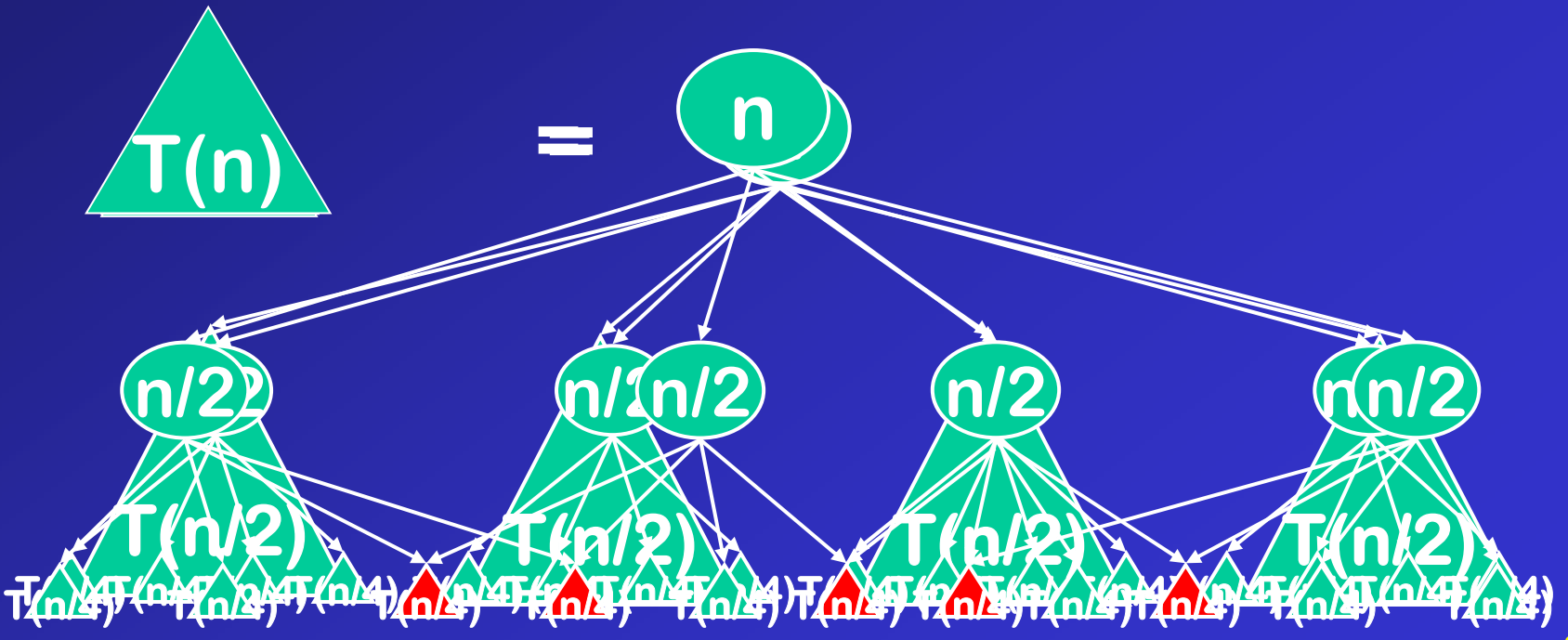
第三章 动态规划

- 本章主要知识点：
  - 3.1 矩阵连乘问题
  - 3.2 动态规划算法的基本要素
  - 3.3 最长公共子序列问题
  - 3.4 最大子段和
  - 3.5 凸多边形的最优三角剖分
  - 3.6 多边形游戏
  - 3.7 图像压缩
  - 3.8 电路布线
  - 3.9 流水作业调度
  - 3.10 0-1背包问题

引言

- 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。
- 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。
- 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。

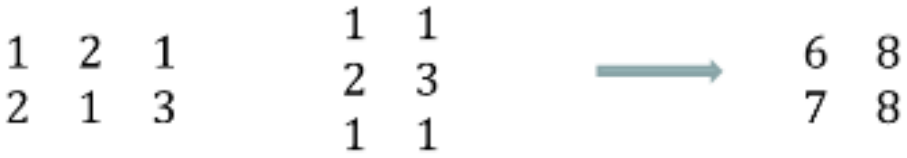


动态规划基本步骤

- 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。

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与A_{i+1}是可乘的，i=1，2...，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。



两个矩阵相乘所需做的数乘次数为 l*n*m

矩阵乘法满足结合律，故矩阵连乘有可以有许多不同的计算顺序。

计算顺序由加括号的方式确定。

加括号的方式决定了矩阵连乘的计算量

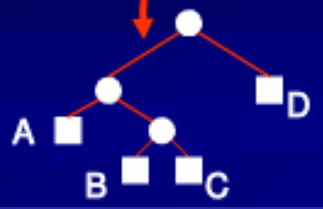
3.1 矩阵连乘问题

- 例如

4个矩阵连乘积ABCD，设它们的维数分别为A:50×10，B:10×40, C:40×30, D:30×5。

矩阵连乘的完全加括号形式以及相应的计算量

完全加括号形式	连乘积所需计算量
(A(B(CD)))	40×30×5+10×40×5+50×10×5=10500
(A((BC)D))	10×40×30+10×30×5+50×10×5=16000
((AB)(CD))	50×10×40+40×30×5+50×40×5=36000
(((AB)C)D)	50×10×40+50×40×30+50×30×5=87500
((A(BC))D)	10×40×30+50×10×30+50×30×5=34500



每种计算次序都可以用一个二叉树来表示

3.1 矩阵连乘问题

- 已确定计算次序的矩阵连乘积称为已完全加括号
- 由于每种加括号方式都可以分解为两个子连乘积的加括号问题，因此完全加括号的矩阵连乘积可递归地定义为：
  - 矩阵连乘积A是完全加括号的，则A可表示为两个已完全加括号的矩阵连乘积B和C的乘积，即A=(BC)。
  - 其中单个矩阵视为已完全加括号的；

$$A=A_1 \times A_2 \times A_3 \times \ldots \times A_n$$

$$A=\underbrace{(A_1 \times A_2 \times \ldots \times A_k)}_{B} \times \underbrace{(A_{k+1} \times A_{k+2} \times \ldots \times A_n)}_{C}$$

3.1 矩阵连乘问题---穷举搜索法

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

    $$\Rightarrow \quad P(n)=C(n-1), \text{ 其中，} \quad C(n)=\frac{1}{n+1}\binom{2n}{n}=\Omega\left(4^n/n^{3/2}\right)$$
  - 也就是说，$P(n)$是随n的增长成指数增长的。

3.1 矩阵连乘问题---分治法

- 分治法
  - 将矩阵连乘积$A_iA_{i+1}…A_j$简记为**A[i:j]**。
  - 设**n**个矩阵的连乘积$A_1A_2…A_n$的计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，**1≤k<n**，则其相应的完全加括号方式为$(A_1…A_k)(A_{k+1}…A_n)$。
  - 完全加括号是一个递归定义，可递归地计算**A[1:k]**和**A[k+1:n]**，然后将计算结果相乘得到**A[1:n]**。

3.1 矩阵连乘问题---分治法

可以递归地定义m[i,j]为：

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

前部分的时间　　后部分的时间　　合并时间

k 的位置只有 $j-i$ 种可能

3.1 矩阵连乘问题---分治法

```
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;
}
```

3.1 矩阵连乘问题---分治法

- 该递归算法的的计算时间**T(n)**可递归定义如下：

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

- 当**n>1**时，

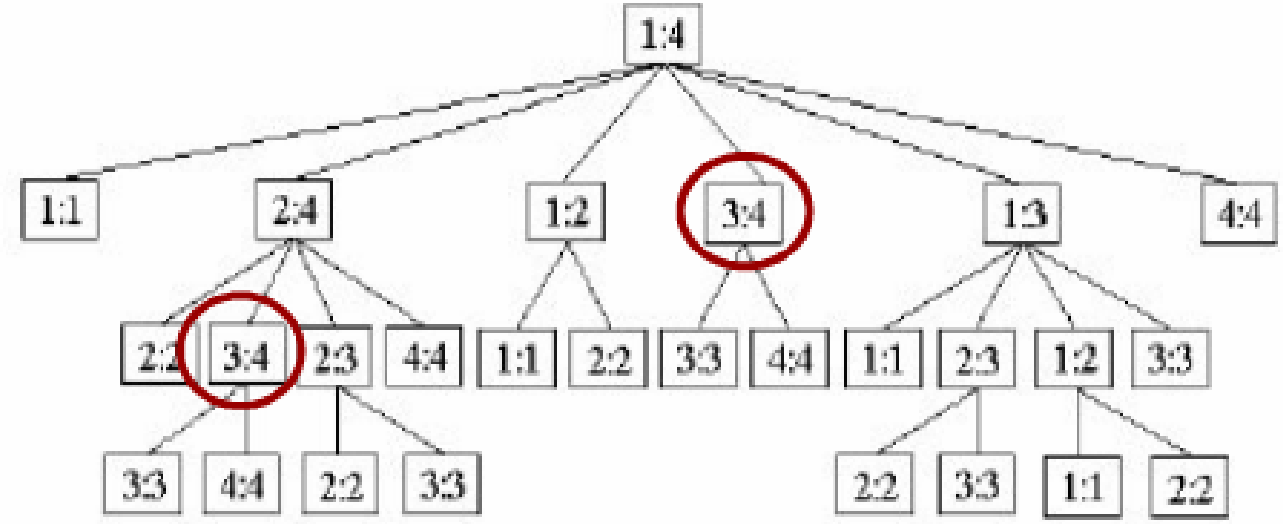
$$T(n)\geq 1+(n-1)+\sum_{k=1}^{n-1}T(k)+\sum_{k=1}^{n-1}T(n-k)=n+2\sum_{k=1}^{n-1}T(k)$$

$$T(n)\geq 2^{n-1}=\Omega(2^n)$$

- 该算法的计算时间**T(n)**有指数下界。
- 分治法是该问题的一个可行方法，但不是一个有效算法。

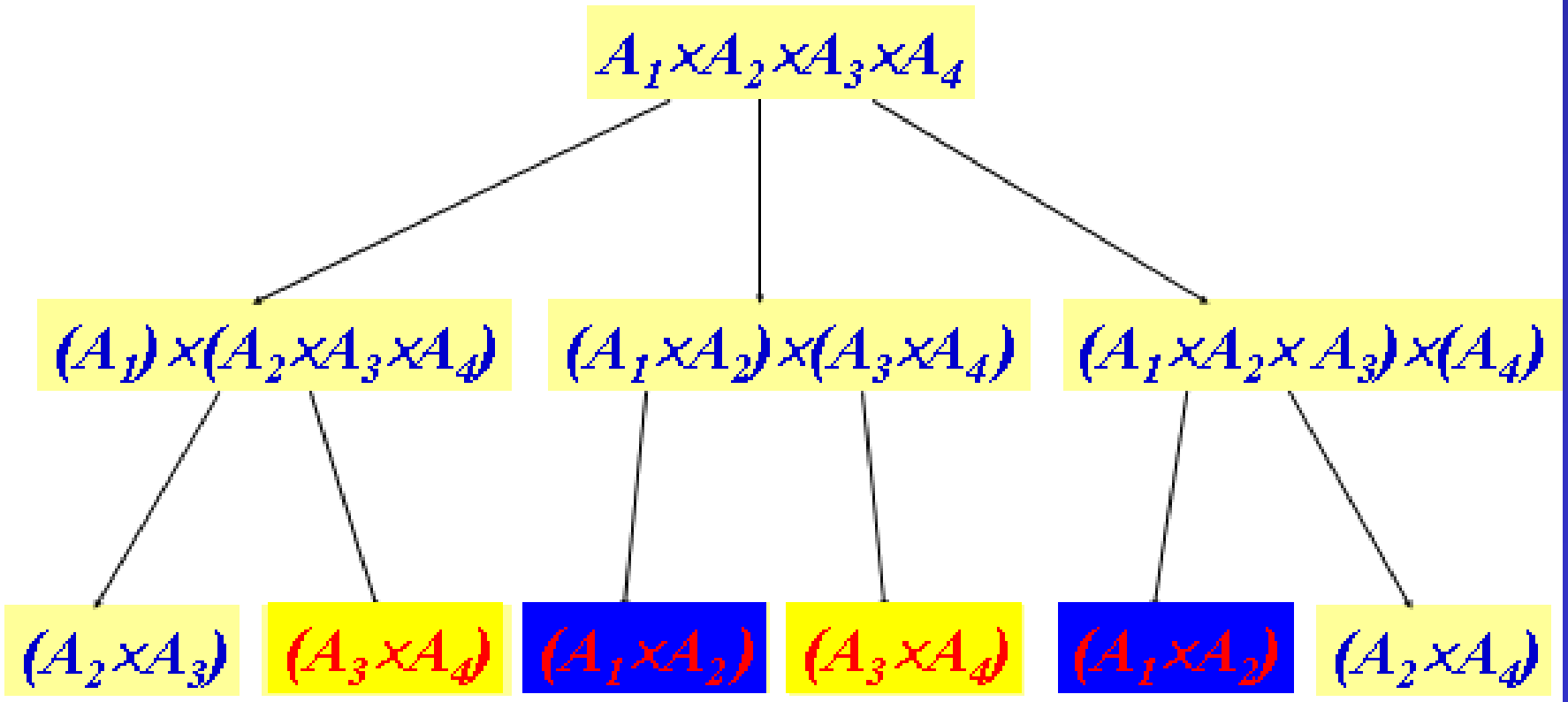
3.1 矩阵连乘问题---分治法

- 为何分治法的效率如此低下？
- **recurmatrixChain(1,4)**计算**A[1：4]**的递归树。



- 从图中可看出，许多子问题被重复计算。
- 这是分治法效率低下的根本原因。

3.1 矩阵连乘问题---动态规划法



- 该问题可用分治思想解决，并存在大量冗余，可用动态规划求解。

3.1 矩阵连乘动态规划法——1.分析最优解的结构

- 下面我们考虑用动态规划求解。
- 预处理：
  - 将矩阵连乘积$A_iA_{i+1}...A_j$简记为$A[i:j]$，这里$i \le j$。
  - 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，$i \le k < j$，则其相应完全加括号方式为（$A_iA_{i+1}... A_k$）（$A_{k+1} A_{k+2}... A_j$）。
  - 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。

3.1 矩阵连乘动态规划法—— 1.分析最优解的结构

1、划分阶段（子问题）并刻画

$(A((BC)D)) \Longrightarrow$ (A) ((BC)(D))　$((A(BC))D) \Longrightarrow$ ((A)(BC)) (D)

$(((AB)C)D) \Longrightarrow$ ((AB)(C))(D)

将原问题划分成两个子问题。如果原问题获得最优值，则子问题的应该也是最优的。

将矩阵连乘积 $A_1A_2...A_n$ 简记为A(1:n)
设最优计算次序在矩阵
A_k和A_{k+1}之间将矩阵链断开，$1 \le k < n$，则其相应完全加括号方式为

$$(A_1...A_k)(A_{k+1}A_{k+2}...A_n)$$

3.1 矩阵连乘动态规划法—— 2.建立递归关系

- 设计算$A[i:j]$，$1\le i\le j\le n$，所需要的最少数乘次数$m[i,j]$，则原问题的最优值为$m[1,n]$。
- 当$i=j$时，$A[i:j]=A_i$，因此，$m[i,i]=0$，$i=1,2,\ldots,n$。
- 当$i<j$时，$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$，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。

3.1 矩阵连乘动态规划法—— 3.计算最优值

- 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。
- 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。

3.1 矩阵连乘动态规划法——算法描述

- 算法描述：

```
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++)
        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;}
            }
        }
```

3.1 矩阵连乘动态规划法——实例分析

例　四个乘 50*10 10*40 40*30 30*5的矩阵连乘。

P[0]=50 P[1]=10 P[2]=40 P[3]=30 P[4]=5

m

0			
	0		
		0	
			0

```
for (int i = 1; i <= n; i++)  m[i][i] = 0;
```

3.1 矩阵连乘动态规划法——实例分析

例　四个乘 50*10 10*40 40*30 30*5的矩阵连乘。

P[0]=50 P[1]=10 P[2]=40 P[3]=30 P[4]=5

m

0			
	0		
		0	6000
			0

•i=n-1=3 ,j=4, k=3

m[3,4]=m[3,3]+m[4,4] +p[2]*p[3]*p[4]

= 40*30*5=6000

i=2, j=3,k=2

m[2,3]=m[2,2]+m[3,3] +p[1]*p[2]*p[3]

= 10*40*30=12000　k=2

0			
	0	12000	8000
		0	6000
			0

i=2, j=4,k=2~3

m[2,4]=min{m[2][2]+m[3][4] +p[1]*p[2]*p[4]

, m[2][3]+m[4][4] +p[1]*p[3]*p[4]

=min{6000+10*40*5, 12000+10*30*5}

=8000

K=2

3.1 矩阵连乘动态规划法——实例分析

P[0]=50 P[1]=10 P[2]=40 P[3]=30 P[4]=5

m[1,2]=m[1,1]+m[2,2]+p[0]*p[1]*p[2]
=50*10*40=20000

0	20000	27000	10500
	0	12000	8000
		0	6000
			0

m[1,3]=min {m[1,1]+m[2,3]+p[0]*p[1]*p[3]
, m[1,2]+m[3,3]+p[0]*p[2]*p[3]}
={12000+50*10*30,20000+50*40*30}
=27000
k=1

m[1,4]=min {m[1,1]+m[2,4]+p[0]*p[1]*p[4]
, m[1,2]+m[3,4]+p[0]*p[2]*p[4],
m[1,3]+m[4,4]+p[0]*p[3]*p[4] }
=min{8000+2500,20000+8000+10000,
27000+7500}
=10500

S=

0	1	1	1
	0	2	2
		0	3
			0

k=1 $(A_1)(A_2A_3A_4)$

k=2, $A_1\ (A_2\ (A_3\ A_4))$

$$m[1,4]=\min\{m[1,1]+m[2,4]+p[0]*p[1]*p[4],\ m[1,2]+m[3,4]+p[0]*p[2]*p[4],\ m[1,3]+m[4,4]+p[0]*p[3]*p[4]\}$$
$$=\min\{8000+2500,20000+8000+10000,\ 27000+7500\}$$
$$=10500$$

k=1 $(A_1)(A_2A_3A_4)$

k=2, $A_1\ (A_2\ (A_3\ A_4))$

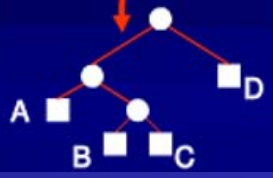
- 例如

4个矩阵连乘积ABCD，设它们的维数分别为

A:50×10，B:10×40, C:40×30, D:30×5。

矩阵连乘的完全加括号形式以及相应的计算量

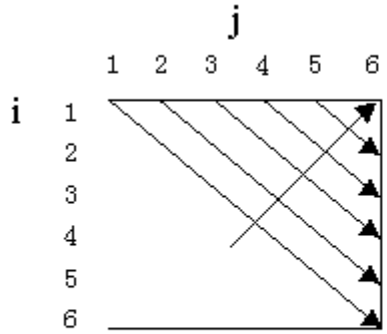
完全加括号形式	连乘积所需计算量
(A(B(CD)))	40×30×5+10×40×5+50×10×5=10500
(A((BC)D))	10×40×30+10×30×5+50×10×5=16000
((AB)(CD))	50×10×40+40×30×5+50×40×5=36000
(((AB)C)D)	50×10×40+50×40×30+50×30×5=87500
((A(BC))D)	10×40×30+50×10×30+50×30×5=34500



每种计算次序都可以用一个二叉树来表示

3.1 矩阵连乘动态规划法——实例分析

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



(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]

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

用动态规划法求最优解 ---- s数组的作用

因此，从s[1][n]记录的信息可知计算A[1:n]的最优加括号方式为：(A[1 : s[1][n]])(A[s[1][n]+1 : n])

如：根据s[i][j]的记录，可以计算A[1:6]的最优加括号方式：

(A1A2A3)(A4A5A6)　s[1][n]=3

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

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

3.1 矩阵连乘动态规划法——练习题

矩阵连乘实例

例：设要计算矩阵连乘$A_1A_2A_3A_4A_5$，其中各矩阵的维数分别为：

A_1：3×5　　A_2：5×10

A_3：10×4　　A_4：4×2

A_5：2×5

3.1 矩阵连乘动态规划法——练习题

矩阵连乘实例

解：

m[i][j]	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

m[1][2] = m[1][1] + m[2][2] + 3×5×10=150

s[1][2] = 1

m[2][3] = m[2][2] + m[3][3] + 5×10×4 = 200

s[2][3] = 2

m[3][4] = m[3][3] + m[4][4] + 10×4×2 = 80

s[3][4] = 3

m[4][5] = m[4][4] + m [5][5] + 4×2×5 = 40

s[4][5] = 4

m[i][j]	1	2	3	4	5
1	0	150 (1)			
2		0	200 (2)		
3			0	80 (3)	
4				0	40 (4)
5					0

3.1 矩阵连乘动态规划法——练习题

$$m[1][3] = \min\{m[1][1] + m[2][3] + 3\times5\times4 = 260,$$
$$m[1][2] + m[3][3] + 3\times10\times4 = 270\}$$
$$= 260 \qquad s[1][3] = 1$$

$$m[2][4] = \min\{m[2][2] + m[3][4] + 5\times10\times2 = 180,$$
$$m[2][3] + m[4][4] + 5\times4\times2 = 240\}$$
$$= 180 \qquad s[2][3] = 2$$

$$m[3][5] = \min\{m[3][3] + m[4][5] + 10\times4\times5 = 240,$$
$$m[3][4] + m[5][5] + 10\times2\times5 = 180\}$$
$$=180 \qquad s[3][5] = 4$$

m[i][j]	1	2	3	4	5
1	0	150 (1)	260 (1)		
2		0	200 (2)	180 (2)	
3			0	80 (3)	180 (4)
4				0	40 (4)
5					0

3.1 矩阵连乘动态规划法—— 练习题

$$m[1][4] = \min\{m[1][1] + m[2][4] + 3\times5\times2 = 210,$$
$$m[1][2] + m[3][4] + 3\times10\times2 = 290$$
$$m[1][3] + m[4][4] + 3\times4\times2 = 284\}$$
$$= 210 \qquad s[1][4] = 1$$
$$m[2][5] = \min\{m[2][2] + m[3][5] + 5\times10\times5 = 430,$$
$$m[2][3] + m[4][5] + 5\times4\times5 = 300,$$
$$m[2][4] + m[5][5] + 5\times2\times5 = 230\}$$
$$= 230 \qquad s[2][5] = 4$$

m[i][j]	1	2	3	4	5
1	0	150 (1)	260 (1)	210 (1)	
2		0	200 (2)	180 (2)	230 (4)
3			0	80 (3)	180 (4)
4				0	40 (4)
5					0

3.1 矩阵连乘动态规划法——练习题

$$m[1][5] = \min\{m[1][1] + m[2][5] + 3\times5\times5 = 305,$$
$$m[1][2] + m[3][5] + 3\times10\times5 = 480,$$
$$m[1][3] + m[4][5] + 3\times4\times5 = 360,$$
$$m[1][4] + m[5][5] + 3\times2\times5 = 240\}$$
$$= 240 \qquad s[1][5] = 4$$

m[i][j]	1	2	3	4	5
1	0	150 (1)	260 (1)	210 (1)	240 (4)
2		0	200 (2)	180 (2)	230 (4)
3			0	80 (3)	180 (4)
4				0	40 (4)
5					0

最优值 = 240

回溯求最优解：

s[1][5] = 4　　$(A_1A_2A_3A_4)A_5$

s[1][4] = 1　　$(A_1(A_2A_3A_4))A_5$

s[2][4] = 2　　$(A_1(A_2(A_3A_4))A_5$

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。**从一般意义上讲，问题的这两个重要性质是该问题可以用动态规划算法求解的基本要素：
  - **最优子结构**
  - **重叠子问题**
  - **备忘录方法**

3.2 动态规划算法的基本要素--备忘录方法

- 属于动态规划算法的变形。也用一个表格来保存已解决的子问题的答案。
- 区别：备忘录方法的递归方式是自顶向下的，而动态规划算法是自底向上的。

```
int LookupChain(int i，int j)
{
    if (m[i][j] > 0) return m[i][j];
    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;
}
```

关于动态规划算法和备忘录方法的适用条件

- 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方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，对每个子问题两种算法都只解一次并记录答案。当再次遇到该子问题时，简单地取用已得到的答案，节省了计算量，提高了算法的效率。
- 适用条件：一般来说，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，还可以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。当子问题空间中部分子问题可以不必求解时，易用备忘录方法则较为有利，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

- 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：$z_j=x_{i_j}$。
- 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}。

3.3 最长公共子序列

- 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：
- X={A,B,C,B,D,A,B}，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。

3.3 最长公共子序列

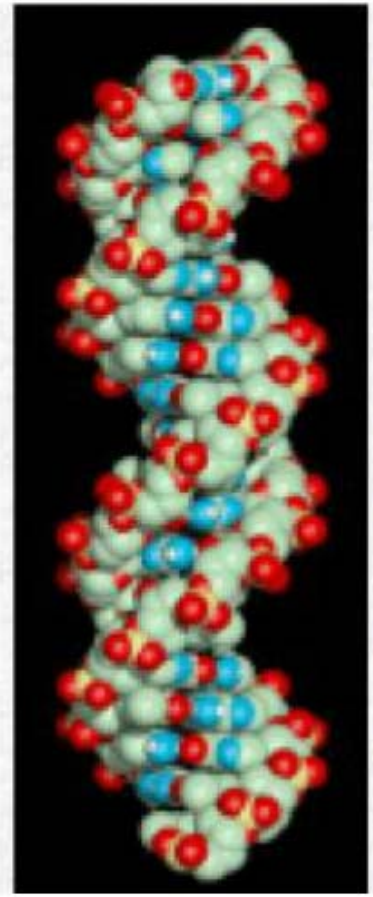
例：

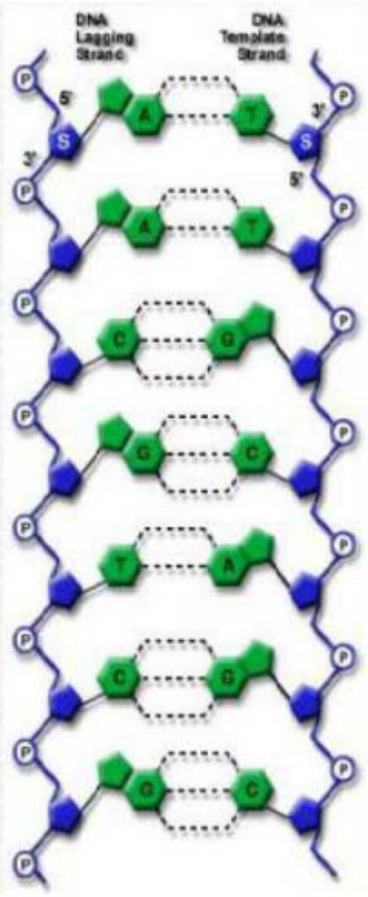
- 两个DNA序列

S1=ACCGGTCGAGTGCGC
GGAAGCCGGCCGAA

S2=GTCGTTCGGAATGCC
GTTGCTCTGTAAA

LCS=GTCGTCGGAAGCCG
GCCGAA





3.3 最长公共子序列

- 在软件工程应用中，两个串可能来自同一个程序的源代码的两个版本，希望确定两个版本之间有哪些变化。
- 搜索引擎中的数据收集系统(也称**web**蜘蛛或爬虫)必定能区分相似的**web**页面，避免不必要的**web**页面请求。
- **Unin/linux**中提供**diff**程序，用于比较文本文件。

3.3 最长公共子序列

穷举搜索法

- 对**X**的所有子序列，检查它是否也是**Y**的子序列。
- **X**的每个子序列相应于下标集**{1,2,…,m}**的一个子集，因此共有2^m个不同子序列，穷举搜索法需要指数时间。

3.3 最长公共子序列

1. 最长公共子序列的结构

- 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。

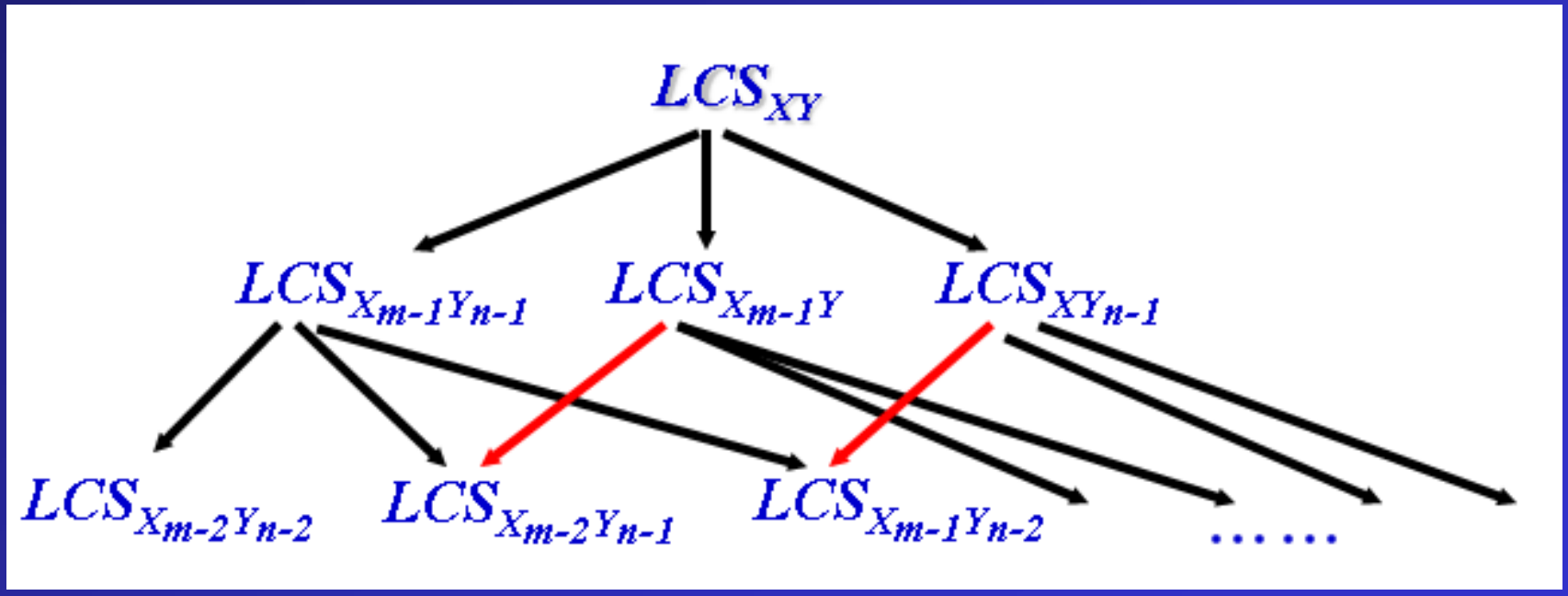
由此可见，2个序列的最长公共子序列包含了这2个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有最优子结构性质。

X和Y的LCS的优化解结构为

$$LCS_{XY}=LCS_{X_{m-1}Y_{n-1}}+\langle x_m=y_n\rangle \quad \text{if } x_m=y_n$$

$$LCS_{XY}=LCS_{X_{m-1}Y} \quad \text{if } x_m\neq y_n,\ z_k\neq x_m$$

$$LCS_{XY}=LCS_{XY_{n-1}} \quad \text{if } x_m\neq y_n,\ z_k\neq y_n$$



LCS问题具有子问题重叠性

3.3 最长公共子序列

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建立子问题最优值的递归关系。用**c[i][j]**记录序列的最长公共子序列的长度。其中，$X_i=\{x_1,x_2,\ldots,x_i\}$；$Y_j=\{y_1,y_2,\ldots,y_j\}$。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**。其它情况下，由最优子结构性质可建立递归关系如下：

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

3.3 最长公共子序列

3. 计算最优值

- 由于在所考虑的子问题空间中，总共有**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。
- 输入：**x,y** （序列数组）
- 输出：
  - c[i][j]，存储x[1:i]和y[1:j]的最长公共子序列的长度；
  - b[i][j]，记录上面c[i][j]的值是由哪个子问题的解得到的。

3.3 最长公共子序列

4. 构造最长公共子序列

- 从**b[m][n]** 开始，依其值在数组**b**中搜索。
- **b[i][j]**的值为：
  - 1，表示X_i和Y_i的最长公共子序列是由X_{i-1}和Y_{i-1}的最长公共子序列在尾部加上x_i所得到的。
  - 2，表示X_i和Y_i的最长公共子序列与X_{i-1}和Y_i的最长公共子序列相同。
  - 3， 表示X_i和Y_i的最长公共子序列与X_i和Y_{i-1}的最长公共子序列相同。

计算最优值

- 由于在所考虑的子问题空间中，总共有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。

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

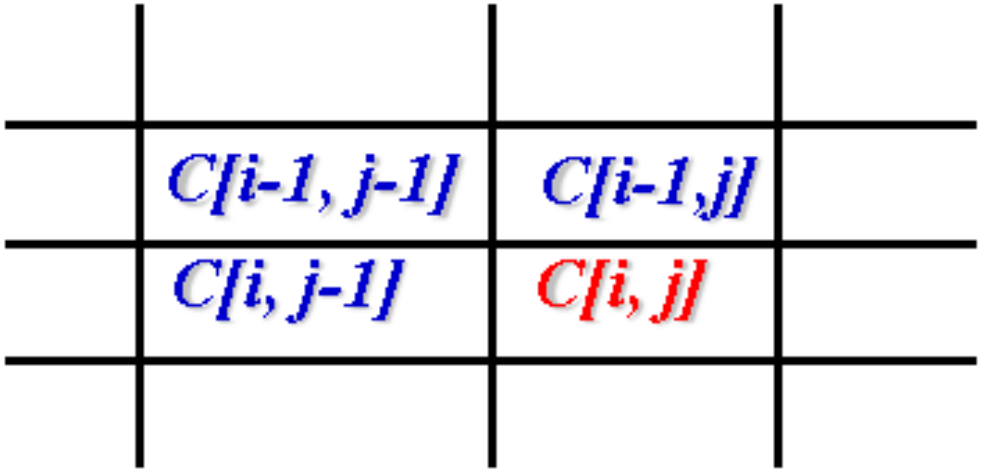
构造最长公共子序列

- 算法描述：

```
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);
}
```

自底向上计算LCS的长度

- 基本思想



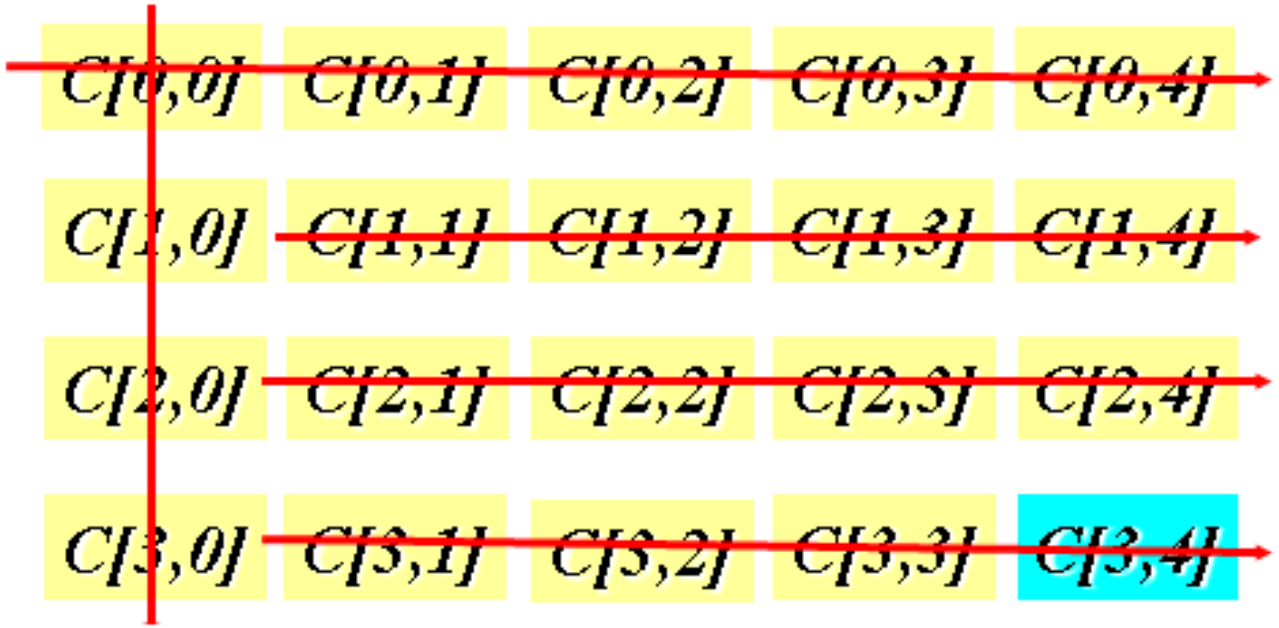
- *LCS*长度的递归方程

$C[i,j] = 0$ if $i=0$ 或 $j=0$

$C[i,j] = C[i-1,j-1] + 1$ if $i,j>0$ and $x_i = y_j$

$C[i,j] = Max(C[i,j-1], C[i-1,j])$ if $i,j>0$ and $x_i \neq y_j$

- 计算过程



- *LCS*长度的递归方程

$$C[i,j] = 0 \quad \text{if } i=0 \text{ 或 } j=0$$

$$C[i,j] = C[i-1,j-1] + 1 \quad \text{if } i,j>0 \text{ and } x_i = y_j$$

$$C[i,j] = Max(C[i,j-1], C[i-1,j]) \quad \text{if } i,j>0 \text{ and } x_i \neq y_j$$

```
LCS-LENGTH(X, Y, m, n)
1  for i ← 1 to m   do  c[i, 0] ← 0
2  for j ← 0 to n    do c[0, j] ← 0
3  for i ← 1 to m    do
4       for j ← 1 to n do
5           if x_i = y_j
6           then 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                then c[i, j] ← c[i - 1, j]
10                    b[i, j] ← "↑"
11               else c[i, j] ← c[i, j - 1]
12                    b[i, j] ← "←"
13 return c and b
```

运行时间: $\Theta(nm)$

构造优化解

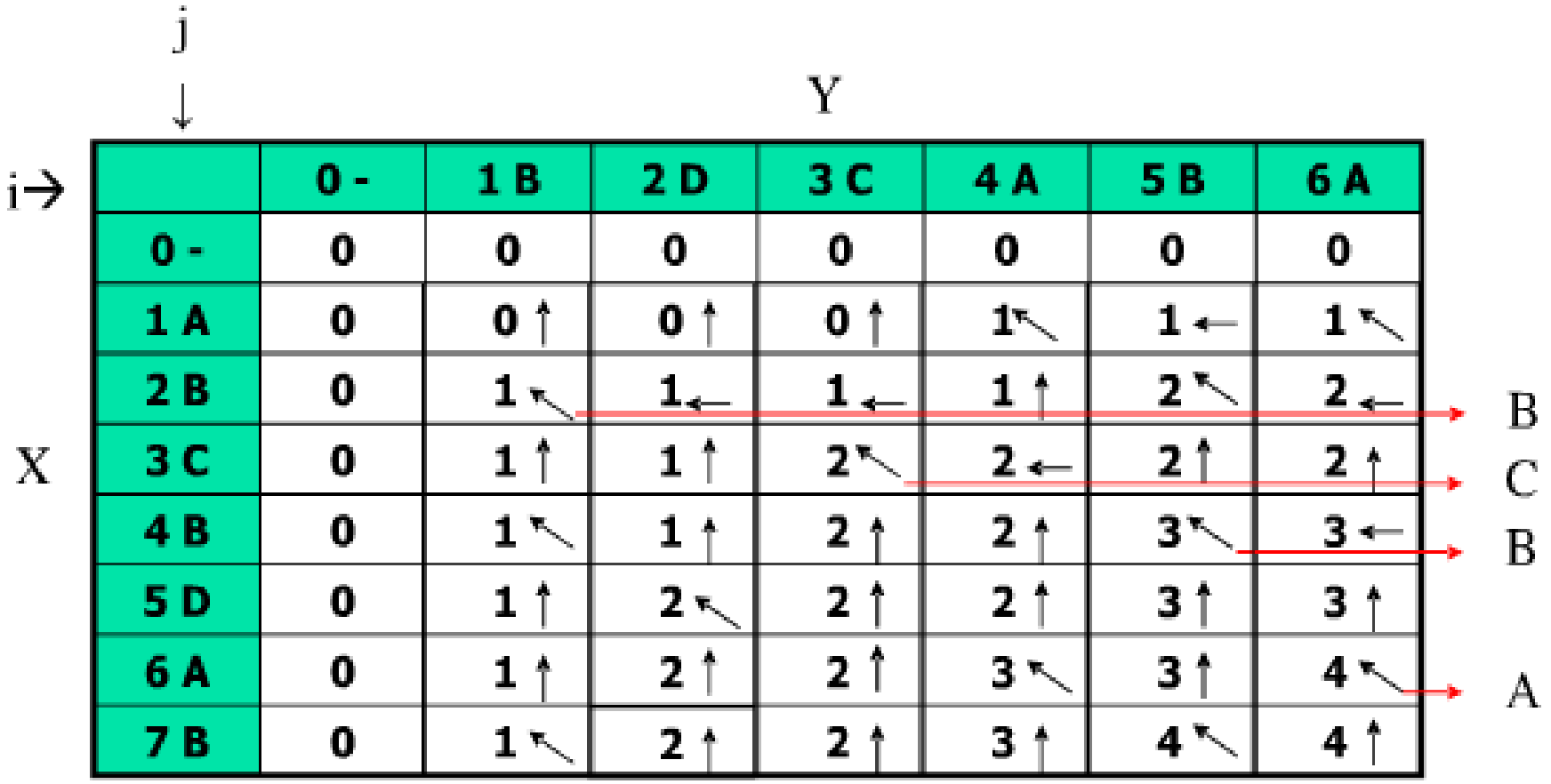
- 基本思想
  - 从B[m, n]开始按指针搜索
  - 若B[i, j]=“↖”，则$x_i=y_j$是LCS的一个元素
  - 如此找到的“LCS”是X与Y的LCS的Inverse

例子

X = (B, D, C, A, B, A)
Y = (A, B, C, B, D, A,B)

if $x_i = y_j$
then $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
$b[i, j] \leftarrow$ "↖"
else if $c[i - 1, j] \geq c[i, j - 1]$
then $c[i, j] \leftarrow c[i - 1, j]$
$b[i, j] \leftarrow$ "↑"
else $c[i, j] \leftarrow c[i, j - 1]$
$b[i, j] \leftarrow$ "←"

		0	1	2	3	4	5	6
		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



对b递归推导得解：BCBA

最长公共子序列----练习题

X={A,B,C,B,D,A,B}， Y={B,D,C,A,B,A}

最长公共子序列----课前练习

例：序列$X=(a, b, c, b, d, b)$，$Y=(a, c, b, b, a, b, d, b, b)$，建立两个$(m+1)\times(n+1)$的二维表L和表S，分别存放搜索过程中得到的子序列的长度和状态。

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))。