算法设计与分析

第 3 章 动态规划

金英　编写

计算机科学技术学院
软件学院

2021 年 3 月

目　录

第 3 章 动态规划法（Dynamic Programming）

3.1 基本思想

动态规划法与分治法类似，也是将要求解的问题一层一层地分解成一级一级、规模逐步缩小的子问题，直到可以直接求解其解的子问题为止。所有子问题按层次关系构成一棵子问题树。树根是原问题。原问题的解依赖于子问题树中所有子问题的解。

与分治法不同的是，子问题往往不是相互独立的。动态规划法所针对的问题有一个显著的特征，即它所对应的子问题树中的子问题呈现大量的重复。因此动态规划法的相应特征是，对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解。

设原问题的规模为 n，容易看出，当子问题树中的子问题总数是 n 的超多项式函数，而不同的子问题数只是 n 的多项式函数时，动态规划法显得特别有意义。

动态规划法通常用于求一个问题在某种意义下的最优解。适合采用动态规划方法的优化问题必须具备最优子结构性质和子问题重叠性质。

当一个问题的优化解包含了子问题的优化解时，则称该问题具有优化子结构性质。在求解一个问题的过程中，很多子问题的解被多次调用，则称该问题具有子问题的重叠性质。

设计一个动态规划算法，通常可按以下几个步骤进行：

⑴定义子问题（用参数表达子问题的边界）。

⑵分析最优解的性质，并刻画其结构特征。

⑶定义最优解的代价（每个最优解都有一个值，称最优值，也称代价——最大值或最小值；目标函数、优化函数）。

⑷列出关于优化函数的递推式和边界条件（即递推式的初值）。

⑸根据递推式分解子问题，直到不可分为止。

⑹自底向上计算最优值（迭代实现），以备忘录方式（表格）存储，并记录构造最优解所需的信息。

⑺根据计算最优值时得到的信息，构造一个最优解。

步骤⑴~⑹是动态规划算法的基本步骤。在只需要求出最优值的情形，步骤⑺可以

省去。若需要求出问题的一个最优解，则必须执行步骤⑺。此时，在步骤⑹中计算最优值时，通常需要记录更多的信息，以便在步骤⑺中，根据所记录的信息，快速地构造出一个最优解。

下面我们以具体的例子来说明如何运用动态规划算法的设计思想，并分析可以用动态规划算法求解的问题所应具备的一般特征。

3.2 最长公共子序列问题（Longest-Common-Subsequence, LCS）

一个给定序列的子序列是在该序列中删去若干元素后得到的序列。

例：X=<A,B,C,B,D,B>

Z=<B,C,D,B>是 X 的子序列。

W=<B,D,A>不是 X 的子序列。

如果序列 Z 是序列 X 的子序列，也是序列 Y 的子序列，则 Z 是序列 X 与 Y 的公共子序列。

例：X=<A,B,C,B,D,A,B>

Y=<B,D,C,A,B,A>

Z=<B,C,A>是 X 与 Y 的公共子序列。

最长公共子序列（LCS）问题是给定两个序列 $X = <x_1,x_2,\ldots,x_m>$ 和 $Y = <y_1,y_2,\ldots,y_n>$，要求找出 X 和 Y 的一个最长公共子序列。

例：X=<A,B,C,B,D,A,B>

Y=<B,D,C,A,B,A>

Z=<B,C,A>是 X 与 Y 的一个公共子序列。

L=<B,C,B,A>是 X 与 Y 的一个最长公共子序列。因为 X 与 Y 没有长度大于 4 的公共子序列。

V=<B,C,A,B>也是 X 与 Y 的一个最长公共子序列（即 LCS 不唯一!）。

下面用动态规划方法解最长公共子序列问题，计算步骤如前所述。

⑴ 最长公共子序列（LCS）结构分析

第 i 前缀：设 $X=<x_1,x_2,\ldots,x_m>$是一个序列，X 的第 i 前缀 X_i 是一个序列，定义为 $X_i=<x_1,x_2,\ldots,x_i>$。约定 X_0 是个空序列。

定理（LCS 的最优子结构性质）：

设 $X=<x_1,x_2,\ldots,x_m>$和 $Y=<y_1,y_2,\ldots,y_n>$是两个序列，$Z=<z_1,z_2,\ldots,z_k>$是 X 与 Y 的 LCS，则

1）若 $x_m=y_n$，则 $z_k=x_m=y_n$，且 Z_{k-1} 是 X_{m-1} 是 Y_{n-1} 的 LCS。

2）若 $x_m\neq y_n$，且 $z_k\neq x_m$，则 Z 是 X_{m-1} 和 Y 的 LCS。

3）若 $x_m\neq y_n$，且 $z_k\neq y_n$，则 Z 是 X 和 Y_{n-1} 的 LCS。

证明：用反证法证明。

1）设 $z_k\neq x_m$，则$<z_1,z_2,\ldots,z_k,x_m>$是 X 和 Y 的长度为 k+1 的公共子序列。这与 Z 是 X 与 Y 的 LCS 矛盾。$\therefore z_k=x_m=y_n$。

现在证明 Z_{k-1} 是 X_{m-1} 与 Y_{n-1} 的 LCS。显然 Z_{k-1} 是 X_{m-1} 与 Y_{n-1} 的长度为 k-1 的公共子序列，我们需要证明 Z_{k-1} 是 X_{m-1} 与 Y_{n-1} 的 LCS。若 X_{m-1} 与 Y_{n-1} 有一个长度大于 k-1 的公共子序列 W，则将 x_m 加在其尾部将产生 X 与 Y 的一个长度大于 k 的公共子序列，矛盾。所以，Z_{k-1} 是 X_{m-1} 与 Y_{n-1} 的 LCS。

2）由于 $z_k\neq x_m$，那么 Z 是 X_{m-1} 和 Y 的一个公共子序列。若 X_{m-1} 和 Y 有一个长度大于 k 的公共子序列 W，则 W 也是 X 与 Y 的长度大于 k 的公共子序列。这与 Z 是 X 与 Y 的 LCS 矛盾。$\therefore$Z 是 X_{m-1} 和 Y 的 LCS。

3）与 2）类似。

这个定理告诉我们，两个序列的最长公共子序列包含了这两个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有最优子结构性质。

由递归结构容易看出 LCS 问题具有子问题的重叠性，如图 3-1。

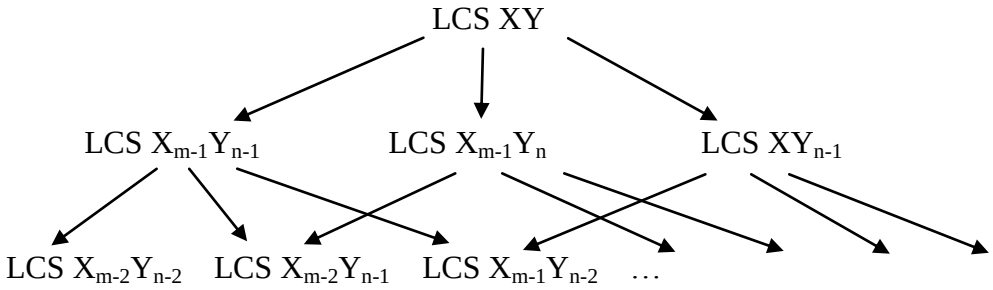


图 3-1　LCS 问题的子问题重叠性

因此可用动态规划算法解此问题（此问题可用穷举法求解，但需要指数时间）。

⑵ 建立求解 LCS 长度的递推关系式

令 c[i][j]=X_i 与 Y_j 的 LCS 的长度。

LCS 长度的递归方程如下：

$$c[i][j]=\begin{cases}0 & \text{当}i=0\text{或}j=0\text{时}\\ c[i-1][j-1]+1 & \text{当}i,j>0\text{且}x_i=y_i\text{时}\\ \max(c[i][j-1],c[i-1][j]) & \text{当}i,j>0\text{且}x_i\neq y_i\text{时}\end{cases}$$

⑶ 分解子问题

在分解子问题时，根据 c[i][j]的递归方程自顶向下分解子问题，直到不可分为止。例如，对于计算 X_3 与 Y_4 的最长公共子序列问题，根据上述原则分解出的子问题空间如图 3-2 所示。

c[0][0]	c[0][1]	c[0][2]	c[0][3]	c[0][4]
c[1][0]	c[1][1]	c[1][2]	c[1][3]	c[1][4]
c[2][0]	c[2][1]	c[2][2]	c[2][3]	c[2][4]
c[3][0]	c[3][1]	c[3][2]	c[3][3]	c[3][4]

图 3-2 计算 X_3 与 Y_4 的最长公共子序列问题的子问题空间

⑷ 自底向上计算 LCS 长度

根据 c[i][j]的递归定义式，容易写一个递归程序来计算 c[m][n]。简单地递归计算将耗费指数计算时间。然而，不同的子问题个数只有 $\Theta(mn)$个。由此可见，在递归计算时，许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。

用动态规划算法解此问题，可依据递推式以自底向上的方式进行计算，在计算过程中，保存已解决的子问题答案，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，最终得到多项式时间的算法。下面要给出的计算 c[i][j]的动态规划算法涉及到的数据结构如下：

c[0:m,0:n]: c[i][j]是 X_i 和 Y_j 的的 LCS 的长度。

b[1:m,1:n]: b[i][j]指向计算 c[i][j]时所选择的子问题的优化解所对应的 c 表的表项，这个表项对应于在计算 c[i][j]时所选择的最优子问题的解（由表 b 构造 X 与 Y 的一个 LCS）。

计算 LCS 长度的算法如下：

```
void LCSLength(int m, int n, char *x, char *y, int **c, Type **b)
  { int i, j;
    for(i=0; i<=m; i++) c[i][0]=0;
    for(j=0; j<=n; j++) c[0][j]=0;
```

```
for(i=1; i<=m; i++)
    for(j=1; j<=n; j++)
        if (x[i]==y[j])
            { c[i][j]= c[i-1][ j-1]+1;      b[i][j]='↖';}
         else   if (c[i-1][j]>= c[i][j-1])
                    { c[i][j]= c[i-1][j];      b[i][j]='↑';}
                else   { c[i][j]= c[i][j-1];      b[i][j]='←';}
}
```

⑸ 构造 LCS（构造优化解）

b[i][j]是标记，记录 X_i 和 Y_j 的最长公共子序列的元素是怎样选取的。有三种标记，用“↖”表示序列 X_i 和 Y_j 的最后一项 $x_i=y_j$，已被选入最长公共子序列；用“↑”表示在序列 X_i 和 Y_j 的最长公共子序列选择时不考虑 x_i，它们的最长公共子序列就是 X_{i-1} 和 Y_j 的最长公共子序列；用“←”表示在序列 X_i 和 Y_j 的最长公共子序列选择时不考虑 y_j，它们的最长公共子序列就是 X_i 和 Y_{j-1} 的最长公共子序列。设立这些标记是为了追踪解。

基本思想：

- 从 b[m][n]开始按指针搜索（追踪）；
- 若 b[i][j]＝“↖”，则 $x_i= y_j$ 是 LCS 的一个元素（如果遇到其他两种标记，表示没有元素加入最长公共子序列）；
- 如此找到的序列是 X 与 Y 的 LCS 的反序。

构造最长公共子序列算法：

```
void LCS(int i, int j, char *x, Type **b)
{ if( (i==0) || (j==0) )    return ;
  if (b[i][j]=='↖')
        { LCS(i-1, j-1, x, b);    cout<< x[i];}
  else   if (b[i][j]=='↑')    LCS(i-1, j, x, b);
   else                        LCS(i, j-1, x, b);
}
```

例：设 X=<A,B,C,B,D,A,B>, Y=<B,D,C,A,B,A>

按算法 LCSLength 算法求得的 c 矩阵与 b 矩阵如图 3-3 所示。

依据 LCS 算法，从 b[7][6]开始按指针搜索；得到 X 与 Y 的最长公共子序列<B,C,B,A>。

⑹ 算法复杂性分析

- **时间复杂性分析**

计算代价的时间：i, j 两层循环，i 循环 m 步，j 循环 n 步。由于每个数组单元的计算耗费 O(1) 时间，算法 LCSLength 耗时 O(mn)。

构造最优解的时间：算法 LCS 中，每一次递归调用使 i 和 j 同时减 1，或者 i 减 1，j 减 1，总之 i+j 至少减 1。由于初始 i+j=m+n，因此至多 m+n 次在标记上的操作，于是算法的计算时间为 O(m+n)。

∴T(n)= O(mn)。

- **空间复杂性**

使用了二维数组 b 和 c，∴S(n)= O(mn)。

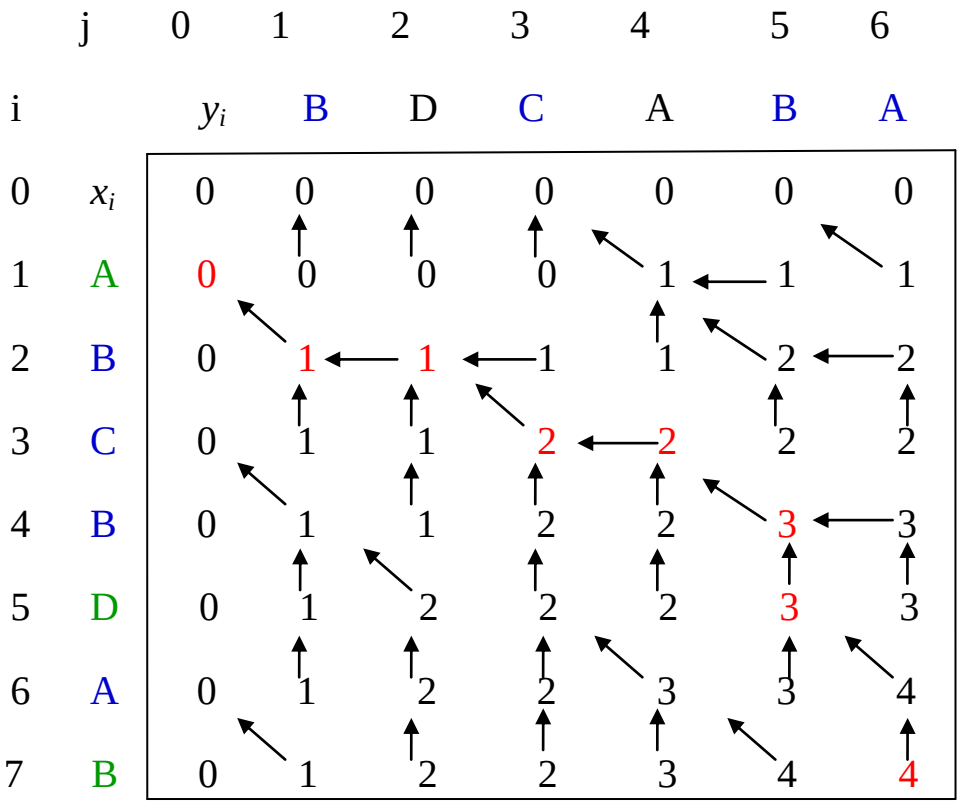


图 3-3　c 矩阵与 b 矩阵的叠加

3.3 矩阵连乘最佳计算次序问题

矩阵连乘积问题是给定 n 个矩阵{A_1，A_2，…，A_n}。其中 A_i 和 A_{i+1} 是可乘的，i =1，2，…，n-1。要求这 n 个矩阵连乘积 $A_1A_2…A_n$ 具有最小代价的计算次序（矩阵乘法的

代价/复杂性：数乘次数或标量乘法的次数）。

由于矩阵乘法满足结合律，故连乘积的计算可以有许多不同的计算次序。这种计算次序可以用加括号的方式来确定。若一个矩阵连乘积的计算次序已完全确定，也就是说该连乘积已完全加括号，则我们可以通过反复调用两个矩阵相乘的标准算法计算出矩阵连乘积。完全加括号的矩阵连乘积可以递归地定义为：

⑴ 单个矩阵是完全加括号的；

⑵ 若矩阵连乘积 A 是完全加括号的，则 A 可表示为两个完全加括号的矩阵连乘积 B 和 C 的乘积并加括号，即 A=(BC)。

例如，矩阵连乘积 $A_1A_2A_3A_4$ 可以有以下 5 种不同的完全加括号方式：

（A_1（A_2（A_3A_4））），

（A_1（（A_2A_3）A_4））），

（（A_1A_2）（A_3A_4））），

（（A_1（A_2A_3））A_4），

（（（A_1A_2）A_3）A_4）。

每一种完全加括号的方式对应于一种矩阵连乘积的计算次序，而这种计算次序与计算矩阵连乘积的计算量有着密切的关系。

首先，我们来看计算两个矩阵乘积所需的计算量。若 A 是一个 $p\times q$ 矩阵，B 是一个 $q\times r$ 矩阵，则其乘积 C=AB 是一个 $p\times r$ 矩阵。在计算 C 时总共需要 pqr 次数乘。

为了说明在计算矩阵连乘积时加括号方式对整个计算量的影响，我们来看一个计算 3 个矩阵{A_1，A_2，A_3}的连乘积的例子。设这 3 个矩阵的维数分别为 10×100，100×5，和 5×50。若按第一种加括号方式（（A_1A_2）A_3）来计算，总共需要 $10\times100\times5+10\times5\times50=7500$ 次的数乘（标量乘法运算）。若按第二种加括号的方式（A_1（A_2A_3））来计算，则需要的数乘次数为 $100\times5\times50+10\times100\times50=75000$。第二种加括号的方式是第一种加括号方式的计算量的 10 倍。由此可见，在计算矩阵连乘积时，加括号方式，即计算次序对计算量有很大影响。

因而人们提出了矩阵连乘积的最优计算次序问题，即对于给定的相继 n 个矩阵{A_1，A_2，…，A_n}（设其中 A_i 的维数为 $p_{i-1}\times p_i$，i＝1，2，…，n），如何确定计算矩阵连乘积 $A_1A_2\ldots$，A_n 的一个计算次序（完全加括号方式），使得依此次序计算矩阵连乘积需要

的数乘次数最少。

解矩阵连乘积的最优计算次序问题最容易想到的方法是穷举搜索法。也就是列出所有可能的计算次序，并计算出每一种计算次序相应需要的计算量，然后找出较小者。然而，这样做计算量太大。事实上，对于 n 个矩阵的连乘积，设有 P(n)个不同的计算次序。由于我们可以首先在第 k 个和第 k+1 个矩阵之间将原矩阵序列分为两个矩阵子序列，k=1,2,…,n-1；然后分别对这两个矩阵子序列完全加括号；最后对所得的结果加括号，得到原矩阵序列的一种完全加括号方式。关于 P(n)的递归式如下：

$$p(n)=\begin{cases}1 & \text{当}n=1\text{时}\\ \sum_{k=1}^{n-1}P(k)P(n-k) & \text{当}n\geq 2\text{时}\end{cases}$$

解此递归方程可得，P(n)实际上是卡特兰（Catalan）数，即 p(n)=C(n-1)，其中，

$$C(n)=\frac{1}{n+1}\binom{2n}{n}=\Omega(4^n/n^{3/2})$$（根据 Stirling 公式）

也就是说，P(n)随着 n 的增长是指数增长的。因此，穷举搜索法不是有效算法。

下面用动态规划方法解矩阵连乘积的最优计算次序问题，计算步骤如前所述。

⑴ 定义子问题、分析最优解的结构

设计求解具体问题的动态规划算法的第一步是刻画该问题的最优解结构特征。对于矩阵连乘积的最优计算次序问题也不例外。首先，为方便起见，将矩阵连乘积 $A_iA_{i+1}\cdots A_j$ 简记为 A[i:j]。我们来看计算 A[1:n]的一个最优次序。设这个计算次序在矩阵 A_k 和 A_{k+1} 之间将矩阵链断开，$1\leq k<n$，则完全加括号方式为（（$A_1\cdot\cdot A_k$）（$A_{k+1}\cdot\cdot A_n$））。照此，我们要先计算 A[1:k]和 A[k+1:n]，然后，将所得的结果相乘才得到 A[1:n]。显然其总计算量为计算 A[1:k]的计算量加上计算 A[k+1:n]的计算量，再加上 A[1:k]和 A[k+1:n]相乘的计算量。

这个问题的一个关键特征是：计算 A[1:n]的一个最优次序所包含的计算 A[1:k]和 A[k+1:n]的次序也是最优的。事实上，若有一个计算 A [1:k]的次序需要的计算量更少，则用此次序替换原来计算 A[1:k]的次序，得到的计算 A[1:n]的次序需要的计算量将比最优次序所需计算量更少，这是个矛盾。同理可知，计算 A[1:n]的一个最优次序所包含的计算矩阵子链 A[k+1:n]的次序也是最优的。

因此， 矩阵连乘积计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。一个问题的最优子结构性质是该问题可用动态规划算法求解的显著特

征。

⑵ 建立递推关系式

设计一个动态规划算法的第二步是递归地定义最优值。对于矩阵连乘积的最优计算次序问题，设计算 A[i: j]，1≤i≤j≤n，所需的最少数乘次数为 m[i][j]，则原问题的最优值为 m[1][n]。

当 i=j 时，A [i:j]=A_i 为单一矩阵，无需计算，因此 m[i][i]=0，i=1,2,…,n。

当 i<j 时，可利用最优子结构性质来计算 m[i][j]。事实上，若计算 A[i:j]的最优次序在 A_k 和 A_{k+1} 之间断开，i≤k<j，则 m[i][j]= m[i][k]+ m[k+1][j]+ $p_{i-1}\,p_k\,p_j$。

由于在计算时并不知道断开点 k 的位置，所以 k 还未定。不过，k 的位置只有 j-i 个可能，即 k∈{i,i+1,…,j-1}。因此，k 是这 j-i 个位置中计算量达到最小的那一个位置。从而，m[i][j]可以递归地定义为：

$$m[i][j]=\begin{cases}0 & \text{当} i=j\\ \min\limits_{i\le k<j}\{m[i][k]+m[k+1][j]+p_{i-1}p_kp_j\} & \text{当} i<j\end{cases}$$

m[i][j]记录计算 A [i:j]所需的最少数乘次数，即最优值。在计算最优值的同时还要确定计算 A [i:j]的最优次序中的断开位置 k，也就是说， 若 m[i][j]= m[i][k]+ m[k+1][j]+ $p_{i-1}\,p_k\,p_j$，则将对应于 m[i][j]的最佳断开位置 k 记录在 s[i][j]中，以便在第⑸步可以据此构造出最优解。

⑶ 分解子问题

在分解子问题时，根据 m[i][j]的递归方程自顶向下分解子问题，直到不可分为止。例如，对于计算矩阵连乘积 $A_1A_2…A_5$ 的最佳计算次序问题，根据上述原则分解出的子问题空间如图 3-4 所示。

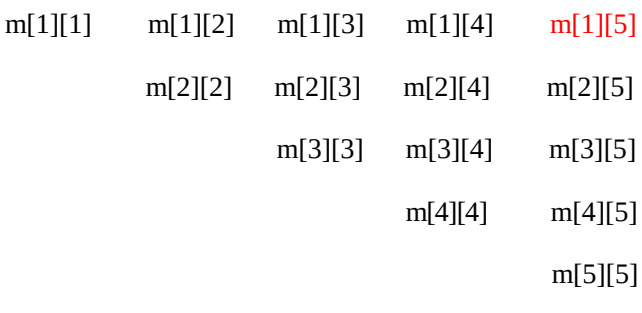


图 3-4 计算矩阵连乘积 $A_1A_2…A_5$ 问题的子问题空间

⑷ 计算最优值

根据 m[i][j]的递归定义式，容易写一个递归程序来计算 m[1][n]。稍后将看到，简单

地递归计算将耗费指数计算时间。然而，我们注意到，在递归计算过程中，不同的子问题个数只有$\Theta(n^2)$个。事实上，对于 $1\le i\le j\le n$ 不同的有序对(i,j)对应于不同的子问题。因此，不同子问题的个数最多只有$\binom{n}{2}+n=\Theta(n^2)$个。由此可见，在递归计算时，许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。

用动态规划算法解此问题，可依据递归式以自底向上的方式进行计算，在计算过程中，保存已解决的子问题答案，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，最终得到多项式时间的算法。下面将给出计算 m[i][j]的动态规划算法 MatrixChain。

输入：序列 $P=\{p_0, p_1, \ldots, p_n\}$

输出：除了每个子问题 A [i:j]的最优值 m[i][j]外，还有使 m[i][j]= m[i][k]+ m[k+1][j]+ $p_{i-1}\ p_k\ p_j$ 达到最优的断开位置 k（s[i][j] =k, $1\le i\le j\le n$）。

算法描述如下：

```
void MatrixChain(int *p, int n, int **m, int **s)
{ for (int i=1; i<=n; i++) m[i][i]=0;    //长度为 1 的链的最小代价
  for(int r=2; r<=n; r++)                //长度为 r 的链的最小代价
     for( int i=1; i<=n-r+1; i++)        // n-r+1 为对角线的长度
        { int j=i+r-1;                   //长度为 r 的链上第一个最优值 m[i][j]的列标
          m[i][j] = m[i+1][j]+p[i-1]*p[i]*p[j];     //省略了 m[i][i]
          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; }
              }
        }
}
```

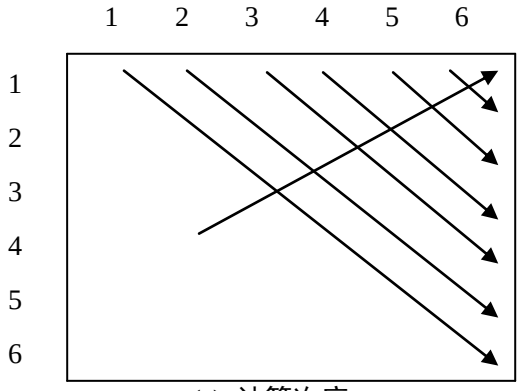
此算法首先计算出 m[i][i]=0，i=1,2,…,n，然后，根据递归式，让矩阵链长递增，依次计算 m[i][i+1]，i=1,2,…,n-1（矩阵链长度为 2，即 r=2 时）；m[i][i+2]，i=1,2,…,n- 2（矩

阵链长度为 3，即 r=3 时）；…。在计算 m[i][j]时，只用到已计算出的 m[i][k]和 m[k+1][j]。

例：确定计算矩阵连乘积 $A_1A_2A_3A_4A_5A_6$ 的最优次序。其中各个矩阵的维数为：

A_1	A_2	A_3	A_4	A_5	A_6
30×35	**35×15**	**15×5**	**5×10**	**10×20**	**20×25**

计算 m[i][j]先后次序如图 3-5(a)所示，计算结果 m[i][j]和 s[i][j]，1≤i≤j≤n，分别如图 3-5(b)和 3-5(c)所示。



(a) 计算次序

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) m 矩阵

	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

(c) S 矩阵

图 3-5 计算 m[i][j]（包括 s[i][j]）的次序

例如，在计算 m[2][5]时，依 m[i][j]的递归式有：

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

$$=7125$$

对应的最佳断开点 k=3，因此，s[2][5]＝3。

MatrixChain 算法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)，而三重循环的总次数为 $O(n^3)$。因此，该算法的计算时间上界为 $O(n^3)$。算法所占用的空间显然为 $O(n^2)$。由此可见，动态规划算法比穷举搜索法要有效的多。

⑸ 构造最优解

动态规划算法的第⑸步是构造问题的一个最优解。算法 MatrixChain 只是计算出了最优值，并未给出最优解。也就是说，通过 MarixChain 的计算，我们只知道计算给定的矩阵连乘积所需的最少数乘次数，还不知道具体应按什么次序来做矩阵乘法才能达到数乘次数最少。

然而，MatrixChain 已记录了构造一个最优解所需要的全部信息。事实上，s[i][j]中的数 k 告诉我们计算矩阵 A[i:j]的最佳方式应在矩阵 A_k 和 A_{k+1} 之间断开，即最优的加括号方式应为(A[i:k]) (A[k+1:j])。因此，从 s[1][n]记录的信息可知计算 A[1:n]的最优加括号方式为 (A[1:s[1][n]]) (A[s[1][n]+1:n]) 。而 A[1:s[1][n]] 的最优加括号方式为 (A[1:s[1][s[1][n]]]) (A[s[1][s[1][n]]+1: s[1][n])。同理可以确定计算 A[s[1][n]+1: n]的最优加括号方式为(A[s[1][n]+1: s[s[1][n]+1][n]]) (A[s[s[1][n]+1][n]+1: n])。

下面的 Traceback 算法按 MatrixChain 算法计算出的断点矩阵 s 指示的加括号方式输出计算 A[i:j]的最优计算次序。

```
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;
}
```

要输出 A[1:n]的最优计算次序只要调用 Traceback(1, n ,s)即可。对于上面所举的例子，通过调用 Traceback(1, 6, s)，即可输出最优计算次序（（A_1（A_2 A_3））（（A_4 A_5）A_6））计算出所要求的连乘积 $A_1A_2A_3A_4A_5A_6$。

构造最优解的时间为 O(n)。

3.4 最大子段和问题

在第 2 章我们用分治法求解了最大子段和问题，因此对最大子段和问题就不在这里赘述了。下面将按动态规划法求解最大子段和问题。

⑴ 定义子问题并定义优化函数

定义 $b[j]$是子段 $a_1, a_2, \ldots, a_j$ 中必须包含最后元素 a_j 的最大子段和，即

$$b[j]=\max_{1\le i\le j}\sum_{k=i}^{j}a[k]\text{，}\quad 1\le j\le n$$

⑵ 分析问题最优解的结构

$b[j]$可以通过 $b[j\text{-}1]$直接得到。因为如果 $a_1, a_2, \ldots, a_j$ 的子段 $a_i, a_{i+1}, \ldots, a_j$ 是使得 $b[j]$达到最大和的子段，那么 $a_i, a_{i+1}, \ldots, a_{j-1}$ 一定是使得 $b[j\text{-}1]$达到最大和的子段。如若不然，存在一个使得 $b[j\text{-}1]$达到更大和的子段 $a_t, a_{t+1}, \ldots, a_{j-1}$，那么在 $a_t, a_{t+1}, \ldots, a_{j-1}$ 后面附上 a_j 所得到的子段 $a_t, a_{t+1}, \ldots, a_{j-1}, a_j$ 之和将大于 $b[j]$。这与 $b[j]$是 $a_1, a_2, \ldots, a_j$ 以元素 a_j 作为最后元素的最大子段和矛盾。这恰好验证了这样定义的优化函数满足最优子结构性质。

⑶ 建立递推关系式

根据以上分析，在考虑怎样选择才能使得 $b[j\text{-}1]$加到 a_j 上？而这取决于 $b[j\text{-}1]$是否大于 0。即

$$b[j]=\begin{cases}b[j-1]+a[j] & b[j-1]>0\\ a[j] & b[j-1]\le 0\end{cases}$$

于是，可得到下面计算最优值 $b[j]$的动态规划递推关系式

$$b[j]=\begin{cases}0 & j=0\\ \max\{b[j-1]+a[j],a[j]\} & 1\le j\le n\end{cases}$$

⑷ 计算最优值

因为所求最大子段和为

$$\max_{1\le i\le j\le n}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}\max_{1\le i\le j}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}b[j]$$

所以，$b[0], b[1], b[2], \ldots, b[n]$恰好枚举了以任何元素为最后元素的所有子段的最大和。显然，要找的那个具有最大和的子段一定在里面，只要对 $n+1$ 个和 $b[j]$进行比较，一定可以找到其中的最大和，这才是问题所要求的最大和。

根据上述递推关系，可设计出求最大子段和的动态规划算法如下：

```
int besti = 0, bestj = 0;     // 分别记录最大子段的开始下标和结束下标
int MaxSum(int n, int *a)
{   int sum = 0, b = 0;
    for (int j =1; j <= n; j++)
        {   if (b > 0) b += a[j];
            else {b = a[j]; i = j;}
```

```
            if (b > sum)
                {sum = b; besti = i; bestj = j;}
        }
    return sum;
}
```

上述算法的时间复杂性 T(n)=O(n)，空间复杂性 S(n)=O(1)。

例 用动态规划法求 a=(-2, 11, -4, 13, -5, -2)的最大子段和。

设第 j 个子问题的最优值记在 $b[j]$中，根据 $b[j]$的递归方程求出的 $b[j]$ $(0 \le j \le n)$， 最大子段和 sum、最大子段起始下标 besti、最大子段终止下标 bestj 见图 3-6。

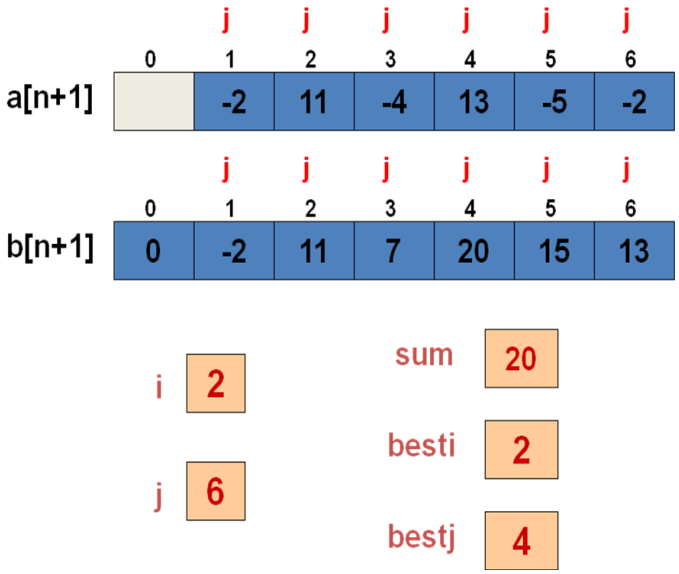


图 3-6 求最大子段和过程示意图

3.5 0-1 背包问题

0-1 背包问题：给定 n 种物品和一个背包。物品 i 的重量是 w_i，价值 v_i，背包承重为 C。每种物品只能选择完全装入背包或不装入，一个物品至多装入一次，因此该问题称为 0-1 背包问题。问如何选择装入背包的物品，使装入背包中的物品的总价值最大？

0-1 背包问题形式化描述为：

输入：整数 $C>0$，整数 $w_i>0$，$v_i>0$，$1 \le i \le n$

输出：$(x_1, x_2, \ldots, x_n)$，$x_i \in \{0, 1\}$，满足 $\sum_{i=1}^{n} w_i x_i \le C$，且使 $\sum_{i=1}^{n} v_i x_i$ 最大。

0-1 背包问题等价 0-1 规划问题（整数规划问题，属于组合优化问题），目标函数和

约束条件是：

目标函数 $\max\sum_{i=1}^{n} v_i x_i$

约束条件 $\begin{cases}\sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n\end{cases}$

⑴ 定义子问题、优化解的结构分析

① 划分子问题

背包问题有两个参数，物品种类和背包重量的限制。可以根据物品种类和背包的重量限制进行子问题划分。

不妨假设已经对物品 1, 2, …, *i*-1 做出决策 $x_1, x_2, \ldots, x_{i-1}$，$x_k \in \{0,1\}, 1 \le k \le i-1$。背包中物品重量 $\sum_{k=1}^{i-1} w_k x_k$，价值 $\sum_{k=1}^{i-1} v_k x_k$，背包余下容量 $C_i = C - \sum_{k=1}^{i-1} w_k x_k$。接下来问题就变为关于物品 *i*, *i*+1, …, *n* 而背包容量为 C_i 的 **0-1** 背包问题了，即

$$\max \sum_{k=i}^{n} v_k x_k$$

$$\begin{cases}\sum_{k=i}^{n} w_k x_k \le C_i \\ x_k \in \{0,1\}, i \le k \le n\end{cases}$$

可以将这个子问题简记为

$$P_i = [\{i, i+1, \ldots, n\}, C_i = C - \sum_{k=1}^{i-1} w_k x_k]$$

如第一次划分得到的子问题就是 $P_2 = [\{2,3,\ldots,n\}, C_2 = C - w_1 x_1]$，也就是

$$\max \sum_{k=2}^{n} v_k x_k$$

$$\begin{cases}\sum_{k=2}^{n} w_k x_k \le C - w_1 x_1 \\ x_k \in \{0,1\}, 2 \le k \le n\end{cases}$$

0-1 背包问题的子问题划分方式如图 3-7 所示。

② 问题的最优子结构性质

为了确定 **0-1** 背包问题具有最优子结构性质，我们先证明下述定理。

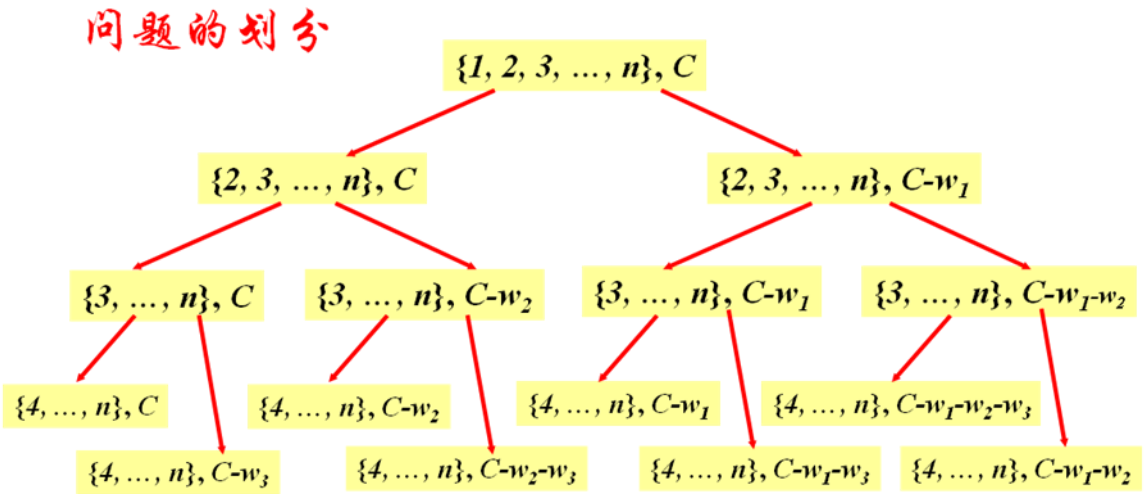


图 3-7 背包问题的子问题划分示意图

定理: 如果 $S_i=(y_i, y_{i+1}, \ldots, y_n)$是 **0-1** 背包子问题 $P_i=[\{i,i+1,\ldots,n\},C_i=C-\sum_{k=1}^{i-1}w_kx_k]$ 的优化解，则 $S_{i+1}=(y_{i+1}, \ldots, y_n)$是子问题 P_{i+1} 的优化解，即为如下问题的优化解

$$\max \sum_{k=i+1}^{n} v_k x_k$$

$$\begin{cases} \sum_{k=i+1}^{n} w_k x_k \le C_i - w_i y_i \\ x_k \in \{0,1\}, i+1 \le k \le n \end{cases}$$

证明: 如果 $S_{i+1}=(y_{i+1}, \ldots, y_n)$不是子问题 P_{i+1} 的优化解，设 $S'_{i+1}=(z_{i+1}, \ldots, z_n)$是 P_{i+1} 的优化解。则 $\sum_{k=i+1}^{n} v_k z_k > \sum_{k=i+1}^{n} v_k y_k$，且 $w_i y_i + \sum_{k=i+1}^{n} w_k z_k \le C_i$。因此

$$v_i y_i + \sum_{k=i+1}^{n} v_k z_k > \sum_{k=i}^{n} v_k y_k$$

$$w_i y_i + \sum_{k=i+1}^{n} w_k z_k \le C_i$$

这说明 $S'_i=(y_i, z_{i+1}, \ldots, z_n)$是问题 P_i 之比 S_i 更优的解，从而 S_i 不是问题 P_i 的最优解，此为矛盾。

可见，**0-1** 背包问题具有最优子结构性质。

③ 问题的子问题重叠性

从 **0-1** 背包问题的子问题划分图示 3-8 看到，有许多子问题大量重复出现。**0-1** 背包问题具有子问题重叠性。

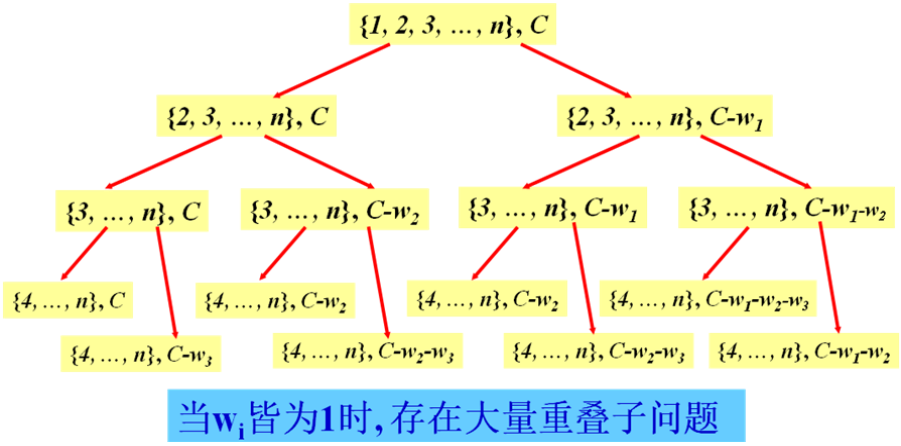


图 3-8　**0-1 背包问题的子问题重叠性**

⑵ 定义优化值（最优解的代价）并建立其递推关系式

0-1 背包问题需要确定 $x_1, x_2, \ldots, x_n$ 的值。假设按 $i=1, 2, \ldots, n$ 的次序来确定 x_i 的值，对应 n 次决策即 n 个阶段。

在决策 x_1 时问题处于以下两种状态：

① 背包不装入物品 1，则 $x_1=0$，背包不增加重量和价值，背包余下容量 C 不变，问题转变为相对于其余物品（即物品 $2, 3, \ldots, n$），背包容量仍为 C 的背包问题；

② 背包中装入物品 1，则 $x_1=1$，背包增加重量 w_1 和价值 v_1，背包余下容量 $C-w_1$，问题就变为关于物品 $2, 3, \ldots, n$ 而背包容量为 $C-w_1$ 的背包问题。

在决策 x_i 时问题处于以下两种状态：

① 背包不装入物品 i，则 $x_i=0$，背包不增加重量和价值，背包余下容量 j 不变，问题转变为相对于其余物品（即物品 $i+1, i+2, \ldots, n$），背包容量仍为 j 的背包问题；

② 背包中装入物品 i，则 $x_i=1$，背包增加重量 w_i 和价值 v_i，背包余下容量 $j-w_i$，问题就变为关于物品 $i+1, i+2, \ldots, n$ 而背包容量为 $j-w_i$ 的问题。

设所给 **0-1** 背包问题的子问题

$$\max \sum_{k=i}^{n} v_k x_k$$

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, i \le k \le n \end{cases}$$

的最优值为 $m[i][j]$，即 $m[i][j]=\max\sum_{k=i}^{n}v_k x_k$ 是背包容量为 j，可选物品为 $i, i+1, \ldots, n$ 时的 **0-1** 背包问题的最优值（即最优解$(x_i, x_{i+1}, \ldots, x_n)$对应的最优函数值$\sum_{k=i}^{n}v_k x_k$ ）。由 **0-1** 背包问题的最优子结构性质，可以建立计算最优值 $m[i][j]$的递推关系式如下：

$m[i][0]=0, 1\le i\le n$ （背包不能装入任何物品，总价值为 0）

$m[0][j]=0, 1\le j\le C$ （没有任何物品可装入，总价值为 0）

$$m[n][j]=\begin{cases}v_n & j\ge w_n\\ 0 & 0\le j<w_n\end{cases}$$ （只有物品 n，放下装入，放不下不装入）

$$m[i][j]=\begin{cases}\max\{m[i+1][j], m[i+1][j-w_i]+v_i\} & j\ge w_i\\ m[i+1][j] & 0\le j<w_i\end{cases}$$ i=n-1,n-2,…,1

（当 $0\le j<w_i$ 时，物品 i 放不下，$m[i][j]=m[i+1][j]$；否则 $j\ge w_i$，在不放入和放入物品 i 之间选最优值 $m[i][j]=\max\{m[i+1][j], m[i+1][j-w_i]+v_i\}$）

⑶ 递归地划分子问题

根据计算最优值 $m[i][j]$的递推关系自顶向下划分子问题，一直划分到原子问题为止。从 $m[1][C]$开始自顶向下划分子问题的过程如图 3-9 所示。

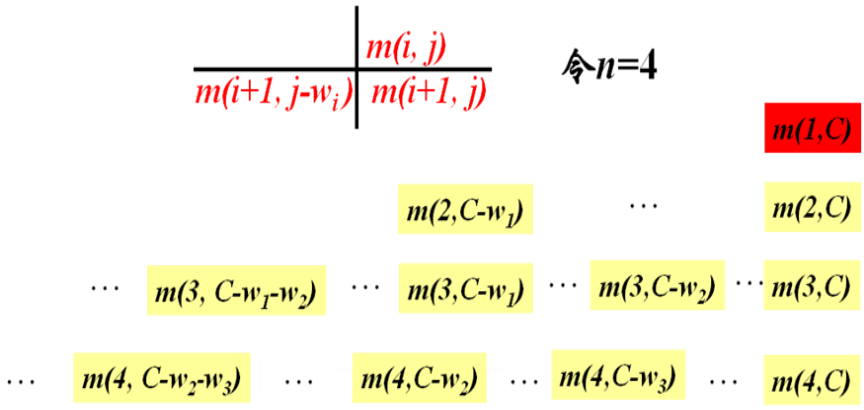


图 3-9 划分子问题过程示意图

⑷ 自底向上计算优化解的代价，记录优化解的构造信息

根据递推关系，自底向上从规模最小的子问题的最优解的代价开始计算，然后不断利用子问题的最优解代价构造更大问题最优解代价，直到计算出 $m[1][C]$为止，如图 3-10 所示。

⑸ 算法描述

基于以上讨论，当 $w_i(1 \le i \le n)$ 为正整数且 $w_i - 1 < C(1 \le i \le n)$ 时，可设计 **0-1** 背包问题的动态规划算法。

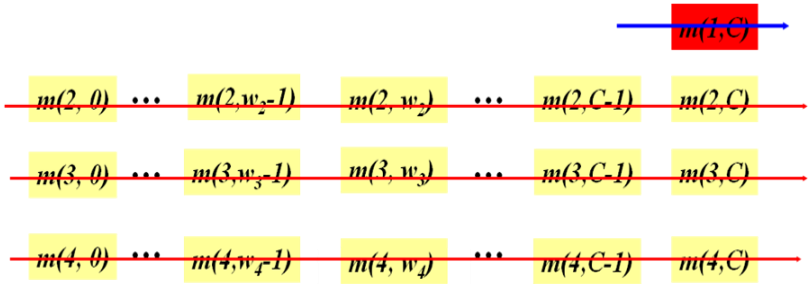


图 3-10 自底向上计算优化解的代价

① 计算代价的算法

```
void Knapsack(float v[], int w[], int C, int n, float m[n][])
{    for ( j = 0 ; j < w[n]; j++)
         m[n][j] = 0;
     for ( j = w[n]; j <= C; j++)
         m[n][j] = v[n];
     for ( i = n-1 ; i >= 2 ; i--)
       { for ( j = 0 ; j <= w[i] -1; j++)
             m[i][j] = m[i+1][j];
         for (j = w[i] ; j <=C ; j++)
             m[i][j] = max{m[i+1][j], m[i+1][j-w[i]]+v[i] };
       }
     if (C < w[1] ) m[1][C]=m[2][C];
     else          m[1][C]=max{m[2][C], m[2][C-w[1]]+v[1]};
}
```

② 构造优化解的算法

构造最优解的基本思想：

1. $m[1][C]$是最优解代价值，相应解计算如下：

if $m[1][C] = m[2][C]$

$x_1 = 0$;

else $x_1 = 1$;

2. 如果 x_1=0, 由 m[2][C]继续构造 x_2；

3. 如果 x_1=1, 由 m[2][C-w_1]继续构造 x_2；

……

算法描述：

```
void Traceback (float m[n][], int w[], int C, int n, int &x[])
{    for ( int i = 1; i < n; i++)
          if (m[i][C] == m[i+1][C])   x[i] =0；
          else                        x[i] =1；
     x[n] = m[n][C] ? 1 : 0 ;
}
```

⑹ 算法复杂性分析

① 时间复杂性

计算代价的时间=$O(nC)$

构造最优解时间=$O(nC)$

总时间复杂性为：$T(n)=O(nC)$

这是一个**伪多项式算法**!

当 $C=2^n$ 时: $T(n)=O(n2^n)$

当 w_i 不限定为正整数时: $T(n)=O(2^n)$

② 空间复杂性

使用二维数组 m，所以需要空间 $S(n)=O(n^2)$。

本章习题

1．给定两个序列 X=<A,B,B,D,A>，Y=<B,D,A,B>。采用动态规划策略求出其最长公共子序列。要求：

（1）根据 LCSlength 算法计算出表 c 和 b。

（2）根据表 b 计算出最长公共子序列。

（注：在 LCSLength 算法中，用 c[i][j]存储序列 X_i 和 Y_j 的最长公共子序列长度，并用 b[i][j]指向一个表项，这个表项对应于计算 c[i][j]时所选择的最优子问题的解。）

2. 设要计算矩阵连乘积 $A_1 A_2 A_3 A_4$

矩阵	A_1	A_2	A_3	A_4
矩阵的阶	3×2	2×4	4×3	3×2

采用动态规划策略完成下列任务：

(1) 请根据动态规划算法 MatrixChain 计算 m 矩阵和 s 矩阵（注：在该算法中用 m[i][j]记录计算矩阵链 $A_iA_{i+1}…A_j$ 所需的最小数乘次数，并用 s[i][j]记录计算 m[i][j]时取得最优代价处 k 的值）。

(2) 根据 m 矩阵和 s 矩阵来确定矩阵连乘积 $A_1A_2A_3A_4$ 的一个最佳计算次序及最少乘法次数。

3. 根据下述动态规划算法计算{3,-2,-1,9,-3,5}的最大子段和，并将计算过程中的相关数据填入表 1，最后给出最大子段和以及这个最大子段和的起始下标索引和终止下标索引。

计算最大子段和的动态规划算法：

```
int MaxSum（int n, int *a）// 假设上述序列中的数从下标 1 开始存入数组 a
{
int sum=0，i=0，besti=0，bestj=0，b[n+1]；
b[0]=0;
    for (int j=1; j<=n; j++)
    {
   if (b[j-1]>0)
        b[j] =b[j-1]+ a[j];
        else
         { b[j]=a[j];   i=j; }
          if (b[j] > sum)
         {sum=b[j]; besti=i; bestj=j;}
 }
     return sum;
}
```

表 1　用动态规划算法计算最大子段和的过程表

j	b[j]	i	j	besti	bestj	sum
0						

1						
2						
3						
4						
5						
6						

最大字段和 sum =_____；最大字段和起始下标=_____ ，终止下标=_____。

4. 设物品个数 n=3, 背包容量 C=8, 各物品的重量 W={6，5，3},各物品的价值 V={12，9，5}。设 m[i][j]是背包容量为 j，可选物品为 i,i+1,…,n 时问题最优的代价。用动态规划求解 0-1 背包问题，填写下列表格。

m[i][j]　j / i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
3									
2									
1									

5. 动态规划法为什么需要最优子结构性质和子问题的重叠性？

6. 输入表达式 $a_1\ O_1\ a_2\ O_2.....O_{n-1}\ a_n$，其中 a_i 是整数（$1\le i\le n$），$O_j\in\{+,-,\times\}$（$1\le i\le n-1$）。试设计一个动态规划算法，输出一个带括号的表达式（不改变操作数和操作符的次序），使得表达式的值达到最大。你的答案应包括：

(1) 用简明的语言表述这个问题的优化子结构；

(2) 根据优化子结构写出代价方程；

(3) 根据代价方程写出动态规划算法（伪代码）

(4) 写出最优解构造算法；

(5) 分析算法的时间复杂性。

7. 证明：如果 $S_i=(y_i, y_{i+1}, ..., y_n)$是 **0-1** 背包子问题 $P_i=[\{i,i+1,...,n\},C_i=C-\sum_{k=1}^{i-1}w_k x_k]$ 的优化解，则 $S_{i+1}=(y_{i+1}, ..., y_n)$是子问题 P_{i+1} 的优化解，即为如下问题的优化解

$$\max \sum_{k=i+1}^{n} v_k x_k$$

$$\begin{cases} \sum_{k=i+1}^{n} w_k x_k \le C_i - w_i y_i \\ x_k \in \{0,1\}, i+1 \le k \le n \end{cases}$$