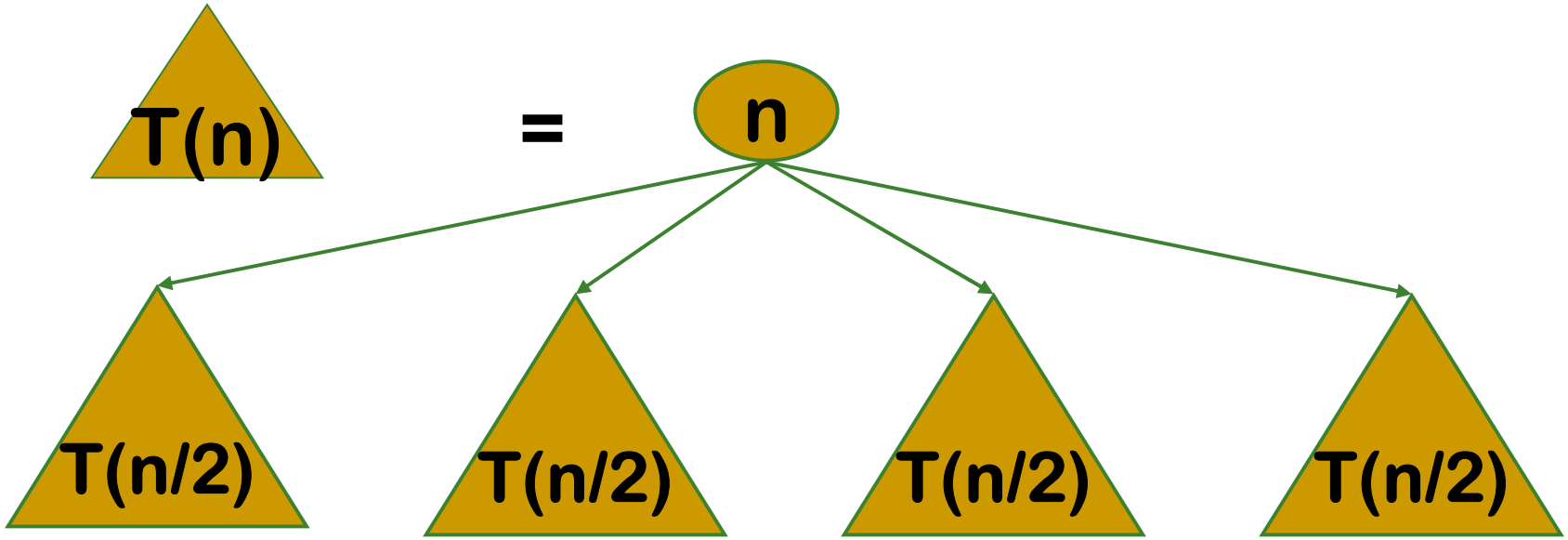
第3章　动态规划

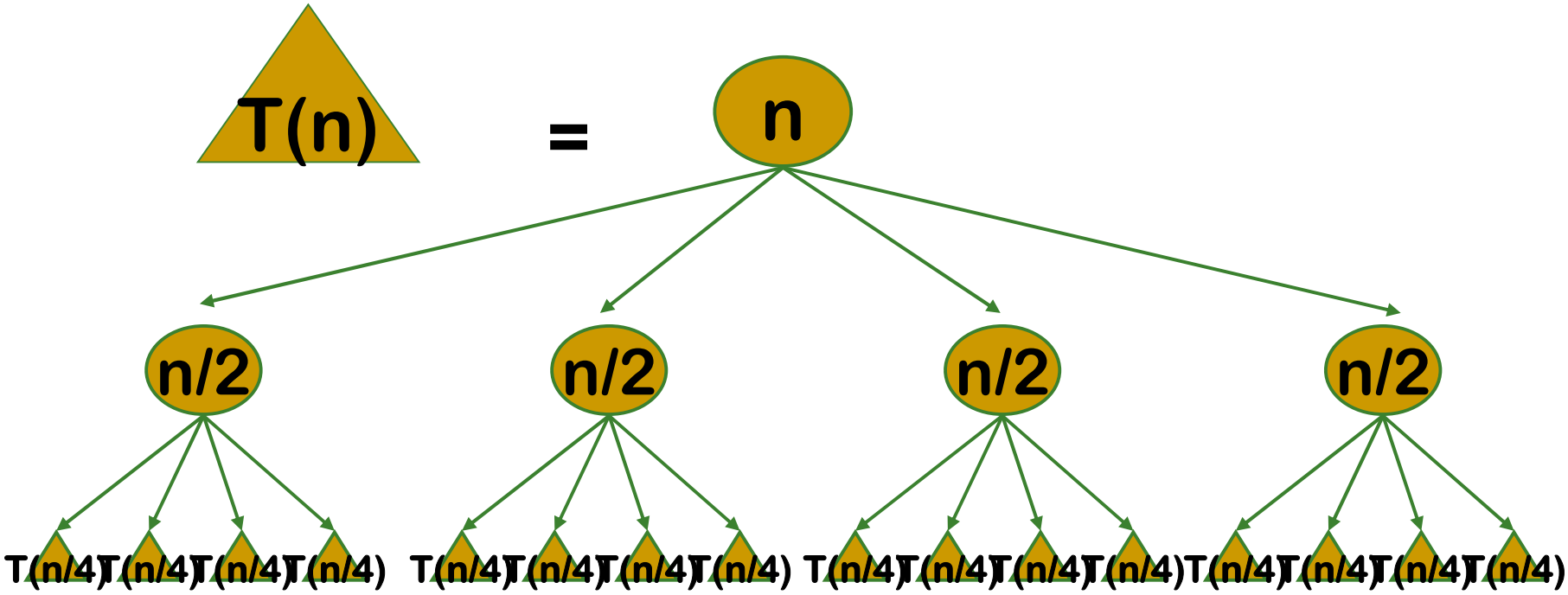
算法总体思想

- 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



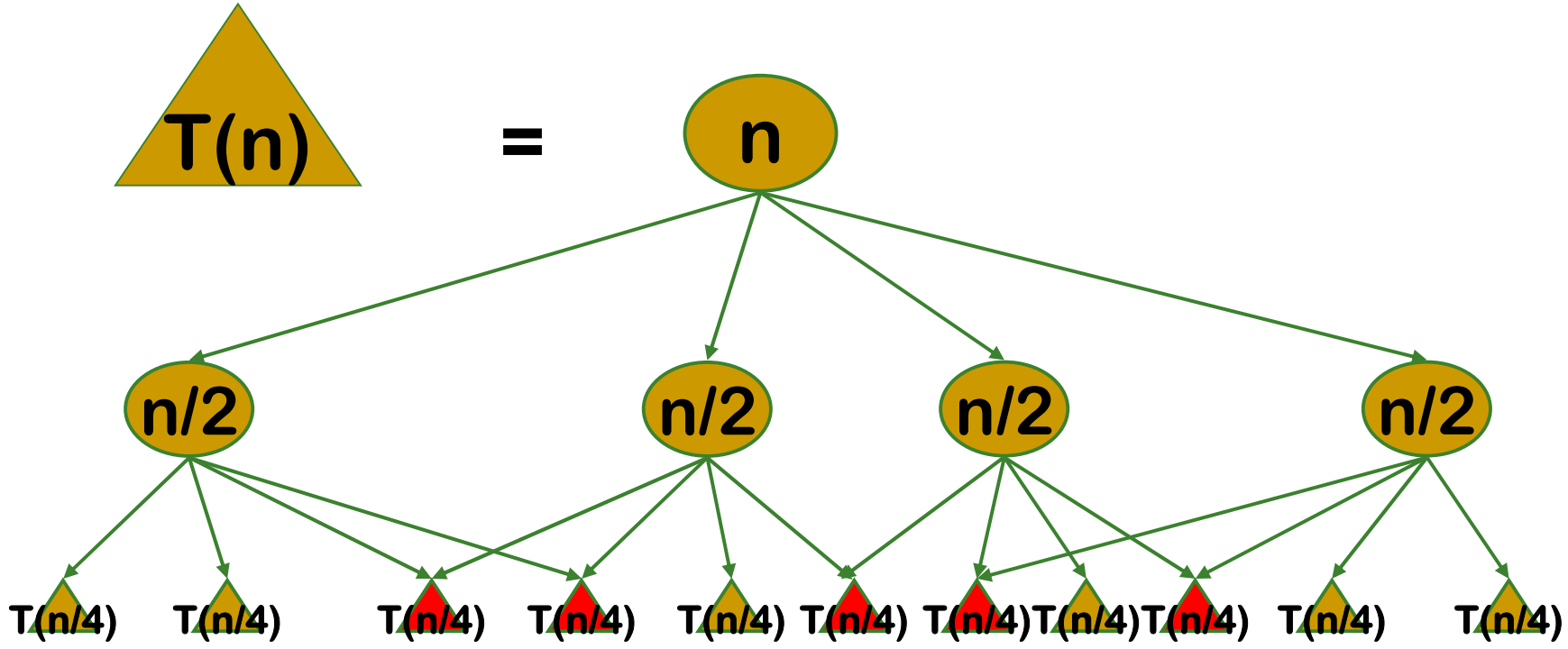
算法总体思想

- 但是经分解得到的子问题往往不是互相独立的。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。



算法总体思想

- 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。



矩阵连乘问题

- 给定n个矩阵$\{A_1, A_2, \ldots, A_n\}$，其中$A_i$与$A_{i+1}, i = 1,2, \ldots, n-1$，是可乘的。考察这n个矩阵的连乘积

$$A_1 A_2 \ldots A_n$$

- 由于矩阵乘法满足结合律，所以计算矩阵的连乘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。

矩阵连乘问题

- 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A是完全加括号的，则A可表示为2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) 。

矩阵连乘问题

- 设有四个矩阵A,B,C,D，它们的维数分别是：

  A=50×10　　B=10×40　　C=40×30　　D=30×5

- 总共有五种完全加括号的方式

(A((BC)D))　(A(B(CD)))　((AB)(CD))　(((AB)C)D)　((A(BC))D)

- 计算所需要的乘法次数分别为：

  16000，10500，36000，87500，34500

给定n个矩阵$\{A_1, A_2, \ldots, A_n\}$，其中$A_i$与$A_{i+1}$是可乘的，$i = 1,2, \ldots, n-1$。如何确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的数乘次数最少。

矩阵连乘问题

◆穷举法：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。

算法复杂度分析：

对于n个矩阵的连乘积，设其不同的计算次序为P(n)。

由于每种加括号方式都可以分解为两个子矩阵的加括号问题：

$(A_1 \ldots A_k)(A_k+1 \ldots A_n)$,可以得到关于P(n)的递推式如下：

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & n>1\end{cases} \Rightarrow P(n)=\Omega(4^n/n^{3/2})$$

矩阵连乘问题

- ◆穷举法
- ◆动态规划

将矩阵连乘积 $A_i A_{i+1} \ldots A_j$ 简记为A[i:j]，这里i≤j

考察计算A[i:j]的最优计算次序。设这个计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，i≤k<j，则其相应完全加括号方式为 $(A_i A_{i+1} \ldots A_k)(A_{k+1} A_{k+2} \ldots A_j)$

计算量：A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k]和A[k+1:j]相乘的计算量

分析最优解的结构

- 特征：计算A[i:j]的最优次序所包含的计算矩阵子链A[i:k]和A[k+1:j]的次序也是最优的。
- 矩阵连乘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。

建立递归关系

设计算A[i:j]，1≤i≤j≤n，所需要的最少数乘次数为m[i,j]，则原问题的最优值为m[1,n]

- 当i=j时，A[i:j]=Ai，因此，m[i,i]=0，i=1,2,…,n
- 当i<j时，$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$

这里 A_i 的维数为 $p_{i-1}\times p_i$

- 可以递归地定义m[i,j]为：

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

k 的位置只有 $j-i$ 种可能

用动态规划法求最优解

```
public static void matrixChain(int [] p, int [][] m, int [][] s) {
//将m[i][j]的断开位置k记为s[i][j]，可递归地由s[i][j]构造出相应的最优解。
    int n=p.length-1;
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++) //r为链长
          for (int i = 1; i <= n - r+1; i++) //i 左边界
               int j = i+r-1;                //j 右边界
               m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j]; //k=i
               s[i][j] = i;
               for (int k = i+1; k < j; k++) //遍历所有划分
                     int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                     if (t < m[i][j]) //更新解
                           m[i][j] = t;
                           s[i][j] = k;
}
```

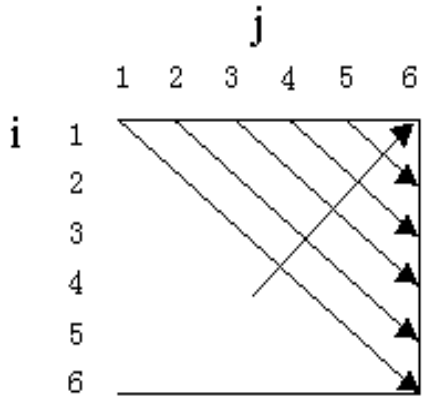
初值

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

用动态规划法求最优解

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

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



(a) 计算次序

j

i	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]

j

i	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]

动态规划算法的基本要素

一、最优子结构

- 矩阵连乘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**。
- 在分析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。

动态规划算法的基本要素

一、最优子结构

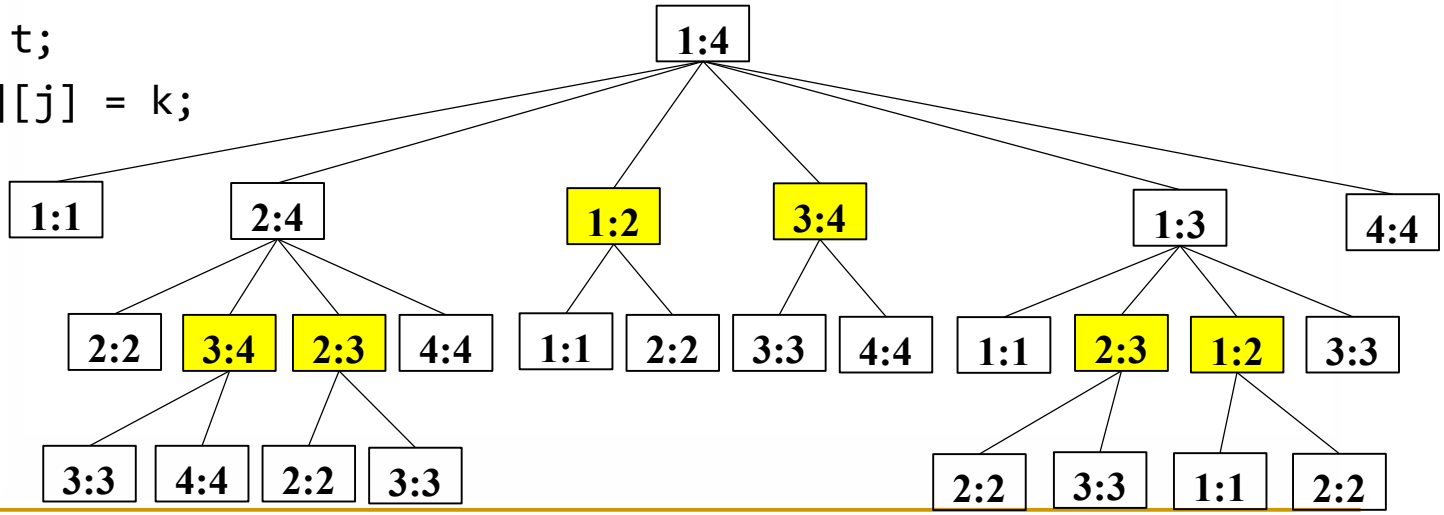
- 利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的前提。

注意：同一个问题可以有多种方式刻划它的最优子结构，有些表示方法的求解速度更快（空间占用小，问题的维度低）

二、重叠子问题

计算矩阵连乘积最优计算次序的递归算法

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



二、重叠子问题

递归算法的计算时间：

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

当n>1时，

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

证明得：$T(n)\geq 2^{n-1}=\Omega(2^n)$

二、重叠子问题

动态规划算法复杂度分析：

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$O(n^2)$。

```
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;
}
```

二、重叠子问题

- 对于1≤i≤j≤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$\theta(n^2)$

- 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。

二、重叠子问题

- 递归算法求解问题时，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。这种性质称为**子问题的重叠性质**。
- 动态规划算法，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只需要多项式时间，从而获得较高的解题效率。

三、备忘录方法

•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。

```
m←0
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;
}
```

动态规划基本步骤

- 找出最优解的性质，并刻划其结构特征。
- 递归地定义最优值。
- 以自底向上的方式计算出最优值。
- 根据计算最优值时得到的信息，构造最优解。

最长公共子序列

- DNA序列分析

X先生找到了其失散多年的兄弟Y先生。为了确定血缘关系，X先生决定做DNA鉴定，比较两组基因，

X先生基因片段 {A,C,T,C,C,T,A,…,G},

Y先生基因片段 {C,A,T,T,C,A,G,…,C},

找出两人基因片段中最长相同的部分 ⟶ 最长公共子序列

子序列

设序列X，Z，

$$X = \{x_1, x_2, \dots, x_m\},$$

$$Z = \{z_1, z_2, \dots, z_k\},$$

若存在X的元素构成的严格递增下标序列$\{i_1, i_2, \dots, i_k\}$使得

$$z_j = x_{i_j}, \quad j = 1,2, \dots, k$$

则称Z是X的子序列

X与Y的公共子序列Z：Z是X和Y的子序列

子序列的长度：子序列的元素个数

最长公共子序列

问题：给定序列

$X = \{x_1, x_2, \ldots, x_m\}$,

$Y = \{y_1, y_2, \ldots, y_n\}$,

求X和Y的最长公共子序列

提问：穷举算法求解的复杂度?

最坏情况下：$O(m2^n)$

实例：

$$X = \{A, T, C, T, G, A, T\}$$

$$Y = \{T, G, C, A, T, A\}$$

最长公共子序列：T，C，T，A，长度4

子问题界定

- 参数i和j界定子问题
- X的终止位置是i，Y的终止位置是j
- $X_i = \{x_1, x_2, \ldots, x_i\}$，　$Y_j = \{y_1, y_2, \ldots, y_j\}$

子问题间的依赖关系

- 设$X=\{x_1,x_2,\dots,x_m\}$，$Y=\{y_1,y_2,\dots,y_n\}$ $Z=\{z_1,z_2,\dots,z_k\}$为X和Y的LCS，那么

(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。

满足优化原则和子问题的重叠性

优化函数的递推方程

令X与Y的子序列

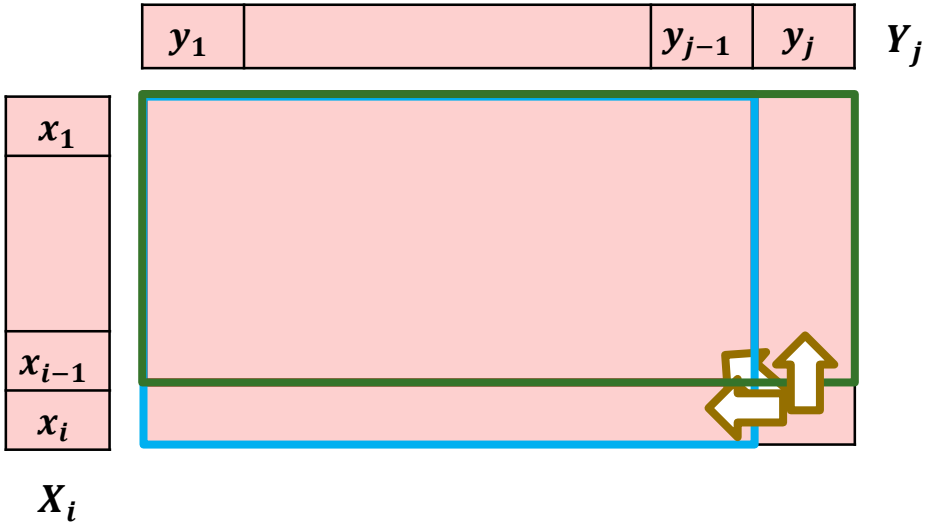
$X_i = \{x_1, x_2, \dots, x_i\}$，$Y_j = \{y_1, y_2, \dots, y_j\}$

$c[i][j]$: X_i与Y_j的LCS的长度

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

标记函数

- 标记函数$B[i][j]$，值为↖、←、↑
- $c[i][j]=c[i-1][j-1]+1$:↖　　$c[i][j]=c[i][j-1]$:←
- $c[i][j]=c[i-1][j]$:↑



实例—优化函数和标记函数

构建C[*i*, *j*]、 B[*i*, *j*]数组

C[1, 1]: $x_1 \neq y_1$, 即A ≠ T

C[1, 1]= max{C[0, 1],

C[1, 0]}

= C[0, 1]=0

$B[1,1] = \uparrow$

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

	Y	0	T	G	C	A	T	A
X	[*i*,*j*]	*j* = 0	*j* = 1	*j* = 2	*j* = 3	*j* = 4	*j* = 5	*j* = 6
0	*i* = 0	0	0	0	0	0	0	0
A	*i* = 1	0	C[1,1] B[1,1]					
T	*i* = 2	0						
C	*i* = 3	0						
T	*i* = 4	0						
G	*i* = 5	0						
A	*i* = 6	0						
T	*i* = 7	0						

实例—优化函数和标记函数

构建C[i, j]、 B[i, j]数组

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

C[1, 2]: $x_1 \neq y_2$, 即A ≠ G

C[1, 2]= max{C[0, 2],

C[1, 1]}

=C[0, 2]=0

$B[1, 2]$ =↑

		Y	0	T	G	C	A	T	A
	X	[i, j]	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
0		$i=0$	0	0	0	0	0	0	0
A		$i=1$	0	0, ↑	C[1,2] B[1,2]				
T		$i=2$	0						
C		$i=3$	0						
T		$i=4$	0						
G		$i=5$	0						
A		$i=6$	0						
T		$i=7$	0						

实例—优化函数和标记函数

构建C[*i*, *j*]、 B[*i*, *j*]数组

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

		Y	0	T	G	C	A	T	A
X	[*i*,*j*]	*j* = 0	*j* = 1	*j* = 2	*j* = 3	*j* = 4	*j* = 5	*j* = 6
0	*i* = 0	0	0	0	0	0	0	0
A	*i* = 1	0	0, ↑	0, ↑	C[1,3] B[1,3]			
T	*i* = 2	0						
C	*i* = 3	0						
T	*i* = 4	0						
G	*i* = 5	0						
A	*i* = 6	0						
T	*i* = 7	0						

C[1, 3]: $x_1 \neq y_3$, 即A ≠ C

C[1, 3]= max{C[0, 3],

C[1, 2]}

=C[0, 3]=0

$B[1,3]$ =↑

实例—优化函数和标记函数

构建C[*i*, *j*]、 B[*i*, *j*]数组

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

	Y	0	T	G	C	A	T	A
X	[*i*,*j*]	*j* = 0	*j* = 1	*j* = 2	*j* = 3	*j* = 4	*j* = 5	*j* = 6
0	*i* = 0	0	0	0	0	0	0	0
A	*i* = 1	0	0, ↑	0, ↑	0, ↑	C[1,4] B[1,4]		
T	*i* = 2	0						
C	*i* = 3	0						
T	*i* = 4	0						
G	*i* = 5	0						
A	*i* = 6	0						
T	*i* = 7	0						

C[1, 4]: $x_1 = y_4$, 即A = A

C[1, 4]= C[0, 3] + 1 = 1

$B[1,4] = \nwarrow$

实例—优化函数和标记函数

构建C[*i*, *j*]、 B[*i*, *j*]数组

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

	Y	0	T	G	C	A	T	A
X	[*i*, *j*]	*j* = 0	*j* = 1	*j* = 2	*j* = 3	*j* = 4	*j* = 5	*j* = 6
0	*i* = 0	0	0	0	0	0	0	0
A	*i* = 1	0	0, ↑	0, ↑	0, ↑	1, ↖		
T	*i* = 2	0						
C	*i* = 3	0						
T	*i* = 4	0						
G	*i* = 5	0						
A	*i* = 6	0						
T	*i* = 7	0						

以此类推

……

实例

构建C[i,j]、 B[i,j]数组

输入：$X=\{A,\ T,\ C,\ T,\ G,\ A,\ T\}$

$Y=\{T,\ G,\ C,\ A,\ T,\ A\}$

	Y	0	T	G	C	A	T	A
X	[i,j]	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
0	$i=0$	0	0	0	0	0	0	0
A	$i=1$	0	0, ↑	0, ↑	0, ↑	1, ↖	1, ←	1, ↖
T	$i=2$	0	1, ↖	1, ←	1, ←	1, ↑	2, ↖	2, ←
C	$i=3$	0	1, ↑	1, ↑	2, ↖	2, ←	2, ↑	2, ↑
T	$i=4$	0	1, ↖	1, ↑	2, ↑	2, ↑	3, ↖	3, ←
G	$i=5$	0	1, ↑	2, ↖	2, ↑	2, ↑	3, ↑	3, ↑
A	$i=6$	0	1, ↑	2, ↑	2, ↑	3, ↖	3, ↑	4, ↖
T	$i=7$	0	1, ↖	2, ↑	2, ↑	3, ↑	4, ↖	4, ↑

实例—构建最优解

输入：$X=\{A,\ T,\ C,\ T,\ G,\ A,\ T\}$

$Y=\{T,\ G,\ C,\ A,\ T,\ A\}$

		T	G	C	A	T	A
		1	2	3	4	5	6
A	1	B[1][1] =↑	B[1][2] =↑	B[1][3] =↑	B[1][4] =↖	B[1][5] =←	B[1][6] =↖
T	2	B[2][1] =↖	B[2][2] =←	B[2][3] =←	B[2][4] =↑	B[2][5] =↖	B[2][6] =←
C	3	B[3][1] =↑	B[3][2] =↑	B[3][3] =↖	B[3][4] =←	B[3][5] =↑	B[3][6] =↑
T	4	B[4][1] =↖	B[4][2] =↑	B[4][3] =↑	B[4][4] =↑	B[4][5] =↖	B[4][6] =←
G	5	B[5][1] =↑	B[5][2] =↖	B[5][3] =↑	B[5][4] =↑	B[5][5] =↑	B[5][6] =↑
A	6	B[6][1] =↑	B[6][2] =↑	B[6][3] =↑	B[6][4] =↖	B[6][5] =↑	B[6][6] =↖
T	7	B[7][1] =↖	B[7][2] =↑	B[7][3] =↑	B[7][4] =↑	B[7][5] =↖	B[7][6] =↑

自顶向下设计

自底向上求解

解：$X[2],X[3],X[4],X[6]$，即T，C，T，A

伪码

算法 $\boldsymbol{lcsLength}(X, Y, m, n)$

```
1: for i ← 1 to m do c[i][0] ← 0;
2: for i ← 1 to n do c[0][i] ← 0;
3: for (int i = 1; i <= m; i++)
4:     for (int j = 1; j <= n; j++)
5:         if (X[i] == Y[j])
6:             c[i][j] = c[i − 1][j − 1] + 1;
7:             b[i][j] = " ↖ ";
8:         else if (c[i − 1][j] >= c[i][j − 1])
9:                 c[i][j] = c[i − 1][j];
10:                b[i][j] = " ↑ ";
11:            else
12:                c[i][j] = c[i][j − 1];
13:                b[i][j] = " ← ";
```

初值

子问题 X_i, Y_j

追踪解

算法 Structure Sequence(B, i, j)
输入：$B[i][j]$
输出：X与Y的最长公共子序列
1: ***if*** $(i == 0 \,||\, j == 0)$ ***return*** //序列为空
2: ***if*** $(b[i][j] ==$ " ↖ ")
3: 输出$X[i]$;
4: **Structure Sequence$(B, i-1, j-1)$**
5: ***else if*** $(b[i][j] ==$ "↑")
6: **Structure Sequence$(B, i-1, j)$**
7: ***else*** **Structure Sequence$(B, i, j-1)$**
8: }

算法的时空复杂度

计算优化函数和标记函数：

赋初值，为$O(m)+O(n)$

计算优化、标记函数迭代$\Theta(mn)$次

循环体内常数次运算，时间为$\Theta(mn)$

构造解：

每步缩小X或Y的长度，时间$\Theta(m+n)$

算法时间复杂度： $\Theta(mn)$

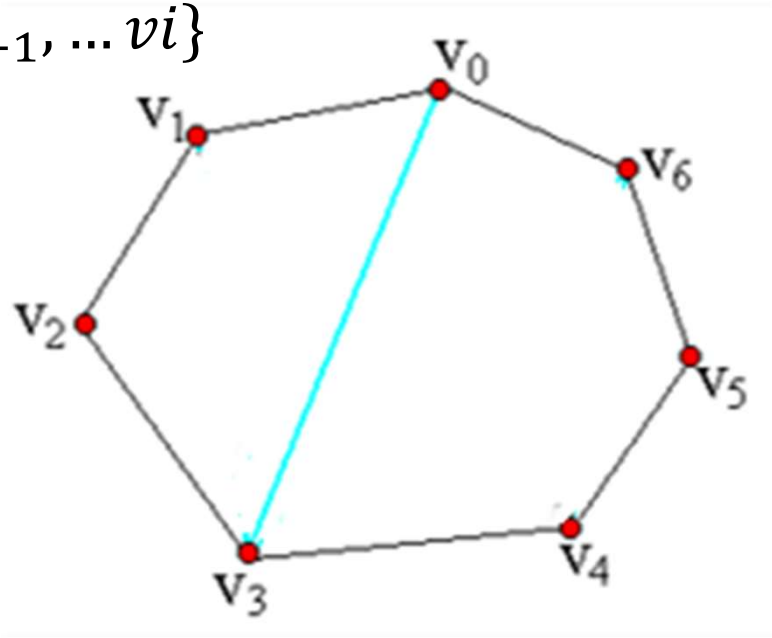
空间复杂度： $\Theta(mn)$

算法的改进

- 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。
- 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2行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至O(min(m,n))。

凸多边形最优三角剖分

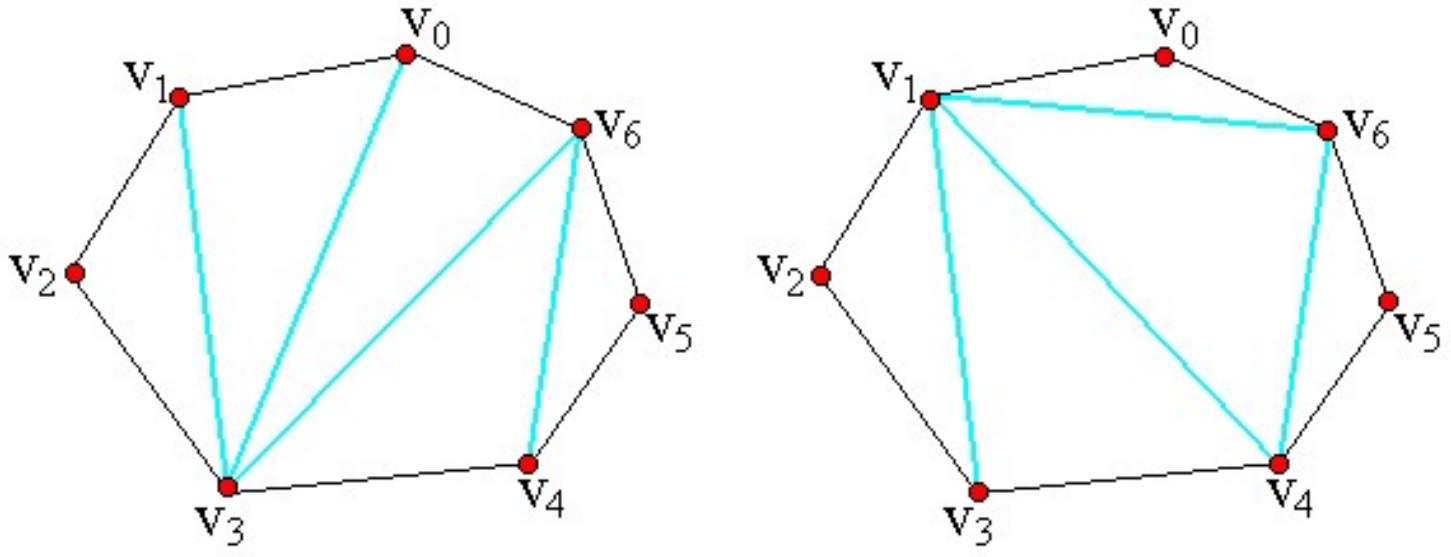
- 用多边形顶点的逆时针序列表示凸多边形，即$P = \{v_0, v_1, \ldots, v_{n-1}\}$表示具有n条边的凸多边形。
- 若v_i与v_j是多边形上不相邻的2个顶点，则线段$v_i v_j$称为多边形的一条弦。弦将多边形分割成2个多边形$\{v_i, v_{i+1}, \ldots, v_j\}$和$\{vj, v_{j+1}, \ldots vi\}$



$$P = \{v_0, v_1, \ldots, v_6\}$$

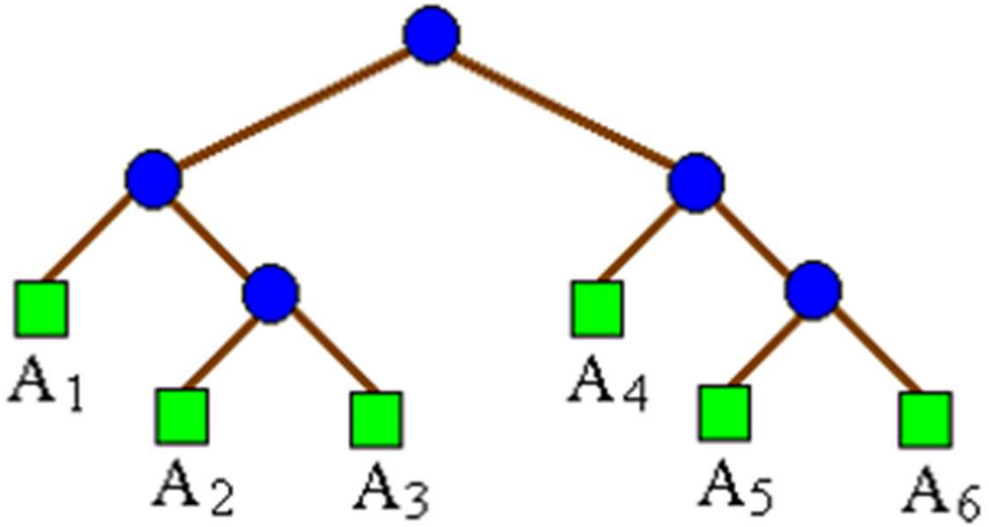
凸多边形最优三角剖分

- **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。
- 给定凸多边形P，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，使得该三角剖分中诸三角形上权之和为最小。



三角剖分的结构及其相关问题

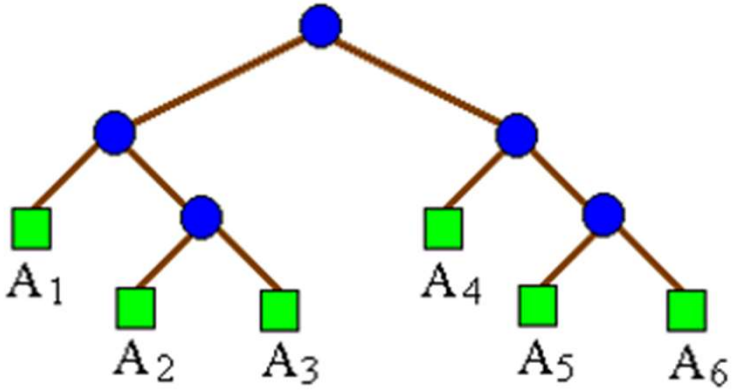
- 一个表达式的完全加括号方式相应于一棵完全二叉树，称为表达式的语法树。例如，完全加括号的矩阵连乘积 $((A_1(A_2A_3))(A_4(A_5A_6)))$所相应的语法树如图 (a)所示。



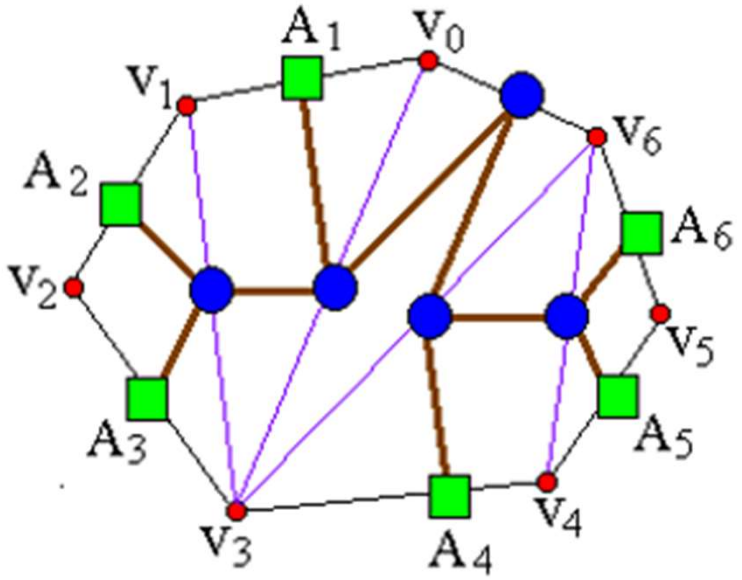
(a)

三角剖分的结构及其相关问题

- 凸多边形$\{v_0,v_1,\ldots v_{n-1}\}$的三角剖分也可以用语法树表示。例如，图 (b)中凸多边形的三角剖分可用图 (a)所示的语法树表示。
- 矩阵连乘积中的每个矩阵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，对应于矩阵连乘积A[i+1:j]。



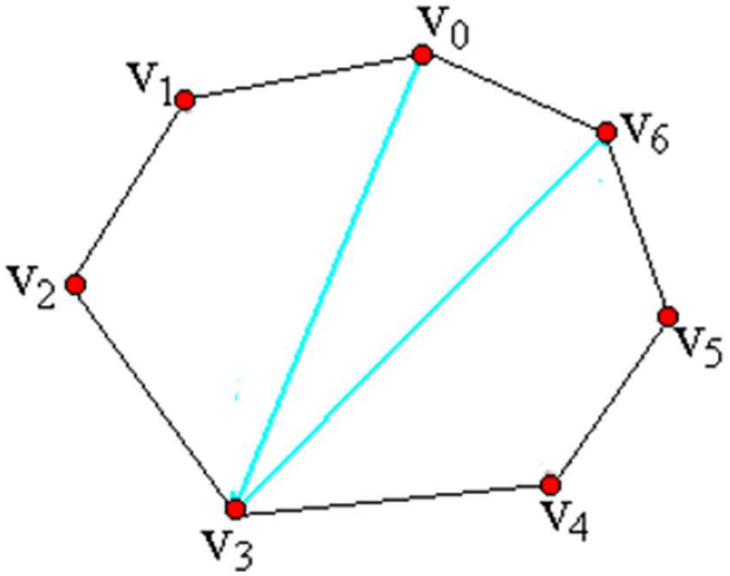
(a)



(b)

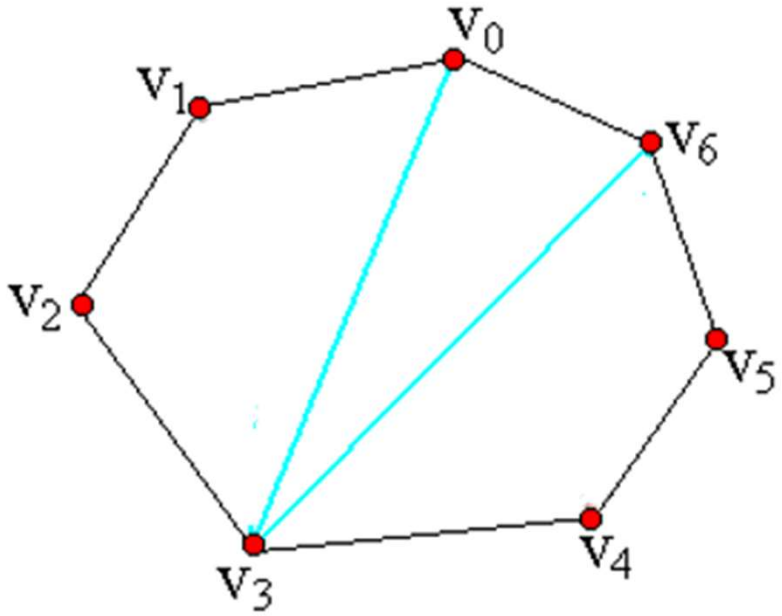
最优子结构性质

- 凸多边形的最优三角剖分问题有最优子结构性质。
- 若凸(n+1)边形P={$v_0,v_1,\ldots,v_n$}的最优三角剖分T包含三角形$v_0v_kv_n$，$1\le k\le n-1$，则T的权为3个部分权的和：三角形$v_0v_kv_n$的权，子多边形{$v_0,v_1,\ldots,v_k$}和{$v_k,v_{k+1},\ldots,v_n$}的权之和。



最优子结构性质

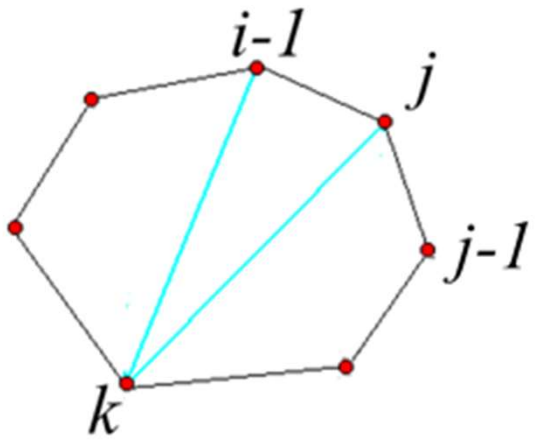
- 可以断言，由T所确定的这2个子多边形的三角剖分也是最优的。因为若有比$\{v_0,v_1,\ldots,v_k\}$或$\{v_k,v_{k+1},\ldots,v_n\}$更小权的三角剖分将导致T不是最优三角剖分的矛盾。



最优三角剖分的递归结构

•定义t[i][j]，1≤i<j≤n为凸子多边形$\{v_{i-1},v_i,\ldots,v_j\}$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]。

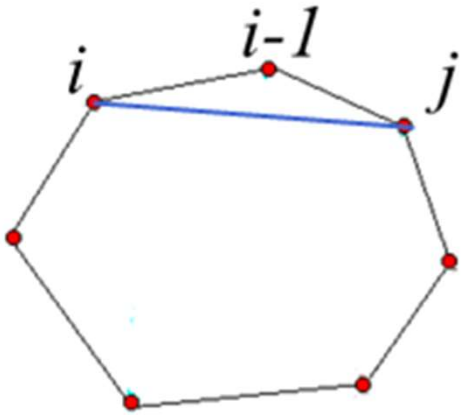
•t[i][j]的值可以利用最优子结构性质递归地计算。当j-i≥1时，凸子多边形至少有3个顶点。由最优子结构性质，t[i][j]的值应为t[i][k]的值加上t[k+1][j]的值，再加上三角形$v_{i-1}v_kv_j$的权值，其中i≤k≤j-1。



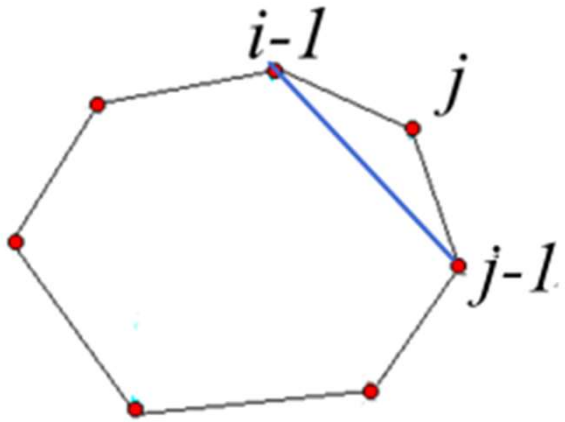
最优三角剖分的递归结构

•由于在计算时还不知道k的确切位置，而k(i≤k≤j-1)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：

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



k = i



k = j-1

用动态规划法求最优解

```
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 < j; 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;
}
```