第5章 动态规划

林 海

Lin.hai@whu.edu.cn

引例：费氏数列

- 费氏数列是由13世纪的意大利数学家、来自Pisa的 Leonado Fibnacci发现。
- 费氏数列是由0，1开始，之后的每一项等于前两项之和：

  0，1，1，2，3，5，8，13，21，34，55，89，144..... 。

引例：费氏数列

- 这个数列有如下一些特性：
  - 前2个数相加等于第3个数
  - 前1个数除以后一个数越往后越无限接近于0.618 (黄金分割)
  - 相邻的两个比率必是一个小于0.618一个大于0.618
  - 后1个数除以前一个数越往后越无限接近于1.618
  - …

$$f(n)=\begin{cases}1 & ,\ if\ n=1,2\\ f(n-1)+f(n-2), & if\ n\geq 3\end{cases}$$

引例：费氏数列

$$f(n)=\begin{cases}1 & ,\ if\ n=1,2\\ f(n-1)+f(n-2), & if\ n\geq 3\end{cases}$$

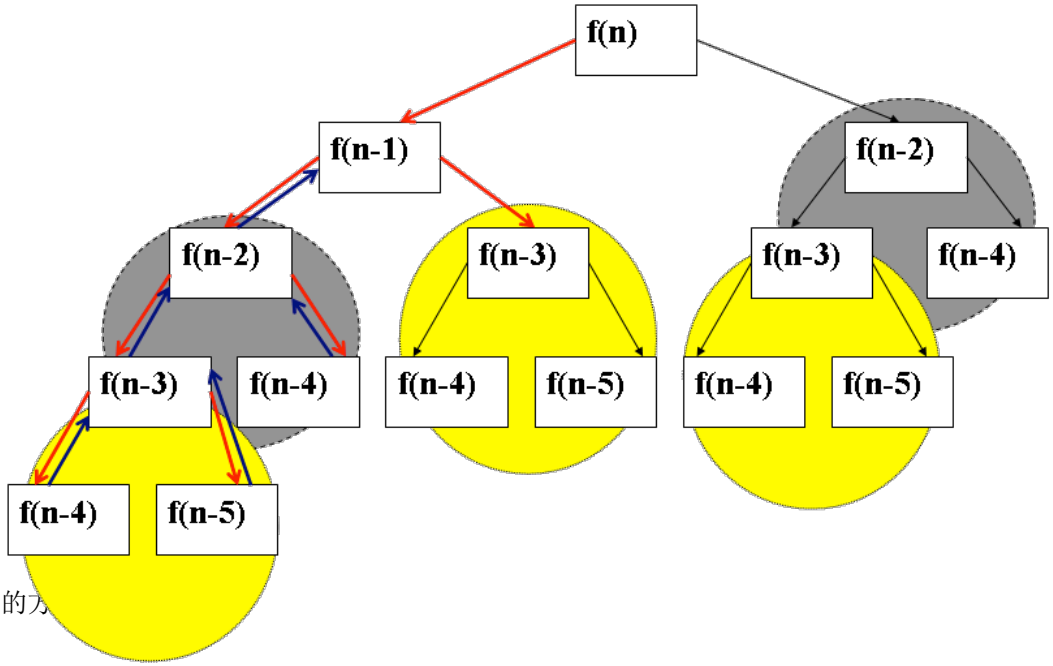
递归形式的算法：

```
procedure fib(n)
    if n=1 or n=2 then return 1
    else return fib(n-1)+fib(n-2)
```

优点： → 简洁，容易书写以及调试。

缺点： → 效率低下 。

为何效率低下？



- 使用直观的方

存在大量重复计算

为何效率低下？

- 使用时间复杂性的方式分析

$$T(n)=\begin{cases}1 & if\ n=1,2\\ T(n-1)+T(n-2) & if\ n\geq 3\end{cases}$$

$$\longrightarrow\quad T(n)\approx\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n}\approx 0.447(1.618)^{n}$$

即时间复杂度为输入规模的指数形式。当n=100时，用递归求解的时间T(100)≈3.53×10^{20}, 若每秒计算10^{8}次，需111,935年!

解决方法

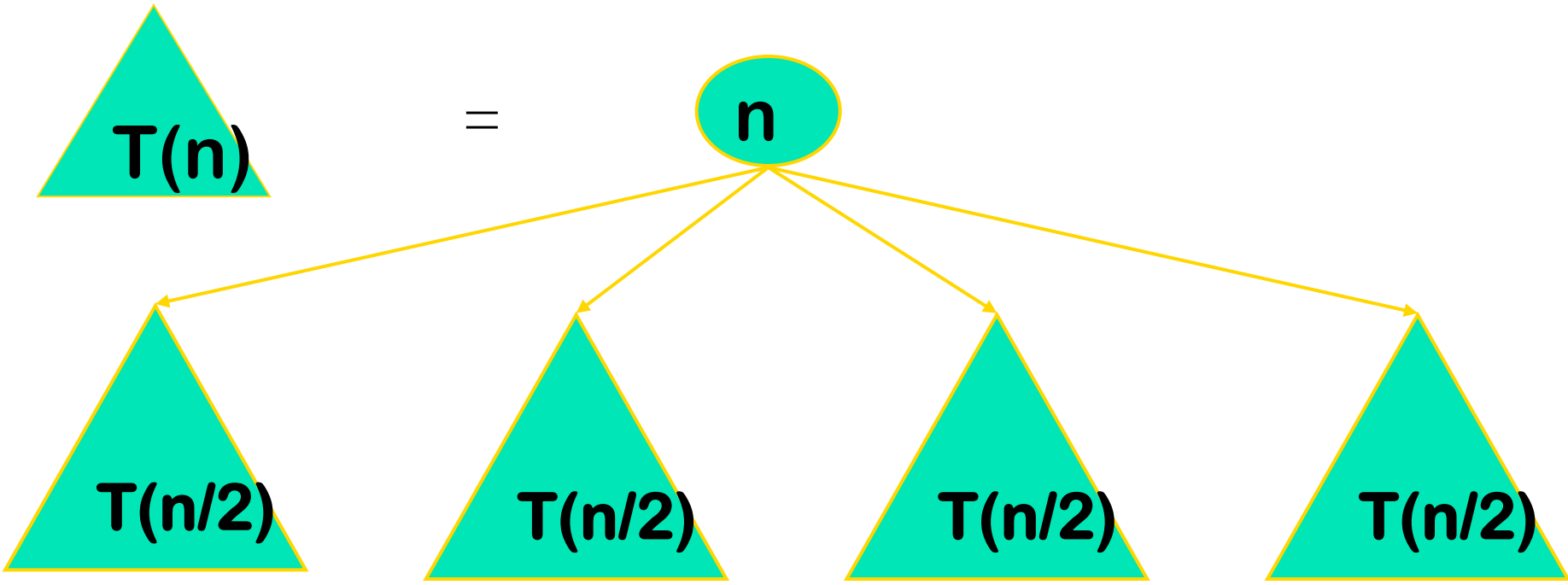
- 借助于变量存储中间计算结果，消除重复计算。代码片断如下：

```
f1 ← 1
f2 ← 1
for  i ← 3  to n
   result ← f1+f2
   f1 ← f2
   f2 ← result
end for
return result
```

$$T(n) = \Theta(n)$$

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

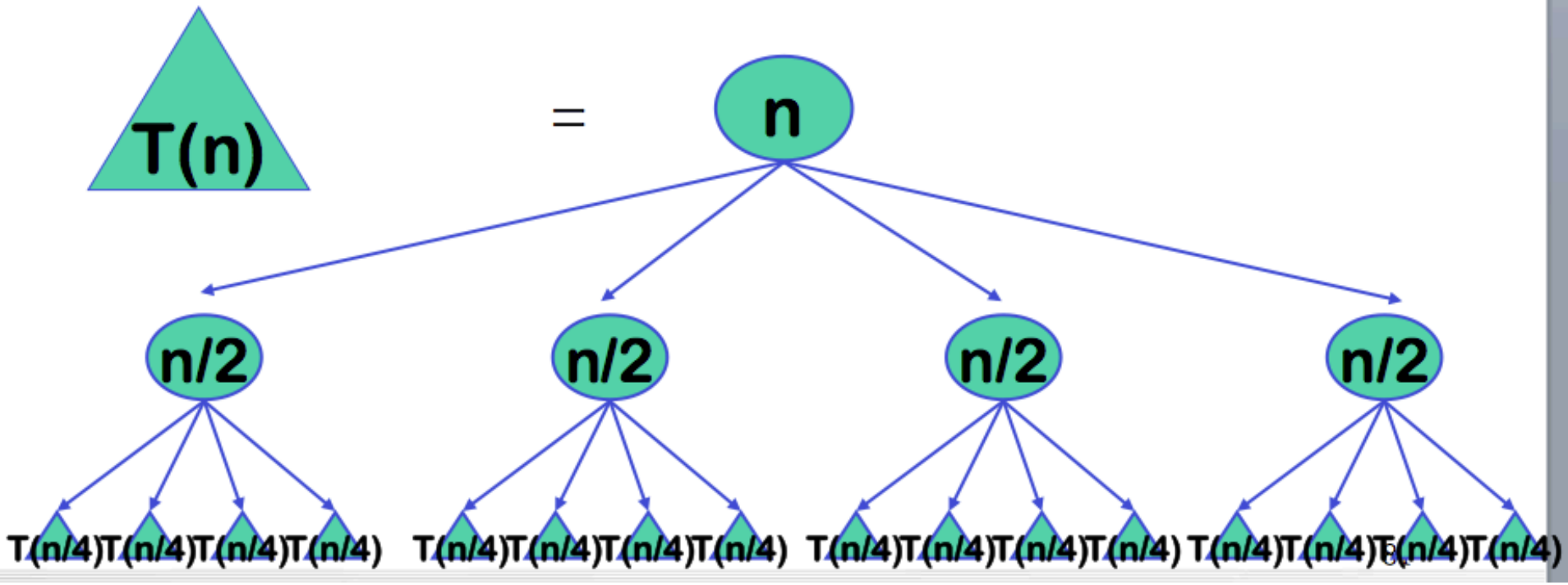


算法总体思想

- 和分治法的区别
  - 主要用于优化问题（求最优解）
  - 子问题并不独立，即子问题是可能重复的
    - 重复的子问题，不需要重复计算

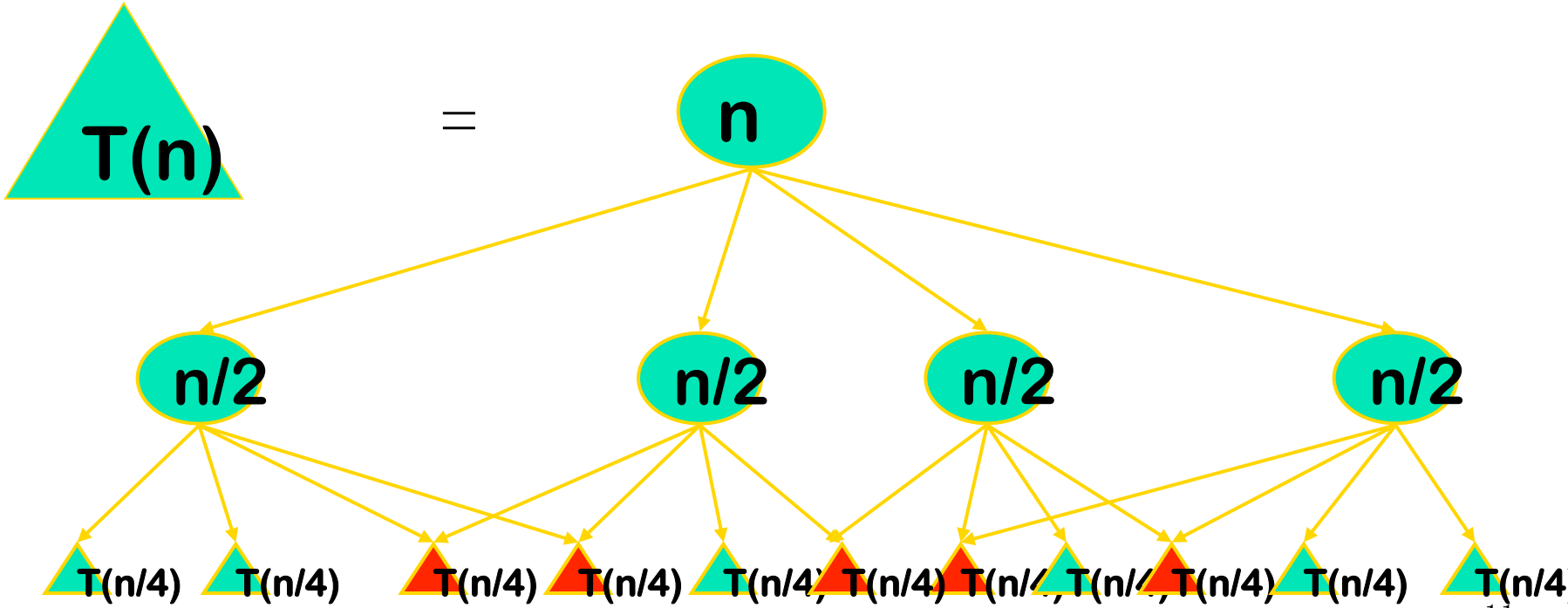
算法总体思想

- 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。



动态规划的基本思想

- 动态规划的实质是分治和消除冗余，是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解以避免计算重复的子问题，来解决最优化问题的算法策略。
- 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。

完全加括号的矩阵连乘积

- 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 $A=(BC)$

- 设有四个矩阵 $A, B,, C, D$：

- 总共有五中完全加括号的方式

$$(A((BC)D)) \quad (A(B(CD))) \quad ((AB)(CD))$$
$$(((AB)C)D) \quad ((A(BC))D)$$

Example

$A = A_1 \quad A_2 \quad A_3 \quad A_4$

$10 \times 20 \quad 20 \times 50 \quad 50 \times 1 \quad 1 \times 100$

- Order 1

$$A_1 \times (A_2 \times (A_3 \times A_4))$$

$$\text{Cost}(A_3 \times A_4) = 50 \times 1 \times 100$$

$$\text{Cost}(A_2 \times (A_3 \times A_4)) = 20 \times 50 \times 100$$

$$\text{Cost}(A_1 \times (A_2 \times (A_3 \times A_4))) = 10 \times 20 \times 100$$

Total Cost = 125000

- Order 2

$$(A_1 \times (A_2 \times A_3)) \times A_4$$

$$\text{Cost}(A_2 \times A_3) = 20 \times 50 \times 1$$

$$\text{Cost}(A_1 \times (A_2 \times A_3)) = 10 \times 20 \times 1$$

$$\text{Cost}((A_1 \times (A_2 \times A_3)) \times A_4) = 10 \times 1 \times 100$$

Total Cost = 2200

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（不是说要把矩阵连乘求出来）。

◆**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 $A_iA_{i+1}...A_j$ 简记为A[i:j]，这里i≤j

矩阵A_i 的维度为 $p_{i-1}\times p_i$；矩阵A_{i+1} 的维度为 $p_i\times p_{i+1}$

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 $(A_iA_{i+1}...A_k)(A_{k+1}A_{k+2}...A_j)$，这两个子矩阵乘也必须是最优的

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

- 计算A[i:j]，1≤i≤j≤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]=Ai，因此，m[i,i]=0，i=1,2,…,n
- 当i<j时，设k为最优断开点

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

这里 A_i 的维数为 $p_{i-1}\times p_i$

- $(A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6)$

$m[1,3]$ $\qquad$ $m[4,6]$

$(A_1 \quad A_2 \quad A_3)$ $\quad$ $(A_4 \quad A_5 \quad A_6)$

$p_0\times p_3$ $\qquad$ $p_3\times p_6$

$$m[1,3]+m[4,6]+p_0p_3p_6$$

建立递归关系

- 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

计算最优值

- 如果通过简单的递归来求解此问题，其复杂度为

$$\min_{i \le k < j} \{ m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \}$$

$$T(n) \ge 1 + \sum_{1 \le k < n} (T(k) + T(n-k) + 1) = \Omega(2^n), \text{ for } n > 1$$

计算最优值

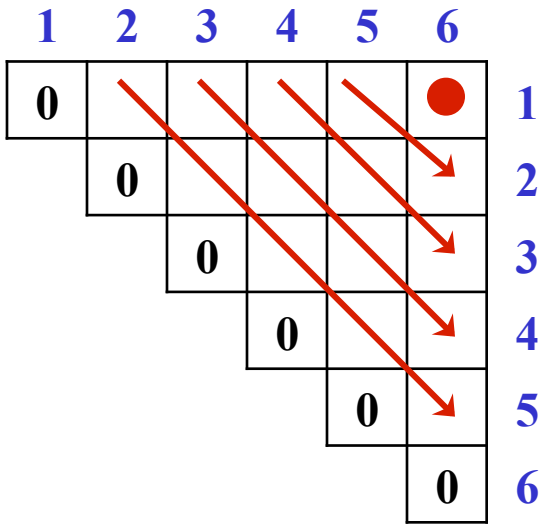
- 对于1≤i≤j≤n不同的有序对(i,j)对应于不同的子问题。因此，不同子问题的个数最多只有

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

- 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

计算最优值

- 计算所有的 ***m[i,j]***:
  --Start by setting ***m[i,i]***=0 for ***i*** = 1,…,n.
  --Then compute ***m***[1,2], ***m***[2,3],…,***m***[*n-1,n*].
  --Then ***m***[1,3], ***m***[2,4],…,***m***[*n-2,n*],…
  --… so on till we can compute ***m***[1,*n*].



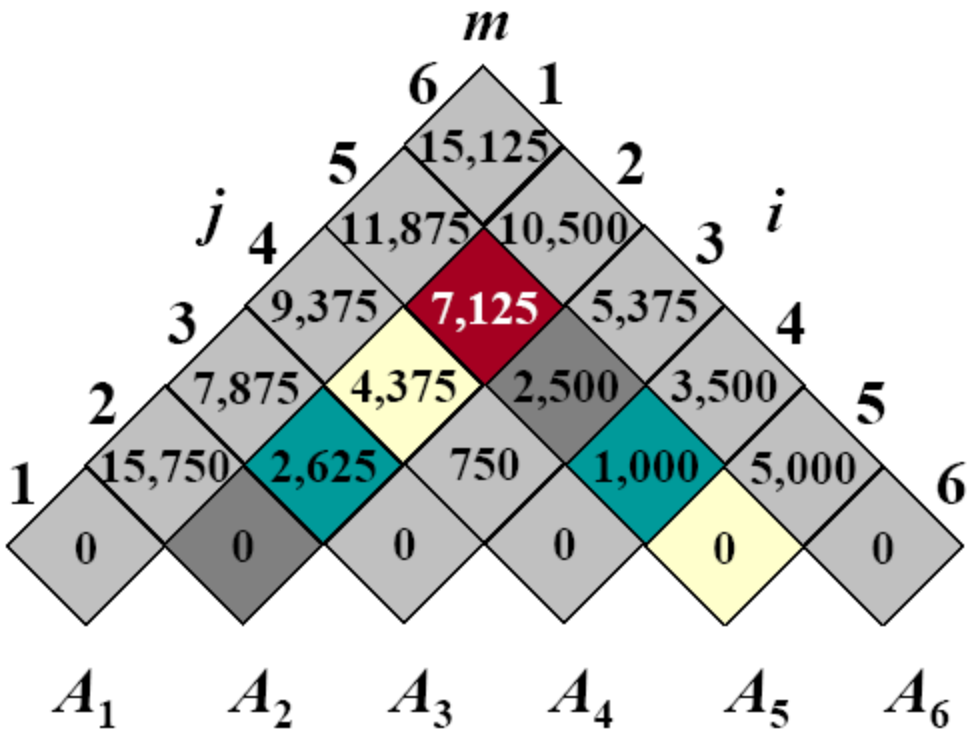
- The input a sequence $p=<p_0,p_1,\ldots,p_n>$, we use an auxiliary table *s[1..n,1..n]* that records which index of k achieved the optimal cost in computing *m[i,j]*.

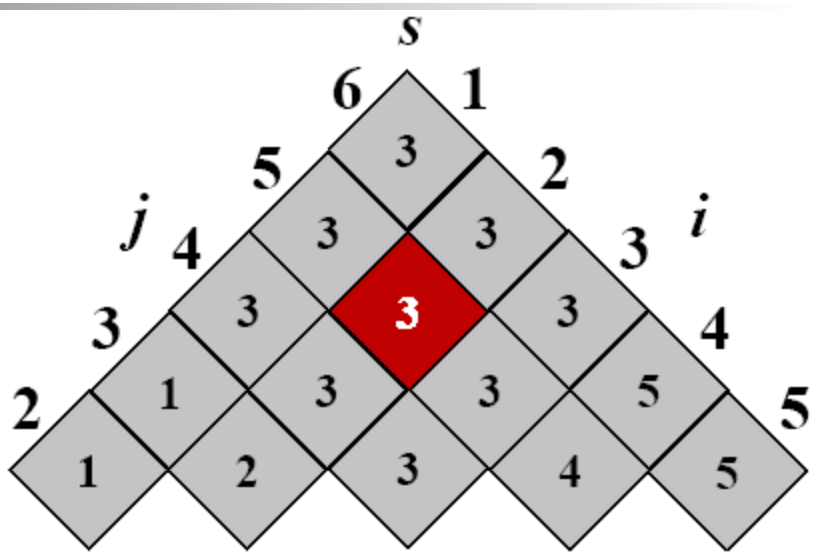
```
public static void matrixChain(int [] p, int [][] m, int [][] s)
  {
    int n=p.length-1;
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
      for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = 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;}
          }
        }
  }
```

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

$$m[2][5] = \min\begin{cases} m[2][2] + m[3][5] + p_1 p_2 p_5 = 0 + 2500 + 35 \times 15 \times 20 = 13000 \\ m[2][3] + m[4][5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \times 5 \times 20 = 7125 \\ m[2][4] + m[5][5] + p_1 p_4 p_5 = 4375 + 0 + 35 \times 10 \times 20 = 11375 \end{cases}$$

Example： The optimal solution is $((A_1(A_2A_3))((A_4A_5)A_6)$





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

```
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;}
          }
        }
  }
```

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$O(n^2)$。

构造最优解

表 $s[1..n-1, 2..n]$ 记录了构造所需的信息。每个表项 $s[i, j]$ 记录了一个 k 值，指出 $A_iA_{i+1}\cdots A_j$ 的最优括号化方案的分在 A_k 和 A_{k+1} 之间。因此，我们知道 $A_{1..n}$ 的最优计算方案中最后一次矩阵乘法运算 $A_{1..s[1,n]}A_{s[1,n]+1..n}$。我们可以用相同的方法递归地求出更早的矩阵乘法的具体计算过程 $s[1, s[1, n]]$ 指出了计算 $A_{1..s[1,n]}$ 时应进行的最后一次矩阵乘法运算；$s[s[1, n]+1, n]$ 计算 $A_{s[1,n]+1..n}$ 时应进行的最后一次矩阵乘法运算。

PRINT-OPTIMAL-PARENS

```
PRINT-OPTIMAL-PARENS(s,i,j)
1   if i=j
2      then print "A"_i
3      else print "("
4           PRINT-OPTIMAL-PARENS(s,i,s[i,j])
5           PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
6           print ")"
```

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。

三、自顶向下（备忘录方法）

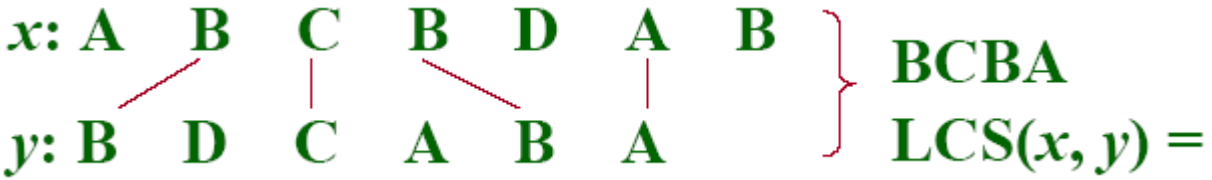
备忘录方法的控制结构与直接递归方法的控制结构相同，区别在于备忘录方法为每个解过的子问题建立了备忘录以备需要时查看，避免了相同子问题的重复求解。

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;
  }
```

最长公共子序列

- 若给定序列X={x_1,x_2,…,x_m}，则另一序列Z={z_1,z_2,…,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,…,i_k}使得对于所有j=1,2,…,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}。
- 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**。
- 给定2个序列X={x_1, x_2, …, x_m}和Y={y_1, y_2, …, y_n}，找出X和Y的最长公共子序列。



最长公共子序列问题

- 穷举法(Brute-Force)：
  - 找出X字符串所有可能的子序列(2^n)；
  - 对于X的每一个子序列， 判断其是否是Y的一个子序列,需要的时间为$\Theta(m)$;
  - 求max;总的时间为Θ ($m\ 2^n$).

最长公共子序列的结构

前缀定义：

前缀的严谨定义如下：给定一个序列 $X=\langle x_1, x_2, \cdots, x_m\rangle$，对 $i=0, 1, \cdots, m$，定义 X 的第 i 前缀为 $X_i=\langle x_1, x_2, \cdots, x_i\rangle$。例如，若 $X=\langle A, B, C, B, D, A, B\rangle$，则 $X_4=\langle A, B, C, B\rangle$，$X_0$ 为空串。

c[i, j]定义：

$c[i, j]$表示 X_i 和 Y_j 的 LCS 的长度。如果 $i=0$ 或 $j=0$，即一个序列长度为 0，那么 LCS 的长度为 0。

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**。

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]记录序列和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：

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

自底向上计算c[i][j]

- The sequences are X=<A,B,C,B,D,A,B> and Y=<B,D,C,A,B,A>

i		j	0	1	2	3	4	5	6
		y_i		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

计算最优值

COMPUTING the LENGTH of LCS

```
LCS-LENGTH(X,Y)
1  m ← length[X]
2  n ← length[Y]
3  for i ← 1 to m
4      do c[i,0] ← 0
5  for j ← 0 to n
6      do c[0,j] ← 0
7  for i ← 1 to m
8      do for j ← 1 to n
9           do if x[i] = y[j]
10                then c[i,j] ← c[i-1,j-1]+1
11                     b[i,j] ←"↖"
12                else if c[i-1,j] ≥ c[i,j-1]
13                        then c[i,j] ← c[i-1,j]
14                             b[i,j] ← "↑"
15                        else c[i,j] ← c[i,j-1]
16                             b[i,j] ← "←"
17 return c and b
```

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。

构造最长子序列

CONSTRUCTING an LCS

```
PRINT-LCS(b,X,i,j)
1 if i = 0 or j = 0
2    then return
3 if b[i,j] = "↖"
4    then PRINT-LCS(b,X,i-1,j-1)
5      print x[i]
6 else if b[i,j] = "↑"
7     then PRINT-LCS(b,X,i-1,j)
8 else PRINT-LCS(b,X,i,j-1)
```

DP方法求最大子数组问题

$$b[j]=\max_{1\le i\le j}\left\{\sum_{k=i}^{j}a[k]\right\},1\le j\le n$$

记 ，则所求的最大子段和为：

$$\max_{1\le i\le j\le n}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}\ \max_{1\le i\le j}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}b[j]$$

由b[j]的定义知，当b[j-1]>0时，b[j]=b[j-1]+a[j]，否则b[j]=a[j]。由此可得b[j]的动态规划递推式如下：

$$b(j)=\begin{cases} b(j-1)+a(j) & \text{当}b(j-1)>=0 \\ a(j) & \text{当}b(j-1)<0 \end{cases}$$

DP方法求最大子数组问题

例如，假如有一序列：-2，11，-4，13，-5，-2
则

a(1) = -2, a(2) = 11, a(3) = -4, a(4) = 13, a(5) = -5, a(6) = -2

b(1) = -2 ,b(2) = 11, b(3) = 7, b(4) = 20, b(5) = 15, b(6) = 13

b(1) < 0　b(2) > 0　b(3) > 0　b(4) > 0　b(5) > 0　b(6) > 0

DP方法求最大子数组问题

DP-MAX-SUM

```
int DPMaxSum(int n, int a, int &besti, int &bestj){
    int sum=a[1], b[1]=a[1];
    for(int i=2; i <= n; ++i){
        if(b[i-1]>0)
                b[i] = b[i-1]+a[i];
        else
                b[i] = a[i];
        if(b[i]>sum)
                sum = b[i];
    }
    return sum;
}
```