第 3 章

动态规划法

3.1 动态规划法概述

3.1.1 最优性原理

一个问题可以看作是一个前后关联的、具有链状结构的多阶段决策过程，如图 3-1 所示。

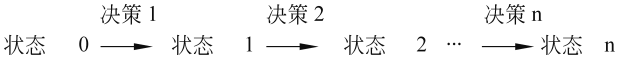

图 3-1 多阶段决策过程

在多阶段决策问题中，各阶段采取的决策一般与时间有关，决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，故有“动态”的含义。通常称这种解决多阶段决策最优化的过程为动态规划法。

最优性原理(principle of optimality)是指多阶段决策过程的最优决策序列具有这样的性质：不论初始状态和初始决策如何，对于前面决策所造成的某一状态而言，其后各阶段的决策序列必须构成最优策略。也就是说，不论前面的状态和决策如何，后面的最优策略只取决于由最初策略所确定的当前状态。若一个决策序列中包含有非局部最优的决策子序列时，该决策序列一定不是最优的。这个最优性原理是动态规划的基础。

3.1.2 动态规划法的基本步骤

动态规划法的基本步骤如下。

(1) 找出最优解的性质，并描绘其结构特征。

(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。

下面先看动态规划法的两个简单例子。

例 3-1 数塔问题：设有一个三角形数塔，如图 3-2 所示。求一条从三角形的塔顶到塔底

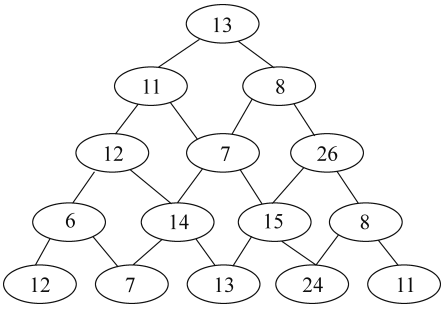

图 3-2 数塔示意图

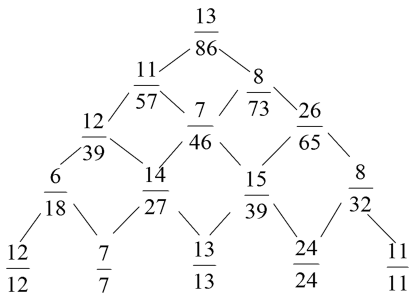

图 3-3　数塔问题求解过程

的路径，使该路径上结点的值之和最大。

数塔问题求解过程，如图 3-3 所示。

例 3-2　最小代价子母树：设有 n 堆沙子(n≥2)排成一排，每堆沙子的质量为 w。现要将 n 堆沙子归并成一堆，规则是：每次只能将相邻的两堆沙子归成一堆，经过 n－1 次归并之后成为一堆，其总代价为进行过程中新产生的沙堆的质量之和。这个代价最小的归并树为最小代价子母树。

解：当 n＝2 时，仅有 1 种归并方法；当 n＝3 时，有 2 种归并方法，如图 3-4 所示。

当 n＝4 时，有 5 种归并方法，如图 3-5 和图 3-6 所示。

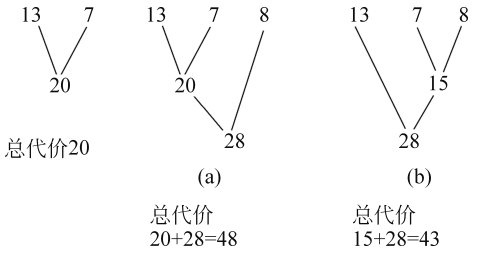

图 3-4　n＝2 与 n＝3 时的归并方法

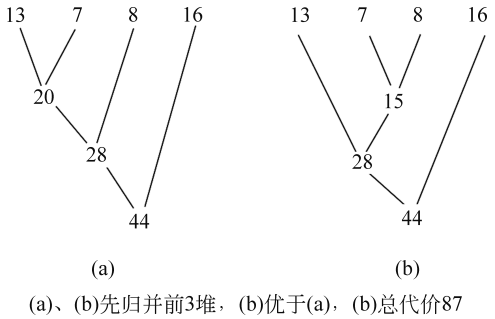

图 3-5　n＝4 时 5 种归并方法中的 2 种

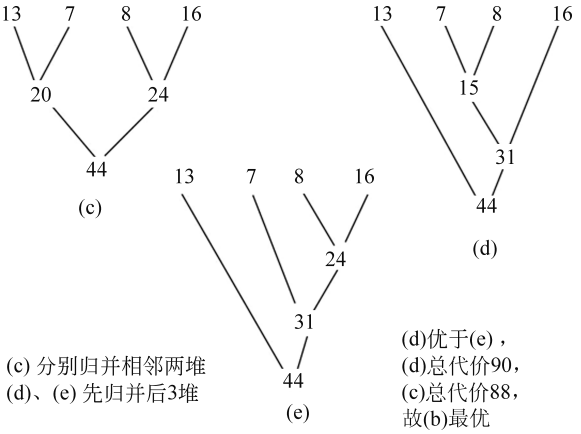

图 3-6　n＝4 时 5 种归并方法中的 3 种

定义 f(i,j)表示第 i 堆到第 j 堆沙子的归并最小代价，g(i,j)表示第 i 堆到第 j 堆沙子的质量和，则有

$$f(1,4)=\min\{f(1,3),f(1,2)+f(3,4),f(2,4)\}+g(1,4)$$

$$f(1,3)=\min\{f(1,2),f(2,3)\}+g(1,3)$$

$$f(1,2)=g(1,2)$$

$$f(3,4)=g(3,4)$$
$$f(2,3)=g(2,3)$$
$$f(2,4)=\min\{f(2,3),f(3,4)\}+g(2,4)$$

推广到一般情形：

$$f(1,n)=\min\{f(1,n-1),f(1,2)+f(3,n),\cdots f(1,3)+f(4,n),\cdots f(2,n)\}+g(1,n)$$
$$f(i,j)=\min\{f(i,j-1),f(i,i+1)+f(i+2,j),\cdots f(i,i+2)+f(i+3,j),\cdots f(i+1,j)\}+g(i,j)$$

将 f(1,n)用在 w=(13,7,8,16,21,4,18)上，得到如图 3-7 所示的最小代价子母树。

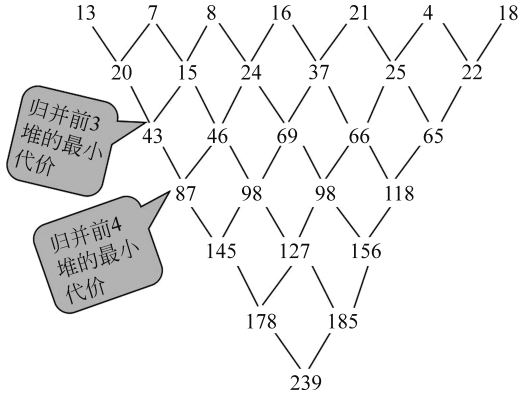

图 3-7　最小代价子母树

例 3-2 中最小代价子母树的动态规划算法实现的 C 程序如下：

```
#include <stdio.h>
#define N 7
int w[N+1]={0,13,7,8,16,21,4,18};
int g[N+1][N+1]={0},f[N+1][N+1]={0};
gg()
{
    int i,j,k,s;
    for (i=1;i<=N;i++)
        for (j=1;j<=N;j++)
        {
            if (i>j)
                continue;
            s=0;
            for(k=i;k<=j;k++)
                s=s+w[k];
            g[j][i]=g[i][j]=s;
        }
        for (i=1;i<=N;i++)
```

```
        {
            printf("\n");
            for (j=1;j<=N;j++)
                if (i>j)
                    printf("%11c",32);
                else
                    printf("g[%d][%d]=%-3d",i,j,g[i][j]);
         }
}

int mincpt(int start, int end)
{
    int i,j,x,min=0;
    for (i=start;i<=end;i++)
        f[i][i]=0;
    for (i=start;i<=end-1;i++)          /*计算相邻两堆石子合并的得分*/
        f[i][i+1]=w[i]+w[i+1];
    for (j=start+1;j<=end;j++)
        for (i=j-1 ;i>=1;i--)
        {
             min=f[i][j-1];
             for (x=i;x<=j-1;x++)
                 if (min >f[i][x]+f[x+1][j])
                     min=f[i][x]+f[x+1][j];
            f[i][j]=min+g[i][j];
        }
    return(f[start][end]);
}

void tree(int b[N+1][N+1])
{
    int i,j,k,a;
    printf("\n\n");
    for (i=1;i<=N;i++)
        printf("%7d",w[i]);
    for (i=1,a=1;i<=N;i++)
    {
        printf("\n");
        for (k=1;k<=i;k++)
            printf("%3c",32);
        for (k=1;k<=N-i;k++)
            printf("%6c%c",92,47);
        printf("\n");
```

```
            for (k=1;k<=i;k++)
                printf("%3c",32);
            for (j=1;j<=N-i;j++)
                printf("%7d",b[j][j+a]);
            a++;
        }
    }
    int main()
    {
        int i,j,b[N+1][N+1];
        gg();
        for (i=1;i<=N;i++)
            for (j=i+1;j<=N;j++)
                b[i][j]=mincpt(i,j);
        tree(b);
        return 0;
    }
```

3.2　矩阵连乘问题

3.2.1　问题描述

在科学计算中经常要计算矩阵的乘积。矩阵 A 和 B 可乘的条件是矩阵 A 的列数等于矩阵 B 的行数。若 A 是一个 p×q 的矩阵，B 是一个 q×r 的矩阵，则其乘积 C=AB 是一个 p×r 的矩阵。其标准计算公式为：

$$C_{ij}=\sum_{k=1}^{n}A_{ik}B_{kj},\quad 1\leqslant i\leqslant p,1\leqslant j\leqslant r$$

由该公式知计算 C=AB 总共需要 pqr 次的数乘。若 A 是一个 2×3 的矩阵，B 是一个 3×2 的矩阵，则其乘积 C=AB 是一个 2×2 的矩阵，计算 C 总共需要 2×3×2=12 次的数乘。

例如，$\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}\times\begin{bmatrix}7&8\\2&5\\3&4\end{bmatrix}=\begin{bmatrix}20&30\\56&81\end{bmatrix}$

计算两个矩阵乘积的代码如下：

```
int a[2][3]={1,2,3,4,5,6};
int b[3][2]={7,8,2,5,3,4};
int c[2][2]={0};
int p=2,q=3,r=2;
void MatrixMulti(int a[2][3],int b[3][2],int p,int q,int r)
{
```

```
    int i,j,k,sum;
    for (i=0;i<p;i++)
      for (j=0;j<r;j++)
      {
          c[i][j]=a[i][0]*b[0][j];
          for (k=1;k<q;k++)
              c[i][j]+=a[i][k]*b[k][j];
      }
}
```

矩阵连乘问题：给定 n 个矩阵$\{A_1,A_2,\cdots,A_n\}$，其中 A_i 与 A_{i+1} 是可乘的，i＝1，2，…，n－1。要求计算出这 n 个矩阵的连乘积 $A_1A_2\cdots A_n$。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，则可以通过反复调用两个矩阵相乘的标准算法来计算出矩阵连乘积。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) 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 $A_1A_2A_3\ A_4$ 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。例如，计算 3 个矩阵$\{A_1,A_2,A_3\}$的连乘积 $A_1\ A_2\ A_3$。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 10×100×5＋10×5×50＝7500 次的数乘。若按第二种加括号方式$(A_1(A_2A_3))$来计算，则需要的数乘次数为 100×5×50＋10×100×50＝75000。第二种加括号方式的计算量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，不同加括号方式导致不同的计算次序，对计算量的影响很大。

矩阵连乘积的最优计算次序问题描述为对于给定的相继 n 个矩阵$\{A_1,A_2,\cdots,A_n\}$(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 $A_1A_2\cdots A_n$ 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。

说明：计算两个均为 n×n 的矩阵(即 n 阶方阵)相乘还有一种 Strassen 矩阵乘法，利用分治思想将 2 个 n 阶矩阵乘积所需时间从标准算法的 $O(n^3)$ 改进到 $O(n^{\log 7})=O(n^{2.81})$。目前计算两个 n 阶方阵相乘最好的计算时间上界是 $O(n^{2.367})$。但无论如何，所需的乘法次数总随两个矩阵的阶而递增。在以上问题中只考虑采用标准公式计算两个矩阵的乘积。

解这个问题的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，然后找出最小者。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。所以P(n)的递推式如下：

$$P(n)=\begin{cases}1, & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k), & n\geqslant 2\end{cases}$$

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

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

也就是说，P(n)随着n的增长是指数增长的。因此，穷举搜索法不是一个有效算法。

下面来考虑用动态规划法求解矩阵连乘积的最优计算次序问题。此问题是动态规划的典型应用之一。

3.2.2　分析最优解的结构

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≤k<n，则完全加括号方式为$((A_1\cdots A_k)(A_{k+1}\cdots A_n))$。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]与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：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，这是一个矛盾。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。根据该问题的指标函数的特征也可以知道该问题满足最优化原理。另外，该问题显然满足无后向性，因为前面的矩阵链的计算方法和后面的矩阵链的计算方法无关。

3.2.3　建立递归关系

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]，原问题的最优值为m[1,n]。

① 当i=j时，$A_{i..j}=A_i$为单一矩阵，无须计算，因此m[i,i]=0，i=1,2,…,n；

② 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$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$。

由于在计算时我们并不知道断开点A的位置，所以A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：

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

m[i,j]给出了最优值,即计算 $A_{i..j}$所需的最少数乘次数。同时还确定了计算 $A_{i..j}$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] + $p_{i-1}p_kp_j$。若将对应于 m[i,j]的断开位置 k 记录在 s[i,j]中,则相应的最优解便可递归地构造出来。

3.2.4 计算最优值

根据 3.2.3 节中 m[i,j]的递归定义,容易编写一个递归程序来计算 m[1,n],但由于在递归计算时,许多子问题被重复计算多次,即使简单的递归计算也将导致耗费指数级计算时间。事实上,对于 1≤i≤j≤n,不同的有序对(i,j)对应于不同的子问题,不同的子问题个数最多只有 $\theta(n^2)$个,即

$$\binom{n}{2}+n=\theta(n^2)$$

该问题可以用动态规划算法求解。用动态规划算法解此问题,可依据 3.2.3 节中建立的递归关系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。在下面所给出的计算 m[i,j]的动态规划算法中,输入是序列 $P=\{p_0,p_1,\cdots,p_n\}$,输出除了最优值 m[i,j]外,还有使 m[i,j] = m[i,k] + m[k+1,j] + $p_{i-1}p_kp_j$ 达到最优的断开位置 k=s[i,j],1≤i≤j≤n。

设有 5 个矩阵,大小分别是 10×3、3×12、12×15、15×8、8×2,则有

```
int p[]={10,3,12,15,8,2};          //计算 5 个矩阵连乘
int n=5;
```

计算矩阵连乘的最优断开位置的代码如下:

```
int s[6][6];
void MatrixChain(int p[],int n)
{
    long m[6][6]={0};
    int i,r,j,k,t;
    for (i=1;i<=n;i++)
        m[i][i]=0;
    for (r=2;r<=n;r++)
    {
        for (i=1;i<=n-r+1;i++)
        {
            j=i+r-1;
            m[i][j]=m[i+1][j]+p[i-1]*p[i]*p[j];
```

```
                s[i][j]=i;
                for (k=i+1;k<j;k++)
                {
                    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;              \\s[i,j]记录计算 A[i..j]最优断开位置 k
                    }
                }
            }
        }
    }
```

该算法按照

```
m[1,1]
m[2,2] m[1,2]
m[3,3] m[2,3]m[1,3]
  ...
m[n,n] m[n-1,n] ...m[1,n]
```

的顺序计算 m[i,j]。该算法的计算时间上界为 $O(n^3)$，所占用的空间为 $O(n^2)$。由此可见，动态规划算法比穷举搜索法要有效得多。

3.2.5　构造最优解

算法 MatrixChain 只是计算出了最优值，即计算给定的矩阵连乘积所需的最少数乘次数，并未给出最优解。也就是说，通过 MatrixChain 的计算，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 告诉我们计算矩阵链 $A_{i..j}$ 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$(A_{1..k})(A_{k+1..n})$。因此，从 s[i,j]记录的信息可知计算 $A_{1..n}$ 的最优加括号方式为 $(A_{1..s[1,n]})(A_{s[1,n]+1..n})$，而计算 $A_{1..s[1,n]}$ 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 $A_{s[1,n]+1..n}$ 的最优加括号方式在 s[s[1,n]+1,n]处断开。照此递推下去，最终可以确定 $A_{s[1,n]+1..n}$ 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(i,j)是按 s 指示的加括号方式计算矩阵链 $A=\{A_1, A_2, \cdots, A_n\}$ 的子链 $A_{i..j}$ 的连乘积的算法。要计算 $A_{1..n}$ 只要调用 MatrixChain(p,n)和 Traceback(1,n)即可。

```
void Traceback(int i,int j)
{
    if (i==j) return;
     Traceback(i,s[i][j]);
     Traceback(s[i][j]+1,j);
     printf("\n(A[%d:%d])",i,s[i][j]);
     printf("*(A[%d:%d])",s[i][j]+1,j);
}
```

3.3 动态规划算法的基本要素

3.2 节 MatrixChain 算法的有效性依赖于问题本身所具有的三个重要性质：最优子结构性质、无后向性和子问题重叠性质。一般来说，问题所具有的这三个重要性质是该问题可用动态规划算法求解的基本要素，在设计求解具体问题的算法时，这对于是否选择动态规划算法具有指导意义。本节将着重研究最优子结构性质和子问题重叠性质以及动态规划法的一个变形——备忘录方法。

3.3.1 最优子结构

设计动态规划算法的第 1 步通常是要描述最优解的结构。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，若 $A_{1..n}$ 的最优完全加括号方式在 A_k 和 A_{k+1} 之间将矩阵链断开，则由该次序确定的子链 $A_{1..k}$ 和 $A_{k+1..n}$ 的完全加括号方式也是最优的，所以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，我们所用的方法具有普遍性。首先假设由问题的最优解导出的其子问题的解不是最优的，然后再设法证明在这个假设下可构造出一个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，问题的最优子结构性质使我们能够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。同时，它也使我们能在相对小的子问题空间中考虑问题。例如，在矩阵连乘积最优计算次序问题中，子问题空间是输入的矩阵链的所有不同的子链，它们的个数为 $\theta(n^2)$。因而子问题空间的规模仅为 $\theta(n^2)$。

3.3.2 重叠子问题

可用动态规划算法求解的问题应具备的另一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此问题时，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，对每一个子问题只求解一次，而后将其解保存在一个表格中，当再次需要求解此子问题时，只是简单地用常数时间查看一下结果。通常，不同的子问题的个数随输入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。

在计算矩阵连乘积最优计算次序时，利用 m[i,j]=m[i,k]+m[k+1,j]+$p_{i-1}p_kp_j$递归定义公式直接计算 $A_{i..j}$的递归算法。

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

用算法 RecurMatrixChain (p,1,4)计算 $A_{1..4}$的递归树如图 3-8 所示，许多子问题被重复计算。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：

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

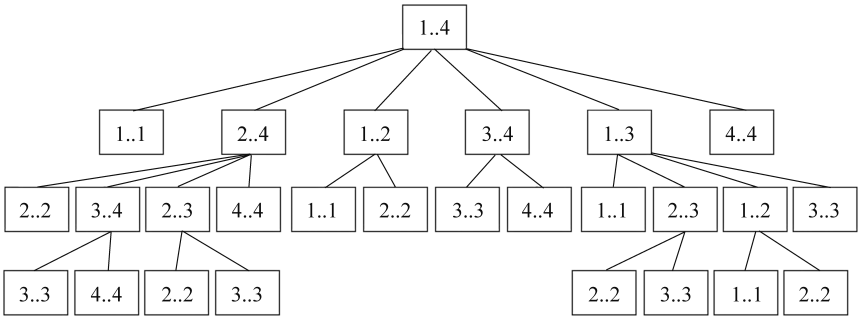

图 3-8　计算 $A_{1..4}$的递归树

因此，当 n>1 时，

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

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(p,1,n)的计算时间随 n 指数增长。相比之下，求解同一问题的动态规划算法只需计算时间 $O(n^2)$。动态规划算法的有效性就在于它充分利用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，不是重复计算多次。由此也可看出，当求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。

3.3.3 备忘录方法

动态规划算法的一个变形是备忘录方法。备忘录方法用一个表格来保存已解决的子问题的答案，当再碰到该子问题时，只要简单地查看该子问题的解答，而不必重新求解。与动态规划算法不同的是：备忘录方法采用自顶向下的递归方式，而动态规划算法采用自底向上的非递归方式。备忘录方法的控制结构与直接递归方法的控制结构相同，区别仅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，从而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的值，表示该子问题尚未求解。在求解过程中对遇到的每个子问题，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，此时需要对该子问题进行求解，并把得到的解保存在其相应的记录项中，以备以后查看。若记录项中存储的不是初始化时存入的特殊值，则表示该子问题已被求解过，其相应的记录项中存储的是该子问题的解答。此时，只要从记录项中读取子问题的解答，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的备忘录方法。

```
int MemoizedMatrixChain(int p[])
{
    n=length[p]-1;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            m[i,j]=0;
      return(LookupChain(p,1,n));
}
int LookupChain(int p[],int i,int j)
{
    if (m[i][j]>0)
        return(m[i][j]);
    if (i==j)
      m[i][j]=0;
    m[i][j]=LookupChain(p,i+1,j)+p[i-1]*p[i]*p[j];
    s[i][j]=i;
```

```
        for (k=i+1;k<j;k++)
        {
            q=LookupChain(p,i,k)+LookupChain(p,k+1,j)+p[i-1]*p[k]*p[j];
            if (q<m[i][j])
            {
                m[i][j]=q;
                s[i][j]=k;
            }
        }
        return(m[i][j]);
    }
```

与动态规划算法 MatrixChain 一样，备忘录算法 MemoizedMatrixChain 用数组 m[1…n，1…n]的单元 m[i,j]来记录解子问题 $A_{i..j}$ 的最优计算量。M[i,j]初始化为∞，表示相应于 $A_{i..j}$ 的子问题还未被计算。在调用 LookupChain(p,i,j)时，若 m[i,j]>0，则表示 m[i,j]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，LookupChain(p,i,j)总能正确返回 m[i,j]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。

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]，其 i=1,2,…,n ,j=i,i+1,…,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)。因此，LookupChain(p,1,n)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 $O(n^3)$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，再遇到该问题时，不重新求解，而是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。

通常，当一个问题的所有子问题都至少要解一次时，用动态规划算法解比用备忘录方法好。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，常常可利用其规则的表格存取方式，来减少在动态规划算法中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，该方法只求解那些确实需要求解的子问题。

3.4 最长公共子序列问题

3.4.1 问题描述

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 $Z=\{z_1,z_2,\cdots,z_k\}$是 X 的子序列是指存在一个严

格递增的下标序列 $<i_1,i_2,\cdots,i_k>$，使得对于所有 $j=1,2,\cdots,k$ 有 $X_{ij}=Z_j$。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 $X=\{A,B,C,B,D,A,B\}$ 和 $Y=\{B,D,C,A,B,A\}$，则序列 $\{B,C,A\}$ 是 X 和 Y 的一个公共子序列，序列 $\{B,C,B,A\}$ 也是 X 和 Y 的一个公共子序列。而且，后者是 X 和 Y 的一个最长公共子序列，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(Longest Common Sequence，LCS)问题描述为：给定两个序列 $X=\{x_1,x_2,\cdots,x_m\}$ 和 $Y=\{y_1,y_2,\cdots,y_n\}$，要求找出 X 和 Y 的一个最长公共子序列。

动态规划算法可有效地求解此问题。下面按照动态规划算法设计的各个步骤来设计一个求解此问题的有效算法。

3.4.2 最长公共子序列的结构

求解最长公共子序列问题时最容易想到的算法是穷举搜索法，即对 X 的每一个子序列，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，并且在检查过程中选出最长公共子序列。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 的一个子序列相应于下标序列 $\{1,2,\cdots,m\}$ 的一个子序列，因此，X 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，最长公共子序列具有最优子结构性质。

定理：最长公共子序列具有最优子结构性质。设序列 $X=\{x_1,x_2,\cdots,x_m\}$ 和 $Y=\{y_1,y_2,\cdots,y_n\}$ 的一个最长公共子序列 $Z=\{z_1,z_2,\cdots,z_k\}$，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。

(2) 由于 $z_k\neq x_m$，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 和 Y 的一个长度大于 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) 证明过程与(2)类似，略。

上述定理说明，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，最长公共子序列具有最优子结构性质。

3.4.3 子问题的递归结构

由最长公共子序列具有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，可按以下方式递归地进行：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，必须求解两个子问题，即找出 X_{m-1} 和 Y 的最长公共子序列及 X 和 Y_{n-1}的最长公共子序列。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，可能要计算出 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。

与矩阵连乘积最优计算次序问题类似，我们来建立子问题的最优值的递归关系。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 c[i,j]=0。其他情况下，由定理可建立 c[i,j]的递归关系如下：

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

3.4.4 计算最优值

直接利用 c[i,j]的递归关系式容易写出一个计算 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，总共只有 θ(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 LCS_LENGTH(X,Y)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[0..m ,0..n]和 b[1..m ,1..n]，其中 c[i,j]存储 X_i 与 Y_j 的最长公共子序列的长度，b[i,j]记录指示 c[i,j]的值是由哪一个子问题的解达到的，这在构造最长公共子序列时要用到。最后，X 和 Y 的最长公共子序列的长度记录于 c[m,n]中。

```
void LCS_LENGTH(int m,int n,int x[ ],int y[ ],int c[ ][ ],int b[ ][ ])
{
    int i,j;
    for (i=1;i<=m;i++)
        c[i][0]=0;
    for (j=1;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]='←'
                }
        }
}
```

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

3.4.5 构造最长公共子序列

由算法 LCS_LENGTH 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，沿着其中的箭头所指的方向在数组 b 中搜索。当 b[i,j]中遇到“↖”时，表示 X_i 与 Y_j 的最长公共子序列是由 X_{i-1}与 Y_{j-1}的最长公共子序列在尾部加上 x_i 得到的子序列；当 b[i,j]中遇到“↑”时，表示 X_i 与 Y_j 的最长公共子序列和 X_{i-1}与 Y_j 的最长公共子序列相同；当 b[i,j]中遇到“←”时，表示 X_i 与 Y_j 的最长公共子序列和 X_i 与 Y_{j-1}的最长公共子序列相同。

下面的算法 LCS(b,X,i,j)实现根据 b 的内容打印出 X_i 与 Y_j 的最长公共子序列。通过算法调用 LCS(b,X,length[X],length[Y])，便可打印出序列 X 和 Y 的最长公共子序列。

```
void LCS(int b[][],int x[ ],int i,int j)
{
    if (i==0 || j==0)
        return;
    if (b[i,j]=='↖')
    {
        LCS(b,x,i-1,j-1);
        print("%d",x[i]);          //打印 x[i]
    }
    else
        if (b[i,j]=='↑')
```

```
            LCS(b,x,i-1,j);
        else
            LCS(b,x,i,j-1);
    }
```

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

例如，设所给的两个序列为 X=<A,B,C,B,D,A,B>和 Y=<B,D,C,A,B,A>。由算法 LCS_LENGTH 和 LCS 计算出的结果如图 3-9 所示。

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	5

图 3-9　算法 LCS 的计算结果

3.4.6　算法的改进

对于一个具体问题，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可以改进。这种改进，通常是利用具体问题的一些特殊性。

例如，在算法 LCS_LENGTH 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]三个值之一确定，而数组元素 b[i,j]也只是用来指示 c[i,j]究竟由哪个值确定。因此，在算法 LCS 中，可以不借助于数组 b 而借助于数组 c 本身临时判断 c[i,j]的值是由 c[i−1,j−1]，c[i−1,j]和 c[i,j−1]中哪一个数值元素所确定，代价是 O(1)时间。既然 b 对于算法 LCS 不是必要的，那么算法 LCS_LENGTH 便不必保存它。这一来，可节省 $\theta(mn)$的空间，而 LCS_LENGTH 和 LCS 所需要的时间分别仍然是 O(mn)和 O(m+n)。不过，由于数组 c 仍需要 O(mn)的空间，因此这里所做的，只是在空间复杂度的常数因子上的改进。

另外，如果只需要计算最长公共子序列的长度，则算法的空间需求还可大大减少。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，只要用 2 行的数组空间就可以计算出最长公共子序列的长度。更进一步的分析还可将空间需求减至 min(m,n)。

3.5　凸多边形的最优三角剖分问题

3.5.1　问题描述

多边形是平面上一条分段线形成的闭曲线，是由一系列首尾相接的直线段组成的。组成多边形的各直线段称为该多边形的边。多边形相接两条边的连接点称为多边形的顶点。若多边形的边之间除了连接顶点外没有别的公共点，则称该多边形为简单多边形。一个简单多边形将平面分为 3 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一个闭凸集时，称该简单多边形为凸多边形。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的点均在该凸多边形的内部或边界上。

通常，用多边形顶点的顺时针序列来表示一个凸多边形，即 $P=<v_0, v_1, \cdots, v_{n-1}>$ 表示具有 n 条边 $v_0v_1, v_1v_2, \cdots, v_{n-1}v_n$ 的一个凸多边形，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。弦将多边形分割成两个子凸多边形 $<v_i, v_{i+1}, \cdots, v_j>$ 和 $<v_j, v_{j+1}, \cdots, v_i>$。多边形的三角剖分是一个将多边形分割成互不重叠的三角形的弦的集合 T。图 3-10 是一个凸多边形的两个不同的三角剖分。

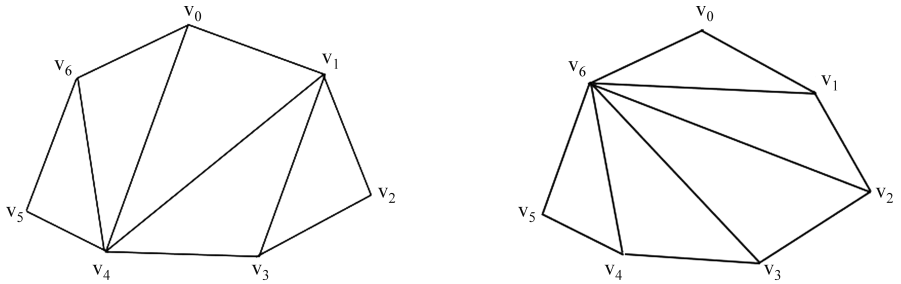

图 3-10　一个凸多边形的两个不同的三角剖分

在凸多边形 P 的一个三角剖分 T 中，各弦互不相交且弦数已达到最大，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一个三角剖分中，恰好有 n−3 条弦和 n−2 个三角形。

凸多边形最优三角剖分的问题是：给定一个凸多边形 $P=<v_0, v_1, \cdots, v_{n-1}>$ 以及定义在由多边形的边和弦组成的三角形上的权函数 ω，要求确定该凸多边形的一个三角剖分，使得该三角剖分对应的权（即剖分中诸三角形上的权）之和为最小。

可以定义三角形上各种各样的权函数 W。例如，定义 $\omega(\Delta v_iv_jv_k)=|v_iv_j|+|v_iv_k|+|v_kv_j|$，其中，$|v_iv_j|$ 是点 v_i 到 v_j 的欧几里得距离。相应于此权函数的最优三角剖分即为最小弦长三角剖分。

注意：*解决此问题的算法必须适用于任意的权函数。*

用动态规划算法也能有效地求解凸多边形的最优三角剖分问题。尽管这是一个计算几何学问题，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。

3.5.2　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。正如所看到过的，矩阵连乘积的最优计算次序问题等价于矩阵链的完全加括号方式。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。这里的所谓完全二叉树是指叶结点以外的所有结点的度数都为 2 的二叉树（注意与满二叉树和近似满二叉树的区别）。

一个表达式的完全加括号方式对应于一棵完全二叉树，称这棵二叉树为表达式的语法树。例如，与完全加括号的矩阵连乘积$((A_1(A_2A_3))(A_4(A_5A_6)))$相对应的语法树如图3-11(a)所示。

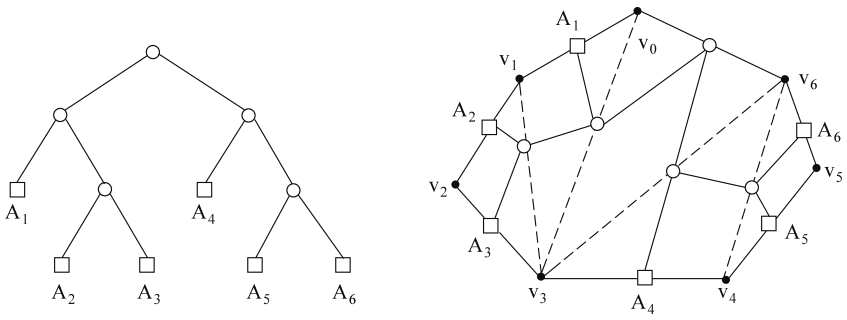

图3-11　表达式语法树与三角剖分的对应

语法树中每一个叶子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来表示。例如，图3-11(a)中凸多边形的三角剖分可用图3-11(b)所示的语法树来表示。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部结点。多边形中除v_0v_6边外的每一条边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3个部分：三角形$v_0v_3v_6$、凸多边形$<v_0,v_1,\cdots,v_3>$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_3v_6和v_0v_3为根的两个儿子。以它们为根的子树分别表示凸多边形$<v_0,v_1,\cdots,v_3>$和凸多边形$<v_3,v_4,\cdots,v_6>$的三角剖分。

在一般情况下，一个凸n边形的三角剖分对应于一棵有n－1个叶子的语法树。反之，也可根据一棵有n－1个叶子的语法树产生相应的一个凸n边形的三角剖分。也就是说，凸n边形的三角剖分与n－1个叶子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个叶子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)边形的三角剖分之间存在一一对应关系。图3-11(a)和图3-11(b)表示出了这种对应关系，这时n＝6。矩阵连乘积$A_1A_2..A_6$中的每个矩阵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$v_iv_j(i<j-1)$，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一个特殊情形。对于给定的矩阵链$A_1A_2..A_n$，定义一个与之相应的凸(n+1)边形$P=<v_0,v_1,\cdots,v_n>$，使得矩阵A_i与凸多边形的边$v_{i-1}v_i$一一对应。若矩阵A_i的维数为$p_{i-1}\times p_i$，i＝1，2，…，n，则定义三角形$v_iv_jv_k$上的权函数值为$\omega(\Delta v_iv_jv_k)=p_ip_jp_k$。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了矩阵链$A_1A_2..A_n$的最优完全加括号方式。

3.5.3 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 $P=<v_0,v_1,\cdots,v_n>$ 的一个最优三角剖分 T 包含三角形 $v_0v_kv_n$，其中 $1\leqslant k\leqslant n-1$，则 T 的权为 3 个部分权之和，即三角形 $v_0v_kv_n$ 的权，子多边形 $<v_0,v_1,\cdots,v_k>$ 的权与 $<v_k,v_{k+1},\cdots,v_n>$ 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，将会导致 T 不是最优三角剖分。

3.5.4 最优三角剖分对应的权的递归结构

首先，定义 t[i,j]，其中 $1\leqslant i<j\leqslant n$，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值，即最优值。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)边多边形 P 对应的权的最优值为 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，其中 i=1,2,…,n。当 $j-i\geqslant 1$ 时，子多边形 $<v_{i-1},v_i,\cdots,v_j>$ 至少有 3 个顶点。由最优于结构性质，t[i,j]的值应为 t[i,k]的值加上 t[k+1,j]的值，再加上 $\Delta v_{i-1}v_kv_j$ 的权值，并在 $i\leqslant k\leqslant j-1$ 的范围内取最小。由此，t[i,j]可递归地定义为：

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

3.5.5 计算最优值

将上式与矩阵连乘积的最优计算次序问题中计算 m[i,j]的递归定义公式进行比较容易看出，除了权函数的定义外，两个递归式是完全一样的。因此只要对计算 m[i,j]的算法 MatrixChain 做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 $P=<v_0,v_1,\cdots,v_n>$ 的三角剖分最优权值的动态规划算法 Minweigh_triangle，输入是凸多边形 $P=<v_0,v_1,\cdots,v_n>$ 的权函数 ω，输出是最优值 t[i,j]和使得 $t[i,k]+t[k+1,j]+\omega(\Delta v_{i-1}v_kv_j)$ 达到最优的位置(k=)s[i,j]，其中 $1\leqslant i\leqslant j\leqslant n$。

```
void Minweigh_triangle(int p[])
{
    n=length[p]-1;
    for (i=1;i<=n;i++)
        t[i][i]=0;
    for (r=2;r<=n;r++)
        for (i=1;i<=n-r+1;i++)
```