Лекция 1

Динамично оптимиране

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.

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 й 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:

```
                         ------------ F(6) ----------
                      /                                \
               ---- F(5) ---                             F(4)
             /                \                        /        \
          F(4)                 F(3)                F(3)           F(2)
       /      \               /    \              /    \        /    \
    F(3)        F(2)       F(2)      F(1)      F(2)     F(1)   F(1)    F(0)
   /   \       /    \      /   \               /  \
  F(2)  F(1)  F(1)  F(0)  F(1)  F(0)          F(1)  F(0)
 /    \
F(1)  F(0)
```

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

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

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;}
```

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 варианти. 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 предишния. 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:

```
                C(5,3)
               /      \
          C(4,2)     C(4,3)
         /      \   /      \
    C(3,1)       C(3,2)     C(3,3)
```

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.

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, $\ldots$, A_{N-1}, то получаваме алгоритъм за решаване на задачата за всяко N. Така, започвайки от няколко известни начални стойности A_0, $\ldots$, A_k, намираме последователно следващите A_{k+1}, A_{k+2}, $\ldots$.

3. 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)`:

```
            B(5)
          /      \
     B(3)         B(4)
     /  \         /   \
B(1)  B(2)    B(2)  B(3)
                    /  \
                B(1)  B(2)
```

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.