第 3 章 动态规划

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划法求解的问题，经分解得到的子问题往往不是互相独立的。若用分治法解这类问题，则分解得到的子问题数目太多，以至于最后解决原问题需要耗费指数时间。然而，不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次。如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。为了达到这个目的，可以用一个表来记录所有已解决的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思想。具体的动态规划算法是多种多样的，但它们具有相同的填表格式。

动态规划算法适用于解最优化问题。通常可以按以下步骤设计动态规划算法：

(1) 找出最优解的性质，并刻画其结构特征；

(2) 递归地定义最优值；

(3) 以自底向上的方式计算出最优值；

(4) 根据计算最优值时得到的信息，构造最优解。

步骤(1)～(3)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(4)可以省去。若需要求问题的最优解，则必须执行步骤(4)。此时，在步骤(3)中计算最优值时，通常需记录更多的信息，以便在步骤(4)中，根据所记录的信息，快速构造出最优解。

下面以具体例子说明如何运用动态规划算法的设计思想，并分析可用动态规划算法求解的问题应该具备的一般特征。

3.1 矩阵连乘问题

给定 n 个矩阵$\{\boldsymbol{A}_1,\boldsymbol{A}_2,\cdots,\boldsymbol{A}_n\}$，其中，$\boldsymbol{A}_i$ 与 $\boldsymbol{A}_{i+1}$是可乘的，$i=1,2,\cdots,n-1$。考查这n 个矩阵的连乘积 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_n$。由于矩阵乘法满足结合律，故计算矩阵的连乘积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完全确定，也就是说该连乘积已完全加括号，则可以依此次序反复调用 2 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递归地定义为：

(1) 单个矩阵是完全加括号的；

(2) 矩阵连乘积 $\boldsymbol{A}$ 是完全加括号的,则 $\boldsymbol{A}$ 可表示为 2 个完全加括号的矩阵连乘积 $\boldsymbol{B}$ 和 $\boldsymbol{C}$ 的乘积并加括号,即 $\boldsymbol{A}=(\boldsymbol{BC})$。

例如,矩阵连乘积 $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4$ 可以有以下 5 种不同的完全加括号方式:

$$(\boldsymbol{A}_1(\boldsymbol{A}_2(\boldsymbol{A}_3\boldsymbol{A}_4)))$$
$$(\boldsymbol{A}_1((\boldsymbol{A}_2\boldsymbol{A}_3)\boldsymbol{A}_4))$$
$$((\boldsymbol{A}_1\boldsymbol{A}_2)(\boldsymbol{A}_3\boldsymbol{A}_4))$$
$$((\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))\boldsymbol{A}_4)$$
$$(((\boldsymbol{A}_1\boldsymbol{A}_2)\boldsymbol{A}_3)\boldsymbol{A}_4)$$

每一种完全加括号方式对应于一个矩阵连乘积的计算次序,而矩阵连乘积的计算次序与其计算量有密切关系。

首先考虑计算 2 个矩阵乘积所需的计算量。

计算 2 矩阵乘积的标准算法如下,其中,ra,ca 和 rb,cb 分别表示矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的行数和列数。

```
public static void matrixMultiply(int [][]a, int [][]b, int [][]c, int ra, int ca, int rb, int cb)
{
    if (ca!=rb)
      throw new IllegalArgumentException ("矩阵不可乘");
    for (int i=0; i<ra; i++)
     for (int j=0; j<cb; j++){
       int sum=a[i][0]*b[0][j];
       for (int k=1; k<ca; k++)
         sum+=a[i][k]*b[k][j];
       c[i][j]=sum;
       }
}
```

矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 可乘的条件是矩阵 $\boldsymbol{A}$ 的列数等于矩阵 $\boldsymbol{B}$ 的行数。若 $\boldsymbol{A}$ 是一个 $p\times q$ 矩阵,$\boldsymbol{B}$ 是一个 $q\times r$ 矩阵,则其乘积 $\boldsymbol{C}=\boldsymbol{AB}$ 是一个 $p\times r$ 矩阵。在上述计算 $\boldsymbol{C}$ 的标准算法中,主要计算量在 3 重循环,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,考查计算 3 个矩阵 $\{\boldsymbol{A}_1,\boldsymbol{A}_2,\boldsymbol{A}_3\}$ 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$((\boldsymbol{A}_1\boldsymbol{A}_2)\boldsymbol{A}_3)$计算,3 个矩阵连乘积需要的数乘次数为 $10\times100\times5+10\times5\times50=7500$。若按第二种加括号方式$(\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))$计算,3 个矩阵连乘积总共需要 $10\times5\times50+10\times100\times50=75\,000$ 次数乘。第二种加括号方式的计算量是第一种加括号方式计算量的 10 倍。由此可见,在计算矩阵连乘积时,加括号方式,则计算次序对计算量有很大影响。于是,自然提出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$\{\boldsymbol{A}_1,\boldsymbol{A}_2,\cdots,\boldsymbol{A}_n\}$(其中矩阵 $\boldsymbol{A}_i$ 的维数为 $p_{i-1}\times p_i$,$i=1,2,\cdots,n$),如何确定计算矩阵连乘积 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_n$ 的计算次序(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。

穷举搜索法是最容易想到的方法。也就是列举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数乘次数,从中找出一种数乘次数最少的计算次序。这样做计算量太大。事实上,对于 n 个矩阵的连乘积,设其不同的计算次序为 $P(n)$。由于可以先在第 k

个和第$k+1$个矩阵之间将原矩阵序列分为2个矩阵子序列,$k=1,2,\cdots,n-1$;然后分别对这2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。由此,可以得到关于$P(n)$的递推式如下:

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

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

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

也就是说,$P(n)$是随n的增长呈指数增长的。因此,穷举搜索法不是一个有效算法。

下面考虑用动态规划法解矩阵连乘积的最优计算次序问题。如前所述,按以下几个步骤进行。

1. 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$\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_j$简记为$\boldsymbol{A}[i:j]$。考查计算$\boldsymbol{A}[1:n]$的最优计算次序。设这个计算次序在矩阵$\boldsymbol{A}_k$和$\boldsymbol{A}_{k+1}$之间将矩阵链断开,$1\leqslant k<n$,则其相应的完全加括号方式为$((\boldsymbol{A}_1\cdots\boldsymbol{A}_k)(\boldsymbol{A}_{k+1}\cdots\boldsymbol{A}_n))$。即依此次序,先计算$\boldsymbol{A}[1:k]$和$\boldsymbol{A}[k+1:n]$,然后将计算结果相乘得到$\boldsymbol{A}[1:n]$。依此计算次序,总计算量为$\boldsymbol{A}[1:k]$的计算量加上$\boldsymbol{A}[k+1:n]$的计算量,再加上$\boldsymbol{A}[1:k]$和$\boldsymbol{A}[k+1:n]$相乘的计算量。

这个问题的一个关键特征是:计算$\boldsymbol{A}[1:n]$的最优次序所包含的计算矩阵子链$\boldsymbol{A}[1:k]$和$\boldsymbol{A}[k+1:n]$的次序也是最优的。事实上,若有一个计算$\boldsymbol{A}[1:k]$的次序需要的计算量更少,则用此次序替换原来计算$\boldsymbol{A}[1:k]$的次序,得到的计算$\boldsymbol{A}[1:n]$的计算量将比按最优次序计算所需计算量更少,这是一个矛盾。同理可知,计算$\boldsymbol{A}[1:n]$的最优次序所包含的计算矩阵子链$\boldsymbol{A}[k+1:n]$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。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。

2. 建立递归关系

设计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$\boldsymbol{A}[i:j]$,$1\leqslant i\leqslant j\leqslant n$,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,$\boldsymbol{A}[i:j]=\boldsymbol{A}_i$为单一矩阵,无需计算,因此,$m[i][i]=0$,$i=1,2,\cdots,n$。

当$i<j$时,可利用最优子结构性质计算$m[i][j]$。事实上,若计算$\boldsymbol{A}[i:j]$的最优次序在$\boldsymbol{A}_k$和$\boldsymbol{A}_{k+1}$之间断开,$i\leqslant k<j$,则$m[i][j]=m[i][k]+m[k+1][j]+p_{i-1}\times p_k\times p_j$。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$j-i$种可能,即$k\in\{i,i+1,\cdots,j-1\}$。因此,$k$是这$j-i$个位置中使计算量达到最小的那个位置。从而$m[i][j]$可以递归地定义为

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

$m[i][j]$给出了最优值，即计算 $\mathbf{A}[i:j]$所需的最少数乘次数。同时还确定了计算 $\mathbf{A}[i:j]$的最优次序中的断开位置 k，也就是说，对于这个 k 有

$$m[i][j]=m[i][k]+m[k+1][j]+p_{i-1}\times p_k\times p_j$$

若将对应于 $m[i][j]$的断开位置 k 记为 $s[i][j]$，在计算出最优值 $m[i][j]$后，可递归地由 $s[i][j]$构造出相应的最优解。

3. 计算最优值

根据计算 $m[i][j]$的递归式，容易写一个递归算法计算 $m[1][n]$。稍后将看到，简单地递归计算将耗费指数计算时间。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 $1\leqslant i\leqslant j\leqslant n$ 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，可依据其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。下面所给出的动态规划算法matrixChain中，输入参数$\{p_0,p_1,\cdots,p_n\}$存储于数组 p 中。算法除了输出最优值数组 m 外还输出记录最优断开位置的数组 s。

```
public static void matrixChain(int []p, int [][]m, int [][]s)
{
    int n=p.length-1;
    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;
        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;}
            }
        }
}
```

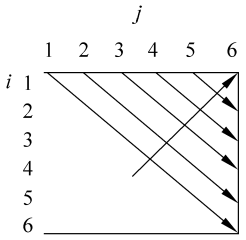
算法 matrixChain，首先计算出 $m[i][i]=0,i=1,2,\cdots,n$。然后，根据递归式，按矩阵链长递增的方式依次计算 $m[i][i+1],i=1,2,\cdots,n-1$，(矩阵链长度为 2)；$m[i][i+2],i=1,2,\cdots,n-2$，(矩阵链长度为 3)；……。在计算 $m[i][j]$时，只用到已计算出的 $m[i][k]$和

$m[k+1][j]$。

例如,设要计算矩阵连乘积 $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4\boldsymbol{A}_5\boldsymbol{A}_6$,其中各矩阵的维数分别为

$\boldsymbol{A}_1$	$\boldsymbol{A}_2$	$\boldsymbol{A}_3$	$\boldsymbol{A}_4$	$\boldsymbol{A}_5$	$\boldsymbol{A}_6$
30×35	35×15	15×5	5×10	10×20	20×25

动态规划算法 matrixChain 计算 $m[i][j]$先后次序如图 3-1(a)所示,计算结果$m[i][j]$和 $s[i][j]$,$1\leqslant i\leqslant j\leqslant n$,分别如图 3-1(b)和(c)所示。



(a) 计算次序

i \ j	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) $m[i][j]$

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

(c) $s[i][j]$

图 3-1　计算 $m[i][j]$的次序

例如,在计算 $m[2][5]$时,依递归式有

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

且 $k=3$,因此,$s[2][5]=3$。

算法 matrixChain 的主要计算量取决于算法中对 r,i 和 k 的 3 重循环。循环体内的计算量为 $O(1)$,而 3 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有效得多。

4. 构造最优解

动态规划算法的第四步是构造问题的最优解。算法 matrixChain 只是计算出了最优值,并未给出最优解。也就是说,通过算法 matrixChain 的计算,只知道最少数乘次数,还不知道具体应按什么次序做矩阵乘法才能达到最少的数乘次数。

事实上,算法 matrixChain 已记录了构造最优解所需要的全部信息。$s[i][j]$中的数 k 表明计算矩阵链 $\boldsymbol{A}[i:j]$的最佳方式应在矩阵 $\boldsymbol{A}_k$和 $\boldsymbol{A}_{k+1}$之间断开,即最优的加括号方式应为$(\boldsymbol{A}[i:k])(\boldsymbol{A}[k+1:j])$。因此,从 $s[1][n]$记录的信息可知计算 $\boldsymbol{A}[1:n]$的最优加括号方式为$(\boldsymbol{A}[1:s[1][n]])(\boldsymbol{A}[s[1][n]+1:n])$。而 $\boldsymbol{A}[1:s[1][n]]$的最优加括号方式为$(\boldsymbol{A}[1:s[1][s[1][n]]])(\boldsymbol{A}[s[1][s[1][n]]+1:s[1][s[1][n]]])$。同理可以确定 $\boldsymbol{A}[s[1][n]+1:n]$的最优加括号方式在 $s[s[1][n]+1][n]$处断开,……,照此递推下去,最终可以确定 $\boldsymbol{A}[1:n]$的最优完全加括号方式,即构造出问题的一个最优解。

下面的算法 traceback 按算法 matrixChain 计算出的断点矩阵 s 指示的加括号方式输出计算 $\boldsymbol{A}[i:j]$的最优计算次序。

```
public static void traceback(int [][]s, int i, int j)
{
    if (i==j) return;
      traceback(s, i, s[i][j]);
      traceback(s, s[i][j]+1, j);
      System.out.println("Multiply A"+i+","+s[i][j]+
                  "and A"+(s[i][j]+1)+","+j);
}
```

要输出 $\boldsymbol{A}[1:n]$的最优计算次序只要调用上面的 traceback(1,n,s)即可。对于上面所举的例子,通过调用算法 traceback(1,6,s),可输出最优计算次序$((\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))((\boldsymbol{A}_4\boldsymbol{A}_5)\boldsymbol{A}_6))$。

3.2 动态规划算法的基本要素

从计算矩阵连乘积最优计算次序的动态规划算法可以看出,该算法的有效性依赖于问题本身所具有的两个重要性质:最优子结构性质和子问题重叠性质。从一般的意义上讲,问题所具有的这两个重要性质是该问题可用动态规划算法求解的基本要素。这对于在设计求解具体问题的算法时,是否选择动态规划算法具有指导意义。下面着重研究动态规划算法的这两个基本要素以及动态规划法的变形——备忘录方法。

1. 最优子结构

设计动态规划算法的第一步通常是刻画最优解的结构。当问题的最优解包含了其子问题的最优解时,称该问题具有最优子结构性质。问题的最优子结构性质提供了该问题可用动态规划算法求解的重要线索。

在矩阵连乘积最优计算次序问题中,注意到,若 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_n$ 的最优完全加括号方式在 $\boldsymbol{A}_k$ 和 $\boldsymbol{A}_{k+1}$之间将矩阵链断开,则由此确定的子链 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_k$ 和 $\boldsymbol{A}_{k+1}\boldsymbol{A}_{k+2}\cdots\boldsymbol{A}_n$ 的完全加括号方式也是最优的。也就是说该问题具有最优子结构性质。在分析该问题的最优子结构性质时,所用的方法具有普遍性。首先假设由问题的最优解导出的子问题的解不是最优的,然后再设法说明在这个假设下可构造出比原问题最优解更好的解,从而导致矛盾。

在动态规划算法中,利用问题的最优子结构性质,以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。算法考查的子问题空间的规模较小。例如,在矩阵连乘积最优计算次序问题中,子问题空间由矩阵链的所有不同子链组成。所有不同子链的个数为 $\theta(n^2)$,因而子问题空间的规模为 $\theta(n^2)$。

2. 重叠子问题

可用动态规划算法求解的问题应该具备的另一个基本要素是子问题的重叠性质。也就是说,在用递归算法自顶向下求解问题时,每次产生的子问题并不总是新问题,有些子问题被反复计算多次。动态规划算法正是利用了这种子问题的重叠性质,对每一个子问题只解一次,而后将其解保存在一个表格中,当再次需要解此子问题时,只是简单地用常数时间查看一下结果。通常,不同的子问题个数随问题的大小呈多项式增长。因此,用动态规划算法通常只需要多项式时间,从而获得较高的解题效率。

为了说明这一点,考虑计算矩阵连乘积最优计算次序时,利用递归式直接计算$\boldsymbol{A}[i:j]$的递归算法 recurmatrixChain。

```
public static int recurMatrixChain(int i, int j)
{
    if (i==j) return 0;
    int u=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;
}
```

用算法 recurmatrixChain(1,4)计算 $\boldsymbol{A}[1:4]$的递归树如图 3-2 所示。从该图可以看出,许多子问题被重复计算。

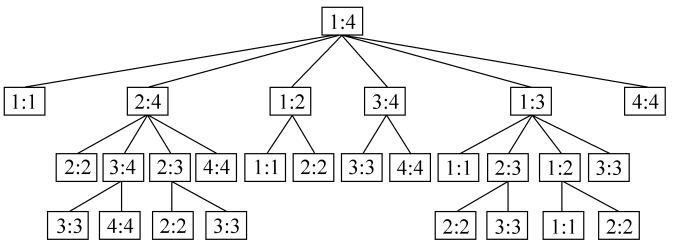


图 3-2　计算 $\boldsymbol{A}[1:4]$的递归树

事实上,可以证明该算法的计算时间 $T(n)$有指数下界。设算法中判断语句和赋值语句花费常数时间,则由算法的递归部分可得关于 $T(n)$的递归不等式如下:

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

因此,当 $n>1$ 时,有

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

据此,可用数学归纳法证明 $T(n)\geqslant 2^{n-1}=\Omega(2^n)$。

因此,直接递归算法 recurmatrixChain 的计算时间随 n 指数增长。相比之下,解同一问题的动态规划算法 matrixChain 只需计算时间 $O(n^3)$。其有效性就在于它充分利用了问题的子问题重叠性质。不同的子问题个数为 $\theta(n^2)$,而动态规划算法对于每个不同的子问题只计算一次,从而节省了大量不必要的计算。由此也可看出,当解某一问题的直接递归算法所产生的递归树中,相同的子问题反复出现,并且不同子问题的个数又相对较少时,用动态规划算法是有效的。

3. 备忘录方法

备忘录方法是动态规划算法的变形。与动态规划算法一样，备忘录方法用表格保存已解决的子问题的答案，在下次需要解此子问题时，只要简单地查看该子问题的解答，而不必重新计算。与动态规划算法不同的是，备忘录方法的递归方式是自顶向下的，而动态规划算法则是自底向上递归的。因此，备忘录方法的控制结构与直接递归方法的控制结构相同，区别在于备忘录方法为每个解过的子问题建立了备忘录以备需要时查看，避免了相同子问题的重复求解。

备忘录方法为每个子问题建立一个记录项，初始化时，该记录项存入一个特殊值，表示该子问题尚未求解。在求解过程中，对每个待求子问题，首先查看其相应的记录项。若记录项中存储的是初始化时存入的特殊值，则表示该子问题是第一次遇到，此时计算出该子问题的解，并保存在其相应的记录项中，以备以后查看。若记录项中存储的已不是初始化时存入的特殊值，则表示该子问题已被计算过，其相应的记录项中存储的是该子问题的解答。此时，只要从记录项中取出该子问题的解答即可，而不必重新计算。

下面的算法 memoizedmatrixChain 是解矩阵连乘积最优计算次序问题的备忘录方法。

```
public static int memoizedmatrixChain(int n)
{
    for (int i=1; i<=n; i++)
      for (int j=i; j<=n; j++)
        m[i][j]=0;
    return lookupChain(1,n);
}

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

与动态规划算法 matrixChain 一样，备忘录算法 memoizedmatrixChain 用数组 m 记录子问题的最优值。m 初始化为 0，表示相应的子问题还未被计算。在调用 lookupChain 时，

若 $m[i][j]>0$,则表示其中存储的是所要求子问题的计算结果,直接返回此结果即可。否则与直接递归算法一样,自顶向下地递归计算,并将计算结果存入 $m[i][j]$后返回。因此,算法 lookupChain 总能返回正确的值,但仅在它第一次被调用时计算,以后的调用就直接返回计算结果。

与动态规划算法一样,备忘录算法 memoizedmatrixChain 耗时 $O(n^3)$。事实上,共有 $O(n^2)$个备忘记录项 $m[i][j]$,$i=1,2,\cdots,n$,$j=i,\cdots,n$。这些记录项的初始化耗费$O(n^2)$时间。每个记录项只填入一次。每次填入时,不包括填入其他记录项的时间,共耗费 $O(n)$时间。因此,算法 lookupChain 填入 $O(n^2)$个记录项总共耗费 $O(n^3)$计算时间。由此可见,通过使用备忘录技术,直接递归算法的计算时间从 $\Omega(2^n)$降至 $O(n^3)$。

综上所述,矩阵连乘积的最优计算次序问题可用自顶向下的备忘录算法或自底向上的动态规划算法在 $O(n^3)$计算时间内求解。这两个算法都利用了子问题重叠性质。总共有 $\theta(n^2)$个不同的子问题。对每个子问题,两种方法都只解一次,并记录答案。再次遇到该子问题时,不重新求解而简单地取用已得到的答案。因此,节省了计算量,提高了算法的效率。

一般地讲,当一个问题的所有子问题都至少要解一次时,用动态规划算法比用备忘录方法好。此时,动态规划算法没有任何多余的计算。同时,对于许多问题,常可利用其规则的表格存取方式,减少动态规划算法的计算时间和空间需求。当子问题空间中的部分子问题可不必求解时,用备忘录方法则较有利,因为从其控制结构可以看出,该方法只解那些确实需要求解的子问题。

3.3 最长公共子序列

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。确切地说,若给定序列 $X=\{x_1,x_2,\cdots,x_m\}$,则另一序列 $Z=\{z_1,z_2,\cdots,z_k\}$,X 的子序列是指存在一个严格递增下标序列$\{i_1,i_2,\cdots,i_k\}$使得对于所有 $j=1,2,\cdots,k$ 有 $z_j=x_{ij}$。例如,序列 $Z=\{B,C,D,B\}$是序列 $X=\{A,B,C,B,D,A,B\}$的子序列,相应的递增下标序列为$\{2,3,5,7\}$。

给定两个序列 X 和 Y,当另一序列 Z 既是 X 的子序列又是 Y 的子序列时,称 Z 是序列 X 和 Y 的公共子序列。

例如,若 $X=\{A,B,C,B,D,A,B\}$,$Y=\{B,D,C,A,B,A\}$,序列$\{B,C,A\}$是 X 和 Y 的一个公共子序列,但它不是 X 和 Y 的最长公共子序列。序列$\{B,C,B,A\}$也是 X 和 Y 的一个公共子序列,它的长度为 4,而且它是 X 和 Y 的最长公共子序列,因为 X 和 Y 没有长度大于 4 的公共子序列。

最长公共子序列问题:给定两个序列 $X=\{x_1,x_2,\cdots,x_m\}$和 $Y=\{y_1,y_2,\cdots,y_n\}$,找出 X 和 Y 的最长公共子序列。

动态规划算法可有效地解此问题。下面按照动态规划算法设计的各个步骤设计解此问题的有效算法。

1. 最长公共子序列的结构

穷举搜索法是最容易想到的算法。对 X 的所有子序列,检查它是否也是 Y 的子序列,从而确定它是否为 X 和 Y 的公共子序列。并且在检查过程中记录最长的公共子序列。X

的所有子序列都检查过后即可求出 X 和 Y 的最长公共子序列。X 的每个子序列相应于下标集$\{1,2,\cdots,m\}$的一个子集。因此，共有 2^m 个不同子序列，从而穷举搜索法需要指数时间。

事实上，最长公共子序列问题具有最优子结构性质。

设序列 $X=\{x_1,x_2,\cdots,x_m\}$和 $Y=\{y_1,y_2,\cdots,y_n\}$的最长公共子序列为 $Z=\{z_1,z_2,\cdots,z_k\}$，则

(1) 若 $x_m=y_n$，则 $z_k=x_m=y_n$，且 Z_{k-1}是 X_{m-1}和 Y_{n-1}的最长公共子序列；

(2) 若 $x_m\neq y_n$ 且 $z_k\neq x_m$，则 Z 是 X_{m-1}和 Y 的最长公共子序列；

(3) 若 $x_m\neq y_n$ 且 $z_k\neq y_n$，则 Z 是 X 和 Y_{n-1}的最长公共子序列。

其中，$X_{m-1}=\{x_1,x_2,\cdots,x_{m-1}\}$；$Y_{n-1}=\{y_1,y_2,\cdots,y_{n-1}\}$；$Z_{k-1}=\{z_1,z_2,\cdots,z_{k-1}\}$。

证明：(1) 用反证法。若 $z_k\neq x_m$，则$\{z_1,z_2,\cdots,z_k,x_m\}$是 X 和 Y 的长度为 $k+1$ 的公共子序列。这与 Z 是 X 和 Y 的最长公共子序列矛盾。因此，必有 $z_k=x_m=y_n$。由此可知 Z_{k-1}是 X_{m-1}和 Y_{n-1}的长度为 $k-1$ 的公共子序列。若 X_{m-1}和 Y_{n-1}有长度大于 $k-1$ 的公共子序列 W，则将 x_m 加在其尾部产生 X 和 Y 的长度大于 k 的公共子序列。此为矛盾。故 Z_{k-1}是 X_{m-1}和 Y_{n-1}的最长公共子序列。

(2) 由于 $z_k\neq x_m$，Z 是 X_{m-1}和 Y 的公共子序列。若 X_{m-1}和 Y 有长度大于 k 的公共子序列 W，则 W 也是 X 和 Y 的长度大于 k 的公共子序列。这与 Z 是 X 和 Y 的最长公共子序列矛盾。由此即知，Z 是 X_{m-1}和 Y 的最长公共子序列。

(3) 证明与(2)类似。

由此可见，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。

2. 子问题的递归结构

由最长公共子序列问题的最优子结构性质可知，要找出 $X=\{x_1,x_2,\cdots,x_m\}$和 $Y=\{y_1,y_2,\cdots,y_n\}$的最长公共子序列，可按以下方式递归计算：当 $x_m=y_n$ 时，找出 X_{m-1}和 Y_{n-1}的最长公共子序列，然后在其尾部加上 $x_m(=y_n)$即可得 X 和 Y 的最长公共子序列。当 $x_m\neq y_n$ 时，必须解两个子问题，即找出 X_{m-1}和 Y 的一个最长公共子序列及 X 和 Y_{n-1}的一个最长公共子序列。这两个公共子序列中较长者即为 X 和 Y 的最长公共子序列。

由此递归结构容易看到最长公共子序列问题具有子问题重叠性质。例如，在计算 X 和 Y 的最长公共子序列时，可能要计算 X 和 Y_{n-1}及 X_{m-1}和 Y 的最长公共子序列。而这两个子问题都包含一个公共子问题，即计算 X_{m-1}和 Y_{n-1}的最长公共子序列。

首先建立子问题最优值的递归关系。用 $c[i][j]$记录序列 X_i 和 Y_j 的最长公共子序列的长度。其中，$X_i=\{x_1,x_2,\cdots,x_i\}$；$Y_j=\{y_1,y_2,\cdots,y_j\}$。当 $i=0$ 或 $j=0$ 时，空序列是 X_i 和 Y_j 的最长公共子序列，故此时 $c[i][j]=0$。在其他情况下，由最优子结构性质可建立递归关系如下：

$$c[i][j]=\begin{cases}0 & i=0,j=0\\ c[i-1][j-1]+1 & i,j>0;x_i=y_j\\ \max\{c[i][j-1],c[i-1][j]\} & i,j>0;x_i\neq y_j\end{cases}$$

3. 计算最优值

直接利用递归式容易写出计算 $c[i][j]$ 的递归算法,但其计算时间是随输入长度指数增长的。由于在所考虑的子问题空间中,总共有 $\theta(mn)$ 个不同的子问题,因此,用动态规划算法自底向上地计算最优值能提高算法的效率。

计算最长公共子序列长度的动态规划算法 lcsLength 以序列 $X=\{x_1,x_2,\cdots,x_m\}$ 和 $Y=\{y_1,y_2,\cdots,y_n\}$ 作为输入。输出两个数组 c 和 b。其中,$c[i][j]$ 存储 X_i 和 Y_j 的最长公共子序列的长度,$b[i][j]$ 记录 $c[i][j]$ 的值是由哪一个子问题的解得到的,这在构造最长公共子序列时要用到。问题的最优值,即 X 和 Y 的最长公共子序列的长度记录于 $c[m][n]$ 中。

```
public static int lcsLength(char []x, char []y, int [][]b)
{
    int m=x.length-1;
    int n=y.length-1;
    int [][]c=new int [m+1][n+1];
    for (int i=1; i<=m; i++) c[i][0]=0;
    for (int i=1; i<=n; i++) c[0][i]=0;
    for (int i=1; i<=m; i++)
      for (int j=1; j<=n; j++){
        if (x[i]==y[j]){
          c[i][j]=c[i-1][j-1]+1;
          b[i][j]=1;
          }
        else if (c[i-1][j]>=c[i][j-1]){
          c[i][j]=c[i-1][j];
          b[i][j]=2;
          }
          else {
            c[i][j]=c[i][j-1];
            b[i][j]=3;
            }
        }
    return c[m][n];
}
```

由于每个数组单元的计算耗费 $O(1)$ 时间,算法 lcsLength 耗时 $O(mn)$。

4. 构造最长公共子序列

由算法 lcsLength 计算得到的数组 b 可用于快速构造序列 $X=\{x_1,x_2,\cdots,x_m\}$ 和 $Y=\{y_1,y_2,\cdots,y_n\}$ 的最长公共子序列。首先从 $b[m][n]$ 开始,依其值在数组 b 中搜索。当 $b[i][j]=1$ 时,表示 X_i 和 Y_j 的最长公共子序列是由 X_{i-1} 和 Y_{j-1} 的最长公共子序列在尾部加上 x_i 所得到的子序列;当 $b[i][j]=2$ 时,表示 X_i 和 Y_j 的最长公共子序列与 X_{i-1} 和 Y_j 的最长公共子序列相同;当 $b[i][j]=3$ 时,表示 X_i 和 Y_j 的最长公共子序列与 X_i 和 Y_{j-1} 的最长公共子序列相同。

下面的算法 lcs 实现根据 b 的内容打印出 X_i 和 Y_j 的最长公共子序列。通过算法调用 lcs(m,n,x,b)便可打印出序列 X 和 Y 的最长公共子序列。

```
public static void lcs(int i,int j,char []x,int [][]b)
{
    if (i==0 || j==0) return;
    if (b[i][j]==1){
      lcs(i-1,j-1,x,b);
      System.out.print(x[i]);
      }
    else if (b[i][j]==2) lcs(i-1,j,x,b);
      else lcs(i,j-1,x,b);
}
```

在算法 lcs 中,每一次递归调用使 i 或 j 减 1,因此算法的计算时间为 $O(m+n)$。

5. 算法的改进

对于具体问题,按照一般的算法设计策略设计出的算法,往往在算法的时间和空间需求上还有较大的改进余地。通常可以利用具体问题的一些特殊性对算法做进一步改进。例如,在算法 lcsLength 和 lcs 中,可进一步将数组 b 省去。事实上,数组元素$c[i][j]$的值仅由 $c[i-1][j-1]$,$c[i-1][j]$和 $c[i][j-1]$这 3 个数组元素的值所确定。对于给定的数组元素 $c[i][j]$,可以不借助于数组 b 而仅借助于 c 本身,在 $O(1)$时间内确定 $c[i][j]$的值是由 $c[i-1][j-1]$,$c[i-1][j]$和 $c[i][j-1]$中哪一个值所确定的。因此,可以写一个类似于 lcs 的算法,不用数组 b 而在$O(m+n)$时间内构造最长公共子序列。从而可节省 $\theta(mn)$的空间。由于数组 c 仍需要$\theta(mn)$的空间,因此,在渐近的意义上,算法仍需要 $\theta(mn)$的空间,所做的改进,只是对空间复杂性的常数因子的改进。

另外,如果只需要计算最长公共子序列的长度,则算法的空间需求可大大减少。事实上,在计算 $c[i][j]$时,只用到数组 c 的第 i 行和第 $i-1$ 行。因此,用两行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可以将空间需求减至 $O(\min\{m,n\})$。

3.4 凸多边形最优三角剖分

用动态规划算法能有效地解凸多边形的最优三角剖分问题。尽管这是一个几何问题,但在本质上它与矩阵连乘积的最优计算次序问题极为相似。

多边形是平面上一条分段线性闭曲线。也就是说,多边形是由一系列首尾相接的直线段所组成的。组成多边形的各直线段称为该多边形的边。连接多边形相继两条边的点称为多边形的顶点。若多边形的边除了连接顶点外没有别的交点,则称该多边形为一简单多边形。一个简单多边形将平面分为 3 个部分:被包围在多边形内的所有点构成了多边形的内部;多边形本身构成多边形的边界;而平面上其余包围着多边形的点构成了多边形的外部。当一个简单多边形及其内部构成闭凸集时,称该简单多边形为一凸多边形。也就是说,凸多边形边界上或内部的任意两点所连成的直线段上所有点均在凸多边形的内部或边界上。

通常,用多边形顶点的逆时针序列表示凸多边形,即 $P=\{v_0,v_1,\cdots,v_{n-1}\}$表示具有 n 条

边 $v_0v_1, v_1v_2, \cdots, v_{n-1}v_n$ 的凸多边形。其中,约定 $v_0 = v_n$。

若 v_i 与 v_j 是多边形上不相邻的两个顶点,则线段 v_iv_j 称为多边形的一条弦。弦 v_iv_j 将多边形分割成两个多边形 $\{v_i, v_{i+1}, \cdots, v_j\}$ 和 $\{v_j, v_{j+1}, \cdots, v_i\}$。

多边形的三角剖分是将多边形分割成互不相交的三角形的弦的集合 T。图 3-3 是一个凸七边形的两个不同的三角剖分。

在凸多边形 P 的三角剖分 T 中,各弦互不相交,且集合 T 已达到最大,即 P 的任一不在 T 中的弦必与 T 中某一弦相交。在有 n 个顶点的凸多边形的三角剖分中,恰有 $n-3$ 条弦和 $n-2$ 个三角形。

凸多边形最优三角剖分的问题:给定凸多边形 $P=\{v_0, v_1, \cdots, v_{n-1}\}$,以及定义在由多边形的边和弦组成的三角形上的权函数 w。要求确定该凸多边形的三角剖分,使得该三角剖分所对应的权,即该三角剖分中诸三角形上权之和为最小。

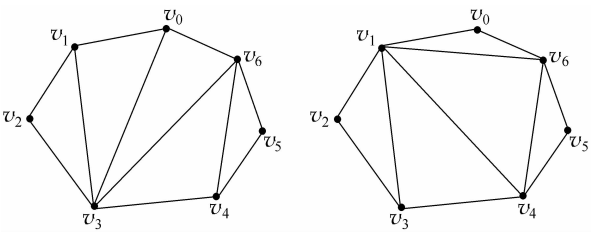


图 3-3　一个凸七边形的两个不同的三角剖分

可以定义三角形上各种各样的权函数 w。例如,

$$w(v_iv_jv_k) = |v_iv_j| + |v_jv_k| + |v_kv_i|$$

其中,$|v_iv_j|$ 是点 v_i 到 v_j 的欧氏距离。相应于此权函数的最优三角剖分即为最小弦长三角剖分。

本节所述算法可适用于任意权函数。

1. 三角剖分的结构及其相关问题

凸多边形的三角剖分与表达式的完全加括号方式之间具有十分紧密的联系。正如所看到的,矩阵连乘积的最优计算次序问题等价于矩阵链的最优完全加括号方式。这些问题之间的相关性可从它们所对应的完全二叉树的同构性看出。

一个表达式的完全加括号方式相应于一棵完全二叉树,称为表达式的语法树。例如,完全加括号的矩阵连乘积 $((\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))(\boldsymbol{A}_4(\boldsymbol{A}_5\boldsymbol{A}_6)))$ 相应的语法树如图 3-4 (a)所示。

语法树中每一个叶结点表示表达式中一个原子。在语法树中,若一结点有一个表示表达式 E_l 的左子树,以及一个表示表达式 E_r 的右子树,则以该结点为根的子树表示表达式 (E_lE_r)。因此,有 n 个原子的完全加括号表达式对应于唯一的一棵有 n 个叶结点的语法树,反之亦然。

凸多边形 $\{v_0, v_1, \cdots, v_{n-1}\}$ 的三角剖分也可以用语法树表示。例如,图 3-4 (a)中凸多边形的三角剖分可用图 3-4 (b)所示的语法树表示。该语法树的根结点为边 v_0v_6。三角剖分中的弦组成其余的内结点。多边形中除 v_0v_6 边外的各边都是语法树的一个叶结点。树根 v_0v_6 是三角形 $v_0v_3v_6$ 的一条边。该三角形将原多边形分为三个部分:三角形 $v_0v_3v_6$,凸

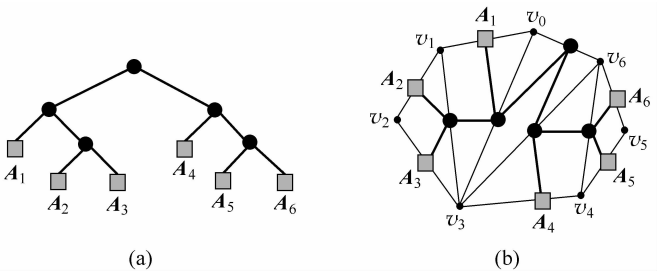


图 3-4　表达式语法树与三角剖分的对应

多边形$\{v_0,v_1,\cdots,v_3\}$和凸多边形$\{v_3,v_4,\cdots,v_6\}$。三角形 $v_0v_3v_6$ 的另外两条边，即弦 v_0v_3 和 v_3v_6 为根的两个儿子。以它们为根的子树表示凸多边形$\{v_0,v_1,\cdots,v_3\}$和$\{v_3,v_4,\cdots,v_6\}$的三角剖分。

在一般情况下，凸 n 边形的三角剖分对应于一棵有 $n-1$ 个叶结点的语法树。反之，也可根据一棵有 $n-1$ 个叶结点的语法树产生相应的凸 n 边形的三角剖分。也就是说，凸 n 边形的三角剖分与有 $n-1$ 个叶结点的语法树之间存在一一对应关系。由于n 个矩阵的完全加括号乘积与 n 个叶结点的语法树之间存在一一对应关系，因此，n 个矩阵的完全加括号乘积也与凸$(n+1)$边形中的三角剖分之间存在一一对应关系。图 3-4(a) 和(b)表示出这种对应关系。矩阵连乘积 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_n$ 中的每个矩阵 $\boldsymbol{A}_i$ 对应于凸$(n+1)$边形中的一条边 $v_{i-1}v_i$。三角剖分中的一条弦 v_iv_j，$i<j$，对应于矩阵连乘积 $A[i+1:j]$。

事实上，矩阵连乘积的最优计算次序问题是凸多边形最优三角剖分问题的特殊情形。对于给定的矩阵链 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_n$，定义与之相应的凸$(n+1)$边形 $P=\{v_0,v_1,\cdots,v_n\}$，使得矩阵 $\boldsymbol{A}_i$ 与凸多边形的边 $v_{i-1}v_i$ 一一对应。若矩阵 $\boldsymbol{A}_i$ 的维数为 $p_{i-1}\times p_i$，$i=1,2,\cdots,n$，则定义三角形 $v_iv_jv_k$ 上的权函数值为：$w(v_iv_jv_k)=p_ip_jp_k$。依此权函数的定义，凸多边形 P 的最优三角剖分所对应的语法树给出矩阵链 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_n$ 的最优完全加括号方式。

2. 最优子结构性质

凸多边形的最优三角剖分问题有最优子结构性质。

事实上，若凸$(n+1)$边形 $P=\{v_0,v_1,\cdots,v_n\}$的最优三角剖分 T 包含三角形 $v_0v_kv_n$，$1\leqslant k\leqslant n-1$，则 T 的权为三个部分权的和：三角形 $v_0v_kv_n$ 的权，子多边形$\{v_0,v_1,\cdots,v_k\}$ 和$\{v_k,v_{k+1},\cdots,v_n\}$的权之和。可以断言，由 T 所确定的这两个子多边形的三角剖分也是最优的。因为若有$\{v_0,v_1,\cdots,v_k\}$或$\{v_k,v_{k+1},\cdots,v_n\}$的更小权的三角剖分将导致 T 不是最优三角剖分的矛盾。

3. 最优三角剖分的递归结构

首先，定义 $t[i][j]$，$1\leqslant i<j\leqslant n$ 为凸子多边形$\{v_{i-1},v_i,\cdots,v_j\}$的最优三角剖分所对应的权函数值，即其最优值。为方便起见，设退化的多边形$\{v_{i-1},v_i\}$具有权值 0。据此定义，要计算的凸$(n+1)$边形 P 的最优权值为 $t[1][n]$。

$t[i][j]$的值可以利用最优子结构性质递归地计算。由于退化的 2 顶点多边形的权值为 0，所以 $t[i][i]=0$，$i=1,2,\cdots,n$。当 $j-i\geqslant 1$ 时，凸子多边形$\{v_{i-1},v_i,\cdots,v_j\}$至少有 3 个

顶点。由最优子结构性质,$t[i][j]$的值应为$t[i][k]$的值加上$t[k+1][j]$的值,再加上三角形$v_{i-1}v_kv_j$的权值,其中,$i \leqslant k \leqslant j-1$。由于在计算时还不知道$k$的确切位置,而$k$的所有可能位置只有$j-i$个,因此,可以在这$j-i$个位置中选出使$t[i][j]$值达到最小的位置。由此,$t[i][j]$可递归地定义为

$$t[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\leqslant k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j\end{cases}$$

4. 计算最优值

与矩阵连乘积问题中计算$m[i][j]$的递归式进行比较,容易看出,除了权函数的定义外,$t[i][j]$与$m[i][j]$的递归式完全一样。因此,只要对计算$m[i][j]$的算法matrixChain进行很小的修改就完全适用于计算$t[i][j]$。

下面描述的计算凸$(n+1)$边形$P=\{v_0,v_1,\cdots,v_n\}$的最优三角剖分的动态规划算法minWeightTriangulation以凸多边形$P=\{v_0,v_1,\cdots,v_n\}$和定义在三角形上的权函数w作为输入。

```
public static void minWeightTriangulation(int n, int [][] t, int [][] s)
{
    for (int i=1; i<=n; i++) t[i][i]=0;
    for (int r=2; r<=n; r++)
      for (int i=1; i<=n-r+1; i++){
        int j=i+r-1;
        t[i][j]=t[i+1][j]+w(i-1,i,j);
        s[i][j]=i;
        for (int k=i+1; k<i+r-1; k++){
          int u=t[i][k]+t[k+1][j]+w(i-1,k,j);
          if (u<t[i][j]){
            t[i][j]=u;
            s[i][j]=k;}
            }
          }
}
```

与算法matrixChain一样,算法minWeightTriangulation占用$O(n^2)$空间,耗时$O(n^3)$。

5. 构造最优三角剖分

算法minWeightTriangulation在计算每一个凸子多边形$\{v_{i-1},v_i,\cdots,v_j\}$的最优值时,用数组$s$记录了最优三角剖分中所有三角形信息。$s[i][j]$记录了与$v_{i-1}$和$v_j$一起构成三角形的第3个顶点的位置。据此,用$O(n)$时间就可构造出最优三角剖分中的所有三角形。

3.5 多边形游戏

多边形游戏是一个单人玩的游戏,开始时有一个由n个顶点构成的多边形。每个顶点被赋予一个整数值,每条边被赋予一个运算符+或*。所有边依次用整数从1到n编号。

游戏第1步，将一条边删除。

随后的 $n-1$ 步按以下方式操作：

(1) 选择一条边 E 以及由 E 连接着的两个顶点 V_1 和 V_2；

(2) 用一个新的顶点取代边 E 以及由 E 连接着的两个顶点 V_1 和 V_2。将由顶点 V_1 和 V_2 的整数值通过边 E 上的运算得到的结果赋予新顶点。

最后，所有边都被删除，游戏结束。游戏的得分就是所剩顶点上的整数值。

问题：对于给定的多边形，计算最高得分。

该问题与上一节中讨论过的凸多边形最优三角剖分问题类似，但二者的最优子结构性质不同。多边形游戏问题的最优子结构性质更具有一般性。

1. 最优子结构性质

设所给的多边形的顶点和边的顺时针序列为

$$\mathrm{op}[1],\ v[1],\ \mathrm{op}[2],\ v[2],\ \cdots,\ \mathrm{op}[n],\ v[n]$$

其中，op[i]表示第 i 条边所对应的运算符，$v[i]$表示第 i 个顶点上的数值，$i=1\sim n$。

在所给多边形中，从顶点 $i(1\leqslant i\leqslant n)$开始，长度为 j(链中有 j 个顶点)的顺时针链 $p(i,j)$可表示为

$$v[i],\ \mathrm{op}[i+1],\ \cdots,\ v[i+j-1]$$

如果这条链的最后一次合并运算在 op[$i+s$]处发生($1\leqslant s\leqslant j-1$)，则可在op[$i+s$]处将链分割为两个子链 $p(i,s)$和 $p(i+s,j-s)$。

设 m_1 是对子链 $p(i,s)$的任意一种合并方式得到的值，而 a 和 b 分别是在所有可能的合并中得到的最小值和最大值。m_2 是 $p(i+s,j-s)$的任意一种合并方式得到的值，而 c 和 d 分别是在所有可能的合并中得到的最小值和最大值。依此定义有

$$a\leqslant m_1\leqslant b,\quad c\leqslant m_2\leqslant d$$

由于子链 $p(i,s)$和 $p(i+s,j-s)$的合并方式决定了 $p(i,j)$在 op[$i+s$]处断开后的合并方式，在 op[$i+s$]处合并后其值为

$$m=(m_1)\mathrm{op}[i+s](m_2)$$

(1) 当 op[$i+s$]=′+′时，显然有

$$a+c\leqslant m\leqslant b+d$$

换句话说，由链 $p(i,j)$合并的最优性可推出子链 $p(i,s)$和 $p(i+s,j-s)$的最优性，且最大值对应于子链的最大值，最小值对应于子链的最小值。

(2) 当 op[$i+s$]=′*′时，情况有所不同。由于 $v[i]$可取负整数，子链的最大值相乘未必能得到主链的最大值。但是注意到最大值一定在边界点达到，即

$$\min\{ac,ad,bc,bd\}\leqslant m\leqslant\max\{ac,ad,bc,bd\}$$

换句话说，主链的最大值和最小值可由子链的最大值和最小值得到。例如，当 $m=ac$ 时，最大主链由它的两条最小子链组成；同理当 $m=bd$ 时，最大主链由它的两条最大子链组成。无论哪种情形发生，由主链的最优性均可推出子链的最优性。

综上可知多边形游戏问题满足最优子结构性质。

2. 递归求解

由前面的分析可知，为了求链合并的最大值，必须同时求子链合并的最大值和最小值。

因此,在整个计算过程中,应同时计算最大值和最小值。

设 $m[i,j,0]$是链 $p(i,j)$合并的最小值,而 $m[i,j,1]$是最大值。若最优合并在op$[i+s]$处将 $p(i,j)$分为两个长度小于 j 的子链 $p(i,i+s)$和 $p(i+s,j-s)$,且从顶点 i 开始的长度小于 j 的子链的最大值和最小值均已计算出。为叙述方便,记

$$a = m[i,i+s,0]$$
$$b = m[i,i+s,1]$$
$$c = m[i+s,j-s,0]$$
$$d = m[i+s,j-s,1]$$

(1) 当 op$[i+s]=' + '$时,

$$m[i,j,0] = a + c$$
$$m[i,j,1] = b + d$$

(2) 当 op$[i+s]=' * '$时,

$$m[i,j,0] = \min\{ac,ad,bc,bd\}$$
$$m[i,j,1] = \max\{ac,ad,bc,bd\}$$

综合(1)和(2),将 $p(i,j)$在 op$[i+s]$处断开的最大值记为 maxf(i,j,s),最小值记为 minf(i,j,s),则

$$\text{minf}(i,j,s) = \begin{cases} a+c & \text{op}[i+s] = '+' \\ \min\{ac,ad,bc,bd\} & \text{op}[i+s] = '*' \end{cases}$$

$$\text{maxf}(i,j,s) = \begin{cases} b+d & \text{op}[i+s] = '+' \\ \max\{ac,ad,bc,bd\} & \text{op}[i+s] = '*' \end{cases}$$

由于最优断开位置 s 有 $1 \leqslant s \leqslant j-1$ 的 $j-1$ 种情况,由此可知

$$m[i,j,0] = \min_{1 \leqslant s < j}\{\text{minf}(i,j,s)\} \qquad 1 \leqslant i,j \leqslant n$$
$$m[i,j,1] = \max_{1 \leqslant s < j}\{\text{maxf}(i,j,s)\} \qquad 1 \leqslant i,j \leqslant n$$

初始边界值显然为

$$m[i,1,0] = v[i] \qquad 1 \leqslant i \leqslant n$$
$$m[i,1,1] = v[i] \qquad 1 \leqslant i \leqslant n$$

由于多边形是封闭的,在上面的计算中,当 $i+s>n$ 时,顶点 $i+s$ 实际编号为$(i+s) \bmod n$。按上述递推式计算出的 $m[i,n,1]$即为游戏首次删去第 i 条边后得到的最大得分。

3. 算法描述

基于以上讨论可设计解多边形游戏问题的动态规划算法如下:

```
private static void minMax(int i, int s, int j)
{
    int []e=new int [5];
    int a=m[i][s][0],
        b=m[i][s][1],
        r=(i+s-1)%n+1,
        c=m[r][j-s][0],
        d=m[r][j-s][1];
```

```
        if (op[r]=='t'){
            minf=a+c;
            maxf=b+d;
            }
        else
        {
            e[1]=a*c;
            e[2]=a*d;
            e[3]=b*c;
            e[4]=b*d;
            minf=e[1];
            maxf=e[1];
            for (int k=2;k<5;k++){
                if (minf>e[k]) minf=e[k];
                if (maxf<e[k]) maxf=e[k];
            }
        }
    }

    public static int polyMax()
    {
        for (int j=2;j<=n;j++)
            for (int i=1;i<=n;i++)
                for (int s=1;s<j;s++){
                    minMax(i,s,j);
                    if (m[i][j][0]>minf) m[i][j][0]=minf;
                    if (m[i][j][1]<maxf) m[i][j][1]=maxf;
                    }
        int temp=m[1][n][1];
        for (int i=2;i<=n;i++)
            if (temp<m[i][n][1]) temp=m[i][n][1];
        return temp;
    }
```

4. 计算复杂性分析

与凸多边形最优三角剖分问题类似，上述算法需要 $O(n^3)$ 计算时间。

3.6 图像压缩

在计算机中常用像素点灰度值序列 $\{p_1, p_2, \cdots, p_n\}$ 表示图像。其中，整数 $p_i(1 \leqslant i \leqslant n)$ 表示像素点 i 的灰度值。通常灰度值的范围是 0～255。因此，需要用 8 位表示一个像素。

图像的变位压缩存储格式将所给的像素点序列 $\{p_1, p_2, \cdots, p_n\}$ 分割成 m 个连续段 S_1，$S_2, \cdots, S_m$。第 i 个像素段 S_i 中($1 \leqslant i \leqslant m$)，有 $l[i]$ 个像素，且该段中每个像素都只用 $b[i]$ 位

表示。设 $t[i]=\sum_{k=1}^{i-1} l[k], 1\leqslant i\leqslant m$，则第 i 个像素段 S_i 为

$$S_i=\{p_{t[i]+1},\cdots,p_{t[i]+l[i]}\} \qquad 1\leqslant i\leqslant m$$

设 $h_i=\lceil \log(\max_{t[i]+1\leqslant k\leqslant t[i]+l[i]} p_k+1)\rceil$，则 $h_i\leqslant b[i]\leqslant 8$。因此需要用3位表示 $b[i]$，$1\leqslant i\leqslant m$。如果限制 $1\leqslant l[i]\leqslant 255$，则需要用8位表示 $l[i]$，$1\leqslant i\leqslant m$。因此，第 i 个像素段所需的存储空间为 $l[i]*b[i]+11$ 位。按此格式存储像素序列 $\{p_1,p_2,\cdots,p_n\}$，需要 $\sum_{i=1}^{m} l[i]*b[i]+11m$ 位的存储空间。

图像压缩问题要求确定像素序列 $\{p_1,p_2,\cdots,p_n\}$ 的最优分段，使得依此分段所需的存储空间最少。其中，$0\leqslant p_i\leqslant 256$，$1\leqslant i\leqslant n$。每个分段的长度不超过256位。

1. 最优子结构性质

设 $l[i],b[i],1\leqslant i\leqslant m$ 是 $\{p_1,p_2,\cdots,p_n\}$ 的最优分段。显而易见，$l[1],b[1]$ 是 $\{p_1,\cdots,p_{l[1]}\}$ 的最优分段，且 $l[i],b[i],2\leqslant i\leqslant m$ 是 $\{p_{l[1]+1},\cdots,p_n\}$ 的最优分段。即图像压缩问题满足最优子结构性质。

2. 递归计算最优值

设 $s[i],1\leqslant i\leqslant n$ 是像素序列 $\{p_1,\cdots,p_i\}$ 的最优分段所需的存储位数。由最优子结构性质易知

$$s[i]=\min_{1\leqslant k\leqslant \min\{i,256\}}\{s[i-k]+k*\mathrm{bmax}(i-k+1,i)\}+11$$

其中，$\mathrm{bmax}(i,j)=\lceil \log(\max_{i\leqslant k\leqslant j}\{p_k\}+1)\rceil$。

据此可设计解图像压缩问题的动态规划算法如下：

```
static final int lmax=256;
static final int header=11;
static int m;

public static void compress(int p[], int s[], int l[], int b[])
{
    int n=p.length-1;
    s[0]=0;
    for (int i=1; i<=n; i++){
      b[i]=length(p[i]);
      int bmax=b[i];
      s[i]=s[i-1]+bmax;
      l[i]=1;
      for (int j=2; j<=i && j<=lmax; j++){
        if (bmax<b[i-j+1]) bmax=b[i-j+1];
        if (s[i]>s[i-j]+j*bmax){
            s[i]=s[i-j]+j*bmax;
            l[i]=j;
```

```
            }
          }
        s[i]+=header;
        }
}

private static int length(int i)
{
    int k=1;
    i=i/2;
    while (i>0){
      k++;
      i=i/2;
      }
    return k;
}
```

3. 构造最优解

算法 compress 中用 $l[i]$和 $b[i]$记录了最优分段所需的信息。最优分段的最后一段的段长度和像素位数分别存储于 $l[n]$和 $b[n]$中。其前一段的段长度和像素位数存储于 $l[n-l[n]]$和 $b[n-l[n]]$中。依次类推，由算法计算出的 l 和 b 可在 $O(n)$时间内构造出相应的最优解。具体算法可实现如下：

```
private static void traceback(int n, int s[], int l[])
{
     if (n==0) return;
     traceback(n-l[n],s,l);
     s[m++]=n-l[n];
}

public static void output(int s[], int l[],int b[])
{
    int n=s.length-1;
    System.out.println("The optimal value is"+s[n]);
    m=0;
    traceback(n,s,l);
    s[m]=n;
    System.out.println("Decomposed into"+m+"segments");
    for (int j=1;j<=m;j++){
      l[j]=l[s[j]];
      b[j]=b[s[j]];
      }
    for (int j=1;j<=m;j++)
      System.out.println(l[j]+","+b[j]);
```