# 动态规划-1

夏艳

2018年春

# 内容提要--第3章 动态规划

- 动态规划的设计思想及基本要素
- 动态规划算法的典型应用：

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

# 回顾：分治法

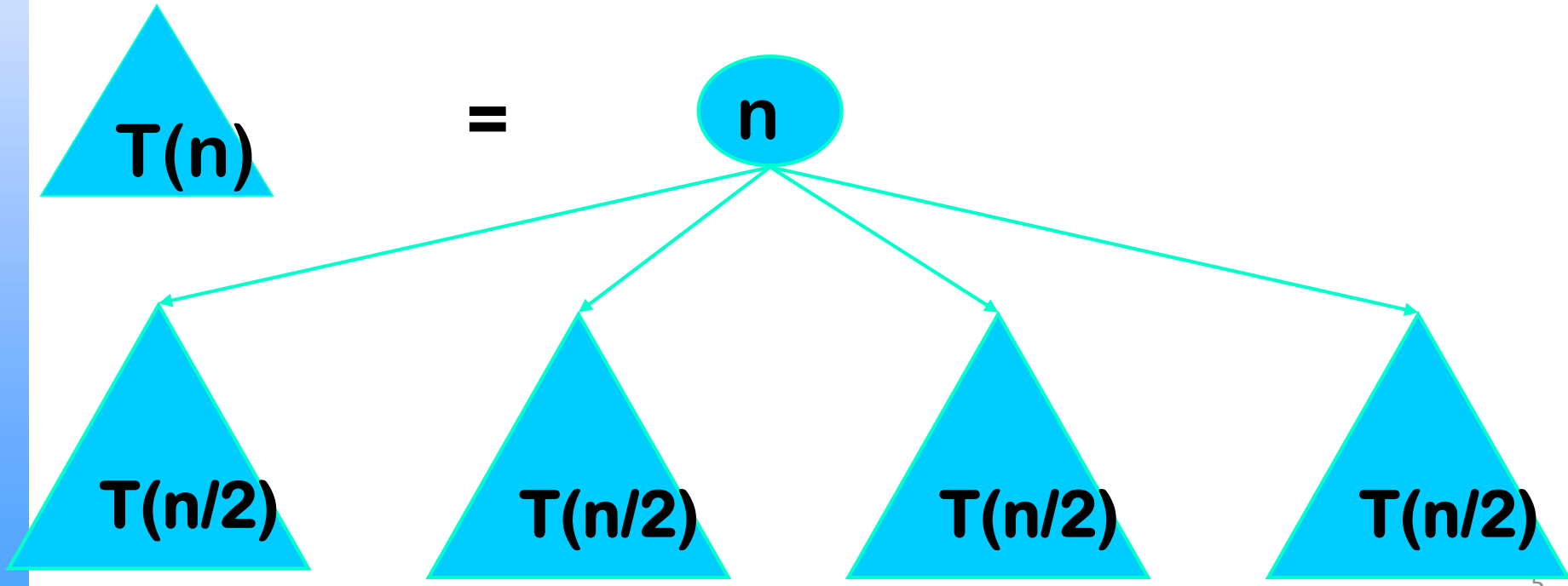
分治法所能解决的问题一般具有以下几个特征：

- 1, 该问题的规模缩小到一定的程度就可以容易地解决；
- 2, 该问题可以分解为若干个规模较小的相同问题.
- 3, 利用该问题分解出的子问题的解可以合并为该问题的解；
- 4, 分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

如果只具备了前两条特征，而不具备后二条特征，则可以考虑本章**动态规划**或下章**贪心算法**

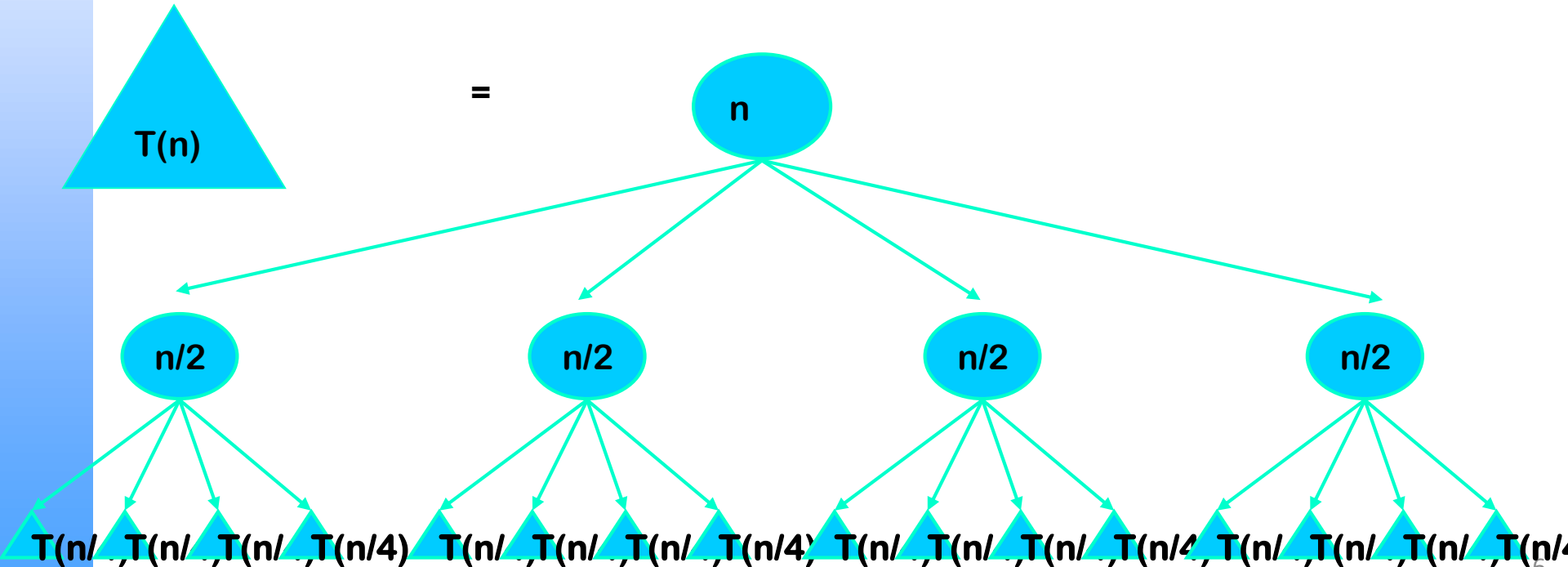
# 算法总体思想

- 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



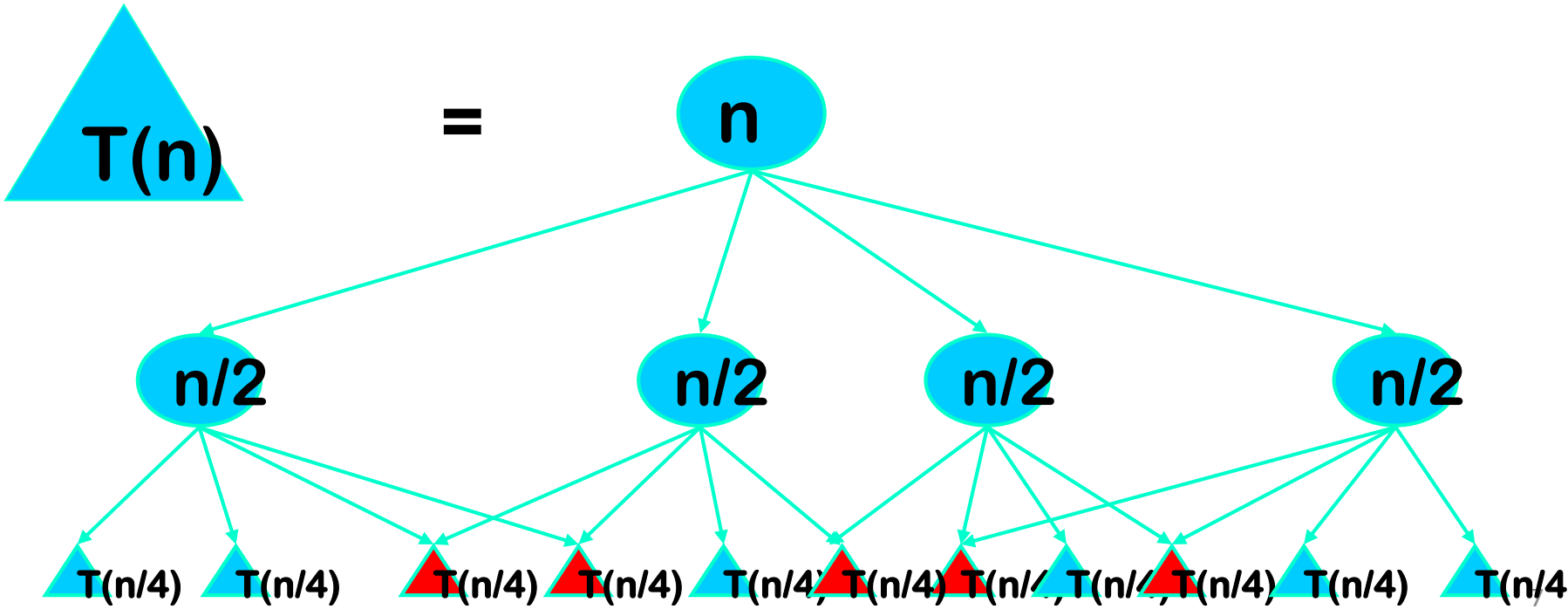
# 算法总体思想

- 分治法要求子问题互相独立。子问题数目太多，最后T(n)可能指数级。有些子问题被重复计算。



# 算法总体思想

- 动态规划保存已解决的子问题的答案，需要时引用，避免重复计算，从而T(n)为多项式时间。



# 动态规划基本步骤

- 找出最优解的性质，并刻划其结构特征。 即判断整体最优是否有局部最优。
- 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：
  - 单个矩阵是完全加括号的；
  - 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，它们的维数分别是：

A=50×10，B=10×40，C=40×30，D=30×5

- 总共有五中完全加括号的方式

(A((BC)D))　　(A(B(CD)))　　((AB)(CD))　(((AB)C)D)　((A(BC))D)

- 10500，36000，87500，34500

# 矩阵连乘问题

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

$$A_1A_2\ldots A_n$$

- 因满足结合律，故有多种计算次序。用加括号确定每种计算次序。
- 若次序完全确定则称该连乘积完全加括号，则依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积。

# 穷举搜索法

- 问题描述：如何确定计算次序，使得乘次数最少。
- 穷举法：列举出所有计算次序的乘次数，找出最少的。
- 穷举法复杂度分析：
  - 对于n个矩阵的连乘积，设其不同的计算次序为*P(n)*。
  - 每种加括号方式可分解为两个子矩阵的加括号问题：$(A_1 \ldots A_k)(A_{k+1} \ldots A_n)$，故*P(n)*的递推式如下：

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

$$\Rightarrow \quad P(n)=\frac{1}{n+1}\binom{2n}{n}=\Omega\left(4^n/n^{3/2}\right)$$

  - 也就是说，*P(n)*是随*n*的增长成指数增长的。至少有

# 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，$i=1,2,\ldots,n-1$。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。
- **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**
- **动态规划基本思路：**
  - 将矩阵连乘积$A_iA_{i+1}...A_j$简记为$A[i:j]$，这里$i \le j$。
  - 将计算次序在$A_k$和$A_{k+1}$之间断开，$i \le k < j$，则完全加括号方式为$(A_iA_{i+1}... A_k)(A_{k+1}A_{k+2}... A_j)$。
  - **计算量：*A*[*i*:*k*]**的计算量+***A*[*k*+1:*j*]**的计算量+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

- 设计算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$A[i:j]=A_i$，不乘，$m[i,i]=0, i=1,2,\ldots,n$。
- 当$i<j$时，$m[i,j]=\min(m[i,k]+m[k+1,j]+p_{i-1}p_kp_j)$，$k=i,i+1,\ldots,j-1$，其中$A_i$的维$p_{i-1}*p_i$
- 递归式

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

$k$ 的位置只有 $j-i$ 种可能

# 计算最优值

- $m[i,j]=\min(m[i,k]+m[k+1,j]+p_{i-1}p_kp_j)$, $1\le i\le j\le n$
- 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 $\binom{n}{2}+n=\Theta(n^2)$
- 递归计算时，**许多子问题被重复计算多次**。
- 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(根据i,j的值直接获取)，从而避免大量重复计算，最终得到多项式时间的算法

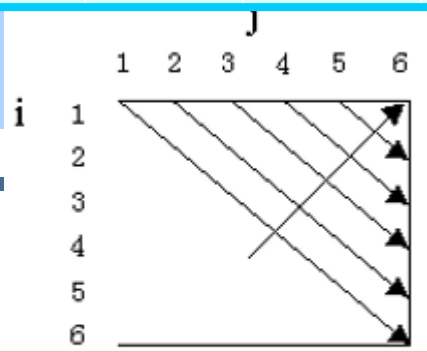
# 计算最优值

$$m[i,j]=\min(m[i,k]+m[k+1,j]+p_{i-1}p_kp_j),\quad 1\le i\le j\le n$$

```
void MatrixChain(int *p，int n，int **m，int **s){
 for(int i=1;i<=n;i++){for(int j=1;j<=n;j++) {m[i][j]=0;}}
   for (int r=2; r<=n; r++)                    //两两相乘 ~n个相乘
      for (int i=1; i<=n-r+1; i++) {       //起点i从1起,最后余r
         int j=i+r-1;                              //终点j从i起共r个
          m[i][j]=m[i][i]+m[i+1][j]+p[i-1]*p[i]*p[j]; //初值
         s[i][j]=i;                          //断开点的初值
         for (int k=i+1; k<j; k++) {      //断点k=i+1~j-1
              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; //断点为k
          }}
      }}//r=2时 j=i+2-1=i+1,不执行for(int k-…)为p[i-1]*p[i]*p[j]
```

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

# 计算最优值




```
void MatrixChain(int *p，int n，int **m，int **s){
 for(int i=1;i<=n;i++){for(int j=1;j<=n;j++)
       {m[i][j]=0;}}
   for (int r=2; r<=n; r++)
      for (int i=1; i<=n-r+1; i++) //起点
         int j=i+r-1; //终点j从i起共r个
          m[i][j]=m[i][i]+m[i+1][j]+p[i-1
       初值
         s[i][j]=i;          //断开点的初值
         for (int k=i+1; k<j; k++) {   //断点k=i+1~j-1
    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; //断点为k
          }}
     }}//r=2时 j=i+2-1=i+1,不执行for(int k-…)为p[i-
      1]*p[i]*p[j]
```

**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$O(n^2)$。

(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 |

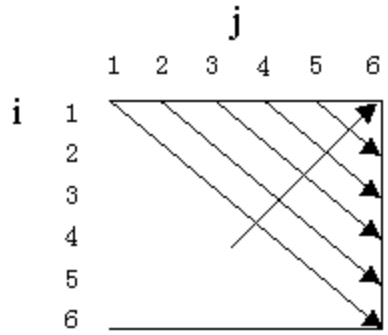
(c) s[i][j]

# 示例

$$m[i,j]=\min(m[i,k]+m[k+1,j]+p_{i-1}p_kp_j),\quad 1\le i\le j\le n$$

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

| p0 | p1 | p2 | p3 | p4 | p5 | p6 |
|---|---|---|---|---|---|---|
| [30 | 35 | 15 | 5 | 10 | 20 | 25 ] |




(a) 计算次序

j

| 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]

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]

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

# 4.构造最优解

| | 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 |

- 最少乘法次数为15125。
- 计算次序
  - S[1][n]为得到n矩阵相乘的断开点为3，全式为**(A[1:3])(A[4:6])**。
  - **s[1][3]**值为1，则A[1:3] 为**(A[1:1])(A[2:3])**，
  - s[2][3]为2，则A[2:3]为(A[2:2])(A[3:3]。
  - **s[4][6]**为5，则A[4:6]为(A[4:5])(A[6:6])
  - s[4][5]为4，则(A[4:4])(A[5:5])
  - **(A[1:3])(A[4:6])= (A[1](A[2]A[3])) ((A[4]A[5])A[6])**

# 构造最优解

**s[1][6]=3故**
**A[1:6]=(A[1:3])(A[4:6])**
**查1:s[1][6],s[1][6]+1:6**

```
void traceBack(int i,int j, int **s){
  if  (i==j) return ;
  traceBack(i,s[i][j],s);  //递归调用
  traceBack(s[i][j]+1,j,s);
  cout<<"Multiply A"<<i<<","<<s[i][j]<<" and A"
<<(s[i][j]+1)<<j<<endl;
}
```

# 3.2 动态规划算法的基本要素

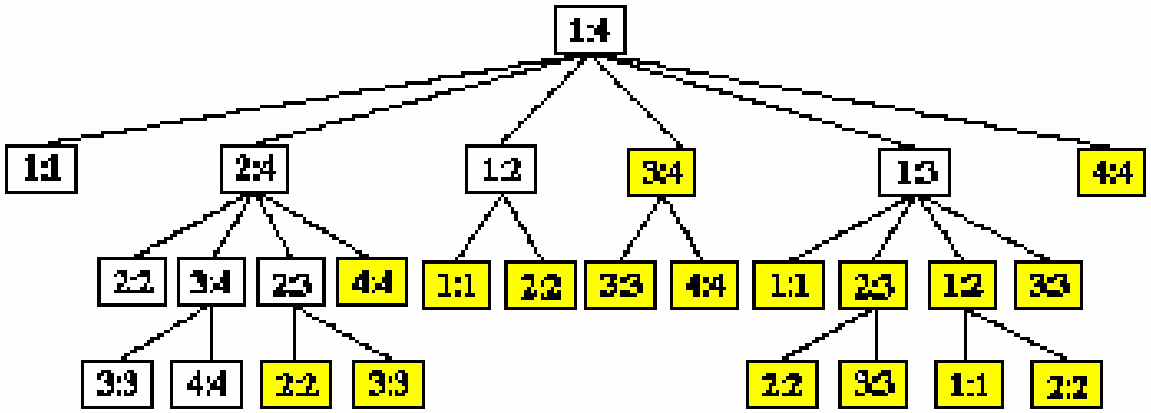
- 基本要素：最优子结构、子问题重叠
  - 最优子结构
  - 重叠子问题
- 备忘录方法
  - 动态规划算法与备忘录方法的适用条件

# 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**。
  - 若整体最优时局部最优则具有最优子结构性质。
- 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。
- 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）

# 二、重叠子问题

- 递归算法求解问题时，每次子问题并不总是新问题，有些子问题被反复计算多次。该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，获得较高的解题效率。




P54的递归算法求A[1:4]的过程，重复计算很多

# 三、备忘录方法

- 备忘录方法的控制结构与直接递归方法相同。
- 保存子问题的解，需要时查看，避免了重复求解。

```
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;
   return LookupChain(1,n);}
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;}
```

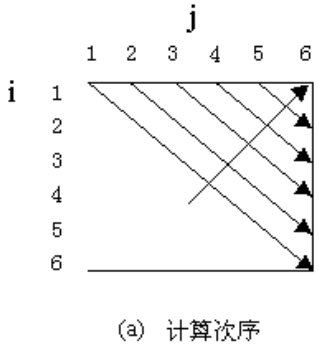
# 关于动态规划算法和备忘录方法的适用条件

- 自顶向下的备忘录方法，时间复杂度为$O(n^3)$
- 自底向上的动态规划，时间复杂度为$O(n^3)$
- 这两个算法都利用了子问题重叠即重复性质。总共有$\theta(n^2)$个不同的子问题。
- 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：当一个问题的所有子问题都至少要解一次时，用动态规划算法好。
- 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。

# 课堂练习

- A1=60×20，A2=20×50，A3=50×25，A4=25×15，请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。

| $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ |
|---|---|---|---|---|---|
| 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]

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

# 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,…,k有：$z_j=x_{i_j}$。
- 例:序列X={A，B，C，B，D，A，B}的一个子序列是Z={B，C，D，B}，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**。
- **问题：**给定2个序列X={$x_1,x_2,\ldots,x_m$}和Y={$y_1,y_2,\ldots,y_n$}，找出X和Y的最长公共子序列。

# 课堂练习

请找出下列两个序列的最长公共子序列

X={A，B，C，B，D，A，B}

Y={B，D，C，A，B，A}

# 最长公共子序列的结构

- 设$X=\{x_1,x_2,\ldots,x_m\}$和$Y=\{y_1,y_2,\ldots,y_n\}$
- 最长公共子序列$Z=\{z_1,z_2,\ldots,z_k\}$，则
  - 若$x_m=y_n$(最后者同则必属公串)，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的最长公子串
  - 若$x_m\neq y_n$且$z_k\neq y_n$，则Z是X和$Y_{n-1}$的最长公子串

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：

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

# 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。

```
Algorithm lcsLength(x,y,b)
1: m←x.length-1;
2: n←y.length-1;
3: c[i][0]=0; c[0][i]=0;
4: for (int i = 1; i <= m; i++)
5:    for (int j = 1; j <= n; j++)
6:       if (x[i]==y[j])
7:          c[i][j]=c[i-1][j-1]+1;
8:          b[i][j]=1;
9:       else if (c[i-1][j]>=c[i][j-1])
10:         c[i][j]=c[i-1][j];
11:         b[i][j]=2;
12:      else
13:         c[i][j]=c[i][j-1];
14:         b[i][j]=3;
```

**构造最长公共子序列**

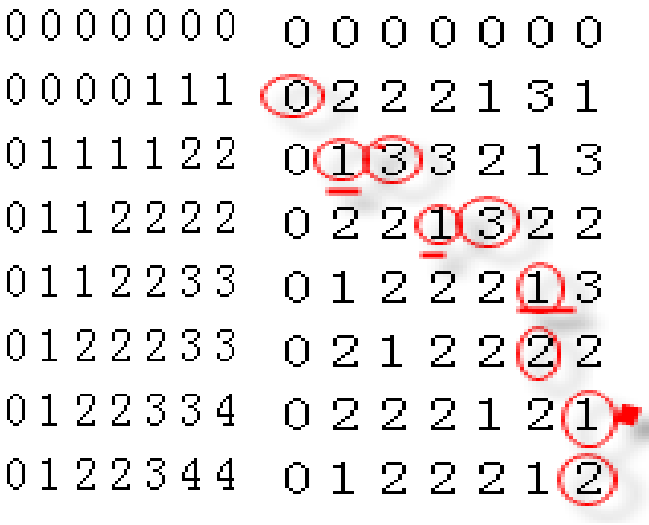
```
Algorithm lcs(int i,int j,char [] x,int [][] b)
  {
    if (i ==0 || j==0) return;
    if (b[i][j]== 1){
      lcs(i-1,j-1,x,b);
      System.out.print(x[i]);
      }
    else if (b[i][j]== 2) lcs(i-1,j,x,b);
      else lcs(i,j-1,x,b);
  }
```

# 计算最优值

- c[i][j]的下标取值范围可知，共有θ(mn)个不同的子问题，每个子问题都会算到，故动态规划算法自底向上，比备忘递归法效率更高。

```
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;//初值
    for (i = 1; i <= n; i++) c[0][i] = 0;//初值
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++) {
            if (x[i]==y[j]) {                    //(Xn,Yn)=(Xn-1,Yn-1)的公串
              +xn.
            c[i][j]=c[i-1][j-1]+1; b[i][j]=1;   }
            else if (c[i-1][j]>=c[i][j-1]) { //(Xm,Yn)=(Xm-1,Yn)
                c[i][j]=c[i-1][j]; b[i][j]=2;}
            else {                               //(Xm,Yn)=(Xm,Yn-1)
            c[i][j]=c[i][j-1]; b[i][j]=3; }
        }
}
```

# 4 构造最长公共子序列




X= ABCBDAB

Y= BDCABA

- 根据b[i][j]构造最长公子串。
- 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){   //(Xn,Yn)=(Xn-1,Yn-1)的公串+xn.
    LCS(i-1，j-1，x，b); cout<<x[i]; }
    else if (b[i][j]== 2){   //(Xm,Yn)=(Xm-1,Yn)
     LCS(i-1，j，x，b);}
    else {                    //(Xm,Yn)=(Xm,Yn-1)
    LCS(i，j-1，x，b);}
}
```

# 算法的改进

- 在算法lcsLength和lcs中，可进一步将数组b省去。
  - c[i][j]的值仅由c[i-1][j-1]，c[i-1][j]和c[i][j-1] 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))。

# 算法的改进

- 在算法lcsLength和lcs中，将数组b省去。

```
void LCSB(int i，int j，char *x，int **b)
{
    if (i ==0 || j==0) return;
    if (c[i][j]==c[i-1][j-1]+1){
      //(Xn,Yn)=(Xn-1,Yn-1) +xn.
    LCSB(i-1，j-1，x，b); cout<<x[i]; }
    else if (c[i-1][j]>c[i][j-1]){     //(Xm,Yn)=(Xm-1,Yn)
     LCSB(i-1，j，x，b);}
    else {                          //(Xm,Yn)=(Xm,Yn-1)
    LCSB(i，j-1，x，b);}
}
```

# 算法的改进

- 改lcs返回最长公子串

```
String  LCSC(int i，  int j，  char *x，  int **b)
{
    if (i ==0 || j==0) return "";
    if (b[i][j]==1){
      //(Xn,Yn)=(Xn-1,Yn-1) +xn.
    return LCSC(i-1，  j-1，  x，  b)+x[i]; }
    else if (b[i][j]==2){     //(Xm,Yn)=(Xm-1,Yn)
     return LCSC(i-1，  j，  x，  b);}
    else {                          //(Xm,Yn)=(Xm,Yn-1)
    return LCSC(i，  j-1，  x，  b);}
}
```

# 算法的改进

- 改lcs返回最长公子串

```
String  LCSD(int i，  int j，  char *x，  int **b)
{
    if (i ==0 || j==0) return "";
    if (c[i][j]==c[i-1][j-1]+1){
      //(Xn,Yn)=(Xn-1,Yn-1) +x_n.
    return LCSD(i-1，  j-1，  x，  b)+x[i]; }
    else if (c[i-1][j]>c[i][j-1]){     //(Xm,Yn)=(Xm-1,Yn)
     return LCSD(i-1，  j，  x，  b);}
    else {                          //(Xm,Yn)=(Xm,Yn-1)
    return LCSD(i，  j-1，  x，  b);}
}
```

# 3.4 最大子段和

- 问题描述：
  - 给定由n个整数(可为负)组成的序列$a_1,a_2,...,a_n$，求该序列子段(连续元素)和的最大值。当和为负值时约定为0。
  - 依此定义，所求的最优值为：
  - $$\max\left\{0, \max_{1\le i\le j\le n}\sum_{k=i}^{j} a_k\right\}$$
- 例如，当$(a_1,a_2, a_3, a_4, a_5,a_6)=(-2,11,-4,13,-5,-2)$时，最大子段和为：
  - $$\sum_{k=2}^{4} a_k = 11+(-4)+13 = 20$$

# 1. 一个简单算法

- 简单算法：连续元素和

```
int MaxSum(int n, a, &besti, &bestj)
{     //起点为i，结束点为j,一段试过去
      int sum=0;
      for(i=1;i<=n;i++)  //sum(i,j)
            for(j=i;j<=n;j++){
                  int thissum=0;
                  for(k=i;k<=j;k++)thissum+=
                     a[k];
                  if(thissum>sum){
                        sum=thissum;
                        besti=i;
                        bestj=j;
                  }}
      return sum;
}
```

- 算法有3重循环，复杂性为$O(n^3)$

由于和是逐渐累加

$$\sum_{k=i}^{j} a_k = a_j + \sum_{k=i}^{j-1} a_k$$

- 改进：

```
int MaxSum(int n, a, &besti, &bestj)
{
      int sum=0;
      for(i=1;i<=n;i++){
            int thissum=0;
            for(j=i;j<=n;j++){
                  thissum+=a[j];
                  if(thissum>sum){
                        sum=thissum;
                        besti=i;
                        bestj=j;
                  }
            }}}
```

- 改进后的算法复杂性为$O(n^2)$。

# 2. 分治方法求解

- 分治策略求解：
  - 将a[1:n]分为a[1:n/2]和a[n/2+1:n]，分别求出各段最大子段和，则a[1:n]的最大子段和分成：
    A. a[1:n]的最大子段和与a[1:n/2]的最大子段和相同；
    B. a[1:n]的最大子段和与a[n/2+1:n]的最大子段和相同；
    C. a[1:n]的最大子段和横跨两段
  - A、B这两种情形可递归求得。
  - 对于情形C，a[n/2]与a[n/2+1]在最优子序列中，在a[1:n/2]和a[n/2+1:n]中分别计算出s1和s2。则s1+s2即为情形C。

$$C.\quad \sum_{k=i}^{j} a_k, 1 \le i \le \frac{n}{2}, \frac{n}{2}+1 \le j \le n \qquad s_1 = \max_{1 \le i \le \frac{n}{2}} \sum_{k=i}^{\frac{n}{2}} a_k \qquad s_2 = \max_{\frac{n}{2}+1 \le i \le n} \sum_{k=\frac{n}{2}+1}^{j} a_k$$

- 从而设计出下面所示的分治算法。

# 2. 分治方法求解

```
int MaxSubSum(int a, int left, int right)
{ int sum=0;
  if (left==right) sum=a[left]>0?a[left]:0;
  else{int center=(left+right)/2;
    int leftsum=MaxSubSum(a,left,center);
    int rightsum=MaxSubSum(a,center+1,right);
    int s1=0;lefts=0;
    for (int i=center;i>=left; i--)
    { lefts+=a[i];
      if (lefts>s1) s1=lefts;
    }
    int s2=0;rights=0;
    for (int i=center+1;i<=right; i++)
    { rights+=a[i];
      if (rights>s2) s2=rights;
    }
    sum=s1+s2;
    if (sum<leftsum) sum=leftsum;
    if (sum<rightsum) sum=rightsum;
  }
  return sum;
}
```

复杂性分析：

$$T(n)=\begin{cases} O(1) & n \le c \\ 2T\left(\frac{n}{2}\right)+O(n) & n > c \end{cases}$$

$$T(n)=O(n\log n)$$

# 3. 动态规划方法求解

最大子段和为 $\max\limits_{1\le i\le j\le n}\sum\limits_{k=i}^{j}a_k=\max\limits_{1\le j\le n}\max\limits_{1\le i\le j}\sum\limits_{k=i}^{j}a_k=\max\limits_{1\le j\le n}b_j$

$$b_j=\max_{1\le i\le j}\sum_{k=i}^{j}a_k=\max\{\sum_{k=1}^{j}a_k,\sum_{k=2}^{j}a_k,\sum_{k=3}^{j}a_k,...,\sum_{k=j}^{j}a_k\}$$

$$=\max\{\sum_{k=1}^{j-1}a_k+aj,\sum_{k=2}^{j-1}a_k+aj,\sum_{k=3}^{j-1}a_k+aj,...,\sum_{k=j-1}^{j-1}a_k+aj,aj\}$$

$$=\max\{\sum_{k=1}^{j-1}a_k,\sum_{k=2}^{j-1}a_k,\sum_{k=3}^{j-1}a_k,...,\sum_{k=j-1}^{j-1}a_k,0\}+aj$$

$$b_{j-1}=\max_{1\le i\le j-1}\sum_{k=i}^{j-1}a_k=\max\{\sum_{k=1}^{j-1}a_k,\sum_{k=2}^{j-1}a_k,\sum_{k=3}^{j-1}a_k,...,\sum_{k=j-1}^{j-1}a_k\}$$

- 由以上$b_j$推理可知，当$b_{j-1}>0$时$b_j=b_{j-1}+a_j$，否则$b_j=a_j$。
- 动态规划递归式 $b_j=\max\{b_{j-1}+a_j，a_j\}$，$1\le j\le n$。
- 算法如下：

```
int MaxSum(int n, int
    a){
      int sum=0; b=0;
      for
         (j=1;j<=n;j++){
            if (b>0)
               b+=a[j];
               else
               b=a[j];
            if (b>sum)
               sum=b;}
      return sum;
}
```

- 计算时间为$O(n)$。

# 4. 算法的推广

- 最大矩阵和问题(*P.62*)
  - 给定一个m行n列的整数矩阵A，试求矩阵A的一个子矩阵，使其各元素之和为最大。

- 最大m子段和问题(*P.63*)
  - 给定由n个整数（可能为负整数）组成的序列a1,a2,…,an，以及一个正整数m，要求确定序列a1,a2,…,an的m个不相交子段，使这m个子段的总和为最大。

# Thank you !