第 3 章 动态规划

动态规划是本书介绍的五种算法设计方法中难度最大的一种，它建立在最优原则的基础上。采用动态规划方法，可以优雅而高效地解决许多用贪婪算法或分而治之算法无法解决的问题。在介绍动态规划的原理之后，本章将分别考察动态规划方法在解决背包问题、图象压缩、矩阵乘法链、最短路径、无交叉子集和元件折叠等方面的应用。

3.1 算法思想

和贪婪算法一样，在动态规划中，可将一个问题的解决方案视为一系列决策的结果。不同的是，在贪婪算法中，每采用一次贪婪准则便做出一个不可撤回的决策，而在动态规划中，还要考察每个最优决策序列中是否包含一个最优子序列。

例**3-1 [**最短路经**]** 考察图1 2 - 2中的有向图。假设要寻找一条从源节点$s=1$到目的节点$d=5$的最短路径，即选择此路径所经过的各个节点。第一步可选择节点2，3或4。假设选择了节点3，则此时所要求解的问题变成：选择一条从3到5的最短路径。如果3到5的路径不是最短的，则从1开始经过3和5的路径也不会是最短的。例如，若选择的子路径（非最短路径）是3，2，5 (耗费为9)，则1到5的路径为1，3，2，5 (耗费为11)，这比选择最短子路径3，4，5而得到的1到5的路径1，3，4，5 (耗费为9) 耗费更大。

所以在最短路径问题中，假如在的第一次决策时到达了某个节点v，那么不管v 是怎样确定的，此后选择从v 到d 的路径时，都必须采用最优策略。

例**3-2 [0/1**背包问题**]** 考察1 3 . 4节的0 / 1背包问题。如前所述，在该问题中需要决定$x_1 .. x_n$的值。假设按$i=1，2，.，n$ 的次序来确定x_i 的值。如果置$x_1=0$，则问题转变为相对于其余物品（即物品2，3，.，n），背包容量仍为c 的背包问题。若置$x_1=1$，问题就变为关于最大背包容量为$c-w_1$ 的问题。现设$r?\{c，c-w_1\}$ 为剩余的背包容量。

在第一次决策之后，剩下的问题便是考虑背包容量为r 时的决策。不管x_1 是0或是1，$[x_2，.，x_n]$ 必须是第一次决策之后的一个最优方案，如果不是，则会有一个更好的方案$[y_2，.，y_n]$，因而$[x_1，y_2，.，y_n]$是一个更好的方案。

假设$n=3$, $w=[100,14,10]$, $p=[20,18,15]$, $c=116$。若设$x_1=1$，则在本次决策之后，可用的背包容量为$r=116-100=16$ 。$[x_2，x_3]=[0,1]$ 符合容量限制的条件，所得值为1 5，但因为$[x_2，x_3]=[1，0]$ 同样符合容量条件且所得值为1 8，因此$[x_2，x_3]=[0，1]$ 并非最优策略。即$x=[1，0，1]$ 可改进为$x=[1，1，0]$。若设$x_1=0$，则对于剩下的两种物品而言，容量限制条件为116。总之，如果子问题的结果$[x_2，x_3]$不是剩余情况下的一个最优解，则$[x_1，x_2，x_3]$也不会是总体的最优解。

例**3-3 [**航费**]** 某航线价格表为：从亚特兰大到纽约或芝加哥，或从洛杉矶到亚特兰大的费用为$100；从芝加哥到纽约票价$20；而对于路经亚特兰大的旅客，从亚特兰大到芝加哥的费用仅为$20。从洛杉矶到纽约的航线涉及到对中转机场的选择。如果问题状态的形式为（起点，终点），那么在选择从洛杉矶到亚特兰大后，问题的状态变为（亚特兰大，纽约）。从亚特兰大到纽约的最便宜航线是从亚特兰大直飞纽约，票价$100。而使用直飞方式时，从洛杉矶到纽约的花费为$200。不过，从洛杉矶到纽约的最便宜航线为洛杉矶-亚特兰大-芝加哥-纽约，其总花费为$140（在处理局部最优路径亚特兰大到纽约过程中选择了最低花费的路径：亚特兰大-芝加哥-纽约）。

如果用三维数组（tag，起点，终点）表示问题状态，其中tag为0表示转飞，tag为1表示其他情形，那么在到达亚特兰大后，状态的三维数组将变为（0，亚特兰大，纽约），它对应的最优路径是经由芝加哥的那条路径。

当最优决策序列中包含最优决策子序列时，可建立动态规划递归方程（dynamic-programming recurrence equation），它可以帮助我们高效地解决问题。

例**3-4 [0/1**背包**]** 在例3 - 2的0 / 1背包问题中，最优决策序列由最优决策子序列组成。假设$f(i,y)$ 表示例1 5 - 2中剩余容量为y，剩余物品为i，$i+1$，.，n 时的最优解的值，即：和利用最优序列由最优子序列构成的结论，可得到f 的递归式。$f(1,c)$ 是初始时背包问题的最优解。可使用（1 5 - 2）式通过递归或迭代来求解$f(1,c)$。从$f(n,*)$开始迭式，$f(n,*)$由（1 5 - 1）式得出，然后由（1 5 - 2）式递归计算$f(i,*)$ ($i=n-1$，$n-2$，.，2)，最后由（1 5 - 2）式得出$f(1,c)$。

对于例1 5 - 2，若$0\le y<10$，则$f(3,y)=0$；若$y\ge10$，$f(3,y)=15$。利用递归式（1 5 - 2），可得$f(2,y)=0$ ($0\le y<10$)；f（2，y）$=15$（$10\le y<14$）；f（2，y）$=18$（$14\le y<24$）和f（2，y）$=33$（$y\ge24$）。因此最优解$f(1,116)=max\{f$（2，116），f（2，$116-w_1$）$+p_1\}=max\{f$（2，116），f（2，16）$+20\}=max\{33,38\}=38$。

现在计算x_i值，步骤如下：若$f(1,c)=f(2,c)$，则$x_1=0$，否则$x_1=1$。接下来需从剩余容量$c-w_1$中寻求最优解，用$f(2,c-w_1)$ 表示最优解。依此类推，可得到所有的x_i ($i=1.n$) 值。

在该例中，可得出$f(2,116)=33\neq f(1,116)$，所以$x_1=1$。接着利用返回值$38-p_1=18$ 计算x_2 及x_3，此时$r=116-w_1=16$，又由$f(2,16)=18$，得$f(3,16)=14\neq f(2,16)$，因此$x_2=1$，此时$r=16-w_2=2$，所以$f(3,2)=0$，即得$x_3=0$。

动态规划方法采用最优原则（principle of optimality）来建立用于计算最优解的递归式。所谓最优原则即不管前面的策略如何，此后的决策必须是基于当前状态（由上一次决策产生）的最优决策。由于对于有些问题的某些递归式来说并不一定能保证最优原则，因此在求解问题时有必要对它进行验证。若不能保持最优原则，则不可应用动态规划方法。在得到最优解的递归式之后，需要执行回溯（traceback）以构造最优解。

编写一个简单的递归程序来求解动态规划递归方程是一件很诱人的事。然而，正如我们将在下文看到的，如果不努力地去避免重复计算，递归程序的复杂性将非常可观。如果在递归程序设计中解决了重复计算问题时，复杂性将急剧下降。动态规划递归方程也可用迭代方式来求解，这时很自然地避免了重复计算。尽管迭代程序与避免重复计算的递归程序有相同的复杂性，但迭代程序不需要附加的递归栈空间，因此将比避免重复计算的递归程序更快。

3.2 应用

3.2.1 0/1背包问题

1. 递归策略

在例3 - 4中已建立了背包问题的动态规划递归方程，求解递归式（15 - 2）的一个很自然的方法便是使用程序15 - 1中的递归算法。该模块假设p、w 和n 为输入，且p 为整型，F(1,c) 返回$f(1,c)$ 值。

程序15-1 背包问题的递归函数

```
int F(int i, int y)
{// 返回f ( i , y ) .
if (i == n) return (y < w[n]) ? 0 : p[n];
if (y < w[i]) return F(i+1,y);
return max(F(i+1,y), F(i+1,y-w[i]) + p[i]);
}
```

程序15 - 1的时间复杂性$t(n)$满足：$t(1)=a$；$t(n)\le 2t(n-1)+b$（$n>1$），其中a、b 为常数。通过求解可得$t(n)=O(2^n)$。

例**3-5** 设$n=5$，$p=[6,3,5,4,6]$，$w=[2,2,6,5,4]$ 且$c=10$，求$f(1,10)$。为了确定$f(1,10)$，调用函数F (1,10)。递归调用的关系如图15 - 1的树型结构所示。每个节点用y值来标记。对于第j层的节点有$i=j$，因此根节点表示F (1,10)，而它有左孩子和右孩子，分别对应F (2,10)和F (2,8)。总共执行了28次递归调用。但我们注意到，其中可能含有重复前面工作的节点，如$f(3,8)$计算过两次，相同情况的还有$f(4,8)$、$f(4,6)$、$f(4,2)$、$f(5,8)$、$f(5,6)$、$f(5,3)$、$f(5,2)$ 和$f(5,1)$。如果保留以前的计算结果，则可将节点数减至19，因为可以丢弃图中的阴影节点。

正如在例3 - 5中所看到的，程序15 - 1做了一些不必要的工作。为了避免$f(i,y)$的重复计算，必须定义一个用于保留已被计算出的$f(i,y)$值的表格L，该表格的元素是三元组$(i,y,f(i,y))$。在计算每一个$f(i,y)$之前，应检查表L中是否已包含一个三元组$(i,y,*)$，其中*表示任意值。如果已包含，则从该表中取出$f(i,y)$的值，否则，对$f(i,y)$进行计算并将计算所得的三元组$(i,y,f(i,y))$加入表L。L既可以用散列（见7 . 4节）的形式存储，也可用二叉搜索树(见11章)的形式存储。

2. 权为整数的迭代方法

当权为整数时，可设计一个相当简单的算法（见程序15 - 2）来求解$f(1,c)$。该算法基于例3 - 4所给出的策略，因此每个$f(i,y)$ 只计算一次。程序15 - 2用二维数组f [ ][ ]来保存各f 的值。而回溯函数Traceback用于确定由程序15 - 2所产生的x_i 值。函数Knapsack的复杂性为(nc），而Traceback的复杂性为(n)。

程序15-2 f 和x 的迭代计算

```
template<class T>
void Knapsack(T p[], int w[], int c, int n, T** f)
{// 对于所有i和y计算f [ i ] [ y ]
// 初始化f [ n ] [ ]
for (int y = 0; y <= yMax; y++)
f[n][y] = 0;
for (int y = w[n]; y <= c; y++)
f[n][y] = p[n];
// 计算剩下的f
for (int i = n - 1; i > 1; i--) {
for (int y = 0; y <= yMax; y++)
f[i][y] = f[i+1][y];
for (int y = w[i]; y <= c; y++)
f[i][y] = max(f[i+1][y], f[i+1][y-w[i]] + p[i]);
}
f[1][c] = f[2][c];
if (c >= w[1])
f[1][c] = max(f[1][c], f[2][c-w[1]] + p[1]);
}
template<class T>
void Traceback(T **f, int w[], int c, int n, int x[])
{// 计算x
for (int i = 1; i < n; i++)
if (f[i][c] == f[i+1][c]) x[i] = 0;
else {x[i] = 1;
c -= w[i];}
x[n] = (f[n][c]) ? 1 : 0;
}
```

3. 元组方法（选读）

程序15 - 2有两个缺点：1) 要求权为整数；2) 当背包容量c 很大时，程序15 - 2的速度慢于程序15 - 1。一般情况下，若$c>2^n$，程序15 - 2的复杂性为$\Omega(n2^n)$。可利用元组的方法来克服上述两个缺点。在元组方法中，对于每个i，$f(i,y)$ 都以数对$(y,f(i,y))$ 的形式按y的递增次序存储于表$P(i)$中。同时，由于$f(i,y)$ 是y 的非递减函数，因此$P(i)$ 中各数对$(y,f(i,y))$ 也是按$f(i,y)$ 的递增次序排列的。

例**3-6** 条件同例3 - 5。对f 的计算如图15 - 2所示。当$i=5$时，f 由数对集合$P(5)=[(0,0),(4,6)]$表示。而$P(4)$、$P(3)$和$P(2)$分别为[(0 , 0) , (4 , 6) , (9 , 10)]、[(0 , 0) (4 , 6) , (9 , 10) , (10 , 11)] 和[(0 , 0) (2 , 3) (4 , 6) (6 , 9) (9 , 10) (10 , 11)]。

为求$f(1,10)$，利用式（15 - 2）得$f(1,10)=\max\{f(2,10),f(2,8)+p1\}$。由$P(2)$得$f(2,10)=11$、$f(2,8)=9$ ($f(2,8)=9$来自数对(6， 9))，因此$f(1,10)=\max\{11,15\}=15$。现在来求x_i 的值，因为$f(1,10)=f(2,6)+p_1$，所以$x_1=1$；由$f(2,6)=f(3,6-w2)+p_2=f(3,4)+p_2$，得$x_2=1$；由$f(3,4)=f(4,4)=f(5,4)$得$x_3=x_4=0$；最后，因$f(5,4)\neq 0$得$x_5=1$。

检查每个$P(i)$ 中的数对，可以发现每对$(y,f(i,y))$ 对应于变量$x_i,.,x_n$ 的0/1 赋值的不同组合。设（a,b）和（c,d）是对应于两组不同$x_i,.,x_n$ 的0 / 1赋值，若$a\ge c$且$b<d$，则(a,b) 受(b,c) 支配。被支配者不必加入$P(i)$中。若在相同的数对中有两个或更多的赋值，则只有一个

放入$P(i)$。假设$w_n \le C$，$P(n)=[(0,0), (w_n, p_n)]$，$P(n)$中对应于x_n的两个数对分别等于0和1。对于每个i，$P(i)$可由$P(i+1)$得出。首先，要计算数对的有序集合Q，使得当且仅当$w_i \le s \le c$且$(s-w_i, t-p_i)$为$P(i+1)$中的一个数对时，(s,t)为Q中的一个数对。现在Q中包含$x_i=1$时的数对集，而$P(i+1)$对应于$x_i=0$的数对集。接下来，合并Q和$P(i+1)$并删除受支配者和重复值即可得到$P(i)$。

例**3-7** 各数据同例15-6。$P(5)=[(0,0),(4,6)]$, 因此$Q=[(5,4),(9,10)]$。现在要将$P(5)$和Q合并得到$P(4)$。因$(5,4)$受$(4,6)$支配，可删除$(5,4)$，所以$P(4)=[(0,0), (4,6), (9,10)]$。接着计算$P(3)$，首先由$P(4)$得$Q=[(6,5), (10,11)]$，然后又由合并方法得$P(3)=[(0,0), (4,6), (9,10), (10,11)]$。最后计算$P(2)$：由$P(3)$得$Q=[(2,3),(6,9)]$，$P(3)$与$Q$合并得$P(2)=[(0,0), (2,3), (4,6), (6,9), (9,10). (10,11)]$。因为每个$P(i)$中的数对对应于$x_i, ., x_n$的不同0/1赋值，因此$P(i)$中的数对不会超过$2^{n-i+1}$个。计算$P(i)$时，计算$Q$需消耗$(|P(i+1)|)$的时间，合并$P(i+1)$和$Q$同样需要$(|P(i+1)|)$的时间。计算所有$P(i)$时所需要的总时间为：$(\sum_{i=2}^{n}|P(i+1)|=O(2^n)$。当权为整数时，$|P(i)| \le c+1$, 此时复杂性为$O(min\{nc, 2^n\})$。

如6.4.3节定义的，数字化图像是$m \times m$的像素阵列。假定每个像素有一个0～255的灰度值。因此存储一个像素至多需8位。若每个像素存储都用最大位8位，则总的存储空间为$8m^2$位。为了减少存储空间，我们将采用变长模式（variable bit scheme），即不同像素用不同位数来存储。像素值为0和1时只需1位存储空间；值2、3各需2位；值4，5，6和7各需3位；以此类推，使用变长模式的步骤如下：

1) 图像线性化根据图15-3a中的折线将$m \times m$维图像转换为$1 \times m^2$维矩阵。

2) 分段将像素组分成若干个段，分段原则是：每段中的像素位数相同。每个段是相邻像素的集合且每段最多含256个像素，因此，若相同位数的像素超过256个的话，则用两个以上的段表示。

3) 创建文件创建三个文件：*SegmentLength, BitsPerPixel* 和*Pixels*。第一个文件包含在2)中所建的段的长度(减1)，文件中各项均为8位长。文件*BitsPerPixel*给出了各段中每个像素的存储位数（减1），文件中各项均为3位。文件*Pixels*则是以变长格式存储的像素的二进制串。

4) 压缩文件压缩在3)中所建立的文件，以减少空间需求。

上述压缩方法的效率（用所得压缩率表示）很大程度上取决于长段的出现频率。

例**3-8** 考察图15-3b的4×4图像。按照蛇形的行主次序，灰度值依次为10，9，12，40，50，35，15，12，8，10，9，15，11，130，160和240。各像素所需的位数分别为4，4，4，6，6，6，4，4，4，4，4，4，8，8和8，按等长的条件将像素分段，可以得到4个段[10，9，12]、[40，50，35]、[15, 12, 8, 10, 9, 15, 11] 和［130, 160, 240］。因此，文件*SegmentLength*为2，2，6，2；文件*BitsPerSegment*的内容为3，5，3，7；文件*Pixels*包含了按蛇形行主次序排列的16个灰度值，其中头三个各用4位存储，接下来三个各用6位，再接下来的七个各用4位，最后三个各用8位存储。因此存储单元中前30位存储了前六个像素：

1010 1001 1100 111000 110010 100011

这三个文件需要的存储空间分别为：文件*SegmentLength*需32位；*BitsPerSegment*需12位；*Pixels*需82位，共需126位。而如果每个像素都用8位存储，则存储空间需8×16=128位，因而在本例图像中，节省了2位的空间。

假设在2)之后，产生了n个段。段标题（segment header）用于存储段的长度以及该段中每个像素所占用的位数。每个段标题需11位。现假设l_i和b_i分别表示第i段的段长和该段每个像素的长度，则存储第i段像素所需要的空间为$l_i * b_i$。在2)中所得的三个文件的总存储空间为$11n+\sum_{i=1}^{n} l_i b_i$。可通过将某些相邻段合并的方式来减少空间消耗。如当段i和$i+1$被合并时，合并后的段长应为$l_i + l_{i+1}$。此时每个像素的存储位数为$max\{b_i, b_{i+1}\}$位。尽管这种技术增加了文件Pixels的空间消耗，但同时也减少了一个段标题的空间。

例**3-9** 如果将例15-8中的第1段和第2段合并，合并后，文件*SegmentLength*变为5，6，2，*BitsPerSegment*变为5，3，7。而文件Pixels的前36位存储的是合并后的第一段：001010 001001 001100 111000 110010 100011其余的像素（例15-8第3段）没有改变。因为减少了1个段标题，文件*SegmentLength*和*BitsPerPixel*的空间消耗共减少了11位，而文件*Pixels*的空间增加6位，因此总共节约的空间为5位，空间总消耗为121位。

我们希望能设计一种算法，使得在产生n个段之后，能对相邻段进行合并，以便产生一个具有最小空间需求的新的段集合。在合并相邻段之后，可利用诸如LZW法（见7.5节）和霍夫曼编码（见9.5.3节）等其他技术来进一步压缩这三个文件。

令s_q为前q个段的最优合并所需要的空间。定义$s_0=0$。考虑第i段$(i>0)$，假如在最优合并C中，第i段与第$i-1$，$i-2$，.，$i-r+1$段相合并，而不包括第$i-r$段。合并C所需要的空间消耗等于：第1段到第$i-r$段所需空间$+ lsum(i-r+1, i) * bmax(i-r+1, i) + 11$

其中$lsum(a, b)=\sum_{j=a}^{b} l_j$，$bmax(a, b)=max\{b_a, ..., b_b\}$。假如在$C$中第1段到第$i-r$段的合并不是最优合并，那么需要对合并进行修改，以使其具有更小的空间需求。因此还必须对段1到段$i-r$进行最优合并，也即保证最优原则得以维持。故C的空间消耗为：

$s_i = s_{i-r} + lsum(i-r+1, i) * bmax(i-r+1, i) + 11$

r的值介于1到i之间，其中要求$lsum$不超过256(因为段长限制在256之内)。尽管我们不知道如何选择r，但我们知道，由于C具有最小的空间需求，因此在所有选择中，r必须产生最小的空间需求。

假定kay_i表示取得最小值时k的值，s_n为n段的最优合并所需要的空间，因而一个最优合并可用kay的值构造出来。

例**3-10** 假定在2)中得到五个段，它们的长度为[6，3，10，2，3]，像素位数为[1，2，3，2，1]，要用公式（15-3）计算s_n，必须先求出s_{n-1}，.，s_0的值。s_0为0，现计算s_1：$s_1 = s_0 + l_1 * b_1 + 11 = 17$ $kay_1 = 1$ s_2由下式得出：

$s_2 = min\{s_1 + l_2 b_2, s_0 + (l_1 + l_2) * max\{b_1, b_2\}\} + 11 = min\{17 + 6, 0 + 9 * 2\} + 11 = 29$

$kay_2 = 2$

以此类推，可得$s_1.s_5 = [17, 29, 67, 73, 82]$，$kay_1.kay_5 = [1, 2, 2, 3, 4]$。因为$s_5 = 82$，所以最优空间合并需82位的空间。可由$kay_5$导出本合并的方式，过程如下：因为$kay_5 = 4$，所以$s_5$是由公式（15-3）在$k=4$时取得的，因而最优合并包括：段1到段$(5-4) = 1$的最优合并以及段2，3，4和5的合并。最后仅剩下两个段：段1以及段2到段5的合并段。

1. 递归方法

用递归式（15-3）可以递归地算出s_i和kay_i。程序15-3为递归式的计算代码。l，b，和kay是一维的全局整型数组，L是段长限制（256），header为段标题所需的空间(11)。调用S(n)返回s_n的值且同时得出kay值。调用Traceback(kay, n)可得到最优合并。

现讨论程序15-3的复杂性。$t(0)=c$（c为一个常数）；$(n>0)$，因此利用递归的方法可得$t(n)=O(2^n)$。Traceback的复杂性为(n)。

程序15-3 递归计算s , k a y及最优合并

```
int S(int i)
{ / /返回S ( i )并计算k a y [ i ]
if (i == 0 ) return 0;
//k = 1时, 根据公式（ 1 5 - 3）计算最小值
int lsum = l[i],bmax = b[i];
int s = S(i-1) + lsum * bmax;
kay[i] = 1;
/ /对其余的k计算最小值并求取最小值
for (int k = 2; k <= i && lsum+l[i-k+1] <= L; k++) {
lsum += l[i-k+1];
if (bmax < b[i-k+1]) bmax = b[i-k+1];
int t = S(i-k);
if (s > t + lsum * bmax) {
s = t + lsum * bmax;
kay[i] = k;}
}
return s + header;
}
void Traceback(int kay[], int n)
{// 合并段
if (n == 0) return;
Tr a c e b a c k ( k a y, n-kay[n]);
cout << "New segment begins at " << (n - kay[n] + 1) << endl;
}
```

2. 无重复计算的递归方法

通过避免重复计算s_i，可将函数S的复杂性减少到(n)。注意这里只有n个不同的s_i。

例**3 - 11** 再考察例1 5 - 1 0中五个段的例子。当计算s_5时，先通过递归调用来计算s_4，.，s_0。计算s_4时，通过递归调用计算s_3，.，s_0，因此s_4只计算了一次，而s_3计算了两次，每一次计算s_3要计算一次s_2，因此s_2共计算了四次，而s_1重复计算了1 6次！可利用一个数组s来保存先前计算过的s_i以避免重复计算。改进后的代码见程序1 5 - 4，其中s为初值为0的全局整型数组。

程序15-4 避免重复计算的递归算法

```
int S(int i)
{ / /计算S ( i )和k a y [ i ]
/ /避免重复计算
if (i == 0) return 0;
if (s[i] > 0) return s[i]; //已计算完
/ /计算s [ i ]
/ /首先根据公式（1 5 - 3）计算k = 1时最小值
int lsum = l[i], bmax = b[i];
s[i] =S(i-1) + lsum * bmax;
kay[i] = 1;
/ /对其余的k计算最小值并更新
for (int k = 2; k <= i && lsum+l[i-k+1] <= L; k++) {
lsum += l[i-k+1];
if (bmax < b[i-k+1]) bmax = b[i-k+1];
int t = S(i-k);
if (s[i] > t + lsum * bmax) {
s[i] = t + lsum * bmax;
kay[i] = k;}
}
s[i] += header;
return s[i];
}
```

为了确定程序1 5 - 4的时间复杂性，我们将使用分期计算模式（ amortization scheme）。在该模式中，总时间被分解为若干个不同项，通过计算各项的时间然后求和来获得总时间。当计算s_i时，若s_j还未算出，则把调用S(j) 的消耗计入s_j；若s_j已算出，则把$S(j)$的消耗计入s_i(这里s_j依次把计算新s_q的消耗转移至每个s_q)。程序1 5 - 4的其他消耗也被计入s_i。因为L是2 5 6之内的常数且每个l_i至少为1，所以程序1 5 - 4的其他消耗为(1)，即计入每个s_i的量是一个常数，且s_i数目为n，因而总工作量为(n)。

3. 迭代方法

倘若用式（1 5 - 3）依序计算s_1, ., s_n，便可得到一个复杂性为(n)的迭代方法。在该方法中，在s_i计算之前，s_j必须已计算好。该方法的代码见程序1 5 - 5，其中仍利用函数Tr a c e b a c k（见程序1 5 - 3）来获得最优合并。

程序15-5 迭代计算s和k a y

```
void Vbits (int l[], int b[], int n, int s[], int kay[])
{ / /计算s [ i ]和k a y [ i ]
int L = 256, header = 11 ;
s[0] = 0;
/ /根据式（1 5 - 3）计算s [ i ]
```

```
for (int i = 1; i <= n; i++) {
// k = 1时,计算最小值
int lsum = l[i],
bmax = b[i];
s[i] = s[i-1] + lsum * bmax;
kay[i] = 1;
/ /对其余的k计算最小值并更新
for (int k=2; k<= i && lsum+l[i-k+1]<= L; k++) {
lsum += l[i-k+1];
if (bmax < b[i-k+1]) bmax = b[i-k+1];
if (s[i] > s[i-k] + lsum * bmax){
s[i] = s[i-k] + lsum * bmax;
kay[i] = k; }
}
s[i] += header;
}
}
```

3.2.3 矩阵乘法链

$m\times n$矩阵A与$n\times p$矩阵B相乘需耗费$(m\ n\ p)$的时间（见第2章练习16）。我们把$m\ n\ p$作为两个矩阵相乘所需时间的测量值。现假定要计算三个矩阵A、B和C的乘积，有两种方式计算此乘积。在第一种方式中，先用A乘以B得到矩阵D，然后D乘以C得到最终结果，这种乘法的顺序可写为$(A*B)*C$。第二种方式写为$A*(B*C)$,道理同上。尽管这两种不同的计算顺序所得的结果相同，但时间消耗会有很大的差距。

例3-12 假定A为1000×1矩阵，B为1×1000矩阵，C为1000×1矩阵，则$A*B$的时间耗费为10000，得到的结果D为1000×1000矩阵，再与C相乘所需的时间耗费为1 000 000，因此计算$(A*B)*C$的总时间为1 010 000。$B*C$的时间耗费为10 000，得到的中间矩阵为1×1矩阵，再与A相乘的时间消耗为100，因而计算$A*（B*C）$的时间耗费竟只有10 100！而且，计算（$A*B$）$*C$时，还需10 000个单元来存储$A*B$，而$A*（B*C）$计算过程中，只需用1个单元来存储$B*C$。

下面举一个得益于选择合适秩序计算$A*B*C$矩阵的实例：考虑两个3维图像的匹配。图像匹配问题的要求是，确定一个图像需旋转、平移和缩放多少次才能逼近另一个图像。实现匹配的方法之一便是执行约1000次迭代计算，每次迭代需计算12×1个向量T:

$$T=?A(x,y,z)*B(x,y,z)*C(x,y,z)$$

其中A，B和C分别为12×3，3×3和3×1矩阵。(x,y,z)为矩阵中向量的坐标。设t表示计算$A(x,y,z)*B(x,y,z)*C(x,y,z)$的计算量。假定此图像含256×256×256个向量，在此条件中，这1000个迭代所需的总计算量近似为$1000*256^3*t\approx 1.7*10^9 t$。若三个矩阵是按由左向右的顺序相乘的，则$t=12*3*3+12*3*1=144$；但如果从右向左相乘，$t=3*3*1+12*3*1=45$。由左至右计算约需$2.4*10^{11}$个操作，而由右至左计算大概只需$7.5*10^{10}$个操作。假如使用一个每秒可执行1亿次操作的计算机，由左至右需40分钟，而由右至左只需12.5分钟。

在计算矩阵运算$A*B*C$时，仅有两种乘法顺序（由左至右或由右至左），所以可以很容易算出每种顺序所需要的操作数，并选择操作数比较少的那种乘法顺序。但对于更多矩阵相乘来说，情况要复杂得多。如计算矩阵乘积$M_1\times M_2\times\ldots\times M_q$，其中$M_i$是一个$r_i\times r_{i+1}$矩阵$(1\le i\le q)$。不妨考虑$q=4$的情况，此时矩阵运算$A*B*C*D$可按以下方式（顺序）计算:

$A*((B*C)*D)\ A*(B*(C*D))\ (A*B)*(C*D)\ (A*(B*C))*D$

不难看出计算的方法数会随q以指数级增加。因此，对于很大的q来说，考虑每一种计算顺序并选择最优者已是不切实际的。

现在要介绍一种采用动态规划方法获得矩阵乘法次序的最优策略。这种方法可将算法的时间消耗降为(q^3)。用M_{ij}表示链$M_i\times\ldots\times M_j$（$i\le j$）的乘积。设$c(i,j)$为用最优法计算$M_{ij}$的消耗，$kay(i,j)$为用最优法计算$M_{ij}$的最后一步$M_{ik}\times M_{k+1,j}$的消耗。因此$M_{ij}$的最优算法包括如何用最优算法计算$M_{ik}$和$M_{kj}$以及计算$M_{ik}\times M_{kj}$。根据最优原理，可得到如下的动态规划递归式：$kay(i,i+s)=$获得上述最小值的$k$. 以上求$c$的递归式可用递归或迭代的方法来求解。$c(1,q)$为用最优法计算矩阵链的消耗，$kay(1,q)$为最后一步的消耗。其余的乘积可由$kay$值来确定。

1. 递归方法

与求解0/1背包及图像压缩问题一样，本递归方法也须避免重复计算$c(i,j)$和$kay(i,j)$，否则算法的复杂性将会非常高。

例3-13 设$q=5$和$r=(10,5,1,10,2,10)$，式中待求的c中有四个c的$s=0$或1，因此用动态规划方法可立即求得它们的值：$c(1,1)=c(5,5)=0$;$c(1,2)=50$; $c(4,5)=200$。现计算$C(2,5)$：$c(2,5)=min\{c(2,2)+c(3,5)+50,\ c(2,3)+c(4,5)+500,\ c(2,4)+c(5,5)+100\}$（15-5）其中$c(2,2)=c(5,5)=0$；$c(2,3)=50$；$c(4,5)=200$。再用递归式计算$c(3,5)$及$c(2,4)$:$c(3,5)=min\{c(3,3)+c(4,5)+100,\ c(3,4)+c(5,5)+20\}=min\{0+200+100,\ 20+0+20\}=40$ $c(2,4)=min\{c(2,2)+c(3,4)+100,\ c(2,3)+c(4,4)+100\}=min\{0+20+10,\ 50+10+20\}=30$由以上计算还可得$kay(3,5)=4$，$kay(2,4)=2$。现在,计算$c(2,5)$所需的所有中间值都已求得，将它们代入式（15-5）得:

$c(2,5)=\min\{0+40+50,\ 50+200+500,\ 30+0+100\}=90$且$kay(2,5)=2$

再用式（15-4）计算$c(1,5)$，在此之前必须算出$c(3,5)$、$c(1,3)$和$c(1,4)$。同上述过程，亦可计算出它们的值分别为40、150和90，相应的kay值分别为4、2和2。代入式（15-4）得:

$c(1,5)=\min\{0+90+500,\ 50+40+100,\ 150+200+1000,\ 90+0+200\}=190$且$kay(1,5)=2$

此最优乘法算法的消耗为190，由$kay(1,5)$值可推出该算法的最后一步，$kay(1,5)$等于2，因此最后一步为$M_{12}\times M_{35}$，而M_{12}和M_{35}都是用最优法计算而来。由$kay(1,2)=1$知M_{12}等于$M_{11}\times M_{22}$，同理由$kay(3,5)=4$得知M_{35}由$M_{34}\times M_{55}$算出。依此类推，M_{34}由$M_{33}\times M_{44}$得出。因而此最优乘法算法的步骤为:

$M_{11}\times M_{22}=M_{12}$

$M_{33}\times M_{44}=M_{34}$

$M_{34} \times M_{55} = M_{35}$

$M_{12} \times M_{35} = M_{15}$

计算$c(i, j)$和k a y (i, j)的递归代码见程序1 5 - 6。在函数C中，r为全局一维数组变量，k a y是全局二维数组变量，函数C返回$c(i\ j)$之值且置k a y $[a]$ $[b]$ $=k\ ay\ (a\ ,\ b)$ (对于任何$a\ ,\ b$)，其中$c(a\ ,\ b)$在计算$c(i,j)$时皆已算出。函数Traceback 利用函数C中已算出的k a y值来推导出最优乘法算法的步骤。

设$t(q)$为函数C的复杂性，其中$q=j-i+1$（即M_{ij}是q个矩阵运算的结果）。当q为1或2时，$t(q)=d$，其中d为一常数；而$q>2$时，$t(q)=2_{q-1}?_{k=1}t\ (k\ )+e\ q$，其中$e$是一个常量。因此当$q>2$时，$t(q)>2t\ (q-1\ )+e$，所以$t\ (q)=\mathbb{W}\ (2_q)$。函数Traceback 的复杂性为$(q)$。

程序15-6 递归计算$c\ (i, j)$和$kay\ (i, j)$

```
int C(int i, int j)
{ / /返回c(i,j) 且计算k(i,j) = kay[i][j]
if (i==j) return 0; //一个矩阵的情形
if (i == j-1) { //两个矩阵的情形
kay[i][i+1] = i;
return r[i]*r[i+1]*r[r+2];}
/ /多于两个矩阵的情形
/ /设u为k = i 时的最小值
int u = C(i,i) + C(i+1,j) + r[i]*r[i+1]*r[j+1];
kay[i][j] = i;
/ /计算其余的最小值并更新u
for (int k = i+1; k < j; k++) {
int t = C(i,k) + C(k+1,j) + r[i]*r[k+1]*r[j+1];
if (r < u) {//小于最小值的情形
u = t;
kay[i][j] = k;
}
return u;
}
void Traceback (int i, int j ,int **kay)
{ / /输出计算Mij 的最优方法
if ( i == j) return;
Traceback(i, kay[i][j], kay);
Traceback(kay[i][j]+1, j, kay);
cout << "Multiply M" << i << ", "<< kay[i][j];
cout << " and M " << (kay[i][j]+1) << ", " << j << end1;
}
```

2. 无重复计算的递归方法

若避免再次计算前面已经计算过的c（及相应的$k\ a\ y$），可将复杂性降低到（q_3）。而为了避免重复计算，需用一个全局数组c[ ][ ]存储$c(i, j)$值，该数组初始值为0。函数C的新代码见程序1 5 - 7：

程序15-7 无重复计算的$c\ (i, j)$计算方法

```
int C(int i,int j)
{ / /返回c(i,j) 并计算k a y ( i , j ) = k a y [ I ] [ j ]
/ /避免重复计算
/ /检查是否已计算过
if （c[i][j] >） return c[i][j];
/ /若未计算,则进行计算
if(i==j) return 0; //一个矩阵的情形
i f ( i = = j - 1 ) { / /两个矩阵的情形
kay[i][i+1]=i;
c [ i ] [ j ] = r [ i ] * r [ i + 1 ] * r [ i + 2 ] ;
return c[i][j];}
/ /多于两个矩阵的情形
/ /设u为k = i 时的最小值
int u=C(i,i)+C(i+1,j)+r[i]*r[i+1]*r[j+1];
k a y [ i ] [ j ] = i ;
/ /计算其余的最小值并更新u
for (int k==i+1; k<j;k++){
int t=C(i,k)+C(k+1,j)+r[i]*r[k+1]*r[j+1];
if (t<u) {// 比最小值还小
u = t ;
k a y [ i ] [ j ] = k ; }
}
c [ i ] [ j ] = u ;
return u;
}
```

为分析改进后函数C 的复杂性，再次使用分期计算方法。注意到调用C(1, q) 时每个$c\ (i, j)$（$1\le i\le j\le q$）仅被计算一次。要计算尚未计

算过的$c(a,b)$，需附加的工作量$s=j-i>1$。将s计入第一次计算$c(a,b)$时的工作量中。在依次计算$c(a,b)$时，这个s会转计到每个$c(a,b)$的第一次计算时间c中，因此每个$c(i,i)$均被计入s。对于每个s，有$q-s+1$个$c(i,j)$需要计算，因此总的工作消耗为$\sum_{s=1}^{q-1}(q-s+1)=(q^3)$。

3. 迭代方法

c的动态规划递归式可用迭代的方法来求解。若按$s=2,3,\ldots,q-1$的顺序计算$c(i,i+s)$，每个c和kay仅需计算一次。

例**3-14** 考察例3-13中五个矩阵的情况。先初始化$c(i,i)$ $(0\le i\le 5)$为0，然后对于$i=1,\ldots,4$分别计算$c(i,i+1)$。$c(1,2)=r_1r_2r_3=50$，$c(2,3)=5\,0$，$c(3,4)=20$和$c(4,5)=2\,0\,0$。相应的kay值分别为1，2，3和4。

当$s=2$时，可得：

$$c(1,3)=\min\{c(1,1)+c(2,3)+r_1r_2r_4,\ c(1,2)+c(3,3)+r_1r_3r_4\}=\min\{0+50+500,50+0+100\}=150$$

且$kay(1,3)=2$。用相同方法可求得$c(2,4)$和$c(3,5)$分别为30和40，相应kay值分别为2和3。

当$s=3$时，需计算$c(1,4)$和$c(2,5)$。计算$c(2,5)$所需要的所有中间值均已知(见(15-5)式)，代入计算公式后可得$c(2,5)=90$，$kay(2,5)=2$。$c(1,4)$可用同样的公式计算。最后，当$s=4$时，可直接用（15-4）式来计算$c(1,5)$，因为该式右边所有项都已知。

计算c和kay的迭代程序见函数MatrixChain（见程序15-8），该函数的复杂性为(q^3)。计算出kay后同样可用程序15-6中的Traceback函数推算出相应的最优乘法计算过程。

程序15-8 c和kay的迭代计算

```
void MatrixChain(int r[], int q, int **c, int **kay)
{// 为所有的Mij 计算耗费和kay
// 初始化c[i][i], c[i][i+1]和kay[i][i+1]
for (int i = 1; i < q; i++) {
c[i][i] = 0;
c[i][i+1] = r[i]*r[i+1]*r[i+2];
kay[i][i+1] = i;
}
c[q][q] = 0;
// 计算余下的c和kay
for (int s = 2; s < q; s++)
for (int i = 1; i <= q - s; i++) {
// k = i时的最小项
c[i][i+s] = c[i][i] + c[i+1][i+s] + r[i]*r[i+1]*r[i+s+1];
kay[i][i+s] = i;
// 余下的最小项
for (int k = i+1; k < i + s; k++) {
int t = c[i][k] + c[k+1][i+s] + r[i]*r[k+1]*r[i+s+1];
if (t < c[i][i+s]) {// 更小的最小项
c[i][i+s] = t;
kay[i][i+s] = k;}
}
}
}
```

3.2.4 最短路径

假设G为有向图，其中每条边都有一个长度（或耗费），图中每条有向路径的长度等于该路径中各边的长度之和。对于每对顶点(i,j)，在顶点i与j之间可能有多条路径，各路径的长度可能各不相同。我们定义从i到j的所有路径中，具有最小长度的路径为从i到j的最短路径。

例3-15 如图15-4所示。从顶点1到顶点3的路径有

1) 1,2,5,3
2) 1,4,3
3) 1,2,5,8,6,3
4) 1,4,6,3

由该图可知,各路径相应的长度为10、28、9、27，因而路径3)是该图中顶点1到顶点3的最短路径。

在所有点对最短路径问题（all-pair sshorest-paths problem）中，要寻找有向图G中每对顶点之间的最短路径。也就是说，对于每对顶点(i,j)，需要寻找从i到j的最短路径及从j到i的最短路径。因此对于一个n个顶点的图来说，需寻找$p=n(n-1)$条最短路径。假定图G中不含有长度为负数的环路，只有在这种假设下才可保证G中每对顶点(i,j)之间总有一条不含环路的最短路径。当有向图中存在长度小于0的环路时，可能得到长度为$-\infty$的更短路径，因为包含该环路的最短路径往往可无限多次地加上此负长度的环路。

设图G中n个顶点的编号为1到n。令$c(i,j,k)$表示从i到j的最短路径的长度，其中k表示该路径中的最大顶点。因此，如果G中包含边$<i,j>$，则$c(i,j,0)=$边$<i,j>$的长度；若$i=j$，则$c(i,j,0)=0$；如果G中不包含边$<i,j>$，则$c(i,j,0)=+\infty$。$c(i,j,n)$则是从i到j的最短路径的长度。

例**3-16** 考察图15-4。若$k=0,1,2,3$，则$c(1,3,k)=\infty$；$c(1,3,4)=28$；若$k=5,6,7$，则$c(1,3,k)=10$；若$k=8,9,10$，则$c(1,3,k)=9$。因此1到3的最短路径长度为9。对于任意$k>0$，如何确定$c(i,j,k)$呢？中间顶点不超过k的i到j的最短路径有两种可能：该路径含或不含中间顶点k。若不含，则该路径长度应为$c(i,j,k-1)$，否则长度为$c(i,k,k-1)+c(k,j,k-1)$。$c(i,j,k)$可取两者中的最小值。因此可得到如下递归式：

$$c(i,j,k)=\min\{c(i,j,k-1),\ c(i,k,k-1)+c(k,j,k-1)\},\quad k>0$$

以上的递归公式将一个k级运算转化为多个k-1级运算，而多个k-1级运算应比一个k级运算简单。如果用递归方法求解上式，则计算最终结果的复杂性将无法估量。令$t(k)$为递归求解$c(i, j, k)$的时间。根据递归式可以看出$t(k) = 2t(k-1) + c$。利用替代方法可得$t(n) = (2^n)$。因此得到所有$c(i, j, n)$的时间为$(n^2 2^n)$。

当注意到某些$c(i, j, k-1)$值可能被使用多次时，可以更高效地求解$c(i, j, n)$。利用避免重复计算$c(i, j, k)$的方法，可将计算c值的时间减少到(n^3)。这可通过递归方式（见程序15-7矩阵链问题）或迭代方式来实现。出迭代算法的伪代码如图15-5所示。

```
//寻找最短路径的长度
//初始化c(i, j, 1)
for (int i=1; i<=n; i++)
  for (int j=1; j<=n; j++)
    c(i, j, 0) = a(i, j); // a是长度邻接矩阵
    //计算c(i, j, k) (0<k<=n)
for(int k=1;k<=n;k++)
  for (int i=1;i<=n;i++)
    for (int j=1;j<=n;j++)
      if (c(i,k,k-1)+c(k,j,k-1)<c(i,j,k-1))
        c(i,j,k) = c(i,k,k-1) + c(k,j,k-1);
      else c(i,j,k) = c(i,j,k-1);
```

图15-5 最短路径算法的伪代码

注意到对于任意i，$c(i,k,k) = c(i,k,k-1)$且$c(k,i,k) = c(k,i,k-1)$，因而，若用$c(i,j)$代替图15-5的$c(i,j,k)$，最后所得的$c(i,j)$之值将等于$c(i,j,n)$值。此时图15-5可改写成程序15-9的C++代码。程序15-9中还利用了程序12-1中定义的AdjacencyWDigraph类。函数AllPairs在c中返回最短路径的长度。若i到j无通路，则c[i][j]被赋值为NoEdge。函数AllPairs同时计算了kay[i][j]，其中kay[i][j]表示从i到j的最短路径中最大的k值。在后面将看到如何根据kay值来推断出从一个顶点到另一顶点的最短路径（见程序15-10中的函数OutputPath）。

程序15-9的时间复杂性为(n^3)，其中输出一条最短路径的实际时间为O(n)。

程序15-9 c和kay的计算

```
template<class T>
void AdjacencyWDigraph<T>::Allpairs(T **c, int **kay)
{ //所有点对的最短路径
//对于所有i和j，计算c[i][j]和kay[i][j]
//初始化c[i][j] = c(i, j, 0)
for (int i = 1; i <= n; i++)
for (int j = 1; j <= n; j++) {
c[i][j] = a[i][j];
kay[i][j] = 0;
}
for (i = 1; i <= n; i++)
c[i][i] = 0;
// 计算c[i][j] = c(i,j,k)
for (int k = 1; k <= n; k++)
for (int i = 1; i <= n; i++)
for (int j = 1; j <= n; j++) {
T t1 = c[i][k];
T t2 = c[k][j];
T t3 = c[i][j];
if (t1 != NoEdge && t2 != NoEdge && (t3 == NoEdge || t1 + t2 < t3)) {
c[i][j] = t1 + t2;
kay[i][j] = k;}
}
}
```

程序15-10 输出最短路径

```
void outputPath(int **kay, int i, int j)
{// 输出i到j的路径的实际代码
if (i == j) return;
if (kay[i][j] == 0) cout << j << ' ';
else {outputPath(kay, i, kay[i][j]);
outputPath(kay, kay[i][j], j);}
}
template<class T>
void OutputPath(T **c, int **kay, T NoEdge, int i, int j)
{// 输出从i到j的最短路径
if (c[i][j] == NoEdge) {
cout << "There is no path from " << i << " to " << j << endl;
return; }
cout << "The path is" << endl;
```

```
cout << i << ' ';
o u t p u t P a t h ( k a y,i , j ) ;
cout << endl;
}
```

例**3-17** 图15-6a 给出某图的长度矩阵a，15-6b 给出由程序1 5 - 9所计算出的*c* 矩阵，15-6c 为对应的*k a y*值。根据15-6c 中的*kay* 值，可知从1到5的最短路径是从1到k a y [1] [5] = 4的最短路径再加上从4到5的最短路径，因为k a y [4] [5] = 0，所以从4到5的最短路径无中间顶点。从1到4的最短路径经过k a y [1] [4] = 3。重复以上过程，最后可得1到5的最短路径为：1，2，3，4，5。

3.2.5 网络的无交叉子集

在11 . 5 . 3节的交叉分布问题中，给定一个每边带n 个针脚的布线通道和一个排列C。顶部的针脚i 与底部的针脚C_i 相连，其中$1 \le i \le n$，数对(i, C_i) 称为网组。总共有n 个网组需连接或连通。假设有两个或更多的布线层，其中有一个为优先层，在优先层中可以使用更细的连线，其电阻也可能比其他层要小得多。布线时应尽可能在优先层中布设更多的网组。而剩下的其他网组将布设在其他层。当且仅当两个网组之间不交叉时，它们可布设在同一层。我们的任务是寻找一个最大无交叉子集（Maximum Noncrossing Su b s e t，M N S)。在该集中，任意两个网组都不交叉。因(i, C_i) 完全由i 决定，因此可用i 来指定(i, C_i)。

例**3-18** 考察图1 5 - 7（对应于图1 0 - 1 7）。(1 , 8)和(2 , 7)（也即1号网组和2号网组）交叉，因而不能布设在同一层中。而(1 , 8)，(7,9) 和(9,10) 未交叉，因此可布设在同一层。但这3个网组并不能构成一个M N S，因为还有更大的不交叉子集。图1 0 - 1 7中给出的例子中，集合｛(4 , 2) ,(5 , 5) , (7 , 9) , (9 , 1 0)｝是一个含4个网组的M N S。

设$MNS(i, j)$ 代表一个M N S，其中所有的(u, C_u) 满足$u \le i$，$C_u \le j$。令$size(i,j)$ 表示$MNS(i,j)$的大小(即网组的数目)。显然$MNS(n,n)$是对应于给定输入的M N S，而$size(n,n)$是它的大小。

例**3-19** 对于图1 0 - 1 7中的例子，$MNS(1 0 , 1 0)$是我们要找的最终结果。如例3 - 1 8中所指出的，$size(1 0 , 1 0) = 4$，因为(1 , 8)，(2 , 7)，(7 , 9)，(8 , 3)，(9 , 1 0)和(1 0 , 6)中要么顶部针脚编号比7大，要么底部针脚编号比6大，因此它们都不属于$MNS(7 , 6)$。因此只需考察剩下的4个网组是否属于$MNS(7 , 6)$，如图1 5 - 8所示。子集｛(3 , 4) , (5 , 5)｝是大小为2的无交叉子集。没有大小为3的无交叉子集，因此$size(7 , 6) = 2$。

当$i = 1$时，$(1 , C_1)$ 是$MNS(1 , j)$ 的唯一候选。仅当$j \ge C_1$ 时，这个网组才会是$MNS(1 , j)$ 的一个成员.

下一步，考虑$i > 1$时的情况。若$j < C_i$，则(i, C_i) 不可能是$MNS(i, j)$ 的成员，所有属于$MNS(i,j)$ 的(u, C_u) 都需满足$u < i$且$C_u < j$，因此：$size(i,j) = size(i- 1 , j), j < C_i$ （1 5 - 7）

若$j \ge C_i$，则(i, C_i) 可能在也可能不在$MNS(i,j)$ 内。若(i, C_i) 在$MNS(i,j)$ 内，则在$MNS(i,j)$中不会有这样的(u, C_u)：$u < i$且$C_u > C_i$，因为这个网组必与(i, C_i) 相交。因此$MNS(i,j)$ 中的其他所有成员都必须满足条件$u < i$且$C_u < C_i$。在$MNS(i,j)$ 中这样的网组共有M_{i-1}, C_{i-1} 个。若(i, C_i) 不在$MNS(i,j)$中，则$MNS(i,j)$ 中的所有(u, C_u) 必须满足$u < i$；因此$size(i,j) = size(i- 1 , j)$。虽然不能确定$(i, C_i)$是否在$MNS(i,j)$ 中，但我们可以根据获取更大M N S的原则来作出选择。因此：$size(i,j) = max\{size(i-1 , j), size(i- 1 , C_i-1)+1\}, j \ge C_i$（1 5 - 8）

虽然从（1 5 - 6）式到（1 5 - 8）式可用递归法求解，但从前面的例子可以看出，即使避免了重复计算，动态规划递归算法的效率也不够高，因此只考虑迭代方法。在迭代过程中先用式（1 5 - 6）计算出$size (1 , j)$，然后再用式（1 5 - 7）和（1 5 - 8）按$i = 2, 3, ., n$ 的顺序计算$size (i,j)$，最后再用Traceback 来得到$MNS(n, n)$ 中的所有网组。

例**3-20** 图1 5 - 9给出了图1 5 - 7对应的$size(i,j)$ 值。因$size(1 0 , 1 0) = 4$，可知M N S含4个网组。为求得这4个网组，先从$size (1 0 , 1 0)$入手。可用（1 5 - 8）式算出$size(1 0 , 1 0)$。根据式（1 5 - 8）时的产生原因可知$size (1 0 , 1 0)=size (9 , 1 0)$，因此现在要求$MNS (9 , 1 0)$。由于$MNS (1 0 , 1 0) \ne size(8 , 1 0)$，因此$MNS$ (9,10) 中必包含9号网组。MNS(9,10) 中剩下的网组组成$MNS (8 , C_9- 1)=MNS(8 , 9)$。由$MNS(8 , 9) =MNS$ (7,9) 知，8号网组可以被排除。接下来要求$MNS(7 , 9)$，因为$size(7 , 9) \ne size(6 , 9)$，所以M N S中必含7号网组。$MNS$ (7,9) 中余下的网组组成$MNS (6 , C_7- 1) =MNS(6 , 8)$。根据$size(6 , 8) =size(5 , 8)$可排除6号网组。按同样的方法，5号网组，3号网组加入M N S中，而4号网组等其他网组被排除。因此回溯过程所得到的大小为4的M N S为{ 3 , 5 , 7 , 9 }。

注意到在回溯过程中未用到$size(1 0 , j) (j \ne 1 0)$，因此不必计算这些值。

程序1 5 - 11给出了计算$size (i , j)$ 的迭代代码和输出M N S的代码。函数M N S用来计算$size (i,j)$ 的值，计算结果用一个二维数组M N来存储。size[i][j] 表示$size(i,j)$，其中i=j= n 或$1 \le i < n$，$0 \le j \le n$，计算过程的时间复杂性为(n_2)。函数Traceback 在N et［0 : m - 1］中输出所得到的MNS，其时间复杂性为(n)。因此求解M M S问题的动态规划算法的总的时间复杂性为(n_2)。

程序1 5 - 11 寻找最大无交叉子集

```
void MNS(int C[], int n, int **size)
{ / /对于所有的i 和j，计算s i z e [ i ] [ j ]
/ /初始化s i z e [ 1 ] [ * ]
for (int j = 0; j < C[1]; j++)
size[1][j] = 0;
for (j = C[1]; j <= n; j++)
size[1][j] = 1;
// 计算size[i][*], 1 < i < n
for (int i = 2; i < n; i++) {
for (int j = 0; j < C[i]; j++)
size[i][j] = size[i-1][j];
for (j = C[i]; j <= n; j++)
size[i][j] = max(size[i-1][j], size[i-1][C[i]-1]+1);
}
size[n][n] = max(size[n-1][n], size[n-1][C[n]-1]+1);
}
void Traceback(int C[], int **size, int n, int Net[], int& m)
```

```
{// 在N e t [ 0 : m - 1 ]中返回M M S
int j = n; // 所允许的底部最大针脚编号
m = 0; // 网组的游标
for (int i = n; i > 1; i--)
// i 号n e t在M N S中?
if (size[i][j] != size[i-1][j]){// 在M N S中
Net[m++] = i;
j = C[i] - 1;}
// 1号网组在M N S中?
if (j >= C[1])
Net[m++] = 1; // 在M N S中
}
```

3.2.6 元件折叠

在设计电路的过程中，工程师们会采取多种不同的设计风格。其中的两种为位－片设计（bit-slice design）和标准单元设计（standard-cell design）。在前一种方法中，电路首先被设计为一个元件栈（如图15-10a 所示）。每个元件C_i宽为w_i，高为h_i，而元件宽度用片数来表示。图15-10a 给出了一个四片的设计。线路是按片来连接各元件的，即连线可能连接元件C_i的第j片到元件C_{i+1}的第j 片。如果某些元件的宽度不足j 片，则这些元件之间不存在片j 的连线。当图1 5 -10a 的位－片设计作为某一大系统的一部分时，则在V L SI (Very Large Scale Integrated) 芯片上为它分配一定数量的空间单元。分配是按空间宽度或高度的限制来完成的。现在的问题便是如何将元件栈折叠到分配空间中去，以便尽量减小未受限制的尺度（如，若高度限制为H时，必须折叠栈以尽量减小宽度W）。由于其他尺度不变，因此缩小一个尺度（如W）等价于缩小面积。可用折线方式来折叠元件栈，在每一折叠点，元件旋转1 8 0°。在图15-10b 的例子中，一个1 2元件的栈折叠成四个垂直栈，折叠点为C_6, C_9和C_{10}。折叠栈的宽度是宽度最大的元件所需的片数。在图15-10b 中，栈宽各为4，3，2和4。折叠栈的高度等于各栈所有元件高度之和的最大值。在图15-10b 中栈1的元件高度之和最大，该栈的高度决定了包围所有栈的矩形高度。

实际上，在元件折叠问题中，还需考虑连接两个栈的线路所需的附加空间。如，在图1 5 -10b 中C_5和C_6间的线路因C_6为折叠点而弯曲。这些线路要求在C_5和C_6之下留有垂直空间，以便能从栈1连到栈2。令r_i为C_i是折叠点时所需的高度。栈1所需的高度为$_5?_{i=1}h_i+r_6$，栈2所需高度为$_8?_{i=6}h_i+r_6+r_9$。

在标准单元设计中，电路首先被设计成为具有相同高度的符合线性顺序的元件排列。假设此线性顺序中的元件为C_1，.，C_n，下一步元件被折叠成如图1 5 - 11所示的相同宽度的行。在此图中，1 2个标准单元折叠成四个等宽行。折叠点是C_4，C_6和C_{11}。在相邻标准单元行之间，使用布线通道来连接不同的行。折叠点决定了所需布线通道的高度。设l_i表示当C_i为折叠点时所需的通道高度。在图1 5 - 11的例子中，布线通道1的高度为l_4，通道2的高度为l_6，通道3的高度为l_{11}。位—片栈折叠和标准单元折叠都会引出一系列的问题，这些问题可用动态规划方法来解决。

1. 等宽位－片元件折叠

定义$r_1 = r_{n+1} =0$。由元件C_i至C_j构成的栈的高度要求为$_j?_{k=i}l_k+ r_j+ r_{j+1}$。设一个位－片设计中所有元件有相同宽度$W$。首先考察在折叠矩形的高度H给定的情况下，如何缩小其宽度。设W_i
为将元件C_i到C_n折叠到高为H的矩形时的最小宽度。若折叠不能实现（如当r_i+h_i＞H时），取$W_i=\infty$。注意到W_1可能是所有n 个元件的最佳折叠宽度。

当折叠C_i到C_n时，需要确定折叠点。现假定折叠点是按栈左到栈右的顺序来取定的。若第一点定为C_{k+1}，则C_i到C_k在第一个栈中。为了得到最小宽度，从C_{k+1}到C_n的折叠必须用最优化方法，因此又将用到最优原理，可用动态规划方法来解决此问题。当第一个折叠点k+ 1已知时，可得到以下公式：

$$W_i = w + W_{k+1} \quad (15-9)$$

由于不知道第一个折叠点，因此需要尝试所有可行的折叠点，并选择满足（1 5 - 9）式的折叠点。令$h\ s\ u\ m(i,k)=_k?_{j=i}h_j$。因$k$+ 1是一个可行的折叠点，因此$h\ s\ u\ m(i, k) +r_i+r_{k+1}$一定不会超过$H$。

根据上述分析，可得到以下动态规划递归式：

这里W_{n+1}=0，且在无最优折叠点k+ 1时W_i为∞。利用递归式（1 5 - 1 0），可通过递归计算W_n, W_{n-1}., W_2, W_1来计算W_i。W_i的计算需要至多检查n-i+ 1个W_{k+1}，耗时为O (n-k)。因此计算所有W_i的时间为O (n_2)。通过保留式（1 5 - 1 0）每次所得的k 值，可回溯地计算出各个最优的折叠点，其时间耗费为O (n)。

现在来考察另外一个有关等宽元件的折叠问题：折叠后矩形的宽度W已知，需要尽量减小其高度。因每个折叠矩形宽为w，因此折叠后栈的最大数量为$s=W/w$。令$H_{i,j}$为C_i, ., C_n折叠成一宽度为jw 的矩形后的最小高度，$H_{1,s}$则是所有元件折叠后的最小高度。当j= 1时，不允许任何折叠，因此：$H_{i,1} =h\ s\ u\ m(i,n) +r_i$, $1\le i\le n$

另外，当$i=n$ 时，仅有一个元件，也不可能折叠，因此：$H_{n,j}=h_n+r_n$, $1\le j\le s$

在其他情况下，都可以进行元件折叠。如果第一个折叠点为k+ 1，则第一个栈的高度为
$h\ s\ u\ m(i,k) +r_i+r_{k+1}$。其他元件必须以至多($j$- 1) *$w$ 的宽度折叠。为保证该折叠的最优性，其他元件也需以最小高度进行折叠.

因为第一个折叠点未知，因此必须尝试所有可能的折叠点，然后从中找出一个使式（1 5 - 11）的右侧取最小值的点，该点成为第一个折叠点。

可用迭代法来求解$H_{i,j}$($1\le i\le n$, $1\le j\le s$)，求解的顺序为：先计算j=2 时的$H_{i,j}$，再算j= 3，.，以此类推。对应每个j 的$H_{i,j}$的计算时间为O (n_2)，所以计算所有$H_{i,j}$的时间为O($s\ n_2$)。通过保存由（1 5 - 1 2）式计算出的每个k 值，可以采用复杂性为O (n) 的回溯过程来确定各个最优的折叠点。

2. 变宽位－片元件的折叠

首先考察折叠矩形的高度H已定，欲求最小的折叠宽度的情况。令W_i如式（1 5 - 1 0）所示，按照与（1 5 - 1 0）式相同的推导过程，可得：

$W_i = \min\{w\,min(i,k) + W_{k+1} \mid sum(i,k) + r_i + r_{k+1} \le H, i \le k \le n\}$ （15-13）

其中$W_{n+1}=0$且$w\,min(i,k) = \min_{i \le j \le k}\{w_j\}$。可用与（15-10）式一样的方法求解（15-13）式，所需时间为$O(n^2)$。

当折叠宽度W给定时，最小高度折叠可用折半搜索方法对超过$O(n^2)$个可能值进行搜索来实现，可能的高度值为$h(i,j)+r_i+r_{j+1}$。在检测每个高度时，也可用（15-13）式来确定该折叠的宽度是否小于等于W。这种情况下总的时间消耗为$O(n^2 \log n)$。

3. 标准单元折叠

用w_i定义单元C_i的宽度。每个单元的高度为h。当标准单元行的宽度W固定不变时，通过减少折叠高度，可以相应地减少折叠面积。考察C_i到C_n的最小高度折叠。设第一个折叠点是C_{s+1}。从元件C_{s+1}到C_n的折叠必须使用最小高度，否则，可使用更小的高度来折叠C_{s+1}到C_n，从而得到更小的折叠高度。所以这里仍可使用最优原理和动态规划方法。

令$H_{i,s}$为C_i到C_n折叠成宽为W的矩形时的最小高度，其中第一个折叠点为C_{s+1}。令$w\,sum(i,s)=\sum_{j=i}^{s} w_j$。可假定没有宽度超过$W$的元件，否则不可能进行折叠。对于$H_{n,n}$因为只有一个元件，不存在连线问题，因此$H_{n,n}=h$。对于$H_{i,s}$（$1 \le i < s \le n$）注意到如果$w\,sum(i,s) > W$，不可能实现折叠。若$w\,sum(i,s) \le W$，元件$C_i$和$C_{j+1}$在相同的标准单元行中，该行下方布线通道的高度为$l_{s+1}$（定义$l_{n+1}=0$）。因而：

$$H_{i,s} = H_{i+1,k} \quad (15\text{-}14)$$

当$i=s<n$时，第一个标准单元行只包含C_i。该行的高度为h且该行下方布线通道的高度为l_{i+1}。因C_{i+1}到C_n单元的折叠是最优的.

为了寻找最小高度折叠，首先使用式（15-14）和（15-15）来确定$H_{i,s}$（$1 \le i \le s \le n$）。最小高度折叠的高度为$\min\{H_{1,s}\}$。可以使用回溯过程来确定最小高度折叠中的折叠点。

练习

1. 修改程序15-1，使它同时计算出能导致最优装载的x_i值。
2. 修改程序15-1，使用一个表格来确定$f(i,y)$是否已被计算过。在求$f(i,y)$时,若表中已经存在该值，则直接取用；若不存在该值，则采用一个递归调用来计算该值。
3. 定义0/1/2背包问题为：$\max\{\sum_{i=1}^{n} p_i x_i\}$。限制条件为：$\sum_{i=1}^{n} w_i x_i \le c$且$x_i \in \{0,1,2\}$，$1 \le i \le n$。设$f$的定义同0/1背包问题中的定义。

1) 从0/1/2背包问题中推出类似于（15-1）和（15-2）的公式。
2) 假设ws为整数。编写一个类似于15-2的程序来计算二维数组f，然后确定最优分配的x值。
3) 程序的复杂性是多少?

4. 二维0/1背包问题定义为：$\max\{\sum_{i=1}^{n} p_i x_i\}$。限制条件为：$\sum_{i=1}^{n} v_i x_i \le c$，$\sum_{i=1}^{n} w_i x_i \le d$且$x_i \in \{0,1\}$，$1 \le i \le n$。设$f(i,y,z)$为二维背包问题最优解的值，其中物品为$i$到$n$，$c=y$，$d=z$。

1) 推出类似于（15-1）和（15-2）式的关于$f(n,y,z)$和$f(i,y,z)$的公式。
2) 假设vs和ws为整数。编写一个类似于15-2的程序计算三维数组f，然后确定最优分配的x值。
3) 程序的复杂性是多少?

*5. 编写一个实现元组方法的C++代码，要求提供一个确定最优装载的x_i值的回溯函数。

6. 当取消段长限制时(即在程序15-3中$L=\infty$)，程序15-3的时间复杂性按如下方式递归定义：

$t(0)=c$（c为常数）；当$n>0$时$t(n)=\sum_{i=0}^{n-1} t(j)+n$。

1) 根据$t(n-1)=\sum_{j=0}^{n-2} t(j)+n-1$证明：当$n>0$时，$t(n)=2t(n-1)+1$。
2) 证明$t(n)=(2^n)$

7. 编写函数Traceback (见程序15-3) 的迭代版本。试说明两个版本各自的优缺点。
8. 编写变长图像压缩过程中1) 和2) 的实现代码。
9. 证明：$\sum_{s=0}^{q-1} s(q-s+1)=(q^3)$。
10. 在求解矩阵乘法递归式时仅用到数组c和kay的上三角。重写程序15-6，定义c和kay为UpperMatrix类（见4.3.4节）的成员。
11. 改写程序15-9，把它作为LinkedWDigraph的类成员，其渐进复杂性应与程序15-9相同。
12. 设G为有n个顶点的有向无环图，G中各顶点的编号为1到n，且当〈i,j〉为G中的一条边时有$i<j$。设$l(i,j)$为边〈i,j〉的长度：

1) 用动态规划方法计算图G中最长路径的长度，算法的时间耗费应为$O(n+e)$，其中e为G中的边数。
2) 编写一个函数，利用1) 中所得到的结果来构造最长路径，其复杂性应为$O(p)$，其中p为该路径的顶点数。

13. 改写程序15-9，首先从一个有向图的邻接矩阵开始，然后计算其反身传递闭包矩阵RTC。若从顶点i到顶点j无通路，则RTC[i][j]=1，否则RTC[i][j]=0。要求代码的复杂性为(n^3)，其中n为图中的顶点数。
14. 利用（15-10）式，编写一个复杂性为$O(n^2)$的C++迭代程序，寻找等宽元件栈的最优折叠点。
15. 用递归式（15-12）代替式（15-10）完成练习14，时间复杂性要求为$O(sn^2)$。
16. 用式（15-13）得出一个变宽元件栈的最小宽度折叠法，时间复杂性要求为$O(n^2)$。
17. 利用15.2.6节的设计，给出一个寻找折叠矩形宽度为W的最小高度折叠算法，其复杂性应为$O(n^2 \log n)$。位－片元件宽度不等。
18. 利用式（15-14）和（15-15）来确定一个含n个标准单元的最小高度折叠。算法的时间复杂性应为$O(n^2)$。能否使用这两个公式得到一个时间复杂性为(n)的算法?

*19. 在13.3.3节可知，一个工程可分解为多个任务且这些任务可按拓扑顺序来完成。设任务从1到n编号，首先完成任务1，然后完成任务2，以此类推..。假设我们有两种方法来完成每个任务。$C_{i,1}$为使用第一种方法完成任务i时的代价，$C_{i,2}$为使用第2种方法完成任务i时的代价。令$T_{i,1}$为第一种方法中任务i的时间耗费，$T_{i,2}$为第二种方法中任务i的时间耗费。并设各个T为整数。设计一个动态规划算法，以得到在时间t内完成所有任务的最小代价的方法。假定工程的代价为各任务的代价之和，工程所需的时间是各任务时间耗费之和。（提示：可设$cost(i,j)$为在j时间内完成任务i到n的最小代价）。算法的复杂性是多少?

*20. 某一机器中有n个零件。每个零件有三个供应商，来自供应商j的零件i的重量为$W_{i,j}$，其价格为$C_{i,j}$（$1 \le j \le 3$）。机器的价格等于所有零件价格之和，其重量也为各零件重量之和。设计一个动态规划算法，以决定在总价格不超过C的条件下，从哪些供应商购买零件能组成最轻的机器。（提示：可设$w(i,j)$为价格低于j时由零件i到n组成的最轻机器）。算法的复杂性是多少?

*21. 定义$w(i, j)$ 为价格低于j 时由零件1到i 组成的最轻机器，完成练习2 0。
*22. 串s 为串a 中去掉某些字符而得到的子串。如串“ $o\ n\ i\ o\ n$”为串“$r\ e\ c\ o\ g\ n\ i\ t\ i\ o\ n$”的子串。当且仅当串$s$ 既是a 的子串又是b 的子串时，串s 是串a 和串b 的公共子串。串s 的长度指其所含的字符数。试用动态规划算法得到串a 和串b 的最长公共子串。（提示：设$a=a_1a_2.a_n$，$b=b_1b_2.b_m$。定义$l(i, j)$ 为串$a_i.a_n$ 和$b_j.b_m$ 最长公共子串的长度）。算法的复杂性是多少?
*23. 若$l(i,j)$定义为串$a_1a_2.a_i$ 和$b_1b_2.b_j$ 的最长公共子串的长度，重做练习2 2。
*24. 在串编辑问题中，给出两个串$a=a_1a_2.a_n$ 和$b=b_1b_2.b_m$ 及三个耗费函数C，D和I。其中$C(i, j)$ 为将a_i 改为b_j 的耗费，$D(i)$ 为从a 中删除a_i 的耗费，$I(i)$为将b_i 插入a 中的耗费。通过修改、删除和插入操作可把串a 改为串b。如，可删除所有a_i，然后插入所有b_i；或者当$n\geq m$ 时，可先把a_i 变成b_i（$1\leq i\leq n$），然后删除其余的a_i。整个操作序列的耗费为各个操作的耗费之和。设计一个动态规划算法来确定一个具有最少耗费的编辑操作序列。（提示：定义$c(i, j)$ 为将$a_1a_2.a_i$ 转变为$b_1b_2.b_j$ 的最少耗费）。算法的复杂性是多少?

（ 说明：本资料是根据《数据结构、算法与应用》（美，Sartaj Sahni著）一书第13-17章编辑、改写的。考虑到因特网传输速度等因素，大部分插图和公式不得不被删除。对于内容不连贯之处，请网友或读者参阅该书，敬请原谅。 ）