第五章 动态规划算法

§1.动态规划算法的基本思想

动态规划方法是处理分段过程最优化问题的一类及其有效的方法。在实际生活中，有一类问题的活动过程可以分成若干个阶段，而且在任一阶段后的行为依赖于该阶段的状态，与该阶段之前的过程是如何达到这种状态的方式无关。这类问题的解决是多阶段的决策过程。20 世纪 50 年代，贝尔曼（Richard Bellman）等人提出了解决这类问题的“最优化原则”，从而创建了求解最优化问题的一种新的算法－－动态规划算法。最优化原则指出，多阶段过程的最优决策序列具有性质：

无论过程的初始状态和初始决策是什么，其余的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。

这要求原问题计算模型的最优解需包含其（相干）子问题的一个最优解（称为***最优子结构性质***）。

动态规划算法采用最优化原则来建立递归关系式（关于求最优值的），在求解问题时有必要验证该递归关系式是否保持最优化原则。若不保持，则动态规划算法不适合求解该计算模型。在得到最优值的递归式之后，需要执行回溯以构造最优解。在使用动态规划算法自顶向下（Top-Down）求解时，每次产生的子问题并不总是新问题，有些子问题反复计算多次，动态规划算法正是利用了这种***子问题重叠性质***，对每一个子问题只计算一次，将其解保存在一个表格中，当再次要解此子问题时，只是简单地调用（用常数时间）一下已有的结果。最优子结构性质和子问题重叠性质是计算模型采用动态规划算法求解的两个基本要素。

例 5.1.1. 多段图问题

设 G=(V,E)是一个赋权有向图,其顶点集 V 被划分成 k>2 个不相交的子集 V_i: $1\le i\le k$，其中，V_1 和 V_k 分别只有一个顶点 s(称为源)和一个顶点 t(称为汇)，图 5-1-1 中所有的边(u,v)的始点和终点都在相邻的两个子集 V_i 和 V_{i+1} 中，而且 $u\in V_i$，$v\in V_{i+1}$。**多阶段图问题是：**求由 s 到 t 的最小成本路径（也叫最短路径）。

对于每一条由 s 到 t 的路径，可以把它看成在 k-2 个阶段做出的某个决策序列的相应结果：第 i 步决策就是确定 V_{i+1} 中哪个顶点在这条路径上。假设

$$s,\ v_2,\ v_3,\ \cdots,\ v_{k-1},\ t$$

是一条由 s 到 t 的最短路径，再假定从源点 s(初始状态)开始，已经作出了到顶点 v_2 的决策(初始决策)，则 v_2 就是初始决策产生的状态。若将 v_2 看成是原问题的子问题的初始状态，这个子问题就是找一条由 v_2 到 t 的最短路径。事实上，路径 v_2，v_3，…，v_{k-1}，t 一定是 v_2 到 t 的一条最短路径。不然，设

$$v_2,\ q_3,\ \cdots,\ q_{k-1},\ t$$

是一条由 v_2 到 t 的比 v_2，v_3，…，v_{k-1}，t 更短的路径，则 s，v_2，q_3，…，q_{k-1}，t 是一条由 s 到 t 的比 s，v_2，v_3，…，v_{k-1}，t 更短的路径，与前面的假设矛盾。这说明多段图问题具有最优子结构性质。

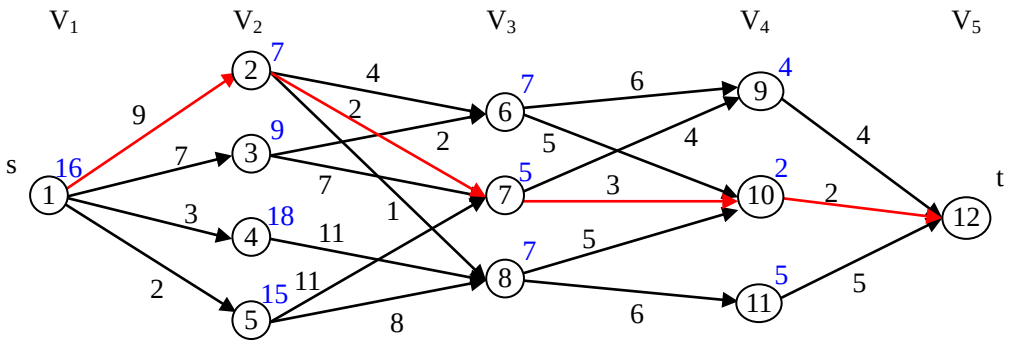


图 5-1-1 一个 5 段图

例5.1.2 0/1 背包问题

有 n 件物品，第 i 件重量和价值分别是 w_i 和 p_i，i=1，2，…，n。要将这 n 件物品的某些件装入容量为 c 的背包中，要求每件物品或整个装入或不装入，不许分割出一部分装入。0/1 背包问题就是要给出装包方法，使得装入背包的物品的总价值最大。这个问题归结为数学规划问题：

$$\max \sum_{1\le i\le n} p_i x_i$$

$$\text{s.t.} \sum_{1\le i\le n} w_i x_i \le c,\quad x_i \in \{0,1\}, i=1,2,\cdots,n \tag{5.1.1}$$

0/1 背包问题的上述计算模型具有最优子结构性质。事实上，若 $y_1, y_2, \cdots, y_n$ 是原问题的最优解，则 $y_2, \cdots, y_n$ 将是 0/1 背包问题的下述子问题：

$$\max \sum_{2\le i\le n} p_i x_i$$

$$\text{s.t.} \sum_{2\le i\le n} w_i x_i \le c - y_1 w_1,\quad x_i \in \{0,1\}, i=2,\cdots,n \tag{5.1.2}$$

的最优解。因为，若 $y'_2, \cdots, y'_n$ 是子问题(5.1.2)的最优解，且使得

$$\sum_{2\le i\le n} p_i y'_i > \sum_{2\le i\le n} p_i y_i \tag{5.1.3}$$

则 $y_1, y'_2, \cdots, y'_n$ 将是原问题(5.1.1)的可行解，并且使得

$$p_1y_1+\sum_{2\le i\le n}p_iy'_i>\sum_{1\le i\le n}p_iy_i \tag{5.1.4}$$

这与 $y_1,y_2,\cdots,y_n$ 是最优解相悖。

例5.1.3　矩阵连乘问题

给定 n 个数字矩阵 A_1，A_2，…，A_n，其中 A_i 与 A_{i+1} 是可乘的，i=1,2,…,n-1. 求矩阵连乘 $A_1A_2\cdots A_n$ 的加括号方法，使得所用的数值乘法运算次数最少。

考察两个矩阵相成的情形：C=AB。如果矩阵 A，B 分别是 p×r 和 r×q 矩阵，则它们的乘积 C 将是 p×q 矩阵，其(i, j)元素为

$$c_{ij}=\sum_{k=1}^{r}a_{ik}b_{kj} \tag{5.1.5}$$

i=1,…,p, j=1,…,q, 因而 AB 所用的数乘次数是 prq。如果有至少 3 个以上的矩阵连乘，则涉及到乘积次序问题，即加括号方法。例如 3 个矩阵连乘的加括号方法有两种：$(A_1A_2)A_3$ 和 $A_1(A_2A_3)$。设 A_1，A_2，A_3 分别是 $p_0\times p_1$，$p_1\times p_2$，$p_2\times p_3$ 矩阵，则以上两种乘法次序所用的数乘次数分别为：

$p_0p_1p_2+p_0p_2p_3$ 和 $p_0p_1p_3+p_1p_2p_3$。

如果 $p_0=10$, $p_1=100$, $p_2=5$, $p_3=50$, 则两种乘法所用的数乘次数分别为：7500 和 75000。可见，由于加括号的方法不同，使得连乘所用的数乘次数有很大差别。对于 n 个矩阵的连乘积，令 P(n)记连乘积的完全加括号数，则有如下递归关系

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & n>1\end{cases} \tag{5.1.6}$$

由此不难算出 P=C(n-1)，其中 C 表示 Catalan 函数：

$$C(n)=\frac{1}{n+1}\binom{2n}{n}=\Omega(4^n/n^{3/2}) \tag{5.1.7}$$

也就是说，P(n)是随 n 指数增长的，所以，我们不能希望列举所有可能次序的连乘积，从中找到具有最少数乘次数的连乘积算法。事实上，矩阵连乘积问题问题的上述计算模型具有最优子结构性质，我们可以采用动态规划的方法，在多项式时间内找到最优的连乘积次序。

用 A[i..j]表示连乘积 $A_iA_{i+1}\cdots A_j$。分析计算 A[1..n]的一个最优加括号方法。设这个加括号方法在矩阵 A_k 和 A_{k+1} 之间将矩阵链分开，即$(A_1A_2\cdots A_k)(A_{k+1}\cdots A_n)$，$1\le k<n$。依次，我们先分别计算 A[1..k]和 A[k+1..n]，然后将计算的结果相乘得到 A[1..n]。可见，A[1..n]的一个最优加括号方法所包含的矩阵子链 A[1..k]和 A[k+1..n]也必定采用了最优加括号方法。也就是说，矩阵连乘问题调的上述计算模型具有最优子结构性质。

如上三个例子都具有最优子结构性质，这个性质决定了解决此类问题的基本思路是：

首先确定问题具有最优子结构性质的计算模型，然后寻找原问题的最优值和子问题的最优值之间的递推关系（自上而下），最后自底向上递归地构造出最优解（自下向上）。

最优子结构性质是最优化原理得以采用的先决条件。一般说来，分阶段选择策略确定最优解的问题往往会形成一个决策序列。Bellman 认为，利用最优化原理以及所获得的递推关系式去求解最优决策序列，可以使枚举数量急剧下降。

这里有一个问题值得注意：最优子结构性质提示我们使用最优化原则产生的算法是递归算法，但只是简单地使用递归算法可能会增加时间与空间开销。例如，下面的用递归式直接计算矩阵连乘积 A[1..n]的算法 RecurMatrixChain 的时间复杂度将是$\Omega(2^n)$：

程序 5-1-1 计算矩阵连乘的递归算法

```
int RecurMatrixChain(int i, int j)
{
   if (i==j) return 0;
   int u=RecurMatrixChain(i, i)
      +RecurMatrixChain(i+1,j)
      +p[i-1]*p[i]*p[j];
  s[i][j]=i;
  for(int k=i+1; k<j; k++){
    int t=RecurMatrixChain(i,k)
        +RecurMatrixChain(k+1,j)
        +p[i-1]*p[k]*p[j];
    if (t<u) {
      u=t;
      s[i][j]=k;}
    }
  return u;
}
```

如果用 T(n)表示该算法的计算 A[1..n]的时间，则有如下递归关系式：

$$T(n) \ge \begin{cases} O(1) & n=1 \\ 1+\sum_{k=1}^{n-1}(T(k)+T(n-k)+1) & n>1 \end{cases}$$

$$T(n) \ge 1+(n-1)+\sum_{k=1}^{n-1}T(k)+\sum_{k=1}^{n-1}T(n-k)=n+2\sum_{k=1}^{n-1}T(k),$$

可用数学归纳法直接证明：$T(n) \ge 2^{n-1}=\Omega(2^n)$，这显然不是我们所期望的。

注意到，在用递归算法自上向下求解具有最优子结构计算模型时，每次产生的子问题并不总是新问题，有些问题被反复计算多次。如果对每一个问题只解一次，而后将其解保存在一个表格中，当再次需要解此问题时，只是简单地用常数时间查看一下结果，则可以节省大量的时间。在矩阵的连乘积问题中，若用m[i][j]表示由第 i 个矩阵到第 j 个矩阵的连乘积 A[i..j]所用的最少数乘次数，则计算 m[1][n]时共有$\Theta(n^2)$个子问题。这是因为，对于$1 \le i < j \le n$，不同的有序对(i, j)对应于不同的子问题，不同子问题有$\binom{n}{2}+n=\Theta(n^2)$个。下面将会看到，用动态规划方法解此问题时，可在多项式时间内找到矩阵连乘积的最优加括号方法。

程序 5-1-2 求矩阵连乘最优次序的动态规划算法

```
void MatrixChain(int p, int n, int * *m, int * *s)
{
  for (int i=1; i<=n; i++) m[i][i]=0;
  for (int r=2; r<=n; r++){
    for (int i=1; i<=n-r+1; i++){
      int j=i+r-1; \\ r 是跨度
      m[i][j]= m[i+1][j]+p[i-1]*p[i]*p[j];
      s[i][j]=i;
      for (int k=i+1; k<j; k++){
        int t= m[i][k]+ m[k+1][j]+p[i-1]*p[k]*p[j];
        if (t< m[i][j]) {
           m[i][j]=t;
           s[i][j]=k; }
        }
      }
```

```
        }
    }
}
```

算法 MatrixChain 的主要计算量取决于程序中对 r，i 和 k 的三重循环，循环体内的计算量为 O(1)，三重循环的总次数是 $O(n^3)$，所以，算法的计算时间复杂性为 $O(n^3)$。

例 5.1.4 求以下 6 个矩阵连乘积最少数乘运算次数及所采用乘法次序。

$$A_1:30\times35;\ A_2:35\times15;\ A_3:15\times5;\ A_4:5\times10;\ A_5:10\times20;\ A_6:20\times25\quad,$$

即

P=[30, 35, 15, 5, 10, 20, 25]

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000\\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125\\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375\end{cases}$$
$$=7125$$

一般的计算 $m[i][j]$ 以及 $s[i][j]$ 的过程如下图所示：

$m[i][j]$

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	5125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

跨度增加方向

$s[i][j]$

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

注意，上述算法只是明确给出了矩阵最优连乘次序所用的数乘次数 m[1][n]，并未明确给出最优连乘次序，即完全加括号方法。但是以 s[i][j]为元素的 2 维数组却给出了足够的信息。事实上，s[i][j]=k 说明，计算连乘积 A[i..j]的最佳方式应该在矩阵 A_k 和 A_{k+1} 之间断开，即最优加括号方式为(A[i..k])(A[k+1..j])。下面的算法 Traceback 按算法 MatrixChain 计算出的断点信息 s 指示的加括号方式输出计算 A[i..j]的最优次序。

程序 5-1-3　根据最优值算法构造最优解

```
Void Traceback(int i, int j, int * * s)
{
   if (i==j) return;
   Traceback(i, s[i][j], s);
   Traceback(s[i][j]+1, j, s);
   cout << "Multiply A" << i << ","  << s[i][j];
   cout << "and A" << (s[i][j] +1) << ","  << j << endl;
}
```

总结上面解矩阵连乘积问题，我们可以归纳出使用动态规划算法的基本步骤：

1.　分析最优解的结构

在这一步中，选定要解决的问题的一个模型，其具有最小子结构性质，这是选择动态规划算法的基础。

2.　建立递归关系

第一步的结构分析已经为建立递归关系奠定了基础。这一步的主要任务是递归地定义最优值，并建立该问题与其子问题最优值之间的递归关系。例如在矩阵连乘积问题中，递归关系为

$$m[i][j]=\begin{cases} 0 & i=j \\ \min\limits_{i\le k\le j}\{\ \mathrm{m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j]}\} & i<j \end{cases}$$

在 0/1 背包问题中，令 $g_j(X)$ 表示 0/1 背包问题：物品编号是 j, j+1, …, n，背包容量是 X 的最优值，则最优值递归关系是

$$g_j(X)=\max\left\{g_{j+1}(X), g_{j+1}(X-w_j)+p_j\right\} \qquad (5.1.8)$$

在多段图问题中的递归关系是

$$COST(i,j)=\min_{l\in V_{i+1},(j,l)\in E}\left\{c(j,l)+COST(i+1,l)\right\} \qquad (5.1.9)$$

这里 j 表示取 V_i 中的顶点 j。

3.　计算最优值

依据递归关系式可以自底向上的方式（或自顶向下的方式－备忘录方法）进行计算，在计算过程中保存已经解决的子问题答案。每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算。

4. 构造最优解

依据求最优值时记录的信息，构造出最优解（如矩阵连乘积）。

上述归纳的 4 个步骤是一个整体，必要时才分步完成，多数情况下是统一来完成的。

§2. 多段图问题

多段图是一种简单而经典的使用动态规划算法的模型，它既能有效地反映动态规划算法基本特征，而且在实际问题中有较多的应用。

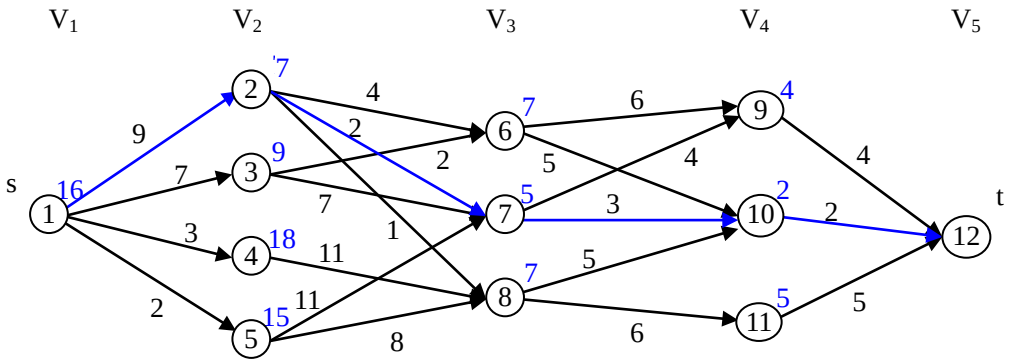


图 5-2-1 多段图的动态规划算法执行过程

最优值递推关系式为

$$COST(i,j)=\min_{l\in V_{i+1},(j,l)\in E}\{c(j,l)+COST(i+1,l)\} \qquad (5.2.1)$$

其中，$COST(i,j)$ 代表 i 段中的顶点 j 到汇点 t 的最短路径的长度。根据(5.2.1)式，我们可以由后向前逐步确定各阶段中的顶点到汇顶点 t 的最短路径。对于如上的 5 阶段网络图，蓝色字体标出了各顶点到汇顶点 t 的最短距离。

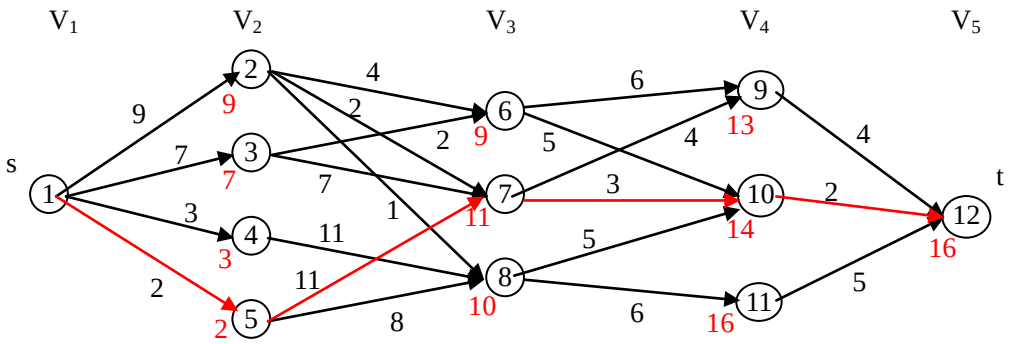


图 5-2-2 由前向后的处理方法（备忘录方法）

事实上，我们也可以由前向后逐步确定由源顶点 s 到各阶段中顶点的最短路径，此时的递归关系为

$$BCOST(i,j)=\min_{l\in V_{i-1},(l,j)\in E}\{BCOST(i-1,l)+c(l,j)\} \qquad (5.2.2)$$

其中，$BCOST(i,j)$代表源节点s到i段中的顶点j的最短路径的长度。上图中的红色字体标出了由源节点 s 到各顶点的最短距离。

程序 5-2-2　多段图的动态规划算法伪代码

```
    proc MultiGraph(E, k, n, P) //有 n 个顶点的 k 段图 G（按段序统
        // 一编号)，E 是边集，c(i, j)是边(i, j)的成本，P[1..k]是最小
        // 成本路径。
1       float COST[1..n]; integer D[1..n-1]，P[1..k], r, j, n;
2       COST[n]:=0;
3       for j from n-1 by -1 to 1 do
4         设 r 是这样一个顶点，(j, r)∈E 且使得 c(j, r)+COST[r]取最小
        值
5         COST[j]:= c(j, r)+COST[r];
6         D[j]:=r;  //指出 j 的后继
7       end{for}
8       P[1]:=1; P[k]:=n; //最短路径的起点为 s，终点为 t
9       for j from 2 to k-1 do
10        P[j]:=D[P[j-1]];  //最短路径上的第 j 个节点是第 j-1 节点的后
        继
11      end{for}
12  end{MultiGraph}
```

这里，D[j]将由 j 到汇顶点 t 的最短路径上 j 后面的顶点记录下来，P[j]则记录由源节点 s 到汇节点 t 的最短路径上处于第 j 阶段中的顶点。语句 10 是根据数组 D 中的信息逐段寻找到由源顶点 s 到汇顶点 t 的最短路径上的各个顶点。如果用邻接链表表示图 G，则语句 4 中 r 的确定可以在与 $d^+(j)$成正比的时间内完成。因此，语句 3－7 的 for 循环所需的时间是$\Theta(n+|E|)$。循环体 9－11 所需时间是$\Theta(k)\le n$，因而算法 MultiGraph 的时间复杂度是$\Theta(n+|E|)$。下面给出的备忘录算法的时间复杂度也是如此。

程序 5-2-3　多段图的备忘录算法伪代码

```
MemorizedMultiGraph(E, k, n, P) //有 n 个顶点的 k 段图 G（按段序统一
```

```
    //编号)，E 是边集，c(i, j)是边(i, j)的成本，P[1..k]是最小成本
    //路径。
    real BCOST[1..n]; integer D[1..n-1], P[1..k], r, j, n;
    BCOST[1]:=0;
    for j from 2 to n do
      设 r 是这样一个顶点，(r, j)∈E 且使得 BCOST(r)+ c(r, j)取最
      小值
      BCOST[j]:= BCOST(r)+ c(r, j);
      [j]:=r;   // 记录节点 j 的前继
    end{for}
    P[1]:=1; P[k]:=n;  // 最短路径的起点为节点 s，终点为节点 t
    for j from k-1 by -1 to 2 do
       P[j]:=D[P[j+1]];  // 最短路径上第 j 个节点是第 j+1 个节点
                              的前继
    end{for}
end{MemorizedMultiGraph}
```

§3. 0/1 背包问题

本节将使用动态规划的由前向后处理方法（也称备忘录算法）处理 0/1 背包问题：

$$\max \sum_{1\le i\le n} p_i x_i$$

$$\text{s.t.} \sum_{1\le i\le n} w_i x_i \le c, \quad x_i \in \{0,1\}, i=1,2,\cdots,n \tag{5.3.1}$$

通过作出变量 x 的取值序列 $x_1, x_2, \cdots, x_n$ 来给出最优解。这个取值序列的对应决策序列是 $\underline{x_n, x_{n-1}, \cdots, x_1}$ 。在对 x_n 作出决策之后，问题处于下列两种状态之一：背包剩余的容量仍为 c，此时未产生任何效益；背包的剩余容量为 $c-w_n$，此时的效益值增长了 p_n。因而

$$m[n][c] = \max\{m[n-1][c], m[n-1][c-w_n]+p_n\} \tag{5.3.2}$$

一般地，如果记 0/1 背包子问题：

$$\max \sum_{1\le i\le k} p_i x_i$$

$$\text{s.t.} \sum_{1\le i\le k} w_i x_i \le X, \quad x_i \in \{0,1\}, i=1,2,\cdots,k \tag{5.3.3}$$

最优解的值为$m[k][X]$，则有

$$m[k][X] = \max\{m[k-1][X], m[k-1][X-w_k]+p_k\} \tag{5.3.4}$$

这里$0 \le X \le c$，而且$k=1,2,\cdots,n$。为使(5.3.4)式能够有效地递推下去，需要延拓 k 和 X 的取值范围：$k=0,1,2,\cdots n; X \in (-\infty,+\infty)$；补充定义：

$$m[0][X] = \begin{cases} -\infty & if \quad X<0; \\ 0 & if \quad X \ge 0; \end{cases} \quad m[k][<0] = -\infty, k=1,2,\cdots n \tag{5.3.5}$$

事实上，我们的主要目的是计算出$m[n][c]$，根据(5.3.4)式，我们可能需要计算$m[n-1][c-w_n]$，而为了计算$m[n-1][c-w_n]$，可能需要计算$m[n-2][c-w_{n-1}-w_n]$，如此等等，在递推公式(5.3.4)中用到的 X 值可能为负数。另外，如果将$m[k][X]$看作 X 的函数，则是一个分段函数，而且当$X \ge \sum_{1\le i\le k} w_i$时取常值。所以将 X 的取值范围延拓至全体实数。因此，0/1 背包问题的求解过程就转化为递归地求解诸函数$m[0][X], m[1][X], \cdots, m[n][X]$的过程，最优值就是$m[n][c]$。

例 5.3.1 $n=3, (w_1,w_2,w_3)=(2,3,4), (p_1,p_2,p_3)=(1,2,5), c=6$。

$$m[0][X] = \begin{cases} -\infty & X<0 \\ 0 & X \ge 0 \end{cases},$$

$$m[1][X] = \begin{cases} -\infty & X<0 \\ 0 & 0 \le X < 2 \\ \max\{0,0+1\}=1 & X \ge 2 \end{cases} \quad m[0][X-w_1]+p_1$$

$$m[2][X] = \begin{cases} -\infty & X<0 \\ 0 & 0 \le X < 2 \\ 1 & 2 \le X < 3 \\ \max\{1,0+2\}=2 & 3 \le X < 5 \\ \max\{1,1+2\}=3 & X \ge 5 \end{cases} \quad m[1][X-w_2]+p_2$$

$$m[2][X]=\begin{cases}-\infty & X<0\\ 0 & 0\le X<2\\ 1 & 2\le X<3\\ 2 & 3\le X<4\\ \max\{2,0+5\}=5 & 4\le X<5\\ \max\{3,0+5\}=5 & 5\le X<6\\ \max\{3,1+5\}=6 & 6\le X<7\\ \max\{3,2+5\}=7 & 7\le X<9\\ \max\{2,3+5\}=8 & X\ge 9\end{cases}\qquad m[2][X-w_3]+p_3$$

由递推式(5.3.5)，$m[k-1][X]\le m[k][X]$，而且当 $X<w_k$ 时等式成立，这一事实可以写成

$$m[k][X]=\begin{cases}m[k-1][X], & X<w_k\\ \max\{m[k-1][X],m[k-1][X-w_k]+p_k\}, & X\ge w_k\end{cases}$$
$$k=1,2,\cdots,n$$

上述诸函数 $m[k][X]$ 具有如下性质：

1. 每个阶梯函数 $m[k][X]$ 都是由其跳跃点偶 $(b,m[k][b])$ 决定的。如果有 $r+1$ 个跳跃点：$b_0<b_1<\cdots<b_r$，各点的函数值分别是 $v_0,v_1,\cdots,v_r$，则

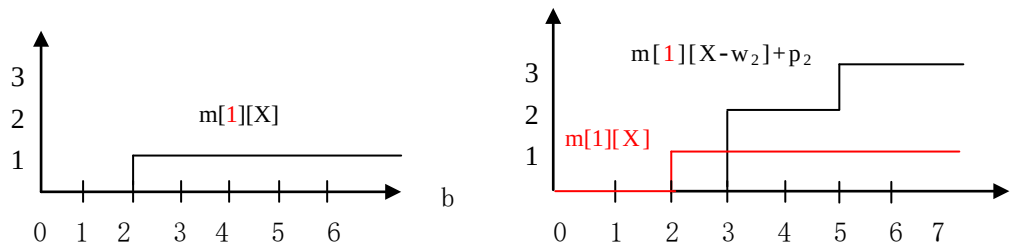
$$m[k][X]=v_i \quad if \quad b_i\le X<b_{i+1},\quad i=0,1,\cdots,r \tag{5.3.6}$$

这里约定 $b_{r+1}=+\infty$。

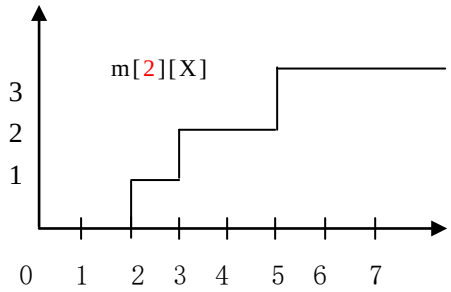
2. 令 S^k 是阶梯函数 $m[k][X]$ 的跳跃点偶的集合，则阶梯函数 $m[k-1][X-w_k]+p_k$ 的跳跃点偶之集去掉点偶(0,0)后，恰是集合

$$S_a^i=\{(b,v)\,|\,(b-w_k,v-p_k)\in S^{k-1}\} \tag{5.3.7}$$

这是因为函数 $m[k-1][X-w_k]+p_k$ 的图象恰是由函数 $m[k-1][X]$ 的图象先向右平移 w_k，然后再向上移动 p_k 而得。下图给出了前面例子的阶梯函数图像示意：



$S^1=\{(0,0),\ (2,1)\}$　　$(w_2,\ p_2)=(3,\ 2)$　　$S^2_a=\{\ (3,2),\ (5,3)\}$



$S^2=\{(0,0),\ (2,1),\ (3,2),\ (5,3)\}$

根据递推公式(5.3.4)，S^k是从$S^{k-1}\cup S^k_a$中去掉那些点偶(b_i,v_i)：在$S^{k-1}\cup S^k_a$中存在点偶(b_k,v_k)使得$b_i\ge b_k$而且$v_i\le v_k$，此时我们说(b_k,v_k)淹没(b_i,v_i)。前面例子的诸S^k计算如下：

$$S^0=\{(0,0)\};\quad (w_1,p_1)=(2,1), S_a^1=(w_1,p_1)+S^0=\{(2,1)\}$$
$$S^1=\{(0,0),(2,1)\};\quad (w_2,p_2)=(3,2), S_a^2=(w_2,p_2)+S^1=\{(3,2),(5,3)\}$$
$$S^2=\{(0,0),(2,1),(3,2),(5,3)\};\quad (w_3,p_3)=(4,5),$$
$$S_a^3=(w_3,p_3)+S^2=\{(4,5),(6,6),(7,7),(9,8)\}$$
$$S^3=\{(0,0),(2,1),(3,2),(4,5),(6,6),(7,7),(9,8)\}.$$

在由S^2、S_a^3向S^3的归并过程中，点偶(5, 3)被(4, 5)淹没。

3．在处理实际问题时，由于要求$X\le c$，在计算$S_a^k=(w_k,p_k)+S^{k-1}$时应该去掉那些使得$b>c$的跳跃点偶(b,v)。根据前面提到的淹没规则，如果将S^k中的元素按照第一个分量的递增次序排列，则第二个分量也呈递增的次序，而且S^n的最后一个元素，设为(b,v)，的v值即是 0/1 背包问题(5.3.1)的最优值$m[n][c]$。

4．最优解的获得可以通过检查诸S^k来确定。设(b_{k_n},v_{k_n})是S^n的最后一个元素。若$(b_{k_n},v_{k_n})\in S^{n-1}$，则$x_n=0$．因为此时函数$m[n][X]$和函数$m[n-1][X]$在 X≤c 范围内的最大值一致，表明 0/1 背包问题(5.3.1)与其子问题

$$\max\sum_{1\le i\le n-1}p_ix_i$$
$$\text{s.t.}\ \sum_{1\le i\le n-1}w_ix_i\le c,\quad x_i\in\{0,1\}, i=1,2,\cdots,n-1 \qquad (5.3.8)$$

有相同的最优值。若$(b_{k_n},v_{k_n})\notin S^{n-1}$，则$x_n=1$。因为，此时函数$m[n][X]$在 X≤c

范围内的最大值是函数$m[n-1][X]$的最大值加p_n，相应地，0/1 背包问题(5.3.1)的最优值是其子问题

$$\max \sum_{1\le i\le n-1} p_i x_i$$
$$\text{s.t.} \sum_{1\le i\le n-1} w_i x_i \le c - w_n, \quad x_i \in \{0,1\}, i = 1,2,\cdots,n-1 \qquad (5.3.9)$$

的最优值加p_n。注意到此时跳跃点偶一定具有形式

$$(b_{k_n}, v_{k_n}) = (w_n, p_n) + (b_{k_{n-1}}, v_{k_{n-1}}) \qquad (5.3.10)$$

其中$(b_{k_{n-1}}, v_{k_{n-1}}) \in S^{n-1}$。于是，我们可以依$(b_{k_{n-1}}, v_{k_{n-1}}) \in S^{n-2}$与否而决定$x_{n-1}$取 0 或 1。如此等等，逐一确定$x_n, x_{n-1}, \cdots, x_1$的值。

在上述例子中，整理后的诸S^k为：

$$S^0 = \{(0,0)\};$$
$$S^1 = \{(0,0),(2,1)\};$$
$$S^2 = \{(0,0),(2,1),(3,2),(5,3)\};$$
$$S^3 = \{(0,0),(2,1),(3,2),(4,5),(6,6)\}.$$

(6, 6)是S^3的最后一个跳跃点偶，所以该 0/1 问题的最优值是 6。这个点偶不在S^2中，因而$x_3 = 1$；又 $(6,6)-(4,5)=(2,1)\in S^2$，据此判断x_2的取值。因为$(2,1)\in S^1$，所以$x_2 = 0$；最后由$(2,1)\notin S^0$知$x_1 = 1$，所以最优解为$(1,0,1)$.

为实现上述算法，可以用两个一维数组 B 和 V 来存放所有的跳跃点偶(b, v)，跳跃点偶集$S^0, S^1, \cdots, S^{n-1}$互相邻接地存放（将诸$S^i$中的全部元素统一编号，从$S^i$生成$S^{i+1}$时，元素也是递升地产生，而且使用淹没规则），并用指针 F(k)来指示集合 S^k，即 S^k的第一个元素的位置，而 F(n)-1 是S^{n-1}中最后一个元素的位置（这里不存放 S^n是由于我们只需要它的最后一个元素，而这个元素或者是 S^{n-1}的最后一个元素，此时 S^{n-1}与 S^n有相同的最后元素；或者具有形式(5.3.10)，而且v_{k_n}是满足$b_{k_{n-1}} + w_n \le c$的最大值。所以，确定 S^n的最后元素不必将 S^n的元素全部求出来。而且确定最优解的其它变量$x_{n-1}, \cdots, x_1$时也不使用数据 S^n）。因为产生 S^k仅使用信息 S^{k-1}和(w_k, p_k)，所以不必存储S_a^k。根据以上分析，我们不难给出求解 0/1

背包问题模型的动态规划算法。

程序 5-3-1　0/1 背包问题动态规划算法伪代码

```
DKnapsack(w, p, c, n, m) //数组 w, p 中的元素分别代表各件物品的
 //重量和价值，n 是物品个数，c 代表背包容量，m 代表跳跃点个数。
  float p[1..n], w[1..n], B[1..m], V[1..m], ww, pp, c;
  integer F[0..n], l, h, i, j, next;
  F[0]:=1; B[1]:=0; V[1]:=0;
  s:=1; t:=1; //S^0 的首和尾
  F[1]:=2; next:=2; // B、V 中的第一个空位
  for i to n-1 do
     k:=s; //k 指示扫描 S^(i-1) 的游标，它的起始位置为 s，终止位置为 t
     u:=最大指标 r，使得 s≤ r ≤ t, 而且 B[r]+w_i ≤ c;
     for j from s to u do
       (ww, pp):=( B[j]+w_i, V[j]+p_i); //S^i_a 中的下一个元素
        while k ≤ t && B[k] < ww do //从 S^(i-1) 中取来元素归并
           B[next]:=B[k]; V[next]:=V[k];
           next:=next+1; k:=k+1;
        end{while}
        if k ≤ t && B[k]=ww then
          pp:=max(V[k], pp);
          k:=k+1;
        end{if}
        if pp > V[next-1] then
           (B[next], V[next]):=(ww, pp);
           next:=next+1;
        end{if}
         while k ≤ t && V[k] < V[next-1] do //清除
           k:=k+1;
         end{while}
      end{for}
      while k ≤ t do // 将 S^(i-1) 剩余的元素并入 S^i
        (B[next], V[next]):= (B[k], V[k]);
        next:=next+1; k:=k+1;
      end{while}
```

```
    s:=t+1; t:=next-1; F[i+1]:=next; //为S^(i+1)赋初值
  end{for}
  traceparts // 逐一确定x_n,x_(n-1),…,x_1
end{DKnapsack}
```

算法 DKnapsack 的主要工作是产生诸 S^i。在 i > 0 的情况下，每个 S^i 由 S^{i-1} 和 S^i_a 归并而成，并且 $|S^i_a| \le |S^{i-1}|$。因此 $|S^i| \le 2|S^{i-1}|$，在最坏情况下，没有序偶会被清除，所以

$$\sum_{1\le i\le n-1} |S^i| = \sum_{1\le i\le n} 2^{i-1} = 2^n - 1$$

由此知，算法 DKnapsack 的空间复杂度是 $O(2^n)$。由 S^{i-1} 生成 S^i 需要 $\Theta(|S^{i-1}|)$ 的时间，因此，计算 $S^0, S^1, \cdots, S^{n-1}$ 总共需要的时间为

$$\sum_{1\le i\le n-1} |S^{i-1}| \le \sum_{1\le i\le n-1} 2^{i-1} = 2^{n-1} - 1$$

算法 DKnapsack 的时间复杂度为 $O(2^n)$。

如果物品的重量 w_i 和所产生的效益值 p_i 都是整数，那么，S^i 中的元素 (b,v) 的分量 b 和 v 也都是整数，且 $b \le c, v \le \sum_{1\le k\le i} p_k$。又 S^i 中不同的元素对应的分量也都是不同的，故

$$|S^i| \le 1 + \sum_{1\le k\le i} p_k$$

$$|S^i| \le 1 + \min\left\{\sum_{1\le k\le i} w_k, c\right\}$$

此时，算法 DKnapsack 的时间复杂度为 $O(\min\left\{2^n, nc, n\sum_{i=1}^{n} p_k\right\})$。

附：从组合的角度来理解 0/1 背包问题的动态规划算法

先假定背包的容量是充分大的，考虑有 k 件物品往背包里装时，背包中物品总重量的各种可能出现的情况：

$k=0$： $W_0 = \{0\}$

$k=1$： $W_1 = \{0, w_1\}$

$k=2$： $W_2 = \{0, w_1, w_2, w_1 + w_2\}$

$k=3$： $W_3 = \{0, w_1, w_2, w_1 + w_2, w_3, w_1 + w_3, w_2 + w_3, w_1 + w_2 + w_3\}$

一般地，

$$W_k = \left\{ \sum_{i=1}^{k} x_i w_i \mid x_i \in \{0,1\} \right\} \qquad (5.3.11)$$

$$W_{k+1} = W_k \cup \left(w_{k+1} + W_k \right), \qquad k = 1,2,\cdots,n-1 \qquad (5.3.12)$$

递推关系式(5.3.10)正是我们计算诸S^k和S_a^k的依据，这只需将各种“重量”情况所对应的总价值带上，即产生元素对(b,v)，就能获得诸S^k和S_a^k。如上面的例子：$n=3,(w_1,w_2,w_3)=(2,3,4),(p_1,p_2,p_3)=(1,2,5),c=6$

$S^0=\{(0,0)\}$；　$(w_1,p_1)=(2,1)$,

$S_a^1=(w_1,p_1)+S^0=\{(2,1)\}$

$S^1=S^0\cup\left((w_1,p_1)+S^0\right)=\{(0,0),(2,1)\}$；　$(w_2,p_2)=(3,2)$,

$S_a^2=(w_2,p_2)+S^1=\{(3,2),(5,3)\}$

$S^2=S^1\cup\left((w_2,p_2)+S^1\right)=\{(0,0),(2,1),(3,2),(5,3)\}$；　$(w_3,p_3)=(4,5)$,

$S_a^3=(w_3,p_3)+S^2=\{(4,5),(6,6),(7,7),(9,8)\}$

$S^3=S^2\cup\left((w_3,p_3)+S^2\right)=\{(0,0),(2,1),(3,2),(4,5),(6,6),(7,7),(9,8)\}$　去掉被淹没的(5,3)

最后将S^3截断（根据约束条件），即去掉“重量”大于背包容量的点对，就得到所需要的点对集

$$S=\{(0,0),(2,1),(3,2),(4,5),(6,6)\}$$

得到最大价值是 6，因为$(6,6)\in S^3\setminus S^2$，应该是在前面的基础上增加了$(w_3,p_3)=(4,5)$而获得的，这个“基础”就是$(6,6)-(4,5)=(2,1)\in S^2$，所以，最优解中，$x_3=1$。再考虑$(2,1)\in S^2\cap S^1$的情况，在最优解中应有$x_2=0$。最后由$(2,1)\in S^1\setminus S^0$知，在最优解中$x_1=1$。所以，$(1,0,1)$就是上面背包问题的一个最优解。

§4. 流水作业调度问题

问题描述：已知 n 个作业{1，2，．．．，n}要在由两台机器 M_1 和 M_2 组成的流水线上完成加工。每个作业加工的顺序都是先在 M_1 上加工，然后在 M_2 上加工。M_1 和 M_2 加工作业 i 所需的时间分别为 a_i 和 b_i，$1\le i\le n$。流水作业调度问题要求确定这 n 个作业的最优加工次序，使得从第一个作业在机器 M_1 上开始加工，到最后一个作业在机器 M_2 上加工完成所需的时间最少。

记 N={1, 2, . . . , n}，S 为 N 的子集合。一般情况下，当机器 M_1 开始加工 S 中的作业时，机器 M_2 可能正在加工 S 以外的作业，要等待时间 t 后才可用来加工 S 中的作业。将这种情况下流水线完成 S 中的作业所需的最短时间记为 T(S, t)，则流水作业调度问题的最优值即是 T(N, 0)。

流水作业调度问题的上述计算模型具有最优子结构性质。事实上，设 π 是 n 个流水作业的一个最优调度（实际上是作业的一个排序），其所需要的加工时间为 $a_{\pi(1)}+T'$，其中，T' 是在机器 M_2 的等待时间为 $b_{\pi(1)}$ 时，调度 π 安排作业 $\pi(2),\cdots,\pi(n)$ 所需的加工时间。若记 $S=N\setminus\{\pi(1)\}$，我们证明 $T'=T(S,b_{\pi(1)})$。

首先由 T' 的定义知 $T'\geq T(S,b_{\pi(1)})$。如果 $T'>T(S,b_{\pi(1)})$，设 π' 是作业集 S 开始在机器 M_1，而机器 M_2 等待时间为 $b_{\pi(1)}$ 情况下的一个最优调度，则

$$\pi(1),\pi'(2),\cdots,\pi'(n)$$

是 N 的一个调度，该调度所需的时间为 $a_{\pi(1)}+T(S,b_{\pi(1)})<a_{\pi(1)}+T'$，这与 π 是 N 的最优调度矛盾。所以 $T'=T(S,b_{\pi(1)})$，说明流水作业调度问题的上述计算模型具有最优子结构性质。

可以如下建立流水作业调度问题的最优值递推关系式。容易看出

$$T(N,0)=\min_{1\leq i\leq n}\{a_i+T(N\setminus\{i\},b_i)\} \tag{5.4.1}$$

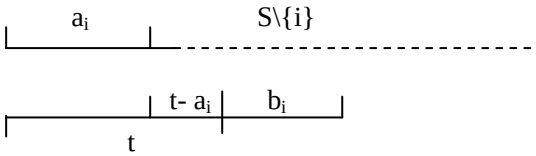
推广到一般情形：

$$T(S,t)=\min_{i\in S}\left\{a_i+T\left(S\setminus\{i\},b_i+\max\{t-a_i,0\}\right)\right\} \tag{5.4.2}$$

其中，$\max\{t-a_i,0\}$ 这一项是由于机器 M_2 必须在 $\max\{t,a_i\}$ 时间之后才能开始加工 S 中的作业。因此，在机器 M_1 完成作业 i 之后，机器 M_2 还需等待

$$b_i+\max\{t,a_i\}-a_i=b_i+\max\{t-a_i,0\}$$

时间后才能开始加工 S\{i}中的作业。



按照递推关系(5.4.2)，我们容易给出流水调度问题的动态规划算法，但是其时间复杂度将是 $O(2^n)$。以下我们将根据这一问题的特点，采用 Johnson 法则简化

算法，使得到时间复杂度为$O(n\log n)$。

设π是机器 M_1开始加工 S 中的作业而机器 M_2的等待时间为 t 时的任一最优调度。若在这个调度中，安排在最前面的两个作业分别是i和j，即$\pi(1)=i,\pi(2)=j$。则由递推关系式(5.4.2)得

$$T(S,t)=a_i+T\big(S\setminus\{i\},b_i+\max\{t-a_i,0\}\big)=a_i+a_j+T(S\setminus\{i,j\},t_{ij})$$

其中

$$\begin{aligned}t_{ij}&=b_j+\max\{b_i+\max\{t-a_i,0\}-a_j,0\}\\&=b_j+b_i+\max\{\max\{t-a_i,0\}-a_j,-b_i\}\\&=b_j+b_i+\max\{t-a_i-a_j,-a_j,-b_i\}\end{aligned}$$

注：上述推导过程中用到了关系式：

$$A+\max\{B_1,\cdots,B_l\}=\max\{A+B_1,\cdots,A+B_l\}$$

如果作业i和j满足

$$\min\{a_i,b_j\}\le\min\{a_j,b_i\}\tag{5.4.3}$$

则称作业i和j满足 Johnson 不等式；如果作业i和j不满足 Johnson 不等式，则交换作业i和j的加工次序后即满足 Johnson 不等式。在机器 M_1开始加工 S 中的作业而机器 M_2的等待时间为 t 时的调度π中，交换作业i和j的加工次序，得到作业集 S 的另一个调度π'，它所需的加工时间为

$$T'(S,t)=a_i+a_j+T(S\setminus\{i,j\},t_{ji})$$

其中，

$$t_{ji}=b_j+b_i+\max\{t-a_i-a_j,-a_i,-b_j\}$$

当作业i和j满足 Johnson 不等式(5.4.3)时，有

$$\max\{-a_i,-b_j\}\ge\max\{-a_j,-b_i\}$$

从而 $t_{ji}\ge t_{ij}$，可见$T'(S,t)\ge T(S,t)$。换句话说，当作业i和j不满足 Johnson 不等式时，交换它们的加工次序后，作业i和j将满足 Johnson 不等式，而且不会增加加工时间。

由此可知，对于流水作业调度问题，必然存在一个最优调度π，使得作业

$\pi(i)$和$\pi(i+1)$满足 Johnson 不等式：

$$\min\{a_{\pi(i)},b_{\pi(i+1)}\}\le\min\{a_{\pi(i+1)},b_{\pi(i)}\},\quad 1\le i\le n-1$$

一般地，可以证明，上述不等式与不等式

$$\min\{a_{\pi(i)},b_{\pi(j)}\}\le\min\{a_{\pi(j)},b_{\pi(i)}\},\quad 1\le i<j\le n \qquad (5.4.4)$$

等价。以上分析使我们不难理解关于流水作业调度问题的以下 Johnson 算法：

(1) 令 $AB=\{i\mid a_i<b_i\}$, $BA=\{i\mid a_i\ge b_i\}$；

(2) 将 AB 中作业依 a_i 的非减次序排列；将 BA 中作业依 b_i 的非增次序排列；

(3) AB 中作业接 BA 中作业即构成满足 Johnson 法则的最优调度。

程序 5-4-1 流水作业调度问题的 Johnson 算法

```
FlowShop(a,b,n,p)
// 给作业排序，数组 p 记录作业号的一个排列
float a[1..n],b[1..n];
integer c[1..n],d[1..n],p[1..n], n,j,k;
j:=1; k:=n;
for i to n do
  if a[i]<b[i] then
    d[j]:=i; c[j]:=a[i]; j:=j+1;
  else
    d[k]:=i; c[k]:=b[i]; k:=k-1;
  end{if}
end{for}
MergeSortL(c,1,k,q);
MergeSortL(c,k+1,n,r);
j:=q[0];
for i to k do
  p[i]:=d[j];
  j:=q[j];
end{for}
j:=r[0];
for i to n-k do
  p[n-i+1]:=d[k+j];
```

```
      j:=r[j];
    end{for}
 end{FlowShop}
```

上述算法的时间主要花在排序上，因此，在最坏情况下的时间复杂度为 $O(n\log n)$。空间复杂度为 $O(n)$ 更容易看出。

§5. 最优二叉搜索树

设 $S=\{x_1,x_2,\cdots,x_n\}$ 是一个有序集合，且 $x_1<x_2<\cdots<x_n$. 表示有序集合的二叉搜索树是利用二叉树的内节点存储有序集中的元素，而且具有性质：存储于每个节点中的元素大于其左子树中任一个节点中存储的元素，小于其右子树中任意节点中存储的元素。二叉树中的叶节点是形如 (x_i,x_{i+1}) 的开区间。在二叉搜索树中搜索一个元素 x，返回的结果或是在二叉树的内节点处找到：$x=x_i$；或是在二叉树的叶节点中：$x\in(x_i,x_{i+1})$，这里约定 $x_0=-\infty, x_{n+1}=+\infty$.

现在假定在第一种情况出现(即 $x=x_i$)的概率为 b_i；在第二种情况出现，即 $x\in(x_i,x_{i+1})$ 的概率为 a_i.显然

$$a_i\ge 0,\ 1\le i\le n, b_j\ge 0, 1\le i,j\le n;\ \sum_{i=0}^{n}a_i+\sum_{j=1}^{n}b_j=1 \qquad (5.5.1)$$

集合 $\{a_0,b_1,a_1,\cdots,b_n,a_n\}$ 称为有序集 S 的存取概率分布。

在一个表示 S 的二叉搜索树 T 中，设存储元素 x_i 的顶点深度为 c_i，叶顶点 (x_i,x_{i+1}) 的深度为 d_i，则

$$p=\sum_{i=1}^{n}b_i(1+c_i)+\sum_{j=0}^{n}a_jd_j \qquad (5.5.2)$$

表示在二叉搜索树 T 中做一次搜索所需的平均比较次数。p 也称为二叉搜索树 T 的平均路长。最优二叉搜索树问题是:

对于有序集 S 及其存取概率分布 $\{a_0,b_1,a_1,\cdots,b_n,a_n\}$*，在所有表示 S 的二叉搜索树中，找出一棵具有最小平均路长的二叉搜索树。*

为了采用动态规划算法，我们首先要考虑该问题的上述计算模型是否具有最优子结构性质。二叉搜索树 T 的一棵含有顶点 $x_i, x_{i+1}, \cdots, x_j$ 和叶顶点 $(x_{i-1}, x_i), (x_i, x_{i+1}), \cdots, (x_j, x_{j+1})$ 的子树 $\overline{T}$ 可以看作是有序集 $\{x_i, x_{i+1}, \cdots, x_j\}$ 关于全集为实数区间 (x_{i-1}, x_{j+1}) 的一棵二叉搜索树（T 自身可以看作是有序集 $S = \{x_1, x_2, \cdots, x_n\}$ 关于全集为整个实数区间 $(-\infty, +\infty)$ 的二叉搜索树）。

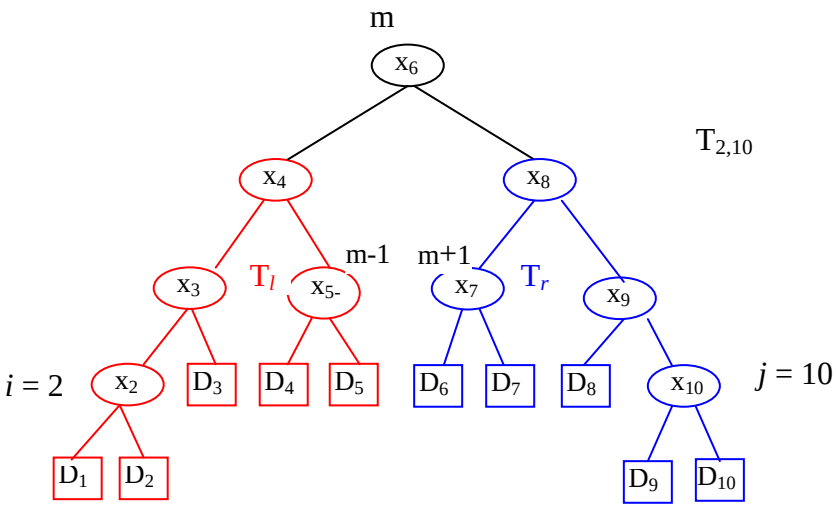


图 5-5-1 二叉搜索树的一棵子树

根据 S 的存取分布概率，x 在子树 $\overline{T}$ 的顶点处被搜索到的概率是

$$w_{ij} = \sum_{i-1 \le k \le j} a_k + \sum_{i \le k \le j} b_k \tag{5.5.3}$$

于是 $\{x_i, x_{i+1}, \cdots, x_j\}$ 的存储概率分布为 $\{\overline{a}_{i-1}, \overline{b}_i, \cdots, \overline{b}_j, \overline{a}_j\}$，其中，$\overline{a}_h, \overline{b}_k$ 分别是下面的条件概率：

$$\overline{b}_k = b_k / w_{ij}, \quad i \le k \le j; \qquad \overline{a}_h = a_h / w_{ij}, \quad i-1 \le h \le j \tag{5.5.4}$$

设 T_{ij} 是有序集 $\{x_i, x_{i+1}, \cdots, x_j\}$ 关于存储概率分布 $\{\overline{a}_{i-1}, \overline{b}_i, \cdots, \overline{b}_j, \overline{a}_j\}$ 的一棵最优二叉搜索树，其平均路长记为 p_{ij}．T_{ij} 的根顶点存储的元素是 x_m，其左子树 T_l 和右子树 T_r 的平均路长分别记为 p_l 和 p_r。由于 T_l 和 T_r 中顶点深度是它们在 T_{ij} 中的深度减 1，如下推导

$$w_{ij} = \sum_{i-1 \le k \le j} a_k + \sum_{i \le k \le j} b_k ,$$

$$p_{ij} = \sum_{i-1\le k\le j} \bar{d}_k\, a_k / w_{ij} + \sum_{i\le k\le j} (\bar{c}_k + 1)\, b_k / w_{ij}$$

$$w_{ij}\, p_{ij} = \sum_{i-1\le k\le j} \bar{d}_k a_k + \sum_{i\le k\le j} (\bar{c}_k + 1) b_k$$

$$w_{i,m-1}\, p_l = \sum_{i-1\le k\le m-1} (\bar{d}_k - 1) a_k + \sum_{i\le k\le m-1} \bar{c}_k b_k$$

$$w_{m+1,j}\, p_r = \sum_{m\le k\le j} (\bar{d}_k - 1) a_k + \sum_{m+1\le k\le j} \bar{c}_k b_k$$

$$w_{ij}\, p_{ij} = w_{ij} + w_{i,m-1} p_l + w_{m+1,j}\, p_r \ (\because \bar{c}_m = 0) \qquad (5.5.5)$$

由于T_l是有序集$\{x_i, \cdots, x_{m-1}\}$的一棵二叉搜索树，故$p_l \ge p_{i,m-1}$.若$p_l > p_{i,m-1}$，则用$T_{i,m-1}$替换T_l可得到平均路长比T_{ij}更小的二叉搜索树。这与T_{ij}是最优二叉搜索树矛盾。同理可证，T_r也是一棵最优二叉搜索树。因此，最优二叉搜索树问题具有最优子结构性质。将(5.5.5)中的p_l换成$p_{i,m-1}$，p_r换成$p_{m+1,j}$，则得到

$$w_{ij}\, p_{ij} = w_{ij} + w_{i,m-1} p_{i,m-1} + w_{m+1,j}\, p_{m+1,j} \qquad (5.5.6)$$

采用上面的记号，则 n 元最优二叉搜索树问题即是求 $T_{1,n}$，其最优值为 $p_{1,n}$。由最优二叉搜索树的最优子结构性质，可以建立最优值 $p_{i,j}$ 的递推公式：

$$w_{ij}\, p_{ij} = w_{ij} + \min_{i\le k\le j}\{w_{i,k-1} p_{i,k-1} + w_{k+1,j}\, p_{k+1,j}\}, \quad i \le j \qquad (5.5.7)$$

初始时，$p_{i,i-1} = 0, \quad 1 \le i \le n$. 记$w_{ij}\, p_{ij}$为$m(i, j)$，则$m(1, n) = w_{1,n}\, p_{1,n} = p_{1,n}$为所求最优值。由(5.5.7)得关于$m(i, j)$的递推公式

$$\begin{aligned} m(i, j) &= w_{ij} + \min_{i\le k\le j}\{m(i, k-1) + m(k+1, j)\}, \quad i \le j \\ m(i, i-1) &= 0, \quad i = 1, 2, \cdots, n \end{aligned} \qquad (5.5.8)$$

据此，可以设计求解最优二叉搜索树问题的动态规划算法。

程序 5-5-1　最优二叉搜索树的动态规划算法

```
void OBSTree( int a, int b, int n, int **m, int **s, int **w)
{
    for (int i = 0; i < n; i++) {
        w[i+1][i] = a[i];
        m[i+1][i] = 0;
```

```
        }
        for (int r = 0; r < n; r++) {
            for (int i = 1; i <= n-r; i++) {
                int j = i+r;
                w[i][j] = w[i][j-1] + a[j] +b[j];
                m[i][j] = m[i+1][j];
                s[i][j] = i;
                for (int k =i + 1; k<= j; k++) {
                    int t = m[i][k-1] + m[k+1][j];
                    if (t < m[i][j]) {
                        m[i][j] = t;
                        s[i][j] = k; }
                  }
                  m[i][j]+=w[i][j];
            }
    }
 }
```

算法 OBSTree 中用 s[i][j]保存最优子树的根顶点中元素。当 s[1][n] = k 时，x_k 为所求二叉搜索树的根顶点元素。其左子树为 T(1,k-1)，因此 i= s[1][k-1]即表示 T(1,k-1)的根顶点为 x_i。依次类推，容易由 s 记录的信息在 O(n)时间内构造出所求的最优二叉搜索树。算法中用到三个 2 维数组 m、s、w，故所需的空间为 $O(n^2)$. 算法的主要计算量在于计算

$$w_{ij} + \min_{i \le k \le j}\{m(i,k-1) + m(k+1,j)\}$$

对于固定的差 $r = j - i$，需要计算时间 $O(j-i+1) = O(r+1)$。因此算法所耗费的总时间为 $\sum_{r=0}^{n-1}\sum_{i=1}^{n-r} O(r+1) = O(n^3)$。

习题 五

1. 最大子段和问题：给定整数序列 $a_1, a_2, \cdots, a_n$，求该序列形如 $\sum_{k=i}^{j} a_k$ 的子段和的最大值： $\max\left\{0, \max_{1 \le i \le j \le n} \sum_{k=i}^{j} a_k\right\}$

1）已知一个简单算法如下：

```
int Maxsum(int n,int a,int& besti,int& bestj)
{
   int sum = 0;
   for(int i=1;i<=n;i++){
     int suma = 0;
    for(int j=i;j<=n;j++){
       suma + = a[j];
       if(suma > sum){
         sum = suma;
         besti = i;
         bestj = j;
         }
      }
    }
    return sum;
}
```

试分析该算法的时间复杂性。

2）试用分治算法解最大子段和问题，并分析算法的时间复杂性。

3）试说明最大子段和问题具有最优子结构性质，并设计一个动态规划算法解最大子段和问题。分析算法的时间复杂度。（提示：令 $b(j)=\max\limits_{1\le i\le j\le n}\sum\limits_{k=i}^{j}a_k,\ j=1,2,\cdots,n$）

2.（双机调度问题）用两台处理机 A 和 B 处理 n 个作业。设第 i 个作业交给机器 A 处理时所需要的时间是 a_i，若由机器 B 来处理，则所需要的时间是 b_i。现在要求每个作业只能由一台机器处理，每台机器都不能同时处理两个作业。设计一个动态规划算法，使得这两台机器处理完这 n 个作业的时间最短（从任何一台机器开工到最后一台机器停工的总的时间）。以下面的例子说明你的算法：

$n=6,(a_1,a_2,a_3,a_4,a_5,a_6)=(2,5,7,10,5,2),(b_1,b_2,b_3,b_4,b_5,b_6)=(3,8,4,11,3,4)$

3．考虑下面特殊的整数线性规划问题

$$\max\sum_{i=1}^{n}c_ix_i$$

$$\sum_{i=1}^{n}a_ix_i\le b,\quad x_i\in\{0,1,2\},1\le i\le n$$

试设计一个解此问题的动态规划算法，并分析算法的时间复杂度。

4. 可靠性设计：一个系统由 n 级设备串联而成，为了增强可靠性，每级都可能并联了不止一台同样的设备。假设第 i 级设备 D_i 用了 m_i 台，该级设备的可靠性是 $g_i(m_i)$，则这个系统的可靠性是$\Pi g_i(m_i)$。一般来说 $g_i(m_i)$都是递增函数，所以每级用的设备越多系统的可靠性越高。但是设备都是有成本的，假定设备 D_i 的成本是 c_i，设计该系统允许的投资不超过 c，那么，该如何设计该系统（即各级采用多少设备）使得这个系统的可靠性最高。试设计一个动态规划算法求解可靠性设计问题。