_4(5)$
4) $t_2 \rightarrow c_3(5)$

Програмата е изпълнена и общата ѝ продължителност е 20 часа.

Вариант 2.

1) $t_1 \rightarrow c_1(5)$, $t_2 \rightarrow c_2(5)$, $t_3 \rightarrow c_3(5)$
2) $t_1 \rightarrow c_2(5)$, $t_2 \rightarrow c_3(5)$, $t_3 \rightarrow c_4(5)$
3) $t_1 \rightarrow c_3(5)$, $t_3 \rightarrow c_2(5)$, t_2 почива.

Тук програмата е изпълнена за 15 часа. Вижда се, че общата продължителност зависи от реда на преподаване.

Двата варианта са показани в *таблица 6.3.6а*.

Вариант 1.

	1-5	6-10	11-15	16-20
t_1	c_2	c_3	c_1	×
t_2	×	×	c_2	c_3
t_3	c_3	c_2	c_4	×

Вариант 2.

	1-5	6-10	11-15
t_1	c_1	c_2	c_3
t_2	c_2	c_3	×
t_3	c_3	c_4	c_2

Таблица 6.3.6а. Варианти за разписание.

Като отправна точка за решаването на задачата ще разгледаме следния комбинаторен алгоритъм: За да намерим програма с минимална продължителност, ще конструираме и оценим *всички* възможни разписания. Нека разгледаме следната таблица (*таблица 6.3.6б.*):

	1	2	3	4	5	6	7	8	...
t_1									
t_2									
...									

Таблица 6.3.6б. Таблица на разпределение.

Таблицата показва заетостта на всеки учител във всеки един час, т. е. в клетката на i-тия ред, j-тия стълб ще се записва класът, на който ще преподава i-тият учител в j-тия час. Последова-

телно ще запълваме стълбовете на таблицата с всички възможни комбинации от класове така, че да бъдат изпълнени условията на задачата:

- във всеки момент всеки учител може да преподава най-много на един клас (т. е. в един стълб не може да има повторение на класове).
- всеки учител да проведе точно толкова часа със съответния клас, колкото са зададени по условие.

Например, за *Вариант 2* по-горе таблицата ще се запълни по начина, показан в *таблица 6.3.6в.*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t_1	c_1	c_1	c_1	c_1	c_1	c_2	c_2	c_2	c_2	c_2	c_3	c_3	c_3	c_3	c_3
t_2	c_2	c_2	c_2	c_2	c_2	c_3	c_3	c_3	c_3	c_3	×	×	×	×	×
t_3	c_3	c_3	c_3	c_3	c_3	c_4	c_4	c_4	c_4	c_4	c_2	c_2	c_2	c_2	c_2

Таблица 6.3.6в. Таблица на разпределение.

Решението с попълване на такава таблица няма да бъде ефективно, тъй като генерираме разписание, като разглеждаме всеки час поотделно. В посочения пример всички учители провеждат по 5 часа — така можем да генерираме разписанието за всеки 5 поредни часа (*таблица 6.3.6г.*):

	5	10	15	20	...
t_1					
t_2					
...					

Таблица 6.3.6г. Таблица на разпределение.

В общия случай, тъй като часовете на всеки учител са произволни естествени числа, на всяка стъпка ще правим следното:

- Изчисляваме $mX = \min(cl[i][j])$, за всяко $i = 1,2,...,c$, $j = 1,2,...,t$, $cl[i][j] \neq 0$.
- Генерираме всички възможни комбинации от присвоявания на класове на учител за периода от mX часа: на учител $t_i \rightarrow$ клас c_j , за всяко $i = 1,2,...,t$, и за всяко присвояване ще намаляваме с mX часовете $cl[i][j]$.

Генерирането продължава, докато всички стойности $cl[i][j]$ станат равни на нула — тогава е налице граничен случай, при който се изчислява, колко дълго е продължила програмата и, ако полученото число е по-малко от най-малката продължителност, намерена до момента, разписанието се запазва.

Алгоритъма ще реализираме като рекурсивна функция `generate(teacher, level, mX, totalHours)` с 4 параметъра: променливата `teacher` съдържа номера на учителя, за който ще се назначава клас, `mX` е броят часове за провеждане, а `totalHours` — общата продължителност на програмата до момента. За всяко ниво (текущото се пази в `level`) първо се изчислява `mX` по алгоритъма, описан по-горе, след което се извиква `generate(0, level+1, mX, totalHours+mX)`. Това означава, че по-нататък трябва за всеки учител да се назначи клас, на който да преподаде `mX` часа. Присвояването на клас `k` на учителят `teacher` се извършва по алгоритъма за комбинации без повторение (*виж 1.3.3.*). След присвояването, часовете за преподаване `cl[k][teacher]` се намалят с `mX`, а когато се назначи клас за всеки учител (`teacher == N`) се преминава на следващо ниво и целият процес се повтаря отначало — преизчислява се `mX` и т.н. Дъното на рекурсията се достига, когато не останат часове за преподаване — тогава се проверява дали `totalHours < minimal` (дали продължителността на току-що генерираната програма е по-малка от минималната до момента) и, ако е изпълнено, разписанието се запазва. Извършва се и рязане на рекурсията:

```
if (totalHours >= minimal) return;
```

т. е., ако продължителността на частично генерираната програма стане по-голяма от минималната намерена до момента, то няма смисъл да се продължава генерирането.

```
#include <stdio.h>

/* Максимален брой учители */
#define MAXT 100
/* Максимален брой класове */
#define MAXC 100

/* Брой учители */
const unsigned t = 3;

/* Брой класове */
const unsigned c = 4;

unsigned cl[MAXC][MAXT] = {
  { 5, 5, 5 }, /* информация за клас 1 */
  { 5, 5, 5 },
  { 5, 0, 0 },
  { 0, 0, 5 }  /* информация за клас C */
};

const unsigned MAX_VALUE = 20000;

char usedC[100][MAXC];
unsigned teach[MAXT], minimal;

void generate(unsigned teacher, unsigned level,
              unsigned mX, unsigned totalHours)
{ unsigned i, j;
  if (totalHours >= minimal) return;
  if (teacher == t) {
    unsigned min = MAX_VALUE;
    for (i = 0; i < c; i++)
      for (j = 0; j < t; j++)
        if (cl[i][j] < min && 0 != cl[i][j]) min = cl[i][j];
    if (min == MAX_VALUE) {
      if (totalHours < minimal) minimal = totalHours;
    }
    else {
      generate(0, level + 1, min, totalHours + min);
    }
    return;
  }

  /* определя клас за учителя teacher, с който той да проведе min часа */
  for (i = 0; i < c; i++) {
    if (cl[i][teacher] > 0 && !usedC[level][i]) {
      cl[i][teacher] -= mX;
      usedC[level][i] = 1;
      generate(teacher + 1, level, mX, totalHours);
      usedC[level][i] = 0;  /* връщане */
      cl[i][teacher] += mX;
    }
  }

  /* Ако не е намерено присвояване за учителя,
```

```
    то не са му останали часове за преподаване */
  if (i == c) generate(teacher + 1, level, mX, totalHours);
}

int main(void) {
  unsigned i, j;
  for (i = 0; i < 100; i++)
    for (j = 0; j < c; j++) usedC[i][j] = 0;
  minimal = MAX_VALUE;
  generate(t, 0, 0, 0);
  printf("Минималното разписание е с продължителност %u часа.\n",
         minimal);
  return 0;
}
```
🖫 program.c

Резултат от изпълнението на програмата:
```
Минималното разписание е с продължителност 15 часа.
```

Задачи за упражнение:

1. Да се модифицира горната програма така, че да отпечатва разпределението за преподаване заедно с минималната продължителност на програмата.
2. Каква е сложността на предложения алгоритъм?

6.3.7. Побуквено превеждане

Морзовата азбука е често използвано, макар малко поостаряло, средство за комуникация — при телеграфи, кораби, за връзка между военни обекти и др. При нея всеки символ (буква, число и др.) се представя като последователност от точки и тирета. Така, символът може да се предава чрез кратки и по-продължителни радио (или светлинни) сигнали, отговарящи на точката и тирето. Съществено е, че при предаването между символите трябва да се прави пауза, в противен случай обратният превод не е еднозначен, например:

На буквата "А" отговаря "01" (с 0 ще означаваме точките, а с 1 — тиретата), на буквата "М" отговаря "11", а на буквата "Й" —"0111". Така, ако се получи кодираната последователност "0111", няма да знаем, как да я декодираме — като "Й" или като "АМ".

Задачата, която ще разгледаме, е следната: Дадени са две азбуки и за всяка буква от едната азбука е известно, коя буква от другата ѝ отговаря ("буквите" от двете азбуки могат да представляват повече от един символ). Така например, ако символите, които образуват буквите от първата азбука, са 0 и 1, а от втората — *a*, *b*, *c*, *d*, то можем да разглеждаме следните две азбуки:

1) "0", "1", "01", "10"
2) "*a*", "*b*", "*c*", "*d*"

Всички тези низове се възприемат като букви и на всяка буква от едната азбука се съпоставя еднозначно буква от другата, например:

"0" ⇔ "*a*"
"1" ⇔ "*b*"
"01" ⇔ "*c*"
"10" ⇔ "*d*"

По дадена *дума* (последователност от букви) от едната азбука търсим превод на езика на другата азбука. Задачата се усложнява от факта, че нямаме разделители между отделните букви. Така, низът "0110" може да се преведе по няколко начина: като "*abba*", "*cd*" и др. Търсят се всички възможни преводи на дадена дума.

Задачата е поставена така, че да изисква пълно изчерпване, затова ще приложим търсене с връщане. Нека думата, която ще превеждаме, е записана в низа `str1`. Ще използваме параметър `count`, който сочи символа, до който сме достигнали до момента (първоначално `count` има стойност 0). Намираме подниз, започващ от `count`, който да бъде буква от първата азбука. Възможно е да намерим няколко такива букви (например, по-горе в низа "0110" ще намерим буквите "0" и "01"). За всяка такава буква рекурсивно изследваме (превеждаме) останалата част от низа, след изключването ѝ. Всъщност, на практика няма да модифицираме низа, а ще увеличаваме `count` с дължината на намерената буква. Ако в даден момент стойността на `count` стане равна на дължината на низа, то е налице граничен случай — получили сме един възможен превод.

В реализацията входните данни се инициализират във функцията `initLanguage()`. Броят на буквите е N, полето на масива `transf[i].st1` съдържа i-тата буква ($1 \leq i \leq n$) от първата азбука и на нея отговаря i-тата буква от втората азбука, записана в полето `transf[i].st2`. Когато извършваме превода, в масива `int translation[]` записваме само номерата на преведените букви и, ако стигнем до решение, печатаме буквите от втората азбука, отговарящи на тези номера. В програмата като примерни входни данни сме използвали българската азбука и цифрите от 0 до 9. Втората азбука се състои от съответния за всеки символ морзов код.

```
#include <stdio.h>
#include <string.h>

#define MAXN   40  /* Максимален брой съответствия между букви */
#define MAXTL  200 /* Максимална дължина на дума за превод */

/* Брой съответствия */
const unsigned n = 38;

/* Дума за превод */
char *str1 = "101001010";

struct transType {
  char *st1;
  char *st2;
};
struct transType transf[MAXN];

unsigned translation[MAXTL], pN, total = 0;

/* В примера се използва Морзовата азбука: 0 е точка, а 1-та е тире */
void initLanguage(void)
{ transf[0].st1 = "А";  transf[0].st2 = "01";
  transf[1].st1 = "Б";  transf[1].st2 = "1000";
  transf[2].st1 = "В";  transf[2].st2 = "011";
  transf[3].st1 = "Г";  transf[3].st2 = "110";
  transf[4].st1 = "Д";  transf[4].st2 = "100";
  transf[5].st1 = "Е";  transf[5].st2 = "0";
  transf[6].st1 = "Ж";  transf[6].st2 = "0001";
  transf[7].st1 = "З";  transf[7].st2 = "1100";
  transf[8].st1 = "И";  transf[8].st2 = "00";
  transf[9].st1 = "Й";  transf[9].st2 = "0111";
  transf[10].st1 = "К"; transf[10].st2 = "101";
  transf[11].st1 = "Л"; transf[11].st2 = "0100";
  transf[12].st1 = "М"; transf[12].st2 = "11";
  transf[13].st1 = "Н"; transf[13].st2 = "10";
  transf[14].st1 = "О"; transf[14].st2 = "111";
  transf[15].st1 = "П"; transf[15].st2 = "0110";
  transf[16].st1 = "Р"; transf[16].st2 = "010";
  transf[17].st1 = "С"; transf[17].st2 = "000";
  transf[18].st1 = "Т"; transf[18].st2 = "1";
```

```
  transf[19].st1 = "У"; transf[19].st2 = "001";
  transf[20].st1 = "Ф"; transf[20].st2 = "0010";
  transf[21].st1 = "Х"; transf[21].st2 = "0000";
  transf[22].st1 = "Ц"; transf[22].st2 = "1010";
  transf[23].st1 = "Ч"; transf[23].st2 = "1110";
  transf[24].st1 = "Ш"; transf[24].st2 = "1111";
  transf[25].st1 = "Щ"; transf[25].st2 = "1101";
  transf[26].st1 = "Ю"; transf[26].st2 = "0011";
  transf[27].st1 = "Я"; transf[27].st2 = "0101";
  transf[28].st1 = "1"; transf[28].st2 = "01111";
  transf[29].st1 = "2"; transf[29].st2 = "00111";
  transf[30].st1 = "3"; transf[30].st2 = "00011";
  transf[31].st1 = "4"; transf[31].st2 = "00001";
  transf[32].st1 = "5"; transf[32].st2 = "00000";
  transf[33].st1 = "6"; transf[33].st2 = "10000";
  transf[34].st1 = "7"; transf[34].st2 = "11000";
  transf[35].st1 = "8"; transf[35].st2 = "11100";
  transf[36].st1 = "9"; transf[36].st2 = "11110";
  transf[37].st1 = "0"; transf[37].st2 = "11111";
}

/* Отпечатва превод */
void printTranslation(void)
{ unsigned i;
  total++;
  for (i = 0; i < pN; i++)
    printf("%s", transf[translation[i]].st1);
  printf("\n");
}

/* Намира следваща буква */
void nextLetter(unsigned
                count)
{ unsigned i, k;
  if (count == strlen(str1)) { printTranslation(); return; }
  for (k = 0; k < n; k++) {
    unsigned len = strlen(transf[k].st2);
    for (i = 0; i < len; i++)
      if (str1[i + count] != transf[k].st2[i]) break;
    if (i == len) {
      translation[pN++] = k;
      nextLetter(count + strlen(transf[k].st2));
      pN--;
    }
  }
}

int main(void) {
  printf("Списък от всички възможни преводи: \n");
  initLanguage();
  pN = 0;
  nextLetter(0);
  printf("Общ брой различни преводи: %u \n", total);
  return 0;
}
```

translat.c

Задача за упражнение:

Да се модифицира горната програма така, че да намира превод с минимална дължина. Какви условия отсичане на дървото с кандидатите за решение могат да се приложат?

6.4. Метод на разклоненията и границите

Частен случай на метода на търсенето с връщане е *методът на разклоненията и границите*. Той се прилага в задачи, при които са в сила следните две условия:

1) На всеки кандидат за решение е съпоставена стойност — *цена*, като целта е да се намери *оптимално* решение (например това с минимална цена).
2) Всяко решение на задачата трябва да може да се представи като получено от последователни разширения на частични решения с монотонно нарастваща цена.

В *глава 5* (*виж 5.4.4.*) решихме чрез метода на разклоненията и границите задачата за търговския пътник. В тази задача двете условия бяха изпълнени:

1) На всеки път от графа (и в частност Хамилтонов цикъл) се съпоставя цена — сбор от теглата на ребрата, които съдържа. Търсим Хамилтонов цикъл с минимална цена.
2) Тъй като въведохме допълнителното ограничение теглата на ребрата на графа да бъдат положителни числа, то е изпълнено и второто условие: Цената на всяко частично решение, в което участват върховете $(v_1,v_2, \ldots ,v_k)$, е по-малка от цената на произволно негово разширение $(v_1,v_2, \ldots, v_k, v_{k+1})$, получено с добавяне на още един връх.

Съществено при метода на разклоненията и границите е, че рекурсивното дърво, получено от кандидатите за решение, може да се "изрязва" въз основа на още един принцип: Ако дадено частично решение има цена по-голяма или равна на най-малката, намерена до момента, то няма смисъл да се разширява повече, тъй като това със сигурност няма да доведе до по-добро решение. Подобно изрязване на безперспективните решения направихме и при задачата за търговския пътник. Ще разгледаме още една *NP*-пълна задача и неин вариант, при който е възможно решението да се търси по метода на разклоненията и границите.

6.4.1. Задача за раницата (оптимална селекция)

Задача 1. Дадена е раница с вместимост (издръжливост) M килограма. Дадени са N предмета, всеки с тегло m_i $(1 \leq i \leq N)$. Да се провери дали съществува подмножество от предмети, които да запълват точно раницата, т. е. сборът от теглата им да бъде *точно* M.

Задача 2. (*обобщен оптимизационен вариант на предишната*) Дадена е раница, която може да побере M килограма. Дадени са N предмета, всеки с тегло m_i и стойност (цена) c_i. Да се намери подмножество от предмети с *максимална обща цена*, които да се побират в раницата, т. е. сборът от теглата им да бъде по-малък или равен на M.

В някои частни случаи, например когато теглата са цели числа, и при наличие на достатъчно памет ($\Theta(M.N)$), могат да се приложат ефективни алгоритми за решаване на горните задачи, основани на принципа на динамичното оптимиране (*глава 8*). В общия случай обаче, когато цените и теглата са реални числа, тези две *NP*-пълни задачи се решават с пълно изчерпване. Ще разгледаме втората от тях и ще я решим с метода на разклоненията и границите.

Ще генерираме всички възможни решения чрез търсене с връщане, като в процеса на генериране ще запазим оптималното. Във втората задача е дадено множество от предмети. Измежду всички 2^N негови подмножества трябва да изберем това, което отговаря на условията на задачата: сумата от теглата на предметите в него да не надхвърля M и сумата от стойностите им да бъде максимална.

Да разгледаме основните моменти при съставянето на алгоритъма. (С цел опростяване на нотацията, на c_i и m_i ще гледаме като на масиви $c[i]$ и $m[i]$ съответно.)

Нека разполагаме с частично решение (подмножество от предмети), при което са взети предметите $A = \{a_1,a_2,...,a_k\}$. С $S(a_1,a_2,...,a_k)$ означаваме сумата $S(a_1,a_2,...,a_k) = m[a_1]+m[a_2]+...+m[a_k]$ от теглата им. Към подмножеството A опитваме да добавим нов предмет a_{k+1} измежду тези, които не са взети до момента:

- Ако $S(a_1,a_2,...,a_k) + m[a_{k+1}] \leq M$, то следва, че добавянето на предмета е възможно и продължаваме рекурсивно да разширяваме подмножеството A по всички възможни начини.
- Ако не съществува предмет a_{k+1} такъв, че $S(a_1,a_2,...,a_k) + m[a_{k+1}] \leq M$, то следва, че множеството $a_1,a_2,...,a_k$ е гранично (т. е. не може да се разширява повече).

Тогава се изчислява стойността му $c[A] = c[a_1]+c[a_2]+...+c[a_k]$ и, ако е по-голяма от максималната намерена до момента, се запазва:

```
void generate(i) {
  if (общото_тегло_на_взетите_предмети > M) return;
  if (i == N) {
    /* граничен случай, проверка дали е налице по-добро решение
       от максималното, намерено до момента */
    return;
  }
  вземаме_i-тия_предмет;
  generate(i+1);
  изключваме_i-тия_предмет;
  generate(i+1); /* т. е. при избирането на предмети пропускаме i-тия */
}
```

На практика дефинирахме кога едно частично решение може да се разшири. Така осигурихме да не се разглеждат всички 2^N-та възможни подмножества, а само тези, за които сборът от теглата на предметите е по-малък или равен на M. С това обаче въпросът с отсичането на рекурсивното дърво на кандидатите за решение не се изчерпва.

При задачата за Търговския пътник, където се търси *минимален* Хамилтонов цикъл, едно частично решение се обявява за безперспективно, ако дължината на пътя, който се строи, стане по-голяма от минималната намерена до момента. Когато търсим *максимално* решение, както при задачата за оптималната селекция, този вид отрязване трябва да се измени: Нека в процеса на построяване на подмножество от предмети (както в схематичната процедура по-горе) сме взели предметите $A = \{a_1,a_2,...,a_k\}$, а сме пропуснали предметите $B = \{b_1,b_2,...,b_p\}$. С VT ще означим сбора от стойностите на всички предмети, а с $Vmax$ максималната стойност, получена от проверените до момента решения. Тогава, ако

$$VT - c[b_1] - c[b_2] - ... - c[b_p] \leq Vmax,$$

то следва, че няма смисъл да разширяваме повече текущото решение. Това е така, тъй като дори да бъде възможно включването на всички останали предмети, максималната стойност, която ще получим, е:

$$(\textit{стойността на взетите до момента предмети}) + (\textit{стойността на оставащите}) =$$
$$= (c[a_1]+...+c[a_k]) + (VT - (c[a_1]+...+c[a_k]+c[b_1]+...+c[b_p])) =$$
$$= VT - c[b_1] - c[b_2] - ... - c[b_p],$$

при което, ако последният израз се окаже по-малък от $Vmax$, следва, че текущото решение е безперспективно и по-нататъшно разглеждане няма да доведе до по-добри резултати. Следва по-подробна схема на рекурсивната функция `generate()`:

```
float Ttemp;      /* общо тегло на взетите предмети */
float Vtemp;      /* обща цена на взетите предмети */
float VmaX;       /* максимална цена от решенията, проверени до момента */
float totalV;     /* обща цена на оставащите предмети */
```

```
void generate(unsigned i)
{ if (Ttemp > K) return;
  if (Vtemp + totalV < VmaX) return; /* безперспективно решение */
  if (i == N) {
    if (Vtemp > VmaX) {
      запазване_на_решението;
      VmaX = Vtemp;
    }
    return;
  }
  Vtemp += c[i]; Ttemp += m[i]; totalV -= c[i]; /* вземаме предмет i */
  generate(i+1);
  Vtemp -=c [i]; Ttemp -= m[i]; /* изключваме_i-тия_предмет; */
  generate(i+1); /* т. е. при вземането i-тия предмет се "прескача" */
  totalV += c[i];
}

int main(void) {
  totalV = c[1] + c[2] + ... + c[N];
  generate(0);
  return 0;
}
```

За да получим работеща програма, трябва да уточним още как ще извършваме включването и изключването на *i*-тия предмет. За целта ще въведем масив `taken[]` и променлива `tN`, която показва броя на елементите му. Най-доброто решение, намерено до момента, ще запазваме в масива `saveTaken[]`, с `sn` елемента. В края на функцията `main()` ще печатаме намереното оптимално решение. Входните данни `N`, `M`, `c[N]` и `m[N]` се задават като константи в началото на програмата.

```
#include <stdio.h>
#define MAXN 100
const unsigned n = 10;
const double M = 10.5;
const double c[MAXN] = {10.3,9.0,12.0,8.0,4.0,8.4,9.1,17.0,6.0,9.7};
const double m[MAXN] = {4.0,2.6,3.0,5.3,6.4,2.0,4.0,5.1,3.0,4.0};
unsigned taken[MAXN], saveTaken[MAXN], tn, sn;
double VmaX, Vtemp, Ttemp, totalV;

void generate(unsigned i)
{ unsigned k;
  if (Ttemp > M) return;
  if (Vtemp + totalV < VmaX) return;
  if (i == n) {
    if (Vtemp > VmaX) { /* запазване на оптималното решение */
      VmaX = Vtemp; sn = tn;
      for (k = 0; k < tn; k++) saveTaken[k] = taken[k];
    }
    return;
  }
  taken[tn++] = i;  Vtemp += c[i];  totalV -= c[i];  Ttemp += m[i];
  generate(i + 1);
  tn--;  Vtemp -= c[i];  Ttemp -= m[i];
  generate(i + 1);
  totalV += c[i];
}
```

```
int main(void) {
  unsigned i;
  tn = 0; VmaX = 0; totalV = 0;
  for (i = 0; i < n; i++) totalV += c[i];
  generate(0);
  printf("Максимална цена: %.2lf\nИзбрани предмети: \n", VmaX);
  for (i = 0; i < sn; i++) printf("%u ", saveTaken[i] + 1);
  printf("\n");
  return 0;
}
```

🖫 bagrec.c

Резултат от изпълнението на програмата:

```
Максимална цена: 37.40
Избрани предмети:
3 6 8
```

Задачи за упражнение:

1. Да се състави и реализира алгоритъм, решаващ *задача 1*.
2. Да се модифицира горната програма, за да намира всички решения с максимална цена.

6.5. Оптимални стратегии при игри

Примера с шаха, който дадохме в началото на главата, ще разгледаме още два пъти — тук и в *6.5.2*. В *6.3.4.* съвсем схематично показахме как може да се построи дърво на всевъзможните ходове. Сега при разглеждането ще използваме не дърво, а ориентиран граф. Всяка възможна шахматна конфигурация ще се представя с един връх на граф $G(V,E)$. Реброто $(i,j)\in E$ т.с.т.к от конфигурацията, отговаряща на i, може да се получи конфигурацията j с извършване на единствен ход. Ще дефинираме няколко понятия:

- Една шахматна конфигурация се нарича *терминална*, ако играта е завършила (мат на черните, мат на белите или реми. В шахмата реми настъпва при:
  - *пат* — няма възможен ход пред играча, който е на ход.
  - *вечен шах* — повтаряне на един и същи ход три пъти (обикновено форсирано с шах от единият от противниците, който се стреми към реми).
  - *невъзможност* на никой от двата противника да победи (останали са прекалено малко и/или слаби/неманеврени фигури върху дъската).
  - *взаимна договорка* за реми между играчите (дори и този "субективен" случай не бива да се пропуска и е валиден при компютърните реализации на шахмат).

Когато е налице терминална конфигурация, съответният връх от графа няма наследници.

- Една *нетерминална* конфигурация се нарича *печеливша*, ако съществува поне един наследник, който е *губещ* (т. е. само с един ход можем да достигнем до конфигурация, при която на ход е противникът и всеки негов ход води до бъдеща загуба).
- Една нетерминална конфигурация се нарича *губеща*, ако всичките ѝ наследници са печеливши конфигурации (т. е. какъвто и ход да изиграем, нашият противник ще има печеливш ход веднага след това).
- Всяка друга конфигурация е *неопределена* (ако и двамата играчи играят оптимално, играта ще завърши наравно).

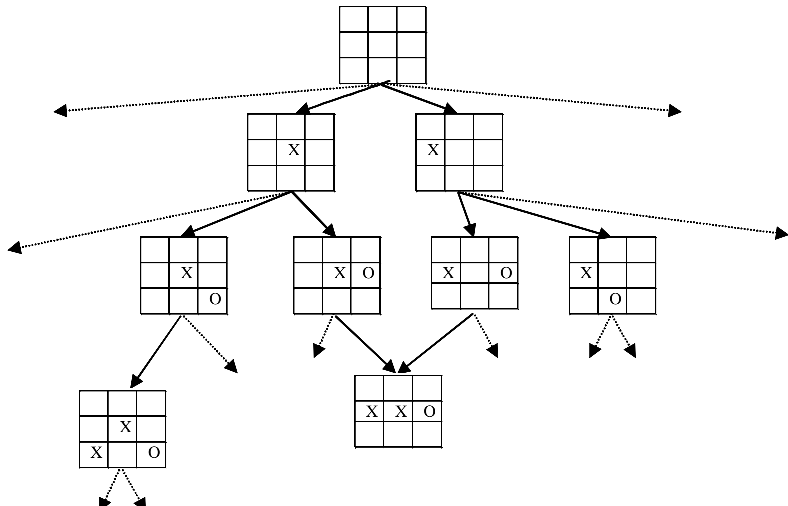
Така, целта на перфектно играещата шах-програма е да се стреми винаги към печеливша конфигурация. Веднъж попадне ли в такава, то победата е само въпрос на време и е абсолютно сигурна, независимо от ходовете на противника.

6.5.1. Игра "X"-чета и "O"

Построяването на граф, представящ пълен анализ на шахматна партия, не е по-силите на сега съществуващата техника (тя не позволява запазването или изследването на граф с приблизително 10^{100} върха). Затова проблемът с печелившите стратегии ще разгледаме с една по-проста, но също добре известна игра:

Играят двама души върху дъска с размер 3×3. На всеки ход играчите се редуват да попълват полета от дъската — играч 1 използва символа "*X"*, а играч 2 — "*O*". Печели този, който пръв попълни цял хоризонтал, вертикал или някой от двата главни диагонала на дъската.

Възможните конфигурации в играта ще представим като възли на граф (забележете, че за разлика от шахмата, при играта "X" и "O" графът е ацикличен). Част от графа е показана на *фигура 6.5.1а.*:



Фигура 6.5.1а. Граф на играта "*X*"-чета и "*O*"

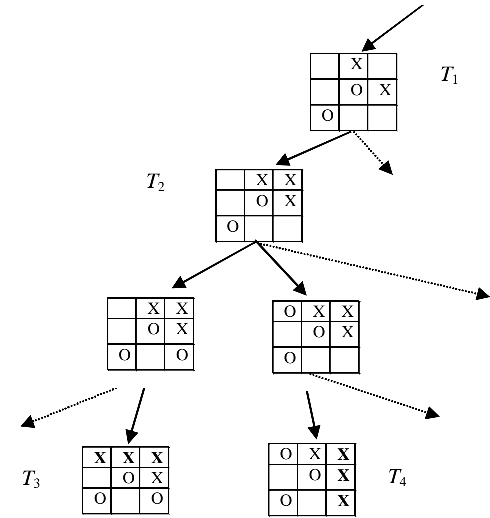
Термините от предходния параграф са валидни и тук — за примера от *фигура 6.5.1б.* конфигурациите T_3 и T_4 са *терминални* (играч 1 е спечелил играта), конфигурацията T_2 е *губеща* (какъвто и ход да изиграе играч 2, той ще загуби), а T_1 е *печеливша*.

Ще предложим алгоритъм, играещ оптимално играта "*X*"-чета и "*O*", извършващ пълно изчерпване (пълно обхождане на графа на играта). Ще използваме рекурсивната функция `checkPosition(int player, board)`, която проверява дали дадена конфигурация `board` е печеливша, губеща или неопределена за играча `player`. С `board1,board2,...,boardn` ще означим всички конфигурации, които могат да се получат от `board` с извършването на един ход:

```
/* Връща: 1 - ако конфигурацията е печеливша за играча player, */
/*        2 - ако е губеща */
/*        3 - ако е неопределена */
int checkPosition(int player, board)
 if (конфигурацията е терминална) {
   if (играта е завършила) return 3; /* реми */
   if (player == играча, който печели) return 1;
   if (player != играча, който печели) return 2;
```

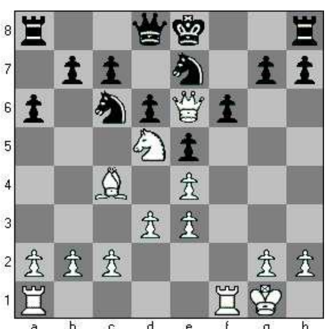
```
    } else {
      if (checkPosition(other_player,boardi)==1 за всяко i=1,..,n) return 2;
      if (checkPosition(other_player,boardi)==2 за някое i=1,..,n) return 1;
      return 3;
    }
  }
```

В горната схема се използва една и съща стойност за означаване на неопределена позиция и терминална позиция, при която играта завършва реми. Както отбелязахме в параграф *6.5.*, това, че дадена позиция е неопределена (и в случай, че дървото на играта се разглежда изцяло) означава, че при оптимална игра и на двамата играчи играта със сигурност ще завърши реми.



Фигура 6.5.1б. Терминални позиции в дървото на играта.

Схемата на функцията `checkPosition()` е универсална. Тя може да се приложи във всички игри, в които участват двама редуващи се играчи и всяка конфигурация може да се получи с един ход от някоя друга. Може се приложи и при шахмат, но, поради размерите на дървото, тя ще помогне само при някои по-прости случаи (например на *фигура 6.5.1в.* е показана печелившa позиция за белите). С нея можем да решаваме и класическите шахматни задачи от вида "мат в 3 хода" и др.



Фигура 6.5.1в. Мат в 3 хода (белите са на ход).

Следва една възможна реализация на функцията `checkPosition()` за играта "*X*"-чета и "*O*". Използваме функция `terminal()`, която проверява дали дадена конфигурация е терминална, и, ако е такава, връща кой е победителят, или че играта е завършила реми:

```
#include <stdio.h>

/* Стартов играч */
const char startPlayer = 2;

/* Стартова конфигурация */
char board[3][3] = {
  { '.', '.', '.' },
  { '.', 'X', '.' },
  { 'X', '.', 'O' }};

/* Връща: 1, ако конфигурацията е терминална и печели играч 1,
 *        2, ако е терминална и печели играч 2,
 *        3, ако е терминална играта е реми
 *        0, ако конфигурацията не е терминална.
 */
char terminal(char a[3][3])
{ unsigned i, j;
  for (i = 0; i < 3; i++) {
    /* проверка на хоризонталите */
    for (j = 0; j < 3; j++)
      if (a[i][j] != a[i][0]) break;
    if (3 == j && a[i][0] != '.') {
      if (a[i][0] == 'X') return 1; else return 2;
    }
    /* проверка на вертикалите */
    for (j = 0; j < 3; j++)
      if (a[j][i] != a[0][i]) break;
    if (3 == j && a[i][0] != '.') {
      if (a[0][i] == 'X') return 1; else return 2;
    }
    /* проверка на диагоналите */
    if (a[0][0] == a[1][1] && a[1][1] == a[2][2] && a[1][1] != '.')
      if (a[0][0] == 'X') return 1; else return 2;
    if (a[2][0] == a[1][1] && a[1][1] == a[0][2] && a[1][1] != '.')
      if (a[2][0] == 'X') return 1; else return 2;
  }
  /* дали не е реми (дали всички позиции са заети) */
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      if (a[i][j] == '.') return 0;
  return 3;
}
/* Връща: 1, ако конфигурацията е печеливша за играча player,
 *        2, ако е губеща и
 *        3, ако е неопределена */
char checkPosition(char player, char board[3][3])
{ unsigned i, j, result;
  int t = terminal(board);
  if (t) {
    if (player == t) return 1;
    if (3 == t) return 3;
    if (player != t) return 2;
  }
```

```
    else {
      char otherPlayer, playerSign;
      if (player == 1) { playerSign = 'X'; otherPlayer = 2; }
        else { playerSign = 'O'; otherPlayer = 1; }

      /* char board[3][3];
       * определя позицията
       */
      for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
          if (board[i][j] == '.') {
            board[i][j] = playerSign;
            result = checkPosition(otherPlayer, board);
            board[i][j] = '.';
            if (result == 2) return 1;  /* конф. е губеща за противника, */
            /* следователно е печеливша за играча player */
            if (result == 3) return 3;  /* конфигурацията е неопределена */
          }
        }
      }
      /* всички наследници са печеливши конфигурации и
       * следователно тази е губеща за играча player */
      return 2;
    }
  }

  int main(void) {
    printf("%u\n", checkPosition(startPlayer, board));
    return 0;
  }
```

tictac.c

В следващите няколко точки ще разгледаме и този въпрос — за сравнение по "качество" на две неопределени позиции, както и за отсичане на част от дървото, подлежащо на изследване.

Задачи за упражнение:

1. Да се напише програма, симулираща игра на “*X*”-чета и “*O*” между човек и компютър, като компютърът трябва да играе оптимално: да печели винаги, когато му се удаде такава възможност, т. е. при грешка на човека, и сам да не греши. Когато позицията на компютъра е неопределена, той може да направи произволен *негубещ* ход.
2. Да се напише програма за игра на “*X*”-чета и “*O*” в общия случай, когато е дадена дъска с размер $n \times n$, и печеливш е играчът, който направи редица, колона или диагонал от m знака. В най-разпространения вариант m е 5.
3. Да се докаже, че при дъска 3×3, ако двамата играчи играят правилно, играта задължително завършва с реми.
4. Да се намери броят на:
   а) всевъзможни конфигурации, които могат да се получат по време на игра
   б) всевъзможните конфигурации от дървото (да се сравни с а))
   в) броят на листата в дървото (т.е. на терминалните конфигурации)

6.5.2. Принцип на минимума и максимума

Ще продължим с играта “*X*”-чета и “*O*”. За разлика от предходния параграф, сега възможните ходове ще представяме не с граф (както на *фигура 6.5.1а.*), а с дърво, на всеки връх на което се съпоставя някаква *оценка*. Тя определя, доколко *оптимална* е дадена позиция за играч 1. Ще

започнем от листата на дървото (терминалните конфигурации): едно листо е с оценка $+\infty$, ако печели играч 1, оценка $-\infty$, ако печели играч 2, и оценка 0, ако играта завършва реми. Оценката на всички останали върхове се определя от техните наследници по следния начин: За връх i с наследници $i_1,i_2,..,i_k$ оценката $V[i]$ се определя въз основата на следния *принцип на минимума и максимума*:

$$V[i]=\begin{cases}\max(V[i_1],V[i_2],...,V[i_k]), \text{ако на ход е играч } 1\\ \min(V[i_1],V[i_2],...,V[i_k]), \text{ако на ход е играч } 2\end{cases}$$

Един връх се нарича *връх за максимизиране*, ако в конфигурацията, която му отговаря, на ход е играч 1, иначе се нарича *връх за минимизиране* (тези дефиниции следват интуитивно от това, че оценката на върха се определя като максимума/минимума от оценките на наследниците му). Следва функция (основана на метода търсене с връщане), която изчислява оценката на произволен връх от дървото.

```
value minimax(връх i) {
   if (i_е_листо) return оценката_на_листото;
   Нека (i₁, i₂, .., iₙ) са наследниците на i;
   if (i_е_връх_за_минимизиране)
     return min(minimax(i₁), .., minimax(iₙ));
   if (i_е_връх_за_максимизиране)
     return max(minimax(i₁), .., minimax(iₙ));
}
```

Така, задачата за извършване на най-добър следващ ход се свежда до намиране на върха-наследник с най-висока оценка. При играта "*X*"-чета и "*O*" оценките на върховете-листа могат да бъдат $-\infty$, $+\infty$ и 0. Оттук оценката на всеки връх от дървото задължително е една от тези три стойности. В много други игри обаче, е възможно различните оценки за листата да са повече, или оценката на връх от дървото да се определя, без да е необходимо да се изследва цялото му поддърво (т. е. оценката да се получава по някакъв субективен критерий — *виж 6.5.4.*). Характерно за функцията `minimax()` е, че тя изследва *всеки* един връх от дървото. Често това не е необходимо.

Задача за упражнение:

Да се напише програма за симулиране на игра на "*X*"-чета и "*O*" между човек и компютър, реализираща принципа на минимума и максимума.

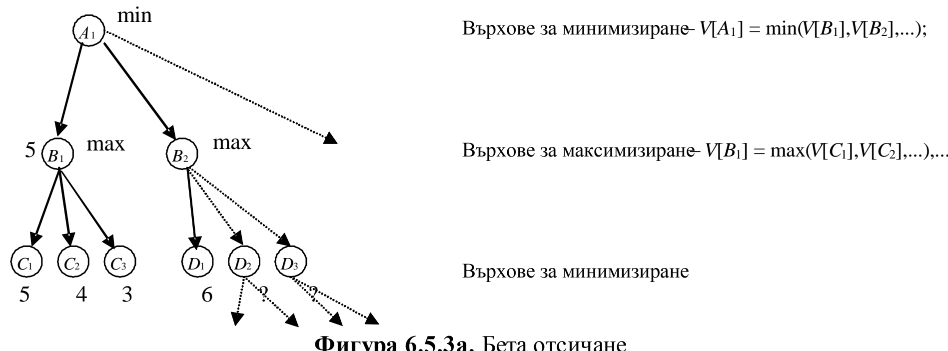
6.5.3. Алфа-бета отсичане

Алфа-бета отсичането е метод за редуциране на върховете, които се изследват по принципа на минимума и максимума. Нека разгледаме два примера.

На *фигура 6.5.3а.* е показано дърво, в което алгоритъмът трябва да пресметне оценката на върха за минимизиране A_1. Оценката на A_1 е минималната от оценките на наследниците му $V[B_1]$, $V[B_2]$, $V[B_3]$ и т.н. Алгоритъмът първо се опитва да намери стойността на оценката $V[B_1]$ — тя се определя като максимума от оценките на наследниците му. Да предположим, че оценките на наследниците на B_1 са вече изчислени, така $V[B_1] = max(V[C_1,C_2,C_3]) = max(5,4,3) = 5$. Тъй като A_1 е връх за минимизиране, от $V[B_1] = 5$ следва, че оценката на A_1 ще бъде по-малка или равна на 5. По-нататък обхождането продължава с върха B_2. Стойността $V[B_2]$ е максималната от стойностите на наследниците му $D_1,D_2,D_3,...$. Стига се до същинската част на *алфа-бета отсичането* — в разгледания пример приемаме, че оценката, която се е получила след изследването на поддървото с корен D_1, е $V[D_1] = 6$. От това следва, че оценката на B_2 ще бъде поне 6 (търсим максимум). От друга страна вече видяхме, че $V[A_1] \leq 5$. Така може да се приключи с изследването на B_2 (казва се, че правим *бета-отсичане*) — поддърветата с корен наследниците

му D_2, D_3,... не ни интересуват и няма да се изследват, тъй като вече получихме, че $V[B_2] \geq 6$, а $V[A_1] \leq 5$.

Алфа отсичането се прилага, когато се разглежда връх за максимизиране. Да разгледаме примера от *фигура 6.5.3б.*: Да предположим, че сме пресметнали $V[C_1] = 4$, $V[C_2] = 5$, $V[C_3] = 6$. Така $V[B_1] = \min(V[C_1], V[C_2], V[C_3]) = \min(4, 5, 6) = 4$. От това следва, че $V[A_1] \geq 4$. По-нататък се получава, че $V[D_1] = 3$, от това следва, че $V[B_2] \leq 3$. От $V[A_1] \geq 4$ и $V[B_2] \leq 3$ следва, че по-нататъшното изследване на наследниците на B_2 е излишно (*алфа-отсичане*).



Фигура 6.5.3а. Бета отсичане.

Реализацията на алфа и бета отсичането се извършва по следния начин: За всеки връх, който се обхожда, освен неговата оценка се изчисляват още две стойности — *алфа-стойност* и *бета-стойност* на върха:

Алфа-стойност на връх

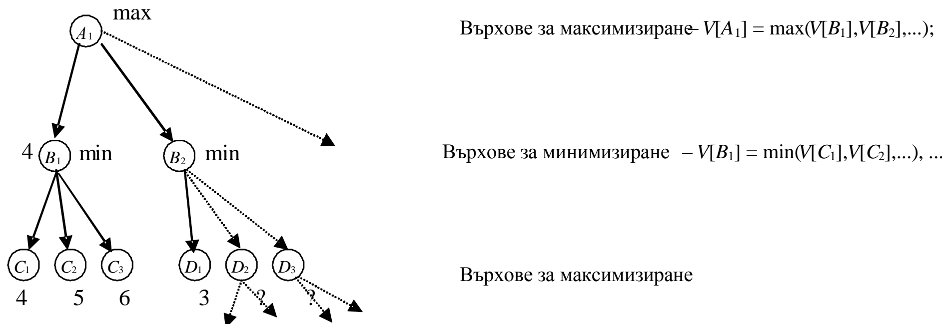
Инициализираме алфа-стойностите по следния начин: Ако върхът е листо, алфа-стойността е $-\infty$, в противен случай тя съвпада с оценката му. По-късно, при обхождането, алфа стойността на всеки връх *за максимизиране* се определя като максималната стойност от оценките на наследниците му, изследвани до момента. При върховете за минимизиране алфа-стойността е алфа-стойността на *предшественика* на върха.

Бета-стойност на връх

Бета-стойностите на листата се инициализират с оценките им. Всички останали върхове се инициализират с бета-стойност $+\infty$. При обхождането по-късно, бета-стойността на връх за минимизиране се определя като минималната стойност от оценките на наследниците му, изследвани до момента. Бета-стойността на всеки връх за максимизиране е бета-стойността на *предшественика* му.

Това присвояване гарантира, че:

- Оценката на връх ще бъде винаги не по-малка от алфа-стойността на върха и не по-голяма от бета стойността му.
- Алфа и бета стойностите могат да се променят, при обхождането, но е сигурно, че алфа-стойността никога няма да намалее, а бета-стойността — да се увеличи.
- Алфа-стойността на всеки връх (с изключение на корена) е винаги по-голяма или равна на алфа-стойността на предшественика му.
- Бета-стойността на всеки връх (с изключение на корена) е винаги по-малка или равна на бета-стойността на предшественика му.



Фигура 6.5.3б. Алфа отсичане.

Ще покажем схема, по която може да се извършва обхождането на дървото. Функцията `minimaxCutoff()` се извиква с параметри корена на дървото, $-\infty$ и $+\infty$ (съответно за алфа и бета стойностите):

```
value minimaxCutoff(N, A, B) {
   Na = A; /* Na е алфа-стойността на върха, която ще се изчислява */
   Nb = B; /* Nb е бета-стойността на върха, която ще се изчислява */
   if (N_е_листо) return оценката_на_листото;
   if (N_е_връх_за_минимизиране)
      for (всеки_наследник_Ni_на_N) {
        val = minimaxCutoff(Ni, Na, Nb);
        Nb = min(Nb, val);
        if (Nb <= Na) break; /* алфа-бета отсичане */
      }
      return Nb;
   } else { /* if (N_е_връх_за_максимизиране) */
      for (всеки_наследник_Ni_на_N) {
        val = minimaxCutoff(Ni, Na, Nb);
        Na = max(Na, val);
        if (Na >= Nb) break; /* алфа-бета отсичане */
      }
      return Na;
   }
}
```

В двата подчертани реда се прекъсва по-нататъшното алфа-бета изследване, тъй като то е безперспективно.

Задача за упражнение:

Да се напише програма за симулиране на игра на "*X*"-чета и "*O*" между човек и компютър, реализираща принципа на минимума и максимума с алфа-бета отсичане. Да се сравни с варианта *без* алфа-бета отсичане.

6.5.4. Алфа-бета изследване до определена дълбочина

Ще се спрем отново на примера с шахмата. За да извършим алфа-бета обхождане за определяне на оценката на връх, ще трябва да обходим огромно дърво, при което дори оптимизации като алфа-бета отсичане няма да ни помогнат съществено. Често в подобни случаи се прилага *обхождане до определена дълбочина*. При достигането ѝ обхождането се прекратява и оценката на върха, при който то е прекъснато, се изчислява по някакъв друг критерий. За шахмата този критерий може да бъде, какви фигури са останали на дъската, свободата на движението им, кой

владее центъра на дъската и т.н. Функцията, която пресмята оценката на конфигурацията, не трябва да изисква много изчислително време, тъй като ще се изпълнява често. Един възможен начин за оценяване е, като съпоставим стойности на фигурите [*Brassard,Bratley–1996*]:

Дама – 10; Топ – 5; Офицер, Кон – 3,25; Пешка – 1;

Оценката на дадена конфигурация (положителна или отрицателна) можем да дефинираме като сумата от точките на белите фигури минус сумата от точките на черните. За терминалните позиции приемаме, че, когато черните са в положение мат, оценката на конфигурацията е $+\infty$, а когато белите са мат — оценката е $-\infty$. Когато играта завърши наравно — стойността е 0.

Тогава функцията за оценяване на връх прилича на тази от *6.5.2.* с единствената разлика, че се предизвиква изчисляване на стойността на конфигурацията при достигане на определена дълбочина:

```
value eval(връх i) {
   return Изчислената_по_субективен_критерий_стойност_на_конфигурацията;
}
value minimax(връх i, depth) {
   if (i_е_листо) return оценката_на_листото(+∞,-∞,0);
   if (depth > maxDepth) return eval(i); /* форсирано изчисление */
   Нека_(i1, i2, .., in)_са_наследниците_на_i;
   if (i_е_връх_за_минимизиране)
     return min(minimax(i1,depth+1), .., minimax(in,depth+1));
   if (i_е_връх_за_максимизиране)
     return max(minimax(i1,depth+1), .., minimax(in,depth+1));
}
```

Колкото по-голяма е дълбочината `maxDepth` на обхождането, толкова по-точно оценяване ще получим. Никога обаче няма да бъдем сигурни, че оценката е абсолютно точна. Например, винаги е възможно да съществува развитие на играта, при което в конфигурация на дълбочина `maxDepth` да се жертва фигура (което ще намали оценката на `eval(i)`) с цел спечелване на по-добра позиция: В някой близък следващ ход (дори и `maxDepth` + 1), който няма да бъде разгледан, може да се спечели по-силна фигура или дори да се матира противникът.

Успоредно с алфа-бета обхождане е възможно да се приложи и алфа-бета отсичане. През 1997 скоростта на компютъра *Deep Blue*, победил световния шампион, е позволявала извършването на 200,000,000 проверки на шахматни конфигурации в секунда. С добавянето на алфа-бета отсичане дълбочината на дървото, която може да изследва този компютър, е била доста висока. Наред с това функцията `eval()` е била изключително прецизно съставена (за това са се погрижили няколко професионални шахматисти). Може да се каже, че компютърният хардуер в момента е преминал критичната точка на развитие, при която човек вече не може да бъде конкурент в играта на шахмат с една добре съставена шахматна програма.

Задачи за упражнение:

1. Да се предложи схема за оценяване на качеството на конфигурация при “*X*”-чета и “*O*”.
2. Да се реализира вариант на играта “*X*”-чета и “*O*” с използване на алфа-бета отсичане до определена дълбочина.

6.6. Въпроси и задачи

6.6.1. Задачи от текста

Задача 6.1.

Да се определи броят на различните шахматни конфигурации (*виж 6.1.*).

Задача 6.2.

Да се дадат примери за задачи от теорията на графите, които имат логаритмична, полиномиална или експоненциална сложност (*виж 6.1.*).

Задача 6.3.

На един и същи клас (*виж 6.1.*) ли принадлежат следните две задачи:

- Да се провери дали дадено естествено число n е просто.
- Да се провери дали съществува делител на дадено естествено число n, по-малък от дадено естествено число q ($q > 1$, $q < n$).

Задача 6.4.

Като се използва само *дефиниция 6.2.* да се докаже, че ако дадена NP-пълна задача A е сводима към друга задача B, то B е NP-пълна. (*виж 6.2.*)

Задача 6.5.

Да се дадат примери за задачи от теорията на графите, които принадлежат на класа NP-пълни задачи (*виж 6.2.*) посредством свеждане до търсене на Хамилтонов цикъл в граф (допускаме, че NP-пълнотата на задачата за търсене на Хамилтонов цикъл е доказана).

Задача 6.6.

Задачата "Да се провери дали съществува делител на дадено естествено число n, по-малък от дадено естествено число q ($q > 1$, $q < n$)." принадлежи към класа NP. А принадлежи ли към класа NP-пълни?

Задача 6.7.

Къде в схемата от *6.3.* е подходящо да се направи проверка за оптималност (например, ако се търси Хамилтонов цикъл *с минимална дължина*)?

Задача 6.8.

Да се предложат други критерии за отсичане на дървото на кандидатите за решение. (*виж 6.3.1.*)

Задача 6.9.

Удовлетворим ли е изразът $(\overline{X}_2 \vee X_3 \wedge X_1) \wedge X_2 \wedge (\overline{X}_1 \vee \overline{X}_3)$?

Задача 6.10.

Да се докаже тъждеството $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$, като се използват таблиците на истинност. (*виж 6.3.1.*)

Задача 6.11.

Да се докажат законите на Де Морган. (*виж 6.3.1.*)

Задача 6.12.

Като се използват законите на Де Морган да се докаже равенството $A \vee B = \overline{\overline{A} \wedge \overline{B}}$.

Задача 6.13. Равенството от предходната задача изразява дизюнкцията чрез конюнкция и отрицание. Да се предложи и докаже формула, която изразява *конюнкцията* чрез дизюнкция и отрицание.

Задача 6.14. Като се използват предходните две задачи, да се докаже, че множествата {¬, ∧} и {¬, ∨} имат същата изразителност като {¬, ∧, ∨}.

Задача 6.15. Вярно ли е, че множеството {∧, ∨} има същата изразителност като {¬, ∧, ∨}?

Задача 6.16.

Да се предложи алгоритъм за минимално оцветяване на върховете на граф, който намира минималния брой цветове *директно*, т.е. без да проверява последователно за всяко r ($1 \leq r \leq n$) дали графът може да се оцвети с r цвята. (*виж 6.3.2.*)

Задача 6.17.

Да се предложат още критерии за отсичане на дървото от кандидатите за решение в *6.3.2*.

Задача 6.18.

Да се предложи и реализира алгоритъм за минимално оцветяване на *ребрата* на граф. (*виж 6.3.2.*)

Задача 6.19.

Променя ли се принципно задачата за намиране на най-дълъг прост път в ориентиран претеглен цикличен граф, ако дължината на пътя се пресмята като *произведение* от теглата на ребрата, които съдържа. А ако дължината се определя от теглото на минималното (максималното) ребро, участващо в пътя? (*виж 6.3.3.*)

Задача 6.20.

Променя ли се принципно задачата за намиране на най-дълъг прост път в ориентиран цикличен граф, ако дължината на пътя се пресмята като сума от теглата на *върховете*, които съдържа (приемаме, че за всеки връх е зададено тегло — естествено число). (*виж 6.3.3.*)

Задача 6.21.

Да се модифицира програмата от *6.3.3.* така, че да отпечатва *всички* пътища с максимална дължина.

Задача 6.22.

Да се модифицира програмата от *6.3.4.* така, че да намира *всички* решения.

Задача 6.23.

Да се модифицира програмата от *6.3.4.* така, че да намира *всички различни несиметрични* решения. За симетрични ще считаме решения, които могат да се получат едно от друго чрез завъртане или чрез симетрия относно хоризонталите, вертикалите или двата главни диагонала на дъската.

Задача 6.24.

Да се намери броят на различните обхождания на дъската (*виж 6.3.4.*):

а) всички обхождания

б) всички различни несиметрични обхождания

Задача 6.25.

Дадено е естествено число n ($n > 4$) и обобщена шахматна дъска с размер $n \times n$ клетки. Да се намери обхождане на дъската с хода на коня. Всяка клетка трябва да бъде посетена точно веднъж, като "обиколката" започва от *произволна* предварително зададена клетка от дъската. (*виж 6.3.4.*)

Задача 6.26.

Дадено е естествено число n ($n > 4$) и обобщена шахматна дъска с размер $n \times n$ клетки, някои от които са *забранени*. Да се намери обхождане на дъската с хода на коня. Всяка незабранена клетка от дъската трябва да бъде посетена точно веднъж, като "обиколката" започва от клетката, намираща се в долния ляв ъгъл. (*виж 6.3.4.*)

Задача 6.27.

Дадено е естествено число n ($n > 4$) и обобщена шахматна дъска с размер $n \times n$ клетки. Да се намери обхождане на дъската с хода на коня с минимално самопресичане на траекторията на коня. (*виж 6.3.4.*)

Задача 6.28.

Да се определи класът на задачата за намиране на Хамилтонов път в граф с ограничена степен на върховете (т.е. степента на всеки връх е по-малка от дадено фиксирано естествено число k).

Задача 6.29.

Възможно ли е *задача 6.28.* да се сведе полиномиално до *задача 6.26*?

Задача 6.30.

Да се предложи и реализира алгоритъм за решаване на следната разновидност на задачата от *6.3.5.*: Да се отпечатат всички възможни *несиметричните* разположения на царините върху дъската. Две решения се смятат за симетрични, ако едното може да се получи от другото чрез завъртане на шахматната дъска или симетрия относно редовете, стълбовете, главния или вторичния диагонал.

Задача 6.31.

Има ли смисъл (и ако да — при какви допълнителни уточнения) задачата за обобщения случай на правоъгълна дъска? (*виж 6.3.5.*)

Задача 6.32.

Да се модифицира програмата от *6.3.6.* така, че да отпечатва разпределението за преподаване заедно с минималната продължителност на програмата.

Задача 6.33.

Каква е сложността на предложения в *6.3.6.* алгоритъм?

Задача 6.34.

Да се модифицира програма от *6.3.7.* така, че да намира превод с минимална дължина. Какви условия отсичане на дървото с кандидатите за решение могат да се приложат?

Задача 6.35.

Да се състави и реализира алгоритъм, решаващ *задача 1* от *6.4.1.*

Задача 6.36.

Да се модифицира програмата от *6.4.1.* така, че да намира всички решения с максимална цена.

Задача 6.37.

Да се напише програма, симулираща игра на "*X*"-чета и "*O*" между човек и компютър, като компютърът трябва да играе оптимално: да печели винаги, когато му се удаде такава възможност, т. е. при грешка на човека, и сам да не греши. Когато позицията на компютъра е неопределена, той може да направи произволен *негубещ* ход. (*виж 6.5.1.*)

Задача 6.38.

Да се напише програма за игра на "*X*"-чета и "*O*" в общия случай, когато е дадена дъска с размер $n \times n$, и печеливш е играчът, който направи редица, колона или диагонал от m знака. В най-разпространения вариант m е 5. (*виж 6.5.1.*)

Задача 6.39.

Да се докаже, че при дъска 3×3, ако двамата играчи играят правилно, играта задължително завършва с реми. (*виж 6.5.1.*)

Задача 6.40.

Да се намери броят на:

а) всевъзможни конфигурации, които могат да се получат по време на игра

б) всевъзможните конфигурации от дървото (да се сравни с а))

в) броят на листата в дървото (т.е. на терминалните конфигурации)

(*виж 6.5.1.*)

Задача 6.41.

Да се напише програма за симулиране на игра на "*X*"-чета и "*O*" между човек и компютър, реализираща принципа на минимума и максимума.

(*виж 6.5.2.*)

Задача 6.42.

Да се напише програма за симулиране на игра на "*X*"-чета и "*O*" между човек и компютър, реализираща принципа на минимума и максимума с алфа-бета отсичане. Да се сравни с варианта *без* алфа-бета отсичане. (*виж 6.5.3.*)

Задача 6.43.

Да се предложи схема за оценяване на качеството на конфигурация при "*X*"-чета и "*O*". (*виж 6.5.4.*)

Задача 6.44.

Да се реализира вариант на играта "*X*"-чета и "*O*" с използване на алфа-бета отсичане до определена дълбочина. (*виж 6.5.1.*)

6.6.2. Други задачи

- NP-пълни задачи

В последната част от главата ще приведем списък с най-известните *NP*-пълни задачи, придружен с кратки пояснения към някои от тях. *NP*-пълните задачи се появяват често на практика и тяхното разпознаване като такива може да ни спести доста усилия. Фактът, че дадена задача е *NP*-пълна, ще ни позволи да не губим време в търсене на бърз алгоритъм, а да се съсредоточим върху един от следните стандартни подходи за решаване:

- *Използване на евристичен алгоритъм* (*виж глава 9*). Ако задачата не може да се реши бързо във всички случаи, то все пак е възможно да съществува бърз метод, който да дава решение само в част от случаите.
- *Приблизително решаване на задачата* (*виж глава 9*). За някои *NP*-пълни задачи съществуват алгоритми за решаване с приближение. Те няма да решат проблема точно, но винаги ще намерят решение, за което може да се докаже, че е достатъчно близко до точното.
- Много практически задачи, които по своята същност са *NP*-пълни, можем да разделим на няколко *основни частни случаи*. За някои случаи е възможно да съществува

алгоритъм с полиномиална сложност с използването например на допълнителна памет (*динамично оптимиране — глава 8*).

- *Използване на пълно изчерпване*. Тук начинът на реализация също е важен. Възможно е за част от случаите в задачата да бъде сигурно, че няма да доведат до решение, което позволява изключването им от процеса на изследване и съответно намалява времевата сложност. Такива бяха разгледани в настоящата глава.

Голяма част от задачите, които ще разгледаме, са публикувани за пръв път през 1979 в "енциклопедията" на *NP*-пълните задачи — [*Garey,Johnson-1979*]. За пълнота, списъкът съдържа и задачите, които вече бяха разгледани (някои от тях решени). Всяка от задачите по-долу е разгледана по-следната схема:

- *Име на задачата* (и съкращение, ако такова е известно). Това е името, под което задачата може да се намери най-често в литературата.
- *Условие на задачата*. Състои се от две части: входни данни (често те са специфични структури — графи, множества, логически изрази и др.) и какво се търси. Последното е формулирано като въпрос, на който, за дадени конкретни входни данни, алгоритъмът трябва да отговори с *да* или *не*.
- *Коментар на задачата*. Тази част (която не винаги присъства) се състои от кратък коментар по задачата, който може да включва упътване (или по-малко известни подходи) за решаване на задачата, нейни разновидности, приложения и др. Тук е възможно да има и препратки към друга литература, разглеждаща задачата. Както казахме, много от задачите в този списък могат да бъдат намерени класифицирани в [*Garey,Johnson-1979*], затова няма да споменаваме тази книга при всяка една конкретна задача. Следва един пример:

Задача 6.45. *3-оцветяване на граф* (*3-COL*)

• Даден е граф.

• Да се провери възможно ли е всеки връх на графа да се "оцвети" (да му се съпостави) с един от три цвята: зелен, син, червен, по такъв начин, че да няма два съседни върха, оцветени с един и същ цвят.

Така поставена, задачата предполага намиране на решение за произволен граф. В условието се иска само да се провери, възможно ли е или — не, оцветяване на графа с 3 цвята. При много от задачите в този списък условието може да се обобщи така, че да се търси *оптимално* решение — за конкретния пример е възможно да търсим оцветяване с минимален брой цветове и т.н.

По-нататък задачите следват в нарастваща трудност. Задачите от началото на списъка са зададени в по-проста форма, в тях участва по-прост математически апарат, разгледани са обширно в литературата. Точно обратната ситуация има при задачите от края на списъка — условието е по-сложно и са по-малко известни.

Задача 6.46. *Хамилтонов път в граф*

• Даден е неориентиран граф.

• Да се провери съществува ли прост път, съдържащ всички върхове на графа.

• Задачата е разгледана в: *5.4.4.*, *6.2.*, *9.2.3.* и др.

Задача 6.47. *Максимален цикъл*

• Даден е неориентиран граф с n върха и естествено число k, $1 \leq k \leq n$.

• Да се провери съществува ли прост цикъл в графа, съдържащ поне k върха.

• Това е обобщение на задачата за намиране на Хамилтонов път. Фактически, задачата за проверка дали в граф има Хамилтонов цикъл се получава като частен случай на тази (ако положим $k = n$).

Задача 6.48. *Удовлетворимост на булева функция*

• Дадени са m дизюнкта C_1, C_2, ..., C_m на булевите променливи X_1, X_2, ..., X_n и техните отрицания.

• Да се провери съществува ли присвояване на стойност "истина" или "лъжа" на променливите X_1, X_2, ..., X_n, за което стойността на всички дизюнкти C_1, C_2, ..., C_m да бъде "истина".

• Задачата решихме в *6.3.1.*

Задача 6.49. *Най-дълъг път*

• Даден е неориентиран граф с n върха и естествено число k, $1 \leq k \leq n$. Фиксирани са два върха s, t от графа.

• Да се провери съществува ли прост път от s до t, в който участват поне k ребра.

• Вариант на задачата решихме в *6.3.3*.

Задача 6.50. *Задача за търговския пътник*

• Даден е неориентиран претеглен граф с n върха и естествено число k, $k \geq 1$.

• Да се провери съществува ли прост цикъл, в който участват всички върхове на графа, с дължина (като сума от теглата на ребрата) най-много k.

• Това е вариант на известната задача за намиране на минимален Хамилтонов цикъл в граф, която вече разгледахме неколкократно.

Задача 6.51. *Задача за раницата*

• Дадено е крайно множество A и на всеки елемент $a \in A$ е съпоставено естествено число $s(a)$. Дадено е естествено число k.

• Да се провери съществува ли подмножество $B \subseteq A$ такова, че сумата от стойностите $s(x)$ за $x \in B$ да бъде точно k.

• Това е *задача 1* от *6.4.1.* Нейния оптимизационен вариант решихме по метода на разклоненията и границите. Ще я разгледаме отново в *8.2.1.*, където ще я решим с динамично оптимиране.

Задача 6.52. *Задача за "Алан и Боб"*

• Дадено е крайно множество A. На всеки елемент a е съпоставено естествено число $s(a)$.

• Да се провери дали A може да се разбие на две множества A_1 и A_2, за които

$$\sum_{x \in A_1} s(x) = \sum_{y \in A_2} s(y) \cdot$$

• Името на задачата "Алан и Боб" идва от практическата задача, да се разделят няколко подаръка (всеки от тях с определена стойност) между двама братя (Алан и Боб) така, че сумата от стойностите на подаръците на единия да бъде равна на сумата от стойностите на подаръците на другия. Задачата е разгледана в *8.2.2*.

Задача 6.53. *Произведение от подмножество.*

• Дадено е крайно множество A и на всеки елемент $a \in A$ е съпоставено естествено число $s(a)$. Дадено е естествено число k.

• Да се провери съществува ли подмножество $B \subseteq A$ такова, че произведението на елементите $s(x)$, $x \in B$ да бъде точно равно на k.

• Това е вариант на задачата за раницата.

Задача 6.54. *Пакетиране на кутии*

• Дадено е крайно множество A с n елемента. На всеки елемент $a \in A$ е съпоставено естествено число $s(a)$. Дадени са естествените числа b и k, $1 \leq k \leq n$.

• Да се провери може ли A да се разбие на k множества A_1, A_2, ..., A_k така, че за всяко A_i ($1 \leq i \leq k$) сумата от стойностите на елементите му да не надхвърля b.

• Задачата представлява обобщение на задачата за раницата — дадени са k раници, всяка от които може да побере b килограма, и трябва да разпределят предмети в тях. Възможен е и оп-

тимизационен вариант на задачата — да се намери *минималното* k, за което предметите могат да се разпределят по описания начин.

Задача 6.55. *Покриване на квадрат*

• Дадени са n цвята (означени с целите числа $1,2,...,n$) и списък от оцветени квадрати със страна единица. Всеки квадрат е зададен с четворка (a, b, c, d) показваща, че горната страна е оцветена с цвят a, долната — с цвят b, лявата — с цвят c и дясната — с цвят d. Дадено е естествено число k, $1 \le k \le n$.

• Да се провери съществува ли запълване на квадрат с размери $k \times k$, като се използват дадените квадратчета така, че допиращите им се страни да бъдат оцветени с еднакъв цвят.

• [*Келеведжиев-5/1998*]

Задача 6.56. *Кръстословица*

• Дадено е множество от n символа $S = \{s_1,s_2,...,s_n\}$ и множество $W = \{w_1,w_2,...,w_{2n}\}$ от $2n$ думи такова, че всяко w_i е последователност от точно n символа от S.

• Да се провери може ли от дадените $2n$ думи да се получи кръстословица с размер $n \times n$, т. е. ако C е матрица с размер $n \times n$, може ли във всяка клетка да се запише символ от S така, че всеки ред и стълб от матрицата да представлява някоя дума от W.

Задача 6.57. *Независимо множество в граф*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \le k \le n$.

• Да се провери дали съществува подмножество $U \subseteq V$, съдържащо поне k върха, в което не съществуват два върха $i,j \in U$ такива, че $(i,j) \notin E$.

• Вариант на задачата е разгледан в *5.7.2*.

Задача 6.58. *Граф без триъгълник*

• Даден е неориентиран граф $G(V,E)$.

• Да се провери възможно ли е множеството E да се разбие на две подмножества E_1 и E_2 така, че в никое от тях да няма тройка ребра от вида (i,j), (j,k), (k,i) (т. е. в никой от двата граф $G_1(V,E_1)$, $G_2(V,E_2)$ да няма цикъл с дължина 3).

Задача 6.59. *Доминиращо множество*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \le k \le n$.

• Да се провери дали G съдържа подмножество U, $U \subseteq V$ с най-много k върха и такова, че за всеки връх $i \in V$, $i \notin U$ да съществува връх $j \in U$ такъв, че $(i,j) \in E$.

Задача 6.60. *Ребрено хроматично число*

• Даден е неориентиран граф $G(V,E)$ с m ребра и естествено число k, $1 \le k \le m$.

• Да се провери може ли на всяко ребро да се съпостави уникално естествено число от 1 до k (да се оцвети реброто с цвят от 1 до k) така, че да няма две инцидентни ребра, оцветени с еднакъв цвят.

• *Теоремата на Визинг* показва, че единствения сложен случай при тази задача е, когато k е равно на максималната степен на връх от графа. *NP*-пълнотата на тази задача е доказана наскоро. За решаването ѝ съществуват много *евристични* и *паралелни* алгоритми [*Gibbons,Rytter-1987*].

Задача 6.61. *Най-къс общ подниз*

• Дадено е множество $S = \{s_1,s_2,...,s_n\}$ от двоични низове (последователност от 0 и 1) и естествено число k.

• Да се провери съществува ли низ s с дължина най-много k такъв, че всеки низ $p \in S$ да бъде подниз на s.

• Обобщението на задачата е, когато в низовете могат да участват повече различни символи, но дори и в този по-прост случай задачата остава *NP*-пълна.

Задача 6.62. *Задача за провинциалния пощальон*

• Даден е граф $G(V,E)$ с m ребра, подмножество E_1, $E_1 \subseteq E$ и естествено число k, $1 \leq k \leq m$.

• Да се провери съществува ли цикъл в графа (не задължително прост), съдържащ всяко ребро $e \in E_1$ поне веднъж и с общ брой ребра, по-малък или равен на k.

Задача 6.63. *"Превъзходстващо" деление*

• Дадени са две строго нарастващи редици $A = \langle a_1,a_2,...,a_n \rangle$ и $B = \langle b_1,b_2,...,b_m \rangle$ от цели положителни числа.

• С $Div(x,Y)$ (Y е редица от естествени числа) ще означим броя елементи $y \in Y$, такива че x е точен делител на y. Да се провери съществува ли естествено число c, за което $Div(c, A) > Div(c, B)$.

• Тази и следващите три задачи са специален вид *NP*-пълни задачи — *от целочислено програмиране*. В конкретната задача можем да търсим c последователно, започвайки от 1. Този алгоритъм обаче е неефективен, а задачата е *NP*-пълна. Това се забелязва, ако се сравни размерът на входните данни (броя битове за представяне на числото в двоична бройна система) със сложността на алгоритъма в най-лошия случай — имаме експоненциална функция.

Задача 6.64. *Квадратни Диофантови уравнения*

• Дадени са естествените числа a, b, c.

• Да се провери съществуват ли цели числа x и y такива, че $(a.x^2) + (b.y) = c$

• Тук важи същият коментар, както и при *задача 19* — последователното търсене на числа x и y е с експоненциална сложност по броя на битовете, необходими за представяне на числата a, b и c.

Задача 6.65. *Квадратично сходство*

• Дадени са цели положителни числа a, b и c.

• Да се провери съществува ли естествено число x, $x \leq c$ такова, че $x^2 \equiv a \pmod b$.

Задача 6.66. *Едновременно несходство*

• Дадено е множество от n наредени двойки естествени числа $\{(a_1,b_1), (a_2,b_2), ..., (a_n,b_n)\}$, като $a_i \leq b_i$, $1 \leq i \leq n$.

• Да се провери съществува ли цяло число x такова, че $a_i \neq x \pmod{b_i}$, за всяко $1 \leq i \leq n$.

Задача 6.67. *Компресиране на изображение*

• Дадено е естествено число k и матрица M с размер $n \times n$, съдържаща само 0 и 1.

• Да се провери могат ли всички единици в M да бъдат покрити с най-много k правоъгълника, т. е. съществува ли множество от наредени четворки (a_i, b_i, c_i, d_i), $1 \leq i \leq k$, $a_i \leq b_i$, $c_i \leq d_i$ такива, че:

- За всяка двойка (i,j), $1 \leq i, j \leq n$ стойността на `M[i][j] == 1` тогава и само тогава, когато за някое l $(1 \leq l \leq k)$ е изпълнено $a_l \leq i \leq b_l$ и $c_l \leq j \leq d_l$.
- Правоъгълниците са съставени изцяло от единици: за всяко l $(1 \leq l \leq k)$, i $(a_l \leq i \leq b_l)$, j $(c_l \leq j \leq d_l)$ е изпълнено `M[i][j] == 1`.

• Зад сложното условие, стои една практическа задача (както показва и името й): Да се провери дали двумерно черно-бяло изображение може да се представи с най-много k запълнени правоъгълни блока данни.

Задача 6.68. *Път със забранени двойки*

• Даден е ориентиран граф, два негови фиксирани върха s и t и списък от r двойки върхове $C = \{(a_1,b_1), (a_2,b_2), ..., (a_r,b_r)\}$.

• Да се провери съществува ли път от s до t такъв, че за всяка двойка $(a_i,b_i) \in C$ най-много един от двата върха a_i, b_i да участва в него.

Задача 6.69. *Максимален двуделен подграф*

• Даден е неориентиран граф $G(V,E)$ с n върха и m ребра и естествено число k, $1 \le k \le m$.

• Да се провери съществува ли подмножество E_1 на E с поне k ребра такова, че графът $G(V,E_1)$ да бъде двуделен.

• Оптимизационният вариант на задачата може да изглежда така: Дадени са n студенти $s_1,s_2,\dots,s_n$, като някои се познават помежду си. Да се изберат *максимален* брой от тях така, че да могат да се разделят в две групи и всеки (евентуално) да познава някого (някои) от другата група и никого от своята собствена.

Задача 6.70. *Разбиване на граф*

• Даден е неориентиран граф $G(V,E)$ с $2n$ върха и m ребра и естествено число k, $1 \le k \le m$.

• Да се провери могат ли върховете на G да се разбият на две множества U и W, всяко с по n-елемента и такива, че броят на ребрата $(i,j) \in E$, $i \in U$, $j \in W$ да бъде най-много k.

• Оптимизационният вариант на задачата намира приложение при дизайна на чипове, тя представлява естествено формулиране на задачата за минимизиране на броя на връзките между отделните чипове в електронна схема.

Задача 6.71. *Циклично нареждане*

• Дадено е крайно множество A с n елемента и списък C от наредени тройки (a, b, c) различни елементи от A.

• Нека X и Y са две множества с по n елемента. Ще припомним, че *биекция* между множествата X и Y се нарича функция $f:X \to Y$, съпоставяща еднозначно на *всеки* елемент от X *уникален* елемент от множеството Y и обратно: за всеки елемент от Y има съответен първообраз от X. В частност, $f:X \to \{1,2,\dots,n\}$ съпоставя на всеки елемент от X уникално цяло число от 1 до n.

Да се провери съществува ли биекция $f:X \to \{1,2,\dots,n\}$, при която за всеки елемент $(a,b,c) \in C$ да бъде изпълнено точно едно от трите съотношения:

$$f(a) < f(b) < f(c)$$
$$f(b) < f(c) < f(a)$$
$$f(c) < f(a) < f(b)$$

Задача 6.72. *Вместване*

• Дадено е крайно n-елементно множество A и множество C от наредени тройки (a, b, c), където a, b, c са три различни елемента от A.

• Да се провери съществува ли биекция $f:A \to \{1,2,\dots,n\}$ такава, че за всяка тройка $(a, b, c) \in C$ да бъде изпълнено $f(a) < f(b) < f(c)$ или $f(c) < f(b) < f(a)$.

Задача 6.73. *Централен низ*

• Дадено е множество S от k двоични низа, всеки с дължина n и естествено число r.

• *Разстояние по Хеминг* $H(x,y)$ между два низа x и y се дефинира като броя позиции, в които битът в низа x се различава от съответния бит в y. Да се провери съществува ли низ x с дължина n бита такъв, че за всеки низ y, $y \in S$ да бъде изпълнено $H(x,y) \le r$.

• *NP*-пълнотата на тази задача е доказана съвсем наскоро [*Frances-Litman-1997*].

Задача 6.74. *B-нареждане*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \le k \le n$.

• Да се провери съществува ли линейно нареждане на върховете на G, т. е. биекция $f:V \to \{1,2,\dots,n\}$ такава, че за всяко ребро $(i,j) \in E$ да бъде изпълнено $|f(i)-f(j)| \le k$.

Задача 6.75. *Кубичен подграф*

• Даден е неориентиран граф $G(V,E)$.

• Да се провери съществува ли непразно множество E_1, $E_1 \subseteq E$ такова, че всеки връх от подграфа $G_1(V,E_1)$ е или от степен 3, или от степен 0.

Задача 6.76. *Покритие на върхове*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \leq k \leq n$.

• Да се провери съществува ли подмножество $U \subseteq V$ с най-много k върха, в което за всяко ребро $(i,j) \in E$ поне един от върховете i,j принадлежи на U.

Задача 6.77. *Оптимално линейно нареждане*

• Даден е граф $G(V,E)$ с n върха и естествено число k, $1 \leq k \leq n$.

• Да се провери съществува ли биекция $f:V \to \{1,2,...,n\}$, за което

$$\sum_{(i,j)\in E} | f(i) - f(j) | \leq k$$

Задача 6.78. *Непресичащи се пътища*

• Даден е неориентиран граф $G(V,E)$ с n върха и множество от k-различни двойки върхове $\{(s_1,t_1), (s_2,t_2),..., (s_k,t_k)\}$.

• Да се провери съществуват ли k непресичащи се (без общи върхове) пътища с начало s_i и край t_i.

Задача 6.79. *Разделяне на ациклични подграфи*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \leq k \leq n$.

• Да се провери може ли множеството от върхове V да се разбие на t, $1 \leq t \leq k$ подмножества V_1, V_2, ..., V_t така, че никой индуциран от V_i подграф $G_i(V_i,E_i) \subseteq G$ да не съдържа цикли.

Задача 6.80. *Най-голям общ подграф*

• Дадени са неориентирани графи $G_1(V_1,E_1)$, $G_2(V_2,E_2)$ и естествено число k.

• Да се провери съществуват ли подмножества $F_1 \subseteq E_1$ и $F_2 \subseteq E_2$ такива, че:

- Броят на елементите на F_1 и F_2 да бъде еднакъв и този брой да бъде по-голям или равен на k.
- Графите $G_1(V_1,F_1)$, $G_2(V_2,F_2)$ да бъдат изоморфни.

Задача 6.81. *Максимална 2-удовлетворимост*

• Дадено е множество от дизюнкти $C_1,C_2,...,C_m$ на n булеви променливи $X = \{X_1, X_2, ..., X_n\}$, където всеки дизюнкт се състои от две различни булеви променливи X_i, $X_j \in X$. Дадено е естествено число k, $1 \leq k \leq m$.

• Да се провери съществува ли присвояване на стойност "истина" или "лъжа" на всяка булева променлива такова, че поне k дизюнкта да имат стойност "истина".

• В случая $k = m$ се получава по-прост случай на *задача 6.4.* — тя се нарича *2-SAT* и за нейното решение съществува полиномиален (по m) алгоритъм.

Задача 6.82. *Максимален пълен подграф* (*k-клика*)

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \leq k \leq n$.

• Да се провери съществува ли подмножество $U \subseteq V$ от k върха такова, че индуцираният от U подграф $G'(U,E')$ да бъде пълен, т. е. за всеки върха $i \in U$, $j \in U$ да съществува реброто $(i,j) \in E'$.

Задача 6.83. *Централно разрязване на полуклика*

• Даден е неориентиран граф $G(V,E)$ с $2n$ върха.

• Да се провери изпълнени ли са следните условия:

- Графът има поне $(2n^2 - n)/2 + 1$ ребра.
- Графът има точно $(2n^2 - n)/2 + 1$ ребра и съдържа n-клика.

Задача 6.84. *Ядро на граф*

• Даден е ориентиран граф $G(V,E)$.

• Да се провери съществува ли *ядро*, т.е. подмножество U, $U \subseteq V$, за което е изпълнено:

- никои два върха $i,j \in U$ не са свързани с ребро, т.е $(i,j) \notin E$
- за всеки връх $i \in V \setminus U$, съществува връх $j \in U$ такъв, че $(i,j) \in E$.

Задача 6.85. *Двойкосъчетание в триизмерен граф*

• Дадени са три непресичащи се множества X, Y, Z всяко с по n елемента и множество M от m наредени тройки (x_i, y_i, z_i), $x_i \in X$, $y_i \in Y$, $z_i \in Z$, $1 \le i \le m$.

• Да се провери съществува ли подмножество $Q \subseteq M$ с n елемента и такова, че за всеки две различни тройки $(i, j, k) \in Q$, $(u, v, w) \in Q$ да бъде изпълнено $i \ne u$, $j \ne v$, $k \ne w$.

• Съществуват по-прости варианти на задачата — например двойкосъчетание в двуделен граф (*виж 5.7.6.*), които се решават с полиномиална сложност.

Задача 6.86. *Разделяне на множество*

• Дадени са крайно множество S и m негови подмножества $C_1, C_2, ..., C_m$.

• Да се провери може ли S да се разбие на две подмножества S_1 и S_2 така, че C_i , i = 1,2,...,m да не е подмножество нито на S_1, нито на S_2.

Задача 6.87. *Пакетиране на множество*

• Дадени са крайните множества $C_1, C_2, ..., C_m$ и естествено число k, $1 \le k \le m$.

• Да се провери могат ли да се изберат k множества от дадените така, че никои две от избраните да нямат общ елемент.

Задача 6.88. *Ограничена сума на квадрати*

• Дадено е множество A с n елемента и на всяко $a \in A$ е съпоставено естествено число $s(a)$. Дадени са две естествени числа j и k, $1 \le k \le n$.

• Да се провери може ли A да се разбие на k подмножества $A_1, A_2, ..., A_k$ така, че за i = 1,2,…,k:

$$\sum_{x \in A_i} s(x)^2 \le j$$

Задача 6.89. *3-разделяне*

• Дадено е множество A с $3m$ елемента и естествено число b. На всеки елемент $a \in A$ е съпоставено естествено число $s(a)$, за което е изпълнено $b/4 < s(a) < b/2$. Числата са такива, че $\sum_{a \in A} s(a) = mb \cdot$

• Да се провери може ли A да се разбие на m подмножества A_1, A_2, ..., A_m такива, че всяко множество A_i да съдържа точно 3 елемента от A и сборът от стойностите на елементите в A_i (за i = 1,2,...,m) да бъде точно равен на b.

Задача 6.90. *"Разпознаване на грешки" в ориентиран граф*

• Даден е ориентиран, ацикличен граф $G(V,E)$ и точно един от върховете му t е без изходящи ребра. Нека U е подмножество на V, образувано от всички върхове, в които няма входящи ребра. Дадено е естествено число k.

• Да се провери съществува ли "*тестово множество*" с най-много k елемента, което може да провери всяка "*проста грешка*" в графа, т. е. да се провери съществува ли подмножество T, $T \subseteq U$ такова, че:

- T да съдържа най-много k елемента.
- За всеки връх $i \in V$ да съществува връх $j \in T$, такъв че i принадлежи на прост път от j до t.

Задача 6.91. *Минимално "губене на време"*

• Даден е неориентиран граф $G(V,E)$ с n върха и подмножество $V_0 \subseteq V$.

• Да се провери може ли "съобщение" да бъде "разпратено" от база от върхове V_0 до всички върхове от V за най-много 4 стъпки, т. е. съществува ли последователност V_0, E_1, V_1, E_2, V_2, E_3, V_3, E_4, V_4 такава, че:

- V_i е подмножество на V, за $0 \le i \le 4$.
- E_i е подмножество на E, за $1 \le i \le 4$.
- $V_4 \equiv V$.

- За всяко ребро $(u,v) \in E_i$ е изпълнено $v \in V_{i-1}$, $1 \le i \le 4$.
- Не съществуват две ребра от E_i, инцидентни с един и същ връх, $1 \le i \le 4$.
- $V_i = V_{i-1} \cup \{w: (v,w) \in E_i\}$, $1 \le i \le 4$.

Задача 6.92. *Най-къс ограничен път*

• Даден е неориентиран претеглен граф $G(V,E)$ с n върха. На всяко ребро $(i,j) \in E$ освен *тегло* $e(i,j)$ е съпоставено число $l(i,j)$, което ще наричаме *дължина* на реброто. Дадени са два фиксирани върха s и t от графа и две естествени числа k и w.

• Да се провери съществува ли път от s до t, който:
- Да има сума от тегла на ребрата най-много w.
- Да има сума от дължини на ребрата най-много k.

• Когато всички ребра имат еднаква дължина, се получава стандартната задача за намиране на най-кратък път в граф (*виж 5.4.2.*), която се решава в полиномиална сложност.

Задача 6.93. *Минимално-максимално двойкосъчетание*

• Даден е неориентиран граф $G(V,E)$ с m ребра и естествено число k, $1 \le k \le m$.

• Да се провери съществува ли подмножество $E_1 \subseteq E$ с най-много k ребра, което да формира максимално двойкосъчетание в графа, т. е. никои две ребра от E_1 да не са инцидентни с един и същ връх и всички ребра $e \in E$, $e \notin E_1$ да са инцидентни с поне един връх от E_1.

Задача 6.94. *Разделяне на триъгълници*

• Даден е неориентиран граф $G(V,E)$ с $3n$ върха.

• Да се провери може ли V да се разбие на n непресичащи се подмножества V_1, V_2,...,V_n (такива, че обединението им да дава V), всяко съдържащо точно 3 елемента, по-такъв начин, че за всяко $V_i = \{u_i,v_i,w_i\}$ и трите ребра (u_i, v_i), (v_i, w_i), (w_i, u_i) да принадлежат на E, $1 \le i \le n$.

Задача 6.95. *Разделяне на пълни подграфи*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \le k \le n$.

• Да се провери може ли множеството от върхове V да се разбие на t, $1 \le t \le k$ непресичащи се негови подмножества V_1, V_2,...,V_t (такива, че обединението им да дава V), по такъв начин, че всеки индуциран от V_i подграф $G_i(V_i,E_i) \subseteq G$ да бъде пълен.

Задача 6.96. *Разделяне на перфектно двойкосъчетание*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \le k \le n$.

• Да се провери може ли множеството от върхове V да се разбие на t, $1 \le t \le k$ непресичащи се подмножества V_1, V_2,...,V_t (така, че обединението им да дава V) по такъв начин, че всеки индуциран от V_i подграф $G_i(V_i,E_i) \subseteq G$ да съдържа перфектно двойкосъчетание, т. е. всеки връх от него да бъде инцидентен точно с едно ребро.

Задача 6.97. *Покритие с пълни подграфи*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \le k \le n$.

• Да се провери съществуват ли t, $t \le k$ множества V_1, V_2,...,V_t, подмножества на V (не задължително непресичащи се), такива че да са изпълнени следните две условия:
- Индуцираният от V_i подграф $G_i(V_i,E_i)$ на G да бъде пълен.
- За всяко ребро $(u,v) \in E$ да съществува множество V_i, съдържащо едновременно u и v.

Задача 6.98. *Ограничен по степен свързан подграф*

• Даден е неориентиран граф $G(V,E)$ с n върха и m ребра и естествено число k, $1 \le k \le m$. Дадено е естествено число d, $d \le n$.

• Да се провери съществува ли подмножество $E_1 \subseteq E$ с k елемента такова, че подграфът $G(V,E_1) \subseteq G$ да бъде свързан и в него да няма връх от степен, по-голяма от d.

Задача 6.99. *Несвързан подграф*

• Даден е ориентиран граф $G(V,E)$ с m ребра и естествено число k, $1 \le k \le m$.

• Да се провери съществува ли подмножество $E_1 \subseteq E$ с поне k елемента такова, че подграфът $G(V,E_1) \subseteq G$ да съдържа най-много един път между всяка двойка върхове $i, j \in V$.

Задача 6.100. *Минимален k-свързан подграф*

• Даден е неориентиран граф $G(V,E)$ с n върха и m ребра. Дадени са естествените числа k и b, $1 \le k \le n$ и $1 \le b \le m$.

• Да се провери съществува ли подмножество $E_1 \subseteq E$ най-много b елемента такова, че подграфът $G_1(V,E_1) \subseteq G$ да бъде k-свързан.

Задача 6.101. *Линейно нареждане с минимален разрез*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k.

• Да се провери съществува ли биекция $f:V \to \{1,2,...,n\}$ такава, че за всяко i, $1 \le i \le n$ броят на ребрата $(u,v) \in E$, за които $f(u) \le i \le f(v)$, да бъде най-много k.

Задача 6.102. *Разграничаване на пътища*

• Даден е ориентиран ацикличен граф $G(V,E)$ m ребра, два негови върха s, t и естествено число k, $1 \le k \le m$.

• Да се провери съществува ли подмножество $E_1 \subseteq E$ с най-много k ребра и такова, че за всяка двойка пътища P и Q с начален връх s и краен връх t (без общи върхове) да съществува ребро $e \in E_1$, което участва в точно един от пътищата P и Q.

Задача 6.103. *Индуциран път*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \le k \le n$.

• Да се провери съществува ли множество U, $U \subseteq V$ с p ($p \ge k$) върха и такова, че подграфът $G_1(U,E_1) \subseteq G$, индуциран от U, да представлява прост път през точно p върха.

Задача 6.104. *Ограничена покриваща гора*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $k \le n$.

• Да се провери съществуват ли t, $t \le k$ подмножества V_1, V_2,...,V_t на V (не задължително непресичащи се) такива, че да са изпълнени следните две условия (за $1 \le i \le t$):

- Подграфът $G_i(V_i,E_i) \subseteq G$, индуциран от V_i, да бъде свързан.
- Броят на елементите на V_i да бъде най-много 4.

Задача 6.105. *Максимално съчетание на подграфи*

• Дадени са ориентираните графи $G_1(V_1,E_1)$ и $G_2(V_2,E_2)$ и естествено число k.

• Да се провери съществува ли подмножество U на $V_1 \times V_2$ (т. е. множество от двойки върхове i,j такива, че $i \in V_1$, $j \in V_2$) с поне k елемента и такова, че за всеки две двойки $<u,v> \in U$ и $<x,y> \in U$, реброто $(u,x) \in E_1$ тогава и само тогава, когато реброто $(v,y) \in E_2$.

Задача 6.106. *Числено триизмерно съчетание*

• Дадени са непресичащи се множества W, X, Y, всяко с по m елемента. На всеки елемент $u \in W \cup X \cup Y$ е съпоставено естествено число $f(u)$. Дадено е естествено число b.

• Да се провери може ли $3m$-елементното множество $W \cup X \cup Y$ да бъде разбито на m непресичащи се подмножества $A_1,A_2,...,A_m$ по такъв начин, че да бъде изпълнено:

- Всяко множество A_i да съдържа точно един елемент от W, точно един от X и точно един от Y.
- $\sum_{u \in A_i} f(u) = b$, за всяко A_i, $1 \le i \le n$.

Задача 6.107. *Клъстеризация на множество*

• Дадено е крайно множество X. На всяка двойка елементи (x,y), $x,y \in X$ е съпоставено естествено число $d(x,y)$. Дадено е естествено число b.

• Да се провери съществува ли разбиване на X на три непресичащи се множества X_1, X_2, X_3 такива, че за всяко X_i $(1 \leq i \leq 3)$ и всяка двойка елементи $x,y \in X_i$ да бъде изпълнено $d(x,y) \leq b$.

Задача 6.108. *2-покриване*

• Даден е неориентиран граф $G(V,E)$ и естествено число s.

• Да се провери съществува ли подмножество $E_1 \subseteq E$ с най-много s ребра и такова, че за всяко ребро $(v,w) \in E$ да следва, че: 1) $(v,w) \in E_1$; или 2) $(v,x) \in E_1$ и $(x,w) \in E_1$ за някое $x \in V$.

Ако такова множество бъде намерено, графът $G_1(V,E_1)$ се нарича *2-покриване* на графа $G(V,E)$.

• *2-покриване* (и в по-общия случай *t-покриване*) е естествено обобщение на идеята за намиране на покриващо дърво в граф — за t-покриване не се иска нито да бъде дърво, нито свързан граф.

Задача 6.109. *Максимално по брой листа покриващо дърво*

• Даден е неориентиран граф $G(V,E)$ и естествено число k, $1 \leq k \leq n$.

• Да се провери дали G съдържа покриващо дърво, в което поне k ребра са от *първа степен*, т.е. инцидентни точно с един връх.

• [*Dunne-1987*]

Задача 6.110. *Свиване на степента на покриващо дърво*

• Даден е неориентиран граф $G(V,E)$ и естествено число k, $1 \leq k \leq n$.

• Да се провери дали G съдържа покриващо дърво, в което няма връх от степен, по-висока от k.

Задача 6.111. *Граничен диаметър на покриващо дърво*

• Даден е неориентиран претеглен граф с тегла на ребрата естествени числа. Дадено е естествено число b.

• Да се провери дали графът съдържа покриващо дърво такова, че:

- да не съдържа прост път с повече от 5 върха.
- сумата от теглата на ребрата му да бъде по-малка от b.

Задача 6.112. *Ориентиран диаметър*

• Даден е неориентиран граф $G(V,E)$ с n върха и естествено число k, $1 \leq k \leq n$.

• Да се провери може ли ребрата на графа да се ориентират по такъв начин, че за получения ориентиран граф $G_1(V,E_1)$ да бъде изпълнено:

- $G_1(V,E_1)$ да бъде силно свързан.
- диаметърът на $G(V,E_1)$ да бъде най-много k.

Задача 6.113. *Минимален "и-или" подграф*

• Даден е ориентиран ацикличен граф $G(V,E)$ с n върха, съдържащ връх s без изходящи ребра. Всеки връх на графа е маркиран като *"и-връх"* или *"или-връх"*. Дадено е естествено число k.

• Да се провери съществува ли подграф $G_1(V_1,E_1) \subseteq G$, за който е изпълнено:

- $s \in V_1$
- ако $w \in V_1$ и w е *"и-връх"*, тогава *всички* ребра, излизащи от w, трябва да принадлежат на E_1.
- ако $w \in V_1$ и w е *"или-връх"*, тогава *поне едно* ребро, излизащо от w, трябва да принадлежи на E_1.
- E_1 съдържа най-много k ребра.

Задача 6.114. *Минимално покриващо дърво от пътища*

• Даден е неориентиран граф $G(V,E)$ и естествено число b.

• Да се провери дали графът съдържа покриващо дърво, за което сумата от дължината на пътя между върховете u и v, за всички $u,v \in V$, да бъде по-малка или равна на b.

Задача 6.115. *Двупроцесорно разписание*

• Дадено е множество T от *задачи*. Всяка задача $t \in T$ е с времетраене $l(t)$ равно на 1 или 2. В множеството от задачите T има частична наредба. Дадено е още естествено число d (*deadline*).

• *Двупроцесорно разписание,* за множеството от задачи T и ограничител d се нарича биекция $\Xi : T \to \{1,2,...,n\}$, за която е изпълнено:

- За всяко $u \geq 0$ броят задачи t от T, за които $\Xi(t) \leq u \leq \Xi(t) + l(t)$ е най-много 2.
- За всяка задача $t \in T$, $\Xi(t) + l(t) \leq d$.

Да се провери съществува ли двупроцесорно разписание Ξ такова, че ако i е по-напред в наредбата от j да бъде изпълнено $\Xi(j) \geq \Xi(i) + l(i)$.

Задача 6.116. *Двупроцесорно разписание - 2*

• Дадено е множество T от задачи. Всяка задача $t \in T$ е с времетраене $l(t)$. Дадено е естествено число d.

• Да се провери съществува ли *двупроцесорно* разписание, което завършва за време по-кратко от d.

Задача 6.117. *Достатъчно регистри*

• Даден е ориентиран ацикличен граф $G(V,E)$ с n върха, в който всеки връх има полустепен на изхода най-много 2 (т. е. има най-много две изходящи ребра). Дадено е естествено число k.

• Да се провери съществува ли нареждане на върховете $(v_1,v_2,...,v_n)$ и списък $(V_0,V_1,...,V_n)$ от подмножества на V, за които:

- V_i съдържа най-много k върха от V $(1 \leq i \leq n-1)$.
- V_0 не съдържа върхове, V_n съдържа всички върхове без входящи ребра (с полустепен на входа 0).
- За всяко $i = 1,2,...,n$ е изпълнено: $v_i \in V_i$; множеството $V_i \backslash \{v_i\}$ е подмножество на V_{i-1}; V_{i-1} съдържа всички върхове u, за които $(v_i,u) \in E$.

• Тази задача намира приложение при компилаторите за езици от високо ниво. Изпълнението на аритметични операции над операнди в регистрите води до значително по-бързи операции, отколкото ако те се намират в паметта. Тези регистри обаче са ограничен брой (обикновено 16 или 32), поради което при компилиране и превеждане до асемблерски код е необходимо регистрите да се използват ефикасно. Първата стъпка от компилирането на аритметични изрази е те да се представят като "праволинейни програми" (от англ. *straight-line programs*), които могат да се моделират с ориентиран ацикличен граф. Задачата "Достатъчно регистри" се състои в това, да се определи "праволинейна програма", която да може да работи като използва само наличните k регистри.

Задача 6.118. *Дърво на решенията*

• Дадена е булева логическа функция f на n променливи $X_1, X_2,...,X_n$, описана с множество от двоични вектори $A = <a_1, a_2,...,a_n>$, за които $f(a_1, a_2,...,a_n) = 1$. Дадено е естествено число k.

• Да се провери съществува ли *дърво на решенията* за f такова, че *средната дължина на пътя* да бъде най-много k.

• *Дърво на решенията* е двоично дърво, в което:

- на всеки връх, който не е листо, е съпоставена една от променливите $X_1, X_2,...,X_n$.
- на всяко листо е съпоставена стойност 0 или 1.
- реброто от връх, който не е листо, към неговия ляв наследник се маркира с 0.
- реброто от връх, който не е листо, към неговия десен наследник се маркира с 1.

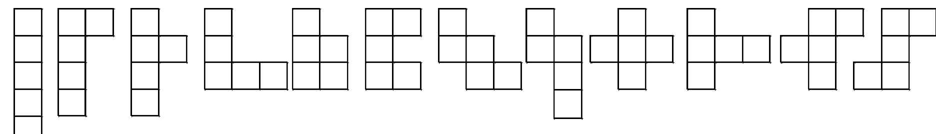
За дадено присвояване на булеви стойности на променливите $(X_1, X_2,...,X_n)$ път може да бъде проследен, като се започне от корена и се следват ребрата със стойности, които съвпадат със стойността на съответната променлива. Пътят завършва във връх-листо и стойността му дава

стойността на функцията. Дървото на решенията изчислява булевата функция $f(X_1, X_2,...,X_n)$, ако за всяко присвояване A на променливите проследеният път от корена стига до листо, чиято стойност е $f(A)$. *Средна дължина на пътя* на двоично дърво с t върха-листа и корен v е сумата от дължините на пътищата от v до всички върхове листа, разделена на t.

- Изчерпващи задачи

Задача 6.119. *Пентамино*

Да се намерят всички начини за покриване на правоъгълна дъска с размер $m \times n$ клетки с плочки от пентамино без застъпване. Плочките от пентамино представляват 5 свързани квадрата със страна 1 и са показани (без ротации и симетрии) на *фигура 6.6.2.*



Фигура 6.6.2. Плочките от пентамино.

Така зададената задача е обобщение на оригиналната задача за пентамино, където площта на дъската е фиксирана — 60 (т.е. $m.n = 60$) и се търсят всички решения, в които всяка плочка от даден вид участва точно един път в решението.

Задача 6.120. *n-мино*

n-мино са всички фигури, съставени от n-свързани квадрата със страна 1. Да се генерират всички различни такива фигури за дадено n. Две фигури от n-мино са еднакви, ако при налагане съвпадат или едната се получава от другата чрез ротация или симетрия.

Задача 6.121. *Прост магически квадрат*

Прост магически квадрат от ред n е таблица с размер $n \times n$, във всяка клетка от която е записано едно цяло число от 1 до n^2 така, че сумата от числата, записани във всеки хоризонтал и вертикал, е една и съща и е равна на $(n/2).(n^2+1)$. Да се намерят всички прости магически квадрати с размер $n \times n$.

Задачи 6.122. *Латински квадрат*

Да се намерят всички латински квадрати от ред n (*виж задача 1.30.*).

Други задачи, при които успешно може да се прилага търсене с връщане и/или методът на разклоненията и границите, бяха разгледани в *глава 1 — комбинаторни задачи* (*1.5.3.*).

Глава 7
Разделяй и владей

"Fools ignore complexity.
Pragmatists suffer it.
Some can avoid it.
Geniuses remove it."

~ Epigrams in Programming

Корените на идеята за разделяне на сложната задача на няколко по-прости, които се атакуват по-лесно поотделно и чието решаване позволява лесно конструиране на решение на изходната задача, се крият далеч в Древността. Тя достига своя връх по времето на Римската империя, която формулира и издига "Разделяй и владей" като основен принцип на външната си политика по отношение на съседните на Империята враждуващи помежду си племена. Всъщност римляните съвсем не са откриватели на принципа и той е в основата на експанзията на всяка голяма империя преди и след Римската.

Как стоят нещата в програмирането? Прилагането на метода протича на три стъпки. Първо се извършва разбиване на изходната задача на няколко подзадачи, най-често две (*разделяй*). Следва решаване на всяка от тях поотделно. На третата стъпка, въз основа на решенията на подзадачите, се конструира решение на изходната задача (*владей*).

Разбира се, принципът не е универсален и не при всяка задача прилагането му е възможно и подходящо. Причините, поради които методът не може да се приложи, могат да бъдат най-разнообразни. Така например, задачата може да не позволява подходящо разбиване на подзадачи, които могат да се решават поотделно. Възможно е решаването на отделните подзадачи да не облекчава съществено решаването на изходната задача или дори да няма нищо общо с него. Макар идеята на метода да е проста, прилагането му, както и определянето кога може да бъде приложен, е много или малко изкуство, което най-лесно се усвоява на основата на конкретни примери.

7.1. Намиране на k-ия по големина елемент

Един от първите алгоритми, с които се запознава начинаещият програмист, е този за намиране на максималния елемент на масив. Става въпрос за достатъчно прост алгоритъм, изискващ $n-1$ сравнения:

```
int findMax(int m[], unsigned n) /* Намира максималния елемент */
{ unsigned i;
  int max;
  for (max = m[0], i = 1; i < n; i++)
    if (m[i] > max) /**/
      max = m[i];
  return max;
}
```
🖫 maximum.c

Бихме ли могли да подобрим този резултат? Не е трудно да се убедим, че предложеният алгоритъм извършва теоретично минималния брой сравнения на елементи. Последното се вижда лесно, ако мислим за елементите като за участници в турнир с пряко елиминиране. Действително, за определяне на победителя е необходимо всеки един участник, с изключение на точно един (по-

бедителят), да загуби поне веднъж, за да може да отпадне. Оттук вече лесно се забелязва, че броят на необходимите срещи е $n-1$. Освен броят на *сравненията* важна характеристика на всеки подобен алгоритъм е броят на присвояванията на стойност. От кода на предложения програмен фрагмент непосредствено се вижда, че броят на присвояванията не надвишава броя на сравненията. Не е толкова очевидно обаче, какъв е *средният* брой присвоявания. Оставяме на читателя да покаже, че той е от порядъка на $\Theta(\log_2 n)$.

Нека си представим програма за визуализация на съвкупност от точки върху екрана на компютъра. Тъй като броят и позициите на точките могат да варират в широки граници, програмата ще трябва да мащабира по някакъв начин техните координати така, че да изглеждат добре върху екрана. Екранът има правоъгълна форма, поради което би било удобно точките да се "затворят" в подходящ правоъгълник, откъдето да се пресметне съответният мащаб. Ясно е, че този правоъгълник се определя от максималната и минималната x и y координати. Тук възниква задачата за *едновременно* намиране на минималния и максималния елемент.

Очевидно новата задача също има линейно решение. Действително, за независимото намиране на максималния, а след/преди това и на минималния елемент по горния алгоритъм биха били необходими $2n-1$ сравнения (При търсене на минималния елемент просто се обръща знакът за сравнение в реда, отбелязан с `/**/`.). Едно такова решение обаче е неоптимално, тъй като в процеса на търсенето на минимума по никакъв начин не се отчитат известните вече съотношения, получени при търсенето на максимума.

По-долу ще приведем друг алгоритъм, за който в най-лошия случай ще ни бъдат необходими не повече от $\lceil 3n/2 \rceil$ сравнения. Ще разглеждаме елементите по двойки: първо ще сравняваме двата елемента помежду им, а след това по-големия — с текущия максимум, а по-малкия — с текущия минимум, извършвайки три сравнения за всяка двойка елементи.

```
void findMinMax(int *min, int *max, const int m[],const unsigned n)
/* Намира едновременно максималния и минималния елементи */
{ unsigned i, n2;
  for (*min = *max = m[n2 = n/2], i = 0; i < n2; i++)
    if (m[i] > m[n-i-1]) {
      if (m[i] > *max)
        *max = m[i];
      if (m[n-i-1] < *min)
        *min = m[n-i-1];
    }
    else {
      if (m[n-i-1] > *max)
        *max = m[n-i-1];
      if (m[i] < *min)
        *min = m[i];
    }
}
```
🖫 maximum.c

Как стоят нещата, ако търсим *втория по големина* елемент? Очевидно бихме могли да се възползваме от същата стратегия, модифицирайки по подходящ начин функцията така, че да запазва не само една, а *двете* най-добри стойности. Идеята е проста: ще сравняваме поредния елемент с текущия втори най-голям и, *само* при откриване на по-голям от него елемент, ще извършваме допълнителна проверка дали той не е по-голям и от текущия *първи* най-голям:

```
/* Разменя стойностите */
void swap(int *el1, int *el2)
{ int tmp = *el1; *el1 = *el2; *el2 = tmp; }

int findSecondMax(int m[], unsigned n)
{ int x,y;
```

```
  unsigned i;
  x = m[0]; y = m[1];
  if (y > x)
    swap(&x,&y);
  for (i = 2; i < n; i++)
    if (m[i] > y)
      if ((y = m[i]) > x)
        swap(&x,&y);
  return y;
}
```

🖫 maximum.c

Да се опитаме да оценим сложността на алгоритъма. Не е трудно да се забележи, че *най-лошият* случай е обратно подреденият масив, при което на всяка от (n–2)-те итерации на цикъла се извършват по две сравнения. Прибавяйки сравнението преди цикъла, за общия брой необходими сравнения получаваме $2n$–3.

Нека сега разгледаме по-общата задача за намиране на *k-ия най-малък елемент* на дадено n-елементно множество. Разглежданата задача е еквивалентна на намирането на (n–k+1)-ия най-голям елемент. При това, ако разполагаме с алгоритъм, решаващ изходната задача, обръщайки сравненията $<$ (или $\leq$) и $>$ (или $\geq$), с минимални модификации бихме могли да намерим k-ия *най-голям* елемент.

Тривиално решение на проблема е пълното сортиране на множеството, в резултат на което k-ият елемент ще заеме k-та позиция. Основен недостатък на този подход е, че, независимо от стойността на k, изисква брой сравнения от порядъка на $n.\log_2 n$. Разбира се, бихме могли да намалим този брой до $\Theta(n + k.\log_2 n)$, сортирайки само първите k елемента с *пирамидално сортиране* (построяването на пирамидата изисква време от порядъка на $\Theta(n)$, а пирамидалното сортиране има сложност $\Theta(n.\log_2 n)$ както в средния, така и в *най-лошия* случай, *виж 3.1.9.*). Доразвивайки идеята, получаваме сложност $\Theta(\min(n + k.\log_2 n, n+(n-k+1).\log_2 n))$. Действително, ако $k > [n/2]$, можем да обърнем посоката на сортиране в намаляващ ред. Въпреки това, ако k е достатъчно близо до $n/2$, полученият алгоритъм очевидно ще бъде неефективен, тъй като ще води до пълно сортиране на k или съответно (n–k+1) елемента от масива, при което ще се получава голямо количество ненужна информация.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100
int m[MAX];

const unsigned n = 100; /* Брой елементи в масива */
const unsigned k = 10;  /* Пореден номер на търсения елемент */

void init(int m[], unsigned n) /* Запълва със случайни числа */
{ unsigned i;
  for (i = 0; i < n; i++)
    m[i] = rand() % (2*n + 1);
}

void siftMin(int l,int r) /* Отсява елем. от върха на пирамидата */
{ int i,j;
  int x;
  i = l; j = i + i + 1; x = m[i];
  while (j <= r) {
    if (j < r)
      if (m[j] > m[j+1])
        j++;
```

```
    if (x <= m[j])
      break;
    m[i] = m[j];
    i = j;
    j = j*2 + 1;
  }
  m[i] = x;
}
void siftMax(int l,int r) /* Отсява елем. от върха на пирамидата */
{ int i,j;
  int x;
  i = l; j = i + i + 1; x = m[i];
  while (j <= r) {
    if (j < r)
      if (m[j] < m[j+1])
        j++;
    if (x >= m[j])
      break;
    m[i] = m[j];
    i = j;
    j = j*2 + 1;
  }
  m[i] = x;
}

void heapFindK(unsigned k) /* Търсене на k-ия елемент с пирамида */
{ int l,r;
  char useMax;
  if (useMax = (k > n/2))
    k = n - k - 1;
  l = n/2; r = n - 1;
  /* Построяване на пирамидата */
  while (l-- > 0)
    if (useMax) siftMax(l,r); else siftMin(l,r);
  /* (k-1)-кратно премахване на минималния елемент */
  for (r = (int)n-1; r >= (int)(n-k); r--) {
    m[0] = m[r];
    if (useMax) siftMax(0,r); else siftMin(0,r);
  }
}

void print(int m[], unsigned n) /* Извежда масива на екрана */
{ unsigned i;
  for (i = 0; i < n; i++)
    printf("%8d", m[i]);
}

int main(void) {
  init(m,n);
  printf("Масивът преди търсенето:"); print(m,n);
  printf("\nТърсим k-ия елемент: k=%u", k);
  heapFindK(k);
  printf("\nk-ият елемент е: %d", m[0]);
  return 0;
}
```

🖫 heap.c

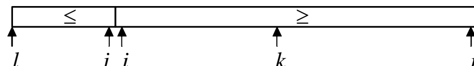
При някои по-силни ограничителни условия бихме могли лесно да получим дори линейни алгоритми. Така например, използвайки идеята на алгоритъма за *сортиране чрез броене* (*виж 3.2.2.*), можем да намерим броя на срещанията c_j за всеки ключ j измежду допустимите стойности *на ключовете*. Да предположим, че ключовете са цели числа в интервала $[l,r]$. Тогава можем да сумираме последователно c_l, c_{l+1}, c_{l+2}, ... до достигане на индекс j, за който сумата за пръв път става по-голяма или равна на k. Търсеният среден елемент ще има стойност j. Въпреки това, отново се натрупва голямо количество излишна информация, а и линейността на алгоритъма силно зависи от разпределението на стойностите на ключовете.

Макар това на пръв поглед да не се вижда, идеята на *бързото сортиране* (*виж 3.1.6.*) за разделяне на масива на дялове, дава ефективен алгоритъм за решаване на задачата. Става въпрос за алгоритъм от тип *разделяй и владей*, предложен от Хоор — автора на бързото сортиране (което също е основано на *разделяй и владей*). Извършва се разделяне на масива на две части с лява граница $l = 0$, дясна граница $r = n-1$ и $x = m[k]$ като разделящ елемент (предполагаме, че масивът се индексира от 0 до $n-1$). След разделянето по алгоритъма на бързото сортиране за индексите i и j са изпълнени неравенствата:

$$m[h] \le x, \text{ за } h < i$$
$$m[h] \ge x, \text{ за } h > j$$
$$i > j$$

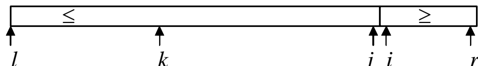
Възможни са три случая:

- $i < k$. Тогава x дели масива в отношение *по-малко* от търсеното. Разделянето следва да продължи, като се положи $l = i$. (*виж фигура 7.1а.*)



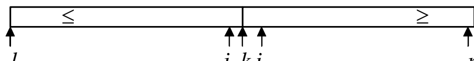
Фигура 7.1а. Масивът в случай на $i < k$.

- $j > k$. Тогава x дели масива в отношение *по-голямо* от търсеното. Разделянето следва да продължи, като се положи $r = j$. (*виж фигура 7.1б.*)



Фигура 7.1б. Масивът в случай на $j > k$.

- $j < k < i$. Тогава x разделя масива в исканите пропорции и следователно е търсеният елемент. Наистина, всички елементи вляво от него са по-малки или равни на него, а тези отдясно — по-големи или равни. Процесът на разделяне се прекратява. (*виж фигура 7.1в.*)



Фигура 7.1в. Масивът в случай на $j < k < i$.

Оставяме на читателя да покаже, че не са възможни други случаи на взаимно положение на i, j и k. При достигане до случай 3 k-ият елемент ще се намира на k-та позиция в масива.

```
void find(int m[],unsigned n,unsigned k) /* намира k-ия елемент */
{ int i,j,l,r;
  int x;
  l = 0; r = n - 1;
  while (l < r) {
    x = m[k]; i = l; j = r;
    for(;;) {
```

```
        while (x > m[i]) i++;
        while (x < m[j]) j--;
        if (i > j)
          break;
        swap(m + i, m + j);
        i++;
        j--;
      }
      if (j < (int)k)
        l = i;
      if ((int)k < i)
        r = j;
    }
  }
```

🖫 mid_elem.c

Описаният алгоритъм изисква брой сравнения от порядъка на $2n$, в случай че на всяка стъпка дялът, съдържащ k-ия елемент, се разполовява. Това му качество го определя като по-ефективен в сравнение с предложения по-горе, основан на пирамидалното сортиране, което изискваше брой сравнения от порядъка на $\Theta(\min(n + k.\log_2 n,\ n + (n-k+1).\log_2 n))$, както в най-лошия, така и в най-добрия случай. За съжаление, в най-лошия случай, когато на всяка стъпка областта намалява само с 1, сложността му ще бъде от порядъка на $\Theta(n^2)$. Така, наследявайки основния недостатък на бързото сортиране на Хоор, макар и много добър в *общия*, описаният алгоритъм се оказа изключително неефективен в *най-лошия* случай.

Ще предложим друг, този път рекурсивен алгоритъм, основаващ се на бързото сортиране на Хоор със сходни характеристики. Ще извършваме разделяне относно някакъв елемент `x`, без значение кой (в програмата по-долу е избран `m[r]`), и нека при разделянето той попада на позиция `mid`, считано от началото на масива, или на позиция `p`, считано от началото `l` на разглеждания дял. Ако $k \leq$ `p`, то следва да търсим k-ия елемент в левия дял `(l,mid)`. В противен случай следва да търсим (k–`p`)-ия елемент в десния дял `(mid+1,r)`. Процесът продължава рекурсивно до достигане на дял, съдържащ единствен елемент (а именно търсения):

```
unsigned partition(unsigned l,unsigned r) /* Разделяне по Ломуто */
{ int i;
  unsigned j;
  int x;
  i = l - 1; x = m[r];
  for (j = l; j <= r; j++)
    if (m[j] <= x) {
      i++;
      swap(m+i,m+j);
    }
  if (i == (int)r) /* Всички са <= x. Стесняване на областта с 1. */
    i--;
  return i;
}

unsigned find(int l, int r, unsigned k) /* Търсене по Хоор */
{ unsigned mid, p;
  if (l == r)
    return l;
  mid = partition(l,r);
  p = mid - l + 1;
  return k < p ? find(l,mid,k) : find(mid+1,r,k-p);
}
```

🖫 mid_el2.c

Описаният алгоритъм има същото поведение, като предходния — много добър в общия случай, и изключително неефективен — в най-лошия. Лошото поведение и на двата алгоритъма се дължи на начина, по който избираме елемента *x*. Добре би било *x* да се избира така, че да разделя разглежданата област на две почти равни части, т. е. да бъде максимално близо до средния елемент.

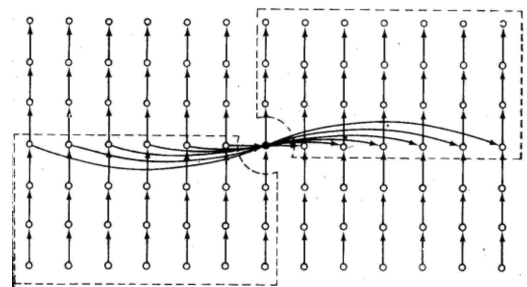
Как да изберем такова *x*? Ще разгледаме алгоритъма, предложен от Блум, Флойд, Прат, Ривест и Тарджан. Нека разсъждаваме индуктивно. Да предположим, че ни е известен *линеен* алгоритъм за намиране на *k*-ия елемент с $28n$ сравнения. При $n \leq 55$ не е трудно да се посочи един такъв конкретен алгоритъм. Например, *методът на мехурчето* (*виж 3.1.4.*), изисква $n(n-1)/2$ на брой сравнения за пълно сортиране на масива, а при $n \leq 55$ е в сила неравенството $28n \geq n(n-1)/2$. Нека предположим, че $28t$ сравнения са ни достатъчни за $t < n$. Разделяме масива на $\lceil n/7 \rceil$ подмасива, всеки от които съдържа по 7 елемента, като евентуално при нужда сме допълнили последния подмасив с $-\infty$ (*разделяй*), след което сортираме *изцяло* всеки от подмасивите. Ако използваме метода на мехурчето, ще ни бъдат необходими не повече от $7(7-1)/2 = 21$ сравнения. Така, общият брой необходими сравнения не надвишава $21(n/7) = 3n$. Следва рекурсивно прилагане на алгоритъма за избор към $n/7$-те сортирани таблици за намиране на медианата на медианите (*владей*). Това изисква $28(n/7) = 4n$ сравнения.

Алгоритъм за намиране на медиана на медианите:

1. Разделяме елементите на групи по 7. Нека ги означим с S_i, за $i = 1, 2, ..., \lceil n/7 \rceil$. Последната група евентуално съдържа по-малко от 7 елемента.
2. Сортираме напълно всяка група S_i и така намираме медианата ѝ m_i.
3. Намираме медианата M на медианите m_i, чрез рекурсивно обръщение към същия алгоритъм.

Ще опишем горния алгоритъм малко по-внимателно с псевдокод. Ще считаме, че разполагаме с функция `partition()`, която по зададен елемент `x` разделя масива `L[]` на три области: лява `L1[]`, в която елементите са по-малки от `x`, средна `L2[]`, в която са равни на `x`, и дясна `L3[]`, в която са по-големи от `x`.

```
unsigned select(int L[],     /* Масив */
                unsigned k) /* Пореден номер на търсения елемент */
{ n = length(L); /* Брой елементи в L[] */
  if (n <= 7) {
    bubbleSort(L,n);
    return L[k];
  }

  /* Разделяме L на n/7 на брой групи S[i] с по 7 елемента. */
  split(L, S = {S[i] | i = 1,2,...,n/7});

  for (i = 0; i < n/7; i++)
    x[i] = select(S[i],3); /* средният елемент е третият */
  M = select({x[i]| i=1,2,...,n/7}, n/14);

  partition(L,L1,L2,L3);

  if (k <= length(L1)) then
    return select(L1,k);
  else if (k > length(L1) + length(L2)) then
        return select(L3,k-length(L1)-length(L2));
      else
   return M;
}
```



Фигура 7.1г. Избор на медиана.

Избраната медиана на медианите x е достатъчно близка до средния елемент, тъй като съществуват поне $2n/7$ по-малки от x, както и поне $2n/7$ по-големи от x елемента (*виж фигура 7.1г.*). Дотук използвахме $3n$ сравнения за сортиране на таблиците, още $4n$ — за намиране на медианата, n сравнения — за разделяне на таблицата по метода на Хоор, и не повече от $28(5n/7)$ — сравнения за рекурсивното прилагане на алгоритъма (при всяко рекурсивно обръщение областта намалява с поне $2n/7$ елемента). В крайна сметка се оказва, че са ни били необходими не повече от $28n$ сравнения. Т. е. нашият алгоритъм е линеен. Разбира се, 28 е достатъчно сериозна константа, далеч от 2, в оптималния случай на предходните алгоритми. Нещо повече, при $n \leq 16384$ предложеният алгоритъм извършва повече сравнения от *пълното пирамидално сортиране* на масива. Въпреки това, при $n > 16384$ алгоритъмът се държи достатъчно добре както в общия, така и в най-лошия случай. Освен това оценката $28n$ лесно би могла да се понижи допълнително, например до $15n$. Блум, Флойд, Прат, Ривест и Тарджан успяват да я понижат до $391/72 \approx 5{,}4306.n$. Шьонхагер, Патерсън и Пипенгер постигат $3n$, а Дор и Цвик успяват да подобрят този резултат до $2{,}95.n$ сравнения! [*Dor,Zwick-1996*]

Задачи за упражнение:

1. Да се реализира програма за намиране на k-ия по големина елемент, основана на сортиране чрез броене.
2. Да се докаже, че при алгоритъма за намиране на k-ия по големина елемент, основан на разделяне, подобно на това при бързото сортиране, не са възможни други взаимни положения на променливите i, j и k, освен показаните на фигури *7.1а.*, *7.1б.* и *7.1в.*
3. Да се сравнят двата предложени варианта на алгоритъма за намиране на k-ия по големина елемент, основани на разделяне, подобно на това при бързото сортиране.
4. Следва ли да се очаква подобрение при разделяне на елементите в групи по 5 при намиране на медиана на медианите? А по 3? Какъв е очакваният брой сравнения във всеки отделен случай? Да се определи оптималният брой елементи в група.
5. Да се реализира алгоритъмът за намиране на медиана на медианите.

7.2. Мажорант

Дефиниция 7.1. Нека е дадено n-елементно *мултимножество* (т. е. множество, в което се допуска повторение на елементи). Ще казваме, че даден елемент на множеството е негов *мажорант*, ако се среща *строго повече* от $n/2$ пъти.

Така мултимножеството {2, 3, 3, 1, 3} има за мажорант 3, а мултимножеството {1, 1, 2, 3} няма мажорант.

Очевидно, съгласно горната дефиниция, мултимножеството не би могло да има два мажоранта. По-долу ще разгледаме няколко различни начина за намиране на мажорант, като мултимножеството ще представяме с едномерен масив. Освен ако изрично не сме казали друго, няма

да предполагаме някакъв конкретен тип на елементите, нито съществуване на наредба (частична или пълна) в този тип, още по-малко пък наредба на елементите в масива. В програмите по-долу за целта ще използваме макроса `CDataType`, който ще дефинираме като `char`. Независимостта на алгоритъма от типа `char` може да се провери, като се промени макросът, например на `int` или на `float`.

Първият алгоритъм, който може би хрумва на читателя, е да се премине през елементите на масива и за всеки от тях да се провери дали се среща повече от $n/2$ пъти. В предложената реализация мажорантът се намира от функцията `findMajority()`, която получава като параметър масива, размера му и адрес, в който да запише мажоранта, в случай, че в масива има такъв (функция със страничен ефект). Функцията връща стойност 1, ако е намерен мажорант, и 0 — в противен случай. Намирането на броя на срещанията на даден елемент в масива е отделено във функцията `count()`, която ще използваме по-долу и при други алгоритми. В случай на преждевременно намиране на мажоранта по-нататъшната работа се прекратява незабавно.

```
#include <stdio.h>

#define CDataType char

unsigned count(CDataType m[], unsigned size, CDataType candidate)
{ unsigned cnt, i;
  for (i = cnt = 0; i < size; i++)
    if (m[i] == candidate)
      cnt++;
  return cnt;
}

char findMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned i, size2 = size / 2;
  for (i = 0; i < size; i++)
    if (count(m, size, m[i]) > size2) {
      *majority = m[i];
      return 1;
    }

  return 0;
}

int main(void) {
  CDataType majority;
  if (findMajority("AAACCBBCCCBCC", 13, &majority))
    printf("Мажорант: %c\n", majority);
  else
    printf("Няма мажорант.\n");
  return 0;
}
```
🖫 major1.c

Очевидно сложността на предложения алгоритъм е $\Theta(n^2)$, тъй като всеки от n-те елемента се тества за мажорант, а тестът изисква още едно преминаване. Бихме могли "да се изхитрим" и да намалим проверките, без по същество да променяме алгоритъма. За целта е необходимо да забележим, че при всяко следващо извикване функцията `count()` върши все повече излишна работа, преминавайки през целия масив. Не е трудно да се съобрази, че броенето би могло спокойно да пропусне елементите, вляво от текущия. Наистина, така бихме могли да изпуснем срещания на тествания елемент. В такъв случай обаче той не би бил мажорант, защото на някоя предишна стъпка, би трябвало вече да сме го разгледали и да сме го отхвърлили. Същите разсъждения ни позволяват да приключим проверката на $(n/2)$-ия елемент на масива. Наистина,

той е последният, вдясно от който има достатъчно други, потенциално равни на него, които биха могли да го направят мажорант.

```
char findMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned i, j, cnt, size2 = size / 2;
  for (i = 0; i <= size / 2; i++) {
    for (cnt = 0, j = i; j < size; j++)
      if (m[i] == m[j]) cnt++;
    if (cnt > size2) {
      *majority = m[i];
      return 1;
    }
  }
  return 0;
}
```

🖫 major2.c

За съжаление, въпреки направените подобрения, както в средния, така и в най-лошия случай, при който първите $n/2+1$ елемента са различни помежду си, сложността си остава $\Theta(n^2)$. Бихме могли да опитаме да ограничим още повече разглежданите в процеса на броене елементи. Идеята е да прекратяваме броенето в момента, в който стане ясно, че броят на оставащите елементи е недостатъчен, за да направи разглеждания елемент мажорант, дори ако всички те съвпадат с него.

```
char findMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned i, j, cnt, size2 = size / 2;
  for (i = 0; i <= size / 2; i++) {
    for (cnt = 0, j = i; j < size; j++)
      if (cnt + size - j <= size2)
        break;
      else if (m[i] == m[j])
        cnt++;
    if (cnt > size2) {
      *majority = m[i];
      return 1;
    }
  }
  return 0;
}
```

🖫 major3.c

Ползата от това "подобрение" в общия случай обаче е доста съмнителна, тъй като то изисква допълнително: едно събиране, едно изваждане и едно сравнение на всяка стъпка от броенето. Т. е. в най-лошия случай това решение е по-лошо от предходното. Макар че асимптотичната му сложност отново е $\Theta(n^2)$.

Няма да се задълбочаваме повече в тази посока, тъй като съществуват по-ефективни алгоритми. Да видим какво става, ако отслабим строгите ограничения, наложени в началото, и допуснем тип на елементите, за който е дефинирана пълна наредба. В такъв случай бихме могли да сортираме елементите на масива и да проверим дали средният му елемент е мажорант. Получаваме следния алгоритъм (Доказателството, че ако сортираме масива, средният му елемент се заема от мажоранта, ако има такъв, оставяме на читателя като леко упражнение.):

```
char findMajority(CDataType m[],unsigned size,CDataType *majority)
{ heapSort(m, size);  /* или mergeSort(m,size); */
  if (count(m, size, m[size / 2]) > size2) {
    *majority = m[size / 2];
```

```
    return 1;
  }
  return 0;
}
```
🖫 major4.c

Тъй като проверката дали елементът е мажорант е линейна, сложността на алгоритъма се доминира от сложността на сортирането. Ако използваме подходящ алгоритъм като *пирамидално сортиране* (*виж 3.1.9.*) или *сортиране чрез сливане* (*виж 7.4.* по-долу), които имат гарантирана сложност $\Theta(n.\log_2 n)$ в най-лошия случай, получаваме обща сложност на алгоритъма $\Theta(n.\log_2 n)$. Ще отбележим, че бързото сортиране на Хоор не би ни свършило работа. Макар в средния случай да има сложност $\Theta(n.\log_2 n)$ и да бие пирамидалното от 2 до 3 пъти, в най-лошия сложността му е $\Theta(n^2)$.

За да намерим средния (и изобщо k-ия по големина) елемент на масив обаче, не е необходимо да го сортираме. Да си припомним, че алгоритъмът на Блум, Флойд, Прат, Ривест и Тарджан намира средния елемент с линейна сложност $\Theta(n)$, извършвайки $5{,}43n-163$ сравнения, $n > 32$. (*виж 7.1.*)

```
char findMajority(CDataType m[],unsigned size,CDataType *majority)
{ CDataType med;
  med = findMedian(m, size);
  if (count(m, size, med) > size2) {
    *majority = med;
    return 1;
  }
  return 0;
}
```
🖫 major5.c

Така, получихме линеен алгоритъм за намиране на мажорант, но при допълнителното предположение, че между елементите съществува линейна наредба. Да видим какво ще получим, ако се откажем от наредбата, но поискаме броят на възможните различни стойности на елементите на масива да бъде предварително известен и да бъде достатъчно малко число. В такъв случай отново можем да получим линеен алгоритъм. По същество се използва идеята на алгоритъма за *сортиране чрез броене* (*виж 3.2.2.*). С едно преминаване през елементите на масива се намира броят на срещанията за всички кандидати за мажорант. Следва преминаване през възможните стойности на кандидатите и проверка за всеки един дали се среща строго повече от $n/2$ пъти. Това става за време $\Theta(k)$, независещо от броя на елементите в масива, а само от броя на различните стойности k, които могат да приемат. Така за общата сложност на алгоритъма получаваме $\Theta(n+k)$. При достатъчно малки стойности на k (например $k < n$) имаме $\Theta(n)$.

```
#define MAX_NUM 127
CDataType cnt[MAX_NUM + 1];

char findMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned i, j, size2 = size / 2;
  /* Инициализация */
  for (i = 0; i < MAX_NUM; i++)
    cnt[i] = 0;
  /* Броене */
  for (j = 0; j < size; j++)
    cnt[m[j]]++;
  /* Проверка за мажорант */
  for (i = 0; i < MAX_NUM; i++)
```

```
      if (cnt[i] > size2) {
        *majority = i;
        return 1;
      }
    return 0;
}
```

🖫 major6.c

Да се върнем обаче към изходното условие, при което не можем да разчитаме нито на наредба, нито на предварително известен малък брой възможни стойности на елементите, и да опитаме да приложим стратегията *разделяй и владей*. Да разделим масива на две почти равни части (в случай на нечетен брой елементи, едната част ще има един елемент повече). Основната идея е, че ако *x* е мажорант на изходния масив, той ще бъде мажорант на поне една от двете части. Оттук лесно получаваме съответен рекурсивен алгоритъм. Разделяме масива на две почти равни части и намираме мажоранта на всяка от тях. Налице са три случая:

1) И двата подмасива нямат мажорант. Тогава и изходният масив няма да има.
2) Единият подмасив има мажорант *x*, а другият — няма. Проверява се дали *x* е мажорант на изходния масив.
3) И двата подмасива имат мажоранти *x* и *y*, евентуално различни. Проверяват се както *x*, така и *y*. (Ако *x* се окаже мажорант, проверката за *y* може да се пропусне.)

За намирането на мажоранта на всеки от подмасивите се прилага рекурсивно същият алгоритъм, като процесът приключва при достигане на масив с един елемент, в който случай този елемент се явява мажорант. Процесът би могъл да приключи и по-рано (например при два елемента) от съображения за ефективност, но няма да се спираме на разсъждения в тази посока. Ще отбележим, че поддържането на лява и дясна граница изисква малка промяна в параметрите и на двете функции, като началното извикване става с лява граница на разглеждания масив 0, и дясна — `size-1`.

```
unsigned count(CDataType m[], unsigned left,
               unsigned right, CDataType candidate)
{ unsigned cnt;
  for (cnt = 0; left <= right; left++)
    if (m[left] == candidate)
      cnt++;
  return cnt;
}

char findMajority(CDataType m[], unsigned left,
                  unsigned right, CDataType *majority)
{ unsigned mid;
  if (left == right) {
    *majority = m[left];
    return 1;
  }
  mid = (left + right) / 2;
  if (findMajority(m, left, mid, majority))
    if (count(m, left, right, *majority) > (right - left + 1) / 2)
      return 1;
  if (findMajority(m, mid + 1, right, majority))
    if (count(m, left, right, *majority) > (right - left + 1) / 2)
      return 1;
  return 0;
}
```

🖫 major7.c

Какво постигнахме? В общия случай на всяко ниво на рекурсията имаме по две рекурсивни обръщения с масиви, с наполовина по-малко елементи. Това е частта *разделяй*. Обединението на резултатите на двете рекурсивни обръщения (частта *владей*) изисква 0, n или $2n$ сравнения, съответно за случаите 1), 2) и 3). В граничния случай на един елемент имаме константна сложност без сравнения. Ако означим броя на извършваните сравнения за n-елементен масив с $T(n)$, в най-лошия случай получаваме следните рекурентни зависимости:

$$T(1) = 0$$
$$T(n) = T(n/2) + 2n$$

С помощта на *основната теорема* (*виж 1.4.10*) получаваме, че $T(n) \in \Theta(n.\log_2 n)$. Дали не бихме могли да получим още по-добри резултати? Оказва се, че лесно можем да получим линейна сложност на алгоритъма, забелязвайки, че ако масивът има мажорант, то са в сила следните твърдения:

Твърдение 1. Ако масивът има мажорант и премахнем два *различни* елемента, мажорантът на новия масив ще съвпада с този на изходния.

Твърдение 2. Ако всички елементи в масива се срещат по двойки, то, ако запазим единия представител и изхвърлим другия за всяка двойка, мажорантът на новия масив ще съвпада с мажоранта на изходния.

Доказателството на горните твърдения е леко и го оставяме на читателя. Ще отбележим, че при премахване на два различни елемента от масив *без* мажорант можем да получим масив с мажорант. Така например, ако от масива {1,1,2,3,4} (който не съдържа мажорант) премахнем 3 и 4 (които са различни), получаваме масива {1,1,2}, където 1 е мажорант.

Нека отново опитаме да опростим условието, като поискаме: 1) масивът гарантирано да съдържа мажорант, и 2) да се позволява промяна на масива така, че след приключване работата на програмата той (почти сигурно) да се различава от изходния. Нека за момент предположим, че масивът има *четен* брой елементи. Да опитаме отново да приложим римската стратегия, разделяйки масива на две равни части: част 1 – елементите от нечетни позиции и част 2 – тези от четни. Вместо да търсим мажорант във всяка от тях обаче, този път ще сравняваме елементите по двойки: единият елемент е от едната част, а другият — от другата. Разглеждаме една конкретна двойка (последователни елементи). Ако двата елемента са различни, ги изключваме от понататъшни разглеждания, а ако са равни — запазваме единия. Прилагайки това за всички двойки, получаваме нов масив с най-много $n/2$ елемента. За него се прилага същата стратегия, като процесът продължава до получаване на масив с един елемент: мажорантът (напомняме, че предполагаме, че със сигурност има мажорант).

Проблем възниква при нечетен брой елементи, при което един от елементите няма да може да бъде включен в двойка. Тъй като този елемент може да бъде критичен и да определя мажоранта, не бихме искали да го изхвърлим. (Например: АА ВВ **А**). От друга страна, не бихме искали и винаги да го пазим, защото така можем да разрушим мажоранта (Например: АА **В**).

Едно възможно решение е да го запазим, давайки му по-малко тегло и да го вземем предвид, само ако в новия масив без него не може да се определи мажорантът. Получаваме алгоритъм с линейна сложност, тъй като на всяка стъпка броят на елементите намаляват поне наполовина. Ако означим броя на извършваните сравнения за n-елементен масив с $T(n)$, в най-лошия случай получаваме следните рекурентни зависимости:

$$T(1) = 0$$
$$T(n) = T(n/2) + n/2$$

С помощта на *основната теорема* (*виж 1.4.10.*) получаваме, че $T(n) \in \Theta(n)$. Предложената програмна реализация не използва допълнителен масив, а на всяка стъпка последователно копира по един представител на елементите от двойките с еднакви елементи в началото на същия масив. Лесно се вижда, че това не е опасно, тъй като копирането става “зад гърба ни”. За съжаление, така се разрушава оригиналният масив, което ни лишава и от възможността за извършване на проверка дали намереният елемент наистина е мажорант.

```
void findMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned i, curCnt;
  char part = 0;
  do {
    for (curCnt = 0, i = 1; i < size; i += 2)
      if (m[i - 1] == m[i])
        m[curCnt++] = m[i];
    if (i == size) {
      m[curCnt++] = m[i - 1];
      part = 1;
    }
    else if (part)
      m[curCnt] = m[size - 2];
    else if (m[size - 2] == m[size - 1])
      m[curCnt] = m[size - 2];
    else
      curCnt--;
    size = curCnt;
  } while (size > 1);
  *majority = m[0];
}
```

🖫 major8.c

Друга възможност е да запазим последния елемент на масива при нечетен брой елементи в специална отделна променлива. Ако на някоя следваща стъпка масивът отново се окаже с нечетен брой елементи, ще я унищожим, записвайки отгоре ѝ новия последен елемент на масива. В един момент масивът остава с нула елемента, а кандидатът за мажорант ще се намира в специалната променлива. Дали това работи? Преминавайки към следващата стъпка, ще загубим мажоранта, само ако в новия масив точно половината елементи са равни на мажоранта, при което нашият запазен елемент по необходимост ще трябва да бъде равен на мажоранта (знаем, че в масива със сигурност има мажорант). По-нататък на всяка следваща стъпка поне половината елементи ще бъдат равни на мажоранта, така че, ако в един момент се окажем с нечетен брой елементи, то оставащият елемент отново ще трябва да бъде равен на мажоранта, което означава, че спокойно можем да премахнем старата стойност. След изчерпване елементите на масива, мажорантът ще се намира в специалната променлива. Броят на елементите отново намалява поне наполовина на всяка стъпка, т. е. имаме линейна сложност.

```
void findMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned i, curCnt;
  do {
    for (curCnt = 0, i = 1; i < size; i += 2)
      if (m[i - 1] == m[i])
        m[curCnt++] = m[i];
    if (size & 1)
      *majority = m[size - 1];
    size = curCnt;
  } while (size > 0);
}
```

🖫 major9.c

Дали не бихме могли да се освободим от нуждата от допълнителен елемент? Нека помислим кога трябва да вземем предвид последния елемент. Не е трудно да се съобрази, че той няма да е нужен, ако броят на елементите в новия масив е нечетен, защото в такъв случай не може да има два кандидата за мажорант. Ако обаче този брой е четен, то е възможно да не може да се определи мажорант (т. е. елементите да съдържат две стойности, разпределени по равно), поради което той именно ще определя мажоранта. И така: в случай на нечетен брой елементи последният

елемент се прибавя към края на новия масив, само ако иначе броят на елементите му би станал четен. Така, веднъж достигнали до масив с нечетен брой елементи, ще съхраняваме нечетен брой, докато в него не остане единствен елемент: мажорантът. Описаният алгоритъм е проста вариация на предходния и сложността му отново е линейна: При нужда последният елемент на масива поема функциите на специалната променлива от предишния алгоритъм.

```
void findMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned i, curCnt;
  do {
    for (curCnt = 0, i = 1; i < size; i += 2)
      if (m[i - 1] == m[i])
        m[curCnt++] = m[i];
    if (!(curCnt & 1))
      m[curCnt++] = m[size - 1];
    size = curCnt;
  } while (size > 1);
  *majority = m[0];
}
```

🖫 major10.c

Друг възможен (но излишно по-сложен) вариант на решаване на проблема с нечетния брой елементи е с всеки елемент да асоциираме брояч, показващ колко елемента представлява той в действителност. В началото броячите на всички елементи се инициализират с 1. При сравняване на два елемента `m[i-1]` и `m[i]`, с броячи съответно `cnt[i-1]` и `cnt[i]`, имаме две възможности:

1. `m[i-1] == m[i]`
   Запазваме едно копие на елемента, с брояч `cnt[i-1] + cnt[i]`.
2. `m[i-1] != m[i]`
   2.1. `cnt[i-1] == cnt[i]`
   Премахваме и двата елемента.
   2.2. `cnt[i-1] < cnt[i]`
   Запазваме `m[i]` с брояч `cnt[i] - cnt[i-1]`.
   2.3. `cnt[i-1] > cnt[i]`
   Запазваме `m[i-1]` с брояч `cnt[i-1] - cnt[i]`.

Сега можем *винаги* да запазваме последния елемент в случай на нечетен брой елементи. В случай на четен брой, разбира се, няма да го пазим, защото той ще бъде включен в двойка. Лесно се вижда, че полученият алгоритъм отново е линеен и на всяка стъпка мажорантът се запазва. Следва примерна реализация:

```
void findMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned i, curCnt;
  unsigned *cnt = (unsigned *) malloc(size * sizeof(*cnt));
  for (i = 0; i < size; i++) cnt[i] = 1;
  do {
    for (curCnt = 0, i = 1; i < size; i += 2) {
      if (m[i - 1] == m[i]) {
        cnt[curCnt] = cnt[i - 1] + cnt[i];
        m[curCnt++] = m[i];
      }
      else if (cnt[i] > cnt[i - 1]) {
        cnt[curCnt] = cnt[i] - cnt[i - 1];
        m[curCnt++] = m[i];
      }
      else if (cnt[i] < cnt[i - 1]) {
        cnt[curCnt] = cnt[i - 1] - cnt[i];
```

```
        m[curCnt++] = m[i - 1];
      }
    }
    if (size & 1) {
      cnt[curCnt] = cnt[i - 1];
      m[curCnt++] = m[i - 1];
    }
    size = curCnt;
  } while (size > 1);
  free(cnt);
  *majority = m[0];
}
```

🖫 major11.c

Алтернативен вариант за решаване на задачата за време $\Theta(n)$ е да се използва стек (*виж 2.1.*). Този път ще се освободим от изискването масивът задължително да съдържа мажорант. Започваме с празен стек, в който в началото постъпва първият елемент на масива. След това на всяка стъпка от масива се извлича следващият елемент и се сравнява с елемента на върха на стека. Ако елементите са еднакви, новият елемент постъпва в стека. В противен случай, се изключва елементът на върха на стека, което практически означава отхвърляне и на двата елемента. В случай на празен стек следващият елемент постъпва в него. Процесът продължава до изчерпване елементите на масива. Ако има мажорант, той е на върха на стека. Наистина, елементите се унищожават едновременно по двойки, при това *само ако са различни*. Така, мажорантът не може да бъде унищожен, защото няма достатъчно немажорантни елементи в масива. Наистина, по определение мажорантът е равен на *строго повече* от половината от елементите.

Възможно е обаче масивът да не съдържа мажорант и все пак на върха на стека да има елемент, например:

“**A**BC”(стек: ∅)
“**B**C” (стек: A)
“**C**” (стек: ∅)
“” (стек: C)

Това налага едно допълнително преминаване през масива, за да се провери дали елементът наистина е мажорант. Забележете, че в този случай това е възможно, защото оригиналният масив *не се разрушава*, но това става за сметка на допълнителна памет за стек. Ще отбележим, че стекът може да съдържа и повече от един елемент. В този случай всички елементи в него ще имат една и съща стойност, съгласно разсъжденията по-горе. Следва примерна реализация на алгоритъма:

```
/* Променливи, функции и дефиниции за работа със стек */
#define STACK_SIZE  100
CDataType stack[STACK_SIZE];
unsigned stIndex;
void stackInit(void) { stIndex = 0; }
void stackPush(CDataType elem) { stack[stIndex++] = elem; }
CDataType stackPop(void) { return stack[--stIndex]; }
CDataType stackTop(void) { return stack[stIndex - 1]; }
char stackIsEmpty(void) { return 0 == stIndex; }

char findMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned i, cnt;
  stackInit();
  for (stackPush(m[0]), i = 1; i < size; i++) {
    if (stackIsEmpty())
      stackPush(m[i]);
    else if (stackTop() == m[i])
```

```
      stackPush(m[i]);
    else
      stackPop();
  }
  if (stackIsEmpty()) return 0;
  for (*majority = stackPop(), i = cnt = 0; i < size; i++)
    if (m[i] == *majority)
      cnt++;
  return(cnt > size / 2);
}
```

🖫 major12.c

Дали имаме наистина нужда от стек? Не е трудно да се види, че в предложения алгоритъм стекът винаги съдържа елементи с *еднаква* стойност. Наистина, ако текущият елемент е различен от този на върха на стека (предполагаме, че стекът не е празен), той не може да се добави, а вместо това взаимно се унищожава с елемента на върха му. Но тогава можем да се освободим от стека и да го заместим с двойка променливи от вида: (текущ елемент, брояч). В началото кандидатът има неопределена стойност, а броячът има стойност 0. Алгоритъмът преминава последователно през елементите на масива и на всяка стъпка извършва следните проверки и свързани с тях действия:

1) Ако броячът е 0, кандидат става текущият елемент, а броячът става 1.
2) Ако броячът е различен от 0:
   2.1) Ако кандидатът съвпада с текущия елемент, броячът се увеличава с 1.
   2.2) Ако са различни, броячът се намалява с 1.

Процесът продължава до изчерпване елементите на масива. Ако накрая броячът се окаже 0, масивът със сигурност не съдържа мажорант. Ако обаче е различен от 0, оттук не следва, че кандидатът непременно е мажорант и следва да се направи съответна проверка. От друга страна, ако масивът има мажорант, то това задължително е кандидатът.

Следва пример за прилагане на алгоритъма върху конкретна редица, съдържаща мажорант.

```
A A A C C B B C C C B C C
 ^
?:0

A A A C C B B C C C B C C
 ^
A:1

A A A C C B B C C C B C C
   ^
  A:2

A A A C C B B C C C B C C
     ^
    A:3

A A A C C B B C C C B C C
       ^
      A:2

A A A C C B B C C C B C C
         ^
        A:1

A A A C C B B C C C B C C
```

```
           ^
          ?:0

A A A C C B B C C C B C C
             ^
            B:1

A A A C C B B C C C B C C
               ^
              ?:0

A A A C C B B C C C B C C
                 ^
                C:1

A A A C C B B C C C B C C
                   ^
                  C:2

A A A C C B B C C C B C C
                     ^
                    C:1

A A A C C B B C C C B C C
                       ^
                      C:2

A A A C C B B C C C B C C
                         ^
                        C:3
```

В нашия случай мажорантът е *C* и алгоритъмът правилно го посочва като кандидат. Забележете, че ако заменим например първото *C* с *A*, алгоритъмът отново ще посочи *C* като *кандидат* за мажорант, но в този случай мажорант просто няма. Т. е. необходима е стандартната крайна *проверка*, за да се уверим, че посоченият кандидат действително е мажорант. Следва примерна реализация:

```
char FindMajority(CDataType m[],unsigned size,CDataType *majority)
{ unsigned cnt, i;
  for (i = cnt = 0; i < size; i++) {
    if (0 == cnt) {
      *majority = m[i];
      cnt = 1;
    }
    else if (m[i] == *majority)
      cnt++;
    else
      cnt--;
  }
  if (cnt > 0) {
    for (i = cnt = 0; i < size; i++)
      if (m[i] == *majority)
        cnt++;
    return(cnt > size / 2);
  }
  return 0;
}
```

🖫 major13.c

Забележете, че този алгоритъм, за разлика от някои предишни, използва *константна* допълнителна памет (за две променливи) и *не променя* масива. Въпреки, че може да се разглежда като вариант на предишния, ще се опитаме да го обясним и обосновем от още една гледна точка.

Да предположим, че алгоритъмът е започнал работа и е достигнал до някаква позиция. Можем да считаме, че е разделил прегледаните елементи на масива на две области. Едната област съдържа елементи, групирани по двойки, така че членовете на всяка двойка са различни. Останалите елементи са равни помежду си и равни на кандидата за мажорант. Лесно се вижда, че ако масивът съдържа мажорант, негов представител непременно ще попадне във втората група. Наистина, противното не е възможно, тъй като няма достатъчно елементи, различни от мажоранта, които да влязат в двойка с него, така че всички срещания на мажоранта да попаднат в първата група. Ако масивът няма мажорант обаче, е възможно във втората група все пак да попадне елемент-кандидат (например: *ABC*, *C* ще бъде кандидат). Ето защо и тук се налага познатото ни допълнително преминаване през масива. Ако втората група се окаже празна, то в масива гарантирано няма мажорант.

Ще отбележим, че макар да бяха разгледани редица линейни алгоритми, не трябва да се забравят скритите константи. Така например, последният алгоритъм е по-добър от алгоритъма на Блум, Флойд, Прат, Ривест и Тарджан за намиране на медиана, който работеше при по-строги ограничения. Наистина, тук броят на сравненията е $2n$, докато при алгоритъма на Блум, Флойд, Прат, Ривест и Тарджан е $5{,}43n-163$, $n > 32$. Към това трябва да добавим още $n-1$ сравнения за проверка дали медианата е мажорант, при което получаваме $6{,}43n-164$, $n > 32$. (*виж 7.1.*)

Задачи за упражнение:

1. Да се докаже, че, ако се сортира масив, съдържащ мажорант, средният му елемент се заема от мажоранта.
2. Да се докаже, че ако x е мажорант на даден масив и масивът се раздели на две почти равни части, то x е мажорант на поне едната от тях.
3. Да се докаже, че ако масивът `m[]` има мажорант, то, ако елементите `m[i]` и `m[j]` са различни и бъдат премахнати, мажорантът на новия масив ще съвпада с мажоранта на стария.
4. Да се докаже, че ако даден масив има мажорант, то, ако всички елементи в масива се срещат по двойки и ако се запази единият представител, като се изхвърли другият за всяка двойка, мажорантът на новия масив ще съвпада с мажоранта на стария.

7.3. Сливане на сортирани масиви

Нека са дадени две сортирани последователности *A* и *B*, подобласти на масива `a[]`. Задачата е да се обединят по подходящ начин така, че получената последователност *C* също да бъде сортирана. На пръв поглед изглежда, че това би могло да се извърши на място, т. е. направо в масива `a[]`. Оставяме на читателя сам да се убеди, че това не е толкова лесно. В действителност, такъв метод съществува, но той е достатъчно сложен и изисква особено внимание и допълнителни усилия. Вместо това, тук ще предложим класическия алгоритъм с допълнителен масив `b[]`, в който се копират елементите на изходния `a[]`, след което двата дяла се сливат в `a[]`.

Да разгледаме първо по-общия случай на сливане на два сортирани масива `a[]` и `b[]` (с n и m елемента съответно) в нов сортиран масив `c[]`. Това може да стане така: Инициализираме 3 индекса: по един за всеки масив. Сравняваме първите елементи на двата масива `a[]` и `b[]` и прехвърляме по-малкия от тях в `c[]`, след което увеличаваме индексите на `c[]` и на масива, в който се е съдържал по-малкият елемент. На следващата стъпка отново сравняваме съответните елементи на двата масива и копираме по-малкия от тях в `c[]`, като увеличаваме съответните индекси. Процесът приключва при изчерпване на единия от масивите, след което останалите елементи на неизчерпания масив следва да се изкопират в `c[]`. Следва примерна реализация на описания алгоритъм (последните два *while*-цикъла могат да бъдат заменени с `memcpy()`) :

```
i = j = k = 0;
while (i < n && j < m)
  c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
if (i == n)
  while(j < m)
    c[k++] = b[j++];
else
  while(i < n)
    c[k++] = a[i++];
```

Бихме могли да опитаме да оптимизираме горния програмен фрагмент, съкращавайки броя на извършваните сравнения в първия *while*-цикъл. Това може да се постигне с помощта на *ограничител* (допълнителен стоп-елемент ∞) в изходните масиви `a[]` и `b[]`, в резултат на което не може да се премине през краищата им. Недостатък на реализацията са излишните сравнения, извършвани при изчерпване елементите на `a[]` или `b[]`:

```
i = j = k = 0; a[m] = b[n] = INFINITY; mn = m + n;
while (k < mn)
  c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
```

Новият алгоритъм изисква непременно *точно* $n + m$ сравнения, докато първият изисква по-малко, ако се използва функцията `memcpy()`. Всъщност, функцията `memcpy()` вътрешно също е реализирана чрез цикъл, свързан със съответни сравнения, но реализацията му е достатъчно ефективна, така че може би следният вариант е за предпочитане (тук обаче има едно допълнително сравнение и някои събирания):

```
i = j = k = 0;
while (i < n && j < m)
  c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
if (i == n)
  memcpy(c+k,b+j,m-j);
else
  memcpy(c+k,a+i,n-i);
```

Описаният алгоритъм се реализира най-добре при използване на динамични структури от данни, тъй като прехвърлянето на елемент от A или B в C става с просто пренасочване на указатели, т. е. не се изисква допълнителна памет за C, а при изчерпване елементите на едната последователност елементите на другата се залепят по тривиален начин към края на C. Все пак нещата не са толкова розови, тъй като указателите изискват допълнителна памет, а и в резултат на работата на алгоритъма A и B се разрушават.

Горният алгоритъм би могъл да се обобщи за произволен брой последователности, като всяка от тях поддържа свой собствен индекс-указател с функция, аналогична на тази на `i` и `j` от горната реализация. Предложената по-долу програма е минимална по отношение броя на извършваните сравнения. Последователностите се пазят статично (в масиви), а самите масиви са организирани в динамичен линеен списък. На всяка стъпка се сравняват текущите елементи на масивите и се определя минималният елемент. Следва извеждане на екрана и изключване. Ако някой масив се окаже празен, той се изключва незабавно от списъка на масивите и не участва в следващите сравнения. Този път ще работим с елементи от тип `struct CElem` и ще сравняваме ключовете `key`.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX    12
#define ARRAYS 6
```

```
struct CElem {
  int key;
  /* Някакви данни */
};
struct CList {
  unsigned len, point;
  struct CElem data[MAX];
  struct CList *next;
};

struct CList *init(unsigned mod)  /* Запълва със случайни цели числа */
{ struct CList *head, *p;
  unsigned i, j;
  srand(time(NULL));
  for (head = NULL, i = 0; i < ARRAYS; i++) {
    p = (struct CList *) malloc(sizeof(struct CList));
    p->len = MAX;
    p->point = 0;
    p->data[0].key = (rand() % mod);
    for (j = 1; j < MAX; j++)
      /* Генерира сортирана последователност */
      p->data[j].key = p->data[j-1].key + (rand() % mod);
    p->next = head;
    head = p;
  }

  return head;
}

void merge(struct CList *head)
{ struct CList *p, *q, *pMin;
  struct CElem k1, k2;
  int i;
  printf("\n");
  p = (struct CList *) malloc(sizeof(struct CList));
  p->next = head;
  head = p;
  for (i = 0; i < MAX*ARRAYS; i++) {
    p = head; pMin = head;
    while (NULL != p->next) {
      k1 = p->next->data[p->next->point];
      k2 = pMin->next->data[pMin->next->point];
      if (k1.key < k2.key)
        pMin = p;
      p = p->next;
    }
    printf("%8d", pMin->next->data[pMin->next->point].key);
    if (pMin->next->len-1 == pMin->next->point) {
      q = pMin->next;
      pMin->next = pMin->next->next;
      free(q);
    }
    else
      pMin->next->point++;
  }
}

void print(struct CList *head)
{ unsigned i;
```

```
  for (; NULL != head; head = head->next) {
    for (i = 0; i < MAX; i++)
      printf("%6d", head->data[i].key);
    printf("\n");
  }
  printf("\n");
}

int main(void)
{ struct CList *head;
  head = init(500);
  printf("\nМасивите преди сортирането:\n");
  print(head);
  printf("Резултатът от сливането:");
  merge(head);
  return 0;
}
```

mergearr.c

a_1,a_2,a_3,a_4,a_5 | a_6,a_7,a_8,a_9,a_{10} | $a_{11},a_{12},a_{13},a_{14},a_{15}$ | $a_{16},a_{17},a_{18},a_{19},a_{20}$

b_1 b_2 b_3

Очевидно, функцията `merge()` извършва $n + m$ на брой сравнения (m и n са дължините на двете последователности). При близки стойности на m и n това е добре. Ако $m = n$ в общия случай ще ни бъдат необходими $2n–1$ сравнения. Какво става обаче при $m = 1$? В този случай, използвайки *двоично търсене* (*виж 4.3.*) за откриване на позицията, където да се вмъкне B_1, бихме могли да решим проблема значително по-ефективно — достатъчни ще ни бъдат $\log_2 n$ сравнения, вместо $\Theta(n+1)$.

Задачата, която ще си поставим, е да извлечем максимална полза от двата алгоритъма: простото сливане и двоичното търсене, съставяйки алгоритъм, работещ не по-лошо от тях, който при $m = 1$ се държи като двоично търсене, а при $m = n$ — като просто сливане.

Фигура 7.3. Двоично сливане на сортирани масиви.

Този път C ще нараства отзад-напред — като стек. Нека за определеност предположим, че $n > m$ и да разделим A на $m + 1$ групи (подпоследователности) с приблизително равен брой елементи. Сравняваме B_m с последния елемент A_k от предпоследната група в A. Ако B_m се окаже по-малко от A_k, то бихме могли да прехвърлим A_k в C, заедно с всички елементи, вдясно от A_k, т. е. цялата последна група. В противен случай ($B_m \geq A_k$) търсим съответната позиция на вмъкване на B_m чрез двоично търсене, след което прехвърляме в C B_m и стоящите вдясно от B_m елементи на A. Тъй като двоичното търсене работи най-добре при масиви с размер, кратен на 2, то би било добре последната група да съдържа $2^{\lfloor \log_2 n/m \rfloor}$ елемента, вместо n/m. Процесът продължава до изчерпване елементите на едната последователност, при което елементите на другата се прехвърлят в C. (*виж фигура 7.3.*)

Използването на двоично търсене твърдо предполага директен достъп до всеки от елементите на масива и неизбежно води до статична реализация, както е и в приложената функцията `binaryMerge()`, извършваща двоично сливане на масиви съгласно описания алгоритъм. Тя приема като параметри двата сортирани масива `a[]` и `b[]`, както и броя елементи във всеки от тях — `m` и `n` съответно, и връща масив `c[]`, съдържащ слятата последователност:

```
int binarySearch(struct CElem m[], int left,
                 int right, struct CElem elem)
{ int middle;
  do {
    middle = (left + right) / 2;
    if (m[middle].key < elem.key)
```

```
        left = middle + 1;
      else
        right = middle - 1;
    } while (left <= right);
    return right;
}

void binaryMerge(struct CElem a[], struct CElem b[],
                 struct CElem c[], int n, int m)
{ int t, t2, cind, k, j;
  cind = n + m;
  while (n > 0 && m > 0) {
    if (m <= n) {
      t = (int) (log(n / m) / log(2));
      t2 = 1 << t;  /* T2 <-- 2^T */
      if (b[m - 1].key < a[n - t2].key) {
        /* Прехвърляме a[n-t2-1],...,a[n] в изходната последователност */
        cind -= t2;
        n -= t2;
        for (j = 0; j < t2; j++)
          c[cind + j] = a[n + j];
      }
      else {
        k = binarySearch(a, n - t2, n - 1, b[m - 1]);
        for (j = 0; j < n - k - 1; j++)
          c[cind - n + k + j + 1] = a[k + j + 1];
        cind -= n - k - 1;
        n = k + 1;
        c[--cind] = b[--m];
      }
    }
    else {
      t = (int) (log(m / n) / log(2));
      t2 = 1 << t;  /* T2 <-- 2^T */
      if (a[n - 1].key < b[m - t2].key) {
        /* Прехвърляме b[m-t2-1],...,b[m] в изходната последователност */
        cind -= t2;
        m -= t2;
        for (j = 0; j < t2; j++)
          c[cind + j] = b[m + j];
      }
      else {
        k = binarySearch(b, m - t2, m - 1, a[n - 1]);
        for (j = 0; j < m - k - 1; j++)
          c[cind - m + k + j + 1] = b[k + j + 1];
        cind -= m - k - 1;
        m = k + 1;
        c[--cind] = a[--n];
      }
    }
  }
  if (n == 0)
    for (j = 0; j < m; j++)
      c[j] = b[j];
  else
    for (j = 0; j < n; j++)
      c[j] = a[j];
}
```

binmerge.c

Задача за упражнение:

Да се предложи и реализира алгоритъм за сливане на две сортирани последователности от масив на място, т.е. без използване на допълнителен масив.

7.4. Сортиране чрез сливане

Ще разработим алгоритъм за сортиране на масив, а след това и на свързан списък, основан на принципа *разделяй и владей*. Първо ще разгледаме случая на масив. Идеята е да го разделим на две части, които да сортираме поотделно (*разделяй*), след което да ги обединим в общ сортиран масив (*владей*). Сливането на двете части може да стане с единствено преминаване през двата сортирани подмасива. Разделянето ще правим по средата на масива, като за всяка част ще прилагаме рекурсивно същия алгоритъм. Процесът приключва при достигане на подмасив, съдържащ 0 или 1 елемента.

Очевидно тук двоичното сливане (*виж 7.3.*) не би могло да ни бъде от полза, тъй като сливаме дялове с равен брой елементи. Разполагаме с единствен масив `a[]`, който е разделен на две части, които искаме да слеем. В предложената програма се използва *модификация на метода на ограничителя*. (*виж 7.3.*) Алгоритъмът протича на две стъпки. На първата стъпка в помощния масив `b[]` се копира съдържанието на `a[]`, като елементите на втората част се копират в намаляващ ред. След това започват последователни сравнения от двата противоположни края на масива. Забележете, че най-големият елемент ще се окаже в средата на масива (той е бил последният в единия от двата сортирани дяла) и ще служи като ограничител и за двата индекса, след като единият от дяловете се изчерпи. Така получаваме следната програма:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX  100
int a[MAX],          /* Основен масив - за сортиране */
    b[MAX];          /* Помощен масив */

const unsigned n = 100; /* Брой елементи за сортиране */

/* Генерира примерно множество */
void generate(void)
{ unsigned i;
  for (i = 0; i < n; i++)
    a[i] = rand() % (2*n + 1);
}

/* Извежда списъка на екрана */
void printList(void)
{ unsigned i;
  for (i = 0; i < n; i++)
    printf("%4d", a[i]);
}

/* Сортиране */
void mergeSort(unsigned left, unsigned right)
{ unsigned i, j, k, mid;
  if (right <= left) return;  /* Проверка дали има какво да се сортира */
  mid = (right + left) / 2;
  mergeSort(left, mid);       /* Сортиране на левия дял */
  mergeSort(mid + 1, right);  /* Сортиране на десния дял */

  /* Копиране на елементите на a[] в помощния масив b[] */
```

```
  for (i = mid + 1; i > left; i--)
    b[i - 1] = a[i - 1];           /* Права посока */
  for (j = mid; j < right; j++)
    b[right + mid - j] = a[j + 1]; /* Обратна посока */

  /* Сливане на двата масива в a[] */
  for (k = left; k <= right; k++)
    a[k] = (b[i] < b[j]) ? b[i++] : b[j--];
}

int main(void) {
  generate();
  printf("Преди сортирането:\n");
  printList();
  mergeSort(0, n-1);
  printf("След сортирането:\n");
  printList();
  return 0;
}
```

🖫 merge_a.c

Основен недостатък на горната имплементация е копирането на данни на всяка стъпка. Това би могло да се избегне при поддържане на няколко сливащи функции. Идеята е да се редува сливането на `a[]` в `b[]` със сливане на `b[]` в `a[]`, както и на сливане в *нарастващ* със сливане в *намаляващ* ред. Така например, за да получим в `a[]` възходящо сортирана последователност, трябва да слеем две последователности от `b[]`: първата, сортирана в нарастващ (да я означим с `b1[]`), а втората — в намаляващ ред (`b2[]`). За да се получи `b1[]`, трябва да слеем две съответни последователности от първата част на `a[]` (сортирани в нарастващ и намаляващ ред съответно). За получаване на `b2[]`, пък трябва да се слеят други две части на `a[]` (отново сортирани в намаляващ и нарастващ ред съответно) и т.н. Нужни са 4 различни сливащи функции за четирите възможни комбинации: сливане от `a[]/b[]` в `b[]/a[]` в нарастващ/намаляващ ред.

Друг (по-неефективен) начин за избягване на копирането на данни е използването на свързан списък. Сливането на два сортирани свързани списъка става почти по същия начин, както и сливането на два сортирани масива. Следва примерна функция, реализираща сливането.

```
struct list { /* Тип свързан списък */
  int value;
  struct list *next;
} *empty; /* Празен елемент */
const unsigned long n = 100;

struct list *merge(struct list *a, struct list *b)
{ struct list *head, *tail;

  /* Предполага се, че и двата списъка съдържат поне по един елемент */
  tail = head = empty;
  for (;;) {
    if (a->value < b->value) {
      tail->next = a;
      a = a->next;
      tail = tail->next;
      if (NULL == a) {
        tail->next = b;
        break;
      }
    }
    else {
```

```
        tail->next = b;
        b = b->next;
        tail = tail->next;
        if (NULL == b) {
          tail->next = a;
          break;
        }
      }
    }
  return head->next;
}
```

🖫 merge_ll.c

Малко по-неприятно стоят нещата с разделянето на списъка на два подсписъка преди рекурсивните обръщения. Липсата на пряк достъп до елементите на свързан списък изисква явно преминаване през половината от елементите му. За целта е добре броят на елементите в дяла да се подава като параметър, както е направено в следващата функция.

```
struct list *mergeSort(struct list *c, unsigned long n)
{ struct list *a, *b;
  unsigned long i, n2;

  /* Ако списъкът съдържа само един елемент: не се прави нищо */
  if (n < 2)
    return c;
  /* Разделяне на списъка на две части */
  for (a = c, n2 = n / 2, i = 2; i <= n2; i++)
    c = c->next;
  b = c->next;
  c->next = NULL;

  /* Сортиране поотделно на двете части, последвано от сливане */
  return merge(mergeSort(a, n2), mergeSort(b, n - n2));
}
```

🖫 merge_ll.c

Използването на метода на ограничителя позволява оптимизиране на процеса на сливане. За целта ще се откажем от стандартния индикатор за край на списък — `NULL`. Вместо това ще използваме специален елемент `z`, чийто указател `next` ще сочи към `z`, а стойността му `value` ще бъде максималната допустима за съответния тип (в случая `INFINITY`). Всички списъци ще имат за последен елемент `z`.

По-горе споменахме, че е желателно функцията `mergeSort()` да получава като параметър броя на елементите в списъка. В общия случай той може да не бъде известен предварително. Едно възможно решение тогава е броят да бъде предварително намерен с единствено преминаване през списъка, след което да бъде подаден като параметър, както по-горе. Друг възможен подход е да се организира цикъл, в който участват два указателя: единият се движи със стъпка 1, а вторият — със стъпка 2. Когато вторият указател достигне края на списъка, първият ще сочи средата му. Забележете, че ако се използва за ограничител `NULL`, този алгоритъм няма да сработи правилно, тъй като в случая на четен брой елементи ще бъде пропуснат краят на списъка. В нашия случай обаче полето `next` на `z` сочи отново `z`, поради което няма опасност. Така получаваме следния вариант на програмата:

```
#include <stdio.h>
#include <stdlib.h>

#define INFINITY (int)((1 << (sizeof(int)*8 - 1)) - 1)
```

```
const unsigned long n = 100;

struct list { /* Тип свързан списък */
  int value;
  struct list *next;
} *z; /* Празен елемент */

/* Генерира примерно множество */
struct list *generate(unsigned long n)
{ struct list *p, *q;
  unsigned long i;
  for (p = z, i = 0; i < n; i++) {
    q = (struct list *) malloc(sizeof(struct list));
    q->value = rand() % (2*n + 1);
    q->next = p;
    p = q;
  }
  return p;
}

void printList(struct list *p) /* Извежда списъка на екрана */
{ for (; p != z; p = p->next)
    printf("%4d", p->value);
}

struct list *merge(struct list *a, struct list *b)
{ struct list *c;
  c = z;

  /* Предполага се, че и двата списъка съдържат поне по един елемент */
  do {
    if (a->value < b->value) {
      c->next = a;
        c = a;
        a = a->next;
    }
    else {
      c->next = b;
        c = b;
        b = b->next;
    }
  } while (c != z);
  c = z->next;
  z->next = z;
  return c;
}
struct list *mergeSort(struct list *c)
{ struct list *a, *b;

  /* Ако списъкът съдържа само един елемент: не се прави нищо */
  if (c->next == z)
        return c;

  /* Списъкът се разделя на две части */
  for (a = c, b = c->next->next->next; b != z; c = c->next)
    b = b->next->next;
  b = c->next;
  c->next = z;
```

```
  /* Сортиране поотделно на двете части, последвано от сливане */
  return merge(mergeSort(a), mergeSort(b));
}

int main(void) {
  struct list *l;

  /* Инициализация на z */
  z = (struct list *) malloc(sizeof(struct list));
  z->value = INFINITY;
  z->next = z;

  l = generate(n);
  printf("Преди сортирането:\n");
  printList(l);
  l = mergeSort(l); /* Предполага се, че списъкът съдържа поне 1 елем. */
  printf("След сортирането:\n");
  printList(l);
  return 0;
}
```

🖫 merge_l2.c

Добре известно е, че всяка рекурсивна програма има итеративен еквивалент. За някои програми преобразуването от рекурсивен в итеративен вариант става лесно и просто. За други това може да бъде доста сложна задача. Стандартен начин за преобразуване на рекурсивна програма в итеративна е използване на стек. За щастие `mergeSort()` позволява сравнително проста итеративна реализация, без необходимост от стек.

Предложената итеративна реализация не е съвсем еквивалентна на рекурсивната и извършва разделянията по различен начин. Процесът се контролира от външния цикъл *for*. На първата стъпка се формират наредени двойки елементи, на втората — четворки, на третата — осморки и т.н. На всяка следваща стъпка се сливат двойки последователни наредени списъци, при което се получават нови сортирани списъци с дължина, два пъти по-голяма от тази на предходната стъпка. Всички подсписъци ще поддържаме свързани в общ списък. Сливането се извършва във вътрешния цикъл. Формира се списък `todo` от необработени до момента елементи. Вътрешният цикъл *while* последователно търси двойки списъци, получени на предходната стъпка, след което ги слива. Полученият в резултат списък се добавя към края на списъка с обработените елементи. Изпълнението на вътрешния цикъл продължава до прехвърляне на всички елементи в този списък. [*Sedgewick-1992*]

```
struct list *mergeSort(struct list *c)
{ unsigned long i, n, n2;
  struct list *a, *b, *head, *todo, *t;
  head = (struct list *) malloc(sizeof(struct list));
  head->next = c;
  a = z;
  for (n = 1; a != head->next; n <<= 1) {
    todo = head->next;
    c = head;
    while (todo != z) {
      t = todo;
      /* Отделяне на a[] */
      for (a = t, i = 1; i < n; i++)
        t = t->next;
      /* Отделяне на b[] */
      b = t->next; t->next = z;
      for (t = b, i = 1; i < n; i++)
```

```
        t = t->next;
      /* Сливане на a[] и b[] */
      todo = t->next; t->next = z;
      c->next = merge(a, b);
      /* Пропускане на слетия масив */
      for (n2 = n + n, i = 1; i <= n2; i++)
        c = c->next;
    }
  }
  return head->next;
}
```

🖫 merge_l3.c

Каква е ефективността на сортирането чрез сливане? Оказва се, че това е един от най-ефективните известни алгоритми за сортиране с гарантирана средна времева сложност $\Theta(n.\log_2 n)$. Това е теоретически най-добрата алгоритмична сложност, която може да бъде постигната от *универсален* алгоритъм за сортиране (*виж 3.1.10.*). Както вече видяхме в *параграф 3.2.*, при допълнителни предположения за (двоичното) представяне на данните, могат да бъдат получени линейни алгоритми за сортиране. Те обаче не са универсални и не могат да се използват за сортиране на числа с плаваща запетая, например. Това го нарежда непосредствено до *пирамидалното* (*виж 3.1.9.*) и *бързото* (*виж 3.1.6.*) сортирания, които имат същата *средна* времева сложност. Нещо повече, подобно на пирамидалното сортиране, сортирането чрез сливане има същата сложност и в *най-лошия* случай (при това с по-малка скрита константа от пирамидалното). Същевременно *бързото* сортиране в най-лошия случай има сложност $\Theta(n^2)$, което го нарежда сред най-неефективните алгоритми като *метода на мехурчето*, *виж 3.1.4.* (Последният, както вече сме споменавали, стои в основата на *бързото сортиране*.) Когато искаме гарантирана сложност $\Theta(n.\log_2 n)$, включително в най-лошия случай, можем да използваме *пирамидално сортиране* или *сортиране чрез сливане*.

Основен недостатък на сортирането чрез сливане е това, че изисква *допълнителна памет*, пропорционална на n: за допълнителен масив или за указатели съответно. Емпиричните тестове показват, че в средния случай бързото сортиране бие пирамидалното от два до три пъти (*виж* [*Уирт-1980*]). Скоростта на сортирането чрез сливане заема междинно положение между двете, поради което е сериозен кандидат, когато скоростта в най-лошия случай е критична, а същевременно има възможност за заделяне на допълнителна памет, пропорционална на n.

Друго основно предимство на сортирането чрез сливане е, че достъпът до елементите се извършва *строго последователно*, поради което е подходящо за сортиране на списъци, последователни файлове и др. Не на последно място следва да се отбележи, че сортирането чрез сливане е *стабилно*, т. е. запазва относителната наредба на елементите с равни ключове. Това е важно свойство и не е характерно за всеки алгоритъм за сортиране. *Бързото сортиране* например е нестабилно и превръщането му в стабилно изисква доста усилия. Друго интересно свойство на сортирането чрез сливане във варианта му, когато при сливането не се проверява дали някой от двата списъка не е изчерпан, е, че сложността му практически не зависи от предварителната наредба на елементите на множеството. Във варианта с проверката наредбата оказва незначително влияние.

Подобно на повечето съвременни методи за сортиране, сортирането чрез сливане се държи лошо при малък брой елементи. Както и при *бързото сортиране*, добри резултати се получават при комбинирането му с други методи. Стандартен подход е, когато броят на елементите на разглежданото множество падне под 15-20 (този брой е примерен) да се използва *сортиране чрез вмъкване* (*виж 3.1.2.*).

Задачи за упражнение:

1. Да се сравни скоростта на работа на предложените реализации за сортиране чрез сливане за: 100; 1000; 10000; 100000 елемента.

2. Да се напише програма за сортиране чрез сливане, при която множеството се разбива на:
   а) 3 подмножества
   б) k подмножества, $k > 3$
3. Да се пресметне сложността на различните варианти на сортиране чрез сливане:
   а) в най-добрия случай
   б) в най-лошия случай

Да се намерят най-добрата и най-лошата входна последователност.

4. Да се реализира итеративен вариант на сортирането чрез сливане, използващ масиви вместо свързани списъци.
5. Да се установи опитно броят елементи, под който при сортиране чрез сливане следва да се използва сортиране чрез вмъкване.
6. Да се сравни опитно скоростта на работа на сортирането чрез сливане, в комбинация с:
   а) сортиране чрез вмъкване
   б) метод на мехурчето
   в) пряка селекция
7. Да се реализира вариант на сортирането чрез сливане, който се възползва от евентуална естествена подредба на елементите в изходното множество (*виж 7.3.*).

7.5. Бързо повдигане в степен

В *1.1.1.* разгледахме прост итеративен начин за намиране на x^n (x — реално число, n — естествено). Съществува по-бърз алгоритъм (извършващ степенуването с по-малко на брой умножения), основаващ се на следната формула [*Швертнер-1995*]:

$$x^n = \begin{cases} x^{\frac{n}{2}}.x^{\frac{n}{2}}, \text{за } n \text{ четно} \\ x^{n-1}.x, \text{за } n \text{ нечетно} \end{cases}$$

Не е трудно да се види, че горната формула се основава на *разделяй и владей*, макар и в малко странна форма: на всяка стъпка да имаме по две подзадачи, но в действителност се налага решаването само на една нетривиална (за частта *разделяй*). В първия случай, защото двете подзадачи съвпадат, а във втория — защото едната е тривиална. Частта *владей* обаче е класическа и въз основа на решенията на двете подзадачи се получава решение на изходната задача.

```
#include <stdio.h>

const double base = 3.14;
const unsigned d = 11;

double power(double x, unsigned n)
{ if (0 == n) return 1;
   else
     if (n % 2) return x * power(x, n - 1);
       else return power(x * x, n / 2);
}
int main(void) {
  printf("%lf^%u = %lf\n", base, d, power(base, d));
  return 0;
}
```

🖫 powrec.c

Ще отбележим, че горната формула *не* дава минималния брой умножения за извършване на повдигането в степен. Така например, за n = 15 програмите по-горе ще извършат последователно повдигането така:

1) x^1
2) $x^2 = x^1.x^1$
3) $x^3 = x^2.x^1$
4) $x^6 = x^3.x^3$
5) $x^7 = x^6.x^1$
6) $x^{14} = x^7.x^7$
7) $x^{15} = x^{14}.x^1$

т. е. с общо 7 умножения. Съществуват и други решения, например:

1) x^1
2) $x^2 = x^1.x^1$
3) $x^3 = x^2.\ x^1$
4) $x^4 = x^3.\ x^1$
5) $x^7 = x^4.\ x^3$
6) $x^8 = x^7.\ x^1$
7) $x^{15} = x^8.\ x^7$

Съществуват обаче и по-къси последователности от само 6 умножения, например:

1) x^1
2) $x^2 = x^1.x^1$
3) $x^3 = x^2.x^1$
4) $x^5 = x^3.x^2$
5) $x^{10} = x^5.x^5$
6) $x^{15} = x^{10}.x^5$

както и:

1) x^1;
2) $x^2 = x^1.x^1$;
3) $x^4 = x^2.x^2$;
4) $x^5 = x^4.x^1$;
5) $x^{10} = x^5.x^5$;
6) $x^{15} = x^{10}.x^5$

Ако разгледаме само редиците от степенните показатели:

1, 2, 3, 4, 7, 8, 15
1, 2, 3, 5, 10, 15
1, 2, 4, 5, 10, 15

се вижда, че това са редици в които първият член е 1, а всеки следващ е сума от два предхождащи го. Такава редица се нарича *адитивна*. Единственият известен начин за намиране на адитивна редица с минимална дължина, за съжаление, е пълното изчерпване, което не представлява интерес за настоящия параграф. [*Наков-1998*]

Задача за упражнение:

Да се предложи евристичен алгоритъм (*виж глава 9*), който винаги намира адитивна редица, не по-дълга от тази, давана от формулата по-горе.

7.6. Алгоритъм на Щрасен за бързо умножение на матрици

Умножението на матрици е нова възможност за демонстрация на възможностите на стария римски принцип. Ще припомним, че класическият алгоритъм за умножение на матрици $A = (a_{ij})_{m\times n}$ и $B = (b_{ij})_{n\times r}$ изхожда директно от дефиницията и се дава от формулата:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

В случай на квадратни матрици с размер $n\times n$ времето за пресмятане на резултатната матрица C очевидно е $\Theta(n^3)$: имаме n^2 елемента и пресмятането на всеки от тях отнема време $\Theta(n)$. Тук, разбира се, предполагаме, че събирането и умножението имат константа времева сложност $\Theta(1)$, независеща от n.

Този алгоритъм е толкова прост и очевиден, че дълго време никой не е предполагал, че може да съществува друг и не е опитвал да го търси. В края на 60-те години обаче Щрасен предлага интересно подобрение на алгоритъма, намаляващо сложността до $\Theta(n^{\log 7})$. Откритието му е изненадващо за научния свят и води до отдаване на заслуженото на метода *разделяй и владей*, който до този момент е недооценяван като ефективна програмистка техника. Огромният ефект на откритието на Щрасен се дължи на факта, че умножението на матрици, като фундаментална операция от алгебрата, е включена в много числени алгоритми. Всяко подобрение тук води до автоматично намаляване изчислителната сложност на редица други алгоритми.

Да разгледаме матрици A и B с размер 2×2:

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

Използвайки стандартния алгоритъм, получаваме:

$$c_{ij}= a_{i1}b_{1j} + a_{i2}b_{2j},\ i = 1{,}2 \text{ и } j = 1{,}2$$

Вижда се, че са ни необходими 8 умножения и 4 събирания. Щрасен забелязва, че броят на умноженията би могъл да бъде намален до 7 по следния начин (с P_i означаваме помощните променливи):

$$\begin{aligned}
P_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
P_2 &= (a_{21} + a_{22})b_{11} \\
P_3 &= a_{11}(b_{12} - b_{22}) \\
P_4 &= a_{22}(b_{21} - b_{11}) \\
P_5 &= (a_{11} + a_{12})b_{22} \\
P_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
P_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
c_{11} &= P_1 + P_4 - P_5 + P_7 \\
c_{12} &= P_3 + P_5 \\
c_{21} &= P_2 + P_4 \\
c_{22} &= P_1 + P_3 - P_2 + P_6
\end{aligned}$$

Безплатен обяд, разбира се, няма: броят на събиранията (в това число изважданията) нараства до 18. Това поставя под съмнение предимството на метода на Щрасен в конкретния случай, вземайки предвид факта, че на по-старите процесори умножението “струваше” колкото две (по-рядко три) събирания/изваждания. Повечето съвременни процесори обаче извършват събиране и изваждане с еднаква скорост. Т. е. изложеният метод, като че ли не води до подобрение, поне не директно. Щрасен обаче забелязва, че алгоритъмът би могъл да се приложи *рекурсивно*. Преди да обсъдим как би могло да стане това, ще покажем, че горната

последователност от операции не е единствената. Друга възможна реализация на същата идея изглежда така (*виж* [*Brassard*, *Bratley-1987*]):

$$
\begin{aligned}
&P_1 = (a_{21} + a_{22} - a_{11})\,(b_{22} - b_{12} + b_{11})\\
&P_2 = a_{11}b_{11}\\
&P_3 = a_{12}b_{21}\\
&P_4 = (a_{11} - a_{21})(b_{22} - b_{12})\\
&P_5 = (a_{21} + a_{22})(b_{12} - b_{11})\\
&P_6 = (a_{12} - a_{21} + a_{11} - a_{22})b_{22}\\
&P_7 = a_{22}(b_{11} + b_{22} - b_{12} - b_{21})\\
&c_{11} = P_2 + P_3\\
&c_{12} = P_1 + P_2 + P_5 + P_6\\
&c_{21} = P_1 + P_2 + P_4 - P_7\\
&c_{22} = P_1 + P_2 + P_4 + P_5
\end{aligned}
$$

Има, разбира се, и други начини да се постигне същият ефект. Няма да ги разглеждаме всичките, но все пак ще се изкушим да приведем още един ([*Aho, Hopcroft, Ullman-1987*]):

$$
\begin{aligned}
&P_1 = (a_{12} - a_{22})(b_{21} + b_{22})\\
&P_2 = (a_{11} + a_{22})(b_{11} + b_{22})\\
&P_3 = (a_{11} - a_{21})(b_{11} + b_{12})\\
&P_4 = (a_{11} + a_{12})b_{22}\\
&P_5 = a_{11}(b_{12} - b_{22})\\
&P_6 = a_{22}(b_{21} - b_{11})\\
&P_7 = (a_{21} + a_{22})b_{11}\\
&c_{11} = P_1 + P_2 - P_4 + P_6\\
&c_{12} = P_4 + P_5\\
&c_{21} = P_6 + P_7\\
&c_{22} = P_2 - P_3 + P_5 - P_7
\end{aligned}
$$

Ако заместим елементите a_{ij}, b_{ij} и c_{ij} с матрици с размер $n \times n$, получаваме алгоритъм, който умее да умножава матрици с размер $2n$, използвайки 7, вместо 8 умножения на *матрици* с размер $n \times n$. Умножението на две матрици обаче определено е по-тежка операция от събирането. Наистина, ако при използване на класическите алгоритми от дефинициите за събирането са ни необходими n^2 елементарни събирания, то за умножението са ни нужни n^3 елементарни събирания плюс n^3 елементарни умножения. Т. е. спестяването на едно умножение има сериозен ефект.

Защо обаче да спираме дотук? Ако n е четно число, бихме могли да приложим същата схема рекурсивно за матриците с размер $(n/2) \times (n/2)$. В случай, че n е степен на 2, процесът може да продължава до достигане на матрица с размер 2×2. Така получаваме рекурсивен алгоритъм, чието дърво на всяка стъпка има разклоненост 7 и се обхожда в дълбочина. Сложността $T(n)$ на описания алгоритъм в случай на квадратни матрици с размер n, степен на 2, се задава с рекурентната зависимост:

$$T(n) = 7T(n/2) + 18(n/2)^2,\ n > 2$$

Оттук по *основната теорема* (*виж 1.4.10.*) получаваме $T(n) \in \Theta(n^{\log 7})$. Но $\log 7 \approx 2{,}81 < 3$, т.е. алгоритъмът на Щрасен има по-добра времева сложност от класическия.

Какво да правим, ако имаме квадратни матрици, но n не е степен на 2? Стандартно решение на проблема е матриците да се допълнят с нулеви редове и колони до най-малкото n, което е степен на 2. Тази техника е известна като *статично допълване*. Това може да доведе най-много до удвояване размера на матрицата. А извършването на още една итерация, съгласно горната рекурентна зависимост, може да увеличи броя на умноженията най-много 7 пъти.

След публикуването на алгоритъма от Щрасен са правени редица успешни и неуспешни опити за неговото подобрение. Къде са слабите места на алгоритъма и как могат да бъдат подсилени? Единият проблем вече стана ясен: неефективното предварително допълване на матриците, което може да го забави до 7 пъти. Вторият проблем също беше засегнат в началото: при $n = 2$

практически имаме влошаване, вместо подобрение. А как стоят нещата при $n = 3, 4, \ldots$? Къде (за кое n) да спрем да прилагаме рекурсивно алгоритъма на Щрасен и да използваме класическия?

Едно възможно решение на първия проблем (n не е степен на 2) е *да отлагаме* допълването на матриците с нули, докато това е възможно. Т. е. ако първоначално n е четно, можем да извършим една итерация на алгоритъма. Ако и $n/2$ е четно, можем да извършим още една, а ако е нечетно — ще трябва по необходимост да направим допълване. Извършвайки допълване едва сега обаче, евентуално си спестяваме вмъкването на много допълнителни нули. Тази техника е известна като *динамично допълване*.

Въпреки че води до сериозно подобрение, динамичното *допълване* не е оптималният метод. *Хъс* и *Ледерман* прилагат друга техника за справяне с нечетни размери: *динамично обелване*. Идеята е да се отделят един ред и един стълб. Получената "обелена" матрица има четен размер n и за нея може да се приложи рекурсивната стъпка на Щрасен. (*виж фигура 7.6а.*)

$$\left[\begin{array}{c|c} A_{11} & a_{12} \\ \hline a_{21} & a_{22} \end{array}\right]\left[\begin{array}{c|c} B_{11} & b_{12} \\ \hline b_{21} & b_{22} \end{array}\right]$$

Фигура 7.6а. Динамично "обелване" на A и B.

След това полученият резултат трябва да се обедини с резултатната матрица C_{11} (*виж фигура 7.6б.*).

$$\left[\begin{array}{c|c} C_{11} & c_{12} \\ \hline c_{21} & c_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11}B_{11} + a_{12}b_{21} & A_{11}b_{12} + a_{21}b_{22} \\ \hline a_{21}B_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{array}\right]$$

Фигура 7.6б. Получаване на C от произведението на "обелените" A и B.

Вторият проблем не може да бъде атакуван директно, тъй като изисква сравнение между скоростта на класическия алгоритъм и този на алгоритъма на Щрасен при фиксирано n. Това изисква да се вземе предвид времето за достъп до елементите на матрицата при двата метода, което е машинно зависимо. Изследвания от 1996 на *Луома* и *Паука* показват, че обемът на кеш-паметта е от огромно значение и *в повечето практически случаи оригиналният алгоритъм на Щрасен се държи по-зле от класическия*. Те предлагат идея за реализация на алгоритъма, при която междинните пресмятания се извършват *точно* преди да бъдат необходими, което гарантира тяхното оставане в кеш-паметта до следващото им ползване.

Друго неочаквано подобрение на алгоритъма прави *Виноград*. Той успява да намали броя на събиранията/изважданията от 18 на 15, запазвайки броя на умноженията. Неговата схема има вида:

$$\begin{aligned}
S_1 &= a_{21} + a_{22} \\
S_2 &= S_1 - a_{11} \\
S_3 &= a_{11} - a_{21} \\
S_4 &= a_{12} - S_2 \\
T_1 &= b_{12} - b_{11} \\
T_2 &= b_{22} - T_1 \\
T_3 &= b_{22} - b_{12} \\
T_4 &= b_{21} - T_2 \\
P_1 &= a_{11}b_{11}
\end{aligned}$$

$$P_2 = a_{12}b_{21}$$
$$P_3 = S_1T_1$$
$$P_4 = S_2T_2$$
$$P_5 = S_3T_3$$
$$P_6 = T_4b_{22}$$
$$P_7 = a_{22}T_4$$
$$U_1 = P_1 + P_4$$
$$U_2 = U_1 + P_5$$
$$U_3 = U_1 + P_3$$
$$c_{11} = P_1 + P_2$$
$$c_{21} = U_2 + P_7$$
$$c_{22} = U_2 + P_3$$
$$c_{12} = U_3 + P_6$$

Правени са редица други опити за намаляване броя на умноженията и събиранията в класическата схема на Щрасен при $n = 2$, но всичките са безуспешни. През 1971 г. *Хопкрофт* и *Кер* доказват, че това е невъзможно, ако не се ползва комутативността на умножението (Умножението на матрици не е комутативно.). По-късно бива открит начин за умножение на матрици 3×3 с най-много 21 умножения, което дава алгоритъм със сложност $\Theta(n^{\log_3 21})$, т. е. $\Theta(n^{2,771244})$. Следват редица други подобрения при все по-големи размери на матриците и все по-малък шанс за реално практическо приложение. Така например, Пан открива начин за умножение на матрици с размер 70×70 с 143 640 елементарни умножения (срещу 343 000, според класическия алгоритъм). По-късно са открити и още по-ефективни на теория и още по-неприложими на практика алгоритми: $\Theta(n^{2,521813})$ — 1979 г., $\Theta(n^{2,521801})$ — 1980 г., $\Theta(n^{2,376})$ — 1986 г.

Задачи за упражнение:

1. Да се установи опитно кога следва да се прекъсва рекурсията и да се прилага класическият алгоритъм.
2. Да се провери предложената от Виноград последователност за бързо умножение на матрици.
3. Да се предложи друга последователност за бързо умножение на матрици.

7.7. Бързо умножение на дълги числа

Класическият алгоритъм за умножение на числа, изучаван в училище, изисква време от порядъка на $\Theta(n^2)$ за умножение на две n-цифрени числа (предполагаме, че събирането и умножението на едноцифрени числа става за константно време, независещо от n). До такава степен сме свикнали с този алгоритъм, че едва ли някой от нас се е замислял за неговата ефективност и за възможностите за неговото подобрение.

Да видим какво би могъл да ни даде методът *разделяй и владей*. Да предположим, че искаме да умножим две n-цифрени числа X и Y, като за момента считаме, че n е степен на 2 (по-късно ще покажем как да се справим, когато това не е изпълнено). Да разбием X и Y на две части, съответно A, B и C, D с $n/2$ цифри всяка, така че:

$$X = 2^{n/2}.A + B$$
$$Y = 2^{n/2}.C + D$$

Оттук за произведението XY получаваме:

$$XY = 2^n.AC + 2^{n/2}.(AD+BC) + BD$$

Така, за пресмятане стойността на XY по горната формула са необходими 4 умножения на $n/2$ цифрени числа, 3 събирания на числа с най-много n+1 цифри и две измествания наляво на $n/2$

позиции (умножения по 2^n и $2^{n/2}$). Ако за пресмятане произведенията AC, AD, BC и BD използваме същата формула рекурсивно, като процесът продължава до достигане на елементарния случай от две едноцифрени числа, получаваме рекурентната зависимост:

$$T(1) = 1$$
$$T(n) = 4T(n/2) + cn,\ n > 1 \text{ и } n \text{ — степен на 2}$$

Прилагайки *основната теорема* (*виж. 1.4.10.*), получаваме $T(n) \in \Theta(n^2)$, т. е. сложността съвпада с тази на изходния алгоритъм, което означава, че простото разделяне не води до подобрение: нужна е и малко съобразителност. Да си припомним алгоритъма на Щрасен по-горе (*виж 7.6.*), при който подобрението дойде от намаляване броя на *умноженията*. Да видим дали не бихме могли да приложим същия принцип и тук. Ако се вгледаме по-внимателно във формулата, ще видим, че всъщност не ни е непременно необходимо да пресмятаме произведенията AD и BC: достатъчна ни е тяхната *сума*. Да разгледаме формулата:

$$XY = 2^n.AC + 2^{n/2}.[(A\text{–}B)(D\text{–}C) + AC + BD] + BD$$

Тя изисква само 3 умножения, 6 събирания (включително 2 изваждания) и 2 измествания. Сложността на алгоритъма в този случай се дава с рекурентната зависимост (предполагаме, че дъното на рекурсията настъпва при едноцифрени числа, т. е. при $n = 1$, когато се извършва елементарно умножение на едноцифрени числа):

$$T(1) = 1$$
$$T(n) = 3T(n/2) + cn,\ n > 1 \text{ и } n \text{ — степен на 2}$$

Сега по *основната теорема* (*виж 1.4.10.*) получаваме $T(n) \in \Theta(n^{\log_2 3})$. Но $\log_2 3 \approx 1{,}59 < 2$, т. е. полученият алгоритъм е по-добър от класическия. Същия резултат можем да получим, използвайки вариации на горната формула, например:

$$XY = 2^n.AC + 2^{n/2}.[(A+B)(C+D) - AC - BD] + BD$$

Няма да привеждаме работеща реализация на алгоритъма, тъй като това е свързано с операции с "дълги числа" (т. е. непобиращи се в стандартните типове), което ще претрупа кода. (Освен това една добра реализация вероятно няма да използва десетична бройна система, а максималната такава с основа степен на 2, позволяваща резултатът от умножението на две цифри да се събере в една машинна дума. Това обаче е извън интереса на нашите разглеждания.) Вместо това ще се опитаме да го доизясним чрез псевдокод. Предполагаме, че X и Y са цели числа със знак, а n е степен на 2. В граничния случай $n = 1$ ще умножаваме числата директно. [*Aho,Hopcroft,Ullman–1987*]

```
int mult (int X, Y, n)
{
   int s;          /* знак на произведението XY */
   int m1,m2,m3;  /* трите произведения */
   int A,B,C,D;   /* половинките на X и Y */

   /* проверка за граничния случай */
   if (1 == n)
      return (X*Y);
   s = sign(X) *sign(Y);
   X = abs(X);
   Y = abs(Y);
   A = левите n/2 бита от X;
   B = десните n/2 бита от X;
   C = левите n/2 бита от Y;
   D = десните n/2 бита от Y;
   m1 = mult(A,C,n/2);
   m2 = mult(A-B,D-C,n/2);                    /* *** */
   m3 = mult(B,D,n/2);
```

```
    return s * (m1<<n + (m1+m2+m3)<<(n/2) + m3); /* *** */
}
```

Бихме могли да ползваме и втория вариант на формулата, при което редовете, отбелязани с `/* *** */`, трябва да се заменят със следните:

```
m2 = mult(A+B,C+D,n/2);                          /* *** */
return s * (m1<<n + (m2-m1-m3)<<(n/2) + m3); /* *** */
```

Какво става, когато n не е степен на 2? Ако n е четно, на първата стъпка няма проблем и можем да разделим числото на две части с еднаква дължина. Ако и $n/2$ е четно, можем да продължим така, но все едно на някоя стъпка ще получим нечетна дължина и това няма да бъде възможно. Едно очевидно решение на проблема е числото да се допълни отляво с една водеща нула. Макар че това води до влошаване ефективността на алгоритъма, анализът показва, че сложността му остава $\Theta(n^{\log_2 3})$.

Друг интересен момент, който беше пропуснат в анализа по-горе, е фактът, че сумата на две $n/2$-цифрени числа не дава непременно n-цифрено число, а би могла да даде и $(n+1)$-цифрено. Тогава произведението $(A+B)(C+D)$, при втория вариант на формулата, може да бъде $(n+1)$-цифрено (останалите произведения AC и BD нямат такъв шанс.). Да вземем $X = 9999$ и $Y = 9998$. Имаме:

$A = 99$, $B = 99$, $C = 99$ и $D = 98$
$AC = 99.99 = 9801$
$BD = 99.98 = 9702$
$(A+B)(C+D) = 198.197 = 39006$ (т. е. 5-цифрено число)

За щастие нарастването е най-много с 1 и крайната сложност на алгоритъма и в този случай ще бъде $\Theta(n^{\log_2 3})$. Няма да се спираме подробно на анализа на тези случаи. Любознателният читател би могъл да намери повече подробности в [*Brassard, Bratley – 1987*].

Какво да правим, когато двете числа имат дължини m и n, $m \neq n$? Нека за определеност предположим, че $m < n$. Ако запълним по-късото число с водещи нули, получаваме алгоритъм със сложност $\Theta(n^{\log_2 3})$ срещу $\Theta(mn)$ при класическия алгоритъм. Оттук лесно се вижда, че класическият алгоритъм е по-добър от подобрения, при $m < n^{(\log_2 3)-1}$. Едно добро решение е да разделим цифрите на по-дългото число на блокове с дължина m. Всеки от тези блокове се умножава с първото число с използване на подобрения алгоритъм, след което с помощта на $\lceil n/m \rceil$ допълнителни събирания и изместване се получава крайният резултат. Общата сложност на така модифицирания алгоритъм е $\Theta(mn^{(\log_2 3)-1})$.

Възниква логичният въпрос: Щом този алгоритъм е наистина толкова добър, защо не го изучаваме в училище? По две причини: първо той е по-сложен за обяснение и значително по-труден за прилагане, особено при по-дълги числа. Понятието "рекурсия" е абстрактно и трудно за схващане. От друга страна, вземайки предвид скритите константи, можем да очакваме, че подобреният алгоритъм е по-добър от класическия при числа с поне 500 цифри. Едва ли някой извършва ръчно такива пресмятания в училище…

Задачи за упражнение:

1. Да се реализира предложеният алгоритъм за бързо умножение на дълги числа.
2. Да се определи експериментално при колко цифри бързият алгоритъм за умножение на дълги числа е по-добър от класическия.
3. Да се подобри алгоритъмът за бързо умножение на дълги числа, като за целта числата се разделят на три, вместо на две части. Да се покаже, че 5 (а не 9) умножения на числа с дължина $n/3$ са достатъчни за пресмятане на произведението. Да се определи сложността на получения алгоритъм.
4. На базата на предходната задача да се покаже, че $2k-1$ умножения са достатъчни за пресмятане произведението в случай на разбиване на групи от по k последователни цифри. Да се

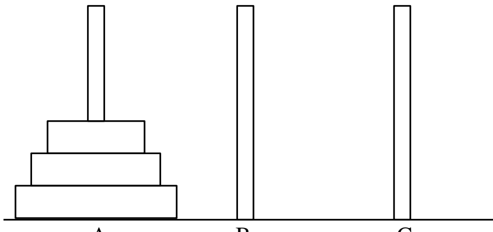
покаже, че оттук следва съществуването на алгоритъм за умножение на дълги числа със сложност $\Theta(n^x)$ за всяко $x > 1$.

7.8. Задача за Ханойските кули

Задача: Дадени са n на брой диска с различни диаметри и три стълба A, B и C. Дисковете са нанизани върху първия стълб по реда на намаляване на размера им и образуват кула. (*виж фигура 7.8.*) Трябва да се прехвърлят от първия стълб върху последния при спазване на следните правила:

1. При всеки ход може да бъде преместен един диск и този диск трябва да бъде най-горен за някой от стълбовете.
2. Диск с по-голям диаметър не може да бъде поставен върху такъв с по-малък.

Ще изложим класическия рекурсивен вариант, основан на метода *разделяй и владей*, както и някои от най-интересните итеративни решения. На читателите, които проявяват допълнителен интерес към проблема, бихме препоръчали да се запознаят с [*Банчев-1993*], където са посочени и някои други алгоритми.



Фигура 7.8. Ханойските кули в начална позиция.

Алгоритъм 1

Това е класическият алгоритъм за решаване на задачата. Идеята е, че, за да преместим n на брой диска от стълб A на стълб C, е необходимо да преместим $n-1$ диска от стълб A на стълб B, след което да преместим диска с номер n (който е най-големият измежду тях) от стълб A на стълб C и накрая да преместим останалите $n-1$ диска от стълб B на стълб C. Така по естествен начин сведохме задачата до решаване на два други нейни екземпляра с по-малък размер (*разделяй*), чието обединение, заедно с една допълнителна операция (прехвърляне на най-големия диск) ни дава търсеното решение (*владей*). Реализацията на този алгоритъм изглежда така:

```
#include <stdio.h>
const unsigned n = 4;

void diskMove(unsigned n, char a, char b)
{ printf("Преместете диск %u от %c на %c.\n", n, a, b); }

void hanoy(char a, char c, char b, unsigned numb)
{ if (1 == numb)
    diskMove(1, a, c);
  else {
    hanoy(a, b, c, numb - 1);
    diskMove(numb, a, c);
    hanoy(b, c, a, numb - 1);
  }
}
int main(void) {
  printf("Брой дискове: %u\n", n);
```

```
  hanoy('A', 'C', 'B', n);
  return 0;
}
```
🖫 hanoy.c

Следват примерни резултати от изпълнението на програмата за n = 2, 3 и 4:

```
Брой дискове: 2
Преместете диск 1 от A на B.
Преместете диск 2 от A на C.
Преместете диск 1 от B на C.

Брой дискове: 3
Преместете диск 1 от A на C.
Преместете диск 2 от A на B.
Преместете диск 1 от C на B.
Преместете диск 3 от A на C.
Преместете диск 1 от B на A.
Преместете диск 2 от B на C.
Преместете диск 1 от A на C.

Брой дискове: 4
Преместете диск 1 от A на B.
Преместете диск 2 от A на C.
Преместете диск 1 от B на C.
Преместете диск 3 от A на B.
Преместете диск 1 от C на A.
Преместете диск 2 от C на B.
Преместете диск 1 от A на B.
Преместете диск 4 от A на C.
Преместете диск 1 от B на C.
Преместете диск 2 от B на A.
Преместете диск 1 от C на A.
Преместете диск 3 от B на C.
Преместете диск 1 от A на B.
Преместете диск 2 от A на C.
Преместете диск 1 от B на C.
```

За дадената задача съществуват най-различни начини за преобразуване на рекурсивното решение в итеративно. Няма да се спираме на стандартния метод на преобразуване на рекурсията в итерация с използването на стек, а ще разгледаме само достатъчно ясни алгоритми, основани на набор от правила, гарантиращи на всяка стъпка наличието на единствен възможен ход. Ще оставим на читателя да се убеди сам, че всички следващи алгоритми строят същото решение като *алгоритъм 1*. Всъщност става въпрос за различен изказ на една и съща идея.

Алгоритъм 2

Конструирането на решението се определя от следните правила:

1. Ако n е нечетно, всеки диск с нечетен номер се премества винаги надясно, а тези с четни номера — винаги наляво. Ако n е четно — посоката е обратна.
2. С един и същ диск не се правят два последователни хода.
3. Не се поставя по-голям диск върху по-малък.

Алгоритъм 3

Правила:

1. При ходовете с нечетен номер най-малкият диск се премества надясно или наляво в зависимост от това дали n е нечетно (наляво) или четно (надясно).
2. При ходовете с четен номер се прави единственото възможно преместване, в което не участва най-малкият диск.
3. Не се поставя по-голям диск върху по-малък.

Алгоритъм 4

Един от стълбовете се избира за белязан и се фиксира посока на промяна на белязания стълб. При нечетно n първоначално белязан е C, а посоката на промяна е наляво; при четно n белязан е B и посоката на промяна е обратна.

1. Ако върху двата небелязани стълба има дискове, то се премества по-малкият от двата най-горни диска върху по-големия. Ако единият от небелязаните стълбове е празен, то върху него се премества най-горният диск от другия стълб. Ако и другият е празен, то задачата е решена.
2. Избира се за белязан съседният на белязания стълб във фиксираната предварително посока на промяна.

Задачи за упражнение:

1. Да се реализират *алгоритми 2*, *3* и *4*.
2. Да се докаже, че *алгоритми 2*, *3* и *4* извършват същите действия като *алгоритъм 1*.

7.9. Организиране на първенства

Друга класическа задача, решавана ефективно с помощта на *разделяй и владей*, е тази за изготвяне план на турнир от типа "всеки срещу всеки". Например:

Футболно първенство (*III Балканиада по информатика Варна'95*)

Министерството на Спорта на една балканска страна трябва да организира футболно първенство с n отбора (номерирани от 1 до n). Шампионатът трябва да се проведе в n кръга, ако n е нечетно, и в n–1 кръга, ако е четно. Всеки два отбора играят един срещу друг точно по веднъж. Съставете програма, която създава програма на първенството.

Резултатът трябва да представлява таблица от цели числа с n реда. Числото от j-тия стълб на i-тия ред на таблицата означава номера на отбора, който играе срещу отбор с номер i в кръг номер j. (Ако отбор i играе срещу отбор k в кръг j, естествено това означава, че отбор k играе срещу отбор i в същия кръг.)

Ако отбор i почива в j-тия кръг, тогава j-тото число в i-тия ред е 0.

Например, при n = 3 и n = 4 съответните разписания ще имат следния вид (по вертикала: отбор, по хоризонтала: кръг):

```
2 3 0        2 3 4
1 0 3        1 4 3
0 1 2        4 1 2
             3 2 1
```

Решение: Ще си позволим леко да изменим формата на резултата. Вместо посочената в условието таблица ще генерираме друга, винаги с размер $n \times n$ (дори и за четно n), от тип "отбор-отбор", като на i-тия ред и j-тия стълб ще се намира кръгът, в който ще се срещнат отборите с номера i и j ($1 \leq i, j \leq n$). Преобразуването на таблицата към искания в задачата формат оставяме на читателя като леко упражнение.

Очевидно задачата няма смисъл при n = 1. При n = 2 се играе точно една игра, което отнема точно един ден: *виж таблица 7.9а*.

отбори	1	2
1	0	1
2	1	0

Таблица 7.9а. Разписание на срещите за 2 отбора.

Въвеждаме съкратения запис i:j, означаващ, че "отбор i среща отбор j" ($1 \leq i, j \leq n$). В случай на четири отбора турнирът завършва за 3 дни. Как да конструираме съответното разписание

на срещите? Бихме могли да разделим отборите по двойки, като през първия ден 1:2 и 3:4, през втория ден 1:3 и 2:4, а през третия — 1:4 и 2:3. Получаваме *таблица 7.9б.*

отбори	**1**	**2**	**3**	**4**
1	0	1	2	3
2	1	0	3	2
3	2	3	0	1
4	3	2	1	0

Таблица 7.9б. Разписание на срещите за 4 отбора.

При по-внимателен поглед забелязваме, че тази таблица съдържа в себе си два пъти таблицата за $n = 2$ и два пъти таблицата

```
2 3
3 2
```

Забележете, че последната се получава от

```
0 1
1 0
```

чрез поелементно добавяне на числото 2.

Досещате ли се вече как можем да получим съответния план на турнира за $n = 8$? Да изкопираме четири пъти таблицата за $n = 4$, след което да прибавим поелементно числото 4 към таблиците с размер 4 извън главния диагонал. (*виж таблица 7.9в.*)

отбори	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
3	2	3	0	1	6	7	4	5
4	3	2	1	0	7	6	5	4
5	4	5	6	7	0	1	2	3
6	5	4	7	6	1	0	3	2
7	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

Таблица 7.9в. Разписание на срещите за 8 отбора.

Непосредствено се проверява, че *таблица 7.9в.* представлява вярно разписание: всеки ред и всеки стълб съдържат пермутация на числата от 0 до 7 без повторение, т. е. турнирът приключва в 7 дни, като всеки отбор играе точно една среща на ден. Таблицата е симетрична, т. е. ако i:j, то е вярно и j:i, и обратно ($1 \leq i, j \leq n$). По главния диагонал стоят нули, т. е. не е вярно i:i, за $1 \leq i \leq n$. По аналогичен начин се получава и таблицата за $n = 16, 32, 64, \ldots$ и изобщо за n, точна степен на 2. Това лесно следва по индукция. Доказателството оставяме на читателя като леко упражнение. Да разсъждаваме обратно. Нека е дадено $n = 2^k$. За да получим таблица за n, трябва да получим таблица за $n/2$ (*разделяй*), след което да я копираме 4 пъти и да добавим константа към две от копията (*владей*). Следва примерна реализация на описания алгоритъм.

```
#include <stdio.h>
#define MAX 100
unsigned m[MAX][MAX];

void copyMatrix(unsigned stX, unsigned stY, unsigned cnt, unsigned add)
{ unsigned i, j;
  for (i = 0; i < cnt; i++)
```

```
    for (j = 0; j < cnt; j++)
      m[i + stX][j + stY] = m[i + 1][j + 1] + add;
}
void findSolution(unsigned n) /* Построява таблицата */
{ unsigned i;
  m[1][1] = 0;
  for (i = 1; i <= n; i <<= 1) {
    copyMatrix(i + 1, 1, i, i);
    copyMatrix(i + 1, i + 1, i, 0);
    copyMatrix(1, i + 1, i, i);
  }
}

void print(unsigned n)        /* Извежда резултата */
{ unsigned i, j;
  for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++)
      printf("%3u", m[i][j]);
    printf("\n");
  }
}

int main(void) {
  const unsigned n = 8;
  findSolution(n);
  print(n);
  return 0;
}
```
🖫 tourn1.c

За съжаление, в този си вид алгоритъмът е непригоден за n, които не са степени на 2. Оставяме на читателя да се опита да го модифицира така, че да работи за всяко естествено n. Подолу ще разгледаме друг алгоритъм, валиден за всяко n.

За нечетно n таблиците ще генерираме по следния начин: Ще попълваме не таблица $n \times n$, а такава с още един допълнителен стълб, т. е. $n \times (n+1)$. Попълването ще извършваме по редове, започвайки от най-левия стълб и придвижвайки се надясно. Когато попълним $(n+1)$-вия стълб от текущия ред, ще преминаваме към следващия ред. Попълването ще започнем със стойност 1 на ред 1 стълб 1, като поредната клетка ще попълваме със следващата по ред стойност в интервала $[1;n]$. При достигане на стойност n ще започваме отново от 1. *Таблица 7.9г.* показва резултата при $n = 5$. Впрочем, вижда се, че можем и да не попълваме $(n+1)$-ия стълб — той съвпада с първия стълб. (*Защо?*)

отбори	**1**	**2**	**3**	**4**	**5**	**6**
1	1	2	3	4	5	1
2	2	3	4	5	1	2
3	3	4	5	1	2	3
4	4	5	1	2	3	4
5	5	1	2	3	4	5

Таблица 7.9г. Начало на попълване на разписанието на срещите за 5 отбора.

След приключване на попълването нулираме всички елементи по главния диагонал на матрицата (*виж таблица 7.9д.*).

отбори	**1**	**2**	**3**	**4**	**5**	**6**
1	0	**2**	**3**	**4**	**5**	1

2	2	0	**4**	**5**	**1**	2
3	3	4	0	**1**	**2**	3
4	4	5	1	0	**3**	4
5	5	1	2	3	0	5

Таблица 7.9д. Разписание на срещите за 5 отбора.

При четни n бихме могли просто да въведем фиктивен отбор. Тук обаче ще приложим друга стратегия. Ще решим задачата за n–1, което е нечетно, след което, докато нулираме главния диагонал, допълнително ще попълним n-тите ред и стълб: Вместо да почива в кръг $m[k][k]$, k-тият отбор трябва да играе с n-тия, защото при четно n почиващи няма.

Така например, за да съставим разписание за n = 4, първо попълваме таблица за n = 3 (*виж таблица 7.9е.*).

отбор	1	2	3	4
1	1	2	3	1
2	2	3	1	2
3	3	1	2	3

Таблица 7.9е. Начало на попълване на разписанието на срещите за 3 отбора.

отбор	1	2	3	4
1	0	**2**	**3**	**1**
2	2	0	**1**	**2**
3	3	1	0	**3**
4	1	2	3	0

Таблица 7.9ж. Разписание на срещите за 4 отбора.

След това ще нулираме главния диагонал и ще попълним n-тия ред така, както е показано на *таблица 7.9ж.* Следва съответно изменение на функцията `findSolution()`:

```
void findSolution(unsigned n) /* Построява таблицата */
{ unsigned i;
  unsigned saveN = n;
  if (n % 2 == 0) /* Ако n е четно, задачата се свежда към n-1 */
    n--;

  /* Попълва се таблицата за n - тук е гарантирано нечетно. */
  for (i = 0; i < n * (n + 1); i++)
    m[i % (n + 1)][i / (n + 1)] = i % n + 1;

  /* Възстановява се стойността на n */
  n = saveN;

  for (i = 0; i < n; i++) {
    if (n%2 == 0) /* Запълват се последният стълб и ред, четно n */
      m[i][n - 1] = m[n - 1][i] = m[i][i];
    m[i][i] = 0;        /* Главният диагонал се запълва с 0 */
  }
}
```

🖫 tourn2.c

Горното решение, макар и ефективно, не позволява директен отговор на въпроса: В кой кръг се срещат отбори i и j? За това се изисква пълно или поне частично попълване на съответната таблица. При по-внимателен поглед върху горните таблици, могат да се изведат съответни формули.

Нека имаме четен брой участници ($n = 2k$). Нека участниците A и B имат номера s и t, които са различни от n. Тогава играят в кръг $s + t - 1$, ако $s + t \leq n$, и в $s + t - n$, ако $s + t > n$. В случай на шахматен турнир има значение още кой играе с белите и кой — с черните фигури. Едно възможно решение е да се следва правилото: с белите ще играе шахматистът с по-малък номер, само ако $s + t$ е нечетно. Участникът с номер n играе с онзи, който има номер l в кръг $2l - 1$, ако $2l \leq n$ и в кръг $2l - n$, когато $2l > n$. В случай на шахматен турнир с участниците с номера от 1 до $n/2$ той играе с черните фигури, а с останалите — с белите. При нечетен брой участници прибавяме фиктивен участник и играещият с него почива в съответния кръг.

Макар последната схема да е по-подходяща за получаване на директен отговор на въпроса "В кой кръг играят i и j?", тя може да се използва и за попълване на таблицата, поради простотата на формулата. Отново можем да се откажем от фиктивния отбор и да попълним директно съответните стойности на последния стълб. Така получаваме:

```
void findSolution(unsigned n) /* Построява таблицата */
{ unsigned i, j;
  unsigned saveN = n;
  if (n % 2 == 0) /* Ако n е четно, задачата се свежда към n-1 */
    n--;

  /* Попълване на таблицата за n - тук е гарантирано нечетно. */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if ((m[i][j] = i + j + 1) > n)
        m[i][j] -= n;

  /* Възстановяване на стойността на n */
  n = saveN;

  for (i = 0; i < n; i++) {
    if (n % 2 == 0) /* Запълване последния стълб и ред, четно n */
      m[i][n - 1] = m[n - 1][i] = m[i][i];
    m[i][i] = 0;        /* Запълване на главния диагонал с 0 */
  }
}
```

🖫 tourn3.c

Горната функция се различава малко от теоретичните разсъждения поради това, че индексирането на масиви в Си започва от 0, вместо от 1.

Задачи за упражнение:

1. Да се преобразува конструираната таблица в таблица от вида, искан в условието на задачата от Балканиадата.
2. Да се докаже коректността на първия алгоритъм за съставяне на разписание на турнир за n степен на 2.
3. Да се реализира вторият алгоритъм за съставяне на разписание на турнир, като за четни n се въвежда фиктивен отбор.
4. Да се докаже коректността на втория алгоритъм за съставяне на разписание на турнир за всяко естествено n, $n \geq 2$.
5. Да се изведат приведените формули за съставяне на разписанието на шахматен турнир.

7.10. Циклично изместване на елементите на масив

През шестдесетте години при разработката на текстов редактор за системата *UNIX* Кърниган и Плоджер се сблъскват с проблема за цикличното изместване елементите на масив. Скромната изчислителна мощ на тогавашните компютри поставя строги изисквания за ефективността на тази операция както по отношение на изразходваното процесорно време, така и по отношение на необходимата оперативна памет. Първоначалният вариант на програмата, основан на свързани списъци, се оказал недостатъчно ефективен и сложен и дори съдържал няколко грешки. След пообстойно вникване в същността на задачата, двамата успяват да създадат кратка програма, ефективна както по време, така и по памет, използвана по-късно в редица други текстови редактори.

Съществуват най-общо пет различни алгоритми за решаване на задачата, като всички ще бъдат разгледани по-долу. Нека обозначим масива с m, а броя на елементите му с n. Как бихме могли да извършим изместването? Съществуват две достатъчно прости решения, всяко от които има своите недостатъци: първото е неефективно по памет, а второто — по време.

Нека разгледаме първото решение. Ако разполагаме с известен обем допълнителна оперативна памет, бихме могли да реализираме следния

Алгоритъм 1

1. Копираме първите k елемента на m в някакъв временен масив x.
2. Изместваме останалите $n-k$ елемента на k позиции вляво.
3. Копираме елементите от x обратно в m (на последните k позиции).

Горният алгоритъм е очевидно неефективен по отношение на използваната оперативна памет. Същевременно е ясно, че в частния случай при $k = 1$ бихме могли да реализираме функция `shiftBy1()`, която да изисква константна допълнителна памет и време от порядъка на $\Theta(n)$. Получаваме нов алгоритъм:

Алгоритъм 2

1. Извикваме k пъти `shiftBy1()`.

За съжаление, предложеният алгоритъм, макар и ефективен по памет, в най-лошия случай изисква време от порядъка на $\Theta(n^2)$.

Не бихме ли могли да намерим алгоритъм, едновременно ефективен и по време, и по памет, съчетаващ предимствата на горните алгоритми? Оказва се, че такъв алгоритъм съществува. Идеята е да преместим първия елемент на `m[]` във временна променлива `tmp`, след което да изместваме последователно `m[k+1]` в `m[1]`, `m[2k+1]` в `m[k+1]` и т. н., вземайки всички индекси на `m[]` по модул n. Процесът продължава до достигане на `m[1]`, при което вместо него при размяната се използва `tmp`. В случай, че не всички елементи са си на местата, повтаряме процеса, започвайки от `m[2]` и т.н. Една примерна реализация на изложената идея изглежда така (Тук индексите са отместени, тъй като индексирането на масива започва от 0.):

```
#include <stdio.h>

#define MAX 100

struct CElem {
  int data;
  /* ... */
} m[MAX];

const unsigned n = 10; /* Брой елементи в масива */
const unsigned k = 2;  /* Брой позиции на отместване */
```

```
void init(void)
{ unsigned i;
  for (i = 0; i < n; i++)
    m[i].data = i;
}

unsigned gcd(unsigned x, unsigned y)
{ while (y > 0) {
    unsigned tmp = y;
    y = x % y;
    x = tmp;
  }
  return x;
}

void shiftLeft1(unsigned k)
{ /* Измества m[] на k позиции наляво, ползва помощна променлива */

  unsigned i, ths, next, gcdNK;
  struct CElem tmp;

  for (gcdNK = gcd(n, k), i = 0; i < gcdNK; i++) {
    ths = i; tmp = m[ths];
    next = ths + k;
    if (next >= n)
      next -= n;
    while (next != i) {
      m[ths] = m[next];
      ths = next;
      next += k;
      if (next >= n)
        next -= n;
    }
    m[ths] = tmp;
  }
}

void print(void)
{ unsigned i;
  for (i = 0; i < n; i++)
    printf("%d ", m[i].data);
  printf("\n");
}

int main(void) {
  init();
  shiftLeft1(k);
  print();
  return 0;
}
```
🖫 shift1.c

Бихме могли да погледнем на нещата и по друг начин. Нека означим с A първите k елемента на m, а с B — останалите $n-k$. Тогава на цикличното изместване на елементите на масива m наляво на k позиции можем да гледаме като на преобразуване на AB в BA. Без ограничение на общността на разсъжденията бихме могли за определеност да считаме, че $k < n/2$, т. е. че A е по-къс от B. Разделяме B на две части B_L и B_R, като B_R съдържа k елемента, т. е. толкова, колкото и A.

Тогава цикличното изместване за свежда до преобразуване на AB_LB_R в B_LB_RA. Ако разменим местата на A и B_R (за което са ни достатъчни k размени), елементите на A ще дойдат на окончателните си позиции. Остава да се реши проблемът с B_L и B_R — двете части на B. За B получаваме нова задача, аналогична на изходната, но с по-малък размер. Бихме могли да продължим процеса рекурсивно, до окончателното решаване на задачата: *разделяй и владей*. Макар на пръв поглед да изглежда сложно, описаният процес е достатъчно прост и може да бъде описан и итеративно. Гриз и Милз предлагат следната итеративна реализация:

```
void swap(unsigned a, unsigned b, unsigned l)
{ /* Разменя местата на подмасивите m[a..a+l-1] и m[b..b+l-1] */
  unsigned i;
  struct CElem tmp;

  for (i = 0; i < l; i++) {
    tmp = m[a + i];
    m[a + i] = m[b + i];
    m[b + i] = tmp;
  }
}

void shiftLeft2(unsigned k)
{ /* Измества масива m[] на k позиции наляво.
   * рекурсивен процес, реализиран итеративно } */
  unsigned i, j, p;
  p = i = k;
  j = n - k;
  while (i != j)
    if (i > j) {
      swap(p - i, p, j);
      i -= j;
    }
    else {
      swap(p - i, p + j - i, i);
      j -= i;
    }
  swap(p - i, p, i);
}
```

🖫 shift2.c

И така, вече разполагаме с два ефективни и по време, и по памет алгоритми. Ще предложим още един ефективен и особено изящен алгоритъм, а именно използвания от Кърниган и Плоджер, основаващ се на следното просто математическо тъждество:

$$BA = (A^R B^R)^R.$$

С A^R по-горе сме означили масива, получен от елементите на A, взети в обратен ред. Аналогичен е смисълът и на B^R и $(A^R B^R)^R$. Горното тъждество ни дава изключително прост и ефективен алгоритъм за решаване на задачата, а именно: за да получим BA следва да обърнем A, да обърнем B, след което да обърнем целия масив. Следва реализация на изложения алгоритъм:

```
void reverse(unsigned a, unsigned b) /* Обръща подмасива m[a..b] */
{ unsigned i, j, k, cnt;
  struct CElem tmp;
  for (cnt = (b-a)/2, k=a, j=b, i=0; i <= cnt; i++, j--, k++) {
    tmp = m[k];
    m[k] = m[j];
    m[j] = tmp;
```

```
  }
}

void shiftLeft3(unsigned k)
{ /* Измества масива m на k позиции наляво, на три стъпки */
  reverse(0, k - 1);
  reverse(k, n - 1);
  reverse(0, n - 1);
}
```
🖫 shift3.c

Последните три алгоритъма изискват константна допълнителна памет и се изпълняват за време $\Theta(n)$. Въпреки това, при по-големи стойности на n те не са равностойни като бързодействие, тъй като е различна константата пред n. Действително, при относително сложния `ShiftLeft1()` всеки елемент на масива се чете и запомня точно веднъж, докато при `ShiftLeft3()` тези операции очевидно се извършват двукратно (*виж* [*Bentley-1990*], [*Наков-1998d*]).

Задачи за упражнение:

1. Да сравни теоретично времето за работа на функциите `shiftLeft1()`, `shiftLeft2()` и `shiftLeft3()`.
2. Да се изведе формулата $BA = (A^R B^R)^R$.

7.11. Покриване с шаблон

Алгоритмите от тип *разделяй и владей* лесно се доказват по индукция и често следват от индуктивни математически разсъждения. По-горе тръгвахме от алгоритъма и едва по-късно давахме формална обосновка. Този път ще тръгнем от математическото доказателство и оттук ще изведем алгоритъма.

Задача: Дадена е квадратна дъска с размер $n \times n$, където n е степен на 2. Едно от квадратчетата е избрано за специално и е запълнено. Дадени са още достатъчно на брой еднотипни шаблона, представляващи дъска с размер 2×2, от която е премахнато едното квадратче. Целта е да се намери покритие на дъската с шаблони, като не се допуска многократно покриване на някое квадратче.

Решение: Първо ще покажем, че задачата винаги има решение.

Не е трудно да се види, че броят на незапълнените квадратчета $2^{2k}-1$ се дели на 3 за всяко естествено k, $k > 0$. За целта е достатъчно да го запишем във вида: $(2^k-1)(2^k+1)$. Да разгледаме числата 2^k-1, 2^k и 2^k+1. Това са последователни числа и са точно три на брой, т. е. поне едно от тях се дели на 3, като това очевидно не е 2^k. Остава да бъде някое от 2^k-1 и 2^k+1. Тогава и произведението $(2^k-1)(2^k+1) = 2^{2k}-1$ ще се дели на 3. Т. е. броят на незапълнените квадратчета винаги се дели на 3 и потенциално би могъл да се запълни с достатъчно на брой шаблони, тъй като броят на квадратчетата, запълвани от шаблоните, се дели на 3. Това обаче е само *необходимо*, но не и *достатъчно* условие.

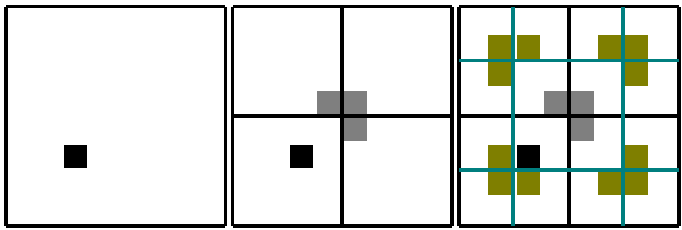
Ще покажем, че задачата винаги има решение за всяко $n = 2^k$, по метода на математическата индукция (по k).

1) *База:* При $k = 0$ имаме $n = 1$ и дъска с единствено запълнено специално квадратче, а при $k = 1$, т. е. $n = 2$, имаме единствен начин за поставяне на шаблона, при което получаваме правилно запълване.

2) *Индуктивно предположение:* Предполагаме, че твърдението е вярно за k, т. е. че винаги можем да запълним дъска с размери $2^k \times 2^k$, съдържаща едно запълнено квадратче.

3) *Индуктивна стъпка:* Ще покажем, че твърдението е вярно и за k+1, т. е. за дъска с размери $2^{k+1} \times 2^{k+1}$, съдържаща едно специално запълнено квадратче. Да разрежем дъската

хоризонтално и вертикално така, че да се получат четири еднакви поддъски. Всяка такава дъска има размери $2^k \times 2^k$, като една от тях съдържа специалното квадратче. Поставяме един шаблон в центъра на изходната дъска така, че всяка от дъските, несъдържаща специалното квадратче, да съдържа едно квадратче от шаблона. Сега всяка от четирите дъски съдържа точно по едно запълнено квадратче и има размери $2^k \times 2^k$. Но тогава по индукционното предположение всяка от тях може да се запълни с достатъчно шаблони, откъдето следва, че и изходната дъска може да се запълни (*виж фигура 7.11.*).



Фигура 7.11. Покриване с шаблон.

Оттук директно следва алгоритъм за решаване на задачата по метода *разделяй и владей*. Програмната реализация оставяме на читателя като леко упражнение.

Задача за упражнение:

Да се реализира алгоритъмът за запълване на област с шаблони.

7.12. Въпроси и задачи

7.12.1. Задачи от текста

Задача 7.1.

Да се реализира програма за намиране на k-ия по големина елемент (*виж 7.1.*), основана на сортиране чрез броене.

Задача 7.2.

Да се докаже, че при алгоритъма за намиране на k-ия по големина елемент, основан на разделяне, подобно на това при бързото сортиране, не са възможни други взаимни положения на променливите i, j и k, освен показаните на фигури *7.1а.*, *7.1б.* и *7.1в.*

Задача 7.3.

Да се сравнят двата предложени варианта на алгоритъма за намиране на k-ия по големина елемент, основани на разделяне, подобно на това при бързото сортиране (*виж 7.1.*).

Задача 7.4.

Следва ли да се очаква подобрение при разделяне на елементите в групи по 5 при намиране на медиана на медианите (*виж 7.1.*)? А по 3? Какъв е очакваният брой сравнения във всеки отделен случай? Да се определи оптималният брой елементи в група.

Задача 7.5.

Да се реализира алгоритъмът за намиране на медиана на медианите (*виж 7.1.*).

Задача 7.6.

Да се докаже, че, ако сортира масив, съдържащ мажорант, средният му елемент се заема от мажоранта (*виж 7.2.*).

Задача 7.7.

Да се докаже, че ако *x* е мажорант на даден масив и масивът се раздели на две почти равни части, то *x* е мажорант на поне едната от тях (*виж 7.2.*).

Задача 7.8.

Да се докаже, че ако масивът `m[]` има мажорант, то, ако елементите `m[i]` и `m[j]` са различни и бъдат премахнати, мажорантът на новия масив ще съвпада с мажоранта на стария (*виж 7.2.*).

Задача 7.9.

Да се докаже, че ако даден масив има мажорант, то, ако всички елементи в масива се срещат по двойки и ако се запази единият представител, като се изхвърли другият за всяка двойка, мажорантът на новия масив ще съвпада с мажоранта на стария (*виж 7.2.*).

Задача 7.10.

Да се предложи и реализира алгоритъм за сливане на две сортирани последователности от масив на място, т.е. без използване на допълнителен масив (*виж 7.3.*).

Задача 7.11.

Да се сравни скоростта на работа на предложените реализации за сортиране чрез сливане за: 100; 1000; 10000; 100000 елемента (*виж 7.4.*).

Задача 7.12.

Да се напише програма за сортиране чрез сливане (*виж 7.4.*), при която множеството се разбива на:

а) 3 подмножества
б) *k* подмножества, $k > 3$

Задача 7.13.

Да се пресметне сложността на различните варианти на сортиране чрез сливане (*виж 7.4.*):

а) в най-добрия случай
б) в най-лошия случай

Да се намерят най-добрата и най-лошата входна последователност.

Задача 7.14.

Да се реализира итеративен вариант на сортирането чрез сливане, използващ масиви вместо свързани списъци (*виж 7.4.*).

Задача 7.15.

Да се установи опитно броят елементи, под който при сортиране чрез сливане (*виж 7.4.*) следва да се използва сортиране чрез вмъкване.

Задача 7.16.

Да се сравни опитно скоростта на работа на сортирането чрез сливане (*виж 7.4.*), в комбинация с:

а) сортиране чрез вмъкване
б) метод на мехурчето
в) пряка селекция

Задача 7.17.

Да се реализира вариант на сортирането чрез сливане (*виж 7.4.*), който се възползва от евентуална естествена подредба на елементите в изходното множество (*виж 7.3.*).

Задача 7.18.

Да се предложи евристичен алгоритъм (*виж глава 9*), който винаги намира адитивна редица, не по-дълга от тази, давана от формулата от *7.5.*

Задача 7.19.

Да се установи опитно кога следва да се прекъсва рекурсията и да се прилага класическият алгоритъм за умножение на матрици (*виж 7.6.*).

Задача 7.20.

Да се провери предложената от Виноград последователност за бързо умножение на матрици (*виж 7.6.*).

Задача 7.21.

Да се предложи друга последователност за бързо умножение на матрици (*виж 7.6.*).

Задача 7.22.

Да се реализира предложеният алгоритъм за бързо умножение на дълги числа (*виж 7.7.*).

Задача 7.23.

Да се определи експериментално при колко цифри бързият алгоритъм за умножение на дълги числа е по-добър от класическия (*виж 7.7.*).

Задача 7.24.

Да се подобри алгоритъмът за бързо умножение на дълги числа, като за целта числата се разделят на три, вместо на две части. Да се покаже, че 5 (а не 9) умножения на числа с дължина $n/3$ са достатъчни за пресмятане на произведението. Да се определи сложността на получения алгоритъм (*виж 7.7.*).

Задача 7.25.

На базата на предходната задача да се покаже, че $2k-1$ умножения са достатъчни за пресмятане произведението в случай на разбиване на групи от по k последователни цифри. Да се покаже, че оттук следва съществуването на алгоритъм за умножение на дълги числа със сложност $\Theta(n^x)$ за всяко $x > 1$.

Задача 7.26.

Да се реализират *алгоритми 2*, *3* и *4* от *7.8.*

Задача 7.27.

Да се докаже, че *алгоритми 2*, *3* и *4* от *7.8.* извършват същите действия като *алгоритъм 1*.

Задача 7.28.

Да се преобразува конструираната таблица от *7.9.* в таблица от вида, искан в условието на задачата от Балканиадата.

Задача 7.29.

Да се докаже коректността на първия алгоритъм от *7.9.* за съставяне на разписание на турнир за n степен на 2.

Задача 7.30.

Да се реализира вторият алгоритъм от *7.9.* за съставяне на разписание на турнир, като за четни n се въвежда фиктивен отбор.

Задача 7.31.

Да се докаже коректността на втория алгоритъм от *7.9.* за съставяне на разписание на турнир за всяко естествено n, $n \geq 2$.

Задача 7.32.

Да се изведат приведените в *7.9.* формули за съставяне на разписанието на шахматен турнир.

Задача 7.33.

Да сравни теоретично времето за работа на функциите `shiftLeft1()`, `shiftLeft2()` и `shiftLeft3()` от *7.10*.

Задача 7.34.

Да се изведе формулата $BA = (A^R B^R)^R$ от *7.10*.

Задача 7.35.

Да се реализира алгоритъмът за запълване на област с шаблони от *7.11*.

7.12.2. Други задачи

Задача 7.36. *По-добро бързо сортиране*

Като се използва алгоритъмът за намиране на медиана на медианите (*виж 7.1.*), да се преработи бързото сортиране (*виж 3.1.6.*) така, че винаги да работи за време $\Theta(n.\log_2 n)$. Да се сравни теоретично и емпирично получената сложност с тази на пирамидалното сортиране (*виж 3.1.9.*).

Задача 7.37. *k-най малки елемента в масив*

Да се разработи алгоритъм за намиране на k-те най-малки числа в масив. Да се сравнят следните алгоритми:

- сортиране и извеждане на първите k числа
- намиране на k-ия по големина елемент x с линеен алгоритъм (*виж 7.1.*), разделяне на масива на ляв и десен дял относно x и цялостно сортиране на левия дял
- построяване на пирамида (приоритетна опашка) и k-кратно извличане на минималния елемент от върха й.

Задача 7.38. *Най-близка двойка точки*

Дадени са n точки в равнината, зададени с координатите си. Да се намери двойка точки, между които разстоянието е минимално.

Ще работи ли Вашият алгоритъм в тримерното пространство? А в n-мерното?

Задача 7.39. *Изпъкнала обвивка на множество точки*

Дадени са n точки в равнината, зададени с координатите си. Да се намери *изпъкнал* многоъгълник с минимално лице (минимално изпъкнало множество), който ги съдържа.

Ще работи ли Вашият алгоритъм в тримерното пространство? А в n-мерното?

Задача 7.40. *Други примери за разделяй и владей*

Вярно ли е, че изброените по-долу алгоритми се основават на *разделяй и владей*? Защо?

- бързо сортиране на Хоор (*виж 3.1.6.*)
- пирамидално сортиране (*виж 3.1.9.*), отчасти (*защо?*)
- побитово сортиране (*виж 3.2.3.*), рекурсивният вариант
- двоично търсене (*виж 4.3.*)
- Фибоначиево търсене (*виж 4.4.*)
- интерполационно търсене (*виж 4.5.*)
- алгоритъм на Шенън-Фано (*виж 10.4.1.*)
- алгоритъм на Хъфман (*виж 10.4.2.*)

Кои други разгледани алгоритми биха могли да се добавят?

Глава 8
Динамично оптимиране

"A mind that is stretched
to a new idea never returns
to its original dimension."

~ Oliver Wendell Holmes

8.1.Въведение

Често за решаването на една задача се оказва удачно тя да бъде разбита на по-малки подзадачи (от същия вид), които се решават по-лесно. По-малките подзадачи често са по-прости за решаване и дават възможност за съсредоточаване върху някои детайли и частни случаи, които не са така очевидни при изходната задача. Освен това подходящото разбиване може да даде съответно рекурсивно решение. Съществуват най-общо две ефективни алгоритмични схеми, основаващи се на разбиване на изходната задача на подзадачи: *динамично оптимиране* и *разделяй и владей*. Ще припомним, че в програмирането римският принцип "*Разделяй и владей*" най-често се свежда до разделяне на изходната задача на две подзадачи с еднакъв или приблизително еднакъв размер (*виж глава 7*). На практика няма никакви ограничения върху броя на подзадачите, както и върху размера им, като единственото изискване, което се поставя, е никои две подзадачи *да не се пресичат* и комбинирането на решенията на всички подзадачи да дава възможност за "лесно" получаване на решение на изходната задача. От своя страна динамичното оптимиране е свързано най-често с премахване на *единствен* елемент (възможно е и повече), последвано от решаване на няколко *задължително пресичащи се* подзадачи. По-долу въпросът ще бъде разгледан по-строго и ще бъдат приведени две необходими условия за прилагане на метода.

Става въпрос за проста програмистка техника, подобна на *разделяй и владей*: задачата се разбива на подзадачи, те от своя страна отново се разбиват на подзадачи и т.н. до достигане на достатъчно прости задачи, които могат да се решат директно. След това решенията на подзадачите се комбинират по подходящ начин, така че да се получи решение на изходната задача. За разлика от *разделяй и владей* обаче тук не се изисква множествата на подзадачите да са непресичащи се, което означава, че динамичното оптимиране е по-обща техника от *разделяй и владей*. Отказът от изискването за непресичане на подмножествата от задачи, характерно за динамичното оптимиране, обаче води до проблеми. В общия случай са налице полиномиален брой различни подзадачи на изходната, но в процеса на декомпозиция се оказва, че едни и същи задачи се решават няколко пъти, което в крайна сметка води до експоненциална сложност на алгоритъма. Типичен пример в това отношение е добре познатата ни функция за пресмятане на числата на Фибоначи (*виж 1.2.2.*):

```
unsigned long fib(unsigned n)
{ if (n < 2) return n;
  else return fib(n-1) + fib(n-2);
}
```

Какво може да се направи, за да се ускори функция, подобна на горната? Едно възможно решение е да се въведе таблица на всички вече пресметнати стойности. Всеки път, когато трябва да пресметнем `fib(n)` за някоя конкретна стойност на n, първо ще проверяваме дали вече не

сме я записали в таблицата и едва тогава, в случай че задачата все още не е била решавана, ще извършваме съответните пресмятания. Така достигаме до следната чисто "програмистка" дефиниция.

Дефиниция 8.1. Програмистката техника, при която се извършва запълване на таблица с резултатите от решенията на вече решени подзадачи с цел избягване на повторни пресмятания се нарича *динамично оптимиране*.

Макар принципът на динамичното оптимиране да е приложим и при задачи с единствено решение, като тази за намиране на n-тото число на Фибоначи (*виж 8.3.1.*), основното му приложение е решаването на оптимизационни задачи. Впрочем, фактът, че този метод може да се използва и за решаване на неоптимизационни задачи поставя под въпрос името му "оптимиране". В англоезичната литература се използва терминът *динамично програмиране* (англ. *dynamic programming*), който изглежда по-точен. Впрочем, той се използва и в българската и руската математическа литература. Ние обаче ще продължим да използваме *динамично оптимиране*, предвид на по-широката му популярност в българските информатически среди.

В класическия оптимизационен вариант става въпрос за задачи с множество *допустими кандидати за решения*, измежду които трябва да се направи оптимален избор. На всеки кандидат за решение се съпоставя някакво число с помощта на предварително дефинирана функция. Целта е да се намери решение, за което функцията приема своята екстремална (максимална или минимална) стойност. Прието е тази функция да се нарича *целева функция*. Възможно е задачата да има повече от едно оптимални решения, като в общия случай прилагането на динамично оптимиране ще ни даде само едно от тях.

Предимствата на динамичното оптимиране често се превъзнасят. Отсега обаче трябва да бъде ясно, че "безплатен обяд" няма (освен в някои софтуерни компании като *SAP* и *Microsoft*). В основата на динамичното оптимиране стои решаването на подзадачи на изходната задача с по-малък размер и запазване на вече пресметнатите резултати, т. е. печели се скорост за сметка на памет. В някои случаи е необходима константна памет (числа на Фибоначи, *виж 8.3.1*), докато в други случаи необходимата памет може да бъде линейна (задача за раницата, *виж 8.2.1.*), квадратична (задача за оптимално умножение на матрици, *виж 8.2.3.*), а понякога и още по-голяма.

Следва да се отбележи още, че динамичното оптимиране не винаги е приложимо. От една страна не винаги решението на изходната задача може да се получи чрез комбиниране на резултатите от решаването на част или всички нейни подзадачи. От друга страна, дори и когато такова комбиниране е възможно, може да се окаже, че броят на подзадачите, които следва да се разгледат, е недопустимо голям. Към това следва да се прибави и липсата на ясен критерий, характеризиращ задачите, които могат да бъдат решени с помощта на описания метод: оказва се, че за редица задачи, стандартните алгоритми се оказват далеч по-ефективни от динамичното оптимиране, а за други — методът изобщо не е приложим.

Съществуват две необходими условия за прилагане на метода: *оптимална подструктура на решението* и *припокриване на подзадачите*. Оптималната подструктура на решението означава, че оптималното решение на изходната задача може да бъде намерено като функция на *оптималните* решения на подзадачите. Обикновено изпълнението на този критерий води до намиране на подходящо "естествено" подпространство на подзадачите, които следва да бъдат разгледани. Така например, при решаването на задачата за оптимално умножение на матрици (*виж 8.2.3.*) ще видим, че е достатъчно да се разглеждат само подпоследователности на изходната последователност от матрици, а не произволни нейни подмножества. Това води до силно ограничаване на множеството на подзадачите, откъдето и до по-голяма ефективност на метода. Второто необходимо условие за прилагане на динамичното оптимиране е *припокриване на подзадачите*. Динамичното оптимиране умее да се възползва умело от припокриването на подзадачите, пресмятайки решението на всяка от тях *само веднъж*, силно ограничавайки по този начин действителния брой решавани подзадачи. Колкото по-рядко се налага действително решаване на нова подзадача, толкова по-ефективен е методът. Липсата на припокриване на подзадачите е сигурен индикатор, че прилагането на динамичното оптимиране е неподходящо. В такъв случай е

добре да се опита с метода *разделяй и владей*, който, в случай че е приложим, ще доведе до по-добри резултати. (*Защо?*)

Често обектите, които се разглеждат при решаване на дадена задача, представляват множества с частична или пълна линейна наредба на елементите. Примери за такива обекти са символните низове, различни комбинаторни конфигурации (пермутации, комбинации и др.), листата на наредени дървета за търсене, точки върху права, върхове на многоъгълник и др. В тези случаи обикновено динамичното оптимиране води до ефективен алгоритъм.

Обикновено при динамично оптимиране задачите се решават итеративно отдолу-нагоре, т. е. първо се разглеждат *тривиалните подзадачи* (тези, които се решават директно, без допълнително разбиване), след това тези по-големи подзадачи, за които всички подзадачи вече са решени и т.н. Основен недостатък на този подход е, че изисква решаване на *всички* подзадачи с размер, по-малък от този на изходната задача, което води до излишни пресмятания, тъй като в процеса на пресмятане може да се окаже, че някои от подзадачите не са нужни. По-долу ще разгледаме конкретни примери. Обикновено това не е толкова опасно и не води до съществено забавяне. Понякога обаче премахването на излишните пресмятания може да доведе до значително ускоряване. В такъв случай се прилага специален вариант на динамичното оптимиране, известен в англоезичната литература като *memoization*. Идеята е да се използва рекурсивна функция, при което да се обърне редът на пресмятане на решенията на подзадачите: *отгоре-надолу*. За целта се поддържа специална таблица с вече намерените решения на подзадачите. Вместо да пристъпи към директно решаване на някоя подзадача, рекурсивната функция първо търси решението ѝ в таблицата и, ако го намери, го взема наготово. Така, подзадачата се решава, само ако никога преди това не е била решавана, след което резултатът незабавно се попълва в таблицата. Решават се само тези подзадачи, които *реално* могат да участват в оптималното решение.

Понякога при реализиране на алгоритми, основаващи се на динамично оптимиране, рекурсията съзнателно се избягва. В действителност няма обективно основание за пренебрегването ѝ в полза на итерацията в този случай. Причините за "дискриминацията" се крият по-скоро в някои неосъзнати предразсъдъци, според които рекурсията традиционно се свързва с неефективни програмни схеми като търсене с връщане назад (*виж 6.3.*). Някои програмисти дори са склонни да поставят знак за равенство между рекурсия и неефективност. От друга страна се счита, че итеративната схема е в самата основа на динамичното оптимиране и крие някакъв особен чар. В действителност става въпрос за две различни техники: при итерацията решението се строи отдолу-нагоре, докато при рекурсията — отгоре-надолу. И двата подхода имат своите предимства и недостатъци. Обикновено рекурсивните решения са по-общи и по-естествени. За сметка на това изискват допълнителен булев идентификатор, удостоверяващ валидността на всеки елемент на таблицата на пресметнатите стойности на целевата функция, т. е. изискват допълнителна памет. (Това не е задължително и, както ще видим по-късно, лесно се заобикаля, като за целта се използва някаква специална стойност, указваща, че подзадачата не е била решавана. Това, разбира се, ограничава с една възможните полезни стойности на елементите в таблицата, но пести памет.) Впрочем рекурсивните обръщения също изискват памет за запазване на локалните променливи, параметрите на рекурсивната функция, както и адреса на връщане. Това води и до известно забавяне на изпълнението. От друга страна рекурсивният вариант обикновено е по-прост и пресмята *само* тези стойности на целевата функция, които *действително са нужни* в процеса на пресмятането на търсената стойност. Итеративният вариант — напротив, пресмята *всички* последователни стойности на целевата функция до достигане на търсената. В този смисъл нерядко рекурсивното решение може да се окаже по-ефективно.

Динамичното оптимиране често се подценява, или по-точно — недооценява. Сред някои програмисти битува мнението, че то е сложно и объркано. В действителност става въпрос за една от най-простите и същевременно ефективни алгоритмични техники. След като веднъж е разбрал основните му принципи, за читателя ще бъде по-лесно съставянето на собствени схеми, отколкото откриването им в литературата. Тъй като подобни знания се усвояват най-лесно на базата на конкретни примери, ще се постараем да разгледаме максимален брой класически, както и някои по-малко известни задачи.

Задачи за упражнение:

1. Задължително ли е подзадачите да се припокриват, за да може да се прилага динамично оптимиране?
2. Да се сравнят търсенето с връщане, "разделяй и владей" и динамичното оптимиране.
3. Само за оптимизационни задачи ли може да се прилага динамично оптимиране?

8.2.Класически оптимизационни задачи

8.2.1. Задача за раницата

Ще започнем нашето пътешествие в света на динамичното оптимиране с една от най-известните задачи, решавана ефективно с помощта на метода. Задачата за раницата се появява в литературата в множество различни формулировки, предполагащи различни решения (задачата вече беше разгледана в *6.4.1* и ще се върнем отново на нея в *параграф 9.1.5.*). В най-популярния си вариант (0-1 задача за раницата) тя е класически пример за задача, която се решава с динамично оптимиране. Да припомним нейното условие.

Дадена е раница с вместимост M килограма и N предмета, за всеки от които са дадени две тегло m_i и стойност c_i. Да се избере множество от предмети, чиято сумарна стойност е максимална, а сумата от теглата им не надвишава M. Числата M, N, c_i и m_i са естествени ($1 \leq i \leq N$).

Задачата се свежда до намиране на максимума на сумата:

$$\sum_{i=1}^{N} x_i c_i$$

при ограничителните условия:

$$\sum_{i=1}^{N} x_i m_i \leq M,$$

$$c_i > 0, m_i > 0, x_i \in \{0,1\}, i = 1,2,\ldots,n$$

Ограниченията върху c_i и m_i са ограничения върху *условието* на задачата (върху *конкретния екземпляр*), докато тези върху x_i следва да се възприемат като ограничения върху *търсеното решение*.

Решение: Ще дефинираме рекурентна целева функция $F(i)$, представляваща решение за раница с вместимост i. Тогава

$$F(i) = \begin{cases} 0 & i = 0 \\ \max\limits_{j=1,2,\ldots,N;\, m_j \leq i} \left[c_j + F(i - m_j) \right] & i > 0 \end{cases}$$

Бихме могли, изхождайки от горната рекурентна формула, да реализираме рекурсивна функция за пресмятане на F. Ще отбележим, че така можем да избираме един и същ предмет няколко пъти. Ето защо по-долу за всяко $F(i)$ ще поддържаме множество `set[i]`, съдържащо конкретните предмети, чието вземане води до тази максимална стойност. Освен че ще ни позволи да намерим едно конкретно множество от предмети, формиращо максималната стойност на целевата функция, `set[i]` ще ни предпази от повторното включване на един и същ предмет.

Един възможен подход е стойността на функцията да се пресмята рекурсивно, а за избягване на повторни изчисления да се използва таблица (*memoization*). Таблицата на вече пресметнатите стойности ще съдържа стойността на целевата функция, ако вече е била пресмятана или специалната стойност `NOT_CALCULATED` — в противен случай. Всеки път, когато ни потрябва стойността на $F(i)$ за някое i = 1, 2, …, N, първо ще проверяваме дали `Fn[i]` (запазената таблична стойност) е различно от `NOT_CALCULATED`, в който случай ще вземаме

стойността `Fn[i]` от таблицата. В противен случай ще извършваме рекурсивно обръщение към `F()`. Входните данни на програмата ще заложим в кода ѝ като константи. Основната работа се извършва от рекурсивната функция `F()`, която пресмята стойностите на целевата функция. Обръщението към нея не е пряко, а посредством функцията `calculate()`, която проверява дали вместимостта на раницата позволява вземането на всички предмети и, едва ако това не е така, извършва обръщение към `F()`. Останалите подробности по реализацията читателят може да види от предложения програмен код:

```
#include <string.h>
#include <stdio.h>
#define NOT_CALCULATED  (unsigned) (-1)
#define MAXN            30  /* Максимален брой предмети */
#define MAXM          1000  /* Максимална вместимост на раницата */

char set[MAXM][MAXN];       /* Множество от предмети за k = 1..M */
unsigned Fn[MAXM];          /* Целева функция */

const unsigned m[MAXN] = {0,30,15,50,10,20,40,5,65};    /* Тегла */
const unsigned c[MAXN] = {0,5,3,9,1,2,7,1,12};      /* Стойности */
const unsigned M = 70;            /* Обща вместимост на раницата */
const unsigned N = 8;                             /* Брой предмети */

/* Пресмята стойността на функцията за k */
void F(unsigned k)
{ unsigned i, bestI, fnBest, fnCur;
  /* Пресмятане на най-голямата стойност на F */
  for (bestI = fnBest = 0, i = 1; i <= N; i++) {
    if (k >= m[i]) {
      if (NOT_CALCULATED == Fn[k - m[i]]) F(k - m[i]);
      if (!set[k - m[i]][i])
        fnCur = c[i] + Fn[k - m[i]];
      else
        fnCur = 0;
      if (fnCur > fnBest) {
        bestI = i;
        fnBest = fnCur;
      }
    }
  }

  /* Регистриране на най-голямата стойност на функцията */
  Fn[k] = fnBest;
  if (bestI > 0) {
    memcpy(set[k], set[k - m[bestI]], N);
    set[k][bestI] = 1;
  }
}

/* Пресмятане на стойността на целевата функция */
void calculate(void) {
  unsigned i, sumM;

  /* Инициализиране */
  memset(set,0,sizeof(set)); /* Иниц. на множествата с предмети */
  for (i = 0; i <= M; i++)  /* Иниц. на целевата функция */
    Fn[i] = NOT_CALCULATED;

  /* Дали не можем да вземем всички предмети? */
  for (sumM = m[1], i = 2; i <= N; i++) sumM += m[i];
```

```
    if (M >= sumM) {
      printf("\nМожете да вземете всички предмети!");
      return;
    }
    else {
      F(M); /* Пресмятане на стойността */

      /* Отпечатване на резултата */
      printf("\nВземете предметите с номера:\n");
      for (i = 1; i <= N; i++)
        if (set[M][i])
          printf("%5u", i);
      printf("\nМаксимална постигната стойност: %u", Fn[M]);
    }
  }
  int main(void) {
    printf("%s%u", "\nБрой предмети: ", N);
    printf("%s%u", "\nВместимост на раницата: ", M);
    calculate();
    return 0;
  }
```

🖫 knapsack 1.c

Резултат от изпълнението на програмата:

```
Вземете предметите с номера:
    2    3    7
Максимална постигната стойност: 13
```

Забележка: В настоящия параграф (с малки изключения) масивите, които използваме, са индексирани от 1 (тъй като при някои от реализациите нулевата стойност се използва за служебни пресмятания). Препоръчваме на читателя да бъде предпазлив в случаите, когато N (съответно M) имат стойности, близки до дефинираните `MAXN` и `MAXM`.

Вижда се, че намереното множество от предмети не е единствено. Така например, бихме могли да вземем предмети с номера 7 и 8, постигайки същата стойност на целевата функция, без при това да нарушаваме ограничителното условие, наложено от издръжливостта на раницата.

Бихме могли да избегнем рекурсията чрез последователно пресмятане на стойностите на $F(1)$, $F(2)$, ..., $F(M)$, без да навлизаме в дълбочина. Действително, рекурентната формула извършва обръщения само назад, т. е. само към стойности, които вече са били пресметнати. Така достигаме до естествено итеративно решение. Тук ролята на рекурсивната функция `F()` се поема от таблицата `Fn[]` и всички пресмятания се реализират направо във функцията `calculate()`.

```
void calculate(void)
{ unsigned maxValue;          /* Максимална постигната стойност */
  unsigned maxIndex;          /* Индекс, за който е постигната */
  unsigned i, j;
  memset(set,0,sizeof(set)); /* Иниц. на множествата предмети */

  /* Пресмятане на стойностите на целевата функция */
  for (i = 1; i <= M; i++) { /* Търсим макс. стойност на Fn(i) */
    maxValue = maxIndex = 0;
    for (j = 1; j <= N; j++)
      if (m[j] <= i && !set[i - m[j]][j])
        if (c[j] + Fn[i - m[j]] > maxValue) {
          maxValue = c[j] + Fn[i - m[j]];
          maxIndex = j;
        }
```

```
    if (maxIndex > 0) { /* Има ли предмет с тегло по-малко от i? */
      Fn[i] = maxValue;
      /* Новото множество set[i] се получава от set[i-m[maxIndex]]
       * чрез добавяне на елемента maxIndex */
      memcpy(set[i], set[i - m[maxIndex]], N);
      set[i][maxIndex] = 1;
    }

    if (Fn[i] < Fn[i - 1]) { /* Побират се всички предмети и още */
      Fn[i] = Fn[i - 1];
      memcpy(set[i], set[i - 1], N);
    }
  }

  /* Извеждане на резултата */
  printf("\nВземете предметите с номера:\n");
  for (i = 1; i <= N; i++)
    if (set[M][i])
      printf("%5u", i);
  printf("\n%s%u", "Максимална постигната стойност: ", Fn[M]);
}
```
🖫 knapsack2a.c

Вторият, итеративен вариант на функцията `calculate()`, позволи съкращаване на необходимата памет (но не и асимптотично), правейки излишна специалната стойност, както и изобщо предварителната инициализация на таблицата на вече пресметнатите стойности. Бихме могли да намалим допълнително заеманата памет при по-прецизна реализация на множеството `set[]`. Там за присъствието на всеки елемент се отделяше цял байт (тип `char`). Същевременно за съхраняване на булева стойност е достатъчен един бит. Това означава, че паметта, заемана от множеството, може да бъде съкратена 8 пъти. За целта се налага използване на съответни побитови или аритметични операции, позволяващи извличане или установяване състоянието на отделен бит. Ще се спрем на побитови операции, поради по-голямата им ефективност. По-принцип въвеждането им следва да забави алгоритъма, макар и незначително. От друга страна операциите по инициализиране и копиране на множества (`memset()` и `memcpy()`) ще работят с 8 пъти по-малко елементи, т. е. ще бъдат по-бързи.

В какво се състои промяната? При проверката дали предметът `j` участва в множеството, съответстващо на максималната стойност на *F*(*i*), по-горе сравнявахме `set[i][j]` с 1. Сега ще трябва да проверяваме дали е установен в 1 битът `j%8` на `set[i][j]`. Това може да стане с помощта на израза `set[i][j/8]&(1 << (j%8))`. Основен недостатък тук са тежките аритметични операции / (деление) и % (остатък по модул). За запознатите с особеностите на побитовите операции е ясно, че те лесно могат да бъдат заменени. Така например, делението на 8 е еквивалентно на побитово изместване надясно на 3 позиции, а остатъкът по модул 8 може да бъде получен чрез поразредно логическо "и" с числото 7. Получаваме `set[i][j>>3]&(1<<(j&7))`. Аналогично, установяването на `m[i][j]` в 1 става с `set[i][j>>3] |= 1 << (j & 7)`. Сега копирането на множества се извършва върху 8 пъти по-малко елементи. След нанасяне на съответните промени получаваме:

```
char set[MAXM][MAXN / 8]; /* Множество от предмети за k = 1..M */

void calculate(void)
{ unsigned maxValue;          /* Максимална постигната стойност */
  unsigned maxIndex;          /* Индекс, за който е постигната */
  unsigned i, j;
  memset(set,0,sizeof(set)); /* Иниц. множествата с предмети */
```

```
    /* Пресмятаме стойностите на целевата функция */
    for (i = 1; i <= M; i++) { /* Търсене макс. стойност на F(i) */
      maxValue = maxIndex = 0;
      for (j = 1; j <= N; j++) {
        if (m[j] <= i && !(set[i - m[j]][j >> 3] & (1 << (j & 7))))
          if (c[j] + Fn[i - m[j]] > maxValue) {
            maxValue = c[j] + Fn[i - m[j]];
            maxIndex = j;
          }
      }
      if (maxIndex > 0) { /* Има ли предмет с тегло по-малко от i? */
        Fn[i] = maxValue;

        /* Новото множество set[i] се получава от set[i-m[maxIndex]]
         * чрез прибавяне на елемента maxIndex */
        memcpy(set[i], set[i - m[maxIndex]], (N >> 3) + 1);
        set[i][maxIndex >> 3] |= 1 << (maxIndex & 7);
      }
      if (Fn[i] < Fn[i - 1]) { /* Побират се всички предмети и още */
        Fn[i] = Fn[i - 1];
        memcpy(set[i], set[i - 1], (N >> 3) + 1);
      }
    }

    /* Извеждане на резултата */
    printf("\nВземете предметите с номера:\n");
    for (i = 1; i <= N; i++)
      if (set[M][i >> 3] & (1 << (i & 7)))
        printf("%5u", i);
    printf("\n%s%u", "Максимална постигната стойност: ", Fn[M]);
  }
```

🖫 knapsack2b.c

Често при решаването на задачи с помощта на динамично оптимиране е възможно да съществува повече от една схема. Задачата за раницата е добър пример в това отношение. Да изоставим рекурентната формула от началото на параграфа и да се опитаме да намерим друга — двуаргументна. Този път ще използваме двумерна таблица `F[i][j]` ($1 \leq i < N$, $0 \leq j < M$), съдържаща по един ред за всеки предмет и по една колона — за всяка възможна стойност на вместимостта на раницата M. `F[i][j]` ще съдържа максималната стойност, която може да бъде постигната при положение, че е позволено вземането само на първите i предмета, а вместимостта на раницата е j. Очевидно в такъв случай решението на изходната задача ще се съдържа в `F[N][M]`. Таблицата ще попълваме по формулата:

$$F[i][j] = \max(F[i-1][j],\ F[i-1][j-m_i]+c_i)$$

Ще се опитаме да обосновем предложената формула. Смисълът ѝ най-общо е следният: На всяка стъпка, при пресмятането на `F[i][j]`, можем да не включим или да включим предмета i. В първия случай `F[i][j]` ще съвпадне с `F[i-1][j]`, а при втория — с $F[i-1][j-m_i]+c_i$.

Попълването може да стане както по редове, така и по колони. Ще наложим естественото гранично условие `F[0][j] = 0`, за $j = 0, 1, ..., M$. Така получаваме следната функция:

```
unsigned F[MAXN][MAXM]; /* Целева функция */

void calculate(void) /* Пресмята стойностите на целевата функция */
{ unsigned i,j;
  for (j = 0; j <= M; j++)
      F[0][j] = 0;
```

```
    for (i = 1; i <= N; i++)
      for (j = 0; j <= M; j++)
        if (j >= m[i] && F[i-1][j] < F[i-1][j-m[i]] + c[i])
            F[i][j] = F[i-1][j-m[i]] + c[i];
          else
            F[i][j] = F[i-1][j];
}
```
🖫 knapsack3a.c

Да разгледаме един конкретен пример. Да предположим, че е дадена раница с вместимост 15 килограма и 6 предмета, чиято стойност и тегло могат да се видят от следващите дефиниции (Нулевите елементи на масивите не се ползват!):

```
const unsigned m[MAXN] = {0,1,2,3,5,6,7};            /* Тегла */
const unsigned c[MAXN] = {0,1,10,19,22,25,30}; /* Стойности */
const unsigned M = 15;         /* Обща вместимост на раницата */
const unsigned N = 6;                         /* Брой предмети */
```
🖫 knapsack3a.c

Да изведем на екрана таблицата `F[][]`, получена в резултат на работата на функцията `calculate()`. За целта можем да реализираме проста функция `printTable()`:

```
void printTable(void) /* Извежда съдържанието на таблицата F[][] */
{ unsigned i, j;
  for (i = 1; i <= N; i++) {
    printf("\n");
    for (j = 0; j <= M; j++)
      printf("%4u",F[i][j]);
  }
}
```
🖫 knapsack3a.c

N\M	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
1	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	**0**	1	10	~~11~~	11	11	11	11	11	11	11	11	11	11	11	11
3	0	1	10	**19**	20	29	30	30	~~30~~	30	30	30	30	30	30	30
4	0	1	10	19	20	29	30	32	**41**	42	51	52	52	52	52	52
5	0	1	10	19	20	29	30	32	**41**	44	51	54	55	57	66	~~67~~
6	0	1	10	19	20	29	30	32	41	44	51	54	59	60	66	**71**

Таблица 8.2.1а. Стойности на целевата функция за входните данни от *knapsack3a.c*.

Таблица 8.2.1а. съдържа стойностите на целевата функция. Числото долу вдясно е търсената максимална стойност на целевата функция. Използвайки горната таблица, лесно можем да намерим едно примерно множество от предмети. В нашия пример `F[6][15]` ≠ `F[5][15]`, но `F[6][15] == F[5][15–`m_6`] + `c_6, откъдето следва, че предметът с номер 6 задължително участва в множеството. По-нататък `F[5][8] == F[4][8]` и `F[5][8]` ≠ `F[4][8–`m_5`] + `c_5, което означава, че предметът с пореден номер 5 не може да участва в множеството. Сега разглеждаме `F[4][8]`. Имаме `F[4][8]` ≠ `F[3][8]`, но `F[4][8] == F[3][8–`m_4`] + `c_4, т. е. предметът 4 участва в множеството. Сега `F[3][3]` ≠ `F[2][3]` и `F[3][3] == F[2][3–`m_3`] + `c_3 т.е. предметът 3 участва в множеството. Достигнахме до елемента `F[2][0]`. Тъй като вторият аргумент е 0, следва да прекратим търсенето. (Ако продължим, ще получим `F[2][0] = F[1][0] = F[0][0]`).

Реализирането на описаната функция не представлява проблем:

```
void printSet(void) /* Извежда възможно множество от предмети, */
```

```
{                    /* за което се постига максимална стойност */
                     /* на целевата функция */
   unsigned i = N,
            j = M;
   while (j != 0) {
     if (F[i][j] == F[i-1][j])
       i--;
     else {
       printf("%u ",i);
       j -= m[i];
       i--;
     }
   }
}
```

🖫 knapsack3a.c

Забележка: Показаният фрагмент не разглежда случая, когато задачата няма решение. Предоставяме на читателя да модифицира по подходящ начин реализацията по-горе.

И така, търсеното множество включва предметите с номера 3, 4 и 6. В този случай то е единствено. Ще припомним, че е възможно оптималната стойност да се получава по няколко различни начина. В нашия пример това би могло да се получи например, ако прибавим още един предмет със същото тегло и стойност като предмет 6.

Макар сложността на алгоритъма по време и памет при двете схеми на пресмятане да е сходна, двумерната таблица има някои предимства. От една страна, не се налага тежката операция по преместване на цели области от паметта, при копиране на множества. По-същественото обаче е, че позволява намиране не само на едно-единствено, а на *всички* решения. Това свойство е уникално за тази схема и нито един друг алгоритъм измежду разгледаните в настоящия параграф не позволява получаване на всевъзможните оптимални множества. Да разгледаме като пример следните входни данни:

```
const unsigned m[MAXN] = {0,6,3,10,2,4,8,1,13,3};      /* Тегла */
const unsigned c[MAXN] = {0,5,3,9,1,2,7,1,12,3};       /* Стойности */
const unsigned M = 14;              /* Обща вместимост на раницата */
const unsigned N = 9;                          /* Брой предмети */
```

🖫 knapsack3b.c

Максималната стойност на целевата функция е 13 и се постига за четири различни множества от предмети, а именно: {2, 3, 7}, {2, 6, 9}, {3, 7, 9} и {7, 8}. Как да ги намерим? Да разгледаме *таблица 8.2.1б.*, съответстваща на горните входни данни.

N\M	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
1	0	0	0	0	0	0	5	5	5	5	5	5	5	5	5
2	0	0	0	3	3	3	5	5	5	8	8	8	8	8	8
3	0	0	0	3	3	3	5	5	5	8	9	9	9	12	12
4	0	0	1	3	3	4	5	5	6	8	9	9	10	12	12
5	0	0	1	3	3	4	5	5	6	8	9	9	10	12	12
6	0	0	1	3	3	4	5	5	7	8	9	10	10	12	12
7	0	1	1	3	4	4	5	6	7	8	9	10	11	12	13
8	0	1	1	3	4	4	5	6	7	8	9	**10**	11	12	**13**
9	0	1	1	3	4	4	6	7	7	8	9	10	11	12	**13**

Таблица 8.2.1б. Стойности на целевата функция за входните данни от *knapsack3b.c*.

На първата стъпка `F[9][14] == F[8][14]` и `F[9][14] == F[8][14-m₉] + c₉`. Това означава, че предметът 9 би могъл както да участва, така и да не участва в оптималното множество. Предложената по-горе функция `printSet()` в такъв случай предпочиташе да смята, че

предметът не принадлежи на множеството и продължаваше в посока `F[8][14]`. Ясно е, че ако искаме да намерим всички решения, в такъв момент трябва да тръгнем и в двете посоки. За целта ще използваме рекурсия и списък (стек) `set[]`, в който ще пазим текущото множество от предмети. Функцията ще има три параметъра: *i*, *j* и *k*. Числата *i* и *j* имат смисъл на текущи координати в таблицата `F[][]`, а *k* — на индекс в `set[]`.

```
unsigned set[MAXN]; /* Множество от предмети, постигащи max */

void printAll(unsigned i, unsigned j, unsigned k)
{  /* Извежда ВСИЧКИ възможни множества от предмети, за които */
   /* се постига максимална стойност на целевата функция      */

   if (0 == j) {
     printf("\nВземете следните предмети: ");
     for (i = 0; i < k; i++)
       printf("%u ",set[i]);
   }
   else {
     if (F[i][j] == F[i-1][j])
       printAll(i-1,j,k);
     if (j >= m[i] && F[i][j] == F[i-1][j-m[i]] + c[i]) {
       set[k] = i;
       printAll(i-1,j-m[i],k+1);
     }
   }
}

/* ... */
printAll(N,M,0);
/* ... */
```
🖫 knapsack3b.c

Функцията `printAll()` намира следните решения за данните от *таблица 8.2.1б.*: {7, 3, 2}, {8, 7}, {9, 6, 2} и {9, 7, 3}. изписали сме ги в намаляващ ред, защото в такъв ред ги намира функцията. Оставяме на читателя да се убеди, че всички те и само те са решения в този случай.

Ако се вгледаме по-внимателно във формулата, по която пресмятаме стойността на целевата функция `F[i][j] = max(F[i-1][j],F[i-1][j-mi]+ci)`, лесно можем да забележим, че необходимата памет може да се съкрати значително. Действително, едната размерност на двумерния масив `F[][]` е напълно излишна, тъй като при пресмятане на стойностите за *i*-ия ред `F[i][.]` се извършват обръщения само към предходния ред `F[i-1][.]`. Останалите редове не се използват и биха могли да не се пазят. За целта освен масива `F[]` (този път едномерен) ще въведем и втори — `OldF[]`. Тогава масивът `F[]` ще съответства на *i*-ия ред на `F[i][.]`, а `OldF[]` — на (*i*–1)-ия ред на `F[i-1][.]`.

```
unsigned calculate(void) /* Пресмята стойността на функкията */
{ unsigned F[MAXM], OldF[MAXM]; /* Целева функция */
  unsigned i,j,k;

  for (j = 0; j <= M; j++)
      OldF[j] = 0;
  for (i = 1; i <= N; i++) {
     for (j = 0; j <= M; j++)
        if (j >= m[i] && OldF[j] < OldF[j-m[i]] + c[i])
           F[j] = OldF[j-m[i]] + c[i];
        else F[j] = OldF[j];
     for (k = 0; k < M; k++)
        OldF[k] = F[k];
```

```
  }
  return F[M];
}
```
🖫 knapsack3c.c

Направената модификация води до значително съкращаване на необходимата памет — от $\Theta(N.M)$ на $\Theta(N+M)$. За сметка на това обаче се оказва *невъзможно* директното възстановяване на едно *конкретно множество* от предмети. В случай, че се интересуваме само от стойността на целевата функция $F(k)$ за някакво конкретно k, но не и от множеството от предмети, за които се постига, горната реализация напълно ни удовлетворява и е за предпочитане, тъй като намалената памет ще ни позволи да пресметнем стойността на целевата функция за по-големи стойности на k.

Забележете, че при първата формула, която използвахме, не е възможно така лесно да се освободим от изискването за всяка стойност на целевата функция да пазим съответното множество от предмети, за които се постига. Начинът, по който извършвахме там пресмятанията, изискваше *непременно* запазване на множеството, за да се предпазим от повторното включване на един и същ предмет в раницата.

Анализът на предложените по-горе решения не представлява проблем. Необходимата памет при всички е $\Theta(N.M)$. Изключение прави само *knapsack3c.c*, който обаче не умее да намира конкретно множество от предмети. По същия начин се оценява и алгоритмичната им сложност — $\Theta(N.M)$. Това на пръв поглед може да не изглежда така очевидно за рекурсивното решение (*knapsack1.c*). Не е трудно обаче да се убедим, че сложността е именно тази, припомняйки си, че всяка стойност на целевата функция се пресмята от функцията `F()` най-много *веднъж*. Намирането на едно конкретно множество от предмети става за време $\Theta(N)$, за *knapsack1.c*, *knapsack2a.c* и *knapsack2b.c*, и $\Theta(N+M)$ — за *knapsack3a.c*. Намирането на всички възможни множества, извършвано от *knapsack3b.c*, изисква време $\Theta(N+M+s)$, където s е броят на оптималните множества от предмети.

Както споменахме в началото на параграфа, задачата за раницата може да се появи в най-разнообразни модификации. Така например, е възможно да са дадени *не конкретни* предмети, а *типове* предмети. Тогава всеки тип ще се характеризира с още едно число: $\#i$, представляващо брой налични предмети от съответния тип. Решението на задачата в този случай не се различава съществено от решението ѝ в разгледания по-горе случай. Действително, ако гледаме на предметите от всеки тип като на $\#i$ на брой различни отделни предмета, то получаваме изходната задача.

Сложността на решението на задачата в този случай се запазва. Възможно е обаче задачата да се формулира по такъв начин, че решението ѝ да се опрости. Така например, ако гледаме на предметите като на *типове предмети*, от всеки от които са налични достатъчно много (неограничен брой) предмети, то получаваме по-проста задача. Действително, в такъв случай не се налага да се тревожим за това колко пъти вземаме даден предмет и е в сила първата формула. Използвайки итеративната схема с едномерна таблица, получаваме следното решение:

```
unsigned F[MAXM];    /* Целева функция */
unsigned best[MAXM]; /* Последен нов предмет за максимума */

void calculate(void) /* Пресмята стойностите на целевата функция */
{ unsigned i,j;
  /* Инициализиране */
  for (i = 0; i <= M; i++)
       F[i] = 0;

  /* Основен цикъл */
  for (j = 1; j <= N; j++)
    for (i = 1; i <= M; i++)
      if (i >= m[j])
        if (F[i] < F[i-m[j]] + c[j]) {
          F[i] = F[i-m[j]] + c[j];
```

```
            best[i] = j;
          }
}
```

🖫 knapsack4.c

В началото алгоритъмът опитва да запълни цялата раница само с предмети от тип 1. След това — с използването на предмети от типове 1 и 2, и. т. н. Накрая опитва да използва всички налични типове предмети. За входните данни на първата програма, горната функция ще намери по-добро решение — със стойност 14:

```
Максимална постигната стойност: 14
Вземете следните предмети:    7    7    2    2    2    2
```

Това не бива да ни учудва, тъй като условието на задачата е различно и входните данни имат друг смисъл, а именно на *категории* предмети, от всяка от които има в наличност неограничен брой. Това се вижда и от предложеното от програмата множество от предмети: два от тип 7 и четири — от тип 2. Очевидно решението не е единствено. Стойност 14 можем да получим и вземайки 14 предмета от тип 7.

Нещата силно се опростяват от това, че не се налага да се поддържат множества, което спестява памет от порядъка на $\Theta(M.N)$, изразходвана за множеството `set[][]` в предходните реализации. Вместо това масивът `best[]` съхранява последния добавен предмет за достигане на максимума. На практика той не е необходим за пресмятането на максималната стойност на целевата функция. Заедно с масива `F[]` обаче `best[]` позволява да бъде получено едно възможно множество от предмети:

```
void printSet(void)  /* Извежда възможно множество от предмети, */
                     /* за което се постига максимална стойност */
                     /* на целевата функция */
{  unsigned value = M;
   printf("\nВземете следните предмети: ");
   while (value) {
     printf("%4u ", best[value]);
     value -= m[best[value]];
   }
}
```

🖫 knapsack4.c

Лесно се вижда, че сложността и в този случай е $\Theta(N.M)$. За разлика от предишните реализации обаче тук допълнителната памет е $\Theta(M)$. В действителност необходимата памет е $\Theta(N+M)$, тъй като това е паметта, необходима за съхраняване на числата c_i и m_i.

Бихме могли да използваме същата техника (освобождаване от множеството `set[][]` и въвеждане на масива `best[]`) за реализиране на по-ефективен по отношение на памет алгоритъм за решаване на изходната задача за раницата. За целта в началото ще инициализираме масива `best[]` с 0. Очевидно нулата е безсмислена като стойност на индекс в множеството на предметите (номерирането започва от 1), поради което е подходяща за използване като индикатор за това дали стойността на `best[]` е валидна.

```
char used(unsigned i, unsigned j)
{ /* Проверява дали j участва в множеството, съответно на F[i] */
   while (i != 0 && best[i] != 0)
     if (best[i] == j)
       return 1;
     else
      i -= m[best[i]];
   return 0;
}
```

```
void calculate() /* Пресмята стойностите на целевата функция */
{ unsigned i,j;

  /* Инициализиране */
  for (i = 0; i <= M; i++)
       best[i] = 0;

  /* Основен цикъл */
  for (i = 1; i <= M; i++)
    for (j = 1; j <= N; j++)
      if (i >= m[j])
        if (F[i] < F[i-m[j]] + c[j])
          if (!used(i-m[j],j)) {
            F[i] = F[i-m[j]] + c[j];
            best[i] = j;
          }
}
```

🖫 knapsack5.c

Задачата за раницата дава възможност да се видят някои особености на динамичното оптимиране въобще. Предложените по-горе реализации се държат добре, при сравнително малки стойности на M. При по-големи стойности обаче времето за работа може да нарасне значително. В такъв случай особено ярко проличават предимствата на рекурсивния вариант, който беше разгледан пръв. Действително, той пресмята само тези стойности на функцията, които наистина му трябват. За да разбере читателят какво означава това, може да опита да подаде на горните програми входни данни, съдържащи достатъчно малък брой предмети и същевременно големи стойности на M и m_i (стойностите на c_i не са от значение). В такъв случай може да се окаже, че рекурсивният вариант ще пресметне едва десетина стойности на функцията — само тези, които действително са му *нужни*, докато итеративният вариант ще намери последователно *всичките* няколко хиляди. Проблемът с необходимото количество памет обаче остава. Едно възможно решение е за съхранението на пресметнатите стойности на функцията да се използва *хеш-таблица* (*виж 2.5.*): така се запазва практически константна скорост на достъп (почти директен) при силно намалена памет.

Друга характерна особеност на динамичното оптимиране е неговият "дискретен характер". Ограничението, което наложихме в началото, а именно M, N, c_i и m_i да бъдат естествени, е изключително съществено. Не съществува ефективен начин за решаване на задачата при положение, че числата M и m_i са реални. При това не само с помощта на динамично оптимиране, а изобщо. На пръв поглед нещата може да не се сторят толкова страшни на читателя. В такъв случай ще му предложим да се опита да реши задачата за: $c_i = 1$, $m_i = \sqrt{i}$ и $M = N/2$.

Задачи за упражнение:

1. Да се покаже, че предложените формули и програмни реализации водят до правилно решение на задачата.
2. Да се преформулира първата целева функция така, че да не допуска повторение на елементи (първа предложена реализация).
3. Необходимо ли е множеството `set[]` в първата реализация? Възможно ли е да се гарантира неповтаряне на предмети по друг начин?
4. Работят ли правилно предложените реализации в случай, когато задачата няма решение? Да се нанесат съответни промени в програмите, в които това е нужно.
5. Нека оптималното решение се състои от k предмета с еднакви тегла и стойности. Колко пъти ще се генерира това решение при всеки от предложените алгоритми?
6. Да се реши задачата за раницата за $c_i = 1$, $m_i = \sqrt{i}$ и $M = N/2$.

8.2.2. Братска подялба

Пряко свързана със задачата за раницата е следната задача:

Братята Алан и Боб искат да си поделят комплект от n подаръка. Всеки от подаръците трябва да се даде или на Алан, или на Боб, като нито един от подаръците не може да се раздели на две. Всеки подарък има стойност цяло положително число. Нека a и b означават сумата подаръци, получени съответно от Алан и Боб. Целта е да се минимизира абсолютната стойност на разликата $a–b$. Напишете програма, която изчислява стойностите на a и b.

Решение:

Нека с p означим общата сума на стойностите на всички налични подаръци, а с S — множеството от всевъзможните стойности на сумите при избор на някои от предметите. Тогава Алан ще вземе подаръци на стойност a — най-големият елемент (сума) от S, ненадвишаващ $p/2$, а Боб — подаръци на стойност $b = p–a$.

Как да намерим елементите на множеството S? Ще предложим един елементарен алгоритъм. Първоначално S ще бъде празно. Да означим с m_k стойността на k-ия подарък. Вземаме първия предмет и включваме в множеството S стойността m_1. След това вземаме втория предмет и включваме в S числата m_2 и m_2+m_1. Така S ще съдържа всевъзможните суми на стойностите на първите два предмета. Преминаваме към третия предмет, при което добавяме към S числата m_3, m_1+m_3, m_2+m_3 и $m_1+m_2+m_3$. Сега S съдържа всевъзможните суми от стойностите на първите 3 предмета. И така, на всяка стъпка k ($k = 1, 2, ..., n$) прибавяме към S числата от вида $s + m_k$, където s е елемент от S.

Ще използваме масив `can[]`. Всеки елемент `can[i]` ще може да приема две стойности — 1 или `0`, в зависимост от това дали съответната стойност `i` може да бъде получена като сума от стойности на някои от предметите. В началото ще инициализираме `can[0]` с 1, а останалите елементи — с 0. На всяка стъпка j при разглеждането на поредния предмет k за подялба ще проверяваме последователно дали `can[j]` е 1, в който случай ще установяваме в 1 и `can[j+m[k]]`. Преглеждането на масива `can[]` ще извършваме от по-големите индекси към по-малките. Така ще се предпазим от повторно включване на един и същ предмет в множеството. Наистина, ако преглеждаме елементите на `can[]` по посока на нарастване на индексите, при включване на k-тия подарък ще установим `can[m[k]]` в 1. По-късно, достигайки до `can[m[k]]`, ще установим в 1 и `can[m[k]+m[k]]` (защото `can[m[k]]` вече ще има стойност 1), след това и `can[m[k]+m[k]+m[k]]` (защото `can[m[k]+m[k]]` е 1) и т.н. Проблемът е в това, че не можем да установим дали стойността 1 е била установена на текущата стъпка (в който случай нямаме право да правим ново включване) или вече е съдържала 1 на някоя предходна стъпка. Проблемът би могъл да се реши леко с използването на трета стойност, но едва ли си струва. По-лесно е просто да се обърне посоката на обхождане.

Накрая остава да намерим най-близкия до $p/2$ индекс `k` на `can[]`, за който `can[k] == 1`. Ясно е, че можем да ограничим търсенето само в едната посока, например сред стойностите, помалки от $p/2$, тъй като единият брат със сигурност ще трябва да вземе предмети на стойност наймного половината на p.

```
#include <stdio.h>
#define MAX       100 /* Максимален брой предмети */
#define MAXVALUE  200 /* Максимална стойност на отделен предмет */

unsigned char can[MAX*MAXVALUE]; /* Може ли да се получи сумата? */

const unsigned m[MAX] = {3,2,3,2,2,77,89,23,90,11}; /* Стойности */
const unsigned n = 10;      /* Общ брой на предметите за поделяне */

void solve(void)
{ unsigned long p; /* Обща стойност на предметите за поделяне */
  unsigned i, j;
```

```
  /* Пресмятаме p */
  for (p = i = 0; i < n; p += m[i++])
    ;

  /* Начално инициализиране */
  for (i = 1; i <= p; i++)
      can[i] = 0;
  can[0] = 1;

  /* Намиране на всевъзможните суми от стойности на подаръците */
  for (i = 0; i < n; i++)
    for (j = p; j+1 > 0; j--)
      if (can[j])
             can[j + m[i]] = 1;

  /* Намиране на най-близката до p/2 стойност */
  for (i = p / 2; i > 1; i--)
    if (can[i]) {
      printf("\n%s%u%s%lu", "сума за Алан:",i," сума за Боб:",p-i);
      return;
    }
}

int main(void)
{ solve();
  return 0;
}
```

🖫 alanbob.c

Непосредствено от кода на програмата се вижда, че сложността ѝ както по време, така и по памет е $\Theta(n.p)$. Нека предположим, че максималната стойност на предмет от множеството за подялба не надвишава някаква константа c, независеща от n. Тогава е в сила неравенството $p \leq c.n$. Оттук за сложността на алгоритъма по време и памет получаваме $\Theta(c.n^2)$, т. е. $\Theta(n^2)$. Ще отбележим, че паметта, необходима за съхраняване на битовата карта `can[]`, може да се намали 8 пъти при използване на техниките от *8.2.1*.

Бихме могли да обобщим настоящата задача и освен сумата от стойностите, да намираме и конкретните подаръци, които ще вземе всеки от братята. За целта е необходимо да пазим и допълнителна информация за всяка сума `can[j]` от `can[]`, а именно прибавянето на кой елемент е довел до установяването на `can[j]` в 1. След намиране на a (търсената обща стойност предмети, взети от Алан) ще можем по тази допълнителна информация да намерим последния предназначен за него подарък — p. Ако извадим стойността `m[p]` на p от a, въз основа на запазената допълнителна информация можем да определим предпоследния добавен подарък, както и сумата, към която е добавен. По аналогичен начин можем да намерим кой е предпредпоследният подарък и т.н., докато не възстановим едно конкретно множество от предмети.

Ще отбележим още, че не е необходимо да обхождаме всички елементи на масива `can[]`, а само тези с индекс, ненадминаващ общата сума на прегледаните до момента елементи — това повишава неколкократно ефективността на алгоритъма и е отчетено в настоящата програма. За целта е въведена допълнителна променлива `curSum`, съдържаща максималната постигната до момента сума.

```
unsigned last[MAX*MAXVALUE]; /* Кой предмет е добавен последен? */

void solve(void)
{ unsigned long p; /* Обща стойност на предметите за поделяне */
  unsigned long curSum = 0;
```

```
    unsigned i, j;

    /* Пресмятане на сумата p */
    for (p = i = 0; i < n; p += m[i++])
      ;

    /* Начално инициализиране */
    for (last[0] = 0, i = 1; i <= p; i++)
        last[i] = NOT_SET;

    /* Намиране на всевъзможните суми от стойности на подаръците */
    for (i = 0; i < n; i++) {
      for (j = p; j+1 > 0; j--)
        if (NOT_SET != last[j] && NOT_SET == last[j + m[i]])
              last[j + m[i]] = i;
      curSum += m[i];
    }

    /* Търсим на най-близка до p/2 стойност и извеждане на решение */
    for (i = p / 2; i > 1; i--) {
      if (NOT_SET != last[i]) {
        printf("\n%s%u%s%lu", "Сума за Алан:",i," сума за Боб:",p-i);
        printf("\nАлан взема:");
        while (i > 0) {
          printf(" %u", m[last[i]]);
          i -= m[last[i]];
        }
        printf("\nБоб взема останалите подаръци.");
        return;
      }
    }
}
```

alanbob2.c

Резултат от изпълнението на програмата:

```
сума за Алан: 136; сума за Боб: 166
Алан взема: 11 90 23 2 2 3 2 3
Боб взема останалите подаръци.
```

Задачи за упражнение:

1. Да се реализира функция, която да намира едно възможно разделяне на подаръците между братята Алан и Боб, без да се използва допълнителна информация за последния добавен предмет `last[k]`.
2. Може ли да се твърди, че предложеният алгоритъм е полиномиален?

8.2.3. Умножение на матрици

Задачата за оптималното умножение на матрици, известна още като задача за оптимално поставяне на скоби в аритметичен израз, е важна класическа задача, решавана с динамично оптимиране, поради което ще ѝ обърнем специално внимание.

Дадени са n матрици с размерности, които позволяват да се умножат:

$$M_1 \times M_2 \times \ldots \times M_n$$

Размерностите на матриците се задават с $n+1$ естествени числа: r_0, r_1, ..., r_n такива, че матрицата M_i е с размери $r_{i-1}\times r_i$ (r_{i-1} реда и r_i стълба), $i = 1, 2, ..., n$.

По дефиниция умножението на две матрици A и B с размери съответно $p\times q$ и $q\times r$ се извършва по формулата

$$c_{ij} = \sum_{k=1}^{q} a_{ik}.b_{kj} \text{ , } i = 1, 2, ..., p; j = 1, 2, ..., r.$$

в резултат на което се получава нова матрица C с размери $p\times r$.

Очевидно броят на елементарните умножения в горната формула е $p.q.r$. Ще припомним, че съществуват методи за умножение на матрици, при които броят на умноженията силно намалява, особено при по-големи размери на матрицата. Такъв е например методът на Щрасен (разгледан в *параграф 7.6.*).

Нека са дадени две матрици A и B (с размери съответно $p\times q$ и $s\times r$). С риск да отегчим читателя ще припомним, че само ако броят на стълбовете q на A е равен на броя на редовете s на B, можем да извършим умножението AB (но не и BA, за него се изисква $r = p$). Ако броят на матриците е по-голям, се изисква броят на редовете на всяка от тях да съвпада с броя на стълбовете на следващата.

Ще отбележим, че умножението на матрици е *асоциативна* операция, т. е. за всеки три матрици A, B и C, които имат подходящи размери, е в сила свойството: $(AB)C = A(BC)$. Оттук по индукция лесно получаваме, че редът на умножение не влияе върху крайния резултат, независимо от броя на матриците. Така например, при 4 матрици A, B, C и D са налице следните възможности за реда на извършване на умножението:

$(A(B(CD)))$
$(A((BC)D))$
$((AB)(CD))$
$((A(BC))D)$
$(((AB)C)D)$

Макар редът на умножение да не влияе по никакъв начин на крайния резултат, който винаги ще бъде един и същ, *времето* за извършване на умноженията може съществено да се различава при различните схеми поради различния брой елементарни умножения. Да разгледаме един пример: Нека са дадени матриците $M_1(10\times 20)$, $M_2(20\times 50)$, $M_3(50\times 1)$ и $M_4(1\times 100)$. Ако ги умножим в реда: $M_1\times(M_2\times(M_3\times M_4))$, ще получим:

1. За $L_1 = M_3 \times M_4$ $\Rightarrow$ 50.1.100 = 5000 [50×100]
2. За $L_2 = M_2 \times L_1$ $\Rightarrow$ 20.50.100 = 100000 [20×100]
3. За $M = M_1 \times L_2$ $\Rightarrow$ 10.20.100 = 20000 [10×100]

или общо 125000 умножения. Ако обаче ги умножим в реда $(M_1\times(M_2\times M_3))\times M_4$, ще бъдат нужни общо едва 2200 прости умножения:

1. За $L_1 = M_2 \times M_3$ $\Rightarrow$ 20.50.1 = 1000 [20×1]
2. За $L_2 = M_1 \times L_1$ $\Rightarrow$ 10.20.1 = 200 [10×1]
3. За $M = L_2 \times M_4$ $\Rightarrow$ 10.1.100 = 1000 [10×100]

Задачата, която ще си поставим, е не да извършим умноженията на матриците, а да намерим реда, в който това ще изисква минимален брой елементарни умножения. Така задачата се свежда до намиране на правилното поставяне на скобите. Ще считаме, че са дадени размерностите r_0, r_1,..., r_n.

Преди да продължим, ще се убедим, че двете необходими условия за прилагане на динамично оптимиране са налице. Действително, оптималното решение има оптимална подструктура, т. е. състои се от оптимални решения на задачата с по-малка размерност. Да разгледаме по-общата задача за намиране на оптималното разполагане на скоби в израз, състоящ се не от M_1, M_2, ..., M_n, а от M_i, M_{i+1}, ..., M_j, $1 \le i \le j \le n$. При $i = 1$, $j = n$ получаваме условието на изходната за-

дача. Да предположим, че оптималното решение се получава при разделяне на последователността за някакво k ($i \le k < j$) така, че първо се пресмятат $B = M_i, M_{i+1}, ..., M_k$ и $C = M_{k+1}, M_{k+2}, ..., M_j$, след което крайният резултат се получава като произведение на B и C. Не е трудно да се забележи, че матриците B и C трябва да се получават като *оптимално* решение на задачата за съответната по-малка размерност. Това води до следния алгоритъм:

Нека m_{ij} е минималният брой прости умножения, необходими за умножението на матриците $M_i \times M_{i+1} \times ... \times M_j$, $1 \le i \le j \le n$. Тогава е в сила рекурентната формула ($1 \le i \le j \le n$):

$$m_{i,j} = \begin{cases} 0 & i = j \\ \min\limits_{k=i}^{j-1}\left(m_{i,k} + m_{k+1,j} + r_{i-1}r_k r_j\right) & i < j \end{cases}$$

Тогава, ако знаем $m_{i,k}$ и $m_{k+1,j}$ за всяко k между i и j, бихме могли еднозначно да определим $m_{i,j}$. Директното прилагане на горната формула обаче води до експоненциална сложност, тъй като в процеса на рекурсивно пресмятане ще решаваме една и съща задача многократно.

```
#include <stdio.h>
#define MAX 100
#define INFINITY (unsigned long)(-1)

unsigned long m[MAX][MAX];    /* Таблица - целева функция */

/*Размерности на матриците*/
const unsigned long r[MAX+1] = {12,13,35,3,34,2,21,10,21,6};
const unsigned n = 9;         /* Брой матрици */

/* Неефективна рекурсивна функция */
unsigned long solveRecursive(unsigned i, unsigned j)
{ unsigned k;
  if (i == j) return 0;
  m[i][j] = INFINITY;
  for (k = i; k <= j - 1; k++) {
    unsigned long q = solveRecursive(i, k) +
                      solveRecursive(k + 1, j) +
                      r[i - 1] * r[k] * r[j];
    if (q < m[i][j])
      m[i][j] = q;
  }

  return m[i][j];
}

void printMatrix(void) /* Извежда матрицата на минимумите */
{ unsigned i,j;
  printf("\nМатрица на минимумите:");
  for (i = 1; i <= n; i++) {
    printf("\n");
    for (j = 1; j <= n; j++)
      printf("%8lu", m[i][j]);
  }
}

int main(void) {
  printf("\nМин. брой умножения е: %lu",solveRecursive(1,n));
  printMatrix();
  return 0;
}
```

🖫 matrix1.c

Налице е припокриване на подзадачите, което е именно второто необходимо условие за успешно прилагане на динамично оптимиране. Броят на различните последователности от вида M_i, M_{i+1}, ..., M_j, $1 \leq i \leq j \leq n$ е едва $n(n+1)/2$. Натъкваме се на същия проблем, както при предходните примери. Едно стандартно решение, както знаем, е прилагането на метода *memoization*, което позволява да се запази рекурсията, при съществено подобрение на ефективността на алгоритъма. Лесно се вижда, че сложността в този случай става $\Theta(n^3)$.

```
long solveMemo(unsigned i, unsigned j)
{ unsigned k;
  unsigned long q;
  if (NOT_SOLVED != m[i][j]) /* Стойността вече е пресметната */
    return m[i][j];
  if (i == j)                /* В този интервал няма матрица */
    m[i][j] = 0;
  else {                     /* Пресмятаме рекурсивно */
    for (k = i; k <= j - 1; k++)
      if ((q = solveMemo(i,k)+solveMemo(k+1,j)+r[i-1]*r[k]*r[j])
           < m[i][j])
        m[i][j] = q;
  }
  return m[i][j];
}

long solveMemoization(void)
{ unsigned i, j;
  for (i = 1; i <= n; i++)
    for (j = i; j <= n; j++)
      m[i][j] = NOT_SOLVED;
  return solveMemo(1, n);
}
```
🖫 matrix2.c

При по-внимателен поглед върху кода на горните функции забелязваме, че матрицата (m_{ij}) съдържа нули по главния диагонал и под него. На практика тези стойности, представляващи повече от половината полезни стойности в масива, се инициализират, но не се използват. Едно възможно решение е масивът да се линеаризира, т. е. да се използва едномерен вместо двумерен масив, достъпът до елементите на който ще се извършва с помощта на подходяща двуаргументна функция на i и j.

Бихме могли да използваме част от празните елементи на матрицата (без тези по главния диагонал) далеч по-пълноценно: над главния диагонал, както и досега, ще записваме минимумите, а симетрично на него — конкретната стойност на k, за която се получават. Запазването на конкретните стойности на k. По-късно ще ни позволи да възстановим едно възможно поставяне на скобите в оптималното решение. Матрицата, получена при обработката на дадения по-горе пример, ще придобие вида от *таблица 8.2.3*.

$m_{1,1} = 0$	$m_{1,2} = 10000$	$m_{1,3} = 1200$	$m_{1,4} = 2200$
$k_{2,1} = 1$	$m_{2,2} = 0$	$m_{2,3} = 1000$	$m_{2,4} = 3000$
$k_{3,1} = 1$	$k_{3,2} = 2$	$m_{3,3} = 0$	$m_{3,4} = 5000$
$k_{4,1} = 3$	$k_{4,2} = 3$	$k_{4,3} = 3$	$m_{4,4} = 0$

Таблица 8.2.3. Стойности, получени в резултат на изпълнение на програмата.

Пресмятанията ще започваме от по-близки стойности на i и j, като постепенно ще разширяваме разглежданите интервали и в даден момент ще достигнем до $i = 1$ и $j = n$. Това от своя страна означава, че вече сме изчислили $m_{1,n}$, което ще бъде търсеният краен минимален резултат. Как точно става това, се вижда достатъчно ясно от приложения програмен код (функция `solve()`). Функцията `printMatrix()` извежда получената матрица `m[][]` на екрана. Възстановяването на едно възможно оптимално поставяне на скобите не представлява трудност. За целта се използват индексите под главния диагонал, които сме запазили, както и формулата, по която извършваме пресмятанията (виж функцията `getOrder()`). Не представлява особена трудност и построяването на план за пресмятане на произведението (виж `buildOrder()` и `printMultiplyPlan()`).

За да разнообразим малко нещата и за да ги направим по-ясни, програмата по-долу ще работи по малко по-различна формула, при която горната граница се дава не от j, а от $i+j$, т. е. j означава не *края* на разглеждания интервал, а *броя на елементите* в него, намален с 1. Така получаваме по-удобната формула ($1 \le i \le i+j \le n$):

$$m_{i,j} = \begin{cases} 0 & j = 0 \\ \min\limits_{k=i}^{i+j-1}\left(m_{i,k} + m_{k+1,j} + r_{i-1}r_k r_{i+j}\right) & j > 0 \end{cases}$$

Пресмятането започва със стойност $j = 1$. Това отговаря на умножението на две съседни матрици. След като броят на елементарните умножения за всички произведения на двойки съседни матрици бъде намерен, j получава стойност 2. Сега пресмятането на цената (броят елементарни произведения) на тройките последователни матрици не е проблем, тъй като вече имаме готови пресметнати стойностите за всички двойки. На следващата стъпка j става 3 и т.н. На всяка стъпка се използват само вече пресметнати стойности на целевата матрица.

```
#include <stdio.h>

#define MAX 100
#define INFINITY (unsigned long)(-1)

unsigned long m[MAX][MAX];     /* Таблица - целева функция */

struct {
  unsigned left;
  unsigned right;
} order[MAX * MAX];            /* Ред на умножение на матриците */

unsigned long cnt;             /* Брой действия за пресмятането */

/* Размерности на матриците */
const unsigned long r[MAX+1] = {12,13,35,3,34,2,21,10,21,6};
const unsigned n = 9;          /* Брой матрици */

/* Формира таблица, съдържаща минималния брой умножения,
 * необходими за умножението на всяка двойка матрици,
 * както и индексът, за който се постига */
void solve(void)
{ unsigned i, j, k;

  /* Инициализиране */
  for (i = 1; i <= n; i++)
    m[i][i] = 0;

  /* Основен цикъл */
  for (j = 1; j <= n; j++) {
```

```
    for (i = 1; i <= n - j; i++) {
      m[i][i + j] = INFINITY;
      for (k = i; k < i + j; k++) {
        unsigned long t = m[i][k]+m[k+1][i+j] + r[i-1]*r[k]*r[i+j];

        /* Подобряване на текущото решение */
        if (t < m[i][i + j]) {
          m[i][i + j] = t;
          m[i + j][i] = k;
        }
      }
    }
  }
}

unsigned buildOrder(unsigned ll,unsigned rr) /* Ред на умножение */
{ long ret = cnt++;
  if (ll < rr) {
    order[ret].left = buildOrder(ll, m[rr][ll]);
    order[ret].right = buildOrder(m[rr][ll] + 1, rr);
  }
  else {
    order[ret].left = ll;
    order[ret].right = rr;
  }
  return ret;
}

void printMatrix(void) /* Извежда матрицата на минимумите */
{ unsigned i, j;
  printf("\nМатрица на минимумите :");
  for (i = 1; i <= n; i++) {
    printf("\n");
    for (j = 1; j <= n; j++)
      printf("%8lu", m[i][j]);
  }
}

void printMultiplyPlan(void) /* Извежда план за умножение */
{ unsigned long i;
  printf("\nПлан за умножение на матриците:");
  for (i = 0; i < cnt; i++)
    if (order[i].left == order[i].right)
     printf("\nL[%lu] = M%u", i, order[i].left);
    else
     printf("\nL[%lu]=L[%u]*L[%u]",i,order[i].left,order[i].right);
}

void getOrder(unsigned ll, unsigned rr) /* Редът със скоби */
{ if (ll == rr) printf("M%u", ll);
  else {
    printf("(");
    getOrder(ll, m[rr][ll]);
    printf("*");
    getOrder(m[rr][ll] + 1, rr);
    printf(")");
  }
}
```

```
int main(void) {
  solve();
  cnt = 0; buildOrder(1, n);
  printf("\nМинималният брой умножения е: %lu", m[1][n]);
  printMatrix();
  printMultiplyPlan();
  printf("\nРед на умножение на матриците: ");
  getOrder(1, n);
  return 0;
}
```

matrix3.c

Резултат от изпълнението на програмата:

```
Минималният брой умножения е: 2872
Матрица на минимумите :
 0    5460    1833    3057    1636    2140    2296    2980    2872
 1       0    1365    2691    1324    1870    2004    2710    2572
 1       2       0    3570     414    1884    1534    2724    1926
 3       3       3       0     204     330     684    1170    1332
 1       2       3       4       0    1428    1100    2268    1500
 5       5       5       5       5       0     420     840    1092
 5       5       5       5       5       6       0    4410    2520
 5       5       5       5       5       7       7       0    1260
 5       5       5       5       5       8       7       8       0
План за умножение на матриците:
L[0] = L[1] * L[10]
L[1] = L[2] * L[3]
L[2] = M1
L[3] = L[4] * L[5]
L[4] = M2
L[5] = L[6] * L[7]
L[6] = M3
L[7] = L[8] * L[9]
L[8] = M4
L[9] = M5
L[10] = L[11] * L[16]
L[11] = L[12] * L[15]
L[12] = L[13] * L[14]
L[13] = M6
L[14] = M7
L[15] = M8
L[16] = M9
Ред на умножение на матриците: ((M1*(M2*(M3*(M4*M5))))*(((M6*M7)*M8)*M9))
```

Задачи за упражнение:

1. Да се напише функция, която възстановява един възможен оптимален ред на умножение на матриците, без да използва изрично запазени индекси, само въз основа на стойностите на целевата функция.
2. Да се напише функция, която намира *всички* възможни оптимални решения на задачата.
3. Да се реализира вариант на програмата, който използва *линеаризирана* матрица, т.е. едномерен масив, с $n(n-1)/2$ елемента.
4. Възможно ли е задачата за умножение на матрици да се реши с памет, линейна по n?
5. Може ли да се твърди, че задачата за матриците се решава полиномиално?

8.2.4. Триангулация на многоъгълник. Числа на Каталан

Задача: Разполагаме с изпъкнал многоъгълник, направен от плосък метален лист. Върховете му са номерирани от 1 до n по посока на обхождане на контура. С ножица можем да правим разрез по отсечка, съединяваща два произволни върха на многоъгълника. Целта ни е, след последователност от такива разрези, да получим само триъгълници. Изхабяването на ножицата е правопропорционално на сумата от дължините на направените разрези. Да се състави програма, която въвежда координатите на върховете на многоъгълника и посочва търсената последователност от разрези така, че ножицата да се изхаби най-малко [*Computer-5/98*].

На пръв поглед настоящата задача по нищо не прилича на предходната. Връзка между двете обаче има и тя се дава от *числата на Каталан*. Една възможна дефиниция на тези числа е като броят на различните начини за разрязване (*триангулация*) на изпъкнал n-ъгълник на (n–2) на брой триъгълника. Числата на Каталан се намират по формулата

$$T(n) = \frac{1}{n+1}\binom{2n}{n}$$

Оказва се, че този брой е свързан с броя на различните начини за умножение на n матрици. Да се опитаме да изведем горната формула. За умножението на n матрици опитвахме да разделим последователността на две последователности на позиция k (k = 1, 2, ..., n–1). Оттук за броя на различните поставяния на скобите директно получаваме рекурентната зависимост:

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n \geq 2 \end{cases} \qquad (1)$$

След опростяване получаваме връзката $P(n) = T(n-1)$. Числата на Каталан имат множество различни интерпретации. По-долу ще приведем някои от тях, като при повече въображение списъкът би могъл да се продължи:

1. Броят на начините за правилно поставяне на скобите в неасоциативно произведение на n+1 множителя.
2. Броят на начините за разрязване на (n+2)-ъгълник на n триъгълника с помощта на n–1 непресичащи се негови диагонала.
3. Броят на начините за умножение на n матрици. (*сравнете с* 1)
4. Частното на средния биномен коефициент $(2n)!/(n!n!)$ и n+1.
5. Членовете на редицата c_i, зададена чрез равенствата:

$$c_0 = 1,$$
$$c_{n+1} = c_0c_n + c_1c_{n-1} + c_2c_{n-2} + \ldots + c_nc_0,\ n \geq 0. \qquad (2)$$

6. Членовете на рекурентната редица, зададена чрез равенствата:

$$c_0 = 1 \text{ и } (n+2)c_{n+1} = (4n+2)c_n,\ \ n \geq 0. \qquad (3)$$

7. Коефициентите пред степените на x в развитието на пораждащата функция

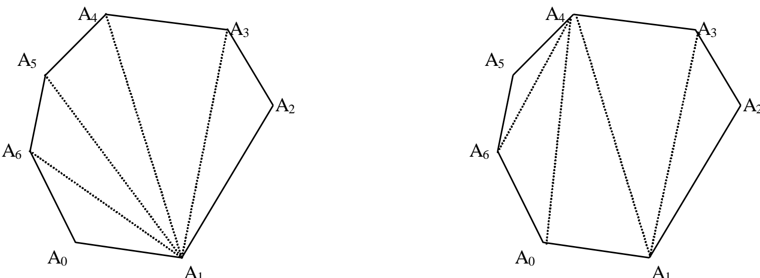
$$\frac{1-\sqrt{1-4x}}{2x} \qquad (4)$$

8. Броят на двоичните дървета с n листа.
9. Броят на кореновите двоични дървета с n върха.
10. Броят на планинските вериги, които могат да бъдат начертани с n черти нагоре и n черти надолу.
11. Броят на отделните планини, които могат да бъдат начертани с помощта на n+1 черти нагоре и n+1 черти надолу (падините между върховете вече не могат да падат до морското равнище).

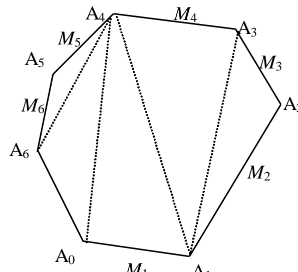
12. Броят на различните пътища от (0,0) до (n,n) в квадратна мрежа, като на всяка стъпка точно една от двете координати x или y нараства с единица и не се позволява *пресичане* на главния диагонал, определен от уравнението $y = x$, в нито една вътрешна точка. (*сравнете с* 10)
13. Броят на различните пътища от (0,0) до $(n+1,n+1)$, като на всяка стъпка точно една от двете координати x или y нараства с единица и не се позволява *докосване* до главния диагонал, определен от уравнението $y = x$, в нито една вътрешна точка. (*сравнете с* 11)
14. Броят на начините, по които всеки един от два кандидата A и B, може да получи общо n гласа, като при това в нито един момент за A не се подадени повече гласове, отколкото за B.
15. Броят на начините, по които $2n$ на брой души, седнали около кръгла маса, могат да се здрависат, без никои две двойки да си кръстосват ръцете. (*сравнете с 2*)

Еквивалентността на някои от горните формулировки е очевидна, други следват непосредствено от някои математически свойства на числата на Каталан и могат лесно да се докажат (например по индукция). Ще разгледаме по-подробно връзката между триангулацията на многоъгълник и умноженията на матрици. Не е трудно да се покаже, че всяка триангулация разделя произволен n-ъгълник на n–2 на брой триъгълника, определени от n–3 на брой разреза. Така например, един седмоъгълник се триангулира на 7–2 = 5 триъгълника с помощта на 7–3 = 4 разреза (*виж фигура 8.2.4а.*).

Ще изясним връзката между триангулацията на многоъгълник, поставянето на скобите в произведението на матриците и двоичните дървета с n листа. Да разгледаме втората триангулация на многоъгълника от *фигура 8.2.4а.*. Да съпоставим на страната $\overline{A_{i-1}A_i}$ матрицата M_i, i = 1, 2, ..., 6. Забележете, че на страната $\overline{A_0A_6}$ не сме съпоставили матрица!



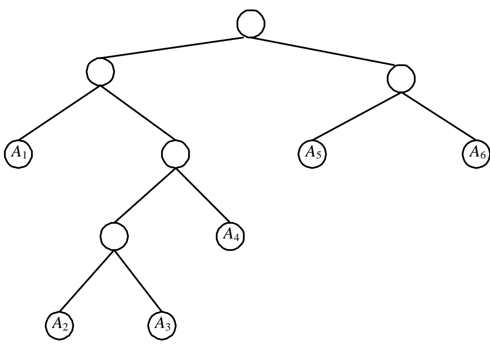
Фигура 8.2.4а. Две възможни триангулации на многоъгълник.



Фигура 8.2.4б. Връзка между умножение на матрици и триангулация.

Да разгледаме диагонала $\overline{A_0A_4}$ (*виж фигура 8.2.4б.*). Вижда се, че 4 от матриците остават в едната полуравнина по отношение на него, а останалите — в другата. Така, диагоналът определя първото разделяне на матриците: $(M_1M_2M_3M_4)(M_5M_6)$. Добавяйки диагонала $\overline{A_1A_4}$,

определяме разделянето $(M_1(M_2M_3M_4))(M_5M_6)$, а добавяйки $\overline{A_1A_3}$ — разделянето $(M_1((M_2M_3)M_4))(M_5M_6)$. Не е трудно да се построи съответното дърво на израза (*виж фигура 8.2.4в.*).



Фигура 8.2.4в. Дърво на израза със скоби $(M_1((M_2M_3)M_4))(M_5M_6)$.

Не е трудно да се види, че на всяко двоично дърво с точно n листа може да се съпостави съответен израз със скоби, т. е. съответен ред на умножение на матриците. Оттук пък можем да получим съответна триангулация на изпъкнал $(n+1)$-ъгълник. Така показахме тристранната връзка: триангулация–матрици–дървета.

Връзката между броя на различните конфигурации в двете задачи ни навежда на мисълта да търсим сходство в начина на решаването им. При матриците стойността на едно умножение на матриците L_1 и L_2 (L_1 е матрицата, получена при умножението на всички матрици от i-тата до k-тата, а L_2 — от $(k+1)$-вата до j-тата) се изразява с `r[i-1].r[k].r[j]`, където `r[.]` са размерности на матриците. На едно такова умножение ще съпоставим триъгълник, определен от три върха на многоъгълника с номера съответно: $i–1$, k и j. Не е трудно да се забележи, че в крайна сметка обединението на така построените триъгълници съдържа всички страни на многоъгълника *точно по веднъж*, както и някои негови диагонали. При това забележете, че диагоналите не се пресичат и всеки съдържащ се в триъгълник диагонал е страна на точно два триъгълника, като при това тези диагонали разбиват многоъгълника на триъгълници по описания в условието на задачата начин.

Нека обозначим с R общата дължина на диагоналите от разреза. Тогава вместо да минимизираме R, бихме могли да минимизираме израза $(P+2R)$, където P е периметърът на многоъгълника. Ясно е, че при всеки разрез P ще бъде една и съща константа. Нека в програмата, решаваща задачата за матриците, заменим израза `r[i-1].r[k].r[j]` с

`d[i-1][k] + d[k][j] + d[i-1][j]`, $1 \le i \le k < j \le n$,

където $d[n_1][n_2]$ е разстоянието (Евклидово разстояние) между точките с номера n_1 и n_2. От използвания запис става ясно, че се предполага, че разстоянията са предварително пресметнати и записани в масив. Това е разумно, тъй като те се ползват многократно, а пресмятането им е относително тежка операция: съдържа две изваждания, две умножения и едно кореннуване.

След приключване на работата на основната функция матрицата M ще съдържа търсения минимум на $F = P + 2R$. За да се намери R, следва да се пресметне P: $R = (F–P)/2$. Получаваме следната целева функция ($1 \le i \le j \le n$):

$$m_{i,j} = \begin{cases} 0 & i = j \\ \min\limits_{k=i}^{j-1}\left(m_{i,k} + m_{k+1,j} + d_{i-1,k} + d_{k,j} + d_{i-1,j}\right) & i < j \end{cases} \qquad (5)$$

В действителност по-удобно е да работим с малко по-различна формула, при която горната граница се дава не от j, а от $i+j$, т. е. j означава не края на разглеждания интервал, а броя на елементите в него, намален с 1. Така, формулата придобива вида ($1 \le i \le i+j \le n$):

$$m_{i,j} = \begin{cases} 0 & j = 0 \\ \min\limits_{k=i}^{i+j-1}\left(m_{i,k} + m_{k+1,j} + d_{i-1,k} + d_{k,i+j} + d_{i-1,i+j}\right) & j > 0 \end{cases} \qquad (6)$$

По-горе на всеки n-ъгълник съпоставихме еднозначно (с точност до преномериране на върховете) $n-1$ последователни матрици: на всяка страна, с изключение на точно една, съпоставихме съответна матрица. Оттук следва, че трябва да изпълним алгоритъма за $n-1$. Това се извършва в началото на функцията `solve()`.

Входните данни на програмата са зададени като константа. Функцията `calcDist()` пресмята разстоянията между всички двойки върхове и ги запазва в масива `d[][]`. Забележете, че цената на минималния разрез сега не се получава в `m[1][n]`, а по формулата `R = (m[1][n]-P)/2`, където `P` е периметърът на многоъгълника. За това се грижи функцията `printResult()`. Отново можем симетрично относно главния диагонал да пазим това k, за което е постигната максималната стойност. Това ще ни позволи да възстановим един възможен максимален разрез: функция `writeCut()`.

```
#include <stdio.h>
#include <math.h>
#define MAX 80
#define INFINITY 1e20

double d[MAX][MAX];   /* Разстояния между върховете */
double m[MAX][MAX];   /* Таблица - целева функция */

const struct {        /* Координати на върховете */
  int x;
  int y;
} coord[MAX] = {{1,1},{5,-2},{10,1},{7,7},{1,7}};
const unsigned n = 5; /* Брой върхове */

/* Пресмята разстоянията между всички двойки върхове */
void calcDist(void)
{ unsigned i, j;
  for (i = 0; i < n - 1; i++)
    for (j = i + 1; j < n; j++)
      d[i][j] = sqrt(
              (coord[i].x - coord[j].x) * (coord[i].x - coord[j].x) +
              (coord[i].y - coord[j].y) * (coord[i].y - coord[j].y));
}

/* Формира таблица, съдържаща минималния брой умножения за
 * умножение на две матрици и индекса, за който се постига */
void solve(void)
{ unsigned i, j, k;
  for (i = 1; i < n; i++)
       m[i][i] = 0; /* Инициализиране */
  for (j = 1; j < n; j++) {         /* Основен цикъл */
    for (i = 1; i < n - j; i++) {
      m[i][i + j] = INFINITY;
      for (k = i; k < i + j; k++) {
        double t = m[i][k] +
          m[k+1][i+j] +
          d[i-1][k] +
```

```
          d[k][i+j] +
          d[i-1][i+j];
        if (t < m[i][i+j]) {  /* Подобряване на текущото решение */
          m[i][i+j] = t;
          m[i+j][i] = k;
        }
      }
    }
  }
}

/* Отпечатване на резултата */
void printResult(void)
{ unsigned i;
  double p = d[0][n-1];  /* Пресмятане на периметъра */
  for (i = 0; i < n; i++)
    p += d[i][i+1];
  printf("\nДължината на мин. разрез е %.2lf", (m[1][n-1] - p)/2);
}

/* Извеждане на минималния разрез на екрана */
void writeCut(unsigned ll, unsigned rr)
{ if (ll != rr) {
    writeCut(ll, (unsigned) m[rr][ll]);
    writeCut((unsigned) m[rr][ll] + 1, rr);
    if (ll != 1 || rr != n-1)
      printf("(%u,%u) ", ll, rr + 1);
  }
}
int main(void) {
  calcDist();
  solve();
  printResult();
  printf("\nДиагонали от минималния разрез: ");
  writeCut(1, n-1);
  return 0;
}
```

🖫 catalan.c

Резултат от изпълнението на програмата:

```
Дължината на минималния разрез е 17.49
Диагонали от минималния разрез: (1,3) (1,4)
```

Задачи за упражнение:

1. Каква е сложността на горния алгоритъм?
2. Да се докажат (1), (2) и (3).
3. Да се докаже еквивалентността на различните дефиниции на числата на Каталан.
4. Защо на *фигура 8.2.4а.* на страната $\overline{A_0A_6}$ не е съпоставена матрица?

8.2.5. Оптимално двоично дърво за претърсване

Ще разгледаме още една задача, свързана с числата на Каталан. По-горе в *параграф 8.2.4.* вече разгледахме взаимно еднозначното съответствие между начините за умножение на n матрици, триангулацията на $(n+1)$-ъгълник и двоичните дървета за претърсване с точно n листа. Сега ще прибавим към този клас и *кореновите* двоични дървета с n върха.

Ще припомним, че двоичните дървета за претърсване бяха вече разгледани в *параграф 2.3*. Тази структура от данни намира широко приложение при реализацията на множество алгоритми. Причината за това се крие от една страна в простотата на организацията ѝ, а от друга — в ефективността на търсенето, вмъкването, актуализирането и изтриването, които в общия случай имат логаритмична сложност. За съжаление, при неподходящо разпределение на входните данни или в резултат на интензивно изтриване и вмъкване дървото може да се изроди в списък, при което изброените операции да изискват линейно време. В такъв случай се използват така наречените *балансирани дървета* (*виж 2.4.*), които предвиждат механизъм за запазване на някаква степен на балансираност на дървото въз основа на прости правила. Така се предотвратява израждането на дървото в списък, като в най-лошия случай търсенето в такова дърво изисква 45% повече време в сравнение с търсенето в съответното идеално балансирано дърво [*Уирт-1980*]. В случай, че дървото не може да се побере в паметта, се използват така наречените *B*-дървета (*виж 2.4.2.*), които имат странична организация и позволяват в паметта да се зарежда само тази част от дървото, която се обработва в момента. Подобно на балансираните дървета *B*-дърветата също позволяват освен търсене, вмъкване, актуализиране и изтриване на елемент. Друг възможен подход е отказът от дървовидни структури и използване на *хеш-таблици* (*виж 2.5.*), чиито предимства и недостатъци няма да обсъждаме.

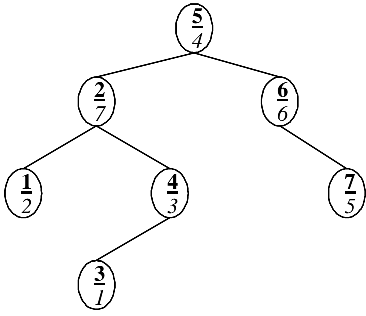
Понякога се налага изграждането на дърво за претърсване, в което операциите добавяне, актуализиране или изключване на елемент не се срещат или са изключително редки. В такъв случай се поставя проблемът за построяването на идеално балансирано дърво, търсенето в което в най-лошия случай изисква $\log_2 n$ сравнения. *Идеално балансирано* е такова двоично дърво, което при предварително зададен брой на върховете n има най-малката височина измежду всички двоични дървета с n върха. То е пълно за всички нива с изключение евентуално на последното. Построяването на идеално балансирано дърво за претърсване (т. е. за което ключът на левия наследник е по-малък, а на десния — по-голям от този на корена) по сортирана последователност $a_1, a_2, ..., a_n$ е просто: За корен се избира средният елемент $a_{\left\lceil \frac{n+1}{2} \right\rceil}$. Негов ляв наследник е средният елемент на $a_1, a_2, ..., a_{\left\lceil \frac{n+1}{2} \right\rceil - 1}$, а десен — средният елемент на $a_{\left\lceil \frac{n+1}{2} \right\rceil + 1}, a_{\left\lceil \frac{n+1}{2} \right\rceil + 2}, ..., a_n$. Изграждането на дървото продължава рекурсивно, като на всяка стъпка се взема средният елемент за съответната последователност. Не е трудно да се види, че така построеното дърво е сортирано и идеално балансирано.

Понякога обаче върховете на дървото се характеризират освен с ключ, който задава наредбата на елементите му, и с някакво допълнително число, например пропорционално на честота на срещане. Такъв случай възниква например при компилаторите, които срещат някои от ключовите думи от езика по-често от други. Така например, в езика Си, ключовите думи `for` и `if` се срещат по-често от `main` или `goto` [*Sedgewick-1992*].Ако искаме да ускорим бързодействието на компилатора, следва да изградим дървото на ключовите думи така, че най-често срещаните да бъдат по-близо до върха му, така че достъпът до тях да бъде най-бърз. При това двоичното дърво трябва да остане сортирано. По същия начин ще се подобри скоростта на работа на една програма за проверка на текст, ако успеем да построим по подходящ начин в дървото думите, които тя разпознава като правилни. Наборът от думи е предварително фиксиран и не се изменя.

Задачата, която ще си поставим, е по даден набор от ключови думи $d_1 < d_2 < ... < d_n$ и съответстващи им честоти на срещане $f_1, f_2, ..., f_n$ да построим оптимално двоично дърво за претърсване, т. е. това, което минимизира сумата по всички ключове на времето за достъп до съответния ключ. (Забележете, че става дума за *наредено* двоично дърво за *претърсване*, т. е. за всеки връх думите от лявото поддърво са преди него, а тези от дясното — след него в лексикографска наредба. За да не забравяме това, ще използваме означението “<”.) Времето за достъп до даден ключ се задава от произведението на броя върхове по пътя от него до корена, умножен по съответната честота на срещане f_i. Да разгледаме дървото от *фигура 8.2.5*. Всеки възел съдържа две числа. Горното е ключът i, а долното — честотата му на срещане f_i. Да пресметнем

стойността на претегления вътрешен път. Времето за достъп до корена е: 1.4 = 4 (преминаваме през един връх и честотата f_5 е равна на 4). Времето за достъп до върха с ключ **1** е 3.2 = 6, тъй като пътят до корена съдържа 3 върха, а честотата f_1 е 2. Аналогично, времето за достъп до върха с ключ **4** е 3.3 = 9, тъй като пътят до корена съдържа 3 върха, а f_4 = 3. Сумирайки за всички върхове, получаваме (*виж фигура 8.2.5.*):

$$2.3 + 7.2 + 1.4 + 3.3 + 4.1 + 6.2 + 5.3 = 64$$



Фигура 8.2.5. Оптимално двоично дърво за претърсване.

Макар това на пръв поглед да не е съвсем очевидно, задачата за построяване на оптимално двоично дърво за претърсване може да се реши с помощта на алгоритъм, аналогичен на този от задачата за умножение на матрици (*виж 8.2.3.*). Да разгледаме по-общата задача за построяване на оптимално двоично дърво за претърсване, съдържащо върховете $d_i < d_{i+1} < ... < d_{i+j}$. Идеята е за всяко k ($1 \leq i \leq k \leq i+j \leq n$) да се пробва построяване на оптимално двоично дърво за претърсване с корен d_k, ляво поддърво, съдържащо $d_i < d_{i+1} < ... < d_{k-1}$ и дясно — съдържащо $d_{k+1} < d_{k+2} < ... < d_{i+j}$ ($1 \leq j \leq n-1$, $1 \leq i \leq n-j$). Дължината на вътрешния претеглен път е сума от дължините на претеглените вътрешни пътища за двете поддървета плюс s, където s е сумата от честотите на върховете $d_i < d_{i+1} < ... < d_{i+j}$. Действително, към дължините на вътрешните претеглени пътища за двете поддървета следва да се прибави честотата на срещане на корена, както и да се отчете, че двете поддървета са на едно ниво под него. Така получаваме s. Забележете, че ако се интересуваме единствено от оптималното дърво за претърсване, но не и от неговата цена, което е стандартният случай, не е нужно да пресмятаме и прибавяме сумата s, тъй като тя не зависи от избора на корена и се прибавя винаги.

Ясно е, че оптималното решение има оптимална подструктура, т. е. за да построим оптимално двоично дърво за претърсване, трябва двете поддървета на корена също да бъдат оптимални. Това е и първото достатъчно условие за използване на динамично оптимиране. Очевидно второто необходимо условие също е налице, тъй като в процеса на изграждане на оптимални поддървета една и съща задача ще се решава многократно.

За разлика от умножението на матрици тук е налице една специфична особеност: Ако разгледаме по-внимателно *фигура 8.2.5*, ще видим, че е възможно някой връх да има единствен наследник (вместо 0 или 2). Това не можеше да се случи при матриците. За да отчетем тази възможност, е необходимо да инициализираме `m[i][i-1]` с 0, $1 \leq i \leq n+1$. Това означава, че се използват стойности на `m[][]`, разположени под главния диагонал, което ще доведе до проблеми при запазване на оптималното решение [*Sedgewick-1992*]. За целта ще се наложи да преместим един ред надолу позицията, на която пазим стойността на k, за която се е постигнал максимумът. Така получаваме функция със сложност по време $\Theta(n^3)$ и по памет — $\Theta(n^2)$. Ще отбележим, че сложността по време на всяка програма е по-голяма или равна на сложността ѝ по памет, тъй като не може за време по-малко от t да се заеме памет t [*Манев-1996*]. (Тук под *заемане* имаме предвид и *инициализиране*. Ще отбележим, че в някои специални случаи съществуват бързи начини за *виртуално* инициализиране, например при *разредени* матрици, при които заемането става по-бързо. В този случай обаче не се ползва цялата памет, което означава, че реалната

сложност по памет е по-малка и се определя от действително ползваната памет, при което цитираното правило е вече в сила.)

Как да намерим оптималното дърво за претърсване? Нещата стоят аналогично на начина, по който възстановявахме реда на умножение на матриците. Трябва обаче да се отчетат две особености: 1) отместването на първата координата на индекса на k с 1; и 2) възможността за поява на връх с единствен наследник. Това е направено в предложената функция `getOrder()`. Дървото ще изведем на екрана, завъртяно на 90°. Ще въведем допълнителен параметър h, който ще показва нивото, на което се намира текущият възел, откъдето ще определяме отстъпа от началото на реда при извеждане на текущия връх. Обръщението към функцията става с `getOrder(1,n,0)`. Следва реализация. Като входни данни сме подали стойностите от дървото от *фигура 8.2.5*.

```
#include <stdio.h>

#define MAX 100

#define INFINITY (unsigned long)(-1)

unsigned long m[MAX][MAX];                /* Таблица - целева функция */

const unsigned long f[MAX+1] = {2,7,1,3,4,6,5}; /* Честоти на срещане */

const unsigned n = 7;                           /* Брой честоти */

/* Построява оптимално двоично дърво за претърсване */
void solve(void)
{ unsigned i, j, k;
  unsigned long t;

  /* Инициализиране */
  for (i = 1; i <= n; i++) {
    m[i][i] = f[i-1];
    m[i][i-1] = 0;
  }
  m[n+1][n] = 0;

  /* Основен цикъл */
  for (j = 1; j <= n - 1; j++) {
    for (i = 1; i <= n - j; i++) {
      m[i][i + j] = INFINITY;
      for (k = i; k <= i + j; k++) {

        /* Подобряваме текущото решение */
        if ((t = m[i][k - 1] + m[k + 1][i + j]) < m[i][i + j]) {
          m[i][i + j] = t;
          m[i + j + 1][i] = k;
        }
      }

      for (k = i-1; k < i + j; k++)
        m[i][i+j] += f[k];
    }
  }
}

/* Извежда матрицата на минимумите на екрана */
void PrintMatrix(void)
{ unsigned i,j;
```

```
  printf("\nМатрица на минимумите:");
  for (i = 1; i <= n+1; i++) {
    printf("\n");
    for (j = 1; j <= n; j++)
      printf("%8lu", m[i][j]);
  }
}

/* Извежда оптималното дърво на екрана */
void getOrder(unsigned ll, unsigned rr, unsigned h)
{ unsigned i;

  if (ll > rr)
    return;
  if (ll == rr) {
    for (i = 0; i < h; i++)
      printf("   ");
    printf("d%u\n", rr);
  }
  else {
    getOrder(ll, m[rr + 1][ll] - 1, h + 1);
    for (i = 0; i < h; i++)
      printf("   ");
    printf("d%lu\n", m[rr + 1][ll]);
    getOrder(m[rr + 1][ll] + 1, rr, h + 1);
  }
}

int main(void) {
   solve();
   printf("\nМинималната дължина на претегления вътрешен път е: %lu",
          m[1][n]);
   PrintMatrix();
   printf("\nОптимално дърво за претърсване:\n"); getOrder(1,n,0);
   return 0;
}
```
🖫 tree.c

В резултат получаваме именно дървото от *фигура 8.2.5*, т. е. то е оптималното. (Оттук не следва непременно, че е *единственото* оптимално.)

Резултат от изпълнението на програмата:

```
Максималната дължина на претегления вътрешен път е: 64
Матрица на минимумите:
       2      11      13      20      32      49      64
       0       7       9      16      28      43      58
       2       0       1       5      13      25      37
       2       2       0       3      10      22      33
       2       2       4       0       4      14      24
       2       2       4       5       0       6      16
       5       5       5       5       6       0       5
       5       5       6       6       6       6       0
Оптимално дърво за претърсване:
      d1
   d2
         d3
      d4
```

```
d5
   d6
      d7
```

Задачи за упражнение:

1. Ще работи ли предложеният алгоритъм, ако вместо честоти са зададени *тегла* на думите и е позволено да бъдат:
   а) отрицателни
   б) рационални
   в) реални
2. Има ли други оптимални решения на задачата за дадените конкретни входни данни освен дървото от *фигура 8.2.5.*?
3. Да се намерят всички оптимални решения.
4. Да се сравнят реализациите на решенията на задачата за умножение на матрици (*виж 8.2.3.*), триангулация на многоъгълник (*виж 8.2.4.*) и построяване на оптимално двоично дърво за претърсване.

8.2.6. Най-дълга обща подредица

Ще разгледаме още една класическа задача от областта на динамичното оптимиране. Дадени са две числови редици $X = (x_1, x_2, ..., x_m)$ и $Y = (y_1, y_2, ..., y_n)$. Искаме да намерим най-дългата редица $Z = (z_1, z_2, ..., z_k)$, която е подредица на X и Y едновременно. Ще считаме, че дадена числова редица Z е подредица на X, ако Z може да бъде получена чрез премахване на нула или повече члена на X. Разместване на членове не се допуска. Бихме могли да дефинираме понятието и по-строго.

Дефиниция 8.2. Ще казваме, че редицата $Z = (z_1, z_2, ..., z_k)$ е *подредица* на $X = (x_1, x_2, ..., x_m)$, ако съществува строго растяща редица $i_1 < i_2 < ... < i_k$, за която $x_{i_j} = z_j$, $j = 1, 2, ..., k$, $1 \leq i_1 < i_k \leq n$.

Да вземем като пример две редици от символи (ще изписваме отделните символи без запетаи и долепени един до друг): $X = aaaabbca$ и $Y = baadcaba$. Тогава редиците *a*, *aaa*, *abca* и *ba* са подредици на X, а *dca*, *bad* и *aaa* са подредици на Y. Редиците *a*, *aa* и *ab* са общи подредици за X и Y, но не са оптимални, тъй като съществува обща подредица с дължина 4, например *aaaa*. Тя обаче също не е оптимална, тъй като съществува още по-дълга обща подредица, а именно *aaaba*, която е едно възможно решение на задачата в случая.

Не е трудно да се убедим, че директният подход: генериране на всички подредици на X и проверка за всяка от тях дали е подредица и на Y, се справя със задачата само за достатъчно къси редици — броят на подредиците на X е $2^{|X|}$ (в нашия случай 2^m). Да означим с $F(i,j)$ дължината на най-дългата обща подредица на $X_i = (x_1, x_2, ..., x_i)$ и $Y_j = (y_1, y_2, ..., y_j)$, $1 \leq i \leq m$, $1 \leq j \leq n$. Тогава очевидно, ако $x_i = y_j$, то е в сила равенството $F(i,j) = F(i-1,j-1) + 1$. Действително, в такъв случай най-дългата обща подредица Z_{ij} на X_i и Y_j задължително съдържа като последен член x_i (или y_j, което е същото). В случай, че $x_i \neq y_j$, е ясно, че Z_{ij} не може да завършва едновременно на x_i и y_j. Ако Z_{ij} не завършва на x_i, то Z_{ij} е най-дълга обща подредица и на $X_{i-1} = (x_1, x_2, ..., x_{i-1})$ и Y_j, т. е. $Z_{ij} = Z_{i-1,j}$. Оттук непосредствено следва равенството: $F(i,j) = F(i-1,j)$. Ако пък Z_{ij} не завършва на y_j, то Z_{ij} е най-дълга обща подредица и на X_i и $Y_{j-1} = (y_1, y_2, ..., y_{j-1})$, т. е. $Z_{ij} = Z_{i,j-1}$ и оттук $F(i,j) = F(i,j-1)$. Ако Z_{ij} не завършва нито на x_i, нито на y_j, то следва, че Z_{ij} е най-дълга обща подредица на X_{i-1} и Y_{j-1}, т. е. $Z_{ij} = Z_{i-1,j-1}$, откъдето $F(i,j) = \max(F(i-1,j),F(i,j-1))$. В граничния случай, когато едната редица е празна, най-дългата обща подредица също е празна. Така достигаме до следната рекурентна формула:

$$F(i,j)=\begin{cases}0 & i=0\,или\,\,j=0\\ F(i-1,j-1)+1 & i>0,j>0,x_i=y_j\\ \max\left(F(i-1,j),F(i,j-1)\right) & i>0,j>0,x_i\neq y_j\end{cases}$$

Очевидно, директното прилагане на горната формула е неудачно, тъй като оптималните решения се препокриват. Освен това от разсъжденията по-горе следва, че оптималното решение има оптимална подструктура. Същевременно броят на различните подзадачи е $m*n$. Всички предпоставки за прилагане на динамично оптимиране са налице.

```
#include <stdio.h>
#include <string.h>
#define MAXN 100
#define MAX(a,b) (((a)>(b)) ? (a) : (b)) /* Връща по-големия аргумент */

char F[MAXN][MAXN];                     /* Целева функция */

const char x[MAXN] = "acbcacbcaba";     /* Първа редица */
const char y[MAXN] = "abacacacababa";   /* Втора редица */

/* Намира дължината на най-дългата обща подредица */
unsigned LCS_Length(void)
{ unsigned i, j, m, n;
  m = strlen(x);  /* Дължина на първата редица */
  n = strlen(y);  /* Дължина на втората редица */

  /* Начално инициализиране */
  for (i = 1; i <= m; i++)
    F[i][0] = 0;
  for (j = 0; j <= n; j++)
    F[0][j] = 0;
  /* Основен цикъл */
  for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
      if (x[i - 1] == y[j - 1])
        F[i][j] = F[i - 1][j - 1] + 1;
      else
        F[i][j] = MAX(F[i - 1][j], F[i][j - 1]);
  return F[m][n];
}

int main(void) {
  printf("\nДължина на най-дългата обща подредица: %u", LCS_Length());
  return 0;
}
```

🖫 lcs1.c

От кода на горната програма непосредствено се вижда, че сложността на предложения алгоритъм е $\Theta(mn)$. На дневен ред идва следващият проблем: намиране на една конкретна редица, подредица на дадената. Оказва се възможно използването на вече отработената техника на въвеждане на допълнителен масив и запазване на допълнителна информация на всяка стъпка. Ако се върнем към разсъжденията от по-горе, ще видим, че всъщност са налице три различни случая (последните два всъщност могат да възникнат едновременно), а именно:

1) $x_i = y_j$, при което следва, че търсената редица Z_{ij} завършва на x_i, т. е. извършено е присвояването `F[i][j] = F[i-1][j-1] + 1;`
2) $x_i \neq y_j$, като `F[i-1][j] ≥ F[i][j-1]`, тогава е извършено присвояването `F[i][j] = F[i-1][j]`, което означава, че x_i не е включено в редицата Z_{ij}

3) $x_i \neq y_j$, като `F[i-1][j] ≤ F[i][j-1]`, тогава е извършено присвояването `F[i][j] = F[i][j-1]`, което означава, че y_j не е включено в редицата Z_{ij}

Ще въведем три стойности: `UPLEFT`, `UP` и `LEFT`, съответстващи на случаи 1), 2) и 3) съответно и указващи в каква посока сме се придвижили в таблицата. На всяка стъпка на `b[i][j]` ще присвояваме съответна стойност измежду посочените три (виж `LCS_Length()` по-долу). С помощта на натрупаната информация по-късно ще бъде възможно да възстановим една възможна търсена редица по тривиален начин. Предложената функция `printLCS()` предлага елементарен итеративен подход, при което търсената оптимална редица се получава в обратен ред. Ако искаме да я получим в правилния ред, трябва да използваме стек в явен или неявен вид (чрез механизма на рекурсията). Функцията `printLCS2()` предлага една възможна реализация на втория подход. При по-внимателен поглед се вижда, че всъщност масивът `b[][]` изобщо не е необходим и редицата може да бъде възстановена само въз основа на информацията в `F[][]` (виж `printLCS3()`).

```
#include <stdio.h>
#include <string.h>
#define MAX     100
#define LEFT    1
#define UP      2
#define UPLEFT  3
char F[MAX][MAX];                    /* Целева функция */
char b[MAX][MAX];                    /* Указател към предходен елемент */
const char x[MAX] = "acbcacbcaba";   /* Първа редица */
const char y[MAX] = "abacacacababa"; /* Втора редица */

/* Намира дължината на най-дългата обща подредица */
unsigned LCS_Length(void)
{ unsigned i, j, m, n;
  m = strlen(x);  /* Дължина на първата редица */
  n = strlen(y);  /* Дължина на втората редица */
  /* Начално инициализиране */
  for (i = 1; i <= m; i++)
    F[i][0] = 0;
  for (j = 0; j <= n; j++)
    F[0][j] = 0;
  /* Основен цикъл */
  for (i = 1; i <= m; i++) {
    for (j = 1; j <= n; j++) {
      if (x[i - 1] == y[j - 1]) {
        F[i][j] = F[i - 1][j - 1] + 1;
        b[i][j] = UPLEFT;
      }
      else if (F[i - 1][j] >= F[i][j - 1]) {
        F[i][j] = F[i - 1][j];
        b[i][j] = UP;
      }
      else {
        F[i][j] = F[i][j - 1];
        b[i][j] = LEFT;
      }
    }
  }
  return F[m][n];
}
/* Намира една възможна максимална обща подредица (обърната) */
void printLCS(void)
{ unsigned i = strlen(x),
```

```
          j = strlen(y);
  while (i > 0 && j > 0)
    switch (b[i][j]) {
      case UPLEFT:  printf("%c", x[i - 1]); i--; j--; break;
      case UP:      i--; break;
      case LEFT:    j--;
    }
}

/* Намира една възможна максимална обща подредица */
void printLCS2(unsigned i, unsigned j)
{
  if (0 == i || 0 == j)
    return;
  if (UPLEFT == b[i][j]) {
    printLCS2(i - 1, j - 1);
    printf("%c", x[i - 1]);
  }
  else if (UP == b[i][j])
    printLCS2(i - 1, j);
  else
    printLCS2(i, j - 1);
}

/* Намира една възможна максимална обща подредица */
void printLCS3(unsigned i, unsigned j)
{ if (0 == i || 0 == j)
    return;
  if (x[i - 1] == y[j - 1]) {
    printLCS3(i - 1, j - 1);
    printf("%c", x[i - 1]);
  }
  else if (F[i][j] == F[i - 1][j])
    printLCS3(i - 1, j);
  else
    printLCS3(i, j - 1);
}

int main(void) {
  printf("\nДължина на най-дългата обща подредица: %u", LCS_Length());
  printf("\nPrintLCS:  Максимална обща подредица (в обратен ред): ");
  printLCS();
  printf("\nPrintLCS2: Максимална обща подредица: ");
  printLCS2(strlen(x), strlen(y));
  printf("\nPrintLCS3: Максимална обща подредица: ");
  printLCS3(strlen(x), strlen(y));
  return 0;
}
```

🖫 lcs2.c

Резултат от изпълнението на програмата:

```
Дължина на най-дългата обща подредица: 9
printLCS:  Максимална обща подредица (в обратен ред): ababcacca
printLCS2: Максимална обща подредица: accacbaba
printLCS3: Максимална обща подредица: accacbaba
```

Забележка: Въпреки направеното подобрение паметта остава от порядъка на $\Theta(nm)$, което налага сериозно ограничение за дължините на входните редици. В случай, че търсим само дъл-

жината на максималната обща подредица, но не се интересуваме от самата подредица, можем да намалим паметта до $2.\min(m,n) + \Theta(1)$ и дори до $\min(m,n) + \Theta(1)$, изхождайки от съображението, че на всяка стъпка се извършва обръщение само към текущия и предходния ред на матрицата [*Cormen*,*Leiserson*,*Rivest–1997*]. По подобен начин постъпихме при задачата за раницата (*виж 8.2.1.*).

Задача за упражнение:

1. Да се реализира рекурсивен вариант на алгоритъма.
2. Да се реализира вариант на алгоритъма, който изисква памет:
   а) $2.\min(m,n) + \Theta(1)$
   б) $\min(m,n) + \Theta(1)$

8.2.7. Най-дълга ненамаляваща подредица

Да разгледаме още една задача, свързана с числови редици. Нека е дадена числовата редица $X = (x_1, x_2, \ldots, x_n)$. Търсим най-дългата ненамаляваща подредица на X.

Дефиниция 8.3. Една числова редица $X = (x_1, x_2, \ldots, x_n)$ ще наричаме *ненамаляваща*, ако всеки неин елемент е по-голям или равен на предхождащия го (с изключение на първия, който няма предшественик), т. е. ако са в сила неравенствата $x_1 \leq x_2 \leq \ldots \leq x_n$.

Ще решим задачата по три различни начина. Преди да се опитаме да съставим някакъв алгоритъм, да се опитаме да я анализираме. Да означим с $LNS(k)$ (от англ. *Longest Non-decreasing Subsequence*) най-дългата ненамаляваща подредица на редицата $X_k = (x_1, x_2, \ldots, x_k)$, съставена от първите k елемента на X. Ще разсъждаваме индуктивно. Пресмятането на $LNS(1)$ е тривиално. Да предположим, че сме успели да намерим $LNS(k)$ и искаме да намерим $LNS(k+1)$. Оказва се, че това не може да стане директно без запазване на някаква допълнителна информация на предходните стъпки. Да разгледаме редицата (10, 40, 20, 30). При разглеждането на 30 е необходимо да знаем не просто *някаква* редица $LNS(3)$, а *именно* (10, 20). Другата възможност (10, 40) очевидно не ни води до решение. Оттук стават ясни две неща: първо, оптималното решение има оптимална подструктура, и второ, нужно е да се съхрани не коя да е оптимална редица, а тази, завършваща на най-малко число.

Да разгледаме друг пример: (1, 2, 5, 3, 4). Оказва се, че познаването на най-дългата ненамаляваща редица, завършваща на минимално число, не ни носи достатъчно информация, за да намерим $LNS(k+1)$. Забележете, че в този случай (в момента на разглеждане на елемента 3) се променя не *дължината* на най-дългата ненамаляваща подредица, а нейният *последен член*, който се заменя с по-малък. Т. е. трябва да запомним оптималната редица с дължина, с единица по-малка от дължината на $LNS(k)$. Ако не го заменим, няма да можем да построим редицата (1, 2, 3, 4), която е оптимална. Ако сега разгледаме редицата (1, 5, 6, 2, 3, 4), ще видим, че по аналогичен начин възниква нужда от съхраняване на най-добрата подредица с дължина, с 2 по-малка от тази на $LNS(k)$. Разсъждавайки индуктивно, получаваме, че е нужно да съхраняваме всички подредици, завършващи на най-малкото възможно число. Така $LNS(k)$ става двуаргументна функция: $LNS(k,s)$, съхраняваща последния най-малък елемент от всяка подредица с дължина s, чиито елементи са измежду тези на $X_k = (x_1, x_2, \ldots, x_k)$, $1 \leq s \leq k \leq n$.

Стойностите на $LNS(k,s)$ могат да се намерят итеративно. Разглеждаме членовете на редицата X един по един. Нека разглеждаме x_k за някое k, $1 \leq k \leq n$. Последователно опитваме да поставим x_k като първи, втори, трети, ..., s-и елемент на някоя от оптималните редици. Необходимо условие за това е $LNS(k, s-1) \leq x_k \leq LNS(k, s)$ за някое s, $1 \leq s \leq k \leq n$. В този случай сме открили по-малък последен елемент за редица с дължина s — заменяме s-ия ѝ член с x_k. В случай, че x_k се окаже по-голям от най-големия елемент на най-дългата известна ни редица, го добавяме в края ѝ. Ако пък се окаже по-малък от най-малкия елемент, единствената възможност е да състави тривиалната подредица, съдържаща само него.

С подходящо предварително инициализиране на все още непресметнатите стойности можем да си спестим някои от горните разсъждения. Идеята е проста и следващият псевдокод показва как става това:

```
/* Начално инициализиране */
for (k = 0; k <= n; k++) {
  for (s = 1; s <= n; s++)
    LNS[k][s] = +∞;
  LNS[k][0] = -∞;
}

/* Основен цикъл */
for (k = 1; k <= n; k++)
  for (s = 1; s <= n; s++)
    if (LNS[k-1][s-1] ≤ x[k] ≤ LNS[k-1][s])
      LNS[k][s] = x[k];
    else
      LNS[k][s] = LNS[k-1][s];
```

По-долу следва пълна реализация. В процеса на пресмятане ще пазим максималната стойност на s, за която сме пресметнали $LNS(k,s)$. За целта ще поддържаме специална променлива r, която функцията `LNS_Length()` ще връща като резултат. От горния алгоритъм лесно се вижда как може да бъде намерена една конкретна максимална по дължина ненамаляваща подредица на X. Функцията `LNS_Print()` е итеративна и извежда подредицата в обратен ред. `LNS_Print2()` е неин рекурсивен вариант, който извежда подредицата в правилния ред.

```
#include <stdio.h>

#define MAX 100

const int x[MAX] = {100, 10, 15, 5, 25, 22, 12, 22}; /* Редица */
                   /* Нулевият елемент на x[] не се използва! */
const unsigned n = 7; /* Брой елементи в редицата */

int LNS[MAX][MAX];    /* Целева функция */

/* Намира дължината на най-дългата ненамаляваща подредица */
unsigned LNS_Length(void)
{ unsigned i, j, r;

  /* Начално инициализиране */
  for (i = 0; i <= n; i++) {
    for (j = 1; j <= n; j++)
      LNS[i][j] = MAX + 1;
    LNS[i][0] = -1;
  }

  /* Основен цикъл */
  r = 1;
  for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
      if (LNS[i - 1][j - 1] <= x[i] && x[i] <= LNS[i - 1][j]
          && LNS[i - 1][j - 1] <= LNS[i - 1][j]) {
        LNS[i][j] = x[i];
        if (r < j)
          r = j;
      }
      else
       LNS[i][j] = LNS[i - 1][j];
```

```
    }
  }

  return r;
}

/* Извежда най-дългата ненамаляваща подредица (в обратен ред) */
void LNS_Print(unsigned j)
{ unsigned i = n;
  do {
    if (LNS[i][j] == LNS[i - 1][j])
      i--;
    else {
      printf("%d ", x[i]);
       j--;
    }
  } while (i > 0);
}

/* Извежда най-дългата ненамаляваща подредица */
void LNS_Print2(unsigned i, unsigned j)
{ if (0 == i) return;
  if (LNS[i][j] == LNS[i - 1][j])
    LNS_Print2(i - 1, j);
  else {
    LNS_Print2(i, j - 1);
    printf("%d ", x[i]);
  }
}

int main(void) {
  unsigned len = LNS_Length();
  printf("\nДължина на най-дългата ненамаляваща подредица: %u", len);
  printf("\nПодредицата (обърната): "); LNS_Print(len);
  printf("\nПодредицата: "); LNS_Print2(n, len);
  return 0;
}
```

🖫 lns1.c

Резултат от изпълнението на програмата:

```
Дължина на най-дългата ненамаляваща подредица: 4
Подредицата (обърната): 22 22 15 10
Подредицата: 10 15 22 22
```

И така, получихме алгоритъм със сложност $\Theta(n^2)$ както по време, така и по памет. По-долу ще се опитаме да подобрим и двата показателя. Да започнем с паметта. Бихме могли значително да я намалим: ако започнем пресмятането отзад-напред, за намиране на дължината на оптималната редица ще бъде достатъчно за всеки елемент да пазим само максималната дължина на редица, на която той е начало. Така използваната памет става $\Theta(n)$ при времева сложност $\Theta(n^2)$.

Разглеждаме отзад-напред елементите на редицата. Нека текущият елемент е x_i. Искаме да намерим дължината на максималната редица с начало x_i. За целта преглеждаме всички елементи x_j, разположени след него в редицата ($i < j \leq n$). Ако x_i е по-малък или равен на x_j, можем да го сложим непосредствено преди максималната редица с начало x_j. Ако дължината на максималната редица с начало x_j е била l, то очевидно дължината на новата редица, с начало x_i, ще бъде $l + 1$. Така, ако искаме да построим максималната редица с начало x_i, трябва да изберем това x_j, за което l е максимално. Ако няма такова x_j, т. е. x_i е последният елемент в редицата или всички

елементи след него са строго по-малки от него, то единствената възможност е да построим тривиалната ненамаляваща редица, съдържаща само x_i. Нека обозначим с $LNS(i)$ дължината на максималната ненамаляваща подредица с начало x_i. Можем да пресметнем $LNS(i)$ по следната формула:

$$LNS(i) = \begin{cases} 1 + \max\limits_{j=i+1, x_j \geq x_i}^{n} (LNS(j)) & \text{ако } \exists j : x_j \geq x_i, 1 \leq i < j \leq n \\ 1 & \text{иначе} \end{cases}$$

Подробностите по реализацията читателят може да види в предложения по-долу програмен код (виж `LNS_Length()`). Ако желаем да възстановим и една конкретна оптимална редица, се оказва достатъчно да въведем още един масив `next[]`, който да съхранява за всеки елемент x_i непосредствено следващия го елемент в оптималната редица с начало x_i. Забележете, че този път редицата се възстановява в правилния ред, без нужда от рекурсия за обръщане на последователността (виж `LNS_Print()`).

```
#include <stdio.h>
#define MAX 100

const int x[MAX] = {100, 10, 15, 5, 25, 22, 12, 22}; /* Редица */
                      /* Нулевият елемент на x[] не се използва! */
const unsigned n = 7; /* Брой елементи в редицата */

unsigned LNS[MAX];    /* Дължина на максималната редица с начало x[i] */
unsigned next[MAX];   /* Индекс на следващ елемент*/

/* Намира дължината на най-дългата ненамаляваща подредица */
unsigned LNS_Length(unsigned *start)
{ unsigned i, j;
  unsigned l;  /* В момента на разглеждане на xi,
               /* l е дължината на максималната подредица с начало xj: */
               /*    1) i < j <= n и */
               /*    2) xi <= xj     */
  unsigned len = 0; /* Максимална (за момента) дължина на подредица */
  for (i = n; i >= 1; i--) {
    for (l = 0, j = i + 1; j <= n; j++)
      if (x[j] >= x[i] && LNS[j] > l) {
        l = LNS[j];
        next[i] = j;
      }
    LNS[i] = l + 1;
    if (LNS[i] > len) {
      len = LNS[i];
      *start = i;
    }
  }
  return len;
}

/* Извежда най-дългата ненамаляваща подредица */
void LNS_Print(unsigned start)
{ for (; LNS[start] >= 1; start = next[start])
    printf(" %d", x[start]);
}

int main(void) {
  unsigned start;
```

```
  printf("Дължина на най-дългата ненамаляваща подредица: %u\n",
          LNS_Length(&start));
  printf("Подредицата: "); LNS_Print(start);
  return 0;
}
```

🖫 lns2.c

Какво постигнахме? Сложност по памет $\Theta(n)$ при сложност по време $\Theta(n^2)$. Сега ще съсредоточим усилията си върху времевата сложност. Оказва се възможно тя да се подобри до $\Theta(n.\log_2 n)$.

Да означим с $LNS(k)$ минималния елемент x_i на X, който може да стои на позиция k в най-дългата ненамаляваща подредица. Очевидно $1 \le k \le i \le n$ и съществуват k–1 елемента преди него в оптималната редица, т. е. съществува ненамаляваща подредица на $X_{i-1} = \{x_1, x_2, ..., x_{i-1}\}$ с дължина k–1, последният член на която е по-малък или равен на x_i. Следва изрично да се подчертае, че $LNS(.)$ не съдържа търсената максимална ненамаляваща подредица. Нека k е максималната моментна дължина на ненамаляваща подредица, т. е. максималната стойност, за която е пресметната стойността $LNS(k)$. Налице са три възможности за x_i.

1. $x_i < LNS(1)$. Тогава актуализираме: $LNS(1) = x_i$.
2. $x_i \ge LNS(k)$. Тогава актуализираме: $LNS(k+1) = x_i$.
3. $LNS(1) \le x_i < LNS(k)$. Търсим такова j $(1 < j \le k)$, за което $LNS(j-1) \le x_i < LNS(j)$, след което актуализираме $LNS(j) = x_i$.

Първите две стъпки се извършват за време $\Theta(1)$, а третата за $\Theta(n)$. При прилагане на двоично търсене времето за изпълнение на третата стъпка намалява до $\Theta(\log_2 n)$. Тъй като за всеки елемент x_i на X се изпълнява точно една от тези стъпки, за общата времева сложност на алгоритъма получаваме $\Theta(n.\log_2 n)$ при сложност по памет $\Theta(n)$ [*Шишков-1995*]. Следва реализация на алгоритъма.

```
unsigned LNS_Length(void)
{ unsigned i, r, k, l, med;
  for (LNS[1] = x[1], k = 1, i = 2; i <= n; i++) {
    if (x[i] < LNS[1])          /* случай 1 */
      LNS[1] = x[i];
    else if (x[i] >= LNS[k])  /* случай 2 */
      LNS[++k] = x[i];
    else {                      /* случай 3 */
      l = 1;
      r = k;                    /* двоично търсене */
      while (l < r - 1) {
        med = (l + r) / 2;
        if (LNS[med] <= x[i])
          l = med;
        else
          r = med;
      }
      LNS[r] = x[i];
    }
  }
  return k;
}
```

🖫 lns3.c

Задачи за упражнение:

1. Последната функция намира дължината на максималната ненамаляваща редица, но не и *самата* редица. Да се модифицира по подходящ начин така, че да намира една оптимална редица за време $\Theta(n)$.
2. Да се сравнят трите предложени алгоритъма. Кои от тях се основават на динамично оптимиране?
3. Да се намерят всички максимални ненамаляващи подредици. Кой от предложените алгоритми е най-подходящ за целта? Защо?

8.2.8. Сравнение на символни низове

Дадени са два символни низа s_1 и s_2. Целта е от първия низ s_1 да се получи вторият s_2, като се използва само ограничен набор от операции, а именно:

replace(i,x)	— замества i-ия ($1 \leq i \leq \|s_1\|$) символ на s_1 със символа x
insert(i,x)	— вмъква символа x на позиция i в s_1
delete(i)	— изтрива i-ия символ на s_1

Всички операции се извършват *само* върху първия низ, а вторият не може да се изменя. За всяка от операциите е дадена цена (естествено число) и търсим редица с минимална цена за трансформиране на s_1 в s_2 (сума от цените на участващите операции).

На пръв поглед задачата изглежда сложна. В действителност тя силно напомня задачата за най-дългата обща подредица (*виж 8.2.6.*). Тази аналогия ни навежда на мисълта да разгледаме символите отзад-напред. Да разгледаме последните символи на двата низа. Те биха могли или да съвпаднат, или да се различават. Във втория случай можем да заменим последния символ на s_1 с последния символ на s_2. Останалите възможности са изтриване или вмъкване. По същия начин можем да разсъждаваме за предходните символи.

Да се опитаме да формализираме горните разсъждения. Да въведем целевата функция $F(i,j)$, която ще ни дава минималната цена за уеднаквяване на началните последователности (префикси) $s_1[1]$, $s_1[2]$, ..., $s_1[i]$ и $s_2[1]$, $s_2[2]$, ..., $s_2[j]$. Нека c_r е цената за заместване на символ на s_1 със символ от s_2, c_i е цената за вмъкване на символ, и c_d — цената за премахване на символ.

Как да пресметнем $F(i,j)$ в общия случай? Да разгледаме следните възможности:

1. $s_1[i] = s_2[j]$. Тогава $F(i,j) = F(i-1,j-1)$.
2. $s_1[i] \neq s_2[j]$. Имаме няколко възможности:
   - $F(i,j) = F(i-1,j-1) + c_r$. Съответните символи се различават: правим заместване.
   - $F(i,j) = F(i,j-1) + c_i$. Вторият низ s_2 съдържа допълнителен символ, поради което го вмъкваме и в s_1.
   - $F(i,j) = F(i-1,j) + c_d$. Първият низ s_2 съдържа допълнителен символ, поради което го изтриваме.

В приложената програма са въведени цени на операциите: изтриване `COST_DELETE == 1` (c_d), вмъкване `COST_INSERT == 2` (c_i) и заместване `COST_REPLACE` (`COST_REPLACE == 0` (c_r), ако символите съвпадат, и `COST_REPLACE == 3` — иначе). В случай, че символите съвпадат, е налице случай 1). При несъвпадение стойността на $F(i,j)$ се пресмята като минимум на $F(i-1,j-1) + c_r$, $F(i,j-1) + c_i$ и $F(i-1,j) + c_d$.

Остава да въведем граничните условия и сме готови да пристъпим към реализация на алгоритъма. Стойността на $F(i,0)$ съответства на случая, когато разглежданата част от първия низ съдържа i символа, докато вторият е празен. Тъй като нямаме право да манипулираме втория низ, единствената възможност е да премахнем първите i символа от s_1. Цената за извършване на това е $F(i,0) = i.c_d$. Аналогично, $F(0,j)$ съответства на случая, когато първият низ е празен. В такъв случай трябва да се вмъкнат първите j символа на s_2, което има цена $j.c_i$. Така е налице всичко необходимо за реализирането на функцията `editDistance()`. Непосредствено от кода на програмата се вижда, че сложността на алгоритъма е $\Theta(nm)$, където n е дължината на s_1, а m — на s_2.

Както и при други подобни задачи въведената таблица `F[][]` съдържа достатъчно информация за възстановяване на едно конкретно решение за време $\Theta(n+m)$. Решението ще търсим като последователност от операции от описания тип: `INSERT`, `DELETE` и `REPLACE`. Следва да се отбележи, че операциите `INSERT` и `DELETE` променят дължината на s_1. Тъй като в процеса на търсене на решението ние не променяме s_1, а само *пресмятаме* цената и последствията от съответната промяна, решението, което ще получим, ще се състои от команди, ориентирани към *изходните* позиции в s_1. Това личи и от примера по-долу.

```
#include <stdio.h>
#include <string.h>

#define MAX                 100
#define COST_DELETE         1
#define COST_INSERT         2
#define COST_REPLACE(i, j)  ((s1[i] == s2[j]) ? 0 : 3)

#define min2(a, b) (((a) < (b)) ? (a) : (b))   /* Мин. на 2 аргумента */
#define min(a, b, c)      min2(min2(a,b), (c)) /* Мин. на 3 аргумента */

unsigned F[MAX+1][MAX+1];   /* Целева функция */
unsigned n1;                /* Дължина на първия низ */
unsigned n2;                /* Дължина на втория низ */

const char *s1="_abracadabra";  /* Низ 1 (първи символ без значение) */
const char *s2="_mabragabra";   /* Низ-цел (първи символ без значение) */

/* Намира разстоянието между два низа */
unsigned editDistance(void)
{ unsigned i, j;
  /* Инициализиране */
  for (i = 0; i <= n1; i++)
      F[i][0] = i * COST_DELETE;
  for (j = 0; j <= n2; j++)
      F[0][j] = j * COST_INSERT;
  /* Основен цикъл */
  for (i = 1; i <= n1; i++)
    for (j = 1; j <= n2; j++)
      F[i][j] = min(F[i - 1][j - 1] + COST_REPLACE(i, j),
                    F[i][j - 1] + COST_INSERT,
                    F[i - 1][j] + COST_DELETE);
  return F[n1][n2];
}

/* Извежда операциите по редактирането */
void printEditOperations(unsigned i, unsigned j)
{
  if (0 == j)
    for (j = 1; j <= i; j++)
        printf("DELETE(%u) ", j);
  else if (0 == i)
    for (i = 1; i <= j; i++)
        printf("INSERT(%u,%c) ", i, s2[i]);
  else if (i > 0 && j > 0) {
    if (F[i][j] == F[i - 1][j - 1] + COST_REPLACE(i, j)) {
      printEditOperations(i - 1, j - 1);
      if (COST_REPLACE(i, j) > 0)
            printf("REPLACE(%u,%c) ", i, s2[j]);
```

```
      }
      else if (F[i][j] == F[i][j - 1] + COST_INSERT) {
        printEditOperations(i, j - 1);
        printf("INSERT(%u,%c) ", i, s2[j]);
      }
      else if (F[i][j] == F[i - 1][j] + COST_DELETE) {
        printEditOperations(i - 1, j);
        printf("DELETE(%u) ", i);
      }
  }
}

int main(void) {
  n1 = strlen(s1) - 1;
  n2 = strlen(s2) - 1;
  printf("\nМинимално разстояние между двата низа: %u\n",
         editDistance());
  printEditOperations(n1, n2);
  return 0;
}
```

🖫 transform.c

Резултат от изпълнението на програмата:

```
Минимално разстояние между двата низа: 7
INSERT(1,m) DELETE(4) DELETE(5) REPLACE(7,g)
```

Горната програма може да се модифицира по подходящ начин така, че необходимата памет да се намали от $\Theta(nm)$ до $\Theta(m)$. Действително, ако извършваме пресмятанията по редове отляво-надясно, ще ни бъдат необходими най-много два реда от матрицата, а именно текущият и предходният. Оставяме на читателя да реализира този алгоритъм като леко упражнение. За съжаление, в този случай съхранената информация се оказва недостатъчна за намиране на конкретна последователност от операции, уеднаквяваща двата низа. В действителност съществува алгоритъм, основан на *разделяй и владей*, който намира такава последователност за време $\Theta(mn)$ и изисква памет $\Theta(m)$. Всъщност необходимата памет е $\Theta(\min(n,m))$, тъй като можем да пресмятаме стойностите на матрицата както по редове, така и по колони. Предлагаме на читателя да се опита да реализира такъв алгоритъм.

Хиршберг предлага друг алгоритъм, основан на *разделяй и владей*, при който, въпреки линейната памет, се оказва възможно да се възстанови едно конкретно оптимално решение. Идеята е при пресмятане на стойността на $F(n,m)$ да се запазва редът с номер $n/2$ и да се запомня кой негов елемент x е бил използван в оптималното решение за $F(n,m)$. Така, задачата се свежда до намиране на оптимален път от $F(1,1)$ до $F(n/2,x)$ и от $F(n/2,x)$ до $F(n/2,m)$. Тези подзадачи се решават рекурсивно. В резултат се получава алгоритъм със сложност по време $\Theta(nm)$. [*Skiena-1997*]

Настоящата задача е добре изследвана и може да се появи в най-разнообразни формулировки, като в повечето случаи решението на вариантите почти не се различава от предложеното по-горе. Едно от най-съществените ѝ приложения е при разработването на програми за автоматично проверяване на правописа на текст. Всяка такава програма трябва освен да умее да разпознава неправилно изписани думи (т. е. липсващи в речника ѝ), така и да подскаже на потребителя правилния правопис. За целта е необходимо да бъдат намерени тези думи от речника, които са най-близо до погрешното изписване, т. е. които са на най-малко "разстояние" от дума в него.

Подобна техника се използва и от програмите за автоматично разпознаване на сканиран печатен текст. Често текстът, който се сканира, не е достатъчно ясен или използва по-особен шрифт, което затруднява работата на програмата, поради което тя може да прибегне до помощта на речник, в който се търсят най-близките низове. Така може да се оценява качеството на програмата за разпознаване, както и типът на най-честите грешки. Във втория случай следва да се про-

ведат няколко пресмятания с различни тегла на операциите. Вмъкването и изтриването на символи се свързват с грешки при определяне границите между отделните букви (т. е. причинени от лошо качество на входния текст), докато замяната на буква се счита за грешка в разпознаващата програма. [*Skiena-1997*]

Друг пример за приложение на метода е в биологията при изследване на близостта на молекули на ДНК на различни биологични видове. Според някои автори корените на задачата могат да се търсят в старите *smart терминали*. Те умеят да преобразуват даден низ, изобразен върху екрана на компютъра чрез серия от елементарни операции, дадени в *таблица 8.2.8.* (в началото курсорът е позициониран върху първия символ на изходния низ):

Елементарни операции	Действие
ADVANCE	Премества курсора на една позиция надясно.
DELETE	Изтрива символа под курсора и се премества върху следващия символ вдясно.
REPLACE	Замества символа под курсора с друг символ, като курсорът остава на място.
INSERT	Вмъква нов символ преди този, върху който е позициониран курсорът. Курсорът не се мести.
KILL	Изтрива всички символи до края на низа, започвайки от този, върху който е позициониран курсорът включително. Тази операция може да бъде само финална.

Таблица 8.2.8. Операции при *smart* терминали.

Времето за извършване на тези операции се различава и с цел подобряване скоростта на работа на терминалите, които си комуникират с такива команди, се поставя задачата за намиране на най-бързия начин (за конкретния тип *smart* терминал) за получаване на низа-цел по зададен изходен низ. Тази задача може да се разглежда като обобщение на разгледания по-горе вариант. Разликата се състои в това, че придвижването един символ надясно също може да има цена, както и че могат да премахват символите от края на думата наведнъж (това обаче си има съответната цена). Оставяме на читателя сам да направи съответната модификация на горната програма.

В случай, че искаме да изобразим изходния низ като подниз на низа-цел, ще трябва да инициализираме $F(i,0) = 0$, вместо $F(i,0) = c_d$, тъй като премахването на начални символи ще може да се извърши безплатно. Ако пък дадем на операцията `REPLACE` по-голяма цена от сумата от цените на `DELETE` и `INSERT`, ще можем да си гарантираме, че операцията `REPLACE` на практика никога няма да се извършва. Така, задачата се свежда до намиране на най-дълга обща подредица (*виж 8.2.6.*).

Задачи за упражнение:

1. Да се състави алгоритъм за решаване на задачата за сравнение на символни низове за време $\Theta(nm)$ и памет $\Theta(\min(m,n))$.
2. Да се реализира алгоритъмът на Хиршберг.
3. Като се използва информацията, събрана от алгоритъма, да се възстанови една възможна оптимална последователност от операции, уеднаквяваща двата низа, но ориентирана към *текущите*, а не към изходните позиции в низовете.
4. Да се модифицира по подходящ начин програмата така, че да решава задачата за *smart* терминалите.
5. Да се покаже, че при подходящи тегла задачата се свежда до задачата за намиране на най-дълга обща подредица (*виж 8.2.6.*).

8.2.9. Задача за разделянето

Научен екип трескаво търси информация по даден проблем. За целта трябва да се претърси дълъг библиотечен рафт, съдържащ n наредени една след друга книги, всяка от които съдържа s_i страници ($1 \leq i \leq n$). С претърсването се е заела група от k научни сътрудника. Наредбата на книгите в библиотеката е изключително важна и не бива да се нарушава. За да се избегнат евентуални проблемите в това отношение, сътрудниците решават да си поделят рафта на области, без да разместват книгите. Всички членове на групата са опитни научни сътрудници и прегледжането на една страница им отнема едно и също фиксирано време. Тъй като времето за приключване на проекта е изключително критично, групата решава да си подели рафта по такъв начин, че претърсването да приключи за минимално време. За целта екипът преглежда последователно книгите и записва на лист хартия броя на страниците за всяка от тях, в резултат на което се получават редица от числа $s_1, s_2, \ldots, s_n$. Така задачата им се свежда до подходящо разбиване на тази редица на k подпоследователности така, че максимумът на сумите от броя на страниците, пресметнат по групите, да бъде минимален.

Веднага можем да посочим един възможен евристичен похват: намираме средната стойност $\frac{1}{k}\sum_{i=1}^{n} s_i$ на сумата за всяка група, след което се опитваме да поставим разделителите така, че максимално да се приближим до нея. За съжаление, при някои входни данни този подход няма да даде оптимален резултат, тъй като не изследва систематично всички възможности.

Да видим как можем да решим задачата с помощта на динамично оптимиране. Ще разсъждаваме по следния начин: Да разделим изходната последователност $s_1, s_2, \ldots, s_n$ на две подпоследователности на позиция i ($1 \leq i \leq n$): получаваме $s_1, s_2, \ldots, s_i$ и $s_{i+1}, s_{i+2}, \ldots, s_n$. Втората последователност ще считаме за окончателно разделена и ще я зачислим на един от научните сътрудници, а първата ще разделим на k–1 части. За целта в нея трябва да бъдат поставени k–2 разделителя. За лявата част можем да разсъждаваме по същия начин: поставяме разделител на някоя позиция j ($1 \leq j \leq i$). Едната част ($s_{j+1}, s_{j+2}, \ldots, s_i$) считаме за окончателно разделена, докато в другата ($s_1, s_2, \ldots, s_j$) следва да се поставят още k–3 разделителя. Така успяхме да разделим задачата на две подзадачи, втората от които се решава директно, а първата е задача с по-малка размерност. Оттук директно получаваме съответно рекурсивно решение.

Ще се опитаме да обобщим и формализираме горните разсъждения. Да въведем двуаргументна целева функция $F(n,k)$, която ще ни дава минималната цена по всевъзможните разделяния на първите n книги в k групи. Цената ще дефинираме като максимум по групите на сумата от елементите в групата. Или формално:

$$F(i, j) = \min_{l=1}^{i} \max\left(F(l, j-1), \sum_{r=l+1}^{i} s_r \right), \ 1 \leq i \leq n, \ 1 \leq j \leq k$$

Горната формула показва, че предмети от $l + 1$ до i са в една група, а останалите се разбиват по оптимален начин. Остава да се наложат подходящи гранични условия по двата аргумента, а именно:

$$F(1,j) = s_1, \ 1 \leq j \leq k \text{ (само един елемент)}$$
$$F(i,1) = s_1 + s_2 + \ldots + s_i, \ 1 \leq i \leq n \text{ (само една група)}$$

Сега вече сме готови да реализираме съответна итеративна програма. Преди да пристъпим към това, нека се опитаме да оценим времевата сложност на алгоритъма. Основна характеристика на динамичното оптимиране е, че $F(i,j)$ се пресмята *точно веднъж* за всяка възможна стойност на аргументите i ($1 \leq i \leq n$) и j ($1 \leq j \leq k$). Броят на различните комбинации от тези стойности е nk. Остава да оценим времето за пресмятане на стойността на функцията за една фиксирана двойка стойности (i,j) при предположението, че всички стойности на функцията за по-малки стойности на аргументите вече са били пресметнати. Тогава от формулата по-горе се вижда, че времето за пресмятане се определя от времето за намиране на сумата $s_{l+1} + s_{l+2} + \ldots + s_i$. На пръв поглед

изглежда, че за това ще бъде необходимо време от порядъка на $\Theta(n)$. В действителност при използване на *кумулативен* масив, съдържащ всевъзможните префиксни суми на елементите на s_i, можем да пресметнем сумата директно. Така максимумът се пресмята за време $\Theta(1)$. Вземайки предвид минимума от формулата, получаваме, че пресмятането на една конкретна стойност отнема време $\Theta(n)$. Окончателно за решаването на задачата получаваме $\Theta(kn^2)$.

Целевата функция $F(i,j)$ ще представим като двумерен масив `F[][]`. Ще въведем и втори масив `b[][]`, който ще съдържа индексите, за които се е получила съответната оптимална стойност на функцията. Това ще ни позволи да възстановим едно възможно оптимално решение по добре познатия ни начин. Подробностите по реализацията се виждат от кода на програмата.

```
#include <stdio.h>
#define MAXN 80
#define INFINITY (unsigned long)(-1)
#define MAX(a,b) (((a)>(b)) ? (a) : (b)) /* Връща по-големия аргумент */
unsigned long p[MAXN];       /* Префиксни суми */
unsigned long F[MAXN][MAXN]; /* Целева функция */
unsigned long b[MAXN][MAXN]; /* За възстановяване на решението */

/* Редица (нулевият елемент не се ползва) */
const unsigned s[MAXN] = {0,23,15,89,170,25,1,86,80,2,27};
const unsigned n = 10;  /* Брой елементи в редицата */
const unsigned k = 4;   /* Брой групи */

/* Извършва оптимално разделяне на k групи */
long doPartition(unsigned k)
{ unsigned i, j, l, m;

  /* Пресмятане на префиксните суми */
  for (p[0] = 0, i = 1; i <= n; i++)
      p[i] = p[i - 1] + s[i];

  /* Установяване на граничните условия */
  for (i = 1; i <= n; i++)
      F[i][1] = p[i];
  for (j = 1; j <= k; j++)
      F[1][j] = s[1];

  /* Основен цикъл */
  for (i = 2; i <= n; i++)
    for (j = 2; j <= k; j++)
      for (F[i][j] = INFINITY, l = 1; l <= i - 1; l++)
          if ((m = MAX(F[l][j - 1], p[i] - p[l])) < F[i][j]) {
          F[i][j] = m;
          b[i][j] = l;
        }
  return F[n][k];
}

void print(unsigned from, unsigned to)
{ unsigned i;
  printf("\n");
  for (i = from; i <= to; i++)
      printf("%u ", s[i]);
}

void printPartition(unsigned n, unsigned k)
{ if (1 == k)
```

```
    print(1, n);
  else {
    printPartition(b[n][k], k - 1);
    print(b[n][k] + 1, n);
  }
}

int main(void) {
  printf("\nМаксимална сума в някоя от групите: %lu", doPartition(k));
  printPartition(n, k);
  return 0;
}
```

🖫 partitio.c

Резултат от изпълнението на програмата:

```
Максимална сума в някоя от групите: 170
23 15 89
170
25 1 86
80 2 27
```

Задачи за упражнение:

1. Нужен ли е непременно масивът `b[][]` за възстановяване на едно възможно решение?
2. Да се намерят всички решения.
3. Да се даде пример, когато предложената в началото на параграфа алгоритъм няма да работи правилно.

8.3.Неоптимизационни задачи

Въпреки името си (*оптимиране*, т. е. търсене на оптималното измежду множества от решения на дадена задача) динамичното оптимиране е техника с широка сфера на приложение, която може да се използва и при решаване на *неоптимизационни* задачи. Това ни навежда на мисълта, че може би терминът *динамично програмиране* е по-подходящ от *динамично оптимиране*.

В настоящия раздел ще бъдат разгледани няколко такива примера. Става въпрос за намиране стойността на функция, зададена с рекурентна формула, намиране броя на различни комбинаторни конфигурации и проверка за удовлетворяване на някакви условия (принадлежност на изречение към контекстно свободен език, зададен по дадена граматика; проверка дали дадена стойност може да бъде получена).

Характерна особеност тук е, че задачите могат да се разглеждат като съставени от припокриващите се подзадачи с по-малка размерност, т. е. налице е едното необходимо условие за прилагане на динамично оптимиране. За второто необходимо условие — оптимална подструктура на оптималното решение, не би могло да се говори, тъй като задачата не е оптимизационна. Динамичното оптимиране в този случай е полезно дотолкова, доколкото позволява спестяване на многократното решаване на едни и същи подзадачи с по-малка размерност. По-долу читателят ще има възможност да се увери, че прилагането на динамично оптимиране в случай на неоптимизационни задачи може да бъде не по-малко ефикасно от прилагането му към оптимизационни.

8.3.1. Числа на Фибоначи

Отново се сблъскваме с добре познатите ни числа на Фибоначи (разгледани обстойно в *параграф 1.2.2.*). Да припомним класическия неефективен рекурсивен алгоритъм:

```
unsigned long fib(unsigned n)
```

```
{ if (n < 2)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

🖫 fibrec.c

Времевата сложност на горното решение е експоненциална и в настоящия параграф ще покажем как, запазвайки елегантността на рекурсията, можем да получим ефективно (линейно) решение. Последното става за сметка на необходима допълнителна памет: За целта трябва да се пази таблица с всички вече пресметнати стойности на функцията и да се пристъпва към изчисляването на дадена стойност, само ако все още не е попълнена — *memoization*. Така всяка стойност ще се пресмята точно по веднъж. Оттук и сложността на алгоритъма ще бъде линейна — имаме n стойности за попълване.

```
#include <stdio.h>
#include <string.h>
#define MAX 256

const unsigned n = 10; /* търсим 10-тото число на Фибоначи */
unsigned long m[MAX + 1];

/* Бърз рекурсивен линеен вариант, запаметяващ вече пресметнатото */
unsigned long fibMemo(unsigned n)
{ if (n < 2)
    m[n] = n;
  else if (0 == m[n])
    m[n] = fibMemo(n - 1) + fibMemo(n - 2);
  return m[n];
}

int main(void) {
  memset(m, 0, MAX * sizeof(*m));
  printf("\n%u-тото число на Фибоначи е: %lu", n, fibMemo(n));
  return 0;
}
```

🖫 fibmemo.c

В следващите два алгоритъма ще предполагаме, че $F_0 = 1$. Това няма особено значение, но от друга страна може леко да облекчи някои детайли от по-нататъшните ни разсъждения.

Ще разгледаме още два подхода за бързо пресмятане на n-тото число на Фибоначи, използващи само целочислено умножение и събиране. Първият се основава на следното матрично равенство:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \quad (1)$$

Горното равенство означава, че за намиране n-тото число на Фибоначи е необходимо да повдигнем матрицата от лявата страна на равенството на n-та степен. Ясно е, че при използване на бърз алгоритъм за степенуване (*виж 7.5*, [*Наков-1998c*]) това може да стане с *логаритмична* сложност $\Theta(\log_2 n)$, тъй като времето, необходимо за умножение на две матрици, не зависи от n. Ако се вгледаме по-подробно в горното матрично равенство, ще забележим, че тъй като изходната матрица е симетрична, то на всеки етап от работата на алгоритъма се получава също симетрична относно главния си диагонал матрица. Това ни позволява известно оптимизиране процеса на умножение, в резултат на което получаваме следния алгоритъм:

```
#include <assert.h>
#include <stdio.h>
#include <string.h>

#define SQR(X) ((X) * (X))

const unsigned n = 10;

unsigned long matrE[2][2] = { { 1, 1 }, { 1, 0 } }; /* Изходна матрица */
unsigned long matr[2][2]; /* Резултатна матрица */

void fibMatr(unsigned n, unsigned long matr[][2])
{ static unsigned long lMatr[2][2]; /* Помощна матрица */
  static unsigned long sq12;        /* Помощна променлива */

  if (n < 2)
    memcpy(matr, matrE, 4 * sizeof(matr[0][0]));
  else if (0 == n % 2) {
    fibMatr(n / 2, lMatr);
    sq12 = SQR(lMatr[0][1]);
    matr[0][0] = SQR(lMatr[0][0]) + sq12;
    matr[1][1] = SQR(lMatr[1][1]) + sq12;
    matr[0][1] = matr[0][0] - matr[1][1];
    matr[1][0] = matr[0][1];
  }
  else {
    fibMatr(n - 1, lMatr);
    matr[1][1] = lMatr[0][1];
    matr[0][1] = lMatr[0][0];
    matr[0][0] = lMatr[0][0] + lMatr[1][0];
    matr[1][0] = lMatr[0][1];
  }
}

int main(void) {
  fibMatr(n - 1, matr);
  printf("\n%u-тото число на Фибоначи е: %lu", n, matr[0][0]);
  return 0;
}
```

🖫 fibmatr.c

Горната реализация не е много добра, тъй като при всяко рекурсивно извикване на функцията в системния стек постъпва подадената като параметър матрица `matr[][]` с размер 2×2. Като задача за упражнение, читателят би могъл да се опита да реализира съответен итеративен алгоритъм, което ще отстрани посочения недостатък. Остава обаче още една възможност за оптимизация: тъй като матриците са винаги симетрични относно главния си диагонал, то бихме могли изобщо да се откажем от елемента под/над главния диагонал, тъй като той само дублира стойността на друг елемент от матрицата, без да носи допълнителна полза, а само проблеми, свързани главно с допълнителна памет в системния стек и допълнителните операции по присвояване на съответна стойност.

Доразвивайки идеята, достигаме до по-проста формула за бързо пресмятане на n-тото число на Фибоначи, силно наподобяваща тази за бързото умножение:

$$F_n = \begin{cases} F_{\frac{n}{2}}^2 + F_{\frac{n}{2}-1}^2, n = 2k \\ F_{n-1} + F_{n-2}, n = 2k+1 \end{cases} \qquad (2)$$

При реализацията ще се откажем от използването на матрица, прилагайки метода на запомняне на вече пресметнатите стойности в допълнителен масив.

```
#include <stdio.h>
#include <string.h>
#define MAX 250
#define SQR(X) ((X)*(X))

unsigned long m[MAX+1];

const unsigned n = 10;

#include <stdio.h>
#include <string.h>

#define MAX 250
#define SQR(X) ((X)*(X))

unsigned long m[MAX+1];

const unsigned n = 10;

/* Бърз рекурсивен логаритмичен вариант, запаметяващ вече изчисленото */
unsigned long fMemo2(unsigned n)
{ if (n < 2)
    m[n] = 1;
  else if (0 == m[n])
    if (1 == n % 2)
      m[n] = fMemo2(n - 1) + fMemo2(n - 2);
    else
      m[n] = SQR(fMemo2(n / 2)) + SQR(fMemo2(n / 2 - 1));
  return m[n];
}
int main(void) {
  memset(m, 0, MAX * sizeof(m[0]));
  printf("\n%u-тото число на Фибоначи е: %lu", n, fMemo2(n - 1));
  return 0;
}
```
🖫 fibmemo2.c

И така, след като разгледахме достатъчно различни варианти за бързо пресмятане на числата на Фибоначи, възниква естественият въпрос: Кой е най-добрият метод? Отговорът на този въпрос е сложен и нееднозначен. Следва да се отбележи, че числата на Фибоначи растат много бързо и за $n = 48$ вече не се побират в стандартните целочислени типове за повечето компютърни системи. Едно възможно решение на проблема е използването на по-големи стандартни типове, например *double* по стандарта *IEEE*, но това отново не води до задоволителен резултат, тъй като въпросният тип има мантиса с ограничен брой значещи цифри — не повече от 18-19. Т. е. по този начин няма да можем да намерим например 1000-ия член на редицата на Фибоначи. Ето защо при по-големи стойности на n се налага използване на нестандартни за компютърната система типове данни: така наречените "дълги" числа. Обикновено наборът от средства за ефективна работа с такива числа се свежда до стандартните аритметични операции събиране, изваждане и умножение. Ето защо в този случай е добре да се насочим към последните два предложени метода.

Задачи за упражнение:

1. Като се използва определението за произведение на матрици да се докаже формула (1).

2. Да се докаже формула (2) по индукция.

3. Да се реализира итеративен вариант на програмата, реализираща алгоритъма, зададен от формула (1). Да се сравни с рекурсивния вариант по отношение на необходима памет.

4. Да се провери емпирично кой е най-бързият метод на Вашата машина.

5. Да се реализират необходимите операции с дълги числа и да се сравнят по скорост предложените алгоритми при пресмятане на n-тото число на Фибоначи за n: 10, 100, 200, 500, 1000.

8.3.2. Биномни коефициенти

Ще припомним (*виж 1.1.5*), че биномните коефициенти се бележат с C_n^k или $\binom{n}{k}$, като второто обозначение е по-общо и има не само комбинаторен смисъл. Пресмятането на биномните коефициенти може да стане по комбинаторната формула:

$$C_n^k = \frac{n!}{k!(n-k)!},\quad 0 \le k \le n,\ n \text{ и } k \text{ — естествени}$$

В *глава 1* от триъгълника на Паскал изведохме и друга рекурентна формула, а именно:

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \text{ или } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 0 & k > n \end{cases}$$

Горната формула дава възможност за директна рекурсивна реализация на съответна функция на Си:

```
unsigned long binom(unsigned n,unsigned k) /* Неефективен рек. вариант */
{ if (k > n) return 0;
    else if (0 == k || k == n) return 1;
    else return binom(n-1, k-1) + binom(n-1, k);
}
```

🖫 binom.c

Горният вариант силно прилича на първия вариант на функцията за търсене на n-тото число на Фибоначи: `fibRec()`. За съжаление, налице са и същите недостатъци: многократно пресмятане на едни и същи стойности. Така например, пресмятането на `binom(6,4)` ще доведе до пресмятане на `binom(5,3)` и `binom(5,4)`. Те от своя страна изискват пресмятането на `binom(4,3)`, поради което тази стойност ще се пресметне на практика два пъти. По-надолу по дървото на рекурсията нещата стоят още по-лошо. Разсъждавайки индуктивно, получаваме, че алгоритмичната сложност на предложената функция е $\Theta\left(C_n^k\right)$.

Бихме могли да ускорим значително горната функция, въвеждайки таблица с вече пресметнатите стойности. На пръв поглед може да изглежда, че е нужно запазването на цялата таблица — функцията е двуаргументна. В действителност се оказва достатъчно да се пази само предходният ред на таблицата, тъй като при пресмятането на k-тия ред функцията се обръща само към $(k–1)$-вия ред. Подробностите могат да се видят в предложената по-долу реализация. Така, получаваме алгоритъм със сложност по време $\Theta(n.k)$ и сложност по памет $\Theta(k)$.

```
#define MAX 200

unsigned long m[MAX];

unsigned long binomDynamic(unsigned n, unsigned k)
{ unsigned i, j;
  for (i = 0; i <= n; i++) {
```

```
    m[i] = 1;
    if (i > 1) {
      if (k < i - 1) j = k; else j = i - 1;
      for (; j >= 1; j--) m[j] += m[j - 1];
    }
  }
  return m[k];
}
```
binom2.c

Задачи за упражнение:

1. Да се реализира рекурсивен вариант, основан на динамично оптимиране.
2. Възможно ли е да се намали сложността по време $\Theta(n.k)$ така, както беше възможно при числата на Фибоначи?

8.3.3. Спортни срещи

Да си представим следната игра: Два отбора A и B играят серия от най-много $2n-1$ срещи, при всяка от които има вероятност p ($0 \leq p \leq 1$) първият отбор да спечели и вероятност $q = 1-p$ вторият отбор да спечели. Победител в сериите е този отбор, който пръв постигне n победи. Равни мачове не се допускат. Да се определи вероятността първият отбор да спечели серията. Нека с $P(i,j)$ означим вероятността първият отбор да спечели серията, ако за това му остават още i победи, а на втория — още j победи. Така например, вероятността A да спечели, преди да е изигран първият мач, е $P(n,n)$, тъй като и двата отбора се нуждаят от n победи. Ако A има нужда от още 0 победи, то очевидно вече е спечелил. Оттук веднага получаваме, че $P(0,j) = 1$ ($1 \leq j \leq n$). Аналогично, ако B се нуждае от още 0 победи, то вече е спечелил и вероятността A да спечели е 0: $P(i,0) = 0$ ($1 \leq i \leq n$). Тогава търсената вероятност се определя по следните формули:

$$P(0,j) = 1,\ j = 1, 2, \ldots, n$$
$$P(i,0) = 0,\ i = 1, 2, \ldots, n$$
$$P(i,j) = p.P(i-1, j) + q.P(i, j-1)\ ,\ \ i > 0\ ,\ j > 0$$

Забележете, че $P(0,0)$ не е дефинирано. Това е така, защото то няма смисъл — в такъв случай и двата отбора ще са спечелили, което е невъзможно (Да си припомним, че равни мачове не се допускат.).

Целта е да се състави програма, която по зададени n и p да пресмята $P(n,n)$, т. е. да намира вероятността първият отбор да спечели серията. Използвайки горните формули, лесно можем да реализираме съответна рекурсивна функция на Си, пресмятаща $P(i,j)$. В частност при $i = j = n$ ще получим $P(n,n)$:

```
/* Неефективен рекурсивен вариант */
float P(unsigned i, unsigned j)
{ if (0 == j)
    return 0.0;
  else if (0 == i)
    return 1.0;
  else
    return p * P(i - 1, j) + (1 - p) * P(i, j - 1);
}
```
series.c

Подобно на числата на Фибоначи (*виж 8.3.1.*) и биномните коефициенти (*виж 8.3.2.*) и тук директната рекурсивна реализация води до неефективен алгоритъм. Действително, за пресмятане на $P(3,4)$ се изчисляват $P(2,4)$ и $P(3,3)$, които от своя страна изискват намирането на

$P(2,3)$. Така $P(2,3)$ се пресмята два пъти. По-нататък препокриването на рекурсивните извиквания става още по-голямо. Сложността на горната функция вече беше оценена в *параграф 1.4.9.*, където получихме, че извикването `P(n,n)` има сложност $\Theta(4^n)$.

При по-прецизен анализ можем да получим, че общият брой извиквания на $P(i,j)$ е $\binom{i+j}{i}$, т. е. броят на начините на избор на i предмета измежду $i + j$ [*Aho,Hopcroft,Ullman–1987*]. Наистина, ако се вгледаме по-внимателно в горната функция, ще видим още, че в *структурно* отношение тя е напълно еквивалентна на първата реализация на функцията за пресмятане на биномните коефициенти. Действително, да заменим $P(i,j)$ с $binom(i+j,j)$. Тогава рекурентната формула

$$P(i,j) = p.P(i-1, j) + q.P(i, j-1)$$

силно напомня програмната конструкция:

$$binom(i+j,j) = binom(i+j-1, j) + binom(i+j-1, j-1).$$

Тук се абстрахираме от умноженията по p и q, тъй като се опитваме да оценим сложността на алгоритъма, а тези умножения са с константна сложност.

Оттук директно се получава, че времето, необходимо за пресмятането на $P(n,n)$, е $\Theta\left(\binom{2n}{n}\right)$. Разбира се, това също ни води към експоненциална сложност, тъй като последното е равно на $(2n)! / (2.n!)$. След като се убедихме аналитично в експоненциалната сложност на горния алгоритъм, да видим как бихме могли да го подобрим. (Впрочем, читателят би могъл да се убеди в това на практика, стартирайки горната програма например за $n = 14$). Съвсем естествено е да използваме таблица на вече пресметнатите стойности. Всеки елемент на таблицата ще представлява запис с две полета: `char calculated` (ползва се като булев тип: била ли е пресмятана вече тази стойност?) и `float value` (пресметнатата стойност). Новата функция ще изградим на основата на тази по-горе. Разликата ще бъде, че преди да извършим рекурсивно обръщение, ще проверяваме дали интересуващата ни стойност не е била вече пресметната. Функцията `pDynamic()` установява полето `calculated` в 0, след което извиква `pDyn()`, която извършва пресмятанията.

```
#define MAX 100

#define q (1 - (p))              /* Вероятност B да спечели отделен мач */

const float p = 0.5;             /* Вероятност A да спечели отделен мач */
const unsigned n = 5;

struct {
  char calculated;               /* Пресметната ли е вече стойността? */
  float value;                   /* Пресметнатата стойност */
} PS[MAX][MAX];

float pDyn(unsigned i, unsigned j)  /* Динамично оптимиране */
{ if (!PS[i][j].calculated) {
    PS[i][j].value = p * pDyn(i - 1, j) + q * pDyn(i, j - 1);
    PS[i][j].calculated = 1;
  }
  return PS[i][j].value;
}

float pDynamic(unsigned i, unsigned j)
{ unsigned k, l;
```

```
  for (k = 1; k <= i; k++)
    for (l = 1; l <= j; l++)
      PS[k][l].calculated = 0;
  for (k = 1; k <= i; k++) {
    PS[0][k].value = 1.0;
    PS[0][k].calculated = 1;
  }
  for (k = 1; k <= j; k++) {
    PS[k][0].value = 0.0;
    PS[k][0].calculated = 1;
  }
  return pDyn(i, j);
}
```

🖫 series2.c

Бихме могли да се освободим от полето `calculated`. Действително, всички вероятности, които се пресмятат, се неотрицателни числа. В такъв случай бихме могли да инициализираме в началото масива с някакво отрицателно число, например –1, след което да познаваме дали дадена стойност е вече пресмятана по знака й. Ако е неотрицателно число, то тя е била вече пресметната, иначе — не е била. Следва да отбележим, че подобен подход е потенциално опасен (сравнение на приближено число с 0), тъй като при работа с приближени числа е възможна загуба на точност. В нашия случай обаче това не би могло да доведе до проблеми (*Защо*?).

```
#define MAX 100
#define NOT_CALCULATED (-1)
#define q (1 - (p))              /* Вероятност B да спечели отделен мач */
const float p = 0.5;             /* Вероятност A да спечели отделен мач */
const unsigned n = 5;
float PS[MAX][MAX];

float pDyn(unsigned i, unsigned j)  /* Динамично оптимиране */
{ if (PS[i][j] < 0)
    PS[i][j] = p * pDyn(i - 1, j) + q * pDyn(i, j - 1);
  return PS[i][j];
}

float pDynamic2(unsigned i, unsigned j)
{ unsigned k, l;
  for (k = 1; k <= i; k++)
    for (l = 1; l <= j; l++)
      PS[k][l] = NOT_CALCULATED;
  for (k = 1; k <= i; k++)
    PS[k][0] = 0.0;
  for (k = 1; k <= j; k++)
    PS[0][k] = 1.0;
  return pDyn(i, j);
}
```

🖫 series3.c

Не е трудно да се забележи, че сложността на последните две функции е $\Theta(n^2)$ както по време, така и по памет. Основно тяхно предимство беше простотата им, а подходът, който използвахме при реализирането им, е универсален за прилагането на *memoization*. Основното неудобство е свързано с необходимостта от предварително инициализиране на таблицата, което доведе до нуждата от въвеждане на допълнителна функция.

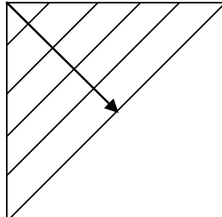
Истинското изкуство на динамичното оптимиране се крие в това, таблицата да се попълва по такъв начин, че да не се преминава два пъти през една и съща нейна клетка, при което ини-

циализацията отпада по естествен начин. Да видим как това може да стане в нашия случай. Нека за целта разгледаме *таблица 8.3.3*.

*j**i*	0	1	2	3	4
0	×	1	1	1	1
1	0	1/2	3/4	7/8	15/16
2	0	1/4	1/2	11/16	13/16
3	0	1/8	5/16	1/2	21/32
4	0	1/16	3/16	11/32	1/2

Таблица 8.3.3. Стойности на вероятността $P(i,j)$ за $n = 4$ и $p = 1/2$ $(0 \leq i, j \leq n)$.

Непосредствено от вида на използваната формула се вижда, че всеки елемент в *таблица 8.3.3.*, с изключение на първия ред и първата колона, се получава само от два елемента: този над него и този вляво от него. Дали не можем да се възползваме от това така, както се възползвахме от триъгълника на Паскал при биномните коефициенти? Ако човек се вгледа по-внимателно, ще види, че и тук има триъгълник. Той съдържа горния ляв ъгъл, както и двата съседни му ъгли (*виж фигура 8.3.3.*).



Фигура 8.3.3. Ред на попълване на таблицата.

Ако започнем да попълваме таблицата по редовете на този триъгълник, всички необходими за текущото пресмятане стойности ще бъдат вече пресметнати на някоя предишна стъпка. (Всъщност по-правилно е да говорим за квадрат, тъй като след втория диагонал има нов триъгълник, чиито редове обаче се свиват.)

```
float pDynamic3(unsigned i, unsigned j)
{ float P[MAX][MAX];
  unsigned s, k;
  P[0][0] = 0.0; /* Това е излишно. Не ни трябва */
  for (s = 1; s <= i + j; s++) {
    P[0][s] = 1.0;
    P[s][0] = 0.0;
    for (k = 1; k <= s - 1; k++)
      P[k][s - k] = p * P[k - 1][s - k] + q * P[k][s - k - 1];
  }
  return P[i][j];
}
```
🖫 series4.c

Предложената реализация не се възползва напълно от открития триъгълник и отново изисква памет $\Theta(n^2)$. Единственото предимство е, че спестихме инициализирането. Алгоритмичната сложност по време отново е $\Theta(n^2)$.

Задачи за упражнение:

1. Защо сравнението с реалното число 0.0 по-горе не е опасно? Кога би било опасно?
2. Да се модифицира последната функция така, че да бъде необходима памет $\Theta(n)$.

3. Да се сравни настоящата задача с тази за биномните коефициенти (*виж 1.1.5.*, *8.3.2.*). Възможно ли е и тук да се дефинира истински триъгълник, подобен на този на Паскал?

8.3.4. Представяне на сума с неограничен брой монети

Задача: Дадени са n типа монети със стойности съответно: c_0, c_1, ..., c_{n-1}, и естествено число s. Да се намери броят на различните представяния на s с монети измежду наличните типове. Стойностите c_0, c_1, ..., c_{n-1} са цели положителни числа. Всеки тип монети може да участва в сумата неограничен брой пъти.

Решение: За съжаление, тук не можем да подходим директно към целта, тъй като комбинаторните конфигурации нямат явна наредба. В действителност лесно можем да въведем такава, ако изискаме монетите в сумата да бъдат наредени по големина в намаляващ ред. Налага се да решим една по-обща задача. Да означим с $F(s,m)$ броя на начините, по които можем да представим сумата s с тези монети измежду наличните, чиято стойност не надвишава m. В сила са следните зависимости:

$$F(s,m) = \begin{cases} 0 & s \le 0 \\ F(s,s) & s < m \\ 1 + \sum\limits_{i=1,2,\dots,m;\exists k: c_k = i} F(s-i,i) & s = m \;\&\; \exists k : c_k = s \\ \sum\limits_{i=1,2,\dots,m;\exists k: c_k = i} F(s-i,i) & \textit{иначе} \end{cases}$$

Повечето от горните формули са очевидни. Първата формула отразява факта, че сума 0 не може да бъде получена. Втората формула показва, че при получаване на сума s не могат да се използват монети с по-голяма стойност от s. Четвъртата формула отразява общия случай. Да вземем една монета, чиято стойност c_k не надвишава нито s, нито m. Тогава, можем да опитаме да получим сумата $s-c_k$, като не допускаме използване на монети, чиято стойност надвишава c_k. Броят на начините, по които може да стане това, се дава от $F(s- c_k, c_k)$. Ясно е, че ако сумираме $F(s-c_k, c_k)$ по всички такива монети със стойност c_k, за които $c_k \le s$ и $c_k \le m$, ще получим $F(s,m)$. Третата формула обработва случая, когато съществува тип монети със стойност точно s.

Функцията `count()` пресмята $F(s,m)$ рекурсивно, като пази вече пресметнатите стойности, т. е. прилага се *memoization*. За намиране на броя на всички представяния на сумата s се извършва обръщението `count(s,s)`, което връща като резултат стойността на $F(s,s)$. Входните данни са зададени като константи в началото на програмата.

```
#include <stdio.h>
#define MAXCOINS  100 /* Максимален брой монети */
#define MAXSUM    100 /* Максимална сума */

unsigned long F[MAXSUM][MAXSUM];                    /* Целева функция */
unsigned char exist[MAXSUM]; /* Съществува ли монета с такава стойност */

const unsigned coins[MAXCOINS] = {1,2,3,4,6}; /* Налични типове монети */
const unsigned sum = 6;               /* Сума, която искаме да получим */
const unsigned n = 5;                       /* Общ брой налични монети */

/* Инициализираща функция */
void init(void)
{ unsigned i, j;
  /* Нулиране на целевата функция */
  for (i = 0; i <= sum; i++)
    for (j = 0; j <= sum; j++)
      F[i][j] = 0;
  /* Друго представяне на стойностите на монетите за по-бърз достъп */
```

```
  for (i = 0; i <= sum; i++)
    exist[i] = 0;
  for (i = 0; i < n; i++)
    exist[coins[i]] = 1;
}

/* Намира броя на представянията на sum */
unsigned long count(unsigned sum, unsigned max)
{ unsigned long i;
  if (sum <= 0)
    return 0;
  if (F[sum][max] > 0)
    return F[sum][max];
  else {
    if (sum < max)
      max = sum;
    if (sum == max && exist[sum]) /* Има монета с такава стойност */
      F[sum][max] = 1;
    for (i = max; i > 0; i--)  /* Пресмятаме всички */
      if (exist[i])
        F[sum][max] += count(sum - i, i);
  }
  return F[sum][max];
}

int main(void) {
  init();
  printf("\nБроят на представянията на %u с наличните монети е %lu",
         sum, count(sum, sum));
  return 0;
}
```

🖫 coin_min.c

Резултат от изпълнението на програмата:

```
Броят на представянията на 6 с наличните монети е 10
```

Действително, има точно 10 различни представяния с наличните типове монети 1, 2, 3, 4 и 6, а именно:

```
6 = 6
6 = 4 + 2
6 = 4 + 1 + 1
6 = 3 + 3
6 = 3 + 2 + 1
6 = 3 + 1 + 1 + 1
6 = 2 + 2 + 2
6 = 2 + 2 + 1 + 1
6 = 2 + 1 + 1 + 1 + 1
6 = 1 + 1 + 1 + 1 + 1 + 1
```

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.
2. Да се реализира итеративен вариант на програмата.

8.3.5. Представяне на сума с ограничен брой монети

Да разгледаме отново задачата от предходния параграф, като този път ще считаме, че броят монети от всеки тип е ограничен, т. е. дадени са ни *не типове* монети, а *конкретно множество*. Целта е да намерим броя на начините, по които може да бъде представена дадена сума S.

Отново ще решим по-обща задача. Да наредим монетите по големина в нарастващ ред. Въвеждаме целевата функция $F(s,k)$, която ни дава броя на начините за получаване на сумата s при използване само на първите k монети: $c_0, c_1, ..., c_{k-1}$. В сила са формулите:

$$F(s,k) = \begin{cases} 0 & s \le 0 \\ 0 & k < 0 \\ 1 + F(s - c_k, k-1) + F(s, j),\ j = \max\limits_i \left[c_u < c_k \right] & \exists l : c_l = s \\ F(s - c_k, k-1) + F(s, j),\ j = \max\limits_i \left[c_u < c_k \right] & \text{иначе} \end{cases}$$

Първите два случая са очевидни. Четвъртата формула отговаря на общия случай, когато не съществува монета със стойност s. Имаме две взаимноизключващи се възможности:

1) k-тата монета *участва* в сумата. Броят на тези конфигурации се дава от първото събираемо.
2) k-тата монета *не участва* в сумата. Броят на тези конфигурации се дава от второто събираемо.

Третата формула отговаря на случая, когато съществува монета със стойност s.

```
#include <stdio.h>
#define MAXCOINS    100 /* Максимален брой монети */
#define MAXSUM      100 /* Максимална сума */
#define SWAP(a, b)  { a = a ^ b; b = a ^ b; a = a ^ b; }

unsigned long F[MAXSUM][MAXSUM]; /* Целева функция */
const unsigned n = 7;    /* Общ брой налични монети */
const unsigned sum = 6; /* Сума, която искаме да получим */
unsigned coins[MAXCOINS] = {1,2,2,3,3,4,6}; /* Налични типове монети */

/* Инициализираща функция */
void init(void)
{ unsigned i, j;
  /* Нулиране на целевата функция */
  for (i = 0; i <= sum; i++)
    for (j = 0; j <= sum; j++)
      F[i][j] = 0;
}

/* Сортира монетите в нарастващ ред */
void sort(void)
{ unsigned i, j;
  for (i = 0; i < n - 1; i++)
    for (j = i + 1; j < n; j++)
      if (coins[i] > coins[j])
        SWAP(coins[i], coins[j]);
}

/* Намира броя представяния на sum с използване на първите k монети */
unsigned long count(int sum, int k)
{ unsigned j;
  if (sum <= 0 || k < 0)
```

```
    return 0;
  if (F[sum][k] > 0)
    return F[sum][k];
  else {
    if (coins[k] == (unsigned)sum)
      F[sum][k] = 1;
    F[sum][k] += count(sum - coins[k], k - 1);
    j = k;
    while (coins[j] == coins[k]) j--;
    F[sum][k] += count(sum, j);
  }
  return F[sum][k];
}

int main(void) {
  init();
  sort();
  printf("\nБроят на представянията на %u с наличните монети е %lu.",
         sum, count(sum, n - 1));
  return 0;
}
```
🖫 coinmin2.c

Резултат от изпълнението на програмата:
```
Броят на представянията на 6 с наличните монети е 4.
```

Действително, съществуват точно 4 различни представяния с наличните типове монети 1, 2, 2, 3, 3, 4 и 6, а именно (сравнете с предходния параграф, където се използваха *същите* типове монети, но *без ограничение за кратността им в сумата*):

```
6 = 6
6 = 4 + 2
6 = 3 + 3
6 = 3 + 2 + 1
```

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.
2. Да се реализира итеративен вариант на програмата.
3. Да се сравнят условията и съответните алгоритми на текущата и на предходната задача.

8.3.6. Разбиване на естествено число

Дефиниция 8.4. Под разбиване на естествено число n ще разбираме всяко мултимножество от естествени числа, не непременно различни, чиято сума е n.

Например за $n = 4$ всевъзможните разбивания са:

```
4
3 1
2 2
2 1 1
1 1 1 1
```

Целта ни е да намерим *броя* на различните разбивания. Да разгледаме по-общата задача за определяне броя $S(n,m)$ на представянията на n като сума от естествени числа, ненадминаващи m.

Тогава интересуващият ни брой на разбиванията ще бъде $S(n,n)$. За $S(n,m)$ е в сила формулата [*Рахнев,Гъров-1988*]:

$$S(n,m) = \begin{cases} 1 & n = 1 \text{ или } m = 1 \\ S(n,n) & n < m \\ 1 + S(n, n-1) & n = m \\ S(n, m-1) + S(n-m, m) & n > m \end{cases}$$

1) $S(n,1) = 1$ (Съществува единствено представяне с най-голямо събираемо 1.)
2) $S(1,m) = 1$ (Единицата има единствено представяне за всяко m.)
3) $S(n,m) = S(n,n)$, $n < m$ (Никое представяне на n не съдържа по-голямо от n събираемо.)
4) $S(n,n) = 1 + S(n,n-1)$ (Само едно представяне на n, съдържа n като събираемо, а броят на останалите е $S(n,n-1)$.)
5) $S(n,m) = S(n,m-1) + S(n-m,m)$, $n > m$ (Броят на представянията на n като сума от числа, по-малки или равни на m, е $S(n,m-1)$, а на останалите — $S(n-m,m)$.)

```
#include <stdio.h>
#define MAX 100
unsigned long M[MAX][MAX];  /* Целева функция */
const unsigned n = 10;

/* Намира броя на представянията на n като сума от естествени числа */
unsigned long getNum(unsigned n)
{ unsigned i, j;
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
      if (1 == j)
        M[i][j] = 1;
      else if (1 == i)
        M[i][j] = 1;
      else if (i < j)
        M[i][j] = M[i][i];
      else if (i == j)
        M[i][j] = 1 + M[i][i - 1];
      else
        M[i][j] = M[i][j - 1] + M[i - j][j];
  return M[n][n];
}

int main(void)
{ printf("Броят представяния на %u като сума от естествени числа е: %lu",
         n, getNum(n));
  return 0;
}
```

🖫 breaknum.c

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.
2. Да се реализира рекурсивен вариант на програмата и да се сравни с итеративния.
3. Позволява ли формулата да се намали на порядък размерът на необходимата памет?

4. Да се сравнят условията и съответните алгоритми на текущата и на предходните две задачи.

8.3.7. Числа без две съседни нули

Да разгледаме друга задача [*Timus, задача 1009*], свързана с броене на комбинаторни конфигурации.

Задача: Разглеждаме n-цифрените числа в k-ична бройна система. Ще казваме, че едно число е "правилно", ако не съдържа две съседни нули. Целта е да се намери броят на "правилните" числа при зададени n и k ($2 \le k \le 10$; $n \ge 2$; $4 \le n + k \le 18$).

Примери:
4308063 е правилно 7 цифрено число (за $k \ge 9$)
1000198 не е правилно число
0002323 не е 7-цифрено, а 4 цифрено число (за $k \ge 4$)

Формулировката на задачата като че ли не ни дава проста рекурсивна зависимост, която да използваме. Ключът към решението е обръщането на целта: Вместо да броим правилните числа, бихме могли да преброим *неправилните*. Да предположим, че разполагаме с алгоритъм, който ни дава броя на неправилните числа $q(n,k)$. Броят на всевъзможните n-цифрени последователности в k-ична бройна система, правилни и неправилни, е n^k. Оттук броят на правилните числа ще бъде $F(n,k) = n^k - q(n,k)$.

Как да намерим броя на неправилните числа? Неправилните числа съдържат поне две съседни нули. Можем да ги разпределим в n–1 категории в зависимост от позицията на най-лявата двойка съседни нули. Така например, за n = 8 имаме:

```
00??????
x00?????
_x00????
__x00???
___x00??
____x00?
_____x00
```

Тук с "0" сме означили цифрата 0, с "?" — произволна k-ична цифра (0, 1, ..., k–1), с "x" — произволна различна от нула k-ична цифра (1, 2, ..., k–1), а "_" означава произволна цифра такава, че никои две съседни "_" не са едновременно нули.

Така, неправилните числа в общия случай имат лява част, несъдържаща две съседни нули, средна част, състояща се от ненулева цифра, последвана от две нули и дясна част, върху която няма ограничения. В първата група липсва лявата част, а в последната — дясната. В първата група средната част е непълна — липсва "x".

Как да намерим броя на неправилните числа за дадена конкретна група? Да вземем например: "___x00??". Да означим с $F(s,k)$ броя на правилните s-цифрени k-ични числа. Броят на различните възможности за лявата част е $F(3,k)$, за средната — $(k-1)$, и за дясната — k^2.Тогава броят на числата в тази група е: $F(3,k).(k-1).k^2$

По аналогичен начин може да се пресметне броят на числата в останалите групи. Така, сумирайки по всички групи, достигаме до следната формула за броя на неправилните числа:

$$q(s,k) = k^{s-2} + \sum_{i=1}^{s-2} F(i-1,k).(k-1).k^{s-i-2}$$

Тук първата група (k^{s-2}) е разгледана отделно. Остава да дадем граничните случаи, които се пресмятат непосредствено:

$$F(0,k) = 1$$
$$F(1,k) = k$$
$$F(2,k) = k(k-1)$$

Броя на правилните числа получаваме по разгледаната по-горе формула:

$$F(s,k) = k^s — q(s,k)$$

Така, пресмятането на търсената стойност $F(n,k)$ може да стане отдолу-нагоре, като се започне с $F(3,k)$ и на всяка стъпка се увеличава стойността на s с 1 и пресметнатите вече стойности на функцията се запазват. Ние обаче ще започнем отгоре-надолу — *memoization*. С цел ефективност степените на k ще пресметнем предварително.

```
#include <stdio.h>
#define MAX         100
#define NOT_SOLVED  (unsigned long)(-1)
unsigned long F[MAX];    /* Целева функция */
unsigned long pow[MAX]; /* Степените на k */
const unsigned n = 10;
const unsigned k = 7;

void init(void)
{ unsigned i;
  for (i = pow[0] = 1; i <= n; i++) {
    pow[i] = k * pow[i - 1];
    F[i] = NOT_SOLVED;
  }
}

unsigned long solve(unsigned s)
{ unsigned i;
  if (NOT_SOLVED == F[s]) {
    F[s] = pow[s - 2];
    for (i = 1; i < s - 1; i++)
      F[s] += solve(i - 1) * (k - 1) * pow[s - i - 2];
    F[s] = pow[s] - F[s];
  }
  return F[s];
}

int main(void) {
  init(); F[0] = 1; F[1] = k; F[2] = k * k - 1;
  printf("%lu\n", (k - 1) * solve(n - 1));
  return 0;
}
```
🖫 no2zero.c

Задачи за упражнение:

1. Каква е сложността на предложената по-горе реализация?
2. Да се реализира итеративен вариант на програмата и да се сравни с рекурсивния.

8.3.8. Разпознаване на контекстно свободен език

Всеки формален език се изгражда въз основа на някакви правила — граматика. В зависимост от това дали дадено изречение съответства на граматиката на даден език, то се определя като правилно или неправилно в смисъла на този език. В основата на всеки *естествен* (т. е. използвания за общуване помежду ни) език лежат думите. В основата на *формалния* (например използвания при диалог с компютъра) език — символите. И при естествения, и при формалния език думите се свързват в изречения въз основа на някакъв синтаксис (На *изречението* в естествения език съответства понятието *дума* във формалния. Налице са и някои други различия,

които не представляват интерес за настоящото изложение.). Синтаксиса на даден език най-общо бихме могли да определим като съвкупност от правила, определящи структурата на изреченията.

Тъй като езиците, които разглеждаме, могат да бъдат безкрайни, то се налага дефинирането на метод за описание на безкрайни езици чрез крайна граматика. Очевидно използването на безкрайна граматика е сложно и безсмислено. За съжаление, не за всички езици може да се намери крайна граматика, което означава, че не можем да обхванем всички езици, а само част от тях. Все пак езиците, които се описват с крайно множество от правила, се оказват напълно достатъчни за практически цели.

Дефиниция 8.5. *Формална граматика* ще наричаме наредената четворка (*виж* [*Манев-1996*]):

$$\Gamma = \langle N, T, S, P\rangle,$$

където:

- N е крайна азбука, чиито елементи наричаме *нетерминали*;
- T е крайна азбука, чиито елементи наричаме *терминали*, като N и T не съдържат общи елементи, т. е. $N \cap T = \varnothing$;
- S се нарича *аксиома* (начален нетерминал), $S \in N$;
- P е множество от правила (продукции) от вида $\alpha \rightarrow \beta$, като:
    - ◊ $\alpha = \alpha' A \alpha''$, като A е нетерминал, а $\alpha', \alpha'' \in (N \cup T)^*$ (α', α'' са последователности от терминали и нетерминали, включително празната);
    - ◊ $\beta \in (N \cup T)^*$;
    - ◊ $d(\alpha) \le d(\beta)$, (дължината на α е по-малка или равна на дължината на β), т. е. правилата са несъкращаващи.

Символът * е знак за итерация (т. е. повторение нула, един или повече пъти), + е знак за повторение един или повече пъти, а с $d(\alpha)$ сме означили дължината на α.

Така например, граматиката на езика $L = \{ac, ad, bc, bd\}$ би могла да се дефинира като наредена четворка по следния начин:

$$\Gamma = \langle \{S, A, B\}, \{a, b, c, d\}, S, \{S \rightarrow AB, A \rightarrow a, A \rightarrow b, B \rightarrow c, B \rightarrow d\}\rangle$$

Съществуват различни начини за дефиниране на граматиката на формален език. Най-често за тази цел се използват следните представяния:

- наредена четворка съгласно *дефиниция 8.5.*;
- графични диаграми;
- *Бекус-Наурова нормална форма* (*БНФ*).

Бекус-Науровата нормална форма е създадена от Бекус и Наур, сътрудници на *IBM*, и използвана за дефиниране синтаксиса на езика Алгол. (Езиците за програмиране и изобщо повечето изкуствено създадени езици са формални. Разбира се, съществуват изкуствени езици, които не са формални, например *Есперанто*.)

Описаният по-горе език L е дефинира посредством БНФ по следния начин:

```
S ::= AB
A ::= a|b
B ::= c|d
```

Тук *S, A* и *B* са *нетерминали* (най-често за означаване на нетерминалите се използват главни латински букви или конструкции от вида <*синтактична_категория*>. Последните означават име на синтактична категория и следва да се възприемат като един символ. Например: <*подлог*>, <*сказуемо*> и др.). *S* е *начален нетерминал*, *a* и *b* са *терминали*, а символите ::=, |, <, > се наричат *метасимволи* на БНФ и имат специално предназначение. Правилата (*S* ::= *AB* например), съгласно които се изграждат изреченията на езика, се наричат *продукции*. Метасимволът ::= служи за дефиниране на такива правила. Отляво на ::= стои нетерминал, а отдясно — една или повече негови алтернативни дефиниции, представляващи последователност от терминали и нетерминали. Символът | служи за отделяне на алтернативните дефиниции една от друга, а символите < и > заграждат нетерминали, състоящи се от повече от един символ.

Обикновено при дефинирането на формални езици, с цел опростяване на разпознаващия алгоритъм, се изисква анализирането да се извършва с *поглед един символ напред*. Последното се свежда най-общо до следните две изисквания:

- изборът на всяка следваща стъпка от анализирането да зависи само от текущото състояние на пресмятанията и от следващия прочетен символ;
- да не се извършва по-късно повторно изпълнение на вече извършена стъпка.

Синтактичният анализ на дадено изречение от формален език е пряко свързан с граматиката, която описва езика. Разпознаването му като правилно или неправилно е сложна задача, в някои случаи непосилна дори за най-мощните съвременни компютри. Сложността на разпознаването се определя в най-значителна степен от сложността на граматиката на езика и по-специално от множеството от правила P. Обикновено при дефинирането на по-сложен израз се използват рекурентни дефиниции, което определя рекурсията като естествен инструмент при подобен род задачи.

Съществуват два най-общи подхода при съставянето на синтактичен анализатор по зададена граматика. При първия подход анализът е таблично управляем, което определя неговата универсалност (разбира се, при спазване на изискването за анализ с поглед един символ напред) и практическа независимост от конкретната граматика. При втория подход нещата стоят точно обратно. Той се определя от правилата на конкретната граматика и се свежда до преобразуване на продукциите от P в поредица от съответни конструкции (функции или последователности от оператори) *едно към едно*. За целта обикновено се построява явно или неявно (с помощта на различни механизми на използвания език за програмиране като рекурсия и др.) съответен *синтактичен граф*. Така, вторият подход се лишава от универсалността на първия, но често води до по-прости програмни реализации. Именно него ще използваме в програмите по-долу.

Повече за формалните езици и пораждащите ги граматики, както и за синтактичния анализ на изречение съгласно дадена граматика, читателят би могъл да намери в [*Денев-1984*], [*Манев-1996*], [*Радевски-1987*] и [*Уирт-1980*].

Йерархия на Чомски

Формалните езици, породени от формални граматики, се класифицират в четири типа в зависимост от типа на използваните правила: тип 0, тип 1, тип 2 и тип 3. За всяко $i = 1, 2, 3$ езикът от тип i се съдържа в езика от тип i–1, поради което класификацията е известна като *йерархия на Чомски*. По-долу следва видът на правилата в граматиките, пораждащи всеки тип език:

1) Граматика от общ тип — тип 0 (поражда език от общ тип)

$$\gamma_1 A\gamma_2 \to \beta,\ \gamma_1,\ \gamma_2 \in (N\cup T)^*,\ \beta \in (N\cup T)^+,\ A\in N$$

2) Контекстно зависима граматика — тип 1 (поражда контекстен език)

$$\gamma_1 A\gamma_2 \to \gamma_1\beta\gamma_2,\ \gamma_1,\ \gamma_2 \in (N\cup T)^*,\ \beta \in (N\cup T)^+,\ A\in N$$

γ_1 и γ_2 се наричат съответно ляв и десен контекст.

3) Безконтекстна граматика — тип 2 (поражда контекстно свободен език)

$$A\to\alpha,\ \alpha\in(N\cup T)^+,\ A\in N$$

4) Автоматна граматика — тип 3 (поражда автоматен език)

$$A\to B,\ A\to aB,\ A\to a,\ a\in \mathrm{T},\ A,B\in N$$

Ако граматиката не съдържа аксиома в дясна част на правило, се допуска и правилото $S\to\varepsilon$ (ε е празната дума). [*Манев-1996*]

Да разгледаме задачата за разпознаване на език, породен от безконтекстна граматика. Правилата на безконтекстната граматика ще разделим условно на два типа: правила от тип 1 и правила от тип 2:

1) правила от тип 1: $A\to a,\ a\in T,\ A\in N$
2) правила от тип 2: $A\to BC,\ A,B,C\in N$

Бихме могли да дефинираме контекстно свободния език, породен от нетерминала A, като:

$$L(A) = \{bc \mid b \in L(B), c \in L(C), A \to BC\} \cup \{a \mid A \to a\}$$

Тогава езикът, породен от безконтекстната граматика $Г = <N,T,S,P>$, се дава от $L(S)$. Целта ни ще бъде да съставим програма, която по зададена безконтекстна граматика да проверява дали дадена сума (символен низ) принадлежи на езика, който тя поражда, или — не.

Не е трудно да се види, че задачата е подходяща за прилагане на динамично оптимиране, тъй като в общия случай една дума може да бъде породена по повече от един начин. Ще поддържаме булева таблица `t[R][1...n][1...n]`, за всяко $R \in N$. С n сме означили дължината на входната дума x (`string`). За всяко i,j ($1 \le i \le j \le n$) `t[R][i][j] == 1` тогава и само тогава, когато низът $x_i, x_{i+1}, ..., x_j$ принадлежи на езика, породен от нетерминала R. В сила е правилото:

`t[R][i][j] == 1`, тогава и само тогава, когато:

1) `i == j` и $R \to x_i$

или

2) съществува k ($i \le k < j$) и правило $R \to TU$ такива, че `t[T][i][k] == 1` и `t[U][k+1][j] == 1`

Формално алгоритъмът би могъл да се запише така (*виж* [*Parberry-1995*]):

```
unsigned cfl(void) {
  for (всеки нетерминал R)
    for (i = 1; i <= n; i++)
      if (R→xi)
        t[R][i][i] = 1;

  /* Основен цикъл по правилата от тип 2 */
  for (d = 1; d < n; d++)
    for (i = 1; i <= n-d; i++) {
      j = i + d;
      for (всеки нетерминал R) {
        t[R][i][j] = 0;
        for (всяко правило от вида R→TU)
          t[R][i][j] = ∨_{k=i}^{j-1} (t[T][i][k]∧t[U][k+1][j])
      }
    }
  return t[S][1][n]; /* S е аксиомата на граматиката */
}
```

По-долу ще реализираме алгоритъма по малко по-различен начин. Ще разделим правилата от тип 1 и 2 и ще ги разглеждаме поотделно. Първоначално ще инициализираме цялата таблица с `0`. Граничното условие 1) ще разглеждаме само върху правила от тип 1, а условието 2 — върху правила от тип 2. Попълването на таблицата е подобно на начина, по който я попълвахме при решаване на задачата за раницата (*виж 8.2.1.*): започва се с $i == j$, като дължината на интервала $[i,j]$ постепенно се увеличава. Останалите подробности по реализацията се виждат лесно от кода на програмата.

```
#include <stdio.h>
#include <string.h>
#define MAX     30          /* Максимален брой правила за извод */
#define LETTERS 26          /* Брой букви */

const struct { /* Продукции, отиващи в терминали: S->a */
  char S, a;
} prodT[MAX+1] = {
      {0,0},     /* не се използва */
      {'S','s'}, /* S->s */
```

```
      {'A','a'}, /* A->a */
      {'B','b'}  /* B->b */
};

const unsigned cntT = 3;               /* Брой правила от вида 1: S->a */
const char string[MAX + 1] = "aaasbbb"; /* Низ, който проверяваме */

const struct { /* Продукции, отиващи в нетерминали: S->AB */
  char S, A, B;
} prodNT[MAX+1] = {
      {0,0,0},       /* не се използва */
      {'S','A','R'}, /* S->AR */
      {'S','A','B'}, /* S->AB */
      {'R','S','B'}, /* R->SB */
};
const unsigned cntNT = 3;           /* Брой правила от вида 2: S->AB */

unsigned char t[LETTERS][MAX][MAX]; /* Целева функция */

/* Проверява */
unsigned cfl(void)
{ unsigned i, j, k, l, d, let, n;
  /* Инициализиране */
  n = strlen(string); /* Дължина на проверявания низ */
  /* Запълваме масива с "неистина" */
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
      for (let = 0; let < LETTERS; let++)
            t[let][i][j] = 0;
  /* Установяване в 1 на директните продукции, които вършат работа */
  for (i = 1; i <= cntT; i++)
    for (j = 1; j <= n; j++)
      if (prodT[i].a == string[j - 1])
             t[prodT[i].S - 'A'][j][j] = 1;
  /* Основен цикъл по правилата от тип 2 */
  for (d = 1; d < n; d++)
    for (i = 1; i <= n - d; i++)
      /* За всеки нетерминал S от лява част на правило */
      for (j = i + d, k = 1; k <= cntNT; k++)
        for (l = i; l <= j - 1; l++)
          if (t[prodNT[k].A - 'A'][i][l] && t[prodNT[k].B - 'A'][l+1][j])
            t[prodNT[k].S - 'A'][i][j] = 1;
  return t['S' - 'A'][1][n];
}

int main(void) {
  printf("\nНизът %s%s", cfl() ? "" : "НЕ ",
         "се извежда от граматиката!");
  return 0;
}
```

🖫 cfl.c

Резултат от изпълнението на програмата:

```
Низът се извежда от граматиката!
```

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.

2. Единствен ли е изводът на `aaasbbb` в граматиката на програмата по-горе?

3. Да се напише функция за намиране на един възможен извод като последователност от правила от граматиката. Необходимо ли е да се пази допълнителна информация?

4. Да се реализира програма за разпознаване на автоматна граматика. Да се сравни с предложената по-горе реализация за случая на контекстно свободна граматика.

5. Възможно ли е прилагане на динамично оптимиране за разпознаване на контекстно зависима граматика? А на език от общ тип?

6. Ще може ли да се използва горният алгоритъм, ако се търси *най-кратък* извод?

7. Към кой тип граматика принадлежи езикът Си? А *XML*? А българският език?

8.3.9. Хедонийски език

Официалният език на остров Хедония, както следва да се очаква, е хедонийският. Един хедонийски професор забелязал, че студентите му не владеят добре синтаксиса на езика. Уморен да проверява множеството студентски синтактични грешки, той решил да предизвика студентите и им поставил за задача да напишат програма за автоматична проверка на синтактичната коректност на изреченията, които пишат. За щастие езикът на хедонийците е приятно прост:

0. Единствените символи в езика са *p,q,r,s,t,u,v,w,x,y,z* и *N,C,D,E,I*.
1. Всеки символ измежду *p,q,r,s,t,u,v,w,x,y,z,* взет отделно, представлява правилно изречение на хедонийски език.
2. Ако ***S*** е правилно изречение, то ***NS*** също е.
3. Ако ***S*** и ***T*** са правилни изречения, то такива са и ***CST***, ***DST***, ***EST*** и ***IST***.
4. Няма други правилни изречения, освен тези, получени по правила 0,1,2 и 3.

Лесно се вижда, че съгласно горните правила *Isz* и *NIsz* са правилни изречения на хедонийски език, докато *Cp* и *Cqpq* не са.

Настоящата задача силно прилича на предходната от *параграф 8.3.8*. Правило 1 се реализира непосредствено. Правило 2 се реализира така: ако първият символ на изречението е *N* (т. е. имаме изречение от вида ***NS***), то изречението е коректно тогава и само тогава, когато е правилно ***S***. Правило 3 обаче е свързано със сериозен проблем: как да намерим тези ***T*** и ***S***, за които ***CST***, ***DST***, ***EST*** или пък ***IST*** са правилни изречения. Очевидно трябва да се изследват всички възможности. Именно тук на помощ идва динамичното оптимиране. Ще въведем двуаргументна функция $F(i,j)$, която ще връща истина, ако частта от входното изречение $s_i, s_{i+1},..., s_j$ е правилно изречение на хедонийски език. Тъй като в процеса на проверка едни и същи поднизове ще се проверяват многократно, е удобно да съхраняваме резултатите от пресмятанията. Получаваме следната целева функция:

1. $F(i,j) = 1$, ако $i = j$ и $s_j \in \{ p, q, r, s, t, u, v, w, x, y, z \}$
2. $F(i,j) = 1$, ако $s_i = N$ и $F(i+1,j) = 1$.
3. $F(i,j) = 1$, ако $s_i \in \{C,D,E,I\}$ съществува k, такова че $F(i+1,k) = 1$ и $F(k+1,j) = 1$
4. иначе $F(i,j) = 0$

Следва реализация, като пресмятанията се извършват отгоре-надолу — *memoization*.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100
#define NOT_CALCULATED 2
unsigned char F[MAX][MAX]; /* Целева функция */
const char *s = "NNNNNNNNECINNxqpCDNNNNNwNNNtNNNNs"; /* За проверка */
unsigned n; /* Дължина на изречението */

void init(void)
{ unsigned i, j;
```

```
  n = strlen(s);
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        F[i][j] = NOT_CALCULATED;
}

unsigned char check(unsigned st, unsigned en)
{ unsigned k;
  if (NOT_CALCULATED != F[st][en])
    return F[st][en];
  else {
    /* Вместо следващите 2 реда */
    if (st == en)
        F[st][en] = (s[st] >= 'p' && s[st] <= 'z');
    else if ('N' == s[st])
      F[st][en] = check(st + 1, en);
    else if('C' == s[st] || 'D' == s[st] || 'E' == s[st] || 'I' == s[st])
    {
      k = st + 1;
      while (k < en && !(check(st + 1, k) && check(k + 1, en)))
             k++;
      F[st][en] = (k != en);
    }
    else
      F[st][en] = 0;
    return F[st][en];
  }
}

int main(void) {
  init();
  printf("\nИзреч. е %s", check(0,n-1) ? "правилно!" : "НЕПРАВИЛНО!!!");
  return 0;
}
```

hedonia.c

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.
2. Правилно ли е според Вас изречението `NNNNNNNNECINNxqpCDNNNNNwNNNtNNNNs`, подадено за проверка на горната програма? Защо?
3. Какъв е типът на граматиката на хедонийския език съгласно йерархията на Чомски?
4. Да се модифицира горната програма така, че да намира:
   а) един извод възможен извод;
   б) всички възможни изводи;
   в) най-краткия извод

8.3.10. Символно умножение

	a	***b***	***c***
a	*b*	*b*	*a*
b	*c*	*b*	*a*
c	*a*	*c*	*c*

Таблица 8.3.10. Примерна таблица на умножение в множеството $A = \{a, b, c\}$.

Нека е даден низ $x = x_1x_2...x_n$ над фиксирана азбука A с k букви. Целта е да се провери дали съществува такова поставяне на скоби в низа, че в резултат на извършване на умножението да бъде получена стойност a, където $a \in A$. Таблицата за умножение в A не е нито комутативна, нито асоциативна, така че редът на умножение е от съществено значение. Единственото изискване е за всеки $a,b \in A$ да бъде в сила $ab \in A$. Да вземем например азбуката $A = \{a, b, c\}$. Една възможна таблица на умножение в A е показана в *таблица 8.3.10*.

Да вземем израза abc. Ако извършим умножението в реда $(ab)c$, ще получим $(ab)c = bc = a$. В случай, че поставим скобите така $a(bc)$, ще получим $a(bc) = aa = b$.

Задачата ще решим с динамично оптимиране. Ще считаме, че азбуката A, таблицата на умножение в азбуката A, буквата-цел a, както и изразът x постъпват като параметри на програмата (в програмата по-долу те са фиксирани).

Ще дефинираме булевата функция `can(i,j,c)`, която връща истина, ако съществува поставяне на скобите, при което изразът, започващ от i-тата буква и завършващ в j-тата буква, получава стойност c ($1 \leq i \leq j \leq n$). В сила са формулите:

1) `can(i,i,c) == 1`, тогава и само тогава, когато $x_i == c$
2) `can(i,j,c) == 1`, тогава и само тогава, когато $i < j$ и съществуват c_1 и c_2 такива, че $c_1 * c_2 == c$ и съществува такова p ($i \leq p < j$), че `can(i,p,c1) == 1` и `can(p+1,j,c2)== 1`

Тук с x_i е означен i-тият знак на x, с * — операцията умножение на два символа. Пресмятането ще се извършва рекурсивно, като всички пресметнати резултати ще се съхраняват в масива `table[][][]`. Ще съхраняваме стойността на p, за която `can()` е станало 1. Това ще ни позволи да възстановим едно възможно поставяне на скобите по начин, подобен на начина, по който постъпихме при умножението на матриците.

```
#include <stdio.h>
#include <string.h>

#define NOT_CALCULATED (unsigned char)(-1)
#define MAXSLEN 100 /* Максимална дължина на низа */
#define LETTS   3   /* Брой букви */

/* Таблица на умножение */
char rel[LETTS][LETTS] = {
  { 'b', 'b', 'a' },
  { 'c', 'b', 'a' },
  { 'a', 'c', 'c' }};
char *s = "bacacbcabbbcacab";

unsigned char table[MAXSLEN][MAXSLEN][LETTS];
unsigned char split[MAXSLEN][MAXSLEN];

unsigned char can(unsigned char i, unsigned char j, unsigned char ch)
{ unsigned char c1, c2, pos;
  if (table[i][j][ch] != NOT_CALCULATED)
      return table[i][j][ch]; /* Вече сметнато */
  if (i == j)
      return(s[i] == ch + 'a');
  for (c1 = 0; c1 < LETTS; c1++)
    for (c2 = 0; c2 < LETTS; c2++)
      if (rel[c1][c2] == ch + 'a')
        for (pos = i; pos <= j - 1; pos++)
          if (can(i, pos, c1))
            if (can(pos + 1, j, c2)) {
```

```
              table[i][j][ch] = 1;
              split[i][j] = pos;
              return 1;
            }
  table[i][j][ch] = 0;
  return 0;
}

void putBrackets(unsigned char i, unsigned char j)
{ /* Поставя скобите с израза */
  if (i == j)
    printf("%c", s[i]);
  else {
    printf("(");
    putBrackets(i, split[i][j]);
    printf("*");
    putBrackets(split[i][j] + 1, j);
    printf(")");
  }
}

int main(void) {
  unsigned char len = strlen(s);
  memset(table, NOT_CALCULATED, sizeof(table));
  if (can(0, len - 1, 0))
    putBrackets(0, len - 1);
  else
    printf("Няма решение");
  return 0;
}
```

relation.c

Резултат от изпълнението на програмата:

```
(((b*a)*(c*a))*(((c*(b*c))*a)*((b*((b*b)*(c*a)))*(c*(a*b)))))
```

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.
2. Да се реализира итеративен вариант на програмата и да се сравни с рекурсивния.
3. Да се намерят всички решения.

8.4. Други интересни оптимизационни задачи

По-долу ще разгледаме някои други интересни оптимизационни задачи, които обикновено не се приемат като класически, но при които прилагането на динамично оптимиране е естествен и правилен подход. Техните решения са интересни и дават възможност на читателя да добие по-пълна представа за начина на прилагане на метода в реални алгоритмични задачи. По наше мнение динамичното оптимиране се усвоява най-добре на базата на конкретни примери, поради което сме се постарали да осигурим възможно по-голям брой, като същевременно сме се опитали да наблегнем и на разнообразието.

8.4.1. Такси компания

Задача: Движението на "Експрес такси" по дълга улица е организирано така, че да има спирка на всеки километър. "Експрес такси" се движи по улицата от всяка спирка 1, 2, ..., 10 километра без спиране. Цената за всяко от разстоянията е означено с c_i (i = 1, 2, ..., 10), например:

c_1 = 12
c_2 = 21
c_3 = 31
c_4 = 40
c_5 = 49
c_6 = 58
c_7 = 69
c_8 = 79
c_9 = 90
c_{10} = 101

Пътник иска да пътува n ($1 \leq n \leq 100$) километра. Какви разстояния на пътуване трябва да избере, така че пътуването да му излезе най-евтино, и каква е цената на цялото пътуване?

Решение: Не е трудно да се забележи, че условието броят на спирките да бъде точно 10 е несъществено. Ето защо ще разгледаме задачата в по-общия случай, когато броят k се задава като параметър.

Лесно се виждам че задачата може да се реши с динамично оптимиране. Да въведем целева функция $F(i)$, даваща минималната цена за i километра. Нека c_j е дадената по условие тарифа за превоз на j километра. Тогава можем да дефинираме $F(i)$ така.

$$F(0) = 0.$$
$$F(i) = \min[F(i-j) + c_j],\ 1 \leq i \leq n,\ 1 \leq j \leq \min(i,k)$$

Стойностите на целевата функция $F(i)$ ще пресмятаме последователно, започвайки от 1. За всяко i ($1 \leq i \leq n$) ще пазим намерената стойност на $F(i)$, както и разстоянието j от рекурентната формула, за което тя се постига. Така, при изчислението на $F(i)$ всички необходими ни предишни стойности на F ще бъдат вече пресметнати. Разстоянието j не ни е необходимо за изчисленията, но трябва да се пази, тъй като то ни позволява за всяко i да намерим едно конкретно пътуване с минимална стойност.

Възстановяването на търсения път се извършва със сложност $\Theta(n)$. Последният елемент от този път ще бъде това j_1, което съответства на $F(n)$. Предпоследният елемент — j_2, съответстващо на $F(n-j_1)$ и т. н. След обръщане на получената последователност получаваме търсения минимален път. В предложената реализация резултатът се извежда директно, без да се обръща. Това се прави основателно, тъй като между отделните елементи не съществува връзка. Тогава при произволна пермутация на елементите, съставящи даден път, отново получаваме валиден път, при това със същата цена. В частност последното важи и при обръщане последователността на елементите.

В реализацията по-долу пресметнатите стойности на целевата функция F, както и последната измината отсечка, се запазват в масива `dist[]`. Основната функция `solve()` съдържа два вложени `for`-цикъла със сложности съответно $\Theta(n)$ и $\Theta(k)$, като сложността на тялото им е константа, т. е. $\Theta(1)$. Оттук за общата им сложност получаваме $\Theta(n.k)$.

```
#include <stdio.h>
#define MAXN    100     /* Максимален брой километри */
#define MAXK    20      /* Максимален брой спирки */
#define INFINITY (unsigned)(-1)

struct {
  unsigned last;          /* Последна измината отсечка */
  unsigned value;         /* Цена на разстоянието */
```

```
} dist[MAXN];

const unsigned values[MAXK+1] = {0,12,21,31,40,49,58,69,79,90,101};
const unsigned n = 15;
const unsigned k = 10;

void solve(unsigned n) /* Решава задачата чрез динамично оптимиране */
{ unsigned i, j;
  dist[0].value = 0;
  for (i = 1; i <= n; i++) {
    dist[i].value = INFINITY;
    for (j = 1; j <= k && j <= i; j++)
      if (dist[i - j].value + values[j] < dist[i].value) {
        dist[i].value = dist[i - j].value + values[j];
        dist[i].last = j;
      }
  }
}

void print(unsigned n)  /* Извежда резултата на екрана */
{ printf("\n%s%u", "Обща стойност на пътуването: ", dist[n].value);
  printf("\nДължина и стойности на отделните отсечки:");
  while (n > 0) {
    printf("\n%u %u", dist[n].last, values[dist[n].last]);
    n -= dist[n].last;
  }
}

int main(void) {
  solve(n);
  print(n);
  return 0;
}
```
🖫 taxi.c

Резултат от изпълнението на програмата:

```
Обща стойност на пътуването: 147
Дължина и стойности на отделните отсечки:
3 31
6 58
6 58
```

Задачи за упражнение:

1. Да се реализира рекурсивен вариант на програмата и да се сравни с итеративния.
2. Да се намерят всички решения.
3. Използва ли се в горната програма фактът, че всяка пермутация на отсечките от дадено решение също е решение? Ако да — къде и как? Ако не — как би могло да стане и нужно ли е?

8.4.2. Билети за влак

Подобна на предходната е следната задача [*Timus, задача 1031*]:

Железопътната линия "Екатеринбург-Свердловск" има множество гари. ЖП линията може да бъде представена като отсечка, а гарите — като точки върху нея. Линията започва от гара "Екатеринбург" и завършва в "Свердловск", като гарите са номерирани в този ред ("Екатеринбург" има номер 1).

Цената на билета между две гари зависи само от разстоянието между тях. Цените са дадени в *таблица 8.4.2*.

Разстояние между гарите — X	**Цена на билета**
$0 < X \le L_1$	C_1
$L_1 < X \le L_2$	C_2
$L_2 < X \le L_3$	C_3

Таблица 8.4.2. Ценоразпис на билетите за различни разстояния между гарите.

Директни билети между две гари могат да бъдат закупени, само ако разстоянието между тях не надвишава L_3. Така понякога се оказва необходимо да се закупуват няколко билета за част от пътя между тях. Ще отбележим, че всеки билет е валиден за пътуване между две гари, т. е. началото и краят му трябва да бъдат гари.

Търси се минималната цена на билетите за придвижване между две зададени гари.

Задачата се решава аналогично на предходната. Разбира се, тук има някои особености: имаме конкретни "заковани" гари, което ограничава възможностите за избор на вид билети.

Ще решим задачата отгоре-надолу с рекурсия и *memoization*. Идеята е, че минималната цена $\min Price(k)$ на билета от началната гара до гара с номер k може да се получи по един от следните 3 начина:

1) $minPrice(i_1) + C_1$, където i_1 е номерът на най-далечната гара, вляво от k, разстоянието до която е по-малко или равно на L_1
2) $minPrice(i_2) + C_2$, където i_2 е номерът на най-далечната гара, вляво от k, разстоянието до която е по-малко или равно на L_2
3) $minPrice(i_3) + C_3$ където i_3 е номерът на най-далечната гара, вляво от k, разстоянието до която е по-малко или равно на L_3

Да означим с s номера на началната гара. Тогава

$$minPrice(k) = 0,\ k = 0, 1, \ldots, s$$
$$minPrice(k) = \min(minPrice(i_1) + C_1, minPrice(i_2) + C_2, minPrice(i_3) + C_3),\ k > s$$

Предлагаме на читателя сам да се убеди във верността на горните разсъждения.

```
#include <stdio.h>

#define MAX 8000
#define NOT_CALCULATED  (unsigned long)(-1)

unsigned long minPrice[MAX];/* Мин. цена от началната до текущата гара */

const unsigned long dist[MAX] = {0,3,7,8,13,15,23}; /* Разстояние от */
                                                     /* началната гара */
const unsigned long l1 = 3,  l2 = 6,  l3 = 8,
                    c1 = 20, c2 = 30, c3 = 40;
const unsigned n = 7;
const unsigned end = 6;
unsigned start = 2;

unsigned long calc(unsigned cur)
{ unsigned i;
  unsigned long price;
  if (NOT_CALCULATED == minPrice[cur]) {

    /* Търсим най-лявата гара и пресмятаме евентуалната цена,
       ако вземем билет тип 1 */
    for (i = cur - 1; i >= start && (dist[cur] - dist[i]) <= l1; i--);
    if (++i < cur)
```

```
      if ((price = calc(i) + c1) < minPrice[cur])
        minPrice[cur] = price;

    /* Търсим най-лявата гара и пресмятаме евентуалната цена,
       ако вземем билет тип 2 */
    for (; i >= start && (dist[cur] - dist[i]) <= l2; i--);
    if (++i < cur)
      if ((price = calc(i) + c2) < minPrice[cur])
        minPrice[cur] = price;

    /* Търсим най-лявата гара и пресмятаме евентуалната цена,
       ако вземем билет тип 3 */
    for (; i >= start && (dist[cur] - dist[i]) <= l3; i--);
    if (++i < cur)
      if ((price = calc(i) + c3) < minPrice[cur])
        minPrice[cur] = price;
  }
  return minPrice[cur];
}

int main(void) {
  unsigned i;

  /* Инициализация */
  for (i = 0; i < start; i++)
      minPrice[i] = 0;
  for (; i < n; i++)
      minPrice[i] = NOT_CALCULATED;
  /* Решаване на задачата */
  start--;
  printf("Минимална цена: %lu\n", calc(end-1));
  return 0;
}
```

railway.c

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.
2. Да се модифицира програмата така, че да намира и отпечатва подробно разписание за движението: начало, край, дължина и тарифа за всяка отсечка.
3. Предложеното решение винаги търси *най-далечната* гара вляво. Съгласни ли сте с това ограничаване на изследваните възможности? Възможно ли е да бъде пропуснато решение? Доколко това ускорява алгоритъма? Може ли да се приложи при друг разгледан алгоритъм, основан на динамично оптимиране?

8.4.3. Числов триъгълник

Задачите за търсене на оптимален път в граф често се решават чрез техниката на динамичното оптимиране. Класически пример в това отношение е задачата за търсене на минимален път в триъгълник:

Задача (Международната олимпиада по информатика в Швеция):
Разглежда се числов триъгълник от вида:

$$
\begin{matrix}
a_1 & & \\
a_2 & a_3 & \\
a_4 & a_5 & a_6
\end{matrix}
$$

a_7 a_8 a_9 a_{10}

с n реда и съдържащ $n(n+1)/2$ положителни числа. Числото x от i-тия ред ($1 \leq i \leq n$) и числото y от (i+1)-вия ред се наричат *съседни*, ако са разположени едно под друго в числовия триъгълник или числото y е с една позиция надясно или наляво спрямо числото x. Например съседни на a_5 са числата a_7, a_8 и a_9. Числото a_2 има за съседни само числата a_4 и a_5.

Да се състави програма, която намира редица от n числа в числовия триъгълник, удовлетворяваща едновременно свойствата:

- първото число от редицата е числото от първи ред;
- всеки две последователни числа на редицата за съседни, съгласно определението за съседство, дадено по-горе;
- сумата от членовете на редицата е максимална.

Решение: Разглеждаме ориентирания граф $G(V,E)$, с върхове $V = \{a_1, a_2, \ldots, a_{n(n+1)/2}\}$ и ребра $E = \{(a_i,a_j) \mid a_i$ и a_j са съседни$\}$. Тогава задачата се свежда до намиране на път в G с дължина n, за който сумата от върховете е минимална. Съгласно условието, произволен път с дължина n ще започва от a_1 (там започват всички пътища) и ще завършва в последния ред на триъгълника, т. е. в някое от $a_{n(n+1)/2-n+1}, a_{n(n+1)/2-n+2}, \ldots, a_{n(n+1)/2}$ (всеки следващ връх от пътя е едно ниво по-надолу от предходния, т. е. на практика имаме *троично* дърво.).

Ясно е, че за да определим търсения минимален път, трябва да намерим всички минимални пътища в G, завършващи във върхове от последния ред на триъгълника. Бихме могли да приложим някакъв вид алгоритъм с изчерпване, който, построявайки всички пътища, завършващи в последния ред на триъгълника, ще построи и търсения минимален път. Тъй като при построяването на пътя на всяка стъпка имаме 2 или 3 възможности за продължение, един такъв алгоритъм очевидно би имал експоненциална сложност.

Не е трудно да се забележи, че задачата може да се реши с техниката на динамичното оптимиране. Тъй като числата от триъгълника са положителни, то добавянето на нов връх към някой частичен път ще води до строго увеличаване цената на пътя. Тогава минималният път p_x до произволен връх x от ниво $k = 2, 3, \ldots, n$ ще има едно особено свойство: Нека y е непосредствен предшественик на x в p_x. Премахвайки последния връх от p_x (т. е. x), получаваме път p_y до върха y. При това полученият път p_y е минимален. Действително, ако допуснем, че съществува път p'_y, с по-ниска цена от тази на p_y, то, добавяйки върха x към p'_y, получаваме път p'_x, с по-ниска цена от тази на p_x, което е противоречие с минималността на p_x. Така установихме, че е налице най-важното условие за да може да се прилага успешно техниката на динамичното оптимиране — оптималното решение има оптимална подструктура, т. е. състои се от оптимални решения на задачата за по-малка размерност.

Тогава намирането на минималния път до даден връх се свежда до намиране на минималния път до всички негови преки предшественици. За да построим минималните пътища с дължина k, трябва да знаем всички минимални пътища с дължина k–1. За да намерим тях, са ни необходими всички пътища с дължина k–2 и т. н. Така, за да решим задачата, възниква необходимостта от построяване на минималните пътища до всеки от върховете на триъгълника.

На пръв поглед може да не личи, че предложеният алгоритъм е по-добър от пълното изчерпване, тъй като тук също се строят пътища до всеки връх. За разлика от пълното изчерпване обаче тук се изграждат *не всевъзможните*, а само *минималните* пътища до даден връх. При това преминаването от минимален път с дължина k към минимален път с дължина k+1 става почти директно и е свързано с не повече от три проверки, тъй като всеки връх, с изключение на тези от последния ред, има два или три наследника. Въвеждаме следните означения:

`v[x]` — цена за преминаване през върха x (стойност на x-тото число от триъгълника);
`p[x]` — цена на минималния път до върха x;
`pred[x]` — пряк предшественик на върха x в минималния път до x.

Удобно е изграждането на минималните пътища да става по редове, като се започне от първия ред. В зависимост от начина на актуализиране и изграждане на минималния път получаваме два различни линейни алгоритъма:

Алгоритъм 1

Фиксира се връх и се актуализират минималните пътища до наследниците му:

1) Инициализиране:

```
p[1] = v[1];
p[x] = +∞, x = 2, 3 ,…, n(n+1)/2.
```

2) Преглеждаме последователно елементите от ниво $k = 1, 2, \ldots, n-1$:

   а) Нека сме намерили минималния път `p[x]` до даден връх x от ниво k ($k < n$). Тогава за минималните пътища на съседните му елементи y от ниво $k + 1$ имаме:

   `p[y] = min(p[y],p[x]+v[y]).`

   б) Ако на *стъпка а* сме актуализирали `p[y]`, то x е непосредствен предшественик на y в текущия минимален път до y:

   `pred[y] = x.`

Алгоритъм 2

Фиксира се връх и директно се намира минималният път до него, като се използват намерените минимални пътища до неговите предшественици:

1) Инициализиране:

```
p[1] = v[1];
p[x] = +∞,  за x = 2,3,…, n(n+1)/2.
```

2) Преглеждаме последователно елементите от ниво $k = 2, 3, \ldots, n$:

   а) Нека сме намерили минимален път `p[y]` до всеки връх y от ниво $k-1$ ($k \leq n$). Тогава за минималния път до произволен връх x от ниво k имаме:

   `p[x] = min(p[y′]+v[x],p[y″]+v[x],p[y‴]+v[x]),`

   където y', y'' и y''' са съседните на x елементи от ниво $k-1$. Могат да бъдат само два или дори само един (виж триъгълника по-горе).

   б) Ако на *стъпка а*) сме актуализирали `p[x]` за някое $y \in \{y', y'', y'''\}$, то y е непосредствен предшественик на x в текущия минимален път до x:

   `pred[x] = y.`

Ясно е, че предложените алгоритми са линейни по броя на върховете N в триъгълника. Вземайки предвид, че $N = n(n+1)/2$, получаваме алгоритмична сложност $\Theta(n^2)$.

Съхраняването на числата от триъгълника в матрица води до неефективно използване на паметта — почти половината елементи са неизползвани. Добре би било да се помисли за линеаризация на матрицата. Едно очевидно решение е използването на едномерен масив, в който числата се разполагат по редове последователно отляво-надясно — така, както ще ги обработваме. Следва реализация на *Алгоритъм 1*:

```
#include <stdio.h>

#define MAX   100
#define MAX2  MAX * (MAX + 1) / 2
#define INFINITY (unsigned)(-1)

unsigned p[MAX2];          /* Минимална цена за достигане на възел I */
unsigned pred[MAX2];       /* Предшественик на възел i в мин. път */

const unsigned v[MAX2] = /* Цени за преминаване през върховете */
    { 0,1,22,33,5,6,77,8,22,7,225,177,738,737,778,39,28,9,10,11,12,13 };
```

```
const unsigned n = 6;              /* Брой редове в триъгълника */

void compare(unsigned ind1, unsigned ind2)
{ if (p[ind1] > p[ind2] + v[ind1]) {
    p[ind1] = p[ind2] + v[ind1];
    pred[ind1] = ind2;
  }
}

/* Намира минималния път до всеки възел */
void findMinPath(void)
{ unsigned i, j, sum;
  /* До първия връх се стига непосредствено */
  for (p[1] = v[1], sum = 0, i = 1; i <= n; i++) {
    for (j = 1; j <= i; j++) {
      if (j > 1)
            compare(sum + j + i - 1, sum + j);
      compare(sum + j + i, sum + j);
      compare(sum + j + i + 1, sum + j);
    }
    sum += i;
  }
}

/* Извежда триъгълника на минималните пътища на екрана */
void print(const unsigned m[])
{ unsigned i, j, sum;
  for (sum = 0, i = 1; i <= n; printf("\n"), sum += i++)
    for (j = 1; j <= i; j++)
        printf("%8u", m[sum + j]);
}

/* Извежда минималния път до върха x */
void writePath(unsigned x)
{ printf("\n%s%u%s%u", "Пътят до връх номер ", x,
                       " е минимален: ", p[x]);
  printf("\nПътят, изведен в обратен ред (индекси): ");
  while (x != 1) {
    printf("%u ", x);
    x = pred[x];
  }
  printf("1");
}

/* Намира връх от последния ред с минимален път */
void findMinLastRow(void)
{ unsigned i, minInd, end = n * (n + 1) / 2;
  for (i = 1 + (minInd = end-n+1); i <= end; i++)
    if (p[i] < p[minInd])
        minInd = i;
  writePath(minInd);
}

int main(void) {
  unsigned i;
  for (i = 0; i < (n + 3)*(n + 2)/2; i++)
      p[i] = INFINITY;
  printf("Изходен триъгълник:\n"); print(v);
  findMinPath();
```

```
  printf("\nТриъгълник на минималните пътища:\n"); print(p);
  findMinLastRow();
  return 0;
}
```

🖫 triangle.c

Резултат от изпълнението на програмата: (допълнително сме почернили пътя)

```
Изходен триъгълник:
      1
     22      33
      5       6      77
      8      22       7     225
    177     738     737     778      39
     28       9      10      11      12      13

Триъгълник на минималните пътища:
      1
     23      34
     28      29     111
     36      50      36     336
    213     774     773     814     375
    241     222     783     386     387     388

Пътят до връх номер 17 е минимален: 222
Пътят, изведен в обратен ред (индекси): 17 11 7 4 2 1
```

Задачи за упражнение:

1. Да се използва линеаризирана матрица за представяне на триъгълника.
2. Да се намерят всички пътища. Каква е сложността в този случай? Зависи ли от броя на пътищата?
3. Да се реши задачата, като се тръгне от последния ред нагоре към върха на триъгълника.

8.4.4. Представяне на сума с минимален брой монети

Задача: Дадени са n монети със стойности съответно: $c_1, c_2, \ldots, c_n$ и естествено число S. Да се намери представяне на S с минимален брой монети измежду наличните. Стойностите $c_1, c_2, \ldots, c_n$ са цели положителни числа.

Решение: Настоящата задача се решава леко по метода на динамичното оптимиране. Да означим с $F(k)$ минималния брой монети, необходими за представяне на k. Тогава са в сила формулите:

$$F(k) = \begin{cases} 0 & k = 0 \\ 1 + \min\limits_{i=1,2,\ldots,n;\, k \ge c_i} \left[F(k - c_i) \right] & k > 0 \end{cases}$$

В частност $F(S)$ ще ни даде решението на задачата.

Стойностите на $F(k)$ ще изчисляваме последователно, започвайки от $k = 1$. Така, при изчислението на $F(k)$ за някакво конкретно k всички необходими ни предишни стойности на F ще бъдат вече пресметнати (Стойностите на монетите са цели и строго положителни). За всяка сума k ще пазим освен минималния брой монети, за който се постига, и индекса на последната добавена монета. Това ще ни позволи за произволно вече разгледано k да възстановим едно конкретно множество от монети. Намирането на такова множество ни е необходимо освен това, за да си гарантираме, че всяка монета участва в сумата *не повече от веднъж*. В приложената програма булевата функция `canJ()` проверява дали j-тата монета може да участва в i-тата сума. Условията за това са:

1. $c_j \leq i$
2. Сумата $i - c_j$ да може да се получи с наличните монети
3. j да не участва в получаването на сумата $i - c_j$

При проверката дали се удовлетворява третото условие (което е еквивалентно на естественото изискване всяка монета да участва не повече от веднъж в сумата) се налага явно намиране на участващите в сумата $i - c_j$ монети. Аналогична операция се извършва и накрая, след изчисляването на $F(S)$, за получаване на конкретно множество от монети, за които се постига.

Възстановяването на конкретно множество се извършва със сложност $\Theta(S)$. Нека последният добавен елемент при получаването на сумата да е бил j_1. Тогава предпоследният j_2 ще бъде последният, съответстващ на сумата $i - c_{j_1}$. Предпредпоследния елемент j_3 получаваме като съответен на сумата $i - c_{j_1} - c_{j_2}$ и т.н. Процесът продължава до достигане на сума без предшественик, а именно нулевата.

Да се опитаме да оценим сложността на предложения алгоритъм. Функцията `findMin()` съдържа два вложени цикъла `for` със сложности съответно $\Theta(S)$ и $\Theta(n)$, в тялото на които се съдържа обръщение към функцията `canJ()`, която от своя страна има сложност от порядъка на $\Theta(n)$. Тогава общата алгоритмична сложност е от порядъка на $\Theta(S.n.n)$. Ако предположим, че максималната допустима стойност на монета е ограничена от известна и предварително фиксирана константа M, то $S \leq M.n$. Оттук получаваме $\Theta(M.n^3)$. Или, след изключване на константата M — $\Theta(n^3)$.

Характерно за горния алгоритъм е, че като страничен ефект намира минималния брой необходими монети, както и самите монети, не само за търсената сума, а и за *всички* суми, по-малки или равни на S. Тогава, ако положим S да бъде общата стойност на наличните монети, то в масива `sums[]` ще получим минималните представяния на всевъзможните суми на монетите — при това отново със сложност $\Theta(S.n^2)$.

```
#include <stdio.h>
#define MAXCOINS  100        /* Максимален брой монети */
#define MAXSUMA   1000       /* Максимална сума */
struct {
  unsigned num;              /* Брой монети в сумата */
  unsigned last;             /* Последна добавена монета */
} sums[MAXSUMA];             /* Целева функция */

/* Стойности на монетите */
const unsigned coins[MAXCOINS] = {0,5,2,2,3,2,2,2,4,3,5,8,6,7,9};
const unsigned sum = 31; /* Сума, чието представяне минимизираме */
const unsigned n = 14;   /* Общ брой налични монети */

/* Дали можем да използваме j-тата монета в i-тата сума? */
char canJ(unsigned i, unsigned j)
{ int k = i - coins[j];
  if (k > 0 && sums[k].num < MAXCOINS)
    while (k > 0) {
      if (sums[k].last == j) break; /* монета j участва в сумата */
      k -= coins[sums[k].last];
    }
  return(0 == k);
}

/* Намира представяне на сумата Sum с минимален брой монети */
void findMin(unsigned sum)
{ unsigned i, j;
  sums[0].num = 0;
  for (i = 1; i <= sum; i++) {
    sums[i].num = MAXCOINS;
```

```
    for (j = 1; j <= n; j++) {
      if (canJ(i, j))
        if ((sums[i - coins[j]].num + 1) < sums[i].num) {
          sums[i].num = 1 + sums[i - coins[j]].num;
          sums[i].last = j;
        }
    }
  }
}

void print(unsigned sum)
{ /* Извежда намереното представяне */
  if (MAXCOINS == sums[sum].num)
    printf("\nСумата не може да се получи с наличните монети.");
  else {
    printf("\nМинимален брой необходими монети: %u", sums[sum].num);
    printf("\nЕто и стойностите на самите монети: ");
    while (sum > 0) {
      printf("%u ", coins[sums[sum].last]);
      sum -= coins[sums[sum].last];
    }
  }
}

int main(void) {
  printf("\n%s %u %s", "Как да получим сума от",
         sum,"лева с минимален брой монети?");
  findMin(sum);
  print(sum);
  return 0;
}
```

coins.c

Резултат от изпълнението на програмата:

```
Минимален брой необходими монети: 5
Ето и стойностите на самите монети: 5 2 8 7 9
```

Задачи за упражнение:

1. Необходимо ли е непременно да се пази последната добавена монета за всяка възможна сума, за да се гарантира, че няма да взема втори път една и съща монета?
2. Да се предложи нова рекурентна формула, която да включва изискването да няма повторение на монетите.
3. Да се сравни текущата задача със задачата за раницата.

8.4.5. Опроводяване на платка

Задача: В две срещуположни страни на правоъгълна платка са разположени симетрично един срещу друг n ($n < 1000$) накрайника, номерирани последователно с числата от 1 до n. Дадена е и пермутацията $Q[1..n]$ на числата от 1 до n. Да означим с P обратната пермутация Q^{-1} на Q. В сила е равенството: $P(i) = j \Leftrightarrow Q(j) = i$. Необходимо е да се свърже с проводник всеки накрайник i от едната страна на платката със съответния му накрайник $P(i)$ от другата страна. Проводниците са прави и никои два не трябва да се пресичат. Да се намери максималният брой непресичащи се проводници, които могат едновременно да свържат съответни двойки накрайници.

Решение: Не е трудно да се забележи, че задачата може да се реши с динамично оптимиране. Въвеждаме двуаргументна целева функция $F(i,j)$, която ще ни дава максималния брой непреси-

чащи се проводници, които могат да се прокарат в частта от платката, включваща накрайниците от i до n в горната част и от j до n в долната част. В сила е рекурентната зависимост:

$$F(i,j) = 1 + \max_{k,(i \le k \le n, P(k) \le n)} F(k+1, P(k)+1)$$

Налагаме и следните естествени гранични условия:

$$F(i,j) = 0, \text{ за } i > n \text{ или } j > n$$

От приведената дефиниция на $F(i,j)$ лесно се вижда, че търсеният максимален брой проводници се дава от $F(1,1)$. Всички възможности за i и j са точно n^2 и за намиране на $F(i,j)$ при фиксирани стойности на аргументите и използване на таблица за съхраняване на вече пресметнатите стойности се извършват най-много n операции — за намиране на максимума в първата формула. (Ще отбележим, че за разлика от останалите алгоритми, които бяха разгледани по-горе, тук пресмятанията се извършват отзад-напред, т. е. рекурсията в дефиницията е обърната.) Оттук лесно следва, че задачата може да бъде решена за време $\Theta(n^3)$ при използване на памет $\Theta(n^2)$. Бихме могли да възстановим и едно възможно оптимално свързване, въвеждайки допълнителна таблица с размери $n \times n$ или пък направо използвайки данните от таблицата на F по добре познатата ни схема.

Дали обаче няма и по ефективен алгоритъм? Да разгледаме друга възможна дефиниция на функцията F:

$$F(i,j) = 0, \text{ за } i > n \text{ или } j > n$$
$$F(i,j) = \max(1 + F(i+1, P(i) + 1), F(i+1, j)) \text{ при } P(i) \le j$$

Това е очевидната формула за динамично оптимиране — вземаме по-добрата от двете възможности — да свържем накрайника i със съответния му накрайник $P(i)$, ако може, или да пропуснем да свържем накрайника i и да опитаме следващия $i+1$. Така можем да решим задачата за време $\Theta(n^2)$ и памет $\Theta(n^2)$.

Несъмнено времевата сложност на един алгоритъм е най-важната му характеристика. При решаване на задачи от областта на динамичното оптимиране не по-малко съществен е и проблемът за сложността по памет, тъй като тя налага директни ограничения върху максималната размерност на задачата, за която е пригоден алгоритъмът. При много задачи, решавани с динамично оптимиране, е естествено за съхраняване на резултатите от пресмятанията да се използва правоъгълна таблица. В повечето случаи, както впрочем вече видяхме (*виж 8.2.1.*), се оказва възможно да се съхраняват само част от редовете на таблицата (най-често два: текущият и предходният).

Този подход се оказва приложим и в нашия случай. Рекурентната зависимост относно параметъра i е много проста: Елементите $F(i,1...n)$ зависят само от елементите $F(i+1,1..n)$, което ни дава основание да помислим за алгоритъм с линейна памет. Да означим с $F(x)$ максималния брой непресичащи се кабели, които могат да се прокарат между накрайниците x, x_1, ..., n. Търсим стойностите на $F(x)$ от n-тата към първата:

```
for (i = 1; i <= n; i++)
  F[i] = 1;
for (k = n; k >= 1; k--)
  for (i = k + 1; i <= n; i++)
    if (p[k] < p[i])
      if (1 + F[i] > F[k])
        F[k] = 1 + F[i];
```

С други думи, ако кабелите `x-P[x]` и `i-P[i]` не се пресичат, можем да подобрим максималната за момента сума за `F[x]`, която в началото е 0, като опитаме да прокараме кабела x–`P[x]` и добавим към сумата броя кабели `F[i]` от i нататък. Търсената оптимална стойност накрая е max(`F[i]`) за $1 \le i \le n$. В случай, че желаем да намерим едно конкретно свързване на множеството

от накрайници, можем да въведем допълнителен масив `next[1..n]`, който да съдържа следващия проводник, за който се е получила максимална стойност на `F[i]`, или 0 — ако няма такъв.

```
#include <stdio.h>
#define MAX 1000
unsigned F[MAX];                             /* Целева функция */
unsigned next[MAX];                          /* Следващ проводник */
const unsigned n = 9;                        /* Брой накрайници */
const unsigned p[MAX]={0,9,1,3,6,2,7,5,4,8}; /* Изх. пермутация */

void solve(void)
{ unsigned k, i;
  /* Инициализиране */
  for (i = 1; i <= n; i++) {
    F[i] = 1;
    next[i] = 0;
  }
  /* Основен цикъл */
  for (k = n; k >= 1; k--)
    for (i = k + 1; i <= n; i++)
      if (p[k] < p[i])
        if (1 + F[i] > F[k]) {
          F[k] = 1 + F[i];
          next[k] = i;
        }
}

/* Извежда резултата на екрана */
void print(void)
{ unsigned i, max, index;
  for (max = F[index = 1], i = 2; i <= n; i++)
    if (F[i] > max) {
      max = F[i];
      index = i;
    }
  printf("\nМаксимален брой кабели: %u\n", max);
  do {
    printf("%u ", index);
    index = next[index];
  } while (index);
}

int main(void) {
  solve();
  print();
  return 0;
}
```

🖫 board.c

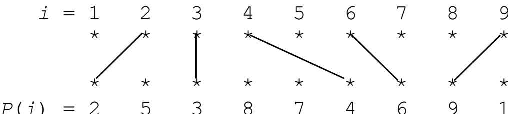
Да разгледаме една примерна схема от проводници ($n = 9$):

```
        i = 1   2   3   4   5   6   7   8   9
            *   *   *   *   *   *   *   *   *

            *   *   *   *   *   *   *   *   *
     P(i) = 2   5   3   8   7   4   6   9   1
```

Резултат от изпълнението на програмата:

```
Максимален брой кабели: 5
2 3 4 6 9
```

И съответната картинка:



Задачи за упражнение:

1. Да се сравнят двата описани алгоритъма.
2. Да се реализира първият алгоритъм.

8.4.6. На опашка за билети

Рокгрупа ще изнася концерт. На едно от бюрата се е оформила опашка от n фена. Всеки фен иска да закупи точно един билет, а касиерът продава най-много по два билета на фен. Знае се, че касиерът губи време T_i, за да обслужи i-тия фен. Същевременно два съседни фена, напр. j-ият и (j+1)-вият ($1 \leq j \leq n-1$) могат да се договорят само един да остане на опашката, а другият да напусне, ако времето R_j, необходимо на касиера да ги обслужи заедно, е по-малко от $T_j + T_{j+1}$. Следователно, за да се минимизира времето за обслужване и да се предпазят феновете от по-продължително чакане, е добре всеки фен да влезе в преговори с фена преди или след него.

Дадени са целите положителни стойности на n, T_i и R_j. Целта е да се минимизира общото време за работа на касиера, като се дефинират двойки от съседи в опашката по най-добрия възможен начин. Не е задължително всеки фен да влезе в двойка с някой от съседите си.

Ще решим задачата с динамично оптимиране. Нека предположим, че сме получили по някакъв начин минималното време за изпълнение на продажбите от първия до (i–1)-вия фен. Тогава можем да получим минималното време за продажба до i-тия фен, като използваме стойностите за i–1-вия и i–2-рия фен. Налице са точно две възможности: фенът i може или да се сдружи с фена i–1, или изобщо да не се сдружава с никого.

Да въведем целевата функция $F(i)$, която ще връща минималното време за обслужване на първите i фена включително. В сила са формулите:

$$F(i) = \min(F(i-1)+T_i,\ F(i-2)+R_{i-1}),\ 2 \leq i \leq n$$

Първата сума в скобите съответства на възможността i-ият фен да не се сдружава с никого, а втората — да се сдружи с фена i–1. Остава да дефинираме естествените гранични условия, а именно: $F(0) = 0$, $F(1) = T_1$.

В приложената програма стойностите на T_i и R_j са зададени като константа. Тъй като в езика Си индексирането на масивите започва винаги от 0, се налага модификация на горните формули така, че те придобиват вида:

$$F(i) = \min(F(i-1)+T_{i-1},\ F(i-2)+R_{i-2}),\ 2 \leq i \leq n$$
$$F(0) = 0,\ F(1) = T_0$$

Ще пресмятаме последователно $F(i)$ за i = 1, 2, ..., n. Използвайки пресметнатите стойности на $F(i)$ за $1 \leq i \leq n$, можем да намерим една възможна оптимална конфигурация от групирания на част от феновете по двойки. Търсенето започва отзад-напред, като дали фенът се е комбинирал или — не, се определя от това коя от двете суми в скобите е по-малка, т. е. съвпада с $F(i)$. Пресмятането на $F(n)$, както и възстановяването на групиранията, става за време $\Theta(n)$ и памет $\Theta(n)$. Следва кодът на програмата.

```
#include <stdio.h>
#define MAXN      1000
#define MIN(a, b) ((a) < (b) ? (a) : (b))
```

```
unsigned F[MAXN];                                    /* Целева функция */
const unsigned T[MAXN] = {8,5,3,9,2,1,4,4,1,17}; /* Време за 1 фен */
const unsigned R[MAXN] = {1,3,9,4,2,4,9,3,8};    /* Време за 2 фена */
unsigned n = 10;                                     /* Брой фенове */

/* Пресмята стойностите на целевата функция */
void solve(void)
{ unsigned i;
  F[0] = 0; F[1] = T[0];
  for (i = 2; i <= n; i++)
    F[i] = MIN(F[i - 1] + T[i - 1], F[i - 2] + R[i - 2]);
}

/* Извежда решението на екрана */
void print(void)
{ printf("\nМинимално време за обслужване на опашката: %u", F[n]);
  do
    if (F[n - 1] + T[n - 1] == F[n])
      printf("\n%u", n--);
    else {
      printf("\n(%u,%u)", n - 1, n);
      n -= 2;
    }
  while (n > 0);
}

int main(void) {
  solve();
  print();
  return 0;
}
```

tickets.c

Резултат от изпълнението на програмата:

```
Минимално време за обслужване на опашката: 24
(9,10)
8
(6,7)
(4,5)
3
(1,2)
```

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.
2. Да се сравни задачата с числата на Фибоначи (*виж 8.3.1.*).
3. Възможно ли е задачата да се реши с константна памет за целевата функция?

8.4.7. Разпределение на ресурси

Задача: Търговска компания разполага с известно количество стока n, което трябва да продаде в k града, като във всеки град може да продаде количество стока, ненадминаващо някаква предварително зададена константа M. Оказва се, че цената, на която могат да се предлагат различни количества стока във всеки град се влияе от редица фактори. Маркетинговите специалисти на фирмата са направили съответните проучвания на пазара и са обобщили

резултатите от изследването си в *таблица 8.4.7.*, в която е записана цената на всяко количество стока за всеки град (по-долу тези стойности ще се дават от функцията *v*).

ГРАД	Количество стока					
	0	***1***	***2***	***3***	***4***	***5***
1	0	10	15	25	40	60
2	0	15	20	30	45	60
3	0	20	30	40	50	60

Таблица 8.4.7. Цени за различни количества стока по градове.

Брой градове — 3
Максимално количество стока, което може да се продаде в един град — 5
Количество стока за разпределяне — 5

Да се състави програма, която разпределя по такъв начин наличната стока между градовете, че печалбата от нея да бъде максимална. За примера максималната печалба е 65 и може да се получи, ако в град 1 се продаде количество 0, в град 2 се продаде количество 4, а в град 3 — количество 1.

На практика от всеки ред на таблицата (от всеки град) трябва да се избере най-много един елемент, така че сумата от избраните елементи да бъде максимална и сумата от индексите на избраните елементи да не надвишава *n*. Допуска се част от стоката да остане непродадена, стига това да не води до намаляване на печалбата.

Задачата ще решим с динамично оптимиране. Въвеждаме целева функция $F(i,j)$, $1 \le i \le k$; $0 \le j \le n$. Функцията ще ни дава максималната печалба, която може да се получи, ако се продаде количество стока *j* в първите *i* града. В сила са формулите ($2 \le i \le k$, $1 \le j \le n$):

1. $F(i,0) = 0,\ 1 \le i \le k$
2. $F(1, j) = \max_l v(1,l),\ 1 \le l \le \min(j, M),\ 1 \le j \le n$
3. $F(i, j) = \max_l (v(i,l) + F(i-1, j-l)),\ 1 \le l \le \min(j, M)$

За първата формула: Ако няма останала стока за някои градове, то няма и печалба от тях. Втората формула се тълкува така: Ако останалата стока *j* трябва да се разпредели само за един град, то в този град се продава количество стока, което ще донесе максимален доход, като размерът на дохода се проверява непосредствено. Ще отбележим, че е възможно да остане непродадена стока, което, както по-горе вече споменахме, е допустимо.

Смисълът на третата формула е: Взема се максималната цена, получена от количество стока *i* в даден град, плюс максималната цена от количеството останала стока *j*–*i* в останалите *i*–1 града. За да се намери не само максималната печалба, но и начинът на получаването ѝ, се въвежда допълнителна матрица `amount[][]`, в която за всеки елемент на $F(i,j)$ се вписва количеството стока, което се заделя за града *i* при наличието на стока *j*. Пресмятанията се извършват рекурсивно, съгласно приведените по-горе формули, като всички пресметнати резултати се запазват, т. е. използва се *memoization*.

```
#include <stdio.h>
#include <string.h>

#define MIN(a, b)    ((a) < (b) ? (a) : (b))
#define MAXPERCITY 100 /* Максимално количество стока за град */
#define MAXCITIES   50 /* Максимален брой градове за търговия */
#define MAXCARGO   200 /* Макс. количество стока за разпределяне */
#define INFINITY    (unsigned)(-1)

unsigned F[MAXCITIES][MAXCARGO];       /* Целева функция */
unsigned amount[MAXCITIES][MAXCARGO];  /* Оптимално колич. стока */
```

```
unsigned inc;

unsigned k = 3;       /* Брой градове за разпределяне на стоката */
unsigned n = 5;            /* Количеството стока за разпределяне */
const unsigned M = 5;   /* Максим. количество за продажба в град */

const unsigned v[MAXCITIES][MAXPERCITY] = {   /* Цени на стоките */
  { 0, 10, 15, 25, 40, 60 },
  { 0, 15, 20, 30, 45, 60 },
  { 0, 20, 30, 40, 50, 60 }
};

unsigned maxIncome(unsigned city, unsigned ccargo)
{ unsigned i, max;
  if (0 == ccargo)
    return 0;               /* Ако няма стока, няма и печалба ;) */
  else if (0 == city) {
    /* Ако ccargo стока трябва да се разпредели само в 1 град, */
    /* се избира максим. печалба за този град от тази стока */
    for (i = max = 0; i <= MIN(ccargo, M); i++)
      if (max < v[city][i]) {
        max = v[city][i];
        amount[city][ccargo] = i;
      }
    F[city][ccargo] = max;
    return max;
  }
  else if (F[city][ccargo] != INFINITY) /* Ако е вече изчислена, */
    return F[city][ccargo];/* се взема стойността й от таблицата */
  else {
    /* Взема се макс. цена за колич. i стока в този град */
    /* плюс останалата (ccargo-i) стока в останалите градове */
    for (i = max = 0; i <= MIN(ccargo, M); i++) {
      inc = v[city][i] + maxIncome(city - 1, ccargo - i);
      if (max < inc) {
        max = inc;
        amount[city][ccargo] = i;
      }
    }
    F[city][ccargo] = max;
    return max;
  }
}

void scheduleCargo(void)
{ unsigned i,j;
  for (i = 0; i <= k; i++)
    for (j = 0; j <= n; j++)
      F[i][j] = INFINITY;
  F[k - 1][n] = maxIncome(k - 1, n);
}

void printResults(void)
{
  printf("\nМаксимален доход: %u", F[--k][n]);
  for(;;) {
    if (0 == n)
      printf("\nВ град %u продайте колич. 0.", k+1);
    else {
```

```
      printf("\nВ град %u продайте колич. %u.", k+1, amount[k][n]);
      n -= amount[k][n];
     }
      if (0 == k--) break;
   }
   if (n > 0)
     printf("\nОстава стока: %u", n);
}

int main(void)
{ scheduleCargo();
  printResults();
  return 0;
}
```

🖫 resource.c

Резултат от изпълнението на програмата:

```
Максимален доход: 65
В град 3 продайте количество 1.
В град 2 продайте количество 4.
В град 1 продайте количество 0.
```

Задачи за упражнение:

1. Да определи сложността на предложената по-горе реализация.
2. Да се реализира итеративен вариант на програмата.
3. Да се реши задачата при условие, че не се позволява да остава непродадена стока.

8.4.8. Семинарна зала

Зала е отпусната за една година на две партии — "сини" и "червени" за провеждане на семинари. Дните за заявка са номерирани от 1 до 365 и всяка заявка се състои от начален и краен ден. За да не възникнат конфликти, семинарите задължително се редуват: един за "сините", следващият — за "червените", следващият отново за сините и т.н. Няма значение коя партия започва първа, но не се допуска в един и същ ден да има повече от един семинар. Целта е при зададени n заявки на "сините" и m заявки на "червените" да се направи такова разписание за използване на залата, което да осигури максималната ѝ заетост.

Задачата можем да решим с динамично оптимиране. Ще въведем *две* целеви функции $B(d)$ и $R(d)$, които ще ни дават максималната сумарна заетост на залата за дните от 1 до d при условие, че семинарите на партиите се редуват и последният семинар в интервала $[1;d]$ е бил проведен от "синята" (съответно "червената") партия. Тогава търсената максимална заетост, изразена в брой заети дни на залата, ще бъде $\max(B(365), R(365))$. Функциите B и R можем да изчислим по следните рекурентни формули (Предполагаме, че годината има 365 дни и всеки от тях би могъл потенциално да попадне в заявката на някоя от двете партии.):

1. $B(0) = 0$
2. $B(d) = \max (B(d-1), R(b-1) + d-b+1)$, при $d > 0$, за всяка "синя" заявка $(b \div d)$
3. $R(0) = 0$
4. $R(d) = \max (R(d-1), B(b-1) + d-b+1)$, при $d > 0$, за всяка "червена" заявка $(b \div d)$)

Така, максималната заетост за първите d дни може да бъде получена по един от следните начини:

1) от максималната заетост за предния ден $d-1$;
2) като се изпълни някоя от заявките на "сините", завършваща в деня d, и се вземе максималната заетост за деня преди нейното начало, като се следи последният семинар до този ден да е бил на "червените";

3) като се изпълни някоя от заявките на "червените", завършваща в деня *d*, и се вземе максималната заетост за деня преди нейното начало, като се изисква последният семинар до този ден да е бил на "сините".

Можем да изчислим стойностите на двете функции за всички дни последователно, започвайки от първия. За да стане търсенето на заявките, завършващи в даден ден, по-бързо, ще сортираме двата списъка със заявки в нарастващ ред по завършването им. След това ще поддържаме две индексни променливи (за всяка партия по една), които ще указват номера на заявката, от която нататък всички заявки в съответния списък са след разглеждания ден. При изчисляване на *B* и *R* за всеки следващ ден *d* всички заявки, които свършват по-рано от *d*, остават вляво, а всички, които свършват по-късно или през *d*, остават вдясно от индексната променлива. В предложената по-долу реализация таблиците на целевите функции *B* и *R* са записи, съдържащи не общата заетост на залата в дни, а броя на дните, в които тя е дадена на сините и червените съответно. Разбира се, общата заетост директно се получава като сума на двете стойности, но така получаваме по-точна представа за разпределението.

Намирането на всички завършващи в деня *d* заявки се извършва, като се движи индексната променлива надясно, докато не се стигне до заявка, завършваща по-късно от този ден. Понеже дните са точно 365 и всяка от заявките се разглежда точно по веднъж, то сложността на алгоритъма за намиране на максимална заетост на залата е линейна относно общия брой заявки. Тъй като използваният алгоритъм за сортиране има сложност от порядъка на $n^2/2 + m^2/2$, общата сложност на алгоритъма е квадратична. При използване на друг подходящ алгоритъм за сортиране може да се очаква значително подобрение. Например можем да направим 365 списъка (за всеки ден по един) и с единично преминаване през елементите да поставим всеки елемент в съответния му списък. По този начин общата сложност на алгоритъма ще бъде линейна (*виж 3.2.4.*).

```
#include <stdio.h>

#define MAXN  5000  /* Максимален брой заявки */
#define MAXD  365   /* Максимален брой дни */

struct TZ { unsigned b, e; };
struct { unsigned cntBlue, cntRed; } B[MAXD], R[MAXD];

const unsigned n = 2; /* Брой сини заявки */
const unsigned m = 4; /* Брой червени заявки */

struct TZ blueOrders[MAXN] = { {1,5}, {12,20} };
struct TZ redOrders[MAXN] = { {2,10}, {6,11}, {15,25}, {26,30} };

/* Сортира заявките */
void sort(struct TZ Z[], const unsigned count)
{ unsigned i, j;
  struct TZ swp;

  for (i = 0; i < count; i++)
    for (j = i + 1; j < count; j++)
      if (Z[i].e > Z[j].e) {
        swp = Z[i];
            Z[i] = Z[j];
            Z[j] = swp;
      }
}

/* Решава задачата с динамично оптимиране */
void solveDynamic(void)
{ unsigned d, bb, be, blueIndex, redIndex;
```

```
  /* Инициализиране */
  B[0].cntBlue = B[0].cntRed = R[0].cntBlue = R[0].cntRed = 0;
  blueIndex = redIndex = 1;

  /* Пресмятане на B[1..MAXD], R[1..MAXD] */
  for (d = 1; d <= MAXD; d++) {
    /* Пресмятане на B[d] */
    B[d] = B[d - 1];
    for (blueIndex = 0; blueIndex < n; blueIndex++) {
      if (blueOrders[blueIndex].e > d)
        break;
      else {
        bb = blueOrders[blueIndex].b;
        be = blueOrders[blueIndex].e;
        if (R[bb-1].cntBlue + R[bb-1].cntRed + (be-bb+1)
           > B[d].cntBlue + B[d].cntRed) {
          B[d].cntBlue = R[bb - 1].cntBlue + (be - bb + 1);
          B[d].cntRed = R[bb - 1].cntRed + 0;
        }
      }
    }

    /* Пресмятане на R[d]: аналогично на B[d] */
    R[d] = R[d - 1];
    for (redIndex = 0; redIndex < m; redIndex++) {
      if (redOrders[redIndex].e > d)
        break;
      else {
        bb = redOrders[redIndex].b;
        be = redOrders[redIndex].e;
        if (B[bb-1].cntBlue + B[bb-1].cntRed + (be-bb+1)
          > R[d].cntBlue + R[d].cntRed) {
          R[d].cntBlue = B[bb - 1].cntBlue;
          R[d].cntRed = B[bb - 1].cntRed + (be - bb + 1);
        }
      }
    }
  }
}

/* Извежда резултата на екрана */
void printResult(void)
{ if (B[MAXD].cntBlue+B[MAXD].cntRed > R[MAXD].cntBlue+R[MAXD].cntRed) {
    printf("\nЗаетост на залата (дни): %u",
           B[MAXD].cntBlue + B[MAXD].cntRed);
    printf("\nБрой дни за червените: %u", B[MAXD].cntRed);
    printf("\nБрой дни за сините: %u", B[MAXD].cntBlue);
  }
  else {
    printf("\nЗаетост на залата (дни): %u",
           R[MAXD].cntBlue + R[MAXD].cntRed);
    printf("\nБрой дни за червените: %u", R[MAXD].cntRed);
    printf("\nБрой дни за сините: %u", R[MAXD].cntBlue);
  }
}

int main(void) {
```

```
  sort(blueOrders, n);
  sort(redOrders, m);
  solveDynamic();
  printResult();
  return 0;
}
```
🖫 room.c

Резултат от изпълнението на програмата:
```
Заетост на залата (дни): 25
Брой дни за червените: 11
Брой дни за сините: 14
```

В примера сините са дали заявки 1÷5 и 12÷20, а червените — 2÷10, 6÷11, 15÷25 и 26÷30.

Задачи за упражнение:

1. Да се предложи и реализира функция за възстановяване на едно възможно оптимално разписание. За примера по-горе една възможност е: R : 1÷5, B : 6÷11, R : 12÷20 и B : 26÷30.
2. Възможно ли е задачата да се реши само с една целева функция, вместо с две?

8.4.9. Крайпътни дървета

От едната страна на магистрала имало много дървета и в съответното пътно управление решили да отсекат някои от тях. В управлението разполагали със списък на дърветата заедно с височините им. Шефът на пътното управление бил човек със странни естетични идеи и решил да бъдат отрязани някои от дърветата по такъв начин, че тези, които останат, първо да нарастват, а след това да намаляват по височина. Освен това поискал средната височина на новата редица да бъде максимална (тайничко се надявал това да води до запазване на по-високите дървета и премахване главно на по-ниски). Наложено било и естественото изискване да се ограничи максимално броят на отсечените дървета.

И така, задачата е по дадена редица от естествени числа (височините на дърветата) да се намери такава нейна подредица с максимална дължина, че тя първо да бъде нарастваща, а от дадено място нататък да бъде намаляваща. От всички такива максимални редици да се намери тази, която има максимална сума от съставящите я елементи: тя ще има максимална средна височина, тъй като броят на елементите е вече фиксиран от това, че редицата е максимална по дължина.

Въвеждаме означението $X_{ij}= \{x_i, x_{i+1}, ..., x_j\}$. Задачата се разделя на 3 части:

(1) Намиране и запазване за всяко k (k = 1, 2, ..., n) на максимална *нарастваща* подредица на X_{1k}, чийто последен (и евентуално единствен) елемент е x_k.

(2) Намиране и запазване за всяко k ($k = n$, n–1, ..., 1) на максимална *намаляваща* подредица на X_{kn}, чийто първи (и евентуално единствен) елемент е x_k.

(3) Намиране на оптималната първо растяща, а после намаляваща редица.

За подзадача (1) се прилага динамично оптимиране. Да означим с $l(k)$, k = 0, 1, ..., n *дължината* на максималната *нарастваща* подредица на X_{1k}, чийто последен (и евентуално единствен) елемент е x_k. Да означим с $s(k)$ сумата от елементите на същата редица. В сила са следните рекурентни зависимости:

$$l(k) = \begin{cases} 0 & k = 0 \\ 1 + \max\limits_{i=0,1,\dots,k-1;\, x_i \le x_k} l(i) & k > 0 \end{cases}$$

Пресмятането на $s(k)$ не представлява проблем. Така, всяка максимална нарастваща подредица на X_{1k}, чийто последен (и евентуално единствен) елемент е x_k, може да се получи от някоя

предходна максимална нарастваща подредица на X_{1i} чрез добавяне на елемент x_k. Следва обаче да отбележим, че при няколко възможности за получаване на $l(k)$ следва да се избира тази, при която сумата от елементите е максимална. Това не влияе на $l(k)$, но е от съществено значение за $s(k)$, както и за намирането на една конкретна редица. За подзадача (2) е достатъчно да обърнем в обратен ред началната редица X и да решим отново подзадача (1).

Т. е. всяка максимална подредица `len[x]` може да се получи от някоя предходна чрез добавяне на текущия елемент `L[x]`. Реализацията е функцията `findIncSequence()`. За подзадача (2) е достатъчно да обърнем в обратен ред началната редица `L[]` и да решим отново подзадача (1).

Подзадача (3) решаваме, като проверяваме всичките n възможности. Пробваме да сложим за връх всеки един елемент от x_1 до x_n и избираме този елемент t, при който дължината на нарастващата подредица на елементите от x_1 до x_t плюс дължината на намаляващата подредица на елементите от x_t до x_n е максимално число. Обединението на нарастващата и намаляващата подредици е решението на задачата. За извеждане на една конкретна оптимална редица от дървета се използва информацията, съхранявана в полето `back`.

```
#include <stdio.h>
#include <string.h>
#define MAX 100

struct ST {
  unsigned len; /* Дължина на макс. ненамаляваща подредица с край i */
  unsigned back; /* Индекс на предишния елемент в макс. редица */
  unsigned long sum; /* Сума на елементите на максималната редица */
} max1[MAX], max2[MAX];
unsigned x2[MAX], rez[MAX];
unsigned top, bestLen, bestSum;

const unsigned n = 9;  /* Брой крайпътни дървета */
const unsigned x[MAX] = {0,10,20,15,40,5,4,300,2,1}; /* Височини */

/* Търси нарастваща редица */
void findIncSequence(struct ST max[], const unsigned x[])
{ unsigned i, j;
  /* Основен цикъл */
  for (i = 1; i <= n; i++)
    for (j = max[i].len = max[i].sum = 0; j < i; j++)
      if (x[j] <= x[i])
        if ((max[j].len + 1 > max[i].len)
            || ((max[j].len + 1 == max[i].len)
             && (max[j].sum + x[i] > max[i].sum)) )
        { max[i].back = j;
          max[i].len = max[j].len + 1;
          max[i].sum = max[j].sum + x[i];
        }
}

/* Построява обърнато копие на редицата */
void reverse(unsigned x2[], const unsigned x[])
{ unsigned i;
  for (i = 1; i <= n; i++)
    x2[i] = x[n-i+1];
}

/* Намира търсената редица */
void solve(void) {
  unsigned i;
  /* стъпка (1) */
```

```
  findIncSequence(max1, x);
  /* стъпка (2) */
  reverse(x2,x);
  findIncSequence(max2, x2);
  /* стъпка (3) */
  for (bestLen = bestSum = 0, i = 1; i <= n; i++) {
    if (((max1[i].len + max2[n-i+1].len > bestLen))
        || ((max1[i].len + max2[n-i+1].len == bestLen)
            && (max1[i].sum + max2[n-i+1].sum > bestSum))) {
      bestLen = max1[i].len + max2[n-i+1].len;
      bestSum = max1[i].sum + max2[n-i+1].sum - x[i];
      top = i;
    }
  }
}

/* Построява търсената редица */
void buildSequence(void)
{ unsigned t, len, l;

  /* Построяване на нарастващата част на редицата */
  for (l = max1[t = top].len, len = 0; t != 0; t = max1[t].back)
    rez[l-len++] = x[t];

  /* Построяване на намаляващата част на редицата */
  for (t = max2[n-top+1].back; t != 0; t = max2[t].back)
    rez[++len] = x2[t];
}

/* Извежда резултата на екрана */
void print(void)
{ unsigned i;
  printf("\nМаксимален брой дървета, които могат да се запазят: %u\n",
         bestLen-1);
  printf("\nСредна височина на редицата: %0.2f\n",
         (float) bestSum / (bestLen-1));
  for (i = 1; i < bestLen; i++) printf("%u ", rez[i]);
  printf("\n");
}

int main(void) {
  solve();
  buildSequence();
  print();
  return 0;
}
```
🖫 trees.c

Резултат от изпълнението на програмата:

```
Максимален брой дървета, които могат да се запазят: 7
Средна височина на редицата: 11.71
10 20 40 5 4 2 1
```

Задачи за упражнение:

1. Каква е сложността на предложената реализация?
2. Да се предложи целева функция $l(k)$, която явно да изисква максимално $s(k)$.

3. Да се реши задачата при условие, че имаме три сегмента: растящ, намаляващ и отново растящ (в оригиналната имахме само 2: растящ и намаляващ).

8.4.10. Разрязване на материали

Дадено е правоъгълно парче материал с размери $X \times Y$ и n изделия, които могат да се изготвят от този материал, като всяко изделие е правоъгълник със зададени размери $a_i \times b_i$ и продажна цена $c(a_i,b_i)$, $i = 1, 2, ..., n$. Разполагаме с машина, която може да разрязва произволно правоъгълно парче на две или хоризонтално, или вертикално. Да се определи какви изделия да се направят от даденото парче материал така, че общата продажна цена да бъде максимална.

Нека с $F(a,b)$ да означим максималния доход, който може да се получи при оптимално разрязване на парчето (a,b). В сила са формулите:

$$F(a,b) = \begin{cases} c(a_1,b_1) & a = a_1, b = b_1 \\ c(a_1,b_1) & a = b_1, b = a_1 \\ \max\left\{ \max\limits_{i=1}^{a/2}\left(F(a-i,b)+F(i,b)\right), \max\limits_{j=1}^{b/2}\left(F(a,b-j)+F(a,j)\right) \right\} & \text{иначе} \end{cases}$$

Стойността на функцията F ще пресмятаме рекурсивно, като ще запазваме вече намерените стойности в масива `F[]`. Първоначално ще инициализираме `F[i][j] = ∞` и след това за всеки правоъгълник (a,b) ще положим `F[a][b] = F[b][a] =` $c(a,b)$. Ако `F[a][b] = ∞`, ще го изчисляваме рекурсивно по формулата, а иначе ще го вземаме от таблицата. За да възстановим после изделията, които сме избрали като оптимално решение, ще запомняме всеки избор — с положително число за номер на хоризонтал, по който режем, и с отрицателно число — за номер на вертикал, ако максималната цена от текущото парче се получава при вертикално разрязване. Сложността на алгоритъма е от порядъка на $\Theta(n^4)$.

```
#include <stdio.h>
#define MAXSIZE 100
#define NOT_CALCULATED (unsigned)(-1)

struct {  /* Целева функция */
  unsigned value;
  int action;
} F[MAXSIZE][MAXSIZE];

unsigned sizeX, sizeY; /* Размери на парчето */

/* Инициализираща функция */
void init(void)
{ unsigned x, y;
  /* Входни данни */
  sizeX = 13;
  sizeY = 9;
  /* Инициализиране */
  for (x = 1; x <= sizeX; x++)
    for (y = 1; y <= sizeY; y++) {
      F[x][y].value = NOT_CALCULATED;
      F[x][y].action = 0;
    }
  /* Входни данни */
  F[11][1].value = 28; F[5][3].value = 31;
  F[1][2].value = 4;  F[2][1].value = 2;
  F[3][1].value = 7;  F[10][1].value = 23;
```

```
  F[3][2].value = 14;  F[3][3].value = 17;
  F[5][4].value = 41;  F[5][7].value = 96;
}

/* Намира максималния доход от парчето (x,y) */
unsigned solve(unsigned x, unsigned y)
{ int bestAction;
  unsigned i, bestSol, x2 = x / 2, y2 = y / 2;
  if (NOT_CALCULATED != F[x][y].value)
    return F[x][y].value; /* Вече е пресмятана */
  bestSol = 0;
  if (x > 1) {
    /* Срязваме го хоризонтално и викаме рекурсия за двете части */
    for (i = 1; i <= x2; i++)
      if (solve(i, y) + solve(x - i, y) > bestSol) {
        bestSol = solve(i, y) + solve(x - i, y);
        bestAction = i;
      }
  }
  if (y > 1) {
    /* Срязваме го вертикално и викаме рекурсия за двете части */
    for (i = 1; i <= y2; i++)
      if (solve(x, i) + solve(x, y - i) > bestSol) {
        bestSol = solve(x, i) + solve(x, y - i);
        bestAction = -(int)i;
      }
  }
  F[x][y].value = bestSol;
  F[x][y].action = bestAction;
  return bestSol;
}

/* Извежда резултата на екрана */
void printSolution(int x, int y)
{
  if (x > 0 && y > 0 && F[x][y].value > 0) {
    if (F[x][y].action > 0) {
      printSolution(F[x][y].action, y);
      printSolution(x - F[x][y].action, y);
    }
    else if (F[x][y].action < 0) {
      printSolution(x, -F[x][y].action);
      printSolution(x, y - (-F[x][y].action));
    }
    else
      printf("(%2u,%2u) --> %2u  ", x, y, F[x][y].value);
  }
}

int main(void) {
  init();
  printf("\nМаксимална цена %u", solve(sizeX, sizeY));
  printf("\nРазмери (X,Y)-->Цена\n");
  printSolution(sizeX, sizeY);
  return 0;
}
```
🖫 cuts.c

Резултат от изпълнението на програмата:

```
Максимална цена 305
Размери (X,Y)-->Цена
( 1, 2) -->  4  ( 1, 2) -->  4  (11, 1) --> 28  (11, 1) --> 28  ( 3, 1) -->  7
( 3, 1) -->  7  ( 3, 1) -->  7  ( 3, 2) --> 14  ( 3, 2) --> 14  ( 5, 7) --> 96
( 5, 7) --> 96
```

Забележка: Внимателният читател вероятно е забелязал един пропуск в разсъжденията по-горе — не се предвижда възможност за допълнително разрязване на съществуващи размери, макар че това би могло да доведе до по-малка цена.

Задачи за упражнение:

1. Да се обоснове предложената формула за пресмятане стойностите на целевата функция.
2. Да се модифицира предложената реализация така, че да позволява допълнително разрязване на материали с размери измежду дадените.
3. Как могат да се намалят пресмятанията в предходната задача, ако се налага разрязване на множество *различни* парчета материал, но при *фиксирани* размери на изделията?

8.4.11. Зациклен израз

Дадени за n цели числа $\{x_1, x_2, ..., x_n\}$, подредени във формата на пръстен, като между всеки две съседни е поставена аритметична операция $\{\oplus_1, \oplus_2, ..., \oplus_n\}$. Така, числата образуват аритметичен израз без начало и край. Всички операции имат еднакъв приоритет. Изчисляването на израза става, като се вземат две съседни числа и се заместят с резултата от операцията между тях, като това се повтаря, докато остане само едно число, което е стойността на израза. При различни начални позиции, т. е. различни места, където се къса изразът, и различен ред на изчисляване се получават различни резултати. За фиксиране на определен ред за изчисляване се използват скоби. В задачата се търси максималната и минималната стойности на израза и как те се получават, зададени като числов израз, съставен от последователност от n числа, свързани помежду си със съответни знаци за аритметични операции, като приоритетът се изразява чрез скоби. Входните данни са n-те числа и съответно n-те знака за аритметични операции между тях. Търсят се най-малкият и най-големият резултат, както и изразите, при пресмятането на които се получават. Разрешените аритметични операции са събиране, изваждане и умножение.

Ще използваме динамично оптимиране. Да означим с $F_{min}(k,l)$ минималната стойност, която може да се получи при изчисляване на част от израза с дължина l, започваща от елемента с индекс k, а с $F_{max}(k,l)$ — максималната стойност. Тогава за $F_{min}(k,l)$ са валидни граничните условия:

$F_{min}(k,l) = x_k$, при $l = 1$

При $l > 1$ стойността на $F_{min}(k,l)$ се пресмята като най-малката измежду ($i = 1, 2, ..., l–1$):

$$F_{min}(k,i) \oplus_{k+i-1} F_{min}(k+i,l-i)$$
$$F_{min}(k,i) \oplus_{k+i-1} F_{max}(k+i,l-i)$$
$$F_{max}(k,i) \oplus_{k+i-1} F_{min}(k+i,l-i)$$
$$F_{max}(k,i) \oplus_{k+i-1} F_{max}(k+i,l-i)$$

По същия начин изчисляваме и $F_{max}(k,l)$:

$$F_{max}(k,l) = x_k \text{ при } l = 1$$

При $l > 1$ стойността на $F_{max}(k,l)$ се пресмята като най-голямата измежду ($i = 1, 2, ..., l–1$):

$$F_{min}(k,i) \oplus_{k+i-1} F_{min}(k+i,l-i)$$
$$F_{min}(k,i) \oplus_{k+i-1} F_{max}(k+i,l-i)$$
$$F_{max}(k,i) \oplus_{k+i-1} F_{min}(k+i,l-i)$$
$$F_{max}(k,i) \oplus_{k+i-1} F_{max}(k+i,l-i), \text{ за } i = 1, 2, ..., l-1 \text{ при } l > 1$$

Приема се, че ако $k > n$, то във формулите по-горе сме положили $k = k - n$, което изразява цикличността на пръстена. Търсените максимална и минимална стойности се дават от $F_{min}(k,n)$ и $F_{max}(k,n)$ съответно, за някоя стойност на $k = 1, 2, ..., n$.

Смисълът на тези рекурентни формули се състои в следното: Разделя се частта от израза (k,l) на две части — (k,i) и $(k+i,l-i)$, които ще наричаме съответно *лява* и *дясна*. Търсеният максимум за израза (k,l) се получава от максимума или минимума на левия израз и максимума или минимума на десния израз, върху които се прилага аритметичната операцията между тях. За да получим максимална сума, трябва да съберем максимални числа. За да получим максимално произведение трябва да умножим максимални по модул числа, т. е. или максимални положителни числа или минимални числа — когато и двете са отрицателни. При изваждане, за да получим максимален резултат, трябва да извадим от максимално число минимално число. С цел опростяване на условията на пресмятане, ще изпробваме *всички* варианти за прилагане на операцията — мин|мин, мин|макс, макс|мин и макс|макс. Предлагаме на читателя да модифицира по подходящ начин програмата по-долу, така че да се взема предвид вида на операцията, при което да се избегнат безперспективните случаи.

Как да изчислим ефективно стойностите на описаните рекурентни функции? Ще работим рекурсивно отгоре-надолу. При попълването на таблицата се съхраняват стойностите на F_{min} и F_{max}, както и за коя стойност на i се получават. Това ни позволява по-късно да намерим и самия израз, от който се получава минимума и максимума по добре познатия ни начин. Естествено, отново се взема предвид, че стойностите $k > n$ трябва да се считат за стойности $k-n$ от съображения за цикличност на пръстена. Сложността на алгоритъма е от порядъка на $\Theta(n^3)$, защото се изчисляват най-много n^2 стойности на функциите F_{min} и F_{max}, а за всяка такава стойност са нужни най-много $\Theta(n)$ изчисления.

```
#include <stdio.h>
#define MAXN 50
#define INFINITY (int)((1 << (sizeof(int)*8 - 1)) - 1)
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

struct {
  long min, max;
  unsigned lenMin, lenMax;
} F[MAXN][MAXN];  /* Целеви функции Fmin() и Fmax() */
const unsigned n = 7;      /* Брой числа */
const long x[MAXN] = { 0, 9, -3, 8, 7, -8, 0, 7 }; /* Числа (без 0) */
char sign[MAXN] = { ' ', '+', '*', '*', '-', '+', '*', '-' }; /* Знаци */

/* Извършва операцията */
long oper(long v1, char sign, long v2)
{ switch (sign) {
    case '+': return v1 + v2;
    case '-': return v1 - v2;
    case '*': return v1 * v2;
  }
  return 0;
}

/* Пресмята стойностите на целевата функция */
void calculate(unsigned beg, unsigned len)
{ unsigned i, beg2;
  long val1, val2, val3, val4, minValue, maxValue;
  if (beg > n)
    beg -= n;
  if (F[beg][len].min != INFINITY) /* Стойността вече е била сметната */
```

```
    return;
  if (1 == len) {
    F[beg][len].min = F[beg][len].max = x[beg];
    F[beg][len].lenMin = F[beg][len].lenMax = 0;
    return;
  }

  /* Стойността трябва да се пресметне */
  F[beg][len].min = INFINITY;
  F[beg][len].max = -INFINITY;
  for (i = 1; i < len; i++) {
    /* Пресмятане на всички стойности F[beg][i] и F[beg+i][len-i] */
    calculate(beg, i);
    if (beg + i > n)
          beg2 = beg + i - n;
      else
         beg2 = beg + i;
    calculate(beg2, len - i);
    val1 = oper(F[beg][i].min, sign[beg2 - 1], F[beg2][len - i].min);
    val2 = oper(F[beg][i].min, sign[beg2 - 1], F[beg2][len - i].max);
    val3 = oper(F[beg][i].max, sign[beg2 - 1], F[beg2][len - i].min);
    val4 = oper(F[beg][i].max, sign[beg2 - 1], F[beg2][len - i].max);
    /* Актуализиране на минималната стойност на F[beg][len] */
    minValue = MIN(val1, MIN(val2, MIN(val3, val4)));
    if (minValue < F[beg][len].min) {
      F[beg][len].min = minValue;
      F[beg][len].lenMin = i;
    }
    /* Актуализиране на максималната стойност на F[beg][len] */
    maxValue = MAX(val1, MAX(val2, MAX(val3, val4)));
    if (maxValue > F[beg][len].max) {
      F[beg][len].max = maxValue;
      F[beg][len].lenMax = i;
    }
  }
}

/* Извежда израз, за който се получава min/max */
void printMinMax(unsigned beg, unsigned len, char printMin)
{ unsigned i, beg2;

  if (beg > n)
    beg -= n;
  if (1 == len)
    printf("%ld", x[beg]);
  else {
    if (len < n) printf("(");
    i = printMin ? F[beg][len].lenMin : F[beg][len].lenMax;
      if ((beg2 = beg + i) > n)
    beg2 -= n;
    printMinMax(beg, i, printMin); /* Рекурсия за лявата част */
    printf("%c", sign[beg2 - 1]); /* Извеждане на операцията */
    printMinMax(beg2, len - i, printMin); /* Рекурсия за дясната част */
    if (len < n)
      printf(")");
  }
}

/* Намира максимума и минимума, както и как се получават */
```

```
void solve(void)
{ unsigned i, j;
  /* Инициализиране */
  sign[0] = sign[n];
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
      F[i][j].min = INFINITY;
  /* Пресмятане на стойностите на целевата функция */
  for (i = 1; i <= n; i++)
    calculate(i, n);
}

/* Търси и извежда максимума и минимума */
void print(void)
{ unsigned i, minIndex, maxIndex;
  for (minIndex = 1, i = 2; i <= n; i++)
    if (F[i][n].min < F[minIndex][n].min)
      minIndex = i;
  for (maxIndex = 1, i = 2; i <= n; i++)
    if (F[i][n].max > F[maxIndex][n].max)
      maxIndex = i;
  printf("\nМинимална стойност: %d\n", F[minIndex][n].min);
  printMinMax(minIndex, n, 1);
  printf("\n\nМаксимална стойност: %d\n", F[maxIndex][n].max);
  printMinMax(maxIndex, n, 0);
}

int main(void) {
  solve();
  print();
  return 0;
}
```

🖫 ring.c

Резултат от изпълнението на програмата:

```
Минимална стойност: -3808
((((-8+0)*7)-9)+-3)*(8*7)

Максимална стойност: 5040
(9+-3)*(8*((7-(-8+0))*7))
```

Задачи за упражнение:

1. Да се модифицира програмата така, че да отчита вида на операцията и да изследва само някои от комбинациите: мин|мин, мин|макс, макс|мин и макс|макс.
2. Може ли да се използва същият алгоритъм в случай на реални числа?
3. В предложеното решение се използват две целеви функции F_{min} и F_{max}. Възможно ли е да се реши задачата с използване на *единствена* целева функция?
4. Да се състави програма, която по дадено цяло число x и зациклен израз c проверява дали c може да получи стойност x.

8.4.12. Домино-редица

Ще разгледаме една интересна задача, при чието решение се използват 10 взаимно рекурентни целеви функции.

Дефиниция 8.6. *Домино-редица* ще наричаме редица от естествени числа, за която най-старшата цифра на i-тия член съвпада с най-младшата на (i–1)-ия. ($i \geq 2$)

Задача: По зададена крайна редица $X = \{x_1, x_2, ..., x_n\}$ от естествени числа да се намери нейна максимална по дължина домино-подредица.

Решение: Ясно е, че съществува поне една домино-подредица на X. Наистина, тъй като за първия член на домино-редицата не се налагат никакви ограничителни условия, то всяка подредица, съдържаща единствен член на X, ще бъде нейна домино-подредица.

Освен това броят на домино-подредиците на X не надвишава броя на всички подредици на X. Броят на последните е 2^n и следователно е краен. Тогава и броят на домино-подредиците на X ще бъде краен и следователно ще съществува *максимална* по дължина домино-подредица.

След като се уверихме, че наистина съществува подредица с исканите свойства, достигаме до същинския проблем — как да я намерим? Разбира се, бихме могли да генерираме всички домино-подредици на X и да изберем най-дългата. Последното обаче ни води към алгоритъм с експоненциална сложност.

Да опитаме с динамично оптимиране. Нека с X_k означим редицата $\{x_k, x_{k+1},..., x_n\}$, а с l_k и r_k — съответно първата и последната десетични цифри на елемента x_k, $1 \leq k \leq n$. Въвеждаме целевите функции $F_i(k)$: дължината на най-дългата домино-подредица на X_k, първият член на която има най-старша цифра i ($0 \leq i \leq 9$). Дължината на най-дългата домино-подредица на X ще се дава от най-голямото измежду числата $F_0(1)$, $F_1(1)$, ..., $F_9(1)$.

За целевите функции $F_i(k)$ са в сила следните дефиниции:

$$F_i(k) = \begin{cases} 0 & k > n \\ F_i(k+1) & l_k \neq i \\ \max\left(F_i(k+1), 1 + F_{r_k}(k+1)\right) & l_k = i \end{cases}$$

Разглеждаме целевата функция $F_i(k)$ и елемента x_k. Ако първата цифра l_k на x_k е различна от i, x_k не може да влияе на стойността на $F_i(k)$. (*Защо?*) Ако $i = l_k$, са налице две възможности — да игнорираме x_k или да го вмъкнем като първи елемент на максималната домино-подредица с първи елемент, започващ с последната цифра на x_k — r_k. Дължината $F_i(k)$ на най-дългата домино-подредица, започваща с първи елемент с начална цифра l_k, ще се актуализира, само ако в резултат на вмъкването се получи по-дълга редица от текущата (припомняме, че разглеждаме случая $i = l_k$).

Внимателният читател сигурно вече е забелязал, че горният алгоритъм се интересува единствено от дължините $F_i(k)$ на редиците, но не и от конкретните им членове. Така, вместо паметта от порядъка на $10n$, необходима за съхранението на редиците, ще ни бъде достатъчна памет за 10 променливи (константа) за запомняне на съответните дължини. Това би ни било напълно достатъчно, ако не се интересуваме от *конкретна* оптимална домино-редица, а само от *дължината* ѝ. Но дори и ако желаем да намерим конкретна редица, пак би ни била достатъчна памет от порядъка на (n + 20), а не $10n$. Тъй като, съгласно алгоритъма, всеки елемент a_k при постъпването си може да се включи като първи в точно една максимална домино-подредица. Тогава ще ни бъде достатъчно да помним само елемента, стоящ непосредствено след x_k. Възниква необходимост от съхранение на първите елементи на всяка от десетте домино-подредици, но това изисква константна допълнителна памет, независеща от n.

```
#include <stdio.h>
#define MAX 100
#define BASE 10 /* Основа на бройната система */

unsigned succ[MAX];  /* Наследници за всеки връх */
unsigned F[BASE];  /* F[i]: текуща макс. дължина на подредица за i */
unsigned ind[BASE];  /* ind[i]: индекс на началото на редицата за i */
```

```
const unsigned n = 17;  /* Брой елементи в редицата */
const unsigned x[MAX] = {0, 72, 121, 1445, 178, 123, 3462, 762, 33434,
          444, 472, 4, 272, 4657, 7243, 7326, 3432, 3465}; /* Редица */

/* Намира максимална домино-редица */
void solve(void)
{ unsigned i, l, r;

  for (i = 0; i < BASE; i++)
    F[i] = ind[i] = 0;

  /* Намиране дължините на редиците, започващи с цифрите от 0 до 9 */
  for (i = n; i > 0; i--) {
    /* Определяне на най-старшата и най-младшата цифри на числото */
    r = x[i] % BASE;
    l = x[i];
    while (l > BASE)
      l /= BASE;
    /* Актуализиране на редицата, започваща със старшата цифра */
    if (F[r] >= F[l]) {
      F[l] = F[r] + 1;
      succ[i] = ind[r];
      ind[l] = i;
    }
  }
}

void print(void)
{ unsigned i, bestInd;
  /* Определяне на най-дългата редица */
  bestInd = 0;
  for (i = 1; i < BASE; i++) /* Никое число не започва с 0 */
    if (F[i] > F[bestInd])
      bestInd = i;

  /* Извеждане на редицата на екрана */
  printf("\nДължина на максималната домино-подредица: %u", F[bestInd]);
  i = ind[bestInd];
  do {
    printf("%u ", x[i]);
    i = succ[i];
  } while (i > 0);
}

int main(void) {
  solve();
  print();
  return 0;
}
```
🖫 domino.c

Резултат от изпълнението на програмата:

```
Дължина на максималната домино-подредица: 8
121 123 33434 444 4 4657 7243 3465
```

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.
2. Възможно ли е да се реши задачата с използване на единствена целева функция?
3. Да се намерят всички решения. Каква минимална памет е необходима?

8.4.13. Трионообразна редица

Задача: Една редица от цели числа се нарича *трионообразна*, ако е изпълнено едно от следните две условия:

1) Всеки елемент с четен номер е по-малък от съседните си елементи, а с нечетен номер — по-голям.
2) Всеки елемент с нечетен номер е по-малък от съседните си елементи, а с четен — по-голям.

Да се напише програма, която по зададена редица от цели числа намира най-дългата ѝ трионообразна подредица. Например за редицата от 5 числа — 8, 3, 5, 7, 0 едно възможно решение е: 8, 3, 5, 0.

Решение: Лесно се вижда, че задачата може да се реши с динамично оптимиране. Не е толкова очевидно обаче как да съставим целевата функция. Ако се досетим да използваме две взаимно рекурентни целеви функции, нещата се опростяват значително.

Дефинираме две целеви функции $F_{\min}$ и $F_{\max}$, всяка от които има аргумент k — индекс на елемент в редицата. $F_{\min}$ ще съдържа дължината на най-дългата трионообразна подредица на редицата $x_1, x_2, \ldots, x_k$, завършваща в елемента x_k *със спад*, а $F_{\max}$ — дължината на най-дългата трионообразна подредица на редицата $x_1, x_2, \ldots, x_k$, завършваща в елемента x_k *със скок*. Оттук лесно получаваме рекурентните формули:

$$F_{\min}(k) = \begin{cases} 1 + \max\limits_{i<k, x_i > x_k} \left(F_{\max}(i)\right) & \exists i : 1 \le i < k \;\&\; x_i > x_k \\ 1 & \text{иначе} \end{cases}$$

$$F_{\max}(k) = \begin{cases} 1 + \max\limits_{i<k, x_i < x_k} \left(F_{\min}(i)\right) & \exists i : 1 \le i < k \;\&\; x_i < x_k \\ 1 & \text{иначе} \end{cases}$$

Лесно се вижда, че горните две функции могат да бъдат пресметнати едновременно и последователно за $k = 1, 2, \ldots, n$. Така, за всеки елемент на редицата получаваме дължините на двете най-дълги трионообразни редици от всеки тип съгласно *дефиниция 8.6.* по-горе, завършващи в този елемент. За да намерим най-дългата трионообразна редица трябва да намерим най-голямата стойност на $F_{\min}(k)$ и $F_{\max}(k)$, за $k = 1, 2, \ldots, n$. По-голямата от тези две стойности ни дава дължината на най-дългата трионообразна подредица на изходната редица. За да възстановим една такава конкретна редица, бихме могли за всяко k да запазваме индекса i, за който се е получила съответната максимална стойност на съответната целева функция ($1 \le i, k \le n$). Това е и подходът, избран в реализацията по-долу. Забележете, че в процеса на възстановяване на редицата трябва на всяка стъпка алтернативно да сменяме целевата функция.

Алтернативен начин за получаване на конкретна редица е да се търси вляво от крайния елемент такъв, който удовлетворява съответното горно уравнение, т. е. стойността на чиято съответна целева функция е с 1 по-малка от текущата. Процесът продължава до достигане на елемент, чиято съответна целева функция (тя се сменя на всяка стъпка) има стойност 1.

Реализацията по-долу допуска трионообразната редица да съдържа или да не съдържа два съседни равни елемента, т. е. дали неравенството да бъде строго/нестрого. За целта е дефиниран макрос `OPERATION`, позволяващ лесно превключване между двете възможности.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX       1000
```

```
#define NO_IND    (unsigned) (-1)
#define OPERATION <

const int x[MAX] = {0,8,3,5,7,0,8,9,10,20,20,20,12,19,11}; /* без 0 */
const unsigned n = 14; /* Брой */

/* Целева функция: максимална дължина на редица, завършваща със спад */
unsigned Fmin[MAX];
/* Предишен индекс от редицата на целевата функция Fmin */
unsigned Fmin_back[MAX];

/* Целева функция: максимална дължина на редица, завършваща със скок */
unsigned Fmax[MAX];

/* Предишен индекс от редицата на целевата функция Fmax */
unsigned Fmax_back[MAX];

void calculateFMinMax(void)
{ unsigned ind, ind2;
  /* Инициализиране */
  Fmin[0] = Fmax[0] = 1;
  Fmin_back[0] = Fmax_back[0] = NO_IND;

  /* Последователно пресмятане на двете целеви функции */
  for (ind = 1; ind < n; ind++) {
    Fmax_back[ind] = Fmin_back[ind] = NO_IND;
    Fmin[ind] = Fmax[ind] = 0;
    for (ind2 = 0; ind2 < ind; ind2++) {
      /* Опит за разширяване на намаляваща редица с нарастващ елемент */
      if (x[ind2] OPERATION x[ind] && Fmin[ind2] > Fmax[ind]) {
        Fmax[ind] = Fmin[ind2];
        Fmax_back[ind] = ind2;
      }
      /* Опит за разширяване на нарастваща редица с намаляващ елемент */
      if (x[ind] OPERATION x[ind2] && Fmax[ind2] > Fmin[ind]) {
        Fmin[ind] = Fmax[ind2];
        Fmin_back[ind] = ind2;
      }
    }
    /* Увеличаване с 1 заради текущия елемент */
    Fmin[ind]++;
    Fmax[ind]++;
  }
}

void markSolutionElements(unsigned *f1, unsigned *f2, unsigned *fInd1,
                          unsigned *fInd2, unsigned indF)
{ if (NO_IND == fInd1[indF])
    return;
  f1[indF] = f2[indF] = NO_IND;
  markSolutionElements(f2, f1, fInd2, fInd1, fInd1[indF]);
}

void findSolution(void)
{ unsigned ind, bestFminInd, bestFmaxInd;
  /* Намиране (края) на най-дългата редица */
  bestFminInd = bestFmaxInd = 0;
  for (ind = 1; ind < n; ind++) {
    if (Fmin[bestFminInd] < Fmin[ind])
```

```
        bestFminInd = ind;
      if (Fmax[bestFmaxInd] < Fmax[ind])
        bestFmaxInd = ind;
    }
    /* Маркиране на елементите й */
    if (Fmin[bestFminInd] > Fmax[bestFmaxInd])
      markSolutionElements(Fmin, Fmax, Fmin_back, Fmax_back, bestFminInd);
    else
      markSolutionElements(Fmax, Fmin, Fmax_back, Fmin_back, bestFmaxInd);
    /* Извеждане на решението на екрана */
    for (ind = 0; ind < n; ind++)
      if (NO_IND == Fmin[ind])
        printf("%d ", x[ind]);
    printf("\n");
  }
  int main(void) {
    calculateFMinMax();
    findSolution();
    return 0;
  }
```

🖫 saw.c

Резултат от изпълнението на програмата:

```
8 3 5 0 20 12 19 11
```

Задачи за упражнение:

1. Да се определи сложността на предложената реализация.
2. Възможно ли е да се реши задачата с използване на единствена целева функция?
3. Да се сравни текущата задача с тази за крайпътните дървета от *8.4.9*.
4. Да се реализира вторият предложен начин за намиране на една конкретна редица.
5. Да се намерят всички решения. Ще работи ли всеки от двата алгоритъма, изложени по-горе? При каква памет?

8.5. Въпроси и задачи

8.5.1. Задачи от текста

Задача 8.1.

Задължително ли е подзадачите да се припокриват, за да може да се прилага динамично оптимиране (*виж 8.1.*)?

Задача 8.2.

Да се сравнят търсенето с връщане, “разделяй и владей” и динамичното оптимиране (*виж 8.1.*).

Задача 8.3.

Само за оптимизационни задачи ли може да се прилага динамично оптимиране (*виж 8.1.*)?

Задача 8.4.

Да се покаже, че предложените в *8.2.1.* формули и програмни реализации водят до правилно решение на задачата за раницата.

Задача 8.5.

Да се преформулира първата целева функция от *8.2.1.* така, че да не допуска повторение на елементи (първа предложена реализация).

Задача 8.6.

Необходимо ли е множеството `set[]` в първата реализация от *8.2.1.*? Възможно ли е да се гарантира неповтаряне на предмети по друг начин?

Задача 8.7.

Работят ли правилно предложените в *8.2.1.* реализации в случай, когато задачата няма решение? Да се нанесат съответни промени в програмите, в които това е нужно.

Задача 8.8.

Нека оптималното решение се състои от k предмета с еднакви тегла и стойности. Колко пъти ще се генерира това решение при всеки от предложените в *8.2.1.* алгоритми?

Задача 8.9.

Да се реши задачата за раницата (*виж 8.2.1.*) за $c_i = 1$, $m_i = \sqrt{i}$ и $M = N/2$.

Задача 8.10.

Да се реализира функция, която да намира едно възможно разделяне на подаръците между братята Алан и Боб, без да се използва допълнителна информация за последния добавен предмет `last[k]` (*виж 8.2.2.*).

Задача 8.11.

Може ли да се твърди, че предложеният в *8.2.2.* алгоритъм е полиномиален?

Задача 8.12.

Да се напише функция, която възстановява един възможен оптимален ред на умножение на матриците, без да използва изрично запазени индекси, само въз основа на стойностите на целевата функция (*виж 8.2.3.*).

Задача 8.13.

Да се напише функция, която намира *всички* възможни оптимални решения на задачата за умножение на матрици (*виж 8.2.3.*).

Задача 8.14.

Да се реализира вариант на програмата от *8.2.3.*, който използва *линеаризирана* матрица, т.е. едномерен масив, с $n(n-1)/2$ елемента.

Задача 8.15.

Възможно ли е задачата за умножението на матрици (*виж 8.2.3.*) да се реши с памет, линейна по n?

Задача 8.16.

Може ли да се твърди, че задачата за матриците (*виж 8.2.3.*) се решава полиномиално?

Задача 8.17.

Каква е сложността на алгоритъма от *8.2.4.*?

Задача 8.18.

Да се докажат (1), (2) и (3) от *8.2.4.*

Задача 8.19.

Да се докаже еквивалентността на различните дефиниции на числата на Каталан (*виж 8.2.4.*).

Задача 8.20.

Защо на *фигура 8.2.4а.* на страната $\overline{A_0A_6}$ не е съпоставена матрица?

Задача 8.21.

Ще работи ли предложеният в *8.2.5.* алгоритъм, ако вместо честоти са зададени *тегла* на думите и е позволено да бъдат:

а) отрицателни
б) рационални
в) реални

Задача 8.22.

Има ли други оптимални решения на задачата за построяване на оптимално двоично дърво за претърсване за дадените конкретни входни данни освен дървото от *фигура 8.2.5.*?

Задача 8.23.

Да се намерят всички оптимални решения на задачата за построяване на оптимално двоично дърво за претърсване за дадените входни данни от *8.2.5.*

Задача 8.24.

Да се сравнят реализациите на решенията на задачата за умножение на матрици (*виж 8.2.3.*), триангулация на многоъгълник (*виж 8.2.4.*) и построяване на оптимално двоично дърво за претърсване (*виж 8.2.5.*).

Задача 8.25.

Да се реализира рекурсивен вариант на алгоритъма за намиране на най-дълга обща подредица (*виж 8.2.6.*).

Задача 8.26.

Да се реализира вариант на алгоритъма от *8.2.6.*, който изисква памет:

а) $2.\min(m,n) + \Theta(1)$
б) $\min(m,n) + \Theta(1)$

Задача 8.27.

Последната функция от *8.2.7.* намира дължината на максималната ненамаляваща редица, но не и *самата* редица. Да се модифицира по подходящ начин така, че да намира една оптимална редица за време $\Theta(n)$.

Задача 8.28.

Да се сравнят трите предложени в *8.2.7.* алгоритъма. Кои от тях се основават на динамично оптимиране?

Задача 8.29.

Да се намерят *всички* максимални ненамаляващи подредици. Кой от предложените в *8.2.7.* алгоритми е най-подходящ за целта? Защо?

Задача 8.30.

Да се състави алгоритъм за решаване на задачата за сравнение на символни низове (*виж 8.2.8.*) за време $\Theta(nm)$ и памет $\Theta(\min(m,n))$.

Задача 8.31.

Да се реализира алгоритъмът на Хиршберг (*виж 8.2.8.*).

Задача 8.32.

Като се използва информацията, събрана от алгоритъма от *8.2.8.*, да се възстанови една възможна оптимална последователност от операции, уеднаквяваща двата низа, но ориентирана към *текущите*, а не към изходните позиции в низовете.

Задача 8.33.

Да се модифицира по подходящ начин програмата от *8.2.8.* така, че да решава задачата за *smart* терминалите .

Задача 8.34.

Да се покаже, че при подходящи тегла задачата от *8.2.8.* се свежда до задачата за намиране на най-дълга обща подредица (*виж 8.2.6.*).

Задача 8.35.

Нужен ли е непременно масивът `b[][]` за възстановяване на едно възможно решение на задачата за разделянето (*виж 8.2.9.*)?

Задача 8.36.

Да се намерят всички решения на задачата за разделянето (*виж 8.2.9.*).

Задача 8.37.

Да се даде пример, когато предложената в началото на *8.2.9.* алгоритъм няма да работи правилно.

Задача 8.38.

Като се използва определението за произведение на матрици да се докаже формула (1) от *8.3.1.*

Задача 8.39.

Да се докаже по индукция формула (2) от *8.3.1.*

Задача 8.40.

Да се реализира итеративен вариант на програмата, реализираща алгоритъма, зададен от формула (1). Да се сравни с рекурсивния вариант по отношение на необходима памет (*виж 8.3.1*).

Задача 8.41.

Да се провери емпирично кой от методите от *8.3.1* е най-бърз на Вашата машина.

Задача 8.42.

Да се реализират необходимите операции с дълги числа и да се сравнят по скорост предложените в *8.3.1* алгоритми при пресмятане на n-тото число на Фибоначи за n: 10, 100, 200, 500, 1000.

Задача 8.43.

Да се реализира *рекурсивен* вариант на програмата за намиране на определен биномен коефициент, основан на динамично оптимиране (*виж 8.3.2.*).

Задача 8.44.

Възможно ли е да се намали сложността по време $\Theta(n.k)$ на програмата за намиране на определен биномен коефициент (*виж 8.3.2.*) така, както беше възможно при числата на Фибоначи (*виж 8.3.1.*)?

Задача 8.45.

Защо в *8.3.3.* сравнението с реалното число 0.0 не е опасно? Кога би било опасно?

Задача 8.46.

Да се модифицира последната функция от *8.3.3.* така, че да бъде необходима памет $\Theta(n)$.

Задача 8.47.

Да се сравни задачата от *8.3.3.* с тази за биномните коефициенти (*виж 1.1.5.*, *8.3.2.*). Възможно ли е и тук да се дефинира истински триъгълник, подобен на този на Паскал?

Задача 8.48.
Да се определи сложността на предложената в *8.3.4.* реализация.

Задача 8.49.
Да се реализира итеративен вариант на програмата от *8.3.4.*

Задача 8.50.
Да се определи сложността на предложената в *8.3.5.* реализация.

Задача 8.51.
Да се реализира итеративен вариант на програмата от *8.3.5.*

Задача 8.52.
Да се сравнят условията и съответните алгоритми на задачите от *8.3.4. и 8.3.5.*

Задача 8.53.
Да се определи сложността на предложената в *8.3.6.* реализация.

Задача 8.54.
Да се реализира рекурсивен вариант на програмата от *8.3.6.* и да се сравни с итеративния.

Задача 8.55.
Позволява ли формулата от *8.3.6.* да се намали на порядък размерът на необходимата памет?

Задача 8.56.
Да се сравнят условията и съответните алгоритми на задачите от *8.3.4.*, *8.3.5.* и *8.3.6.*.

Задача 8.57.
Каква е сложността на предложената в *8.3.7.* реализация?

Задача 8.58.
Да се реализира итеративен вариант на програмата от *8.3.7.* и да се сравни с рекурсивния.

Задача 8.59.
Да се определи сложността на предложената в *8.3.8.* реализация.

Задача 8.60.
Единствен ли е изводът на `aaasbbb` в граматиката на програмата от *8.3.8.*?

Задача 8.61.
Да се напише функция за намиране на един възможен извод като последователност от правила от граматиката (*виж 8.3.8.*). Необходимо ли е да се пази допълнителна информация?

Задача 8.62.
Да се реализира програма за разпознаване на автоматна граматика. Да се сравни с предложената в *8.3.8.* реализация за случая на контекстно свободна граматика.

Задача 8.63.
Възможно ли е прилагане на динамично оптимиране за разпознаване на контекстно зависима граматика (*виж 8.3.8.*)? А на език от общ тип?

Задача 8.64.
Ще може ли да се използва алгоритъмът от *8.3.8.*, ако се търси *най-кратък* извод?

Задача 8.65.
Към кой тип граматика (*виж 8.3.8.*) принадлежи езикът Си? А *XML*? А българският език?

Задача 8.66.
Да се определи сложността на предложената в *8.3.9.* реализация.

Задача 8.67.

Правилно ли е според Вас изречението `NNNNNNNNECINNxqpCDNNNNNNwNNNtNNNNNs`, подадено за проверка на програмата от *8.3.9.*? Защо?

Задача 8.68.

Какъв е типът на граматиката на хедонийския език съгласно йерархията на Чомски (*виж 8.3.8.*, *8.3.9.*)?

Задача 8.69.

Да се модифицира програмата от *8.3.9.* така, че да намира:

а) един извод възможен извод;
б) всички възможни изводи;
в) най-краткия извод

Задача 8.70.

Да се определи сложността на предложената в *8.3.10.* реализация.

Задача 8.71.

Да се реализира итеративен вариант на програмата от *8.3.10.* и да се сравни с рекурсивния.

Задача 8.72.

Да се намерят всички решения на задачата от *8.3.10.*

Задача 8.73.

Да се реализира рекурсивен вариант на програмата от *8.4.1.* и да се сравни с итеративния.

Задача 8.74.

Да се намерят всички решения на задачата от *8.4.1.*

Задача 8.75.

Използва ли се в програмата от *8.4.1.* фактът, че всяка пермутация на отсечките от дадено решение също е решение? Ако да — къде и как? Ако не — как би могло да стане и нужно ли е?

Задача 8.76.

Да се определи сложността на реализацията от *8.4.2.*

Задача 8.77.

Да се модифицира програмата от *8.4.2.* така, че да намира и отпечатва подробно разписание за движението: начало, край, дължина и тарифа за всяка отсечка.

Задача 8.78.

Предложеното в *8.4.2.* решение винаги търси *най-далечната* гара вляво. Съгласни ли сте с това ограничаване на изследваните възможности? Възможно ли е да бъде пропуснато решение? Доколко това ускорява алгоритъма? Може ли да се приложи при друг разгледан алгоритъм, основан на динамично оптимиране?

Задача 8.79.

Да се използва линеаризирана матрица за представяне на триъгълника от *8.4.3.*

Задача 8.80.

Да се намерят *всички* пътища в задачата от *8.4.3.* Каква е сложността в този случай? Зависи ли от броя на пътищата?

Задача 8.81.

Да се реши задачата от *8.4.3.*, като се тръгне от последния ред нагоре към върха на триъгълника.

Задача 8.82.

Необходимо ли е непременно да се пази последната добавена монета за всяка възможна сума, за да се гарантира, че няма да взема втори път една и съща монета (*виж 8.4.4.*)?

Задача 8.83.

Да се предложи нова рекурентна формула за задачата от *8.4.4.*, която да включва изискването да няма повторение на монетите.

Задача 8.84.

Да се сравни задачата от *8.4.4.* със задачата за раницата (*виж 8.2.1.*).

Задача 8.85.

Да се сравнят двата алгоритъма от *8.4.5.*

Задача 8.86.

Да се реализира първият алгоритъм от *8.4.5.*

Задача 8.87.

Да се определи сложността на реализацията от *8.4.6.*

Задача 8.88.

Да се сравни задачата от *8.4.6.* с числата на Фибоначи (*виж 8.3.1.*).

Задача 8.89.

Възможно ли е задачата от *8.4.6.* да се реши с константна памет за целевата функция?

Задача 8.90.

Да определи сложността на предложената в *8.4.7.* реализация.

Задача 8.91.

Да се реализира итеративен вариант на програмата от *8.4.7.*

Задача 8.92.

Да се реши задачата от *8.4.7.* при условие, че не се позволява да остава непродадена стока.

Задача 8.93.

Да се предложи и реализира функция за възстановяване на едно възможно оптимално разписание. За примера от *8.4.8.* една възможност е: R : 1÷5, B : 6÷11, R : 12÷20 и B : 26÷30.

Задача 8.94.

Възможно ли е задачата от *8.4.8.* да се реши само с една целева функция, вместо с две?

Задача 8.95.

Каква е сложността на предложената в *8.4.9.* реализация?

Задача 8.96.

За задачата за крайпътните дървета от *8.4.9.* да се предложи целева функция $l(k)$, която явно да изисква максимално $s(k)$.

Задача 8.97.

Да се реши задачата от *8.4.9.* при условие, че имаме три сегмента: растящ, намаляващ и отново растящ (в оригиналната имахме само 2: растящ и намаляващ).

Задача 8.98.

Да се обоснове предложената в *8.4.10.* формула за пресмятане стойностите на целевата функция.

Задача 8.99.

Да се модифицира предложената в *8.4.10.* реализация така, че да позволява допълнително разрязване на материали с размери измежду дадените.

Задача 8.100.

Как могат да се намалят пресмятанията в предходната задача, ако се налага разрязване на множество *различни* парчета материал, но при *фиксирани* размери на изделията (виж *8.4.10.*)?

Задача 8.101.

Да се модифицира програмата от *8.4.11.* така, че да отчита вида на операцията и да изследва само някои от комбинациите: мин|мин, мин|макс, макс|мин и макс|макс.

Задача 8.102.

Може ли да се използва алгоритъмът от *8.4.11.* в случай на реални числа?

Задача 8.103.

В предложеното в *8.4.11.* решение се използват две целеви функции F_{min} и F_{max}. Възможно ли е да се реши задачата с използване на *единствена* целева функция?

Задача 8.104.

Да се състави програма, която по дадено цяло число x и зациклен израз c проверява дали c може да получи стойност x (*виж 8.4.11.*).

Задача 8.105.

Да се определи сложността на реализацията от *8.4.12.*

Задача 8.106.

Възможно ли е да се реши задачата от *8.4.12.* с използване на единствена целева функция?

Задача 8.107.

Да се намерят всички решения на задачата за домино-редицата (*виж 8.4.12.*). Каква минимална памет е необходима?

Задача 8.108.

Да се определи сложността на реализацията от *8.4.13.*

Задача 8.109.

Възможно ли е да се реши задачата от *8.4.13.* с използване на *единствена* целева функция?

Задача 8.110.

Да се сравни задачата от *8.4.13.* с тази за крайпътните дървета от *8.4.9.*

Задача 8.111.

Да се реализира вторият предложен начин за намиране на една конкретна редица, предложен в *8.4.13.*

Задача 8.112.

Да се намерят всички решения на задачата от *8.4.13.* Ще работи ли всеки от двата алгоритъма, изложени по-горе? При каква памет?

8.5.2. Други задачи

Задача 8.113. *Ясно отпечатване*

Даден е принтер с фиксирани размер на страницата и шрифт. Текстът се състои от поредица от думи с дължини съответно l_1, l_2, ..., l_n букви. Всеки ред, отпечатан от принтера, може да съдържа до M букви. Ако даден ред съдържа думите от i до j включително, то в края на реда ще

остане празно място, което може да се получи по формулата (предполага се, че всеки две думи също са отделени с интервал):

$$M - j + i - \sum_{k=i}^{j} l_k$$

Целта е да се минимизира сумата от кубовете на броя на интервалите за всички редове на текста с изключение на последния. Така например, при k реда, съдържащи $b_1,b_2,...,b_k$ интервала съответно, целта е да се минимизира стойността на израза $b^3_1 + b^3_2 + ... + b^3_{k-1}$. Да се състави алгоритъм за намиране на оптимално разбиване на текста по редове.

Задача 8.114. *Разбиване на низ*

Някои езици за обработване на символни низове позволяват на програмиста да разбие низа на части. Тъй като тази операция е свързана с копиране на стария низ, разбиването на низ на две части с дължина n изисква време от порядъка на n. Да предположим, че програмистът желае да разбие даден низ на повече части. Редът, в който се извършват разбиванията влияе върху общото изразходвано време. Да вземем например низ с дължина 20 символа и да поискаме да го разбием на части след третия, осмия и десетия символ (номерацията започва от 1 и е отляво-надясно). Ако разбиването се извърши отляво-надясно, то първото струва 20 единици време, второто — 17, и третото — 12. Или общо 49 единици време:

```
THI|SISAS|TR|INGOFCHARS
THI SISAS|TR|INGOFCHARS → 20
THI SISAS TR|INGOFCHARS → 17
THI SISAS TR INGOFCHARS → 12
```

Ако обаче се извърши отдясно-наляво, цените на разбиванията са съответно 20, 10 и 8, или общо 38 единици време:

```
THI|SISAS|TR|INGOFCHARS
THI|SISAS|TR INGOFCHARS → 20
THI|SISAS TR INGOFCHARS → 10
THI SISAS TR INGOFCHARS →  8
```

Да се състави алгоритъм, основан на динамичното оптимиране, който по зададен низ и позиции на разделяне, определя най-бързия начин за това.

Да се намерят контрапримери (дължина на низ и позиции на разрязване) за следните алгоритми, претендиращи да решават предходната задача:

- Низът се срязва възможно най-близо до средата, след което процесът се повтаря рекурсивно за двете части.
- Извършват се (най-много) две разрязвания с цел отделяне на най-късите поднизове. Процесът продължава до извършване на всички разрязвания.
- Извършват се (най-много) две разрязвания с цел отделяне на най-дългите поднизове. Процесът продължава до извършване на всички разрязвания.

Задача 8.115. *Валутни курсове*

Да разгледаме задачата за извличане на максимална печалба от разликите във валутните курсове. Възможно е в някакъв интервал от време 1 щатски долар да се търгува за 0,75 британски лири, 1 британска лира — за 2 австралийски франка, а 1 австралийски франк — за 0,70 щатски долара. В такъв случай един умел търговец на валута би могъл, започвайки с 1 щатски долар, да реализира 5% печалба: 0,75.20,7 = 1,05 щатски долара. Да предположим, че са дадени n валути c_1, c_2, ..., c_n, и таблица R с размери $n \times n$ на кръстосаните курсове. $R[i][j]$ показва количеството валута c_j, което може да се закупи с единица валута c_i. Да се състави динамичен алгоритъм, за намиране на максимума на:

$$R[c_1][c_{i_1}].R[c_{i_1}][c_{i_2}]...R[c_{i_{k-1}}][c_{i_k}].R[c_{i_k}][c_1],$$

където $i_r \neq i_s$ за $s \neq r$, $1 \leq i,r \leq n$.

Задача 8.116. *Инвестиране*

Разполагате с 1 лев и желаете да го инвестирате за период от n месеца. В началото на всеки месец можете да избирате измежду следните възможности:

- *Откриване на едномесечен депозит в местната банка.* Така, парите Ви ще престоят 1 месец, като, ако ги депозирате в месеца t, откриването на влог ще изисква такса от $C_S(t)$ лева и след месец ще ви донесе $S(t)$ лева за всеки депозиран лев. Така, ако сте депозирали k лева в месец t, в месец t+1 ще разполагате с $(k–C_S(t)).S(t)$ лева.
- *Закупуване на държавни ценни книжа.* Това ще ангажира парите Ви за следващите 6 месеца. Ако ги закупите в месеца t, ще заплатите такса от $C_B(t)$ и след 6 месеца ще получите $B(t)$ лева за всеки инвестиран лев. Така, ако в месеца t инвестирате k лева, в месеца t+6 ще разполагате с $(k–C_B(t)).B(t)$.
- *Парите — в дюшека.* Въздържате се от инвестиции за един месец и прибирате парите на сигурно под дюшека. Така, ако в месеца t сте разполагали с k лева, в месеца t+1 ще разполагате отново с k лева.

Да предположим, че разполагате с прогнозни стойности на S, B, C_S и C_B за следващите n месеца. Да се състави алгоритъм, основан на динамичното оптимиране, който намира оптимален план за инвестиране на Вашите спестявания за следващите n месеца, със сложност по време $\Theta(n)$.

Задача 8.117. *Фирмено тържество*

След дипломирането си започвате работа в перспективна компания с дървовидна йерархична организация на управление. Първата Ви задача като сътрудник в информационния отдел е да изготвите списък на служителите на фирмата, които да бъдат поканени на новогодишното тържество. За целта са Ви предоставили дървото на йерархията във фирмата, съдържащо имената на всички служители. Шефът на фирмата желае всички присъстващи да се забавляват максимално. Въз основа на наблюдения от предишни празненства за всеки служител е зададена качествена оценка (число от 0 до 11), показваща доколко умее да се забавлява и да създава празнична атмосфера. За съжаление, служителите се притесняват да се отпуснат в компанията на преките си началници (коефициентът става 0). Във връзка с това шефът е решил да максимизира сумата от коефициентите на способност за забавление от *поканените* служители, като изисква за всеки *поканен* служител неговият *пряк* началник да не бъде поканен. Да се намери алгоритъм, основан на динамично оптимиране, за съставяне на търсения списък и да се анализира сложността му.

Задача 8.118. *Фирмено тържество – 2*

Разгледайте още веднъж предложения алгоритъм за предходната задача. Сигурни ли сте, че шефът на фирмата ще бъде поканен на собственото си тържество? Не забравяйте, че той самият се има за голям бохем и едва ли ще оцени по достойнство труда Ви, ако го изпуснете от списъка. Как ще си гарантирате, че той ще бъде поканен?

Задача 8.119. *Опасни кръстовища*

Даден е град с идеално прави улици, образуващи равномерна правоъгълна мрежа $X \times Y$. Искаме да се придвижим от горния ляв ъгъл на правоъгълника до долния десен. За съжаление, движението в града е претоварено и част от кръстовищата трябва да се избягват. Направени са съответни проучвания, резултатите от които са дали възможност за съставяне на таблица `t[i][j]` с размери $X \times Y$. `t[i][j] == 1`, ако кръстовището трябва да се избягва, и `t[i][j] == 0` — иначе.

- Да се даде пример за съдържание на `t[][]`, при което не съществува път от горния ляв до долния десен ъгъл, избягващ опасните кръстовища.
- Да се състави алгоритъм със сложност $\Theta(XY)$, който намира безопасен път, ако такъв съществува.
- Да се състави алгоритъм със сложност $\Theta(XY)$, който намира *най-краткия* безопасен път, ако такъв съществува. Предполагаме, че всички отсечки между две последователни кръстовища имат еднаква дължина.

Задача 8.120. *Опасни кръстовища–2*

Това е разновидност на предходната задача. Отново е даден град с идеално прави улици, образуващи равномерна правоъгълна мрежа $X \times Y$, и искаме да се придвижим от горния ляв ъгъл на правоъгълника до долния му десен ъгъл, като искаме да избягваме кръстовищата, за които `t[i][j] == 1`. Ако всички кръстовища са безопасни, най-краткият път ще има дължина $(X–1) + (Y–1)$ и ще се състои от $X–1$ придвижвания надясно по хоризонтална отсечка и $Y–1$ — по вертикална отсечка. Очевидно в такъв случай ще съществуват множество такива пътища. За съжаление, градът все пак е претоварен. Въпреки това е възможно най-късият път да има дължина $(X–1) + (Y–1)$. Целта да се намери *броят* на всички такива безопасни пътища.

Задача 8.121. *Библиотека*

Дадени са n книги и множество от празни рафтове в библиотека. Целта е книгите да се разположат така, че да заемат минимален брой рафтове. Редът на книгите е строго фиксиран от съществуващата каталогова система и не може да бъде променян. Всяка книга се характеризира с две числа: дебелина t_i, $1 \leq i \leq n$. Дължината на рафтовете в библиотеката е еднаква и е равна на L. Да предположим, че всички рафтове са достатъчно високи, така че всяка книга може да бъде сложена на всеки от рафтовете, т. е. височините им нямат значение. Един алчен алгоритъм (*виж глава 9*) ще попълва всеки рафт дотогава, докато свободното място, което остава, не се окаже недостатъчно за вмъкване на следващата книга. В такъв случай ще преминаваме към следващия рафт. Да се покаже, че този алгоритъм винаги намира оптималното решение и да се оцени сложността му по време и памет. Да се сравни със задачата от *параграф 8.2.9*.

Задача 8.122. *Библиотека–2*

Да се реши предишната задача, но при предположението, че на всяка книга е съпоставено още едно число h_i и височините на рафтовете могат да се коригират така, че да съответстват на височината на най-високата книга в рафта. Така, всяка книга може да бъде поставен на всеки рафт. Целта е да се минимизира *сумата от височините* на рафтовете. Да се покаже, че описаният в предишната задача алчен алгоритъм не винаги намира оптималното решение. Да се намери конкретен пример. Да се реализира алгоритъм, основан на динамично оптимиране, който решава задачата. Да се оцени сложността му по време и памет.

Задача 8.123. *Други примери за динамично оптимиране*

Вярно ли е, че изброените по-долу алгоритми се основават на *динамично оптимиране*? Защо?

- алгоритъм на Форд-Белман (*виж 5.4.2.*)
- алгоритъм на Флойд (*виж 5.4.2.*)
- обобщеният алгоритъм на Флойд (*виж 5.4.2.*)
- алгоритъм на Дейкстра (*виж 5.4.2.*)
- минимален цикъл през връх (*виж 5.4.3.*)
- алгоритъм на Уоршал (*виж 5.5.1.*)
- обратен алгоритъм на Уоршал (*виж 5.5.3.*)
- контрол на компании (*виж 5.5.4.*)
- алгоритъм на Крускал (*виж 5.7.1.*)
- алгоритъм на Прим (*виж 5.7.1.*)

Кои други разгледани алгоритми биха могли да се добавят?

Глава 9
Евристични и вероятностни алгоритми

“... при някои задачи е за предпочитане да се приложи алгоритъм, който не винаги работи коректно, но за сметка на това работи изключително бързо. За тези задачи е пресметнато, че вероятността космическата радиация да попречи на правилно работещ алгоритъм по време на неговата работа, е по-висока от тази евристичният алгоритъм да не намери правилно решението. Някои автори считат, че последната философия е едно от нещата, които разграничават математическото от информатическото мислене ...”

9.1. Алчни алгоритми

В голяма част от задачите, които разгледахме в предходните глави, се търсеше оптималното измежду възможните решения. Нека припомним: при метода *търсене с връщане*, за да се намери оптималното решение, е необходимо да се намерят решенията на всички подслучаи на задачата (не непременно оптимални). Основен недостатък е, че някои от тях евентуално биват пресмятани многократно. Повторното пресмятане на едни и същи подслучаи може да се избегне, като се приложи динамично оптимиране, но за нещастие последното е свързано с необходимост от достатъчно памет за запазване на резултатите (и с необходимост от оптимална подструктура на решението, *виж 8.1.*). Идеята, която стои зад *евристичните алгоритми*, е следната: евристичният алгоритъм се насочва към *един* от всичките подслучаи на задачата и решава единствено него, с "надеждата", че той ще се окаже правилният. Изборът на този подслучай се извършва въз основата на локален критерий за оптималност. Така например, *алчните алгоритми*, както подсказва и самото име, се насочват винаги към най-добрия за момента избор, погледнато *локално*, като съвсем естествено, на по-късен етап може да се окаже, че този избор не е бил най-подходящият, погледнато *глобално*.

Алчните алгоритми се съставят лесно, съответната реализация на алгоритъма не е сложна и единственият недостатък е, че понякога те не гарантират правилното решение на задачата. Последното обаче не намалява ползата от тях — за евристичните алгоритми (и в частност алчните) е характерно, че бързо успяват да намерят решение, *близко до оптималното*. В много практически задачи е невъзможно да се изследват всички случаи и често съставянето на алгоритъм, който намира с 5% по-лошо решение от оптималното, се счита за успех, сравнено с алтернативата за едно почти безкрайно и безперспективно претърсване за “истинско” оптимално решение.

Ще се върнем на алчните алгоритми и ще илюстрираме как те се прилагат върху някои конкретни примери. Първата задача, която ще разгледаме, е следната: Да се намери начин за получаване на дадена сума m (m е естествено число), като се използват минимален брой банкноти, с номинали от множеството $C = \{a_1, a_2, ..., a_n\}$. Например, за българската национална валута стойностите на банкнотите са 1, 2, 5, 10, 20, 50 лева. Да разгледаме следния алгоритъм:

1) Инициализираме $s = 0$

2) Намираме банкнотата i с максимална стойност a_i ($a_i \in C$), такава че $s + a_i \leq m$.
   2.1) Ако няма банкнота, за която $s + a_i \leq m$, следва, че задачата *няма решение*. Край.
   2.2) Иначе, вземаме банкнотата i и увеличаваме s с a_i.
      2.2.1) Ако $s = n$ следва, че *задачата е решена*. Край.
      2.2.2) Ако $s < n$, то още не сме получили цялата сума и повтаряме стъпка 2).

Така например, за сумата 298 лева ще бъдат избрани последователно пет банкноти от по 50 лева, две от 20, една от 5 и по една банкнота от два и един лева (общо 250 + 40 + 5 + 2 + 1 = 298).

Очевидно, описаният алгоритъм отговаря на критериите за алчен алгоритъм: на всяка стъпка той избира максималната по-стойност банкнота, като по този начин се стреми да постигне възможно най-бързо търсената сума. В конкретния пример той води до ефективно решение на задачата.

За съжаление не съществуват достатъчно общи схеми за това какви да бъдат номиналите на банкнотите, за които да може да се твърди, че алчният алгоритъм ще работи правилно за всяка сума (*виж задача 9.43.*). Така например, да разгледаме случая, при който възможните стойности за банкноти са 2, 5, 20 и 30 и искаме да получим сума 40. Алчният алгоритъм първо ще избере банкнота от 30 (максималното възможно), след което ще избере две банкноти по 5, т. е. общо 3 банкноти. Очевидно обаче, съществува и по-добро решение: две банкноти по 20. Освен че може да не намери оптималното решение, възможно е алчният алгоритъм изобщо да не намери решение. Така например, ако искаме да получим сума 6: след банкнотата със стойност 5 пред алгоритъма не остава никаква възможност за следващ избор (а сумата все пак може да бъде получена от 3 банкноти със стойност 2).

Все пак нещата не винаги са толкова лоши и съществуват редица задачи, при които може да се покаже, че алчният алгоритъм *винаги* намира решение. Материалът в този параграф обхваща най-известните задачи, които *могат* да се решат лесно и ефективно с помощта на алчни алгоритми.

Задачи за упражнение:

1. Да се докаже, че алчният алгоритъм винаги работи за множеството от българските монети: 1, 2, 5, 10, 20, 50.
2. Да се дадат други примери, когато алчният алгоритъм ще работи винаги правилно.

9.1.1. Египетски дроби

Ще разгледаме една проста задача, подобна на тази за получаване на парична сума с минимален брой банкноти. Древните египтяни са използвали означение само за дробите с числител единица. Всяка друга дроб p/q представяли и записвали като сума от такива дроби (с числител единица). Например, 7/9 може да се представи като сума по някой от следните начини:

$$7/9 = 1/3 + 1/3 + 1/9$$
$$7/9 = 1/2 + 1/4 + 1/36$$
$$7/9 = 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9$$

Задача: Дадени са две естествени числа p и q ($q \neq 0$, $p < q$; $p,q \in N$). Да се намери представяне на дробта p/q във вид на сума:

$$p/q = 1/a_1 + 1/a_2 + \ldots + 1/a_n,$$

при което знаменателите да бъдат различни ($a_i \neq a_j$, $1 \leq i, j \leq n$, $i \neq j$, $a_i \geq 2$, $a_j \geq 2$, a_i, $a_j \in N$).

Забележка: Възможно е задачата да има повече от едно решение. Например, за дробта 3/7 две възможни решения са:

$$3/7 = 1/3 + 1/11 + 1/231$$
$$3/7 = 1/4 + 1/8 + 1/19 + 1/1064$$

В конкретната задача търсим *произволно* решение, отговарящо на условието знаменателите на намерените дроби да бъдат различни.

Съществува прост алчен алгоритъм за решаване на задачата: На всяка стъпка поредният член в сумата да бъде максималната дроб, която може да се добави към текущата сума така, че резултатът да не надвишава p/q (тъй като числителят е винаги 1, това означава дробта с най-малък знаменател). Например, за p/q = 7/9 най-голямата възможна дроб е 1/2. По-нататък трябва да изберем нова дроб $1/a_2$, така че

$$1/2 + 1/a_2 \leq 7/9, \text{ т. е. } 1/a_2 \leq 7/9 - 1/2, \text{ или } 1/a_2 \leq 5/18.$$

Най-голяма дроб, отговаряща на това условие, е 1/4, при което получаваме:

$$1/a_3 \leq 7/9 - 1/2 - 1/4 = 5/18 - 1/4 = 2/72,$$

т. е. максималното a_3 е 1/36, с което сумирането приключва, тъй като 1/2 + 1/4 + 1/36 = 7/9. Изчисленията в последния пример подсказват каква ще бъде схемата за реализация на алгоритъма:

```
while (p > 1) {
  Намира_се_максималната_дроб_1/r_ненадвишаваща_p/q;
  Отпечатва_се_дробта_1/r;
  p/q = p/q - 1/r;
}
```

В горната схема трябва да уточним две неща. Първото е, как да търсим максималната дроб $1/r$, ненадвишаваща p/q ($q \neq 0$), т. е. минималното r, за което е изпълнено $1/r \leq p/q$, $r \geq 2$, $q \geq 2$. Последното неравенство е еквивалентно на $r \geq q/p$. За да намерим r, можем да извършим делението q/p и да вземем най-малкото цяло число, не по-малко от q/p (в езика Си може да се използва функцията `ceil(q/p)`). За да избегнем използването на реални числа (за по-голямо бързодействие, както и за да си спестим грешки при закръгляне), ще намерим r, като използваме само целочислено деление: `r = (q+p)/p;`.

Разликата $p/q - 1/r$ се пресмята чрез привеждане под общ знаменател. Така, новите стойности за p и q ще бъдат:

```
p = p*r - q;
q = q*r;
```

Възможно е, след пресмятане на разликата да се получи съкратима дроб. Това ще попречи на правилната работа на алгоритъма единствено в случая, когато се получи дроб p/q, която може да се съкрати до $1/x$, т. е. `q % p == 0`. В този случай, ако не се извърши съкращението, условието $p > 1$ ще продължава да бъде изпълнено и търсенето на дроби ще продължи до безкрайност. (*Защо?*) Следва реализация на алгоритъма, където сме се погрижили за този случай.

```
/* ако q е кратно на p, се извършва съответно съкращение */
void cancel(unsigned long *p, unsigned long *q) {
  if (0 == *q % *p) {
    *q /= *p;
    *p = 1;
  }
}

void solve(unsigned long p, unsigned long q) {
  printf("%lu/%lu = ", p, q);
  cancel(&p, &q);
  while (p > 1) {
    /* намира максималната дроб 1/r, 1/r<=p/q */
    unsigned long r;
    r = (q + p) / p;
    printf("1/%lu + ", r);
    /* изчислява p/q - 1/r */
```

```
    p = p * r - q;
    q = q * r;
    cancel(&p, &q);
  }
  if (p > 0)
    printf("%lu/%lu ", p, q);
  printf("\n");
}

int main(void) {
  solve(3, 7);
  return 0;
}
```
🖫 egypt.c

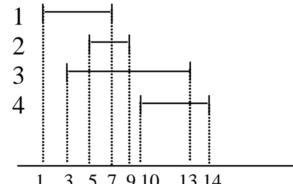
Резултат от изпълнението на програмата:
```
3/7 = 1/3 + 1/11 + 1/231
```

Задачи за упражнение:

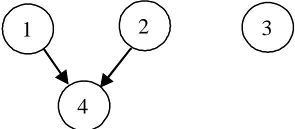
1. Да се докаже, че описаният по-горе алгоритъм работи правилно.
2. В горната реализация се извършва съкращаване единствено в случая, когато може да се съкрати до $1/x$. Какъв недостатък има при този подход? Да се предложи и реализира съответно подобрение.
3. Да се реши оптимизационният вариант на задачата, при който се търси сума с минимален брой събираеми. Съществува ли лаком алгоритъм, който винаги намира решение?
4. Съществуват различни разновидности на задачата за египетските дроби [*Knott-1*]. В един широко известен вариант на задачата е наложено допълнителното ограничение всички знаменатели a_i на редицата от дроби да бъдат *нечетни* числа. Съществува ли представяне като сума от дроби с числител 1 за всяка положителна обикновена дроб при това ограничение?
5. Ще работи ли коректно горната реализация, ако се търси сума от дроби с *нечетни* знаменатели? Да се покаже, че следната модификация на алгоритъма *няма* да решава задачата винаги: Търси се минималното нечетно r, за което $r \geq q/p$. Ако `r = ceil(q/p)` е четно, r се увеличава с единица.
6. Има ли смисъл задачата, ако се наложи условието за четни знаменатели?
7. Да се реши оптимизационен вариант на задачата при изискването за нечетни знаменатели, при който се търси сума с минимален брой събираеми. Съществува ли лаком алгоритъм, който винаги намира решение?

9.1.2. Максимално съчетание на дейности

Задача: Дадени са n лекции, които трябва да бъдат преподадени (дейности, които трябва да бъдат извършени). Всяка лекция i се определя от две числа: фиксирано *начало* s_i и фиксиран *край* f_i. Числата s_i и f_i могат да означават началния и крайния час за лекцията — можем да считаме (без ограничение на общността), че те са естествени числа. Трябва да се изберат максимален брой лекции, така че никои две от избраните да не се провеждат по едно и също време, т. е. ако приемем, че разполагаме само с една зала за лекции, то във всеки един момент t в залата може да се преподава най-много една лекция i ($s_i \leq t \leq f_i$, $1 \leq i \leq n$).



Фигура 9.1.2а. Конфигурация от четири лекции: начало и край.



Фигура 9.1.2б. Граф за конфигурацията.

Задачата е класически пример за ефективен и правилно работещ алчен алгоритъм [*Brassard*,*Bratley-1996*]. Да разгледаме един конкретен пример. На *фигура 9.1.2а.* е показана конфигурация от четири лекции. Лекциите с номера *1* и *4* не се пресичат във времето, докато *1* и *2*, *2* и *3* и др. се пресичат (т. е. не могат да се проведат едновременно).

Алгоритъм 1:

Построяваме ориентиран граф $G(V, E)$ с n върха, в който на всеки две "непресичащи" се лекции i, j ($f_i < s_j$) съответства реброто (i, j) от графа (*виж фигура 9.1.2б.*). Ако намерим най-дълъг път в G (например като използваме разгледания в *5.4.2* алгоритъм) ще получим алгоритъм със сложност $\Theta(n^2)$.

Алгоритъм 2:

Възможно е задачата да се реши и без да се строи граф, като се използва динамично оптимиране (*виж глава 8*). Дефинираме целева функция F, максимизираща броя на избраните лекции за време от 0 до t, по следния начин:

$F(t) = 0$, ако не съществува лекция i, за която $f_i < t$ и

$F(t) = \max\limits_{\substack{i=1,2,\ldots,n \\ f_i<t}} \{F(s_i)+1\}$, в противен случай.

Търсеното решение е стойността на функцията $F(t_0)$ за $t_0 = \max\limits_{i=1,2,\ldots,n} \{f_i\}$.

Ако приложим *memoization* (*виж глава 8*) за горната рекурентна формула, ще се наложи да използваме допълнителна памет. Например, можем да въведем масив `calc[t]`, в който да пазим всяка вече пресметната стойност на t. От него ще се използват най-много $2n$ полета (може да има съвпадения). Ако t е прекалено голямо число, решение по този начин изобщо няма да бъде възможно. За да решим този проблем ще използваме списък `calc[2*n]`, всеки елемент на който има две полета: момент `calc[i].t` и максимален брой лекции, които могат да се изнесат до този момент — `calc[i].maxl`.

Реализацията на описания по-горе алгоритъм предоставяме на читателя. Умишлено няма да се задълбочаваме повече в него, тъй като сложността му е квадратична, а използваната допълнителна памет — линейна относно броя на лекциите n.

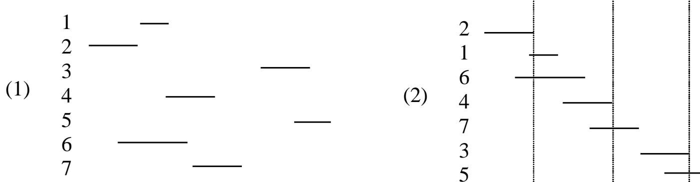
Алгоритъм 3:

Задачата може да се реши просто и ефективно с алчен алгоритъм, при което се постига сложност $\Theta(n.\log_2 n)$, без да се използва допълнителна памет. Сложността на алчния алгоритъм е дори линейна — проблемът се състои в това, че трябва да разполагаме с лекциите, сортирани във възходящ ред по f_i. И тъй като сортирането в общия случай е със сложност $\Theta(n.\log_2 n)$, то определя сложността на решението като цяло.

Алгоритъм:

Разглеждаме лекциите последователно от първата към последната:

1) Избираме се лекция i, за която f_i е минимално (в началото винаги се избира първата, тъй като лекциите са сортирани в нарастващ ред по f_i)
2) Всички лекции j, за които $s_j \leq f_i$ (т. е. които се "засичат" с избраната), се изключват от списъка на разглежданите лекции. Повтаря се стъпка 1) и така, докато се разгледат всички лекции (*виж фигура 9.1.2в.*).



Фигура 9.1.2в. Сортиране на лекциите и намиране на максимално съчетание.

На *фигура 9.1.2в.* в (2) са показани лекциите от (1), сортирани по f_i. Алчният алгоритъм ще избере лекция 2, след това ще пропусне лекции 1 и 6, ще избере 4, ще прескочи 7, ще избере 3 и ще пропусне 5. Схемата на алгоритъма като псевдокод изглежда най-общо така:

```
Избира_се_лекция_1;
i = 1; j = 1;
while (j <= n) {
  j++;
  if (s[j] > f[i]) {
    Избира_се_лекция_j;
    i = j;
  }
}
```

Следва и изходният код на програмата (приемаме, че лекциите вече са сортирани по f_i). Входните данни са зададени в началото като константи.

```
#include <stdio.h>
#define MAXN 100

const unsigned n = 7;
const unsigned s[MAXN] = { 3, 7, 5, 9, 13, 15, 17 };
const unsigned f[MAXN] = { 8, 10, 12, 14, 15, 19, 20 };

void solve(void)
{ unsigned i = 1, j = 1;
  printf(" Избрани лекции: %u ", 1);

  while (j++ <= n)
    if (s[j - 1] > f[i - 1]) printf("%u ", i = j);
  printf("\n");
}

int main(void) {
  solve();
  return 0;
}
```

🖫 aselect.c

Резултат от изпълнението на програмата:

```
Избрани лекции: 1 4 6
```

Задачи за упражнение:

1. Да се обърне внимание, че програмата отпечатва номерата на лекциите според наредбата им, *след* като е приключило сортирането. Една възможност за запазване на оригиналната номерация е да се въведе допълнителен масив `prenum[]`, където в `prenum[j]` да бъде запазен началният номер на лекцията *j*. Така по-късно ще можем вместо *j* да се отпечатва `prenum[j]`. Да се направи съответна модификация на горната програма.
2. Да се реализира *алгоритъм 1* (основан на търсене на най-дълъг път в ориентиран граф).
3. Да се реализира *алгоритъм 2* (основан на динамично оптимиране, *виж глава 8*). Да се реализират два варианта: отгоре-надолу (рекурсивен) и отдолу-нагоре (итеративен).
4. Да се покаже, че *алгоритъм 1* работи винаги правилно.
5. Да се покаже, че *алгоритъм 2* работи винаги правилно.
6. Да се покаже, че *алгоритъм 3* работи винаги правилно.
7. Да се даде пример, когато задачата има няколко оптимални решения. Едно и също оптимално решение ли ще намират *алгоритми 1*, *2* и *3* в такъв случай?

9.1.3. Минимално оцветяване на граф и дърво

Да разгледаме друг вариант на задачата за максимално съчетание на дейности: търсим минималния брой лекционни зали, нужни за изнасянето на всичките *n* лекции.

Решение: Отново строим граф, като, за разлика от предходния параграф, графът този път ще бъде неориентиран и реброто (*i*, *j*) ще съществува единствено, когато лекции *i* и *j* се "засичат по време". По този начин решението се свежда до намиране на минимално оцветяване на върховете в построения граф.

Минималното оцветяване на граф е задача, която се решава с пълно изчерпване. За това вече стана дума на няколко пъти (*виж 5.8.1.*, *6.3.2.*). Целта в настоящия параграф ще бъде да покажем някои алчни алгоритми, които, за разлика от оригиналната задача за максимално съчетание на дейности, не гарантират, че винаги ще бъде намерено правилно решението на задачата (В началото на главата вече стана дума за ползата от подобен род алгоритми. В материала по-нататък, изследвайки и други различни евристични техники, отново ще се убедим в това).

Алгоритъм 1:

Оцветяваме последователно върховете от графа в предварително определен ред. Всеки пореден връх се опитваме да оцветим с някой от използваните до момента цветове. Ако не успеем, прибавяме нов неизползван цвят и върхът се оцветява с него. Изборът, който трябва да извърши алчният алгоритъм, се свежда до две неща:

- Кой цвят да бъде избран, когато даден връх може да се оцвети с повече от един цвят
- В каква последователност да се оцветяват върховете от графа. Най-добри резултати се получават, ако върховете се разглеждат по реда на намаляване на степента им.

Алгоритъм 2:

Оцветяваме произволни несъседни върхове само с първия цвят. Продължаваме, докато не остане нито един връх, който да може да се оцвети с този цвят. Повтаряме същата процедура със следващия цвят и така, докато оцветим всички върхове.

Съществуват и други алчни алгоритми, основаващи се на различни характеристики на графа [*Culberson-1993*][*Wai-2000*].

Ще реализираме двата, описани по-горе, в обща програма: оцветяването по *алгоритъм 1* извършва функцията `solve1()`, а по *алгоритъм 2* — функцията `solve2()`. Входните данни са зададени в началото на програмата като константи.

```
#include <stdio.h>

#define MAXN 200

const unsigned n = 6;

const char A[MAXN][MAXN] = {
  { 0, 1, 0, 0, 0, 0 },
  { 1, 0, 1, 0, 0, 1 },
  { 0, 1, 0, 0, 1, 0 },
  { 0, 0, 0, 0, 0, 1 },
  { 0, 0, 1, 0, 0, 0 },
  { 0, 1, 0, 1, 0, 0 }
};

unsigned color[MAXN];

/* върховете се разглеждат в произволен ред */
void solve1(void)
{ unsigned i, j, c;
  /* оцветява връх i с най-малкия възможен цвят */
  for (i = 0; i < n; i++) {
    c = 0;
    do {
      c++;
      for (j = 0; j < n; j++)
        if (A[i][j] && color[j] == c) break;
    } while (j < n);
    color[i] = c;
  }
}

void solve2(void)
{ unsigned c = 0, cn = 0, i, j;
  /* оцветява само с първия цвят, докато е възможно, след това само с
   * втория и т.н., докато всички върхове бъдат оцветени */
  while (cn < n) {
    c++;
    for (i = 0; i < n; i++) {
      if (!color[i]) {
        int flag = 1;
        for (j = 0; j < n; j++)
          if (A[i][j] && color[j] == c) {
            flag = 0;
            break;
          }

        if (flag) {
          color[i] = c;
          cn++;
        }
      }
    }
  }
}

void showColor(void)
{ unsigned i;
  for (i = 0; i < n; i++) printf("%u-%u; ", i + 1, color[i]);
```

```
  printf("\n");
}

int main(void) {
  unsigned i;
  printf("Оцветяване на върховете по алгоритъм 1: \n");
  for (i = 0; i < n; i++) color[i] = 0;
  solve1();
  showColor();
  printf("Оцветяване на върховете по алгоритъм 2: \n");
  for (i = 0; i < n; i++) color[i] = 0;
  solve2();
  showColor();
  return 0;
}
```

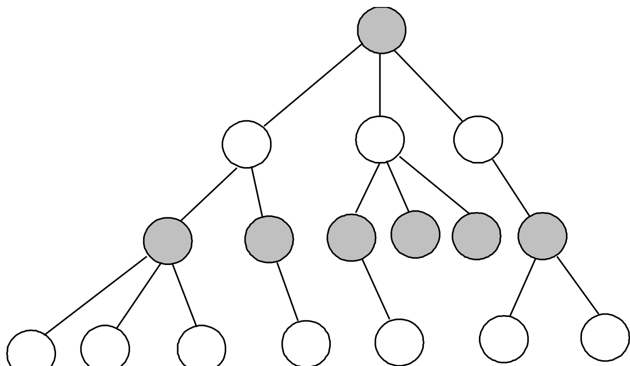
🖫 colorg.c

Резултат от изпълнението на програмата:

```
Оцветяване на върховете по алгоритъм 1:
1-1; 2-2; 3-1; 4-1; 5-2; 6-3;
Оцветяване на върховете по алгоритъм 2:
1-1; 2-2; 3-1; 4-1; 5-2; 6-3;
```

В частния случай, когато графът е ацикличен (и тъй като е неориентиран, следва че можем да го разглеждаме като дърво или гора), съществува прост алчен алгоритъм, който винаги намира минималното оцветяване. Не е трудно да се досетим, че са необходими точно два цвята (с изключение на случаите, когато дървото е празно или се състои единствено от корен без наследници — тогава са достатъчни съответно нула и един цвят).

Алгоритъмът е следният: оцветяваме корена на дървото с първия цвят, неговите наследници — с втория, наследниците на неговите наследници — отново с първия и т.н. Изобщо, върховете, намиращи се на четно разстояние от корена (това включва и самия корен), оцветяваме с първия, а тези на нечетно разстояние — с втория цвят (*виж фигура 9.1.3а.*).



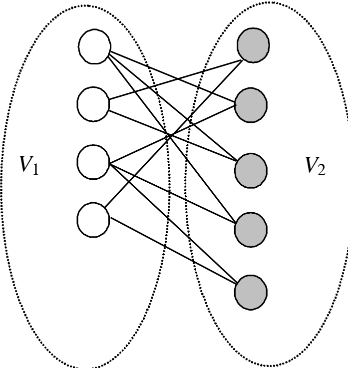
Фигура 9.1.3а. Оцветяване на дърво.

Алгоритъмът може да се реализира по следната примерна рекурсивна схема:

```
void treeColor(връх i, int color) {
  i = color; /* върхът i се оцветява с цвят color */
  for (всеки наследник j на i) {
    treeColor(j, !color);
  }
```

```
}

main() {
  treeColor(root, 0); /* извиква се с корена на дървото */
  return 0;
}
```

Аналогично, в по-общия случай на двуделен граф (фактически всяко дърво представлява двуделен граф, *защо?*), отново може да се състави прост алгоритъм за оцветяване, използващ само два цвята. Върховете от множеството V_1 ще бъдат оцветени с първия, а тези от V_2 — с втория цвят (*фигура 9.1.3б.*).



Фигура 9.1.3б. Оцветяване на двуделен граф.

Реализацията на последните два алгоритъма предоставяме на читателя.

Задачи за упражнение:

1. В предложените реализации на двата алгоритъма за намиране на оцветяване на произволен граф върховете се разглеждат последователно от *1* до *n*. В примерния изход резултатът от двата алгоритъма съвпада. Ще съвпада ли намереното оцветяване и за произволен друг граф? Да се намери пример, при който намерените оцветявания са различни, или да се докаже, че те винаги съвпадат и да се обясни разликата между алгоритмите и реализацията им.
2. Да се модифицира реализацията на *алгоритъм 1* така, че върховете да се разглеждат по реда на намаляване на степента им. Да се сравни резултатът (броят цветове в минималното оцветяване при произволно генерирани графи) с оригиналния вариант. Постига ли се подобрение?
3. Да се определят сложностите на *алгоритми 1* и *2*.
4. Да се покаже, че всяка гора е двуделен граф.

9.1.4. Алгоритми на Прим и Крускал

Алгоритмите на Прим и Крускал за намиране на минимално покриващо дърво (разгледани в *5.7.1*) са добра илюстрация на принципите на алчните алгоритми. Идеята, която е "скрита" и в двата алгоритъма, е следната: покриващо дърво се строи, като на всяка стъпка се избира ребро с *минимално* тегло, свързващо съответно две несвързани досега поддървета от графа (Крускал) или текущото покриващо дърво с нов връх извън него (Прим). Интуитивно ясно е, че подобен подход (характерен за алчните алгоритми) ще намери правилно минимално покриващо дърво, но както при повечето алчни алгоритми, доказателството на този факт не е толкова тривиално.

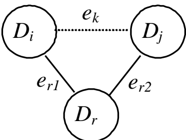
Освен *оптималната подструктура* на решението (необходимо условие, характерно и за динамичното оптимиране), за да бъдем сигурни, че един алчен алгоритъм решава поставената задача, трябва да докажем, че "алчният" избор на всяка стъпка от алгоритъма е *правилен*. Това найчесто се прави с допускане на противното. Показваме, че ако алгоритъмът на някоя фиксирана стъпка *k* не направи своя "алчен" избор, то в края полученото решение няма да бъде оптимално

(точно поради грешния избор на стъпката k). Ето как се прилага тази техника в задачата за минимално покриващо дърво:

При алгоритмите на Прим и Крускал на всяка поредна стъпка добавяме *минималното* ребро (свързващо съответно нов връх към дървото или две несвързани до момента поддървета). Нека на някоя фиксирана стъпка k e_k е минималното ребро, което свързва някои несвързани до момента поддървета D_i и D_j. Нека, противно на логиката на алгоритъма, *пропуснем* реброто e_k и *забраним* да бъде избирано и по-нататък.

Тогава, за да получим покриващо дърво, на някоя следваща стъпка ще се наложи да изберем друго ребро e_r (по-тежко от e_k), за да се свържат върховете от D_i и D_j, или две други ребра (e_r' и e_r'') които да свързват поотделно съответно D_i и D_j с останалата част D_r ($D_r = E\backslash(D_i \cup D_j)$):

1) В първия случай на последната стъпка, когато покриващото дърво е построено, се оказва, че то не е минимално: можем да получим покриващо дърво с по-ниска цена, ако просто заменим реброто e_r с e_k.
2) *Фигура 9.1.4.* показва другата възможна конфигурация.



Фигура 9.1.4. Оптималност на дървото, изграждано от алгоритмите на Прим и Крускал.

Получаваме цикъл, в който участват ребрата e_r' e_r'' и e_k. Отново, ако заменим e_r', или e_r'' с e_k, ще получим покриващо дърво с по-ниска цена (това е така, тъй като ребрата e_r' и e_r'' са с по-високо тегло от e_k — те не са участвали в D_i и D_j, а при нашето разглеждане на тази стъпка минималното ребро беше e_k).

Макар изложените разсъждения да не представляват строго доказателство (такова може да бъде намерено в [*Манев-1996*]), те са полезна схема за действие, когато предполагаме, че поставената задача се решава с алчен алгоритъм. Методът се прилага успешно при много алчни алгоритми, както ще видим по-нататък в тази глава.

Задача за упражнение:

На базата на горните разсъждения да "се сглобят" по-строги доказателства на правилността на работата на алгоритмите на Прим и Крускал.

9.1.5. Дробна задача за раницата

Дробният вариант на известната задача за раницата е друг пример за правилно работещ алчен алгоритъм.

Ще припомним, че в оригиналния вариант на задачата за раницата (*виж 6.4.1.*, *8.2.1.*) са дадени:

- N предмета, всеки от които с тегло m_i и стойност c_i.
- ограничително тегло M (издръжливост на раницата).

При тази задача предметите не могат да се делят на части (поради което е известна като *0-1 задача за раницата* — всеки един предмет или се взема, или — не).

От друга страна, при *дробната задача за раницата* можем да вземаме произволна *част* от всеки предмет. При този вариант на предметите може да се гледа като на нещо, което може да се раздробява на части, и стойността на съответната част се определя пропорционално на общата стойност. Такива предмети например могат да бъдат сребърен прах, стрит магданоз, лист ламарина (който може да бъде разрязан на произволни парчета) и т.н. В задачата отново се търси

подходящ избор на предмети (или части от тях) за “запълване” на раницата така, че общата получена стойност да бъде максимална.

Алгоритъмът, който ще приложим при решаване на тази задача, отговаря напълно на критерия "алчност": ще вземаме *максимална* част от *най-скъпия* предмет (т. е. този с *максимална* стойност на отношението c_i/m_i — общата цена, разделена на количеството, определя себестойността на единица от предмета), след това максимална част от втория по-себестойност предмет и т.н., докато "запълним" раницата. Най-много един предмет ще бъде разделен на две части, а всички останали предмети в раницата ще бъдат взети изцяло. *Таблица 9.1.5.* съдържа информация за параметрите на три предмета.

Предмет	Обща стойност c_i	Налично количество m_i (в килограми)	Себестойност (c_i/m_i)
1	25	10	2,5
2	12	8	1,5
3	16	8	2,0

Таблица 9.1.5. Раницата побира 16 килограма.

Алгоритъмът избира предмета с най-висока себестойност (предмет 1) и взема 10 килограма от него (с обща стойност 25). За да се допълнят оставащите 6 килограма в раницата, се избират 6 килограма от следващия по стойност предмет — този с номер 3 (6 кг × 2 = 12), и получената (максималната възможна) печалба е 25 + 12 = 37.

Отново, без да претендираме за строго доказателство, ще покажем, че алчният алгоритъм винаги намира винаги оптимално решение. За целта ще използваме допускане на противното. Нека на някоя фиксирана стъпка k сме взели предметите $j_1,j_2,\ldots,j_k$ и остават неразгледани $i_1,i_2,\ldots,i_t$. Нека, противно на действието на алчния алгоритъм, от тази стъпка нататък не избираме предмета $i_{\max}$, който е с максимална себестойност ($c_{i_{\max}}/m_{i_{\max}}$). Тогава, при запълване на раницата, няма да сме получили оптимално решение: по-голяма стойност можем да получим, ако заменим дори и минимално количество ε ($\varepsilon > 0$) от който и да е от предметите, взети *след* стъпката k, със същото количество от предмета $i_{\max}$.

Реализацията на алгоритъма не представлява особена трудност, затова няма да се спираме на подробности. Функцията `sort()` сортира предметите по себестойност, а функцията `solve()` осъществява цикъл, избиращ последователно предмета с най-висока себестойност. Входните данни са зададени като константи.

```
#include <stdio.h>
#define MAXN 1000

const unsigned n = 3;             /* брой на предметите */
float c[MAXN] = { 25, 12, 16 };   /* себестойност на предметите */
float m[MAXN] = { 10, 8, 8 };     /* количества от предметите */
const float M = 16;               /* ограничително тегло на раницата */

float ratio[MAXN];

void swap(float *a, float *b)
{ float s = *a; *a = *b; *b = s; }

/* сортира предметите по себестойност */
void sort(void)
{ ...
}

/* намира решението */
void solve(void)
{ unsigned i = 0;
```

```
    float T = 0, V = 0;;
    while (T + m[i] <= M) {      /* взема цели предмети, докато може */
      printf("Вземат се 100%% от предмет със стойност %.2f");
      printf(" и тегло %.2f \n", c[i], m[i]);
      T += m[i]; V += c[i];
      i++;
    }
    printf("Вземат се %.2f%% от предмет със стойност %.2f и тегло %.2f \n",
           ((M - T) / m[i]) * 100, c[i], m[i]);
    V += (M - T) * (c[i] / m[i]);
    printf("Обща получена цена: %.2f\n", V);
  }

  int main(void) {
    unsigned i;
    for (i = 0; i < n; i++) ratio[i] = c[i] / m[i];
    sort();
    solve();
    return 0;
  }
```

fracknap.c

Резултат от изпълнението на програмата:

```
Вземат се 100% от предмет със стойност 25.00 и тегло 10.00
Вземат се 75.00% от предмет със стойност 16.00 и тегло 8.00
Обща получена цена: 37.00
```

Забележка: Не бива да бъдем прекалено "алчни" и да се опитваме да прилагаме алгоритъма в 0-1 задачата за раницата (която е *NP*-пълна). За да се убедим в това, е достатъчно да проверим, какъв ще бъде резултатът за избрания пример от *таблица 9.1.5.*: след избирането на предмет 1 със стойност 25 няма да можем да поберем друг предмет в раницата (тъй като при този вариант не е позволено да се разделят предметите). Така не получаваме оптималния резултат: предмети 2 и 3 се събират заедно в раницата и дават по-голяма обща стойност: 12 + 16 = 28.

Задачи за упражнение:

1. Да се дадат още примери, когато алчният алгоритъм работи и когато не работи за 0-1 задачата.
2. Необходим ли е масивът `ratio[]` в горната реализация? Да се реализира подходяща функция за сортиране на рационални дроби и да се направят съответните модификации във функцията `solve()`. Упътване: вместо a/b и c/d може да се сравняват ad и bc.

9.1.6. Задача за магнитната лента

Разновидност на дробната задача за раницата е *задачата за магнитната лента*. Дадени са n програми с дължини l_1, l_2, ... , l_n и магнитна лента с последователен достъп. Последователният достъп означава, че за да се прочете запис, намиращ се на дадена позиция върху лентата, трябва да се премине (да се превърти лентата) до указаното място.

K_1	K_2	K_3	K_4	....	K_n
l_1	l_3	l_2	l_{K_4}	...	l_{K_n}

Фигура 9.1.6. Магнитна лента с последователен достъп ($K_1 = 1$, $K_2 = 3$, $K_3 = 2$, ...).

Например (*виж фигура 9.1.6.*), за да бъде прочетена третата програма (с дължина l_2), трябва да се "превъртят" всички програми преди нея — тези с дължина l_1 и l_3.

Дадени са вероятностите на изпълнение на всяка програма $p_1, p_2, ..., p_n$, $0 \le p_i \le 1$, $1 \le i \le n$, $\sum_{i=1}^{n} p_i = 1$. Например, вероятност 0,5 означава, че програмата се изпълнява толкова често, колкото всички останали програми, взети заедно. При фиксирано подреждане $K_1, K_2, \ldots, K_n$ на програмите върху лентата *средното време* T, необходимо за зареждането на произволна програма, се дефинира по следния начин:

$$T = \frac{\sum_{i=1}^{n} \left(p_{K_i} \sum_{j=1}^{i} l_{K_j} \right)}{n}$$

Задачата е да се намери подреждане на програмите върху лентата, минимизиращо T.

Ще използваме алчен алгоритъм. Интуитивно ясно е, че колкото по-често се използва една програма, толкова по-напред върху лентата трябва да бъде записана тя. Другият определящ критерий е дължината на програмата — колкото по-дълга е тя, толкова по-назад трябва да бъде върху лентата (за да "пречи" колкото се може по-малко, когато се наложи да се превърта). Така достигаме до следния алгоритъм: Записваме последователно програмите върху лентата, като на всяка стъпка избираме програмата K_i, за която отношението p_{Ki}/l_{Ki} е максимално.

Правилността на последния алгоритъм може да се докаже лесно: достатъчно е да се пресметне как ще се промени сумата, ако се разместят два члена K_i и K_j в крайното подреждане. Непосредствената реализацията предоставяме на читателя.

Частен случай на току-що разгледаната задача е следната

Задача: Сървър трябва да изпълни n задачи. Дадени са времената за изпълнение на всяка от задачите — $t_1, t_2, ..., t_n$. Трябва да се минимизира сумата от времената, които всяка задача изчаква до изпълнението си, т. е. да се минимизира

$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{Kj}$ (ако $K_1, K_2, \ldots, K_n$ е реда, в който сървърът изпълнява задачите).

Решение: На всяка стъпка се избира задача, за която е необходимо най-малко време за изпълнение (т. е. сортиране по времената t_i). Всъщност, задачата за сървъра е частен случай на тази за магнитната лента, при която честотите на изпълнение са равни ($p_1 = p_2 = ... = p_n = \frac{1}{n}$).

Задачи за упражнение:

1. Да се докаже коректността на предложения алгоритъм за решаване на задачата за магнитната лента.
2. Да се реализира алгоритъмът за решаване на задачата за магнитната лента.

9.1.7. Процесорно разписание

Еднозадачен процесор трябва да изпълни n задачи. Задачите се изпълняват последователно, като за всяка една от тях е необходимо време 1. За всяка задача i ($1 \le i \le n$) са дадени две естествени числа: v_i и d_i. Първото от тях показва печалбата, която се акумулира от изпълнението на задачата, а второто — крайния срок за изпълнение (от англ. *deadline*). Т.е. ако задачата изобщо ще се изпълнява, това трябва да стане най-късно на d_i-тата стъпка. Търси се разписание за изпълнение на задачите, при което общата печалба ще бъде максимална.

Таблица 9.1.7а. съдържа конкретен пример с 5 задачи. Всички възможни разписания и съответните доходи от тях са показани в *таблица 9.1.7б*.

задача i	*1*	*2*	*3*	*4*	*5*
доход v_i	40	30	15	20	50
краен срок d_i	1	2	1	1	2

Таблица 9.1.7а. Задачи от разписанието.

изпълнени задачи	*(1)*	*(2)*	*(3)*	*(4)*	*(5)*	*(1,2)*	*(1,5)*	*(2,5)*	*(3,2)*	*(3,5)*	*(4,2)*	*(4,5)*	*(5,2)*
обща печалба	40	30	15	20	50	70	90	80	45	65	50	70	80

Таблица 9.1.7б. Разписание при дадените задачи.

Разписания, като например (1,3), (2,4) и т.н. не са възможни, тъй като крайният срок за задачите 3 и 4 е стъпка 1. От таблицата се вижда, че оптималното разписание (с печалба 90), е (1,5).

• *Първият алчен алгоритъм*, за който веднага можем да се досетим, е следният: На всяка стъпка i ($1 \leq i \leq n$) се изпълнява задача j ($1 \leq j \leq n$) от неизпълнените до момента, за която $d_j \geq i$ и v_j е максимално. Този алчен алгоритъм обаче, не винаги намира оптималното решение. Ако го приложим в примера по-горе, на стъпка 1 ще бъде избрана задача 5 с доход 50 (максималният възможен доход). След това, на втората стъпка, единствената задача, изпълняваща условието $d_j \geq 2$, е задача 2. Получава се общ доход 50 + 30 = 80, което очевидно не е оптималният резултат.

• На базата на разгледания алгоритъм ще построим нов, който ще гарантира намиране на оптималното решение:

Дефиниция 9.1. Множество от k задачи се нарича *изпълнимо*, ако съществува подреждане на задачите, така че всички да бъдат изпълнени преди крайните си срокове.

Алгоритъм

На всяка стъпка, докато това е възможно, алчният алгоритъм ще избира за изпълнение задача с максимален доход, но такава, че множеството от избраните задачи да остава *изпълнимо*.

Нека множеството $A = \{t_1, t_2, ..., t_{k-1}\}$ е изпълнимо. Ще разгледаме необходимо условие, за да бъдем сигурни, че A ще остане изпълнимо и след добавяне на задачата t_k към него:

- Трябва да се провери, дали задачите за изпълнение до i-тата стъпка (за всяко $i = 1,2,...,n$) не стават повече от i (на всяка стъпка може да се изпълнява по една задача).

Нека `index[j]` (индексираме от 0) показва *броя* на избраните до момента задачи, които имат краен срок (j+1)-вата стъпка. За всяко i пресмятаме $S = \sum_{j=0}^{i} index[j]$ и ако $S \leq i+1$, то условието е изпълнено и множеството ще остане изпълнимо и след добавяне на задачата t_k.

И така, в началото сортираме задачите по техния *доход*. Ще приемаме, че операцията сортиране на задачите вече е извършена и информацията се намира в двата масива `v[n]` и `d[n]`, показващи съответно дохода и крайния срок за изпълнението на всяка от задачите. По-нататък опитваме да изберем всяка една задача последователно. Тъй като задачите са сортирани, пробваме последователно дали, ако добавим задачата k ($k = 1, 2, ..., n$), множеството от избраните задачи остава изпълнимо. Проверката дали дадено множество е изпълнимо, се извършва по схемата, описана по-горе: въвели сме масива `index[]` и неговите стойности се променят при добавяне на нова задача k на всяка стъпка — увеличава се стойността на `index[d[k]]` с единица. Масивът `taken[]` се използва за запазване на задачите в последователността, в която са избирани. Сложността на предложената реализация е $\Theta(n^2)$.

За разгледания пример на първата стъпка алгоритъмът ще избере задача 5, а на втората — задача 1, получавайки оптималната сума 90, като и при двата избора множеството от задачите остава изпълнимо. Следва изходният код на програмата:

```
#include <stdio.h>
#define MAXN 1000
const unsigned n = 5;
const unsigned v[MAXN] = { 50, 40, 30, 20, 15 };
const unsigned d[MAXN] = {  2,  1,  2,  2,  1 };
const unsigned p[MAXN] = {5,1,2,4,3}; /* ориг. номерация на задачите */
```

```
unsigned index[MAXN], taken[MAXN], tn;

char feasible(unsigned k)
{ unsigned s = 0, i;
  for (i = 0; i < n; i++) {
    s += index[i];
    if (i == d[k] - 1) s += 1;
    if (s > i + 1) return 0;
  }
  return 1;
}

void solve(void)
{ unsigned k, i, income;
  for (k = 0; k < n; k++)
    if (feasible(k)) {
      taken[tn++] = k;
      index[d[k] - 1]++;
    }

  printf("Оптимално разписание: ");

  income = 0;
  for (i = 0; i < tn; i++) {
    printf("%u ", p[taken[i]]);
    income += v[taken[i]];
  }
  printf("\nОбщ доход: %u\n", income);
}
int main(void) {
  unsigned i;
  for (i = 0; i < n; i++) index[i] = 0;
  tn = 0;
  solve();
  return 0;
}
```

🖫 schedule.c

Резултат от изпълнението на програмата:

```
Оптимално разписание: 5 1
Общ доход: 90
```

Задачи за упражнение:

1. Да се докаже, че описаният по-горе алгоритъм за намиране на оптимално процесорно разписание работи правилно.

2. Вижда се, че горната програма не отпечатва задачите в реда, в който трябва да се изпълнят, а определя единствено *множеството* от задачи, даващо оптималното решение. Да се предложи и реализира алгоритъм, който да намира *реда* на изпълнение на задачите:

а) по време на построяване на решението (в цикъла *for* на функцията `solve()`)

б) след като оптималното решение е вече определено

9.1.8. Разходката на коня. Хиперкуб. Код на Грей

В глава 6 (*виж 6.3.4.*) разгледахме задачата за разходката на шахматния кон: Да се намери път върху обобщена шахматна дъска $n \times n$ с хода на коня, започващ в горното ляво поле и посеща-

ващ всяко едно поле точно по веднъж. Там решихме задачата чрез търсене с връщане назад, като показахме как тази задача може да се сведе до търсене на Хамилтонов път в граф.

Въпреки че на пръв поглед може да изглежда странно, съществува и *полиномиално* решение. Ще покажем алчен алгоритъм, който я решава с n^2 стъпки (като всяка стъпка се състои от еднозначен избор на следващо поле от пътя на коня):

1) Започваме от полето с координати (1, 1).
2) Докато има още необходени полета от дъската, избираме следващото поле (x', y') от пътя така, че да удовлетворява едновременно следните условия:
   - (x', y') е достижимо с един ход на коня от текущото поле (x, y).
   - (x',y') е непосетено поле от дъската.
   - Броят на непосетените полета, до които може да се достигне с един ход на коня от (x', y'), е минимален.

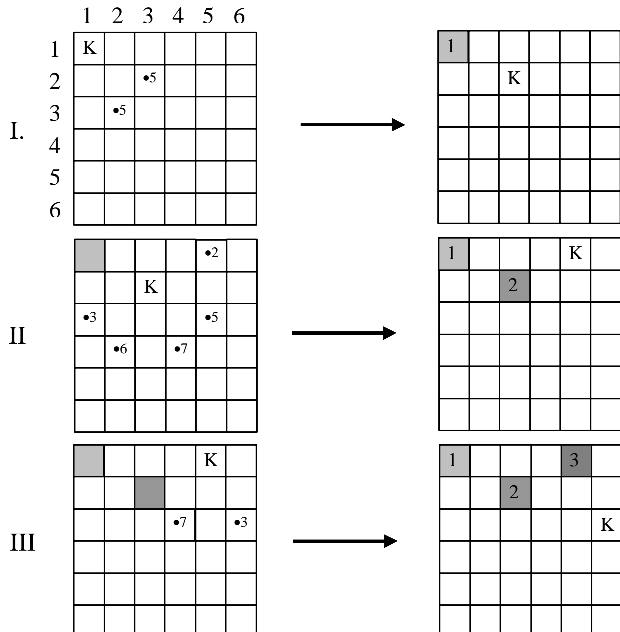
Например (*виж фигура 9.1.8а.*) първите няколко стъпки, които ще извърши алгоритъмът за дъска с размери 6×6, са следните:

I) От (1,1) може да се премине в полетата (2,3) и (3,2) — в случая няма значение кое от двете ще изберем — преминаваме в (2,3).

II) Новите възможности за ход са (1,5), (3,5), (4,4), (4,2) и (3,1). Във всяко от тези полета на фигурата е отбелязан броят на валидните следващи ходове. Избираме полето с минимално такова число: (1,5).

III) От (1,5) възможните ходове са (3,4) и (3,6): отново избираме да посетим това с минимален брой възможности за следващ ход — (3,6) и т.н.

Забележка: Позиция (x,y) означава (ред, колона), като полето (1,1) е в *горния ляв* ъгъл. Забележете, че това е различно от стандартното шахматно номериране на дъската, използвано в *6.3.4.*, където координатите са обърнати, а полето (1,1) е в долния ляв ъгъл.



Фигура 9.1.8а. Първите няколко хода на коня за дъска с размер 6×6.

Следва реализация на описания алгоритъм. Функцията `countMoves(x,y)` връща броя на необходените полета, достижими с един ход на коня от полето (x, y). Обхождането продължава, докато бъдат извършени n^2–1 хода: на всеки ход `countMoves()` се извиква за осемте възможности за следващ ход и се избира този, за който е върната минимална стойност (ако са няколко, се избира първата възможност). Входните данни са зададени в началото на програмата като константи.

```
#include <stdio.h>
#define MAXN      100
#define MAX_MOVES   8

const unsigned n = 12;
const int moveX[MAX_MOVES] = { +1, -1, +1, -1, +2, +2, -2, -2 };
const int moveY[MAX_MOVES] = { +2, +2, -2, -2, +1, -1, +1, -1 };

int a[MAXN][MAXN], x, y, p;

int countMoves(int x, int y)
{ int i, number = 0;
  if (x < 0 || y < 0 || x >= n || y >= n || a[x][y] != 0)
    return MAX_MOVES + 1;  /* невалиден ход */
  for (i = 0; i < MAX_MOVES; i++) {
    int nx = x + moveX[i], ny = y + moveY[i];
    if (nx >= 0 && ny >= 0 && nx < n && ny < n && a[nx][ny] == 0)
      number++;
  }
  return number;
}

void solve(void) {
  unsigned i, j, n2;
  /* инициализиране */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      a[i][j] = 0;
  x = 0;
  y = 0;
  a[0][0] = 1;
  p = 1;

  /* повтаря "алчната" стъпка, докато попълни цялата дъска */
  n2 = n * n;
  while (p < n2) {
    int min = MAX_MOVES + 1, choose;
    for (i = 0; i < MAX_MOVES; i++) {
      int temp = countMoves(x + moveX[i], y + moveY[i]);
      if (temp < min) {
        min = temp;
        choose = i;
      }
    }
    x += moveX[choose];
    y += moveY[choose];
    a[x][y] = ++p;
  }

  /* отпечатва резултата */
  for (i = 0; i < n; i++) {
```

```
    for (j = 0; j < n; j++) printf("%4d", a[i][j]);
    printf("\n");
  }
}

int main(void) {
  solve();
  return 0;
}
```
🖫 knightg.c

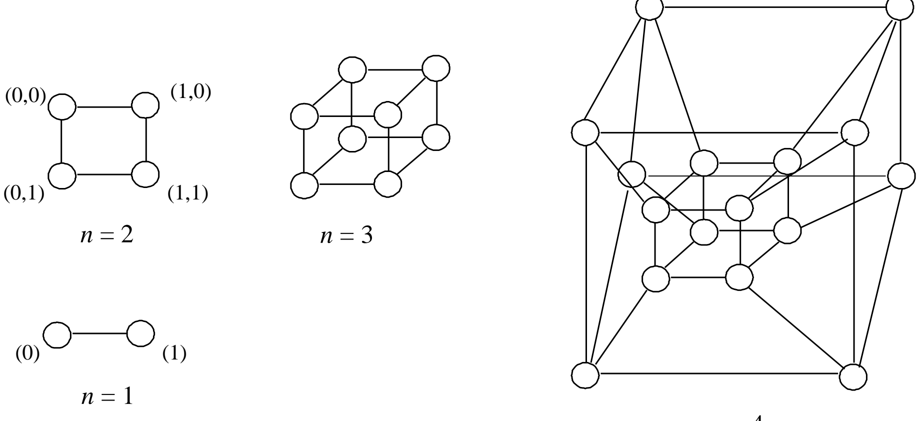
Резултат от изпълнението на програмата:

```
   1  24  61  96   3  26  91 110   5  28  31 112
  62  95   2  25  92 141   4  27 114 111   6  29
  23  60  93 144  97  90 135 142 109  30 113  32
  94  63  98  89 140 143 130 115 136 119 108   7
  59  22 139 100 129 134 137 126 107 116  33 118
  64  99  88 133 138 127 106 131 120 125   8  81
  21  58  65 128 101 132 121 124 105  82 117  34
  46  55 102  87  66  77 104  83 122  75  80   9
  57  20  47  54 103  86 123  76  79  84  35  74
  42  45  56  67  50  53  78  85  70  73  10  13
  19  48  43  40  17  68  51  38  15  12  71  36
  44  41  18  49  52  39  16  69  72  37  14  11
```

Съществува хипотеза (*хипотеза на Варнсдорф*), че описаният алгоритъм винаги открива обхождане на дъската с хода на коня. Последното не е доказано, но досега не е намерен контрапример. Така, въпреки че в *глава 6* решихме задачата с пълно изчерпване, сега се вижда, че тя не е *NP-пълна* и за нея съществува полиномиален (алчен) алгоритъм. Привидното противоречие се обяснява с това, че графът, който построихме (*виж фигура 6.3.4в.*), принадлежи на специален клас графи, в които може да се намери Хамилтонов път с полиномиална сложност. Освен тривиалният случай, когато графът е пълен (тогава произволна пермутация на върховете образува Хамилтонов път), съществуват и много други специфични видове графи, в които търсенето може да се извърши с полиномиална сложност. Такъв например е *n-мерният двоичен куб*, или *хиперкуб*:

Дефиниция 9.2. Свързаният неориентиран граф с 2^n върха, в който степента на всеки връх е точно n и между всеки два върха съществува път с дължина най-много n, G се нарича *n-мерен двоичен куб*.

n-мерните двоични кубове за $n = 1, 2, 3$ и 4 са показани на *фигура 9.1.8б*.



Фигура 9.1.8б. 1, 2, 3 и 4-мерен двоичен куб.

Названието *n-мерен двоичен куб* идва от геометрични съображения. При $n = 1$ графът може да се представи като отсечка върху реалната права с координати на краищата (0) и (1). При $n = 2$ му се съпоставя квадрат в равнината с координати (0,0), (0,1), (1,1) и (1,0). При $n = 3$ отново се прави съответна интерпретация с куб в пространството и т.н. *Хиперкуб* (т.е. n-мерен куб) е съответното обобщение за n-мерното пространство, като в този случай координатите ще бъдат наредени n-торки от 0 и 1 такива, че ако върховете i и j са съседни, то съответстващите им n-торки ще се различават *точно* в една позиция.

Намирането на Хамилтонов път (или дори цикъл) в n-мерен двоичен куб е полиномиално разрешима задача, а последователността от списъците, съпоставени на върховете в цикъла, има малко по-различно значение — в комбинаториката е известна като *двоичен код на Грей*. Съществува прост алгоритъм за намиране на двоичния код на Грей [*Рейнгольд, Нивергельт, Део-1980*]. Да разгледаме няколко примера:

$n = 1$
```
0
1
```

$n = 2$
```
0 0
1 0
1 1
0 1
```

$n = 3$
```
0 0 0
1 0 0
1 1 0
0 1 0
0 1 1
1 1 1
1 0 1
0 0 1
```

Нека $i_1, i_2, \ldots, i_{2^n}$ е код на Грей за $n = k$.

i_1
i_2
. . .
i_{2^n}

Код на Грей за $n = k + 1$ получаваме, като към всички вектори от кода на Грей за $n = k$ добавим 0, след което записваме векторите за $n = k$ в обратен ред и към всеки добавяме 1:

i_1 0
i_2 0
...
i_{2^n} 0
----
i_{2^n} 1
...
i_2 1
i_1 1

Следва реализация на алгоритъма, като генерирането се извършва с двойна рекурсия:

```
#include <stdio.h>

#define MAXN 1000

/* код на Грей, Хамилтонов цикъл в n-мерен двоичен куб (хиперкуб) */
const unsigned n = 3;
char a[MAXN];

void print(void)
{ unsigned i;
  for (i = 1; i <= n; i++) printf("%d ", (int) a[i]);
  printf("\n");
}

void forwgray(unsigned); /* prototype */

void backgray(unsigned k)
{ if (0 == k) { print(); return; }
  a[k] = 1;  forwgray(k - 1);
  a[k] = 0;  backgray(k - 1);
}

void forwgray(unsigned k)
{ if (0 == k) {  print(); return; }
  a[k] = 0;  forwgray(k - 1);
  a[k] = 1;  backgray(k - 1);
}

int main(void) {
  forwgray(n);
  return 0;
}
```

🖫 hamgray.c

Задачи за упражнение:

1. В предложената по-горе реализация на разходката на коня сложността на алгоритъма е $\Theta(m^2.n^2)$, където m е броят на възможните ходове на коня (в програмата по-горе — константата `MAX_MOVES`): Извършват се n^2 стъпки, на всяка от които има m извиквания на `countMoves()`, която от своя страна също има сложност $\Theta(m)$. Как, като се използва допълнително n^2 памет, сложността може да се редуцира до $\Theta(n^2)$?

2. Ще работи ли правилно алгоритъмът за обхождане с ход на коня, ако стартовото поле е в долния ляв ъгъл, както в *6.3.4.*? А при произволно зададено поле от дъската?

3. Да се докаже коректността на предложения алгоритъм за построяване на кода на Грей.

9.2. Търсене с налучкване

Търсенето с налучкване заема централно място сред вероятностните алгоритми. Това е програмистки подход, подчиняващ се на следното просто "правило": всеки път, когато програмата трябва да избира измежду няколко възможности, тя продължава по "произволен" начин.

За да е възможно на всяка стъпка да изберем измежду k ($k \in N$) налични възможности, трябва да разполагаме с функция `int random(k)`, връщаща произволно цяло число между 0 и k–1. В настоящия параграф няма да обсъждаме как и доколко е възможно да бъде съставена "прецизна" функция `random()` (подробно изследване на проблема може да бъде намерено в [*Knuth-2/1968*]). Ще приемем, че стандартната реализация, съдържаща се в заглавния файл `<stdlib.h>`, е достатъчно добра. Ще приведем няколко прости, но показателни примера на търсене с налучкване.

Пример 1: В началото на почти всички курсове по алгоритми и структури от данни се разглежда задачата: Да се провери дали дадено число m се среща като елемент на даден масив `a[n]`. Тъй като числата в масива са подредени по произволен начин (и не можем да приложим двоично търсене — *виж 4.3.*), се спираме на стандартния подход: сравняваме последователно всеки елемент `a[i]` (i = 1,2,...,n) с m. Този алгоритъм има средна сложност $\Theta(n)$, каквато е сложността му и в най-лошия случай. Едва ли би ни хрумнала идеята за прилагане на търсене с налучкване. Все пак един такъв алгоритъм би могъл да изглежда така:

1) Избираме *произволен* елемент `a[i]` от масива: `i = random(n)`.
   1.1) Ако `a[i] == m`, то сме намерили търсения елемент и приключваме.
   1.2) Ако `a[i] != m`, повтаряме стъпка 1.

Забелязваме, че сложността и на този алгоритъм в средния случай е $\Theta(n)$ — вероятността на всяка стъпка да попаднем на търсения елемент е $1/n$. Сложността в най-лошия случай обаче, не може да се определи: В общия случай, ако не налагаме допълнителни ограничения на функцията `random()` нищо не гарантира, че елементът, който търсим, някога ще бъде "уцелен", независимо от това, колко дълго работи програмата. Но каквато и функция `random()` да изберем, все едно, ще изпаднем в безкраен цикъл, ако елементът изобщо липсва. Горният алгоритъм не съобразява това и ще извършва проби до безкрайност. С последния проблем можем да се справим, като въведем битов масив `taken[n]`, инициализиран с нули, и променлива `number`, показваща колко различни елемента сме проверили до момента. Нека на текущата стъпка е бил избран елементът i, ако `taken[i]==0` (т.е. елементът не е разглеждан до момента), повдигаме съответния бит `taken[i]==1` и увеличаваме `number` с 1. Така, ако `number` стане равно на n, следва, че сме проверили всички елементи, и алгоритъмът приключва.

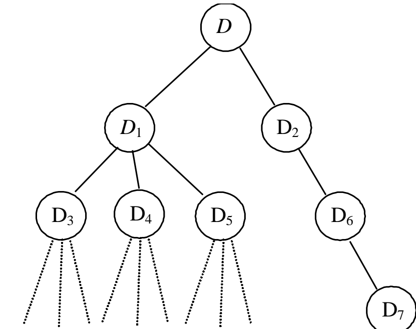
Пример 2: Аналогичен пример е задачата за проверка дали даден граф е Хамилтонов. Едно възможно решение е да избираме произволни пермутации от върхове и да проверяваме дали някоя пермутация не образува Хамилтонов цикъл. Така можем да го намерим още на първата стъпка, но може и въобще да не го открием. Търсенето никога няма да завърши и в случая, когато графът изобщо не съдържа Хамилтонов цикъл.

Двата примера трудно могат да ни убедят, че прилагането на алгоритъм за търсене с налучкване е по-добър подход от съответното систематично изследване. Забелязахме, че ако "нямаме късмет", и при двата алгоритъма можем да попаднем в безкрайно търсене на решение. Въпреки това, съществуват случаи, когато алгоритмите за търсене с налучкване могат да постигнат много добри резултати:

Пример 3: Всяка *NP*-пълна задача (*виж глава 6*) или задача, изискваща пълно изчерпване, се решава с обхождане на дървото на кандидатите за решение. При методичното изследване, на-

пример търсене с връщане (*виж 6.3.*), се извършва претърсване в дълбочина, докато при търсенето с налучкване се изследват произволни клонове.

За примера от *фигура 9.2.*, нека решението на задачата е пътят D-D_2-D_6-D_7, а поддървото с корен D_1 е прекалено голямо (по брой върхове, а не по височина), за да може да бъде изследвано изцяло. В този случай алгоритъмът за търсене в дълбочина никога няма да намери решението, ако се насочи първо към поддървото на D_1. От друга страна, нека приложим алгоритъм, който търси решението с налучкване по следния начин: започва от корена и на всяка стъпка избира произволен наследник, по който да продължи. Когато достигне до листо, проверява дали построеният път е решение и ако не е — започва от корена построяване на нов (произволен) път. Вероятността този алгоритъм да тръгне по върха D_2, вместо по D_1, е 0,5. При проверка на два кандидата тази вероятност е 0,75, при три кандидата е 0,875 (*Защо?*) и т.н., т.е. можем да очакваме с голяма вероятност, че ще открием решението още с първите няколко опита.



Фигура 9.2. Дърво на кандидатите за решение.

Изобщо, основното предимство на алгоритмите, извършващи търсене с налучкване, е *разнообразието* от изследвани възможности. По този начин може да се избегне изследването на огромен брой еднотипни (и безперспективни) случаи, а това води до задоволителни резултати при много практически задачи. В следващите точки ще разграничим няколко основни типа търсене с налучкване и ще покажем някои техни приложения.

Задача за упражнение:

Да се приведат още примери, при които вероятностният алгоритъм превъзхожда систематичното изследване.

9.2.1. Алгоритми Монте Карло и Лас Вегас

Две специфични (и доста сходни) разновидности на метода на търсене с налучкване са алгоритмите Монте Карло и Лас Вегас (двата града са добре известни със своите заведения за хазартни игри, откъдето идва и името на тези две "хазартни" техники).

Алгоритмите Монте Карло винаги "претендират", че са намерили решение на задачата. Понякога обаче, това решение се оказва невярно (в частност, ако разглеждаме оптимизационна задача, е възможно намереното от алгоритъма решение да бъде близко до оптималното, но все пак да съществува по-добро от него). От друга страна, алгоритмите Лас Вегас не винаги намират решение, но когато намерят такова, то със сигурност е вярно (съответно ще е оптималното, ако разглеждаме оптимизационна задача).

- проверка дали число е просто

Класически пример за приложение на алгоритъм Монте Карло е проверката дали дадено число n е съставно. Ако изберем k естествени числа p_1, p_2, ..., p_k ($2 \le p_i \le \sqrt{n}$) и n се дели на някое от тях, следва че n е съставно. Така получихме един съвсем прост алгоритъм Монте Карло:

Алгоритъм 1:

```
for (i = 1,...,k)
  if (n % (random(sqrt(n))+2) == 0) {
    return n_е_съставно;
  }
return n_е_просто;
```

Вижда се, че, ако във фрагмента по-горе се случи n да бъде съставно, резултатът със сигурност ще бъде правилен — можем да посочим един конкретен делител на n. Ако алгоритъмът не намери делител измежду избраните k числа, той ще определи n като просто число — предположение, в което не можем да бъдем сигурни.

Нека пресметнем каква е вероятността p алгоритъмът да определи правилно дадено съставно число като такова, т.е. да намери делител. Първо ще припомним (*виж 1.1.3.*), че ако едно естествено число е съставно, то има поне един делител, по-малък или равен на $\sqrt{n}$. Нека с q означим броя на делителите на n, по-малки или равни на $\sqrt{n}$. Да изберем произволно естествено число от интервала [2, $\sqrt{n}$]. Вероятността то да бъде делител на n е $\frac{q}{\sqrt{n}}$, а вероятността да не бъде: $\left(1-\frac{q}{\sqrt{n}}\right)$. Да повторим опита k пъти: вероятността да не попаднем на делител при нито един от опитите е $\left(1-\frac{q}{\sqrt{n}}\right)^k$ (те са независими). Така получаваме, че вероятността алгоритъмът да определи правилно дадено съставно число като такова е $p = 1-\left(1-\frac{q}{\sqrt{n}}\right)^k$. В останалите случаи с вероятност $1-p$ можем да предполагаме, но не можем да бъдем сигурни, че n е просто.

Вероятността p се използва за характеризиране на всеки алгоритъм Монте Карло, но тя не трябва да зависи от конкретните входни данни (по-горе p зависи от броя на делителите на n). Ако последното условие е изпълнено, се казва, че алгоритъмът Монте Карло е *p-правилен*, т. е. получава правилния резултат с вероятност поне p.

Описаният *алгоритъм 1* е първата проста илюстрация на техниката Монте Карло. По-долу ще разгледаме още два (значително по-ефективни) алгоритъма за решаване на същата задача.

Първият е следствие от малката теорема на Ферма:

Теорема. **(***Ферма***)** Нека n е просто число. Тогава a^{n-1} % $n = 1$ за всяко $a = 1,2,...,n-1$.

Теоремата дава следния практически резултат: Ако за дадено естествено число n намерим естествено число a ($1 < a < n$), такова че a^{n-1} % $n \ne 1$, то следва, че n е съставно:

Алгоритъм 2:

```
for (i = 1,2,...,k) { /* k опита */
  a = random(n-1) + 1;
  if ((a^(n-1)) % n != 1) {
    return n_е_съставно;
  }
}
return n_е_просто;
```

Тук отново не може да се определи вероятност p, за която да твърдим, че *алгоритъм 2* е p-правилен. Това може да се докаже, като се използва твърдението, че за всяко $\delta > 0$ съществуват безброй много съставни числа, за които *алгоритъм 2* ще потвърди факта, че те са съставни с вероятност, по-малка от δ [*Brassard,Bratley–1996*].

Ще приведем трети алгоритъм, чиято p-правилност, за разлика от тази на предишните два, може да бъде определена. Той е един от най-ярките примери за това колко полезни и ефективни могат да бъдат алгоритмите Монте Карло. Нека n е нечетно число, $n > 4$ (ако n е четно, то очевидно е съставно).

Дефиниция 9.3. Нека е дадено естествено число n, $n > 4$. Да го представим във вида $2^s t-1$, където s и t са естествени, $s > 0$, $n > 1$, t е нечетно. Дадено е още естествено число a, $1 < a < n-1$. Числото n се нарича *строго случайно при основа a*, ако $a^t \% n = 1$, или съществува естествено число i ($0 \le i < s$) такова, че $a^{2^i t} \% n = n-1$.

Теорема. Ако при произволна фиксирана база a ($1 < a < n-1$) n не е строго случайно число, то n е съставно. В противен случай n е просто с вероятност, по-висока от 0,75.

Алгоритъм 3:

```
for (i = 1,2,...,k) { /* k опита */
  a = random(n-3) + 2;
  if (n_не_е_строго_случайно_при_база_a) {
    return n_е_съставно;
  }
}
return n_е_просто;
```

Теоремата твърди, че вероятността за намиране на правилното решение е поне 0,75, т. е. алгоритъмът е 0,75-правилен. Така, ако изпълним *алгоритъм 3* за $k = 4$ (т. е. повторим основния опит 4 пъти) ще получим или, че n е съставно, или че n е просто, с вероятност по-висока от 99%. Това превръща алгоритъма в изключително мощно средство за проверка и търсене на прости числа — с него може да се провери дали едно число, с повече от 1000 цифри е просто, като вероятността за грешка е по-ниска от 10^{-100} [*Brassard,Bratley–1996*].

За реализацията на *алгоритъм 3* трябва да покажем как ще се извършва проверката дали n е строго случайно при произволно избрана база a. Това става по следния начин:

1) Намираме числата s и t (определят се еднозначно от *дефиниция 9.3.*):

```
s = 1; t = n-1;
while (t % 2 != 1) { s++; t /= 2; }
```

2) Проверяваме дали $a^t \% n = 1$:

```
x = power(a,t) % n;
if (1 == x) return (n_е_строго_случайно);
```

3) Проверяваме дали съществува i ($0 \le i < s$), за което $a^{2^i t} \% n = n-1$:

```
for (i = 1,2,...,s-1) {
  if (n-1 == x) return (n_е_строго_случайно);
  x = x*x % n;
}
return (n_е_съставно);
```

Вижда се, че на стъпка 2) при операцията $x = a^t \% n$ може да се получи препълване, ако първо се извършва повдигането в степен и едва след това делението с остатък: a^t може да бъде много голямо число, докато резултатът x е винаги в интервала $[0, n-1]$. За да се справим с последния проблем, използваме свойствата на операцията деление с остатък:

$$a.b \% p = [(a \% p).(b \% p)] \% p ,$$

като вместо умножение, в горната формула могат да стоят и други операции (събиране, изваждане и др.). При пресмятането на a^t % n прилагаме горната формула и получаваме:

$$a^t \% n = (a^{t-1}.a) \% n = [(a^{t-1} \% n).(a \% n)] \% n$$

Така продължаваме пресмятането за a^{t-1} % n (рекурсия!), докато достигнем до

$$a^2 \% n = a.a \% n= [(a \% n).(a \% n)] \% n$$

В предложената програма това се извършва от рекурсивната функция `bigmod(a,t,n)`. Следва пълна реализация:

```
#include <stdio.h>
#include <stdlib.h>
const unsigned n = 127; /* проверява се дали n е просто */
const unsigned k = 10;  /* брой опити със случайна база a */

/* пресмята power(a,t) mod n; */
unsigned long bigmod(unsigned long a, unsigned long t, unsigned long n)
{ return (1 == t) ? (a % n) : (bigmod(a, t - 1, n) * (a % n)) % n; }

char strongRandom(unsigned long n, unsigned long a)
{ unsigned long s = 1, t = n - 1, x, i;

  /* частен случай */
  if (n < 2) return 0;
  if (2 == n) return 1;

  /* стъпка 1) */
  while (t % 2 != 1) {
    s++;
    t /= 2;
  }
  /* стъпка 2) x = power(a,t) mod n; */
  x = bigmod(a, t, n);
  if (1 == x) return 1;
  /* стъпка 3 */
  for (i = 0; i < s; i++) {
    if (x == n - 1) return 1;
    x = x * x % n;
  }
  return 0;
}

char isPrime(unsigned long n)
{ unsigned i;
  for (i = 1; i <= k; i++) {
    int a = rand() % (n - 3) + 2;
    if (!strongRandom(n, a)) return 0;
  }
  return 1;
}

int main(void) {
  printf("Числото %u е %s.\n", n, (isPrime(n)) ? "просто" : "съставно");
  return 0;
}
```

🖫 prime_mc.c

Примери за още случаи, където е подходящо да се използва алгоритъм Монте Карло или Лас Вегас, са приведени в края на главата в раздела въпроси и задачи (*9.4.*).

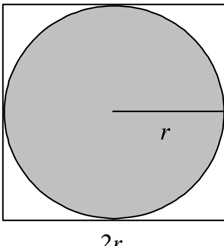
Задачи за упражнение:

1. Да се сравнят *алгоритми 1, 2* и *3*.
2. Да се реализират *алгоритми 1* и *2*.
3. Да се провери експериментално, че *алгоритъм 3* е 0,75-правилен.

9.2.2. Числени алгоритми с приближение

Числените алгоритми с приближение са най-старите алгоритми за търсене с налучкване. Макар да са известни от векове, те получават популярност по време на Втората световна война при извършването на първите ядрени тестове — по проекта "Манхатън". Всъщност, тогава те са били наричани Монте Карло, но в съвременната литература, както и в тази книга, названието Монте Карло се използва за алгоритмите, разгледани в предходния параграф.

Числените алгоритми с приближение най-често комбинират генерирането на случайни числа със статистически методи. Подобни алгоритми намират приложение при много задачи от различни области на науката — физика, химия, математика и др. Основната идея ще изясним, като разгледаме прост пример — изчисляване на константата π. Алгоритъмът, който ще покажем, е модификация на оригиналния числен метод, предложен още през 1768 *г.* от френския математик Бюфон.



Фигура 9.2.2. Вписана окръжност с радиус *r*.

Известно е, че лицето *S* на окръжност с радиус *r* може да се намери по формулата $S = \pi r^2$. Нека разгледаме окръжност с радиус *r*, вписана в квадрат със страна 2*r* (*виж фигура 9.2.2.*). Отношението на лицето на окръжността към лицето на квадрата е $\pi r^2/(2r)^2$ и така, вероятността *p* произволна точка, намираща се в квадрата да бъде същевременно в окръжността също е равна на $\pi r^2/(2r)^2$. Нека сега пресметнем тази вероятност емпирично: ще изберем *t* произволни точки в квадрата и ще проверим, колко от тях се намират и в окръжността (една точка е в окръжността, ако разстоянието от нея до центъра е по-малко от радиуса на окръжността). Полученият брой *k*, разделен на *t*, дава приближение на вероятността *p*, определена по-горе. Така, можем да получим формула за π, която се основа на случайно поставяне на точки:

$$\frac{k}{t} \approx \frac{\pi . r^2}{(2r)^2} \text{, т. е. } \pi \approx \frac{4.k.r^2}{t.r^2} = \frac{4.k}{t}$$

Вижда се, че приближената стойност на π ще зависи от отношението на броя на точките, попаднали в окръжността, към броя на проведените тестове *t*. Приближението ще бъде толкова по-близко до истинската стойност, колкото по-голямо е *t* и по-малко *r* (при по-малко *r* точността при деление ще бъде по-голяма, а при по голям брой тестове *t* — ще разполагаме с по-голяма статистическа извадка). Поради това, за радиуса *r* ще изберем максималното число, такова че r^2 да се побира в типа `long` (в противен случай може да се получи препълване при пресмятане на разстоянието от точката до центъра на окръжността).

Следва изходният код на програма, реализираща описания числен вероятностен метод за намиране на π:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>

int main(void)
{ long t = 1000000;  /* брой тестове */
  long r = 200;  /* радиус на окръжността */
  long r2 = r / 2;
  long k = 0, i;
  for (i = 0; i < t; i++) {
    long a = (rand() % r) - r2 + 1;
    long b = (rand() % r) - r2 + 1;
    if (sqrt(a * a + b * b) <= r2) k++;
  }
  printf("Приближение на p = %.2f\n", (4.0 * k) / t);
  return 0;
}
```

🖫 pi.c

Може да се пресметне, че за да се подобри прецизността при изчисление на π с 1 цифра, броят на проведените тестове трябва да се увеличи 100 пъти [*Brassard,Bratley–1996*]. Този факт води до извода, че алгоритъмът на практика е безполезен, тъй като отстъпва значително по точност на известните детерминистични алгоритми [*Келеведжиев-1994*].

Други приложения числените вероятностни алгоритми намират при пресмятане на интеграли (и съответно лица и обеми на фигури), приблизително намиране броя на комбинаторни конфигурации и др.

Задачи за упражнение:

1. Да се провери експериментално твърдението, че при по-малък радиус на окръжността се получава по-голяма точност.
2. Да се определи теоретично и да се провери експериментално колко тестове са необходими за намиране на π с точност да k-тата цифра за k = 2,3,4,5,6. Вярно ли, че, за да се подобри точността на приближението на π с 1 цифра, броят на проведените тестове трябва да се увеличи 100 пъти?

9.2.3. Генетични алгоритми

Генетичните алгоритми са метод за търсене с налучкване, при който основната идея е да се симулират генетичните и еволюционните процеси в природата. Така, за дадена оптимизационна задача, първоначално се построяват няколко *произволни неоптимални решения*. Най-сполучливите от тях се запазват и на тяхната база се построяват нови с надеждата, че те ще бъдат още по-добри. По-този начин неоптималните и безперспективни решения се изолират от по-нататъшно разглеждане. Аналогията с процеса на еволюция на видовете е очевиден — в природата този модел е известен като "оцеляване на най-доброто". Ще разгледаме как се прилага този метод в добре известната задача за търговския пътник (*виж 5.4.4.*).

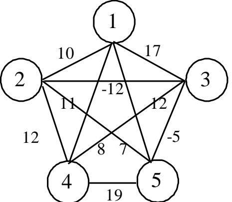
Задача: Даден е претеглен пълен граф $G(V, E)$ с тегла на ребрата реални (положителни или отрицателни) числа. Да се намери Хамилтонов цикъл с минимална дължина.

Така зададената задача е NP-пълна и се решава с пълно изчерпване — за да бъде намерен оптималният цикъл (за пълен граф с n върха), трябва да се проверят n! Хамилтонови цикъла Помислете кои от техниките за оптимизация на пълното изчерпване, разгледани в *глава 6*, могат да се приложат в този вариант на задачата, като се има предвид, че вече не е в сила ограничението теглата на ребрата да бъдат положителни числа.

Дефиниция 9.4. Множеството от кандидатите за решения, които генетичният алгоритъм обработва на всяка стъпка, се нарича *популация*.

В началото построяваме списък $P = \{H_1, H_2, ..., H_q\}$ от *q произволни* Хамилтонови цикли. Те ще бъдат началната популация и ще послужат за основа на по-нататъшната "еволюция". Хамилтоновите цикли ще запазваме като пермутация на числата от 1 до *n*, показваща реда, в който се посещават върховете. На *фигура 9.2.3.* е показан пълен граф с 5 върха. Няколко произволни Хамилтонови цикли в него (числото в скобите означава дължина на съответния цикъл) са *1-2-3-4-5* (25), *3-4-5-1-2* (49), *5-3-1-4-2* (35).

Процесът на “репродукция” при генетичните алгоритми се извършва по следния начин: на всяка стъпка се избира подмножество на текущата популация (*родители*) и (малко по-различно от стандартния начин за получаване на поколение) така избраните елементи се *комбинират* и формират нова група от елементи (*наследници*). Тъй като популацията е ограничена по размер, новополучените елементи ще заменят част от участващите в нея. За целта е необходимо да дефинираме *целева функция*, на базата на която ще сравняваме елементите и ще отсяваме “лошите”.



Фигура 9.2.3. Пълен неориентиран граф с 5 върха.

Така в задачата, която разглеждаме, на всяка стъпка ще избираме за родители половината от циклите в *P*, и то тези с най-малка дължина, след което ще ги комбинираме по двойки — така от всеки два цикъла $H_i = (i_1, i_2,..., i_n)\in P$ и $H_j = (j_1, j_2,..., j_n)\in P$ се получават нови два: избираме произволни позиции *k* и *r* и двата нови цикъла (наследници) ще бъдат: $(i_1, ..., i_{k-1}, j_k, ..., j_r, i_{r+1}, ..., i_n)$ и $(j_1, ..., j_{k-1}, i_k, ..., i_r, j_{r+1}, ..., j_n)$.

Така например, цикълът *2-3-1-4-5*, комбиниран с *5-1-3-4-2* за $k = 2$, $r = 4$, дава като резултат *2-1-3-4-5* и *5-3-1-4-2*, аналогично *1-2-3-4-5*, комбиниран с *1-3-5-4-2* за $k = 2$, $r = 3$, дава *1-3-5-4-5* и *1-2-3-4-2*.

От примера се вижда, че в някои случаи се получават повтарящи се, и съответно има липсващи, върхове. Когато се попадне на такъв случай, повтарящите се върхове се заменят с липсващите. По-нататък новополучените наследници се добавят към популацията *P*, след което от нея се изключва “лошата” половина, т. е. тези с най-голяма дължина. Така, след известен брой стъпки, се очаква алгоритъмът да построи Хамилтонов цикъл с дължина, близка до оптималната.

При работата на генетичния алгоритъм се забелязват някои негативни ефекти. Така например, с многократно прилагане на основната стъпка (комбиниране на най-добрите решения) често се стига до списък *P* от почти идентични цикли. В този случай по-нататъшна "еволюция" няма да доведе до значително подобрение. Съществуват два начина за решаване на този проблем. Единият е да се избере по-голяма начална популация (което може да бъде свързано с евентуален недостиг на памет — предполага се, че генетичният алгоритъм ще се прилага за графи с голям брой върхове, където изчерпващо решение е неприемливо). Вторият е, когато получим два абсолютно еднакви цикъла да променяме единия, като разместваме местата на два произволни съседни върха от него. Това в генетичните алгоритми се нарича *мутация* и съответства на биологичната мутация.

Реализацията на алгоритъма се състои от следните функции:

- Функция `initGraph()`, която генерира произволен граф с *n* върха (използва се за инициализиране на входните данни — вместо статично зададен граф, генерираме произволен).
- Функция `randomCycle()`, която генерира произволен Хамилтонов цикъл.

- Целева функция `evaluate()`, която оценява даден Хамилтонов цикъл (оценката е сума от теглата на участващите в него ребра).
- Функция `combine()`, която от два избрани родителя получава два нови наследника.
- Функция `mutate()`, която проверява дали има съвпадащи елементи в текущата популация и в случай, че има — променя единия от тях, като разменя два произволни върха в Хамилтоновия цикъл.
- Функцията `reproduce()` е основна функция. Тя сортира елементите на популацията и определя родителите, както и елементите, които да бъдат отстранени.

В реализацията по-долу, функцията `combine()` има сложност $\Theta(n^2)$ и се изпълнява `PSIZE`/2 пъти (`PSIZE` е размерът на популацията и е константа, независеща от n). Бихме могли да я включим в анализа на сложността, тъй като колкото по-голяма е стойността на `PSIZE` (което означава повече време и памет, памет от порядъка на $\Theta(n.\texttt{PSIZE})$), толкова по-добро решение можем да очакваме. Сложността на функцията `reproduce()` е $\Theta(n^2.\texttt{PSIZE})$. Общата сложност на програмата зависи линейно и от броя извиквания `maxSteps` на функцията `reproduce()`, т.е. от броя на поколенията, които се генерират. Следва изходният код на програмата:

```
#include <stdio.h>
#include <stdlib.h>

#define MAXN  100
/* трябва да бъде такова, че psize % 4 == 0 */
#define PSIZE 200

const unsigned long maxSteps = 1000;
const unsigned n = 20;    /* брой върхове на графа */
char A[MAXN][MAXN];       /* матрица на теглата */
int population[PSIZE][MAXN]; /* цикли на популацията */
int result[PSIZE];

/* създаване на произволен граф */
void initGraph(void)
{ unsigned i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      A[i][j] = (rand() % 100) + 1;
}

void randomCycle(int t)
{ char used[MAXN];
  unsigned i;
  for (i = 0; i < n; i++) used[i] = 0;
  for (i = 0; i < n; i++) {
    int p = (rand() % (n - i)) + 1;
    int j = 0;
    while (p > 0) {
      while (used[j]) j++;
      p--; j++;
    }
    population[t][i] = j - 1;
    used[j - 1] = 1;
  }
}

int evaluate(int t)
{ unsigned i;
  int res = 0;
```

```
  for (i = 0; i < n - 1; i++)
    res += A[population[t][i]][population[t][i + 1]];
  return res + A[population[t][n - 1]][population[t][0]];
}

/* генерира наследници q1,q2 от родителите p1,p2 */
void combine(int p1, int p2, int q1, int q2)
{ int uq1[MAXN], uq2[MAXN];
  unsigned i, j, k;
  k = (rand() % (n - 1)) + 1;  /* размяна на позиция k */
  for (i = 0; i < n; i++) {
    uq1[i] = 0;
    uq2[i] = 0;
  }

  for (i = 0; i < k; i++) {
    population[q1][i] = population[p1][i];
    uq1[population[p1][i]]++;
    population[q2][i] = population[p2][i];
    uq2[population[p2][i]]++;
  }
  for (i = k; i < n; i++) {
    if (0 == uq1[population[p2][i]]) {
      population[q1][i] = population[p2][i];
      uq1[population[p2][i]]++;
    }
    else {
      for (j = 0; uq1[j] != 0; j++);
      population[q1][i] = j;
      uq1[j]++;
    }

    if (0 == uq2[population[p1][i]]) {
      population[q2][i] = population[p1][i];
      uq2[population[p1][i]]++;
    }
    else {
      for (j = 0; uq2[j] != 0; j++);
      population[q2][i] = j;
      uq2[j]++;
    }
  }
  result[q1] = evaluate(q1);
  result[q2] = evaluate(q2);
}

void mutate(void) {
  unsigned i, k;
  /* ако се получат две поредни еднакви решения, едното "мутира" */
  for (i = 0; i < PSIZE - 1; i++) {
    int flag = 0;
    for (k = 0; k < n; k++)
      if (population[i][k] != population[i + 1][k]) {
        flag = 1;
        break;
      }
    if (!flag) {  /* цикъл i мутира */
      int p1 = rand() % n;
      int p2 = rand() % n;
```

```
      int swap = population[i][p1];
      population[i][p1] = population[i][p2];
      population[i][p2] = swap;
      result[i] = evaluate(i);
    }
  }
}

void reproduce(unsigned s)
{ unsigned i, j, k, swap;

  /* замества най-лошите цикли, като комбинира произволни
   * от първата половина
   */
  for (i = 0; i < (PSIZE-1)/2; i += 2) {
    /* randomCycle(i); */
    combine(i, i+1, PSIZE-i-1, PSIZE-i-2);
    result[i] = evaluate(i);
  }
  /* сортира популацията по оптималност */
  for (i = 0; i < PSIZE - 1; i++) {
    for (j = i + 1; j < PSIZE; j++) {
      if (result[j] < result[i]) {
        for (k = 0; k < n; k++) {
          swap = population[i][k];
          population[i][k] = population[j][k];
          population[j][k] = swap;
        }
        swap = result[i];
        result[i] = result[j];
        result[j] = swap;
      }
    }
  }

  if (maxSteps - 1 == s) return;
  mutate();
}

int main(void) {
  unsigned i, j, s;
  int minRandom;
  initGraph();

  /* Решение с генериране на произволни цикли */
  minRandom = n*101;
  for (s = 0; s < maxSteps; s++) {
    for (i = 0; i < PSIZE; i++) {
      randomCycle(i);
      result[i] = evaluate(i);
    }
    for (j = 0; j < PSIZE; j++)
      if (result[j] < minRandom)
        minRandom = result[j];
  }
  printf("Оптимално решение след произволни %ld цикъла: %d\n",
         PSIZE*maxSteps, minRandom);

  /* Решение с генетичен алгоритъм със същия брой итерации */
```

```
    for (s = 0; s < maxSteps; s++) reproduce(s);
    printf("Най-къси цикли, намерени от генетичния алгоритъм: \n");
    for (i = 10; i > 0; i--) printf("%d, ", result[i]);
    printf("%d\n", result[0]);
    return 0;
}
```

🖫 tspgenet.c

Резултат от изпълнението на програмата:

```
Оптимално решение след произволни 200000 цикъла: 487
Най-къси цикли, намерени от генетичния алгоритъм:
297, 292, 292, 292, 292, 292, 292, 292, 292, 289, 289
```

Горната програма извършва и сравнение между решението с генетичен алгоритъм и решение на принципа генериране на произволни цикли и запазване на минималния. Резултатът от изпълнението на програмата (за произволен граф с 20 върха) дава бегла представа за ползата, от разгледаната в настоящия параграф техника:

Задачи за упражнение:

1. Кои от техниките за оптимизация на пълното изчерпване, разгледани в *глава 6*, могат да се приложат за намиране на Хамилтонов цикъл с минимална дължина, като се има предвид, че вече не е в сила ограничението теглата на ребрата да бъдат положителни числа?
2. Горната програма генерира и тества само пълни графи (виж `initGraph()`). Да се модифицира така, че да работи за произволни свързани графи.
3. Да се определи експериментално кога генетичният алгоритъм е по-добър от този за генериране на произволни решения, като се отчете влиянието на различни фактори, например:
   - брой итерации
   - брой върхове на графа
   - степен на свързаност на графа

9.3. Достигане на фиксирано приближение

Всички разгледани по-горе алгоритми с приближение намираха решение, близко до оптималното. Никъде обаче не изяснихме следните въпроси:

- възможно ли е да се определи колко близко до оптималното е намереното решение и как зависи от големината на входните данни;
- възможно ли е да бъде определена *граница на оптималност*, която със сигурност ще бъде достигната след завършване на алгоритъма.

Оказва се, че съществуват задачи (и съответно алгоритми), за които отговорът на тези въпроси е положителен, и други — за които е отрицателен (или неизвестен).

Дефиниция 9.5. Нека е дадена оптимизационна задача с размер на входните данни n и алгоритъм A, който я решава. Ще казваме, че алгоритъмът я решава с *фиксирано приближение* $p(n)$, ако е изпълнено

$$\max\{\frac{C}{C'}, \frac{C'}{C}\} \leq p(n),$$

където C' е оценката на най-доброто решение, получено от алгоритъма A, а C е оценката на оптималното решение на задачата.

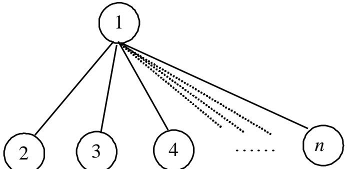
При някои алгоритми, функцията $p(n)$ е константа (тогава се пише само p), докато при други тя зависи съществено от размера n на входните данни. Ще разгледаме и двата случая на базата на една конкретна задача.

9.3.1. Върхово покритие на граф

Дефиниция 9.6. *Минимално върхово покритие* на граф $G(V, E)$ се нарича най-малкото множество от върхове D, за което е изпълнено: за всяко ребро $(i,j)\in E$ поне един от двата върха i и j принадлежи на D. Броят на елементите на D се нарича *минимално върхово число* за G.

Казано с други думи, това е минималното множество, което може да "покрие" всички *ребра* на графа (обърнете внимание, че върхово покритие е различно от разгледаните в *5.7.3. доминиращи множества* — доминиращо е множество от върхове, което покрива всички *върхове* на графа).

Намирането на минималното върхово покритие е *NP*-пълна задача и в настоящия параграф ще покажем два алчни алгоритъма за нейното решаване. Нашата по-важна цел обаче ще бъде да определим най-лошия възможен случай за всеки от тях, т. е. функцията $p(n)$.



Фигура 9.3.1а. Граф с n върха.

Алгоритъм 1:

Нека означим с D най-малкото върхово покритие, което трябва да бъде построено от алчния алгоритъм. Започваме от празното множество: $D = \varnothing$. На всяка стъпка ще добавяме към D произволен връх i от графа. След всяко добавяне се изключват върхът i, както и съседните му върхове. Алгоритъмът приключва, когато в графа не остане нито един връх.

Да разгледаме примера от *фигура 9.3.1а.* Ако в показания граф алгоритъмът избере връх *1* на първата стъпка, то решението е намерено — *1* покрива всички останали върхове и те се изключват. Какво ще се случи обаче, ако алгоритъмът избере връх, различен от връх *1*, например *2*? Ще останат n–2 изолирани върха, които алгоритъмът ще трябва последователно да избира и изключва от графа, при което за върховото число ще получи $g = n-1$, докато оптималното решение е $d = 1$. Вижда се, че ако увеличаваме n (и запазваме вида на графа), ще се увеличава и полученото от алчния алгоритъм число g. Така, очевидно не можем да фиксираме константа p, за която да твърдим, че алгоритъмът е с фиксирано приближение, т. е.

$$\max \{ g/d, d/g \} \leq p,$$

В случая е изпълнено единствено неравенството

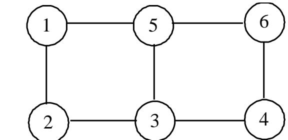
$$\max \{ g/d, d/g \} \leq n-1,$$

където $p(n) = n-1$, т.е. $p(n)$ е линейна функция на n.

Не е трудно да се съобрази как да бъде подобрена ефективността на *алгоритъм 1* така, че да не се получават абсурдни резултати като избор на n–1 върха при условие, че покритието може да се осъществи с един-единствен връх. Ще приведем втори алгоритъм, за който може да се докаже, че намереното от него решение е най-много 2 пъти по-лошо от оптималното.

Алгоритъм 2:

Нека D е върховото покритие, което ще строим (в началото $D = \varnothing$). На всяка стъпка избираме произволно ребро (i, j) от графа. Добавяме върховете i и j в D, след което изключваме всички ребра, инцидентни с i и j. Алгоритъмът приключва, когато в графа не остане нито едно ребро.



Фигура 9.3.1б. Граф с 6 върха.

За графа на *фигура 9.3.1б.* алчният алгоритъм избира последователно:

- реброто (*1*, *2*), след което се премахват ребрата (*2*, *3*) и (*1*, *5*), а към върховото покритие *D* се прибавят върховете *1* и *2*.
- реброто (*3*, *5*) — премахват се ребрата (*3*, *4*) и (*5*, *6*), а към *D* се прибавят върховете *3* и *5*.
- реброто (*4*, *6*) — не се премахват ребра, а към върховото покритие се прибавят върховете *4* и *6*. Така се получи върхово покритие {*1*, *2*, *3*, *4*, *5*, *6*}, на което съответства върхово число 6, докато съществува два пъти по-добро решение: например множеството {*2*, *5*, *4*}.

Ще покажем, че алгоритъмът е с фиксирано приближение $p = 2$:

Нека D' е минималното върхово покритие за графа, а D е върховото покритие, което строи *алгоритъм 2*. Трябва да докажем, че $\frac{|D|}{|D'|} \leq 2$. На всяка стъпка в множеството D добавяме два нови върха от графа i и j, като избираме произволно ребро (i, j). Нека множеството от всички така избрани ребра означим с S, очевидно $|D| = 2|S|$. Валидно е следното свойство: в множеството S никои две ребра не са инцидентни — това е така, тъй като след всеки избор на ребро премахваме всички инцидентни с него ребра. От дефиницията на върхово покритие следва, че всяко ребро (i, j) трябва да се покрива от връх от това множество, т. е. за всяко $(i, j) \in E$ трябва или $i \in D'$, или $j \in D'$. В частност, всяко ребро от S трябва да се покрива от връх от D', но тъй като в S няма инцидентни ребра, то никой връх от D' не е инцидентен с повече от едно ребро от S, т. е. $|S| \leq |D'|$. От $|S| \leq |D'|$ и $|D| = 2|S|$ следва, че $|D| \leq 2|D'|$, което трябваше да се докаже.

За някои *NP*-пълни задачи (например 0-1 задача за раницата) съществуват специални алгоритмични схеми с фиксирано приближение (англ. *approximation schemes*), които за всяко ε ($\varepsilon > 0$) и за произволни входни данни могат с полиномиална сложност да намерят решение с фиксирано приближение, по-малко от ε [*Brassard,Bratley–1996*].

Задачи за упражнение:

1. Да се сравнят *алгоритми 1* и *2*.
2. Да се реализират *алгоритми 1* и *2*.
3. Ще се подобри ли ефективността на *алгоритъм 1*, ако изборът на връх се прави по някакъв критерий, например избира се връх от най-висока степен, както в *алгоритъм 1* от *9.1.3.* при минимално оцветяване на граф?

9.4. Въпроси и задачи

9.4.1. Задачи от текста

Задача 9.1.

Да се докаже, че алчният алгоритъм от *9.1.* винаги работи за множеството от българските монети: 1, 2, 5, 10, 20, 50.

Задача 9.2.

Да се дадат други примери, когато алчният алгоритъм от *9.1.* ще работи винаги правилно.

Задача 9.3.

Да се докаже, че описаният в *9.1.1.* алгоритъм работи правилно.

Задача 9.4.

В реализацията от *9.1.1.* се извършва съкращаване единствено в случая, когато може да се съкрати до $1/x$. Какъв недостатък има при този подход? Да се предложи и реализира съответно подобрение.

Задача 9.5.

Да се реши оптимизационният вариант на задачата за египетските дроби (*виж 9.1.1.*), при който се търси сума с минимален брой събираеми. Съществува ли лаком алгоритъм, който винаги намира решение?

Задача 9.6.

Съществуват различни разновидности на задачата за египетските дроби [*Knott-1*]. В един широко известен вариант на задачата е наложено допълнителното ограничение всички знаменатели a_i на редицата от дроби да бъдат *нечетни* числа. Съществува ли представяне като сума от дроби с числител 1 за всяка положителна обикновена дроб при това ограничение? (*виж 9.1.1.*)

Задача 9.7.

Ще работи ли коректно реализацията от *9.1.1.*, ако се търси сума от дроби с *нечетни* знаменатели? Да се покаже, че следната модификация на алгоритъма *няма* да решава задачата винаги: Търси се минималното нечетно r, за което $r \geq q/p$. Ако `r = ceil(q/p)` е четно, r се увеличава с единица.

Задача 9.8.

Има ли смисъл задачата за египетските дроби (*виж 9.1.1.*), ако се наложи условието за четни знаменатели?

Задача 9.9.

Да се реши оптимизационен вариант на задачата за египетските дроби (*виж 9.1.1.*) при изискването за нечетни знаменатели, при който се търси сума с минимален брой събираеми. Съществува ли лаком алгоритъм, който винаги намира решение?

Задача 9.10.

Да се обърне внимание, че програмата от *9.1.2.* отпечатва номерата на лекциите според наредбата им, *след* като е приключило сортирането. Една възможност за запазване на оригиналната номерация е да се въведе допълнителен масив `prenum[]`, където в `prenum[j]` да бъде запазен началният номер на лекцията j. Така по-късно ще можем вместо j да се отпечатва `prenum[j]`. Да се направи съответна модификация на горната програма.

Задача 9.11.

Да се реализира *алгоритъм 1* от *9.1.2.* (основан на търсене на най-дълъг път в ориентиран граф).

Задача 9.12.

Да се реализира *алгоритъм 2 9.1.2.* (основан на динамично оптимиране, *виж глава 8*). Да се реализират два варианта: отгоре-надолу (рекурсивен) и отдолу-нагоре (итеративен).

Задача 9.13.

Да се докаже, че *алгоритъм 1* от *9.1.2.* работи винаги правилно.

Задача 9.14.

Да се докаже, че *алгоритъм 2* от *9.1.2.* работи винаги правилно.

Задача 9.15.

Да се докаже, че *алгоритъм 3* от *9.1.2.* работи винаги правилно.

Задача 9.16.

Да се даде пример, когато задачата за максимално съчетание на дейности (*виж 9.1.2.*) има няколко оптимални решения. Едно и също оптимално решение ли ще намират *алгоритми 1, 2* и *3* в такъв случай?

Задача 9.17.

В предложените реализации на двата алгоритъма от *9.1.3.* за намиране на оцветяване на произволен граф върховете се разглеждат последователно от *1* до *n*. В примерния изход резултатът от двата алгоритъма съвпада. Ще съвпада ли намереното оцветяване и за произволен друг граф? Да се намери пример, при който намерените оцветявания са различни, или да се докаже, че те винаги съвпадат и да се обясни разликата между алгоритмите и реализацията им.

Задача 9.18.

Да се модифицира реализацията на *алгоритъм 1* от *9.1.3.* така, че върховете да се разглеждат по реда на намаляване на степента им. Да се сравни резултатът (броят цветове в минималното оцветяване при произволно генерирани графи) с оригиналния вариант. Постига ли се подобрение?

Задача 9.19.

Да се определят сложностите на *алгоритми 1* и *2* от *9.1.3.*

Задача 9.20.

Да се покаже, че всяка гора е двуделен граф (*виж 9.1.3.*).

Задача 9.21.

На базата на разсъждения от *9.1.4.* да "се сглобят" по-строги доказателства на правилността на работата на алгоритмите на Прим и Крускал.

Задача 9.22.

Да се дадат още примери, когато алчният алгоритъм от *9.1.5.* работи и когато не работи за 0-1 задачата.

Задача 9.23.

Необходим ли е масивът `ratio[]` в реализацията от *9.1.5.*? Да се реализира подходяща функция за сортиране на рационални дроби и да се направят съответните модификации във функцията `solve()`. Упътване: вместо *a*/*b* и *c*/*d* може да се сравняват *ad* и *bc*.

Задача 9.24.

Да се докаже коректността на предложения в *9.1.6.* алгоритъм за решаване на задачата за магнитната лента.

Задача 9.25.

Да се реализира алгоритъмът от *9.1.6.* за решаване на задачата за магнитната лента.

Задача 9.26.

Да се докаже, че описаният в *9.1.7.* алгоритъм за намиране на оптимално процесорно разписание работи правилно.

Задача 9.27.

Вижда се, че програма от *9.1.7.* не отпечатва задачите в реда, в който трябва да се изпълнят, а определя единствено *множеството* от задачи, даващо оптималното решение. Да се предложи и реализира алгоритъм, който да намира *реда* на изпълнение на задачите:

а) по време на построяване на решението (в цикъла *for* на функцията `solve()`)

б) след като оптималното решение е вече определено

Задача 9.28.

В предложената в *9.1.8.* реализация на разходката на коня сложността на алгоритъма е $\Theta(m^2.n^2)$, където m е броят на възможните ходове на коня (в програмата по-горе — константата `MAX_MOVES`): Извършват се n^2 стъпки, на всяка от които има m извиквания на `countMoves()`, която от своя страна също има сложност $\Theta(m)$. Как, като се използва допълнително n^2 памет, сложността може да се редуцира до $\Theta(n^2)$?

Задача 9.29.

Ще работи ли правилно алгоритъмът за обхождане с ход на коня, ако стартовото поле е в долния ляв ъгъл, както в *6.3.4.*? А при произволно зададено поле от дъската?

Задача 9.30.

Да се докаже коректността на алгоритъма за построяване на кода на Грей от *9.1.8.*

Задача 9.31.

Да се приведат още примери, при които вероятностният алгоритъм превъзхожда систематичното изследване (*виж 9.2.*).

Задача 9.32.

Да се сравнят *алгоритми 1, 2* и *3* от *9.2.1.*

Задача 9.33.

Да се реализират *алгоритми 1* и *2* от *9.2.1.*

Задача 9.34.

Да се провери експериментално, че *алгоритъм 3* от *9.2.1.* е 0,75-правилен.

Задача 9.35.

Да се провери експериментално твърдението от *9.2.2.*, че при по-малък радиус на окръжността се получава по-голяма точност.

Задача 9.36.

Да се определи теоретично и да се провери експериментално колко тестове по метода на Бюфон (*виж 9.2.2.*) са необходими за намиране на π с точност да k-тата цифра за k = 2,3,4,5,6. Вярно ли, че, за да се подобри точността на приближението на π с 1 цифра, броят на проведените тестове трябва да се увеличи 100 пъти?

Задача 9.37.

Кои от техниките за оптимизация на пълното изчерпване, разгледани в *глава 6*, могат да се приложат за намиране на Хамилтонов цикъл с минимална дължина, като се има предвид, че вече не е в сила ограничението теглата на ребрата да бъдат положителни числа (*виж 9.2.3.*)?

Задача 9.38.

Програмата от *9.2.3.* генерира и тества само пълни графи (виж `initGraph()`). Да се модифицира така, че да работи за произволни свързани графи.

Задача 9.39.

Да се определи експериментално кога генетичният алгоритъм (*виж 9.2.3.*) е по-добър от този за генериране на произволни решения, като се отчете влиянието на различни фактори като:

- брой итерации
- брой върхове на графа
- степен на свързаност на графа

Задача 9.40.

Да се сравнят *алгоритми 1* и *2* от *9.3.1.*

Задача 9.41.
Да се реализират *алгоритми 1* и *2* от *9.3.1*.

Задача 9.42.
Ще се подобри ли ефективността на *алгоритъм 1* от *9.3.1.*, ако изборът на връх се прави по някакъв критерий, например избира се връх от най-висока степен, както в *алгоритъм 1* от *9.1.3.* при минимално оцветяване на граф?

9.4.2. Други задачи

Задача 9.43. *Получаване на сума при хиперрастяща редица*
Да се състави алгоритъм за получаване на дадена сума N, като се използват минимален брой банкноти с дадени номинали $C = \{a_1, a_2, ..., a_n\}$, за които още е изпълнено:

$$a_i \geq 2.a_{i-1}, \text{ за } i = 2, ..., n, \text{ и } a_1 = 1.$$

т. е. редицата от стойностите е хиперрастяща.
Да се покаже чрез подходящ контрапример, че алчният алгоритъм, разгледан в увода на главата, няма да работи коректно при произволни входни данни.

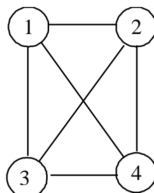
Задача 9.44. *Получаване на сума при геометрична прогресия*
Да се покаже, че алчният алгоритъм за получаване на сума с минимален брой банкноти ще работи винаги правилно, ако номиналите на банкнотите са от вида $\{1, c, c^2, c^3, ..., c^k\}$, $c \geq 2$, $k \geq 0$.

Задача 9.45. *Обобщен ход на коня*
Ще работи ли правилно алчният алгоритъм за намиране на път с ход на коня (*виж 9.1.8.*), ако константите в `moveX` и `moveY` са различни, т. е. ако разглеждаме фигура, подобна на шахматен кон, но извършваща различни Г-образни движения?

Задача 9.46. *Покритие с интервали*
Дадени са n точки върху реалната права. Да се предложи ефикасен алчен алгоритъм, който определя минималния брой интервали с дължина 1 върху правата, които покриват всички точки.
Решение: Нека p_l е най-лявата от всички точки (т. е. точката с най-малка x координата). Трябва да има поне един интервал, който да съдържа тази точка. Очевидно, интервалът трябва да бъде разположен така, че p_l да съвпада с левия му край: ако точката е вътрешна, то частта от интервала, намираща се вляво от нея, ще бъде излишна. След като разположим еднозначно първия интервал, изключваме точките, които той съдържа, и решаваме задачата за останалите (непокрити) точки.

Задача 9.47. *Обобщена задача за магнитната лента*
Дадени са k магнитни ленти, върху които трябва да бъдат записани n програми. За всяка програма i са известни нейната дължина и честота на изпълнение. Да се състави алчен алгоритъм, който определя как да бъдат записани всички програми върху лентите, така че средното време за достъп да бъде минимално.

Задача 9.48. *Минимално оцветяване на граф при дадени допустими цветове за всеки връх*
При тази модификация на задачата за минимално оцветяване на граф за всеки връх е дадено множество от цветове, с които може да бъде оцветен. Отново се търси оцветяване на графа с минимален брой цветове.



Фигура 9.4.2а. Граф с 4 върха.

Възможно е задачата да няма решение. Пример, при който оцветяване е невъзможно, е показан на *фигура 9.4.2а.*: допустимите стойности (номера на цветове), с които могат да бъдат оцветени върховете му, са следните: връх *1*: {*1*}; връх *2*: {*1*, *2*, *3*, *4*}; връх *3*: {*1*, *4*}; връх 4: {*1*, *4*}.

Проверката, дали задачата има решение, се извършва с полиномиална сложност.

Упътване: да се сведе до задача за проверка за съвършено двойкосъчетание — *виж глава 5*, в подходящо построен двуделен граф.

Да се състави алчен алгоритъм, който извършва минималното оцветяване.

Задача 9.49. *Максимално съчетание на дейности от две групи*

Да се реши следната модифицирана задача за максимално съчетание на дейности: Дадени са два вида дейности — например лекции по математика и лекции по информатика. Всяка лекция се определя от две числа: начало s_i и край f_i. Да се състави разписание за провеждане на максимален брой лекции така, че да бъдат изпълнени следните две условия:

- Във всеки един момент не може да се преподава повече от една лекция.
- Лекциите, които са избрани за преподаване, трябва да се редуват: лекция по математика (от първата група), следва лекция по информатика (втората група), следва лекция по математика и т.н.

Задача 9.50. *Проверка на умножение на матрици*

Дадени са три матрици A', A'' и A. Да се състави вероятностен алгоритъм от тип Монте Карло, който проверява дали матрицата A е резултат от умножението на A' с A''.

Задача 9.51. *Обратна матрица*

Дадени са две матрици A', A''. Да се състави алгоритъм от тип Монте Карло за проверка дали A' е обратна матрица на A''. Вижда се, че това е частен случай на горната задача.

Задача 9.52. *Троично булево дърво*

Дадено е троично кореново дърво (всеки връх, освен листата, има точно 3 наследника) с височина h, т. е. всяко листо се намира на разстояние h от корена. С всеки връх е асоциирана по една булева променлива по следния начин:

- с всяко листо е асоциирана произволна булева променлива.
- с всеки друг връх е асоциирана променливата, която се среща най-често в наследниците на съответния връх.

По дадени стойности на листата да се намери стойността на променливата, асоциирана с корена на дървото. Да се покаже, че за всеки детерминистичен алгоритъм, решаващ винаги коректно задачата, съществува пример, при който алгоритъмът трябва да прочете всичките 3^h листа на дървото. Да се състави вероятностен алгоритъм за решаване на задачата.

Задача 9.53. *8 царици*

Да се реши задачата за осемте царици (*виж 6.3.5.*), като се използва вероятностен алгоритъм от тип Лас Вегас. Да се сравни поведението на предложения от Вас алгоритъм с разгледаната в *6.3.5.* детерминистична схема за изчерпване на решенията.

Задача 9.54. *Минимално покриващо дърво от дадени k върха*

Даден е свързан неориентиран претеглен граф $G(V, E)$ и подмножество H от k негови върхове. Да се намери такова множество от ребра, че всеки два от участващите в H върхове да бъдат свързани с път и общата сума от теглата на избраните ребра да бъде минимална.

Да се предложи подходящ евристичен алгоритъм за решаване на задачата.

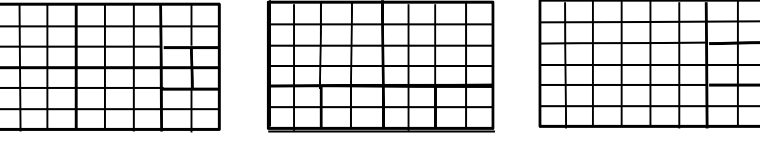
Задача 9.55. *Последователни лекции*

Дадени са n лекции (събития), всяко от които е с продължителност l_i (цяло положително число). Лекциите се провеждат една след друга (т. е. редът им не може да се променя) — лекция $i+1$ ($i = 1,2, ..., n-1$) започва веднага, след като завърши лекция i. Всеки ден залата за лекции е свободна по L часа (дадено цяло число). Какъв е минималният брой дни, за които могат да се изнесат всички лекции (лекциите не могат да се разкъсват — една лекция или трябва да бъде изнесена цялата в деня, когато е започната, или изобщо да не бъде започвана).

Задача 9.56 *Запълване с квадрати*

Даден е правоъгълник с размер $n \times m$. Правоъгълникът е разграфен на квадратчета с размери 1×1. Разполагаме с достатъчен брой квадрати със стандартни размери 1×1, 2×2, 3×3, 4×4, 5×5 и 6×6. Да се определи конфигурация от минимален брой квадрати, които покриват изцяло и без застъпване дадения правоъгълник.

В примера от *фигура 9.4.2б.* оптималният случай, който използва само 4 фигури за покритие, е в).



а) покритие с 10 фигури б) покритие с 6 фигури в) покритие с 4 фигури

Фигура 9.4.2б. Покрития на правоъгълник с размер 8×6.

Задача 9.57. *Покриващо дърво на граф с минимална височина*

Даден е граф $G(V, E)$. Да се намери покриващо дърво на G с минимална височина.

Упътване: Един възможен алчен алгоритъм, който не винаги намира минималното решение е следният: за корен се избира върхът от най-голяма степен [*Манев-1996*]. Така второто ниво от дървото е определено — наследниците на корена в графа. Третото ниво са наследниците на техните наследници, като върховете се разглеждат в намаляващ (по степен на върха) ред и т.н. Идеята е да се минимизира височината, като на всяко ниво се добавят възможно повече върхове.

- Да се намери пример, за който показаният алгоритъм няма да работи правилно.
- Да се пресметне фиксираното приближение $p(n)$, с което предложеният алгоритъм решава задачата.
- Да се предложи алгоритъм, който решава задачата с фиксирано приближение 2.

Задача 9.58. *Избори* [*Timus, задача 1025*]

В Америка, преди да се установи демокрация, изборите се провеждали по следния начин: всеки щат носел по 1 глас, независимо колко души е населението му. Борбата по същество била между 2 партии — републиканци и демократи, като мнозинство печелела тази партия, която спечели изборите в повече щати.

Задача: Нека n е броят на съединените американски щати, а m_i ($1 \leq i \leq n$) е населението на i-тия щат (в милиони). Да се намери минималният брой хора, които трябва да гласуват за демократите, така че те да спечелят изборите.

Пример: Ако щатите са 3, а населението е съответно 5, 7 и 5 милиона, то достатъчно е 6 милиона да гласуват за демократите — по 3 в 1-ви и 3-ти щат. Така победата ще бъде извоювана с 2:1 щата (което всъщност е крайно недемократично, тъй като реалните гласове са 6:11 милиона).

Решение: Очевидно (обратно на здравия разум), борбата ще се води за щатите с най-малко население, тъй като всеки щат участва с точно един глас и с техните гласове победата ще бъде спечелена най-лесно. Алчният алгоритъм, който решава задачата, е следният:

- сортираме щатите по тяхното население в нарастващ ред: $m_1, m_2, \ldots, m_n$.
- минималният брой необходими гласове се дава от сумата $\sum_{i=1}^{\frac{n}{2}+1}\left(\frac{m_i}{2}+1\right)$.

Глава 10
Компресиране

"Какъвто и капацитет да има един твърд диск, той винаги е пълен..."

~ Програмистки фолклор

10.1. Кодиране

През последните години беше постигнат наистина забележителен напредък в областта на технологиите за запис и съхранение на данни. Само допреди десетина години човек можеше да побере на 2-3 дискети (по 180-360 *KB*) целия необходим му за работа софтуер, а капацитетът на тогавашните твърди дискове — 10-20 *MB*, се считаше за астрономически. Наистина, сблъсквайки се за пръв път с подобни размери свободно дисково пространство, обикновеният потребител трудно можеше да си представи разумен начин за оползотворяването му. По-късно нещата се промениха, капацитетът на запаметяващите устройства започна да нараства, а заедно с това увеличиха и изискванията на използвания софтуер, като последствията от този процес днес са особено осезаеми. Всъщност, сега се наблюдава обратната зависимост: изискванията на софтуера често изпреварват възможностите на хардуера.

Успоредно с усъвършенстването на хардуерните технологии започна развитие и на съответни софтуерни решения, позволяващи по-ефективно използване на наличните ресурси на системата, с цел да се задоволят все по-нарастващите нужди от свободно дисково пространство. Появиха се компресиращи програми (*COMPRESS*, *PKZIP*, *ARJ*, *AIN*, *RAR*, *WinZIP* и др.), които позволяваха данните да се преобразуват по подходящ начин така, че да се намали необходимото дисково пространство. Направени бяха и някои повече или по-малко успешни опити за компресиране и декомпресиране в реално време като *Stacker*, *Double Space*, *Drive Space* и др. Успоредно с това бяха положени сериозни усилия за разработване и утвърждаване на технологии за компресиране на графика (*GIF*, *JPEG*, *TIFF*), звук (*WAV*, *MP3*) и видео (*MPEG*, *AVI, DivX*). По-късно, с настъплението на *Интернет*, започна да се чувства все по-осезателната нужда от кодиране не само с цел съкращаване размера на предаваните по мрежата данни. Мрежата наложи свои специфични стандарти и протоколи, повечето от които бяха свързани с някакъв особен вид кодиране (например *7-битово* кодиране). При трансфера на данни от изключително значение се оказаха някои пренебрегвани до момента от масовия потребител кодирания като *шумозащитното* кодиране, поставящо си за цел подобряване качеството на трансфер на данни, както и *криптирането*, целящо защита на данните от непозволен достъп (защита с парола, електронен подпис и др.). Кодирането на данни започна да придобива все по-съществено значение за компютърния свят.

И все пак откъде тръгнаха нещата? Би било погрешно да се смята, че кодирането и нуждата от него възниква в резултат на компютърния бум. Напротив, корените му, най-вече в смисъл на криптография, се крият в най-дълбока Древност. Днес то е силно развита математическа наука на границата между дискретната математика и теоретичната информатика с три основни клона: криптография, компресиране на данни и шумозащитно кодиране. Кодирането е изключително обширна област, поради което няма да го разглеждаме изцяло и до края на настоящата глава ще се ограничим до алгоритмите за *компресиране*. Оттук нататък ще употребяваме *кодиране* (*декодиране*) в по-тесния смисъл на синоним на *компресиране* (*декомпресиране*).

Следва да се отбележи, че алгоритмите за компресиране са по-особени. Действително, основният параметър, от който човек се ръководи при разработването на даден алгоритъм, е минимизиране на необходимото *време* за работа, като размерът на използвана памет обикновено е от

второстепенно значение. Тук нещата стоят точно обратното: целта е преди всичко намаляване *размера* на данните, като времето за постигане на това често не е от особено значение.

Очевидно за произволен компресиращ алгоритъм ще съществуват входни данни (по-долу ще ги наричаме *съобщения*), чиято дължина при компресиране ще се запазва или дори ще нараства. Действително, нека допуснем, че съществува компресиращ алгоритъм, при който дължината на *всяко* входно съобщение строго намалява. Да компресираме произволно съобщение, след което да компресираме отново получения резултат, него също да го компресираме и т. н. Ясно е, че в един момент ще достигнем до празната последователност, което със сигурност ще ни доведе до невъзможност за извършване на обратния процес на декомпресиране.

10.2. Обща класификация

Съществуват множество съществени характеристики, въз основа на които могат да се приведат различни класификации на методите за компресиране на данни. Така например, една възможна класификация е в зависимост от това дали при декомпресирането се получава резултат, идентичен с входните данни преди компресирането. В зависимост от това методите се делят на такива *с* и *без загуба на данни*. Повечето данни, произтичащи от реалния свят, са неточни, т. е. съдържат някакво определено ниво на шум. Прибавянето на малко допълнителен шум обикновено не води до трагични последствия, но може да позволи значително намаляване на размера на компресираните данни. В сила е принципът: колкото повече допълнителен шум добавяме, толкова по-висока степен на компресия постигаме, като съответното ниво следва да се определя в зависимост от конкретните нужди и изисквания. Най-важните представители на компресирането със загуба принадлежат към класа на така наречените *вълнови* методи. Типични примери за данни, при които широко се прилага компресиране със загуба на точност, са: графичните изображения, музикалните и видеофайловете и др. Макар възможността за компресиране със загуба да е примамлива, предвид на по-високата степен на постигана компресия, това не винаги е допустимо: например при изпълними файлове, текстови документи, електронни таблици и др.

Могат да се приведат редица други класификации, основани на различни признаци. Няма да се спираме подробно на всяка една от тях, а ще разгледаме само една, съгласно която съществуват най-общо 6 основни типа компресиране:

Морзов код					
буква	**код**	**буква**	**код**	**буква**	**код**
A	● –	**M**	– –	**Y**	– ● – –
B	– ● ● ●	**N**	– ●	**Z**	– – ● ●
C	– ● – ●	**O**	– – –	**1**	● – – – –
D	– ● ●	**P**	● – – ●	**2**	● ● – –
E	●	**Q**	– – ● –	**3**	● ● ● – –
F	● ● – ●	**R**	● – ●	**4**	● ● ● –
G	– – ●	**S**	● ● ●	**5**	● ● ● ● ●
H	● ● ● ●	**T**	–	**6**	– ● ● ● ●
I	● ●	**U**	● ● –	**7**	– – ● ● ●
J	● – – –	**V**	● ● ● –	**8**	– – – ● ●
K	– ● –	**W**	● – –	**9**	– – – – ●
L	● – ● ●	**X**	– ● ● –	**0**	– – – – –

Таблица 10.2. Морзов код на английската азбука и цифрите.

Кодиране на последователности

Това е прост метод, при който последователности от повтарящи се букви се кодират чрез двойка, съставена от буквата и броя на последователните ѝ появи. Методът може да бъде особено ефективен при черно-бели графични изображения, факсове и др. По-долу ще разгледаме редица негови разновидности (*виж 10.3.*).

Статистически (вероятностни) методи

Основават се на статистически наблюдения за вероятността на поява на буквите (по-рядко групи от букви) във входното съобщение. Идеята е, че ако са известни честотите на срещане за всяка от буквите, бихме могли да ги кодираме с някакъв неравномерен код (най-общо съпоставящ на различните букви кодове с различна дължина), съпоставяйки на най-често срещаните букви по-кратки битови последователности. Типичен представител на този вид кодиране е добре известният Морзов код, съставен от два символа: "•" и "–". В английския му вариант букви като "*e*" и "*t*" се кодират с единствен символ (*виж таблица 10.2.*), тъй като са сред най-често срещаните. Други важни представители, на които ще обърнем особено внимание, са кодовете на Шенън-Фано (*виж 10.4.1.*) и Хъфман (*виж 10.4.2.*), аритметичното кодиране (*виж 10.4.5.*) и др.

Речникови методи

За разлика от статистическите, речниковите методи извършват кодиране не на отделни букви (тук *буква* означава най-често *един символ*, но може да бъде и *последователност* от символи), а на цели думи, изречения и произволни последователности. За целта се съставя таблица (речник) на съответствията. Всяка дума се кодира с индекса си в речника, а в случай, че липсва там, се предава без изменение. Речниковото кодиране може да бъде изключително ефективно при кодиране на текстови файлове, като в някои случаи размерът на входното съобщение може да се намали няколко пъти. Най-известните методи от този клас са *LZ77*, *LZ78* (*виж 10.6.5.*) и *LZW* (*виж 10.6.8.*).

Вълнови методи

Използват се при компресиране на звук, графични и видеоизображения. Входната последователност се описва с помощта на набор от вълни (функции) въз основа на някакви нейни характеристики. Постига се много висока степен на компресия, за сметка на невъзможност за 100% точно декомпресиране. Все пак при декомпресиране се получава резултат, достатъчно близък до оригинала, което за такива типове файлове е допустимо. Най-известните представители на вълновите методи са *JPEG* (графични изображения, *виж 10.7.2.*), *MP3* (звук) и *MPEG* (видео, *виж 10.7.3.*).

Фрактални методи

Тук целта отново е намиране на формула, описваща данните. Фракталните функции позволяват представяне на сложни данни чрез прости функции, при което се постига огромна степен на компресия. За съжаление, намирането на съответна формула е изключително трудно (*виж 10.7.5.*).

Адаптивно компресиране

Адаптивното компресиране позволява самонастройване на кодиращия алгоритъм въз основа на локалните характеристики на кодираната последователност така, че максимално да се възползва от тях. Повечето от широкоразпространените методи за компресиране имат адаптивен вариант. (*виж 10.5.*)

10.3. Кодиране на последователности

10.3.1. Премахване на нулите

Премахването на нулите (англ. *null suppression*) е една от най-ранните техники на компресиране. Днес тя има ограничено приложение, главно при комуникационния протокол *IBM* 3780 и някои текстови редактори.

Основната идея на метода е дългите последователности от нули да се заменят с кодираща двойка, състояща се от специален символ, указващ наличието на компресия, последван (или предшестван) от брояч, указващ броя последователни нули. Процесът на декодиране е тривиален и се свежда до просто заместване на кодиращата двойка със съответния брой нули. Най-често за брояча се отделя един байт. По този начин могат да се кодират последователности с дължини 0, 1, ..., 255. Ясно е, че 0 е безсмислена като стойност на брояча. Ето защо бихме могли да изместим с 1 наляво интервала от стойности, получавайки: 1, 2, ..., 256. Да разгледаме един пример:

12 17 86 93 **0 0** 1 2 **0 0 0 0 0** 19 20 **0** 8 3 12 **0 0 0** 6

След кодиране получаваме:

12 17 86 93 **0 1** 1 2 **0 4** 19 20 **0 0** 8 3 12 **0 2** 6

Понякога броят на последователните нули може значително да надвишава 256. Едно възможно решение в такъв случай е да се използват повече от една кодиращи двойки за една и съща последователност. Така например, последователност от 1000 нули ще се кодира по начина, показан в *таблица 10.3.1а*.

първа двойка		**втора двойка**		**трета двойка**		**четвърта двойка**	
специален символ	*255*	*специален символ*	*255*	*специален символ*	*255*	*специален символ*	*231*

Таблица 10.3.1а. Кодиране на последователност от 1000 нули.

В случай на по-дълги последователности от нули може да се окаже по-ефективно за брояча да се отдели повече от един байт. Това води до премахване на ненужното повторение на специалния символ в случаи, подобни на горния, но пък увеличава пространството, заемано от брояча. Ето защо размерът на брояча следва да се избира много внимателно.

В случай, че специалният символ и броячът заемат по един байт, се оказва неефективно кодирането на единични нули. Действително, в този случай това води не до *намаляване*, а до *увеличаване* размера на кода. Кодирането на последователности от две нули очевидно също не ни дава нищо, тъй като дължината се запазва. Изобщо, кодирането следва да се прилага само към последователности от поне два(три) символа. С увеличаване дължината на брояча ще нараства и общата дължина на кодиращата двойка, а оттук и на последователностите, които няма да се кодират.

Основен проблем при премахването на нулите е изборът на специален символ. Обикновено това е символ, за който може да се гарантира със сигурност, че не се съдържа във входното съобщение. В случай, че текущата кодова таблица не съдържа такъв символ, тя следва да се разшири по подходящ начин. Едно възможно решение е кодиращата двойка да се загради с *двойка* допълнителни редки символи *SI* (от англ. *shift in*) и *SO* (от англ. *shift out*), *виж таблица 10.3.1б*.

SI	*специален символ*	*брояч*	*SO*

Таблица 10.3.1б. Заграждане на кодиращата двойка с друга двойка редки символи.

Така ще могат да се кодират ефективно последователности от поне 4(5) нули. По-късно, при кодирането на последователности (*виж 10.3.9.*), ще бъде предложен друг подход, при който се използва само един допълнителен символ, който ще наричаме *escape*. По този начин се получава кодираща *тройка*.

Каква е ползата от прилагането на описания метод? Това силно зависи от съдържанието на входното съобщение. То ще се компресира добре, само ако съдържа достатъчно дълги последователности от нули. Понякога ползата може да се окаже наистина голяма. В протокола за предаване на данни *IBM* 3780 *BISYNC*, основан на този метод, се постига компресия от порядъка на 30-50%. [*Held-1991*]

Възможни са различни обобщения на метода. Така например, обикновените текстови файлове не съдържат последователни нули, поради което премахването на нулите не носи нищо. Оказва се обаче, че те често съдържат достатъчно дълги последователности от *интервали*. Така, кодирайки интервалите, понякога можем да получим значителни съкращения. Изобщо, при избор на подходящ следва да се отчитат типът и особеностите на кодираната последователност.

В случай, че кодовата таблица съдържа достатъчно свободни (несрещащи се във входното съобщение) символи, биха могли да се резервират един или повече от тях за специални нужди. Някои текстови редактори използват специален символ *tab* (с код 9 в таблицата *ASCII* — от англ. *American Standard Code for Information Interchange*) за кодиране на последователности от 8 интервала. Биха могли да се използват и няколко различни символа едновременно. Така например, един символ би могъл да кодира 5 интервала, друг — 10, а трети — 20, при което да се постигне максимална гъвкавост на кодирането.

Забележка: Следва да отбележим, че символът *tab* има по-специално предназначение. Често при текстови файлове текстът се отмества надясно, за да се оформи таблица. В този случай обикновено текстовите редактори използват символ *tab*, указващ определена позиция на отместване спрямо началото на текущия ред, а не спрямо края на предишната дума. Да предположим, че колоните на таблиците имат ширина 10 символа. Тогава поставянето на един символ *tab* ще означава позициониране във втората колона на таблицата, т. е. на позиция 10, считано от началото на реда. Втора табулация на същия ред, ще означава позициониране в третата колона, т. е. на позиция 20 и т.н. Един символ *tab* всъщност ще може да означава относителни отмествания: 0,1,...,9. Обикновено колоните на таблиците имат различна ширина. В такъв случай някои текстови редактори използват специално предварително описание на таблицата, в което се задава абсолютното отместване спрямо началото за всяка колона поотделно.

Задачи за упражнение:

1. Да се определи максималната степен на компресия, която може да постигне методът.
2. Да се предложи начин за кодиране на символа *escape* — макар и рядък, все пак може да се срещне.
3. Да се предложи начин за кодиране на символите от кодиращата двойка *SI* и *SO*.
4. Да се реализира описаният алгоритъм и да се изпробва върху битова карта на черно-бяло изображение.

10.3.2. Компресиране с битови карти

В случай, че някой от символите от входното съобщение се среща доста по-често от останалите (например интервал, нула и др.) е удобно използването на *битови карти*. Битовата карта представлява последователност от 0 и 1, всеки бит на която показва присъствието (стойност 1) или отсъствието (стойност 0) на въпросния символ. *Кодирането с битови карти* (англ. *bit mapping*) може да се разглежда като обобщение на премахването на нулите. Действително, тук от значение е само *вероятността* на срещане на символа, без значение дали срещанията са в дълги последователности или — не. Ще покажем как с помощта на битова карта може да се реализира премахване на нулите. Да разгледаме последователността:

```
0 13 0 0 89 0 37 0
```

Вижда се, че 0 е доминиращ символ и се среща с вероятност 5/8. Същевременно, компресирането чрез премахване на нулите не ни носи нищо, тъй като най-дългата последователност от нули е с дължина 2. Използването на битова маска обаче води до значително съкращаване дължината на съобщението. В горното съобщение ненулеви елементи имаме на позиции 2, 5 и 7. Така за съответната битова маска получаваме: 01001010. За да получим кодираното съобщение, след битовата карта (тя има стойност 74_{10} в десетична бройна система) трябва да запишем ненулевите символи по реда на появата им:

```
74 13 89 37
```

Постигнахме компресия от 50%, намалявайки броя на необходимите байтове от 8 на 4 (дължината на входното съобщение нарочно беше избрана 8, за да може битовата маска да се побере в 1 байт.). Ефективността на кодирането се основава на високата вероятност за срещане на нулата. Какво става, ако нулата се среща по-рядко? *Таблица 10.3.2.* ни дава представа за това как степента на компресия зависи от честотата на срещане на нулата в последователност от 8 символа. Забележете, че при по-малко от 12,5% нули дължината на кодираното съобщение нараства, вместо да намалее.

брой нули	0	1	2	3	4	5	6	7	8
% нули	0	12,5	25	37,5	50	62,5	75	87,5	100
дължина на кода в байтове	9	8	7	6	5	4	3	2	1
степен на компресия	0,888	1,000	1,143	1,334	1,600	2,000	2,667	4,000	8,000

Таблица 10.3.2. Дължина на кода и степен на компресия при различен брой нули.

Не е трудно да получим конкретен математически израз за степента на компресия като функция на вероятността за срещане на символа. Да означим с p вероятността за поява на нула във входното съобщение, а с n — дължината му. Тогава броят на нулите във входната последователност ще бъде np. Директно се пресмята дължината на кодираното съобщение $n(1-p) + \lceil n/8 \rceil$: извадили сме броя на нулите np и сме прибавили броя на байтовете $\lceil n/8 \rceil$, заемани от битовите карти. (Тук $\lceil n/8 \rceil$ означава горна цяла част на $n/8$). За степента на компресия получаваме отношението:

$$\text{степен на компресия} = \frac{\text{дължина на входното съобщение}}{\text{дължина на кодираното съобщение}} = \frac{n}{n(1-p)+\left\lceil \frac{n}{8} \right\rceil}$$

Да проверим доколко получената формула отговаря на *таблица 10.3.2.* Да вземем входно съобщение с дължина $n = 8$, съдържащо 6 нули. Вероятността за срещане на нула се пресмята като: $p = 6/8 = 0{,}75$. За дължината на кодираното съобщение имаме: $n(1-p) + \lceil n/8 \rceil = 8(1-0{,}75) + \lceil 8/8 \rceil = 3$. Оттук пресмятаме степента на компресия: $8/3 = 2{,}667$.

Основен недостатък на битовите карти е, че са приложими само върху последователности с фиксирана дължина в битове: символи, байтове, думи и др. Директно следствие от това е, че дължината на кодираното съобщение е *обратнопропорционална* на вероятността за срещане на нула. Впрочем, това се вижда добре и от горната формула. В случай, че имаме няколко често срещащи се символи с близки честоти на срещане, битовата карта позволява съкращаване на кода, пропорционално само на *едната* честота и по никакъв начин не отчита високата вероятност за срещане на другия символ. Такава ситуация се среща достатъчно често, например при съобщения, съдържащи числа. Решение на проблема дава кодирането на последователности (*виж 10.3.9.*), което ще разгледаме по-късно.

Как става декодирането? Прочита се битовата карта, след което се прочитат толкова символа, колкото е броят на единиците в нея. Между символите евентуално се вмъкват нули, съобразно данните от битовата карта. След това се прочита нова битова карта и т.н. Процесът продължава до цялостното декодиране на съобщението.

Задачи за упражнение:

1. Да се реализира предложеният алгоритъм.
2. За какъв тип файлове е подходящо компресирането с битови карти?
3. Следва ли да се очаква намаляване на дължината на стандартен текстов файл (приемаме интервала за 0)? А на битова карта на черно-бяло изображение?
4. Възможно ли е да се използва успешно методът, ако някои части от файла съдържат голям процент нули, но, погледнато глобално, честотата им е недостатъчна?
5. Да се сравнят компресирането с битови карти и премахването на нулите (*виж 10.3.1.*). Кое от тях е за предпочитане? Защо?

10.3.3. Полубайтово пакетиране

Полубайтовото пакетиране (англ. *halfbyte packing*) се основава на структурата на кода на някои от символите в кодовата таблица. При някои кодови таблици част от кода на цял набор от важни символи се повтаря. Така например, в *EBCDIC* (от англ. *Extended Binary-Coded Decimal Interchange Code*) старшите четири бита в представянето на цифрите 0, 1, ..., 9 са единици, *виж таблица 10.3.3а*.

цифра	*двоичен запис*	*16-ичен код EBCDIC*	*двоичен код EBCDIC*
0	0000 0000	F0	1111 0000
1	0000 0001	F1	1111 0001
2	0000 0010	F2	1111 0010
3	0000 0011	F3	1111 0011
4	0000 0100	F4	1111 0100
5	0000 0101	F5	1111 0101
6	0000 0110	F6	1111 0110
7	0000 0111	F7	1111 0111
8	0000 1000	F8	1111 1000
9	0000 1001	F9	1111 1001

Таблица 10.3.3а. Представяне на десетичните цифри в *EBCDIC*.

Една възможност за компресиране в този случай ни дава методът на полубайтовото пакетиране. Идеята е в един байт да се записват две цифри. За целта едната цифра се записва в старшия, а другата — в младшия полубайт. Декодирането става чрез отделяне на двата полубайта и прибавяне пред всеки от тях на липсващата четворка битове: в случая 1111_2. Така например, числото 38 ще се кодира като 00111000_2. Впрочем, по подобен начин стоят нещата и при таблицата *ASCII*. Там старшият полубайт на кода на цифрите съдържа 0011_2.

Каква е ефективността на описания метод? За текст, съдържащ само цифри, степента на компресия ще бъде 50%. Впрочем, това е горната граница, която може да бъде достигната. На практика, в текст, съдържащ и други символи, степента на компресия ще бъде по-ниска. Действително, в този случай символите, различни от цифри, ще се подават на изхода без изменения. Възниква необходимост от начин за отличаването им от пакетираните цифри. За целта най-често се използва специална структура, подобна на показаната в *таблица 10.3.3б*.

специален символ	*еднобайтов брояч*	*пакетирана цифра* 1	*пакетирана цифра* 2	...	*пакетирана цифра* n–1	*пакетирана цифра* n

Таблица 10.3.3б. Структура за полубайтово пакетиране на цифри: *еднобайтов* брояч.

Подобно на премахването на нулите и битовите карти тук също се формира *кодираща група*. Тя започва със специален символ, указващ началото на кодирането, последван от брояч. Броячът има дължина един байт и указва броя n на *байтовете*, съдържащи пакетирани цифри. Забележете: броят на *байтовете*, а не на *цифрите*! В следващите n байта се записват самите пакетирани цифри. Обикновено броят на цифрите е малък, поради което отделянето на цял байт за брояч се оказва неразумно. Проблемът се решава чрез използване на 4-битов брояч. Броячът се записва в старшия полубайт на байта, непосредствено след специалния символ, а в младшия полубайт се записва кодът на първата цифра от кодираната последователност (*виж таблица 10.3.3в.*). При празен последен полубайт възниква проблем: как да разберем, че не трябва да го интерпретираме като цифра? Решението е просто: достатъчно е да запишем невалиден код, например 1111_2.

Специален символ	*Полубайтов брояч*	*Пакетирана цифра* 1	*Пакетира - на цифра* 2	*Пакетира - на цифра* 3	...	*Пакетира- на цифра* n–1	*Пакетира - на цифра* n

Таблица 10.3.3в. Структура за полубайтово пакетиране на цифри: *полубайтов* брояч.

Минималната дължина на кодиращата група е 3 байта при еднобайтов брояч и 2 байта — при полубайтов. Така кодираните последователности от цифри трябва да имат дължина поне 4 (*без печалба* и с *пълно използване* на полубайтовете), или 5 (*без печалба* и *без използване* на последния полубайт) или 6 (*с печалба* и с *пълно използване* на полубайтовете) при първия вариант, и съответно 3 (*без печалба* и с *пълно използване* на полубайтовете), или 4 (*без печалба* и *без използване* на последния полубайт), или 5 (*с печалба* и с *пълно използване* на полубайтовете) при втория вариант, за да има смисъл кодирането. Максималната стойност на 4-битовия брояч е 15. В този случай обаче стойностите 0, 1, 2 и 3 се оказват неизползвани. Ето защо можем да изместим интервала надясно, при което на стойност 0 на брояча ще съответства дължина 4. Така ще можем да кодираме последователности с дължина до 37 цифри (*Защо?*). Аналогично, в случая на еднобайтов брояч, при което максималната дължина е 255, се оказва възможно изместване на интервала с 5, за да се достигне 519 (*Защо?*).

Да пресметнем степента на компресия на описания метод. Нека входното съобщение съдържа последователност от n цифри ($n \geq 5$). Тогава дължината му в битове ще бъде $8n$. В случай на еднобайтов брояч компресираното съобщение ще има дължина $16 + 4.\lceil n/2 \rceil$ бита. Тогава за степента на компресия получаваме:

$$\text{степен на компресия} = \frac{8n}{16 + 4\left\lceil \frac{n}{2} \right\rceil}$$

В случай на полубайтов брояч за дължината на кодираното съобщение имаме $12 + 4 . [n/2]$, $n \geq 4$. Оттук лесно се пресмята:

$$\text{степен на компресия} = \frac{8n}{12 + 4\left\lceil \frac{n}{2} \right\rceil}$$

Таблица 10.3.3г. дава представа за ефективността на полубайтовото пакетиране при четирибитов брояч.

брой последователни цифри	*дължина на входното съобщение*	*дължина на кодираното съобщение*	*степен на компресия в проценти*
1	8	16	-

2	16	24	-
3	24	24	-
4	32	32	00.00%
5	40	32	20.00%
6	48	40	16.66%
7	56	40	28.00%
8	64	48	25.00%
9	72	48	33.33%
10	80	56	30.00%
11	88	56	36.36%
12	96	64	33.33%
13	104	64	38.46%
14	112	72	35.71%
15	120	72	40.00%

Таблица 10.3.3г. Дължина на съобщението и степен на компресия при различен брой последователни цифри и полубайтов брояч.

Разгледаният по-горе вариант на полубайтовото пакетиране има сериозни недостатъци. Така например, той работи само с цифри, като при това всяка цифра заема един полубайт. Броят на различните стойности, които могат да се запишат с 4 бита (в един полубайт) е 16. Същевременно, броят на различните десетични цифри е 10. Получаваме потенциална загуба от 6 стойности в рамките на един полубайт (всъщност 5, не бива да забравяме, че ни е необходима още една специална стойност.). Ако разгледаме цял байт, нещата изглеждат още по-лошо: от 256 възможни стойности ние използваме едва 100.

Файлове, съдържащи само десетични цифри, се срещат изключително рядко. Финансовите и статистическите файлове съдържат в изобилие и някои други специални символи като: $, %, +, –, *, /, =, (,), "запетая", "точка" и др. Очевидно, ефективността на полубайтовото пакетиране би могла да се подобри значително, ако към цифрите се прибавят и някои от специалните символи. За съжаление, това не може да стане така директно, както при цифрите, тъй като старшите им полубайтове са различни. Налага се предефиниране на кодовете на символите. Получаваме полубайтово пакетиране, разпознаващо произволен фиксиран набор от 16 различни символа, без значение какви са кодовете им в използваната кодова таблица. *Таблица 10.3.3д.* показва примерно предефиниране на най-често използваните символи при финансови приложения.

символ	*двоичен запис*	*ASCII двоичен код*	*Предефиниран код*
0	0000 0000	0011 0000	0000
1	0000 0001	0011 0001	0001
2	0000 0010	0011 0010	0010
3	0000 0011	0011 0011	0011
4	0000 0100	0011 0100	0100
5	0000 0101	0011 0101	0101
6	0000 0110	0011 0110	0110
7	0000 0111	0011 0111	0111
8	0000 1000	0011 1000	1000
9	0000 1001	0011 1001	1001
+	—	0010 1011	1010

–	—	0010 1101	1011
$	—	0010 0100	1100
,	—	0010 1100	1101
.	—	0010 1110	1110
*	—	0010 1010	1111

Таблица 10.3.3д. Предефиниран код на символи при някои финансови приложения.

Задачи за упражнение:

1. Да се предложи решение на проблема с потенциалната загуба на 156 от 256-те възможни стойности в случай на еднобайтов брояч.
2. Да се реализира алгоритъмът и да се сравни с премахването на нулите (*виж 10.3.1.*) и компресирането с битови карти (*виж 10.3.2.*) върху различни типове файлове.

10.3.4. Съвместно използване на битови карти и полубайтово пакетиране

Някои от по-старите приложения работят със 7-битов *ASCII* код, при който най-старшият бит не се взема предвид, а служи само за проверка по четност при предаване на данни. Така кодовете 00100100 и 10100100 представляват един и същ символ, а именно '$'. В такъв случай можем да опростим кодиращата група, премахвайки специалния символ. Но как ще отличаваме началото на кодиращата последователност, а именно брояча, от останалите символи в кодираното съобщение? Отговорът е прост: броячът ще има най-старши бит 1, а останалите символи — 0. Впрочем, подобна техника е приложима и при битовите карти (*виж 10.3.2.*).

Тази прилика ни навежда на мисълта за създаване на хибриден вариант, съчетаващ предимствата на двата метода. Това може да се постигне лесно при използването на още един бит. Ако стойността му е 1, то имаме кодиране с битова карта, в противен случай е налице полубайтово пакетиране (*виж 10.3.3.*). В първия случай следващите 6 бита следва да се интерпретират като 6-битова карта, а във втория — като 6-битов брояч, приемащ 64 различни стойности. Бихме могли да доразвием метода, позволявайки до 4 различни вида компресия, като за целта ще трябва да отделим за специални цели още един бит. Така размерът на битовата карта ще намалее до 5 бита, а броячът ще приема само до 32 различни стойности.

Задачи за упражнение:

1. Да се реализира предложеният алгоритъм.
2. Да се пресметне степента на компресия на алгоритъма.
3. Да се предложи вариант на алгоритъма в случай, че всичките 8 бита се ползват.

10.3.5. Двуатомно кодиране

Двойка	Общ брой срещания	Среден брой срещания на хиляда символа
E_	328	26,89
_T	292	23,94
TH	249	20,41
_A	244	20,00
S_	217	17,79
RE	200	16,40

IN	197	16,15
HE	183	15,00
ER	171	14,02
_I	156	12,79
_O	153	12,54
N_	152	12,46
ES	148	12,13
_B	141	11,56
ON	140	11,48
T_	137	11,23
TI	137	11,23
AN	133	10,90
D_	133	10,90
AT	119	9,76
TE	114	9,35
_C	113	9,26
_S	113	9,26
OR	112	9,18
R_	109	8,94

Таблица 10.3.5. Най-чести двойки последователни букви в английски текст.

Двуатомното кодиране (англ. *diatomic encoding*) е обобщение на полубайтовото пакетиране. Да предположим, че разполагаме с достатъчно допълнителни символи, несрещащи се във входната последователност. Тогава бихме могли да постигнем до 50% степен на компресия, съпоставяйки на най-често срещаните двойки последователни символи единствен съответен символ. Ключов момент тук е изборът на подходящите двойки, които да бъдат включени в кодирането. Очевидно най-добра ефективност ще се постигне при кодиране на най-често срещаните двойки. Това предполага наличието на някакви предварителна статистика относно честотата на срещане на всяка от тях. В случай, че нямаме нужната статистика, бихме могли да определим двойките с едно преминаване през входното съобщение. Това обаче е възможно само за крайни съобщения, а освен това води до забавяне на алгоритъма и изисква предаване на речника, но от друга страна по-добре моделира съобщението. Ето защо понякога се оказва удачно използването на предварително фиксирана таблица. Това следва да се извършва изключително внимателно и очевидно изисква някаква предварителна статистика относно типа и съдържанието на текста. В случай на текстов файл на някакъв естествен език, например английски, би могла да се състави някаква фиксирана универсална таблица за този език. *Джуел* предлага *таблица 10.3.5.* (знакът “_ “ означава интервал), след като е анализирал 12 198 байтов английски текст. [*Jewell-1976*]

Интерес представлява построяването на такива таблици за компютърни програми на някои по-разпространени езици за програмиране. Двуатомното кодиране е приложимо и в случай, че не разполагаме с допълнителни символи. В този случай се избира някакъв рядък символ, всички негови срещания се удвояват, след което той се използва за кодиране на някоя двойка. Подобен подход ще бъде ефективен, само ако срещанията на символа са били по-малко от тези на двойката.

Задачи за упражнение:

1. Да се построи честотна таблица за най-често срещаните двойки символи за български език по подобие на *таблица 10.3.5.*

2. Да се намерят най-често срещаните двойки букви — кандидати за супербуква за различни видове файлове:

- стандартен текстов файл на английски език;
- стандартен текстов файл на български език;
- програма на *Си, Паскал, Бейсик, Фортран, Java*.

3. Да се оцени максималната и очакваната степен на компресия.
4. Защо срещанията на специалния символ трябва да се удвоят преди да се пристъпи към кодиране?
5. Да се предложи решение на проблема, когато две двойки претендират за една и съща буква.
6. Да се реализира предложеният алгоритъм.
7. Следва ли да се очаква подобрение, ако се разглеждат последователности от 3 (или повече) символа? Какви проблеми могат да възникнат в такъв случай?

10.3.6. Замяна на шаблони

Замяната на шаблони може да се разглежда като обобщение на двуатомното кодиране (*виж 10.3.5.*). Идеята е да се кодират не двойки символи, а *произволни* думи. Така ограничението от 50% за степента на компресия отпада автоматично. Методът демонстрира скромна ефективност при обикновен текстов файл, но дава много добри резултати при кодиране на компютърни програми. Подобно на двуатомното кодиране, и тук се изгражда таблица на съответствията. Намирането на идеалната таблица по принцип е сложен проблем, но в някои конкретни случаи кандидатите за заместване са очевидни. В български език евентуални кандидати за кодиране са думи като: “не”, “на”, “от”, “за”, “който” и др. В английски някои кандидати са: “the”, “for”, “of” и др. Ще отбележим, че тези думи най-често носят силно ограничен собствен смисъл, поради което се включват в списъка на *стоп-думите*, т. е. на пропусканите при индексиране и търсене от търсещите машини в Интернет.

Най-висока степен на компресия обаче се получава при кодиране на компютърни програми, написани на език за програмиране (от високо ниво). Възможни кандидати в този случай са предимно запазените думи на езика, както и някои от често срещаните имена на променливи. *Таблица 10.3.6.* показва някои възможни кандидати за някои езици за програмиране от високо ниво.

По-старите текстови редактори на някои интегрирани програмни среди използват съществено замяната на шаблони при запис на компютърни програми във файл. Като пример можем да приведем средата на *GWBasic*. Други езици за програмиране, например ПЛ/1 на *IBM*, съдържат стотици и дори хиляди запазени или ключови думи, поради което при кодирането им ще се наложи използването на двубайтов код (а може и по-дълъг), което може да се окаже неефективно.

език за програмиране	*възможни кандидати за заместване*
Бейсик	`FOR, GOTO, GOSUB, IF, INPUT, LET, NEXT, PRINT, REM, THEN`
Си	`do, else, for, if, int, void, while`
Фортран	`DO, FORMAT, READ, WRITE`
Паскал	`if, else, for, function, procedure, repeat, then, until, write`

Таблица 10.3.6. Кандидати за заместване при някои езици за програмиране.

Задачи за упражнение:

1. Да се предложи стратегия за намиране на “най-добрата” таблица при зададен текст за кодиране. Да се вземе предвид, че ще се предава и речникът.
2. Необходимо ли е да се кодират всички запазени думи на даден език за програмиране, ако целта е да се състави универсален речник за кодиране на произволна програма?
3. Да се реализира предложеният метод за компресиране.

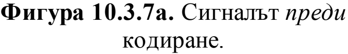
10.3.7. Относително кодиране

Понякога входното съобщение не съдържа дълги последователности от повтарящи се символи, а множество от близки едно до друго числа. В този случай бихме могли да постигнем добри резултати, използвайки друг подход. Нека е дадено входно съобщение, представляващо последователност от години:

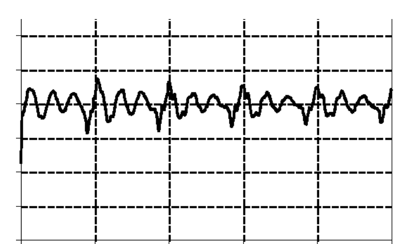
```
1991 1992 1990 1992 1995 1994 1991
```

Ясно е, че за кодирането на всяка година са достатъчни 11 бита ($2^{11} = 2048$). Бихме могли обаче значително да съкратим дължината на съобщението, запазвайки само първата година и представяйки останалите чрез относителното им отместване спрямо нея. Така получаваме:

```
1991 1 –2 2 3 –1 0 –3
```



Фигура 10.3.7а. Сигналът *преди* кодиране.



Фигура 10.3.7б. Сигналът *след* кодиране.

Лесно се вижда, че всички години от горния пример, с изключение на първата, могат да се кодират само с 3 бита. Погледнато по-общо, *относителното кодиране* (англ. *delta encoding*) приема като вход някаква последователност от числа (a_1, a_2,..., a_n) и я преобразува в последователност от вида (a_1, a_2–a_1, ..., a_n–a_1). Тук знакът "–" може да не означава непременно *изваждане*, а следва да се интерпретира като подходяща *аритметична операция*. *Фигури 10.3.7а. и 10.3.7б.* показват аналогов сигнал съответно преди и след относително кодиране.

Относителното кодиране е особено ефективно при архивиране на последователните версии на даден текст. За целта оригиналният текст се запазва изцяло върху диска (лентата), а за всяка следваща редакция се пазят само разликите спрямо предходната версия. Разбира се, възможни са редица вариации на изложения метод. Всички разлики могат да бъдат зададени не спрямо предходната редакция, а спрямо първоначалната версия на текста. Така се премахва необходимостта от изчитане на всички предходни версии при декодирането. На практика се получава проста транслация на текста. Когато броят на версиите започне да нараства, е разумно да се помисли за кодиране, при което на всеки 30 копия например, се сменя началният файл, използван при транслацията, т. е. въвежда се ново начало на "координатната система".

Много добри резултати се постигат при комбиниране на няколко метода на компресиране. Добър пример в това отношение са факс-машините. Те предават страници, всеки ред от които съдържа 1728 бита. Кодирането на страниците става на две стъпки. На първата стъпка се извършва относително кодиране с операцията ⊕ — "побитово изключващо или" (това е поразредната операция ^ в Си), прилагано върху всеки два последователни реда. Пример:

```
00111001 00001101 10110111 --> ред i
   ⊕
10100001 01100001 01110110 --> ред i+1
10011000 01101100 11000001 --> резултат
```

Ще припомним, че операцията ⊕ удовлетворява следните равенства:

```
1 ⊕ 0 = 0 ⊕ 1 = 1
0 ⊕ 0 = 1 ⊕ 1 = 0
```

Целта на тази стъпка е да се получат повече достатъчно дълги еднородни последователности от нули и от единици. Ясно е, че тя запазва дължината на съобщението, но сама по себе си е безсмислена. Така обаче се подготвя по-добра входна последователност за втората стъпка, при която обикновено се прилага разгледаното по-горе премахване на нулите (*виж 10.3.1.*). В случай на предаване на текст този метод работи особено добре, тъй като след първата стъпка ще се получат достатъчно дълги еднородни последователности от "0" и "1". Най-добър резултат ще се получи при бели (черни) страници. Разглеждайки по-общия случай обаче, е ясно, че първата стъпка може да не доведе до задоволителен резултат. Ето защо някои факс-машини използват на втората стъпка статистическо кодиране по Хъфман. (*виж 10.4.2.*)

В този случай ползата от кодирането е очевидна. Действително, стандартното съобщение има разделителна способност 1780 точки на ред и 96 реда на инч, което прави 1 410 048 точки на страница. Без компресия това съобщение би се изпращало 42 секунди при скорост 33 600 *bps* (от англ. *Bits Per Second*), докато с компресия ще бъдат необходими средно едва около 4 секунди.

Друг типичен пример на приложение на относителното кодиране е командата *sccs* на *UNIX*, която служи за централизирано управление при разработката на голям проект, разработван от екип програмисти. Тя предвижда механизъм за поддържане на версии на разработвания продукт, включително автоматично увеличаване поредния номер на версията и възможност за получаване на текущата или произволна предишна. Механизмът се основава на включването на някаква начална версия на продукта, като при всяка следваща промяна се пазят само разликите спрямо предишната, т. е. прилага се относително кодиране.

Задачи за упражнение:

1. Да се сравнят следните две стратегии за запазване на последователните версии на текстов документ:
   а) спрямо предходната редакция;
   б) спрямо първоначалния вариант.
2. Да се предложи начин за определяне и ефективно запазване на разликите между два текстови документа.

10.3.8. Математическо очакване. Кодиране с линейно предсказване

Изложеният по-горе метод на компресиране може да се разшири по естествен начин. При относителното кодиране всеки символ се кодира, като кодът му се сравнява с този на някакъв друг фиксиран символ, най-често непосредствено предхождащия го. Дали използваният метод ще даде добър резултат или — не, т. е. дали при кодирането амплитудата на изменение на стойностите на кодовете ще се намира достатъчно близо до нулата, зависи съществено от избора на този символ. Очевидно най-добри резултати ще се получат, когато елементът е максимално близък до средната стойност, т. е. до математическото очакване на кода на символ от входното съобщение. Едно възможно решение в този случай е входното съобщение да се прочете предварително, при което да се натрупат съответните статистически данни и да се пресметне математическото очакване на кода на символа. След това съобщението се преглежда втори път и се извършва кодиране спрямо това очакване. Недостатък на такъв подход е необходимостта от двукратно прочитане на входното съобщение. Вместо това *кодирането с линейно предсказване LPC* (от англ. *Linear Predictive Code*) пресмята математическото очакване *в момента* на кодирането. Идеята е проста: ако сме прочели 1000 символа, то можем достатъчно точно да определим най-вероятния символ. На изхода се подава разликата между кода на текущия символ и пресметнатото до момента математическо очакване. [*Smith-1998*]

Какво представлява *математическото очакване*? Математическото очакване, известно още като *средна стойност*, е основно понятие в теорията на вероятностите. Математическото очакване на дискретната случайна величина X, приемаща стойности x_1, x_2, …, x_n с вероятности $P(x_1)$, $P(x_2)$, …, $P(x_n)$ се пресмята по формулата:

$$EX = \sum_{i=1}^{n} x_i P(x_i)$$

Статистическото определение за *вероятност* на случайната величина X се свързва с извършване на някои опити и наблюдения над X и се дефинира като отношението на броя на благоприятните изходи към общия брой опити. Да разгледаме случайна величина, която приема стойности 0 или 1 в зависимост от това дали се пада лице или герб при хвърляне на монета. Да предположим, че сме подхвърлили монетата във въздуха 10000 пъти, при което 5037 пъти се е паднал герб. В такъв случай вероятността тази случайна величина да приеме стойност 1, т. е. да се падне герб, е 5037/10000 = 0,5037.

Да разгледаме друг пример: хвърляне на зар. Тук стойностите на случайната величина са 1, 2, 3, 4, 5 и 6, и са равновероятни. Нека пресметнем вероятността да се падне 2. Съгласно определението за вероятност това е $P(x = 2) = 1/6$ (*благоприятни / общ_брой* изходи). Каква е вероятността да се падне четно число? Имаме 3 четни числа: 2, 4 и 6. Отнесено към общия брой изходи — 6, получаваме вероятност $P(x \text{ четно}) = 3/6 = 1/2$.

Да се върнем към математическото очакване на получената стойност при хвърляне на зарче. Както по-горе показахме, вероятността да се падне всяко от числата i е $P(i) = 1/6$, $1 \leq i \leq 6$. Тогава за математическото очакване получаваме:

$$\mathbf{E}X = 1.P(1) + 2.P(2) + 3.P(3) + 4.P(4) + 5.P(5) + 6.P(6)$$
$$\mathbf{E}X = 1.(1/6) + 2.(1/6) + 3.(1/6) + 4.(1/6) + 5.(1/6) + 6.(1/6)$$
$$\mathbf{E}X = (1 + 2 + 3 + 4 + 5 + 6) / 6 = 3,5$$

Така средната стойност или математическото очакване за описания експеримент, т. е. средният брой точки от зарчето, е 3,5. Забележете, че математическото очакване съвсем не е естествено число и не е стойност на случайната величина: ясно е, че не бихме могли да хвърлим 3,5.

Да разгледаме още един пример. Даден е текстов фрагмент, съставен от 200 букви. Текстът съдържа 74 еднобуквени думи, 31 — двубуквени, 10 — трибуквени, 2 — петбуквени и 1 — двадесет и четири буквена. Каква е средната дължина на дума от текста? Непосредствено се пресмята общият брой думи: 118. Оттук за вероятността за поява на k-буквена дума е $P(k) = \#k/118$, $k \in \{1, 2, 3, 5, 24\}$. Тук с $\#k$ сме означили броя на думите с дължина k.

$$\mathbf{E}X = 1.(74/118) + 2.(31/118) + 3.(3/118) + 5.(2/118) + 1.(24/118) = 200/118 \approx 1,694 \text{ букви/дума}$$

След това "лирично" отклонение да се върнем на главния въпрос: Как с помощта на математическо очакване да реализираме кодиране с линейно предсказване? Идеята беше скицирана по-горе: В процеса на преминаване през символите от входното съобщение ще натрупваме статистически данни, позволяващи ни да пресметнем математическото очакване на следващата буква, като на изхода ще подаваме само "грешката", т. е. само разликата между средната стойност и кода на символа. Вместо да натрупваме статистика за символите на входното съобщение, както и за честотата (или вероятността) на срещане на всеки от тях, ще използваме проста рекурентна връзка между старата стойност на $\mathbf{E}X_{стар} = \{x_1, x_2, ..., x_k\}$, и новата $\mathbf{E}X_{нов} = \{x_1, x_2, ..., x_k, x_{k+1}\}$, т. е. след прибавяне на новия символ. В сила е връзката:

$$EX_{нов} = \frac{(n-1)EX_{стар} + код_на_символа}{n}$$

Същата формула се използва и при декодирането. Следва примерна програмна реализация:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100
const char *message="LLLLLLALABALANICAAAABABABBABABABABAAABABALLLLAABB";

/* Извършва LPC кодиране на съобщението */
```

```
void LPCencode(int *code, const char *msg)
{ double exp;
  unsigned n;
  if ('\0' == *msg) return; /* Празно входно съобщение */
  for (exp = *code = *msg++, n = 1; '\0' != *msg; n++, msg++) {
    code[n] = (int) ceil(exp-*msg);
    exp = (exp*n + (unsigned char)*msg)/(n+1);
  }
}

/* Извършва LPC декодиране */
void LPCdecode(char *msg, const int *code, const unsigned n)
{ double exp;
  unsigned i;
  for (exp = *msg++ = *code, i = 1; i < n; i++, msg++) {
    *msg = (char)ceil(exp - code[i]);
    exp = (exp*i + (unsigned char) *msg) / (i+1);
  }
  *msg = '\0';
}

void print(const int *code, const unsigned n)
{ unsigned i;
  for (i = 0; i < n; i++)
    printf("%4d", code[i]);
}

int main(void) {
  int coded[MAX];     /* Кодирано съобщение */
  char decoded[MAX]; /* Декодирано съобщение */
  printf("Входно съобщение:\n%s\n", message);
  LPCencode(coded, message);
  printf("\nКодирано съобщение:\n");
  print(coded, strlen(message));
  LPCdecode(decoded, coded, strlen(message));
  printf("\nДекодирано съобщение:\n%s", decoded);
  return 0;
}
```

🖫 lpc.c

Резултат от изпълнението на програмата:

```
Входно съобщение:
LLLLLLALABALANICAAAABABABBABABABABAAABABALLLLAABB

Кодирано съобщение:
  76   0   0   0   0   0  11  -1  10   8   8  -3   8  -6   0   6   7   7   7   6
   5   6   5   5   4   4   5   4   5   4   4   3   4   3   4   4   4   3   4   3
   4  -8  -7  -7  -7   4   4   3   3
Декодирано съобщение:
LLLLLLALABALANICAAAABABABBABABABABAAABABALLLLAABB
```

Освен еднопасовата обработка изложеният метод има същественото предимство, че отчита локалните особености на входното съобщение. Така например, ако даден сегмент на текста съдържа символа *x* с вероятност 70%, докато вероятността на срещане на *x* в целия текст е едва 3%, то описаният метод на кодиране с линейно предсказване ще успее да отчете тази особеност и да коригира по подходящ начин математическото очакване за този сегмент. Описаният метод може да се разглежда като статистически (*виж 10.4.*) и адаптивен (*виж 10.5.*) едновременно.

Каква все пак е ползата от алгоритъма? На пръв поглед той не води до съкращаване на размера на съобщението. Полза обаче има. От една страна, той води до намаляване на амплитудата на изменение на кодираните стойности, като ги приближава до нулата от двете ѝ страни. Това означава по-малки по абсолютна стойност числа (и по-малко като разнообразие от различни стойности), а оттук — и по-малко необходими двоични битове за запазването им: старшите битове просто могат да се отхвърлят. От друга страна, кодираното съобщение често съдържа дълги последователности от съседни (или почти съседни) еднакви числа. Което е добър вход за други алгоритми като кодирането на последователности (*виж 10.3.9.*) или с битови карти (*виж 10.3.2.*).

Задачи за упражнение:

1. Друг възможен вариант на алгоритъма е да се изчете цялото съобщение и да се намери ентропията му, след което е използвано в процеса на кодиране, вместо частичните ентропии от оригиналния алгоритъм. Какви са предимствата и недостатъците на такъв подход?
2. Ключова характеристика на кодирането с линейно предсказване е неговата адаптивност. Способността за адаптация обаче намалява прогресивно заедно с броя на обработените символи и математическото очакване се изменя все по-трудно. Проблемът е особено изразен в края на дълго съобщение. Какви възможни решения съществуват?
3. Да се сравни предложеният алгоритъм с относителното кодиране от *10.3.7.*

10.3.9. Кодиране на последователности

Кодирането на последователности (англ. *run-length encoding*) е естествено обобщение на премахването на нулите. Класическото премахване на нулите (*виж 10.3.1.*) допускаше евентуалното кодиране на последователните повторения на произволен, но *единствен* и предварително фиксиран символ, обикновено най-често срещаният. За съжаление, най-често срещаните символи могат да бъдат повече от един, например:

```
BBBBBBBCCCCCAAAAAAAAAAAAAACCCABABBCCAAAAAAAAAAAAAAAAAAAA      (*)
```

Нека предположим, че съобщението съдържа само букви. Тогава бихме могли значително да намалим дължината на горния низ, заменяйки всяка последователност от еднакви символи с един-единствен негов екземпляр, предшестван от число, указващо броя на срещанията му. Изложеният метод може да доведе до значително съкращаване дължината на съобщението, особено при по-дълги последователности от еднакви символи. За горния пример съответната кодирана последователност изглежда така:

```
7B5C13A3CABABB2C20A
```

Внимателният читател сигурно е забелязал, че някои от буквите са директно преписани в кодираната последователност, без да са предшествани от число, указващо броя на срещанията им. Действително, обикновено еднобуквените последователности не се кодират, тъй като това води до фактическо удължаване на кода. Двубуквените последователности също могат да не се кодират, тъй като дължината на кодираното съобщение се запазва, т. е. кодирането им не принася полза, макар че не вреди.

В случай, че броят на буквите е достатъчно малък, например 2, бихме могли да използваме модификация на горния метод, при която кодираните символи да се пропускат и да се подразбират. Така например, при кодиране на двоични файлове е ясно, че всяка последователност от нули е непосредствено следвана от последователност от единици и обратно. Последното, разбира се, не се отнася за последната последователност. Нека разгледаме следния пример (вдясно са посочени съответните кодиращи последователности):

```
000000001111111000000  → 8 7 6
000000011111111110000  → 7 10 4
000001111111111111000  → 5 13 3
000001111111111111000  → 5 13 3
```

```
00000001111111111110000  → 7 10 4
00000000111111100000O    → 8 7 6
```

Горният пример представлява кодиране на черно-бяло графично изображение чрез битова карта (англ. *bitmap*). Цветът на всяка точка се задава с един бит: 0 — за бяло, и 1 — за черно. Обикновено в резултат се получават множество достатъчно дълги последователности от едноцветни точки, поради което в общия случай подобен начин на компресиране на графичното изображение води до значително намаляване на дължината на съхраняваните данни. Макар този тип кодиране, често да се използва в практиката (компресиран *bitmap*), все пак той не е универсален и съществуват типове файлове, чието кодиране по описания метод е крайно неефективно. При текстови файлове повторението на повече от 2 последователни еднакви знака е изключително рядко явление, с изключение единствено на интервалите. Ето защо, както вече отбелязахме в *параграф 10.3.1.*, повечето текстови редактори прилагат кодиране на групите от 8 последователни интервала чрез специалния знак за табулация. Ползата от това обаче не е голяма — размерът на файла намалява средно с едва около 4%.

Нека отново се върнем към по-общия случай, когато имаме повече от два различни символа. По-горе беше разгледан случаят, когато кодираното съобщение съдържа само букви, като в този случай преди всяка буква поставяхме някакво число, указващо броя на последователните ѝ повторения. Не е трудно да се забележи, че този метод няма да работи в случая, когато съобщението съдържа цифри. Един възможен начин за решаване на проблема е използването на някакви други символи за представяне на цифрите. В този случай обаче кодирането няма да работи правилно за съобщения, съдържащи въпросните символи.

Ясно е, че както и да избираме множеството от символите, винаги ще съществуват съобщения, които няма да можем да кодираме. Възниква необходимостта от разработване на алгоритъм за кодиране на съобщението, без използване на допълнителни символи. Едно възможно решение на проблема е използването на така наречените *escape последователности*. Подобен подход е изключително удобен и намира широко приложение, например при кодиране на управляващите команди, изпращани към принтера и др. (Ще припомни, че в *параграф 10.3.1.* беше разгледан друг подход, основан на двойка символи *SI* и *SO*.)

Нека отново разгледаме случая, когато кодираното съобщение съдържа само 26-те главни латински букви (по-долу ще ги наричаме просто *азбука*). Ще покажем как с помощта на *escape* последователности можем да извършим кодиране, без използване на символи извън азбуката. Идеята е проста: Избира се произволен символ от азбуката за *escape* символ. Всяка негова поява ще бележи началото на нова *escape* последователност. Това означава, че следващите два символа следва да се интерпретират като двойка от вида *XY* `(брой_срещания,символ)`. Броят срещания на *Y* се дава от поредния номер на *X* в азбуката. Така вместо 7 пишем *G*, вместо 5 — *E*, вместо 13 — *M*, и вместо 20 — *T*. За нашето входно съобщение (*) получаваме следващото кодиране (тук *Z* е *escape* символ):

```
ZGBZECZMACCCABABBCCZTA
```

Тъй като *escape* последователността има дължина точно 3 символа, то кодирането има смисъл само за последователности с дължина поне 4. Какво да правим, ако входното съобщение съдържа *Z*? Действително, няма символ от азбуката, за който да можем да гарантираме, че няма да се срещне във входното съобщение, така че подобна ситуация е напълно възможна. Тя ще възникне, например, при опит за повторно кодиране на вече кодирано съобщение. Едно възможно решение е използване на *escape* последователност с дължина 0, при което дължината на съобщението ще нараства. Впрочем, файловете, съдържащи *Z*, са единствените, които могат да увеличават дължината си.

Понякога се случва дължината на последователността от еднакви символи да бъде достатъчно голяма. В този случай тя се кодира с няколко последователни *escape* последователности. Ако това се случва достатъчно често, подобно кодиране може да се окаже неефективно. Използването на повече от една буква за кодиране дължината на последователността може да доведе до значително подобрение. [*Sedgewick-1990*]

Задачи за упражнение:

1. Нека при реализацията на кодиране на последователности *Z* се използва като *escape* символ, възможно ли в кодираното съобщение да се срещне “*ZZ*”? А “*ZZZ*”? А “*ZZZZ*”?
2. Да се компресира по метода на кодиране на последователности съобщението: “*ZAZZZZZSDHDGSDGHGZGZGZHJGG*”, *Z* е *escape* символ.
3. Да се определи експериментално степента на компресия, постигана при замяна на последователност от *k* интервала със знака за табулация. Каква е оптималната стойност на *k*?
4. Да се сравнят кодирането на последователности и премахването на нулите (*виж 10.3.1.*).
5. При какъв тип файлове следва се очаква, че кодирането на последователности ще бъде ефективно?

10.3.10. Един представителен пример: PackBits

Един добър пример за практическо приложение на кодирането на последователности е алгоритъмът *PackBits*, разработен за потребителите на *Macintosh*. В процеса на кодиране всеки байт (последователност от 8 бита) се заменя с 9 битова последователност. Допълнителният девети бит има специално предназначение и се интерпретира като *знак* на байта. Така на всеки байт от входното съобщение, т. е. на всяко число между 0 и 255, се съпоставя цяло число между –256 и 255 в допълнителен код. В процеса на декодиране в случай, че знаковият бит е 0, т. е. в случай на положително число, следващите 8 бита се подават на изхода без изменение. В случай, че е 1 (отрицателно число), се интерпретира като брой повторения на предходния символ. Така например, последователността 5,5,5,5,5,5,5,5 ще се кодира като двойката (5,–7).

Първият символ е положително число, поради което при декомпресирането ще се интерпретира като себе си: символ с код 5. Вторият е отрицателно число, поради което ще се интерпретира като брой повторения (8) на предходния символ (5).

Основен недостатък на *PackBits* е използването на 9-битови кодиращи последователности. Практически всички съвременни компютри работят с минимална адресируема единица байт, т. е. с 8 битови последователности. Използването на девети бит води до някои проблеми във връзка с необходимостта дължината на кодираната последователност да бъде цяло число байтове, което нормално ще се случва с вероятност 1/9. Известни неудобства възникват и при четене и запис на 9-битови последователности, при което се налага използване на специален буфер. Въпреки това тази схема се справя добре. Освен това, при кодиране на *стандартни ASCII* файлове тя може да се възползва от това, че символите между 128 и 255 включително (така наречената *първа страница*) на таблицата *ASCII* не са стандартизирани. В случай, че във входното съобщение няма символи с код, по-голям от 127, то допълнителният осми бит би могъл да играе ролята на деветия знаков бит в *PackBits*. [*Smith-1998*]

Кодирането на последователности несъмнено е интересен метод, позволяващ множество любопитни обобщения. Ние обаче ще спрем дотук с разглеждането му, тъй като съществуват други, значително по-ефективни.

Задачи за упражнение:

1. Да се сравнят *PackBits* и кодирането на последователности.
2. Да се сравнят *PackBits* и алгоритъмът от *10.3.4*.

10.4. Статистически методи

10.4.1. Алгоритъм на Шенън-Фано

Втората световна война дава силен тласък на развитието на кодирането във всяко едно от трите основни направления: шумозащитно кодиране, криптиране и компресиране. Правят се редица изследвания и теоретични разработки, като постепенно започва да си проправя път идеята, че за ефективното кодиране на едно съобщение е достатъчно да се знае вероятността на срещане на всеки от символите в него. По това време се достига и до идеята за използването на двоичната бройна система като основа на кодиращия алгоритъм. Последното представлява наистина сериозен пробив, особено предвид на факта, че по това време все още няма компютри. Първият универсален ефективен алгоритъм за кодиране е разработен от Клод Шенън, *Bell Labs* и М. Фано, *MIT*, като двамата го предлагат почти едновременно и независимо един от друг. Основната идея на алгоритъма е по зададени вероятности на срещане на всеки от символите във входното съобщение да се съпостави двоичен код. (По-долу ще използваме *код* както за кода на отделна буква, така и за съвкупността от кодовете на буквите от входната азбука.)

Алгоритъмът притежава следните характерни свойства:

1. Дължината на кодовете е променлива.
2. Буквите с по-голяма вероятност се кодират с по-малко битове от тези с по-малка вероятност.
3. Съобщението се декодира еднозначно.

Преди да продължим с изложението на метода, ще въведем някои понятия. Нека разгледаме последното, трето свойство на кода, построяван по алгоритъма на Шенън-Фано. Код, удовлетворяващ това изискване, ще наричаме *разделим*. Очевидно, всички кодове с фиксирана дължина (ще ги наричаме *равномерни* кодове) са разделими. Декодирането в този случай се извършва чрез разбиване на кодираното съобщение на подпоследователности (поддуми) с дължина, равна на дължината на кода, след което се извършва просто заместване на всяка поддума с първообраза ѝ съгласно таблицата на кодиране. В случай, че някоя от поддумите няма първообраз, декодирането е невъзможно, а в останалите случаи винаги се гарантира еднозначност. Пример за равномерен код е *ASCII*.

буква	честота	буква	честота	буква	честота	буква	честота	буква	честота	буква	честота
А	12,68%	Е	9,64%	К	3,43%	П	2,50%	Ф	0,78%	Щ	0,60%
Б	1,08%	Ж	0,76%	Л	3,04%	Р	4,67%	Х	0,35%	Ъ	1,56%
В	4,77%	З	2,21%	М	3,30%	С	4,41%	Ц	0,87%	Ь	0,02%
Г	1,01%	И	8,54%	Н	8,17%	Т	7,80%	Ч	1,62%	Ю	0,10%
Д	3,33%	Й	0,48%	О	9,05%	У	1,44%	Ш	0,18%	Я	1,61%

Таблица 10.4.1а. Средна вероятност за срещане на буквите от българската азбука.

Равномерните кодове дават ограничена възможност за намаляне дължината на входното съобщението. Действително, ако кодираме текст, съдържащ само главните латински букви и интервала (общо 27 на брой символа), бихме могли да използваме 5-битов равномерен код, вместо стандартния 8-битов *ASCII*, съкращавайки по този начин дължината на съобщението с 37,5%. В 5 бита могат да се запишат $2^5 = 32$ различни стойности, а ние трябва да кодираме едва 27 букви.

Използването на равномерни кодове по никакъв начин не отчита честотите на срещане на символите във входното съобщение. Интуитивно обаче е ясно, че ако кодираме с по-малко битове символите с по-голяма вероятност на срещане, бихме могли да получим значително по-добри резултати. По-долу ще видим, че това действително е така. Следва *таблица 10.4.1а.*, съдържаща средните честоти на срещане на буквите в български език в проценти, съгласно [*Манев-1996*].

Въпреки посоченото предимство, неравномерните кодове имат редица недостатъци. На първо място следва да се посочи изключителната им чувствителност към грешни битове. Докато

при равномерните кодове сгрешаването на един бит ще води до повреждане само на един символ, при неравномерните това може да доведе до невъзможност за декодиране на цялото съобщение докрая. Освен това не винаги може да се гарантира еднозначно декодиране. Нека разгледаме например следния код: *a* → 01, *b* →11, *c* → 0111 и кодираното съобщение: 01110111. Вижда се, че можем да го декодираме като: *abab*, *abc*, *cab* или *cc*.

Възниква въпросът: Как да строим разделими кодове? Съществуват различни класове разделими кодове, като най-често използвани са така наречените префиксни кодове. Ще казваме, че един код е *префиксен*, ако кодът на никой символ не е префикс (начало) на кода на никой друг символ. Не е трудно да се покаже, че всеки префиксен код е разделим. Обратното обаче не е вярно. Наистина, нека разгледаме кода: *a* → 01, *b* →00, *c* → 0111. Не е трудно да се покаже, че той е разделим, макар очевидно да не е префиксен (*a* е префикс на *c*). Префиксните кодове са изключително удобни, тъй като не само осигуряват еднозначност на декодирането, но и дават прост и ефективен алгоритъм за това. [*Манев-1996*]

символ	*вероятност*	*код*			
		стъпка 1	*стъпка* 2	*стъпка* 3	*стъпка* 4
f	0,25	1	1		
c	0,20	1	0		
d	0,15	0	1	1	
e	0,15	0	1	0	
a	0,10	0	0	1	
g	0,10	0	0	0	1
b	0,05	0	0	0	0

Таблица 10.4.1б. Алгоритъм на Шенън-Фано за източника: (*a*:0,1), (*b*:0,05), (*c*:0,20), (*d*:0,15), (*e*:0,15), (*f*:0,25) и (*g*:0,10).

Нека си поставим като задача намирането на оптимален неравномерен разделим код. Дали можем да се ограничим само с префиксни кодове? Оказва се, че за всеки разделим непрефиксен код съществува съответен префиксен код със същите дължини на кодиране на буквите на входната азбука (*виж* [*Манев-1996*]). Нека предположим, че са ни известни вероятностите за поява на всеки символ от входната азбука във входното съобщение. Ще считаме, че буквите, които не участват в съобщението, не принадлежат на азбуката и няма да им съпоставяме никакъв код. В случай, че вероятностите на срещане не са ни предварително известни, при крайна дължина на входното съощение, бихме могли сами да си ги пресметнем. За целта е достатъчно да намерим броя (честотата) срещания на всеки символ и да я разделим на броя на символите (т. е. на дължината) на входното съобщение. Всъщност изложеният по-долу алгоритъм работи еднакво добре както с честота, така и с вероятност на срещане.

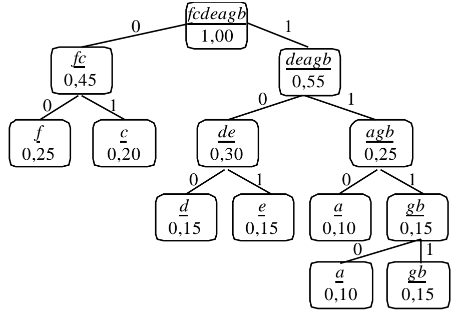
Алгоритъм на Шенън-Фано

1. Сортираме символите по вероятност на срещане в намаляващ ред.
2. Разделяме множеството от символите на две подмножества с (почти) равни вероятности.
3. На едното подмножество съпоставяме 0, а на другото — 1 като поредна буква.
4. Ако някое от подмножествата съдържа повече от един елемент, прилагаме за него същия процес, започвайки от стъпка 2.

Ще илюстрираме горния алгоритъм с помощта на прост пример. Нека е дадено някакво множество от символи заедно с вероятностите на поява на всеки от тях: (*a*: 0,1), (*b*: 0,05), (*c*: 0,20), (*d*: 0,15), (*e*: 0,15), (*f*: 0,25) и (*g*: 0,10). Следвайки горния алгоритъм, извършваме съответно сортиране: *f, c, d, e, a, g, b*. След това последователно разделяме на групи. Резултатът е даден в *таблица 10.4.1б.*

Получаваме кода: (*a*: 001), (*b*: 0000), (*c*: 10), (*d*: 011), (*e*: 010), (*f*: 11) и (*g*: 0001). Ясно е, че описаният процес на разделяне всъщност строи двоично дърво, по върховете на което са разпо-

ложени множествата: в корена е входното множество, съдържащо всички букви от азбуката (с положителна честота на срещане), а в листата са всичките му едноелементни подмножества, т. е. отделните букви. Разделянето на дадено множество на две подмножества се интерпретира като създаване на два наследника на съответния връх. Ще наричаме това дърво *кодиращо*.



Фигура 10.4.1. Кодиращо дърво на Шенън-Фано за източника: (*a*:0,1), (*b*:0,05), (*c*:0,20), (*d*:0,15), (*e*:0,15), (*f*:0,25) и (*g*:0,10).

Как по зададено кодиращо дърво да получим съответния код за всеки от символите? Нека сме фиксирали някое листо и искаме да определим кода, който му е съпоставен. Започвайки от корена, се придвижваме надолу по дървото, избирайки на всяка стъпка този наследник на текущия връх, който съдържа интересуващия ни символ. Всеки път, когато вървим наляво, пишем 0, а когато вървим надясно — 1. Последователността от 0 и 1, която получаваме при достигане до листо, е търсеният код, *виж фигура 10.4.1.* Не е трудно да се забележи, че построеният по този начин код е *префиксен* (виж по-горе). Действително, за да бъде кодът на някой символ префикс на кода на друг символ, трябва поне единият символ да бъде разположен в нелисто, което е невъзможно съгласно конструкцията на дървото.

Как става декодирането? Започваме от корена на дървото: Ако текущият бит е 0, тръгваме по левия наследник, ако е 1 — по десния. При достигане до листо записваме на изхода кода на съдържащия се там символ, след което продължаваме декодирането, започвайки отново от корена.

символ	*вероятност*	*код*		
		стъпка 1	*Стъпка* 2	*стъпка* 3
a	0,20	1	1	
b	0,20	1	0	
c	0,20	0	1	1
d	0,15	0	1	0
e	0,15	0	0	1
f	0,10	0	0	0

Таблица 10.4.1в. Алгоритъм на Шенън-Фано за източника: (*a*:0,20), (*b*:0,20), (*c*:0,20), (*d*:0,15), (*e*:0,15), (*f*:0,10). *Вариант 1.*

Внимателният читател вероятно е забелязал, че алгоритъмът на Шенън-Фано не винаги ще строи оптимални кодове. Основният проблем е в начина на разделяне. В случай на почти равно разделяне е възможно да възникнат две възможности, което да доведе до две различни дървета с различни тегла. Нека разгледаме следния пример: (*a*: 0,20), (*b*: 0,20), (*c*: 0,20), (*d*: 0,15), (*e*: 0,15), (*f*: 0,10). Възможни са 3 различни кода на Шенън-Фано: *виж таблици 10.4.1в., 10.4.1г. и 10.4.1д.*

символ	*вероятност*	*код*		
		стъпка 1	*стъпка* 2	*стъпка* 3

a	0,20	1	1	
b	0,20	1	0	1
c	0,20	1	0	0
d	0,15	0	1	
e	0,15	0	0	1
f	0,10	0	0	0

Таблица 10.4.1г. Алгоритъм на Шенън-Фано за източника: (*a*:0,20), (*b*:0,20), (*c*:0,20), (*d*:0,15), (*e*:0,15), (*f*:0,10). *Вариант 2*.

символ	*вероятност*	*код*		
		стъпка 1	*стъпка* 2	*стъпка* 3
a	0,20	1	1	1
b	0,20	1	1	0
c	0,20	1	0	
d	0,15	0	1	
e	0,15	0	0	1
f	0,10	0	0	0

Таблица 10.4.1д. Алгоритъм на Шенън-Фано за източника: (*a*:0,20), (*b*:0,20), (*c*:0,20), (*d*:0,15), (*e*:0,15), (*f*:0,10). *Вариант 3*.

Да означим с l_i ($1 \leq i \leq n$) дължината на кода на буквата a_i, а с p_i — вероятността за срещането ѝ. Да пресметнем *средния брой битове*, необходими за кодирането на произволна буква от входното съобщение. Тази величина ще наричаме *цена на кода* и ще я бележим с L. Цената на кода се пресмята по формулата:

$$L = \sum_{i=1}^{n} l_i p_i$$

На практика, това е математическото очакване (*виж 10.3.8.*) на дължината на произволна дума след кодирането ѝ. Нека е дадено входно съобщение α и нека означим с $d(\alpha)$ дължината му в брой букви. Да фиксираме някоя от буквите a_i от азбуката. Тогава произведението $l_i p_i d(\alpha)$ ще ни даде *очаквания* (не непременно действителния) брой битове, които буквата a_i внася в кодираното съобщение. Сумирайки по всички символи от входното съобщение, за *очакваната* (не действителната) дължина в битове на кодираното съобщение получаваме ([*Манев-1996*]):

$$\sum_{i=1}^{n} l_i p_i d(\alpha) = d(\alpha) \sum_{i=1}^{n} l_i p_i = d(\alpha).L$$

От последната формула се вижда ясно, че цената на кода еднозначно определя очакваната дължина на кодираното съобщение. Побуквен код, за който, при фиксирана азбука $\{a_1, a_2, ..., a_n\}$ и фиксирани съответни честоти на срещане $\{p_1, p_2, ..., p_n\}$, величината L приема минимална стойност, ще наричаме *оптимален побуквен код*.

Да пресметнем цената на кода за горните три таблици:

$$L_1 = 2.0,20 + 2.0,20 + 3.0,20 + 3.0,15 + 3.0,15 + 3.0,10 = 2,33 \text{ бита}$$
$$L_2 = 2.0,20 + 3.0,20 + 3.0,20 + 2.0,15 + 3.0,15 + 3.0,10 = 2,65 \text{ бита}$$
$$L_3 = 3.0,20 + 3.0,20 + 2.0,20 + 2.0,15 + 3.0,15 + 3.0,10 = 2,65 \text{ бита}$$

Оказва се, че първият код е по-ефективен от останалите два, и съпоставя на всеки символ едва 2,33 срещу 2,65 бита.

Задачи за упражнение:

1. Да се намери оптималният код за *фигура 10.4.1*.

2. Да се дадат още примери за неравномерен разделим код.

3. Какъв е броят на върховете в дървото на Шенън-Фано?

4. Каква е максималната височина на дървото на Шенън-Фано, изразена като функция на броя на буквите от входната азбука? Балансирано ли е?

5. Да се докаже, че кодът, построяван от алгоритъма на Шенън-Фано, е префиксен.

6. Кога алгоритъмът на Шенън-Фано ще строи гарантирано оптимален код?

7. Да се определи сложността на алгоритъма на Шенън-Фано.

8. Възможно ли е алгоритъмът на Шенън-Фано да се модифицира така, че винаги да строи *оптимален* побуквен код?

10.4.2. Алгоритъм на Хъфман

Хъфман успява да избегне недостатъците на алгоритъма на Шенън-Фано, като строи дървото от листата към корена.

Алгоритъм на Хъфман

1. Образуваме от всеки символ тривиално дърво, в корена (единствения връх) на което записваме вероятността на срещане на съответния символ.

2. Намираме двата върха с най-малки вероятности и ги обединяваме в ново дърво с корен, съдържащ сумата от вероятностите им.

3. Ако има поне две дървета, преход към 2.

Алгоритъмът на Хъфман строи дърво със същите свойства като изгражданото от алгоритъма на Шенън-Фано. Търсеният код, както и декодирането на съобщението се извършват абсолютно аналогично. Тук обаче символите с по-голяма честота се включват по-късно в дървото. Така те се оказват по-близо до корена, т. е. съпоставя им се по-кратък код.

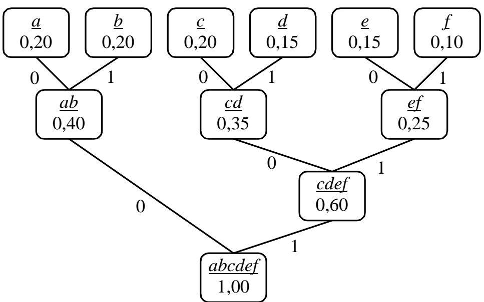
Ще отбележим едно интересно свойство на дървото на Хъфман, което впрочем е свойство и на дървото на Шенън-Фано, при положение че вместо с вероятности сме работили с честота на срещане: Дължината на кодираното съобщение е равна на дължината на *претегления външен път* в дървото на Хъфман. Под "дължина на претегления външен път" ще разбираме сумата по всички листа на честотата, записана в листото, умножена по дължината на пътя до корена. Доказателството на това твърдение е очевидно. Действително, нека фиксираме един символ от входното съобщение. За да изчислим колко бита внася той в кодираното съобщение, е достатъчно да умножим броя срещания на символа по дължината на кода, който му съпоставя дървото на Хъфман. Сумирайки по всички символи от входното съобщение, получаваме общата дължина на кодираното съобщение.

Най-важното свойство на дървото на Хъфман обаче е, че винаги гарантира построяването на оптимален побуквен код, за разлика от алгоритъма на Шенън-Фано, който, както вече видяхме, понякога греши. Ако на някоя стъпка има повече от една възможности, се избира коя да е от тях. Изобщо, алгоритъмът на Хъфман е добър пример за приложение на алчен алгоритъм (*виж 9.1.*). Не е трудно да се докаже, че всички дървета на Хъфман, построени по едни и същи честоти, имат еднаква дължина на претегления външен път.

Ще илюстрираме алгоритъма на Хъфман върху съобщението:

```
afbabcdefacbabcdecde
```

Пресмятаме честотите на срещане на отделните символи: (*a*: 0,20), (*b*: 0,20), (*c*: 0,20), (*d*: 0,15), (*e*: 0,15), (*f*: 0,10) и построяваме дървото на Хъфман. (*виж фигура 10.4.2.*)



Фигура 10.4.2. Кодиращо дърво на Хъфман за източника: (*a*:0,20), (*b*:0,20), (*c*:0,20), (*d*:0,15), (*e*:0,15), (*f*:0,10).

Забележете, че дървото от *фигура 10.4.2.* е начертано обърнато по отношение на дървото на Шенън-Фано (сравнете с *фигура 10.4.1.*). Това е направено нарочно с цел да се подчертае, че се изгражда от листата към корена, за разлика от алгоритъма на Шенън-Фано. Горното дърво ни дава следните кодове: a = 00, b = 01, c = 100, d = 101, e = 110 и f = 111. Кодираното съобщение ще изглежда така (оставили сме разстояния с цел по-лесна читаемост):

```
00 111 01 00 01 100 101 110 111 00 100 01 00 01 100 101 110 100 101 110
```

За съхраняването на входното съобщение в стандартен 8-битов *ASCII* формат ще са необходими 8.20 = 160 бита. От друга страна, тъй като имаме само 6 различни символа, можем да използваме 3-битов равномерен код, при което ще са ни необходими 3.20 = 60 бита. Кодирането по Хъфман обаче изисква едва 52 бита, т. е. средно 52/20 = 2,6 бита за символ.

Каква е времевата сложност на изграждането на дървото на Хъфман? Ясно е, че тази сложност ще зависи от броя n на различните символи, участващи във входното съобщение, но не и от съдържанието му. Тривиалният подход при решаването ни дава сложност $\Theta(n^2)$. Действително, на всяка стъпка се обединяват точно две дървета. Т. е. броят на стъпките е пропорционален на n. Ако на всяка стъпка за избора на двете дървета с минимални честоти ни е необходимо време, пропорционално на n, то получаваме посочената по-горе сложност. Броят на стъпките не би могъл да бъде намален.

По-долу ще съсредоточим усилията си върху съставянето на проста реализация на кодирането по Хъфман. Преди да пристъпим към реализацията на същинския алгоритъм ще съберем нужните ни статистически данни за честотата Ще работим с честота, вместо вероятност на срещане на всеки от символите от входното съобщение `MSG`. Така няма да се налага да работим с реални числа. След това за всеки символ с ненулева честота на срещане ще изградим съответно тривиално дърво, състоящо се от единствен възел — символа, който представя (виж по-долу функцията `initModel()`). Получаваме *Хъфманова гора*. На всяка стъпка ще избираме двете дървета с най-ниски тегла по начина, описан по-горе. Дърветата ще изграждаме като динамични структури от данни и за всяко от тях ще пазим указател в специален масив от дървета `forest[]`. Освен въпросния указател масивът ще съдържа и данни за теглата на всяко от дърветата: сумата от честотите на срещане на символите от листата на дървото. На нелистата няма да съпоставяме символи, а само честоти на срещане. Най-общо алгоритъмът на Хъфман изглежда така:

```
void huffman(void)
{ /* Докато гората съдържа повече от едно дърво */
  while (forest_съдържа_поне_2_дървета) {
    findMins(i,j);       /* 1. Намиране на двата най-редки върха */
    forest[i] = createNew(forest[i],forest[j]);/* 2. Обединяване */
    free(forest[j]);            /* 3. Премахване на j-тото дърво */
  }
}
```

След построяването на кода ще можем да изведем дървото на Хъфман (функция `printTree()`), както и кодовете на всички символи с ненулева честота на срещане (функция `writeCodes()`) на екрана. Следва пълна реализация (функцията `findMins()` намира двете най-редки дървета):

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 256
#define MSG "afbabcdefacbabcdecde"

struct tree {
  char sym;                   /* Символ (буква) */
  unsigned freq;              /* Честота на срещане на символа */
  struct tree *left, *right; /* Ляв и десен наследници */
};

struct {
  unsigned weight;            /* Тегло на дървото */
  struct tree *root;          /* Родител */
} forest[MAX];                /* Гора: масив от дървета */

unsigned treeCnt;             /* Брой дървета в гората */

char code[MAX];               /* Код на Хъфман за съответия символ */

void initModel(char *msg)  /* Намира честотата на срещане на символите */
{ char *c = msg;
  unsigned freqs[MAX];        /* Честоти на срещане на символите */
  unsigned i;

  /* Построяване на таблица на честотите на срещане */
  for (i = 0; i < MAX; i++)
    freqs[i] = 0;
  while (*c)
    freqs[(unsigned char) *c++]++;

  /* За всеки символ с ненулева честота се създава тривиално дърво */
  for (treeCnt = i = 0; i < MAX; i++)
    if (freqs[i]) {
      forest[treeCnt].weight = freqs[i];
      forest[treeCnt].root = (struct tree *) malloc(sizeof(struct tree));
      forest[treeCnt].root->left = NULL;
      forest[treeCnt].root->right = NULL;
      forest[treeCnt].root->freq = freqs[i];
      forest[treeCnt++].root->sym = i;
    }
}

void findMins(unsigned *min,
              unsigned *secondMin) /* Намира двата най-редки елемента */
{ unsigned i;
  if (forest[0].weight <= forest[1].weight) {
    *min = 0;
    *secondMin = 1;
  }
  else {
    *min = 1;
    *secondMin = 0;
```

```
  }
  for (i = 2; i < treeCnt; i++)
    if (forest[i].weight < forest[*min].weight) {
      *secondMin = *min;
      *min = i;
    }
    else if (forest[i].weight < forest[*secondMin].weight)
      *secondMin = i;
}

void huffman(void)
{ unsigned i,j;
  struct tree *t;

  while (treeCnt > 1) {
    findMins(&i,&j);  /* Намиране на двата най-редки върха */

    /* Създаване на нов възел - обединение на двата най-редки */
    t = (struct tree *) malloc(sizeof(*t));
    t->left = forest[i].root;
    t->right = forest[j].root;
    t->freq = forest[i].weight += forest[j].weight;
    forest[i].root = t;

    /* j-тото дърво не е нужно повече. Заместване с последното. */
    forest[j] = forest[--treeCnt];
  }
}

void printTree(struct tree *t, unsigned h) /* Извежда дървото */
{ unsigned i;
  if (t) {
    printTree(t->left,h+1);
    for (i = 0; i < h; i++)
      printf("   ");
    printf("%4d",t->freq);
    if (NULL == t->left)
      printf(" %c",t->sym);
    printf("\n");
    printTree(t->right,h+1); }
}

void writeCodes(struct tree *t, unsigned index) /* Извежда кодовете */
{ if (t) {
    code[index] = '0';
    writeCodes(t->left,index+1);
    if (NULL == t->left) {/* Всеки връх има 0 или 2 наследника */
      code[index] = '\0';
      printf("%c = %s\n",t->sym,code);
    }
    code[index] = '1';
    writeCodes(t->right,index+1);
  }
}

int main(void) {
  initModel(MSG);
  huffman();
  printf("Дърво на Хъфман за %s:\n",MSG);
```

```
  printTree(forest[0].root,0);
  writeCodes(forest[0].root,0);
  return 0;
}
```
🖫 huffman1.c

Внимателният читател вероятно е забелязал, че масивът `forest[]` и върховете на дървета-та дублират данните за честотата на срещане. В действителност съхраняването на тези данни за всеки връх на дървото е абсолютно ненужно и по-горе е направено единствено с цел да се улесни разпечатването на теглото на съответното дърво.

Резултат от изпълнението на програмата:
```
Дърво на Хъфман за afbabcdefacbabcdecde:
         4 b
      8
         4 c
  20
            2 f
         5
            3 d
     12
            3 e
         7
            4 a
b = 00
c = 01
f = 100
d = 101
e = 110
a = 111
```

Не е трудно да се провери, че полученото дърво е Хъфманово, при това *различно* от конструираното в началото на параграфа. Това е така, защото на някои от стъпките сме имали *повече от една възможност за избор* на наредена двойка (*i*,*j*) и просто сме се спрели на друга възможност. Дървото от примера има хубавото свойство, че е "сортирано" в смисъл, че при стандартно обхождане ляво-корен-дясно ще получим елементите му сортирани по азбучен ред. Това не е така при дървото, построено от програмата. За да заставим програмата винаги да строи "сортирани" дървета на Хъфман, е нужно да заменим двете строги неравенства (<) във `findMins()` с нестроги (≤), както и да направим минимална промяна в кода на функцията `huffman()`. Следва кодът на променените функции (с "`/* <-- */`" са отбелязани промените):

```
void findMins(unsigned *min,
              unsigned *secondMin) /* Намира двата най-редки елемента */
{ unsigned i;
  if (forest[0].weight <= forest[1].weight) {
    *min = 0;
    *secondMin = 1;
  }
  else {
    *min = 1;
    *secondMin = 0;
  }
  for (i = 2; i < treeCnt; i++)
      if (forest[i].weight <= forest[*min].weight) {        /* <-- */
      *secondMin = *min;
      *min = i;
    }
```

```
    else if (forest[i].weight <= forest[*secondMin].weight) /* <-- */
      *secondMin = i;
}

void huffman(void)
{ unsigned i,j;
  struct tree *t;
  while (treeCnt > 1) {
    findMins(&i,&j);  /* Намиране на двата най-редки върха */

    /* Създаване на нов възел - обединение на двата най-редки */
    t = (struct tree *) malloc(sizeof(*t));

    if (i < j) {                                            /* <-- */
      t->left = forest[i].root;                             /* <-- */
      t->right = forest[j].root;                            /* <-- */
    }                                                       /* <-- */
    else {                                                  /* <-- */
      t->right = forest[i].root;                            /* <-- */
      t->left = forest[j].root;                             /* <-- */
    }                                                       /* <-- */

    forest[i].weight += forest[j].weight;
    forest[i].root = t;

    /* j-тото дърво не е нужно повече. Заместване с последното. */
    forest[j] = forest[--treeCnt];
  }
}
```

🖫 huffman2.c

Внимателният читател вероятно е забелязал, че горната функция не присвоява стойност на `t->freq`. Както по-горе вече споменахме това поле е напълно излишно, поради което просто сме го премахнали. Това налага и съответна промяна в кода на функцията `printTree()`. Промените са отразени по-долу в кода на *huffman3.c*.

Резултат от изпълнението на програмата:

```
Дърво на Хъфман за afbabcdefacbabcdecde:
       --  a
    --
       --  b
 --
          --  c
       --
          --  d
    --
          --  e
       --
          --  f

a = 00
b = 01
c = 100
d = 101
e = 110
f = 111
```

Можем ли да подобрим алгоритъма? Може би най-естествено е да опитаме да оптимизираме процеса на избор. Той би могъл да се ускори значително при използването на *приоритетна опашка* (*пирамида*: *виж 3.1.9.*). В този случай изборът може да се извършва за време, пропорционално на $\Theta(\log_2 n)$. Ще припомним, че предварителното построяване на пирамида изисква време от порядъка на $\Theta(n)$ и не влияе на общата сложност на алгоритъма, откъдето окончателно получаваме сложност $\Theta(n.\log_2 n)$. Следва описание на псевдокод на функция, изграждаща дървото на Хъфман:

```
void huffman()
{ buildHeap();                          /* Построяване на пирамидата */
  /* Докато гората съдържа поне 2 дървета */
  while (forest_съдържа_поне_2_дървета) {
    min1 = getMinAndRemoveFromHeap();/* 1. Намиране на най-редкия връх */
    min2 = getMinAndRemoveFromHeap();/* 2. Следващ най-рядък връх */
    insert(createNew(min1,min2));/* 3. Обединяване и -> в пирамидата */
  }
}
```

Следва измененият код на програмата. Забележете, че тук нулевият елемент на масива `forest[]` не се използва. Промените отново са отбелязани с “`/* <-- */`“.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 256
#define MSG "afbabcdefacbabcdecde"
struct tree {                                           /* <-- */
  char sym;                 /* Символ */
  struct tree *left, *right; /* Ляв и десен наследници */
};
struct CForest {                                        /* <-- */
  unsigned weight;          /* Тегло на дървото */
  struct tree *root;        /* Родител */
} forest[MAX];              /* Гора: масив от дървета */
unsigned treeCnt;           /* Брой дървета в гората */
char code[MAX];             /* Код на Хъфман за съответния символ */

/*******************************************************/
/************** Функции за работа с пирамиди ***************/
/*******************************************************/
/* "Отсява" елемент НАГОРЕ по пирамидата */
void siftUp(unsigned k)                                       /* <-- */
{ struct CForest save = forest[k]; /* Помощна променлива */
  unsigned parent = k/2;           /* "Баща" на разглеждания елемент */
  while (parent >= 1) {
    /* Придвижване към върха на пирамидата */
    if (save.weight < forest[parent].weight) {
      forest[k] = forest[parent];
      k = parent;
    }
    parent /= 2;
  }
  forest[k] = save; /* Намерено е окончателното място за вмъкване */
}

/* "Отсява" елемент НАДОЛУ по пирамидата */
void siftDown(void)                                     /* <-- */
{ unsigned parent = 1,         /* "Баща" */
           child  = 2;         /* "Дете" */
```

```
  struct CForest save = forest[1]; /* Помощна променлива */
  while (child <= treeCnt) {
    if (child+1 <= treeCnt) /* Търсене на по-големия наследник */
      if (forest[child+1].weight < forest[child].weight)
        child++;
    if (save.weight > forest[child].weight) {
      /* Отсяване на елемента надолу */
      forest[parent] = forest[child];
      parent = child;
      child *= 2;
    }
    else
      break;
  }
  forest[parent] = save; /* Намерено е окончателното място за вмъкване */
}

void removeMin(void) /* Премахва върха на пирамидата */       /* <-- */
{ forest[1] = forest[treeCnt--];
  siftDown();
}

/*********************************************/
/************* Основна програма *************/
/*********************************************/
void initModel(char *msg)  /* Намира честотата на срещане на символите */
{ char *c = msg;
  unsigned freqs[MAX]; /* Честоти на срещане на символите */
  unsigned i;

  /* Построяване на таблица на честотите на срещане */
  for (i = 0; i < MAX; i++)
    freqs[i] = 0;
  while (*c)
    freqs[(unsigned char) *c++]++;

  /* За всеки символ с ненулева честота се създава тривиално дърво */
  treeCnt = 0;
  for (i = 0; i < MAX; i++)
    if (freqs[i]) {
      forest[++treeCnt].weight = freqs[i];                    /* <-- */
      forest[treeCnt].root = (struct tree *) malloc(sizeof(struct tree));
      forest[treeCnt].root->left = NULL;
      forest[treeCnt].root->right = NULL;
      forest[treeCnt].root->sym = i;
      siftUp(treeCnt);                                  /* <-- */
    }
}

void huffman(void)                                      /* <-- */
{ struct CForest min1, min2;
  while (treeCnt > 1) {
    /* Намиране и премахване на двата най-леки върха */
    min1 = forest[1];
    removeMin();
    min2 = forest[1];
    removeMin();
    /* Създаване на нов възел - обединение на двата най-редки */
    forest[++treeCnt].root = (struct tree *) malloc(sizeof(struct tree));
```

```
    forest[treeCnt].root->left = min1.root;
    forest[treeCnt].root->right = min2.root;
    forest[treeCnt].weight = min1.weight + min2.weight;
    /* Вмъкване на възела */
    siftUp(treeCnt);
  }
}

void printTree(struct tree *t, unsigned h)
/* Извежда дървото на екрана, връща като резултат теглото му */
{ unsigned i;
  if (NULL != t) {
    printTree(t->left,h+1);
    for (i = 0; i < h; i++)
      printf("   ");
    printf(" -- ");                                    /* <-- */
    if (NULL == t->left)
      printf(" %c", t->sym);
    printf("\n");
    printTree(t->right,h+1);
  }
}

void writeCodes(struct tree *t, unsigned index) /* Извежда кодовете */
{ if (NULL != t) {
    code[index] = '0';
    writeCodes(t->left,index+1);
    if (NULL == t->left) { /* Всеки връх има 0 или 2 наследника */
      code[index] = '\0';
      printf("%c = %s\n",t->sym,code);
    }
    code[index] = '1';
    writeCodes(t->right,index+1);
  }
}

int main(void) {
  initModel(MSG);
  huffman();
  printf("Дърво на Хъфман за %s:\n",MSG);
  printTree(forest[1].root,0);                           /* <-- */
  writeCodes(forest[1].root,0);                          /* <-- */
  return 0;
}
```

huffman3.c

Някои автори считат, че използването на пирамида не си струва усилията, тъй като броят на различните букви във входната последователност не може да надвишава 256, колкото е броят на символите в таблицата *ASCII*. Действително, при малък брой n на различните букви числото $\Theta(n^2)$ е малко и при това достатъчно близко до $\Theta(n.\log_2 n)$. От друга страна, операциите по поддържането на пирамидата са достатъчно прости и реализирането им не изисква особени усилия. Освен това съображението, че броят на различните букви не надвишава 256, не може да се счита за основателно. Понятието *буква*, използвано при описанието на кодиращия алгоритъм, по никакъв начин не бива да се смесва с понятието *символ* от кодовата таблица на съответната компютърна система. Използването на алгоритъма на Хъфман с многосимволни последователности (най-често двусимволни) съвсем не е рядко явление. Освен това днес съществуват двубайтови кодови таблици, например *Unicode*, съдържащи до 65536 различни символа. Изобщо, понятието

буква в алгоритъма на Хъфман може да бъде достатъчно общо и не бива да се свързва с фиксиран брой байтове или битове. Така например, ако искаме да кодираме по Хъфман програма на Си, се оказва много по-ефективно да разглеждаме като символи запазените думи на езика (*while, for, if* и др.), както и някои по-често срещани променливи. При това е ясно, че колкото повече битове съдържат буквите, толкова по-ефективно ще бъде кодирането. Естествено, не бива да се прекалява, тъй като за успешното декодиране на съобщението се налага запазване на буквите, заедно със съответните кодове. В случай че буквите се подразбират, те може да не се запазват, като се запази само дървото на Хъфман.

При запазването на дървото на Хъфман се налагат две основни изисквания: простота на реализацията и компактност на записа. Ще използваме идеята на Дж. Амстердам (*виж* [*TopTeam-1997*]). Следва функция за запис на Хъфманово дърво на диска:

```
void writeTree(tree *t) { /* Вика се с корена на Хъфмановото дърво */
{
  if (t != NULL)           /* Дали t е непразно? */
    if (isLeaf(t)) {         /* Ако t е листо */
      writeBit(0);           /* Пишем бит 0 */
      writeSym(t->sym);      /* Пишем 8 бита ASCII код на символа */
    }
    else {                 /* t е нелисто */
      writeBit(1);           /* Пишем бит 1 */
      writeTree(t->left);    /* Обхождане на лявото поддърво */
      writeTree(t->right);   /* Обхождане на дясното поддърво */
    }
}
```

Прочитането на дървото от диска става с помощта на следната функция:

```
struct tree *readTree() { /* Прочита Хъфманово дърво от диска */
{ struct tree *l, *r, *p;
  bit b;

  if (1 == (b = readBit())) {      /* Попаднали сме на нелисто */
    l = readTree();      /* Извличане на левия наследник */
    r = readTree();      /* Извличане на десния наследник */
    p = newNode(l,r); /* Конструиране на нов връх p с наследници l и r */
    return p;         /* Връщане на p като резултат */
  }
  else {               /* Попаднали сме на листо */
    sym = getSym();      /* Прочитане на следващите 8 бита като символ */
    p = emptyNode(sym); /* Тривиално дърво с един връх, съдържащ sym */
    return p;            /* Връщане на p като резултат */
  }
}
```

От изложените по-горе алгоритми лесно се вижда, че дърво с n възела изисква памет $10n-1$ бита. Действително, всяко дърво с n листа, построено по алгоритъма на Хъфман, съдържа точно $n-1$ нелиста. (Защо?) Всяко листо заема на диска $8+1 = 9$ бита, а всяко нелисто — 1 бит. Така, цялото дърво ще заема $9.n+1.(n-1) = 10.n-1$ бита. Внимателният читател вероятно е забелязал, че горните алгоритми не запазват *честотата* на срещане на буквите: Тя просто не е необходима.

Запазването на дървото е свързано с допълнително удължаване на кодираното съобщение, поради което алгоритъмът ще бъде ефективен само за достатъчно големи *файлове*. За да се справят с това неудобство, повечето комерсиални продукти при кодиране на малки файлове използват *статично* Хъфманово кодиране, при което се работи с фиксирана честота на срещане на всяка от буквите, независеща от съдържанието на входното съобщение. Основен проблем при

подобен подход е отчитането на типа текст. Така например статичното дърво на Хъфман за стандартен текстов файл на български език ще съдържа достатъчно близо до върха буквата "*а*", тъй като тя е най-често срещаната (*виж таблица 10.4.1а.*). От друга страна при кодиране на програма на Си най-често срещан би могъл да бъде символът ";". Някои "умни" кодиращи програми се опитват да познават типа на файла по разширението му.

Друг проблем при кодирането по Хъфман е, че се работи с битове. За съжаление, при практически всички разпространени днес машини най-малката адресируема единица е байтът, т. е. не е възможно в паметта или на диска да бъде записан или прочетен един бит. За щастие, всички машини дават възможност за манипулация над битовете в рамките на един байт или дума. Това дава възможност за изграждане на буфер, в който да се поставят един по един постъпващите за запис на диска битове. Когато буферът се напълни, се записва на диска. Аналогично, при четене от диска се чете по един байт (или дума), след което последователно се извличат нужните битове. При изчерпване на битовете от буфера се извършва ново четене от диска.

Тук обаче възниква нов проблем: Как да се определи краят на файла? Действително, тъй като записът на диска става побайтово, последният байт евентуално ще бъде полупразен: няма да се използват всичките му 8 бита. За съжаление, декодиращата програма няма как да разбере колко от битовете са валидни. Едно възможно решение е в дървото да се включи специален допълнителен символ за край на файл с честота на срещане 1. Това обаче ще доведе до промяна структурата на дървото, а оттам и до влошаване ефективността на кодирането. Ето защо обикновено вместо това се запазва дължината на кодираното съобщение. Така, декодиращата програма знае точно колко бита да декодира. Още по-добро решение е да не се пази дължината, а само съответният ѝ остатък по модул 8, при което се пести памет: За запазване на *дължината* на съобщението може да бъдат необходими няколко байта, докато за запазване на *остатъка* винаги ще бъде достатъчен един байт: всяко число между 0 и 7 може да се запише с 3 бита.

На теория кодирането по Хъфман ни дава най-добрия код, основан на побуквено кодиране, като средната му ефективност при текстови файлове е от порядъка на 20-25%. На практика обаче, се оказва, че разпространените комерсиални компресиращи програми успяват да постигнат много повече. Привидният парадокс се крие в неподходящия избор на *азбука*, над която се извършва кодирането по Хъфман. Как да изберем азбуката, така че кодирането да бъде най-ефективно? За съжаление, за момента не е известен ефективен начин за това.

Следва да се отбележи, че дори най-добрите комерсиални програми постигат доста скромна ефективност върху истински случайни файлове, при които, както и да се избира азбуката, буквите ѝ се срещат с приблизително еднаква вероятност, например при изпълним файл (за *DOS/Windows*).

Задачи за упражнение:

1. Каква е максималната височина на дървото на Хъфман? Балансирано ли е?
2. Да се докаже, че кодът, построяван от алгоритъма на Хъфман, е префиксен.
3. Да се сравнят алгоритмите на Хъфман и на Шенън-Фано.
4. Да се модифицират предложените програмни реализации така, че да намират всички кодове на Хъфман за дадения източник.
5. Да се построи кодът на Хъфман за следните вероятности:

	a	*b*	*c*	*d*	*e*	*f*
а)	0,70	0,10	0,10	0,05	0,03	0,02
б)	0,10	0,20	0,20	0,10	0,20	0,20
в)	0,30	0,30	0,20	0,15	0,05	0,00
г)	0,17	0,17	0,17	0,17	0,16	0,16

6. Да се реализира кодиране на Хъфман без използване на пирамида и да се сравни по скорост с предложената реализация.
7. Да се определи сложността на последната реализация по време и памет.

8. Нека всички честоти на срещане на буквите за дадено входно съобщение са различни. Може ли да се твърди, че съответното дърво на Хъфман е единствено?

9. Какво ще се получи, ако се разбие кодирано по Хъфман съобщение на последователности от по 5 бита, след което отново се приложи Хъфманово кодиране?

10. Да се напише декодираща програма за Хъфмановото компресиране.

10.4.3. Обобщен алгоритъм на Хъфман

Идеята на разгледания по-горе класически двоичен Хъфманов код може да се обобщи по естествен начин за построяване на троичен, четвъртичен и изобщо n-ичен Хъфманов код. Да разгледаме задачата за намиране на оптимален троичен код, т. е. съставен не от две, а от три различни цифри (например 0, 1 и 2), по зададена вероятност за срещане на всеки от символите от входното съобщение. Как да построим този код? Оказва се, че алгоритъмът на Хъфман работи и в този случай, разбира се, при някои допълнителни условия. За целта се изгражда *тернарно* дърво, т. е. всеки негов връх има до 3 наследника включително. Подобно на двоичното дърво, където имахме ляв и десен наследник, и тук за наследниците има фиксирана наредба. Ще ги наричаме: *ляв*, *среден* и *десен*. Как се изгражда дървото? На всяка стъпка се избират *трите букви* с най-малка вероятност на срещане и се обединяват в нова обобщена буква с вероятност, равна на сумата от вероятностите им. Процесът продължава до обединяване на всички букви в обща супербуква с вероятност 1.

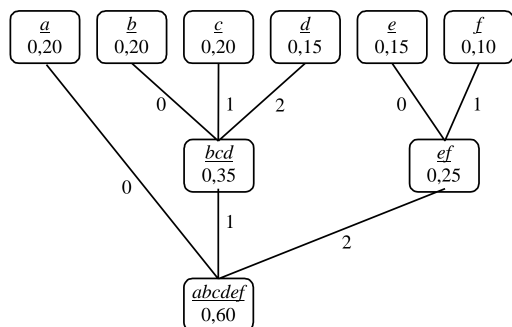
Тъй като на всяка стъпка се извършва обединяване на три (обобщени) букви в една обобщена супербуква, то броят на буквите намалява с 2. В случай на четен брой букви, на последната стъпка ще останат само две букви. Това е доста неприятно, тъй като ни лишава от фундаментален клон на дървото, а оттук евентуално и от някои ценни къси кодове. Очевидно в общия случай такова дърво *няма да бъде оптимално*. Не е трудно да се забележи, че колкото по-далеч от корена е разположено двойното разклонение, толкова повече къси кодове ще имаме на разположение. Остава да съобразим, че алгоритъмът на Хъфман строи дървото от листата към корена. Тогава за получаване на оптимален код на първата стъпка следва да се изберат двете най-малко вероятни букви, при четно n, и трите най-малко вероятни букви, при нечетно n. На всяка следваща стъпка се избират трите най-малко вероятни букви.

Как се получава кодът? Аналогично на двоичния случай: Проследява се пътят от корена до съответното листо. Всеки път, когато преминаваме към ляв наследник пишем 0, към среден — 1, а към десен — 2. Ще илюстрираме метода върху същия пример, върху който илюстрирахме класическото двоично кодиране по Хъфман, *виж фигура 10.4.3*.

За кодовете на буквите получаваме: $a = 0$, $b = 10$, $c = 11$, $d = 12$, $e = 20$ и $f = 21$. Очевидно кодът е разделим (истинските символи са в листата) и има същите свойства като двоичния Хъфманов код. Да пресметнем средния брой троични битове за кодиране на някоя буква (цената на така конструирания троичен код):

$$L = 0{,}20.1 + 0{,}20.2 + 0{,}20.2 + 0{,}15.2 + 0{,}15.2 + 0{,}10.2 = 1{,}8$$

Оказва се, че всяка буква се кодира средно с 1,8 *троични* бита. Сравнено с 2,6 при *двоичния* Хъфманов код. Ясно е, че с увеличаване броя на цифрите този показател ще се подобрява. Следва да обърнем внимание обаче, че тези битове за *троични*!



Фигура 10.4.3. *Тернарно* кодиращо дърво на Хъфман за източника: (*a*:0,20), (*b*:0,20), (*c*:0,20), (*d*:0,15), (*e*:0,15), (*f*:0,10).

Как изглеждат нещата при *k*-ичен код? Оказва се, че алгоритъмът на Хъфман строи оптимални кодове и в този случай. На всяка стъпка се избират *k*-те най-малко вероятни букви и се обединяват в обща супербуква. Първата стъпка евентуално прави изключение по същите съображения, както при 3-знаковия случай. Тук се избират *s*-те най-малко вероятни букви, като *s* е това число измежду 2, 3, ..., *k*, за което $k-1$ дели $n-s$. Кодът, съпоставен на всяка буква, се получава при проследяване на пътя от корена и съпоставяне на съответна цифра, съгласно някаква предварително фиксирана наредба на наследниците. Например, на първия се съпоставя 0, на втория — 1, на третия — 2, ..., на *k*-тия — $(k-1)$.

Задачи за упражнение:

1. Защо в случай на нужда от двойно разклонение, е за предпочитане да бъде възможно най-близо до листата?
2. Какви проблеми възникват при използване на троичен Хъфманов код и как биха могли да се решат?
3. Следва ли да се очаква подобрение при използване на троичен Хъфманов код вместо двоичен? Защо? А при четвъртичен?

10.4.4. Код с разделители

Разгледаните по-горе алгоритми на Шенън-Фано (*виж 10.4.1.*) и Хъфман (*виж 10.4.2.*) притежават безспорни предимства: свойството префиксност, позволяващо еднозначност на декодирането, оптималност и почти оптималност съответно, и др. Същевременно обаче, те имат и някои недостатъци, като най-неприятният от тях е *надеждността*. Действително, повреждането на единствен бит може да доведе до невъзможност за декодирането на цялото съобщение, поради неравномерността на кода. Ето защо при предаването на такова съобщение се налага предвиждането на строг механизъм за контрол и евентуална корекция на грешките.

Как да кодираме така, че промяната на единствен бит да засяга не повече от една-две букви? Съществуват ли изобщо кодове с такова свойство? Всички равномерни кодове, в това число *ASCII*, притежават това свойство. При тях повреждането на единствен бит води до повреждането на единствена буква. За съжаление, равномерните кодове не са особено ефективни. Нека разгледаме като пример входно съобщение със следните символи и честоти: (*a*: 0,4), (*b*: 0,2), (*c*: 0,2), (*d*: 0,15) и (*e*: 0,05). При използване на равномерен код всяка буква ще се записва с 3 бита. Същевременно, използвайки кодиране по Хъфман, получаваме кода: (a = 11), (b = 10), (c = 01), (d = 001), (e = 000). Нека пресметнем цената му:

$$L = 2.0,4 + 2.0,2 + 2.0,2 + 3.0,15 + 3.0,05 = 2,2$$

Не бихме ли могли да получим код, съчетаващ висока надеждност и относително добра ефективност, заемащ междинно положение между равномерните кодове и кодирането по Хъфман? Пример за такъв код е така нареченият *код с разделители* (англ. *comma code*). При него кодът на всеки символ завършва с разделител, указващ края му. Това позволява точно определяне на началото на следващия символ. Как се конструира код с разделители? Идеята е проста: Сортираме буквите по вероятност на срещане. На първия символ съпоставяме код 1, на втория — 01, на третия — 001, на четвъртия — 0001 и т.н. Очевидно така конструираният код е префиксен, позволява бързо и еднозначно декодиране и в общия случай е по-ефективен от равномерните кодове. Повреждането на единствен бит води до невъзможност за декодиране на най-много две букви.

За горния пример получаваме кода: a = 1, b = 01, c = 001, d = 0001, e = 00001. Да пресметнем съответната му цена:

$$L = 1.0,4 + 2.0,2 + 3.0,2 + 4.0,15 + 5.0,05 = 2,25$$

Получихме сравнително надежден код с ефективност, близка до тази на кода на Хъфман.

Задачи за упражнение:

1. Префиксен ли е кодът с разделители?
2. Да се реализира програма за кодиране с код с разделители.
3. Да се сравнят кодът с разделители и кодът на Хъфман.

10.4.5. Аритметично кодиране

Кодирането по Хъфман на теория ни дава оптималния побуквен код, основан на честотата на срещане на всеки от символите във *входното съобщение*. На теория то е оптимално. На практика, ефективността му е доста скромна и при текстови файлове най-често варира между 20% и 25%. Същевременно, практически всички комерсиални програми успяват да се справят по-добре от стандартното кодиране по Хъфман, като в повечето случаи успяват да го бият в пъти. Като че ли теорията лъже… По-горе посочихме една от основните причини за скромната ефективност на кодирането по Хъфман: неподходящият избор на азбука. Оказва се, че има и друг проблем. Нека предположим, че искаме да кодираме факс-изображение. Както по-горе вече споменахме (*виж 10.3.7.*) факс-изображенията са черно-бели, т. е. двуцветни. Да предположим, че изображението е 90% бяло. Очевидно кодирането по Хъфман ще съпостави на единия цвят код 0, а на другия — код 1. Получаваме равномерен код, т. е. нулева компресия. На практика, дължината на оптималния код за белия цвят трябва да бъде 0,15 бита, т. е. около шест пъти по-малко. За съжаление, кодирането по Хъфман съпоставя на всяка буква целочислен брой битове, при което не се отчита достатъчно прецизно вероятността за поява на всяка от тях. Как да се справим с проблема? Едно възможно решение може да се търси по посока на смяна на източника. За целта следва да разглеждаме като букви от входната азбука не отделните точки от изображението, а групи от няколко последователни точки. При групи от 8 точки получаваме азбука с 256 символа, позволяваща ни да се възползваме по-добре от високата честота на срещане на белия цвят.

Въпреки това резултатът не винаги ще бъде задоволителен, тъй като избирането на подходящата входна азбука в общия случай е неразрешим проблем. Но дори и да бъде решен за конкретната задача, проблемът остава: Хъфмановото кодиране съпоставя на всяка буква *целочислен* брой битове, поради което не дава потенциално оптималния резултат (макар да дава оптимален *целочислен* код). За съжаление, за да се възползваме по-добре от точните вероятности за срещане на всяка от буквите, е необходимо да можем да им съпоставяме нецелочислен брой битове. Как може да стане това?

Един възможен отговор на въпроса дава *аритметичното кодиране*. Подобно на Хъфмановото кодиране, то също се основава на статистически наблюдения за честотата на срещане на всеки от символите във входното съобщение, но може да постигне значително по-добри резултати, тъй като изобщо се отказва от побуквеното кодиране. Вместо това на *цялото* входно съобщение се съпоставя *едно-единствено дробно число* между 0 и 1 с необходимата точност. Как

става това? Да илюстрираме метода върху някакво конкретно входно съобщение, например: "*АРИТМЕТИКА*".

Първото нещо, което трябва да направим, е да построим статистика на вероятността за поява на всяка буква от входното съобщение. След това, въз основа на тези данни, на всяка буква съпоставяме някакъв подинтервал от вероятностната крива, която по дефиниция е ограничена в интервала [0;1]. Подинтервалите трябва да удовлетворяват следните условия:

1) затворени са отляво и са отворени отдясно, т. е. имат вида [l;r);
2) не се пресичат;
3) обединението им дава интервала [0;1);
4) дължините им се отнасят така, както се отнасят вероятностите за поява на съответните букви.

Буква	**Вероятност на срещане**	**Долна граница**	**Горна граница**
А	20%	0,00	0,20
Е	10%	0,20	0,30
И	20%	0,30	0,50
К	10%	0,50	0,60
М	10%	0,60	0,70
Р	10%	0,70	0,80
Т	20%	0,80	1,00

Таблица 10.4.5а. Начални *дробни* интервали за буквите при кодиране на “АРИТМЕТИКА”.

Таблица 10.4.5а. показва една от възможните таблици с начални интервали за буквите. Ясно е, че тя не е единствена, тъй като бихме могли да разместим буквите. На практика, съществуват 7! = 5040 различни възможни таблици. Няма значение коя от тях ще бъде избрана, стига при декодирането да се използва същата таблица. По-горе сме избрали тази, при която буквите са подредени по азбучен ред.

Символ	**Долна граница**	**Горна граница**
няма	0,000000000	1,000000000
А	0,000000000	0,200000000
Р	0,140000000	0,160000000
И	0,146000000	0,150000000
Т	0,149200000	0,150000000
М	0,149680000	0,149760000
Е	0,149696000	0,149704000
Т	0,149702400	0,149704000
И	0,149702880	0,149703200
К	0,149703040	0,149703072
А	0,149703040	0,149703046

Таблица 10.4.5б. Кодиране на “АРИТМЕТИКА”: *дробни* интервали за всяка буква.

Как става кодирането? Започваме с интервала [0,00; 1,00), като постепенно го стесняваме. Първата буква от входното съобщение принадлежи на интервала [0,00; 0,20). Тогава търсеното число ще лежи в този интервал. Разглеждаме втората буква *Р*. Тя принадлежи на интервала [0,70; 0,80). Тогава нашето дробно число ще лежи между 70% и 80% от *вече определения* интервал [0,00; 0,20). Така получаваме интервала [0,14; 0,16). Следва буквата *И*, принадлежаща на интервала [0,30; 0,50). Отрязваме интервала [0,14; 0,16) отляво на 30%, а отдясно — на 50%, стеснявайки го до [0,146; 0,150). След това постъпва буквата *Т*, на която е съпоставен интервалът [0,80; 1,00). Извършваме съответно ново стесняване на интервала и получаваме [0,1492; 0,1500). *Таблица 10.4.5б.* описва процеса по-ясно.

Процесът продължава до изчерпване на буквите от входното съобщение, при което достигаме до интервала: [0,149703040; 0,149703046). Той определя еднозначно нашето съобщение. Оказва се, че не е необходимо да пазим и двете граници на интервала, а ни е напълно достатъчно да запазим например долната. На практика, всяко число от интервала [0,149703040; 0,149703046) би могло да се счита за код на нашето съобщение. Формално процесът на кодиране би могъл да се опише с помощта на следния програмен код:

```
l = 0.0; r = 1.0;
while (getSymbol(ch)) {
  range = r - l;
  l = l + range * low_range(ch);
  r = l + range * high_range(ch);
}
output(l);
```

А как става декодирането? Нещата протичат аналогично на кодирането. В процеса на кодиране започвахме с интервала, определен от първата буква на входното съобщение, като после непрекъснато го стеснявахме. Така всеки следващ интервал се включваше в предходния. Това означава, че полученият код принадлежи на интервала на първата буква от съобщението. Тогава бихме могли да я изведем на изхода, след което да премахнем ефекта от появата й. Ще получим ново число между 0 и 1, представляващо аритметичния код на останалата част от съобщението. Сега отново можем еднозначно да определим първата буква от този код (т. е. вторият от нашето съобщение), да го изведем на екрана и да премахнем ефекта от него. Получаваме кода на съобщението, без първите му два символа и т.н. Възниква проблемът: Кога да спрем? Оказва се, че това е съществен проблем, тъй като кодът по никакъв начин не ни показва каква е дължината на съобщението. Съществуват два основни метода за справяне с проблема: въвеждане на стоп-символ, указващ края на съобщението, или запазване на дължината на входното съобщение отделно от получения код. В приложената по-долу реализация е избран вторият подход. *Таблица 10.4.5в.* проследява процеса на декодиране на съобщението.

Аритметичен код	**Буква**	**Долна граница**	**Горна граница**
0.149703040	*А*	0,00	0,20
0.748515200	*Р*	0,70	0,80
0.485152000	*И*	0,30	0,50
0.925760000	*Т*	0,80	1,00
0.628800000	*М*	0,60	0,70
0.288000000	*Е*	0,20	0,30
0.880000000	*Т*	0,80	1,00
0.400000000	*И*	0,30	0,50
0.500000000	*К*	0,50	0,60
0.000000001	*А*	0,00	0,20

Таблица 10.4.5в. Декодиране на "АРИТМЕТИКА" по *дробния* код.

Формално процесът на декодиране би могъл да се опише с помощта на следния код:

```
for (i = 0; i < messageLen; i++) {
  symbol = getSymbol(msg);
  writeOut(symbol);
  range = high_range(symbol) - low_range(symbol);
  msg -= low_range(symbol);
  msg /= range;
}
```

Очевидно, аритметичното кодиране ще се справи по-добре от Хъфмановото в случай на двуцветно изображение с 90% вероятност за поява на бял пиксел. Въпреки това от изложеното по-горе изобщо не става ясно защо и дали то е по-добро в общия случай. Да разгледаме съобщението "*БААААААААА*" и съответната му *таблица 10.4.5г.*

Символ	Долна граница	Горна граница
няма	0,0	1,0
А	0,0	0,9
Б	0,9	1,0

Таблица 10.4.5г. *Дробни* интервали на буквите за съобщението "*БААААААААА*".

Да проследим процеса на кодиране

```
Б    0,900000000  1,000000000
А    0,900000000  0,990000000
А    0,900000000  0,981000000
А    0,900000000  0,972900000
А    0,900000000  0,965610000
А    0,900000000  0,959049000
А    0,900000000  0,953144100
А    0,900000000  0,947829690
А    0,900000000  0,943046721
А    0,900000000  0,938742049
```

За кода на съобщението можем да вземем 0,9! Тъй като нулата и десетичната запетая се подразбират, това означава, че успяхме да сведем съобщение от 10 букви до една-единствена цифра! Ясно е, че при нарастване вероятността за срещане на някой от символите степента на компресия ще нараства значително. При кодиране на последователност от 100000 нули с предварително зададена вероятност за срещане на нула 16382/16383 и вероятност за срещане на символ за край на файла 1/16383, е получава *трибайтов* код! Същевременно, кодирането по Хъфман би изисквало най-малко 12501 байта. Разбира се, подобни примери са в значителна степен "нагласени" и рядко се случват на практика. Въпреки това те ни подсказват, че в общия случай следва да се очаква, че аритметичното кодиране ще се държи по-добре от Хъфмановото. Доказано е, че аритметичното кодиране е *оптимално* при предложения модел. По-късно ще видим, че променяйки модела, бихме могли да постигнем още по-висока степен на компресия. На практика, следва да се очаква подобрение от порядъка на 5–10%. Следва примерна реализация на аритметичното кодиране:

```
#include <stdio.h>
#include <string.h>

#define SHOW_MORE
#define MESSAGE "АРИТМЕТИКА"
#define MAX 256

struct {
  double low, high;
} sym[MAX];

unsigned freq[MAX];

void getStatistics(char *mesg) /* Намира броя срещания на всеки символ */
{ unsigned i;

  for (i = 0; i < MAX; i++)
    freq[i] = 0;
```

```
  while ('\0' != *mesg)
    freq[(unsigned char) *mesg++]++;
}
void buildModel(char *mesg) /* Построява модела */
{ unsigned i, cnt, n;
  for (n = strlen(mesg), cnt = i = 0; i < MAX; i++) {
    sym[i].low = (double) cnt/n;
    cnt += freq[i];
    sym[i].high = (double) cnt/n;
  }
}

void printModel(void)
{ unsigned i;
  printf("\n             ГРАНИЦА");
  printf("\nСИМВОЛ    ДОЛНА   ГОРНА");
  for (i = 0; i < MAX; i++)
    if (freq[i])
      printf("\n%4c     %1.4f  %1.4f", i,sym[i].low,sym[i].high);
}

double arithmeticEncode(char *mesg) /* Извършва аритметично кодиране */
{ double range, low, high;
  low = 0.0, high = 1.0;
  while ('\0' != *mesg) {
    range = high - low;
    high = low + range * sym[(unsigned char) *mesg].high;
    low = low + range * sym[(unsigned char) *mesg].low;
#ifdef SHOW_MORE
    printf("\n%c    %1.9f  %1.9f",*mesg,low,high);
#endif
    *mesg++;
  }
  return low;
}

char getSymbol(double encMsg)
{ unsigned i;
  for (i = 0; i < MAX; i++)
    if (sym[i].low <= encMsg && sym[i].high > encMsg)
      break;
  return (char) i;
}

void arithmeticDecode(double msg, unsigned msgLen) /* Декодиране */
{ double range;
  unsigned char ch, i;
  for (i = 0; i < msgLen; i++) {
    ch = (unsigned char) getSymbol(msg);
#ifdef SHOW_MORE
    printf("\n%c    %1.9f",ch,msg);
#else
    putc(ch,stdout);
#endif
    range = sym[ch].high - sym[ch].low;
    msg -= sym[ch].low;
    msg /= range;
  }
}
```

```
int main(void) {
  double code;
  printf("\n\nИзходно съобщение: %s",MESSAGE);
  getStatistics(MESSAGE);
  buildModel(MESSAGE);
#ifdef SHOW_MORE
  printModel();
  printf("\nНатиснете <<ENTER>>"); getchar();
#endif
  code = arithmeticEncode(MESSAGE);
  printf("\nКодът на съобщението е: %1.8f",code);
  printf("\nДекодиране: ");
  arithmeticDecode(code,strlen(MESSAGE));
  return 0;
}
```

🖫 arithmet.c

Резултат от изпълнението на програмата:

```
Изходно съобщение: АРИТМЕТИКА
             ГРАНИЦА
СИМВОЛ    ДОЛНА   ГОРНА
   А      0.0000  0.2000
   Е      0.2000  0.3000
   И      0.3000  0.5000
   К      0.5000  0.6000
   М      0.6000  0.7000
   Р      0.7000  0.8000
   Т      0.8000  1.0000
Натиснете <<ENTER>>
А    0.000000000  0.200000000
Р    0.140000000  0.160000000
И    0.146000000  0.150000000
Т    0.149200000  0.150000000
М    0.149680000  0.149760000
Е    0.149696000  0.149704000
Т    0.149702400  0.149704000
И    0.149702880  0.149703200
К    0.149703040  0.149703072
А    0.149703040  0.149703046
Кодът на съобщението е: 0.14970304
Декодиране:
А    0.149703040
Р    0.748515200
И    0.485152000
Т    0.925760000
М    0.628800000
Е    0.288000000
Т    0.880000000
И    0.400000000
К    0.500000000
А    0.000000001
```

Предложената програма представлява само демонстрация на аритметичното кодиране и за използването ѝ за практически цели читателят ще трябва да се потруди сам и да я модифицира по подходящ начин. На тези, които преценят, че не си струва усилията да го направят сами, препоръчваме [*Стенли-1998*], където е дадена работоспособна реализация.

На пръв поглед може би изглежда, че описаният по-горе процес на аритметично кодиране е сложен и съвършено неподходящ за практически цели. Действително, кодирането и декодирането изискват тежки аритметични операции над числа с плаваща запетая, което несъмнено ще доведе до скромна ефективност на метода. От друга страна, реализация, подобна на горната, едва ли би могла да се нарече универсална, макар теоретически да е такава. С изключение на достатъчно къси съобщения от порядъка на 15-ина символа, както и на някои други тривиални случаи, реализации, подобни на горните, непременно ще страдат от *загуба на точност*. Максималната дължина на мантисата при числа с плаваща запетая при повечето разпространени днес стандартни реализации е 80 бита, т. е. 10 байта, което поставя горната граница на достижимата точност, а оттам и на дължината на съобщенията, които ще можем да кодираме. Освен това на дневен ред винаги ще стои въпросът за съвместимостта между числените типове, при различни платформи. За щастие 80-битовият тип числа с плаваща запетая е възприет като международен *IEEE* стандарт, което гарантира съвместимост между различните му реализации. Въпреки това при по-дълги съобщения предлаганата от него точност ще се окаже недостатъчна. Едно възможно решение е ръчната реализация на съответните аритметични операции, но това едва ли може да се счита за разумен подход. Друг проблем, който трябва да се има предвид е свързан са междуплатформената преносимост на кодираното съобщение. Проблемът е в начина на адресиране на байтовете: на някои платформи най-старши е най-левият байт, а при други — най-десният.

При по-внимателен поглед върху нещата, се оказва, че картината не е толкова “апокалиптична” и аритметичното кодиране може да се преработи сравнително лесно така, че да използва само целочислени операции, основани на стандартните за съвременните компютри 16 и 32-битова машинна аритметика. По-долу ще покажем, как би могло да се стане това.

Ключов елемент на предложения алгоритъм представлява пресмятането на горната и долната граници на интервала на всяка стъпка. На първата стъпка на алгоритъма тези граници са съответно 0 и 1. Изхождайки от чисто аритметични съображения, можем да считаме, че числата 1 и 0,99999999… са равни помежду си. Тогава за началния интервал получаваме: [0;0,99999…). В двоична бройна система ще получим аналогичен резултат, като деветките ще бъдат заменени с единици. Как да преминем от числа с плаваща към числа с фиксирана запетая? Ще направим още едно наблюдение: На всяка стъпка полученият интервал е число между 0 и 1, т. е. можем спокойно да разглеждаме само дробната част `xxxx` на числата, подразбирайки, че вляво от тях стои 0, т. е. подразбирайки числото `0,xxxx`. Нека предположим, че работим с хипотетична машина с десетична машинна дума, съдържаща числа от 0 до 99999. Тогава, вземайки предвид, че 1 = 0,99999…, за началните граници на разглеждания интервал получаваме съответно 00000 и 99999. Инициализацията на първоначалната таблица не представлява проблем: просто се премахва цялата част, след което дясната граница се намаля с 1. Така, за вече разгледания по-горе пример *"АРИТМЕТИКА"* получаваме *интервалната таблица 10.4.5д*.

Символ	**Вероятност на срещане**	**Долна граница**	**Горна граница**
А	20%	00000	19999
Е	10%	20000	29999
И	20%	30000	49999
К	10%	50000	59999
М	10%	60000	69999
Р	10%	70000	79999
Т	20%	80000	99999

Таблица 10.4.5д. *Целочислени* интервали на буквите за съобщението "*АРИТМЕТИКА*".

При определяне дължината на даден интервал, следва да се отчита, че дясната му граница е намалена с 1. Сега остава да направим основното си наблюдение: Ако началните цифри на двете граници съвпаднат веднъж, те ще продължават да съвпадат на всяка следваща стъпка. Действи-

телно, интервалите могат само да се стесняват, при което всеки следващ ще се влага в предходния. Ако първите k цифри на границите на интервала съвпаднат на някоя стъпка, то те ще съвпадат и на следващата. Противното може да бъде налице, само ако следващият интервал се разшири, което е невъзможно. Това ни дава основание да запишем съвпадащата начална цифра във входното съобщение, след което да изместим цифрите в регистрите, съдържащи двете граници на една позиция наляво, допълвайки отдясно с подразбиращата се цифра. Изместването наляво в десетична бройна система става с умножаване по 10, а в двоична — с умножение по 2. Изобщо, за произволна бройна система с основа p умножението по p става с долепяне на 0 отдясно към цифрите на числото. При разработка на реална програма е удобно да се използва поразредната операция <<. Що се отнася до подразбиращата се цифра, с която следва да се допълнят цифрите на числото (която следва да се прибави към числото след изместването), то очевидно за долната граница тя ще бъде 0, а за горната — 9 (в двоична бройна система ще бъде 1). Нека проследим така модифицирания процес на кодиране върху нашето входно съобщение *"АРИТМЕТИКА"*. Ще отбележим, че след изчерпване на входните символи следва да изведем още няколко цифри към изхода, така че да получим началния интервал (*виж таблица 10.4.5е.*).

За съжаление, предложеното подобрение също има някои недостатъци. Проблем отново е точността. Оказва се, че описаната схема при някои по-особени обстоятелства може да доведе до невъзможност за правилно кодиране на съобщението. Проблемът отново е в точността, с която извършваме пресмятанията, и възниква при силно доближаване на двете граници на интервала. Ако в даден момент лявата и дясната граница се окажат например 39999 и 40000 съответно, кодирането няма да може да продължи правилно. От една страна двете числа се намират изключително близо едно до друго, поради което постъпването на нито един входен символ няма да може да окаже влияние, а от друга — все пак нямаме съвпадащи първи цифри! Как да се справим с такъв проблем? Не е трудно да се види, че веднъж достигнали до такова положение, не можем да направим нищо. Какво ни остава тогава? Да не го допускаме! За целта ще модифицираме функцията за изхвърляне на цифра. Очевидно, проблемът възниква само при съседни старши цифри, последвани от 0 и 9 съответно. В случай на еднакви цифри ще процедираме така, както сме действали досега. В случай на съседни цифри обаче (например 4 и 5) "ще изхвърляме" вторите цифри, изместваийки десните така, че да запълнят мястото, и допълвайки отдясно с подразбиращата се цифра. Въвеждаме си брояч, указващ броя на изхвърлените цифри, без да запазваме самите цифри. Всеки път, когато най-старшите цифри на двата интервала се окажат съседни, ще извършваме подобна операция и ще увеличаваме брояча. Когато накрая съвпаднат, ще изведем на изхода общата им стойност, последвана от изхвърлените цифри. Броят на изхвърлените цифри ще се съдържа в брояча, а самите цифри ще имат еднаква стойност, която ще бъде 0 или 9 в зависимост от това, дали най-левите цифри са станали равни при по-ниската или при по-високата стойност.

Символ/ Операция	**Долна граница**	**Горна граница**	**Дължина**	**На изхода**
Няма	00000	99999	100000	0,
А	00000	19999	20000	0,
Р	14000	15999	2000	0,
изместване	**40000**	**59999**	**20000**	**0,1**
И	46000	49999	4000	0,1
изместване	**60000**	**99999**	**40000**	**0,14**
Т	92000	99999	8000	0,14
изместване	**20000**	**99999**	**80000**	**0,149**
М	68000	75999	8000	0,149
Е	69600	70399	800	0,149
Т	70240	70399	160	0,149
изместване	**02400**	**03999**	**1600**	**0,1497**
изместване	**24000**	**39999**	**16000**	**0,14970**

И	28800	31999	3200	0,14970
К	30400	30719	320	0,14970
изместване	**04000**	**07199**	**3200**	**0,149703**
изместване	**40000**	**71999**	**32000**	**0,1497030**
А	40000	45999	2000	0,1497030
изместване	**00000**	**19999**	**20000**	**0,14970304**
изместване	**00000**	**99999**	**100000**	**0,149703041**

Таблица 10.4.5е. Кодиране на "АРИТМЕТИКА": *целочислени* интервали за всяка буква.

Процесът на декодиране също изисква някои модификации. Така например, границите следва да се изменят по същия начин, както при кодирането, с аналогични тестове за недостатъчна точност и др. Входното съобщение трябва да се чете на части. Най-добре е за целта да се използва подходящо буфериране. При определянето колко цифри от входното съобщение да се разглеждат едновременно трябва да се спазва условието: разгледани като едно число, те да попадат в интервала между долната и горната граница. Освен това не бива да се забравя, че дясната част на интервала е скъсена с 1. [*Стенли-1998*].

Задачи за упражнение:

1. Да се сравни аритметичното кодиране с кодирането на Хъфман (*виж 10.4.2.*). Винаги ли аритметичното кодиране ще постига по-добра степен на компресия? Защо?
2. Да се реализира работоспособен вариант на аритметичното кодиране.

10.5. Адаптивно компресиране

Разгледаните по-горе статистически методи на кодиране бяха статични. Действително, независимо от това дали се вземаха предвид вероятностите за поява на всяка от буквите в конкретното съобщение или се работеше с едни и същи фиксирани вероятности за всякакви входни съобщения, все едно, кодът се фиксираше преди началото на кодирането и оставаше неизменен по време на процеса.

Основен проблем при статичните методи с *фиксирана* вероятност е разминаването между *очакваната* и *действителната* вероятност на поява на отделните букви. Така се оказва, че при кодиране на често срещани букви се използват по-дълги кодове, а при по-рядко срещани — по-къси. Подобна ситуация възниква дори при отчитане на вероятностите за поява на всяка буква в конкретното съобщение. Нека предположим, че дадена буква се среща в някаква начална част на съобщението с голяма вероятност, докато в останалата част вероятността ѝ за поява е достатъчно малка. При използване на фиксиран метод на кодиране, основан на глобалната вероятност на поява, в началото на съобщението ще бъде използван достатъчно дълъг код, което локално погледнато е неефективно.

За постигане на по-добра статистически побуквена компресия е необходимо използваният код да се изменя в процеса на кодиране, отчитайки действителната честота на срещане на буквите. Едно възможно решение е *адаптивното компресиране*. При него се фиксира някакъв начален код за всяка буква, като в процеса на компресиране кодовете за отделните букви могат да се разменят така, че да се отчитат локалните честоти на срещане. Наборът от кодиращи последователности не се променя, а само се пермутират елементите на таблицата на съответствието между буква и побуквен код. Най-често началната таблица се инициализира с код на Хъфман, построен въз основа на честотата на срещане на буквите от входната последователност.

При статичните методи на побуквено компресиране кодирането се извършва въз основа на таблица на съответствието между буква и код. При адаптивното компресиране също се използва подобна таблица, разширена с още една колона, съдържаща честотата на срещане на отделните букви. В процеса на кодиране таблицата се поддържа сортирана в намаляващ ред по тази колона. При всяко търсене на буква в таблицата се извършва актуализиране на честотата ѝ на

срещане. Обосновката на това е, че търсенето е възникнало, защото буквата се е срещнала още веднъж в текста. Следва последователно сравняване на тази честота с честотата на предходния елемент дотогава, докато буквата не заеме позицията, съответстваща на новата ѝ честота. В случай на равенство на честотите, буквата се поставя възможно по-нагоре в таблицата. В процеса на размяна се разменят *само* буквите заедно с честотите, като кодовете остават по местата си. Така по-често срещаните букви ще имат по-къс код. Ще илюстрираме метода върху съобщението: *bdaacacabab*, *виж таблици 10.5а., 10.5б., 10.5в., 10.5г. и 10.5д.*

И така, на всяка стъпка таблицата се изменя заедно с изменението на честотата на срещане на отделните букви. Същевременно, за да бъде успешно декодирането, във всеки един момент трябва да бъде известна текущата таблица на съответствията. Очевидно запазването на всички таблици е невъзможно, тъй като това би довело до значително нарастване на дължината на кода. Оказва се, че не е и необходимо. Действително, ако е известна началната таблица, можем да получаваме всички останали по начина, по който ги получавахме и при кодирането: поддържа се допълнителната колона с честотите на срещане, която на всяка стъпка се актуализира, като това води до евентуално разместване на някои от редовете и т.н.

Понякога може да се окаже, че дължината на кодираното съобщение е достатъчно голяма. Впрочем, теоретично погледнато, всички разгледани по-рано статистически методи на кодиране можеха да работят върху безкрайни последователности (при фиксирана таблица). Тъй като адаптивното кодиране поддържа броячи на честотата на срещане на всяка от буквите, които на практика могат да приемат краен брой стойности, то няма да може да кодира безкрайни съобщения. Един възможен подход за справяне с проблема е при достигане на максимална стойност на някой от броячите, да се извършва целочислено разделяне на 2 на цялата колона с честотите. Основен недостатък тук е необходимостта от проверка за препълване. Това може лесно да се избегне, като за целта е достатъчно да се следи броят на постъпващите букви и да се извършва деление на 2, когато се достигне максималната стойност, побираща се в броячите. В този случай обаче ще се извършва много по-често деление на 2.

буква	**честота**	**код**
a	0	0
b	0	10
c	0	110
d	0	111

Таблица 10.5а. Адаптивно кодиране на “*bdaacacabab*”: начална таблица.

буква	**честота**	**код**
b	1	0
a	0	10
c	0	110
d	0	111

Таблица 10.5б. Адаптивно кодиране на “*bdaacacabab*”: постъпва *b*, на изхода — 10.

буква	**честота**	**код**
d	1	0
b	1	10
c	0	110
a	0	111

Таблица 10.5в. Адаптивно кодиране на “*bdaacacabab*”: постъпва *d*, на изхода — 111.

буква	**честота**	**Код**
a	1	0

d	1	10
b	1	110
c	0	111

Таблица 10.5г. Адаптивно кодиране на "*bdaacacabab*": постъпва *a*, на изхода — 111.

буква	честота	код
a	2	0
d	1	10
b	1	110
c	0	111

Таблица 10.5д. Адаптивно кодиране на "*bdaacacabab*": постъпва *a*, на изхода — 10.

Задачи за упражнение:

1. Да се предложи начин за намаляване на броя на размените при обновяването на таблицата.
2. Да се реализира описаният алгоритъм.
3. Да се сравни адаптивното компресиране със стандартното компресиране по Хъфман. Наблюдава ли се подобрение?
4. Колко често следва да се извършва деление на две, така че от една страна да се дава възможност за натрупване на достатъчна статистика, а от друга — да се отчитат локалните особености?

10.5.1. Адаптивно компресиране по Хъфман

Един от основните недостатъци на Хъфмановото кодиране във вида, в който беше разгледано в *параграф 10.4.2.*, беше начинът на построяване на кодиращото дърво, а оттам и на самия код. Действително, изграждането му изисква предварително изграждане на таблица на честотите (вероятностите) на срещане на всяка от буквите от входното съобщение. Бяха посочени два основни пътя за изграждане на такава таблица: предварително преглеждане на подлежащото на кодиране съобщение и натрупване на необходимата статистическа информация или използване на фиксиран код, зависещ евентуално от *типа* или *дължината* на съобщението, но не и от конкретното му *съдържание*. Предимство на първия подход беше получаването на по-добър код, а оттук и на по-добра степен на компресия, а на втория – отпадането на необходимостта от предаване на кода. За съжаление, и двата метода не отчитат евентуалните *локални особености* на входното съобщение. На практика, се оказва, че честотите на срещане на отделните букви в някои части на съобщението се отличават от средностатистическите за цялото съобщение. Това прави използването на един и същ код неефективно. По-горе беше предложено едно добро решение: динамична промяна на таблицата при фиксиран код на Хъфман. Сега ще разгледаме друг подход, при който съобщението се разбива на отделни части, за всяка от които се прилага различен код на Хъфман. Подобен подход извежда на преден план два основни проблема: 1) Как да се определят броят и границите на отделните части? и 2) Как да се предпазим от прекомерно нарастване на кода в резултат на предаването на повече кодиращи таблици?

Ще предложим специален адаптивен метод, който се справя успешно и с двата проблема. Идеята е кодирането да започва с някакъв начален код, който впоследствие да се променя динамично заедно с изменението на вероятностите за срещане на буквите от входното съобщение. Идеята на адаптивното компресиране е универсална и лесно приложима към практически всеки статистически алгоритъм на компресиране. Една адаптивна компресираща схема изглежда най-общо така:

```
initModel();
while(!eof(input)) {
```

```
  sym = getSymbol(input);
  encSym = encode(sym);
  writeOut(output, encSym);
  updateModel(sym);
}
```

И съответно декомпресиращата схема:

```
initModel();
while(!eof(input)) {
  sym = getSymbol(input);
  decSym = decode(sym);
  writeOut(output, decSym);
  updateModel(sym);
}
```

Синхронизацията между кодирането и декодирането се осъществява посредством използване на общи функции за изграждане и поддържане на модела: `initModel()` и `updateModel()`. Какво точно се крие зад тях зависи от конкретния алгоритъм. По-долу ще се спрем на по-подробното описание на прилагането на адаптивни техники при Хъфманово кодиране.

Реализацията на адаптивно Хъфманово кодиране е изправена пред сериозен проблем. На пръв поглед изглежда, че тъй като на всяка стъпка моделът се актуализира, то всеки път ще трябва да строим ново дърво на Хъфман, съответстващо на новите честоти на буквите. Разбира се, това не е точно така. На всяка стъпка се променя честотата на *единствена* буква, като това не винаги ще води до нужда от промяна на структурата на дървото, т. е. в част от случаите то ще продължи да бъде Хъфманово, като ще се налага единствено увеличаване на съответните *броячи* по пътя от корена до листото, съдържащо буквата. За съжаление, в останалите случаи ще се налага промяна и на *структурата* на дървото.

Как да се справим по-лесно с реорганизирането на дървото? Дървото на Хъфман има точно два наследника за всяко нелисто. Да си припомним как го изграждахме: На първата стъпка се вземат двете най-редки букви и се обединяват в обща супербуква. На следващата стъпка отново се избират двете най-редки букви (Едната евентуално може да бъде супербуквата от предишната стъпка.). Те също се обединяват в супербуква и т.н. Процесът продължава до обединяване на всички букви в единствена супербуква. Да въведем обща номерация на буквите и супербуквите. Най-рядката обявяваме за първа по ред, а следващата по големина — за втора. Това са именно буквите от първата стъпка на алгоритъма по изграждането на дървото. На следващата стъпка се избират отново двете най-редки букви, които получават номера съответно 3 и 4 и т. н. Коренът на дървото, т. е. последната супербуква получава най-голям номер, съвпадащ с общия брой $2n-1$ на върховете в дървото.

Оказва се, че така въведената номерация би могла да бъде изключително полезна и да ни позволи да реорганизираме дървото на сравнително ниска цена. Върховете на дървото ще съхраняваме в масив, в който индексът на всеки връх ще съвпада с присвоения му номер съгласно описаната номерация. Т. е. получаваме сортиран във възходящ ред масив. Реорганизирането при постъпване на нова буква от входното съобщение ще протича на две стъпки. Първо се увеличава броячът на буквата, както и на върховете по целия път до корена. Това може да бъде постигнато лесно, ако за всеки връх съхраняваме указател към родителя му. В процеса на актуализиране на броячите може да се случи наредбата на елементите на масива да се наруши. В този случай се налага извършването на подходящи размени така, че масивът отново да се окаже сортиран. Това означава размяна на проблемния елемент с минималния елемент, който е по-голям от него. Подобна стратегия ще ни гарантира минимален брой размени и поддържане на Хъфмановото дърво на сравнително ниска цена. Оставяме на читателя да се убеди сам, че предложената схема работи правилно.

Процесът на актуализация на дървото би могъл да се раздели най-общо на две основни стъпки. На първата се извършва просто обхождане, започващо от листото, съдържащо текущата

буква от входното съобщение, и завършващо в корена, при което се увеличава броячът във всеки връх по пътя. Очевидно една такава операция може да доведе до нарушаване на Хъфмановите свойства на дървото, т. е. нарушаване на свойството, въз основа на което беше въведена горната номерация на върховете. В такъв случай се налага прилагане на съответна операция по възстановяването му. Идеята е проста: за всеки връх, при който сме извършили увеличаване, проверяваме, дали стойността на брояча му не е по-голяма от тази на върха, вдясно от него. Ако това е така, извършваме последователни размени на елемента с дясностоящия, докато не попаднем на подходящата позиция. (На практика, можем да минем и с единствена размяна, като преди това определим точната позиция, на която трябва да отиде нашият връх, с последователни сравнения, но без размени.)

И така, след като видяхме как можем да реорганизираме дървото в движение, като че ли нещата вече са ясни. Остава обаче да бъдат разрешени още два съществени проблема: Как да се третират излишните букви и какво да се прави при препълване? Да се спрем по-подробно на първия проблем. При разглеждането на класическото кодиране по Хъфман изрично заявихме, че в дървото влизат само буквите, които действително се срещат във входното съобщение, т. е. тези с ненулева вероятност (честота) на срещане. За съжаление, подобна стратегия тук е трудно приложима, тъй като дървото за всяка стъпка е различно. Нека предположим, че множеството на буквите от входното съобщение ни е предварително известно. Тогава можем да започнем да строим дърво на Хъфман, включващо всички тези букви, като на началните стъпки повечето от тях ще имат нулева честота на срещане, което ще води до понижаване на ефективността на Хъфмановия код. Добре би било вместо това кодирането да започне от празното дърво, като при постъпване на несрещана до момента буква тя се прибавя към дървото по подходящ начин. Това би довело до по-ефективен код. Основният проблем пред този подход възниква при декодирането: Как декодиращата функция да бъде уведомена за това, че в дървото е постъпила нова буква, както и какъв е кодът й? Идеята е в първоначалното дърво на Хъфман да се предвиди специална буква с честота на срещане 0, която ще наричаме *escape*. Когато кодиращата функция прочете несрещана до момента буква l от входното съобщение, тя ще предаде на изхода кода на *escape*, последван от кода на l. Когато декодиращата функция срещне *escape*, тя ще знае, че следва некодирана буква, и ще предприеме съответни действия по допълване на собственото си дърво.

Вторият основен проблем при адаптивното кодиране на Хъфман е свързан с вероятността от препълване на броячите на честотата на срещане в отделните възли. Всъщност директно застрашен от препълване от такъв тип е коренът. Оказва се, че съществува опасност и от друг вид препълване. Обикновено при обхождането на дървото от листата към корена, се използва променлива от тип *unsigned*, в която се съхраняват битовете, получени по пътя към корена, т. е. кодът на буквата. Тази променлива може да се препълни преди препълването на корена. Едно добро решение на проблема е стойностите на честотите на срещане да се делят например на 2, при вероятност за препълване. За съжаление, делението на две е свързано с поне два нови проблема. От една страна трябва да се съобщи по подходящ начин на декодиращата програма, че е извършено деление. От друга страна делението може да доведе до нужда от реорганизиране на дървото. Проблемът е в това, че стойностите, записани в нелистата, следва да се пресметнат наново, тъй като простото им разделяне на 2 може да доведе до неправилен резултат. Действително, ако двата наследника на даден връх имат нечетна честота на срещане, то при делението им на две ще се загуби общо цяла единица, докато при делението на стойността в родителя им няма да има никаква загуба. Възниква необходимост от цялостно преизчисляване на стойностите във върховете. За съжаление, това не е всичко и може да се наложат редица други промени в структурата на дървото, поради което може би най-доброто решение е, в случай на деление на 2, то да се построи изцяло наново: Все пак става въпрос за относително рядка операция. Един страничен ефект на делението на 2 е намаляването на относителната стойност на старите статистически данни в сравнение с новите. Действително, оттук нататък те се вземат с коефициент 1/2, което води до повишаване на ефективността, тъй като локалните особености на входното съобщение се отчитат по-добре. А нали именно това беше основната идея на адаптивното компресиране.

Задачи за упражнение:

1. Да се реализира описаният алгоритъм.
2. Да се сравни адаптивното компресиране с използване на код на Хъфман от *10.5.* с адаптивното компресиране по Хъфман, предложено тук. Кое отчита по-добре локалните особености?
3. Да се оценят предимствата и недостатъците на използването на *escape* символ.

10.5.2. Модели на Марков

Разглежданите до момента модели на побуквено кодиране не използват контекста, в който се среща буквата във входното съобщение. Оказва се обаче, че това може да бъде от значение и да подобри нивото на компресия. Да разгледаме като пример стандартен текстов файл в *DOS/Windows*. Както е известно, всеки ред на файла завършва с двойка символи с *ASCII* кодове 13 и 10 съответно. Вероятността за срещане на символ с *ASCII* код 10 обикновено е в рамките на 1–5%. Ако обаче предходният символ е бил 13, то вероятността следващият го символ да бъде 10 става вече 100%! (Напомняме, че става въпрос за стандартен *текстов* файл.) Биха могли да се приведат множество подобни примери, когато отчитането на левия контекст води до силно отклонение от *безусловната* (*априорна*) вероятност за срещане на дадена буква. По-долу ще видим как, отчитайки *левия контекст*, можем да повишим степента на компресия.

Всички разгледани до момента статистически алгоритми работеха с таблица на честотите на срещане на отделните букви (най-често символи), без да се интересуват от контекста. В случай на 256 различни символа, беше необходима единствена таблица на честотите, съдържаща по един брояч за всеки символ. По същество това представлява *модел от нулев ред*. В случай, че решим да отчитаме предходния символ, получаваме модел на Марков от първи ред. За реализирането му, в случай на 256 различни символа, са ни необходими 256 различни таблици, всяка от които съдържа 256 различни елемента. Т. е. имаме нужда от 65536 единици (напр. байтове или думи) памет. i-тата таблица съдържа условните вероятности за срещане на всеки от символите, при условие, че непосредствено предхождащият го символ е бил с *ASCII* код i. Така в нашия пример таблица номер 10 ще съдържа в 13-ия си ред (номерацията на редовете и таблиците започва от 0) вероятност 1 (условната вероятност да се срещне 10 при предходен символ 13). За съжаление, преминаването към модел на Марков от по-висок ред е свързано с редица проблеми. Така например, ако решим да преминем към модел от ред 2, т. е. да отчитаме каква е била предходната *двойка* символи, ще ни бъдат необходими 16777216 единици оперативна памет или 65536 таблици с по 256 реда. Вижда се, че когато редът на модела расте линейно, необходимата памет расте експоненциално. Това силно ограничава възможностите са прилагане на модели от по-висок ред. Нещо още по-лошо: по-голямата част (с малко изключения) от клетките в таблицата ще бъдат празни, тъй като тази последователност от букви никога не се е срещала.

Ще припомним, че един възможен подход при прилагането на статистически методи на компресиране е входното съобщение да се прочете, при което да се построят необходимите ни статистики. Следва повторно прочитане на съобщението и съответно компресиране. Основен недостатък на този подход, както вече нееднократно сме споменавали, е нуждата от предаване на статистиките на декодиращата програма. Очевидно в случай на използване на модел от по-висок ред това е крайно нежелателно, тъй като размерът на статистиките може значително да надвиши размера на самото съобщение. Остават двете познати ни решения: използване на *фиксирана честотна таблица* и *адаптивно компресиране*. Тук са в сила всички приведени по-горе разсъждения относно предимствата и недостатъците на двата подхода. Ще припомним, че за адаптивното компресиране основните проблеми са два: започва се с неоптимален модел и след всеки символ моделът следва да се актуализира.

Използването на модели от по-висок ред, например трети, е свързано с поддържането на 16777216 различни честотни таблици, всяка с по 256 елемента. На пръв поглед реализирането на подобен метод може да изглежда нереалистично: заделянето на толкова памет (дори и да я имаме в наличност) е чисто прахосничество. Действително далеч не всички възможни последователности от четворки последователни символи ще се срещнат изобщо в текста, т. е. огромна част от еле-

ментите на таблиците и дори цели таблици ще бъдат празни. Така например, при компресиране на текст на български език е излишно да предвиждаме място за последователности от символи от вида “ьнбх”, “ййьз” или “ъьбб”, просто защото такива едва ли ще се срещнат някога. (Освен, разбира се, ако като входно съобщение не се предаде текстът на настоящата книга. ;)))

Макар това да не се вижда на пръв поглед, при адаптивно компресиране с използване на модели от трети ред, например, се налага поддържане на съответните таблици за *всички* модели от по-нисък ред: нулев, първи и втори. Това е така, тъй като не винаги можем да кодираме новопостъпилия символ в текущия контекст. Компресирането започва с празна таблица, като всеки символ се добавя в нея, едва когато възникне нужда. Когато дадена четворка символи постъпи за пръв път, се извършва установяване на 1 в съответния брояч в таблицата. На изхода се подава специален *escape* код, след което се прави опит символът да бъде закодиран в таблица, съответстваща на контекст от ред, по-нисък с 1. Ако кодирането отново се окаже невъзможно, се генерира нов *escape* и се прави опит за кодиране в модел от първи ред. Ако кодирането се окаже невъзможно дори в нулев ред, то се преминава към специален контекст от (–1)-ви ред, чиито броячи първоначално са били инициализирани с единици, т. е. там със сигурност ще можем да кодираме символа. Актуализирането се извършва само в тези таблици, които са участвали в кодирането на текущия символ, т. е. чиито броячи са били променяни.

Обикновено честотата на *escape* се приема за единица и не се променя в процеса на кодиране. Това означава, че на *escape* се съпоставя най-дългият достъпен код. Подобен подход не е много ефективен, тъй като символът *escape* се среща доста често, особено при модели от по-висок ред. Понякога на *escape* се съпоставя честота на срещане, равна на броя на символите от текущата таблица, без значение на честотата им. Това работи добре, тъй като с увеличаване на броя на попълнените символи в таблицата вероятността за срещане на *escape* намалява. В случай на пълно запълване вероятността става нула. От друга страна е добре да се отчита и вероятността на "познаване" на символ от таблицата, т. е. с каква вероятност постъпващите символи се откриват в таблицата. С нарастването на тази вероятност вероятността за срещане на *escape* намалява. Тимоти Стенли (*виж* [*Стенли-1998*]) предлага формула за пресмятане честотата на *escape*, като въвежда *мярка на случайността* на таблицата: максималната честота, разделена на средната. Колкото по-малко е частното, толкова по-малко случайна е таблицата. Честотата на *escape* се пресмята с помощта на формулата:

```
честота = (256-#срещнати_символи) * #срещнати_символи/(256*най_висока_честота)
ако (честота < 1) то честота = 1
```

Възможно е компресирането да става с постепенно увеличаване на реда на модела. Започва се с модел от нулев ред, като към модел от първи ред се преминава, едва след натрупване на достатъчно статистики. Аналогично към модел от втори ред се преминава, едва след някаква степен на запълване на таблиците от втори ред и т. н. Това намалява значително броя на генерираните *escape* символи.

Остава да решим най-сериозния проблем: как да поддържаме таблиците, като се предпазваме от заделяне на излишна памет. Налице са два вида проблемни контекстни таблици: изцяло празни и частично пълни. Едно решение е контекстните таблици да се организират като дърво. В корена на дървото стои контекстната таблица от ред 0 (тя е единствена). За всеки ред от таблицата се поддържа освен брояч и указател към съответна таблица от първи ред. В случай, че за някой символ не съществува съответна таблица от първи ред, то указателят му сочи `NULL`. От своя страна таблиците от първи ред също съдържат указатели: този път към таблици от втори ред. Таблиците от втори ред сочат към съответни таблици от трети ред. Последните представляват листата на нашето дърво (предполагаме, че работим с модел от трети ред). Нова таблица се създава при първо срещане на съответния контекст. Освен това всяка таблица ще съдържа точно толкова реда, колкото е броят на контекстите с ненулеви броячи. За целта ще се поддържа специален брояч на максималния достъпен индекс в таблицата, а всеки ред, освен брояч и указател към таблица от по-висок ред, ще съдържа и символа, съответен на реда. Листата на дървото, т. е. контекстните таблици от последно (трето) ниво изобщо не съдържат указатели към следваща таблица. По принцип дървото на таблиците ще се обхожда от корена към листата, но в случай на *escape* сим-

вол ще ни бъде необходимо придвижване в обратната посока, т. е. към родителската таблица. С цел ускоряване на този процес е удобно да въведем за всяка таблица съответен указател към родителя. Отделно от това дърво ще поддържаме контекстна таблица от минус първи ред, чиято роля беше обсъдена по-горе.

```
struct CRow { /* ТИП: ред от контекстна таблица */
   char symbol;                    /* символ */
   unsigned cnt;                   /* брояч */
   struct CContextTable *nextLevel; /* следваща контекстна таблица */
};
struct CContextTable { /* ТИП: контекстна таблица */
   unsigned char cnt;                /* брой елементи в таблицата */
   struct CRow *rows;                /* масив от структури */
   struct CContextTable *parentTable; /* родителска контекстна таблица */
};
```

Задачи за упражнение:

1. Да се реализира описаният алгоритъм.
2. Какъв друг проблем освен с паметта възниква при преминаване към модел от по-висок ред?
3. Да се посочат предимствата и недостатъците на *постепенното* преминаване към модел от по-висок ред (едва след натрупване на достатъчно статистики).
4. Да се анализира възможността за ползване на ляв и десен контекст едновременно.

10.5.3. Един представителен пример: MNP-5

По-долу ще бъдат разгледани някои интересни методи на компресиране. Те са важни поне по две причини: първо, става въпрос за ефективни алгоритми, утвърдени като стандарт и с реално практическо приложение, и второ, те са илюстрация на принципа, че най-добра компресия не се получава при използване на *един* от стандартните алгоритми, а при подходящото им *комбиниране*.

Първият метод, на който ще се спрем, е *MNP* (от англ. *Microcom Networking Protocol*). Става въпрос за стандартен комуникационен протокол, разработен от *Microcom Inc.* и предназначен за комуникация между модеми. След първоначалната разработка на протокола, той търпи редица подобрения, всяко от които си има свой пореден номер. По-долу разглеждаме петата му модификация — *MNP* от клас 5. Подобно на кодирането при факс-машините, процесът се извършва на две стъпки. Първата стъпка представлява модификация на кодирането на последователности (*виж 10.3.9.*). За разлика от класическия вариант, тук се работи с *тройки* последователни символи. След всяка тройка се вмъква брояч, указващ броя на повторенията, а самите повторения се премахват. Така се формира кодираща четворка. В случай, че тройката символи се среща точно веднъж, броячът получава стойност 0. Максималната му стойност е 250, а в случай на повече повторения се използват няколко кодиращи четворки.

На втората стъпка получената последователност се кодира отново при използване на *адаптивно компресиране*. Входното съобщение се разбива на последователности от 8 бита, всяка от които се интерпретира като един *ASCII* символ, и ѝ се съпоставя побуквено някакъв код. Особеното тук е кодът. За разлика от разгледания по-горе пример, където беше използван стандартен код на Хъфман, тук се използва друг неравномерен префиксен побуквен код. Идеята, която стои в основата на кодирането е, че от 256 възможни символи, 128 започват с първи бит 0, т. е. могат да се представят със 7-битов равномерен код чрез просто премахване на водещата нула. Доразвивайки идеята, получаваме, че бихме могли да постигнем значително намаляване на дължината на съобщението чрез премахване на всички водещи нули от кодовете на *ASCII* символите. Впрочем, подобен запис е напълно естествен и е най-често използваният в реалния

живот. Свикнали сме да пишем 10523+5, а не 10523 + 00005. За съжаление, при премахването на нулите възникват два сериозни проблема:

1) пълната нула (00000000) няма код;
2) получава се *непрефиксен* код, чието декодиране е *нееднозначно*.

За да се справи с този проблем, *MNP*-5 въвежда допълнителни разделители така, че кодът да се запази префиксен. За целта, след премахване на водещите нули, пред кода на символа се поставя *трибитов префикс*, избран за всеки символ по такъв начин, че като цяло кодът да бъде *префиксен*. Всъщност, се използва малко по-различна и по-проста конструкция, нагледна представа за която читателят може да получи от *таблица 10.5.3*.

Вижда се, че кодът, построен по *таблица 10.5.3.*, се справя достатъчно просто и с двата изложени по-горе проблема. С построяването на горната таблица нещата едва започват. *MNP*-5 я разглежда само като начална таблица за прилагането на адаптивно компресиране. Алгоритъмът счита, че 00000000 е най-често срещаният в момента символ, поради което му съпоставя най-краткия достъпен код — с дължина 4 бита. Изобщо, колкото е по-близо до началото на таблицата даден код, толкова е по-често срещан. По-нататък се използват стандартните за адаптивното компресиране техники на поддържане на специални броячи, пермутиране на символите, защита от препълване на броячите и др., които бяха вече разгледани.

Възниква още един проблем, който е общ за всички неравномерни кодове, и вече беше разгледан по-горе. Поради неравномерността на кода, може да се случи (при това с вероятност 7/8) дължината в битове на кодираното съобщение да не бъде кратна на 8. Тъй като данните се предават побайтово, т. е. на порции от по 8 бита, се налага последният байт евентуално да бъде допълнен с подпълващи битове. В такъв случай възниква опасност от опит за декодирането им. За да се справи с този проблем, *MNP*-5 поставя в края на съобщението специален *стоп-код*: 11111111111. При достигането му процесът на декодиране се прекратява.

***ASCII* код на символа**		**код**	
десетичен	**двоичен**	**префикс**	**тяло**
0	00000000	000	0
1	00000001	000	1
2	00000010	001	0
3	00000011	001	1
4	00000100	010	00
5	00000101	010	01
6	00000110	010	10
7	00000111	010	11
8	00001000	011	000
9	00001001	011	001
10	00001010	011	010
11	00001011	011	011
12	00001100	011	100
13	00001101	011	101
14	00001110	011	110
15	00001111	011	111
16	00010000	100	0000
17	00010001	100	0001
18	00010010	100	0010
19	00010011	100	0011
20	00010100	100	0100
21	00010101	100	0101

22	00010110	100	0110
23	00010111	100	0111
24	00011000	100	1000
25	00011001	100	1001
26	00011010	100	1010
27	00011011	100	1011
28	00011100	100	1100
29	00011101	100	1101
30	00011110	100	1110
31	00011111	100	1111
32	00100000	101	00000
33	00100001	101	00001
...	...	...	...
247	11110111	111	1110111
248	11111000	111	1111000
249	11111001	111	1111001
250	11111010	111	1111010
251	11111011	111	1111011
252	11111100	111	1111100
253	11111101	111	1111101
254	11111110	111	1111110
255	11111111	111	11111110

Таблица 10.5.3. *MNP-5*. Кодиране с префикс и тяло.

Каква е ефективността на разгледания метод? Лесно се вижда, че максималната степен на компресия, която може да бъде постигната е 50%, тъй като най-късият код, с който се заменя даден *ASCII* символ, е 4 бита, т. е. два пъти по-малко. От друга страна вероятността съобщението да нарасне изглежда достатъчно голяма: само първите 32 символа се кодират с кодове с дължина, по-малка от 8, като за останалите 224 символа това кодиране би следвало да води до нарастване на кода. На практика обаче, това може да се случи рядко, тъй като кодирането е *адаптивно*. За текстов файл се очаква 15-те най-често срещащи се символи да съставляват над 50% от съдържанието му. 15% от 256 = 38,4. Т. е. на 38,4 от символите се пада 50% от съдържанието на файла. Но първите 64 символа от таблицата се кодират с намаляващи кодове. Тогава адаптивното кодиране следва да бъде достатъчно ефективно и може да се очаква средна компресия между 3:2 и 2:1. На практика модемите, използващи *MNP-5*, могат да постигнат обмен между 14400 и 19200 *bps* при действителна скорост на модема 9600 *bps*.

Задачи за упражнение:

1. Да се обсъдят предимствата и недостатъците на метода. Как би могъл да се подобри?
2. Да се обясни защо се строи таблица *10.5.3*. Защо не се използва направо код на Хъфман?

10.6. Речниково кодиране

Речниковото кодиране се основава на изграждането и поддържането на таблица, съдържаща някои от думите от входното съобщение. Алгоритъмът се изпълнява на две стъпки и предполага кодиране на крайни входни последователности. На първата стъпка входното съобщение се изчита и анализира, за да се определят кодовите думи. На втората стъпка то се преглежда отново, като думите от речника се заменят с индекса си в него. При подходящ избор те ще се окажат по-

къси от индекса си, т. е. ще се постигне компресиране. Какво значи *подходящ избор* и как да определяме думите от речника? За съжаление, не съществува алгоритъм, който да ни дава оптимални речник. Различните методи решават по различен начин този проблем, като никой от тях не гарантира оптималност. Интуитивно са ясни две неща: първо, в речника трябва да влязат най-често срещаните думи, и второ, при по-дълги думи следва да се очаква по-добра степен на компресия.

Обикновено размерът на речника е 2^n, тъй като тогава индексът има размер n бита. Някои варианти на алгоритъма изискват фиксиран размер на думите от речника. Променливият размер обаче по принцип осигурява по-висока ефективност. Ясно е, че не всички думи от входната последователност могат да присъстват в речника. В случай, че някоя дума липсва, тя ще постъпи на изхода без изменения. Получава се смесване на кодирани и некодирани думи, което явно може да доведе до проблеми при декодирането. Един възможен подход за решаване на проблема е, преди (кода на) думата да се поставя специален разделител, указващ дали следващите битове следва да се възприемат като дума или като индекс в речника. Използването на отделни такива разделители може да се окаже неефективно, поради което те често се групират. Най-често за разделителите се отделя цял байт. Ако всички негови битове са 0, след него следва индекс от речника. В противен случай следва дума с дължина, указана от числото, записано в байта.

Съществуват множество повече или по-малко ефективни модификации на описания метод. Бихме могли да кодираме думите от речника не чрез индекса им, а чрез код на Хъфман, основан на честотата им на срещане. Освен това бихме могли да обновяваме динамично съдържанието на речника в процеса на кодиране, отчитайки локалните особености и др. Преди да продължим, ще въведем важното понятие *ентропия*: фундаментално понятие в теорията на информацията и на кодирането в частност.

Задачи за упражнение:

1. Да се обсъдят предимствата и недостатъците на използване на речник с *фиксирана* и с *променлива* дължина на думите.
2. Да се обсъдят предимствата и недостатъците на използване на *фиксиран* срещу *динамичен* (изграждан на основата на текущия текст) речник.
3. Следва ли да се очаква подобрение спрямо Хъфман и аритметичното кодиране?

10.6.1. Ентропия

Нека предположим, че входното съобщение съдържа само две букви: a и b, като a се среща с вероятност 90%. Да разгледаме всевъзможните двойки последователни букви: aa, ab, ba и bb. Вероятността за поява на всяка от тях се пресмята тривиално и е дадена в *таблица 10.6.1а*.

Как можем да кодираме ефективно входното съобщение, при положение че съдържа само две букви? Очевидно стандартното кодиране по Хъфман (*виж 10.4.2.*) не ни носи нищо. Действително, то съпоставя еднобитови кодове на a и b, т. е. получаваме равномерен код, съпоставящ на всяка буква точно по един бит. Не бихме ли могли да получим по добър резултат?

двойка	*aa*	*ab*	*ba*	*bb*
вероятност	0,81	0,09	0,09	0,01

Таблица 10.6.1а. Вероятности за срещане на *двойки* букви при азбука $\{a,b\}$ и 90% вероятност за срещане на буквата a.

Да разгледаме отново *таблица 10.6.1а.* Ако гледаме на двойките като на супербукви, то тя ни дава честотата на срещане на всяка от тях във входното съобщение. Извършвайки кодиране по Хъфман с новите 4 супербукви (предполагайки още, че дължината на входното съобщение е кратна на 2), получаваме кода: aa = 0, ab = 10, ba = 110, bb = 111. Да пресметнем средния брой битове, необходими за кодирането на една двойка:

$$L_2 = 1.0{,}81 + 2.0{,}09 + 3.0{,}09 + 3.0{,}01 = 1{,}29$$

За да получим броя битове, необходими ни за кодирането на една първоначална буква (*a* или *b*), трябва да разделим на 2. Така получаваме средна дължина 0,645 бита/буква, което е много добра степен на компресия.

тройка	*aaa*	*aab*	*aba*	*baa*	*abb*	*bab*	*bba*	*bbb*
вероятност	0,729	0,081	0,081	0,081	0,009	0,009	0,009	0,001
код	0	100	101	110	11100	11101	11110	11111

Таблица 10.6.1б. Вероятности за срещане на *тройки* букви при азбука {*a*,*b*} и 90% вероятност за срещане на буквата *a*.

Нека сега кодираме същото входно съобщение с тройки букви. Получаваме *таблица 10.6.1б.* Лесно се пресмята, че в този случай средният брой битове, необходими за кодирането на една тройка, е L_3 = 1,598. Разделяйки на 3, получаваме, че за кодирането са ни необходими средно 0,533 бита/буква. Оказва се, че групирането на буквите по тройки води до още по-висока степен на компресия. А какво би станало при групиране по четворки? А по петици? Изобщо, колко пъти можем да извършваме успешно този процес и до каква минимална дължина на кода на съобщението може да ни доведе той?

двойка	*aa*	*ab*	*ba*	*bb*
вероятност	0,64	0,16	0,16	0,04
код	0	10	110	111

Таблица 10.6.1в. Вероятности за срещане на *двойки* букви при азбука {*a*,*b*} и 80% вероятност за срещане на буквата *a*.

Преди да отговорим на горните въпроси, нека видим какво ще стане, ако намалим разликата между вероятностите за поява на всяка от буквите. Нека *a* се среща с вероятност 80%. Тогава очевидно се променя и вероятността за поява на всяка двойка, а оттук евентуално и кодът на Хъфман, построен за двойките. Получаваме *таблица 10.6.1в.* за двойките и *таблица 10.6.1г.* — за тройките. Забележете, че кодът на Хъфман и в двата случая се запазва.

Да пресметнем очаквания среден брой битове за всеки от случаите. Получаваме: L_2 = 1,56 и L_3 = 2,184. Разделяйки съответно на 2 и 3, окончателно имаме 0,78 и 0,728 бита/буква. Вижда се, че при намаляване разликата между вероятностите на отделните букви, степента на компресия също намалява. Ако увеличим още повече разликата между вероятностите на буквите, изхвърляйки изобщо едната от тях, получаваме тривиално съобщение, представляващо многократно повторение на една и съща буква. Ясно е, че бихме могли да го кодираме изключително ефективно: например копие на буквата, последвано от броя повторения (*виж 10.3.9.*).

тройка	*aaa*	*aab*	*aba*	*baa*	*abb*	*bab*	*bba*	*bbb*
вероятност	0,512	0,128	0,128	0,128	0,032	0,032	0,032	0,008
код	0	100	101	110	11100	11101	11110	11111

Таблица 10.6.1г. Вероятности за срещане на *тройки* букви при азбука {*a*,*b*} и 80% вероятност за срещане на буквата *a*.

От какво зависи ефективността на кодирането и защо с увеличаване разликата между вероятностите, степента на компресия нараства? За да отговори на този въпрос, през 1940 г. Клод Шенън въвежда понятието *количество информация*. Преди да го определим, нека поразсъждаваме интуитивно. Очевидно съобщение, съдържащо единствена буква със 100% вероятност, носи минимално количество информация. Допускайки някакъв нисък процент други

букви, получаваме съобщение, което ни носи малко повече информация. Ако намалим още повече разликата във вероятностите, ще получим съобщение, носещо още повече информация. И тук веднага проличава връзката между количество информация и степен на компресия. Действително, от горните примери веднага се вижда, че колкото по-малко информация носи едно съобщение, толкова по-добре се компресира. [*Adámek-1991, Held-1991, Wayner-1996*]

Понятието *информация*, съдържаща се в едно съобщение, може да се разглежда както *синтактично* (как изглежда външно), така и *семантично* (какво всъщност означава). Очевидно, за целите на компресирането смисълът на съобщението е напълно ненужен. Можем да компресираме произволно съобщение, без да ни е необходимо да знаем какво точно означава, т. е. при кодирането ние гледаме на него строго синтактично и само като на последователност от букви. За да се избегнат подобни недоразумения, свързани със смесване на двете интерпретации на понятието *информация*, Шенън въвежда термина *ентропия*. Опитвайки се по някакъв начин да го дефинира, той забелязва, че то зависи изцяло от вероятностите за поява на отделните букви във входното съобщение. Поставяйки и някои допълнителни естествени изисквания, Шенън достига до идеята за дефиниция на ентропията като функция $H(p_1,p_2,...,p_n)$ на вероятностите $p_1,p_2,...,p_n$ за поява на всяка от буквите, със следните свойства:

- *неотрицателност*: $H \geq 0$;
- *непрекъснатост*: Малките изменения на вероятностите да водят до малки изменения на ентропията;
- *симетричност*: $H(p_1,...,p_i,...,p_j,...,p_n) = H(p_1,...,p_j,...,p_i,...,p_n)$; $1 \leq i, j \leq n$
- *кохерентност*:

$$H(p_1, p_2,..., p_n) = H(p_1 + p_2, p_3,..., p_n) + (p_1 + p_2).H\left(\frac{p_1}{p_1 + p_2}, \frac{p_2}{p_1 + p_2}\right)$$

Кохерентността ни дава правило за пресмятане на ентропията на n букви чрез ентропията на $n-1$ и 2 букви. Това е полезно и в някаква степен естествено свойство от когнитивна гледна точка. Като пример можем да посочим как човек разчита нечетливо написана дума. В началото прави опит за прочитане на думата, като пропуска неясните букви. Да предположим, че са разчетени всички букви, с изключение на първите две. Сега, на втората стъпка, човек прави опит да разчете *само* двете неясни му букви. Горната формула отразява точно такъв процес.

Оказва се, че съществува, при това единствена с точност до константа $c > 0$, функция $H(p_1,p_2,...,p_n)$, удовлетворяваща горните условия, а именно:

$$H(p_1, p_2,..., p_n) = c\sum_{\substack{i=1 \\ p_i>0}}^{n} p_i \ln\frac{1}{p_i}$$

Изразявайки c във вида $c = 1/\ln k$, получаваме:

$$H(p_1, p_2,..., p_n) = \sum_{\substack{i=1 \\ p_i>0}}^{n} p_i \log_k \frac{1}{p_i} = -\sum_{\substack{i=1 \\ p_i>0}}^{n} p_i \log_k p_i$$

Най-често се избира $k = 2$, при което окончателно за ентропията се получава:

$$H(p_1, p_2,..., p_n) = -\sum_{\substack{i=1 \\ p_i>0}}^{n} p_i \log_2 p_i$$

По-долу ще приведем някои важни твърдения, свързани с ентропията, без да се спираме на доказателствата им. Някои от тях са почти очевидни, а доказателства на останалите любознателният читател би могъл да намери например в [*Манев-1996*], [*Adámek -1991*] и др.

Твърдение 1: Минималната стойност на ентропията е 0 и се постига за входно съобщение, съдържащо единствена буква.

Твърдение 2: Максималната стойност на ентропията е $\log_2 n$ (за $p_1 = p_2 = ... = p_n = 1/n$).

Досега разглеждахме ентропията като функция на вероятностите за поява на буквите от входното съобщение. По-долу се налага използването на друга, по-обща дефиниция, съгласно която H е функция на входното съобщение S. Много често в смисъла на тази дефиниция (а и не само) входното съобщение S се нарича *информационен източник*.

Твърдение 3: Да означим с S^k входното съобщение, разглеждано над азбука от k-орки. Тогава $H(S^k) = k.H(S)$.

Твърдение 4: За всеки източник S и за всеки двоичен код, построен по този източник, е в сила неравенството $L(S) \geq H(S)$.

Теорема. (*Шенън*) $H(S) \leq L_{\min}(S) \leq H(S) + 1$.

Следва примерна програма за пресмятане на ентропията на информационен източник, теоретично оптималната дължина на кода, препоръчителни дължини на кода за всяка от буквите, както и действителна цена на кода при тези дължини. Програмата е проста и реализацията ѝ произтича директно от изложения по-горе теоретичен материал.

```
#include <stdio.h>
#include <math.h>

#define log2(x) log(x)/log(2)
#define MAX 100

const double p[MAX] = {0.2, 0.2, 0.15, 0.15, 0.10, 0.10, 0.05, 0.05};
const unsigned n = 8; /* Брой вероятности */

/* Пресмята ентропията на източника */
double calcEntropy(const double *p, const unsigned n)
{ unsigned i;
  double sum;
  for (sum = i = 0; i < n; i++)
    sum -= p[i]*log2(p[i]);
  return sum;
}
/* Пресмята цената на кода */
double calcValue(const double *p, const unsigned *l, const unsigned n)
{ unsigned i;
  double sum;
  for (sum = i = 0; i < n; i++)
    sum += p[i]*l[i];
  return sum;
}

/* Пресмята дължините на кодовете на отделните букви */
void calcLengths(unsigned *l, const double *p, const unsigned n)
{ unsigned i;
  for (i = 0; i < n; i++)
    l[i] = (unsigned) ceil(log2(1.0 / p[i]));
}

int main(void) {
  unsigned i;
  double entr;
  unsigned l[MAX];

  printf("\n\nИзточник, зададен с честоти на срещане: ");
  for (i = 0; i < n; i++)
```

```
    printf("%2.2lf ",p[i]);

  entr = calcEntropy(p,n);
  printf("\nЕнтропия на източника: %8.5lf",entr);
  printf("\nТеоретична цена на кода: %8.5lf",entr + 1);

  calcLengths(l,p,n);
  printf("\nДължини на кодовете: ");
  for (i = 0; i < n; i++)
    printf("%u ",l[i]);

  printf("\nЦена на кода при горните дължини: %2.2lf",calcValue(p,l,n));
  return 0;
}
```

🖫 entropy.c

Резултат от изпълнението на програмата:

```
Източник,зададен с честоти на срещане: 0.20 0.20 0.15 0.15 0.10 0.10 0.05 0.05
Ентропия на източника:  2.84644
Теоретична цена на кода:  3.84644
Дължини на кодовете: 3 3 3 3 4 4 5 5
Цена на кода при горните дължини: 3.40
```

Нека сега се върнем към първия пример. Там, кодирайки двойки букви, получихме средна дължина от L_2 = 1,29 бита/двойка. Не е трудно да се види, че този процес е еквивалентен на кодиране на S^2. Тогава за кодирането на една буква получаваме средно $L_{min}(S^2)$ = 0,654 бита. Разглеждайки тройки букви, успяхме да сведем този брой до $L_{min}(S^3)$ = 0,533. Защо степента на компресия се повиши? Отговор на този въпрос дават твърдения 3 и 4, от които директно следва, че кодирането на низове (речниково кодиране) е по-ефективно от побуквеното.

Задачи за упражнение:

1. Защо средният брой необходими битове за кодиране на една буква намалява с увеличаване на дължината на използваната супербуква? Винаги ли ще бъде така?
2. Какви проблеми възникват при по-дълги супербукви?
3. Да се докажат твърдения 1 и 2, като се изходи от дефиницията на понятието ентропия.

10.6.2. Малко история

Началото на речниковото компресиране се поставя през 1975 г. в Израел от Абрахам Лемпел и Якоб Зив, които разработват, а по-късно, през 1977 г. и публикуват в *IEEE Transactions on Information Theory*, най-популярния алгоритъм за речниково кодиране. На практика, те поставят началото на изследванията в този клон от теорията на информацията. До този момент практически почти всички усилия са се насочвали по посока на подобряване на класическото Хъфманово кодиране и на негови варианти. Предложеният от тях алгоритъм допуска *променлива*, но *ограничена отгоре* от някаква константа, дължина на думите от речника. В процеса на компресиране всяка кодова дума се кодира с индекса си в речника, който е число с фиксирана дължина като брой битове.

По същество *LZ77* (по имената на двамата му автори и годината на публикацията) представлява алгоритъм с плъзгащ се прозорец с размер от 2 *KB* до 16 *KB* и променлива дължина на думите между 16 и 64 символа (тук обикновено под *буква* се разбира един символ *ASCII*). Речникът съдържа думи с фиксирана дължина, извлечени от прочетения до момента текст. Лемпел и Зив продължават работата си и през следващата 1978 г. двамата публикуват нова статия, в която описват нов вариант на алгоритъма, получил известност като *LZ78*, при който дължината на думата се увеличава при всяко следващо срещане. Широкоразпространена заблуда е схващането,

че *LZ78* е просто подобрение на *LZ77*. Макар в основата им действително да е залегнала една и съща идея, както ще видим по-късно, те се различават значително, поради което е по-правилно да се разглеждат независимо един от друг. В подкрепа на това твърдение може да се приведе и фактът, че повечето съвременни комерсиални продукти предпочитат *LZ77*, което поставя под въпрос доколко *LZ78* наистина може да се разглежда като *подобрение*.

Въпреки безспорните си предимства и силното влияние, което оказва в чисто теоретичен аспект, поставяйки началото на цял клон от теорията на кодирането, речниковото кодиране остава още няколко години пренебрегнато от компютърния свят, доминиран по това време от *SQ* и подобни компресиращи програми, основани на не особено ефективното кодиране по Хъфман с модел от първи ред. Едва в средата на 80-те години процесорната мощ и размерът на оперативната памет нарастват достатъчно и става възможно разработването на практически реализации, отварящи пътя на речниковото кодиране към затворения за него до момента компютърен свят. За това е необходима още една статия през 1984 г. в *IEEE Computer*. Автор на статията този път е Тери Уелч, служител на *Unisys*, а самата тя представлява описание на вариант на *LZ78*, наречен от Уелч *LZW* по имената на Лемпел-Зив-Уелч, и по-специално на някои аспекти на нейното приложение при лентови устройства и контролери.

Още тогава за компютърния свят става ясно, че изложеният метод има голямо бъдеще и редица софтуерни и хардуерни компании пристъпват към разработка на конкретни продукти. Началото се поставя година по-късно, когато на пазара навлиза знаменитата програма *COMPRESS*, първоначално разработена за *VAX* и работеща под *UNIX*. Скоро след това се появява и версия за *PC* под *MS-DOS*. Опитът се оказва изключително успешен и на пазара започват да навлизат все повече играчи. През 1985 г. *System Enhancement Associates* пуска своята *ARC*, основана на *LZ78*, която бързо печели позиции. Следва поява на редица конкурентни продукти като *PKARC* на *PKWare*, представляващи по същество слаби подобрения, определяни по-скоро като имитации, за да се стигне до *LHarc* на Хируяши Йошизаки, *ARJ* на Робърт Юнг и *PKZIP* на *PKWare*, които изхвърлят *ARC* от борбата и успяват да си поделят пазара. За голяма изненада тези програми се основават не на *LZ78*, а на по-ранния му вариант *LZ77*, считан първоначално за по-несъвършен, дори от самите му автори Лемпел и Зив!

По-късно настъпва часът и на *LZW*, който започва да пробива в редица области като компресиране на графика, звук, видео и компресиране в реално време. Алгоритъмът е изполван от *CompuServe Information Service* за компресиране на растерни изображения. Разработеният от тях файлов формат *GIF* използва *LZW* за компресиране на области с еднакъв цвят. Други широкоразпространени стандарти като *TIFF* (графични изображения) и *PostScript* (електронни документи) предлагат *LZW* компресията като възможност (опция).

10.6.3. Стандарти и патенти

Бързото навлизане на речниковото кодиране като практически универсален метод скоро започва да буди притеснения, свързани със съвместимостта между отделните му варианти. Особено силни са страховете относно съвместимостта при вграждане на алгоритъма в хардуерни устройства. С навлизането на мрежовите технологии бързо става ясна силата на компресирането при пренос на данни по мрежата. Първоначално се предлагат различни софтуерни решения, а по-късно се ражда и идеята за вграждане на алгоритъма в самите устройства за предаване и приемане на данни: модемите (от англ. ***MO**dulator* ***DEM**odulator*). Налага се изискването устройствата на двата края на линията да бъдат съвместими, т. е. да използват един и същ алгоритъм, във връзка с което започват да се въвеждат съответни стандарти. Като пример в това отношение може да се посочи стандартът *CCITT V*.42, разработен от служители на *IBM*, *British Telecom*, *Bell Laboratories*, *Unisys* и други компании. Стандартът изисква минимален размер на речника поне 512 *KB* и максимална дължина на думите до 250 символа. Речникът се изгражда динамично, което позволява кодирането да става с единствено прочитане на входната последователност. В началото той съдържа само едносимволни думи. На всяка стъпка множеството от думите в речника може да се разшири чрез прибавяне на текущия символ към някой от низовете в речника.

Изобщо, след появата на речниковото кодиране в тази насока се провеждат редица изследвания, като резултатите от повечето от тях се защитават с патенти. Така например, подобрението на *LZ78* на Тери Уелч, което ражда и *LZW*, е защитено с американски патент номер 4558302 от 10.12.1985 г., притежание на *Sperry Corp.* (по-късно *Unisys Corp.*). По-късно това води до някои проблеми при разработването на споменатия по-горе стандарт *V.42bis* във връзка с настояването на *Unisys Corp.* да им бъде заплащано за всяко използване на *LZW*.

10.6.4. Статични срещу адаптивни методи

В началото на главата споменахме, че почти всички по-важни методи на компресиране имат статичен и адаптивен вариант. Това, разбира се, важи и за речниковите методи. Както по-горе споменахме, речниковото компресиране се свежда до заместване на някои низове с индекс в речника. Вечният въпрос "Кое е за предпочитане?", няма еднозначен отговор. Адаптивните методи по принцип имат предимството сами да построяват необходимата им статистика в процеса на кодиране, при което се отчитат особеностите на конкретния текст и се постига по-добра степен на компресия. От друга страна им е необходимо известно време, за да успеят да си я построят, което не им позволява да разгърнат възможностите си при кратки файлове. Същевременно, начинът на кодиране при статичните методи не се влияе от размера на входното съобщение, което се оказва предимство при кратки файлове, и недостатък — при по-големи. Недостатък на статичните методи е необходимостта от предаване на речника. Същият проблем стои и при статичния речник, освен ако кодиращата и декодиращата програма "не са се разбрали" предварително за речника, който ще използват, без да си го предават. Това е налице например при използване на един и същ речник върху определен вид (например текстови) или върху всички видове файлове.

На практика, почти всички разпространени днес комерсиални продукти използват адаптивни методи. Обикновено с цел избягване на недостатъците при кратки файлове те не започват от празен, а от някакъв минимален начален речник, винаги един и същ (за конкретния тип файл). В процеса на кодиране речникът се обогатява с нови думи. Най-общо адаптивните речникови методи следват следната схема:

```
initTable();
while (!eof(input)) {
  word = getWord(input);
  if ((dictionaryIndex = findIndex(dictionary,word)) < 0) {
    writeOut(output,word);
    add2Dictionary(dictionary,word);
  }
  else
    writeOut(output,dictionaryIndex);
}
```

Предложената схема следва да се разглежда като най-обща универсална рамка за адаптивните речникови методи на компресиране. На практика, отделните детайли в реализацията ѝ могат силно да варират. Въпреки това, тя ни дава най-обща представа за основните ѝ характеристики, а именно:

1. Входът се чете на части (думи), като се търсят съвпадения с думите от речника.
2. В процеса на компресиране речникът се обогатява с нови думи.
3. Кодирането на индексите в речника и обикновените думи се различават.

Третото изискване е изключително важно, тъй като в противен случай не бихме могли да си гарантираме еднозначност на декомпресирането. Декомпресирането силно прилича на компресирането. Алгоритъмът започва с празен или малък фиксиран речник, идентичен с този на компресиращия алгоритъм, като в процеса на декодиране той се актуализира по съответния начин. По-долу ще видим, че в някои конкретни случаи това може да доведе до проблеми, свързани с получаването на код, несъдържащ се все още в речника и др. На практика, процесът

на декодиране се свежда до разпознаване на индексите и заместването им със съответните думи от речника, при подаване на изхода.

Задача за упражнение:

Кога и защо би могъл да се получи код, който липсва в речника?

10.6.5. LZ77. Компресиране с плъзгащ се прозорец

Нека пристъпим към разглеждане на най-ранния представител на речниковите методи за компресиране — *LZ77*. Той е адаптивен метод, който, вместо да изгражда речник, използва като такъв прочетения до момента текст. Ефективността му се влияе от редица фактори като размер на прозореца и буфера, дължина на съвпадащите думи, ентропия на източника и др.

В процеса на компресиране алгоритъмът разглежда само част от текста, наречена *прозорец*. Прозорецът се състои от две части: *блок* от току-що закодиран текст и *буфер* от незакодиран текст. Най-често размерът на блока е няколко хиляди байта, докато размерът на буфера е от порядъка на няколко десетки байта. Основната идея на алгоритъма е да се търси съвпадение на началото на буфера в текста от блока. При откриване на такова на изхода се извежда код (на него следва да се гледа като на индекс в речника), състоящ се от кодираща тройка (p,l,c), а именно:

1. позиция p на съвпадение в прозореца
2. дължина l на общата последователност (*дума*)
3. символ c от буфера, непосредствено следващ *думата*

Следва придвижване на прозореца на l позиции надясно, при което най-левите l символа на прозореца го напускат, отдясно постъпват l нови незакодирани символа, а *думата* преминава от буфера в блока. *Таблица 10.6.5.* показва прозорец с 42-байтов блок от току-що кодиран текст и 16-байтов некодиран буфер.

Блок от вече кодиран текст	**Буфер**
Ала-бала_портокала,_кой_изяде_кашкавала_и_	кой_е_канибалът?

Таблица 10.6.5. *LZ77*: Примерен блок от вече кодиран текст и некодиран буфер.

Най-дългото съвпадение е четирибуквената дума "кой_", която се намира на позиция 21. Кодът, който ще бъде изведен е: *21, 4, "е"*. Следва преместване на прозореца на 4 позиции надясно. Сега се търси ново съвпадение и т.н.

А какво става при липса на съвпадение? В този случай се извежда по-особен триелементен код, а именно: *0*, *0*, c, където c е първият символ от буфера, след което прозорецът се измества на една позиция. Подобен подход не е особено ефективен, но пък гарантира просто и еднозначно декодиране.

Задачи за упражнение:

1. Да се реализира *LZ77*.
2. Защо на изхода се подава c, т.е. защо се извежда тройка (p,l,c), а не двойката (p,l)?
3. Подаването на $(0,0,c)$ при липса на съвпадение е очевидно неразумно, още повече че ще се случва често. Да се предложи по-икономично решение.

10.6.6. LZSS. Едно подобрение

LZSS представлява модификация на *LZ77* в две основни направления: ускоряване на търсенето на съвпадения и намаляване размера на извеждания код в случай на несъвпадение. След като дадена дума бъде намерена в речника, кодът ѝ бъде изведен на изхода и тя напусне буфера, *LZSS* я прибавя към специално двоично дърво с цел ускоряване на търсенето. Тъй като всяка

дума попада в *блока*, само *след* като е била разпозната в буфера, то всички думи от речника ще бъдат записани в дървото, което ще позволи по-ефективно търсене.

Нека означим с l_w дължината на прозореца, а с l_s — дължината на думата. Тогава времето, необходимо за откриването на най-дългия съвпадащ низ, ще бъде пропорционално на $\log_2 (l_w.l_s)$, вместо на $l_w.l_s$, както беше при *LZ77*. Това дава възможност да се работи с по-големи прозорци.

LZSS в общия случай строи обикновено сортирано дърво за търсене, т. е. левият наследник на всеки връх има по-малък ключ, а десният — по-голям от този във върха. Както вътрешните върхове, така и листата на дървото съдържат по една дума от речника. При неблагоприятно стечение на обстоятелствата може да се случи дървото силно да се разбалансира и дори да се изроди в списък, при което търсенето може да се забави значително. В такъв случай следва да се вземат специални мерки за неговото балансиране. Едно възможно решение е използване на специални техники за недопускане силно разбалансиране на дървото, например чрез използване на балансирани дървета (*виж 2.4.*). На практика се оказва, че в общия случай дърветата са сравнително добре балансирани. В случай, че все пак в даден момент се случи силно разбалансиране или дори израждане на дървото, то това по всяка вероятност е временно явление и бързо ще се оправи. Действително, дървото е изключително динамично: в него непрекъснато постъпват и го напускат думи, което бързо компенсира подобни локални проблеми.

Освен това *LZSS* премахва очевидно неефективния начин за кодиране на неразпознатите символи, позволявайки свободно смесване на текст и индекс от речника в генерирания код. За целта се въвежда еднобитов префикс, указващ дали следващите битове следва да се интерпретират като код на символ, или като индекс в речника. Това позволява по-голяма гъвкавост и намалява дължината на кодираното съобщение. Това е особено полезно в началото, когато блокът е празен, а съвпаденията — невъзможни или изключително редки. Понякога префиксите се групират в групи по 8, което позволява записването им в отделен байт с цел по-лесното им отделяне от кодираните данни.

Задачи за упражнение:

1. Да се сравнят *LZSS* и *LZ77*.
2. Защо се очаква дървото да бъде “добре балансирано”?

10.6.7. FLZ. Друг вариант на LZ77

Става въпрос за сравнително прост вариант на *LZ77*, в чиято основа са залегнали три интересни идеи: емулация на кодиране на последователности (*виж 10.3.9.*), поглед в двете посоки и *"мързеливо"* компресиране. Да започнем с “мързеливото компресиране”. Когато *LZ77* открие съвпадение на думи от дадена позиция, достатъчно дълга, за да бъде кодирана и същевременно, достатъчно кратка, за да може това да принесе голяма полза, той продължава сканирането, опитвайки се да я продължи. Нека разгледаме следния пример:

"back␣acon␣bacon␣"

Когато думата "bacon␣" попадне в буфера на *LZ77*, тази техника ще му позволи да открие две съвпадения с дължина 3, а именно "bac" и "on␣", което при 12-битов индекс ще доведе до 50% компресия. Един “мързелив” алгоритъм, като *FLZ*, ще съобрази, че намереното съвпадение не е достатъчно дълго и ще се опита да го разшири. За целта ще изпусне първия символ на "bacon␣", при което ще открие по-дълго съвпадение, а именно: "acon␣", с дължина 5 байта! Разбира се, това ще изисква допълнително битове за кодиране на броя на изпуснатите букви, но въпреки това често може да доведе до добри резултати. Впрочем, днес тази техника се радва на широка популярност — прилага се например в *PKZIP* за отмествания с дължина до 258 байта.

Друга интересна техника, прилагана от *FLZ*, е така нареченото *поглеждане в двете посоки*. За разлика от *LZ77*, който гледа само назад към вече кодираните символи, *FLZ* позволява да се гледа и в двете посоки. Нещо повече — кодираното съобщение става част от прозореца, т. е. програмата гледа не само входното съобщение, но и генерирания код. Получава се нещо като емулация на кодиране на последователности. Да разгледаме входно съобщение, съставено от

единствена буква: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx". Да го кодираме чрез *FLZ*. На първата стъпка се открива единствено еднобуквено съвпадение. На следващата стъпка програмата се премества върху следващата двойка, която ще бъде едновременно текущ символ и предходен символ. Процесът продължава, като дължината на разпознатата последователност на всяка стъпка нараства.

Задачи за упражнение:

1. Да се сравнят *FLZ* и *LZ77*.
2. Коя от двете идеи е по-перспективна: “мързеливо” компресиране или поглед в двете посоки?
3. Да се анализира взаимодействието на трите нови идеи на алгоритъма.

10.6.8. LZW. Модификацията на Уелч

Начинът, по който работи *LZW*, се различава от този на *LZ77* и *LZ78*. На първо място *LZW* използва таблица в явен вид, като най-често размерът ѝ е от порядъка на 4096 реда. Първите 256 реда съдържат кодовете на всеки от 256-те символа от таблицата *ASCII*. Следващите редове съответстват на кодове на думи с дължина, по-голяма от 1. При размер на таблицата от 4096 символа, дължината на съответния индекс е $\log_2 4096 = 12$ бита. Ако използваме само първите 256 кода, в резултат на компресирането получаваме средно 50% нарастване на дължината на съобщението, тъй като кодираме 8-битови последователности, каквито са отделните символи, с 12 бита. За останалите думи от таблицата (по-долу ще я наричаме *речник*) кодът е по-къс от входната дума. Така например, за двубайтова дума от речника, т. е. с дължина 16 бита, при кодиране получаваме 12 бита. 12 бита обаче получаваме и при трибайтова, и при 50-байтова дума… Изобщо, колкото е по-дълга думата от речника, толкова по-висока е степента на компресия. Тук възникват двата стари проблема:

1. Как да се определят думите, които да влязат в речника?
2. Как да се подсигури използване на един и същ речник при компресиране и декомпресиране?

Спецификацията на *LZW*, предложена от Уелч, дава готова рецепта (при това една и съща!) за справяне и с двата проблема. Идеята е, речникът да се строи динамично по време на кодирането. В началото се започва с таблица от 256-те символа от стандартната кодова таблица, като в процеса на работа речникът се допълва. Входното съобщение се преглежда буква по буква, като се търсят съвпадения с думите от речника. В случай, че текущата дума се съдържа в речника, към нея се прибавя следващата буква от входното съобщение, а на изхода не се извежда нищо. В един момент се достига до дума α, която не се съдържа в речника. Да обозначим с β думата α без последния ѝ символ, β се съдържа в речника. Прибавяме α към речника, като ѝ присвояваме някой от свободните кодове (предполагаме, че таблицата не е запълнена), а на изхода подаваме кода на β. Започваме нова текуща дума с първи символ: последният символ на α. Фактът, че подаваме на изхода кода на β, а не на α, е съществен, тъй като това ще ни позволи еднозначно декодиране, без нужда от предаване на речника. Получаваме следния примерен код:

```
initTable();
ch = getChar(input);
str = ch;
while(!eof(input)) {
   ch = getChar(input);
   if ((dictionaryIndex = findIndex(dictionary,str+ch)) < 0) {
      writeOut(output,findIndex(str));
      add2Dictionary(dictionary,str+ch);
      str = ch;
   }
   else
```

```
        str = str + ch;
    }
    writeOut(output,findIndex(dictionary,str));
```

Функцията `initTable()` извършва начално инициализиране на речника с 256-те символа, след което се инициализира начален низ `str`. До изчерпване на символите от входния файл от него се чете нов символ `ch` с помощта на `getChar()`. След това се извършва обръщение към функцията `findIndex()`, която връща индекса на подадения като аргумент низ `str+ch` или –1, ако такъв низ не се съдържа в речника. Ако низът не бъде намерен, той се прибавя към речника (`add2Dictionary()`), а на изхода се извежда индексът на предходната дума. Ако думата е в речника, се извършва съответно обновяване на `str`. Следва четене на нов символ, търсене в таблицата на `str+ch` и т.н. Процесът продължава до изчерпване на думите от входното съобщение.

За разлика от *LZ77* при *LZW* дължината на думите е променлива и може да расте до няколко хиляди символа. Това е свързано с разход на памет, особено при по-голям размер на речника. Възниква нужда от намаляване на размера му. Оказва се, че това може да се постигне без особени усилия. По-горе видяхме, че α попадна в речника, едва след като там се е оказала β. Т. е. всяка новопостъпила в речника дума се получава от друга дума от речника с прибавяне на *единствен* символ. Ще наричаме β родител на α. И тук възниква идеята в речника вместо дума да се пазят само последната буква от думата и индекс на родителя. Така, всяка дума ще се представя с 20 бита, а именно: 12 бита индекс на родителя и 8 бита код на последната ѝ буква. За първите 256 символа ще считаме, че нямат родител, т. е. че родителят им има индекс –1. Така успяхме с помощта на памет от порядъка на $\Theta(n)$ да запишем n различни низа, без значение каква е дължината им! *Таблица 10.6.8.* илюстрира по-подробно метода, като показва всички действия по кодирането на съобщението: *abracadabragabramabracadabragabra*.

Получихме съобщението: 0, 1, 6, 0, 2, 0, 3, 7, 9, 4, 14, 0, 5, 17, 11, 13, 8, 0, 16, 23. Доколко е добър полученият резултат? Ако предположим, че входното съобщение е зададено в стандартен *ASCII* формат, то за кодирането на всеки от 33-те символа ще бъдат необходими 33.8 = 264 бита. Ако отчетем, че входното съобщение съдържа само 7 различни букви, можем да го кодираме с 33.3 = 99 бита. От своя страна генерираният код се състои от 20 числа. Ако закодираме всяко от тях в един байт (символ), то ще получим дължина 20.8 = 160 бита. Отчитайки, че имаме само 16 различни числа, всяко от които можем да закодираме с 4 бита, получаваме дължина 4.20 = 80 бита. Вижда се, че и в двата случая имаме съществено намаляване на сължината. Не трябва обаче да се забравя, че се налага и предаване на речника, което променя нещата, особено при достатъчно къси файлове.

Оказва се обаче, че предаването на речника при декодиране съвсем не е нужно и алгоритъмът е способен да си го изгради сам. Процесът протича аналогично на кодирането. Започва се с речник, съдържащ само буквите. Всеки път, когато на входа постъпи нов код, той се декодира с помощта на речника, след което към речника се прибавя предишната дума, последвана от първия символ на току-що декодираната. Тези действия водят до изграждане на речник, идентичен с този, изграждан от кодиращия алгоритъм. Следва прочитане на нов код, после на съответен низ от речника и т.н. Процесът продължава до изчерпване на кодовете.

#	*ch*	*str+ch*	търсим		намерен?	кодова таблица			на изхода	намерен код	нов *str*
			символ	баща		код	символ	баща			
						0	*A*	–1			
						1	*B*	–1			
						2	*C*	–1			
						3	*D*	–1			
						4	*G*	–1			
						5	*M*	–1			
						6	*R*	–1			
1	***a***	*a*	*a*	–1	да	—	—	—	—	0	*a*

2	**b**	*ab*	*b*	0	не	7	*B*	0	0	—	*b*
3	**r**	*br*	*r*	1	не	8	*R*	1	1	—	*r*
4	**a**	*ra*	*a*	6	не	9	*A*	6	6	—	*a*
5	**c**	*ac*	*c*	0	не	10	*C*	0	0	—	*c*
6	**a**	*ca*	*a*	2	не	11	*A*	2	2	—	*a*
7	**d**	*ad*	*d*	0	не	12	*D*	0	0	—	*d*
8	**a**	*da*	*a*	3	не	13	*A*	3	3	—	*a*
9	**b**	*ab*	*b*	0	да	—	—	—	—	7	*ab*
10	**r**	*abr*	*r*	7	не	14	*R*	7	7	—	*r*
11	**a**	*ra*	*a*	6	да	—	—	—	—	9	*ra*
12	**g**	*rag*	*g*	9	не	15	*G*	9	9	—	*g*
13	**a**	*ga*	*a*	4	не	16	*A*	4	4	—	*a*
14	**b**	*ab*	*b*	0	да	—	—	—	—	7	*ab*
15	**r**	*abr*	*r*	7	да	—	—	—	—	14	*abr*
16	**a**	*abra*	*a*	14	не	17	*A*	14	14	—	*a*
17	**m**	*am*	*m*	0	не	18	*M*	0	0	—	*m*
18	**a**	*ma*	*a*	5	не	19	*A*	5	5	—	*a*
19	**b**	*ab*	*b*	0	да	—	—	—	—	7	*ab*
20	**r**	*abr*	*r*	7	да	—	—	—	—	14	*abr*
21	**a**	*abra*	*a*	14	да	—	—	—	—	17	*abra*
22	**c**	*abrac*	*c*	17	не	20	*C*	17	17	—	*c*
23	**a**	*ca*	*a*	2	да	—	—	—	—	11	*ca*
24	**d**	*cad*	*d*	11	не	21	*D*	11	11	—	*d*
25	**a**	*da*	*a*	3	да	—	—	—	—	13	*da*
26	**b**	*dab*	*b*	13	не	22	*B*	13	13	—	*b*
27	**r**	*br*	*r*	1	да	—	—	—	—	8	*br*
28	**a**	*bra*	*a*	8	не	23	*A*	8	8	—	*a*
29	**g**	*ag*	*g*	0	не	24	*G*	0	0	—	*g*
30	**a**	*ga*	*a*	4	да	—	—	—	—	16	*ga*
31	**b**	*gab*	*b*	16	не	25	*B*	16	16	—	*b*
32	**r**	*br*	*r*	1	да	—	—	—	—	8	—
33	**a**	*bra*	*a*	8	да	—	—	—	—	23	—
34	—	—	—	—	—	—	—	—	23	—	—

Таблица 10.6.8. *LZW*: Кодиране на съобщението "*abracadabragabramabracadabragabra*".

На пръв поглед подобна схема работи правилно. Въпреки това, понякога може да се случи така, че постъпилият вход да не фигурира в речника. Да разгледаме например последователност от вида *cScScp*, където *S* е низ (последователност от символи), а *c* и *p* са символи. Да предположим, че *cS* вече е бил включен в речника, но *cSc* все още не е. Нека при тези условия да проследим работата на компресиращия алгоритъм върху нашия низ *cScScp*. Първо, алгоритъмът открива, че *c* е в речника (защо?), след това, за още няколко стъпки, открива, че и *cS* е в речника, след което открива, че *cSc* не е в речника. На изхода постъпва индексът на *cS*, а *cSc* постъпва в речника. Кодирането продължава с едносимволния низ *c*, който принадлежи на речника. След това на входа постъпва *S*, при което кодиращият алгоритъм открива, че и *cS* е там. После последователно постъпват *c* и *p*. При постъпването на *c cSc* се оказва в речника, а при постъпване на *p cScp* вече не е там. В този момент компресиращата програма ще изведе на изхода кода на *cSc*, който току-що е бил добавен в таблицата. При декомпресиране обаче се оказва, че речникът не съдържа *cSc*. Защо се получава така? Отговорът е прост: Декомпресирането работи *обратно* на компресирането, т. е. започва с *кода*, при което върви една стъпка назад. Въпреки нашето старание да извеждаме кода едва при повторното срещане на низа в текста, което би трябвало да ни предпази от подобни ситуации, проблемът успя да се промъкне. За компресиращия алгоритъм не е проблем да изведе на изхода току-що генерирания код в ситуации, подобни на горната, но декомпресорът разчита всеки код да се съдържа в речника. На

практика, подобен проблем възниква само в тази ситуация, което силно улеснява решаването му: ако прочетеният код не е в речника, от речника се прочита низът, съответстващ на предишния код, след което към края му се прибавя първият му символ. Това е именно низът, съответстващ на липсващия код. Получаваме следния окончателен алгоритъм за декомпресиране:

```
initTable();
getCode(oldCode);
writeOut(output,getStr(dictionary,oldCode));
while(!eof(input)) {
  getCode(input,code);
  if (!inTable(code)) {
    str = getStr(dictionary,oldCode);
    str = str + str[0];
  }
  else
    str = getStr(dictionary,code);
  writeOut(output,str);
  add2Dictionary(oldCode,str[0]); /* двойка, съотв. на един низ */
  oldCode = code;
}
```

Следва проста реализация на изложените *LZW* алгоритми:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <process.h>

#define MAX_S 10
#define CHAR_CNT 256
#define DICT_SIZE 4096
#define ENCODED_SIZE 1000
#define MAX_MSG_LEN 1000
#define NOT_FOUND (unsigned)(-1)

const char *msg =
   "abracadabragabramabracadabragabraLALALALALALALALALALALALALALA";

char dictionary[DICT_SIZE][MAX_S]; /* Речникът */
unsigned dictIndex;                /* Максимален индекс в речника */
unsigned encoded[ENCODED_SIZE];    /* Кодирано съобщение */
unsigned encIndex;                 /* Максимален индекс на кода */

void initTable(void) /* Инициализира таблицата */
{ unsigned i;
  for (i = 0; i < CHAR_CNT; i++) {
    dictionary[i][0] = i;
    dictionary[i][1] = '\0';
  }
  dictIndex = CHAR_CNT;
}
unsigned findIndex(char *s) /* Търси индекс в таблицата */
{ unsigned i;
  for (i = 0; i < dictIndex; i++)
    if (0 == strcmp(s,dictionary[i]))
      return i;
  return NOT_FOUND;
}
```

```
void add2Dictionary(char *s) /* Добавя нов низ в таблицата */
{ strcpy(dictionary[dictIndex++],s); }

void LZWencode(const char *msg) /* Извършва кодиране по LZW */
{ unsigned dictIndex, len;
  char ch, str[MAX_S], strch[MAX_S];

  initTable();
  encIndex = 0;
  *str = ch = *msg++; *(str+1) = '\0';
  while ('\0' != *msg) {
    ch = *msg++;
    strcpy(strch,str);
    strch[len = strlen(strch)] = ch; strch[len+1] = '\0';
    dictIndex = findIndex(strch);
    if (NOT_FOUND == dictIndex) {
      encoded[encIndex++] = findIndex(str);
      add2Dictionary(strch);
      *str = ch; *(str+1) = '\0';
    }
    else {
      str[len = strlen(str)] = ch; str[len+1] = '\0';
    }
  }
  encoded[encIndex++] = findIndex(str);
}

void LZWdecode(char *decodedMsg) /* Извършва декодиране по LZW */
{ unsigned code, oldCode, ind, len;
  char str[MAX_S], str2[MAX_S];

  initTable(); ind = 0;
  oldCode = encoded[ind++];

  strcpy(decodedMsg,dictionary[oldCode]);
  while(ind < encIndex) {
    code = encoded[ind++];
    if (code >= dictIndex) {
      strcpy(str,dictionary[oldCode]);
      str[len = strlen(str)] = *str; str[len+1] = '\0';
    }
    else
      strcpy(str,dictionary[code]);
    strcat(decodedMsg,str);
    strcpy(str2,dictionary[oldCode]);
    str2[len = strlen(str2)] = *str; str2[len+1] = '\0';
    add2Dictionary(str2);
    oldCode = code;
  }
}

void printEncodedMsg(const unsigned *encoded, const unsigned encIndex)
{ unsigned ind;

  printf("\n\nКодирано съобщение:\n");
  for (ind = 0; ind < encIndex; ind++)
    printf("%u ", encoded[ind]);
  printf("\n");
```

```
}

int main(void) {
  char decodedMsg[MAX_MSG_LEN];
  printf("\n\nВходно съобщение за кодиране:\n%s",msg);

  LZWencode(msg);
  printEncodedMsg(encoded, encIndex);

  LZWdecode(decodedMsg);
  printf("\nДекодирано съобщение:\n%s", decodedMsg);
  return 0;
}
```

🖫 lzw.c

Резултат от изпълнението на програмата:

```
Входно съобщение за кодиране:
abracadabragabramabracadabragabraLALALALALALALALALALALALALALALA

Кодирано съобщение:
97 98 114 97 99 97 100 256 258 103 263 97 109 266 260 262 257 97 265 272
76 65 276 278 277 280 279 282 281 284

Декодирано съобщение:
abracadabragabramabracadabragabraLALALALALALALALALALALALALALALA
```

Вижда се, че както компресирането, така и декомпресирането са достатъчно прости като алгоритми. Елементарна компресираща/декомпресираща програма ще съдържа няколко десетки реда. За съжаление, тя едва ли ще бъде особено ефективна. Основният проблем е начинът на управление на речника. Както по-горе беше споменато, алгоритъмът *LZW* не позволява на речника да расте до безкрайност, а му поставя горна граница, най-често от 4096 елемента. Това предполага предвиждане на механизми за освобождаване на място в речника при евентуалното му препълване. Съществуват множество различни подходи. Един възможен подход е, при препълване на речника, новите низове да не се включват в него, като същевременно се следи постиганото ниво на компресия. Когато то спадне, речникът се инициализира с началните 256 символа, след което започва се изгражда наново. Това обаче изисква синхронизация на компресиращия и декомпресиращия алгоритъм, така че те да нулират речника си в едно и също време.

Друг основен проблем при *LZW* е търсенето. Действително, търсене се извършва поне веднъж за всеки символ от входното съобщение. Ето защо последователното преглеждане на речника очевидно е неефективно решение. Най-добре е да се използва хеш-таблица (*виж 2.4.*). В този случай трябва да се внимава нивото ѝ на запълване да не надвишава 80%, тъй като времето за търсене започва да нараства бързо. Алтернативно решение е използването на частично нареден речник, разделен на секции. Ключът за влизане в дадена секция се задава от символа *ch* (да си припомним представянето на думите в речника: *код_на_родител+ch*).

Други проблеми пред ефективната реализация на алгоритъма са ефективното четене на входните и записът на изходните данни, натрупването на подходящи статистики и други. Редица комерсиални продукти използват описания метод, комбинирайки го успешно с други ефективни методи, като Хъфманово кодиране, както и редица други техники, които ревниво пазят като търговска тайна. Читателят не би следвало да очаква лесно постигане на степен и скорост на компресия, близки до тези на комерсиалните продукти, без сериозни допълнителни усилия и проучване на достатъчно допълнителна литература. Освен това не бива да се забравя, че *LZW* е *патентован* алгоритъм, което би могло в перспектива да доведе до проблеми, поради което обикновено се избягва. Да си припомним, че най-популярните компресиращи програми като *ZIP* и *ARJ* използват *LZ77*. (Впрочем, по наше скромно мнение, главно по тази причина.)

Задачи за упражнение:

1. Следва ли да се очаква подобрение, ако се остави речникът да расте неограничено? Какви проблеми възникват и как могат да се разрешат?
2. Защо на изхода се подава кодът на β, а не на α (виж началото на параграфа)?
3. Да се оцени степента на компресия за примера от *таблица 10.6.8.* при положение, че се предава речникът.
4. Да се докаже, че описаният по-горе случай е единственият, когато при декодиране може да се получи код, за който няма дума в речника.
5. Каква е сложността на *LZW*? А на предложената реализация?
6. Да се сравнят *LZW* и *LZ77*.
7. Да се предложи и реализира адаптивен вариант на *LZW*.
8. Да се предложи начин за комбиниране на *LZW* и кодиране по Хъфман.

10.6.9. GIF. Гледната точка на CompuServe

По-долу ще разгледаме варианта на *LZW*, използван от *CompuServe* в техния графичен файлов формат: *GIF*. За разлика от разгледаните по-горе универсални алгоритми *GIF* е файлов формат за компресиране и съхранение на графични изображения. Съществуват два широкоизползвани стандарта на *GIF*, а именно *GIF87* и *GIF89*. Няма да се спираме подробно на техните особености. Ще отбележим само, че най-съществената разлика между двата е, че *GIF89* може да съхранява не само *статични изображения*, но и прости *анимации*. За целта се допуска файлът да съдържа няколко изображения, както и допълнителна информация за времетраенето на всяко от тях, начина на преминаване помежду им и др. Алгоритъмът за компресиране *LZW* е вграден в *GIF*, като са направени две интересни модификации [*Blackstock-1993*]:

1. В речника са въведени два допълнителни кода със специално предназначение.
2. Индексите в речника имат различна дължина в байтове.

Първо, следва да се отбележи, че при кодиране на изображения началната таблица не е непременно от 256 символа. Впрочем, това е така и при кодирането на обикновен *ASCII* текст: не всеки символ се среща във входното съобщение, поради което не на всеки следва да се съпоставя индекс в речника. При изображенията на буквата от текстовото съобщение съответства пикселът. В зависимост от броя на достъпните цветове броят битове за пиксел може да варира, но винаги е степен на двойката. Ако например броят на битовете за пиксел за дадено конкретно изображение е n, то ще му бъдат достъпни 2^n на брой различни цвята с кодове от 0 до $2^n–1$. Тези 2^n различни цвята ще представляват началната таблица. Към тях ще прибавим още два допълнителни символа, които условно ще бележим с <*EOI*> (от англ. *End Of Information*) и <*CC*> (от англ. *Clear Code*), като ще им присвоим кодове 2^n и 2^n+1 съответно. Така индексите в нашата таблица ще бъдат дълги поне $n + 1$ бита. При *GIF* началните индекси са с дължина точно $n + 1$ бита.

Нека се спрем по-подробно на смисъла на двата нововъведени кода. <*EOI*> представлява специален маркер за край на входните данни: <*EOI*>. Той указва на декомпресиращата програма края на кодираното съобщение. <*CC*> служи за синхронизация между компресиращия и декомпресиращия алгоритъм. Тъй като двата допълнителни кода заемат съответно индекси 2^n и $2^n + 1$, то първият новодобавен код ще бъде от позиция $2^n + 2$. Компресирането започва с извеждане на <*CC*> на изхода като първи код, с което се гарантира, че декомпресиращата програма ще започне с инициализиран речник. Следващият <*CC*> ще бъде изведен при цялостно запълване на речника, т. е. при достигане на индекс 4095. (*GIF* не позволява индексът на речника да надвишава 12 бита.) Въвеждането на <*CC*> се оказва изключително удобно, тъй като позволява преждевременно изпразване на речника. Това може да се окаже необходимо в редица случаи, например при използване на хеш-таблици, където не бива да се допуска запълването ѝ над определена степен, тъй като това я прави неефективна.

Освен двата допълнителни кода *GIF* предвижда променлива дължина на кодовете. Идеята е ясна: когато речникът съдържа например 40 числа, би било най-добре да ги кодираме с 6 бита, вместо с 12. При кодиране в началото се използват $k = n + 1$ бита. Когато максималният индекс в

речника достигне стойност 2^k-1, се извършва присвояването $k = k + 1$, след което кодирането продължава с индекс с дължина, с 1 по-голяма. Забележете, че този по-дълъг индекс се използва не само за новите, големите стойности, но и за тези, които досега са се кодирали с по-малко битове. В противен случай се губи възможността за еднозначно декодиране. При декомпресиране отново се започва с $k = n + 1$, като впоследствие стойността на k нараства. Следва да се отбележи, че стойността на k следва да се увеличи в момента на включване на 2^k-1 в речника, а не при включване на 2^k, в който случай ще бъде изгубен един бит. Причината отново е в това, че компресиращата програма върви една стъпка преди декомпресиращата.

Задачи за упражнение:

1. Какви други подобрения биха могли да се направят?
2. Да се сравнят настоящият алгоритъм и *LZW*.

10.6.10. Оптимални срещу алчни алгоритми

Всички разгледани по-горе варианти на речниковото кодиране са алчни (*виж 9.1.*) и по никакъв начин не се стремят да търсят оптималната комбинация от некодиран текст и индекси. Да разгледаме вариант на речниковото кодиране, при който кодирането на единствен символ изисква 8 бита, а на индекс в речника — 16 бита. Да предположим, че речникът съдържа следните думи: "ПРО", "_ПР", "РОСТ" и "ПРИМЕР". Горните алчни алгоритми биха закодирали съобщението “ПРОСТ_ПРИМЕР”, както е показано в *таблица 10.6.10а.*

Индекс на	*Символ*	*Символ*	*Индекс на*	*Символ*	*Символ*	*Символ*	*Символ*
ПРО	С	Т	_ПР	И	М	Е	Р

Таблица 10.6.10а. Речниково кодиране (с помощта на алчен алгоритъм) на съобщението “*ПРОСТ_ПРИМЕР*” при речник: "*ПРО*", "*_ПР*", "*РОСТ*" и "*ПРИМЕР*".

Така ще се получи код с обща дължина: 16 + 8 + 8 + 16 + 8 + 8 + 8 + 8 = 80 бита. Същевременно, оптималният вариант (*виж таблица 10.6.10б.*) изисква едва 48 бита (8 + 16 + 8 + 16 = 48), т. е. получаваме подобрение от 40%.

Символ	*Индекс на*	*Символ*	*Индекс на*
П	РОСТ	_	ПРИМЕР

Таблица 10.6.10б. *Оптимално* речниково кодиране на съобщението “*ПРОСТ_ПРИМЕР*” при речник: "*ПРО*", "*_ПР*", "*РОСТ*" и "*ПРИМЕР*".

В реалните практически примери обаче резултатът от изпълнението на алчен алгоритъм се различава слабо от този на оптималната схема, като разликите обикновено варират в рамките на няколко процента. Същевременно, намирането на оптималната схема е изключително сложен проблем, водещ до силно снижаване скоростта на работа на алгоритъма, поради което се счита, че търсенето на оптималната кодираща схема не си струва усилията.

Задачи за упражнение:

1. Решава ли мързеливото кодиране на *FLZ* (*виж 10.6.7.*) описания проблем?.
2. Каква е алгоритмичната сложност на потенциален алгоритъм за намиране на оптималната кодираща схема?
3. Нека при кодиране с *LZW* в даден момент речникът съдържа думите "*ПРО*", "*_ПР*", "*РОСТ*" и "*ПРИМЕР*". Кои други думи задължително също трябва да са там?

10.6.11. Компресиране в реално време

Речниковото кодиране се оказва наистина универсално и започва да навлиза с редица други специфични области. Така например, *Stac Electronics* го въвежда в популярния си продукт *Stacker* за компресиране на данните върху диска в реално време. По-късно този метод е въведен и от *Central Point Software* в техния *PC Backup*. *Microsoft* също скоро се намесва на пазара, предлагайки конкурентните на *Stacker* продукти *DoubleSpace* (по-късно *DriveSpace*) стандартно към своя *MS-DOS* версии 5.0 и следващи, а по-късно успява и да закупи *Stac Electronics* и да вгради съответната функционалност в *Windows*. Компресирането и декомпресирането на целия диск или части от него в реално време има своите безспорни предимства. Главното е увеличаването на размера на свободното дисково пространство, като обикновено се постига степен на компресия между 1,7:1 и 2,1:1 в зависимост от типа данни, съхранявани на диска. Методът се оказва изключително ефективен при големи дискове, съдържащи множество малки файлове. Повечето файлови системи изискват всеки файл да заема цяло число сектори, като всеки сектор принадлежи на най-много един файл. При големи дискове размерът на секторите нараства, при което някои достатъчно малки файлове могат да заемат незначителна част от сектора, например 5-10%, което води до големи разхищения. Впрочем, подобен проблем често се среща и при по-големи файлове, при които обикновено последният сектор е почти незапълнен. Програми като *Stacker* и *DoubleSpace* решават проблема, като съхраняват всички файлове в единствен компресиран файл.

Реализирането на такъв метод е сложен проблем по няколко причини. Първо, той рядко започва да компресира добре от самото начало, поради използването на адаптивни схеми, изискващи натрупването на съответни статистики. Алтернативната възможност за използване на фиксирани статични речници очевидно е по-лоша в перспектива. Основният проблем обаче е друг: За разлика от модемите, лентовите устройства и последователните файлове твърдите дискове са устройства с *произволен* достъп. Това означава, че трябва да се гарантира на програмите начин за пряк достъп до произволна позиция на всеки файл, което поражда редица проблеми. Очевидно, декомпресирането на целия файл, последвано от даване на достъп до исканата позиция, е крайно неефективен подход, особено при големи файлове. Повечето компресиращи програми решават проблема, компресирайки на ниво сектор, като при всеки нов сектор моделът се инициализира наново. Тъй като данните, така или иначе *физически* се четат и записват на сектори, такъв подход се оказва изключително ефективен.

Разбира се, компресирането в реално време има и своите противници. Едно от основните им опасения се отнася до сигурността на данните в случай на грешки, породени от прекъсване на тока, вирус и др. Вторият довод против е свързан с евентуалното забавяне на дисковите операции, тъй като всяка такава операция е свързана с процес на компресиране или декомпресиране. На практика обаче, се оказва, че това е несъстоятелно, тъй като в повечето случаи вместо забавяне, се наблюдава *ускоряване*. Причината е проста — компресирането означава, че реално на твърдия диск се записват и съответно прочитат *по-малко* данни. И тъй като времето, необходимо за компресирането/декомпресирането, е по-малко от спестеното в резултат на намаленото време за четене/писане от твърдия диск, в крайна сметка се получава ускорение.

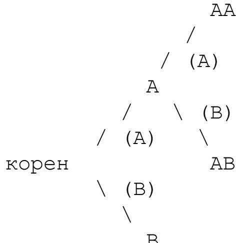
Задачи за упражнение:

1. Кои други разгледани алгоритми са подходящи за компресиране в реално време на ниво сектор?
2. Да се предложи начин за компресиране в реално време, различен от компресирането на ниво сектор.

10.6.12. LZW срещу Марков

Ще направим опит за сравнение между речниковия метод на кодиране на Лемпел, Зив и Уелч (*LZW, виж 10.6.8.*) и моделите на Марков 5от по-висок ред (*виж 10.5.2.*).

На пръв поглед изглежда, че алгоритмите на Лемпел-Зив-Уелч и Марков са дотолкова различни като концепция, че едва ли има база за сравнение. Оказва се обаче, че *LZW* може да се разглежда като частен случай на алгоритмите на Марков. По-долу ще покажем най-общо как може да стане това.



Фигура 10.6.12а. Речниково дърво за речника: {*A*, *B*, *AB*, *AA*}.

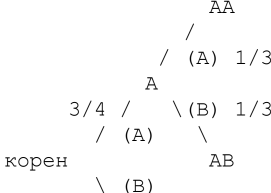
Ще се спрем по-подробно на *LZW* и ще покажем как може да бъде сведен до алгоритъм на Марков. Както знаем, *LZW* поддържа вечно растящ (в общия случай) речник. Алгоритъмът сканира текста и евентуално го обогатява с нова дума, съставена от текущата дума (която е дума от речника), конкатенирана с текущия символ. По-долу на речника ще гледаме като на дърво. Всеки негов връх с изключение на корена (който е празен) съдържа низ, представляващ дума от речника. Ясно е, че произволен речник може да се представи в такъв вид. На ребрата на дървото ще съпоставим символа, който трябва да се прибави към думата, съдържаща се в бащата, така че да бъде получена думата, съдържаща се в сина. Така например, за речника {*A*, *B*, *AB*, *AA*} получаваме речниковото дърво от *фигура 10.6.12а.* (символите, съпоставени на ребрата, сме заградили в скоби).

Нека предположим, че всяка дума от речника се появява с еднаква вероятност 1/4 (*виж фигура 10.6.12б.*).

```
                AA 1/4
               /
             / (A)
            A 1/4
          /   \ (B)
        / (A)   \
корен           AB 1/4
        \ (B)
          \
            B 1/4
```

Фигура 10.6.12б. Речниково дърво за речника: {*A*, *B*, *AB*, *AA*}, с вероятности на *върховете*.

Сега лесно можем да пресметнем вероятностите на ребрата (На последно ниво имаме 1/3, защото върхът *A* представлява мислена дъга и също има вероятност 1/3, *виж 10.6.12в.*):



```
              1/4 \
                   B
```

Фигура 10.6.12в. Речниково дърво за речника: {*A*, *B*, *AB*, *AA*}, с вероятности на *ребрата*.

Преминаването през дървото, извършвано от *LZW* с цел откриване на най-дългото съвпадение и добавянето на нова кодова дума, може да се разглежда като серия *еднобуквени предвиждания*, всяко от които се основава на *собствена вероятностна таблица* и може да бъде закодирано поотделно с използване на *аритметично кодиране*, при което би се постигнала сходна ефективност, т.е. *LZW* може да се разглежда като алгоритъм на Марков. При това показахме и връзката му с аритметичното кодиране.

Нека сега разгледаме по-подробно един конкретен възел *X* от дървото. При всяко преминаване през *X LZW* добавя нов връх някъде в поддървото на *X*. Колкото по-голяма е вероятността за поява на *X*, толкова повече върхове ще съдържа поддървото му. Така, всеки връх се оказва корен на дърво с големина, пропорционална на вероятността му за срещане. По-късно на всеки връх ще бъде съпоставен код с дължина, обратно пропорционална на размера на поддървото му. Оставяйки дървото да расте до безкрайност върху безкрайно съобщение, ще достигнем *ентропията* на входното съобщение.

Вижда се, че алгоритъмът се справя доста добре. И все пак, защо ефективността му е толкова ниска в сравнение с тази на алгоритмите на Марков? Причините за това са няколко. От една страна вероятностите в близост до листата са неверни, а от друга – приблизително половината от кодовото пространство (кодовете на листата) във всеки един момент не се използва.

Друг не по-малко съществен проблем е неефективният начин на изработване на код на първите две стъпки. За разлика от алгоритмите на Марков, където всяко кодиране се основава на трибуквен контекст (ако е възможно), на първите две стъпки на *LZW* се използва контекст от ред 0 и 1 съответно. Това по-късно дава отражение при кодирането на произволна дума. За сметка на това обаче *LZW* позволява *неограничено* по принцип нарастване на дървото, като се опитва да компенсира по този начин това, което губи при първите две букви. Т. е. опитва се чрез увеличаване на средната дължина на думата да намали вероятността за срещане на първите ѝ две букви. За съжаление, в общия случай дълбочината на дървото не нараства достатъчно бързо, за да може да компенсира успешно неефективното кодиране в началото.

И накрая – още една причина за лошото представяне на *LZW* в сравнение с Марков. *LZW* преминава през дървото *само веднъж*. Това означава, че вероятността на срещане на първата буква от дадена дума се пресмята на базата само на първата буква от думите в речника. Ако същата буква се появи някъде по-надолу в дървото, появата ѝ не се взема предвид. Същевременно, алгоритъмът на Марков взема предвид *всички появи* на всички букви. Такъв проблем съществува на всички нива в речниковото дърво на *LZW* (*виж* [*Williams-1991*]).

Задачи за упражнение:

1. Да се обясни по-подробно връзката на *LZW* с аритметичното кодиране.
2. Да се предложи подходяща комбинация на *LZW* и модел на Марков от по-висок ред.
3. Кои от проблемите на *LZW* по отношение на моделите на Марков от по-висок ред се отнасят и за *LZ77*?

10.7. Компресиране със загуба

10.7.1. Изрязване и квантифициране

Най-простият метод за компресиране със загуба е прякото изрязване на някои от битовете и/или квантифициране, известен в англоезичната литература като *CS&Q* (от англ. *coarser sampling and/or quantization*). Да предположим, че искаме да компресираме графично изображение, цветът на всяка точка от което се определя от 24-бита. Ако "изрежем" 3 бита по подходящ начин, качес-

твото на изображението почти няма да се промени. Същевременно, ще сме успели да намалим размера на файла с изображението с 1/8. Такива техники са приложими не само при графични изображения, но и при звук, видео и др. Изобщо, навсякъде където се работи с неточни данни от вълнов характер, произтичащи от заобикалящата ни действителност.

Ще приведем още един пример. Да предположим, че в резултат на някакви измервания сме получили две различни извадки за изменението на температурата в стоманолеярна пещ, измерени през едно и също време от два различни сензора. Всяка от извадките се състои от последователност от 12-битови числа. Т. е. на входа постъпват 24-битови данни. В процеса на компресиране на всяка стъпка по някакъв критерий ще избираме стойността, отчетена от единия от двата сензора, а другата ще отхвърляме. Освен това ще изрязваме най-младшите 4 бита. В резултат получаваме 8-битов код, постигайки трикратно намаление на размера на входното съобщение.

Понякога вместо (или заедно с) изрязване на битове се извършва умножение/деление на някакъв коефициент, който може да бъде различен за различните части на входната последователност. Този процес се нарича *квантифициране*. Подобна техника се прилага например при компресирането на графични изображения *JPEG*. [*Smith-1998*]

Задача за упражнение:

Да се обясни как квантифицирането може да помогне за постигане на по-добра компресия и да се даде конкретен пример.

10.7.2. JPEG

Понякога, при компресиране на звук, графика или видеоизображение, се оказва допустимо прилагането на компресиране, при което част от данните да се губят и да не могат да се декомпресират с точност 100%. Такива методи могат да бъдат изключително ефективни. Съществува цял клас алгоритми за компресиране със *загуба*, като най-широко приложение имат алгоритмите за компресиране чрез трансформация.

Най-често използваната трансформация е *преобразуване на Фурие*. Преобразуването на Фурие разполага реалната функция на времето $X(t)$ върху комплексна честотна функция $F(f)$. Преобразуването се задава с формулите (права и обратна посока):

$$F(f) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(t) e^{-i2\pi ft} dt$$

$$X(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(f) e^{i2\pi ft} df$$

На практика, обикновено не се работи по тези формули, а вместо това се използват следните техни дискретни варианти:

$$F(f) = \frac{1}{N} \sum_{t=0}^{N-1} X(t) e^{\frac{-i2\pi ft}{N}}$$

$$X(f) = \sum_{f=0}^{N-1} F(t) e^{\frac{i2\pi ft}{N}}$$

След извършване на трансформацията, данните придобиват друго представяне и донякъде друг смисъл: Главен носител на информация се оказват ниско-честотните компоненти, докато високочестотните се оказват до голяма степен маловажни. Сега, отрязвайки 50% от битовете с висока честота, ще загубим едва около 5% от данните.

Един най-известните алгоритми за кодиране чрез трансформация е *JPEG*, получил името си от англ. *Joint Photographers Experts Group*. *JPEG* използва дискретно косинусово преобразуване *DCT* (от англ. *Discrete Cosine Transform*). (По принцип най-добра степен на компресия се постига при използване на трансформацията на Кархунен-Лоеве, чиято реализация, за съжаление,

е свързана с редица трудности, поради което почти не се прилага.) Основно предимство на *DCT* е, че работи с реални, вместо с комплексни числа, което значително опростява и ускорява пресмятанията. Как се постига това? Няма да се спираме на всички известни варианти на *DCT*, а само на идеята на най-простия от тях. Да предположим, че искаме да преобразуваме дискретен сигнал със 129 измервания (точки) с номера от 0 до 128. Ще конструираме симетричен 256-точков сигнал, удвоявайки и обръщайки точките от 1 до 127, при което получаваме: 0, 1, 2, 3, …, 127, 128, 127, …, 3, 2, 1. Тъй като времевите моменти са симетрични, то, ако приложим преобразуване на Фурие, всички имагинерни части ще станат нули. Получихме 129 точков честотен спектър, с косинусови амплитуди.

Алгоритъмът на *JPEG* разделя входното изображение на групи от по 64 пиксела. Всяка група представлява двумерен масив 8×8 пиксела и се третира отделно от останалите групи в процеса на компресиране и декомпресиране. Да вземем например изображение с 256 степени на сивото. Тогава цветът на всяка точка от изображението може да приема стойности от 0 до 255 и за кодирането ѝ е достатъчен 1 байт. Така размерът на всяка от групите ще бъде 64 байта. В процеса на кодиране те ще бъдат редуцирани до например 20 байта (или дори 2 при висока степен на компресия). При декомпресиране въз основа на съхраненото в тези 20 (2) байта, ще се извърши приблизителното възстановяване на цялата група. *JPEG* използва следния клас базисни функции при преобразуването (x и y са индекси в групата 8×8, а u и v са индекси в честотния спектър):

$$b(x, y) = \cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

На всяка от групите 8×8 съответства спектър 8×8, т. е. става въпрос за преобразуване на 64 реални в други 64 също реални числа (нямаме комплексна аритметика). В резултат на преобразуването в горния ляв ъгъл на спектъра се оказват ниските честоти, а в долния десен — високите.

Подобно на повечето съвременни методи на компресиране на данни *JPEG* представлява успешна комбинация на няколко метода. На първата стъпка на алгоритъма се формират групи 8×8 и към всяка от тях се прилага *DCT*. След това всеки от получените спектри се компресира чрез премахване на някои от битовете и елиминиране на някои от компонентите. Това се извършва с помощта на *квантифицираща таблица*, отново с размери 8×8. Квантифициращата таблица играе изключително съществена роля, тъй като управлява степента на компресия, която ще бъде постигната. По-долу са показани две примерни квантифициращи таблици. Първата (*виж фигура 10.7.2а.*) съответства на ниска степен на компресия, а втората (*виж фигура 10.7.2б.*) — на висока.

1	1	1	1	1	2	2	4
1	1	1	1	1	2	2	4
1	1	1	1	2	2	2	4
1	1	1	1	2	2	4	8
1	1	2	2	2	2	4	8
2	2	2	2	2	4	8	8
2	2	2	4	4	8	8	16
4	4	4	4	8	8	16	16

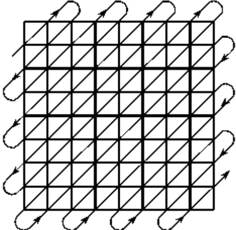
Таблица 10.7.2а. Квантифицираща таблица, съответстваща на *ниска степен на компресия*.

1	2	4	8	16	32	64	128
2	4	4	8	16	32	64	128
4	4	8	16	32	64	128	128
8	8	16	32	64	128	128	256
16	16	32	64	128	128	256	256
32	32	64	128	128	256	256	256

64	64	128	128	256	256	256	256
128	128	128	256	256	256	256	256

Таблица 10.7.2б. Квантифицираща таблица, съответстваща на *висока степен на компресия*.

Всяка от стойностите на спектъра се дели на съответната стойност в квантифициращата таблица и резултатът се закръгля до най-близкото цяло число. Следва процес на линеаризация, извършвана по начина, показан на *фигура 10.7.2а.* (*виж* [*Smith-1998*]). В резултат се получават относително дълги последователности от еднакви числа, които на следващата стъпка се кодират с помощта на алгоритъма за *кодиране на последователности* (*виж 10.3.9.*). Заключителната стъпка на алгоритъма е *Хъфманово кодиране* (*виж 10.4.2.*), в резултат на което се получава крайният компресиран файл. *Фигура 10.7.2б.* дава най-обща представа за резулатите от прилагането на метода (сравнете с *фигура 10.7.4а.*).



Фигура 10.7.2а. Схема на линеаризация на групата.

Фигура 10.7.2б. Компресиране на графично изображение с *JPEG*: 1:1, 1:10 и 1:45.

Задачи за упражнение:

1. Да се сравнят алгоритмите на *GIF* и *JPEG*. Кога е за предпочитане да се използва единият и кога — другият?
2. Да се обясни ползата от всеки от алгоритмите в *JPEG* и да се обоснове последователността им.

10.7.3. Компресиране на видеоизображение. MPEG

Компресирането на *видеоизображение* се различава от компресирането на *статично* изображение. Става въпрос за сложни входни съобщения, представляващи звук и картина едновременно. От една страна степента на компресия е ограничена от изискването компресирането и декомпресирането да работят в *реално време*, т. е. достатъчно бързо, за да могат да се прожектират директно, без нужда от предварително декомпресиране. От друга страна, видеоизображенията се състоят от множество кадри, като най-често разликите между два последователни кадъра са минимални. Това дава потенциална възможност за значителна степен на компресия.

Един от най-популярните алгоритми за компресиране на видеоизображения е *MPEG* (от англ. *Moving Pictures Experts Group*). Подобно на *JPEG MPEG* комбинира по подходящ начин множество методи, като *относително кодиране* (*виж 10.3.7.*), *линейно предсказване* (*виж 10.3.8.*) и др. На практика, самият *JPEG* е съставна част от *MPEG*. Няма да се спираме подробно на начина на работа на алгоритъма, тъй като е прекалено сложен, а само ще изложим някои от идеите му.

MPEG използва два вида компресия: в *рамките на кадъра* и *между кадрите*. Кодирането в рамките на кадъра представлява компресиране на *JPEG*, с някои незначителни изменения. Отделен компресиран по този начин кадър се нарича *I-picture* (от англ. *intra-coded picture*). В *MPEG* са въведени още две подобни понятия като *P-picture* (от англ. *predictive-coded picture*) и *B-picture* (от англ. *bidirectional predictive-coded picture*).

Както по-горе беше споменато, при видеоизображенията с голяма вероятност разликите между два последователни кадъра са незначителни. За да се възползва от това наблюдение, *MPEG* прилага относително кодиране между последователните кадри. След като е закодирала даден кадър като *I-picture*, *MPEG* кодира следващите няколко с помощта на прогнозен код, т. е. като *P-picture*. Освен това *MPEG* допуска *P-picture* да се обръща към *I-picture*, изместена на няколко пиксела, както и използване на двупосочен прогнозен код: *B-picture*, която се обръща (предсказва) както към бъдещия, така и към предходния кадър.

При декомпресиране на *MPEG* се поддържа буфер от поне три кадъра: по един за предсказване напред и назад. Третият съдържа текущото изображение, което трябва да бъде получено. На пръв поглед тук възниква проблем: *B-picture* извършва обръщение както към предходния, така и към следващия кадър. Ако следващият кадър не е зададен директно (т. е. имаме *P-picture* или *B-picture*), ще се наложи временно потискане на декодирането на текущия кадър, кодиран с *B-picture*, до достигане на *I-picture*. Едва тогава ще бъде възможно връщане назад и декодиране на пропуснатите кадри. Проблемът е решен в *MPEG* чрез разместване на кадрите така, че тези, към които се извършват обръщения винаги са разположени *преди* извършващите обръщенията. Така например, следната последователност от кадри: IBBBPBBBP ще бъде предадена от *MPEG* в последователността IPBBBPBBB. След декомпресиране програмата следва да се погрижи за възстановяване на вярната последователност. За целта, при кодирането на всеки кадър се присвоява номер (по модул 1024).

Какво представлява предсказването с помощта на *P-picture*? Както по-горе споменахме, при видеоизображенията често се случва два поредни кадъра да се различават главно по отместването си един спрямо друг. Идеята на *P-picture* е да се намери съответен вектор, задаващ отместването. Стойностите на координатите на вектора могат да варират в границите $-64, -63, \ldots, 63$.

За съжаление, изображението в следващия кадър би могло не само да се измести, но и да се завърти на някакъв ъгъл спрямо това от предходния кадър. В този случай се налага изчисляването на специална грешка при отместването, която следва да се прибави допълнително за конкретния пиксел. За целта се създава матрица на грешките. При декодирането на *P-picture* първо се извършва транслацията с определения при кодирането вектор, след което за всеки пиксел се прибавя съответната грешка от матрицата. Ще отбележим, че поддържането на матрицата на грешките е по-ефективно от запазването на цялото изображение, тъй като голяма част от елементите ѝ са нули и могат да се пропуснат при предаването (например чрез *кодиране на последователности*, *виж 10.3.9.*). Освен това матрицата на грешките се кодира с дискретна косинусова трансформация, разгледана по-горе (*виж 10.7.2.*).

При извършване на предвиждане *MPEG* разделя кадъра на макроблокове с размери 16×16. За всеки макроблок се поддържа отделен вектор на транслация и отделна матрица на грешките. В случай, че кодирането му се окаже неефективно, той може да се подаде на изхода без изменение, т. е. в рамките на *P-picture* можем да имаме некодирани макроблокове. Всеки макроблок се разделя на 6 блока с размери 8×8: 4 отговарят за яркостта и 2 — за цвета. За всеки пиксел се поддържа съответна стойност на яркостта. Що се отнася до цвета — тук нещата стоят по-особено. Благодарение на някои особености на човешкото око се оказва, че не е необходимо да се предават цветови данни за всеки пиксел. Вместо това пикселите се групират по четворки и на всяка четворка се съпоставя единствен цвят (но с различна яркост, съхранена в другите 4 блока).

Цветът се разделя на две части, които изграждат двата блока 8×8 за цвят. Независимо от това дали макроблокът е бил кодиран или не, следва прилагане на дискретна косинусова трансформация.

Това са само част от прилаганите в *MPEG* техники. Става въпрос за мощен метод, съчетаващ в себе си възможността за компресиране и декомпресиране в *реално време*, както при *нормална скорост*, така и на *бързи "обороти"*; за *пряк достъп* до определен кадър или група кадри (това дава възможност за редактиране на изображението); възможност за декомпресиране както в *права*, така и в *обратна посока* и др.

Задачи за упражнение:

1. Да се сравнят *JPEG* и *MPEG*.
2. Какви идеи от *FLZ* и модела на Марков могат да бъдат открити в *MPEG*?

10.7.4. Уейвлети

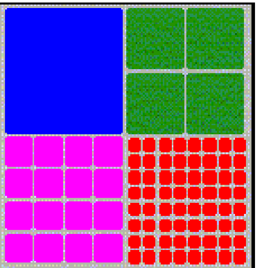
Идеята на уейвлетите е видеоизображението да се раздели на 42 честотни ленти (2×14 за цвят и 1×14 за яркост). Следва интелигентно квантифициране, отрязващо високите честоти, невидими за човешкото око. В резултат се получава изображение с изключително висока степен на компресия, като дори при порядъка от 300:1 няма забележима загуба на качество (*виж фигура 10.7.4а.*). Компресирането с уейвлети не използва техники, подобни на *MPEG*, за кодиране на групи от последователни кадри. Всеки кадър се компресира сам за себе си и по никакъв начин не зависи от останалите. Макар потенциално този подход да дава по-лоши резултати, той има своите несъмнени предимства. От една страна е налице директен достъп до всеки кадър, което силно улеснява редактирането, а от друга — по-висока надеждност. Действително, повреждането на *единствен* кадър по никакъв начин няма да повлияе на останалите кадри.

Фигура 10.7.4а. Компресиране на графично изображение с *уейвлети*: 1:1, 1:20 и 1:50.

Идеята на уейвлетите е сложна и изисква познания по линейна алгебра и по-специално по теория на *линейните пространства*. Препоръчваме на незапознатия с теорията читател да прескочи следващите редове.

Ще скицираме съвсем повърхностно основната идея. Нека V е пространството на изображението. Ще го разбием на подпространства V_i, такива че $V_i \subset V_{i+1}$ и $\bigcup_i V_i = V$. V_i са пространства с прогресивно нарастваща разделителна способност. Колкото по-голям е индексът i, толкова по-плътно се приближаваме до изображението. Да означим с W_i ортогоналното допълнение на V_i до V. Пространството V_{i+1} се представя като ортогонална сума на V_i и W_i. Целта е да намерим базиси ψ и ϕ на V_i и W_i съответно. Всички базисни функции за различните пространства се получават от общ уейвлет-баща чрез транслация и дилатация: $\psi_{ab}(x) = \psi(ax+b)$. С помощта на транслацията се постига отчитане на сигнала в различни точки, а с помощта на дилатацията — в различни скали. Получава се клас ортогонални функции, за които $<\psi_{ab}(x), \psi_{cd}(x)> = \delta_{ac}\delta_{db}$. Тук $\delta_{xy} = 1$ тогава и само тогава, когато $x = y$. Най-популярният пример са функциите на Хаар (*фигура 10.7.4б.* показва случая за стандартния интервал [–1,1]). Те са ортогонални при $a = 2^j$ и $b = k$. В резултат на преобразуването в горния ляв ъгъл на квадрата се получават малко, но важни коефициенти. Обратно — в долния десен ъгъл получаваме повече, но по-маловажни коефициенти, повечето от които са нули. Оказва се възможно да се премахнат част

от тях, без това да се отрази съществено на качеството на изображението след обратно преобразуване.



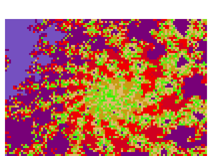
Фигура 10.7.4б. Преобразуване на Хаар.

Уейвлетите си пробиват все по-уверено път не само в мултимедийния, но и в научния свят: преди всичко в областта на ядрената физика, неврологията, музиката, оптиката, радарната техника, астрономията, акустиката и др. Уейвлети се използват в системите за изследване и ранно предупреждение при земетресения, при съхраняване на отпечатъците от пръсти във ФБР и др. Комитетът *JPEG*-2000 е обявил компресирането с уейвлети за компресиращата техника на XXI век.

Задача за упражнение:

Да се сравни компресирането в уейвлети с алгоритмите на *JPEG* и *MPEG*.

10.7.5. Компресиране с фрактали



Компресирането с фрактали се използва при компресиране на графично изображение. Основната му идея е да се намерят подходящи математически функции, които го описват. За целта се използват така наречените *итеративни функционални системи*. Математическите основи на метода са разработени от Барнсли. Идеята е да се търсят части от изображението, които могат да се получат от други части с помощта на различни афинни преобразувания (ротация, транслация, симетрия, хомотетия и др.), което е свързано с огромни математически пресмятания. Понякога се използват готови шаблони, избирани от човека по време на компресиране и позволяващи значително подобряване на качеството на изображението при декомпресиране при една и съща степен на компресия. Компресиране в рамките на 20:1 до 50:1 гарантира високо качество на декомпресираното изображение без забележими за човешкото око загуби. Компресиране в рамките на 50:1 до 90:1 дава добро качество. При степен на компресия 100:1 или повече качеството се влошава видимо, поради което се препоръчва главно за предварителен изглед от изображението (англ. *preview*), но не и за съхранение на оригинала.

Голяма част от изображенията, които се компресират, произлизат от реалния живот и имат ясно изразена фрактална структура, поради което компресирането с фрактали се оказва изключително ефективно. Впрочем, то може да се използва не само при компресиране на статични графични изображения, а и на видеообрази по начина, по който *JPEG* е интегриран в *MPEG* (*виж 10.7.2.* и *10.7.3.*). Подобно на *JPEG* може да се задава различна степен на компресия, разбира се, за сметка на качеството на декомпресираното изображение. Тук съществува още една интересна възможност: постигане на по-висока степен на компресия за сметка на необходимо *време* за работа на компресиращия алгоритъм, при същото качество на декомпресираното изображение. Това прави метода идеален при създаване на архиви. Още повече, че е силно несиметричен — декомпресирането се извършва значително по-бързо от компресирането, при това многократно по-бързо от декомпресирането при *JPEG*. Докато при изображение, кодирано

по *JPEG*, размерът на компресирания файл нараства чисто линейно при нарастване на размера на входното изображение, при компресирането с фрактали нарастването обикновено е по-бавно. Това означава, че методът е особено ефективен при големи по размер изображения, каквито са съвременните висококачествени 24-битови (и дори повече) цветни изображения.

Разбира се, всяко нещо си има цена: "безплатен обяд" отново няма. Наред с изброените по-горе предимства, описаният метод има и редица слаби страни. На първо място софтуерното компресиране отнема много дълго време, поради което се налага използването на специален хардуер. Освен това, методът не е стандартизиран. Нещо повече: за момента алгоритмите са строго защитени като търговски продукт и не са достъпни за свободно използване.

Задачи за упражнение:

1. Да се сравни компресирането в уейвлети с компресирането с фрактали.
2. Защо декомпресирането е значително по-бързо от компресирането?

10.8. Въпроси и задачи

10.8.1. Задачи от текста

Задача 10.1.

Да се определи максималната степен на компресия, която може да се постигне с премахване на нулите (*виж 10.3.1.*).

Задача 10.2.

Да се предложи начин за кодиране на символа *escape* — макар и рядък, все пак може да се срещне (*виж 10.3.1.*).

Задача 10.3.

Да се предложи начин за кодиране на символите от кодиращата двойка *SI* и *SO* (*виж 10.3.1.*).

Задача 10.4.

Да се реализира алгоритъмт от *10.3.1.* и да се изпробва върху битова карта на черно-бяло изображение.

Задача 10.5.

Да се реализира алгоритъмът от *10.3.2.*

Задача 10.6.

За какъв тип файлове е подходящо компресирането с битови карти (*виж 10.3.2.*)?

Задача 10.7.

Следва ли да се очаква намаляване на дължината на стандартен текстов файл (приемаме интервала за 0) при компресиране с битови карти (*виж 10.3.2.*)? А на битова карта на черно-бяло изображение?

Задача 10.8.

Възможно ли е да се използва успешно методът на компресиране с битови карти (*виж 10.3.2.*), ако някои части от файла съдържат голям процент нули, но, погледнато глобално, честотата им е недостатъчна?

Задача 10.9.

Да се сравнят компресирането с битови карти (*виж 10.3.2.*) и премахването на нулите (*виж 10.3.1.*). Кое от тях е за предпочитане? Защо?

Задача 10.10.

Да се предложи решение на проблема с потенциалната загуба на 156 от 256-те възможни стойности в случай на еднобайтов брояч (*виж 10.3.3.*).

Задача 10.11.

Да се реализира алгоритъмът от *10.3.3.* и да се сравни с премахването на нулите (*виж 10.3.1.*) и компресирането с битови карти (*виж 10.3.2.*) върху различни типове файлове.

Задача 10.12.

Да се реализира алгоритъмът от *10.3.4.*

Задача 10.13.

Да се пресметне степента на компресия на алгоритъма от *10.3.4.*

Задача 10.14.

Да се предложи вариант на алгоритъма от *10.3.4.* в случай, че всичките 8 бита се ползват.

Задача 10.15.

Да се построи честотна таблица за най-често срещаните двойки символи за български език по подобие на *таблица 10.3.5.*

Задача 10.16.

Да се намерят най-често срещаните двойки букви — кандидати за супербуква за различни видове файлове (*виж 10.3.5.*):

- стандартен текстов файл на английски език;
- стандартен текстов файл на български език;
- програма на *Си*, *Паскал*, *Бейсик*, *Фортран*, *Java*.

Задача 10.17.

Да се оцени максималната и очакваната степен на компресия (*виж 10.3.5.*).

Задача 10.18.

Защо срещанията на специалния символ трябва да се удвоят преди да се пристъпи към кодиране (*виж 10.3.5.*)?

Задача 10.19.

Да се предложи решение на проблема, когато две двойки претендират за една и съща буква (*виж 10.3.5.*).

Задача 10.20.

Да се реализира алгоритъмът от *10.3.5.*

Задача 10.21.

Следва ли да се очаква подобрение, ако се разглеждат последователности от 3 (или повече) символа (*виж 10.3.5.*)? Какви проблеми могат да възникнат в такъв случай?

Задача 10.22.

Да се предложи стратегия за намиране на "най-добрата" таблица при зададен текст за кодиране (*виж 10.3.6.*). Да се вземе предвид, че ще се предава и речникът.

Задача 10.23.

Необходимо ли е да се кодират всички запазени думи на даден език за програмиране, ако целта е да се състави универсален речник за кодиране на произволна програма (*виж 10.3.6.*)?

Задача 10.24.

Да се реализира предложеният в *10.3.6.* метод за компресиране.

Задача 10.25.

Да се сравнят следните две стратегии за запазване на последователните версии на текстов документ (*виж 10.3.7.*):

а) спрямо предходната редакция;
б) спрямо първоначалния вариант.

Задача 10.26.

Да се предложи начин за определяне и ефективно запазване на разликите между два текстови документа (*виж 10.3.7.*).

Задача 10.27.

Друг възможен вариант на алгоритъма от *10.3.8.* е да се изчете цялото съобщение и да се намери ентропията му, след което е използвано в процеса на кодиране, вместо частичните ентропии от оригиналния алгоритъм. Какви са предимствата и недостатъците на такъв подход?

Задача 10.28.

Ключова характеристика на кодирането с линейно предсказване (виж *10.3.8.*) е неговата адаптивност. Способността за адаптация обаче намалява прогресивно заедно с броя на обработените символи и математическото очакване се изменя все по-трудно. Проблемът е особено изразен в края на дълго съобщение. Какви възможни решения съществуват?

Задача 10.29.

Да се сравни алгоритъмът от *10.3.8.* с относителното кодиране от *10.3.7.*

Задача 10.30.

Нека при реализацията на кодиране на последователности (*виж 10.3.9.*) Z се използва като *escape* символ, възможно ли в кодираното съобщение да се срещне "ZZ"? А "ZZZ"? А "ZZZZ"?

Задача 10.31.

Да се компресира по метода на кодиране на последователности (*виж 10.3.9.*) съобщението: "*ZAZZZZZSDHDGSDGHGZGZGZHJGG*", *Z* е *escape* символ.

Задача 10.32.

Да се определи експериментално степента на компресия, постигана при замяна на последователност от *k* интервала със знака за табулация (*виж 10.3.9.*). Каква е оптималната стойност на *k*?

Задача 10.33.

Да се сравнят кодирането на последователности (*виж 10.3.9.*) и премахването на нулите (*виж 10.3.1.*).

Задача 10.34.

При какъв тип файлове следва се очаква, че кодирането на последователности (*виж 10.3.9.*) ще бъде ефективно?

Задача 10.35.

Да се сравнят *PackBits* (*виж 10.3.10.*) и кодирането на последователности (*виж 10.3.9.*).

Задача 10.36.

Да се сравнят *PackBits* (*виж 10.3.10.*) и алгоритъмът от *10.3.4.*

Задача 10.37.

Да се намери оптималният код за *фигура 10.4.1.*

Задача 10.38.

Да се дадат още примери за неравномерен разделим код (*виж 10.4.1.*).

Задача 10.39.

Какъв е броят на върховете в дървото на Шенън-Фано (*виж 10.4.1.*)?

Задача 10.40.

Каква е максималната височина на дървото на Шенън-Фано (*виж 10.4.1.*), изразена като функция на броя на буквите от входната азбука? Балансирано ли е?

Задача 10.41.

Да се докаже, че кодът, построяван от алгоритъма на Шенън-Фано, е префиксен (*виж 10.4.1.*).

Задача 10.42.

Кога алгоритъмът на Шенън-Фано (*виж 10.4.1.*) ще строи гарантирано оптимален код?

Задача 10.43.

Да се определи сложността на алгоритъма на Шенън-Фано (*виж 10.4.1.*).

Задача 10.44.

Възможно ли е алгоритъмът на Шенън-Фано да се модифицира така, че винаги да строи *оптимален* побуквен код (*виж 10.4.1.*)?

Задача 10.45.

Да се построи кодът на Хъфман (*виж 10.4.2.*) за следните вероятности:

	a	*b*	*c*	*d*	*e*	*f*
а)	0,70	0,10	0,10	0,05	0,03	0,02
б)	0,10	0,20	0,20	0,10	0,20	0,20
в)	0,30	0,30	0,20	0,15	0,05	0,00
г)	0,17	0,17	0,17	0,17	0,16	0,16

Да се сравни със съответния код на Шенън-Фано (*виж 10.4.1.*). Каква е дължината на кодираното съобщение?

Задача 10.46.

Каква е максималната височина на дървото на Хъфман (*виж 10.4.2.*)? Балансирано ли е?

Задача 10.47.

Да се докаже, че кодът, построяван от алгоритъма на Хъфман (*виж 10.4.2.*), е префиксен.

Задача 10.48.

Да се сравнят алгоритмите на Хъфман (*виж 10.4.2.*) и на Шенън-Фано (*виж 10.4.1.*).

Задача 10.49.

Да се модифицират предложените в *10.4.2.* програмни реализации така, че да намират всички кодове на Хъфман за дадения източник.

Задача 10.50.

Да се реализира кодиране на Хъфман без използване на пирамида и да се сравни по скорост с предложената в *10.4.2.* реализация.

Задача 10.51.

Да се определи сложността на последната реализация от *10.4.2.* по време и памет.

Задача 10.52.

Нека всички честоти на срещане на буквите за дадено входно съобщение са различни. Може ли да се твърди, че съответното дърво на Хъфман е единствено (*виж 10.4.2.*)?

Задача 10.53.

Какво ще се получи, ако се разбие кодирано по Хъфман съобщение на последователности от по 5 бита, след което отново се приложи Хъфманово кодиране (*виж 10.4.2.*)?

Задача 10.54.

Да се напише декодираща програма за Хъфмановото компресиране (*виж 10.4.2.*).

Задача 10.55.

Защо в случай на нужда от двойно разклонение, е за предпочитане да бъде възможно най-близо до листата (*виж 10.4.3.*)?

Задача 10.56.

Какви проблеми възникват при използване на троичен Хъфманов код (*виж 10.4.3.*) и как биха могли да се решат?

Задача 10.57.

Следва ли да се очаква подобрение при използване на троичен Хъфманов код (*виж 10.4.3.*) вместо двоичен? Защо? А при четвъртичен?

Задача 10.58.

Префиксен ли е кодът с разделители (*виж 10.4.4.*)?

Задача 10.59.

Да се реализира програма за кодиране с код с разделители (*виж 10.4.4.*).

Задача 10.60.

Да се сравнят кодът с разделители (*виж 10.4.4.*) и кодът на Хъфман (*виж 10.4.2.*).

Задача 10.61.

Да се сравни аритметичното кодиране (*виж 10.4.5.*)с кодирането на Хъфман (*виж 10.4.2.*). Винаги ли аритметичното кодиране ще постига по-добра степен на компресия? Защо?

Задача 10.62.

Да се реализира работоспособен вариант на аритметичното кодиране (*виж 10.4.4.*).

Задача 10.63.

Да се предложи начин за намаляване на броя на размените при обновяването на таблицата при адаптивно кодиране (*виж 10.5.*).

Задача 10.64.

Да се реализира алгоритъмът от *10.5.*

Задача 10.65.

Да се сравни адаптивното компресиране (*виж 10.5.*) със стандартното компресиране по Хъфман (*виж 10.4.2.*). Наблюдава ли се подобрение?

Задача 10.66.

Колко често при адаптивно кодиране (*виж 10.5.*) следва да се извършва деление на две така, че от една страна да се дава възможност за натрупване на достатъчна статистика, а от друга — да се отчитат локалните особености?

Задача 10.67.

Да се реализира алгоритъмът от *10.5.1.*

Задача 10.68.

Да се сравни адаптивното компресиране с използване на код на Хъфман от *10.5.* с адаптивното компресиране по Хъфман, предложено в *10.5.1.* Кое отчита по-добре локалните особености?

Задача 10.69.

Да се оценят предимствата и недостатъците на използването на *escape* символ при адаптивното компресиране по Хъфман (*виж 10.5.1.*).

Задача 10.70.

Да се реализира алгоритъмът от *10.5.2.*

Задача 10.71.

Какъв друг проблем освен с паметта възниква при преминаване към модел от по-висок ред (*виж 10.5.2.*)?

Задача 10.72.

Да се посочат предимствата и недостатъците на *постепенното* преминаване към модел от по-висок ред (едва след натрупване на достатъчно статистики), *виж 10.5.2.*

Задача 10.73.

Да се анализира възможността за ползване на ляв и десен контекст едновременно при модел от по-висок ред (*виж 10.5.2.*).

Задача 10.74.

Да се обсъдят предимствата и недостатъците на *MNP*-5 (*виж 10.5.3.*). Как би могъл да се подобри?

Задача 10.75.

Да се обясни защо се строи таблица *10.5.3.* Защо не се използва направо код на Хъфман?

Задача 10.76.

Да се обсъдят предимствата и недостатъците на използване на речник с *фиксирана* и с *променлива* дължина на думите (*виж 10.6.*).

Задача 10.77.

Да се обсъдят предимствата и недостатъците на използване на *фиксиран* срещу *динамичен* (изграждан на основата на текущия текст) речник (*виж 10.6.*).

Задача 10.78.

Следва ли да се очаква подобрение на речниковия метод (*виж 10.6.*) спрямо кодирането по Хъфман (*виж 10.4.2.*) и аритметичното кодиране (*виж 10.4.5.*)?

Задача 10.79.

Защо средният брой необходими битове за кодиране на една буква намалява с увеличаване на дължината на използваната супербуква (*виж 10.6.1.*)? Винаги ли ще бъде така?

Задача 10.80.

Какви проблеми възникват при по-дълги супербукви (*виж 10.6.1.*)?

Задача 10.81.

Да се докажат твърдения 1 и 2 от *10.6.1.*, като се изходи от дефиницията на понятието ентропия.

Задача 10.82.

Кога и защо би могъл да се получи код, който липсва в речника (*виж 10.6.4.*)?

Задача 10.83.

Да се реализира *LZ77* (*виж 10.6.5.*).

Задача 10.84.

Защо при *LZ77* (*виж 10.6.5.*) на изхода се подава c, т.е. защо се извежда тройка (p,l,c), а не двойката (p,l)?

Задача 10.85.

Подаването на (0,0,*c*) при *LZ77* (*виж 10.6.5.*) на изхода при липса на съвпадение е очевидно неразумно, още повече че ще се случва често. Да се предложи по-икономично решение.

Задача 10.86.

Да се сравнят *LZSS* (*виж 10.6.6.*) и *LZ77* (*виж 10.6.5.*).

Задача 10.87.

Защо се очаква дървото на *LZSS* да бъде "добре балансирано" (*виж 10.6.6.*)?

Задача 10.88.

Да се сравнят *FLZ* (*виж 10.6.7.*) и *LZ77* (*виж 10.6.5.*).

Задача 10.89.

Коя от двете идеи на *FLZ* (*виж 10.6.7.*) е по-перспективна: "мързеливо" компресиране или поглед в двете посоки?

Задача 10.90.

Да се анализира взаимодействието на трите нови идеи на *FLZ* (*виж 10.6.7.*).

Задача 10.91.

Следва ли да се очаква подобрение, ако се остави речникът на *LZW* (*виж 10.6.8.*) да расте неограничено? Какви проблеми възникват и как могат да се разрешат?

Задача 10.92.

Защо на изхода се подава кодът на β, а не на α (виж началото на *10.6.8.*)?

Задача 10.93.

Да се оцени степента на компресия за примера от *таблица 10.6.8.* при положение, че се предава речникът.

Задача 10.94.

Да се докаже, че описаният в *10.6.8.* случай е единственият, когато при декодиране на *LZW* може да се получи код, за който няма дума в речника.

Задача 10.95.

Каква е сложността на *LZW*? А на предложената в *10.6.8.* реализация?

Задача 10.96.

Да се сравнят *LZW* (*виж 10.6.8.*) и *LZ77* (*виж 10.6.5.*).

Задача 10.97.

Да се предложи и реализира адаптивен вариант на *LZW* (*виж 10.6.8.*).

Задача 10.98.

Да се предложи начин за комбиниране на *LZW* (*виж 10.6.8.*) и кодиране по Хъфман (*виж 10.4.2.*).

Задача 10.99.

Какви други подобрения биха могли да се направят в *GIF* (*виж 10.6.9.*)?

Задача 10.100.

Да се сравнят алгоритмите на *GIF* (*виж 10.6.9.*) и *LZW* (*виж 10.6.8.*).

Задача 10.101.

Решава ли мързеливото кодиране на *FLZ* (*виж 10.6.7.*) описания в *10.6.10.* проблем?.

Задача 10.102.

Каква е алгоритмичната сложност на потенциален алгоритъм за намиране на оптималната кодираща схема (*виж 10.6.10.*)?

Задача 10.103.

Нека при кодиране с *LZW* (*виж 10.6.8.*) в даден момент речникът съдържа думите "*ПРО*", "*_ПР*", "*РОСТ*" и "*ПРИМЕР*". Кои други думи задължително също трябва да са там?

Задача 10.104.

Кои други разгледани досега алгоритми са подходящи за компресиране в реално време на ниво сектор (*виж 10.6.11.*)?

Задача 10.105.

Да се предложи начин за компресиране в реално време, различен от компресирането на ниво сектор (*виж 10.6.11.*).

Задача 10.106.

Да се обясни по-подробно връзката на *LZW* с аритметичното кодиране (*виж 10.6.12.*).

Задача 10.107.

Да се предложи подходяща комбинация на *LZW* и модел на Марков от по-висок ред (*виж 10.6.12.*).

Задача 10.108.

Кои от проблемите на *LZW* по отношение на моделите на Марков от по-висок ред се отнасят и за *LZ77* (*виж 10.6.12.*)?

Задача 10.109.

Да се обясни как квантифицирането (*виж 10.7.1.*) може да помогне за постигане на по-добра компресия и да се даде конкретен пример.

Задача 10.110.

Да се сравнят алгоритмите на *GIF* (*виж 10.6.9.*) и *JPEG* (*виж 10.7.2.*). Кога е за предпочитане да се използва единият и кога — другият?

Задача 10.111.

Да се обясни ползата от всеки от алгоритмите в *JPEG* и да се обоснове последователността им (*виж 10.7.2.*).

Задача 10.112.

Да се сравнят *JPEG* (*виж 10.7.2.*) и *MPEG* (*виж 10.7.3.*).

Задача 10.113.

Какви идеи от *FLZ* (*виж 10.6.7.*) и модела на Марков (*виж 10.5.2.*) могат да бъдат открити в *MPEG* (*виж 10.7.3.*)?

Задача 10.114.

Да се сравни компресирането в уейвлети (*виж 10.7.4.*) с алгоритмите на *JPEG* (*виж 10.7.2.*) и *MPEG* (*виж 10.7.3.*).

Задача 10.115.

Да се сравни компресирането в уейвлети (*виж 10.7.4.*) с компресирането с фрактали (*виж 10.7.5.*).

Задача 10.116.

Защо при фракталите декомпресирането е значително по-бързо от компресирането (*виж 10.7.5.*)?

10.8.2. Други задачи

Задача 10.117. *Тестване на комерсиални програми*

Да се извършат експерименти с различни универсални компресиращи програми (*WinZip*, *ARJ*, *AIN*, *RAR* и др.) и да се направят съответни изводи за скоростта и степента на компресиране на използваните от тях алгоритми.

Задача 10.118. *Реализиране на програми*

Да се напишат компресираща и декомпресираща програма за:

- премахване на нулите (*виж 10.3.1.*);
- компресиране с битови карти (*виж 10.3.2.*);
- полубайтово пакетиране на числови данни (*виж 10.3.3.*);
- двуатомно кодиране (*виж 10.3.5.*);
- замяна на шаблони (*виж 10.3.6.*);
- относително кодиране (*виж 10.3.7.*);
- кодиране на последователности от еднакви символи (*виж 10.3.9.*), като за *escape* символ се използва *Z*;

Задача 10.119. *Тестване на програми*

Да се експериментира с програмите от предходната задача и да се сравни получената степен на компресия с теоретичната, както и между различните програми.

Задача 10.120. *Най-подходящ тип файлове*

За всеки тип компресиране от *задача 10.118.* да се определи най-подходящият тип файлове.

Задача 10.121. *Комбиниране на алгоритми*

Да се експериментира с различни комбинации на алгоритми от настоящата глава по подобие на начина, по който това е направено в *10.3.4.*, *10.5.3.*, *10.7.2.* и *10.7.3.*

Задача 10.122. *Сравнение*

Да се сравни кодирането с линейно предсказване (*виж 10.3.8.*) със статистическите (*виж 10.4.*) и адаптивните (*виж 10.5.*) методи на компресиране.

Задача 10.123. *Кодиране по Хъфман*

Дадена е азбуката A = {а(51), б(18), в(53), г(10), д(55), е(3), ж(50), з(29), и(20), к(52), л(54), м(500)}. В скоби след всяка буква са дадени честотите на срещането ѝ в определен текст.

а) Да се построят дървото и кодът на Хъфман за този текст;

б) Нека с $|\alpha|$ означим дължината на думата $\alpha \in A^*$, а с $[\beta]$ — нейния Хъфманов код, построен в а). Да се намери множеството $B = \{\alpha \mid \alpha \in A^*, |\alpha| = \max \{|\beta| \mid \beta \in A^*, [\beta] \in P\}\}$, където P е множеството от двоични низове, в които броят на единиците е равен на броя на нулите и е равен на 60 (*виж* [*Шишков-1995*]).

Задача 10.124. *Варианти на кодиране по Хъфман*

Да се сравнят предимствата и недостатъците на следните варианти на кодиране по Хъфман при компресиране на различни по дължина и тип файлове:

- Статичен модел с едни и същи вероятности за всеки текст;
- Статичен модел с едни и същи вероятности за всеки текст от даден тип: Например един модел за текст на български език, друг — за програма на *Си*;
- Статичен модел, построен съгласно честотата на срещане на буквите от входното съобщение;
- Адаптивен модел.

Задача 10.125. *Речниково кодиране*

Да се сравни ефективността на различните *речникови* методи на компресиране, разгледани в настоящата глава.

Задача 10.126. *Повреждане на единствен бит*

Кои от разгледаните досега алгоритми са силно чувствителни към повреждане на единствен бит?

Задача 10.127. *Оптимално речниково кодиране при фиксиран речник*

Да се разработи алгоритъм, получаващ като вход съобщение за кодиране, речник, дължина на индекса в речника, брой битове, необходими за кодиране на буква, непринадлежаща на речника, и даващ като резултат оптимална кодираща последователност от букви и индекси в речника, т. е. такава, за която общият брой битове в кодираното съобщение да бъде минимален (*виж 10.6.10.*).

Литература

[*Аладжем-1992*] Аладжем М., "Приложно програмиране с Турбо Паскал", "Техника", София, 1992.

[*Банчев-1993*] Банчев Б., "Ханойските кули и превъплъщенията на рекурсията", сп. "Математика плюс", 1/1993, София, 1993.

[*Бонев-1996*] Бонев Ст., "Итерация или рекурсия — предимства и недостатъци", сп. "PC&MAC World", 4/1996, София, 1996.

[*Брудно,Каплан-1980*] Брудно А., Л. Каплан, "Олимпиады по программированию для школьников", "Наука", Москва, 1980.

[*Болобаш-1989*] Болобаш Б., "Теория на графите", "Наука и изкуство", София, 1989.

[*Денев-1984*] Денев Й., "Дискретна математика", "Наука и изкуство", София, 1984.

[*Дилова, Стоянов-1973*] Дилова В, М. Стоянов, "Висша математика", I част, "Техника", София,1973.

[*Димитров,Сълев-1990*] Димитров В., К. Сълев, "За колко време се изпълнява една програма", сп. "Обучението по математика и информатика", 4/1990, София, 1990.

[*Калихман-1979*] Калихман И., М. Войтенко, "Динамическое программирование в примерах и задачах", "Высшая школа", Москва, 1979.

[*Келеведжиев-1993*] Келеведжиев Е., "Намиране на маршрути", сп. "Математика+", 1/1993, София, 1993.

[*Келеведжиев-1994*] Келеведжиев Е., "Пресмятане на числото пи", сп. "Математика +", 3/1994, София, 1994.

[*Келеведжиев-1998*] Келеведжиев Е., "Маршрути на Ойлер и Хамилтън", сп. "Computer", 10/1998, София, 1998.

[*Келеведжиев-5/1998*] Келеведжиев Е., "Задочен конкурс по информатика, задача 5/1998", сп. "Computer", 1998.

[*Келеведжиев-2000*] Келеведжиев Е., "Динамично оптимиране", "Анубис", София, 2000.

[*Корн, Корн-1977*] Корн Г., Т. Корн, "Справочник по математике", "Наука", Москва, 1977.

[*Липский-1980*] Липский В., "Комбинаторика для программистов", "Мир", Москва, 1980.

[*Май-1989*] Май К. *Винету,* том 1, "Отечество", София, 1989.

[*Манев-1996*] Манев К., "Увод в дискретната математика", "Издателство на Нов български университет ", София, 1996.

[*Наков-1998*] Наков П., "Основи на компютърните алгоритми", "TopTeam Co", София, 1998.

[*Наков-1998a*] Наков П., "Сливане на сортирани масиви", сп. "Computer", 8/1998, София, 1998.

[*Наков-1998b*] Наков П., "Максимална сума на подредица на дадена числова редица", сп. "Computer", 1/1998, София, 1998.

[*Наков-1998c*] Наков П., "Числа на Фибоначи", сп. "PC Magazine/България", 12/1998, София, 1998.

[*Наков-1998d*] Наков П., "Циклично изместване елементите на масив", сп. "Computer", 10/1998, София, 1998.

[*Наков-1998e*] Наков П., "Сливане на сортирани масиви", сп. "Computer", 8/1998, София, 1998.

[*Наков-1998f*] Наков П., "Намиране всевъзможните суми на дадено множество от монети. Проверка дали може да бъде получена дадена сума", сп. "Byte/България", 7-8/1998, София, 1998.

[*Наков-1998g*] Наков П., "Пресмятане стойността на аритметичен израз чрез преобразуване в обратен полски запис", в-к "ComputerNews", 27/1998, София, 1998.

[*Наков-1999a*] Наков П., "Динамично оптимиране, сп. "PC Magazine/България", 11/1999, София, 1999.

[*Наков-1999b*] Наков П., "По-сложни методи на търсене", сп. "PC Magazine/България", 10/1999, София, 1999.

[*Наков-1999c*] Наков П., "Статистическо побуквено кодиране на данни", сп. "Byte/България",
9-10/1999, София, 1999.

[*Наков-1999d*] Наков П., "Кодиране на информацията", сп. "Byte/България", 9-10/1999, София, 1999.

[*Наков-1999f*] Наков П., "Намиране на *k*-ия по големина елемент", сп. "PC Magazine/България", 3/1999, София, 1999.

[*Наков-1999g*] Наков П., "Бързо сортиране", PC Magazine/България, 1/1999, София, 1999.

[*Наков-1999h*] Наков П., "Максимална сума на подредица на дадена числова редица", сп. "Computer" 1/1999, София, 1999.

[*Наков-1999i*] Наков П., "Пирамида, пирамидално сортиране и приоритетна опашка", сп. "Byte/България", 1/1999, София, 1999.

[*Наков-1999j*] Наков П., "Циклично изместване елементите на масив", сп. "Computer", 1/1999, София, 1999.

[*Наков-1999k*] Наков П., "Бързо сортиране", сп. "PC Magazine/България", 1/1999, София, 1999.

[*Наков-1999l*] Наков П., "Намиране на *k*-ия по големина елемент", сп. "PC Magazine/България", 3/1999, София, 1999.

[*Наков-1999k*] Наков П., "Динамично оптимиране. Задача за разделянето", сп. "Byte/България", 11/1999, София, 1999.

[*Наков-2000a*] Наков П., "Най-дълга обща подредица", сп. "Byte/България", 3/2000, София, 2000.

[*Наков-2000b*] Наков П., "Задача за раницата", сп. "PC Magazine/България", 2/2000, София, 2000.

[*Наков-2001a*] Наков П., "Алгоритъм на Щрасен за бързо умножение на матрици", сп. "Обучението по математика и информатика", София, 2001.

[*Наков-2001b*] Наков П., "Оптимално двоично дърво за търсене", сп. "Byte/България", София, 2001.

[*Наков-2001c*] Наков П., "Триангулация на многоъгълник. Числа на Каталан", сп. "Byte/България", София, 2001.

[*Наков-2001d*] Наков П., "Бързо умножение на матрици", сп. "Byte/България", София, 2001.

[*Наков-2001e*] Наков П., "Мажорант", сп. "Computer", София, 2001.

[*Наков-2001f*] Наков П., "Разделяй и владей", сп. "PC Magazine/България", София, 2001.

[*Наков-2001g*] Наков П., "Паралелни алгоритми за сортиране", сп. "PC Magazine/България", София, 2001.

[*Огрнянов, Тотков-1989*] Огнянов Ф., Г. Тотков, "Рекурентни масиви", сп. "Математика", 2/1989, София, 1998.

[*Павлов-1988*] Павлов Ц., "Числа на Мерсен", сп. "Математика", 4/1988, София, 1988.

[*Раденски-1987*] Раденски А., "Компютър, език за програмиране, транслатор", "Наука и изкуство", София, 1987.

[*Раденски и др.-1989*] под ред. на А. Раденски, "Задачи по програмиране с решения на Паскал", университетско издателство "Климент Охридски", София, 1989.

[*Рахнев,Гъров-1988*] Рахнев А., К. Гъров, "Програмиране на рекурентни формули", сп. "Математика", 4/1988, София, 1988.

[*Рахнев, Гъров, Гаврилов-1995*] Рахнев А., К. Гъров, О. Гаврилов, "Бейсик в задачи", "Asio", София, 1995.

[*Рейнгольд, Нивергельт, Део-1980*] Рейнгольд Э., Ю. Нивергельт, Н. Део, "Комбинаторные алгоритмы. Теория и практика", "Мир", Москва, 1980.

[*Сидеров-1995*] Сидеров П., "Записки по алгебра. Групи, пръстени и полиноми", "Веди", София, 1995.

[*Симеон-II*] http://www.b-info.com/places/Bulgaria/Royal

[*Смит-1996*] Смит Т., "Програмирането с Pascal. Принципи и методи", "Техника", София 1996

[*Стенли-1998*] Стенли Т., "Компресиране на данни", "Интерфейс-България" ООД, София, 1998.

[*Терзиева-1993*] Терзиева Д., "Прост ли е алгоритъмът за търсене на прости числа?", сп. "Обучението по математика и информатика", 1/1993, София, 1993.

[*Тинчев-1987*] Тинчев Т., "Дървета", сп. "Математика", 1/1987, София, 1987.

[*Тодорова-1995*] Тодорова М., "Програмиране на Паскал", София, 1995.

[*Уирт-1980*] Уирт Н., "Алгоритми + структури от данни = програми", "Техника", София, 1980.

[*Швертнер-1995*] Швертнер Й., "Структури от данни и алгоритми", университетско издателство "Св. Климент Охридски", София, 1995.

[*Шишков-1995*] Шишков Д. и колектив, "Структури от данни", "Интеграл", Добрич, 1995.

[*Adams-1980*] Adams D. *Hitchhikers guide to the galaxy*. Harmony books. 1980.

[*Adámek-1991*] Adámek J. *Foundations of Coding*. A Wiley-Interscience Publications. 1991.

[*Aho, Hopcroft, Ullman-1974*] Aho A., J. Hopcroft, J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company. 1974.

[*Aho,Hopcroft,Ullman–1987*] Aho A., J. Hopcoft, J. Ullman. *Data Structures and Algorithms*. Addison-Wesley. 1987.

[*ANSIC*] *ISO/IEC 9899:1999 Programming languages-C*. http://www.ansi.org; http://www.iso.ch

[*ANSIC-2*] *Rationale for ANSI Systems*. http://www.lysator.liu.se/c/rat/title.html

[*Ayres-1962*] Ayres F. *Theory and Problems of Matrices*. Schaum. 1962.

[*Bentley-1986*] Bentley J. *Pearls of Programming*. Addison-Wesley. 1986. (руски превод: Бентли Д., "Жемчужины творчества программистов", "Радио и связь", Москва, 1990)

[*Bentley-1990*] Bentley J. *More Programming Pearls*. Addison-Wesley Publishing Company. 1990.

[*Blackstock-1993*] Blackstock S. *LZW and GIF explained*. http://www.lexitech.com/bobrich/steve.htm

[*Bollobas-1979*] Bollobas B. *Graph Theory, an introduction course*. Verlag. 1979.

[*Bondy, Murty-1976*] Bondy J., U. Murty. *Graph Theory with Applications*. Elsevier North-Holland. 1976.

[*Brassard,Bratley-1996*] Brassard G., P. Bratley. *Fundamentals of Algorithmics*. Prentice-Hall. 1996.

[*Brown-1972*] R. J. Brown. *Chromatic scheduling and the chromatic number problem*. Management Science, 19:451-463, 1972.

[*Channon-1993*] Channon C. *Claude Elwood Channon Collected Papers*. IEEE Press. 1993.

[*Chiba,Nishizeki,Abe,Ozawa -1985*] Chiba N., T. Nishizeki. *A Linear Algorithm for Embedding Planar Graphs Using PQ-Trees*. J. of Comp. and Sys. Sci.. vol. 30. pp. 54-76. 1985.

[*Christofides-1975*] Christofides N. *Graph Theory-an Algorithmic Approach*. Academic Press. 1975. (руско издание: Кристофидес Н., "Теория графов - алгорифмический подход", "Мир", Москва, 1978.)

[*ComputerNews-1994–1*] в-к ComputerNews, брой 5, София, 1994.

[*ComputerNews-1994–2*] в-к ComputerNews, брой 6, София, 1994.

[*Cormen, Leiserson, Rivest-1997*] Cormen T., C. Leiserson, R. Rivest. *Introduction to algorithms*. MIT Press. 1997.

[*Crochemore-1994*] Crochemore M., R. Wojcieh. *Text Algorithms*. Oxford University Press. Oxford. 1994.

[*Culberson-1993*] J. Culberson. *Iterated Greedy Graph Coloring and the Difficulty Landscape*. Tech. Rep. TR 92-07, Dept. Comp. Sci., Univ. Alberta, Edmonton, Alberta T6G 2H1. Canada. 1992.

[*Dor, Zwick-1996*] Dorit D., U. Zwick. *Median Selection Requires (2+epsilon)n Comparisons*. In Proc. IEEE Symposium on Foundations of Computer Science, pp. 125-134. 1996.

[*Dunne-1987*] Dunne P. *A result of k-valent graphs and its application to a graph embedding problem*. Acta Informatica, 24, pp.447-459. 1987.

[*Evans,Minieka-1998*] Evans J., E. Minieka. *Optimisation Algorithms for Networks and Graphs*. Marcel Dekker Inc. 1998.

[*Fibonacci-1*] http://www.ee.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibFormula.html

[*Flamig-1993*] Flamig B. *Practical Data Structures in C++*. Azarona Software. 1993.

[*France,Litman-1997*] Frances M., A. Litman. *On covering problems of codes*. Theory of Computing Systems, 30(2):pp.113-119, March/April. 1997.

[*Fredman,Tarjan-1987*] Fredman M., R. Tarjan. *Fibonacci heaps and their uses in improved network optimization algorithms*. Journal of ACM 34, 3(July), pp. 596-615. 1987.

[*Gabow*, *Galil*, *Spencer*, *Tarjan-1986*] Gabow H., Z. Galil, T. Spencer, R. Tarjan. *Efficient algorithsms for finding minimum spanning trees in undirected and directed graphs*. Combinatorica 6, pp.109-122. 1986.

[*Garey*, *Johnson-1979*] Garey M., D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman. 1979.

[*Garland-1998*] Garland T. *Fascinating Fibonaccis: Mystery and Magic in Numbers*. Dale Seymour Publications. 1998.

[*Gibbons,Rytter-1987*] Gibbons A., W. Rytter. *Efficient parallel algorithms*. Cambridge University Press. 1987.

[*Gregory, Rawlins - 1997*] Gregory J., E. Rawlins. *Compared to What? An Introdution to the Analysis of Algorithms*. Computer Science Press. 1997.

[*Guinier-1991*] Guinier D. *The Multiplication of Very Large Integers Using the Discrete Fast Fourier Transform*. ACM-SIGSAC Review, Special Group on Security Audit and Control, vol. 9, no. 3, pp. 26-36. 1991.

[*Held-1991*] Held G. *Data Compression*. John Wiley & Sons. 1991.

[*HLJJTT-1996*] Huss-Lederman S., E. Jacobson, J. Johnson, A. Tsao, T. Turnbull. *Implementation of Strassen's Algorithm for Matrix Multiplication*, preprint. 1996.

[*Hoffman-1997*] Hoffman R. *Data Compression in Digital Systems*. Chapman&Hall. 1997.

[*Hopcroft-Karp-1973*] Hopcroft J., R. Karp. *An algorithm for maximum matchings in bipartite graphs*. SIAM J. Computing, 2(4):pp.225-230. 1973.

[*Horowitz-1977*] Horowitz E., S. Sahni, *Fundamentals of Data Structures*. Pitman. 1977.

[*Infoman*] http://infoman.musala.com

[*Jewell-1976*] Jewell G. *Text Compaction for Information Retrieval Systems*. IEEE SMC Newsletter, 5, No 1. 1976.

[*Knuth-1/1968*] Knuth D. *The Art of Computer Programming. Volume I. Fundamental Algorithms*. MIT press. 1968. (руски превод: Кнут Д. "Искусство программирования для ЭВМ, том 1. Основные алгоритмы", Москва, "Мир", 1976.).

[*Knuth-2/1968*] Knuth D. *The Art of Computer Programming. Volume II. Seminumerical algorithms*. MIT press. 1968. (руски превод: Кнут Д., "Искусство программирования", том 2. "Сортировка и поиск", "Мир", Москва, 1978.).

[*Knuth-3/1968*] Knuth D. *The Art of Computer Programming. Volume III. Sorting and Searching*. MIT press. 1968. (руски превод: Кнут Д., "Искусство программирования", том 3. "Сортировка и поиск", "Мир", Москва, 1978.).

[*Knott-1*] http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fractions/egyptian.html

[*Korman-1979*] Korman S. *The graph-coloring problem*. In: N. Christophides, P. Toth, and C. Sandi, editors, Combinatorial Optimization, pp. 211-235. Wiley.1979.

[*Kučera-1998*] Kučera L., *Combinatorial Algorithms*. Adam Hilger. 1998.

[*Loviglio-1997*] Lociglio J., *Deep Blue team awarded 100,000$ Fredkin price*, http://www.rci.rutgers.edu/~cfs/472_html/Intro/NYT_Intro/ChessMatch/DeepBlueTeamAwarded.html

[*Magic-1*] http://user.chollian.net/~brainstm/four.html

[*Mathworld-a*] http://www.math.utu.fi/research/automata/decidres.html

[*Micali,Vazirani-1980*] Micali S., V. Vazirani. *An O(square root of V * E) Algorithm for Finding Maximum Matching in General Graphs*. Proc. 21st Annual Symposium on Foundation of Computer Science, IEEE. 1980.

[*MIT*] Massachusetts Institute of Technology, *http://www.mit.edu*

[*Motwani, Ragharan-1998*] Motwani R., P. Ragharan. *Randomized Algorithms*. Cambridge University Press. 1998.

[*Nelson-1996*] Nelson M., J.-L. Gailly. *The Data Compression Book*. M&T Books. 1996.

[*Papernov, Stasevic-1965*] Papernov A., G. Stasevic. *A method of information sorting in computer memories*. Problems of Information Transmission 1, pp. 63-75. 1965.

[*Parberry-1995*] Parberry I. *Problems on Algorithms*. Prentice-Hall. 1995.

[*Pratt-1979*] Pratt V. *Shellsort and Sorting Networks*. Garland. 1979.

[*Primes-1*] Ribenboim P. *Selling primes*. Math. Mag. 68:pp.175-182. 1995.

[*Primes-2*] *http://www.utm.edu/research/primes/notes/conjectures/*

[*Primes-3*] *http://www.utm.edu/research/primes/notes/6972593*

[*Prize-2000*] Millenium Prize Problems *http://www.claymath.org/prize_problems/index.htm*

[*Reinelt-1994*] Reinelt G. *The Traveling Salesman. Computational Solutions for TSP Applications*. Springer-Verlag. 1994.

[*Riesel-1994*] Riesel H. *Prime Numbers and Computer Methods for Factorization*. Progress in Mathematics. vol. 57, 1985; and vol. 126, 1994**.**

[*RSA-78*] Rivest R., A. Shamir, L. Adleman. *A method of obtaining digital signatures and public key cryptosystems*. Communications of the ACM, 21(2):120-126. 1978.

[*Russell,Norvig-1995*] Russell S., P. Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall. New Jersey. 1995.

[*Sedgewick-1990*] Sedgewick R. *Algorithms in C*. Addison-Wesley Publishing Company. 1990.

[*Sedgewick–1992*] Sedgewick R. *Algorithms in C++*. Addison-Wesley Publishing Company. 1992.

[*Sedgewick-1996*] Sedgewick R. *Analysis of Shellsort and Related Algorithms*. In: Josep Díaz, Maria Serna (Eds.): Algorithms - ESA '96, Fourth Annual European Symposium, Barcelona, Lecture Notes in Computer Science, Vol. 1136, Springer, 1-11. 1996.

[*Shell-1959*] Shell D. *A high-speed sorting procedure*. Communications of the ACM 2 (7), 30-32. 1959.

[*Shioura, Tamura, Uno-1997*] Shioura A., A. Tamura, T. Uno. *An Optimal Algorithm for Scanning All Spanning Trees of Undirected Graphs*. SIAM Journal on Computing,Volume 26, Number 3, pp. 678-692. 1997.

[*Skiena-1997*] Skiena S. *The Algorithm Design Manual*. Springer-Telos. 1997.

[*Smith-1998*] Smith S. *The Scientist and Engineer's Guide to Digital Signal Processing*. Addison-Wesley Publishing Company. 1997.

[*Thulassiramann, Swamy-1998*] Thulassiramann K., M. Swamy. *Graphs: Theory and Algorithms*. A Wilew-Interscience Publication John Wiley & Sons Inc. 1998.

[*Timus*] *Ural State University Problem Set Archive* http://www.timus.ru

[*TopTeam-1997*] TopTeam Co., "Най-доброто от списание BYTE", София, 1997.

[*Wai-2000*] Wai, L. *Approximate Graph Colouring*, *http://web.singnet.com.sg/~melvin/graph.html*

[*Wayner-1996*] Wayner P. *Disappearing Cryptography*. AP Professional. Boston. 1996.

[*Web-d*]. Диаметър на Интернет. *http://www.chronicle.com/free/99/09/99090901t.htm*

[*Williams-1991*] Williams R. *LZRW4: Ziv and Lempel meet Markov*, *http://www.cs.pdx.edu/~idr/unbzip2.cgi?compression/lzrw4.html.bz2*.

Предметен указател

0
0-1 задача за раницата, 462, 577, 600

2
2-3-4 дърво, 166

3
3-COL, 394
3-оцветяване на граф, 394

4
4-мерен двоичен куб, 585

A
ASCII, 170, 172, 611, 624, 630, 637, 640, 653
AVL, 165

B
bidirectional predictive-coded picture, 680
bit mapping, 611
bitmap, 623
B-picture, 680, 681
B-дърво, 166

C
coarser sampling, 677
comma code, 641

D
DCT, 678, 679
deadline, 29, 403, 580
Deep Blue, 353, 389
diatomic encoding, 616
Discrete Cosine Transform, 678
DivX, 607
DoubleSpace, 675

E
EBCDIC, 613
escape последователности, 623
escape символ, 623, 653, 686
exp-space, 352

F
FLZ, 666, 674, 681, 689

G
GIF, 607, 663, 673, 680, 690, 695

H
halfbyte packing, 613

I
IEEE, 34, 506, 646, 663, 695
intra-coded picture, 680
I-picture, 680

J
Java, 23, 26, 617, 685
Javascript, 28
JPEG, 607, 678, 690

K
k-връх в дърво, 166
k-клика, 399
k-свързаност, 317, 344

L
LPC, 620, 621
LZ77, 609, 663
LZ78, 609, 663
LZSS, 666, 689
LZW, 609, 663, 695

M
memoization, 69, 79, 105, 461, 462, 477, 504, 510, 512, 517, 522, 528, 539, 571
MP3, 607
MPEG, 607, 680, 691

N
non-deterministic polynomial time, 351
NP-задачи, 351, 352, 355
NP-complete, 355
NP-пълни задачи, 355
NP-трудни, 356
null suppression, 609
n-мерен двоичен куб, 584

P
Perl, 28
PHP, 28
PKZIP, 607
PostScript, 663

P-picture, 680, 681
predictive-coded picture, 680
P-space, 352
P-задачи, 351
p-радиус, 332
p-център, 332, 345

Q

quantization, 677

R

run-length encoding, 622
r-оцветяване, 337, 362

S

shift in, 610
shift out, 610
SI, 610, 623, 684
smart терминали, 500, 558
SO, 610, 623, 684
Stacker, 607, 675

T

TIFF, 607, 663
t-клика, 254

U

Unicode, 637

А

абстрактни структури от данни, 137
автоматен език, 520
автоматна граматика, 522, 560
адаптивен метод, 651, 665
адаптивно компресиране, 654
Аделсън, Велски и Ландис, 163
адитивна редица, 436, 454
адитивно хеширане, 172
Алан и Боб, 395, 472, 557
Алгол, 519
алгоритми с приближение, 591, 598
алгоритъм на Дейкстра, 188, 276, 566
алгоритъм на Евклид, 67, 127
алгоритъм на Крускал, 318, 566
алгоритъм на Пнуели, Лемпел и Ивена, 299
алгоритъм на Прим, 322, 566
алгоритъм на Уоршал, 280, 303, 343, 566
алгоритъм на Флойд, 272, 341, 566
алгоритъм на Форд-Белман, 271, 566
алгоритъм на Хъфман, 456, 639
алгоритъм на Шенън-Фано, 456
алгоритъм на Щрасен, 438
алфа-бета изследване, 388
алфа-бета изследване до определена дълбочина, 388
алфа-бета отсичане, 386
алфа-стойност на връх, 387
алчни алгоритми, 567
амортизационен анализ, 112
аритметично кодиране, 645, 676
асимптотична нотация, 94
асимптотични функции, 100

Б

база в граф, 329
база от върхове, 347
балансирани дървета, 164, 185, 247, 485, 666
Барнсли, 683
безконтекстна граматика, 520
безплатен обяд, 460, 683
Бейсик, 617, 685, 694
Бекус, 518
Бекус-Наурова нормална форма, 518
Бел, 90
Белман, 272
Бентли, 244, 695
бета-стойност на връх, 387
Бетчер, 228
биекция, 223, 397, 401
биномни коефициенти, 52, 507
битови карти, 611, 615, 622, 684
битонични последователности, 227
Блум, 412
БНФ, 518
бройни системи, 56
брояч на достъпа, 239, 249
бързо повдигане в степен, 434
бързо сортиране, 23, 204, 211, 219, 234, 455
бързо умножение на дълги числа, 441, 455
бързо умножение на матрици, 40, 436, 439, 454, 694

В

вариации, 82
Вен, 31
вероятностни алгоритми, 567, 593
вечен шах, 381
Виноград, 438, 454
връщане от рекурсията, 73
възстановяване на дърво, 189
възстановяване на израз, 188
външен път, 629
върхово покритие на граф, 598

Г

генетични алгоритми, 22, 593

геометрична прогресия, 57, 197
Голдбах, 43, 116
граница на оптималност, 598
граф, 251
Грей, 582, 586, 602
Гриз, 450

Д

двоичен код, 585, 613, 625, 661
двоичен код на Грей, 585
двоично дърво, 153, 178, 188
двоично дърво за претърсване, 156, 160, 164, 189, 485, 486
двоично сливане, 427
двоично търсене, 24, 151, 195, 242, 255, 352, 427, 456, 496, 587
двойкосъчетание, 336, 399
двойно сортиране, 234
двойно хеширане, 176
двуатомно кодиране, 691
двуделен граф, 336, 372, 399, 575, 601
двупроцесорно разписание, 403
двусвързан списък, 145, 187
двусвързаност, 314
Дейкстра, 23, 271, 341, 347, 352
дек, 140, 145, 151, 183, 672, 691
Диаграми на Вен, 32
Диаметър на граф, 254, 349
диаметър на покриващо дърво, 402
дизюнкт, 359, 399
динамична реализация, 148, 184
динамично оптимиране, 459
ДКЛ, 158
ДЛК, 158
доминиращи множества, 327
домино-редица, 552, 553, 563
допускане на противното, 226, 271, 576
допълване на ацикличен граф, 309
допълнителна памет за колизии, 177
Дор, 414
достижимост и свързаност, 310
дробна задача за раницата, 577
Дъглас Адамс, 17
дълги числа, 439, 455, 507, 559
дължина на път, 153, 278, 279, 349, 404
дървета на Фибоначи, 163, 247
дърво на кандидатите за решение, 379
дърво на решенията, 404
дърво на сравненията, 208
дърво на Фибоначи, 162

Е

Евклид, 25, 42, 67, 119
евристичен алгоритъм, 393, 436, 454, 605
Египетски дроби, 568
еднобайтов брояч, 613, 684
едносвързан списък, 145, 151, 184
език от общ тип, 519, 560
експоненциални задачи, 351
екстремален път в граф, 270
елементарна операция, 101
елементарни методи за сортиране, 234
ентропия, 658, 665, 689
Ератостен, 25, 44, 116

З

задача за магнитната лента, 579
задача за осемте царици, 369
задача за раницата, 460, 471, 577, 579
задача за раницата, 379, 394, 462, 694
задача за търговския пътник, 286, 394
задачата на Сколем, 354
задачи за проверимост, 351
законите на Де Морган, 357, 390
Замунда, 235
замяна на шаблони, 691
запълване с квадрати, 605
зтворено хеширане, 175
зациклен израз, 551, 562
Зив, 663, 675
златното сечение, 68, 119, 170
Зобрист, 175, 185

И

Ивена, 302, 343
игра, 382
идеално балансирано дърво, 166, 189, 208, 247, 485
извличане на цифри, 175
изключване на връх по даден ключ, 156
изключване от списък, 148
изоморфизъм на графи, 348
изрязване, 378, 677
изчерпващи задачи, 404
изчисти наполовина, 227
индуциран от V' подграф, 253
индуциран път, 401
инективна, 212, 222
интерполационно търсене, 456
информационен източник, 661
итеративни функционални системи, 683
итерация, 13, 27, 63, 198, 219, 265, 319, 438, 443, 518

Й

йерархия на Чомски, 519

К

капацитет на върховете, 297
капацитет на хеш-таблица, 170
Карл Май, 22
Каталан, 480, 557, 694
квадратично пробване, 176
квадратично търсене, 242, 249
квантифициране, 677
квантифицираща таблица, 679
КДЛ, 160
Кер, 439
Китайския пощальон, 346
класификация на задачите, 351
класификация по памет, 353
класически хеш-функции, 170
класове от задачи, 351
КЛД, 160, 189
кликово число, 254, 324
клъстеризация на множество, 402
Кнут, 170, 191, 197, 696
код на Грей, 585
код с разделители, 641, 687
кодиране на последователности, 611, 624, 666, 679, 686, 691
кодиране с линейно предсказване, 621
кодираща група, 613
кодираща двойка, 609
кодиращо дърво, 627, 640
колизия, 174, 180
комбинации, 85
компаратор, 225
компоненти на свързаност, 257, 311
компоненти на силна свързаност, 312
компресиране, 607
компресиране с битови карти, 684, 691
компресиране със загуба, 608, 677
контекстно зависима граматика, 522, 560
контекстно свободен език, 503, 518
контекстно свободна граматика, 522, 560
контрол на компании, 566
конюнктивна нормална форма, 358
кореново дърво, 152, 255, 605
краен неориентиран граф, 252
краища на път, 254
криптиране, 42, 625
Крускал, 318, 352, 576, 577, 601
кръстословица, 395
Кубичен подграф, 398
Кук, 355
кумулативен масив, 502
Куратовски, 339
Кърниган, 449
Кьонигсберг, 289

Л

лаком алгоритъм, 570, 600
Лас Вегас, 588, 605
латински квадрат, 132, 405
ЛДК, 160, 189, 315
Ледерман, 438
леймър, 13, 22
Лемер, 50, 117
Лемпел, 302, 343, 663
линеаризирана матрица, 480, 532, 557
линеен списък, 139, 145, 148, 151, 183, 187, 189, 425
линейна алгебра, 19, 682
линейна наредба, 211, 225, 233, 417
линейно пробване, 175
листо, 152
литерали, 358
ЛКД, 160, 189
логаритмична сложност, 104
Лукас, 50, 117
Луома, 438

М

магически квадрат, 131, 404
мажорант, 414, 453, 454
максимален двуделен подграф, 397
максимален поток, 293, 296
максимален пълен подграф, 399
максимален път, 270, 281
максимални независими множества, 325, 345
максимално двойкосъчетание, 336
максимално съчетание на дейности, 570, 604
мантиса, 506
Марков, 653
математическа индукция, 226
математическо очакване, 620
матрица на достижимост, 257, 280, 298, 302, 303, 343
матрица на инцидентност, 257
матрица на съседство, 256, 258, 265, 291
матрица на теглата, 256, 258
матрици, 39
медиана, 204, 232, 413, 424, 453
медиана на медианите, 413,453, 456
Мерсен, 694
Мерсенови прости числа, 50
Мерсенови числа, 49, 117
метасимволи, 519
метод на бройните системи, 220
метод на вълната, 349
метод на мехурчето, 199, 434, 454
метод на ограничителя, 194
метод на разклоненията и границите, 378

мехурче без флаг, 199, 231
мехурче с флаг, 199, 231
минимален k-свързан подграф, 401
минимален цикъл през k върха, 347
минимален цикъл през връх, 286, 342, 566
минимално r-оцветяване, 362
минимално върхово покритие, 598
минимално оцветяване на граф, 363, 573, 604
минимално покриващо дърво, 318, 403, 605
множества, 31
модел на Марков, 654, 677, 690
Монте Карло, 588, 604
морзов код, 608, 609
морзовата азбука, 376, 377
МПД, 318, 322, 324
мултиграф, 253, 280, 310, 342
мултипликативно хеширане, 170

Н

надеждност на път, 275
най-голям общ делител, 70
най-голям общ подграф, 398
най-дълга обща подредица, 489, 500, 558
най-дълъг прост път, 283, 355, 364, 391
най-дълъг път в ацикличен граф, 281, 341
най-кратък път, 264, 340, 400
най-кратък път между два върха, 264, 340
най-къс общ подниз, 396
най-малко общо кратно, 72
намаляваща подредица, 544
намиране на всички прости пътища, 268, 340
нарастващ път, 276, 294, 336, 345
нарастваща подредица, 544
Наур, 519
начален нетерминал, 518
независими множества, 324
ненамаляваща подредица, 347, 493, 544
неопределена позиция, 383
неопределени задачи, 354
Неперовото число, 56
неравенство на триъгълника, 271
нерешими задачи, 352
нетерминал, 518
ниво на връх, 153
НОД, 70, 119, 176
НОК, 72, 119

О

обединение на множества, 32
обобщен алгоритъм на Флойд, 275
обобщен ход на коня, 603
обобщена схема на Флойд, 272
обобщено хеширане по CRC, 173
обратен алгоритъм на Уоршал, 566
обратен полски запис, 160, 184, 188, 693
обратна матрица, 604
обратната функция на Акерман, 351
обхождане в дълбочина, 262, 313
обхождане в ширина, 259, 261
обхождане до определена дълбочина, 388
обхождане корен-ляво-дясно, 160
обхождане на граф, 259, 262, 284, 290, 306, 311, 340, 352, 382
обхождане на дърво, 160, 587
обхождане на списък, 146
обща подредица, 489, 497, 694
ограничител, 194, 231, 237, 403, 425
Ойлер, 51, 289, 291, 342, 693
Ойлеров граф, 290
Ойлерови цикли, 289, 349
опашка, 140
опашка за билети, 140, 537
Оперман, 43, 116
оптимален побуквен код, 628, 686
оптимална подструктура, 460, 476, 487, 503, 530, 567
оптимална селекция, 379
оптималната подструктура, 576
оптимални пътища в граф, 264
оптимални стратегии, 381
оптимално двоично дърво за претърсване, 486, 557
оптимално линейно нареждане, 398
ориентиран диаметър, 403
ориентиран претеглен граф, 251, 303
ориентиране на ребрата, 299
ориентирани ребра, 251, 300
осемте царици, 605
основна теорема, 47, 112
остатък при деление с размера на таблицата, 170
отворено хеширане, 177
относително кодиране, 618, 680, 691
отрицателен цикъл, 270, 340
отсяване, 45, 209
оцветяване на граф, 337, 362
оцветяване на двуделен граф, 575

П

пакетиране на кутии, 395
пакетиране на множество, 399
паралелни алгоритми, 224, 396
Паскал, 28, 91, 122, 191, 507, 511, 559, 617, 618, 685, 693
пат, 381
Паука, 438

пентамино, 404
пермутации, 76, 81
пермутационна схема, 230, 234
пирамида, 130, 188, 208, 209, 210, 233, 234, 279, 410, 456, 634, 637, 639, 687
пирамидално сортиране, 188, 207, 210
ПЛ/1, 618
планарен граф, 338, 339, 348
Плоджер, 449
Пнуели, 302, 343
побитово сортиране, 217, 220, 456
побуквен код, 641, 649, 656
побуквено превеждане, 376
повдигане в степен, 279, 341
повдигане в степен на матрицата на съседство, 279
повдигане на средата в квадрат, 171
поглед един символ напред, 519
поддърво, 152
покриване на квадрат, 395
покритие с интервали, 603
полиномиално разрешими задачи, 352
положителен цикъл, 270
полубайтово пакетиране, 613, 691
полустепен на входа, 253, 330, 348, 403
полустепен на изхода, 290, 348, 403
последователно търсене, 238, 245, 249
последователно търсене в сортиран списък, 238, 249
последователно търсене с преподреждане, 240, 249
построяване на граф, 310
поток с минимална цена, 293
потоци, 292
потоци в граф, 264, 336
прав и обратен полски запис, 160
Прат, 197, 412
премахване на нулите, 611, 619, 622, 684, 691
преобразувание на Фурие, 50, 678
префиксен код, 626
префиксни суми, 502
признак за оптималност, 270
Прим, 322, 344, 352, 576, 577, 601
принцип на минимума и максимума, 386
приоритетна опашка, 140, 151, 188, 456, 634, 694
принцип на чекмеджетата, 170
припокриване на подзадачите, 460, 477
проверка дали граф е цикличен, 267, 340
продукции, 518
прости операции с граф, 255
прости числа, 42, 127
прости числа в интервал, 44, 116
процесорно разписание, 580
пълен и празен неориентиран граф, 254
пълна линейна наредба, 460
пълно топологично сортиране, 307
път с най-голяма надеждност, 276
път с най-голяма пропускливост, 276

Р

равномерен код, 625, 630, 640, 656, 659
равномерно разпределение, 173, 248
радиус в граф, 332
разбиване на граф, 397
разбиване на естествено число, 515
разбиване на числа, 87
разделим код, 626, 686
разделяй и владей, 407
разделяне на триъгълници, 400
разделяща точка, 314
разлагане на число, 47
разлика на множества, 33, 114
разписание на училищна програма, 372
разрязване на материали, 548, 562
разходка на коня, 366
ребрено хроматично число, 337
ребро, 251
редица на Фибоначи, 65
рекурентни функции, 549
рекурентни функции, 38
рекурсия, 704
речникови методи, 664, 692
решето на Ератостен, 44
Ривест, 412
Римски цифри, 61
родословно дърво, 154
ротация, 164
ротирано хеширане, 172

С

свиване на пътя, 320, 321
свързан списък, 145, 148, 161, 177, 189, 195, 238, 428
сгъване, 171
симетрична разлика, 33, 114
симулация на рекурсия, 188
синтактичен анализ, 188, 519
синтактичен граф, 519
сливане на сортирани масиви, 427
сливаща схема, 228, 229
сложност на алгоритъм, 25, 93, 101, 123
сложност по време, 351
сложност по памет, 352, 496

случайна величина, 620
смесена бройна система, 222
сортиране чрез броене, 217, 410, 414, 453
сортиране чрез вмъкване, 114, 193, 204, 230, 454
сортиране чрез клатене, 199, 205, 231
сортиране чрез множество, 212, 233
сортиране чрез сливане, 211, 416, 433, 454
сортиране чрез сравнение, 191, 210, 234
сортиране чрез трансформация, 212, 222
сортираща схема, 228
списък на наследниците, 256
списък на препълванията, 177
списък на ребрата, 255
справяне с колизии, 175, 183, 187
сравнение на символни низове, 501, 558
среден елемент, 410
средна стойност, 620
статична реализация, 151, 219, 257, 427
статично допълване, 438
стек, 139
Стенли, 646, 655, 695
Стирлинг, 91, 122, 129, 211
стоп-елемент, 425
строго случайно число, 590
структури от данни, 13, 27, 63, 137, 191
сума при геометрична прогресия, 603
супеизточник, 297
схема от компаратори, 225
съвършени числа, 50
създаване на празен граф, 257
сюрективна, 223

Т

Тарджан, 412, 424
теорема за четирите цвята, 339, 346
теория на графите, 250
терминал, 500
терминална позиция, 383
тернарна пирамида, 210, 233
тернарно дърво, 639
топологично сортиране, 298, 305
транзитивна редукция, 302
транзитивно затваряне, 298
транзитивно ориентиране, 299
транспозиционна схема, 230
триангулация, 480, 489, 558
триангулация на многоъгълник, 482, 489, 558
трионообразна редица, 554
трихотомичност, 100, 225
триъгълник на Паскал, 52
троичен код, 639
троично булево дърво, 605
троично дърво, 530
търсене по даден ключ, 156
търсене с връщане, 351
търсене с налучкване, 587
търсене със стъпка, 240

У

увеличаващ път, 294, 342
удовлетворимост на булева функция, 355
уейвлет, 682
Уелч, 663, 675
Уилсън, 43, 116
Уилямс, 188, 207
Уирт, 28, 145, 160, 166, 188, 191, 195, 204, 239, 243, 433, 485, 519, 695
умножение на матрици, 279, 436, 454, 460, 475, 557, 604, 694
унгарски алгоритъм, 293
универсален алгоритъм за сортиране, 198, 203, 229, 433
универсална колекция от хеш-функции, 174
универсално хеширане, 174
Уоршал, 280, 298, 303, 341
устойчив вариант, 216, 233

Ф

факториел, 38, 64
факторизация, 52
Ферма, 354, 589
Фибоначи, 65, 91, 105, 119, 128, 162, 185, 189, 246, 459, 504, 538, 559
Фибоначиево търсене, 245, 456
фиксирано приближение, 598, 606
Флойд, 271, 332, 341, 352, 412
Фон-Нойман, 20
Форд, 271, 279, 293, 297, 340
Форд-Белман, 271, 340
формален език, 518
формална граматика, 518
формула на Хорнер, 60
Фортран, 617, 685
Фулкерсон, 279, 293, 298, 341
фундаментално множество от цикли, 284
функция на нареждане, 192, 212
Фурие, 678

Х

Хаар, 682
Хамилтонов път, 367, 391, 394, 582
Хамилтонов цикъл, 286, 342, 378, 390, 586, 593, 597, 603
Характеристични уравнения, 107

хедонийски език, 522
хеширане, 168
хеширане едно по едно, 173
хеширане по Пиърсън, 173
хеш-множество, 180
хеш-функции върху символни низове, 171
хеш-функции върху части от ключа, 170
хеш-функция, 168, 174, 180
хеш-функция на Зобрист, 174
Хилберт, 25
хиперкуб, 582, 585
хипотеза на Варнсдорф, 584
хипотези на Голдбах, 42
Хиршберг, 499, 558
ход на коня, 367, 582, 602
Хоор, 198, 204, 219, 231, 410, 456
Хопкрофт, 439
Хорнер, 60
хроматично число, 337
Хъс, 438
Хъфман, 609, 619, 629, 644, 649, 651, 663, 673, 686, 692

Ц

Цвик, 414
целева функция, 460
целочислено деление, 35
цена на кода, 628, 661
център в граф, 332
цикли в граф, 264, 284, 357
цикличен списък, 187

Ч

частично (непълно) решение, 356
частично МПД, 324
червено-черни дървета, 164
четно-нечетна сливаща схема, 229
четно-нечетна сортираща схема, 229
числа, 33, 90, 105, 128, 480, 504, 516, 693
числа на Бел и Стирлинг, 90
числа на Фибоначи, 65
числени алгоритми с приближение, 591
число на доминиране, 327
число на независимост, 325
числов триъгълник, 529
числова база в граф, 347
Чомски, 519, 523, 560

Ш

Шел, 195, 205, 232
Шенън-Фано, 609, 625, 639, 686
шумозащитно кодиране, 607, 625

Щ

Щрасен, 40, 436, 475, 694

Я

ядро на граф, 399