

第3章 动态规划

学习要点:

理解动态规划算法的概念。

掌握动态规划算法的基本要素

(1) 最优子结构性质

(2) 重叠子问题性质

掌握设计动态规划算法的步骤。

(1) 找出最优解的性质，并刻画其结构特征。

(2) 递归地定义最优值。

(3) 以自底向上的方式计算出最优值。

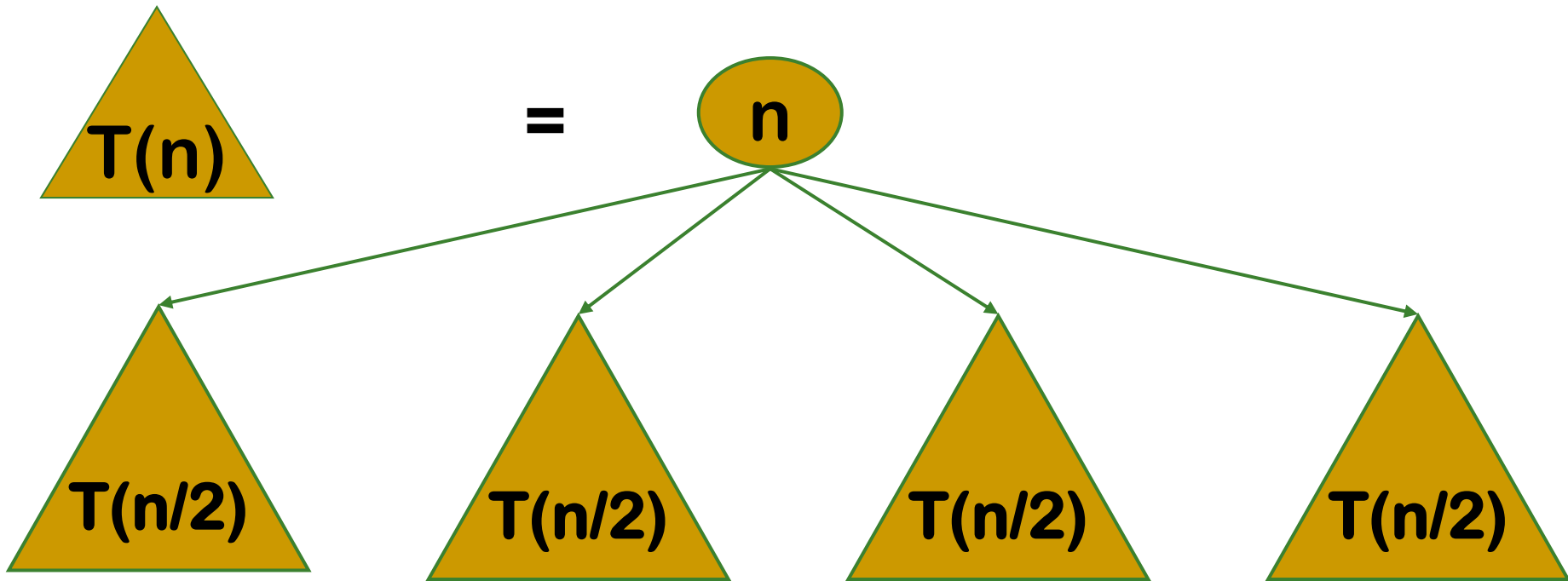
(4) 根据计算最优值时得到的信息，构造最优解。

通过应用范例学习动态规划算法设计策略。

- (1) 矩阵连乘问题;
- (2) 最长公共子序列;
- (3) 凸多边形最优三角剖分;
- (4) 多边形游戏;
- (5) 图像压缩;
- (6) 流水作业调度;
- (7) **0-1**背包问题;
- (8) 最优二叉搜索树。

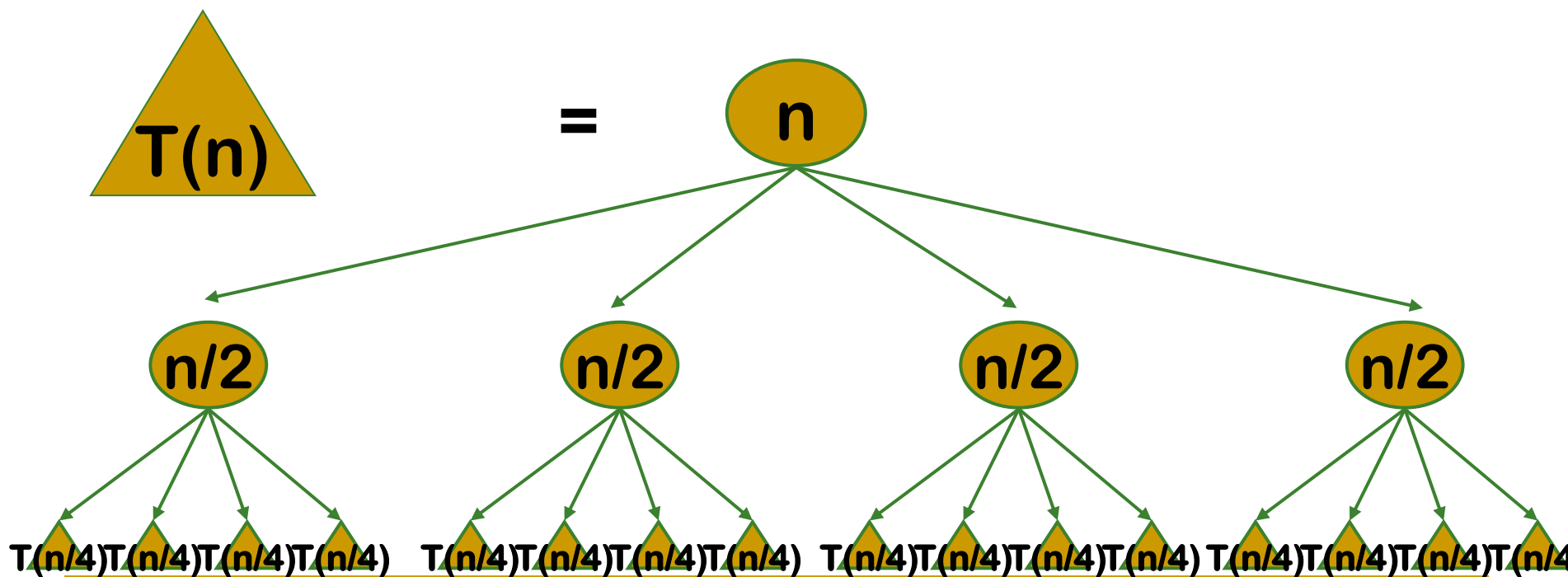
算法总体思想

- 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



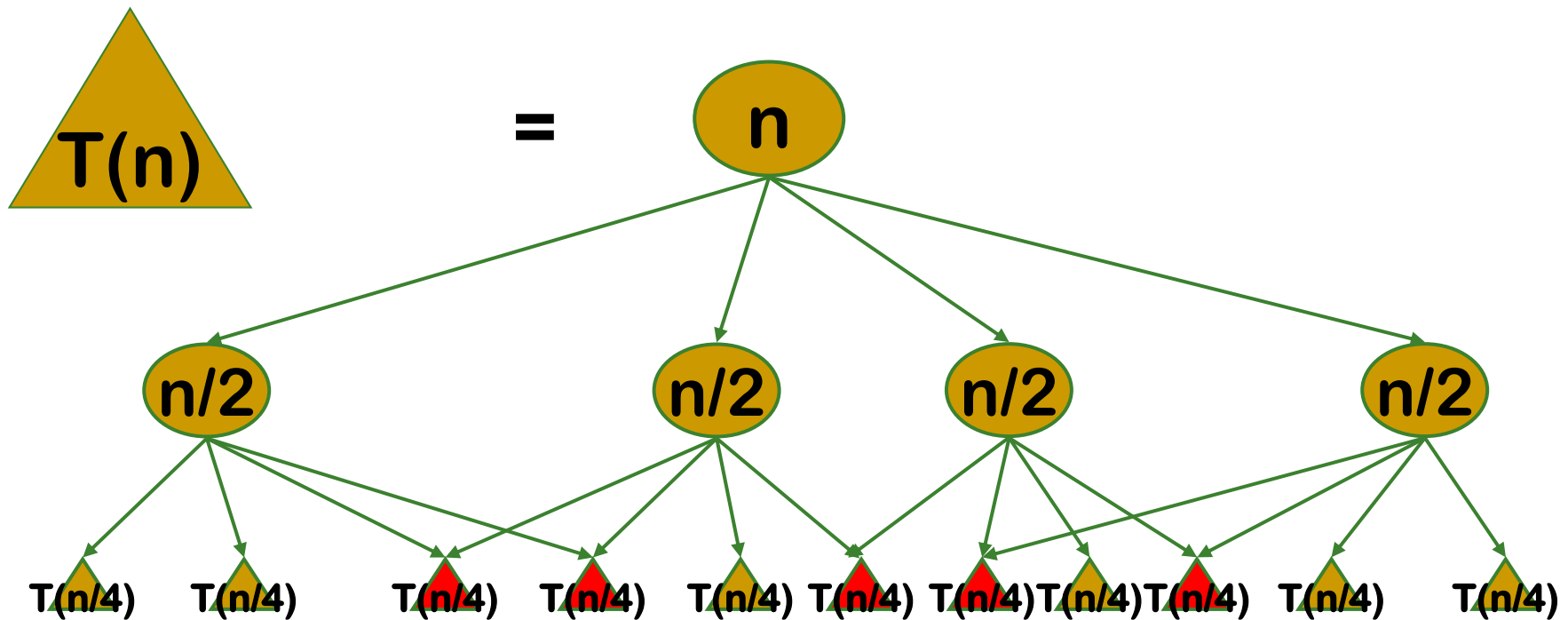
算法总体思想

- 但是在动态规划问题中，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。如果使用分治法求解，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。



动态规划基本步骤

- ① 找出最优解的性质，并刻画其结构特征。
- ② 递归地定义最优值。
- ③ 计算出最优值，通常采用自底向上的方式。
- ④ 根据计算最优值时得到的信息，构造最优解。

3.0 简单示例——斐波那契数列

$\text{Fibonacci}(n) = 1; n = 0$

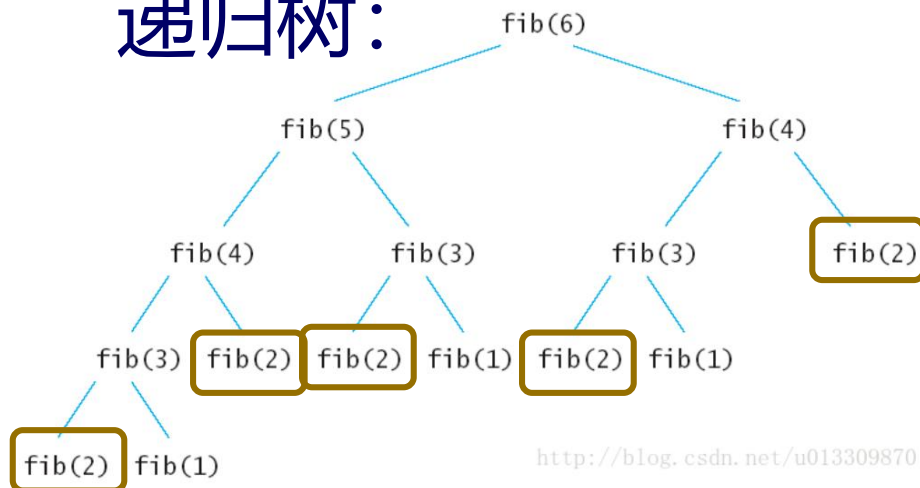
$\text{Fibonacci}(n) = 1; n = 1$

$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$

■ 自顶向下的递归版本：

```
public int fib(int n) {  
    if(n<=0) return 0;  
    if(n==1) return 1;  
    return fib(n-1)+fib(n-2);  
} //测试fib(6)
```

递归树：



上面的递归树中的每一个子节点都会执行一次，很多重复的节点被执行，fib(2)被重复执行了5次。由于调用每一个函数的时候都要保留上下文，所以空间上开销也不小。这么多的子节点被重复执行，如果在执行的时候把执行过的子节点保存起来，后面要用到的时候直接查表调用的话可以节约大量的时间。

■ 自顶向下的备忘录方法

```
public static int Fibonacci(int n){
    if(n<=0)        return n;
    int []Memo=new int[n+1];
    for(int i=0;i<=n;i++)        Memo[i]=-1; //memo数组赋初值-1
    return fib(n, Memo);
}

public static int fib(int n,int []Memo)  {
    if(Memo[n]!=-1)        return Memo[n]; //不等于-1表示该值已计算过
    if(n<=2)        Memo[n]=1;
    else        Memo[n]=fib( n-1,Memo)+fib(n-2,Memo);
    return Memo[n];
}
```

创建了一个 $n+1$ 大小的数组来保存求出的斐波拉契数列中的每一个值，在递归的时候如果发现前面 $\text{fib}(n)$ 的值计算出来了就不再计算，如果未计算出来，则计算出来后保存在Memo数组中，下次在调用 $\text{fib}(n)$ 的时候就不会重新递归了。

■ 自底向上的方法

```
public static int fib(int n){  
    if(n<=0)        return n;  
    int []Memo=new int[n+1];  
    Memo[0]=0;  
    Memo[1]=1;  
    for(int i=2;i<=n;i++) {  
        Memo[i]=Memo[i-1]+Memo[i-2];  
    }  
    return Memo[n];  
}
```

先计算子问题，再由子问题计算父问题。

■ 自底向上的方法——改进

参与循环的只有 i , $i-1$, $i-2$ 三项, 因此该方法的空间可以进一步的压缩:

```
public static int fib(int n) {  
    if(n<=1)        return n;  
    int Memo_i_2=0;  
    int Memo_i_1=1;  
    int Memo_i=1;  
    for(int i=2;i<=n;i++){  
        Memo_i=Memo_i_2+Memo_i_1;  
        Memo_i_2=Memo_i_1;  
        Memo_i_1=Memo_i;  
    }  
    return Memo_i;  
}
```

一般来说由于备忘录方式的动态规划方法使用了递归, 递归的时候会产生额外的开销, 使用自底向上的动态规划方法要比备忘录方法好。

3.1 矩阵连乘问题

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

$$A_1 A_2 \dots A_n$$

- 由于矩阵乘法满足结合律, 所以计算矩阵的连乘可以有許多不同的计算次序。这种计算次序可以用给矩阵加括号的方式来确定。

3.1 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 $A = (BC)$

$$B = 10 \times 40 \quad C = 40 \times 30$$

- ◆ 则 $A = (BC) = 10 \times 30$ ，使用到的乘法次数为 $10 \times 40 \times 30$ 次
- ◆ 共12000次

3.1 完全加括号的矩阵连乘积

- ◆ 设有四个矩阵 A, B, C, D ，它们的维数分别是：
 $A = 50 \times 10$ $B = 10 \times 40$ $C = 40 \times 30$ $D = 30 \times 5$
- ◆ 总共有五种完全加括号的方式

$$\begin{array}{lll}(A((BC)D)) & (A(B(CD))) & ((AB)(CD)) \\ (((AB)C)D) & ((A(BC))D) & \end{array}$$

所需数乘次数分别为16000, 10500, 36000, 87500, 34500次

3.1 矩阵连乘问题

给定 n 个矩阵 $\{A_1, A_2, \dots, A_n\}$ ，其中 A_i 与 A_{i+1} 是可乘的， $i=1, 2, \dots, n-1$ 。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，即计算量最少。

◆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。

穷举法

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)种。

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：
(A₁...A_k)(A_{k+1}...A_n)可以得到关于P(n)的递推式如下：

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

- 因此，**计算次序的数量**与n呈指数关系，通过穷举所有可能的方案来寻找最优方案，是一个糟糕的策略
- 注意区分这里的计算次序数量与上文的数乘次数

3.1 矩阵连乘问题

◆穷举法

◆动态规划

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

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

计算量（数乘次数）： $A[i:k]$ 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

$$\blacksquare (A_1 A_2 \dots A_k) (A_{k+1} \dots A_n)$$

- 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。
- 反证法：如果 $A[i:k]$ 和 $A[k+1:j]$ 的计算次序不是最优的，则原矩阵连乘的计算次序也不可能是最优的。
- 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。
 - 所谓的最优子结构，是指一个问题的解，可以由他的子问题的解来确定，而要想达到当前状态的最优解，则子问题的解也应该是最优的，即全局最优解包含局部最优解

建立递归关系

- 设计算 $A[i:j]$, $1 \leq i \leq j \leq n$, 所需要的最少数乘次数 $m[i,j]$, 则原问题的最优值为 $m[1,n]$ 。用表 $s[i,j]$ 记录 $m[i,j]$ 对应的分割点 k
- 假设 A_i 的维数为 $p_{i-1} \times p_i$
- 当 $i=j$ 时, $A[i:j]=A_i$, 不需要数乘, 因此, $m[i,i]=0$, $i=1,2,\dots,n$
- 当 $i < j$ 时,

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

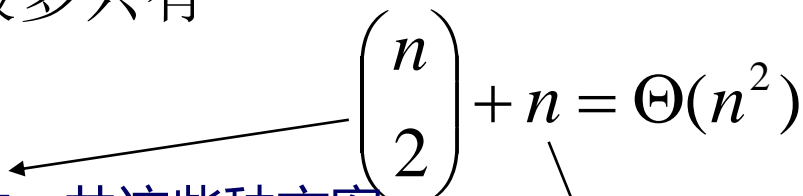
■ 新构成的两个子矩阵的大小分别为 $p_{i-1} \times p_k$ 和 $p_k \times p_j$, 这两个矩阵相乘, 需要的数乘次数为 $p_{i-1} p_k p_j$

- 可以递归地定义 $m[i,j]$ 为:

$$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

计算最优值

- 对于 $1 \leq i \leq j \leq n$ 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 $[1, n]$ 内任取两个不相同的数，共这些种方案

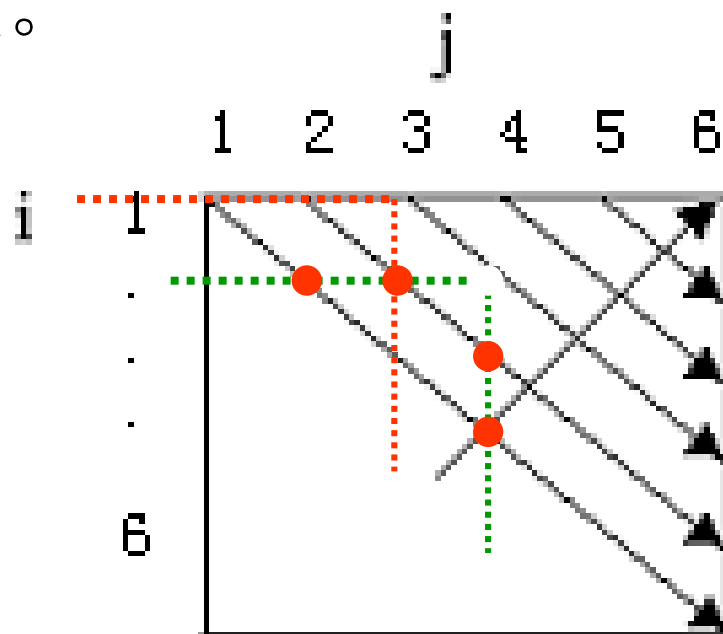
- 在 $[1, n]$ 内任取两个相同的数，共 n 种方案

- 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，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

求解过程

■ 为避免重复计算，自底向上求解

在求解长度为 r 的矩阵链前，先把所有长度 $r-1$ 的矩阵链都求完。以此类推。



用动态规划法求最优解

```
void MatrixChain(int *p, int n, int **m, int **s)
```

```
{
```

```
    for i=1 to n
```

```
        m[i][i]=0
```

```
        for r = 2 to n
```

```
            for i=1 to n-r+1
```

```
                j=i+r-1;
```

```
                m[i][j]=∞
```

```
                for k=i to j-1
```

```
                    q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
```

```
                    if (q < m[i][j])
```

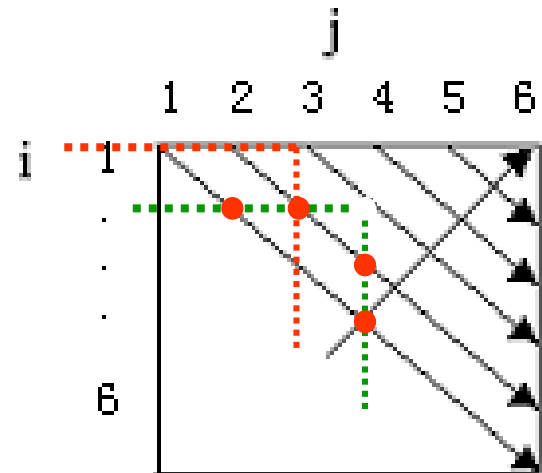
```
                        m[i][j] = q;
```

```
                        s[i][j] = k;
```

■先对 $m[i][i]$ 置零

■在这个循环里, 第一次循环, 对 $i=1\dots n-1$ 计算 $m[i][i+1]$; 第二次循环, 对 $i=1\dots n-2$ 计算 $m[i][i+2]$; 即每次循环里, 计算长度为 r 的矩阵链的最小计算代价

k 依次取值, 寻找使 $m[i][j]$ 最小的切割方案



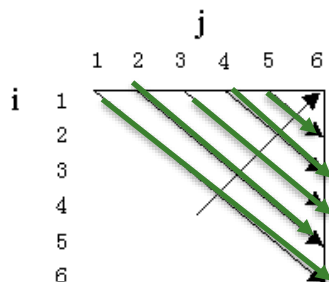
用动态规划法求最优解

■ 例：6个矩阵，其大小分别为：

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

■ 在算 $m[2][5]$ 时，其过程为：

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



(a) 计算次序

	j	1	2	3	4	5	6
i	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	1	2	3	4	5	6
i	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) $s[i][j]$

■ 体现了“自底向上”

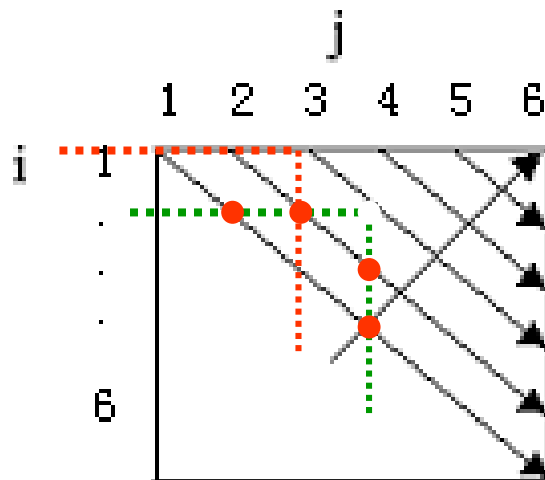
算法复杂度分析：

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 $O(n^2)$ 。

计算过程演示

■ $r=2$:

- $m[1][2]=P_0*P_1*P_2=30*35*15=15750$;
- $m[2][3]=P_1*P_2*P_3=35*15*5=2625$;
- $m[3][4]=P_2*P_3*P_4=15*5*10=750$;
- $m[4][5]=P_3*P_4*P_5=5*10*20=1000$;
- $m[5][6]=P_4*P_5*P_6=10*20*25=5000$;



■ $r=3$:

$$m[1][3] = \min \left\{ \begin{array}{l} m[1][2] + m[3][3] + P_0 P_2 P_3 \\ m[1][1] + m[2][3] + P_0 P_1 P_3 \end{array} \right\} = \min \left\{ \begin{array}{l} 18000 \\ 7875 \end{array} \right\} = 7875$$

■ $s[1][3]=1$;

$$m[2][4] = \min \left\{ \begin{array}{l} m[2][3] + m[4][4] + P_1 P_3 P_4 \\ m[2][2] + m[3][4] + P_1 P_2 P_4 \end{array} \right\} = \min \left\{ \begin{array}{l} 4375 \\ 6000 \end{array} \right\} = 4375$$

■ $s[2][4]=3$;

$$m[3][5] = \min \left\{ \begin{array}{l} m[3][4] + m[5][5] + P_2 P_4 P_5 \\ \underline{m[3][3] + m[4][5] + P_2 P_3 P_5} \end{array} \right. = \min \left\{ \begin{array}{l} 3750 \\ \underline{2500} \end{array} \right. = 2500$$

■ $s[3][5]=3;$

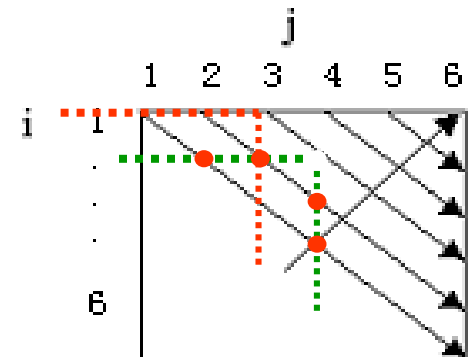
$$m[4][6] = \min \left\{ \begin{array}{l} m[4][5] + m[6][6] + P_3 P_5 P_6 \\ \underline{m[4][4] + m[5][6] + P_3 P_4 P_6} \end{array} \right. = \min \left\{ \begin{array}{l} 3500 \\ \underline{6250} \end{array} \right. = 3500$$

■ $s[4][6]=5;$

■ $r=4:$

$$m[1][4] = \min \left\{ \begin{array}{l} m[1][1] + m[2][4] + P_0 P_1 P_4 \\ m[1][2] + m[3][4] + P_0 P_2 P_4 \\ \underline{m[1][3] + m[4][4] + P_0 P_3 P_4} \end{array} \right. = \min \left\{ \begin{array}{l} 14875 \\ 21000 \\ \underline{9375} \end{array} \right. = 9375$$

■ $s[1][4]=3;$



$$m[2][5] = \min \begin{cases} m[2][2] + m[3][5] + P_1 P_2 P_5 \\ m[2][3] + m[4][5] + P_1 P_3 P_5 \\ m[2][4] + m[5][5] + P_1 P_4 P_5 \end{cases} = \min \begin{cases} 13000 \\ \underline{7125} \\ 11375 \end{cases} = 7125$$

■ $s[2][5]=3;$

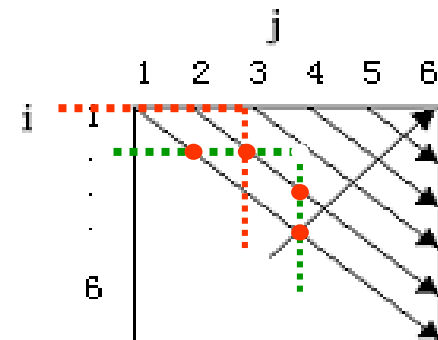
$$m[3][6] = \min \begin{cases} m[3][3] + m[4][6] + P_2 P_3 P_6 \\ m[3][4] + m[5][6] + P_2 P_4 P_6 \\ m[3][5] + m[6][6] + P_2 P_5 P_6 \end{cases} = \min \begin{cases} \underline{5375} \\ 9500 \\ 10000 \end{cases} = 5375$$

■ $s[3][6]=3;$

■ $r=5:$

$$m[1][5] = \min \begin{cases} m[1][1] + m[2][5] + P_0 P_1 P_5 = 28125 \\ m[1][2] + m[3][5] + P_0 P_2 P_5 = 27250 \\ m[1][3] + m[4][5] + P_0 P_3 P_5 = \underline{11875} \\ m[1][4] + m[5][5] + P_0 P_4 P_5 = 15375 \end{cases} = 11875$$

■ $s[1][5]=3;$



$$m[2][6] = \min \begin{cases} m[2][2] + m[3][6] + P_1 P_2 P_6 = 18500 \\ m[2][3] + m[4][6] + P_1 P_3 P_6 = \underline{10500} \\ m[2][4] + m[5][6] + P_1 P_4 P_6 = 18125 \\ m[2][5] + m[6][6] + P_1 P_5 P_6 = 24625 \end{cases} = 10500$$

■ $s[2][6]=3$;

■ $r=6$:

$$m[1][6] = \min \begin{cases} m[1][1] + m[2][6] + P_0 P_1 P_6 = 36750 \\ m[1][2] + m[3][6] + P_0 P_2 P_6 = 32375 \\ m[1][3] + m[4][6] + P_0 P_3 P_6 = \underline{15125} \\ m[1][4] + m[5][6] + P_0 P_4 P_6 = 21875 \\ m[1][5] + m[6][6] + P_0 P_5 P_6 = 26875 \end{cases} = 15125$$

■ $s[1][6]=3$;

■ 从而可知最优乘积次序为: $[A_1(A_2A_3)][(A_4A_5)A_6]$

■ $s[1][3]=1$

■ $s[4][6]=5$

计算过程演示

- 验证数乘次数如下:
- A_2A_3 数乘 $P_1*P_2*P_3=2625$ 次,矩阵 (A_2A_3) 的维数: P_1*P_3 ;
- $A_1(A_2A_3)$ 数乘 $P_0*P_1*P_3=5250$ 次,矩阵 $A_1(A_2A_3)$ 的维数: P_0*P_3 ;
- A_4A_5 数乘 $P_3*P_4*P_5=1000$ 次,矩阵 A_4A_5 的维数: P_3*P_5 ;
- $(A_4A_5)A_6$ 数乘 $P_3*P_5*P_6=2500$ 次,矩阵 $(A_4A_5)A_6$ 的维数: P_3*P_6 ;
- $(A_1(A_2A_3))((A_4A_5)A_6)$ 数乘 $P_0*P_3*P_6=3750$ 次,其维数: P_0*P_6
- 故总的数乘次数 $=2625+5250+1000+2500+3750=15125$.

3.2 动态规划算法的基本要素

一、最优子结构

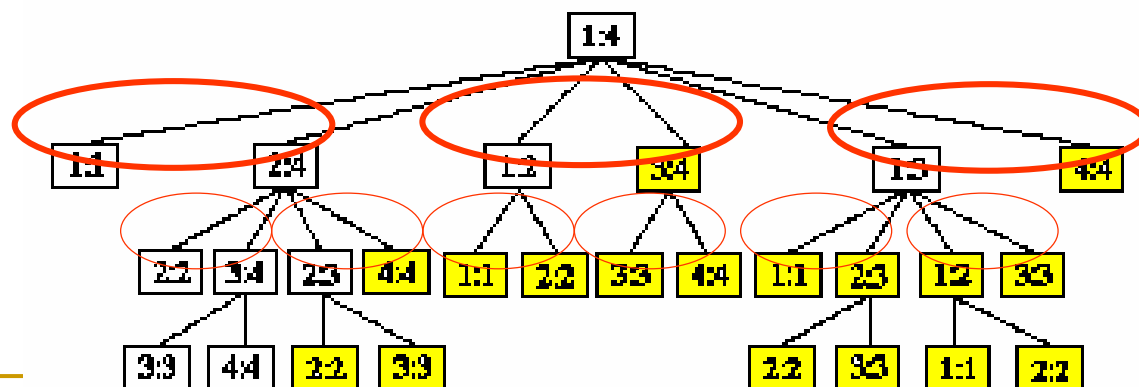
- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为**最优子结构性质**。
- 在分析问题的最优子结构性质时，所用的方法具有普遍性：首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。
- 利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的前提。

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

动态规划算法的基本要素

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。



动态规划算法的基本要素

三、除了自底向上方法，也可以用带有备忘录的自顶向下方法

- 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。

```
int LookupChain(int i, int j)
{
    if (m[i][j] > 0) return m[i][j]; //子问题已解
    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; //记录最优分解位置
    for (int k = i+1; k < j; k++) { //遍历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; //记录最优分解位置
        }
    }
    m[i][j] = u; return u;
}
```

■ 赋m[i][j]初值为-1，则若m[i][j]>0，说明该子问题已解