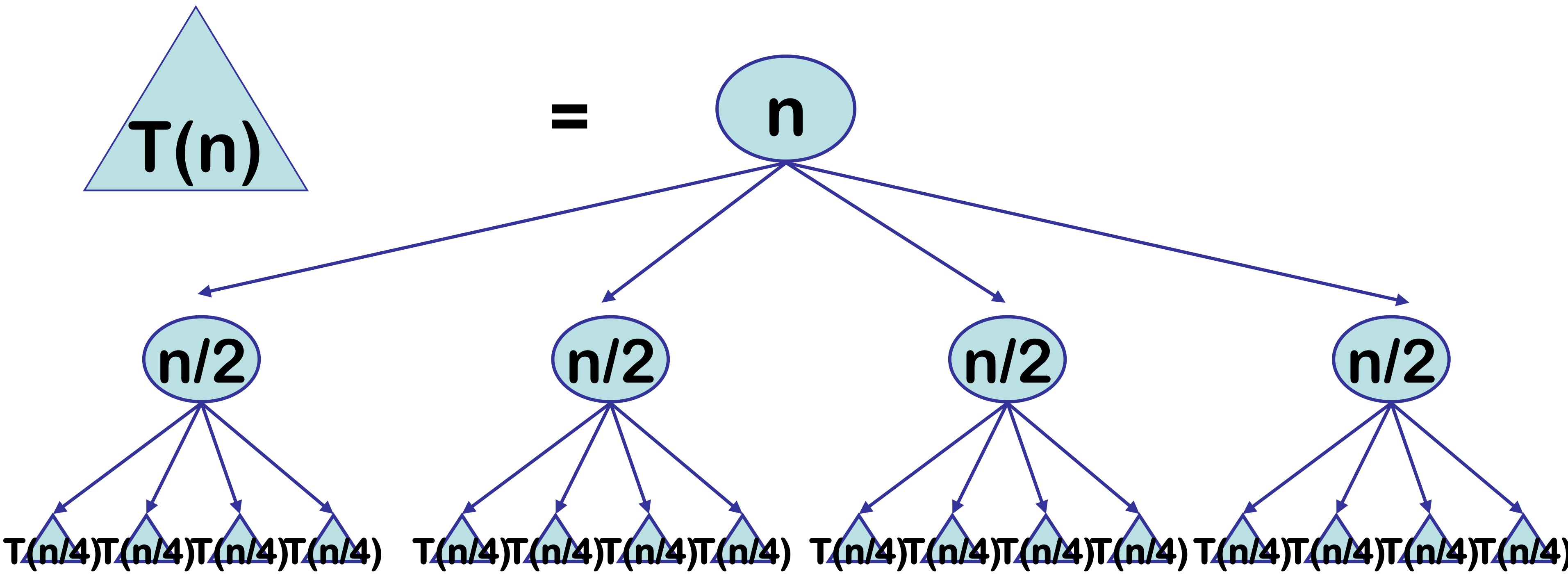
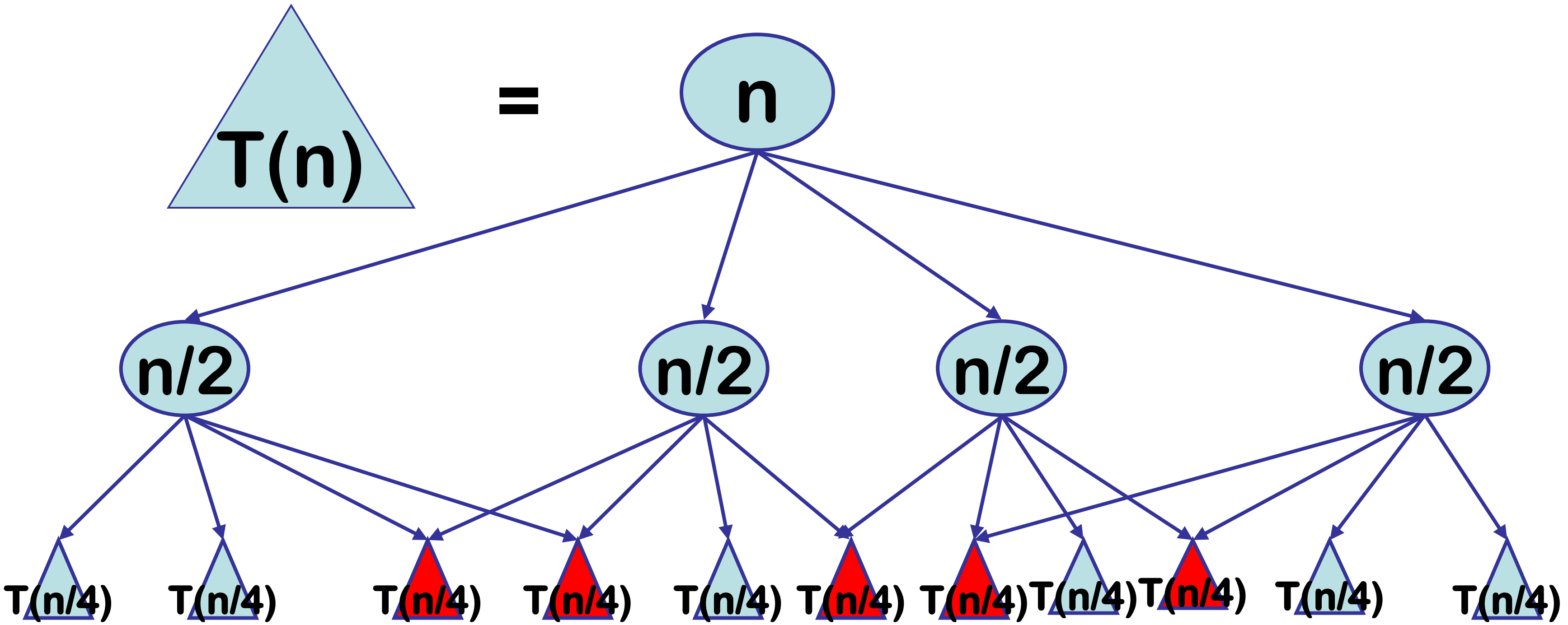
- 分治技术的问题
  - 大问题分解为若干个子问题
  - 子问题相互独立
- 问题：
  - 如果子问题不是相互独立的，分治方法将重复计算公共子问题，效率很低，甚至在多项式量级的子问题数目时也可能耗费指数时间

- 例：将待求解问题分解成若干个子问题，如果经分解得到的子问题**不是互相独立**的，效率很低



- 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。

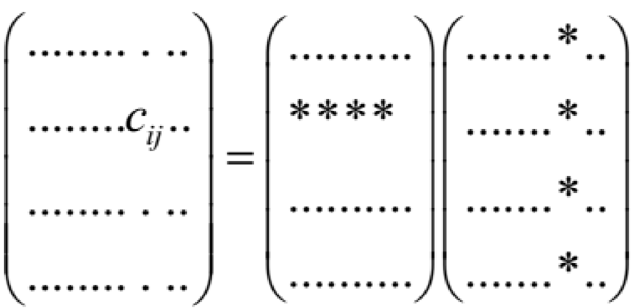


3.1 矩阵连乘问题

给定n个矩阵$\{A_1, A_2, \ldots, A_n\}$，其中$A_i$是一个$r_{i-1} \times r_i$矩阵($1 \le i \le n$)，即$A_i \times A_{i+1}$是**可乘**的，求出$n$个矩阵相乘的最优计算次序，使得**计算量最小**。

两个矩阵可乘：仅当矩阵A和B相容时(A矩阵的列数=B矩阵的行数)，A和B能相乘

- 两个矩阵乘积所需要的计算量
  - 如果 A 矩阵 p×q, B矩阵q×r，那么C矩阵p×r.
  - 求得矩阵C 的计算时间主要由第7行的标量乘所决定，相乘次数为pqr

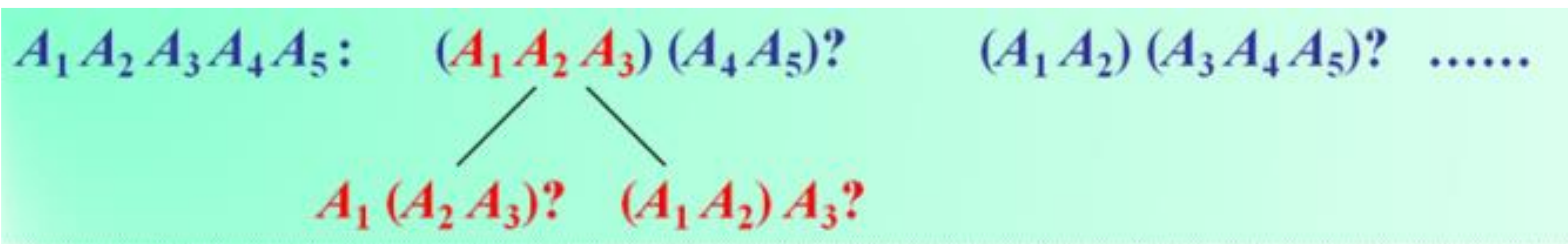


```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2      then return "error: incompatible dimensions"
3      else for i ← 1 to rows[A]
4               for j ← 1 to columns[B]
5                   C[i, j] ← 0
6                   for k ← 1 to columns[A]
7                       C[i, j]←C[i, j]+A[i, k]·B[k, j]
8      return C
```

- 分析:乘法次数由什么决定?
  - 矩阵乘的次序决定乘法次数
    - 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。
    - 对所有加括号的方式，矩阵连乘的积相同。

- For $A_{p \times q}$, $B_{q \times r}$, C=AB is $p \times r$. 乘法次数是 pqr .
- 考虑三个矩阵连乘 $\langle A_1, A_2, A_3 \rangle$.
- 假设 A_1: 10 × 100; A_2: 100 × 5; A_3: 5 × 50
  -
  -
  -

- **矩阵连乘问题:**
  - 给定一个矩阵链<$A_1, A_2, \ldots, A_n$>，其中矩阵A_i的维数为$p_{i-1} \times p_i$，寻找一种矩阵连乘完全加括号方式，使得矩阵连乘积的乘法次数最少



- 注意:该问题的目的是仅仅确定最少乘法个数的矩相乘顺序，而不是实际进行矩阵相乘

计算括号化方案的数量

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。

分析:对于n个矩阵的连乘积，设其不同的计算次序为P(n)。

- n=1 ,只有一个矩阵,只有一种方案

■n>1,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：$(A_1...A_k)(A_{k+1}...A_n)$可以得到关于P(n)的递推式如下：

k为断开位置

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

由此不难算出P(n)=C(n-1)，其中C表示卡特兰(Catalan)数

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

结果:P(n)随着n的增长呈指数增长

有没有更好的方法？

动态规划法

- 动态规划特点
  - 把原始问题划分成一系列子问题
  - 求解每个子问题仅一次，并将其结果保存**在一个表中**，以后用到时直接存取
  - 不重复计算，节省计算时间
  - 自底向上地计算最优值

- 适用范围
  - 一类**优化问题**：可分为多个相关子问题，子问题的解被重复使用

- 动态规划法基本步骤
  - 分析最优解的结构：找出最优解的性质，并刻划其结构特征。
  - 建立递归关系：递归地定义最优值。
  - 计算最优值：以自底向上的方式计算出最优值。
  - 构造最优解：根据计算最优值时得到的信息，构造最优解。

矩阵连乘问题使用的数据结构

- 输入：
  - p数组——存储矩阵行列下标
- 处理：
  - m数组——记录所需要的最小连乘次数
  - s数组——记录最优括号化方案的分割点位置

步骤1：分析最优解的结构

将矩阵连乘积 $A_iA_{i+1}\ldots A_j$ 简记为**A[*i*:*j*]**，*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 $(A_iA_{i+1}\ldots A_k)(A_{k+1}A_{k+2}\ldots A_j)$

- 计算量：
  - *A*[*i*:*k*]的计算量+*A*[*k*+1:*j*]的计算量+*A*[*i*:*k*]和*A*[*k*+1:*j*]相乘的计算量

- 关键特征：计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k] 的次序也是最优的。
  - 反证法证明:若存在一个计算A[i:k]的次序所需计算量更少，则进行替代，得到的计算A[i:j]比最优次序所需计算量少，存在矛盾。
- 同理，计算A[i:j]的最优次序所包含的计算矩阵子链 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**。
- 因此任何最优解都是由子问题实例的最优解构成，可以将问题A[i:j]划分为两个子问题A[i:k] 和A[k+1:j]。

可用动态规划算法求解的显著特征

步骤二:建立递归关系

- 最优子结构性质提示我们使用最优化原则产生的算法是递归算法。
- 但是，简单地使用递归算法可能会增加时间与空间开销。
- 使用数组记录已解决的子问题的答案。

定义:

- 数组P：存放矩阵A_i的行、列数，矩阵A_i的行列值p_{i-1},p_i，初始值为 $p_0,p_1\ldots\ldots p_{n-1},p_n$
- **m[i][j]：** 计算A[i:j] 所需要的最少乘次数，原问题为求m[1][n]；
  - 当**i=j**时,A[i:j]=A_i，只有一个矩阵A_i，没有矩阵元素相乘，因此，

    m[i,i]=0，i=1,2,…,n

- 当i<j时
  - 设矩阵连乘$A_{i..j}$最优全括号的分割点位于A_k与A_{k+1}之间，则…

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

这里A_i的维数为 $p_{i-1}\times p_i$

k为最佳断开位置

- 由于递归方程假设已知矩阵连乘最优全括号问题的分隔点*k*，实际上*k*是未知的。
- *k*的位置只有*j-i*种可能， k∈{*i*, *i*+1, ..., *j*-1}
- 递归地定义**m[i][j]**为：

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

k **的位置只有** $j-i$ **种可能**

步骤三:计算最优值

- 定义:
  - 计算A[1:n]的最小代价:

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

  - s[i][j]保存$A_{i..j}$ 最优括号化方案的分割点位置 *k*

递归方法

```
int RecurMatrixChain(int i, int j){
  if (i==j) return 0;                    //初始化i为最佳断开位置
  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];                //递归计算最优解
    if (t<u) {        u=t;
                      s[i][j]=k;        }
  }
  return u;}
```

如果用T(n)表示该算法的计算A[1:n]的时间，则有如下递归关系式:

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

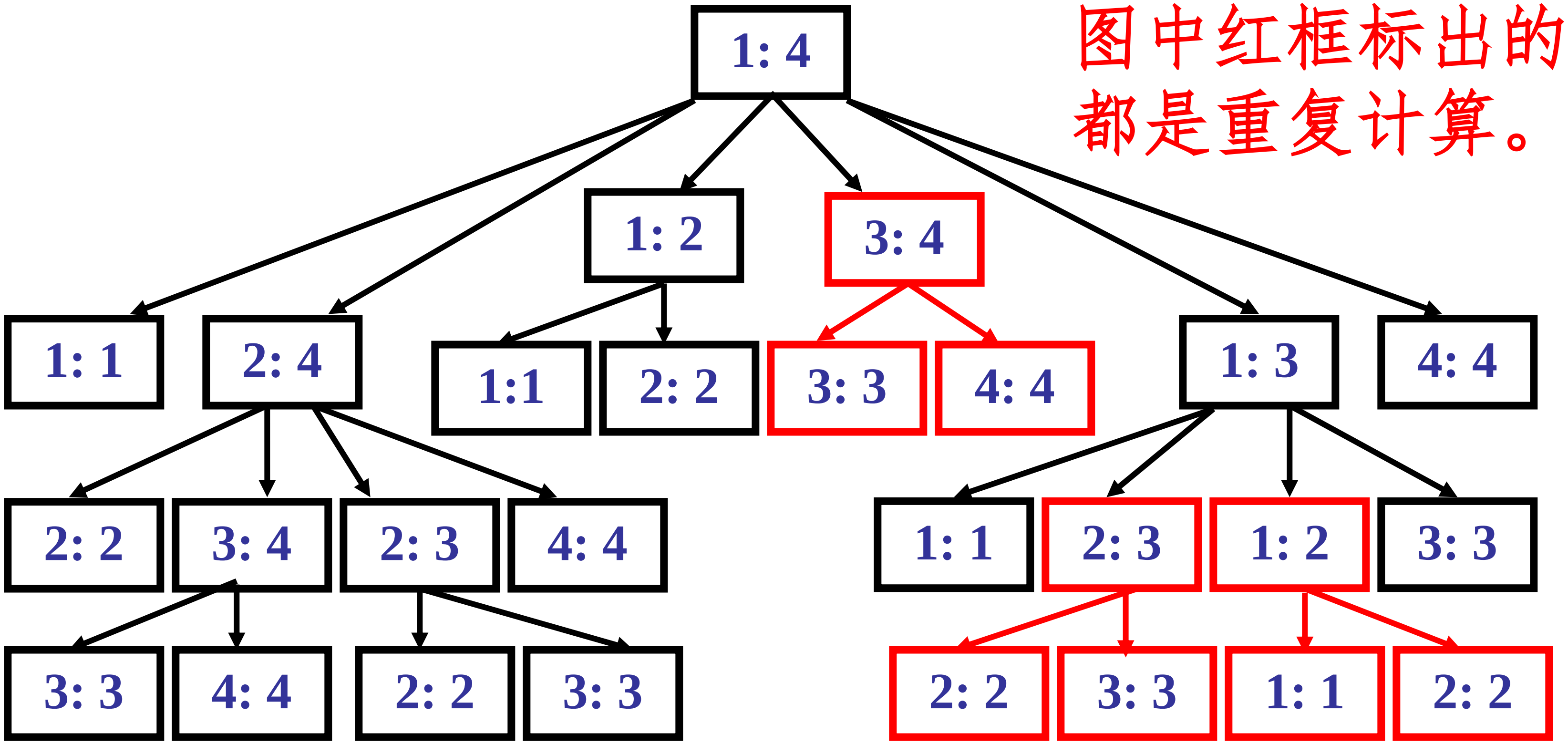
$$T(n) \ge 1+(n-1)+\sum_{k=1}^{n-1}T(k)+\sum_{k=1}^{n-1}T(n-k) = n+2\sum_{k=1}^{n-1}T(k)$$

可用数学归纳法直接证明：$T(n) \ge 2^{n-1} = \Omega(2^n)$

$$m[i][j] = \begin{cases} 0 \\ \min\limits_{i \le k < j}\{m[i][k]+m[k+1][\ldots \end{cases}$$

显然不是期望结果

- 直接递归中有大量重复计算，如A[1: 4]计算中：



图中红框标出的都是重复计算。

- 注意到在此问题中，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

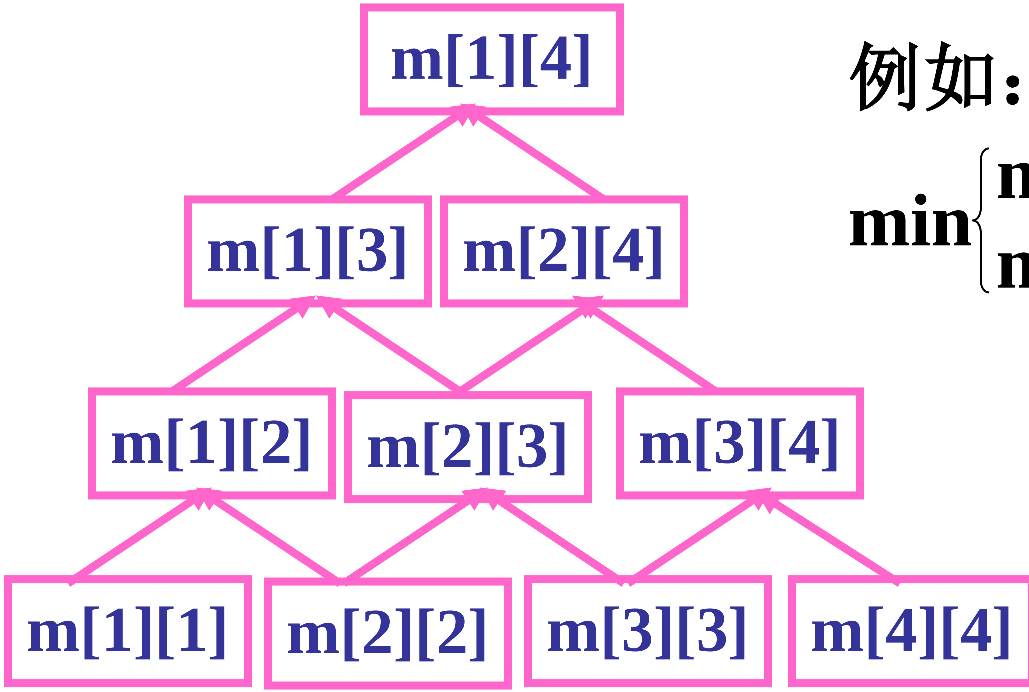
$$\binom{n}{2}+n=\theta(n^2)$$

- 由此可见，在递归计算时，许多子问题**被重复计算多次**。

可用动态规划算法求解的显著特征

- 用动态规划算法解此问题，消除重复计算，可依据其递归式以自底向上的方式进行计算。可以在计算过程中保存已解决的子问题的答案。
- 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。

- 例如对于$A_1A_2A_3A_4$，依据递归式以自底向上的方式计算出各个子问题，其过程如下：



例如： m[1][3] =

$$\min\begin{cases} m[1][1]+m[2][3]+p_0p_1p_3 \\ m[1][2]+m[3][3]+p_0p_2p_3 \end{cases}$$

$$m[i][i+1] = p_{i-1}\,p_i\,p_{i+1}$$

$$m[i][i] = 0$$

- 当i=j时，$A[i:j]=A_i$， 因此， $m[i,i]=0$， $i=1,2,\ldots,n$
- 当i<j时， $m[i,j]=m[i,k]+m[k+1,j]+p_0p_kp_5$

m[1,1]	m[1,2]	m[1,3]	m[1,4]	**m[1,5]**
	m[2,2]	m[2,3]	m[2,4]	m[2,5]
		m[3,3]	m[3,4]	m[3,5]
			m[4,4]	m[4,5]
				m[5,5]

```
void matrixChain(int  *p, int  n, int m[ ][ ] , int  s[][] ){
 for (int i = 1; i <= n; i++) m[i][i] = 0;
 for (int r = 2; r <= n; r++)          /*循环
  for (int i = 1; i <= n-r+1; i++) {
   int j=i+r-1;
   m[i][j] = m[i][i]+m[i+1][j]+ p[i-1]*p[i]*p[j];
   s[i][j] = i;
   for (int k=i+1; k < j; k++) {      /*k为断点位置*/
    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;}
} }}
```

计算m[i][j]不含对角线的上三角矩阵,利用变量i、j设置按照斜线顺序计算m[i][j]

初始化时i为断开位置

k从i+1到j-1循环查找,不断替换，找到最优值和最佳断开位置

例：

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

m[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

s[i][j]

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

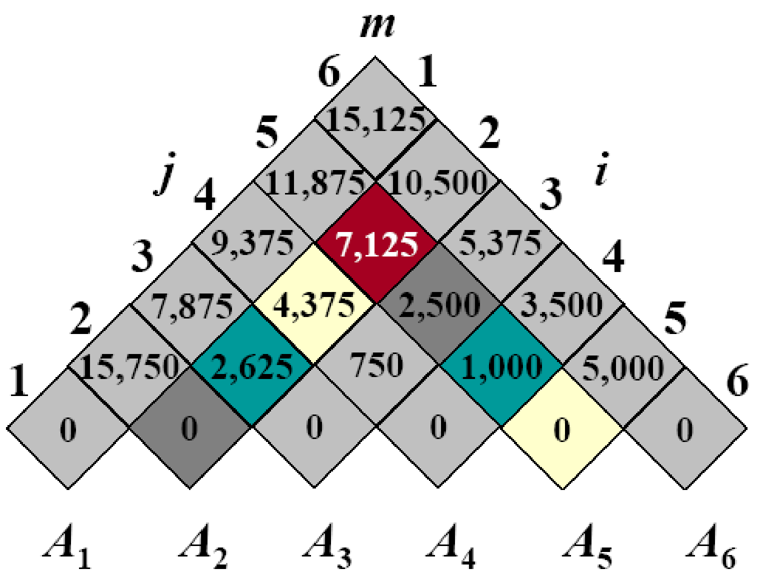
$$m[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i][k]+m[k+1][j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

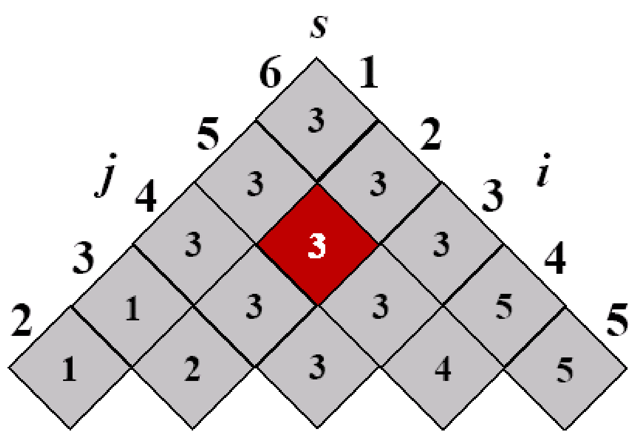
例：在计算m[2][5]时，依公式：

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

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

且k=3,因此s[2][5]=3





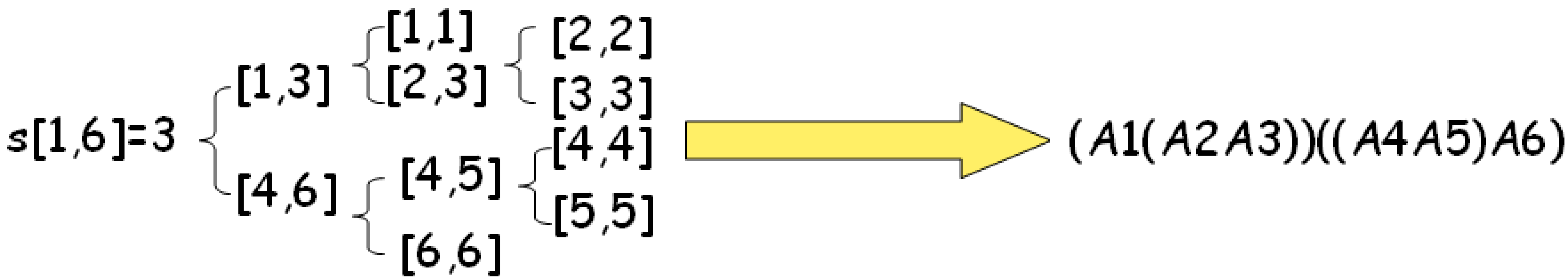
算法复杂度分析：

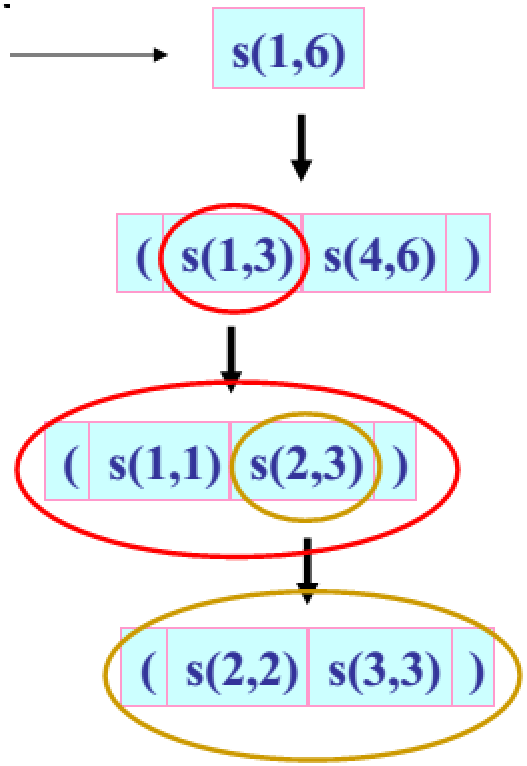
- 算法matrixChain的主要计算量取决于算法中对r，i和k的3重循环。
- 循环体内的计算量为O(1)，而3重循环的总次数为$O(n^3)$。
    - 算法的计算时间上界为$O(n^3)$。
    - 算法占用的空间上界为$O(n^2)$。

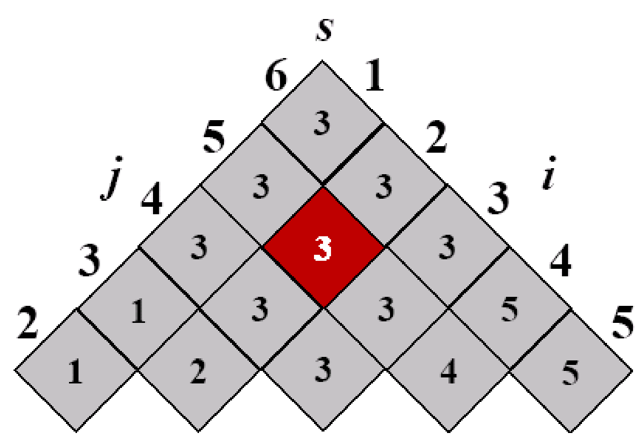
步骤四:用动态规划法求最优解

- 注意:算法MatrixChain只是明确给出了矩阵最优连乘次序所用的数乘次数m[1][n]，并未明确给出最优连乘次序，即完全加括号方法。

- s[i][j]=k 说明了计算连乘积A[i:j]的最佳方式应该在矩阵A_k和A_{k+1}之间断开，即最优加括号方式为(A[i:k])(A[k+1:j])。


s[1,6]=3
[1,3]
[1,1]
[2,3]
[2,2]
[3,3]
[4,6]
[4,5]
[4,4]
[5,5]
[6,6]
(A1(A2A3))((A4A5)A6)


s(1,6)
(s(1,3) s(4,6))
(s(1,1) s(2,3))
(s(2,2) s(3,3))


s
j
i
6 5 4 3 2
1 2 3 4 5
3
3 3
3 3 3
1 3 3 5
1 2 3 4 5

算法Traceback按算法MatrixChain计算出的断点信息s指示的加括号方式输出计算A[i:j]的最优次序。

```
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;
  }
```

3.2 动态规划算法的基本要素

1. 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**。
  - 缩小子问题集合，降低实现复杂性
  - 最优子结构使用自底而上地完成求解过程

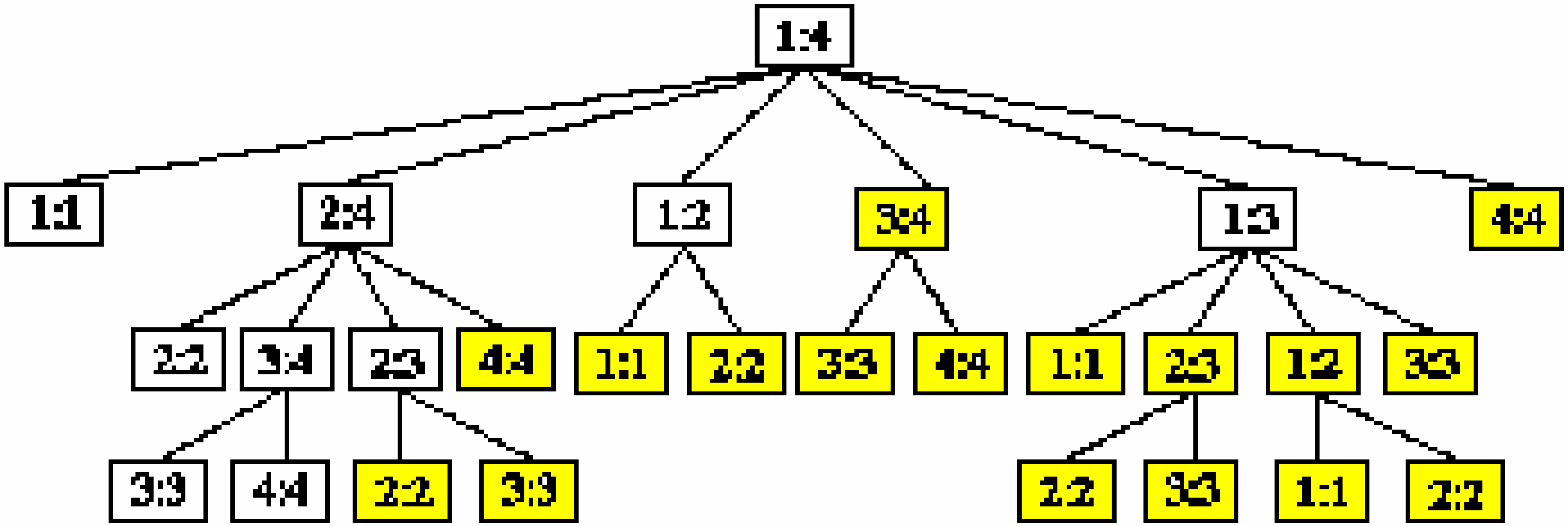
- 在分析问题的最优子结构性质时，所用的方法具有普遍性：
    - 首先假设由问题的最优解导出的子问题的解不是最优的；
    - 然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。

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）

2. 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。



用递归式求解，算法的时间$T(n)=\Omega(2^n)$

3. 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，在下次需要解此问题时，只要简单地查看该问题的解答，而不必重新计算.
  - 备忘录方法的控制结构与直接递归方法的控制结构相同
  - 递归方式：**自顶向下**

计算时间从Ω (2^n)降至$O(n^3)$

```
Int MemoizedMatrixChain( int n, int **m, int **s)
 { for(int i=1; i<=n, i++)
    for (int j=i; j<=n,  j++)
       m[i[j]=0;                    //初始化
   return lookupChain(1,n);
 }
```

备忘录方法的控制结构与直接递归方法的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。

```
int lookupChain(int i, int j)
 {
    if (m[i][j] > 0) return m[i][j];    //已经计算过,返回结果
    if (i == j) return 0;
    int u = lookupChain(i,i) + 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;
 }
```

表示该子问题已经求解

递归求解断开位置为i时的计算量

循环每个可取断开位置求计算量

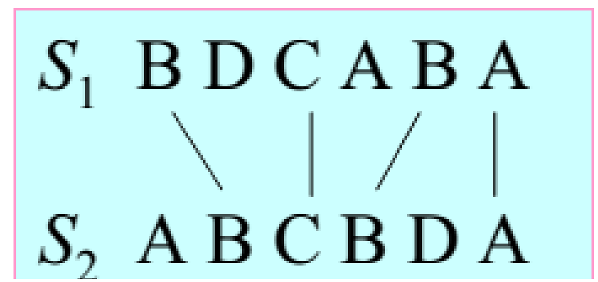
	动态规划法	备忘录法
求解次数(适用场合)	所有子问题都至少要解一次	部分子问题可不必求解
递归方式	自底向上	自顶向下

- 矩阵连乘积的最优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。
- 这两个算法都用了子问题的重叠性质，共有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，简单的取用已得到的答案，提高了算法的效率。

3.3最长公共子序列

- 应用
  - 论文相似度检查
  - 比较两个不同生物体的DNA
    - S1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
    - S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA
    - 研究目标：确定两个DNA序列的相似程度?

- 如何定义S_1和S_2的相似度?
  - 寻找第3个串S_3，S_3中的所有bases都包含在S_1和S_2中，这些bases在S_1和S_2中不一定连续排列，但必须是按顺序排列的。S_3越长，则S_1和S_2的相似度就越大.



- **子序列:** 若给定序列$X=\{x_1,x_2,\ldots,x_m\}$，则另一序列$Z=\{z_1,z_2,\ldots,z_k\}$，是X的子序列是指存在一个严格递增下标序列$\{i_1,i_2,\ldots,i_k\}$使得对于所有$j=1,2,\ldots,k$有：$z_j=x_{ij}$。
- 例如:
  - Z={B，C， D， B}
  - 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}。

- **公共子序列:给定2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:
  - **X = A, B, C, B, D, A, B**
  - **Z_1= B, C, A**
  - **Y = B, D, C, A, B, A**

- **问题描述：** **最长公共子序列（Longest Common Subsequence，LCS)问题**
  - **给定两个序列$X=<x_1,x_2,\ldots,x_m>$和$Y=<y_1, y_2, \ldots, y_n>$,如何寻找X 和Y 的长度最大的公共子序列。**
- **输入：** $X=\{x_1,x_2,\ldots,x_m\}$，$Y=\{y_1, y_2, \ldots, y_n\}$
- **输出：** **Z=X和Y最长公共子序列**

- **LCS 问题可以用动态规划法有效解决**

- 问题求解的步骤
  - 步骤1:最长公共子序列的结构
  - 步骤2:子问题的递归结构
  - 步骤3:计算最优值
  - 步骤4:构造最优解

1、最长公共子序列的结构

已知$X = <x_1, x_2, ..., x_m>$ 和 $Y = <y_1, y_2, ..., y_n>$

- **原始方法**
  - **列举出X 的所有子序列，逐项核查这些子序列是否为Y 的子序列**
  - **对于X 的一个索引的子集{1, 2, ..., m}，有2^m个X 的子序列。需要指数运算时间，当序列较大时实际不可行**

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

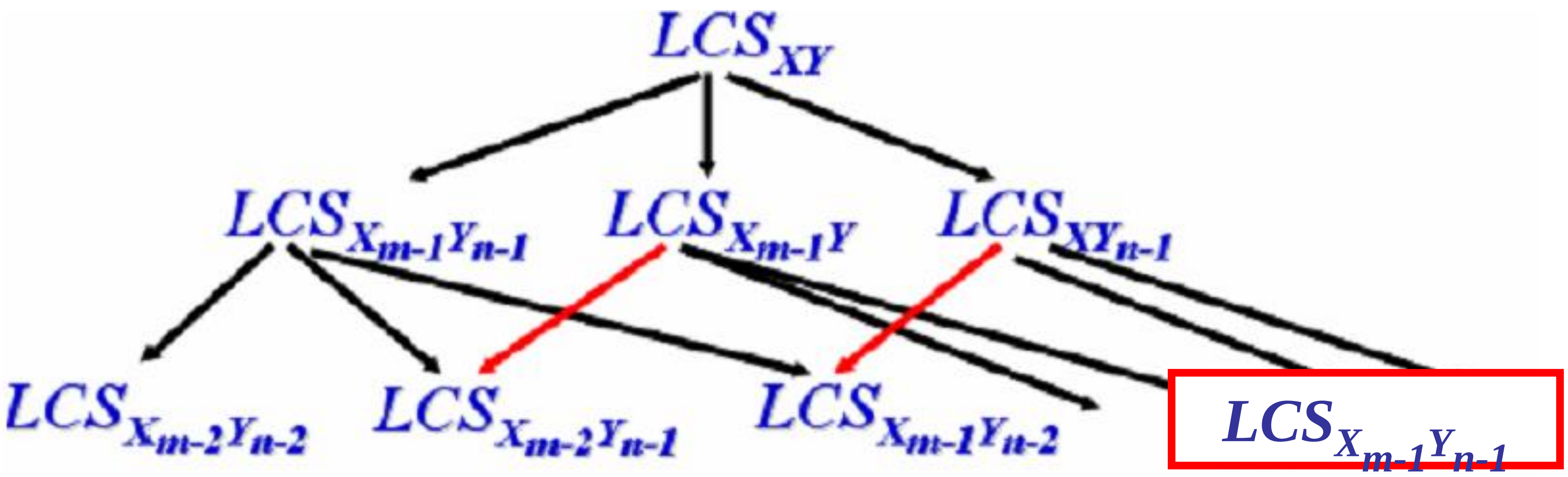
(3)若$x_m \neq y_n$且$z_k \neq y_n$，则Z 是X和Y_{n-1}的最长公共子序列。

$$X=<x_1, \ldots, x_i, \ldots, x_m>$$
$$Z=<z_1, \ldots, z_k>$$
$$Y=<y_1, \ldots, y_j, \ldots, y_n>$$

➢**根据分析要找出序列**$X=\{x_1,x_2,\ldots,x_m\}$**和**$Y=\{y_1, y_2, \ldots,y_n\}$**的最长公共子序列，可以按以下方式递归地进行：**

- ➢**当**$x_m=y_n$**，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**就可以得到**X**和**Y**的最长公共子序列。**
- ➢**当**$x_m \neq y_n$**，找出**X_{m-1}**和**Y_n**以及**X_m**和**Y_{n-1}**最长公共子序列，得到**X**和**Y**的最长公共子序列。**

✓重叠子问题



✓在所考虑的子问题空间中，总共有θ(mn)个不同的子问题。

2. 子问题的递归结构

- 定义：**c[i, j]**记录序列X_i和Y_j的最长公共子序列的长度(c[i,j]=0, i·j=0).

$X_i = <x_1, x_2, \ldots, x_i>$
$Z_k = <z_1, z_2, \ldots, z_k>$
$Y_j = <y_1, y_2, \ldots, y_j>$

- $x_i = y_j$：
  $X_{i-1} = <x_1, x_2, \ldots, x_{i-1}>$
  $Z_{k-1} = <z_1, z_2, \ldots, z_{k-1}>$
  $Y_{j-1} = <y_1, y_2, \ldots, y_{j-1}>$
  $c[i,j] = c[i-1,j-1]+1$

- $x_i \neq y_j$，$z_k \neq x_i$：
  $X_{i-1} = <x_1, x_2, \ldots, x_{i-1}>$
  $Z_k = <z_1, z_2, \ldots, z_k>$
  $Y_j = <y_1, y_2, \ldots, y_j>$

- $x_i \neq y_j$，$z_k \neq y_j$：
  $X_i = <x_1, x_2, \ldots, x_i>$
  $Z_k = <z_1, z_2, \ldots, z_k>$
  $Y_{j-1} = <y_1, y_2, \ldots, y_{j-1}>$

  $c[i,j] = \max(c[i-1,j], c[i,j-1])$

- **建立子问题最优值的递归关系：**
  - **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。**
  - **其他情况下，由最优子结构性质可建立递归关系如下：**

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

3. 计算最优值

✓**定义：b[i][j]记录c[i][j]是由哪个子问题的解得到的。**

- ✓**b[i][j]=1：表示$x_i=y_j$，　　c[i][j]= c[i-1][j-1]+1;**
- ✓**b[i][j]=2：表示$x_i\neq y_j$且$z_k\neq x_i$，　c[i][j]=c[i-1][j];**
- ✓**b[i][j]=3：表示$x_i\neq y_j$且$z_k\neq y_j$，　c[i][j]=c[i][j-1];**

```
void LCSLength(int m,int n, char *x, char *y,int c[][], int b[][]) {
 int i, j;
 for(i=1;i<m;i++) {  c[i][0]=0;}
 for(i=1;i<n;i++) {  c[0][j]=0;}                       //初始化
 for (int i = 1; i <= m; i++)
  for (int j = 1; j <= n; j++)                         //m,n是X和Y的长度
    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;   }
}
```

算法复杂度分析：
算法的计算时间上界为 **O(mn)**。
算法所占用的空间显然为 **O(mn)**。

例如，若X= { A, B, C, B, D, A, B}和Y= {B, D, C, A, B, A},

	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	↑4

4. 构造最长公共子序列

- **由算法LCSLength计算得到的数组b可用于快速的构造X、Y的最长公共子序列。**
- **首先从b[m][n]开始，依其值在数组b中寻找：**
  - **当在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。**
  - **当在b[i][j]=2时，表示X_i和Y_j的最长公共子序列是由X_{i-1}和Y_j的最长公共子序列相同。**
  - **当在b[i][j]=3时，表示X_i和Y_j的最长公共子序列是由X_i和Y_{j-1}的最长公共子序列相同。**

```
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);
      cout<< x[i];
      }
    else if (b[i][j]== 2) lcs(i-1,j,x,b);
      else lcs(i,j-1,x,b);
  }
```

- **如果只需要计算最长公共子序列的长度，空间复杂度分析如下：**
  - **b[m][n]数组不需要；**
  - **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*))。**

- 从运行结果中可以看出，算法LCS回溯算法仅仅打印了其中一条最大公共子序列，如果存在多条公共子序列的情况下怎么解决?
- 对b[i][j]二维数组的取值添加一种可能，等于4，代表多支情况，那么回溯的时候可以根据这个信息打印更多可能的选择。

5. 算法的改进

- **在算法lcsLength和LCS中，可进一步将数组b省去。**
  - **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。**

课堂练习：

1.设计一个$O(n^2)$算法，找出n个数组成的序列的最长单调递增子序列。

2.求<1,0,0,1,0,1,0,1>和<0,1,0, 1, 1,0, 1, 1,0>的一个最长公共子序列。

3.对矩阵规模序列<5,10,3,12, 5, 50, 6>,求矩阵链最括号化方案,写出m[i][j]距阵的值。

1.递归式：b[0:n-1]记录a[i],0≤i<n为结尾元素的最长递增子序列的长度，b[0]=1, b[i]=max{b[k]}+1 , 0≤k<i a[k] ≤a[i]

```
Public static int LISdyna(){
  int i,j,k;
  for(i=1,b[0]=1; i<n; i++){
     for(j=0,k=0; j<i; j++)  if (a[j] <a[i]&&k<b[j])  k=b[j];
     b[i]=k+1;}
   return maxL(n);
}
Static int maxL(n){
  int temp=0;
  for(int i=0; i<n; i++)  if (b[i]>temp) temp=b[i];
  return temp;
}
```

2、 *The LCS is* <1, 0, 0, 1, 1, 0>

3、最终答案：$((A_1A_2)((A_3A_4)(A_5A_6)))$

i\j	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0