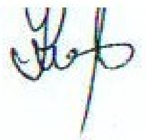


н.с. Емил Келеведжиев

Динамично оптимиране

Ръководство за решаване на задачи по програмиране

Библиотека ENTER

Под общата редакция на Красимир Манев

Съвместно издание
на Мусала Софт и ИК „Анубис"
София, 2001

Учебното пособие е предназначено за любители на задачи по програмиране. Към кръга на тези читатели несъмнено принадлежат участниците в национални и международни конкурси и олимпиади по информатика, както и тези които се подготвят да участват в такива състезания. Изложеното може да се ползва от ученици в средните училища, от студенти в университетите, от преподаватели по информатика и от професионалисти, търсещи ефективни алгоритми при разработване на софтуер.

Генерален спонсор на заглавията от „Синя серия" е Мусала Софт ООД.

н.с. Емил Келеведжиев

Динамично оптимиране

Ръководство за решаване на задачи по програмиране

Редактор *Красимир Манев*
Библиотечно оформление *Борислав Кьосев*
Технически редактор *Симеон Айтов*
Коректор *Румяна Стефанова*

Българска. Издание първо.
Излязла от печат 2001 г.
Формат 60x90/16. Печатни коли 5,5.

Издателска къща „Анубис", 1124 София,
ул. „Младен Павлов" №1, тел. 443 503, 944 16 43
Търговски отдел – ул. „Никола Ракитин" №2
тел. 946 16 76, тел./факс 946 14 32

Печатница „Симолини-94"

© Емил Келеведжиев, 2001
© Борислав Кьосев, библиотечно оформление, 2001

ISBN 954-426-312-8

Съдържание

Увод 5
Глава 1. Таблица от стойности вместо рекурсия 7
1.1. Числа на Фибоначи 8
1.2. Биномни коефициенти 10
Глава 2. „Едномерни задачи" 13
2.1. Редици от 0 и 1 13
2.2. Възстановяване на скоби 17
2.3. Образуване на суми 19
2.4. Ходове с коня 21
Глава 3. „Двумерни задачи" 22
3.1. Движение на североизток 23
3.2. Триъгълник от числа 29
3.3. Управление на врата 31
Глава 4. Основни приложения 35
4.1. Оптимална триангулация 35
4.2. Оптимално умножаване на няколко матрици 41
4.3. Разпределяне на числа 43
4.4. Най-дълга растяща подредица 47
4.5. Най-дълга обща подредица 49
4.6. Търсене на подниз 52
4.7. Оптимални дървета за търсене 55
4.8. Най-къси пътища в графи 59
Глава 5. Задачи от дискретното оптимиране 67
5.1. Задача за раницата 67
5.2. Задача за търговския пътник 73
Теми, задачи и въпроси за самостоятелна работа 80
Литература 86

Увод

През последните години на състезанията по информатика от национален и международен мащаб става обичайно да се предлагат проблеми, които успешно могат да се решат чрез метода на динамичното оптимиране.

В настоящето ръководство този метод е представен като подход за алгоритмизация и програмиране, отнасящ се за определени задачи. Когато е възможно да бъде приложен, той води до значително намаляване на времето, необходимо на компютъра да намери решението. Без метода на динамичното оптимиране такива задачи в общия случай трябва да се атакуват чрез изчерпващо търсене измежду всички възможни варианти от кандидати за решения. Това обаче предизвиква драстично нарастване (например експоненциално) на времето при увеличаване стойността на параметрите или на размера на входните данни. В такъв смисъл динамичното оптимиране е междинен по ефективност метод, който е много по-добър от „backtracking"-а, но е по-слаб от „greedy"-алгоритъма (ако такъв съществува за разглежданата задача).

Динамичното оптимиране (наричано още и *динамично програмиране*) е започнало да се разработва и прилага систематично след 1955 г. от американския математик Ричард Белман. Възникването и развитието му са в непосредствена връзка с появата и усъвършенстването на компютрите. Почти е немислимо динамичното оптимиране да се ползва за „ръчно" смятане, понеже то е свързано с обработване на голям обем данни. Днес този метод е известен главно като теория и технология за решаване на сложни *оптимизационни* задачи, имащи практическа ценност. Към тях се отнасят някои от задачите на целочислената (дискретна) оптимизация, възникващи в области, свързани с изследване на операциите, а също и определени задачи за оптимално управление на динамични системи.

При написването на настоящето пособие съм предполагал, че читателят е запознат с базовите алгоритми, осигуряващи основа за решаване на конкурсни задачи по информатика, и може да ги програмира на език от типа на C/C++ или Pascal. Част от тези умения могат да се придобият чрез посочените в края на книгата учебни пособия. За да се разберат предложените решения, не са необходими почти никакви конкретни предварителни сведения по математика и информатика, излизащи извън елементарната програма от средното училище. При четенето на книгата читателят не е задължен да спазва точно последователността на материала, представен в съдържанието — затова в някои от разделите са дадени частични повторения на основните идеи.

В изложението са вмъкнати фрагменти от програмни текстове, а на отделни места са публикувани и завършени програми. Всички те са написани на алгоритмичния език C, като са използвани съвсем малко някои допълнителни улеснения, предлагани от C++. Проверката на програмите е извършена чрез системата Borland C++, 3.0. Поради алгоритмичното естество на разглежданите въпроси от читателя не се изискват задълбочени познания за използвания език, а само способност да възприема записаните инструкции. Предполагам, че по този начин пособието ще е достъпно и за тези, които ползват Pascal като основен език за програмиране.

* * *

Авторът изказва своята благодарност към членовете на Екипа за извънкласна работа по информатика към Съюза на математиците в България, а също и към фирма MusalaSoft и към издателство „Анубис“, без подкрепата на които написването и цялостната подготовка на настоящето пособие нямаше да бъдат възможни.

София,
6 януари 2001 г.

Емил Келеведжиев
(keleved@math.bas.bg)
Институт по математика
и информатика при БАН

Глава 1

Таблица от стойности вместо рекурсия

Колкото очевиден, толкова и полезен подход за решаване на една задача е „разбиването“ ѝ на по-малки, лесно решаващи се части (подзадачи). Важен случай имаме, когато в процеса на намиране на подзадачите открием, че първоначалната задача се разлага на нови задачи, които са от същия вид като изходната, но в някакъв смисъл по-прости или с по-малки стойности на числените си параметри. Тогава рекурсивният метод става естествено приложим.

Прилагат се две основни алгоритмични конструкции при разлагането на една задача на подзадачи:

1. Алгоритъм, основан на подхода *разделяй и владей*, обикновено разделя задачата на две еднакви части, решава всяка от тях и след това съединява двете частни решения, за да получи цялостното решение. Типичен пример за разделяй и владей е двоичното търсене и различните негови варианти.

2. Алгоритъм, базиращ се на идеята на *динамичното оптимиране*, в повечето случаи премахва един „елемент“ от задачата, решава получената по-малка задача и след това използва това решение, за да се върне към временно премахнатия „елемент“ и оттам да намери цялостното решение.

Веднъж разбрана, идеята на динамичното оптимиране е може би най-лесният подход за конструиране на алгоритми. Даже е по-лесно човек да изобрети сам отново този метод, отколкото да го изучи, прочитайки го от учебниците. Но преди да го разбере, той изглежда като магия. И както е при усвояването на фокуси, придобиване на умението да се ползва динамичното оптимиране най-добре се получава чрез внимателно проследяване на серия от примери.

1.1. Числа на Фибоначи

Необходимостта от балансиране между използвания обем памет и времето за работа на една програма често възниква при решаване на задачи по програмиране. Този компромис ясно се илюстрира при пресмятания, свързани с рекурентни зависимости.

Редицата от числата на Фибоначи е разгледана от италианския математик Фибоначи през тринадесети век. Чрез тях той е моделирал нарастването на популацията на зайците. Фибоначи е забелязал, че броят на двойките зайци, родени през дадена година, е равен на сумата от броя на двойките зайци, родени през двете предишни години. За да изразим числено този процес, дефинираме за n-тата година следната рекурентна зависимост:

$$F_n = F_{n-1} + F_{n-2}$$

заедно с началните условия $F_0 = 0$ и $F_1 = 1$. Това дава $F_2 = 1$, $F_3 = 2$, и след това редицата продължава като 3, 5, 8, 13, 21, 34, 55, 89, 144, ... Оказва се, че тази редица, освен за преброяването на популацията от зайци, има и други многобройни приложения.

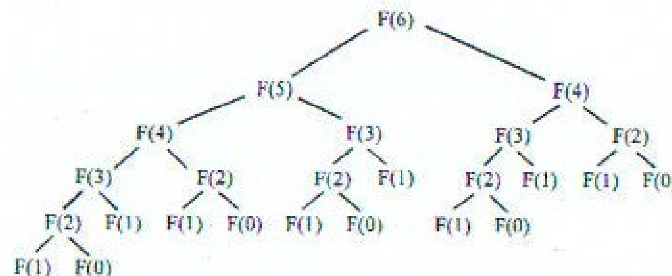
Понеже дефиницията на редицата на Фибоначи се дава чрез рекурсивна формула, очевиден начин да се програмира пресмятането ѝ е чрез рекурсивна функция на алгоритмичен език:

```
int F (int n)
{ if (n == 0) return 0;
  else if (n == 1) return 1;
  else return F(n-1) + F(n-2);
}
```

Тази проста програма обаче води до прекалено много пресмятания. Даже извикването `F(6)` поражда сравнително голямо дърво от извиквания на функцията с по-малки стойности на аргумента (фиг. 1.1.1).

За да изразим количествено времето за работа на алгоритъма, използваме една известна формула ([3]), която няма да обосноваваме тук:

$$\frac{F_{n+1}}{F_n} \approx \phi = \frac{1+\sqrt{5}}{2} \approx 1.61803.$$



Фиг. 1.1.1. Извикването на `F(6)` поражда дърво от следващи извиквания на същата функция

Появилото се при нея число ϕ е известно като *златното сечение*.

От формулата се получава, че $F_n > 1.6^n$. Понеже при пресмятанията, съгласно изобразеното дърво на рекурсията, се тръгва от листа, които са 0 и 1, за да получим чрез събиране F_n, трябва да сме употребили поне 1.6^n извиквания на функцията `F`. Това показва, че така съставената програма изисква експоненциално време за работа.

Пресмятането на числата на Фибоначи може да се организира така, че да е необходимо линейно време по n, като запазваме всички пресметнати стойности в масив `F[i]`:

```
F[0] = 0;
F[1] = 1;
for (i = 2; i <= n; i++) F[i] = F[i-1] + F[i-2];
```

Макар и тривиален, този програмен фрагмент илюстрира основната идея — как може да се пресмятат числата на Фибоначи последователно от по-малките към по-големите и същевременно да се запазват предишните резултати, така че когато ни е необходимо да пресметнем F_n, ние вече да имаме пресметнати F_{n-1} и F_{n-2} и да ги използваме.

Веднага можем да забележим, че не е необходимо да запазваме всички пресметнати до текущия момент числа, за да пресметнем F_n. Достатъчно е да имаме на разположение само двете предишни стойности. Това може да се осъществи без да използваме масив, а само две променливи и по подходящ начин да им разменяме стойностите:

```
f0 = 0;
f1 = 1;
for (i = 2; i <= n; i++) {
  f = f1 + f0;
  f0 = f1;
  f1 = f;
}
```

1.2. Биномни коефициенти

Следващата рекурсивна функция пресмята броя на комбинациите $C(n,k)$, които могат да се съставят от n елемента в групи по k:

```
int C (int n, int k)
{ if ((k == 0) || (k == n)) return 1;
  else return C(n-1, k-1) + C(n-1, k);
}
```

Числата $C(n,k)$ са известни още и като биномни коефициенти и за тях се използва също и означението $\binom{n}{k}$.

Като пример да посочим, че от 4 елемента (1,2,3,4) могат да се образуват 6 комбинации в групи от по 2 елемента:

$$12, 13, 14, 23, 24, 34$$

и следователно $C(4,2) = 6$.

$C(n,k)$ е равно и на броя на k-елементните подмножества на n-елементно множество. Във верността на съотношението

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

се убеждаваме, като фиксираме произволен елемент на n-елементното множество и поотделно преброим k-елементните подмножества, включващи и невключващи фиксирания елемент. Таблицата от стойностите на $C(n,k)$


```
        1
       1 1
      1 2 1
     1 3 3 1
    1 4 6 4 1
   . . . . . . . . . . .
```

се нарича триъгълник на Паскал. Вижда се, че всеки елемент, освен крайните единици, е равен на сумата от двата стоящи над него елемента.

За пресмятане на биномните коефициенти може теоретично да се ползва формулата

$$C(n,k) = \frac{n!}{k!(n-k)!},$$

но при програмирането ѝ трябва да се вземат мерки за избягване на препълванията, които настъпват при големи стойности на n или k.

По-общ метод за избягване на рекурсията е използването на таблица. Съставяме таблица от стойностите на функцията $C(n,k)$, като я запълваме последователно за $n = 0, 1, 2, \ldots$, докато стигнем до интересуващата ни стойност. По-долу е дадена програма, която запълва таблицата `t[n][k]` с всички стойности на биномните коефициенти до $n = 4$ включително и след това ги отпечатва:

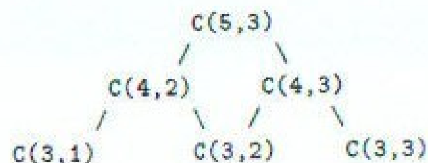
```
const N = 4;
int t[N+1][N+1];

void main ()
{ int n,k;
  for (n = 0; n <= N; n++) {
    t[n][0] = 1;
    t[n][n] = 1;
  }

  for (n = 1; n <= N; n++)
    for (k = 1; k < n; k++)
      t[n][k] = t[n-1][k-1] + t[n-1][k];

  for (n = 0; n <= N; n++) {
```

```
        for (k = 0; k <= n; k++) cout << t[n][k] << ' ';
        cout << endl;
    }
}
```

Оценяване на времето за работа и на необходимата памет за нерекурсивния и рекурсивния вариант. При нерекурсивната програма е необходима таблица (масив за стойностите `t[N+1][N+1]`). Тази таблица заема място от порядъка на n^2 числа, но то може да се намали до n, ако забележим, че за пресмятането на всеки ред от триъгълника на Паскал се използва само предишният. Времето за работа на програмата и в двата случая има порядък n^2 операции събиране.

Рекурсивната програма изисква съществено повече време за работа: всяко извикване `C(n,k)` води до две следващи извиквания: `C(n-1,k-1)` и `C(n-1,k)`, те на свой ред извикват общо 4 пъти функцията със следващите параметри и т. н. Така времето се оказва експоненциално (от порядъка на 2^n). Паметта, използвана от разглежданата рекурсивна програма, е пропорционална на n. Тя се получава, като се умножи дълбочината на рекурсията n с количеството памет, необходима на един екземпляр на функцията, а тя е константа.

Основната причина за кардиналната печалба от време при преминаване от рекурсивната версия към нерекурсивната се дължи на факта, че при рекурсията едни и същи пресмятания се извършват многократно. Например извикването на `C(5,3)` в крайна сметка поражда двукратно извикване на `C(3,2)`. Това се вижда от следната схема:



При използването на таблицата всяка клетка се запълва само веднъж — и ето откъде идва икономията на време. Този принцип лежи в основата на метода на динамичното оптимиране и е приложим в случаите, когато обемът на информацията, която трябва да се съхранява (т. е. размерът на таблицата), не е прекалено голям.

Глава 2

„Едномерни задачи"

Идеята на метода на динамичното оптимиране се състои в преобразуване на дадената задача към фамилия от подзадачи с по-малки размери и използване на таблична техника за съхраняване на отговорите им. Предимствата на този подход са, че щом веднъж една подзадача е решена, нейният отговор се запазва и не се пресмята отново и така се ползва при решаването на други подзадачи.

В случая когато дадената задача се определя от един параметър N, разглеждан като „размер на задачата", динамичното оптимиране има идея, сходна с математическата индукция: Да предположим, че вече знаем отговора A_k на всяка от задачите с размер $k < N$ и искаме да намерим A_N, т.е. отговора за $k = N$. Ако успеем да го изразим чрез вече известните $A_0, A_1, \dots, A_{N-1}$, то получаваме алгоритъм за решаване на задачата за всяко N. Така, започвайки от няколко известни начални стойности $A_0, \dots, A_k$, намираме последователно следващите $A_{k+1}, A_{k+2}, \dots$.

2.1. Редици от 0 и 1

Задача. *Да се намери броят на редиците с дължина N, състоящи се от нули и единици.*

Решение. Ако допуснем, че знаем броя B_{k-1} на редиците от търсения вид с дължина $k-1$, тогава броят на редиците от същия вид с дължина k е равен на $B_k = 2 \cdot B_{k-1}$, защото от всяка редица с дължина $k-1$ се получават две нови редици — едната с присъединяване на 0, а другата с присъединяване на 1. Като вземем предвид, че $B_1 = 2$, можем да напишем дадения по-долу фрагмент за пресмятане на B_N. При него търсената стойност се получава в последния елемент на масива `b[i]`:

```
b[1] = 2;
for (i = 2; i <= N; i++) b[i] = 2*b[i-1];
```

Използването на масив в случая не е наложително. Понеже за пресмятането на всяка следваща стойност се използва само непосредствено предишната, достатъчно е да организираме цикъл с една променлива b:

```
b = 2;
for (i = 2; i <= N; i++) b = 2*b;
```

Задача. *Да се намери броят на редиците с дължина N, състоящи се от нули и единици и такива, че в тези редици не се срещат никъде две единици, непосредствено разположени една до друга.*

Решение. Да означим с B_k броя на редиците от разглеждания вид, които са с дължина k. Да се опитаме да изразим този брой чрез броя на редиците от същия вид, но имащи по-малка дължина. За целта да видим как може да се построи една такава редица с дължина k.

Ако за последен елемент изберем 0, то предишните $k-1$ елемента са някаква редица от разглеждания вид. Броят на тези редици е B_{k-1}.

При другия случай, ако за последен елемент е избран 1, то на предпоследното място с номер $k-1$ задължително има 0. Тогава предишните $k-2$ елемента са някаква редица от разглеждания вид, като броят на тези редици е B_{k-2}.

От казаното дотук следва, че $B_k = B_{k-1} + B_{k-2}$, защото всяка редица от разглеждания вид завършва или с 0, или с 1.

След като получихме рекурентната формула, лесно можем да организираме пресмятанията. Трябва да зададем първите две стойности при $k=1$ и $k=2$. За тях е очевидно, че $B_1 = 2$ и $B_2 = 3$. Всъщност получихме формула, по която се пресмятат известните числа на Фибоначи (виж Глава 1).

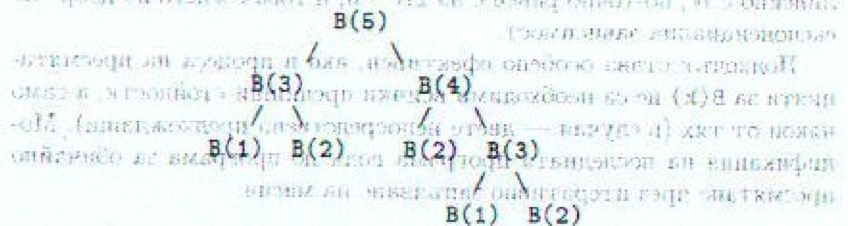
Пресмятането става чрез следния фрагмент:

```
b1 = 2;
b2 = 3;
for (i = 3; i <= N; i++) {
  b = b2 + b1;
  b1 = b2;
  b2 = b;
}
cout << b;
```


Възможно е, стойностите да не се пресмятат чрез цикъл, а с помощта на рекурсивна функция:

```
int B (int k)
{ if (k == 1) return 2;
  if (k == 2) return 3;
  return B(k-1) + B(k-2);
}
```

Тази реализация на алгоритъма обаче е неприемлива, защото даже и за малки стойности на N той извършва многократно пресмятане на едни и същи стойности. Това води до експоненциално нарастване на времето за работа. Например стойността B(N-2) се пресмята два пъти, когато искаме да пресметнем B(N); стойността B(N-3) се пресмята 3 пъти и т.н. Следната схема показва какви извиквания стават при пресмятане на B(5):



Вижда се, че B(3) е извикван 2 пъти, B(2) — 3 пъти, и B(1) — 2 пъти. Общият брой извиквания за пресмятането на B(5), както се вижда от схемата, е 9. Може да се преброи още, че за да се получи B(10), трябва да се използват 109 извиквания, за B(20) — 13529 извиквания, и т. н.

Въпреки безнадеждността, следваща от горните факти, все пак е възможно да се използва рекурсивна функция за пресмятане при по-големи стойности на N. Това може да стане, ако вече веднъж пресметнатите стойности се запомнят в статичен (глобален) масив

и след това, при нужда от тях, те да не се пресмятат отново, а да се вземат наготово. В следващата реализация за тази цел е използван масивът v[i]:

```
const Nmax = 21;
int v[Nmax];

int B(int k)
{ if (v[k] == 0) v[k] = B(k-1) + B(k-2);
  return v[k];
}

void main ()
{ v[1] = 2;
  v[2] = 3;
  for (int i = 3; i < Nmax; i++) v[i] = 0;
  cout << B(20);
}
```

Програмата пресмята B(20) само с 37 извиквания на функцията B() и лесно може да се съобрази, че броят на тези извиквания расте линейно с N, по-точно равен е на $2N-3$, и това е много по-добре от експоненциална зависимост.

Подходът става особено ефективен, ако в процеса на пресмятанията за B(k) не са необходими всички предишни стойности, а само някои от тях (в случая — двете непосредствено предхождащи). Модификация на последната програма води до програма за обичайно пресмятане чрез итеративно запълване на масив:

```
void main ()
{ v[1] = 2;
  v[2] = 3;
  for (int i = 3; i <= N; i++)
    v[i] = v[i-1] + v[i-2];
  cout << v[N];
}
```

Като важна забележка трябва да се спомене, че предишните програмни фрагменти работят правилно само в рамките на стойностите,

които могат да се „вместят" в стандартния тип за цели числа int. Дължината на числата B_k расте доста бързо спрямо k и например при $k = 100$ числото B_{100} има 21 цифри. За да бъде програмата правилно функционираща, в такива случаи тя трябва да се модифицира за работа с т. нар. „дълги" цели числа — тема, която излиза извън обхвата на настоящето ръководство.

2.2. Възстановяване на скоби

Задача. *Даден е шаблон от кръгли скоби и въпросителни знаци. Трябва да се намери по колко начина е възможно въпросителните знаци да се заместят с кръгли скоби, така че да се получи правилен израз от скоби.*

Тук няма да се спираме върху точната дефиниция на понятието правилен израз от скоби. Читателят може да си мисли, че ако вземе един правилен аритметичен израз със скоби и след това изтрие всичко друго освен скобите, ще остане това, което се нарича „правилен израз от скоби".

Пример. При даден шаблон '????(?' отговорът на задачата е 2.

Решение. Да разгледаме една редица s от отварящи и затварящи кръгли скоби с общ брой на скобите (дължина на редицата), равен на n. Означаваме с $d_i(s)$ разликата между броя на отварящите скоби и броя на затварящите скоби, които се намират измежду първите i знака (включително) в тази редица. Очевидно е, че s е правилен израз от скоби тогава и само тогава, когато $d_i(s) \geq 0$ за всяко $i = 1, 2, \ldots, n$ и освен това $d_n(s) = 0$.

Нека $F(k, c)$ да означава броя на всевъзможните редици s от скоби с дължина k, които се съгласуват с първите k знака на дадения шаблон и които са такива, че $d_i(s) \geq 0$ за всяко $i = 1, 2, \ldots, k$ и $d_k(s) = c$. Сега е ясно, че отговорът на първоначално поставената задача е равен на $F(N, 0)$, където N е дължината на дадения шаблон.

Остава да намерим начин за пресмятане на числата $F(k, c)$. Забелязваме, че

— ако k-тият знак на шаблона е отваряща скоба, то

$$F(k, c) = F(k-1, c-1);$$

— ако k-тият знак на шаблона е затваряща скоба, то

$$F(k, c) = F(k-1, c+1);$$

— ако k-тият знак на шаблона е въпросителен знак, то на негово място може да се постави както отваряща, така и затваряща скоба и следователно

$$F(k,c) = F(k-1,c-1) + F(k-1,c+1);$$

Лесно се съобразяват началните условия: $F(0,c) = 0$, $F(k,-1) = 0$ и $F(0,0) = 1$ (празната редица).

Пресмятането се извършва чрез два вложени цикъла: външен — в който се променя индексът k, и вътрешен — в който се променя индексът c. Така последователно се пресмятат числата $F(k,c)$ за всички k от 1 до N и за всички c от 0 до N. Стойността $F(N,0)$ е отговор на задачата.

Една реализация на програма, запълваща масива `F[k][c]`, е следната:

```
const N = 6;
char s[] = " ????(?";

void main ()
{ int F[N+1][N+2];
  int c, k;
  for (c = 0; c <= N+1; c++) F[0][c] = 0;
  for (k = 0; k <= N; k++) F[k][N+1] = 0;
  F[0][0] = 1;

  for (k = 1; k <= N; k++) {
    c = 0;
    if (s[k] == '(') F[k][c] = 0;
    else if (s[k] == ')' ) F[k][c] = F[k-1][c+1];
    else F[k][c] = F[k-1][c+1];

    for (c = 1; c <= N; c++)
      if(s[k] == '(') F[k][c] = F[k-1][c-1];
      else if (s[k] ==')' ) F[k][c] = F[k-1][c+1];
      else F[k][c] = F[k-1][c-1] + F[k-1][c+1];
  }
  cout << F[N][0];
}
```


Привеждаме и пример за рекурсивна програма, решаваща задачата:

```
const N = 6;
char s[] = " ????(?";

int F (int k, int c)
{ if ((k == 0) && (c == 0)) return 1;
  if (k == 0) return 0;
  if (c == -1) return 0;
  if (c == N+1) return 0;
  if (s[k] == '(' ) return F(k-1, c-1);
  else if (s[k] ==')' ) return F(k-1, c+1);
  else return F(k-1, c-1) + F(k-1, c+1);
}

void main ()
{ cout << F(N, 0);
}
```

Като упражнение предлагаме на читателя да преобразува горната рекурсивна програма в такава, при която пресметнатите веднъж стойности на функцията `F` не се пресмятат повторно, а се вземат от масив, където са били запомнени.

Забележка: Всяка програма, която решава тази задача, трябва при нечетен брой знаци в шаблона, независимо какви са те, да извежда винаги 0 за броя на възможните правилни изрази от скоби. Това е така, защото в един правилен израз от скоби броят на отварящите скоби трябва да е равен на броя на затварящите, т. е. общият брой трябва да е четно число.

2.3. Образуване на суми

Задача. *Разполагаме с по една пощенска марка от 1, 2, 3, 4 и 5 лева. Колко са начините за облепване на писмо, за да го таксуваме с 10 лева?*

За да решим задачата, „потапяме" я във фамилия от подзадачи, зависещи от два параметъра: S и k, където S е парична сума за таксуване, а k показва колко от първите няколко марки ще използваме,

за да образуваме сумата. По-общо можем да означим стойностите на марките с $a_1, a_2, \ldots, a_N$. Нека $F(k, S)$ е броят на начините за таксуване при фиксирано k и S.

Разделяме всички начини за таксуване на два вида — със и без използване на марката със стойност a_k. Ако е използвана, то остава да се образува сумата $S-a_k$ чрез останалите марки $a_1, \ldots, a_{k-1}$; ако не е използвана, тогава със същите марки трябва да се образува сумата S. Първата възможност се осъществява по $F(k-1, S-a_k)$ начина, а втората — по $F(k-1, S)$. Тогава общият брой на възможностите е равен на сбора от двете подзадачи: $F(k, S) = F(k-1, S-a_k)+F(k-1, S)$. Тази формула, заедно с очевидни начални условия, води естествено към рекурсивно програмиране (стойностите $a_1, \ldots, a_N$ са заредени в елементите a[1], ..., a[N]):

```
int F (int k, int S)
{ if (S < 0) return 0;
  if (S == 0) return 1;
  if (k == 1)
    if (a[1] == S) return 1;
    else return 0;
  return F(k-1, S-a[k]) + F(k-1, S);
}
```

Не е трудно да избегнем експоненциалния ръст при рекурсията чрез използване на таблична техника. Следващият програмен фрагмент отпечатва числото 3, което е възможният брой начини за образуване на сумата от задачата:

```
const S = 10;
const N = 5;
int a[N+1] = {0,1,2,3,4,5};
int T[N+1][S+1], k, j, w;
for (j = 0; j <= S; j++) T[0][j] = 0;
for (k = 1; k <= N; k++) T[k][0] = 1;
for (k = 1; k <= N; k++)
  if (a[1] == 1) T[k][1] = 1;
  else T[k][1] = 0;
for (k = 1; k <= N; k++)
  for (j = 2; j <= S; j++) {
```



```
    if (j-a[k] < 0) w = 0;
    else w = T[k-1][j-a[k]];
    T[k][j] = w + T[k-1][j];
  }
cout << T[N][S];
```

2.4. Ходове с коня

Едно шахматно дружество поръчало на телефонната компания да осигури за шахматистите телефонни номера, които да се избират чрез ходове на шахматния кон върху телефонната клавиатура. Стандартната клавиатура е изобразена вдясно:

```
7 8 9
4 5 6
1 2 3
  0
```

Пример за телефонен номер от описания вид е 340-49-27.

Задача. *Напишете програма, която намира броя на различните телефонни номера с дължина N, които да се набират с ходове на коня по описания начин. Номерата не могат да започват с 0.*

Упътване: Нека $F(k, d)$ да означава броя на телефонните номера от търсения вид с дължина (брой на цифрите на номера), равна на k, като първата цифра е d. Напишете рекурентни зависимости за числата $F(k, d)$. Например $F(k + 1, 6) = F(k, 0) + F(k, 1) + F(k, 7)$, защото бутонът 6 може да се достигне с ход на коня от бутоните 0, 1 или 7.

След като намерим $F(N, d)$ за всяко $d = 1, 2, \ldots, 9$, отговорът на задачата е

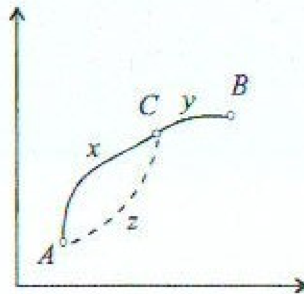
$$\sum_{d=1}^{9} F(N, d).$$

Глава 3

„Двумерни задачи“

При конструиране на алгоритми за решаване на задачи, свързани с оптимални траектории за преминаване от една до друга точка в равнината или в пространството, се използва основополагащата идея, на която е базиран методът на динамичното оптимиране: *произволна част от оптимална траектория също е оптимална траектория*.

За да разгледаме по-подробно този принцип, нека да допуснем, че ни е наложено да се движим така, че да преминем от т. A до т. B по *оптимален* начин (виж фиг. 3.0.1.). За илюстрацията на принципа не е необходимо уточняването на понятието *оптимален*, но читателят може да счита, че става дума например за най-къс път, който може да се осъществи или за минимален разход на гориво, ако движението се извършва с превозно средство.



Фиг. 3.0.1. Част от оптимална траектория също е оптимална

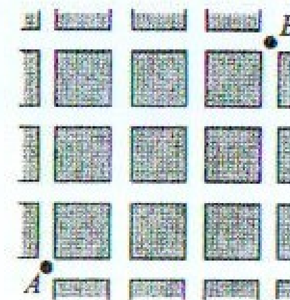
Да приемем, че изобразената траектория, съставена от частите x и y, е оптимална измежду всички възможни, които свързват A и B.

Нека C е междинна точка, съчленяваща x и y. С прости разсъждения можем да се убедим, че траекторията x е оптимална измежду всички други, които свързват A и C. За целта допускаме, че това не е така. Означаваме с f критерия за оптималност и за определеност приемаме, че оптимумът е минимум. Понеже съгласно допускането x не е оптимална, тогава може да считаме, че съществува друга траектория z от A до C, която е по-добра от x, т. е. $f(z) < f(x)$. Сега, ако съединим z и y, получаваме траектория от A до B, по-добра от първоначално разгледаната оптимална траектория: $f(z) + f(y) < f(x) + f(y)$. Стигаме до противоречие с допускането.

Да отбележим, че при нашите разсъждения използвахме *адитивността* на критерия за оптималност f и че за точките A и C *съществува* свързваща ги оптимална траектория. Обикновено свойството адитивност почти винаги е изпълнено при критериите за оптималност, а споменатият проблем за съществуване не възниква при дискретните задачи, защото при тях в съвкупността от краен (макар понякога и много голям) брой варианти винаги има такъв с най-добра стойност.

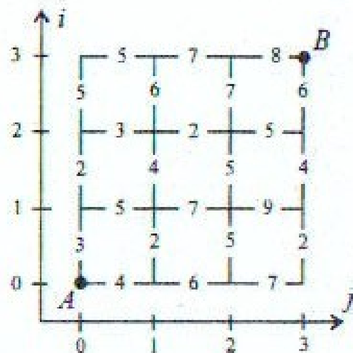
3.1. Движение на североизток

Един град има правилна правоъгълна система от улици, както е показано на фиг. 3.1.1.

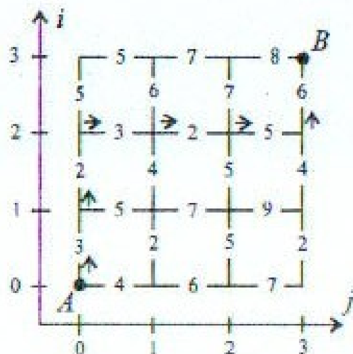


Фиг. 3.1.1. Движение от A до B в един град

Пешеходец трябва да започне движението си от т. A и да отиде до т. B, като спазва правилото, че от всяко кръстовище може да тръгне или на север или на изток. За преминаването на всяка отсечка от улица, заключена между две кръстовища, се заплаща определена такса. Схематично това може да се представи, както е изобразено на фиг. 3.1.2., където са зададени конкретни числени стойности на пътните такси.



Фиг. 3.1.2. Пример с пътни такси за всяка улица



Фиг. 3.1.3. Възможен маршрут с цена 21

Какъв маршрут да се избере, така че общата платена сума да е минимална?

На фиг. 3.1.3. е показан един възможен маршрут, който изисква 21 единици такса.

Този маршрут, както и всеки друг възможен маршрут от A до B, се състои от 6 стъпки: три на север и три на изток. В дадения пример броят на всички възможни маршрути е $20 = \binom{6}{3}$. Тази формула се обяснява с факта, че за шестте участъка на маршрута трябва да определим кои три от тях да водят на север, а това може да се направи по $\binom{6}{3}$ различни начина.

Решаването на задачата с „метода на грубата сила“ се извършва, като образуваме всички допустими маршрути, пресмятаме разходите при всеки от тях и избираме този, за който се достига минималната стойност. За задачи с неголеми размери, като дадения пример, при който маршрутът се строи от 3×3 броя хоризонтални и вертикални отсечки, описаният подход е осъществим. Но в общия случай от $N \times N$ такива отсечки, при големи стойности на N, методът на грубото (всеобхватното) изчерпване на вариантите не може да се приложи на практика. Причина за това е много бързото нарастване на необходимото време за пресмятане даже и при използване на все по-мощни компютри. Например за $N = 30$ броят на допустимите маршрути е $60!/(30! \cdot 30!) \approx 10^{17}$. За пресмятане стойността за един маршрут необходимият брой аритметични операции е от порядъка на $2N$. Това означава, че за $N = 30$ порядъкът на операциите за решаване на задачата е $60 \cdot 10^{17}$, или както лесно може да се оцени, съвременен компютър с бързодействие от няколко милиона аритметични операции в секунда ще трябва да работи няколко хиляди години непрекъснато, за да реши задачата.

Прилагането на идеите на динамичното оптимиране довежда до значително („драстично“) намаляване времето за пресмятане. Формулираме една фамилия от подзадачи, а именно — за всяко кръстовище (i, j) означаваме с $v[i][j]$ оптималното решение на задачата за намиране на маршрут с минимална такса, тръгвайки от т. A и стигайки до кръстовището с координати (i, j). Очевидно е, че $v[0][0] = 0$, т.е. цената за преминаване от т. A до същата точка е нула. Някои други стойности на $v[i][j]$ също лесно могат да се пресметнат. Например $v[0][1]$ е равно на таксата за преминаване по отсечката от т. A до съседното ѝ кръстовище с координати $(0,1)$.

За да решим първоначално поставената задача, трябва да пресметнем $v[N][N]$. Това може да направим, като последователно пресмятаме стойностите на $v[i][j]$ чрез запълването на този масив по редове (първо зареждаме стойности в $v[0][0], v[0][1], \ldots, v[0][N]$, след това в $v[1][0], v[1][1], \ldots, v[1][N]$ и т. н.).

Основната формула, която ще използваме, се основава на факта, че за да достигнем кръстовището $X = (i, j)$, трябва преди това, съгласно условието на задачата, да сме били или в $Y = (i-1, j)$, или в $Z = (i, j-1)$. Ако сме били в Y, за да отидем в X трябва да платим таксата $|YX|$ за преминаване на отсечката $|YX|$; в другия случай платената такса е $|ZX|$. Но в $X = (i, j)$ сме дошли по най-евтиния възможен начин, тръгвайки от A. Следователно пристигнали сме оттам, откъдето е по-евтино. Понеже възможностите са две, дошли сме от тази от двете точки Y или Z, за която е по-малка съответната такса $|YX|$ или $|ZX|$, плюс разходите съответно от A до Y или от A до Z.

За да напишем общата формула, нека да означим с $a[i][j]$ таксата за преминаване по „вертикалната" на чертежа отсечка от кръстовище $(i, j-1)$ до кръстовище (i, j), а с $b[i][j]$ да означим таксата за преминаване по „хоризонталната" отсечка от $(i-1, j)$ до (i, j). Тогава:

$$v[i][j] = \min\{v[i][j-1] + a[i][j],\ v[i-1][j] + b[i][j]\}.$$

При използването на тази формула за създаване на завършена програма трябва да знаем още как да заредим стойностите на граничните елементи на масива. Но за тях е очевидно, че е в сила следната рекурентна зависимост за първия ред:

$$v[i][0] = v[i-1][0] + b[i][0]$$

и аналогична зависимост за първия стълб:

$$v[0][j] = v[0][j-1] + a[0][j].$$

За начална стойност, необходима за да осъществим итерациите, трябва да вземем $v[0][0] = 0$.

Програмният фрагмент, който зарежда стойностите на масива `v[i][j]` и по този начин решава задачата, изглежда така:


```
v[0][0] = 0;

for (i = 1; i <= N; i++)
   v[i][0] = v[i-1][0] + b[i][0];

for (j = 1; j <= N; j++)
  v[0][j] = v[0][j-1] + a[0][j];

for (i = 1; i <= N; i++)
  for (j = 1; j <= N; j++) {
    north = v[i][j-1] + a[i][j];
    east  = v[i-1][j] + b[i][j];
    if (north < east) v[i][j] = north;
    else v[i][j] = east;
  }
```

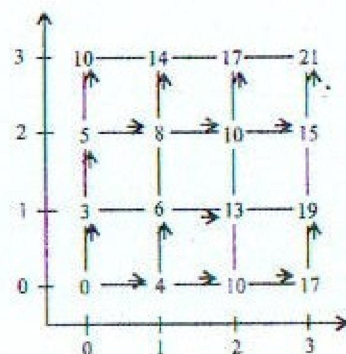
Дотук всъщност намерихме само самата оптимална стойност, т.е. минималната цена. Въпросът за намирането на самия маршрут засега остава открит. Но с този проблем лесно можем да се справим, ако по време на решаването на всяка подзадача, т.е. при зареждането на всяка от стойностите на `v[i][j]`, запомняме откъде е получена тази стойност; по-точно — при кой от двата случая е бил достигнат минимумът: дали при идване „отдолу" или при идване „отляво" върху кръстовището (i, j).

За целта в програмата трябва да използваме допълнителен масив `w[i][j]`, чиито елементи да зареждаме с `1`, ако в точката (i, j) оптималният маршрут е дошъл „отляво" и с `0`, ако в същата точка оптималният маршрут е дошъл „отдолу". Освен това за начална стойност присвояваме `w[0][0]=-1`.

Ето как изглежда модифицираният програмен фрагмент:

```
v[0][0] = 0;
w[0][0] = -1;
for (i = 1; i <= M; i++) {
  v[i][0] = v[i-1][0] + b[i][0];
  w[i][0] = 1;
}
for (j = 1; j <= N; j++) {
```

```
    v[0][j] = v[0][j-1] + a[0][j];
    w[0][j] = 0;
  }

  for (i = 1; i <= M; i++)
    for (j = 1; j <= N; j++) {
      north = v[i][j-1] + a[i][j];
      east  = v[i-1][j] + b[i][j];
      if (north < east) {
        v[i][j] = north;
        w[i][j] = 0;
      }
      else {
        v[i][j] = east;
        w[i][j] = 1;
      }
    }
```



Фиг. 3.1.4. Оптимални преминавания през кръстовищата

След като масивът `w[i][j]` е зареден с необходимите стойности, намирането на маршрута може да се извърши чрез рекурсивна функция. Една такава функция `findback` е дефинирана по-долу. При извикване `findback(N,N)` тя започва да се връща назад по маршрута,

докато стигне началото. След това, при последователните обратни излизания от извикванията, отпечатва координатите на кръстовищата, през които преминава маршрутът (виж също фиг. 3.1.4).

```
void findback (int i, int j)
{ if (w[i][j] == -1) return;
  if (w[i][j] == 0) findback(i,j-1);
  else findback(i-1, j);
  cout << i << " " << j << endl;
}
```

3.2. Триъгълник от числа

По-долу е изобразена една триъгълна схема от числа:

```
    7
   3 8
  8 1 0
 2 7 4 4
4 5 2 6 5
```

Задача. *Напишете програма, която пресмята най-голямата възможна сума от числа, разположени на някои от пътищата, започващи от най-горната точка на триъгълника и завършващи в точка от основата на триъгълника. Изисква се пътищата да са такива, че при всяка от стъпките движението да се осъществява надолу — в посока по диагонала наляво или по диагонала надясно.*

За да решим задачата в общия случай, нека да запишем данните в двуиндексен масив `D[i][j]` с размери $N \times (N+1)$, `i` $= 0, 1, \ldots, N-1$, `j` $= 0, 1, \ldots, N$. По технически причини, за по-лесно боравене с индексите в програмата, този масив го дефинираме с един стълб в повече, като допълнителният нулев стълб остава зареден с нули. При конкретно зададените числа масивът представя следната таблица от 5 реда и 6 стълба ($N = 5$):

```
0 7 0 0 0 0
0 3 8 0 0 0
0 8 1 0 0 0
0 2 7 4 4 0
0 4 5 2 6 5
```

За всеки елемент D[i][j], включен в дадената триъгълна конфигурация от числа, да пресметнем най-голямата стойност R[i][j], която може да се постигне, като се движим според правилата, тръгвайки от върха. Стойностите пресмятаме последователно по редове. Очевидно имаме:

$$R[0][1] = D[0][1];$$
$$R[i][j] = \max\{D[i][j] + R[i-1][j-1],\ D[i][j] + R[i-1][j]\}$$

за всички $i = 1, \ldots, N-1$ и $j = 1, \ldots, i+1$. Остава да намерим най-голямата стойност в последния ред на масива R.

```
const N = 5;
int D[N][N+1] = {
  {0,7,0,0,0,0},
  {0,3,8,0,0,0},
  {0,8,1,0,0,0},
  {0,2,7,4,4,0},
  {0,4,5,2,6,5}
};
int R[N][N+1], P[N][N+1], max, imax;

void show (int i, int j)
{ if (i > 0) show(i-1, P[i][j]);
  printf("%i,%i - %i\n", i, j, D[i][j]);
}

void main ()
{ int i, j;
  for (i = 0; i < N; i++)
    for (j = 0; j<= N; j++) {
      R[i][j] = 0;
      P[i][j] = 0;
    }
  R[0][1]=D[0][1];
  for (i = 1; i < N; i++) {
    for (j = 1; j <= i+1; j++)
      if (D[i][j]+R[i-1][j-1] > D[i][j]+R[i-1][j]) {
        R[i][j] = D[i][j] + R[i-1][j-1];
```



```
        P[i][j] = j - 1;
      }
      else {
        R[i][j] = D[i][j] + R[i-1][j];
        P[i][j] = j;
      }
  }
  max = R[N-1][1];
  imax = 1;
  for (j = 2; j <= N; j++)
    if (max < R[N-1][j]) {
      max = R[N-1][j];
      imax = j;
    }
  show(N-1, imax);
}
```

Спомагателният масив P[i][j] служи за възстановяване на самия път, по който се достига най-голямата сума от търсения вид.

3.3. Управление на врата

Задача. ([7]) *В един ресторант се срещат N бандити. Бандитът с номер i, $i = 1, 2, \ldots, N$ идва в момента от време T_i и носи P_i долара. Входната врата на ресторанта има K състояния, различаващи се по степента на отвореност. Състоянието на вратата може да се променя с една единица за всяка една единица време, т. е. степента на отваряне на вратата или се увеличава с единица, или се намалява с единица, или остава в същото състояние. В началния момент от време вратата е затворена (състояние 0). Бандитът с номер i може да влезе в ресторанта само ако вратата е отворена специално за него, т. е. когато степента на отвореност на вратата съвпада със степента S_i на пълнота на бандита. При едновременно идване на няколко бандити с еднаква пълнота, съвпадаща със степента на отвореност на вратата, влизат всичките бандити с тази пълнота. Ако в момента на идване на един бандит състоянието на вратата не съвпада със степента на пълнота на бандита, той си отива и повече не се връща.*

Ресторантът работи в течение на време T. Напишете програма, която да покаже по какъв начин вратата да се отваря или затваря във всяка стъпка от времето, така че в ресторанта да се съберат бандити с максимално количество долари.

Входни данни за задачата са:

— целите числа N, K и T (примерни ограничения: $1 \le N \le 100$, $1 \le K \le 100$, $0 \le T \le 3000$);

— моментите от време на пристигане на бандитите $T_1, T_2, \ldots, T_N$, зададени като цели числа от интервала $[1, T]$.

— количеството долари, които носи всеки бандит: $P_1, P_2, \ldots, P_N$, зададени като цели числа (например между 0 и 300).

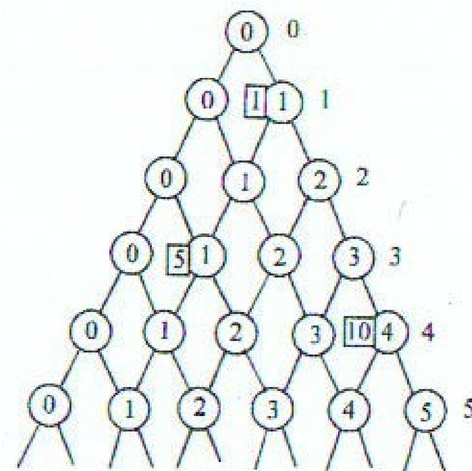
— степента на пълнота на всеки от бандитите: $S_1, S_2, \ldots, S_N$, които са цели числа такива, че $1 \le S_i \le K$ за всяко $i = 1, 2, \ldots, N$.

За да съставим алгоритъм, който пресмята максималното количество долари, които могат да се съберат у влезлите в ресторанта бандити, ще приложим метода на динамичното оптимиране. Състоянието на отвореност на вратата изобразяваме като триъгълна решетка (фиг. 3.3.1.). На фигурата са представени следните входни данни за задачата:

$$N = 3,\ K = 5,\ T = 5;$$
$$T_1 = 3,\ T_2 = 4, T_3 = 1;$$
$$P_1 = 5,\ P_2 = 10, P_3 = 1;$$
$$S_1 = 1,\ S_2 = 4, S_3 = 1.$$

Всеки връх определя степента q на отвореност в момента време t. Моментите от време са отбелязани като числа отдясно на решетката. При някои от върховете в правоъгълна рамка е дадено количеството долари които има този бандит (с пълнота q), който идва в момента време t. За всеки връх това количество долари ще наричаме тегло на върха. За върховете, които нямат съпоставено количество долари, приемаме, че теглото им е нула.

За да решим задачата, трябва да намерим път по решетката, започващ от най-горния връх и минаващ през върхове така, че сумата от теглата на върховете по пътя да е максимална. За дадения пример, изобразен на фигурата, тази максимална стойност лесно може да бъде намерена и тя е равна на 11.



Фиг. 3.3.1. Решетка, илюстрираща възможните степени за отваряне на вратата във времето

За програмната реализация ще отбележим, че не е необходимо да се пазят оценките за всеки връх. За да ги намерим в момента t, необходимо е да ползваме само стойностите им в предишния момент $t - 1$. Това се осъществява в програмата чрез двата едномерни масива `S_old` и `S_new`. В основния цикъл променливата `i` пробягва моментите от време. В тялото на този цикъл първоначално елементите на масива `S_new` се зареждат с по-голямата от двете „горни" стойности. След това програмата преглежда времената на пристигане на бандитите. Ако намери бандит, който пристига в текущия момент (`T[j]==i`), и установи, че съществува степен на отвореност на вратата, съответстваща на пълнотата на бандита (`S_new[S[j]] != MINVALUE`), програмата коригира стойността на `S_new`.

```
const N = 3, K = 5, T_end = 5;
int T[N+1] = {0,3,4,1};
int P[N+1] = {0,5,10,1};
int S[N+1] = {0,1,4,1};
```

```
int S_old[K+1], S_new[K+1];

void main ()
{ int i, j, res;
  S_old[0] = 0;
  for (j = 1; j <= K; j++) S_old[j] = MINVALUE;
  for (j = 0; j <= K; j++) S_new[j] = S_old[j];
  for (i = 1; i <= T_end; i++) {
    for (j = 1; j <= i; j++)
      S_new[j] = MAX(S_old[j-1], S_old[j]);
    for (j = 1; j <= N; j++)
      if ((T[j] == i) && (S_new[S[j]] != MINVALUE))
        S_new[S[j]] += P[j];
    for (j = 0; j <= K; j++) S_old[j] = S_new[j];
  }
  res = MINVALUE;
  for (i = 1; i <= K; i++) res = MAX(res, S_new[i]);
  cout << res;
}
```

Промените в програмата, които трябва да се направят, за да се намери как да се управлява вратата, така че в ресторанта да се съберат бандити с най-много долари, оставяме за читателя. Също за читателя оставяме модификацията на програмата, когато T_end > K.

Глава 4

Основни приложения

Методът на динамичното оптимиране, приложен към оптимизационна задача, изисква 3 етапа: съставяне на рекурентно уравнение за стойността (цената) на задачата; написване на програма за решаването на това уравнение, с която да намерим оптималната стойност; и евентуален „обратен ход" за определяне на самото решение.

В настоящата глава е направен опит да се съберат на едно място няколко основни задачи, станали вече „класически" и най-добре характеризиращи разглеждания метод.

4.1. Оптимална триангулация

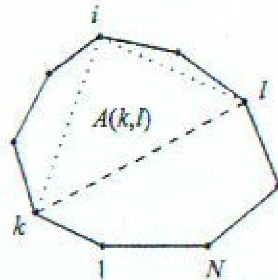
За да илюстрираме по-подробно трите етапа при прилагането на метода на динамичното оптимиране ще използваме една задача, публикувана като Задача 2 от 1998 г. в Задочния конкурс по информатика на списание Computer (брой 9 от 1998 г.) [8].

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 пропорционално на сумата от дължините на направените разрези.*

Съставете програма, която въвежда координатите на върховете на многоъгълника и посочва търсената последователност от разрези, така че ножицата да се изхаби най-малко.

Съставяне на уравнението. Нека k и l, $k < l$ са номерата на два върха от многоъгълника. Да означим с $A(k,l)$ тази част от дадения многоъгълник, която се отрязва от хордата (k,l) и не съдържа

върховете 1 и N (виж фиг. 4.1.1.). При това означение изходният многоъгълник е $A(1,N)$. При $l = k+1$ се получава „двуъгълник" $A(k,l)$, състоящ се от две съвпадащи страни.



Фиг. 4.1.1. Хорди в многоъгълник

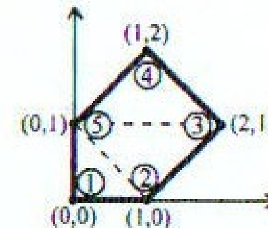
Означаваме с $f(k,l)$ изхабяването на ножицата при разрязването на $A(k,l)$. Това число е равно на сумата от дължините на диагоналите, чрез които многоъгълникът $A(k,l)$ се разрязва на триъгълници. Ще изведем рекурентна формула за $f(k,l)$. При $l = k+1$ няма какво да се разрязва и очевидно $f(k,l) = 0$. При $l = k+2$ съответният многоъгълник е триъгълник и отново имаме $f(k,l) = 0$. Да разгледаме по-интересния случай $l > k+2$. Нека i е връх между k и l, т. е. $k < i < l$. Тогава

$$f(k,l) = \min(d(k,i) + d(i,l) + f(k,i) + f(i,l)),$$

където $d(k,i)$ и $d(i,l)$ са дължините на хордите (k,i) и (i,l), а минимумът се пресмята за всички i, пробягващи върховете $k+1, \ldots, l-1$. При това трябва да се има предвид, че $d(q,q+1) = 0$ за всеки връх q, защото хордата $(q,q+1)$ всъщност е страна и не участва в разрязването.

Решаване на уравнението. а) Чрез рекурсия. Написаното рекурентно уравнение се решава по естествен начин чрез рекурсивна програма. В приложената програма (виж `example 4.1.1.`) това се извършва от функцията `f`. В програмния фрагмент масивът `p` съдържа върховете на многоъгълника, показан на фигура 4.1.2.

Функцията `d(i,j)` пресмята разстоянието между върховете `i` и `j` чрез техните координати.



Фиг. 4.1.2. Многоъгълник, използван като тестов пример за програмата

```
// example 4.1.1
const int N = 5;
double p[N][2] = {
  {0.0,0.0},
  {1.0,0.0},
  {2.0,1.0},
  {1.0,2.0},
  {0.0,1.0}
};
#define SQR(a) ((a)*(a))

double d (int i, int j)
{ if (j <= i + 1) return 0.0;
  i--; j--;
  return sqrt(SQR(p[i][0] - p[j][0]) +
              SQR(p[i][1] - p[j][1]));
}

double f (int k, int l)
{ double m, s;
  int i;
  if(l <= k + 1) return 0;
```

```
    m = MAXDOUBLE;
    for (i = k + 1; i <= l - 1; i++) {
      s = d(k, i) + d(i, l) + f(k, i) + f(i, l);
      if (s < m) m = s;
    }
    return m;
  }
```

Извикването на `f(1,N)` връща оптималната цена.

Рекурсивната реализация има недостатък поради прекалено големия брой извиквания на едни и същи пресмятания. Даже за простия пример, зададен в текста на програмата, се оказва, че извикването на `f(2,3)` се повтаря 5 пъти. Тази реализация води до експоненциално нарастване на времето за работа, което става пречка за решаването на задачата при увеличаване на размера ѝ.

б) Чрез рекурсия, но със запомняне на вече пресметнати резултати. Модифицираната функция `fm` използва допълнителен масив `t[k][l]` за запазване на вече пресметнати стойности (виж `example 4.1.2.`). Елементите на този масив трябва първоначално да бъдат заредени с отрицателната стойност `-1.0`, за да бъдат подходящо разпознавани.

```
// example 4.1.2
double fm (int k, int l)
{ double m, s;
  int i;
  if (t[k][l] != -1.0) return t[k][l];
  if (l <= k + 1) {
    t[k][l] = 0;
    return 0;
  }
  m = MAXDOUBLE;
  for (i = k + 1; i <= l - 1; i++) {
    s = d(k, i) + d(i, l) + fm(k, i) + fm(i, l);
    if (s < m) m = s;
  }
  t[k][l] = m;
  return m;
}
```


Този подход вероятно е най-лошият, защото изисква експоненциално време за работа (обръщенията към `fm` са толкова) и квадратично количество памет спрямо размера `N`. Прилагаме го, когато не можем да конструираме итеративно запълване на таблицата или когато пресмятането на някои елементи, например на `d(k,i)`, е много трудоемко (в конкретния случай обаче това не е така).

в) Чрез итеративно попълване на таблица. Конструирането на такова запълване е най-важното при метода на динамичното оптимиране. Основната трудност е предлагането на начин да запълним таблицата така, че при всяка стъпка поредният нов елемент да се пресмята лесно чрез стойностите на вече запълнени клетки. За разглежданата задача се постига използване на памет от порядъка на N^2 и време за пресмятане от порядъка на N^3, защото еднократното прилагане на рекурентната формула изисква намирането на най-малкото измежду не повече от N числа.

В приложената по-долу програма (`example 4.1.3.`) се използват два масива `t` и `b`. Елементите `t[k][l]` служат за записване на оптималната стойност на съответната подзадача, а `b[k][l]` съхранява индекса, за който се е достигнал максимумът във формулата. Това ще ни улесни при обратния ход, за да намерим самото решение във вид на списък от разрязани диагонали. Запълването на двете таблици става, като се започне от главния диагонал, вървящ от горния ляв ъгъл към долния десен, и след това се запълват по-малките, успоредни на него диагонали, намиращи се отгоре. На фигура 4.1.3. е показан запълненият масив `t[k][l]`. Стойността на решението се намира в `t[1][N]`.

j=0 j=1 ... j=N−1

	l=1	...			l=N	
k=1	0	0	0	2	3.41	...
k=2		0	0	0	2	
...			0	0	0	
				0	0	
k=N					0	

Фиг. 4.1.3. Масивът `t[k][l]`

```
// example 4.1.3
double t[N+1][N+1];
int b[N+1][N+1];

void filltab_dissect ()
{ int i, j, k, z;
  double m, s;
  for (k = 1; k <= N; k++) t[k][k] = 0.0;
  for (k = 1; k < N; k++) t[k][k+1] = 0.0;
  for (j = 2; j <= N - 1; j++)
  for (k = 1; k <= N - j; k++) {
    m = MAXDOUBLE;
    for (i = 1; i <= j - 1; i++) {
      s = d(k, k+i) + d(k+i, k+j) +
          t[k][k+i] + t[k+i][k+j];
      if (s < m) {
        m = s;
        z = k + i;
      }
    }
    t[k][k+j] = m;
    b[k][k+j] = z;
  }
}
```

Обратният ход. Извикването на дадената по-долу функция `back(1,N)` извежда разрезите, които осъществяват оптималното решение (виж `example 4.1.4.`). За конкретния пример това са диагоналите 2–5 и 3–5. Използва се единствено информацията, записана в масива `b[i][j]` от функцията `filltab_dissect()`.

```
// example 4.1.4
void back (int i, int j)
{ int l, k;
  if (j <= i + 1) return;
  k = b[i][j];
  if (i + 1 < k) cout << i << "-" << k << endl;
  if (k + 1 < j) cout << k << "-" << j << endl;
  back(i, b[i][j]);
  back(b[i][j], j);
}
```


4.2. Оптимално умножаване на няколко матрици

Матрица с размер $m \times n$ наричаме правоъгълна таблица от m реда и n стълба, чиито клетки са запълнени с по едно число. Една матрица, която има размер $m \times n$, може да се умножи с друга матрица, имаща размер $n \times k$ (ширината на таблицата, съответстваща на левия множител, трябва да е равна на височината на таблицата, съответстваща на десния множител), и като резултат се получава матрица с размер $m \times k$. Цената на едно такова умножение считаме, че е равна на произведението от трите числа: mnk. Тази цена всъщност е равна на броя умножения между числа, които трябва да се извършат при стандартния начин за умножение на матрици. Въпреки че за формулировката на задачата за оптимално умножаване на няколко матрици не е необходимо да знаем точната дефиниция за умножаване на матрици (а само цената на това умножаване), ние ще дадем тази дефиниция. А именно, по дефиниция в матрицата произведение елементът c_{ij}, който стои на i-тия ред и на j-тия стълб, се изразява чрез елементите a_{ij} и b_{ij} на двете матрици множители по следното правило (известно още и като правилото „ред по стълб“):

$$c_{ij} = \sum_{t=1}^{n} a_{it} b_{tj}$$

Умножението на матрици е асоциативно и затова произведението на няколко матрици може да се пресметне, като се използва различна последователност на умноженията. Например ако означим с A, B и C три матрици, съответно с размери 2×3, 3×4 и 4×5, тогава произведението им може да се получи по два начина:

Първи начин: умножаваме A и B, а резултата умножаваме по C, т. е. пресмятаме $(AB)C$.

Втори начин: умножаваме B и C и след това умножаваме A по този резултат, т. е. $A(BC)$.

При двата начина резултатът, разглеждан като матрица, е един и същ, но общият брой на извършените действия умножения между

числа е различен. Общата цена в първия случай е $2 \cdot 3 \cdot 4 + 2 \cdot 4 \cdot 5 = 24 + 40 = 64$, а във втория случай е $3 \cdot 4 \cdot 5 + 2 \cdot 3 \cdot 5 = 60 + 30 = 90$.

Когато броят на матриците стане по-голям, тогава значително нараства и броят на възможните комбинирания, водещи до различно поставяне на скобите в произведението от матриците.

Задача. *Дадени са s матрици $A_1, A_2, \ldots, A_s$ с дадени размери такива, че всеки две съседни матрици от редицата могат да бъдат умножени. Да се намери минимална обща цена, за която може да се осъществи умножението на всичките матрици.*

Тривиалният алгоритъм за решаване на задачата е следният: образуваме всички възможни начини за умножение на дадените матрици, пресмятаме цената при всеки един от тези начини и след това избираме този начин, при който цената е минимална. За да избягаме от този метод на пълно изчерпване на вариантите, трябва да приложим динамичното оптимиране.

Нека за нагледност си представим, че първата матрица е съпоставена с отсечката $[0, 1]$ от числовата ос, втората матрица — с отсечката $[1, 2]$ от същата ос, $\ldots$, s-тата — с отсечката $[s-1, s]$. Матриците, съответстващи на отсечките $[i-1, i]$ и $[i, i+1]$ имат такива размери, че те могат да бъдат умножени помежду си. Означаваме този общ размер (за едната от матриците това е броят на стълбовете, а за другата — броят на редовете) с $d[i]$. Така за начални данни на задачата считаме числата $d[0], d[1], \ldots, d[s]$.

Означаваме с $a(i, j)$ минималната цена за пресмятането на произведението на матриците, съответстващи на отсечката $[i, j]$ (при $0 \leq i < j \leq s$). Стойността, която търсим в задачата, е $a(0, s)$. За да съставим схема за пресмятането ѝ, най-напред забелязваме, че $a(i, i+1) = 0$, понеже в този случай се касае за една матрица и няма умножения. Рекурентната формула, която дава основа за пресмятания по метода на динамичното оптимиране, е:

$$a(i, j) = \min_{k=i+1,\ldots,j-1} \{a(i, k) + a(k, j) + d[i]d[k]d[j]\},$$

Минимумът се взема по всички възможни места, където може да се осъществи последното умножение. Произведението на матриците, съответстващи на отсечката $[i, k]$, е матрица с размер $d[i] \times d[k]$, произведението на матриците, съответстващи на отсечката $[k, j]$, е матрица с размер $d[k] \times d[j]$ и цената на пресмятането на произведението на тези две матрици е $d[i]d[k]d[j]$.

Изоморфизъм със задачата за оптимална триангулация. Разглежданата задача е много сходна (изоморфна) със задачата от предишния параграф. Формално погледнато, еднакви са формулите за съответните рекурентни зависимости. Това сходство става още по-очевидно, ако запишем матриците множители върху страните 1–2, 2–3, $\ldots$, $(s-1)$–s на многоъгълника и на всяка хорда i–j съпоставим произведението от всички матрици, съответстващи на страните на многоъгълника, които тази хорда обхваща.

4.3. Разпределяне на числа

Задача. *Дадени са n цели и положителни числа $s_1, s_2, \ldots s_n$. Тези числа са написани едно до друго, разделени със запетайки. Без да се прави разместване, числата да се разпределят в k групи. Това става, като някои от запетайките се заместват с вертикални чертички и освен това се сложи по една вертикална черта най-отпред и най-отзад. Числата, разположени между две чертички, образуват една група. Намираме сумата на числата във всяка от групите. Търси се такова групиране, че максималната от сумите, които се получават, да е възможно най-малка в сравнение с всички други начини на групиране. Приемаме, че нито една от групите не трябва да бъде празна, макар че при някои модификации на задачата е смислено да се допусне, че една или повече от групите може да бъде такава, т. е. сумата от числата в тази група да е нула.*

Пример ([3]). Да предположим, че трима служители получават нареждане да потърсят в един рафт от книги определена информация. За да се извърши работата най-добре, книгите, в които всеки ще търси, трябва да бъдат така разпределени между служителите, че по възможност всеки да получи приблизително еднакъв брой страници за преглеждане. Но за да се избегне пренареждането на книгите, разпределението трябва да стане, като за всеки служител се посочи от коя до коя книга да преглежда.

Какъв е най-добрият начин да се направи това. Ако всичките книги са с равен брой страници, например 100, и броят на книгите е

кратен на 3, например книгите са 9, решението на проблема е ясно: всеки служител получава по 300 страници:

```
| 100, 100, 100 | 100, 100, 100 | 100, 100, 100 |
```

Задачата се усложнява, когато книгите имат различен брой страници. Да предположим, че използваме същия начин за разпределяне и в случай, когато броят на страниците изглежда така:

```
| 100, 200, 300 | 400, 500, 600 | 700, 800, 900 |
```

Първият служител ще трябва да обработи само 600 страници, а за третия остават 2400. Най-справедливият начин за разпределяне е следният:

```
| 100, 200, 300, 400, 500 | 600, 700 | 800, 900 |
```

Сега най-малкото натоварване е 1300 страници, а най-голямото — 1700.

Подход за решаване. Означаваме с $M[n,k]$ оптималната стойност за задачата при даден брой n на числата и даден брой k на групите. В сила е следното съотношение, което дава връзка между споменатата стойност и стойността на задачата за предишната (с единица по-малка) стойност на k:

$$M[n,k] = \min_{i=1,\ldots,n} \max\left(M[i,k-1], \sum_{j=i+1}^{n} s_j \right).$$

Формулата изразява, че най-доброто разбиване на k части е такова, че първите му $k-1$ части са най-доброто разбиване за принадлежащите им елементи $s_1, \ldots, s_i$; а кои са тези елементи (т.е. колко да е i), се определя от стойността на i, за която се достига минимумът във формулата.

За да реализираме пресмятане по горната формула, трябва да вземем предвид и следните две гранични условия:

$$M[1,k] = s_1, \text{ за } k > 0,$$
$$M[n,1] = \sum_{i=1}^{n} s_i.$$

Според дефиницията на рекурентната зависимост тя ще пресметне величината на оптималното разделяне. Следният нерекурсивен начин ([3]) за пресмятане на стойностите $M[i,j]$ се нуждае от време, пропорционално на kn^2. При него се използва масивът `p[i]` за съхраняване на сумите $\sum_{i=1}^{k} s[i]$. Този масив е нужен, защото елементите му се използват за получаване на сумите $\sum_{l=i}^{j} s[l] = p[j] - p[l]$:

```
p[0] = 0;
for (i = 1; i <= n; i++) p[i] = p[i-1] + s[i];
for (i = 1; i <= n; i++) M[i][1] = p[i];
for (j = 1; j <= k; j++) M[1][j] = s[1];
for (i = 2; i <= n; i++)
  for (j = 2; j <= k; j++) {
    M[i][j] = INFINITY;
    for (x = 1; x < i; x++) {
      w = MAX(M[x][j-1], p[i] - p[x]);
      if (M[i][j] > w) {
        M[i][j] = w;
        D[i][j] = x;
      }
    }
  }
```

След завършване на работата на програмата стойността, пресметната в `M[n][k]`, е равна на сумата от числата в най-голямото от подразделенията, получени като оптимално решение на задачата. Масивът `D` служи за реконструиране на самите оптимални подразделения. Едновременно с пресмятането на поредната стойност на `M[i][j]` програмата записва в `D[i][j]` позицията на „разделителя“, благодарение на която се достига стойността. При входни данни

```
const n = 9;
const k = 3;
int s[n+1] = {0,1,2,3,4,5,6,7,8,9};
```

масивите `M` и `D` имат следните стойности на елементите си след завършване работата на програмата:

M[i][j]	j		
i	1	2	3
1	1	1	1
2	3	2	2
3	6	3	3
4	10	6	4
5	15	9	6
6	21	11	9
7	28	15	11
8	36	21	15
9	45	24	17

D[i][j]	j		
i	1	2	3
1	-	-	-
2	-	1	1
3	-	2	2
4	-	3	3
5	-	3	4
6	-	4	5
7	-	5	6
8	-	5	6
9	-	6	7

Зареденият масив `D[i][j]` може да се използва рекурсивно „на обратен ход" за възстановяване на подразделенията чрез следната функция:

```
void reconstruct (int n1, int n2, int k)
{ int i;
  if (k == 1) {
    for (i = n1 + 1; i <= n2; i++)
      cout << s[i] << ' ';
    cout << '\n';
  }
  else {
    reconstruct(D[n1][k-1], n1, k-1);
    for (i = n1 + 1; i <= n2; i++)
      cout << s[i] << ' ';
    cout << '\n';
  }
}
```

Тази функция трябва да се извика в главната програма като `reconstruct(D[n][k],n,k)`. Резултатът е:

```
1 2 3 4 5
6 7
8 9
```


Поради спецификата на метода на динамичното оптимиране, решавайки задачата за $n = 9$ и $k = 3$, ние всъщност сме я решили и за всички по-малки положителни стойности на тези параметри. Така при извикване на `reconstruct(D[7][2],7,2)` ще получим оптималното подразделяне за $n = 7$ и $k = 2$:

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

4.4. Най-дълга растяща подредица

Задача. *Дадена е редица от N числа: $a[0], a[1], \ldots, a[N-1]$. Да се намери нейна най-дълга растяща подредица.*

Формулирана по друг начин, задачата изглежда така: колко наймалко членове на дадената редица да изтрием, щото останалите да образуват растяща подредица. Да отбележим, че в задачата не се изисква елементите на търсената подредица да са съседни в първоначално дадената редица.

Например ([3]), ако дадената редица е

$$9, 5, 2, 8, 7, 3, 1, 6, 4,$$

то нейната най-дълга растяща подредица има 3 елемента и е или (2,3,4) или (2,3,6). Най-дългата растяща подредица, чиито елементи са съседни в първоначално дадената редица, е или (2,8), или (1,6).

Ще приложим метода на динамичното оптимиране, за да съставим програма, която решава задачата за време, пропорционално на N^2.

Означаваме с $s[i]$ дължината на най-дългата растяща подредица, която има за последен елемент $a[i]$. По този начин разглеждаме подзадача за намиране на най-дългата растяща подредица измежду $a[0], \ldots, a[i]$ и задължително завършваща с $a[i]$.

Очевидно $s[0] = 1$. По-нататък е ясно, че най-дългата растяща подредица, завършваща с $a[i+1]$, ще има дължина, с 1 по-голяма от дължината на най-дългата от растящите подредици, които завършват с $a[j]$ за някое $j \leq i$, но при условие че $a[j] < a[i+1]$:

$$s[i+1] = \max_{j:\ 0 \leq j \leq i,\ a[j] < a[i+1]} (s[j] + 1)$$

Дължината на търсената подредица е

$$\max_{j:\ 0 \le j < N} s[j],$$

защото търсената подредица трябва все пак да завърши някъде между $a[0]$ и $a[N-1]$.

Следва програмата, която решава задачата с данните от горния пример:

```
const N = 9;
int a[N] = {9,5,2,8,7,3,1,6,4};
int s[N], p[N], i0;

void find ()
{ int i, j, m, L;
  s[0] = 1;
  for (i = 0; i < N; i++) p[0] = 0;
  for (i = 1; i < N; i++) {
    m = 0;
    for (j = 0; j < i; j++)
      if (a[j] <= a[i])
        if (m < s[j]) {
          m = s[j];
          p[i] = j;
        }
    s[i] = m + 1;
  }
  L = 0;
  for (i = 0; i < N; i++)
    if (L < s[i]) {
      L = s[i];
      i0 = i;
    }
  cout << L << endl;
}

void backShow (int i)
{ if (p[i] == 0) {
```



```
    cout << a[i] << " ";
    return;
  }
  backShow(p[i]);
  cout << a[i] << " ";
}

void main ()
{ find();
  backShow(i0);
}
```

За да отпечата намерената подредица, програмата запазва за всеки елемент `a[i]` индекса `p[i]` на елемента, който се намира непосредствено преди `a[i]` в най-дългата растяща подредица, завършваща с `a[i]`.

Следната таблица показва какви стойности имат елементите на масивите `a`, `s` и `p`, след като бъдат заредени:

Редицата a	9	5	2	8	7	3	1	6	4
Дължината s	1	1	1	2	2	2	1	3	3
Предшественик p	–	–	–	2	2	3	–	6	6

4.5. Най-дълга обща подредица

Задача. *Дадени са две редици от числа* (`int`) *или от знаци* (`char`), *имащи дължина, съответно M и N елемента: $a[0], \ldots, a[M-1]$ и $b[0], \ldots, b[N-1]$. Да се намери най-дългата обща подредица.*

За да приложим метода на динамичното оптимиране, разглеждаме фамилията от подзадачи, зависеща от два параметъра, която се получава, като ограничим първата редица до i-тия ѝ елемент, а втората до j-тия. Означаваме $c[i][j]$ дължината на най-дългата обща подредица, която се съдържа в така ограничените редици, т.е. в редиците: $a[0], \ldots, a[i]$ и $b[0], \ldots, b[j]$.

Ако приемем, че вече сме пресметнали числата $c[i'][j']$ за всички индекси i' и j' такива, че $i' \le i$ и $j' \le j$, където поне едно от две-

те неравенства е строго, то по-нататъшното пресмятане може да се организира по формулата:

$$c[i][j] = \begin{cases} \min(c[i-1][j], c[i][j-1]), & \text{ако } a[i] \neq b[j], \\ c[i-1][j-1]+1, & \text{ако } a[i] = b[j]. \end{cases}$$

Тази формула изразява, че когато преминаваме от „по-малка" към „по-голяма" задача, дължината на общата подредица се увеличава с едно в случая, когато двата последни елемента $a[i]$ и $b[j]$ са равни. В противен случай се взема по-голямата дължина, която се получава при двете възможности. Стойностите на $c[i][j]$, когато поне единият от индексите е равен на 1 (т. нар. гранични стойности), са достатъчно очевидни и затова преминаваме към програмния фрагмент, който запълва масива `c[i][j]`:

```
f = 0;
for (i = 0; i < M; i++) {
  if (a[i] == b[0]) {
    d[i][0] = 1;
    f = 1;
  } else d[i][0] = 0;
  if (f) c[i][0] = 1;
  else c[i][0] = 0;
}
f = 0;
for (j = 0; j < N; j++) {
  if (a[0] == b[j]) {
    d[0][j] = 1;
    f = 1;
  } else d[0][j] = 0;
  if (f) c[0][j] = 1;
  else c[0][j] = 0;
}
for (i = 1; i < M; i++)
  for (j = 1; j < N; j++)
    if (a[i] == b[j]) {
      c[i][j] = c[i-1][j-1] + 1;
      d[i][j] = 1;
      max = c[i][j];
```



```
    }
    else
      if (c[i][j-1] > c[i-1][j])
        c[i][j] = c[i][j-1];
      else c[i][j] = c[i-1][j];
```

Масивът `d[i][j]` служи за маркиране на местата, където `a[i]==b[j]`, а променлива `max` в крайна сметка ще съдържа дължината на търсената подредица. При входни данни

```
const M = 6, N = 6;
char a[M] = "abcdef", b[N] = "bacaef";
```

запълненият чрез програмата масив `c[i][j]` има следния вид, където звездичките показват, че `d[i][j]==1` за съответните клетки:

	b	a	c	a	e	f
a	0	*1	1	*1	1	1
b	*1	1	1	1	1	1
c	1	1	*2	2	2	2
d	1	1	2	2	2	2
e	1	1	2	2	*3	3
f	1	1	2	2	3	*4

За да намерим самата най-дълга обща подредица, използваме рекурсивната функция `find`, която работи заедно със спомагателния масив `buf`.

```
char buf[M+N];

void find (int k)
{ int i, j, p;
  if (k > max)
  { for (p = 1; p <= max; p++) cout << buf[p];
    cout << endl;
    return;
  }
  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
```

```
            if ((c[i][j] == k) && (d[i][j] == 1)) {
              buf[k] = a[i];
              find(k+1);
            }
        }
```

Извикана като **find(1)**, функцията всъщност намира *всичките* най-дълги общи подредици, които за конкретния пример са две: **acef** и **bcef**.

4.6. Търсене на подниз

Дейност, често извършвана в практиката, например при текстообработка, е откриването в даден текст на знаков низ, който е „близък" до определен шаблон (образец). За да поставим задачата по-прецизно, нека да означим шаблона с P, а знаковия низ в който ще търсим — с T. Дефинираме *разстояние* между P и T като най-малкия брой промени, които са достатъчни за един подниз от T да се трансформира в P. Промените могат да бъдат от следните 3 вида (за пример вземаме шаблон $P =$'CAT'):

1. *Замяна* — ако една двойка съответни знаци, единият от T, а другият от P, се различават, тогава може да ги уеднаквим. Например ако $T =$'KAT', за да получим P, извършваме преобразованието 'KAT'→'CAT'.

2. *Вмъкване* — добавяне на един знак към T, който е в P. Например ако $T =$'CT', чрез вмъкване на A получаваме P: 'CT'→'CAT'.

3. *Изтриване* — премахване на един знак от T, който не е в P. Например ако $T =$'$CAAT$', за да получим P, изтриваме едната буква A: '$CAAT$'→'CAT'.

По-общ пример ([3]): $P =$'$abcdefghijkl$' може да се преобразува в $T =$'$bcdeffghixkl$' чрез точно 3 промени — по една от всеки от горните три вида.

Задачата за намиране на най-малкия брой трансформации от описаните видове, чрез които подниз от T се трансформира в P, т.е. намирането на разстоянието между P и T, на пръв поглед изглежда трудно. Не е ясно къде да вмъкваме и къде да изтриваме знаци. Но ако разгледаме същата задача отзад напред, можем да се запитаме

каква информация трябва да имаме, за да вземем последното решение, т.е. какво става с последните знаци във всеки от низовете. Те могат или да са еднакви, или да се заменят един с друг, или пък единият от тях да бъде изтрит или вмъкнат.

За да формализираме тези разсъждения, нека да означим с $D[i][j]$ числото, изразяващо разстоянието между сегмента от P, завършващ с i-тия му знак, и сегмента от T, завършващ с j-тия му знак. Да се опитаме да изразим $D[i][j]$ чрез някои от стойностите $D[i'][j']$ при по-малки индекси i' и j'. За целта да вземем предвид, че $D[i][j]$ е минимумът, получаващ се при трите различни начина за идване от по-късите низове:

1. Ако i-тият елемент на P е еднакъв с j-тия елемент на T, т.е. $P[i] = T[j]$, тогава $D[i][j] = D[i-1][j-1]$, а в противен случай $D[i][j] = D[i-1][j-1] + 1$. Това показва, че ако последните знаци са различни, трябва да добавим 1 към разстоянието $D[i-1][j-1]$ поради необходимостта от трансформацията „замяна".

2. $D[i][j] = D[i-1][j] + 1$, което изразява промяната на разстоянието при извършване на „вмъкване".

3. $D[i][j] = D[i][j-1] + 1$ и това изразява промяната на разстоянието при извършване на „изтриване".

От казаното дотук става ясно, че за да пресметнем $D[i][j]$ за конкретни стойности на i и j, трябва да знаем колко са трите числа $D[i-1][j-1]$, $D[i-1][j]$ и $D[i][j-1]$. Това прави възможно пресмятането да се извърши чрез таблица, в която всеки ред съответства на елемент от P, а всеки стълб — на елемент от T. Ако в началото имаме запълнени първия ред и първия стълб на таблицата, след това ред по ред може да я запълним цялата. В следния програмен фрагмент се предполага, че броят на редовете е n, а броят на стълбовете m, което означава, че шаблонът P се състои от n знака, а T — от m, броени от 1 нататък.

```
    for (i = 2; i <= n; i++)
      for (j = 2; j <= m; j++)
        D[i][j] = MIN(D[i-1][j-1] + (P[i] == T[j]),
                      D[i-1][j] + 1, D[i][j-1] + 1);
```

Идентификаторът `MIN` означава функция или макрос, който пресмята минимума от трите си аргумента.

Остава да намерим как да попълним първия ред и първия стълб на таблицата. Очевидно е, че

$$D[1][1] = \begin{cases} 0, & \text{ако } P[1] = T[1], \\ 1, & \text{в противоположния случай.} \end{cases}$$

По-нататък става ясно, че ако P е едноелементен шаблон, а T е низ от j знака, който не съдържа $P[1]$, ще трябват j трансформации ($j-1$ „изтривания" и една „замяна"), за да се преобразува P в T. Трансформациите ще бъдат с единица по-малко, ако T съдържа $P[1]$. Следващият фрагмент пресмята $D[1][j]$ за всяко $j = 1, \ldots, m$:

```
D[1][0] = 0;
for (j = 1; j <= m; j++)
  if ((T[j] == P[1]) && (D[1][j-1] == j - 1))
    D[1][j] = D[1][j-1];
  else D[1][j] = D[1][j-1] + 1;
```

Аналогично се извършва и пресмятането на първия стълб в таблицата $D[i][1]$. Ето как изглежда тази таблица за низовете, дадени по-горе като пример:

P \ T	*b*	*c*	*d*	*e*	*f*	*f*	*g*	*h*	*i*	*x*	*k*	*l*
a	1	2	3	4	5	6	7	8	9	10	11	12
b	**1**	2	3	4	5	6	7	8	9	10	11	12
c	2	**1**	2	3	4	5	6	7	8	9	10	11
d	3	2	**1**	2	3	4	5	6	7	8	9	10
e	4	3	2	**1**	2	3	4	5	6	7	8	9
f	5	4	3	2	**1**	**2**	3	4	5	6	7	8
g	6	5	4	3	2	2	**2**	3	4	5	6	7
h	7	6	5	4	3	3	3	**2**	3	4	5	6
i	8	7	6	5	4	4	4	3	**2**	3	4	5
j	9	8	7	6	5	5	5	4	3	**3**	4	5
k	10	9	8	7	6	6	6	5	4	4	**3**	4
l	11	10	9	8	7	7	7	6	5	5	4	**3**

Ако задачата е формулирана така, че се търси само разстоянието между низовете P и T, тогава нашата програма трябва да отпечата стойността `D[n][m]`. Ако задачата изисква да се намери кой подниз на T е най-близък до P, тогава най-напред лесно можем да отговорим на въпроса: на колко е равно съответното най-малко разстояние — то се получава като най-малкото от числата в последния ред на таблицата.

По-интересно е да намерим самия подниз, който е най-близък до P, а също и последователността от трансформации, с които се стига до него. Всичко това може да се възстанови от попълнената таблица. От клетката в последния ред, която има минимална стойност, тръгваме в една от трите посоки: нагоре, наляво или по диагонала „нагоре и наляво", като избираме да посетим тази от трите клетки, в която има записана равна или по-малка стойност от текущата. Посоката на всяка стъпка показва каква трансформация се прави и дали тя е съответно „вмъкване", „изтриване", „замяна" (или съвпадение на знаците). Това движение продължаваме, докато достигнем горния ляв ъгъл на таблицата и така напълно възстановим последователността на трансформациите. Да отбележим, че решението не винаги е единствено, т. е. при определени случаи са възможни няколко начина за извършването на тези трансформации — например когато T съдържа на две различни места поднизове, идентични с P.

Изпълнението на описания обратен ход изисква време от порядъка на $n + m$, докато при правия ход времето има порядък nm. Паметта, необходима при реализация с таблица, има порядъка на nm клетки, но съществено може да се намали, ако забележим, че при пресмятането на един ред от таблицата всъщност се използва само предишният. Така обемът на използваната памет приема порядък m, а ако организираме пресмятане по стълбове — този порядък е n. При решаване на задача, в която n е много по-малко от m, е ясно, че трябва да предпочетем втория начин.

4.7. Оптимални дървета за търсене

Разработването на ефективни механизми за търсене често съпътства създаването на различни приложни програми. Като основен пример може да се посочи търсенето в речник. Ясно е, че едни думи ще бъдат търсени и намирани много по-често от други. Ес-

тествена идея за подобряване на бързодействието е да се направи предварително подреждане на обектите. За целта обикновено се използва дървовидна структура, в която по-често срещаните елементи се поставят по-близо до корена.

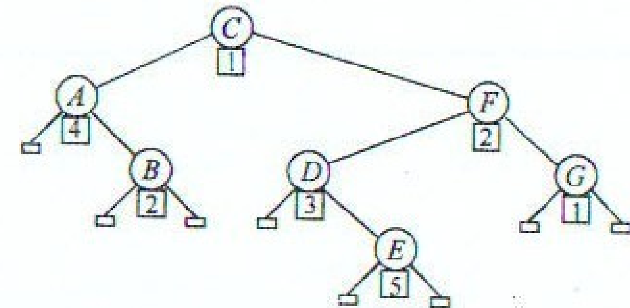
На фигура 4.7.1 е изобразено дърво за търсене, съдържащо обектите A, B, C, D, E, F и H. За всеки от тези обекти е отбелязан етикет с едно цяло положително число, показващо честотата, с която този обект се среща при подаване на заявка за търсене. За примера тези числа показват, че от общо 18 заявки средно очаквано е 4 да бъдат за A, 2 за B, 1 за C и т.н. Алгоритъмът за търсене се основава на естествената азбучна подредба на обектите. Когато търсим обект x, започваме от корена на дървото и всеки път, посещавайки някакъв връх y, ако $x \neq y$, продължаваме наляво или надясно в зависимост от това, дали $x < y$ или $x > y$.

В примера всяко търсене на A изисква преглеждане на два върха, всяко търсена на B — 3 върха и т.н. Може да се пресметне обща мярка за ефективността на търсенето в дървото, наречена *средна цена*, като се умножи честотата на всеки връх по разстоянието му от корена и всичките получени числа се съберат. Така за дадения пример средната цена е $4 \cdot 2 + 2 \cdot 3 + 1 \cdot 1 + 3 \cdot 3 + 5 \cdot 4 + 2 \cdot 2 + 1 \cdot 3 = 51$. Ясно е, че когато тази сума е по-малка, тогава процесът на търсене, разгледан в средностатистически аспект, е по-ефективен. Задачата, която възниква, е да се преструктурира двоичното дърво така, че средната цена да е минимална. Един пример за оптимално дърво за търсене ([1]), съдържащо същите обекти със същите честоти както в предишната фигура е даден на фиг. 4.7.2.

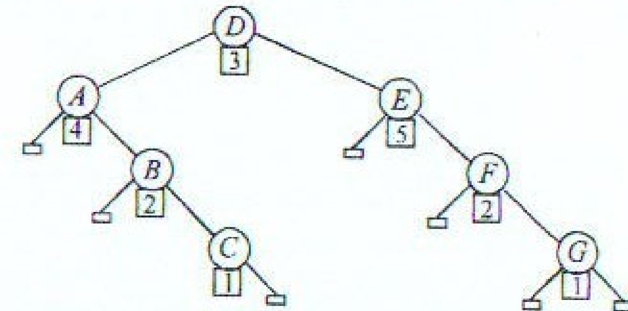
Задача. *Дадени са в сортиран ред N обекта и тегла $t[i]$ за всеки от тях. Да се подредят тези обекти като възли на двоично дърво за търсене с минимална сума $\sum_{i=1}^{N} d[i] \cdot t[i]$, където $d[i]$ е разстоянието от корена на дървото до връх с номер i.*

Идеята за прилагане на метода на динамичното оптимиране е следната: ако $p[i][j]$ е подсумата за едно поддърво (съдържащо обекти с номера от i до j), то в това поддърво избираме върха k ($i \leq k \leq j$) за корен, ако за него се достига минимумът в

$$p[i][j] = \min_{k:\ i \leq k \leq j} (p[i][k-1] + t[k] + p[k+1][j])$$



Фиг. 4.7.1. Двоично разклоняващо се дърво с честоти



Фиг. 4.7.2. Дърво за оптимално търсене

Следващият програмен фрагмент илюстрира запълването на таблицата `p[i][j]`. Спомагателният масив `r[i][j]` служи за отбелязване на индекса, за който се достига минимумът във формулата за пресмятането на `p[i][j]`. Във функцията `tree` масивът `r` се използва за отпечатване на дървовидната структура. Всяко следващо ниво от дървото се извежда подравнено след толкова на брой шпации, на колкото е равен номерът му.

```
const N = 7;
int t[N] = {4,2,1,3,5,2,1};
int p[N][N], r[N][N];
```

```
void solve ()
{ int i, j, k, l, m;
  for (k = 0; k < N; k++) p[k][k] = t[k];
  for (l = 1; l < N; l++)
    for (i = 0; i < N - l; i++) {
      j = i + l;
      m = MAXINT;
      for (k = i; k <= j; k++) {
        int v1 = (k - 1 < 0) ? 0 : p[i][k-1];
        int v2 = (k + 1 >= N) ? 0 : p[k+1][j];
        int v = v1 + v2;
        if (m > v) {
          m = v;
          r[i][j] = k;
        }
      }
      for (k = i; k <= j; k++) m += t[k];
      p[i][j] = m;
    }
}

void tree (int i, int j, int lev)
{ if (i > j) return;
  for (int l = 0; l < lev; l++) cout << " ";
  if (i == j) {
    cout << i << endl;
    return;
  }
  int k = r[i][j];
  cout << k << endl;
  tree(i, k-1, lev+1);
  tree(k+1, j, lev+1);
}

void main ()
{ solve();
  cout << p[0][N-1] << endl;
```



```
  tree(0, N-1, 1);
}
```

Оптималната стойност на цената се получава в `p[0][N-1]` и при конкретните данни, заложени в програмата, тя е 41. По-долу е даден изходът, генериран от функцията `tree` за тези конкретни данни. Числата от 0 до 6 съответстват на обектите $A, \dots, G$ от фиг. 4.7.2:

```
3
 0
  1
   2
 4
  5
   6
```

4.8. Най-къси пътища в графи

В заключителния параграф от тази глава накратко ще се спрем върху задачи от алгоритмичната теория на графите. Тук няма да излагаме подробно тази теория, защото тя излиза далеч от рамките на това учебно пособие. Нашата цел е да акцентираме вниманието върху приложенията на метода на динамичното оптимиране ([5]).

Да разгледаме n града, номерирани с числата от 1 до n. За всяка двойка градове с номера i и j е известна цената на прекия превоз от единия град до другия (например цената на самолетния билет). Тези цени са зададени като елементи на матрицата $a[i][j]$. Счита се, че от всеки един град до всеки друг има директен превоз със съответната цена. При практически приложения в случаите, когато такъв директен транспорт не съществува (т. е. за да се отиде в един град, тръгвайки от друг, непременно трябва да се мине през още един или няколко града), елементите на матрицата $a[i][j]$ се зареждат с достатъчно голямо число („безкрайна" цена).

Винаги считаме, че $a[i][i] = 0$ за всяко $i = 1, 2, \dots, n$, но изобщо казано, стойността $a[i][j]$ може да е различна от $a[j][i]$.

Най-малката стойност на пътуването от град i до град j се дефинира като най-малката сума на цените за пътуванията между тези два града при използване на всевъзможните маршрути с междинни градове такива, че да се стигне от i до j. Тази най-малка стой-

ност очевидно не надминава $a[i][j]$ (но може да е строго по-малка от $a[i][j]$).

Някои от цените $a[i][j]$ може да са зададени и като отрицателни числа. Но когато не съществуват затворени маршрути, за които сумата от цената по отсечките им е отрицателна, тогава винаги за всяка двойка градове съществува маршрут с минимална цена. Това твърдение се обосновава с факта, че маршрут с брой на ребрата по-голям от n ще съдържа цикъл, и затова въпросният минимум трябва да се търси измежду маршрутите с брой на ребрата, не по-голям от n, а те са краен брой.

При следващото изложение в този параграф ще предполагаме, че данните на задачите са такива, че отсъстват цикли с отрицателни суми.

Разгледаната съвкупност от градове и свързващи ги пътища се обобщава абстрактно с математическото понятие **граф**. Предполагаме, че читателят има известна представа за графите и за елементарните им свойства, въпреки че за разбирането на следващия материал това няма да е необходимо.

Задача. *Да се намери най-малката стойност за превоз от града с номер 1 до всеки друг град.*

От отбелязаното по-горе следва тривиален метод за решаване на задачата — просто се генерират всички възможни маршрути и се избира този, който е с най-малка цена. Този метод обаче е силно неефективен, защото броят на вариантите, подлежащи на проверка, нараства експоненциално с увеличаването на n. Смислено е да търсим друг подход, като поставим изискване решението да се намира с извършване на полиномиален спрямо n брой операции. Ще покажем, че това може да стане с брой от порядъка на n^3.

Алгоритъм на Форд–Белман. Да означим с $M(1,s,k)$ най-малката цена за пътуване от град 1 до град s, като се използват по-малко от k прекачвания. Тогава е изпълнено равенството:

$$M(1,s,k+1) = \min\left(M(1,s,k), \min_{i=1,\ldots,n}\left(M(1,i,k) + a[i][s]\right)\right).$$

Отговорът на задачата се получава, когато се пресметне $M(1,i,n)$ за всяко $i = 1,\ldots,n$.

Публикуваме програмен фрагмент, реализиращ описания дотук метод на динамичното оптимиране, който, отнесен към разглежданата задача, е известен и като метод на Форд и Белман за намиране на най-къси пътища в граф.

```
k = 1;
for (i = 1; i <= n; i++) x[i] = a[1][i];
// x[i] == M(1, i, k)
while (k != n) {
  for (s = 1; s <= n; s++) {
    y[s] = x[s];
    for (i = 1; i <= n; i++)
      if (y[s] > x[i] + a[i][s])
        y[s] = x[i] + a[i][s];
      // y[s] == M(1, s, k+1)
  }
  for (i = 1; i <= n; i++) x[s] = y[s];
  k++;
}
```

Може да се покаже, че програмата ще продължи да работи правилно и без да се използва масив `y`, а всички промени да се извършват направо в самия масив `x`:

```
k = 1;
for (i = 1; i <= n; i++) x[i] = a[1][i];
while (k != n) {
  for (s = 1; s <= n; s++)
    for (i = 1; i <= n; i++)
      if (x[s] > x[i] + a[i][s])
        x[s] = x[i] + a[i][s];
  k++;
}
```

Алгоритъм на Флойд. Този алгоритъм намира най-ниската цена за преминаване от i до j едновременно за всички i и j, като използва брой на аритметичните операции от порядъка на n^3.

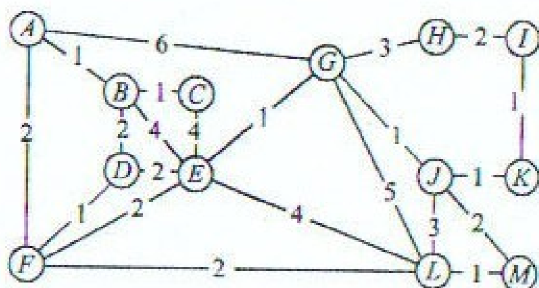
Да означим с $A(i,j,k)$, за $k = 0,\ldots,n$, най-ниската цена на маршрут от i до j, като за пунктове, в които е разрешено да се правят

прекачвания, могат да се ползват градове с номера, не по-големи от k. Тогава

$$A(i, j, 0) = a[i][j],$$

$$A(i, j, k + 1) = \min\left(A(i, j, k),\ A(i, k + 1, k) + A(k + 1, j, k)\right).$$

Двата варианта съответстват на неизползването и използването на пункта $k + 1$ за прекачване. Да отбележим, че няма смисъл да бъдем там повече от веднъж.



Фиг. 4.8.1. Пример за неориентиран граф с положителни стойности по ребрата

Алгоритъм на Дийкстра. Прилага се, при условие че всички дадени цени са неотрицателни. Тогава чрез него се намира най-ниската цена за превоз от 1 до i за всички $i = 1, \ldots, n$, като необходимото време за работа на алгоритъма (т. е. броят аритметични операции) е от порядъка на n^2.

Описание на алгоритъма: За всеки град се използва допълнителна булева променлива, чрез която в процеса на работата една част от градовете се маркират като *отбелязани*. В началото отбелязан е само градът 1, а в края на процеса — такива стават всичките градове.

- За всеки отбелязан град i се записва най-евтината цена $v[i]$ за маршрут, съединяващ $1 \to i$; освен това е осигурено, че този минимум се достига за маршрут, минаващ само през отбелязани градове.

- За всеки неотбелязан град i се записва най-евтината цена $v[i]$, получена измежду всички маршрути $1 \to i$, при които като междинни градове се използват само отбелязани градове.

Множеството на отбелязаните градове се разширява постепенно по време на работа на алгоритъма. Това става въз основа на следното свойство: Ако измежду неотбелязаните градове вземем този град i, за който съответното му число $v[i]$ е минимално, то това число е равно и на най-евтината цена за преминаването $1 \to i$. За да докажем, че това е вярно, допускаме, че съществува по-къс (т. е. с по-ниска цена) път от 1 до i. Да разгледаме първия неотбелязан град i' върху този път. Ясно е, че цената на маршрута $1 \to i'$ е по-голяма от цената на $1 \to i$, и следователно (понеже всички цени са неотрицателни) още по-голяма е цената на прехода $1 \to i' \to i$. Противоречие.

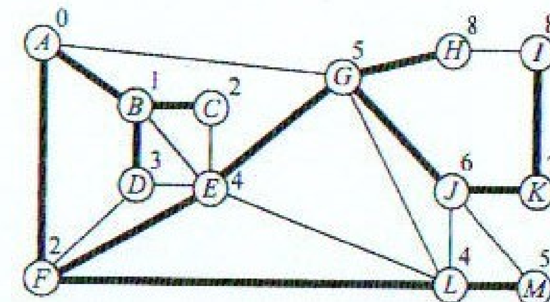
В приложената програма данните ([1]) са взети от фиг. 4.8.1 и са заредени в масива `a[i][j]`. Масивите `b[i]`, `v[i]` и `p[i]` служат съответно за маркиране на отбелязан връх `i`, за съхраняване на дължината на най-късия маршрут от 1 до `i` и за записване на предходния на `i` връх в този най-къс маршрут. Да отбележим, че върховете с имена $A, B, \ldots, M$ от чертежа съответстват на номера `0, 1, ..., N-1` в програмата.

```
const MAXINT = 9999;
const N = 13;
int a[N][N], b[N], v[N], p[N];

void init ()
{ int i, j;
  for (i = 0; i < N; i++) {
    v[i] = 0;
    b[i] = 0;
  }
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) a[i][j] = MAXINT;
// A B C D E F G H I J  K  L  M
// 0 1 2 3 4 5 6 7 8 9 10 11 12
  a[0][1] = 1; a[0][5] = 2; a[0][6] = 6;
  a[1][2] = 1; a[1][3] = 2; a[1][4] = 4;
  a[2][4] = 4;
```

```
    a[3][4] = 2; a[3][5] = 1;
    a[4][5] = 2; a[4][6] = 1; a[4][11] = 4;
    a[5][11] = 2;
    a[6][7] = 3; a[6][9] = 1; a[6][11] = 5;
    a[7][8] = 2;
    a[8][10] = 1;
    a[9][10] = 1; a[9][11] = 3; a[9][12] = 2;
    a[11][12] = 1;
//
    for (i = 0; i < N; i++)
      for (j = i + 1; j < N; j++) a[j][i] = a[i][j];
    b[0] = 1;
    p[0] = 0;
}

void show ()
{ int i;
  for (i = 0; i < N; i++)
    cout<<' '<<' '<< char(i+'A');
  cout << '\n';
  for (i = 0; i < N; i++) cout << setw(3) << v[i];
  cout << '\n';
  for (i = 0; i < N; i++)
    cout<<' '<<' '<< char(p[i]+'A');
  cout << '\n';
}

void solve ()
{ int i, j, i0, j0, m;
  do {
    m = MAXINT;
    for (i = 0; i < N; i++) if (!b[i])
      for (j = 0; j < N; j++) if (b[j])
        if (m > a[i][j] + v[j]) {
          m = a[i][j] + v[j];
          i0 = i;
          j0 = j;
        }
```



```
    if (m < MAXINT) {
      b[i0] = 1;
      v[i0] = m;
      p[i0] = j0;
    }
  } while (m < MAXINT);
}

void main ()
{ init();
  solve();
  show();
}
```



Фиг. 4.8.2. Дървото на най-късите пътища от A до всеки друг връх. При всеки от върховете е изобразено число, показващо дължината на най-късия път от A до този връх

Изходът на програмата, осъществен чрез функцията `show()`, показва за всеки връх `i` дължината `v[i]` на най-късия маршрут до началния връх A и освен това изобразява непосредствения му съсед `p[i]` в този най-къс маршрут по посока към началото. Информацията, съдържаща се в двойките `(i, p[i])`, позволява да се построи дървото от най-къси пътища, изобразено на фиг. 4.8.2.

```
  A  B  C  D  E  F  G  H  I  J  K  L  M
```

```
0 1 2 3 4 2 5 8 8 6 7 4 5
A A B B F A E G K G J F L
```

Програмната реализация изисква време за работа, пропорционално на N^3 поради двойния цикъл `for...for` във функцията `solve()`, който се изпълнява още N пъти чрез цикъла `do...while`. Оставяме за читателя да подобри програмата, като се стреми да постигне време от порядъка на N^2.

Глава 5

Задачи от дискретното оптимиране

В тази глава са разгледани две важни задачи от областта на целочислената (дискретна) оптимизация — задачата за раницата и задачата за търговския пътник. При тяхното решаване методът на динамичното оптимиране показва своите предимства.

5.1. Задача за раницата

Подход чрез жадни (greedy) алгоритми. Повечето алгоритми за решаване на оптимизационни задачи минават през последователност от стъпки, във всяка от които се избира измежду няколко възможности. Ако при всеки от тези избори търсим максималната изгода за момента, то даже обичайният човешки опит показва, че така не винаги може да се стигне до оптималното решение на първоначалната задача. Нали за да достигнем Луната, не е необходимо първо да се качим на някое дърво, за да намалим разстоянието?

Но все пак в теорията и практиката често се използва принцип, според който на всяка стъпка се избира това, което е най-доброто за момента. Алгоритмите, реализиращи тази идея, се наричат greedy (грийди) — английски термин, превеждан на български като „жаден“ или „алчен“. Изобщо казано, такива алгоритми не винаги намират оптимално решение, но прилагането им понякога води до решения, които са доста близки до оптималните. При определени задачи полученото решение наистина е оптимално и гаранция за това е наличието на съответно математическо доказателство.

Ще приведем формулировките на две близки разновидности на задачата за раницата, за едната от които greedy-подходът винаги намира оптимално решение, докато за другата това не може да се твърди.

Задача за „0–1 раница“: *Има n предмета, i-тият от тях струва c_i лева и тежи a_i килограма. Дадените числа са цели и положителни. Разполагаме с раница, в която могат да се сложат предметите, но не винаги всичките заедно, защото капацитетът на раницата не позволява общото тегло на поставените в нея предмети да надхвърли дадено цяло число M. Кои от предметите да изберем за раницата, така че сумата от стойностите им да е възможно най-голяма?*

В названието на задачата присъства „0–1“, защото всеки от предметите може да бъде взет или оставен; не се разрешава вземането на части от един предмет. Последното обаче е позволено при следната

Задача за „дробна раница“: *Постановката на задачата е същата като при „0–1 раница“, но сега е разрешено да се вземе и част от предмет. Така за всеки предмет в решението на задачата, вместо да посочваме 0 или 1, можем да предложим произволно дробно число от интервала $[0, 1]$, включително 0 или 1. Това число ще покаже каква част от предмета да бъде взета.*

При моделиране на реални ситуации първата задача за раницата се прилага при неделими предмети, а втората — когато е възможно предметите да се вземат на части, например чрез разрязване.

Да разгледаме greedy–стратегия, при която, за да запълним раницата, вземаме предметите по ред, определен от отношението c_i/a_i, т. е. според относителната стойност на предмета спрямо единица тегло. Първо избираме това, което има най-голяма относителна ценност. Когато наличността на тези предмети се изчерпи, продължаваме със следващите по относителна стойност предмети и т. н., докато раницата се запълни. Ясно е, че на последната стъпка може да се наложи да вземем само част от предмета, а не целия. Разбира се, ако задачата е „0–1 раница“, на тази последна стъпка няма да можем да вземем нецял предмет и раницата ще остане недокрай запълнена. За така описания greedy–алгоритъм се вижда, че сложността му се определя от сложността на процедурата за сортиране на относителните цени на предметите.

Известно е, че за задачата „дробна раница“ току-що описаният алгоритъм работи винаги правилно. Доказателството може да се намери в литературата (виж например [1] или [4]). Но за „0–1 раница“ не е така. Това се потвърждава от пример, при който са дадени 3 предмета с тегла и цени съгласно следната таблица:

Номер на предмет	1	2	3
Тегло	1	2	3
Цена	6	10	12
Относителна цена	6	5	4

и капацитет на раницата 5.

При прилагане на greedy–стратегията първо вземаме предмет 1, защото той е с най-голяма относителна цена. След него вземаме предмет 2, при което раницата вече е вместила тегло 3 и не остава място за третия предмет. Така стойността, събрана в раницата, е 16 единици. Но очевидно това не е оптималното решение. Вижда се, че ако вземем втория и третия предмет, общата им стойност ще е 22 единици, като при това сумата от теглата им е 5 и не надвишава капацитета на раницата.

Ако разглеждаме задачата „дробна раница“, greedy–стратегията ще ни доведе до решение да вземем целия първи и целия втори предмет и 2/3 части от третия предмет, което е и оптималното за тази задача с цена 24.

Подход чрез динамично оптимиране. Може да постигнем усъвършенстване на разгледания greedy-метод, като опитаме да направим зависими един от друг изборите, извършвани в отделните стъпки. Тази идея води до прилагания в предишните глави на настоящето ръководство метод на динамичното оптимиране. За него основополагащ принцип е намирането на оптимална субструктура на задачата, което ще рече да опишем как (ако е възможно) оптималното решение на задачата съдържа оптималното решение на подзадача. И по-общо, да опишем оптималното решение на всяка подзадача като получено от оптималното решение на „по-малка“ подзадача.

Като пример при задачата за „0–1 раница“ да разгледаме множество от нейни подзадачи $Z(k, t)$, $(k = 0, \ldots, n;\ t = 0, \ldots, M)$, които зависят от двата параметъра k и t. Всяка от подзадачите има същата формулировка като основната задача, но с тази разлика, че сега се използват само първите k предмета измежду общо дадените n и освен това капацитетът на раницата е променливото t вместо фиксираното M. Да означим със $z(k, t)$ оптималната цена на съответната подзадача. За да сметнем $z(n, M)$, допускаме, че знаем колко е $z(k, t)$ за всяко $k < n$ и за всяко $t \leq M$, т. е. допускаме, че е известно числото $z(n-1, t)$. Тогава, за да преминем към $z(n, M)$, разсъждаваме

така:

В оптималното решение на първоначално формулираната задача $Z(n,M)$ предметът с последния номер n може да не участва или да участва. В първия случай $z(n,M) = z(n-1,M)$, а във втория $z(n,M) = c_n + z(n-1, M-a_n)$. Обяснението е, че ако предметът n не участва, то оптималното решение е същото, както на $Z(n-1,M)$, а ако предметът n участва, то оптималната цена се получава от неговата цена, сумирана с оптималната цена на подзадачата $Z(n-1, M-a_n)$. Двата случая можем да обобщим с една формула

$$z(n,M) = \max\{z(n-1,M), c_n + z(n-1, M-a_n)\}$$

и по аналогия да напишем, че е в сила съотношението

$$z(k,t) = \max\{z(k-1,t), c_k + z(k-1, t-a_k)\},$$

за $k \leq n$ и $t \leq M$. Разбира се, когато се окаже, че максимумът се достига за втория си аргумент, трябва допълнително да проверим, дали добавянето на предмета с номер k няма да надвиши вместимостта на раницата. Ако това се случи, тогава „максимума“ трябва да го считаме равен на първия си аргумент.

Така решавайки задачата $Z(n,M)$ по метода на динамичното оптимиране, трябва предварително да решим всички задачи $Z(k,t)$ за $k \leq n$ и $t \leq M$. За всяка от тях трябва да намерим оптималната цена $z(k,t)$ и списъка на предметите, от които е формирана тази цена. От гледна точка на по-добра програмна реализация е възможно вместо споменатите списъци да се пазят само сумите от теглата на предметите, решаващи всяка от задачите $Z(k,t)$. Тогава обаче е нужна допълнителна работа (наричана обратен ход) за намиране на списъка за първоначалната задача.

Предлагаме фрагмент от програма за решаване на задачата по метода на динамичното оптимиране (виж `example 5.1.1.`). Използват се 3 масива, `z`, `w` и `y`, съответно за записване на оптималната цена, сумата от теглото на участващите предмети и за индикация, кои предмети участват при всяка от подзадачите. Програмата запълва елементите на тези масиви, спазвайки подходящ ред, така че вече пресметнати стойности да се ползват за пресмятане на следващи. Първоначално масивите трябва да са запълнени с нули. Входните данни за задачата се определят от константите `n` и `M`, както и

от масивите `c` и `a`. Въведените числени стойности са същите като в дадения по-горе пример.

```
// example 5.1.1
const n = 3;
const M = 5;
int c[n+1] = {0,6,10,12};
int a[n+1] = {0,1, 2, 3};
int z[n+1][M+1];
int w[n+1][M+1];
int y[n+1][M+1][n+1];

void filltab_knapsack ()
{ int k, t, i, v1, v2, t0;
  k = 1;
  for (t = 0; t <= M; t++)
    z[k][t] = (a[k] <= t) ? c[k] : 0;
  for (t = 0; t <= M; t++)
    w[k][t] = (a[k] <= t) ? a[k] : 0;
  for (t = 0; t <= M; t++)
    y[k][t][k] = (a[k] <= t) ? 1 : 0;

  for (k = 2; k <= n; k++)
    for (t = 0; t <= M; t++) {
      v1 = z[k-1][t];
      t0 = t - a[k];
      if (t0 < 0) t0 = 0;
      v2 = c[k] + z[k-1][t0];
      if ((v1 < v2) && (a[k] + w[k-1][t0] <= t)) {
        z[k][t] = v2;
        w[k][t] = w[k-1][t0] + a[k];
        for (i = 1; i < k; i++)
          y[k][t][i] = y[k-1][t0][i];
        y[k][t][k]=1;
      }
      else {
        z[k][t] = v1;
        w[k][t] = w[k-1][t];
```

```
        for (i = 1; i < k; i++)
          y[k][t][i] = y[k-1][t][i];
      }
    }
  }
```

След извикване на `filltab_knapsack()` в `z[n][M]` се получава пресметнатата оптимална цена, в `w[n][M]` — общото тегло на взетите предмети, а елементите `y[n][M][i]`, `i=1,...,n`, са 0 или 1 според това, дали `i`-тият предмет е сложен в раницата или не е.

Ето как ще изглеждат елементите на масивите `z`, `w` и `y`. Те ще описват решенията на всички подзадачи — за брой предмети k и капацитет на раницата t:

`z[k][t]`

k \ t	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16	18	22

`w[k][t]`

k \ t	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	1	1	1	1
2	0	1	2	3	3	3
3	0	1	2	3	4	5

`y[k][t]`

k \ t	0	1	2	3	4	5
0	000	000	000	000	000	000
1	000	100	100	100	100	100
2	000	100	010	110	110	110
3	000	100	010	110	101	011

Считаме, че читателят може да усъвършенства предложената програма, като се опита да намали броя на използваните масиви и даже да ги сведе до едномерни. Това може да стане чрез допълване на програмата с фрагмент за извършване на обратен ход.

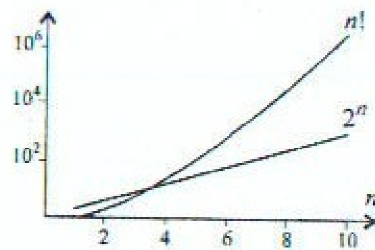
Дали за конкретно въведени стойностите на n и M е необходимо да се решават всичките подзадачи, т. е. дали трябва да се пресметнат и запълнят всичките елементи на масивите? Предлагаме на читателя да изследва този въпрос.

Публикуваната програма, основана на метода на динамичното оптимиране за „0–1 раница", извършва аритметични операции, броят на които има порядък, изразяващ се чрез полином от n и M. По-точно, както се вижда от циклите в текста на програмата, този порядък е $M \cdot n^2$. Същото е вярно и за порядъка на обема на използваната памет. Ако се обърнем към метод за изчерпващо търсене, например чрез генериране на всички n-торки от 0 и 1, тогава броят на операциите ще расте експоненциално като 2^n. Оттук проличава голямото предимство на динамичното оптимиране. Благодарение на този метод задачата за раницата разполага с полиномиално ефективен алгоритъм. Известен недостатък на динамичното оптимиране е нуждата от заделянето на повече памет, отколкото би изисквала една по-неефективна програма за изчерпващо търсене.

5.2. Задача за търговския пътник

Разгледаните по-горе задачи за раницата са пример за избиране измежду подмножества на дадено множество. По-сложен е проблемът, когато трябва да се избира измежду пермутации. В този случай, при дадени n елемента, броят на възможностите е $n!$, докато броят на множествата, които могат да се образуват от същите елементи, е 2^n. Както е известно, при големи n стойността на $n!$ расте много по-бързо от 2^n (виж фигура 5.2.1.).

Задачата за търговския пътник. *Тя се формулира за граф, върху чиито ребра са отбелязани цени $c_{\{i,j\}}$. Ще разглеждаме маршрути, които съдържат всичките върхове на графа точно по веднъж. Цената („дължината") на един такъв маршрут е сумата от цените на участващите в него ребра. Целта на търговския пътник е да намери маршрут от описания вид с минимална цена.*



Фиг. 5.2.1. Функцията $n!$ расте по-бързо от 2^n

Очевидна е важността на приложенията, в които може да се използва тази задача. Един от неефективните начини за решаването ѝ е да се проверят всички възможни маршрути чрез генериране на всевъзможните пермутации от върховете на графа. Ние ще приложим по-ефективния подход на динамичното оптимиране.

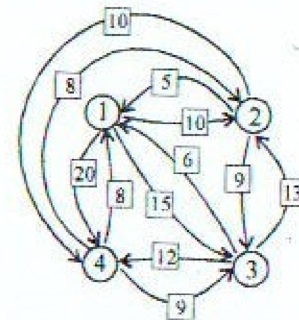
Да означим с V съвкупността от всичките върхове на дадения граф и да приемем, че броят им е $n+1$. Без ограничение на общността ще считаме, че търсеният маршрут започва и завършва във връх 1. Всеки такъв маршрут може да се разглежда като състоящ се от две части: реброто $\langle 1, k\rangle$ за някое k от $V-\{1\}$ и пътя от k до 1. Втората от тези части съдържа всеки от върховете от $V-\{1,k\}$ точно веднъж. Очевидно в сила е следният принцип за оптималност: щом маршрутът е оптимален, то разглежданият път от k до 1 е най-късият път, минаващ през всички върхове от множеството $V-\{1,k\}$. Оттук извеждаме основното уравнение на динамичното оптимиране за тази задача:

Да означим с $g(i,S)$ дължината на най-късия път, започващ от връх i, минаващ през всички върхове от S и завършващ във връх 1. Тогава $g(1, V-\{1\})$ ще е дължината на оптималния маршрут за търговския пътник и очевидно:

$$g(1, V-\{1\}) = \min_{2\le k\le n}\{c_{\{1,k\}} + g(k, V-\{1,k\})\}.$$

Обобщавайки, получаваме за всяко $i \notin S$:

$$g(i,S) = \min_{j\in S}\{c_{\{i,j\}} + g(j, S-\{j\})\}.$$



Фиг. 5.2.2. Пример за задача за търговски пътник

При фиксирано i и S стойността на $g(i,S)$ може да се пресметне, ако знаем $g(j, S-\{j\})$ за всеки избор на j от S.

Ако S е празното множество, то $g(i,S) = c_{\{i,1\}}$, $1 \le i \le n$. След това по формулата може да пресметнем $g(i,S)$ за всички множества S, състоящи се от един елемент. Следва да пресметнем $g(i,S)$ за всички множества S с $|S| = 2$, т. е. с брой елементи, равен на 2, и т. н., докато стигнем до стъпката, в която $|S| = n-1$.

Пример (взет от [4]). Да разгледаме следния граф от 4 върха с ориентирани ребра и със зададени цени върху тях (виж фигура 5.2.2.):

От фигурата образуваме следната таблица за цените $c_{\{i,j\}}$ между двойките съседни върхове:

i \ j	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Решаваме задачата за търговския пътник:

При $|S| = 0$:

$$g(2,S) = 5; g(3,S) = 6; g(4,S) = 8.$$

За всички S, такива, че $|S| = 1$, използваме формулата:

$$g(2,\{3\}) = c_{\{2,3\}} + g(3,\{\}) = 15;$$
$$g(3,\{2\}) = 18;$$
$$g(4,\{2\}) = 13;$$
$$g(2,\{4\}) = 18;$$
$$g(3,\{4\}) = 20;$$
$$g(4,\{3\}) = 15.$$

Пресмятаме $g(i,S)$ за тези S, за които $|S| = 2$, $i \neq 1$, $1 \notin S$ и $i \notin S$:

$$g(2,\{3,4\}) = \min\{c_{\{2,3\}} + g(3,\{4\}),\ c_{\{2,4\}} + g(4,\{3\})\} = 25;$$
$$g(3,\{2,4\}) = \min\{c_{\{3,2\}} + g(2,\{4\}),\ c_{\{3,4\}} + g(4,\{2\})\} = 25;$$
$$g(4,\{2,3\}) = \min\{c_{\{4,2\}} + g(2,\{3\}),\ c_{\{4,3\}} + g(3,\{2\})\} = 23.$$

Накрая

$$g(1,\{2,3,4\}) = \min\{\ c_{\{1,2\}} + g(2,\{3,4\}),\ c_{\{1,3\}} + g(3,\{2,4\}),\ c_{\{1,4\}} + g(4,\{2,3\})\ \} = 35.$$

Оптималният маршрут на търговския пътник за дадения граф има дължина (цена) 35. Самият маршрут щяхме да можем да посочим, ако при намирането на всяка от стойностите $g(i,S)$ си бяхме направили труда едновременно да запазим и стойността на j, която минимизира дясната страна на формулата. За да направим това, нека да означим тези стойности с $J(i,S)$. Тогава $J(1,\{2,3,4\}) = 2$, т. е. маршрутът, започвайки от връх 1, минава през връх 2. Останалата част се намира от $g(2,\{3,4\})$. Имаме $J(2,\{3,4\}) = 4$. Така следващият връх е 4 и целият маршрут има вида: 1,2,4,3,1.

За да пресметнем порядъка на броя на операциите, които трябва да извърши описаният алгоритъм, ще отбележим, че броят на различните подмножества S, $|S| = k$, които не съдържат 1 и i, съгласно известна формула от комбинаториката се изразява чрез биномните коефициенти

$$\binom{n-2}{k} = \frac{(n-2)!}{k!(n-2-k)!}.$$

Понеже за всяко S има $n-1$ избора за i, то общият брой на пресметнатите стойности на $g(i,S)$ е равен на сумата

$$\sum_{k=0}^{n-2}(n-1)\binom{n-2}{k}.$$

Както е известно, такава сума от биномни коефициенти е равна на израз, чийто порядък е 2^n. Следователно, толкова е и времевата сложност на разглеждания алгоритъм.

Това е по-добро от изчерпването на всичките $n!$ пермутации на върховете, определящи всевъзможните маршрути, но за разлика от задачата за раницата тук сложността не е полиномиална, за да можем да наречем този метод силно ефективен. Оценката на необходимата памет сега е $n \cdot 2^n$, което е твърде голямо число, даже и за скромни стойности на n.

Графът в разгледания пример беше пълен, т. е. между всеки два върха бяха зададени ребра в двете посоки. При решаване на задача за непълен граф трябва за всеки два върха, между които няма ребро, да добавим изкуствено такова с много висока цена, равна по стойност на машинната безкрайност. Така, ако след решаване на задачата се окаже, че намерената обща цена е това голямо число, заключаваме, че задачата няма решение. По този начин можем да решим и един частен случай на задачата за търговския пътник, а именно задачата за намиране на хамилтънов цикъл. Известно е (виж например лекциите по програмиране от бр. 10/1998 г. на списание Computer ([9])), че хамилтънов цикъл може да се намери чрез метод за търсене с връщане назад (backtracking). Според казаното тук същата задача с голям успех може да бъде атакувана чрез метода на динамичното оптимиране.

Публикуваме рекурсивна реализация на описания алгоритъм (виж `example 5.2.1.`). Извикването на функцията `g(1,s)` пресмята цената на задачата. Множествата, необходими за работата на алгоритъма, се задават чрез типа `set` във вид на масив от символи. Това улеснява визуализацията им. Входните данни са записани в масива `d[i][j]`.

```
// example 5.2.1
const n = 4;
```

```
typedef char set[n+2];
set s;

int d[n][n] = {
  {0,10,15,20},
  {5, 0, 9,10},
  {6,13, 0,12},
  {8, 8, 9, 0}
};

int g (int v, set s)
{ int m, t;
  m = MAXINT;
  for (int i = 1;i <= n; i++)
    if (s[i] != '0') {
      s[i] = '0';
      t = d[v-1][i-1] + g(i, s);
      s[i] = '0' + i;
      if (t < m) m = t;
    }
  if (m == MAXINT) m = d[v-1][0];
  return m;
}

void main ()
{ s[0] = '0';
  s[1] = '0';
  for (int i = 2; i <= n; i++) s[i] = '0' + i;
  s[n+1] = '\0';
  cout << g(1, s);
}
```

Реализация чрез итеративно запълване на таблица (т. е. истинска реализация на метода на динамичното оптимиране), както и програмиране на пресмятанията за обратния ход, оставяме като упражнение за читателя.

Както е известно, досега за общата задача за търговския пътник няма публикуван полиномиален алгоритъм за решаването ѝ (виж

например [10]). Нещо повече, има сериозни причини, да се счита, че такъв алгоритъм изобщо не съществува. Методът на динамичното оптимиране, приложен за нея, не можа да я реши за полиномиално време, което и трябваше да се очаква. За разлика от други задачи, където този метод с успех заменя изчерпващото търсене, т. е. дава полиномиален алгоритъм вместо експоненциален, за задачата за търговския пътник успехът се изразява само в понижаване на сложността от $n!$ до 2^n. Според някои автори ([3]) философската причина за това е отсъствието на присъща нареденост на частичните подзадачи за разлика например от задачата за раницата.

Теми, задачи и въпроси за самостоятелна работа

1. Нека функцията f има за аргументи и за стойности естествени числа и е дефинирана рекурсивно чрез равенствата:

$$f(0) = a,$$
$$f(x) = h(x, f(g(x))), \ (x > 0),$$

където a е дадено естествено число, а h и g са известни функции. С други думи, стойността на f в точката x се изразява чрез стойността на f в точката $g(x)$. Освен това, предполага се, че за всяко x, редицата $x, g(x), g(g(x)), \ldots$ е такава, че някакъв неин член е равен на нула. Ако допълнително е известно, че $g(x) < x$ за всяко x, тогава пресмятането на f не е трудно: трябва да пресметнем последователно $f(0)$, $f(1)$, $f(2)$ и т. н.

Напишете нерекурсивна програма за пресмятане на f в общия случай.

Упътване: Задачата е по-сложен пример за премахване на рекурсията при пресмятания (виж Глава 1). В случая е достатъчно да образуваме и запомним стойностите на

$$g(x), g(g(x)), g(g(g(x))), \ldots$$

до мястото, където се появява първата нула. След това пресмятаме f в точките от горната редица, тръгвайки от дясно на ляво.

2. Напишете програма за намиране броя на редиците от n елемента, съдържащи само 0 и 1 такива, че в тези редици да не срещат подред 3 единици.

3. Напишете програма за намиране броя на редиците от n елемента, съдържащи само 0 и 1 такива, че в тези редици да не срещат подред k единици, където k е зададено цяло положително число.

4. Дадени са две естествени числа n и k. Трябва да се намери алгебричен израз за пресмятане на степента k^n така, че пресмятането да става възможно най-бързо. Разрешено е да се ползват операциите умножение `*`, степенуване `^`, скоби `()` и променлива с име `k`. Времето за извършване на едно умножение е една единица, а за повдигане на степен q се приема, че е необходимо време от $q-1$ единици. За дадено n намерете израз от описания вид, за който е необходимо минимално време за пресмятане.

Пример: За $n = 5$ са необходими 3 единици време и изразът е `(k*k)^2*k`.

Упътване: Нека в $p[i]$ записваме минималния брой операции, необходим за повдигане на степен i. Очевидно $p[1] = 0$. Да разгледаме операцията, която се извършва последна в реда на пресмятанията с търсения алгебричен израз, който е оптимален за повдигане на степен n. Ако тази последна операция е умножение, то се умножават две степени на k, които дават n-тата степен. Тези степени са с по-малки показатели от n и считаме, че вече сме пресметнали съответните елементи на масива p, показващи най-малкия брой операции, за които тези степени се получават. Полагаме

$$p1 = \min_{j=1,\ldots,n-1} p[j] + p[n-j] + 1.$$

Ако последната операция е била повдигане на степен, то

$$p2 = \min_{j\neq 1, \ n \ \mathtt{mod} \ j=0} p[n \ \mathtt{div} \ j] + j - 1,$$

където операцията `div` пресмята целочислено деление, а `mod` дава остатъка от делението.

За да приложим метода на динамичното оптимиране, забелязваме, че $p[n] = \min(p1, p2)$.

5. Подът на една стая с размери $m \times n$ ($1 \le m \le 20$, $0 \le n \le 8$, m и n са цели числа) трябва да се покрие (без да остават празни места и без да има припокриване) с еднакви правоъгълни парчета от паркет, всяко с размери 2×1. Напишете програма, която намира броя на различните възможни начини за покриване.

Пример: За $m = 2$ и $n = 3$ съществуват 3 различни начина.

6. Вие сте победител в състезание, организирано от Канадските авиолинии. Наградата е безплатно пътешествие из Канада. То трябва за започне от най-западния град, в който могат да кацат самолети, и да продължи на изток, докато достигне най-източния град, до който летят самолети. След това пътешествието продължава на запад, докато се стигне до началния град. Никой град освен него, не трябва да се посещава два пъти. Напишете програма, която при даден списък от градове и списък от директни самолетни рейсове между двойки градове намира маршрут, включващ максимален брой от дадените градове.

7. Шаблон ще наричаме низ, който съдържа латинските букви (a, ..., z, A, ..., Z) и двата знака ? и *. Всеки от знаците ? може да бъде заместен с произволна буква, а всеки знак * може да се замести с произволна последователност (включително и празната) от букви. За всеки низ от букви, който може да се получи по описания начин, казваме, че той удовлетворява шаблона. Напишете програма, която при дадени два шаблона намира низ от букви с минимална дължина, който удовлетворява и двата шаблона.

8. Даден е аритметичен израз, съставен от неотрицателни числа и знаците на операциите +, - и *. Напишете програма, която поставя в този израз скоби така, че пресметнатата стойност да стане максимална. *Пример:* За израза `1+2-3*4` максимална стойност се получава, когато пресметнем `((1+2)-3)*4`.

9. Видоизменете програмата от параграф 4.1 за намиране на оптимална триангулация така, че да се прилага друг критерий за оптималност — например брой на страните на триъгълниците или сума от лицата им, вместо сума от дължините на страните им.

10. Докажете, че всяка триангулация на изпъкнал многоъгълник с n върха има $n-3$ хорди и разделя многоъгълника на $n-2$ триъгълника.

11. Вярно ли е или не, че ако в задачата за оптимално умножение на матрици (параграф 4.2) има две съседни матрици с размери $1 \times k$ и $k \times 1$, то съществува оптимално решение, при което като последно произведение е включено умножение на матрици с указаните размери?

12. Напишете програма за намиране на втория по оптималност начин за умножаване на матрици по критериите от параграф 4.2.

13. Железопътна линия с еднопосочно движение има n гари. Дадени са цените на билетите от i-тата до j-тата гара за всяка двойка номера $i = 1, \ldots, n$, $j = 1, \ldots, n$ и $i < j$. Напишете програма, която намира най-евтиния начин за пътуване от началото до края на железопътната линия, като се използват възможностите за прекачване на гарите.

14. Разглеждаме крайно множество с бинарна операция (изобщо казано, некомутативна и даже неасоциативна). Зададени са n елемента $a_1, \ldots, a_n$ от това множество и още един елемент x. Да се провери дали е възможно да се поставят по подходящ начин скобите в произведението на $a_1, \ldots, a_n$ така, че в резултат да се получи x. Програмата трябва да извършва брой операции от порядъка на n^3.

Упътване: Запълваме таблица, в която за всеки отрез $a_i \ldots a_j$ от разглежданото произведение се пази списък от всичките му стойности, получени при всевъзможните разпределения на скобите.

15. Видоизменете програмата от параграф 4.4 за намиране на най-дълга растяща подредица на дадена редица така, че тя да намира най-дълга ненамаляваща подредица.

16. Каква е времевата сложност на програмата за намиране на най-дълга растяща подредица от параграф 4.4.?

17. Напишете програма с времева сложност от порядъка на $n \log n$ за намиране на най-дълга ненамаляваща подредица на редица от n числа (виж параграф 4.4.).

Упътване: Забележете, че последният елемент на подредицата кандидат с дължина i е поне толкова голям, колкото е последният елемент на подредицата кандидат с дължина $i-1$. Съхранявайте подредиците кандидати чрез подходящо обвързване с първоначалната редица.

18. 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 всеки елемент с нечетен номер е по-голям от своите съседи.
2. Всеки елемент с нечетен номер е по-малък от съседните си елементи, а всеки елемент с четен номер е по-голям от своите съседи.

Напишете програма, която за дадена редица от цели числа намира най-дългата ѝ трионообразна подредица.

Например за редицата от 5 числа: 8,3,5,7,0, едно решение на задачата е 8,3,5,0.

19. Дадени са n цели положителни числа $x_1, \ldots, x_n$ и числото a. Напишете програма, която проверява дали a може да се получи чрез събиране на някои от числата $x_1, \ldots, x_n$. Броят на действията, които програмата трябва да извърши, не бива да надминава порядъка на $a \cdot n$.

Упътване: След i стъпки да се пази множеството от тези числа, намиращи се в отреза $0 \ldots a$, които се представят като сума от някои от $x_1, \ldots, x_i$.

20. Да приемем, че имаме намерено оптимално дърво за търсене при зададени обекти и техните честоти (виж параграф 4.6.). Увеличаваме една или няколко от дадените честоти с 1. Напишете про-

грама, която да преустрои дървото за търсене така, че то да стане оптимално за новите честоти.

21. Да разгледаме проблема за най-красиво отпечатване на един абзац от текст. Данните са последователност от n думи, състоящи се съответно от $l_1, l_2, \ldots, l_n$ знака. Нашият критерий за красота включва изискването тези думи да се отпечатат в редове така, че броя на знаците в един ред да е най-много равен на M. Освен това, ако един ред съдържа думите с номера от i до j включително и между всеки две думи има точно една шпация, то броят на свободните места за знаци в края на реда е $M - j - i - \sum_{k=i}^{j} l_k$. Целта ни е да минимизираме сумата от кубовете на тези числа по всички редове с изключение на последния. Напишете програма, която да използва метода на динамичното оптимиране, за да отпечата абзац от n думи възможно най-красиво, съгласно горните критерии.

22. Дадени са два низа `source` и `target`. Те се състоят от букви и имат дължини съответно m и n. Разполагаме със спомагателен низ `temp`, който в началото е празен. Разглеждаме следните операции:

1. `copy` — взема текущата първа буква от `source` и я премества, като я долепя в края на `temp`.

2. `replace` – взема текущата първа буква от `source`, променя в друга буква и след това я долепя в края на `temp`.

3. `delete` – изтрива текущата първа буква от `source`.

4. `append` – долепя някаква буква на края на `temp`.

5. `twiddle` – взема текущите първи две букви от `source`, разменя им реда и ги долепя в края на `temp`.

Всяка от операциите има определена цена, зададена във вид на входни данни за вашата програма.

Целта е чрез последователност от тези операции да получим в `temp` низ, идентичен с `target`, така че сумата от цените на приложените операции да е минимална (очевидно във входните данни цената на `replace` би трябвало да е по-малка от сумата на цените на `delete` и `append`, защото иначе няма смисъл да се ползва `replace`).

Напишете програма, която чрез метода на динамичното оптимиране решава задачата. След прилагането на всички операции в `source` трябва да остане празният низ.

Пример ([2]) за една възможна последователност от операции, която преобразува '`algorithm`' в '`altruistic`':

–	–	algorithm
copy	a	lgorithm
copy	al	gorithm
replace	alt	orithm
delete	alt	rithm
copy	altr	ithm
append	altru	ithm
append	altrui	ithm
append	altruis	ithm
twiddle	altruisti	hm
append	altruistic	hm
delete	altruistic	m
delete	altruistic	–

23. Какво ще направят програмите за решаване на задачата за раницата от параграф 5.1., ако някои от дадените стойности са отрицателни?

24. Разновидност на задачата за търговския пътник (виж параграф 5.2.) се получава, когато върховете на графа са точки от евклидовата (т. е. обикновената геометрична) равнина и дължините на ребрата са обичайните разстояния между точките. Следваща разновидност (и опростяване на задачата) се получава, когато поискаме търсеният маршрут на търговския пътник да започва от най-лявата точка, да тръгва надясно, минавайки през няколко точки и вървейки все надясно, докато стигне най-дясната точка и след това да се върне, запазвайки посока наляво, докато стигне стартовата точка. Така описаната задача се нарича **битонична евклидова задача за търговския пътник**. Като допълнително (но несъществено) опростяване може да считате, че координатите (x, y) на точките са целочислени и че няма две точки с еднакви x-координати.

Докато за общата задача за търговския пътник не съществува полиномиален алгоритъм за решаването ѝ, то за описания частен случай такъв алгоритъм е възможен. Напишете програма, която решава битоничната евклидова задача за търговския пътник за n града с времева сложност, пропорционална на n^2.

Упътване: Сканирайте от ляво на дясно, обработвайки оптималните възможности за двете части на маршрута.

Литература

[1] Robert Sedgewick. Algorithms in C++. Addison-wesley publishing company, 1992.

В главата, посветена на динамичното оптимиране, първо е разгледан принципът „разделяй и владей". Динамичното оптимиране е представено като метод, който го довежда докрай. Подробно са описани следните алгоритми: алгоритъмът на Флойд за намиране на най-къс път в граф с тегла по ребрата му; задача за раницата; умножаване на матрици с минимален брой операции; оптимални търсещи дървета. Книгата има издадени още два варианта, в които са описани същите алгоритми, но на езиците Pascal и C.

[2] T. Cormen, C. Leiserson, R. Rivest. Introduction to Algorithms, MIT press, 1992.

Тази книга е един от основните учебници по алгоритми, написана от преподаватели в Масачузетския технологичен институт. В главата за „Динамично оптимиране" подробно са описани алгоритми за следните задачи: умножаване на матрици с минимален брой операции; най-дълга обща подредица на две дадени редици; оптимално разрязване на многоъгълник; битонична евклидова задача за търговския пътник.

[3] S. Skiena. The Algorithm Design Manual. Springer-Verlag, 1998.

В първата част на книгата, посветена на методи за създаване на алгоритми, има глава за динамичното оптимиране, в която са включени задачите: числа на Фибоначи; разпределяне на дейности; приближено търсене на низ; най-дълга растяща подредица.

Във втората част на книгата има раздели, ориентирани към различните предметни области, за които се създават алгоритми. Там са разгледани задачи за: умножаване на матрици с минимален брой операции; задача за раницата; алгоритъм на Флойд за най-къс път в граф; хамилтънов цикъл в граф; най-дълъг общ подниз на два низа.

[4] E. Horowitz, S. Sahni. Fundamentals of Computer Algorithms. Computer Science Press, 1978.

В отделна глава е описано динамичното оптимиране и подробно от теоретична гледна точка са разгледани задачите за: раницата; оптималното смесване; най-късия път; търговския пътник.

[5] А. Шень. Програмирование: Теоремы и задачи. МЦНМО, 1995.

Теми в книгата, свързани с динамичното оптимиране са: как да се справим без рекурсия; таблица от стойности; пример за пресмятане на биномни коефициенти; разсъждения върху числата на Фибоначи; задача за минимално разрязване на многоъгълник; стек с отложени стойности; нерекурсивна програма за „Ханойската кула" и др.

[6] В. Беров, А. Лапунов, В. Матухин, А. Пономаров. Особенности национальных задач по информатике. ООО Триада-С, Киров, 2000.

Разгледани са задачите: последователности от 0 и 1; възстановяване на скоби; уравнение с пропуснати цифри; ход с коня и др.

[7] С. Окулов, А. Пестов, О. Пестов. Информатика в задачах. Уеб сайт www.vspu.kirov.ru/~olymp, 1999.

Дадени са задачи за: движение на североизток; триъгълник от числа; повдигане на степен; зареждане с бензин на околовръстното шосе; най-дълъг подниз; разбиване на изпъкнал N-ъгълник; раница и др.

[8] Е. Келеведжиев. Задачи и решение от задочния конкурс по информатика. Списание Computer, 1988–2000 г., изд. Ню Техник Пъблишинг, София.

Част от задачите, предлагани в този конкурс се решават с методите на динамичното оптимиране.

[9] Е. Келеведжиев. Намиране на маршрути. Ойлерови и Хамилтънови цикли. Списание Computer, кн. 10, 1998 г., изд. Ню Техник Пъблишинг, София.

[10] Е. Келеведжиев. Винаги ли ще има трудно решими задачи. Списание Computer, кн. 3, 1998 г., изд. Ню Техник Пъблишинг, София.