动态规划

陈长建

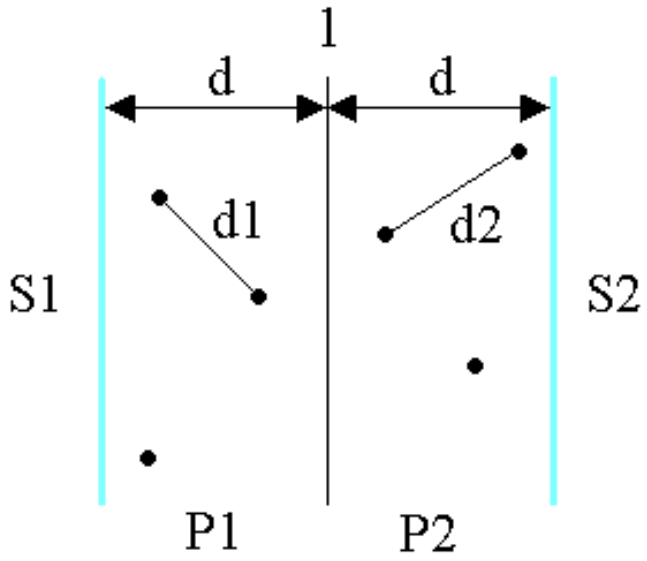
计算机科学系

关于第二次实验

- **院楼103**，11月3日上午8:30-12:00（现场验收）
- 第二次实验离线题（离线准备）
  - 1. 用动态规划法实现0-1背包
  - 2. 半数集问题（实现题2-3）
  - 3. 集合划分问题（实现题2-7）
- 在线题
  - acm.hnu.edu.cn

回顾-最接近点对问题

- 给定平面上n个点的集合S，找其中的一对点，使得在n个点组成的所有点对中，该点对间距离最小。
- 对于P1中的每个点，最多需要检查多少个P2中的点?



本章要点:

- 理解动态规划算法的概念
- 掌握动态规划算法的基本要素
- 掌握设计动态规划算法的步骤
- 通过范例学习动态规划算法设计策略
  - 最大子段和、最长公共子序列、矩阵链连乘、凸多边形最优三角剖分、0-1背包问题

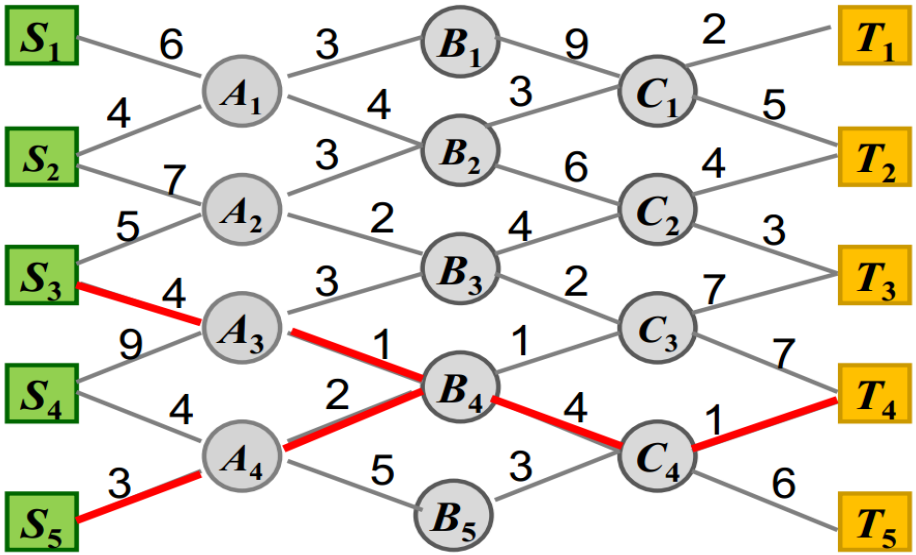
动态规划算法适用于求解最优化问题

通常按如下四步骤设计动态规划算法：

- 找出最优解的性质，并刻画其结构特征；
- 递归定义求最优值的公式；
- 以自底向上方式计算最优值；
- 根据计算最优值时得到的信息，构造最优解。

4个步骤分解

- 找出最优解的性质，并刻画其结构特征；
  - 对于第l列中的某一个节点，其到终点的最短路径是其到第l+1列中任意一个节点加上该节点到终点距离中最短一个 **正确性?**



4个步骤分解

- 递归定义求最优值的公式；

$$F(C_l) = \min_m \{C_l T_m\}$$

$$F(B_k) = \min_l \{B_k C_l + F(C_l)\}$$

$$F(A_j) = \min_k \{A_j B_k + F(B_k)\}$$

$$F(S_i) = \min_i \{S_i A_j + F(A_j)\}$$

4个步骤分解

- 以自底向上方式计算最优值（递归实现）

$$F(C_l) = \min_m \{C_l T_m\}$$

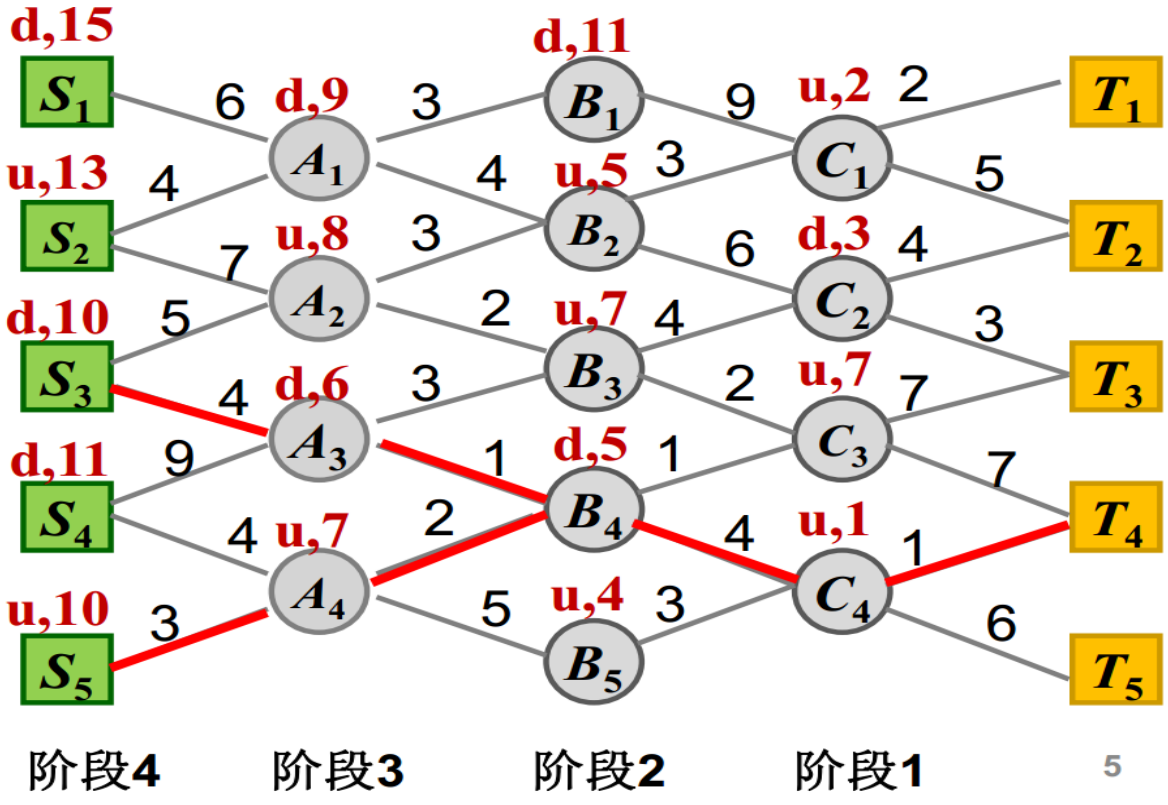
$$F(B_k) = \min_l \{B_k C_l + F(C_l)\}$$

$$F(A_j) = \min_k \{A_j B_k + F(B_k)\}$$

$$F(S_i) = \min_i \{S_i A_j + F(A_j)\}$$

4个步骤分解

- 以自底向上方式计算最优值；



4个步骤分解

- 根据计算最优值时得到的信息，构造最优解。

$$F(S_i) = \min_i \{S_i A_j + F(A_j)\}$$

例2 - 矩阵连乘问题

给定n个矩阵{A_1， A_2，…, A_n},其中A_i与A_{i+1}是可乘的。考察这N个矩阵的连乘积 $A_1A_2...A_n$

分析：

由于矩阵乘法满足结合律，故计算矩连乘积 $A_1A_2...A_n$ 可以有多个计算次序。

例如：
- $(A_1(A_2(A_3A_4)))$ 等等
- $(A_1((A_2A_3)A_4))$
- $((A_1A_2)(A_3A_4))$

先考虑两个矩阵乘积的计算量

设A为ra×ca矩阵 B为rb×cb矩阵，

```
void  matrixMultiply(int **a, int **b,  int **c,  int ra,  int ca,  int rb, int cb )
{   if(ca!=rb) error(“Can’ t multiply”);
    for(i=0; i<ra; ++i)
      for(j=0; j<cb; ++j)
        {    c[i][j] =0;
             for(k=0;k<ca; ++k)  c[i][j]+=a[i][k]*b[k][j];
 }
}
```

复杂度：O(ra × rb × cb)

不同计算次序会导致不同的计算量

例如 $\{A_1, A_2, A_3\}$

A_1 : [10*100]

A_2 : [100*5]

A_3 : [5*50]

- 第1种加括号

  $((A_1A_2)A_3)$

计算量：10*100*5+10*5*50 =7500

- 第2种加括号

  $(A_1(A_2A_3))$

计算量：100*5*50+10*100*50 =75000

所谓矩阵连乘问题：

对于给定n个矩阵{A_1， A_2，…, A_n}，其中A_i的维数是$p_{i-1} \times p_i$

如何确定计算矩阵连乘积$A_1A_2...A_n$的计算次序，使得计算矩阵连乘积的数乘次数最少。

矩阵连乘的递归求解方法

记 A[i:j]为 $A_iA_{i+1}...A_j$ ，考察计算A[1:n]的最优计算次序。

不妨设

计算A[1:n]最优次序的最后一次乘积位置在 K 处

$$(\ (A_1A_2...A_k)\ (A_{k+1}...A_n)\ )\quad k=1,2,...n-1$$

其计算量为:

(1)计算 A[1:k]

(2)计算 A[k+1:n]

(3)最后一次矩阵乘积$p_0 \times p_k \times p_n$

1. 分析最优解的结构

计算A[1:n]的最优次序所包含的子链计算

A[1:k] 和 A[k+1:n]也一定是最优次序的。

事实上

若有一个计算A[1:k]的次序所需的计算量更少，则取代之。类似，计算A[k+1:n]的次序也一定是最优的。

因此，矩阵连乘计算次序问题的最优解，包含了其子问题的最优解。

2. 递归定义求最优值的公式

- 设计算A[i : j]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]。
- 不妨设

计算A[i:j]最优次序的最后一次乘积位置在 K 处

$(\ (A_iA_2\ldots A_k)\ (A_{k+1}\ldots A_j))$ k=i,2,...j-1

则

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

2. 递归定义求最优值的公式

计算最优值：

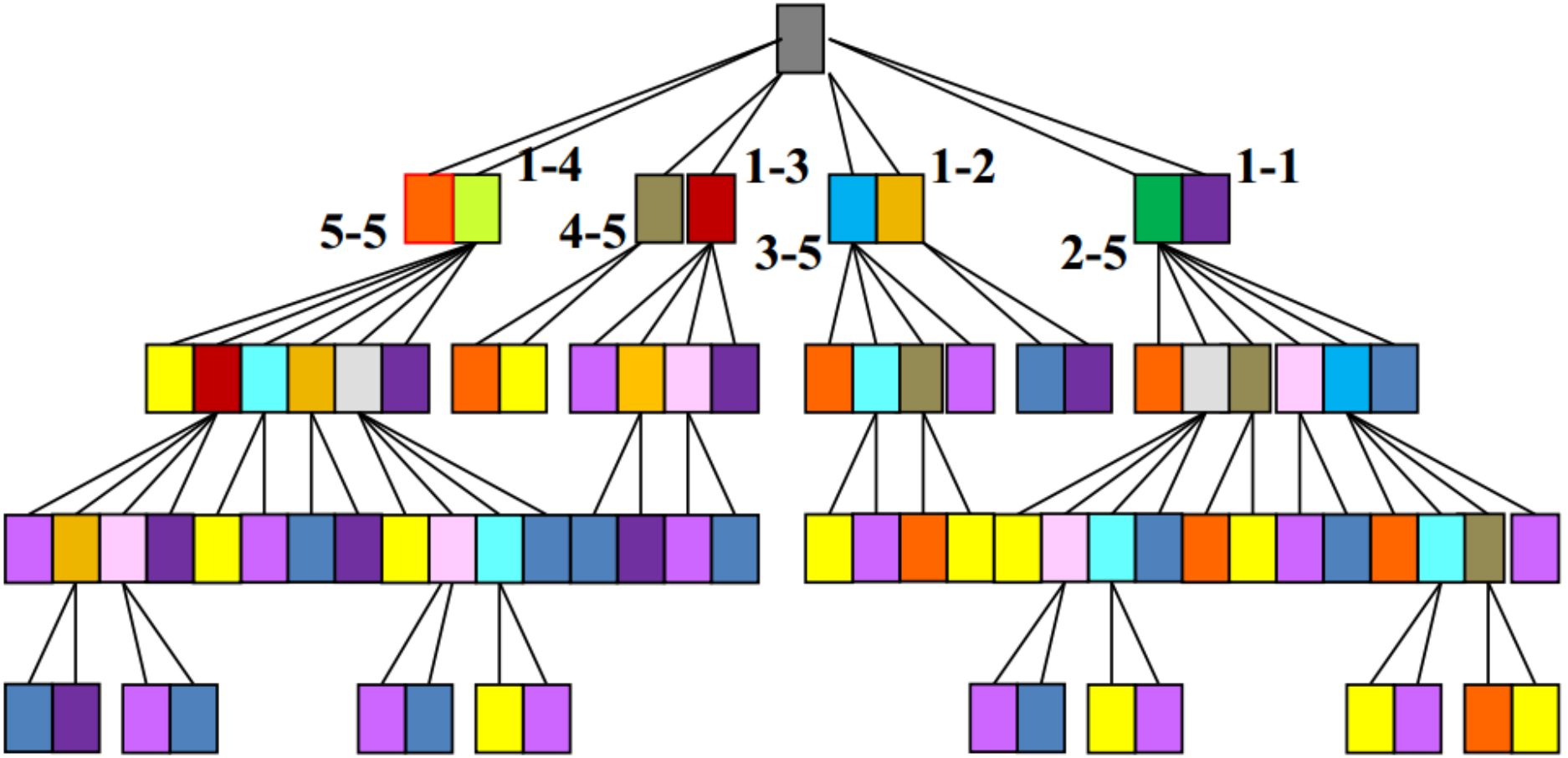
对于1≤i≤j≤n不同的有序对(i,j)对应于不同的子问题。因此，不同子问题的个数最多只有

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

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。

2. 递归定义求最优值的公式



2. 递归定义求最优值的公式

边界	次数	边界	次数	边界	次数
1-1	8	1-2	4	2-4	2
2-2	12	2-3	5	3-5	2
3-3	14	3-4	5	1-4	1
4-4	12	4-5	4	2-5	1
5-5	8	1-3	2	1-5	1

边界不同的子问题：15个

递归计算的子问题：81个

3. 以自底向上方式计算最优值

动态规划实现的关键：

- 每个子问题只计算一次
- 迭代过程
  - 从最小的子问题算起
  - 考虑计算顺序，以保证后面用到的值前面已经计算好
  - 存储结构保存计算结果——备忘录
- 解的追踪
  - 设计标记函数标记每步的决策
  - 考虑根据标记函数追踪解的算法

3. 以自底向上方式计算最优值

矩阵连乘的不同子问题：

长度1：只含1个矩阵，有n个子问题(不需要计算)

长度2：含2个矩阵， n-1个子问题

长度3：含3个矩阵， n-2个子问题

…

长度n-1：含n-1个矩阵， 2个子问题

长度n：原始问题，只有1个

3. 以自底向上方式计算最优值

矩阵链乘法迭代顺序：

长度为1： 初值， $m[i, i] = 0$

长度为2： 1..2, 2..3, 3..4, … , n-1..n

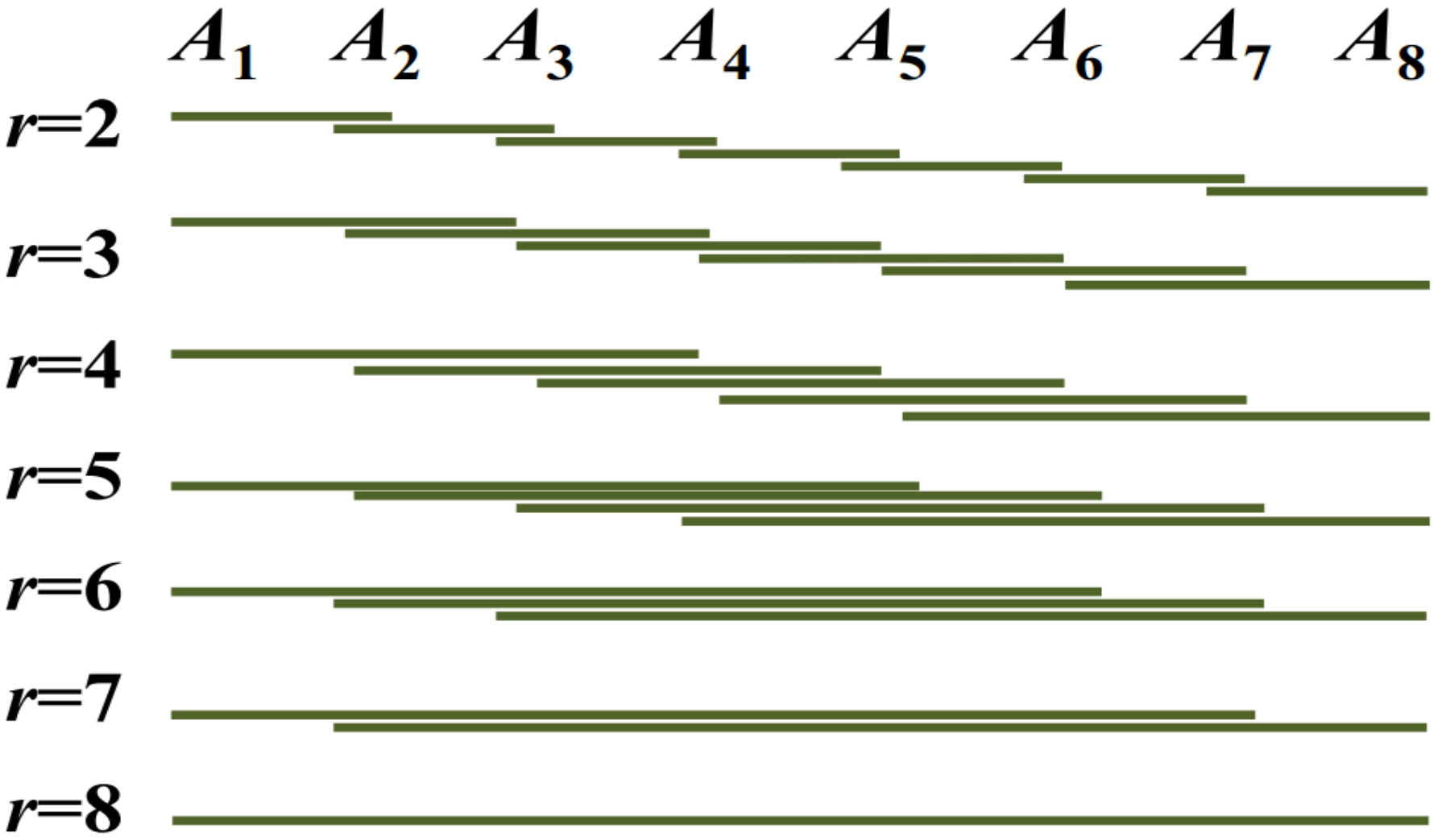
长度为3： 1..3, 2..4, 3..5, … , n-2..n

…

长度为n-1： 1..n-1, 2..n

长度为n： 1..n

3. 以自底向上方式计算最优值



3. 以自底向上方式计算最优值

```
void   MatricChain(int *p,  int n,  int **m,  int **s)
{    for(i=1;i<=n; ++i)  m[i][i]=0; //单个矩阵无计算
     for(r=2; r<=n; ++r)    //连乘矩阵的个数
        for(i=1; i<n-r; ++i)
        {    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(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;
}     }   }
```

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$O(n^2)$。

3. 以自底向上方式计算最优值

实例分析：

输入： $P = < 30, 35, 15, 5, 10, 20 >, n = 5$

矩阵链： $A1\ A2\ A3\ A4\ A5$，其中

$A1$: 30×35， $A2$: 35×15， $A3$: 15×5，

$A4$: 5×10， $A5$: 10×20

备忘录：存储所有子问题的最小乘法次数及得到这个值的划分位置。

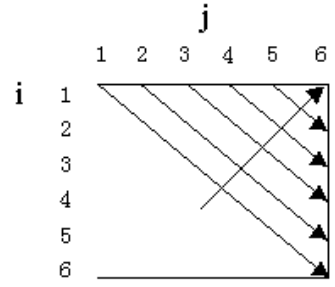
3. 以自底向上方式计算最优值

备忘录m[i,j], P=<30,35,15,5,10,20>

r=1	*m*[1,1]=0	*m*[2,2]=0	*m*[3,3]=0	*m*[4,4]=0	*m*[5,5]=0
r=2	*m*[1,2]=15750	*m*[2,3]=2625	*m*[3,4]=750	*m*[4,5]=1000	
r=3	*m*[1,3]=7875	*m*[2,4]=4375	*m*[3,5]=2500		
r=4	*m*[1,4]=9375	*m*[2,5]=7125			
r=5	*m*[1,5]=11875				

m[2,5] = min{ 0+2500+35×15×20 2625+1000+35×5×20

4375+0+35×10×20 = 7125



(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]

4. 构造最优解

标记函数s[i,j]：

r=2	s[1,2]=1	s[2,3]=2	s[3,4]=3	s[4,5]=4
r=3	s[1,3]=1	s[2,4]=3	s[3,5]=3	
r=4	s[1,4]=3	s[2,5]=3		
r=5	s[1,5]=3			

解的追踪：　$s[1,5]=3 \Rightarrow (A1A2A3)(A4A5)$

$s[1,3]=1 \Rightarrow A1(A2A3)$

输出

计算顺序：　$(A1(A2A3))(A4A5)$

最少的乘法次数：　$m[1,5]=11875$

4. 构造最优解

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

小练习：合并果子-1

在一个果园里，果农已经将所有的果子打了下来，而且按圆形区域堆放了若干堆。最后果农要把所有的果子合并成一堆。

每一次合并，果农总是把两堆相邻果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过n-1次合并之后，就只剩下一堆了。果农在合并果子时总共消耗的体力等于每次合并所耗体力之和。

假定每个果子重量都为1，并且已知果子堆数和每堆的数目，你的任务是设计出合并的次序方案，使得果农耗费的体力最少，并输出这个最小的体力耗费值。

示例

- N=6:　　3　　4　　6　　5　　4　　2

例如：

一、考虑每次选相邻的最小两堆合并

3 4 6 5 4 2 → 5

5 4 6 5 4 → 9

9 6 5 4 → 9

9 6 9 → 15

15 9 → 24

24

62

例如： N=6　　3　　4　　6　　5　　4　　2

二、更好方案

3　4　　6　　5　　4　　2　　→　7

7　6　　5　　4　　2　　→　13

13　5　4　2　　→　6

13　5　6　　→　11

13　11　　→　24

24

61

利用动态规划求解

相邻堆进行合并有多个子序列：

$\{a_1\ a_2\}$　$\{a_2\ a_3\}$　... $\{a_n\ a_1\}$

$\{a_1\ a_2\ a_3\}$ $\{a_2\ a_3\ a_4\}$　$\{a_n\ a_1\ a_2\}$

...

$\{a_1\ a_{2\ldots..}\ a_{n-1}\}$..... $\{a_n\ a_{1\ldots.}\ a_{n-2}\}$

利用动态规划求解2

为了便于运算，用[i，j]表示从第i堆起,顺时针的j堆组成的子序列。

Data[i][j]=a[i]+a[i+1]+...+a[i+j-1]

Fm[i][j] 表示从第i堆起,顺时针的j堆合并成一堆的最小值

显然

Fm[i][1]=0;

Fm[i][j]=MIN{ Fm[i][k] + Fm[i+k][j-k]
+data[i][j] }

k=2,3,...,j-1

求Fm[1][n]

注意下标越界问题

例3-最长公共子序列

定义 一个给定序列的子序列是在该序列中删除若干元素后得到的序列。即

若给定序列 X={x1,x2,...,xm}，则另一序Z={z1,...,zk}是X的子序列是指:

存在一个严格递增下标序列{i1,i2,...,ik}使得对于所有j=1,2,...,k有：$Z_j=X_{ij}$

例如 序列 X={A，B，C，B，D，A，B}

子序列 Z={B，C，D，B}

相应的递增下标序列为{2，3，5，7}。

最长公共子序列

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。

最长公共子序列问题

给定2个序列 X={x1,x2,...,xm}和 Y={y1,y2,...,yn}，找出 X 和 Y 的最长公共子序列。

例如：字符串13455和245576的最长公共子序列为455

字符串acdfg和adfc的最长公共子序列为adf

最长公共子序列

LCS（Longest Common Subsequence）的应用

- 求两个序列中最长的公共子序列算法，广泛的应用在图形相似出路、媒体流的相似比较、计算生物学方面。生物学家常常利用该算法进行基因序列比对，由此推测序列的结构、功能和演化过程。
- LCS可以描述两段文字之间的“相似度”，即它们的雷同程度，从而能够用来辨别抄袭。另一方面，对一段文字进行修改之后，计算改动前后文字的最长公共子序列，将除此子序列外的部分提取出来，这种方法判断修改的部分，往往十分准确。简而言之，百度知道，百度百科都用得上。

最长公共子序列的结构

求最长公共子序列问题，最容易想到的算法----枚举法

即，对X的所有子序列，检查它是否也是Y的子序列，从中找出最长的子序列。

但 X 共有2^M个不同的子序列。枚举法 $O(2^m)$

事实上，

最长公共子序列问题，具有最优子结构性质。

最优子结构性质

设序列$X_m=\{x_1,x_2,\ldots,x_m\}$和$Y_n=\{y_1,y_2,\ldots,y_n\}$的最长公共子序列为$Z_k=\{z_1, z_2,\ldots, z_k\}$，则

- 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。
- 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
- 若$x_m\neq y_n$ 且 $z_k\neq y_n$ 则 Z是X_m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。

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。

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，

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

由最优子结构性质可建立递归关系如下：

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

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的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 = 1; i <= m; i++)  c[i][0] = 0;
    for (i = 1; 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]="↖";   }
        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]="←";   }
}     }
```

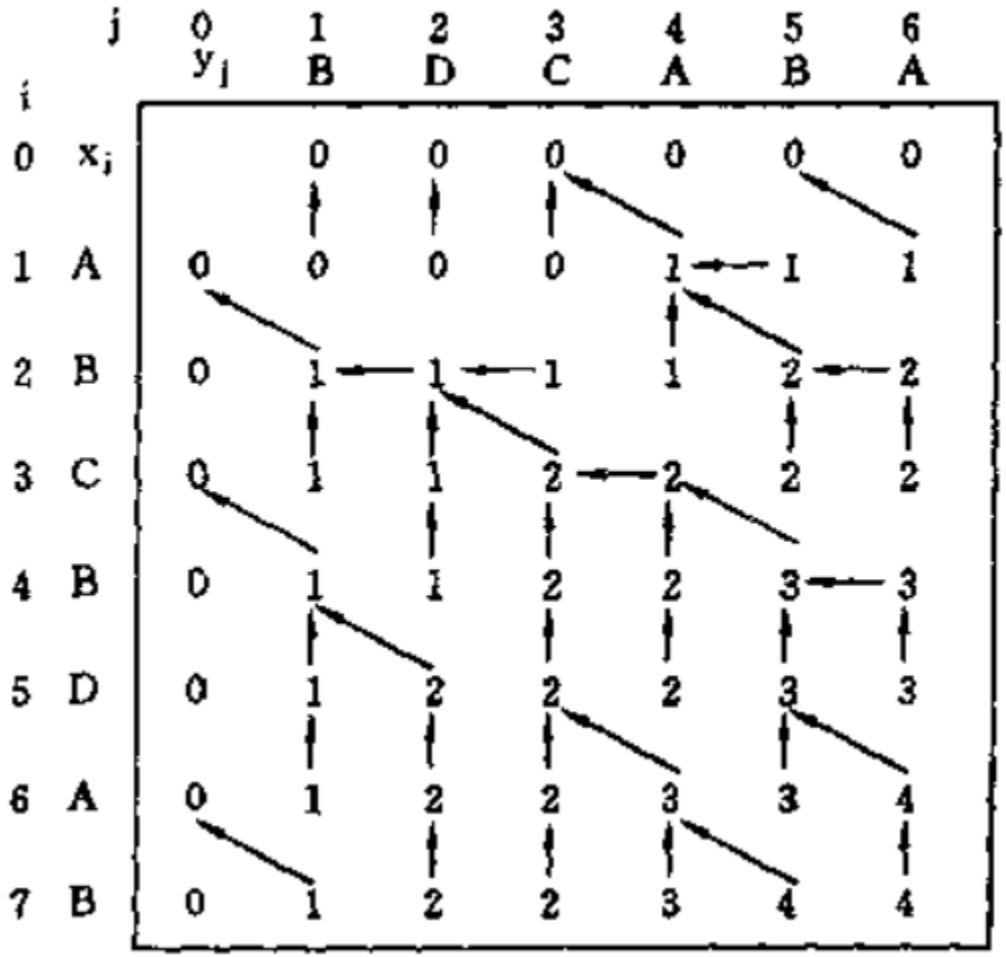
O(mn)

构造最长公共子序列

```
void  LCS(int i，  int j，  char *x，  int **b)
   {
     if (i ==0 || j==0) return;
     if  (b[i][j]== “↖”)
                    {    LCS(i-1，  j-1，  x，  b);
                          cout<<x[i];          }
    else
      if (b[i][j]== “↑”)
            LCS(i-1，  j，  x，  b);
      else
            LCS(i，  j-1，  x，  b);
}
```

O(m+n)

结果



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

LCS:<B,C,B,A>

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。

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))。

Question?

•题目：给定一个长度为N的数组，找出一个最长的单调递增子序列。

•例如：给定数组{5，6，7，1，2，8}则其最长的单调递增自序列为{5，6，7，8}，长度为4。

•求解：怎么用LCS解决这个问题?

•分析：原数组={5，6，7，1，2，8}

排序后={1，2，5，6，7，8}