算　法

1. 算法的基本概念
2. 递归
3. 排序问题和离散集合的操作
4. 回溯
5. 分支与限界
6. 动态规划
7. 背包九问

第一章　算法的基本概念

1.1　引言

算法设计与分析在计算机科学与技术中的地位
算法（Algorithm）一词的由来。

1.1.1　算法的定义和特征

欧几里德算法：

算法 1.1　欧几里德算法
输入：正整数 m,n
输出：m,n 的最大公因子

```
1. int euclid(int m,int n)
2. {
3.     int    r;
4.     do {
5.         r = m % n;
6.         m = n;
7.         n = r;
8.     } while(r)
9.     return m;
10. }
```

一、算法的定义：

定义 1.1　算法是解某一特定问题的一组有穷规则的集合。

二、算法的特征：

1．有限性。算法在执行有限步之后必须终止。

2．确定性。算法的每一个步骤，都有精确的定义。要执行的每一个动作都是清晰的、无歧义的。

3．输入。一个算法有 0 个或多个输入，它是由外部提供的，作为算法开始执行前的初始值，或初始状态。算法的输入是从特定的对象集合中抽取的。

4．输出。一个算法有一个或多个输出，这些输出，和输入有特定的关系，实际上是输入的某种函数。不同取值的输入，产生不同结果的输出。

5．能行性。算法的能行性指的是算法中有待实现的运算，都是基本的运算。原则上可以由人们用纸和笔，在有限的时间里精确地完成。

1.1.2　算法设计的例子，穷举法

一、穷举法，是从有限集合中，逐一列举集合的所有元素，对每一个元素逐一判断和处理，从而找出问题的解。

二、例

例 1.1　百鸡问题。

“鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一。百钱买百鸡，问鸡翁、母、雏各几何？”

a：公鸡只数，b：母鸡只数，c：小鸡只数。约束方程：

$$a+b+c=100 \qquad (1.1.1)$$

$$5a+3b+c/3=100 \qquad (1.1.2)$$

$$c\%3=0 \qquad (1.1.3)$$

1。第一种解法：

a、b、c的可能取值范围：0 ~ 100，对在此范围内的，a、b、c的所有组合进行测试，凡是满足上述三个约束方程的组合，都是问题的解。

把问题转化为用n元钱买n只鸡，n为任意正整数，则方程（1.1.1）、（1.1.2）变成：

$$a+b+c=n \qquad (1.1.4)$$

$$5a+3b+c/3=n \qquad (1.1.5)$$

算法 1.2　百鸡问题

输入：所购买的三种鸡的总数目 n

输出：满足问题的解的数目 k,公鸡,母鸡,小鸡的只数 g[],m[],s[]

```
 1. void chicken_question(int n,int &k,int g[],int m[],int s[])
 2. {
 3.    int  a,b,c;
 4.    k = 0;
 5.    for (a=0;a<=n;a++)
 6.       for (b=0;b<=n;b++)
 7.          for (c=0;c<=n;c++) {
 8.             if ((a+b+c==n)&&(5*a+3*b+c/3==n)&&(c%3==0)) {
 9.                g[k] = a;
10.                m[k] = b;
11.                s[k] = c;
12.                k++;
13.             }
```

```
14.            }
15.        }
16.    }
17. }
```

执行时间：外循环：$n+1$次，

中间循环：$(n+1)\times(n+1)$次，

内循环：$(n+1)\times(n+1)\times(n+1)$次。

当$n=100$时，内循环的循环体执行次数大于 100 万次。

2。第二种解法：

公鸡只数：$0\sim n/5$

母鸡只数：$0\sim n/3$

母鸡只数：$c=n-a-b$。

算法 1.3 改进的百鸡问题

输入：所购买的三种鸡的总数目 n

输出：满足问题的解的数目 k,公鸡,母鸡,小鸡的只数 g[],m[],s[]

```
 1. void chicken_problem(int n,int &k,int g[],int m[],int s[])
 2. {
 3.    int    i,j,a,b,c;
 4.    k = 0;
 5.    i = n / 5;
 6.    j = n / 3;
 7.    for (a=0;a<=i;a++)
 8.       for (b=0;b<=j;b++) {
 9.          c = n – a – b;
10.          if ((5*a+3*b+c/3==n)&&(c%3==0)) {
11.             g[k] = a;
12.             m[k] = b;
13.             s[k] = c;
14.             k++;
15.          }
16.       }
17.    }
18. }
```

执行时间：外循环：$n/5+1$

内循环：$(n/5+1)\times(n/3+1)$

当 $n=100$ 时，内循环的循环体的执行次数为 $21\times 34=714$ 次。

对某类特定问题，在规模较小的情况下，穷举法往往是一个简单有效的方法。

例 1.2 货郎担问题。

n 个城市，分别用 1 到 n 的数字编号，问题归结为在有向赋权图 $G=<V,E>$ 中，寻找一条路径最短的哈密尔顿回路。其中，$V=\{1,2,\cdots\cdots n\}$，表示城市顶点，边 $(i,j)\in E$ 表示城市 i 到城市 j 的距离，$i,j=1,2,\cdots\cdots n$。

图的邻接矩阵 C：表示各个城市之间的距离，称为费用矩阵。

数组 T：表示售货员的路线，依次存放旅行路线中的城市编号。

售货员的每一条路线，对应于城市编号 $1,2\cdots\cdots n$ 的一个排列。n 个城市共有 $n!$ 个排列，采用穷举法逐一计算每一条路线的费用，从中找出费用最小的路线，便可求出问题的解。

算法 1.4 穷举法版本的货郎担问题

输入：城市个数 n，费用矩阵 c[][]

输出：旅行路线 t，最小费用 min

```
1. void salesman_problem(int n,float &min,int t[],float c[][])
2. {
3.     int p[n],i = 1;
4.     float  cost;
5.     min = MAX_FLOAT_NUM;
6.     while (i <= n!) (
7.         产生 n 个城市的第 i 个排列于 p;
8.         cost =  路线 p 的费用;
9.         if (cost < min) {
10.            把数组 p 的内容拷贝到数组 t;
11.            min = cost;
12.        }
13.        i++;
14.    }
15. }
```

执行时间： while 循环执行 $n!$ 次。

表 1.1 算法 1.4 的执行时间随 n 的增长而增长的情况

n	$n!$	n	$n!$	n	$n!$	n	$n!$
5	120μs	9	362ms	13	1.72h	17	11.27year
6	720μs	10	3.62s	14	24h	18	203year

7	5.04ms	11	39.9s	15	15day	19	3857year
8	40.3ms	12	479.0s	16	242day	20	77146year

1.1.3 算法的复杂性分析

问题：效率和方法。

问题一：如何设计算法，算法的设计方法。

问题二：如何分析算法，算法的复杂性分析。

用算法的复杂性来衡量算法的效率。算法的时间复杂性和算法的空间复杂性。算法的时间复杂性越高，算法的执行时间越长；反之，执行时间越短。算法的空间复杂性越高，算法所需的存储空间越多；反之越少。

1.2 算法的时间复杂性

一、算法复杂性的度量？

二、如何分析和计算算法的复杂性？

1.2.1 算法的输入规模和运行时间的阶

一、算法的输入规模和运行时间

令百鸡问题的第一、二两个算法，其最内部的循环体每执行一次，需 1μs 时间。

	$n=100$ 的内循环次数	时间	$n=10000$ 的内循环次数	时间
第一个算法	100 万次	1s	10000^3	11 天零 13 小时
第二个算法	714 次	714μs	$(10000/5+1)\times(10000/3+1)$	6.7 秒

$n=2^{20}$　选择排序需 6.4 天　合并排序需 20 秒

算法的执行时间随问题规模的增大而增长的情况。

二、算法运行时间的评估

不能准确地计算算法的具体执行时间

不需对算法的执行时间作出准确地统计（除非在实时系统中）

1、计算模型：RAM 模型（随机存取机模型）、图灵机模型等

2、初等操作：所有操作数都具有相同的固定字长；所有操作的时间花费都是一个常数时间间隔。算术运算；比较和逻辑运算；赋值运算，等等；

例：输入规模为 n，百鸡问题的第一个算法的时间花费，可估计如下：

$$T_1(n)\le 1+2(n+1)+n+1+2(n+1)^2+(n+1)^2+16(n+1)^3+4(n+1)^3$$
$$=20n^3+63n^2+69n+27 \qquad (1.1.6)$$

可把 $T_1(n)$ 写成：

$$T_1^*(n)\approx c_1n^3 \qquad c_1>0 \qquad (1.1.7)$$

这时，称$T_1^*(n)$的阶是n^3。

百鸡问题的第一个算法的时间花费：

$$T_2(n)\le 1+2+2+1+2\times(n/5+1)+n/5+1+2\times(n/5+1)\times(n/3+1)+$$
$$(3+10+4)\times(n/5+1)\times(n/3+1)$$
$$=\frac{19}{15}n^2+\frac{161}{15}n+28 \qquad (1.1.8)$$

同样，随着n的增大，$T_2(n)$也可写成：

$$T_2^*(n)\approx c_2n^2 \qquad c_2>0 \qquad (1.1.9)$$

这时，称$T_2^*(n)$的阶是n^2。把$T_1^*(n)$和$T_2^*(n)$进行比较，有：

$$T_1^*(n)/T_2^*(n)=\frac{c_1}{c_2}n \qquad (1.1.10)$$

当n很大时，c_1/c_2的作用很小。

3、算法时间复杂性的定义：

定义 1.2 设算法的执行时间$T(n)$，如果存在$T^*(n)$，使得：

$$\lim_{n\to\infty}\frac{T(n)-T^*(n)}{T(n)}=0 \qquad (1.1.11)$$

就称$T^*(n)$为算法的渐近时间复杂性。

表 1.2 表示时间复杂性的阶为$\log n$, n, $n\log n$, n^2, n^3, 2^n，当$n=2^3, 2^4, \cdots\cdots, 2^{16}$ 时，算法的渐近运行时间。这里假定每一个操作是$1ns$。

表 1.3 表示对不同时间复杂性的算法，计算机速度提高后，可处理的规模n_2和n_1的关系。

表 1.2　不同时间复杂性下不同输入规模的运行时间

n	$\log n$	n	$n\log n$	n^2	n^3	2^n
8	3 n	8 n	24 n	64 n	512 n	256 n
16	4 n	16 n	64 n	256 n	4.096 μ	65.536 μ
32	5 n	32 n	160 n	1.024 μ	32.768 μ	4294.967 ms
64	6 n	64 n	384 n	4.096 μ	262.144 μ	5.85 c
128	7 n	128 n	896 n	16.384 μ	1997.152 μ	10^{20} c
256	8 n	256 n	2.048 μ	65.536 μ	16.777 ms	10^{58} c
512	9 n	512 n	4.608 μ	262.144 μ	134.218 ms	10^{135} c
1024	10 n	1.024 μ	10.24 μ	1048.576 μ	1073.742 ms	10^{289} c
2048	11 n	2.048 μ	22.528 μ	4194.304 μ	8589.935 ms	10^{598} c
4096	12 n	4.096 μ	49.152 μ	16.777 ms	68.719 s	10^{1214} c
8192	13 n	8.196 μ	106.548 μ	67.174 ms	549.752 s	10^{2447} c

16384	14 n	16.384 μ	229.376 μ	268.435 ms	1.222 h	10^{4913} c
32768	15 n	32.768 μ	491.52 μ	1073.742 ms	9.773 h	10^{9845} c
65536	16 n	65.536 μ	1048.576 μ	4294.967 ms	78.187 h	10^{19709} c

n：纳秒　μ：微秒　ms：毫秒　s：秒　h：小时　d：天　y：年　c：世纪

表 1.3　计算机速度提高 10 倍后，不同算法复杂性求解规模的扩大情况

算法	A_1	A_2	A_3	A_4	A_5	A_6
时间复杂性	n	$n\log n$	n^2	n^3	2^n	$n!$
n_2 和 n_1 的关系	$10\,n_1$	$8.38\,n_1$	$3.16\,n_1$	$2.15\,n_1$	$n_1+3.3$	n_1

4、多项式时间算法和指数时间算法。

1.2.2　运行时间的上界，O 记号

一、O 记号的定义：

定义 1.3　令 N 为自然数集合，R_+ 为正实数集合。函数 $f: N\to R_+$，函数 $g: N\to R_+$，若存在自然数 n_0 和正常数 c，使得对所有的 $n\ge n_0$，都有 $f(n)\le cg(n)$，就称函数 $f(n)$ 的阶至多是 $O(g(n))$。

因此，如果存在 $\lim\limits_{n\to\infty} f(n)/g(n)$，则：

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}\neq\infty$$

即意味着：

$$f(n)=O(g(n))$$

含义：$f(n)$ 的增长最多象 $g(n)$ 的增长那样快。称 $O(g(n))$ 是 $f(n)$ 的上界。

二、例：百鸡问题的第二个算法，由式（1.1.8）有：

$$T_2(n)\le\frac{19}{15}n^2+\frac{161}{15}n+28$$

取 n_0=28，对 $\forall n\ge n_0$，有：

$$\begin{aligned}T_2(n)&\le\frac{19}{15}n^2+\frac{161}{15}n+n\\&=\frac{19}{15}n^2+\frac{176}{15}n\\&\le\frac{19}{15}n^2+\frac{176}{15}n^2\\&=13n^2\end{aligned}$$

令 $c=13$，并令 $g(n)=n^2$，有：

$$T_2(n) \le cn^2 = cg(n)$$

所以，$T_2(n) = O(g(n)) = O(n^2)$。

这时，如果有一个新算法，其运行时间的上界低于以往解同一问题的所有其它算法的上界，就认为建立了一个解该问题所需时间的新上界。

1.2.3 运行时间的下界，Ω记号

一、Ω记号的定义：

定义 1.4 令 N 为自然数集合，R_+ 为正实数集合。函数 $f: N \to R_+$，函数 $g: N \to R_+$，若存在自然数 n_0 和正常数 c，使得对所有的 $n \ge n_0$，都有 $f(n) \ge cg(n)$， 就称函数 $f(n)$ 的阶至少是 $\Omega(g(n))$。

因此，如果存在 $\lim_{n\to\infty} f(n)/g(n)$，则：

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \ne 0$$

即意味着：

$$f(n) = \Omega(g(n))$$

含义： $f(n)$ 的增长至少象 $g(n)$ 那样快。表示解一个特定问题的任何算法的时间下界。

二、例：百鸡问题的第二个算法第 11、12、13、14 行，仅在条件成立时才执行，其执行次数未知。假定条件都不成立，这些语句一次也没有执行，该算法的执行时间至少为：

$$\begin{aligned} T_2(n) &\ge 1+2+2+1+2\times(n/5+1)+n/5+1+2\times(n/5+1)\times(n/3+1)+ \\ &\quad (3+10)\times(n/5+1)\times(n/3+1) \\ &= n^2 + \frac{43}{5}n + 24 \\ &\ge n^2 \end{aligned}$$

当取 $n_0 = 1$ 时， $\forall n \ge n_0$，存在常数 $c = 1$， $g(n) = n^2$，使得：

$$T_2(n) \ge n^2 = cg(n)$$

三、结论 1.1 $f(n)$ 的阶是 $\Omega(g(n))$，当且仅当 $g(n)$ 的阶是 $O(f(n))$。

1.2.4 运行时间的准确界，Θ记号

百鸡问题的第二个算法，运行时间的上界是 $13n^2$，下界是 n^2，这表明不管输入规模如何变化，该算法的运行时间都界于 n^2 和 $13n^2$ 之间。这时，用记号 Θ 来表示这种情况，认为这个算法的运行时间是 $\Theta(n^2)$。Θ 记号表明算法的运行时间有一个较准确的界。

一、Θ记号的定义如下：

定义 1.5 令 N 为自然数集合，R_+ 为正实数集合。函数 $f: N \to R_+$，函数 $g: N \to R_+$，若存在自然数 n_0 和两个正常数 $0 \le c_1 \le c_2$，使得对所有的 $n \ge n_0$， 都有：

$$c_1 g(n) \le f(n) \le c_2 g(n)$$

就称函数 $f(n)$ 的阶是 $\Theta(g(n))$。

因此，如果存在 $\lim_{n\to\infty} f(n)/g(n)$，则：

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=c$$

即意味着：

$$f(n)=\Theta(g(n))$$

其中，c 是大于 0 的常数。

二、例：

例 1.3 常函数 $f(n)=4096$。

令 $n_0=0$，$c=4096$，使得对 $g(n)=1$，对所有的 n 有：

$f(n)\le 4096\times 1=cg(n)$ ∴ $f(n)=O(g(n))=O(1)$

$f(n)\ge 4096\times 1=cg(n)$ ∴ $f(n)=\Omega(g(n))=\Omega(1)$。

$cg(n)\le f(n)\le cg(n)$ ∴ $f(n)=\Theta(1)$。

例 1.4 线性函数 $f(n)=5n+2$。

令 $n_0=0$，当 $n\ge n_0$ 时，有 $c_1=5$，$g(n)=n$，使得：

$f(n)\ge 5n=c_1g(n)$ ∴ $f(n)=\Omega(g(n))=\Omega(n)$。

令 $n_0=2$，当 $n\ge n_0$ 时，有：$c_2=6$，$g(n)=n$

$$\begin{aligned} f(n) &\le 5n+n \\ &= 6n \\ &= c_2g(n) \end{aligned}$$ ∴ $f(n)=O(g(n))=O(n)$。

同时，有：

$c_1g(n)\le f(n)\le c_2g(n)$ ∴ $f(n)=\Theta(n)$。

例 1.5 平方函数 $f(n)=8n^2+3n+2$。

令 $n_0=0$，当 $n\ge n_0$ 时，有 $c_1=8$，$g(n)=n^2$，使得：

$f(n)\ge 8n^2=c_1g(n)$ ∴ $f(n)=\Omega(g(n))=\Omega(n^2)$。

令 $n_0=2$，当 $n\ge n_0$ 时，有：$c_2=12$，$g(n)=n^2$

$$\begin{aligned} f(n) &\le 8n^2+3n+n \\ &\le 12n^2 \\ &= c_2g(n) \end{aligned}$$ ∴ $f(n)=O(g(n))=O(n^2)$。

同时，有：

$c_1g(n)\le f(n)\le c_2g(n)$ ∴ $f(n)=\Theta(n^2)$。

结论 1.2 令：

$$f(n)=a_kn^k+a_{k-1}n^{k-1}+\cdots\cdots+a_1n+a_0 \qquad a_k>0$$

则有：$f(n)=O(n^k)$，且 $f(n)=\Omega(n^k)$，因此，有：$f(n)=\Theta(n^k)$。

例 1.6 指数函数 $f(n)=5\times 2^n+n^2$。

令 $n_0=0$，当 $n\geq n_0$ 时，有 $c_1=5$，$g(n)=2^n$，使

$$f(n)\geq 5\times 2^n=c_1 g(n) \qquad \therefore\ f(n)=\Omega(g(n))=\Omega(2^n)。$$

令 $n_0=4$，当 $n\geq n_0$ 时，有：$c_2=6$，$g(n)=2^n$

$$\begin{aligned} f(n) &\leq 5\times 2^n+2^n \\ &\leq 6\times 2^n \\ &= c_2 g(n) \end{aligned} \qquad \therefore\ f(n)=O(g(n))=O(2^n)。$$

同时，有：

$$c_1 g(n)\leq f(n)\leq c_2 g(n) \qquad \therefore\ f(n)=\Theta(2^n)。$$

例 1.7 对数函数 $f(n)=\log n^2$。

因为：$\log n^2=2\log n$

令 $n_0=1$，$c_1=1$，$c_2=3$，$g(n)=\log n$，有：

$$c_1 g(n)\leq 2\log n\leq c_2 g(n) \qquad \therefore\ \log n^2=\Theta(\log n)。$$

结论 1.2' 对任何正常数 k，都有：$\log n^k=\Theta(\log n)$

例 1.8 函数 $f(n)=\sum_{j=1}^{n}\log j$

$$\begin{aligned} \sum_{j=1}^{n}\log j &\leq \sum_{j=1}^{n}\log n \\ &= n\log n \end{aligned}$$

令 $n_0=1$，$c_1=1$，$g(n)=n\log n$，有：

$$\sum_{j=1}^{n}\log j\leq c_1 g(n) \qquad \therefore\ \sum_{j=1}^{n}\log j=O(g(n))=O(n\log n)$$

另一方面，假定 n 是偶数，

$$\begin{aligned} \sum_{j=1}^{n}\log j &\geq \sum_{j=1}^{n/2}\log\frac{n}{2} \\ &= \frac{n}{2}\log\frac{n}{2} \\ &= \frac{n}{2}(\log n-1) \\ &= \frac{n}{4}(\log n+\log n-2) \end{aligned}$$

因此，令 $n_0=4$， $c_2=1/4$， $g(n)=n\log n$，对所有的 $n\ge n_0$，都有：

$$\sum_{j=1}^{n}\log j \ge \frac{1}{4}n\log n$$
$$=c_2 g(n)$$
$$=\Omega(g(n))$$
$$=\Omega(n\log n)$$

因此，有：

$$c_2 g(n)\le \sum_{j=1}^{n}\log j \le c_1 g(n)$$

所以，

$$\sum_{j=1}^{n}\log j=\Theta(g(n))=\Theta(n\log n)$$

结论 1.2’’ $\log n!=\Theta(n\log n)$。

1.2.5 复杂性类型和 o 记号

一、复杂性的分类

定义 1.6 令 R 是函数集合 F 上的一个关系， $R\subseteq F\times F$，有

$$R=\{<f,g>\mid f\in F\wedge g\in F\wedge f(n)=\Theta(g(n))\}$$

则 R 是自反、对称、传递的等价关系，它诱导的等价类，称阶是 $g(n)$ 的复杂性类型的等价类。

所有常函数的复杂性类型都是 $\Theta(1)$；

所有线性函数的复杂性类型都是 $\Theta(n)$；

所有的 2 阶多项式函数的复杂性类型都是 $\Theta(n^2)$，如此等等。

例： $f(n)=4096$， $g(n)=3n+2$， $\exists\ n_0=1$， $c=1365$， $\forall\ n\ge n_0$，有 $f(n)\le cg(n)$。 $\therefore f(n)=O(g(n))=O(n)$。

又：

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{4096}{3n+2}=0$$ $\therefore f(n)\ne\Omega(g(n))$，则： $f(n)\ne\Theta(g(n))$。

$\therefore$ $f(n)=4096$ 和 $g(n)=3n+2$ 是属于不同复杂性类型的函数。

例： $\log 2^n=n$ $\log n!=\Theta(n\log n)$ $\log n!\le cg(n\log n)$

由 $n<n\log n$ $\exists\ n_0\ge 0$， $c\ge 0$ ，$\forall\ n\ge n_0$

有 $2^n\le cn!$ $\therefore\ 2^n=O(n!)$

$$\lim_{n\to\infty}\frac{2^n}{n!}=\lim_{n\to\infty}\frac{2\cdot 2\cdots 2}{1\cdot 2\cdots n}=0 \qquad \therefore\ 2^n\neq\Omega(n!)$$

2^n 和 $n!$ 是属于不同复杂性类型的函数。

例： $\log n!=\Theta(n\log n)$ $\log 2^{n^2}=n^2>n\log n$

$n!=O(2^{n^2})$ $2^{n^2}\neq O(n!)$。

这两个函数也是属于不同复杂性类型的函数。

二、o 记号的定义

定义 1.7 令 N 为自然数集合，R_+ 为正实数集合。函数 $f: N\to R_+$，函数 $g: N\to R_+$，若存在自然数 n_0 和正常数 c，使得对所有的 $n\geq n_0$，都有

$$f(n)<cg(n)$$

就称函数 $f(n)$ 是 $o(g(n))$。

由此，如果存在 $\lim_{n\to\infty} f(n)/g(n)$，则

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0 \qquad \text{即意味着} \qquad f(n)=o(g(n))$$

结论 1.3 $f(n)=o(g(n))$ 当且仅当 $f(n)=O(g(n))$ 而 $g(n)\neq O(f(n))$。

复杂性类型体系：用偏序关系 $f(n)\prec g(n)$ 表示 $f(n)=o(g(n))$。

$$1\prec\log\log n\prec\log n\prec\sqrt{n}\prec n^{3/4}\prec n\prec n\log n\prec n^2\prec 2^n\prec n!\prec 2^{n^2}$$

1.3 算法的时间复杂性分析

1.3.1 循环次数的统计

一、循环次数表示乘以一个常数因子的运行时间

例 1.9 计算多项式：

$$P(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots\cdots+a_1x+a_0$$

Horner 法则改写： $P(x)=(\cdots(a_nx+a_{n-1})x+\cdots\cdots+a_1)x+a_0$

算法 1.5 计算多项式

输入：存放多项式系数的数组 A[],实数 x,多项式的阶 n

输出：多项式的值

```
1. float polynomial(float A[],float x,int n)
2. {
3.     int  i;
4.     float value;
5.     for (i=n;i>0;i--) {
```

```
6.        value = A[i] * x + A[i-1];
7.    return value;
8. }
```

c_1：循环控制变量 i 赋初值所花费的单位时间

c_2：变量 i 的测试及递减、以及值 *value* 的计算所花费的单位时间

算法的执行时间 $T(n)$ 为：　　$T(n)=c_1+c_2n$

$$=\Theta(n)$$

例 1.10　把数组中 n 个元素由小到大进行排序。

算法 1.6　冒泡算法

输入：数组 A[],元素个数 n

输出：按递增顺序排序的数组 A[]

```
1. template <class T>
2. void bubble(Type A[],int n)
3. {
4.    int i,k;
5.    for (k=n-1;k>0;k--) {
6.       for (i=0;i<k;i++) {
7.           if (A[i] > A[i+1]) {
8.               swap(A[i],A[i+1]);
9.           }
10.       }
11.   }
12. }
13. void swap(Type &x,Type &y)
14. {
15.   Type temp;
16.   temp = x;
17.   x = y;
18.   y = temp;
19. }
```

$\bar{c}$：辅助操作的执行时间

c：循环体的平均执行时间

算法总的执行时间 $T(n)$ 为：

$$T(n)=((n-1)+(n-2)+\cdots\cdots+1)\ c+\bar{c}$$

$$= \frac{c}{2} n(n-1) + \bar{c}$$

$$= \Theta(n^2)$$

例 1.11 选手的竞技淘汰比赛。

有 $n = 2^k$ 位选手进行竞技淘汰比赛，最后决出冠军的选手。假定用如下的函数：

```
BOOL comp(Type mem1,Type mem2)
```

模拟两位选手的比赛，若 *mem*1 胜则返回 *TRUE* ，否则返回 *FALSE* 。

并假定可以在常数时间 *c* 内完成函数 comp 的执行。

算法 1.7 竞技淘汰比赛

输入：选手成员 group[],选手个数 n

输出：冠军的选手

```
 1. Type game(Type group[],int n)
 2. {
 3.     int j,i = n;
 4.     while (i>1) {
 5.         i = i / 2;
 6.         for (j=0;j<i;j++)
 7.             if (comp(group[j+i],group[j]);
 8.                 group[j] = group[j+i];
 9.     }
10.     return group[0];
11. }
```

因为 $n = 2^k$ ，第 4 行的 while 循环的循环体共执行 *k* 次。

在每一次执行时，第 6 行的 for 循环的循环体，其执行次数分别为 $n/2, n/4, \cdots\cdots, 1$，

函数 comp 可以在常数时间内完成。

算法的执行时间 $T(n)$ 为：

$$T(n) = \frac{n}{2} + \frac{n}{4} + \cdots\cdots + \frac{n}{n}$$

$$= n\left(\frac{1}{2} + \frac{1}{4} + \cdots\cdots + \frac{1}{2^k}\right)$$

$$= n\left(1 - \frac{1}{2^k}\right)$$

$$= n - 1$$

$$= \Theta(n)$$

例 1.12 对 *n* 张牌进行 *n* 次洗牌，洗牌规则如下：在第 *k* 次洗牌时（ $k = 1 \cdots n$ ），对第 *i*

张牌（$i=1\cdots n/k$）随机地产生一个小于 n 的正整数 d，互换第 i 张牌和第 d 张牌的位置。

算法 1.8 洗牌

输入：牌 A[],牌的张数 n

输出：洗牌后的牌 A[]

```
 1. template <class Type>
 2. void shuffle(Type A[],int n)
 3. {
 4.    int  i,k,m,d;
 5.    random_seed(0);
 6.    for (k=1;k<=n;k++) {
 7.       m = n / k ;
 8.       for (i=1;i<=m;i++) {
 9.          d = random(1,n);
10.          swap(A[i],A[d]);
11.       }
12.    }
13. }
```

函数 random_seed：为随机数发生器产生随机数种子，需常数时间

函数 random：产生一个 1 到 n 之间的随机数，需常数时间

第 6 行开始的 for 循环的循环体共执行 n 次。

第 8 行开始的内部 for 循环的循环体，其执行次数依次为：

$$n,\lfloor n/2\rfloor,\lfloor n/3\rfloor,\cdots\cdots\lfloor n/n\rfloor$$

算法的执行时间 $T(n)$ 为内部 for 循环的循环体的执行次数乘以一个常数时间，因此，有：

$$T(n)=\sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor$$

因为：

$$\sum_{i=1}^{n}(\frac{n}{i}-1)\ \le\ \sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor\le\sum_{i=1}^{n}\frac{n}{i}$$

由调和级数的性质，有：

$$\ln(n+1)\le\sum_{i=1}^{n}\frac{1}{i}\le\ln n+1$$

因此：

$$\frac{\log(n+1)}{\log e} \le \sum_{i=1}^{n} \frac{1}{i} \le \frac{\log n}{\log e} + 1$$

所以：

$$\frac{1}{\log e} n\log(n+1) - n \le \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \le \frac{1}{\log e} n\log n + n$$

由此得出：

$$T(n) = \Theta(n\log n)$$

1.3.2 基本操作频率的统计

一、基本操作的定义

定义 1.8 算法中的某个初等操作，如果它的最高执行频率，和所有其它初等操作的最高执行频率，相差在一个常数因子之内，就说这个初等操作是一个基本操作。

初等操作的执行频率，可正比于任何其它操作的最高执行频率

基本操作的选择，必须反映出该操作随着输入规模的增加而变化的情况

二、用基本操作的执行频率估计算法的时间复杂性

例 1.13 合并两个有序的子数组

假定 A 是一个具有 m 个元素的整数数组，给定三个下标：p,q,r，$0 \le p \le q \le r < m$，使得 $A[p] \sim A[q]$，$A[q+1] \sim A[r]$ 分别是两个以递增顺序排序的子数组。把这两个子数组按递增顺序合并在 $A[p] \sim A[r]$ 中。

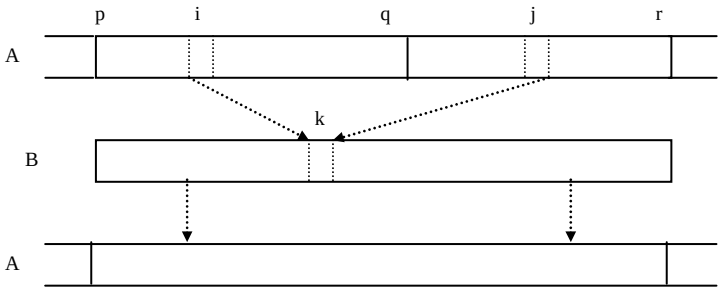
算法 1.9 合并两个有序的子数组

输入：整数数组 A[],下标 p,q,r,元素个数 m。A[p]~A[q]及 A[q+1]~A[r]已按递增顺序排序

输出：按递增顺序排序的子数组 A[p]~A[r]

```
 1. void merge(int A[],int p,int q,int r,int m)
 2. {
 3.     int  *bp = new int[m];          /* 分配缓冲区,存放被排序的元素 */
 4.     int  i,j,k;
 5.     i = p;   j = q + 1;   k = 0;
 6.     while (i<=q && j<=r) {          /* 逐一判断两子数组的元素 */
 7.         if (A[i]<=A[j])             /* 按两种情况,把小的元素拷贝到缓冲区*/
 8.             bp[k++] = A[i++];
 9.         else
10.             bp[k++] = A[j++];
11.     }
12.     if (i==q+1)                     /* 按两种情况,处理其余元素 */
```

```
13.        for (;j<=r;j++)
14.            bp[++] = A[j++];;       /* 把A[j]~A[r]拷贝到缓冲区 */
15.    else
16.        for (;i<=q;i++)
17.            bp[++] = A[i++];        /* 把A[i]~A[q]拷贝到缓冲区 */
18.    k = 0;
19.    for (i=p;i<=r;i++)    /* 最后,把数组bp的内容拷贝导A[p]~A[r] */
20.        A[i++] = bp[k++];
21.    delete bp;
22. }
```



while 循环的循环次数、for 循环的循环次数未知

基本操作的选择：

1、数组元素的赋值操作作为基本操作，操作频率：$2n$，

1）、随输入规模的增大而增加

2）、执行频率与其它操作的执行频率相差一个常数因子

算法的时间复杂性为$\Theta(n)$。

2、数组元素的比较操作作为基本操作

令两个子数组的大小分别为n_1和n_2，其中，$n_1+n_2=n$。

合并两个数组时，数组元素的比较次数，最少为n_1，最多为$n-1$次。

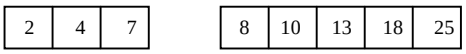


图1.1　合并两个有序数组时, 元素比较次数最少的情况

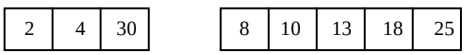


图1.2　合并两个有序数组时, 元素比较次数最多的情况

如果合并两个大小接近相同的有序数组，例如$n_1=\lfloor n/2 \rfloor$，$n_2=\lceil n/2 \rceil$，

数组元素的比较操作，操作频率：$n/2$

满足上述 1）2）。算法的时间复杂性仍然是$\Theta(n)$。

例 1.14　菜园四周种了n棵白菜，并按顺时针方向由 1 到n编号。收割时，从编号 1 开

始，按顺时针方向每隔两棵白菜收割一棵，直到全部收割完毕为止。按收割顺序列出白菜的编号。

数组 A：存放白菜的编号，初值为$1,\cdots,n$。当白菜被收割后，从数组中删去相应元素

数组 B：按收割顺序存放被收割白菜的编号

算法 1.10 收割白菜

输入：白菜棵数 n

输出：按收割顺序存放白菜编号的数组 B[]

```
 1. void reap(int B[],int n)
 2. {
 3.    int i,j,k,s,t;
 4.    int *A = new int[n];
 5.    j = 0;   k = 3;   s = n;
 6.    for (i=0;i<n;i++)
 7.       A[i] = i + 1;
 8.    while (j<n) {
 9.       t = s;   s = 0;
10.       for (i=0;i<t;i++) {
11.          if (--k!=0)
12.             A[s++] = A[i];              /* 未被收割的白菜 */
13.          else {
14.             B[j++] = A[i];   k = 3;     /* 被收割的白菜 */
15.          }
16.       }
17.    }
18.    delete A;
19. }
```

while 循环 for 循环的循环次数未知

基本操作的选择：14 行的赋值操作需要执行 n 次，12 行的赋值操作，需要执行 $2n$ 次

算法的运行时间为 $\Theta(n)$。

1.3.3 计算步的统计

一、计算步

定义 1.9 计算步是一个语法或语义意义上的程序段，该程序段的执行时间与输入实例无关。

例：

```
flag=(a+b+c==n)&&(5*a+3*b+c/3==n)&&(c%3==0);
a=b;
```

和输入规模无关。连续 200 个乘法操作可为一个计算步，*n* 次加法不能作为一个计算步计算步所表示的计算量，可能有很大的差别。

二、计算步的统计

把全局变量 *count* 嵌入实现算法的程序中，每执行一个计算步，*count* 就加 1。算法运行结束时，*count* 的值，就是算法所需执行的计算步数。

随着输入实例的不同，按这种方式统计出来的计算步数也不同。它有助于了解算法的执行时间随输入实例的变化而变化的情况。如果输入实例的规模增大 10 倍，所需执行的计算步数也增加 10 倍，就可以认为运行时间随着 *n* 的增大而线性增加。

1.3.4 最坏情况和平均情况

一、影响运行时间的因素

问题规模的大小、输入的具体数据（除算法的性能外）

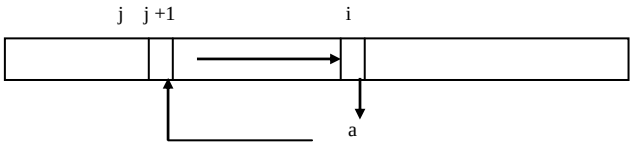
例 1.15 用插入法对 *n* 个元素的数组 A，按递增顺序进行排序。

算法 1.11 用插入法按递增顺序排序数组 A

输入：n 个元素的整数数组 A[],数组元素个数 n

输出：按递增顺序排序的数组 A[]

```
 1. void insert_sort(int A[],int n)
 2. {
 3.    int  a,i,j;
 4.    for (i=1;i<n;i++) {
 5.       a = A[i];
 6.       j = i – 1;
 7.       while (j>=0 && A[j]>a) {
 8.          A[j+1] = A[j];
 9.          j--;
10.       }
11.       A[j+1] = a;
12.    }
13. }
```



1. 初始数组已按递增顺序排列，执行时间既是$O(n)$的，也是$\Omega(n)$，所以，是$\Theta(n)$的。

2. 初始数组按递减顺序排列，每一个元素$A[i],1\le i\le n-1$，都和它前面的i个元素进行比较，则整个算法执行的元素比较次数为：

$$\sum_{i=1}^{n-1} i=\frac{1}{2}n(n-1)$$

在这种情况下，算法的执行时间是$O(n^2)$的，也是$\Omega(n^2)$的，所以是$\Theta(n^2)$的。

二、算法时间复杂性的三种分析

最坏情况的分析、平均情况的分析、和最好情况的分析。

1.3.5 最坏情况分析

下界和上界不一致的情况：

例 1.16 对已经排序过的、具有n个元素的数组A，检索是否存在元素x。当n是奇数时，用二叉检索算法检索；当n是偶数时，用线性检索算法检索。

算法 1.12 分别采用线性检索算法和二叉检索算法进行检索的算法

输入：给定 n 个已排序的元素的数组 A[],及元素 x

输出：若 x = A[j],1≤j≤n,输出 j,否则输出 0

```
1. int linear_search(int A[],int n,int x);
2. int binary_search(int A[],int n,int x);
3. int serach(int A[],int n,int x)
4. {
5.    if ((n%2)==0)
6.        return linear_search(A,n,x);
7.    else
8.        return binary_search(A,n,x);
10. }
```

这个算法在n是偶数时，调用线性检索算法进行检索；在n是奇数时，调用二叉检索算法进行检索。线性检索算法如下：

算法 1.13 线性检索算法

输入：给定 n 个已排序过的元素的数组 A[],及元素 x

输出：若 x = A[j],0≤j≤n-1,输出 j,否则输出-1

```
1.  int linear_search(int A[],int n,int x);
2.  {
3.     int j = 0;
4.     while (j<n && x!=A[j])
```

```
5.          j++;
6.      if ((j<n)&&(x==A[j]))
7.          return j;
8.      else
9.          return -1;
10.  }
```

二叉检索算法如下：

算法 1.14 二叉检索算法

输入：给定 n 个已排序过的元素的数组 A[],及元素 x

输出：若 x = A[j],0≤j≤n-1,输出 j,否则输出-1

```
1.  int binary_search(int A[],int n,int x)
2.  {
3.      int mid,low = 0,high = n – 1,j = -1;
4.      while (low<=high && j<0) {
5.          mid = (low + high) / 2;
6.          if (x==A[mid]) j = mid;
7.          else if (x<A[mid]) high = mid –1;
8.          else low = mid + 1;
9.      }
10.     return j;
11.  }
```

线性检索算法的最坏情况：数组中不存在元素 x，或元素 x 是数组的最后一个元素

时间复杂性：$O(n)$、$\Omega(n)$、$\Theta(n)$

二叉检索算法的最坏情况：数组中不存在元素 x，或元素 x 是数组的第一个元素、或最后一个元素

时间复杂性是 $O(\log n)$、$\Omega(\log n)$、$\Theta(\log n)$

search 在最坏情况下的时间复杂性。$O(n)$ 的，也是 $\Omega(\log n)$。

1.3.6 平均情况分析

在平均情况下，算法的运行时间取算法在所有可能输入的平均运行时间。

预先知道输入的出现概率，即所有输入的分布情况。

例 1.17 插入排序算法 insert_sort 的平均情况分析。

数组 A 中的元素为 $\{x_1, x_2, \cdots\cdots, x_n\}$，并且 $x_i \neq x_j, 1 \leq i, j \leq n, i \neq j$。

n 个元素共有 $n!$ 种排列，假定，每一种排列的概率相同。

前面$i-1$个元素已按递增顺序排序，把元素x_i插入到合适位置的i种可能：

$j=1$： x_i是序列中最小的，需执行$i-1$次比较；

$j=2$： x_i是这个序列中第二小的，仍需执行$i-1$次比较；

$j=3$： x_i是这个序列中第三小的，需执行$i-2$次比较；

……

$j=i$： x_i是这个序列中最大的，需执行 1 次比较。

当$2\le j\le i$时，算法需执行的比较次数为$i-j+1$。

这i种可能性的概率相同，都是$1/i$。元素x_i插入到合适的位置的平均比较次数T_i：

$$T_i=\frac{i-1}{i}+\sum_{j=2}^{i}\frac{i-j+1}{i}$$

$$=\frac{i-1}{i}+\sum_{j=1}^{i-1}\frac{j}{i}$$

$$=1-\frac{1}{i}+\frac{1}{2}(i-1)$$

$$=\frac{1}{2}+\frac{i}{2}-\frac{1}{i}$$

分别把$x_2, x_3, \cdots\cdots, x_n$插入到序列中的合适位置，所需的平均比较总次数$T$为：

$$T=\sum_{i=2}^{n}T_i=\sum_{i=2}^{n}(\frac{1}{2}+\frac{i}{2}-\frac{1}{i})$$

$$=\frac{1}{2}(n-1)+\frac{1}{2}\sum_{i=2}^{n}i-\sum_{i=1}^{n}\frac{1}{i}+1$$

$$=\frac{1}{2}(n-1)+\frac{1}{4}(n\ (n+1)-2)+1-\sum_{i=1}^{n}\frac{1}{i}$$

$$=\frac{1}{4}(n^2+3n)-\sum_{i=1}^{n}\frac{1}{i}$$

因为：

$$\ln(n+1)\le\sum_{i=1}^{n}\frac{1}{i}\le\ln n+1$$

所以：

$$T\approx\frac{1}{4}(n^2+3n)-\ln n$$

由此可得，插入排序算法 insert_sort 在平均情况下的时间复杂性是 $\Theta(n^2)$。

例 1.18　冒泡排序算法在平均情况下的下界分析。

算法 1.15　改进的冒泡算法

输入：被排序的数组 A[],数组的元素个数 n

输出：按递增顺序排序的数组 A[]

```
template <class Type>
void bubble_sort(Type A[],int n)
{
    int i,k,flag;
    k = n - 1,   flag = 1;
    while (flag) {
        k = k – 1,  flag = 0;
        for (i=0;i<=k;i++) {
            if (A[i] > A[i+1]) {
                swap(A[i],A[i+1]);
                flag = 1;
            }
        }
    }
}
```

1、最好的情况：所有初始数据都顺序排序。至少是 $\Omega(n)$。

2、最坏的情况：所有初始数据都逆序排序。执行次数为：

$$(n-1)+(n-2)+\cdots+1=\sum_{i=1}^{n-1} i=\frac{1}{2}n(n-1)$$

最坏情况下的运行时间至多是 $O(n^2)$。

3、平均情况下运行时间下界：

定义 1.10　设 $a_1,a_2,\cdots,a_n$ 是集合 $\{1,2,\cdots,n\}$ 的一个排列，如果 $i<j$ 且 $a_i>a_j$，则对偶 (a_i,a_j) 称为该排列的一个逆序。

例：排列 3，4，1，5 有逆序 $(3,1)$ 及 $(4,1)$。若使元素按序排列，至少必须交换两次。

交换两个相邻元素，则逆序的总数将增 1 或减 1。

不断地交换两个相邻元素，使其逆序个数往减少的方向改变，当逆序个数减少为 0 时，就是一个有序的排列了。

排列中逆序的数目，是算法所执行的元素比较次数的下界。

n 个元素共有 $n!$ 种排列，所有排列的平均逆序的个数，也就是算法所执行的平均比较次数的下界。

例如，集合 $A=\{1,2,3\}$ 有如下 3!= 6 种排列：

排 列	逆序数目 k
1 2 3	0
1 3 2	1
2 1 3	1
2 3 1	2
3 1 2	2
3 2 1	3

令 $S(k)$ 是逆序个数为 k 时的排列数目，则有：

$$S(0)=1 \qquad S(1)=2 \qquad S(2)=2 \qquad S(3)=1$$

记 $mean(n)$ 为 n 个元素集合的所有排列的逆序的平均个数。则具有 3 个元素集合的逆序的平均个数为：

$$\begin{aligned} mean(3) &= \frac{1}{3!}(S(0)\cdot 0+S(1)\cdot 1+S(2)\cdot 2+S(3)\cdot 3) \\ &= \frac{1}{6}(1\cdot 0+2\cdot 1+2\cdot 2+1\cdot 3) \\ &= 1.5 \end{aligned}$$

n 个元素的集合的所有排列：

最好的情况下，所有的元素都已经是顺序排列的了，该排列的逆序个数为 0；

最坏的情况下，所有的元素都是逆序排列的，该排列的逆序个数为 $n(n-1)/2$。

逆序的平均个数为：

$$mean(n)=\frac{1}{n!}\sum_{k=0}^{n(n-1)/2} k\,S(k)$$

Donald E.Knuth 对逆序的分布规律进行了研究，他利用生成函数的性质进行了复杂的推导，得出了下面的公式：

$$\begin{aligned} mean(n) &= \sum_{k=1}^{n}\frac{k-1}{2} \\ &= \frac{1}{4}n(n-1) \end{aligned}$$

因此，冒泡排序在平均情况下的运行时间的下界是 $\Omega(n^2)$。Donald E.Knuth 对冒泡排序在平均情况下的运行时间也进行了研究，可参考文献[6]。

1.4　算法的空间复杂性

一、算法的空间复杂性，指的是为解一个问题实例而需要的存储空间。

二、两种处理方法

1、算法所需要存储空间，仅是算法所需要的工作空间。

例：线性检索算法 linear_search、叉检索算法 binary_search，空间复杂性是 $\Theta(1)$。

合并算法 merge，空间复杂性是 $\Theta(n)$。

令 $T(n)$ 和 $S(n)$ 分别表示算法的时间复杂性和空间复杂性，那么，一般情况下有 $S(n)=O(T(n))$。

2、算法所需要存储空间为算法在运行时所占用的内存空间的总和，包括存放输出入数据的变量单元、程序代码、工作变量、常数、以及运行时的引用型变量所占用的空间、及递归栈所占用的空间。

算法所需要的存储空间 S_A 可表示为：

$$S_A = c + S(n)$$

c：程序代码、常数等固定部分，

$S(n)$：是与输入规模有关的部分。输出入数据所占用的空间、工作空间、递归栈

在分析算法的空间复杂性时，主要考虑的是 $S(n)$。

在很多问题中，时间和空间是一个对立面。为算法分配更多的空间，可以使算法运行得更快。反之，当空间是一个重要因素时，有时，需要用算法的运行时间去换取空间。

1.5　最优算法

1、已知问题Π的任何算法的运行时间是 $\Omega(f(n))$，则对以时间 $O(f(n))$ 求解问题Π的任何算法，都认为是最优算法。

2、运行时间同阶的算法，常数因子小的算法，优于常数因子大的算法。

3、时间复杂性渐近阶的确定，与 n_0 及常数 c 的选取有关，当规模很小时，复杂性阶低的算法，不一定比复杂性阶高的算法更有效。

第二章　常用的数学工具

2.2　用生成函数求解递归方程

2.2.1　生成函数及其性质

一、生成函数的定义

定义 2.1　令 $a_0, a_1, a_2, \cdots$ 是一个实数序列，构造如下的函数：

$$G(z) = a_0 + a_1 z + a_2 z^2 + \cdots = \sum_{k=0}^{\infty} a_k z^k \qquad (2.2.1)$$

则函数 $G(z)$ 称为序列 $a_0, a_1, a_2, \cdots$ 的生成函数。

例：函数

$$(1+x)^n = C_n^0 + C_n^1 x + C_n^2 x^2 + \cdots + C_n^n x^n$$

则函数 $(1+x)^n$ 便是序列 $C_n^0, C_n^1, C_n^2, \cdots, C_n^n$ 的生成函数。

二、生成函数的性质

1. 加法　设 $G(z) = \sum_{k=0}^{\infty} a_k z^k$ 是序列 $a_0, a_1, a_2, \cdots$ 的生成函数，$H(z) = \sum_{k=0}^{\infty} b_k z^k$ 是序列 $b_0, b_1, b_2, \cdots$ 的生成函数，则 $\alpha G(z) + \beta H(z)$

$$\alpha G(z) + \beta H(z) = \alpha \sum_{k=0}^{\infty} a_k z^k + \beta \sum_{k=0}^{\infty} b_k z^k$$

$$= \sum_{k=0}^{\infty} (\alpha a_k + \beta b_k) z^k \qquad (2.2.2)$$

是序列 $\alpha a_0 + \beta b_0, \alpha a_1 + \beta b_1, \alpha a_2 + \beta b_2, \cdots$ 的生成函数。

2．移位　设 $G(z) = \sum_{k=0}^{\infty} a_k z^k$ 是序列 $a_0, a_1, a_2, \cdots$ 的生成函数，则 $z^m G(z)$

$$z^m G(z) = \sum_{k=m}^{\infty} a_{k-m} z^k \qquad (2.2.3)$$

是序列$0,\cdots,0,a_0,a_1,a_2,\cdots$的生成函数。

3．乘法　设$G(z)=\sum_{k=0}^{\infty}a_k z^k$是序列$a_0,a_1,a_2,\cdots$的生成函数，$H(z)=\sum_{k=0}^{\infty}b_k z^k$是序列$b_0,b_1,b_2,\cdots$的生成函数，则$G(z)H(z)$

$$\begin{aligned}G(z)H(z)&=(a_0+a_1z+a_2z^2+\cdots)(b_0+b_1z+b_2z^2+\cdots)\\&=a_0b_0+(a_0b_1+a_1b_0)z+(a_0b_2+a_1b_1+a_2b_0)z^2+\cdots\\&=\sum_{k=0}^{\infty}c_k z^k\end{aligned}\qquad(2.2.4)$$

是序列$c_0,c_1,c_2,\cdots$的生成函数，其中，$c_n=\sum_{k=0}^{n}a_k b_{n-k}$

4. z 变换　设$G(z)=\sum_{k=0}^{\infty}a_k z^k$是序列$a_0,a_1,a_2,\cdots$的生成函数，则$G(cz)$

$$\begin{aligned}G(cz)&=a_0+a_1(cz)+a_2(cz)^2+a_3(cz)^3+\cdots\\&=a_0+ca_1z+c^2a_2z^2+c^3a_3z^3+\cdots\end{aligned}\qquad(2.2.5)$$

是序列$a_0,ca_1,c^2a_2,\cdots$的生成函数。

特别地，有：

$$\frac{1}{1-cz}=1+cz+c^2z^2+c^3z^3+\cdots\qquad(2.2.6)$$

所以，$\frac{1}{1-cz}$是序列$1,c,c^2,c^3,\cdots$的生成函数。

当$c=1$时，有：

$$\frac{1}{1-z}=1+z+z^2+\cdots\qquad(2.2.7)$$

则$\frac{1}{1-z}$是序列 1,1,1,…的生成函数。

利用：

$$\frac{1}{2}(G(z)+G(-z))=a_0+a_2z^2+a_4z^4+\cdots\qquad(2.2.8)$$

可以去掉级数中的奇数项；同样，利用

$$\frac{1}{2}(G(z)-G(-z))=a_1z+a_3z^3+a_5z^5+\cdots\qquad(2.2.9)$$

可以去掉级数中的偶数项。

5．微分和积分　设$G(z)=\sum_{k=0}^{\infty}a_k z^k$是序列$a_0,a_1,a_2,\cdots$的生成函数，对$G(z)$求导数

$$G'(z)=a_1+2a_2z+3a_3z^2+\cdots=\sum_{k=0}^{\infty}(k+1)a_{k+1}z^k \qquad (2.2.10)$$

显然，$G'(z)$是序列$a_1,2a_2,3a_3,\cdots$的生成函数。同样，对$G(z)$求积分

$$\int_0^z G(t)dt=a_0z+\frac{1}{2}a_1z^2+\frac{1}{3}a_2z^3+\cdots=\sum_{k=1}^{\infty}\frac{1}{k}a_{k-1}z^k \qquad (2.2.11)$$

则积分$\int_0^z G(t)dt$是$a_0,\frac{1}{2}a_1,\frac{1}{3}a_2,\cdots$的生成函数。

如果对（2.2.7）式求导数，可得：

$$\frac{1}{(1-z)^2}=1+2z+3z^2+\cdots=\sum_{k=0}^{\infty}(k+1)z^k \qquad (2.2.12)$$

则$\frac{1}{(1-z)^2}$是算术级数1,2,3…的生成函数。

如果对（2.2.7）式求积分，可得：

$$\ln\frac{1}{1-z}=z+\frac{1}{2}z^2+\frac{1}{3}z^3+\cdots=\sum_{k=1}^{\infty}\frac{1}{k}z^k \qquad (2.2.13)$$

则$\ln\frac{1}{1-z}$是调和数$1,\frac{1}{2},\frac{1}{3},\cdots$的生成函数。

2.2.2　用生成函数求解递归方程

例 2.1　汉诺塔（Hanoi）问题。

宝石针的编号为A、B、C，A针串着64片金盘。希望把它们移到B针或C针。有可n是金盘的数量，$h(n)$是移动n个金盘的移动次数

1．当$n=1$时，　$h(1)=1$。

2．当$n=2$时，　$h(2)=2h(1)+1$。

3．当$n=3$时，　$h(3)=2h(2)+1$。

递归关系式：

$$\begin{cases}h(n)=2h(n-1)+1\\ h(1)=1\end{cases} \qquad (2.2.14)$$

用$h(n)$作为系数，构造一个生成函数：

$$G(x)=h(1)x+h(2)x^2+h(3)x^3+\cdots$$

$$=\sum_{k=1}^{\infty}h(k)x^k$$

令

$$G(x)-2xG(x)=h(1)x+h(2)x^2+h(3)x^3+\cdots-2h(1)x^2-2h(2)x^3-\cdots$$

$$=h(1)x+(h(2)-2h(1))x^2+(h(3)-2h(2))x^3+\cdots$$

由（2.2.14）及（2.2.7）式得：

$$(1-2x)G(x)=x+x^2+x^3+\cdots$$

$$=\frac{x}{1-x}$$

所以，

$$G(x)=\frac{x}{(1-x)(1-2x)}$$

令：

$$G(x)=\frac{A}{(1-x)}+\frac{B}{(1-2x)}=\frac{A-2Ax+B-Bx}{(1-x)(1-2x)}$$

有：

$$A+B=0 \qquad -2A-B=1$$

求得 $A=-1,B=1$。所以：

$$G(x)=\frac{1}{(1-2x)}-\frac{1}{(1-x)}$$

$$=(1+2x+2^2x^2+2^3x^3+\cdots)-(1+x+x^2+x^3+\cdots)$$

$$=(2-1)x+(2^2-1)x^2+(2^3-1)x^3+\cdots$$

$$=\sum_{k=1}^{\infty}(2^k-1)x^k$$

$h(n)=2^n-1$，它是式中第 n 项的系数。当 $n=64$ 时，移动次数为 $2^{64}-1$。

例 2.2　菲波那契序列问题。

$t(n)$、$T(n)$、$f(n)$表示第 n 个月小兔子、大兔子的数目，及第 n 个月兔子的总数目。则：

$$T(n)=T(n-1)+t(n-1) \qquad (2.2.15)$$

$$t(n)=T(n-1) \qquad (2.2.16)$$

$$f(n)=T(n)+t(n) \qquad (2.2.17)$$

第一式：第 n 个月大兔子的数量，为前一个月大兔子的数量加上前一个月小兔子的数量；

第二式：第 n 个月小兔子的数量，为前一个月大兔子的数量；

第三式：表示第 n 个月兔子的总量为该月大兔子的数量及小兔子的数量之和。递归方程：

$$\begin{cases} f(n)=f(n-1)+f(n-2) \\ f(1)=f(2)=1 \end{cases} \qquad (2.2.18)$$

用 $f(n)$ 作为系数，构造生成函数：

$$F(x)=f(1)x+f(2)x^2+f(3)x^3+\cdots\cdots$$

$$=\sum_{k=1}^{\infty} f(k)x^k$$

令：

$$\begin{aligned} &F(x)-xF(x)-x^2F(x) \\ &=f(1)x+f(2)x^2+f(3)x^3+\cdots-x(f(1)x+f(2)x^2+\cdots)-x^2(f(1)x+\cdots) \\ &=f(1)x+(f(2)-f(1))x^2+(f(3)-f(2)-f(1))x^3+\cdots\cdots \\ &=x \end{aligned}$$

所以，有：

$$F(x)=\frac{x}{1-x-x^2}=\frac{-x}{x^2+x+\frac{1}{4}-\frac{5}{4}}=\frac{-x}{\left(x+\frac{1}{2}\right)^2-\left(\frac{1}{2}\sqrt{5}\right)^2}$$

$$=\frac{-x}{\left(x+\frac{1}{2}(1-\sqrt{5}\right)\left(x+\frac{1}{2}(1+\sqrt{5}\right)}$$

$$=\frac{A}{x+\frac{1}{2}(1-\sqrt{5})}+\frac{B}{x+\frac{1}{2}(1+\sqrt{5})}$$

$$=\frac{Ax+\frac{1}{2}(1+\sqrt{5})A+Bx+\frac{1}{2}(1-\sqrt{5})B}{\left(x+\frac{1}{2}(1-\sqrt{5}\right)\left(x+\frac{1}{2}(1+\sqrt{5}\right)}$$

有：$A+B=-1$，$(1+\sqrt{5})A+(1-\sqrt{5})B=0$

解得：$A=\frac{1}{2\sqrt{5}}(1-\sqrt{5})$，$B=\frac{-1}{2\sqrt{5}}(1+\sqrt{5})$

把 A、B 代入 $F(x)$，得到：

$$F(x)=\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2x+1-\sqrt{5}}-\frac{1+\sqrt{5}}{2x+1+\sqrt{5}}\right)$$

$$=\frac{1}{\sqrt{5}}\left(\frac{1}{1-\frac{2x}{\sqrt{5}-1}}-\frac{1}{1-\frac{-2x}{\sqrt{5}+1}}\right)$$

令 $\alpha=\frac{2}{\sqrt{5}-1}=\frac{1}{2}(1+\sqrt{5})$，$\beta=\frac{-2}{\sqrt{5}+1}=\frac{1}{2}(1-\sqrt{5})$

则有：

$$F(x)=\frac{1}{\sqrt{5}}((\alpha-\beta)x+((\alpha^2-\beta^2)x^2+\cdots\cdots$$

所以，第 n 项系数为：

$$f(n)=\frac{1}{\sqrt{5}}(\alpha^n-\beta^n)$$

2.3 用特征方程求解递归方程

2.3.1 k 阶常系数线性齐次递归方程

一、k 阶常系数线性齐次递归方程

1、递归方程的形式：

$$\begin{cases} f(n)=a_1f(n-1)+a_2f(n-2)+\cdots a_kf(n-k) \\ f(i)=b_i \qquad\qquad 0\le i<k \end{cases} \qquad (2.3.1)$$

2、递归方程的特征方程

x^n 取代 $f(n)$：

$$x^n=a_1x^{n-1}+a_2x^{n-2}+\cdots+a_kx^{n-k}$$

两边分别除以 x^{n-k}，可得：

$$x^k=a_1x^{k-1}+a_2x^{k-2}+\cdots+a_k$$

把上式写成：

$$x^k-a_1x^{k-1}-a_2x^{k-2}-\cdots-a_k=0 \qquad (2.3.2)$$

则式（2.3.2）称为递归方程（2.3.1）的特征方程。

二、k 阶常系数线性齐次递归方程的求解

1、$q_1,q_2,\cdots,q_k$ 是特征方程的 k 个互不相同的根。则递归方程（2.3.1）的通解为：

$$f(n)=c_1q_1^n+c_2q_2^n+\cdots+c_kq_k^n \qquad (2.3.3)$$

2、特征方程的 k 个根中有 r 个重根 $q_i, q_{i+1}, \cdots, q_{i+r-1}$，递归方程（2.3.1）的通解形式为：

$$f(n)=c_1q_1^n+\cdots+c_{i-1}q_{i-1}^n+(c_i+c_{i+1}n+\cdots+c_{i+r-1}n^{r-1})q_i^n+\cdots+c_kq_k^n \qquad (2.3.4)$$

在（2.3.3）及（2.3.4）中，$c_1, c_2, \cdots, c_k$ 为待定系数。

3、求解过程：把递归方程的初始条件代入（2.3.3）或（2.3.4）中，建立联立方程，确定系数 $c_1, c_2, \cdots, c_k$，从而可求出通解 $f(n)$。

例 2.3 三阶常系数线性齐次递归方程如下：

$$\begin{cases} f(n)=6f(n-1)-11f(n-2)+6f(n-3) \\ f(0)=0 \qquad f(1)=2 \qquad f(2)=10 \end{cases}$$

解 特征方程为：

$$x^3-6x^2+11x-6=0$$

把方程改写成：

$$x^3-3x^2-3x^2+9x+2x-6=0$$

对特征方程进行因式分解，得：

$$(x-1)(x-2)(x-3)=0$$

则有特征根：

$$q_1=1 \qquad q_2=2 \qquad q_3=3$$

所以，递归方程的通解为：

$$\begin{aligned} f(n)&=c_1q_1^n+c_2q_2^n+c_3q_3^n \\ &=c_1+c_2\,2^n+c_3 3^n \end{aligned}$$

由初始条件得：

$$\begin{aligned} f(0)&=c_1+c_2+c_3=0 \\ f(1)&=c_1+2c_2+3c_3=2 \\ f(2)&=c_1+4c_2+9c_3=10 \end{aligned}$$

解此联立方程，得：

$$c_1=0 \qquad c_2=-2 \qquad c_3=2$$

则递归方程的解为：

$$f(n)=2(3^n-2^n)$$

例 2.4 三阶常系数线性齐次递归方程如下：

$$\begin{cases} f(n)=5f(n-1)-7f(n-2)+3f(n-3) \\ f(0)=1 \qquad f(1)=2 \qquad f(2)=7 \end{cases}$$

解 特征方程为：

$$x^3-5x^2+7x-3=0$$

把特征方程改写成：

$$x^3-5x^2+6x+x-3=0$$

进行因式分解：

$$(x-3)(x^2-2x+1)=0$$

最后得：

$$(x-1)(x-1)(x-3)=0$$

求得特征方程的根为：

$$q_1=1 \qquad q_2=1 \qquad q_3=3$$

所以，递归方程的通解为：

$$f(n)=(c_1+c_2\,n)\,q_1^n+c_3\,q_3^n$$
$$=c_1+c_2\,n+c_3 3^n$$

代入初始条件：

$$f(0)=c_1+c_3=1$$
$$f(1)=c_1+c_2+3c_3=2$$
$$f(2)=c_1+2c_2+9c_3=7$$

解此联立方程，得：

$$c_1=0 \qquad c_2=-1 \qquad c_3=1$$

则递归方程的解为：

$$f(n)=(c_1+c_2\,n)\,q_1^n+c_3\,q_3^n$$

$$=3^n-n$$

2.3.2 k 阶常系数线性非齐次递归方程

一、k 阶常系数线性非齐次递归方程

1、递归方程的形式：

$$\begin{cases} f(n)=a_1 f(n-1)+a_2 f(n-2)+\cdots a_k f(n-k)+g(n) \\ f(i)=b_i \qquad\qquad 0\le i<k \end{cases} \qquad (2.3.5)$$

2、通解形式：

$$f(n)=\overline{f(n)}+f^*(n)$$

其中，$\overline{f(n)}$ 是对应齐次递归方程的通解，$f^*(n)$ 是原非齐次递归方程的特解。

3、特解的求取

1） $g(n)$ 是 n 的 m 次多项式，即

$$g(n)=b_1n^m+b_2n^{m-1}+\cdots+b_mn+b_{m+1} \qquad (2.3.6)$$

其中， $b_i, i=1,2,\cdots,m+1$ 是常数。特解 $f^*(n)$ 也是 n 的 m 次多项式：

$$f^*(n)=A_1n^m+A_2n^{m-1}+\cdots+A_mn+A_{m+1} \qquad (2.3.7)$$

其中， $A_i, i=1,2,\cdots,m+1$ 为待定系数。

2） $g(n)$ 是如下形式的指数函数：

$$g(n)=(b_1n^m+b_2n^{m-1}+\cdots+b_mn+b_{m+1})a^n \qquad (2.3.8)$$

其中， a 、 $b_i, i=1,2,\cdots,m+1$ 为常数。

a） a 不是特征方程的重根，特解 $f^*(n)$ 为：

$$f^*(n)=(A_1n^m+A_2n^{m-1}+\cdots+A_mn+A_{m+1})a^n \qquad (2.3.9)$$

其中， $A_i, i=1,2,\cdots,m+1$ 为待定系数。

b） a 是特征方程的 r 重特征根，特解的形式为：

$$f^*(n)=(A_1n^m+A_2n^{m-1}+\cdots+A_mn+A_{m+1})n^ra^n \qquad (2.3.10)$$

其中， $A_i, i=1,2,\cdots,r+1$ 是待定系数。

例 2.5 二阶常系数线性非齐次递归方程如下：

$$\begin{cases} f(n)=7f(n-1)-10f(n-2)+4n^2 \\ f(0)=1 \qquad\qquad f(1)=2 \end{cases}$$

解 对应的齐次递归方程的特征方程为：

$$x^2-7x+10=0$$

把此方程转换为：

$$(x-2)(x-5)=0$$

得到特征根为：

$$q_1=2 \qquad q_2=5$$

所以，对应的齐次递归方程的通解为：

$$\overline{f(n)}=c_12^n+c_25^n$$

令非齐次递归方程的特解为：

$$f^*(n)=A_1n^2+A_2n+A_3$$

代入原递归方程，得：

$$A_1n^2+A_2n+A_3-7(A_1(n-1)^2+A_2(n-1)+A_3)+10(A_1(n-2)^2+A_2(n-2)+A_3)=4n^2$$

化简后得到：

$$4A_1n^2+(-26A_1+4A_2)n+33A_1-13A_2+4A_3=4n^2$$

由此，得到联立方程：

$$\begin{aligned}4A_1&=4\\-26A_1+4A_2&=0\\33A_1-13A_2+4A_3&=0\end{aligned}$$

解此联立方程，可得：

$$A_1=1\qquad A_2=6\frac{1}{2}\qquad A_3=12\frac{7}{8}$$

所以，非齐次递归方程的通解为：

$$f(n)=c_1 2^n+c_2 5^n+n^2+6\frac{1}{2}n+12\frac{7}{8}$$

把初始条件代入，有：

$$f(0)=c_1+c_2+12\frac{7}{8}=1$$

$$f(1)=2c_1+5c_2+20\frac{3}{8}=2$$

解此联立方程，得：

$$c_1=-13\frac{2}{3}\qquad c_2=1\frac{19}{24}$$

最后，非齐次递归方程的通解为：

$$f(n)=-13\frac{2}{3}\cdot 2^n+1\frac{19}{24}\cdot 5^n+n^2+6\frac{1}{2}n+12\frac{7}{8}$$

例 2.6 二阶常系数线性非齐次递归方程如下：

$$\begin{cases}f(n)=7f(n-1)-12f(n-2)+n2^n\\ f(0)=1\qquad\qquad f(1)=2\end{cases}$$

解 对应齐次递归方程的特征方程为：

$$x^2-7x+12=0$$

此方程可改写成：

$$(x-3)(x-4)=0$$

所以，方程的解为：

$$q_1=3\qquad q_2=4$$

齐次递归方程的通解为：

$$\overline{f(n)}=c_1 3^n+c_2 4^n$$

令非齐次递归方程的特解为：

$$f^*(n)=(A_1n+A_2)\,2^n$$

把特解代入原非齐次递归方程，得：

$$(A_1 n + A_2)2^n - 7(A_1(n-1) + A_2)2^{n-1} + 12\,(A_1(n-2) + A_2)2^{n-2} = n2^n$$

整理得：

$$2A_1 n + 2A_2 - 10A_1 = 4n$$

可得联立方程：

$$2A_1 = 4$$
$$2A_2 - 10A_1 = 0$$

解此联立方程得：

$$A_1 = 2 \qquad A_2 = 10$$

所以，非齐次递归方程的通解为：

$$f(n) = c_1 3^n + c_2 4^n + (2n+10)2^n$$

用初始条件代入：

$$f(0) = c_1 + c_2 + 10 = 1$$
$$f(1) = 3c_1 + 4c_2 + 24 = 2$$

解此联立方程得：

$$c_1 = -14 \qquad c_2 = 5$$

最后，非齐次递归方程的解为：

$$\begin{aligned} f(n) &= -14 \cdot 3^n + 5 \cdot 4^n + (2n+10)2^n \\ &= -14 \cdot 3^n + 5 \cdot 4^n + (n+5)2^{n+1} \end{aligned}$$

2.4 用递推方法求解递归方程

2.4.1 递推

非齐次递归方程：

$$\begin{cases} f(n) = b\,f(n-1) + g(n) \\ f(0) = c \end{cases} \qquad (2.4.1)$$

其中，b、c是常数，$g(n)$是n的某一个函数。直接把公式应用于式中的$f(n-1)$，得到：

$$\begin{aligned} f(n) &= b(b\,f(n-2) + g(n-1)) + g(n) \\ &= b^2\,f(n-2) + b\,g(n-1) + g(n) \\ &= b^2(b\,f(n-3) + g(n-2)) + b\,g(n-1) + g(n) \\ &= b^3 f(n-3) + b^2 g(n-2) + b\,g(n-1) + g(n) \end{aligned}$$

$$
\begin{aligned}
&= \cdots\cdots\cdots\cdots \\
&= b^n f(0) + b^{n-1} g(1) + \cdots + b^2 g(n-2) + b\, g(n-1) + g(n) \\
&= c\, b^n + \sum_{i=1}^{n} b^{n-i} g(i) \qquad (2.4.2)
\end{aligned}
$$

例 2.7 汉诺塔问题。

由（2.2.14），汉诺塔的递归方程为：

$$
\begin{cases} h(n) = 2h(n-1) + 1 \\ h(1) = 1 \end{cases}
$$

直接利用（2.4.2）式于汉诺塔的递归方程。此时，

$$
b = 2 \qquad c = 1 \qquad g(n) = 1
$$

从 n 递推到 1，有：

$$
\begin{aligned}
h(n) &= c\, b^{n-1} + \sum_{i=1}^{n-1} b^{n-1-i} g(i) \\
&= 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 + 1 \\
&= 2^n - 1
\end{aligned}
$$

2.4.2 用递推法求解变系数递归方程

一、变系数齐次递归方程：

$$
\begin{cases} f(n) = g(n) f(n-1) \\ f(0) = c \end{cases} \qquad (2.4.3)
$$

利用递推方法，容易得到：

$$
f(n) = c\, g(n)\, g(n-1) \cdots g(1) \qquad (2.4.4)
$$

例 2.8 解如下递归函数：

$$
\begin{cases} f(n) = n\, f(n-1) \\ f(0) = 1 \end{cases}
$$

由（2.4.4）式，容易得到：

$$
\begin{aligned}
f(n) &= n(n-1)(n-2) \cdots 1 \\
&= n!
\end{aligned}
$$

二、变系数非齐次递归方程：

$$
\begin{cases} f(n) = g(n) f(n-1) + h(n) \\ f(0) = c \end{cases} \qquad (2.4.5)
$$

其中，c 是常数，$g(n)$ 和 $h(n)$ 是 n 的函数。利用（2.4.5）式对 $f(n)$ 进行递推，有：

$$
f(n) = g(n) f(n-1) + h(n)
$$

$$
\begin{aligned}
&= g(n)(g(n-1)f(n-2)+h(n-1))+h(n)\\
&= g(n)g(n-1)f(n-2)+g(n)h(n-1)+h(n)\\
&= \cdots\cdots\\
&= g(n)g(n-1)\cdots g(1)f(0)+g(n)g(n-1)\cdots g(2)h(1)+\cdots\\
&\quad +g(n)h(n-1)+h(n)\\
&= g(n)g(n-1)\cdots g(1)f(0)+\frac{g(n)g(n-1)\cdots g(2)g(1)h(1)}{g(1)}+\cdots\\
&\quad +\frac{g(n)g(n-1)\cdots g(2)g(1)h(n-1)}{g(n-1)\cdots g(2)g(1)}+\frac{g(n)g(n-1)\cdots g(2)g(1)h(n)}{g(n)g(n-1)\cdots g(2)g(1)}\\
&= g(n)g(n-1)\cdots g(1)\left(f(0)+\sum_{i=1}^{n}\frac{h(i)}{g(i)g(i-1)\cdots g(1)}\right) \qquad (2.4.6)
\end{aligned}
$$

例 2.9 解如下的递归函数：

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

对方程进行递推，有：

$$
\begin{aligned}
f(n) &= n((n-1)\,f(n-2)+(n-1)!)+n!\\
&= n(n-1)\,f(n-2)+2n!\\
&= \cdots\cdots\\
&= n!\,f(0)+nn!\\
&= nn!
\end{aligned}
$$

如果直接使用公式（2.4.6），此时，$g(n)=n, h(n)=n!$，有：

$$
\begin{aligned}
f(n) &= n(n-1)\cdots 1\sum_{i=1}^{n}\frac{i!}{i(i-1)\cdots 1} \qquad f(0)=0\\
&= nn!
\end{aligned}
$$

得到同样结果。

例 2.10 解如下的递归方程

$$
\begin{cases} f(n)=2\,f(n-1)+n \\ f(0)=0 \end{cases}
$$

解 对方程进行递推，有：

$$
\begin{aligned}
f(n) &= 2(2\,f(n-2)+(n-1))+n\\
&= 2^2\,f(n-2)+2(n-1)+n\\
&= \cdots\cdots
\end{aligned}
$$

$$= 2^n\ f(0) + 2^{n-1} + 2^{n-2}2 + 2^0 n$$

$$= \sum_{i=0}^{n-1} 2^i (n-i)$$

$$= \sum_{i=1}^{n} i \cdot 2^{n-i}$$

$$= 2^n \sum_{i=1}^{n} \frac{i}{2^i}$$

由公式（2.1.24），得

$$f(n) = 2^n \left(2 - \frac{n+2}{2^n}\right) = 2^{n+1} - n - 2$$

如果直接使用公式（2.4.6），此时，$g(n) = 2, h(n) = n$，同样有：

$$f(n) = 2^n \sum_{i=1}^{n} \frac{i}{2^i} = 2^{n+1} - n - 2$$

2.4.3 换名

例 2.11 解如下的递归方程：

$$\begin{cases} f(n) = 2\,f(n/2) + n/2 - 1 \\ f(1) = 1 \end{cases}$$

其中，$n = 2^k$。

解 把n表示成k的关系，原递归方程改写为：

$$\begin{cases} f(2^k) = 2\,f(2^{k-1}) + 2^{k-1} - 1 \\ f(2^0) = 1 \end{cases}$$

再令：

$$g(k) = f(2^k) = f(n)$$

于是，原递归方程可写为：

$$\begin{cases} g(k) = 2\,g(k-1) + 2^{k-1} - 1 \\ g(0) = 1 \end{cases}$$

对上面方程进行递推，有：

$$g(k) = 2(2\,g(k-2) + 2^{k-2} - 1) + 2^{k-2} - 1$$

$$=2^2\,g(k-2)+2\cdot 2^{k-1}-2-1$$
$$=2^3\,g(k-3)+3\cdot 2^{k-1}-2^2-2-1$$
$$=\cdots\cdots\cdots$$
$$=2^k\,g(0)+k\cdot 2^{k-1}-\sum_{i=0}^{k-1}2^i$$
$$=2^k(1+\frac{k}{2}-\sum_{i=0}^{k-1}2^{i-k})$$
$$=2^k(1+\frac{k}{2}-\sum_{i=1}^{k}\frac{1}{2^i})$$
$$=2^k(1+\frac{k}{2}-(1-\frac{1}{2^k}))$$
$$=2^k(\frac{k}{2}+\frac{1}{2^k})$$
$$=\frac{1}{2}2^k k+1$$
$$=\frac{1}{2}n\log n+1$$

如果直接使用（2.4.6）式，可得：

$$f(n)=g(k)=2^k(1+\sum_{i=1}^{k}\frac{2^{i-1}-1}{2^i})$$
$$=2^k(1+\sum_{i=1}^{k}(\frac{1}{2}-\frac{1}{2^i}))$$
$$=2^k(\frac{k}{2}+\frac{1}{2^k})$$
$$=\frac{1}{2}n\log n+1$$

结果一样。

例 2.12　解如下的递归方程：

$$\begin{cases} f(n)=2\,f(n/2)+bn \\ f(1)=c \end{cases}$$

其中，b、c为常数，$n=2^k$。

解　把n表示成k的关系，原递归方程改写为：

$$\begin{cases} f(2^k)=2f(2^{k-1})+b2^k \\ f(2^0)=1 \end{cases}$$

再令：

$$g(k)=f(2^k)=f(n)$$

于是，原递归方程可写为：

$$\begin{cases} g(k)=2g(k-1)+b2^k \\ g(0)=c \end{cases}$$

直接使用（2.4.6）式，可得：

$$f(n)=g(k)=2^k(c+\sum_{i=1}^{k}\frac{b2^i}{2^i})$$

$$=2^k(c+\sum_{i=1}^{k}b)$$

$$=2^k(c+bk)$$

$$=bn\log n+cn$$

第三章　排序问题和离散集合的操作

3.1　合并排序

3.1.1　合并排序算法的实现

假定有 8 个元素，第一步，划分为四对，每一对两个元素，用 merge 算法合并成四个有序的序列；第二步，把四个序列划分成两对，用 merge 算法合并成两个有序的序列；最后，再利用 merge 算法合并成一个有序的序列。

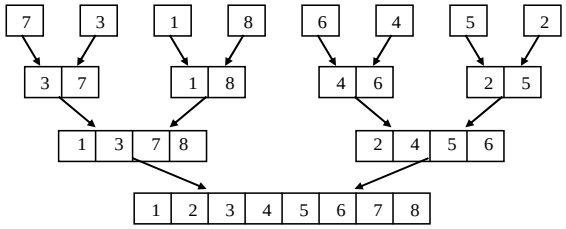


图 3.1　合并 8 个元素的过程

算法 3.1　合并排序算法

输入：具有 *n* 个元素的数组 A[]

输出：按递增顺序排序的数组 A[]

```
 1. template <class Type>
 2. void merge_sort(Type A[],int n)
 3. {
 4.     int  i,s,t = 1;
 5.     while (t<n) {
 6.         s = t; t = 2 * s;  i = 0;
 7.         while (i+t<n) {
 8.             merge(A,i,i+s-1,i+t-1,t);
 9.             i = i + t;
10.         }
11.         if (i+s<n)
12.             merge(A,i,i+s-1,n-1,n-i);
13.     }
14. }
```

i：开始合并时第一个序列的起始位置；

s：合并前序列的大小；

t：合并后序列的大小；

i、$i+s-1$、$i+t-1$定义被合并的两个序列的边界。

例如，当n=11 时，算法的工作过程如图 3.2 所示，过程如下：

1. 在第一轮循环，s=1、t=2，有 5 对 1 个元素的序列进行合并，当i=10 时，$i+t=12>n$ ，退出内部的 while 循环。但$i+s=11$，不小于n，所以，不执行第 12 行的合并工作，余留一个元素没有处理。
2. 在第二轮，$s=2,t=4$，有两对两个元素的序列进行合并，在$i=8$时，$i+t=12>n$ ，退出内部的 while 循环。但$i+s=10<n$，所以执行第 12 行的合并工作，把一个大小为 2 的序列和另外一个元素合并，产生一个 3 个元素的有序序列。
3. 在第三轮，$s=4,t=8$，有一对四个元素的序列合并，在$i=8$时，$i+t=16>n$ ，退出内部的 while 循环。而$i+s=12>n$，所以，不执行第 12 行的合并工作，余留一个序列没有处理。
4. 在第四轮，$s=8,t=16$。在$i=0$时，$i+t=16>n$，所以不执行内部的 while 循环，但$i+s=8<n$，所以执行第 12 行的合并工作，产生一个大小为 11 的有序序列。
5. 在进入第五轮时，因为$t=16>n$，所以退出外部的 while 循环，结束算法。

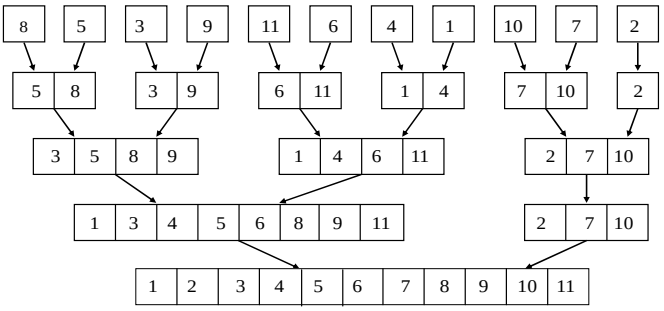


图 3.2　n=11 时的合并排序的工作过程

3.1.2　合并排序算法的分析

一、时间复杂性

假定n是 2 的幂。

外部 while 循环的循环体的执行次数：$k=\log n$次。

	内部 while 循环 merge 执行次数	merge 执行的比较次数	所产生序列数	序列长度	元素比较总次数 最少	最多
第 1 轮	$n/2$	1	$n/2$	2	$(n/2)*1$	$(n/2)*1$
第 2 轮	$n/4=n/2^2$	2，$4-1=3$	$n/2^2$	4	$(n/2^2)*2^1$	$(n/2^2)*(2^2-1)$
第 3 轮	$n/2^3$	4，$8-1=7$	$n/2^3$	8	$(n/2^3)*2^2$	$(n/2^3)*(2^3-1)$
第 j 轮	$n/2^j$	2^{j-1}，2^j-1	$n/2^j$	2^j	$(n/2^j)*2^{j-1}$	$(n/2^j)*(2^j-1)$

合并排序算法的执行时间，至少为：

$$\sum_{j=1}^{k}\frac{n}{2^j}\cdot 2^{j-1}=\sum_{j=1}^{k}\frac{n}{2}$$

$$=\frac{1}{2}k\,n$$

$$=\frac{1}{2}n\log n$$

至多为：

$$\sum_{j=1}^{k}\frac{n}{2^j}(2^j-1)=\sum_{j=1}^{k}(n-\frac{n}{2^j})$$

$$=k\,n-n\sum_{j=1}^{k}\frac{1}{2^j}$$

$$=k\,n-n\left(1-\frac{1}{2^k}\right)$$

$$=k\,n-n\left(1-\frac{1}{n}\right)$$

$$=n\log n-n+1$$

合并排序算法的运行时间，是$\Omega(n\log n)$，也是$O(n\log n)$，因此，是$\Theta(n\log n)$。

二、空间复杂性

每调用一次 merge 算法，便分配一个适当大小的缓冲区，退出 merge 算法便释放它。在最后一次调用 merge 算法时，所分配的缓冲区最大，此时，它把两个序列合并成一个长度为n的序列，需要$\Theta(n)$个工作单元。所以，合并排序算法所使用的工作空间为$\Theta(n)$。

3.2　基于堆的排序

3.2.1　堆

一、堆的定义

定义 3.2　n个元素称为堆，当且仅当它的关键字序列$k_1,k_2,\cdots,k_n$满足：

$$k_i\le k_{2i}\quad k_i\le k_{2i+1}\qquad 1\le i\le\lfloor n/2\rfloor \qquad (3.2.1)$$

或者满足：

$$k_i\ge k_{2i}\quad k_i\ge k_{2i+1}\qquad 1\le i\le\lfloor n/2\rfloor \qquad (3.2.2)$$

把满足(3.2.1)式的堆称为最小堆(min_heaps)；把满足(3.2.2)式的堆称为最大堆

(max_heaps)。

二、堆的性质：可看成是一棵完全二叉树。如果树的高度为 d

1．所有的叶结点不是处于第 d 层，就是处于第 $d-1$ 层；

2．当 $d \geq 1$ 时，第 $d-1$ 层上有 2^{d-1} 个结点；

3．第 $d-1$ 层上如果有分支结点，则这些分支结点都集中在树的最左边；

4．每个结点所存放元素的关键字，都大于（最大堆）或小于（最小堆）它子孙结点所存放元素的关键字。

三、用数组 H 存放具有 n 个元素的堆

1．根结点存放在 $H[1]$；

2．假定结点 x 存放在 $H[i]$，如果它有左儿子结点，则它的左儿子结点存放在 $H[2i]$；如果它有右儿子结点，则它的右儿子结点存放在 $H[2i+1]$；

3．非根结点 $H[i]$ 的父亲结点存放在 $H[\lfloor i/2 \rfloor]$。

例：

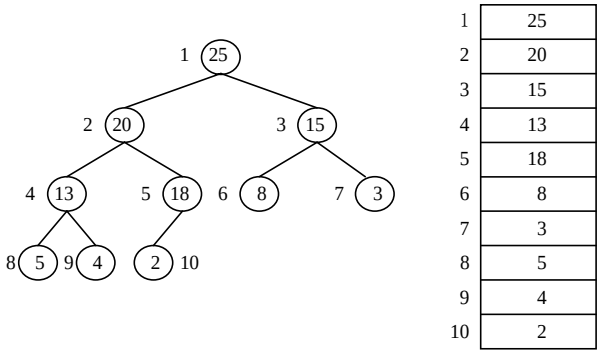


图 3.3　堆及其数组表示

3.2.2　堆的操作

一般来说，对于堆这样的数据结构，需要下面几种操作：

- void sift_up(Type H[ ]，int i)；　　把堆中的第 i 个元素上移
- void sift_down(Type H[ ]，int n，int i)；　　把堆中的第 i 个元素下移
- void insert(Type H[ ]，int &n，Type x)；　　把元素 x 插入堆中
- void delete(Type H[ ]，int &n，int i)；　　删去堆中第 i 个元素
- Type delete_max(Type H[ ]，int &n)；　　从非空的最大堆中删除并回送关键字最大的元素
- void make_head(Type H[ ]，int n)；　　使数组 H 中的元素按堆的结构重新组织

3.2.2.1　元素上移操作

沿 $H[i]$ 到根的路线，把 $H[i]$ 向上移动。移动过程中，如果大于它的父亲结点，就与父亲结点交换位置。否则，操作结束。

算法 3.2 元素上移操作

输入：数组 H[]及被上移的元素下标 i

输出：维持堆的性质的数组 H[]

```
 1. template <class Type>
 2. void sift_up(Type H[],int i)
 3. {
 4.    BOOL done = FALSE;
 5.    if (i!=1) {
 6.       while (!done && i!=1) {
 7.          if (H[i] > H[i/2])
 8.             swap(H[i],H[i/2]);
 9.          else  done = TRUE;
10.          i = i / 2;
11.       }
12.    }
13. }
```

执行时间：共$\lfloor \log n \rfloor$层，每层一个元素比较操作，$O(\log n)$

工作单元： $\Theta(1)$

例 3.1 如果在图 3.3 中，把结点 9 的内容修改为 28 的工作过程。

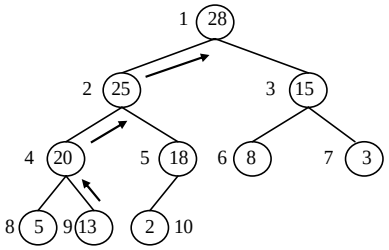


图 3.4 sift_up 操作的工作过程

3.2.2.2 元素下移操作

在向下移动的过程中，把它的关键字和它两个儿子中关键字大的儿子比较，如果小于它儿子结点的关键字，就与儿子结点交换位置。否则，操作结束。

算法 3.3 元素下移操作

输入：数组 H[],数组的元素个数 n,被下移的元素下标 i

输出：维持堆的性质的数组 H[]

```
 1. template <class Type>
 2. void sift_down(Type H[],int n,int i)
 3. {
```

```
 4.    BOOL done = FALSE;
 5.    if ((2*i)<=n) {
 6.       while (!done && (i=2*i<=n)) {
 7.          if (i+1<=n && H[i+1]>H[i])
 8.             i = i + 1;
 9.          if (H[i/2] < H[i])
10.             swap(H[i/2],H[i]);
11.          else done = TRUE;
12.       }
13.    }
14. }
```

执行时间：共$\lfloor \log n \rfloor$层，每层两个元素比较操作$O(\log n)$

工作单元：$\Theta(1)$

例 3.2　如果在图 3.3 中，把结点 2 的内容由 20 改为 1 的工作过程。

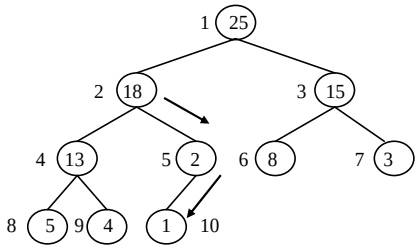


图 3.5　sift_down 操作的工作过程

3.2.2.3　元素插入操作

堆的大小增 1，把 x 放到堆的末端，对 x 做上移操作。借助于 sift_up 操作，既把元素插入堆中，又维持了堆的性质。

算法 3.4　元素插入操作

输入：数组 H[],数组的元素个数 n,被插入的元素 x

输出：维持堆的性质的数组 H[],及插入后的元素个数 n

```
1. template <class Type>
2. void insert(Type H[],int &n,Type x)
3. {
4.    n = n + 1;
5.    H[n] = x;
6.    sift_up(H,n);
7. }
```

执行时间： sift_up 操作的执行时间， $O(\log n)$

工作单元： $\Theta(1)$ 。

3.2.2.4 元素删除操作

为删除堆中的元素 $H[i]$，用堆中最后一个元素取代 $H[i]$，堆的大小减一。再根据被删除元素和取代它的元素的大小，确定对取代它的元素是做上移操作、还是做下移操作，

算法 3.5 元素删除操作

输入：数组 H[],数组的元素个数 n,被删除元素的下标 i

输出：维持堆的性质的数组 H[],及删除后的元素个数 n

```
 1. template <class Type>
 2. void delete(Type H[],int &n,int i)
 3. {
 4.    Type  x,y;
 5.    x = H[i];   y = H[n];
 6.    n = n - 1;
 7.    if (i<=n) {
 8.       H[i] = y;
 9.       if (y>=x)
10.          sift_up(H,i);
11.       else
12.          sift_down(H,n,i);
13.    }
14. }
```

执行时间： sift_up 操作、或 sift_down 操作的执行时间， $O(\log n)$

工作单元： $\Theta(1)$

3.2.2.5 删除关键字最大的元素

在最大堆中，关键字最大的元素位于根结点，借助 delete 操作，既做删除操作，又维持堆的性质。

算法 3.6 删除关键字最大元素

输入：数组 H[],数组的元素个数 n

输出：维持堆的性质的数组 H[],被删除的元素、及删除后的元素个数 n

```
 1. template <class Type>
 2. Type delete_max(Type H[],int &n)
 3. {
 4.    Type  x;
 5.    x = H[1];
```

```
6.    delete(H[],n,1);
7.    return x;
8. }
```

执行时间：$O(\log n)$

工作单元： $\Theta(1)$

3.2.3 堆的建立

一、建造堆的两种方法

1、用 insert 操作建造堆

算法 3.7 建造堆的第一种算法

输入：数组 A[],数组的元素个数 n

输出：n 个元素的堆 H[]

```
1. template <class Type>
2. void make_heap1(Type A[],Type H[],int n)
3. {
4.    int i,m = 0;
5.    for (i=0;i<n;i++)
6.       insert(H,m,A[i]);
7. }
```

执行时间：插入第 i 个元素需花费 $O(\log i)$，插入 n 个元素，需花费 $O(n\log n)$ 时间

工作单元：$\Theta(n)$

2、把数组本身构造成一个堆。

调整过程：从最后一片树叶找到它上面的分支结点，从这个分支结点开始作下移操作，一直到根结点为止。

算法 3.8 建造堆的第二种算法

输入：数组 H[],数组的元素个数 n

输出：n 个元素的堆 A

```
1. template <class Type>
2. void make_heap(Type A[],int n)
3. {
4.    int i;
5.    A[n] = A[0];
6.    for (i=n/2;i>=1;i--)
7.       sift_down(A,i);
```

```
8. }
```

例 3.3　图 3.6 表示把一个具有 11 个元素的数组，调整成一个堆的过程。

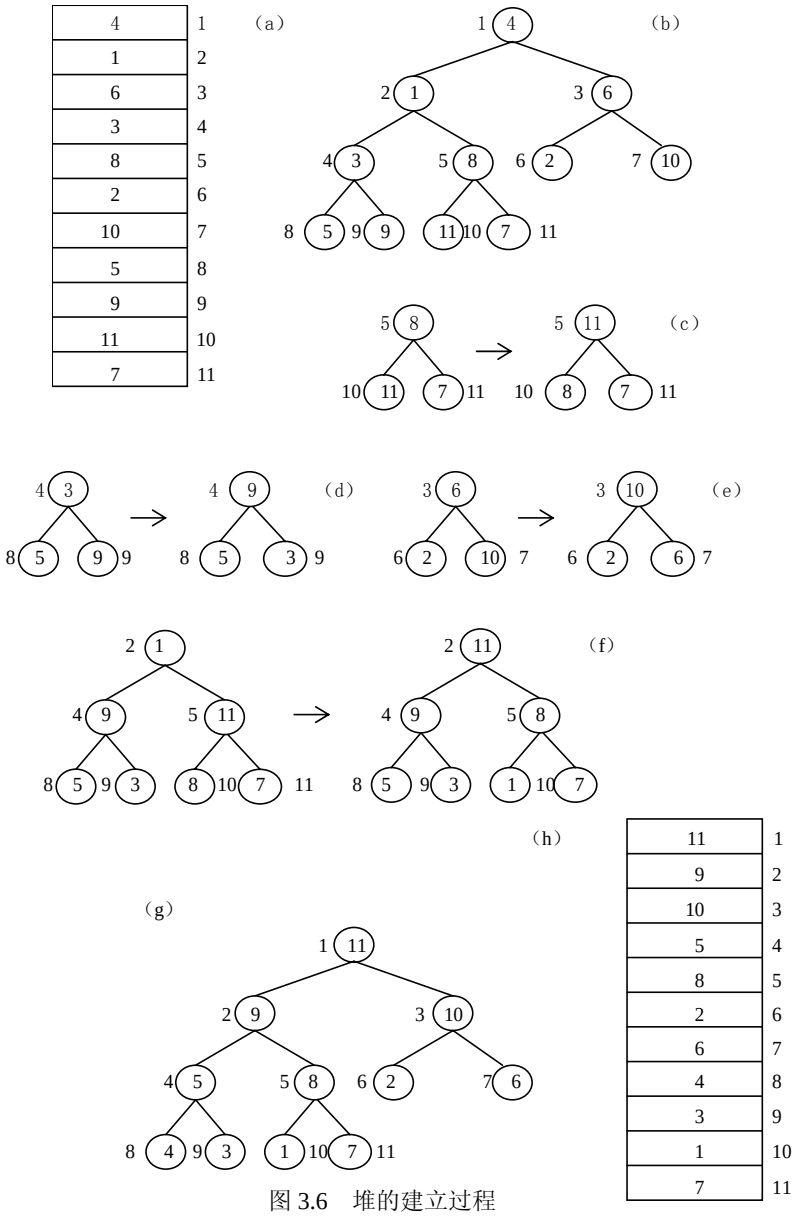


图 3.6　堆的建立过程

二、算法 make_heap 的运行时间分析

1. 数组有 n 个元素，所构成的二叉树的高度为 $k = \lfloor \log n \rfloor$；
2. 第 i 层的元素 $A[j]$ 最多下移 $k-i$ 层，最多执行 $2(k-i)$ 次元素比较；
3. 第 i 层上共有 2^i 个结点，第 i 层上所有结点最多执行 $2(k-i)2^i$ 次元素比较；
4. 第 k 层上的元素，都是叶子结点，无需执行下移操作。最多只需对第 0 层到第 $k-1$ 层

的元素执行下移操作。

由此，算法 make_heap 所执行的元素比较次数为：

$$\sum_{i=0}^{k-1} 2(k-i)2^i = 2k\sum_{i=0}^{k-1} 2^i - 2\sum_{i=0}^{k-1} i\,2^i$$

如果令 $n=2^k$，即 $k=\log n$。由公式（2.1.20）及（2.1.23），有：

$$\begin{aligned}\sum_{i=0}^{k-1} 2(k-i)2^i &= 2k(2^k-1)-2((k-1)2^{k+1}-(k-1)2^k-2^k+2)\\ &= 2(k2^k-k)-2(k2^k-2^{k+1}+2)\\ &= 4\cdot 2^k-2k-4\\ &= 4n-2\log n-4\\ &< 4n\end{aligned}$$

所以执行时间为 $O(n)$。

共 $\lfloor n/2 \rfloor$ 个结点作下移操作，至少需要 $2\lfloor n/2 \rfloor$ 次元素比较。所以执行时间是 $\Omega(n)$。

make_heap 的执行时间是 $\Theta(n)$。

工作单元个数为 $\Theta(1)$。

3.2.4 堆的排序

一、算法描述

算法 3.9 基于堆的排序

输入：数组 H[],数组的元素个数 n

输出：按递增顺序排序的数组 A[]

```
 1. template <class Type>
 2. void heap_sort(Type A[],int n)
 3. {
 4.    int  i;
 5.    make_heap(A,n);
 6.    for (i=n,i>1;i--) {
 7.       swap(A[1],A[i]);
 8.       sift_down(A,i-1,1);
 9.    }
10. }
```

二、算法分析

1、执行时间：make_heap 的执行时间为 $\Theta(n)$

sift_down 执行 $n-1$ 次，每次花费 $O(\log n)$ 时间，总花费时间 $O(n\log n)$。

所以， heap_sort 的运行时间是 $O(n\log n)$

3、工作空间：$\Theta(1)$。

3.3 基数排序

基于比较的排序算法，下界为 $\Omega(n\log n)$。

基数排序方法可以按线性时间运行。

3.3.1 基数排序算法的思想方法

n 个元素的链表 $L=\{a_1,a_2,\cdots\cdots,a_n\}$，每个元素关键字的值有如下形式：

$$d_k d_{k-1}\cdots\cdots d_1 \qquad 0\le d_i\le 9,\ \ 1\le i\le k$$

1、$m=1$

2、按关键字的数字 d_m，把元素分布到 10 个链表 $L_0,L_1,\cdots\cdots,L_9$，使得关键字的 $d_m=i$ 的元素，都分布在链表 L_i 中；

3、把 10 个链表，按照链表的下标由 0 到 9 的顺序重新链接成一个新的链表 L。

4、$m=m+1$，若 $m\le k$，转 2，否则结束

例 3.4 假设链表 L 中有如下 10 个元素，其关键字值分别为：3097、3673、2985、1358、6138、9135、4782、1367、3684、0139。

第一步，按关键字中的数字 d_1，把 L 中的元素分布到链表 $L_0\sim L_9$ 的情况如下：

L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
		4782	3673	3684	2985		3097	1358	0139
					9135		1367	6138	

把 $L_0\sim L_9$ 的元素顺序链接到 L 后，在 L 中的元素顺序如下：

L ： 4782 3673 3684 2985 9135 3097 1367 1358 6138 0139

第二步，按数字 d_2，把 L 中的元素分布到 $L_0\sim L_9$ 的情况如下：

L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
			9135		1358	1367	3673	4782	3097
			6138					3684	
			0139					2985	

把 $L_0\sim L_9$ 的元素顺序链接到 L 后，在 L 中的元素顺序如下：

L ： 9135 6138 0139 1358 1367 3673 4782 3684 2985 3097

第三步，按数字 d_3，把 L 中的元素分布到 $L_0\sim L_9$ 的情况如下：

L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
3097	9135		1358			3673	4782		2985
	6138		1367			3684			
	0139								

把 $L_0 \sim L_9$ 的元素顺序链接到 L 后，在 L 中的元素顺序如下：

L ： 3097 9135 6138 0139 1358 1367 3673 3684 4782 2985

第四步，按数字 d_4，把 L 中的元素分布到 $L_0 \sim L_9$ 的情况如下：

L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
0139	1358	2985	3097	4782		6138			9135
	1367		3673						
			3684						

把 $L_0 \sim L_9$ 的元素顺序链接到 L 后，在 L 中的元素顺序如下：

L ： 0139 1358 1367 2985 3097 3673 3684 4782 6138 9135

在第四步之后，链表中的所有关键字都已经排序了。

3.3.2 基数排序算法的实现

用双循环链表，用成员变量 prior 指向前一个元素，用成员变量 next 指向下一个元素。

算法 3.10 基数排序

输入：存放元素的链表 L，元素个数 n，及关键字的数字位数 k

输出：按递增顺序排序的链表 L

```
 1. template <class Type>
 2. void radix_sort(Type *L,int k)
 3. {
 4.     Type  *Lhead[10],*p;
 5.     int  i,j;
 6.     for (i=0;i<10;i++)              /* 分配 10 个链表的头结点 */
 7.        Lhead[i]  = new Type;
 8.     for (i=0;i<k;i++) {
 9.        for (j=0;j<10;j++)           /* 把 10 个链表置为空表 */
10.           Lhead[j]->prior = Lhead[j]->next = Lhead[j];
11.        while (L->next!=L) {
12.           p = del_entry(L);        /* 取 L 的第一个元素于 p 并把它从 L 删去 */
13.           j = get_digital(p,i);    /* 从 p 所指向的元素关键字取第 i 个数字 */
14.           add_entry(Lhead[j],p);   /* 把 p 加入链表 Lhead[j]的表尾 */
15.        }
```

```
16.        for (j=0;j<10;j++)
17.            append(L,Lhead[j]);     /* 把 10 个链表的元素链接到 L */
18.    }
19.    for (i=0;i<10;i++)              /* 释放 10 个链表的头结点 */
20.        delete(Lhead[i]);
21. }
```

算法 3.11 取下并删去双循环链表的第一个元素

输入：链表的头结点指针 L

输出：被取下第一个元素的链表 L,指向被取下元素的指针，

```
 1. template <class Type>
 2. Type *del_entry(Type *L)
 3. {
 4.    Type  *p;
 5.    p = L->next;
 6.    if (p!=L) {
 7.        p->prior->next = p->next;
 8.        p->next->prior = p->prior;
 9.    }
10.    else p = NULL;
11.    retuen p;
12. }
```

算法 3.12 把一个元素插入双循环链表的表尾

输入：链表头结点的指针 L,被插入元素的指针 p

输出：插入了一个元素的链表 L

```
 1. template <class Type>
 2. void add_entry(Type *L,Type *p)
 3. {
 4.    p->prior = L->prior;
 5.    p->next = L;
 6.    L->prior->next = p;
 7.    L->prior = p;
 8. }
```

算法 3.13 取 p 所指向元素关键字的第 i 位数字(最低位为第 0 位)

输入：指向某元素的指针 p,该元素关键字的的第 i 位数字

输出：该元素关键字的的第 i 位数字

```
 1. template <class Type>
```

```
 2. int get_digital(Type *p,int i)
 3. {
 4.    int key;
 5.    key = p->key;
 6.    if (i!=0)
 7.       key = key / power(10,i);
 9.    return key % 10;
10. }
```

算法 3.14 把链表 L1 附加到链表 L 的末端

输入：指向链表 L 及 L1 的头结点指针

输出：附加了新内容的链表 L

```
 1. template <class Type>
 2. void append(Type *L,Type *L1)
 3. {
 4.    if (L1->next!=L1) {
 5.       L->prior->next = L1->next;
 6.       L1->next->prior = L->prior;
 7.       L1->prior->next = L;
 8.       L->prior = L1->prior;
 9.    }
10. }
```

算法 3.11、3.12、3.14 的执行时间是常数时间。

算法 3.13 的执行时间取决于函数 power(x,y)的执行时间，power 函数计算以 x 为底的 y 次幂。假定，x 是有限长度的整数，后面将说明，该函数的执行时间将是 $\Theta(\log y)$，如果 y 是一个大于 0 的常整数，则该函数的执行时间也是常数。所以，它们都是 $\Theta(1)$。

3.3.3 基数排序算法的分析

一、复杂性分析

算法的执行时间是 $\Theta(kn)$。当 k 是常数时，它的执行时间是 $\Theta(n)$。

工作单元为 $\Theta(1)$。

二、正确性证明

用归纳法证明，算法经过 k 步（假定元素的关键字有 k 位数字）的重新分布和重新链接之后，序列中的元素是按顺序排列的：

$i=1$：L 中的元素按其关键字的最低位数字分布到 10 个链表，然后，再把这些链表按顺序链接成一个链表 L，则 L 中的元素将按其关键字的最低数字排序；

$i=2$： L 中的元素再按其关键字的十位数字分布到 10 个链表

令 x 和 y 是 L 序列中任意两个元素，

x 的关键字的最低两位数字分别为 a 、 b ，

y 的关键字的最低两位数字分别为 c 、 d 。

1）若 $a>c$ ，则 x 被分布到序号较高的链表， y 被分布到序号较低的链表。

重新链接到 L 去时， y 先于 x 被链接到 L ，

它们是按最低两位数字的顺序排序的。

2）若 $c>a$ 同理可证。

3） $a=c$ ，则它们分布在同一个链表。

这时，若 $b>d$ ，则 y 先于 x 被分布到这个链表。

重新链接到 L 去时，仍维持这个顺序，

它们也按最低两位数字的顺序排列。

x 和 y 是任意的，所以，链表中的元素都按最低两位数字的顺序排列。

归纳步的证明类似，留作练习。

3.4 离散集合的操作

例：对集合 $S=\{1,2,\cdots,8\}$ 定义如下的等价关系：

$$R=\{<x,y>|x\in S\wedge y\in S\wedge(x-y)\%3=0\}$$

求 S 关于 R 的等价类，

1. 初始化：$\{1\}\ \{2\}\ \{3\}\ \{4\}\ \{5\}\ \{6\}\ \{7\}\ \{8\}$；
2. $1R4$，有：$\{1,4\}\ \{2\}\ \{3\}\ \{5\}\ \{6\}\ \{7\}\ \{8\}$；
3. $4R7$，有：$\{1,4,7\}\ \{2\}\ \{3\}\ \{5\}\ \{6\}\ \{8\}$；
4. $2R5$，有：$\{1,4,7\}\ \{2,5\}\ \{3\}\ \{6\}\ \{8\}$；
5. $5R8$，有：$\{1,4,7\}\ \{2,5,8\}\ \{3\}\ \{6\}$；
6. $3R6$，有：$\{1,4,7\}\ \{2,5,8\}\ \{3,6\}$；

find 操作：把元素 x 和 y 所在的集合找出来，

union 操作：把两个集合合并成一个集合。

3.4.1 离散集合的数据结构

一、第一种数据结构

```
struct Tree_node {
    struee Tree_node *p;        /* 指向父亲结点的指针 */
    Type x;                     /* 存放在结点中的元素 */
}
```

集合可以由集合中的元素来命名，这个元素就称为该集合的代表元。

集合中的所有元素，都有资格作为集合的代表元。

要把元素 *x* 所代表的集合，与元素 *y* 所代表的集合合并起来，只要分别找出元素 *x* 和元素 *y* 所在集合的根结点，使元素 *y* 的根结点的父指针指向元素 *x* 的根结点即可。

图 3.7（a）表示由集合{1, 3, 5, 8}, {2, 7, 10}, {4, 6}, {9} 所组成的森林；

图 3.7（b）表示由元素 1 所代表的集合、与元素 7 所代表的集合合并的例子。

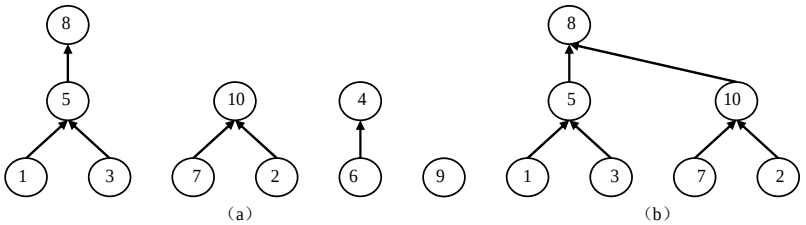


图 3.7　离散集合的表示形式

由此，可以把离散集合中 find 操作和 union 操作的含义定义如下：

- find(x)：寻找元素 *x* 所在集合的根结点；
- union(x, y)：把元素 *x* 和元素 *y* 所在集合合并成一个集合。

缺点：树的高度可能很大，变成退化树，成为线性表。如图 3.8(a)。

find 操作可能需要 $\Omega(n)$ 时间。

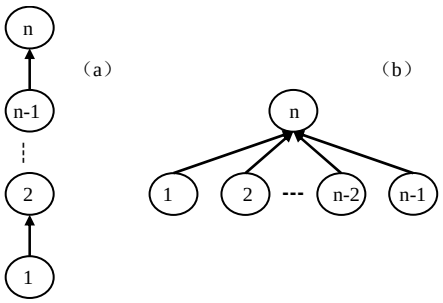


图 3.8　n 个集合合并的两种情况

二、改进的数据结构

```
struct Tree_node {
    struee Tree_node *p;          /* 指向父亲结点的指针 */
    int rank;                     /* 结点的秩 */
    Type x;                       /* 存放在结点中的元素 */
};
typedef struct Tree_node NODE;
```

结点的秩等于以该结点作为子树的根时，该子树的高度。

union(x, y)操作：令 x 和 y 是当前森林中两棵不同树的根结点，

如果 $rank(x) > rank(y)$，就把 x 作为 y 的父亲，并使 $rank(y)$ 加 1

例：图 3.8（b）表示采用这个方法对 n 个集合进行合并时的情况。

三、用数组存放元素

```
struct Tree_node {
   int index;                    /* 指向父亲结点的下标 */
   int rank;                     /* 结点的秩 */
   Type x;                       /* 存放在结点中的元素 */
};
struct Tree_node node[n];
```

这时，父结点的指针，用该结点在数组中的下标表示。

3.4.2 union、find 操作及路径压缩

一、路径压缩

find 操作时，找到根结点 y 之后，再沿着这条路径，改变路径上所有结点的父指针，使其直接指向 y 。如图 3.9 所示。

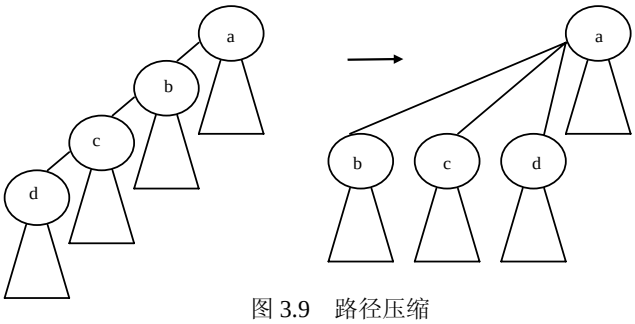


图 3.9　路径压缩

二、算法描述

算法 3.15　离散集合的 find 操作

输入：指向结点 x 的指针 xp

输出：指向结点 x 所在集合的根结点的指针 yp

```
1. NODE *find(NODE *xp)
2. {
3.     NODE *wp, *yp = xp, *zp = xp;
4.     while (yp->p!=NULL) {              /* 寻找 xp 所在集合的根结点 yp */
5.         yp = yp->p;
```

```
 6.    while (zp->p!= NULL) {            /* 路径压缩 */
 7.        wp = zp->p
 8.        zp->p = yp;
 9.        zp = wp;
10.    }
11.    return yp;
12. }
```

算法 3.16　离散集合的 union 操作

输入：指向结点 x 和结点 y 的指针 xp 和 yp

输出：结点 x 和结点 y 所在集合的并集,指向该并集根结点的指针

```
 1. NODE *union(NODE *xp,NODE *yp)
 2. {
 3.    NODE *up,*vp;
 4.    up = find(xp);
 5.    vp = find(yp);
 6.    if (up->rank<=vp->rank) {
 7.        up->p = vp;
 8.        if (up->rank==vp->rank)
 9.            vp->rank++;
10.        up = vp;
11.    }
12.    else
13.        vp->p = up;
14.    return up;
15. }
```

例 3.5　集合{1, 2, 3, 4}, {5, 6, 7, 8}，如图 3.10（a）所示，在执行了 union（1, 5）之后，结果如图 3.10（b）所示。在 union 操作中，对结点 1 和 5 执行了 find 操作，结点 1 和 5 的路径都被压缩了。

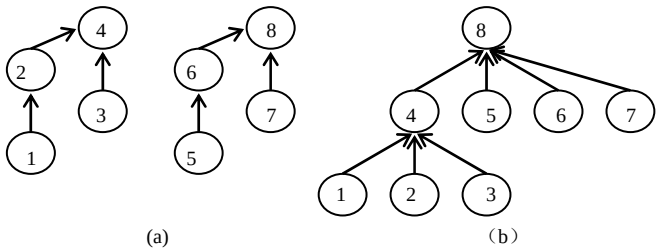


图 3.10　集合 union 操作的例子

三、算法分析

x 是树中的任意结点， $x.p$ 指向 x 的父亲结点。得到下面两个结论。

1、**结论 3.1** $x.p->rank \geq x.rank+1$ 。

2、**结论 3.2** $x.rank$ 的初始值为 0，在一系列的 union 操作中递增，直到 x 不再是树的根结点为止。一旦 x 变为另一个结点的儿子，它的秩就不再改变。

3、**引理 3.1** 若结点 x 的秩为 $x.rank$ ，则以 x 为根的树，其结点数至少为 $2^{x.rank}$ 。

（含义：结点数至少为 $n=2^{x.rank}$ 的树，其高度至多为 $\log n = x.rank$ ）

证明 用归纳法证明。

1. 开始时， x 本身是一棵数，其秩 $x.rank=0$ ，其结点数等于 $2^0=1$ ，引理成立。
2. 假定 x 和 y 分别是两棵树的根结点，其秩分别是 $x.rank$ ，和 $y.rank$ 。

   在 union(x, y)操作之前， x 和 y 为根的树，其结点数分别至少为 $2^{x.rank}$ 和 $2^{y.rank}$ 。

   在 union(x, y)操作之后，有三种情况：

   (1) 若 $x.rank < y.rank$ ，在 union 操作之后，新的树以 y 为根结点，且 y 的秩不变，而树的结点数增加。因此，新树的结点数至少为 $2^{y.rank}$ 。引理成立。

   (2) 若 $x.rank > y.rank$ ，同理可证。

   (3) 若 $x.rank = y.rank$ ，则两棵树的结点数至少都是 $2^{y.rank}=2^{x.rank}$ 。

   在 union 操作之后，新树的结点数至少为 $2\cdot 2^{y.rank}=2^{y.rank+1}=2^{x.rank+1}$ 。

   若新树以 y 为根结点，则 y 的秩 $y.rank$ 增 1；

   否则， x 的秩 $x.rank$ 增 1；在这两种情况下，引理都成立。

4、**结论 3.3** find 操作的执行时间为 $O(\log n)$ 。

证明：如果 x 是树的根， x 的秩就是树的高度。

根据引理 3.1，结点数为 n ，则该树的高度至多为 $\lfloor \log n \rfloor$ 。

find 操作最多执行 $\log n$ 次判断根结点的操作、

以及 $\log n$ 次对非根结点进行的路径压缩操作

5、**结论 3.4** union 操作的执行时间为 $O(\log n)$ 。

证明： union 操作除了执行两次 find 操作外，其余花费 $O(1)$ 时间。

6、**定理 3.1** 连续执行 m 次 union 和 find 操作，在最坏情况下，所需要的执行时间是 $O(m\log^* n)\approx O(m)$ 。

其中， $\log^* n$ 定义为：

$$\log^* n=\begin{cases} 0 & n=0,1 \\ \min\{i\geq 0 \mid \underbrace{\log\log\cdots\log}_{i\text{次}} n\leq 1\} & n\geq 2 \end{cases}$$

例如， $\log^* 2=1$ ， $\log^* 2^2=2$ ， $\log^* 2^4=3$ ， $\log^* 2^{16}=4$ ， $\log^* 2^{65536}=5$ 。在几乎所有的实际应用中， $\log^* n\leq 5$ 。所以，它所需要的执行时间实际上将是 $O(m)$ 。

回溯法的思想方法

问题的解空间和状态空间树

一、解空间

问题的解向量为 $X=(x_1,x_2,\cdots,x_n)$。 x_i 的取值范围为有穷集 S_i。把 x_i 的所有可能取值组合，称为问题的解空间。每一个组合是问题的一个可能解

例：0/1 背包问题， $S=\{0,1\}$，当 $n=3$ 时，0/1 背包问题的解空间是：

$$\{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}$$

当输入规模为 n 时，有 2^n 种可能的解。

例：货郎担问题， $S=\{1,2,\cdots,n\}$，当 $n=3$ 时， $S=\{1,2,3\}$。货郎担问题的解空间是：

$$\{(1,1,1),(1,1,2),(1,1,3),(1,2,1),(1,2,2),(1,2,3),\text{---},(3,3,1),(3,3,2),(3,3,3)\}$$

当输入规模为 n 时，它有 n^n 种可能的解。

考虑到约束方程 $x_i \neq x_j$。因此，货郎担问题的解空间压缩为：

$$\{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$$

当输入规模为 n 时，它有 $n!$ 种可能的解。

二、状态空间树：问题解空间的树形式表示

当 $n=4$ 时，货郎担问题的状态空间树。

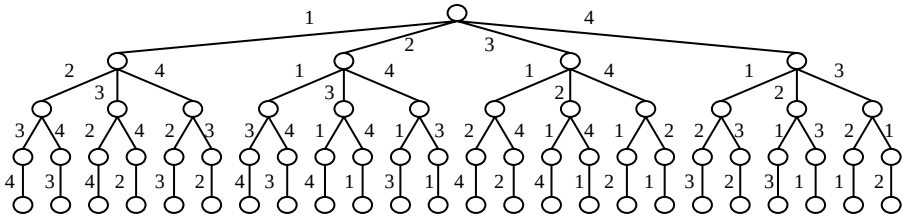


图 7.1　n=4 时货郎担问题的状态空间树

$n=4$ 时，0/1 背包问题的状态空间树

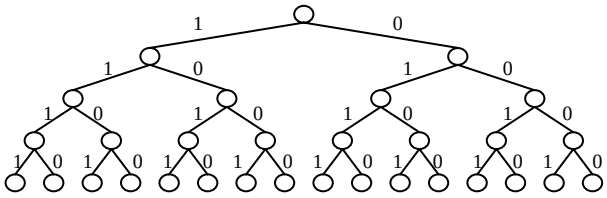


图 7.2　n=4 时背包问题的状态空间树

状态空间树的动态搜索

一、可行解和最优解

可行解：满足约束条件的解，解空间中的一个子集

最优解：使目标函数取极值（极大或极小）的可行解，一个或少数几个

例：货郎担问题，有 n^n 种可能解。$n!$种可行解，只有一个或几个解是最优解。

例：背包问题，有 2^n 种可能解，有些是可行解，只有一个或几个是最优解。

有些问题，只要可行解，不需要最优解，例如八后问题和图的着色问题

二、状态空间树的动态搜索

l _结点（活结点）：所搜索到的结点不是叶结点，且满足约束条件和目标函数的界，其儿子结点还未全部搜索完毕，

e _结点（扩展结点）：正在搜索其儿子结点的结点，它也是一个 l _结点；

d _结点（死结点）：不满足约束条件、目标函数、或其儿子结点已全部搜索完毕的结点、或者叶结点，。以 d _结点作为根的子树，可以在搜索过程中删除。

例 7.1　有 4 个顶点的货郎担问题，其费用矩阵如图 7.3 所示，求从顶点 1 出发，最后回到顶点 1 的最短路线。

∞	∞	1	7
8	∞	5	1
7	2	∞	6
2	5	3	∞

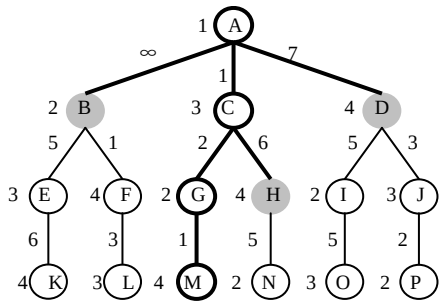


图 7.3　4 个顶点的货郎担问题的费用矩阵及搜索树

回溯法的一般性描述

题的解向量 $X=(x_0,x_1,\cdots,x_{n-1})$，

x_i 的取值范围 S_i，　$S_i=\{a_{i.0},a_{i.1},\cdots,a_{i.m_i}\}$。

问题的解空间由笛卡尔积 $A=S_0\times S_1\times\cdots\times S_{n-1}$ 构成。

状态空间树看成为一棵高度为 n 的树，

第 0 层有 $|S_0|=m_0$ 个分支结点，构成 m_0 棵子树，每一棵子树都有 $|S_1|=m_1$ 个分支结点。

第 1 层，有 $m_0\times m_1$ 个分支结点，构成 $m_0\times m_1$ 棵子树。

第 $n-1$ 层，有 $m_0\times m_1\cdots\times m_{n-1}$ 个结点，它们都是叶子结点。

初始化令解向量 X 为空。

在第 0 层，置 $x_0=a_{0.0}$，

$m[i]$：集合 S_i 的元素个数，　$|S_i|=m[i]$；

$x[i]$：解向量 X 的第 i 个分量；
$k[i]$：当前算法对集合 S_i 中的元素的取值位置。
回溯方法作如下的一般性描述：

```
1. void backtrack_item()
2. {
3.    initial(x);
4.    i = 0;   k[i] = 0;   flag = FALSE;
5.    while (i>=0) {
6.       while (k[i]<m[i]) {
7.          x[i] = a(i,k[i]);
8.          if (constrain(x)&&bound(x)) {
9.             if (solution(x)) {
10.               flag = TRUE;   break;
11.            }
12.            else {
13.               i = i + 1;   k[i] = 0;
14.            }
15.         }
16.         else k[i] = k[i] + 1;
17.      }
18.      if (flag) break;
19.      i = i – 1;
20.   }
21.   if (!flag)
22.      initial(x);
23. }
```

initial(x)把解向量初始化为空；
a(i, k[i])取 S_i 的第 $k[i]$ 个值，赋给解向量的分量 $x[i]$。
函数 constrain(x)判断解向量是否满足约束条件，如果满足，返回值为真。
bound(x)判断解向量是否满足目标函数的界，如果满足，返回值为真。
solution(x)判断解向量是否为问题的最终解，如果是，标志 *flag* 置为真，
回溯法解题时，包含下面三个步骤：

1. 对所给定的问题，定义问题的解空间；
2. 确定状态空间树的结构；
3. 用深度优先搜索方法搜索解空间，用约束方程和目标函数的界对状态空间树进行修剪，生成搜索树, 取得问题的解。

0/1 背包问题

不需把背包的载重量划分为 m 等分、物体的重量是背包载重量 m 等分的整数倍的限制。

回溯法解 0/1 背包问题的求解过程

一、解空间和状态可树

n 个物体 v_i，重量 w_i、价值 p_i，$0 \le i \le n-1$，背包的载重量 M。

x_i：物体 v_i 被装入背包的情况，$x_i = 0,1$。

约束方程和目标函数：

$$\sum_{i=1}^{n} w_i x_i \le M \qquad (7.5.1)$$

$$optp = \max \sum_{i=1}^{n} p_i x_i \qquad (7.5.2)$$

解向量：$X = (x_0, x_1, \cdots, x_{n-1})$，

状态空间树：高度为 n 的完全二叉树，其结点总数有 $2^{n+1}-1$ 个。

根结点到叶结点的路径，是问题的可能解。

假定：第 i 层的左儿子子树，物体 v_i 被装入背包的情况；右儿子子树，物体 v_i 未被装入背包的情况。

二、求解过程

初始化：目标函数上界为 0，物体按价值重量比的非增顺序排序，

搜索过程：尽量沿左儿子结点前进，当不能沿左儿子继续前进时，就得到问题的一个部分解，并把搜索转移到右儿子子树。

估计由部分解所能得到的最大价值，

估计值高于当前上界：继续由右儿子子树向下搜索，扩大部分解，直到找到可行解；

保存可行解，用可行解的值刷新目标函数的上界，向上回溯，寻找其它可行解；

若估计值小于当前上界：丢弃当前正在搜索的部分解，向上回溯。

三、部分解的最大估价值

假定，当前部分解是 $\{x_0, x_1, \cdots, x_{k-1}\}$，同时，有：

$$\sum_{i=0}^{k-1} x_i w_i \le M \quad \text{且} \quad \sum_{i=0}^{k-1} x_i w_i + w_k > M \qquad (7.5.3)$$

将得到部分解 $\{x_0, x_1, \cdots, x_k\}$，其中，$x_k = 0$。由这个部分解继续向下搜索，将有：

$$\sum_{i=0}^{k} x_i w_i + \sum_{i=k+1}^{k+m-1} w_i \le M \quad \text{且} \quad \sum_{i=0}^{k} x_i w_i + \sum_{i=k+1}^{k+m-1} w_i + w_{k+m} > M \qquad (7.5.4)$$

$m=1,2,\cdots,n-k-1$，当 $m=1$ 时，表示继续装入物体 v_{k+1}，仍然将超过背包的载重量。

能够找到的可能解的最大值不会超过：

$$\sum_{i=0}^{k} x_i p_i + \sum_{i=k+1}^{k+m-1} x_i p_i + (M - \sum_{i=0}^{k-1} x_i w_i - \sum_{i=k+1}^{k+m-1} x_i w_i) \times p_{k+m} / w_{k+m} \qquad (7.5.5)$$

四、回溯的两种情况

当估计值小于目标函数的上界（它是已经得到的可行解中的最大值），向上回溯：

当前的结点是左儿子分支结点，就转而搜索相应的右儿子分支结点；

当前的结点是右儿子分支结点，就沿右儿子分支结点向上回溯，直到左儿子分支结点为止，然后，再转而搜索相应的右儿子分支结点。

五、步骤

w_cur：部分解中装入背包物体的总重量

p_cur：部分解中装入背包物体的总价值；

p_est：部分解可能达到的最大估计值；

p_total：当前搜索到的所有可行解中的最大价值，目标函数的上界。

x_k：部分解的第 k 个分量

y_k：部分解的第 k 个分量的拷贝

k：搜索深度。

回溯法解 0/1 背包问题的步骤：

1. 物体按价值重量比的非增顺序排序；
2. w_cur、p_cur、p_total、k 初始化为 0，部分解初始化为空；
3. 按（7.5.4）和（7.5.5）式估计从当前的部分解可取得的最大价值 p_est；
4. 如果 $p\_est > p\_total$，转 5；否则转 8；
5. 从 v_k 开始把物体装入背包，直到没有物体可装、或装不下物体 v_i 为止，生成部分解 $y_k,\cdots,y_i$，$k \le i < n$；刷新 p_cur；
6. 如果 $i \ge n$，得到新的可行解，所有 y_i 拷贝到 x_i，$p\_total = p\_cur$；
   令 $k=n$，转 3，以便回溯搜索其它的可能解；
7. 否则，得到一个部分解，令 $k=i+1$，舍弃物体 v_i，从物体 v_{i+1} 继续装入，转 3；
8. 当 $i \ge 0$ 并且 $y_i = 0$，执行 $i=i-1$，直到条件不成立；即沿右儿子分支结点方向向上回溯，直到左儿子分支结点；
9. 如果 $i<0$，算法结束；否则，转 10；
10. 令 $y_i = 0$，$w\_cur = w\_cur - w_i$，$p\_cur = p\_cur - p_i$，$k=i+1$，转 3；从左儿子分支结点转移到相应的右儿子分支结点，继续搜索其它的部分解或可能解；

例 7.4 有载重量 $M=50$ 的背包，物体重量分别为 5, 15, 25, 27, 30，物体价值分别为 12, 30, 44, 46, 50。求最优装入背包的物体及价值。

1. p_total 为 0，计算 $p\_est=94.5$，大于 p_total，生成结点 1, 2, 3, 4，部分解(1, 1, 1, 0)；
2. 在结点 4 计算 $p\_est=94.3$，大于 p_total，继续向下搜索生成结点 5，得到价值为 86 的可行解(1, 1, 1, 0, 0)，保存在解向量 X 中， p_total 更新为 86；

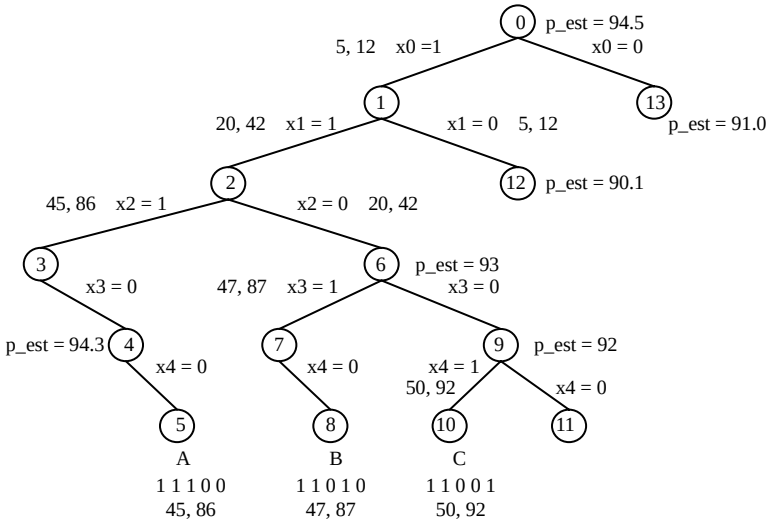


图 7.13 例 7.4 中 0/1 背包问题的搜索树

3. 由叶结点 5 继续搜索时， p_est 被置为 86，不大于 p_total 的值，因此，沿右儿子分支结点回溯到左儿子分支结点 3，生成右儿子分支结点 6，得到部分解(1, 1, 0)；
4. 在结点 6 计算 $p\_est=93$，大于 p_total，因此，生成结点 7, 8, ，得到价值为 87 的可行解(1, 1, 0, 1, 0)，更新解向量 X， p_total 更新为 87；
5. 由叶结点 8 继续搜索时， p_est 被置为 87，不大于 p_total 的值，因此，沿右儿子分支结点回溯到左儿子分支结点 7，生成右儿子分支结点 9，得到部分解(1, 1, 0, 0)；
6. 在结点 9 计算 $p\_est=92$，大于 p_total，生成结点 10，得到价值为 92 的可行解(1, 1, 0, 0, 1)，更新解向量 X， p_total 更新为 92；
7. 由叶结点 10 继续搜索时，p_est 被置为 92，不大于 p_total 的值，因此，进行回溯，因为结点 10 是左儿子结点，生成右儿子结点 11，得到可行解(1, 1, 0, 0, 0)；
8. 由叶结点 11 继续搜索时， p_est 被置为 42，不大于 p_total 的值，因此，沿右儿子分支结点回溯到左儿子分支结点 2，生成右儿子分支结点 12，得到部分解(1, 0)；
9. 在结点 12 计算 $p\_est=90.1$，小于 p_total，因此，回溯到左儿子分支结点 1，生成右儿子分支结点 13，得到部分解(0)；
10. 在结点 13 计算 $p\_est=91.0$，小于 p_total，因此，向上回溯到根结点 0，结束算法。最后，由向量 X 中的内容，得到最优解(1, 1, 0, 0, 1)，从 p_total 中得到最大价值 92。

状态空间树的 63 个结点
被访问的结点数为 14 个。

回溯法解 0/1 背包问题算法的实现

数据结构和变量：

```
typedef struct {
   float     w;          /* 物体重量 */
   float     p;          /* 物体价值 */
   float     v;          /* 物体的价值重量比 */
} OBJECT;
OBJECT   ob[n];
float    M;             /* 背包载重量 */
int      x[n];          /* 可能的解向量 */
int      y[n];          /* 当前搜索的解向量 */
float    p_est;         /* 当前搜索方向装入背包物体的估计最大价值 */
float    p_total;       /* 装入背包的物体的最大价值的上界 */
float    w_cur;         /* 当前装入背包的物体的总重量 */
float    p_cur;         /* 当前装入背包的物体的总价值 */
```

0/1 背包问题的回溯算法：

算法 7.4 0/1 背包问题的回溯算法
输入：背包载重量 M,问题个数 n,存放物体的价值和重量的结构体数组 0b[]
输出：0/1 背包问题的最优解 x[]

```
 1. float knapsack_back(OBJECT ob[],float M,int n,BOOL x[])
 2. {
 3.    int i,k;
 4.    float w_cur,p_total,p_cur,w_est,p_est;
 5.    BOOL *y = new BOOL[n];
 6.    for (i=0;i<=n;i++) {              /* 计算物体的价值重量比 */
 7.       ob[i].v = ob[i].p / ob[i].w;
 8.       y[i] = FALSE;                  /* 当前的解向量初始化 */
 9.    }
10.    merge_sort(ob,n);                 /* 物体按价值重量比的非增顺序排序*/
11.    w_cur = p_cur = p_total = 0;      /* 当前背包中物体的价值重量初始化*/
12.    k = 0;                            /* 已搜索到的可能解的总价值初始化*/
13.    while (k>=0) {
```

```
14.       w_est = w_cur;   p_est = p_cur;
15.       for (i=k;i<n;i++) {            /* 沿当前分支可能取得的最大价值 */
16.          w_est = w_est + ob[i].w;
17.          if (w_est<M) {
18.             p_est = p_est + ob[i].p;
19.          else {
20.             p_est = p_est + ((M – w_est + ob[i].w) / ob[i].w) * ob[i].p;
21.             break;
22.          }
23.       }
24.       if (p_est>p_total) {                  /* 估计值大于上界 */
25.          for (i=k;i<n;i++) {
26.             if (w_cur+ob[i].w<=M) {         /* 可装入第 i 个物体 */
27.                w_cur = w_cur + ob[i].w;
28.                p_cur = p_cur + ob[i].p;
29.                y[i] = TRUE;
30.             }
31.             else {
32.                y[i] = FALSE;   break;       /* 不能装入第 i 个物体 */
33.             }
34.          }
35.          if (i>=n) {                        /* n 个物体已全部装入 */
36.             if (p_cur>p_total) {
37.                p_total = p_cur;   k = n;    /* 刷新当前上限 */
38.                for (i=0;i<n;i++)            /* 保存可能的解 */
39.                   x[i] = y[i];
40.             }
41.          }
42.          else k = i + 1;                /* 继续装入其余物体 */
43.       }
44.       else {                             /* 估计价值小于当前上限*/
45.          while ((i>=0)&&(y[i]==0)       /* 沿着右分支结点方向回溯*/
46.             i = i – 1;                  /* 直到左分支结点 */
47.          if (i<0) break;                /* 已到达根结点,算法结束 */
48.          else {
49.             w_cur = w_cur – ob[i].w;    /* 修改当前值 */
50.             p_cur = p_cur – ob[i].p;
51.             y[i] = FALSE;   k = i + 1;  /* 搜索右分支子树 */
52.          }
```

```
53.        }
54.    }
55.    delete y;
56.    return p_total;
57. }
```

工作空间为$\Theta(n)$。

算法在最坏情况下所花费的时间：$O(n2^n)$

合并排序，需花费$\Theta(n\log n)$时间；

在最坏情况下，状态空间树有$2^{n+1}-1$个结点，有$O(2^n)$儿子结点，

每个右儿子结点都需估计继续搜索可能取得的目标函数的最大价值，每次估计时间需花费$O(n)$时间，因此，右儿子结点需花费$O(n2^n)$时间，

分支与限界

分支与限界法的基本思想

一、基本思想：

1、在$e_$结点估算沿着它的各儿子结点搜索时，目标函数可能取得的“界”，

2、把儿子结点和目标函数可能取得的“界”，保存在优先队列或堆中，

3、从队列或堆中选取“界”最大或最小的$e_$结点向下搜索，直到叶子结点，

4、若叶子结点的目标函数的值，是结点表中的最大值或最小值，则沿叶子结点到根结点的路径所确定的解，就是问题的最优解，由该叶子结点所确定的目标函数的值，就是解这个问题所得到的最大值或最小值

二、目标函数“界”的特性：

$(x_1)\cdots(x_1,x_2,\cdots,x_k)$是部分解，$bound(x_1),\cdots bound(x_1,x_2,\cdots,x_k)$是相应的界

1、对最小值问题，称为下界，意思是向下搜索所可能取得的值最小不会小于这些下界。

若$X=(x_1,x_2,\cdots,x_k)$是所得到的部分解，满足：

$$bound(x_1)\le bound(x_1,x_2)\le\cdots\le bound(x_1,x_2,\cdots,x_k) \qquad (8.1.1)$$

2、对最大值问题，称为上界，意思是向下搜索所可能取得的值最大不会大于这些上界。

若$X=(x_1,x_2,\cdots,x_k)$是所得到的部分解，满足：

$$bound(x_1)\ge bound(x_1,x_2)\ge\cdots\ge bound(x_1,x_2,\cdots,x_k)$$

三、两种分支方法：

设解向量$X=(x_1,x_2,\cdots,x_n)$，x_i的取值范围为有穷集S_i，$|S_i|=n_i$，$1\le i\le n$。

1、每棵子树都有n_i个分支：

最坏情况下，结点表的空间为$O(n_1\times n_2\times\cdots\times n_n)$，

若状态空间树是完全n叉树，$n_1=n_2=\cdots=n_n=n$，结点表的空间为$O(n^n)$。

2、每棵子树只有两个分支，x_i取特定值的分支、及不取特定值的分支：

状态空间树是完全二叉树，最坏情况下结点表的空间为$O(2^n)$

货郎担问题

有向赋权图$G=(V,E)$，顶点集$V=(v_0,v_1,\cdots,v_{n-1})$。

c为图的邻接矩阵，c_{ij}表示顶点v_i到顶点v_j的关联边的长度，又把c称为费用矩阵。

费用矩阵的特性及归约

l：图G的最短哈密尔顿回路，

$w(l)$：回路的费用。因为中的元素c_{ij}表示顶点v_i到顶点v_j的关联边的费用，

一、哈密尔顿回路与费用矩阵的关系：

引理 8.1 令$G=(V,E)$是一个有向赋权图，l是图G的一条哈密尔顿回路，c是图G的费用矩阵，则回路上的边对应于费用矩阵c中每行每列各一个元素。

证明 图G有n个顶点，

费用矩阵第i行元素：顶点v_i到其它顶点的出边费用；

费用矩阵第i列元素：其它顶点到顶点v_i的入边费用。

l是图G的一条哈密尔顿回路，

v_i是回路中的任意一个顶点，$0\le i\le n-1$，

v_i在回路中只有一条出边，对应于费用矩阵中第i行的一个元素；

v_i在回路中只出现一次，费用矩阵的第i行有且只有一个元素与其对应。

v_i在回路中只有一条入边，费用矩阵中第i列也有且只有一个元素与其对应。

回路中有n个不同顶点，费用矩阵的每行每列都有且只有一个元素与回路中的顶点的出边与入边一一对应。

例：，图 8.1(a)中 5 城市的货郎担问题的费用矩阵，

令$l=v_0v_3v_1v_4v_2v_0$是哈密尔顿回路，回路上的边对应于费用矩阵中的元素$c_{03},c_{31},c_{14},c_{42},c_{20}$。

	0	1	2	3	4
0	∞	25	41	32	28
1	5	∞	18	31	26
2	20	16	∞	7	1
3	10	51	25	∞	6
4	23	9	7	11	∞

(a)

	0	1	2	3	4	
0	∞	0	16	7	3	lh0 = 25
1	0	∞	13	26	21	lh1 = 5
2	19	15	∞	6	0	lh2 = 1
3	4	45	19	∞	0	lh3 = 6
4	16	2	0	4	∞	lh4 = 7

(b)

	0	1	2	3	4
0	∞	0	16	3	3
1	0	∞	13	22	21
2	19	15	∞	2	0
3	4	45	19	∞	0
4	16	2	0	0	∞

ch3 = 4

(c)

图 8.1　5 城市货郎担问题的费用矩阵及其归约

二、费用矩阵的归约

1、行归约和列归约

定义 8.1 费用矩阵c的第i行（或第j列）中的每个元素减去一个正常数lh_i（或ch_j），得到一个新的费用矩阵$\bar{c}$，使得$\bar{c}$中第i行（或第j列）中的最小元素为 0，称为费用矩阵的行归约(或列归约)。称lh_i为行归约常数，称ch_j为列归约常数。

例：把图 8.1(a)中归约常数$lh_0=25$，$lh_1=5$，$lh_2=1$，$lh_3=6$，$lh_4=7$。

列归约常数$ch_3=4$，所得结果如图 8.1(c)所示。

2、归约矩阵

定义 8.2 对费用矩阵 c 的每一行和每一列都进行行归约和列归约，得到一个新的费用矩阵 $\bar{c}$，使得 $\bar{c}$ 中每一行和每一列至少都有一个元素为 0，称为费用矩阵的归约。矩阵 $\bar{c}$ 称为费用矩阵 c 的归约矩阵。称常数 h

$$h=\sum_{i=0}^{n-1} lh_i+\sum_{i=0}^{n-1} ch_i \qquad (8.2.1)$$

为矩阵 c 的归约常数。

例：对图 8.1(a)中的费用矩阵进行归约，得到图 8.1(c)所示归约矩阵。

归约常数 h 为

$$h=25+5+1+6+7+4=48$$

3、归约矩阵与哈密尔顿回路的关系

定理 8.1 有向赋权图 $G=(V,E)$，G 的哈密尔顿回路 l，G 的费用矩阵 c，$w(l)$ 是以 c 计算的回路费用。$\bar{c}$ 是 c 的归约矩阵，归约常数为 h，$\bar{w}(l)$ 是以 $\bar{c}$ 计算的回路费用，有：

$$w(l)=\bar{w}(l)+h \qquad (8.2.2)$$

证明 c_{ij} 和 $\bar{c}_{ij}$ 分别是 c 和 $\bar{c}$ 的第 i 行第 j 列元素，

$c_{ij}=\bar{c}_{ij}+lh_i+ch_j \qquad i,j,0\le i,j\le n-1$，

$w(l)$ 是以 c 计算的哈密尔顿回路的费用，令

$$w(l)=\sum_{i,j\in l} c_{ij}$$

$\bar{w}(l)$ 是 $\bar{c}$ 计算的同一条哈密尔顿回路的费用，令

$$\bar{w}(l)=\sum_{i,j\in l} \bar{c}_{ij}$$

由引理 8.1，回路上的边对应于 c 中每行每列各一个元素。有

$$w(l)=\sum_{i,j\in l} c_{ij}=\sum_{i,j\in l} \bar{c}_{ij}+\sum_{i=0}^{n-1} lh_i+\sum_{j=0}^{n-1} ch_j=\bar{w}(l)+h$$

定理证毕。

定理 8.2 有向赋权图 $G=(V,E)$，l 是 G 的最短哈密尔顿回路，c 是 G 的费用矩阵，$\bar{c}$ 是 c 的归约矩阵，令 $\bar{c}$ 是图 $\bar{G}$ 的邻接矩阵，则 l 也是 $\bar{G}$ 的最短的哈密尔顿回路。

证明 用反证法证明。

若 l 不是图 $\bar{G}$ 的最短的哈密尔顿回路，

则 $\bar{G}$ 中必存在另一条回路 l^*，是 $\bar{G}$ 中最短的哈密尔顿回路，

同时，它也是 G 中的一条回路。

$\bar{w}(l)$ 和 $\bar{w}(l^*)$ 分别是以 $\bar{c}$ 计算的 l 和 l^* 的费用，有：

$$\bar{w}(l)=\bar{w}(l^*)+\delta \qquad \text{其中，}\ \delta \text{ 是正整数。}$$

l^* 是 G 的一条回路，令 $w(l)$ 和 $w(l^*)$ 是分别以 c 计算的回路 l 和 l^* 的费用。

由定理 8.1，有

$$w(l)=\overline{w}(l)+h \qquad w(l^*)=\overline{w}(l^*)+h$$

其中，h 是费用矩阵 c 的归约常数。因此

$$w(l)=\overline{w}(l)+h=\overline{w}(l^*)+\delta+h$$
$$=w(l^*)+\delta$$

l^* 是 G 中比 l 更短的哈密尔顿回路，与定理的前提相矛盾。

所以，l 也是 $\overline{G}$ 的最短的哈密尔顿回路。

界限的确定和分支的选择

先求图 G 费用矩阵 c 的归约矩阵 $\bar{c}$，得到归约常数 h

再转换为求取与 $\bar{c}$ 相对应的图 $\overline{G}$ 的最短哈密尔顿回路问题。

$w(l)$ 和 $\overline{w}(l)$ 分别是 G 和 $\overline{G}$ 的最短哈密尔顿回路的费用，

有 $w(l)=\overline{w}(l)+h$。

G 的最短哈密尔顿回路的费用，最少不会少于 h。

h 是货郎担问题状态空间树中根结点 X 的下界。$w(X)=h$

例：图 8.1(a)中归约常数 48 便是该问题的下界。该问题的最小费用不会少于 48。

8.2.2.1 界限的确定

1、搜索策略

选取沿某一边出发的路径，作为分支结点 Y；

不沿该边出发的其它所有路径集合，作为另一个分支结点 $\overline{Y}$。

2、选取沿 (i,j) 方向的路径时，结点 Y 下界 $w(Y)$ 的确定

G 的哈密尔顿回路 l，费用矩阵 c，以 c 计算的回路费用 $w(l)$。

$\bar{c}$ 是 c 的归约矩阵，归约常数为 h，以 $\bar{c}$ 计算的回路费用 $\overline{w}(l)$，

$w(l)=\overline{w}(l)+h=\overline{w}(l')+\bar{c}_{ij}+h$

1）$\bar{c}_{ji}=\infty$，处理不可能经过的边

2）矩阵降阶，删去第 i 行第 j 列所所有元素，得到降阶后的矩阵 c'

3）归约 c'，得归约常数 h'，有 $\overline{w}(l')=\overline{w}(\bar{l}')+h'$

$w(l)=\overline{w}(\bar{l}')+\bar{c}_{ij}+h+h'$

$w(Y)=h+h'$

例：图 8.1(a)及图 8.1(c)的 5 城市货郎担问题的费用矩阵、及其归约矩阵。

选取从顶点 v_1 出发，沿着 (v_1,v_0) 的边前进，

则该回路的边包含费用矩阵中的 $\bar{c}_{10}$。

删去 $\bar{c}$ 中的第 1 行和第 0 列的所有元素，

素 $\bar{c}_{01}$ 置为 ∞。

图 8.1(c)中 5×5 的归约矩阵，降阶为图 8.2(b)所示的 4×4 的矩阵。

进一步进行归约，得到图 8.2(c)所示的归约矩阵，其归约常数为 5。

表明沿 v_1 出发，经边 (v_1, v_0) 的回路，其费用至少不会小于 48＋5 = 53。

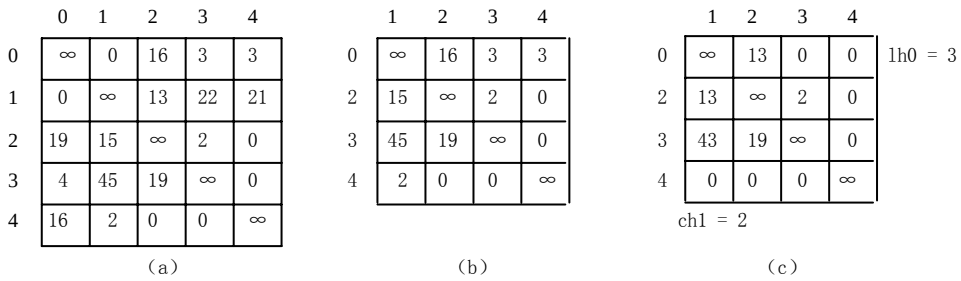


图 8.2　Y 结点对费用矩阵的降阶处理

4）处理不可能经过的边：

(1) $v_i v_j$ 不和其它已经选择的边相连接，把 c_{ji} 置为∞，如图 8.3(a)所示；

(2) 和以前选择的边连接成 $v_i v_j v_k v_l$，把 c_{li} 置为∞，如图 8.3(b)所示；

(3) 和以前选择的边连接成 $v_k v_i v_j v_l$，把 c_{lk} 置为∞，如图 8.3(c)所示；

(4) 和以前选择的边连接成 $v_k v_l v_i v_j$，把 c_{jk} 置为∞，如图 8.3(d)所示；

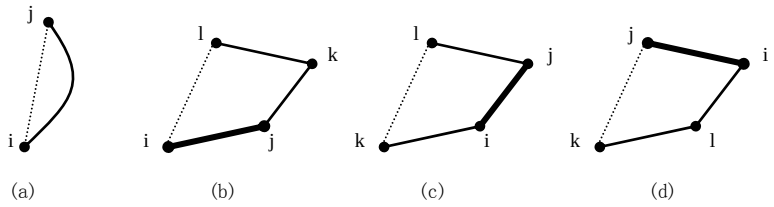


图 8.3　选择有向边时的几种可能情况

5） 父亲结点 X，下界 $w(X)$，降阶后的归约常数为 h，结点 Y 的下界为

$$w(Y) = w(X) + h \qquad (8.2.3)$$

3、不沿 (i, j) 方向的结点 $\overline{Y}$ 下界 $w(\overline{Y})$ 的确定

1）回路不包含 $v_i v_j$ 边，　c_{ij} 置为∞。（不降阶）

2） $d_{ij} = \min\limits_{0 \le k \le n-1, k \ne j} \{c_{ik}\} + \min\limits_{0 \le k \le n-1, k \ne i} \{c_{kj}\}$ （8.2.4）

3）结点 $\overline{Y}$ 的下界为：

$$w(\overline{Y}) = w(X) + d_{ij} \qquad (8.2.5)$$

例：在图 8.1(a)中，根结点作为父亲结点 X，则 $w(X) = 48$。

选择边 (v_1, v_0) 向下搜索作为结点为 Y，结点为 Y 的下界为：

$$w(Y) = w(X) + h = 48 + 5 = 53$$

结点 $\overline{Y}$ 的下界为：

$$w(\overline{Y}) = w(X) + d_{ij} = 48 + 4 + 13 = 65$$

8.2.2.2 分支的选择

选择分支的思想方法：

1. 沿 $c_{ij}=0$ 的方向选择，使所选择的路线尽可能短；
2. 沿 d_{ij} 最大的方向选择，使 $w(\overline{Y})$ 尽可能大；

令 S 是 $c_{ij}=0$ 的元素集合， D_{kl} 是 S 中使 d_{ij} 达最大的元素 d_{kl} ，即：

$$D_{kl}=\max_{S}\{d_{ij}\} \qquad (8.2.6)$$

边 $v_k v_l$ 就是所选择的分支方向。

例：图 8.1(a)中的费用矩阵归约为 8.1(c)中矩阵，根结点的下界 $w(X)=48$

有 $c_{01}=c_{10}=c_{24}=c_{34}=c_{42}=c_{43}=0$，搜索方向的选择如下：

$$d_{01}=3+2=5 \qquad d_{10}=13+4=17 \qquad d_{24}=2+0=2$$

$$d_{34}=4+0=4 \qquad d_{42}=0+13=13 \qquad d_{43}=0+2=2$$

$D_{kl}=d_{10}=17$ 。

所选择的方向为边 $v_1 v_0$，据此建立结点 Y 和 $\overline{Y}$ 。此时，

$$w(\overline{Y})=w(X)+D_{kl} \qquad (8.2.7)$$

货郎担问题的求解过程

结点数据结构：

```
typedef struct node_data {
    Type   c[n][n];            /* 费用矩阵 */
    int    row_init[n];        /* 费用矩阵的当前行映射为原始行 */
    int    col_init[n];        /* 费用矩阵的当前列映射为原始列 */
    int    row_cur[n];         /* 费用矩阵的原始行映射为当前行 */
    int    col_cur[n];         /* 费用矩阵的原始列映射为当前列 */
    int    ad[n];              /* 回路顶点邻接表 */
    int    k;                  /* 当前费用矩阵的阶 */
    Type   w;                  /* 结点的下界 */
} NODE;
```

分支限界法求解货郎担问题的求解过程：

1. 分配堆缓冲区，初始化为空堆；
2. 建立结点 X， c 拷贝到 $X.c$， $X.k$ 初始化为 n；归约 $X.c$，计算归约常数 h，下界 $X.w=h$；初始化回路的顶点邻接表 $X.ad$；
3. 按(8.2.4)式，由 $X.c$ 中所有 $c_{ij}=0$ 的元素 c_{ij}，计算 d_{ij}；
4. 按(8.2.6)式，选取使 d_{ij} 达最大的元素 d_{kl} 作为 D_{kl}，选择边 $v_k v_l$ 作为分支方向；
5. 建立儿子结点 $\overline{Y}$，$X.c$ 拷贝到 $\overline{Y}.c$，$X.ad$ 拷贝到 $\overline{Y}.ad$，$X.k$ 拷贝到 $\overline{Y}.k$；把 $\overline{Y}.c$ 中的 c_{kl}

置为∞，归约$\overline{Y}.c$；计算结点$\overline{Y}$的下界$\overline{Y}.w$；把结点$\overline{Y}$按$\overline{Y}.w$插入最小堆中；

6. 建立儿子结点Y， $X.c$拷贝到$Y.c$， $X.ad$拷贝到$Y.ad$，$X.k$拷贝到$Y.k$； $Y.c$的有关元素置为∞；
7. 降阶$Y.c$， $Y.k$减1，归约降阶后的$Y.c$，按(8.2.3)式计算结点Y的下界$Y.w$；
8. 若$Y.k=2$，直接判断最短回路的两条边，并登记于路线邻接表$Y.ad$，使$Y.k=0$；
9. 把结点Y按$Y.w$插入最小堆中；
10. 取下堆顶元素作为结点X，若$X.k=0$，算法结束；否则，转3；

例 8.1　求解图8.1(a)所示的5城市货郎担问题。

该问题的求解过程如图8.4所示，过程如下：

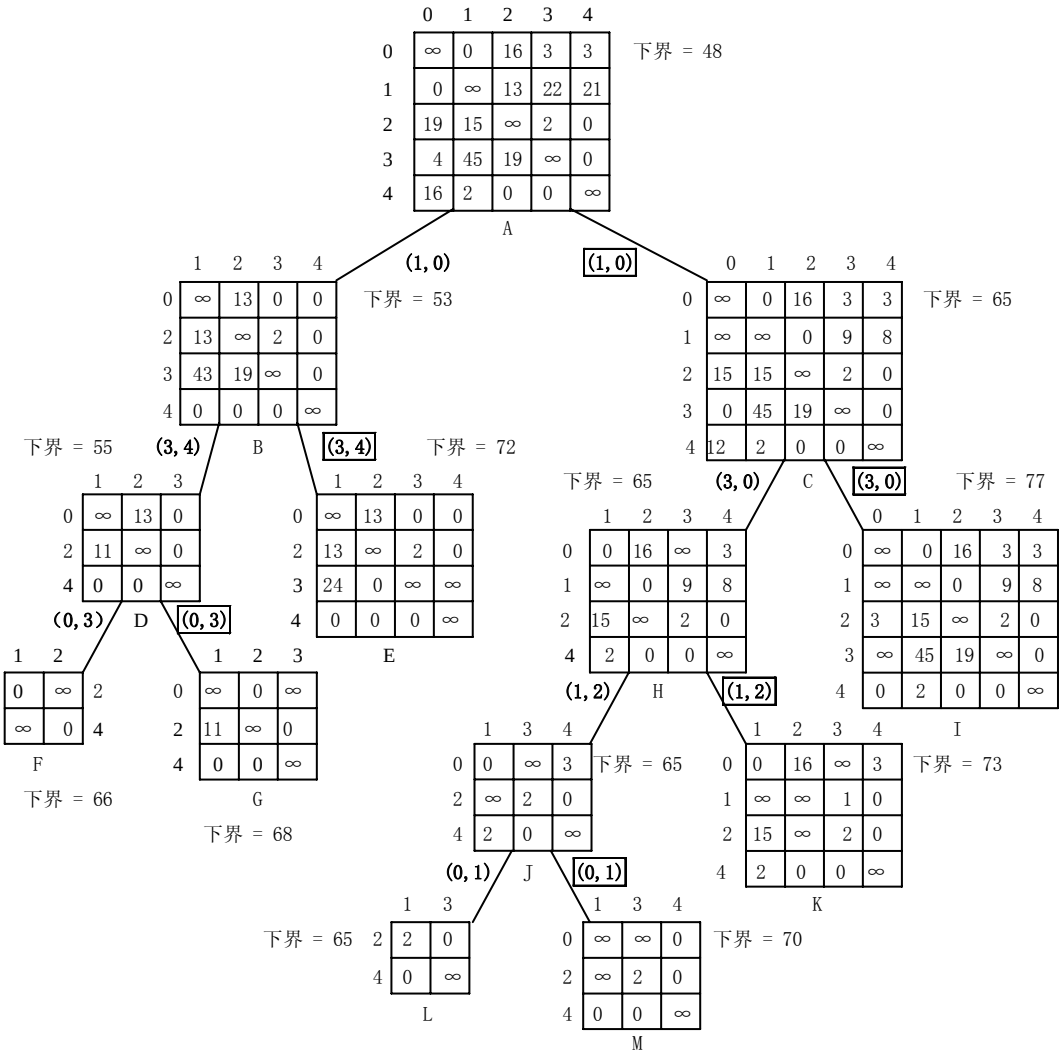


图 8.4　用分支限界法解5城市货郎担问题的过程

几个辅助函数的实现

数据结构：

```
typedef struct node_data {
   Type   c[n][n];            /* 费用矩阵 */
   int    row_init[n];        /* 费用矩阵的当前行(下标)映射为原始行(内容) */
   int    col_init[n];        /* 费用矩阵的当前列(下标)映射为原始列(内容)*/
   int    row_cur[n];         /* 费用矩阵的原始行(下标)映射为当前行(内容) */
   int    col_cur[n];         /* 费用矩阵的原始列(下标)映射为当前列(内容) */
   int    ad[n];              /* 回路顶点邻接表 */
   int    k;                  /* 当前费用矩阵的阶 */
   Type   w;                  /* 结点的下界 */
} NODE;
NODE      *xnode;             /* 父亲结点指针 */
NODE      *ynode;             /* 儿子结点指针 */
NODE      *znode;             /* 儿子结点指针 */
int       n_heap;             /* 堆元素个数 */
typedef struct {              /* 堆结构数据 */
   NODE  *p;                  /* 指向结点元素的指针*/
   Type   w;                  /* 所指向结点的下界,堆元素的关键字 */
} HEAP;
```

$ad[i]$：与顶点 i（出）相邻接的顶点（入）序号。

例：的回路由边 v_3v_0、v_1v_2、v_2v_4、v_0v_1、v_4v_3 组成，数组 *ad* 中的登记情况：

	0	1	2	3	4
ad	1	2	4	0	3

图 8.5　回路顶点邻接表的登记情况

算法中使用下面的几个函数：

Type row_min（NODE * node , int row, Type &second ）；	计算费用矩阵行的最小值
Type col_min（NODE * node , int col, Type &second）；	计算费用矩阵列的最小值
Type array_red（NODE * node）；	归约 *node* 所指向结点的费用矩阵
Type edge_sel（NODE * node, int &vk, int &vl）；	计算 D_{kl}，选择搜索分支的边
void del_rowcol（NODE * node, int vk, int vl）；	删除费用矩阵第 *vk* 行、*vl* 列
void edge_byp（NODE * node, int vk, int vl）；	登记回路顶点邻接表，旁路有

关的边

NODE * initial（Type c[ ][ ], int n）； 初始化

1、row_min（NODE * node , int row, Type &second）函数返回 *node* 所指向结点的费用矩阵中第 *row* 行的最小值，次小值回送于引用变量 sec*ond* 。

```
1. Type row_min(NODE *node,int row,Type &second)
2. {
3.    Type temp;
4.    int i;
5.    if (node->c[row][0]<node->c[row][1]) {
6.       temp = node->c[row][0];   second = node->c[row][1];
7.    }
8.    else {
9.       temp = node->c[row][1];   second = node->c[row][0];
10.   }
11.   for (i=2;i<node->k;i++) {
12.      if (node->c[row][i]<temp) {
13.         second = temp;   temp = node->c[row][i];
14.      }
15.      else if (node->c[row][i]<second)
16.         second = node->c[row][i];
17.   }
18.   return temp;
19. }
```

运行时间：$O(n)$。

工作单元个数：$\Theta(1)$ 。

2、Type col_min（NODE * node , int col, Type &second）返回 *node* 所指向的结点的费用矩阵中第 *col* 列的最小值，次小值回送于引用变量 sec*ond* 。

3、Type array_red(NODE *node)归约 *node* 所指向的结点的费用矩阵，返回值为归约常数

```
1. Type array_red(NODE *node)
2. {
3.    int i,j;
4.    Type temp,temp1,sum = 0;
5.    for (i=0;i<node->k;i++) {            /* 行归约 */
6.       temp = row_min(node,i,temp1);     /* 行归约常数 */
7.       for (j=0;j<node->k;j++)
```

```
 8.           node->c[i][j] -= temp;
 9.        sum += temp;                          /* 行归约常数累计 */
10.    }
11.    for (j=0;j<node->k;j++) {                 /* 列归约 */
12.        temp = col_min(node,j,temp1);         /* 列归约常数 */
13.        for (i=0;i<node->k;i++)
14.           node->c[i][j] -= temp;
15.        sum += temp;                          /* 列归约常数累计 */
16.    }
17.    return sum;                               /* 返回归约常数*/
18. }
```

运行时间：$O(n^2)$时间。

工作单元个数：$\Theta(1)$ 。

4、函数 edge_sel 计算 D_{kl}，选择搜索分支的边。返回 D_{kl} 的值，出边顶点序号 *vk* 和入边顶点序号 *vl* 。

```
 1. Type edge_sel(NODE * node,int &vk,int &vl)
 2. {
 3.    int i,j;
 4.    Type temp,d = 0;
 5.    Type *row_value = new Type[node->k];
 6.    Type *col_value = new Type[node->k];
 7.    for (i=0;i<node->k;i++)                   /* 每一行的次小值 */
 8.       row_min(node,i,row_value[i]);
 9.    for (i=0;i<node->k;i++)                   /* 每一列的次小值 */
10.       col_min(node,i,col_value[i]);
11.    for (i=0;i<node->k,i++) {                 /* 对费用矩阵所有值为0的元素*/
12.       for (j=0;j<node->k;j++) {              /* 计算相应的 temp 值 */
13.          if (node->c[i][j]==0) {
14.             temp = row_value[i] + col_value[j];
15.             if (temp>d) {                    /* 求最大的 temp 值于 d */
16.                d = temp;   vk = i;   vl = j;
17.             }                                /* 保存相应的行、列号 */
18.          }
19.       }
20.    }
21.    delete row_value;
22.    delete col_value;
```

```
23.     return d;
24. }
```

运行时间：$O(n^2)$时间。

工作单元：$O(n)$。

5、函数 del_rowcol 删除费用矩阵当前第 *vk* 行、第 *vl* 列的所有元素

```
 1. void del_rowcol(NODE *node,int vk,int vl)
 2. {
 3.     int i,j,vk1,vl1;
 4.     for (i=vk;i<node->k-1;i++)        /* 元素上移 */
 5.        for (j=0;j<vl;j++)
 6.           node->c[i][j] = node->c[i+1][j];
 7.     for (j=vl;j<node->k-1;j++)        /* 元素左移 */
 8.        for (i=0;i<vk;i++)
 9.           node->c[i][j] = node->c[i][j+1];
10.     for (i=vk;i<node->k-1;i++)        /* 元素上移及左移 */
11.        for (j=vl;j<node->k-1;j++)
12.           node->c[i][j] = node->c[i+1][j+1];
13.     vk1 = node->row_init[vk];         /* 当前行 vk 转换为原始行 vk1*/
14.     node->row_cur[vk1] = -1;          /* 原始行 vk1 置删除标志 */
15.     for (i= vk1+1;i<n;i++)    /*vk1 之后的原始行,其对应的当前行号减 1*/
16.        node->row_cur--;
17.     vl1 = node->col_init[vl];         /* 当前列 vl 转换为原始列 vl1*/
18.     node->col_cur[vl1] = -1;          /* 原始列 vk1 置删除标志 */
19.     for (i=vl1+1;i<n;i++)     /* vl1 之后的原始列,其对应的当前列号减 1*/
20.        node->col-cur--;
21.     for (i=vk;i<node->k-1;i++)    /* 修改 vk 及其后的当前行的对应原始行号*/
22.        node->row_init[i] = node->row_init[i+1];
23.     for (i=vl;i<node->k-1;i++)/* 修改 vl 及其后的当前列的对应原始列号 */
24.        node->col_init[i] = node->col_init[i+1];
25.     node->k--;                /* 当前矩阵的阶数减 1 */
26. }
```

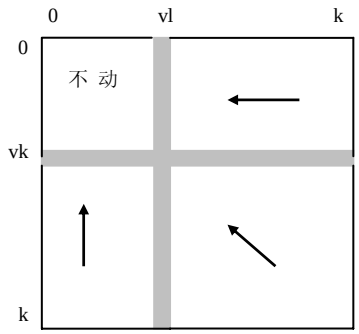


图 8.5　矩阵降阶时元素的移动过程

运行时间：$O(n^2)$时间。

工作单元个数：$\Theta(1)$ 。

6、函数 edge_byp 把*vk* 行、*vl* 列所表示的边，登记到回路顶点邻接表，旁路矩阵中有关的边：

```
1. void edge_byp(NODE *node,int vk,int vl)
2. {
3.    int i,j,k,l;
4.    vk = row_init[vk];            /* 当前行号转换为原始行号 */
5.    vl = col_init[vl];            /* 当前列号转换为原始列号 */
6.    node->ad[vk] = vl;            /* 登记回路顶点邻接表*/
7.    for (i=0;i<n;i++) {           /* 检索顶点邻接表 */
8.       j = i;
9.       while(node->ad[j]!=-1)     /* 检查从顶点 i 开始的通路 */
10.         j = node->ad[j];
11.      if (i!=j) {                /* 存在一条起点为 i 终点为 j 的通路 */
12.         l = node->row_cur[j];   /* j 转换为当前行号 l */
13.         k = node->col_cur[i];   /* i 转换为当前列号 k */
14.         if ((k>0)&&(l>0))       /* 当前行、列号均处于当前矩阵中 */
15.            node->c[l][k] = MAX_VALUE_OF_TYPE;
16.      }                          /* 相应元素置为无限大,旁路相应的边*/
17.   }
18. }
```

运行时间：$O(n^2)$时间。

工作单元个数：$\Theta(1)$ 。

初始化函数 NODE * initial（Type c[ ][ ], int n）叙述如下：

```
 1. NODE *initial(Type c[][],int n)
 2. {
 3.    int i,j;
 4.    NODE *node = new NODE;          /* 分配结点缓冲区 */
 5.    for (i=0;i<n;i++)               /* 拷贝费用矩阵的初始数据 */
 6.       for (j=0;j<n;j++)
 7.          node->c[i][j] = c[i][j];
 8.    for (i=0;i<n;i++) {             /* 建立费用矩阵原始行、列号与 */
 9.       node->row_init[i] = i;       /* 初始行、列号的初始对应关系 */
10.       node->col_init[i] = i;
11.       node->row_cur[i] = i;
12.       node->col_cur[i] = i;
13.    }
14.    for (i=0;i<n;i++)               /* 回路顶点邻接表初始化为空 */
15.       node->ad[i] = -1;
16.    node->k = n;
16.    return node;                    /* 返回结点指针 */
17. }
```

执行时间：$O(n^2)$。

不把结点缓冲区所需存储空间包括在内，工作单元个数是$\Theta(1)$ 。

货郎担问题分支限界算法的实现

算法 8.1 货郎担问题的分支限界算法

输入：城市顶点的邻接矩阵 c[][],顶点个数 n

输出：最短路线费用 w 及回路的邻接顶点表 ad[]

```
 1. template <class Type>
 2. Type traveling_salesman(Type c[][],int n,int ad[])
 3. {
 4.    int i,j,vk,vl,n_heap = 0;
 5.    Type d,w;
 6.    NODE *xnode,*ynode,*znode;
 7.    HEAP *heap = new HEAP[n*n];   /* 分配堆的缓冲区 */
 8.    HEAP x,y,z;                   /* x,y,z 结点的堆元素 */
 9.    xnode = initial(c,n);         /* 初始化父亲结点—-x 结点 */
10.    xnode->w = array_red(xnode);  /* 归约费用矩阵 */
11.    while (xnode->k!=0) {
12.       d = edge_sel(xnode,vk,vl); /* 选择分支方向并计算 D_kl */
```

```
13.        znode = new NODE;          /* 建立分支结点—-z 结点(右儿子结点) */
14.        *znode = *xnode;           /* x 结点数据拷贝到 z 结点 */
15.        znode->c[vk][vl] = MAX_VALUE_OF_TYPE; /* 旁路 z 结点的边 */
16.        array_red(znode);          /* 归约 z 结点费用矩阵 */
17.        znode->w = xnode->w + d;   /* 计算 z 结点的下界 */
18.        z.w = znode->w;            /* 构造 z 结点的堆元素 */
19.        z.p = znode;
20.        insert(heap,n_heap,z);     /* z 结点插入堆中 */
21.        ynode = new NODE;          /* 建立分支结点—-y 结点(左儿子结点) */
22.        *ynode = *xnode;           /* x 结点数据拷贝到 y 结点 */
23.        edge_byp(ynode,vk,vl);     /* 登记回路邻接表,旁路有关的边 */
24.        del_rowcol(ynode,vk,vl);   /* 删除 y 结点费用矩阵当前 vk 行 vl 列*/
25.        ynode->w = array_red(xnode);   /* 归约 y 结点费用矩阵 */
26.        ynode->w += xnode->w;          /* 计算 y 结点的下界 */
27.        y.w = ynode->w;                /* 构造 y 结点的堆元素 */
28.        y.p = ynode;
29.        if (ynode->k==2) {             /* 费用矩阵只剩 2 阶 */
30.           if (ynode->c[0][0]==0) {    /* 登记最后的两条边 */
31.              ynode->ad[ynode->row_init[0]] = ynode->col_init[0];
32.              ynode->ad[ynode->row_init[1]] = ynode->col_init[1];
33.           }
34.           else {
35.              ynode->ad[ynode->row_init[0]] = ynode->col_init[1];
36.              ynode->ad[ynode->row_init[1]] = ynode->col_init[0];
37.           }
38.           ynode->k = 0;
39.        }
40.        insert(heap,n_heap,y);         /* y 结点插入堆中 */
41.        delete xnode;                  /* 释放没用的 x 结点缓冲区 */
42.        x = delete_min(heap,n_heap);   /* 取下堆顶元素作为 x 结点*/
43.        xnode = x.p;
44.     }
45.     w = xnode->w                      /* 保存最短路线费用 */
46.     for (i=0;i<n;i++)                 /* 保存路线的顶点邻接表 */
47.        ad[i] = xnode->ad[i];
48.     delete xnode;                     /* 释放 x 结点缓冲区*/
49.     for (i=1;i<=n_heap;i++)           /* 释放堆的缓冲区*/
50.        delete heap[i].p;
51.     delete heap;
```

```
52.     return w;                              /* 回送最短路线费用 */
53. }
```

算法的时间花费估计如下：

第 9 行初始化父亲结点，第 10 行归约父亲结点费用矩阵，都需 $O(n^2)$ 时间。

第 11 行开始的 while 循环，在最坏情况下，循环体执行 2^n 次。

在 while 循环内部：

12 行选择分支方向，需 $O(n^2)$ 时间。

14 行把 x 结点数据拷贝到 z 结点， 16 行归约 z 结点费用矩阵，都需 $O(n^2)$ 时间。

20 行把 z 结点插入堆中，在最坏情况下，有 2^n 个结点，需 $O(\log 2^n)=O(n)$ 时间。

22 行把 x 结点数据拷贝到 y 结点，需 $O(n^2)$ 时间。

23 行登记回路邻接表，旁路有关的边， 24 行删除 y 结点费用矩阵当前 vk 行 vl 列，

25 行归约 y 结点费用矩阵，这些操作都需 $O(n^2)$ 时间。

40 行把 y 结点插入堆中， 42 行删除堆顶元素，都需 $O(\log 2^n)=O(n)$ 时间。

其余花费为 $O(1)$ 时间。

整个 while 循环，在最坏情况下需 $O(n^2 2^n)$ 。

第 46 行的 for 循环保存路线的顶点邻接表于数组 ad 需 $O(n)$ 时间。

第 49 行释放堆的缓冲区，在最坏情况下，需 $O(n)$ 时间。

算法的运行时间： $O(n^2 2^n)$ 。

算法所需要的空间：

每个结点需要 $O(n^2)$ 空间存放费用矩阵，共有 2^n 个结点，需 $O(n^2 2^n)$ 空间。

0/1 背包问题

分支限界法解 0/1 背包问题的思想方法和求解过程

n 个物体重量分别为 $w_0, w_1, \cdots, w_{n-1}$，价值分别为 $p_0, p_1, \cdots, p_{n-1}$，背包载重量为 M

物体按价值重量比递减的顺序，排序后物体序号的集合为 $S=\{0,1,\cdots,n-1\}$ 。

S_1：选择装入背包的物体集合，

S_2：不选择装入背包的物体集合，

S_3：尚待选择的物体集合。

$S_1(k)$、$S_2(k)$、$S_3(k)$：搜索深度为 k 时的三个集合中的物体。开始时，

$$S_1(0)=\varphi \qquad S_2(0)=\varphi \qquad S_3(0)=S=\{0,1,\cdots,n-1\}$$

一、分支的选择及处理

s：比值 p_i / w_i 最大的物体序号， $s \in S_3$ 。

把物体 s 装入背包的分支结点，不把物体 s 装入背包的分支结点。

s 就是集合 $S_3(k)$ 中的第一个元素。

搜索深度为 k 时，物体 s 的序号就是集合 S 中的元素 k 。

物体 s 装入背包的分支结点作如下处理：

$$S_1(k+1)=S_1(k)\cup\{k\}$$
$$S_2(k+1)=S_2(k)$$
$$S_3(k+1)=S_3(k)-\{k\}$$

不把物体 s 装入背包的分支结点则做如下处理：

$$S_1(k+1)=S_1(k)$$
$$S_2(k+1)=S_2(k)\cup\{k\}$$
$$S_3(k+1)=S_3(k)-\{k\}$$

二、上界的确定

$b(k)$：搜索深度为 k 时，分支结点的背包中物体的价值上界

$S_3(k)=\{k,k+1,\cdots,n-1\}$。若：

$$M<\sum_{i\in S_1(k)}w_i \qquad \text{令} \qquad b(k)=0 \qquad (8.3.1)$$

若：

$$M=\sum_{i\in S_1(k)}w_i+\sum_{i=k}^{l-1}w_i+x\cdot w_l \qquad 0\le x<1,k<l,k\in S_3(k),l\in S_3(k)$$

令：

$$b(k)=\sum_{i\in S_1(k)}p_i+\sum_{i=k}^{l-1}p_i+x\cdot p_l \qquad (8.3.2)$$

三、求解步骤

1. 把物体按价值重量比递减顺序排序；
2. 建立根结点 X，令 $X.b=0$，$X.k=0$，$X.S_1=\varphi$，$X.S_2=\varphi$，$X.S_3=S$；
3. 若 $X.k=n$，算法结束，$X.S_1$ 即为装入背包中的物体，$X.b$ 即为装入背包中物体的最大价值；否则，转 4；
4. 建立结点 Y，$Y.S_1=X.S_1\cup\{X.k\}$，$Y.S_2=X.S_2$，$Y.S_3=X.S_3-\{X.k\}$，$Y.k=X.k+1$；按（8.3.1）、（8.3.2）式计算 $Y.b$；把结点 Y 按 $Y.b$ 插入堆中；
5. 建立结点 Z，$Z.S_1=X.S_1$，$Z.S_2=X.S_2\cup\{X.k\}$，$Z.S_3=X.S_3-\{X.k\}$，$Z.k=X.k+1$；按（8.3.1）、（8.3.2）式计算 $Z.b$；把结点 Z 插入堆中；
6. 取下堆顶元素于结点 X，转 3；

例 8.2 有 5 个物体，重量分别为 8, 16, 21, 17, 12，价值分别为 8, 14, 16, 11, 7，背包载重量为 37，求装入背包的物体及其价值。

假定，物体序号分别为 0, 1, 2, 3, 4。最后得到的解是 $S_1=\{1,2\}$，最大价值是 30。

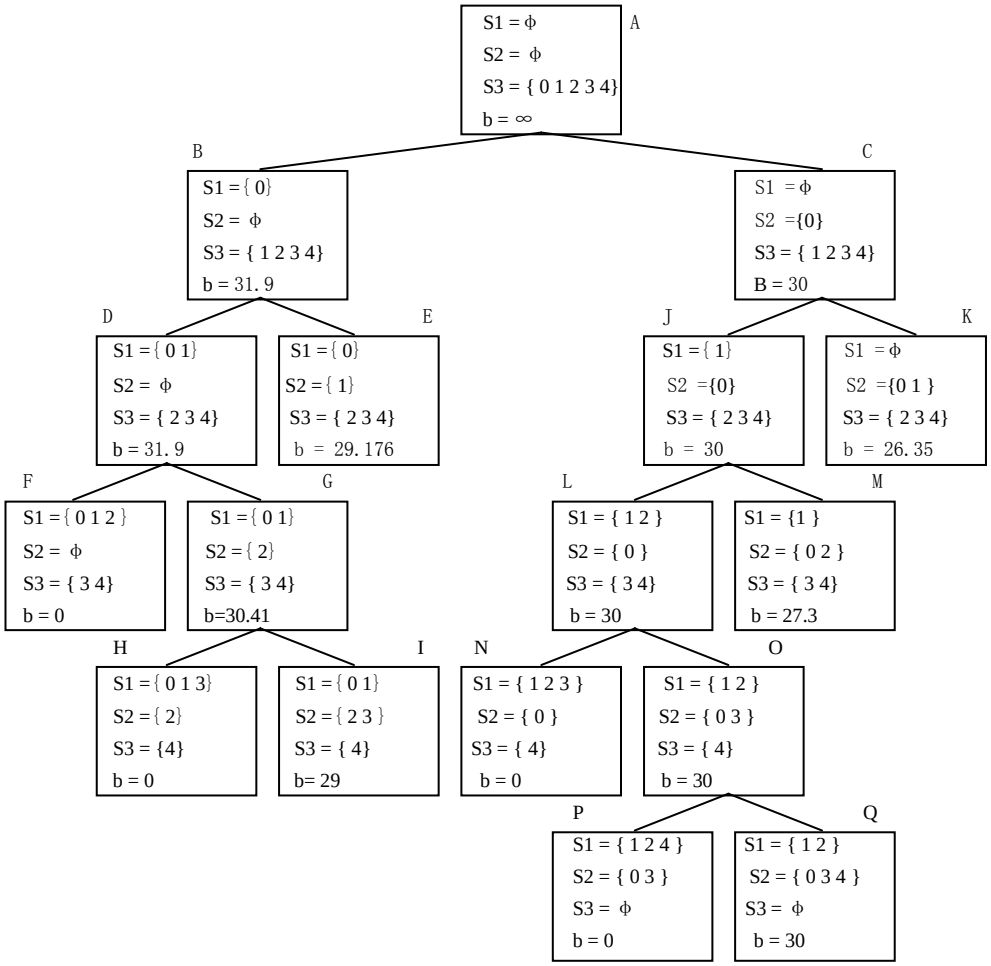


图 8.6　0/1 背包问题分支限界法的求解过程

0/1 背包问题分支限界算法的实现

数据结构：

```
typedef struct {
    float      w;          /* 物体重量 */
    float      p;          /* 物体价值 */
    float      v;          /* 物体的价值重量比 */
    int        num;        /* 物体排序前的初始序号 */
} OBJECT;
OBJECT    ob[n];
float     M;              /* 背包载重量 */
```

```
BOOL      x[n];           /* 最优装入背包的物体 */

typedef  struct {
   BOOL       s1[n];      /* 当前集合 S1 中的物体 */
   int        k;          /* 当前结点的搜索深度 */
   float      b;          /* 当前结点的价值上界 */
   float      w;          /* 当前集合 S1 中的物体重量 */
   float      p;          /* 当前集合 S1 中的物体价值 */
} KNAPNODE;

typedef  struct {
   KNAPNODE   *p;         /* 指向结点的数据 */
   float      b;          /* 所指向结点的上界,堆元素的关键字 */
} HEAP;
```

使用 bound 函数来计算分支结点的上界。bound 函数叙述如下：

```
 1. void bound(KNAPNODE *node,float M,OBJECT ob[],int n)
 2. {
 3.    int i = node->k;
 4.    float w = node->w;
 5.    float p = node->p;
 6.    if (node->w>M)                    /* 物体重量超过背包载重量 */
 7.       node->b = 0;                   /* 上界置为 0 */
 8.    else {                            /* 否则,确定背包的剩余载重量 */
 9.       while (w+ob[i].w<=M)&&(i<n) {  /* 以及继续装入可得到的最大价值 */
10.          w += ob[i].w;
11.          p += ob[i++].p;
12.       }
13.       if (i<n)
14.          node->b = p + (M – w) * ob[i].p / ob[i].w;
15.       else
16.          node->b = p;
17.    }
18. }
```

这个函数的执行时间，在最好的情况下是$O(1)$时间，在最坏的情况下是$O(n)$时间。这样，0/1 背包问题分支限界算法，可叙述如下：

算法 8.2 用分支限界方法实现 0/1 背包问题

输入：包含 n 个物体的重量和价值的数组 ob[],背包载重量 M

输出：最优装入背包的物体 obx[],装入背包的物体最优价值 v

```
 1. float knapsack_bound(OBJECT ob[],float M,int n,int obx[])
 2. {
 3.     int i,j,k = 0;                      /* 堆中元素个数的计数器初始化为 0 */
 4.     float v;
 5.     KNAPNODE *xnode,*ynode,*znode;
 6.     HEAP x,y,z,*heap;
 7.     heap = new HEAP[n*n];               /* 分配堆的存储空间 */
 8.     for (i=0;i<n;i++) {
 9.        ob[i].v = ob[i].p / ob[i].w;     /* 计算物体的价值重量比 */
10.        ob[i].num = i;                   /* 物体排序前的原始序号 */
11.     }
12.     merge_sort(ob,n);                   /* 物体按价值重量比排序 */
13.     xnode = new KNAPNODE;               /* 建立父亲结点 x */
14.     for (i=0;i<n;i++)                   /* 结点 x 初始化 */
15.        xnode->s1[i] = FALSE;
16.     xnode->k = 0;
17.     xnode->w = 0;
18.     xnode->p = 0;
19.     while (xnode->k<n) {
20.        ynode = new KNAPNODE;            /* 建立结点 y */
21.        *ynode = *xnode;                 /* 结点 x 的数据拷贝到结点 y */
22.        ynode->s1[ynode->k] = TRUE;      /* 装入第 k 个物体 */
23.        ynode->w += ob[ynode->k].w;      /* 背包中物体重量累计 */
24.        ynode->p += ob[ynode->k].p;      /* 背包中物体价值累计 */
25.        ynode->k++;                      /* 搜索深度加 1 */
26.        bound(ynode,M,ob,n);             /* 计算结点 y 的上界 */
27.        y.b = ynode->b;
28.        y.p = ynode;
29.        insert(heap,y,k);                /* 结点 y 按上界之值插入堆中*/
30.        znode = new KNAPNODE;            /* 建立结点 z */
31.        *znode = *xnode;                 /* 结点 x 的数据拷贝到结点 z */
32.        znode->k++;                      /* 搜索深度加 1 */
33.        bound(ynode,M,ob,n);             /* 计算结点 z 的上界 */
34.        z.b = znode->b;
35.        z.p = znode;
36.        insert(heap,z,k);                /* 结点 z 按上界之值插入堆中*/
```

```
37.         delete xnode;                    /* 释放结点 x 的缓冲区 */
38.         x = delete_max(heap,k);          /* 取下堆顶元素作为新的父亲结点*/
39.         xnode = x.p;
40.     }
41.     v = xnode->p;
42.     for (i=0;i<n;i++) {                  /* 取装入背包中物体在排序前的序号*/
43.         if (xnode->sl[i]) obx[i] = ob[i].num;
44.         else obx[i] = -1;
45.     }
46.     delete xnode;                        /* 释放 x 结点缓冲区*/
47.     for (i=1;i<=k;i++)                   /* 释放堆中结点的缓冲区*/
48.         delete heap[i].p;
49.     delete heap;                         /* 释放堆的缓冲区 */
50.     return v;                            /* 回送背包中物体的价值 */
51. }
```

算法的时间复杂性估计：

第 8~12 行中，执行排序算法需要花费 $O(n\log n)$；

第 13~18 行对父亲结点进行初始化，需 $O(n)$ 时间；

第 19~40 行的 while 循环，循环体在最坏情况下，可能执行 2^n 次；

21 行和 31 行拷贝结点中的数据，需花费 $O(n)$ 时间；

26 行和 33 行计算上界的工作，需花费 $O(n)$ 时间；

29、36、38 行的堆的操作，需花费 $O(\log n)$；

其余花费 $O(1)$ 时间。

第 19~40 行的 while 循环，在最坏情况下，需花费 $O(n2^n)$ 时间。

第 42~45 行，把在数组 *obx* 中构成解向量，需 $O(n)$ 时间；

第 46~50 行释放堆及存放结点的存储空间，在最坏情况下，需 $O(n)$ 时间。

算法需花费 $O(n2^n)$ 时间。

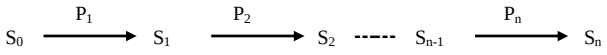
每一个结点需 $O(n)$ 空间，在最坏情况下，有 2^n 个结点，因此，空间复杂性也是 $O(n2^n)$。

动态规划

动态规划的思想方法

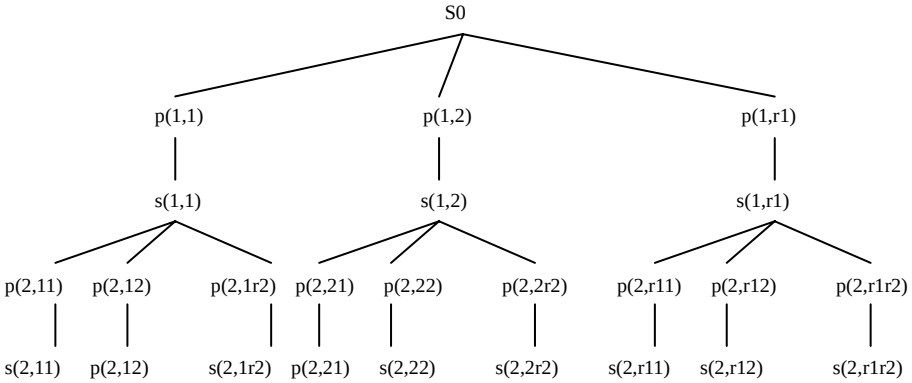
动态规划的最优决策原理

活动过程划分为若干个阶段，每一阶段的决策，依赖于前一阶段的状态，由决策所采取的动作，使状态发生转移，成为下一阶段的决策依据。



动态规划的决策过程

最优性原理：无论过程的初始状态和初始决策是什么，其余决策都必须相对于初始决策所产生的状态，构成一个最优决策序列。



令最优状态为 $s(2,22)$，由此倒推：

$s(2,22) \to p(2,22) \to s(1,2) \to p(1,2) \to s0$

最优决策序列，$p(1,2) \to p(2,22)$

状态转移序列：$s0 \to s(1,2) \to s(2,22)$

赖以决策的策略或目标，称为动态规划函数。

整个决策过程，可以递归地进行，或用循环迭代的方法进行。

动态规划函数可以递归地定义，也可以用递推公式来表达。

最优决策是在最后阶段形成的，然后向前倒推，直到初始阶段；

而决策的具体结果及所产生的状态转移，却是由初始阶段开始进行计算的，然后向后递

归或迭代，直到最终结果。

动态规划实例、货郎担问题

例 货郎担问题。

在有向赋权图 $G=<V,E>$ 中，寻找路径最短的哈密尔顿回路问题。

一、解货郎担问题的过程

令 $d(i,\overline{V})$：从顶点 i 出发，经 $\overline{V}$ 中各顶点一次，最终回到顶点 i 的最短路径的长度，开始时，$\overline{V}=V-\{i\}$。

动态规划函数：

$$d(i,V-\{i\})=d(i,\overline{V})=\min_{k\in V}\{c_{ik}+d(k,\overline{V}-\{k\})\} \quad (6.1.1)$$

$$d(k,\varphi)=c_{ki} \qquad k\neq i \qquad (6.1.2)$$

4 个城市的费用矩阵是：

$$C=(c_{ij})=\begin{pmatrix}\infty & 3 & 6 & 7\\ 5 & \infty & 2 & 3\\ 6 & 4 & \infty & 2\\ 3 & 7 & 5 & \infty\end{pmatrix}$$

根据（6.1.1）式，由城市 1 出发，经城市 2, 3, 4 然后返回 1 的最短路径长度为：

$$d(1,\{2,3,4\})=\min\{c_{12}+d(2,\{3,4\}),c_{13}+d(3,\{2,4\}),c_{14}+d(4,\{2,3\})\}$$

它必须依据 $d(2,\{3,4\}),d(3,\{2,4\}),d(4,\{2,3\})$ 的计算结果：

$$d(2,\{3,4\})=\min\{c_{23}+d(3,\{4\}),c_{24}+d(4,\{3\})\}$$

$$d(3,\{2,4\})=\min\{c_{32}+d(2,\{4\}),c_{34}+d(4,\{2\})\}$$

$$d(4,\{2,3\})=\min\{c_{42}+d(2,\{3\}),c_{43}+d(3,\{2\})\}$$

这一阶段的决策，又必须依据下面的计算结果：

$$d(3,\{4\}),d(4,\{3\}),d(2,\{4\}),d(4,\{2\}),d(2,\{3\}),d(3,\{2\})$$

再向前倒推，有：

$$d(3,\{4\})=c_{34}+d(4,\varphi)=c_{34}+c_{41}=2+3=5$$

$$d(4,\{3\})=c_{43}+d(3,\varphi)=c_{43}+c_{31}=5+6=11$$

$$d(2,\{4\})=c_{24}+d(4,\varphi)=c_{24}+c_{41}=3+3=6$$

$$d(4,\{2\})=c_{42}+d(2,\varphi)=c_{42}+c_{21}=7+5=12$$

$$d(2,\{3\})=c_{23}+d(3,\varphi)=c_{23}+c_{31}=2+6=8$$

$$d(3,\{2\})=c_{32}+d(2,\varphi)=c_{32}+c_{21}=4+5=9$$

有了这些结果，再向后计算，有：

$$d(2,\{3,4\})=\min\{2+5,3+11\}=7$$

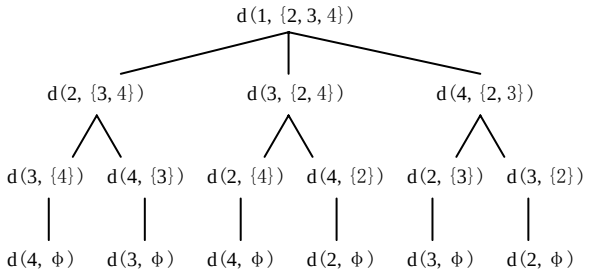
路径顺序是：2, 3, 4, 1

$$d(3,\{2,4\}) = \min\{4+6\ ,2+12\} = 10 \qquad \text{路径顺序是：3, 2, 4, 1}$$

$$d(4,\{2,3\}) = \min\{7+8,5+9\} = 14 \qquad \text{路径顺序是：4, 3, 2, 1}$$

最后：

$$d(1,\{2,3,4\}) = \min\{3+7,6+10,7+14\} = 10 \qquad \text{路径顺序是：1, 2, 3, 4, 1}$$



货郎担问题求解过程示意图

二、复杂性分析

令 N_i 是计算从顶点 i 出发，返回顶点 i 所需计算的形式为 $d(k,\overline{V}-\{k\})$ 的个数。

开始计算 $d(i,V-\{i\})$ 时，集合 $V-\{i\}$ 中有 $n-1$ 个城市。

以后，在计算 $d(k,\overline{V}-\{k\})$ 时，集合 $\overline{V}-\{k\}$ 的城市数目，在不同的决策阶段，分别为 $n-2$, ---, 0。

在整个计算中，需要计算大小为 j 的不同城市集合的个数为 C_{n-1}^{j}， $j=0,1,\cdots n-1$。因此，总个数为：

$$N_i = \sum_{j=0}^{n-1} C_{n-1}^{j}$$

当 $\overline{V}-\{k\}$ 集合中的城市个数为 j 时，为计算 $d(k,\overline{V}-\{k\})$，需 j 次加法， $j-1$ 次比较。从 i 城市出发，经其它城市再回到 i，总的运算时间 T_i 为：

$$T_i = \sum_{j=0}^{n-1} j\cdot C_{n-1}^{j} < \sum_{j=0}^{n-1} n\cdot C_{n-1}^{j} = n\sum_{j=0}^{n-1} C_{n-1}^{j}$$

由二项式定理：

$$(x+y)^n = \sum_{j=0}^{n} C_n^j x^j y^{n-j}$$

令 $x=y=1$；可得：

$$T_i < n\cdot 2^{n-1} = O(n2^n)$$

则用动态规划方法求解货郎担问题，总的花费 T 为：

$$T = \sum_{i=1}^{n} T_i < n^2\cdot 2^{n-1} = O(n^2 2^n)$$

0/1 背包问题

0/1 背包问题的求解过程

一、动态规划函数

x_i：物体 i 被装入背包的情况，$x_i = 0,1$。约束方程和目标函数：

$$\sum_{i=1}^{n} w_i x_i \le M \qquad optp = \max \sum_{i=1}^{n} p_i x_i$$

解向量 $X = (x_1, x_2, \cdots, x_n)$。

背包的载重量：$0 \sim m$

$optp_i(j)$：前 i 个物体中，能装入载重量为 j 的背包中的物体的最大价值，$j = 1, 2, \cdots, m$。

动态规划函数：

$$optp_i(0) = optp_0(j) = 0 \qquad (6.6.1)$$

$$optp_i(j) = \begin{cases} optp_{i-1}(j) & j < w_i \\ \max\{optp_{i-1}(j), optp_{i-1}(j - w_i) + p_i\} & j > w_i \end{cases} \qquad (6.6.2)$$

二、求解过程

1、决策阶段

第一阶段，只装入一个物体，确定在各种不同载重量的背包下，能够得到的最大价值；

第二阶段，装入前两个物体，确定在各种不同载重量的背包下，能够得到的最大价值；

依此类推，直到第 n 个阶段。

最后，$optp_n(m)$ 便是在载重量为 m 的背包下，装入 n 个物体时，能够取得的最大价值。

2、解向量的确定

从 $optp_n(m)$ 的值向前倒推。

递推关系式：

若 $optp_i(j) \le optp_{i-1}(j)$ 则 $x_i = 0$ （6.6.3）

若 $optp_i(j) > optp_{i-1}(j)$ 则 $x_i = 1$，$j = j - w_i$ （6.6.4）

例 6.6 有 5 个物体，其重量分别为 2, 2, 6, 5, 4，价值分别为 6, 3, 5, 4, 6，背包的载重量为 10，求装入背包的物体及其总价值

计算结果，如图所示。

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	***6***	6	6	6	6	6	6
2	0	0	6	6	9	9	***9***	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0	0	6	6	9	9	9	10	11	13	14
5	0	0	6	6	9	9	12	12	15	15	***15***

5 个物体的 0/1 背包问题的例子

装入背包的物体为 $x=\{1,1,0,0,1\}$ 。

6.6.2 0/1 背包问题的实现

数据结构。下面的数据用于算法的输入和输出：

```
int   w[n];              /* n 个物体的重量 */
Type  p[n];              /* n 个物体的价值 */
int   m;                 /* 背包的载重量 */
BOOL  x[n];              /* 装入背包的物体,元素为 true 时,对应物体被装入 */
Type  v;                 /* 装入背包中物体的最大价值 */
```

下面的数据用于算法的工作单元：

```
Type  optp[n+1][m+1];    /* i 个物体装入载重量为 j 的背包中的最大价值 */
```

算法描述如下：

算法 6.5 0/1 背包物体的动态规划算法

输入：物体的重量 w[]和价值 p[],物体的个数 n,背包的载重量 m

输出：装入背包的物体 x[],背包中物体的最大价值 v

```
1. template <class Type>
2. Type knapsack_dynamic(int w[],Type p[],int n,int m,BOOL x[])
3. {
4.    int i,j,k;
5.    Type v,(*optp)[m+1] = new Type[n+1][m+1]; /* 分配工作单元 */
6.    for (i=0;i<=n;i++){                       /* 初始化第 0 列 */
7.       optp[i][0] = 0;   x[i] = FALSE;    /* 解向量初始化为 FALSE */
8.    }
9.    for (i=0;i<=m;i++)                     /* 初始化第 0 行 */
```

```
10.        optp[0][i] = 0;
11.    for (i=1;i<=n;i++) {                  /* 计算 optp[i][j] */
12.        for (j=1;j<=m;j++) {
13.            optp[i][j] = optp[i-1][j];
14.            if ((j>=w[i])&&(optp[i-1,j-w[i]]+p[i]>optp[i-1][j])
15.                optp[i][j] = optp[i-1,j-w[i]]+p[i];
16.        }
17.    }
18.    j = m;                                 /* 递推装入背包的物体 */
19.    for (i=n;i>0;i--) {
20.        if (optp[i][j]>optp[i-1][j]) {
21.            x[i] = TRUE;   j = j – w[i];
22.        }
23.    }
24.    v = optp[n][m];
25.    delete optp;                       /* 释放工作单元 */
26.    return v;                          /* 返回最大价值 */
27. }
```

时间复杂性是$\Theta(nm)$。

第 6~8 行、第 9~10 行都花费$\Theta(m)$时间；

第 11~17 行花费$\Theta(nm)$时间；

第 18~23 行花费$\Theta(n)$时间；

工作空间是$O(nm)$

背包问题九讲

目录

前言

本篇文章是我(dd_engi)正在进行中的一个雄心勃勃的写作计划的一部分，这个计划的内容是写作一份较为完善的 NOIP 难度的动态规划总结，名为《解动态规划题的基本思考方式》。现在你看到的是这个写作计划最先发布的一部分。

背包问题是一个经典的动态规划模型。它既简单形象容易理解，又在某种程度上能够揭示动态规划的本质，故不少教材都把它作为动态规划部分的第一道例题，我也将它放在我的写作计划的第一部分。

读本文最重要的是思考。因为我的语言和写作方式向来不以易于理解为长，思路也偶有跳跃的地方，后面更有需要大量思考才能理解的比较抽象的内容。更重要的是：不大量思考，绝对不可能学好动态规划这一信息学奥赛中最精致的部分。

你现在看到的是本文的 1.0 正式版。我会长期维护这份文本，把大家的意见和建议融入其中，也会不断加入我在 OI 学习以及将来可能的 ACM-ICPC 的征程中得到的新的心得。但目前本文还没有一个固定的发布页面，想了解本文是否有更新版本发布，可以在 OIBH 论坛中以“背包问题九讲”为关键字搜索贴子，每次比较重大的版本更新都会在这里发贴公布。

目录

[第三讲 多重背包问题]

每种物品有一个固定的次数上限。

[第四讲 混合三种背包问题]

将前面三种简单的问题叠加成较复杂的问题。

[第五讲 二维费用的背包问题]

一个简单的常见扩展。

[第六讲 分组的背包问题]

一种题目类型，也是一个有用的模型。后两节的基础。

[第七讲 有依赖的背包问题]

另一种给物品的选取加上限制的方法。

[第八讲 泛化物品]

我自己关于背包问题的思考成果，有一点抽象。

[第九讲 背包问题问法的变化]

试图触类旁通、举一反三。

[附：USACO 中的背包问题]

给出 USACO Training 上可供练习的背包问题列表，及简单的解答。

联系方式

如果有任何意见和建议，特别是文章的错误和不足，或者希望为文章添加新的材料，可以通过 http://kontactr.com/user/tianyi/这个网页联系我。

致谢

感谢以下名单：

- 阿坦
- jason911
- donglixp

他们每人都最先指出了本文第一个 beta 版中的某个并非无关紧要的错误。谢谢你们如此仔细地阅读拙作并弥补我的疏漏。

感谢 XiaQ，它针对本文的第一个 beta 版发表了用词严厉的六条建议，虽然我只认同并采纳了其中的两条。在所有读者几乎一边倒的赞扬将我包围的当时，你的贴子是我的一剂清醒剂，让我能清醒起来并用更严厉的眼光审视自己的作品。

当然，还有用各种方式对我表示鼓励和支持的几乎无法计数的同学。不管是当面赞扬，或是在论坛上回复我的贴子，不管是发来热情洋溢的邮件，或是在即时聊天的窗口里竖起大拇指，你们的鼓励和支持是支撑我的写作计划的强大动力，也鞭策着我不断提高自身水平，谢谢你们！

最后，感谢 Emacs 这一世界最强大的编辑器的所有贡献者，感谢它的插件 EmacsMuse 的开发者们，本文的所有编辑工作都借助这两个卓越的自由软件完成。谢谢你们——自由软件社群——为社会提供了如此有生产力的工具。我深深钦佩你们身上体现出的自由软件的精神，没有你们的感召，我不能完成本文。在你们的影响下，采用自由文档的方式发布本文档，也是我对自由社会事业的微薄努力。

P01：01背包问题

题目

有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使价值总和最大。

基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即f[i][v]表示前i件物品恰放入一个容量为v的背包可以获得的最大价值。则其状态转移方程便是：

f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前i件物品放入容量为v的背包中”这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为v的背包中”，价值为f[i-1][v]；如果放第i件物品，那么问题就转化为“前i-1件物品放入剩下的容量为v-c[i]的背包中”，此时能获得的最大价值就是f[i-1][v-c[i]]再加上通过放入第i件物品获得的价值w[i]。

优化空间复杂度

以上方法的时间和空间复杂度均为O(N*V)，其中时间复杂度基本已经不能再优化了，但空间复杂度却可以优化到O(V)。

先考虑上面讲的基本思路如何实现，肯定是有一个主循环i=1..N，每次算出来二维数组f[i][0..V]的所有值。那么，如果只用一个数组f[0..V]，能不能保证第i次循环结束后f[v]中表示的就是我们定义的状态f[i][v]呢？f[i][v]是由f[i-1][v]和f[i-1][v-c[i]]两个子问题递推而来，能否保证在推f[i][v]时（也即在第i次主循环中推f[v]时）能够得到f[i-1][v]和f[i-1][v-c[i]]的值呢？事实上，这要求在每次主循环中我们以v=V..0的顺序推f[v]，这样才能保证推f[v]时f[v-c[i]]保存的是状态f[i-1][v-c[i]]的值。伪代码如下：

```
for i=1..N
    for v=V..0
        f[v]=max{f[v],f[v-c[i]]+w[i]};
```

其中的f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}，因为现在的f[v-c[i]]就相当于原来的

f[i-1][v-c[i]]。如果将 v 的循环顺序从上面的逆序改成顺序的话，那么则成了 f[i][v]由 f[i][v-c[i]]推知，与本题意不符，但它却是另一个重要的背包问题 P02 最简捷的解决方案，故学习只用一维数组解 01 背包问题是十分必要的。

事实上，使用一维数组解 01 背包的程序在后面会被多次用到，所以这里抽象出一个处理一件 01 背包中的物品过程，以后的代码中直接调用不加说明。

过程 ZeroOnePack，表示处理一件 01 背包中的物品，两个参数 cost、weight 分别表明这件物品的费用和价值。

```
procedure ZeroOnePack(cost,weight)
    for v=V..cost
        f[v]=max{f[v],f[v-cost]+weight}
```

注意这个过程里的处理与前面给出的伪代码有所不同。前面的示例程序写成 v=V..0 是为了在程序中体现每个状态都按照方程求解了，避免不必要的思维复杂度。而这里既然已经抽象成看作黑箱的过程了，就可以加入优化。费用为 cost 的物品不会影响状态 f[0..cost-1]，这是显然的。

有了这个过程以后，01 背包问题的伪代码就可以这样写：

```
for i=1..N
    ZeroOnePack(c[i],w[i]);
```

初始化的细节问题

我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解，有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法，要求恰好装满背包，那么在初始化时除了 f[0]为 0 其它 f[1..V]均设为-∞，这样就可以保证最终得到的 f[N]是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将 f[0..V]全部设为 0。

为什么呢？可以这样理解：初始化的 f 数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为 0 的背包可能被价值为 0 的 nothing “恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，它们的值就都应该是-∞了。如果背包并非必须被装满，那么任何容量的背包都有一个合法解“什么都不装”，这个解的价值为 0，所以初始时状态的值也就全部为 0 了。

这个小技巧完全可以推广到其它类型的背包问题，后面也就不再对进行状态转移之前的初始化进行讲解。

小结

01 背包问题是最基本的背包问题，它包含了背包问题中设计状态、方程的最基本思想，另外，别的类型的背包问题往往也可以转换成 01 背包问题求解。故一定要仔细体会上面基本思路的得出方法，状态转移方程的意义，以及最后怎样优化的空间复杂度。

首页

P02：完全背包问题

题目

有 N 种物品和一个容量为 V 的背包，每种物品都有无限件可用。第 i 种物品的费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本思路

这个问题非常类似于 01 背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取 0 件、取 1 件、取 2 件.....等很多种。如果仍然按照解 01 背包时的思路，令 f[i][v]表示前 i 种物品恰放入一个容量为 v 的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：

f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k*c[i]<=v}

这跟 01 背包问题一样有 O(N*V)个状态需要求解，但求解每个状态的时间已经不是常数了，求解状态 f[i][v]的时间是 O(v/c[i])，总的复杂度是超过 O(VN)的。

将 01 背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明 01 背包问题的方程的确是很重要，可以推及其它类型的背包问题。但我们还是试图改进这个复杂度。

一个简单有效的优化

完全背包问题有一个很简单有效的优化，是这样的：若两件物品 i、j 满足 c[i]<=c[j]且 w[i]>=w[j]，则将物品 j 去掉，不用考虑。这个优化的正确性显然：任何情况下都可将价值小费用高得 j 换成物美价廉的 i，得到至少不会更差的方案。对于随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为有可能特别设计的数据可以一件物品也去不掉。

这个优化可以简单的 O(N^2)地实现，一般都可以承受。另外，针对背包问题而言，比较不错的一种方法是：首先将费用大于 V 的物品去掉，然后使用类似计数排序的做法，计算出费用相同的物品中价值最高的是哪个，可以 O(V+N)地完成这个优化。这个不太重要的过程就不给出伪代码了，希望你能独立思考写出伪代码或程序。

转化为 01 背包问题求解

既然01背包问题是最基本的背包问题，那么我们可以考虑把完全背包问题转化为01背包问题来解。最简单的想法是，考虑到第i种物品最多选V/c[i]件，于是可以把第i种物品转化为V/c[i]件费用及价值均不变的物品，然后求解这个01背包问题。这样完全没有改进基本思路的时间复杂度，但这毕竟给了我们将完全背包问题转化为01背包问题的思路：将一种物品拆成多件物品。

更高效的转化方法是：把第i种物品拆成费用为c[i]*2^k、价值为w[i]*2^k的若干件物品，其中k满足c[i]*2^k<=V。这是二进制的思想，因为不管最优策略选几件第i种物品，总可以表示成若干个2^k件物品的和。这样把每种物品拆成O(log(V/c[i]))件物品，是一个很大的改进。

但我们有更优的O(VN)的算法。

O(VN)的算法

这个算法使用一维数组，先看伪代码：

```
for i=1..N
    for v=0..V
        f[v]=max{f[v],f[v-cost]+weight}
```

你会发现，这个伪代码与P01的伪代码只有v的循环次序不同而已。为什么这样一改就可行呢？首先想想为什么P01中要按照v=V..0的逆序来循环。这是因为要保证第i次循环中的状态f[i][v]是由状态f[i-1][v-c[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个绝无已经选入第i件物品的子结果f[i-1][v-c[i]]。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果f[i][v-c[i]]，所以就可以并且必须采用v=0..V的顺序循环。这就是这个简单的程序为何成立的道理。

这个算法也可以以另外的思路得出。例如，基本思路中的状态转移方程可以等价地变形成这种形式：

```
f[i][v]=max{f[i-1][v],f[i][v-c[i]]+w[i]}
```

将这个方程用一维数组实现，便得到了上面的伪代码。

最后抽象出处理一件完全背包类物品的过程伪代码，以后会用到：

```
procedure CompletePack(cost,weight)
    for v=cost..V
        f[v]=max{f[v],f[v-cost]+weight}
```

总结

完全背包问题也是一个相当基础的背包问题，它有两个状态转移方程，分别在“基本思路”以及“O(VN)的算法“的小节中给出。希望你能够对这两个状态转移方程都仔细地体会，不仅记住，也要弄明白它们是怎么得出来的，最好能够自己想一种得到这些方程的方法。事实上，对每一道动态规划题目都思考其方程的意义以及如何得来，是加深对动态规划的理解、提高动态规划功力的好方法。

首页

P03：多重背包问题

题目

有 N 种物品和一个容量为 V 的背包。第 i 种物品最多有 n[i]件可用，每件费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本算法

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可，因为对于第 i 种物品有 n[i]+1 种策略：取 0 件，取 1 件……取 n[i]件。令 f[i][v]表示前 i 种物品恰放入一个容量为 v 的背包的最大权值，则有状态转移方程：

f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k<=n[i]}

复杂度是 O(V*Σn[i])。

转化为 01 背包问题

另一种好想好写的基本方法是转化为 01 背包求解：把第 i 种物品换成 n[i]件 01 背包中的物品，则得到了物品数为 Σn[i]的 01 背包问题，直接求解，复杂度仍然是 O(V*Σn[i])。

但是我们期望将它转化为 01 背包问题之后能够像完全背包一样降低复杂度。仍然考虑二进制的思想，我们考虑把第 i 种物品换成若干件物品，使得原问题中第 i 种物品可取的每种策略——取 0..n[i]件——均能等价于取若干件代换以后的物品。另外，取超过 n[i]件的策略必不能出现。

方法是：将第 i 种物品分成若干件物品，其中每件物品有一个系数，这件物品的费用和价值均是原来的费用和价值乘以这个系数。使这些系数分别为 1,2,4,...,2^(k-1),n[i]-2^k+1，且 k 是满足 n[i]-2^k+1>0 的最大整数。例如，如果 n[i]为 13，就将这种物品分成系数分别为 1,2,4,6 的四件物品。

分成的这几件物品的系数和为 n[i]，表明不可能取多于 n[i]件的第 i 种物品。另外这种方法也能保证对于 0..n[i]间的每一个整数，均可以用若干个系数的和表示，这个证明可以分 0..2^k-1 和 2^k..n[i]两段来分别讨论得出，并不难，希望你自己思考尝试一下。

这样就将第 i 种物品分成了 O(log n[i])种物品，将原问题转化为了复杂度为 O(V*Σlog n[i])的 01 背包问题，是很大的改进。

下面给出O(log amount)时间处理一件多重背包中物品的过程，其中amount表示物品的数量：

```
procedure MultiplePack(cost,weight,amount)
    if cost*amount>=V
    {
        CompletePack(cost,weight)
        Return
    }
    integer k=1
    while k<amount
    {
        ZeroOnePack(k*cost,k*weight)
        amount=amount-k
        k=k*2
    }
    ZeroOnePack(amount*cost,amount*weight)
```

希望你仔细体会这个伪代码，如果不太理解的话，不妨翻译成程序代码以后，单步执行几次，或者头脑加纸笔模拟一下，也许就会慢慢理解了。

O(VN)的算法

多重背包问题同样有O(VN)的算法。这个算法基于基本算法的状态转移方程，但应用单调队列的方法使每个状态的值可以以均摊O(1)的时间求解。由于用单调队列优化的DP已超出了NOIP的范围，故本文不再展开讲解。我最初了解到这个方法是在楼天成的“男人八题”幻灯片上。

小结

这里我们看到了将一个算法的复杂度由O(V*Σn[i])改进到O(V*Σlog n[i])的过程，还知道了存在应用超出NOIP范围的知识的O(VN)算法。希望你特别注意“拆分物品”的思想和方法，自己证明一下它的正确性，并将完整的程序代码写出来。

首页

P04：混合三种背包问题

问题

如果将 P01、P02、P03 混合起来。也就是说，有的物品只可以取一次（01 背包），有的物品可以取无限次（完全背包），有的物品可以取的次数有一个上限（多重背包）。应该怎么求解呢?

01 背包与完全背包的混合

考虑到在 P01 和 P02 中给出的伪代码只有一处不同，故如果只有两类物品：一类物品只能取一次，另一类物品可以取无限次，那么只需在对每个物品应用转移方程时，根据物品的类别选用顺序或逆序的循环即可，复杂度是 O(VN)。伪代码如下：

```
for i=1..N
    if 第 i 件物品是 01 背包
        for v=V..0
            f[v]=max{f[v],f[v-c[i]]+w[i]};
    else if 第 i 件物品是完全背包
        for v=0..V
            f[v]=max{f[v],f[v-c[i]]+w[i]};
```

再加上多重背包

如果再加上有的物品最多可以取有限次，那么原则上也可以给出 O(VN)的解法：遇到多重背包类型的物品用单调队列解即可。但如果不考虑超过 NOIP 范围的算法的话，用 P03 中将每个这类物品分成 O(log n[i])个 01 背包的物品的方法也已经很优了。

当然，更清晰的写法是调用我们前面给出的三个相关过程。

```
for i=1..N
    if 第 i 件物品是 01 背包
        ZeroOnePack(c[i],w[i])
    else if 第 i 件物品是完全背包
        CompletePack(c[i],w[i])
    else if 第 i 件物品是多重背包
        MultiplePack(c[i],w[i],n[i])
```

在最初写出这三个过程的时候，可能完全没有想到它们会在这里混合应用。我想这体现了编程中抽象的威力。如果你一直就是以这种“抽象出过程”的方式写每一

类背包问题的，也非常清楚它们的实现中细微的不同，那么在遇到混合三种背包问题的题目时，一定能很快想到上面简洁的解法，对吗?

小结

有人说，困难的题目都是由简单的题目叠加而来的。这句话是否公理暂且存之不论，但它在本讲中已经得到了充分的体现。本来 01 背包、完全背包、多重背包都不是什么难题，但将它们简单地组合起来以后就得到了这样一道一定能吓倒不少人的题目。但只要基础扎实，领会三种基本背包问题的思想，就可以做到把困难的题目拆分成简单的题目来解决。

首页

P05：二维费用的背包问题

问题

二维费用的背包问题是指：对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价；对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。设这两种代价分别为代价 1 和代价 2，第 i 件物品所需的两种代价分别为 a[i]和 b[i]。两种代价可付出的最大值（两种背包容量）分别为 V 和 U。物品的价值为 w[i]。

算法

费用加了一维，只需状态也加一维即可。设 f[i][v][u]表示前 i 件物品付出两种代价分别为 v 和 u 时可获得的最大价值。状态转移方程就是：

f[i][v][u]=max{f[i-1][v][u],f[i-1][v-a[i]][u-b[i]]+w[i]}

如前述方法，可以只使用二维的数组：当每件物品只可以取一次时变量 v 和 u 采用逆序的循环，当物品有如完全背包问题时采用顺序的循环。当物品有如多重背包问题时拆分物品。这里就不再给出伪代码了，相信有了前面的基础，你能够自己实现出这个问题的程序。

物品总个数的限制（？？？）

有时，“二维费用”的条件是以这样一种隐含的方式给出的：最多只能取 M 件物品。这事实上相当于每件物品多了一种“件数”的费用，每个物品的件数费用均为 1，可以付出的最大件数费用为 M。换句话说，设 f[v][m]表示付出费用 v、最多选 m 件时可得到的最大价值，则根据物品的类型（01、完全、多重）用不同的方法循环更新，最后在 f[0..V][0..M]范围内寻找答案。

小结

当发现由熟悉的动态规划题目变形得来的题目时，在原来的状态中加一纬以满足新的限制是一种比较通用的方法。希望你能从本讲中初步体会到这种方法。

首页

P06：分组的背包问题

问题

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 c[i]，价值是 w[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

算法

这个问题变成了每组物品有若干种策略：是选择本组的某一件，还是一件都不选。也就是说设 f[k][v]表示前 k 组物品花费费用 v 能取得的最大权值，则有：

f[k][v]=max{f[k-1][v],f[k-1][v-c[i]]+w[i]|物品 i 属于第 k 组}

使用一维数组的伪代码如下：

```
for 所有的组 k
    for v=V..0
        for 所有的 i 属于组 k
            f[v]=max{f[v],f[v-c[i]]+w[i]}
```

注意这里的三层循环的顺序，甚至在本文的 beta 版中我自己都写错了。“for v=V..0”这一层循环必须在“for 所有的 i 属于组 k”之外。这样才能保证每一组内的物品最多只有一个会被添加到背包中。

另外，显然可以对每组内的物品应用 P02 中“一个简单有效的优化”。

小结

分组的背包问题将彼此互斥的若干物品称为一个组，这建立了一个很好的模型。不少背包问题的变形都可以转化为分组的背包问题（例如 P07），由分组的背包问题进一步可定义“泛化物品”的概念，十分有利于解题。

首页

P07：有依赖的背包问题

简化的问题

这种背包问题的物品间存在某种“依赖”的关系。也就是说，i 依赖于 j，表示若选物品 i，则必须选物品 j。为了简化起见，我们先设没有某个物品既依赖于别的物品，又被别的物品所依赖；另外，没有某件物品同时依赖多件物品。

算法

这个问题由 NOIP2006 金明的预算方案一题扩展而来。遵从该题的提法，将不依赖于别的物品的物品称为“主件”，依赖于某主件的物品称为“附件”。由这个问题的简化条件可知所有的物品由若干主件和依赖于每个主件的一个附件集合组成。

按照背包问题的一般思路，仅考虑一个主件和它的附件集合。可是，可用的策略非常多，包括：一个也不选，仅选择主件，选择主件后再选择一个附件，选择主件后再选择两个附件……无法用状态转移方程来表示如此多的策略。（事实上设有 n 个附件，则策略有 2^n+1 个，为指数级。）

考虑到所有这些策略都是互斥的（也就是说，你只能选择一种策略），所以一个主件和它的附件集合实际上对应于 P06 中的一个物品组，每个选择了主件又选择了若干个附件的策略对应于这个物品组中的一个物品，其费用和价值都是这个策略中的物品的值的和。但仅仅是这一步转化并不能给出一个好的算法，因为物品组中的物品还是像原问题的策略一样多。

再考虑 P06 中的一句话：*可以对每组中的物品应用 P02 中“一个简单有效的优化”*。这提示我们，对于一个物品组中的物品，所有费用相同的物品只留一个价值最大的，不影响结果。所以，我们可以对主件 i 的“附件集合”先进行一次 01 背包，得到费用依次为 0..V-c[i]所有这些值时相应的最大价值 f'[0..V-c[i]]。那么这个主件及它的附件集合相当于 V-c[i]+1 个物品的物品组，其中费用为 c[i]+k 的物品的价值为 f'[k]+w[i]。也就是说原来指数级的策略中有很多策略都是冗余的，通过一次 01 背包后，将主件 i 转化为 V-c[i]+1 个物品的物品组，就可以直接应用 P06 的算法解决问题了。

较一般的问题

更一般的问题是：依赖关系以图论中“森林”的形式给出（森林即多叉树的集合），也就是说，主件的附件仍然可以具有自己的附件集合，限制只是每个物品最多只依赖于一个物品（只有一个主件）且不出现循环依赖。

解决这个问题仍然可以用将每个主件及其附件集合转化为物品组的方式。唯一不同的是，由于附件可能还有附件，就不能将每个附件都看作一个一般的 01 背包中的物品了。若这个附件也有附件集合，则它必定要被先转化为物品组，然后用分组的背包问题解出主件及其附件集合所对应的附件组中各个费用的附件所对应的价值。

事实上，这是一种树形 DP，其特点是每个父节点都需要对它的各个儿子的属性进行一次 DP 以求得自己的相关属性。这已经触及到了“泛化物品”的思想。看完 P08 后，你会发现这个“依赖关系树”每一个子树都等价于一件泛化物品，求某节点为根的子树对应的泛化物品相当于求其所有儿子的对应的泛化物品之和。

小结

NOIP2006 的那道背包问题我做得很失败，写了上百行的代码，却一分未得。后来我通过思考发现通过引入“物品组”和“依赖”的概念可以加深对这题的理解，还可以解决它的推广问题。用物品组的思想考虑那题中极其特殊的依赖关系：物品不能既作主件又作附件，每个主件最多有两个附件，可以发现一个主件和它的两个附件等价于一个由四个物品组成的物品组，这便揭示了问题的某种本质。

我想说：失败不是什么丢人的事情，从失败中全无收获才是。

首页

P08：泛化物品

定义

考虑这样一种物品，它并没有固定的费用和价值，而是它的价值随着你分配给它的费用而变化。这就是泛化物品的概念。

更严格的定义之。在背包容量为V的背包问题中，泛化物品是一个定义域为0..V中的整数的函数h，当分配给它的费用为v时，能得到的价值就是h(v)。

这个定义有一点点抽象，另一种理解是一个泛化物品就是一个数组h[0..V]，给它费用v，可得到价值h[V]。

一个费用为c价值为w的物品，如果它是01背包中的物品，那么把它看成泛化物品，它就是除了h(c)=w其它函数值都为0的一个函数。如果它是完全背包中的物品，那么它可以看成这样一个函数，仅当v被c整除时有h(v)=v/c*w，其它函数值均为0。如果它是多重背包中重复次数最多为n的物品，那么它对应的泛化物品的函数有h(v)=v/c*w仅当v被c整除且v/c<=n，其它情况函数值均为0。

一个物品组可以看作一个泛化物品h。对于一个0..V中的v，若物品组中不存在费用为v的的物品，则h(v)=0，否则h(v)为所有费用为v的物品的最大价值 。P07中每个主件及其附件集合等价于一个物品组，自然也可看作一个泛化物品。

泛化物品的和

如果面对两个泛化物品h和l，要用给定的费用从这两个泛化物品中得到最大的价值，怎么求呢？事实上，对于一个给定的费用v，只需枚举将这个费用如何分配给两个泛化物品就可以了。同样的，对于0..V的每一个整数v，可以求得费用v分配到h和l中的最大价值f(v)。也即f(v)=max{h(k)+l(v-k)|0<=k<=v}。可以看到，f也是一个由泛化物品h和l决定的定义域为0..V的函数也就是说，f是一个由泛化物品h和l决定的泛化物品。

由此可以定义泛化物品的和：h、l都是泛化物品，若泛化物品f满足f(v)=max{h(k)+l(v-k)|0<=k<=v}，则称f是h与l的和，即f=h+l。这个运算的时间复杂度取决于背包的容量，是O(V^2)。

泛化物品的定义表明：在一个背包问题中，若将两个泛化物品代以它们的和，不影响问题的答案。事实上，对于其中的物品都是泛化物品的背包问题，求它的答案的过程也就是求所有这些泛化物品之和的过程。设此和为s，则答案就是s[0..V]中的最大值。

背包问题的泛化物品

一个背包问题中，可能会给出很多条件，包括每种物品的费用、价值等属性，物品之间的分组、依赖等关系等。但肯定能将问题对应于某个泛化物品。也就是说给定了所有条件以后，就可以对每个非负整数 v 求得：若背包容量为 v，将物品装入背包可得到的最大价值是多少，这可以认为是定义在非负整数集上的一件泛化物品。这个泛化物品——或者说问题所对应的一个定义域为非负整数的函数——包含了关于问题本身的高度浓缩的信息。一般而言，求得这个泛化物品的一个子域（例如 0..V）的值之后，就可以根据这个函数的取值得到背包问题的最终答案。

综上所述，一般而言，求解背包问题，即求解这个问题所对应的一个函数，即该问题的泛化物品。而求解某个泛化物品的一种方法就是将它表示为若干泛化物品的和然后求之。

小结

本讲可以说都是我自己的原创思想。具体来说，是我在学习函数式编程的 Scheme 语言时，用函数编程的眼光审视各类背包问题得出的理论。这一讲真的很抽象，也许在“模型的抽象程度”这一方面已经超出了 NOIP 的要求，所以暂且看不懂也没关系。相信随着你的 OI 之路逐渐延伸，有一天你会理解的。

我想说：“思考”是一个 OIer 最重要的品质。简单的问题，深入思考以后，也能发现更多。

首页

P09：背包问题问法的变化

以上涉及的各种背包问题都是要求在背包容量（费用）的限制下求可以取到的最大价值，但背包问题还有很多种灵活的问法，在这里值得提一下。但是我认为只要深入理解了求背包问题最大价值的方法，即使问法变化了，也是不难想出算法的。

例如，求解最多可以放多少件物品或者最多可以装满多少背包的空间。这都可以根据具体问题利用前面的方程求出所有状态的值（f 数组）之后得到。

还有，如果要求的是“总价值最小”“总件数最小”，只需简单的将上面的状态转移方程中的 max 改成 min 即可。

下面说一些变化更大的问法。

输出方案

一般而言，背包问题是要求一个最优值，如果要求输出这个最优值的方案，可以参照一般动态规划问题输出方案的方法：记录下每个状态的最优值是由状态转移方程的哪一项推出来的，换句话说，记录下它是由哪一个策略推出来的。便可根据这条策略找到上一个状态，从上一个状态接着向前推即可。

还是以 01 背包为例，方程为 f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。再用一个数组 g[i][v]，设 g[i][v]=0 表示推出 f[i][v]的值时是采用了方程的前一项（也即 f[i][v]=f[i-1][v]），g[i][v]表示采用了方程的后一项。注意这两项分别表示了两种策略：未选第 i 个物品及选了第 i 个物品。那么输出方案的伪代码可以这样写（设最终状态为 f[N][V]）：

```
i=N
v=V
while(i>0)
    if(g[i][v]==0)
        print "未选第 i 项物品"
    else if(g[i][v]==1)
        print "选了第 i 项物品"
        v=v-c[i]
```

另外，采用方程的前一项或后一项也可以在输出方案的过程中根据 f[i][v]的值实时地求出来，也即不须纪录 g 数组，将上述代码中的 g[i][v]==0 改成 f[i][v]==f[i-1][v]，g[i][v]==1 改成 f[i][v]==f[i-1][v-c[i]]+w[i]也可。

输出字典序最小的最优方案

这里“字典序最小”的意思是 1..N 号物品的选择方案排列出来以后字典序最小。以输出 01 背包最小字典序的方案为例。

一般而言，求一个字典序最小的最优方案，只需要在转移时注意策略。首先，子问题的定义要略改一些。我们注意到，如果存在一个选了物品 1 的最优方案，那么答案一定包含物品 1，原问题转化为一个背包容量为 v-c[1]，物品为 2..N 的子问题。反之，如果答案不包含物品 1，则转化成背包容量仍为 V，物品为 2..N 的子问题。不管答案怎样，子问题的物品都是以 i..N 而非前所述的 1..i 的形式来定义的，所以状态的定义和转移方程都需要改一下。但也许更简易的方法是先把物品逆序排列一下，以下按物品已被逆序排列来叙述。

在这种情况下，可以按照前面经典的状态转移方程来求值，只是输出方案的时候要注意：从 N 到 1 输入时，如果 f[i][v]==f[i-v]及 f[i][v]==f[i-1][f-c[i]]+w[i]同时成立，应该按照后者（即选择了物品 i）来输出方案。

求方案总数

对于一个给定了背包容量、物品费用、物品间相互关系（分组、依赖等）的背包问题，除了再给定每个物品的价值后求可得到的最大价值外，还可以得到装满背包或将背包装至某一指定容量的方案总数。

对于这类改变问法的问题，一般只需将状态转移方程中的 max 改成 sum 即可。例如若每件物品均是完全背包中的物品，转移方程即为

f[i][v]=sum{f[i-1][v],f[i][v-c[i]]}

初始条件 f[0][0]=1。

事实上，这样做可行的原因在于状态转移方程已经考察了所有可能的背包组成方案。

最优方案的总数

这里的最优方案是指物品总价值最大的方案。以 01 背包为例。

结合求最大总价值和方案总数两个问题的思路，最优方案的总数可以这样求：f[i][v]意义同前述，g[i][v]表示这个子问题的最优方案的总数，则在求 f[i][v]的同时求 g[i][v]的伪代码如下：

```
for i=1..N
   for v=0..V
        f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}
        g[i][v]=0
        if(f[i][v]==f[i-1][v])
            inc(g[i][v],g[i-1][v]
```

```
if(f[i][v]==f[i-1][v-c[i]]+w[i])
    inc(g[i][v],g[i-1][v-c[i]])
```

如果你是第一次看到这样的问题，请仔细体会上面的伪代码。

求次优解、第 K 优解

对于求次优解、第 K 优解类的问题，如果相应的最优解问题能写出状态转移方程用动态规划解决，那么求次优解往往可以相同的复杂度解决，第 K 优解则比求最优解的复杂度上多一个系数 K。

其基本思想是将每个状态都表示成有序队列，将状态转移方程中的 max/min 转化成有序队列的合并。这里仍然以 01 背包为例讲解一下。

首先看 01 背包求最优解的状态转移方程：f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。如果要求第 K 优解，那么状态 f[i][v]就应该是一个大小为 K 的数组 f[i][v][1..K]。其中 f[i][v][k]表示前 i 个物品、背包大小为 v 时，第 k 优解的值。“f[i][v]是一个大小为 K 的数组”这一句，熟悉 C 语言的同学可能比较好理解，或者也可以简单地理解为在原来的方程中加了一维。显然 f[i][v][1..K]这 K 个数是由大到小排列的，所以我们把它认为是一个有序队列。

然后原方程就可以解释为：f[i][v]这个有序队列是由 f[i-1][v]和 f[i-1][v-c[i]]+w[i]这两个有序队列合并得到的。有序队列 f[i-1][v]即 f[i-1][v][1..K]，f[i-1][v-c[i]]+w[i]则理解为在 f[i-1][v-c[i]][1..K]的每个数上加上 w[i]后得到的有序队列。合并这两个有序队列并将结果（的前 K 项）储存到 f[i][v][1..K]中的复杂度是 O(K)。最后的答案是 f[N][V][K]。总的复杂度是 O(NVK)。

为什么这个方法正确呢？实际上，一个正确的状态转移方程的求解过程遍历了所有可用的策略，也就覆盖了问题的所有方案。只不过由于是求最优解，所以其它在任何一个策略上达不到最优的方案都被忽略了。如果把每个状态表示成一个大小为 K 的数组，并在这个数组中有序的保存该状态可取到的前 K 个最优值。那么，对于任两个状态的 max 运算等价于两个由大到小的有序队列的合并。

另外还要注意题目对于“第 K 优解”的定义，将策略不同但权值相同的两个方案是看作同一个解还是不同的解。如果是前者，则维护有序队列时要保证队列里的数没有重复的。

小结

显然，这里不可能穷尽背包类动态规划问题所有的问法。甚至还存在一类将背包类动态规划问题与其它领域（例如数论、图论）结合起来的问题，在这篇论背包问题的专文中也不会论及。但只要深刻领会前述所有类别的背包问题的思路和状态转移方程，遇到其它的变形问法，只要题目难度还属于 NOIP，应该也不难想出算法。触类旁通、举一反三，应该也是一个 OIer 应有的品质吧。

首页

附：USACO 中的背包问题

USACO 是 USA Computing Olympiad 的简称，它组织了很多面向全球的计算机竞赛活动。

USACO Trainng 是一个很适合初学者的题库，我认为它的特色是题目质量高，循序渐进，还配有不错的课文和题目分析。其中关于背包问题的那篇课文（TEXT Knapsack Problems）也值得一看。

另外，USACO Contest 是 USACO 常年组织的面向全球的竞赛系列，在此也推荐 NOIP 选手参加。

我整理了 USACO Training 中涉及背包问题的题目，应该可以作为不错的习题。其中标加号的是我比较推荐的，标叹号的是我认为对 NOIP 选手比较有挑战性的。

题目列表

- Inflate (+) （基本 01 背包）
- Stamps (+)(!) （对初学者有一定挑战性）
- Money
- Nuggets
- Subsets
- Rockers (+) （另一类有趣的“二维”背包问题）
- Milk4 (!) （很怪的背包问题问法，较难用纯 DP 求解）

题目简解

以下文字来自我所撰的《USACO 心得》一文，该文的完整版本，包括我的程序，可在 DD 的 USACO 征程中找到。

Inflate 是加权 01 背包问题，也就是说：每种物品只有一件，只可以选择放或者不放；而且每种物品有对应的权值，目标是使总权值最大或最小。它最朴素的状态转移方程是：f[k][i] = max{f[k-1][i] , f[k-1][i-v[k]]+w[k]} 。f[k][i]表示前 k 件物品花费代价 i 可以得到的最大权值。v[k]和 w[k]分别是第 k 件物品的花费和权值。可以看到， f[k]的求解过程就是使用第 k 件物品对 f[k-1]进行更新的过程。那么事实上就不用使用二维数组，只需要定义 f[i]，然后对于每件物品 k，顺序地检查 f[i]与 f[i-v[k]]+w[k]的大小，如果后者更大，就对前者进行更新。这是背包问题中典型的优化方法。

题目 stamps 中，每种物品的使用量没有直接限制，但使用物品的总量有限制。求第一个不能用这有限个物品组成的背包的大小。（可以这样等价地认为）设 f[k][i] 表示前 k 件物品组成大小为 i 的背包， 最少需要物品的数量。则 f[k]

[i]= min{f[k-1][i],f[k-1][i-j*s[k]]+j}，其中 j 是选择使用第 k 件物品的数目，这个方程运用时可以用和上面一样的方法处理成一维的。求解时先设置一个粗糙的循环上限，即最大的物品乘最多物品数。

Money 是多重背包问题。也就是每个物品可以使用无限多次。要求解的是构成一种背包的不同方案总数。基本上就是把一般的多重背包的方程中的 min 改成 sum 就行了。

Nuggets 的模型也是多重背包。要求求解所给的物品不能恰好放入的背包大小的最大值（可能不存在）。只需要根据“若 i、j 互质，则关于 x、y 的不定方程 i*x+y*j=n 必有正整数解，其中 n>i*j”这一定理得出一个循环的上限。
Subsets 子集和问题相当于物品大小是前 N 个自然数时求大小为 N*(N+1)/4 的 01 背包的方案数。

Rockers 可以利用求解背包问题的思想设计解法。我的状态转移方程如下：f[i][j][t]=max{f[i][j][t-1] , f[i-1][j][t] , f[i-1][j][t-time[i]]+1 , f[i-1][j-1][T]+(t>=time[i])}。其中 f[i][j][t]表示前 i 首歌用 j 张完整的盘和一张录了 t 分钟的盘可以放入的最多歌数，T 是一张光盘的最大容量，t>=time[i]是一个 bool 值转换成 int 取值为 0 或 1。但我后来发现我当时设计的状态和方程效率有点低，如果换成这样：f[i][j]=(a,b)表示前 i 首歌中选了 j 首需要用到 a 张完整的光盘以及一张录了 b 分钟的光盘，会将时空复杂度都大大降低。这种将状态的值设为二维的方法值得注意。

Milk4 是这些类背包问题中难度最大的一道了。很多人无法做到将它用纯 DP 方法求解，而是用迭代加深搜索枚举使用的桶，将其转换成多重背包问题再 DP。由于 USACO 的数据弱，迭代加深的深度很小，这样也可以 AC，但我们还是可以用纯 DP 方法将它完美解决的。设 f[k]为称量出 k 单位牛奶需要的最少的桶数。那么可以用类似多重背包的方法对 f 数组反复更新以求得最小值。然而困难在于如何输出字典序最小的方案。我们可以对每个 i 记录 pre_f[i]和 pre_v[i]。表示得到 i 单位牛奶的过程是用 pre_f[i]单位牛奶加上若干个编号为 pre_v[i]的桶的牛奶。这样就可以一步步求得得到 i 单位牛奶的完整方案。为了使方案的字典序最小，我们在每次找到一个耗费桶数相同的方案时对已储存的方案和新方案进行比较再决定是否更新方案。为了使这种比较快捷，在使用各种大小的桶对 f 数组进行更新时先大后小地进行。USACO 的官方题解正是这一思路。如果认为以上文字比较难理解可以阅读官方程序或我的程序。

首页

整理 by stntwm

多次背包

多次背包问题：给定 n 种物品和一个背包。第 i 种物品 的价值是 Wi ，其体积为 Vi，数量是 Ki 件，背包的容量为 C。可以任意选择装入背包中的物品，求装入背
包中物品的最大总价值。

方法一：可以把此物品拆分成 Ki 个只能用一次的物品，直接套用 0-1 背包问题的经典动规实现，但是效率太低了，需要寻找更高效的算法。此算法时间复杂度为 O(C*∑(Ki))

方法二：拆分成体积和价值分别为原来 1，2 ，4.. 2^m， Ki-2^m 倍的几个物品，用 0-1 背包求解。 时间复杂度为 O(C*∑([log2Ki]))

方法三（本文重点）：（对单调队列没有了解的请参见原论文[本文结尾链接]）对于第 i 种物品来说，已知体积 v，价值 w，数量 k，那么可以按照当前枚举的体积 j 对 v 的余数把整个动规数组分成 v 份，以下是 v=3 的情况：

```
j              0 1 2 3 4 5 6 7 8 ……
j mod v    0 1 2 0 1 2 0 1 2 ……
```

我们可以把每一份分开处理，假设余数为 d。

```
编号 j          0      1          2              3              4
5          ……
对应体积 d      d+v      d+2*v       d+3*v      d+4*v        d+5*v      ……
```

现在看到分组以后，编号 j 可以从 j-k 到 j-1 中的任意一个编号转移而来（因为相邻的体积正好相差 v），这看上去已经和区间最大值有点相似了。但是注意到由于体积不一样，显然体积大的价值也会大于等于体积小的，直接比较是没有意义的，所以还需要把价值修正到同一体积的基础上。比如都退化到 d，也就是说用 F[j*v+d]- j*w 来代替原来的价值进入队列。

对于物品 i，伪代码如下

```
1. FOR d: = 0 TO v-1                                  //枚举余数，分开处理
2.    清空队列
3.    FOR j: = 0 TO (C-d) div v                       //j 枚举标号，对应体积为
j*v+d
4.     INSERT j , F[ j*v+d ] — j * w                  //插入队列
5.     IF A[ L ] < j - k THEN L + 1 → L //如果队列的首元素已经失效
6.     B[ L ] + j * w → F[ j*v+d ]                    //取队列头更新
7.    END FOR
8. END FOR
```

已知单调队列的效率是O(n)，那么加上单调队列优化以后的多次背包，
效率就是O(n*C)了。
（详细请参见原论文）

==========

完整程序如下(Pasca1)：

```
var
   a,b,f:array[0..100000] of longint;
   m,s,c,n,t,i,j,1,r,d:longint;
procedure insert(x,y:longint);
begin
   while (1<=r)and(b[r]<=y) do dec(r);
   inc(r);a[r]:=x;b[r]:=y;
end;
begin
   readln(n,t);                //读入数据 n 为物品个数 t 为背包容量
   for i:=1 to n do
   begin
      read(m,s,c);          //读入当前物品 m 为物品体积、s 为物品价值、c
为物品可用次数（0 表示无限制）
      if (c=0)or(t div m<c) then c:=t div m;
      for d:=0 to m-1 do
      begin
         1:=1;r:=0;     //清空队列
         for j:=0 to (t-d) div m do
         begin
            insert(j,f[j*m+d]-j*s);   //将新的点插入队列
            if a[1]<j-c then inc(1);   //删除失效点
            f[j*m+d]:=b[1]+j*s;         //用队列头的值更新 f[j*m+d]
         end;
      end;
   end;
   writeln(f[t]);
end.
```

==========