第3章　动态规划

内容纲要

- 动态规划的基本步骤
- 动态规划的基本要求
- 动态规划的应用举例

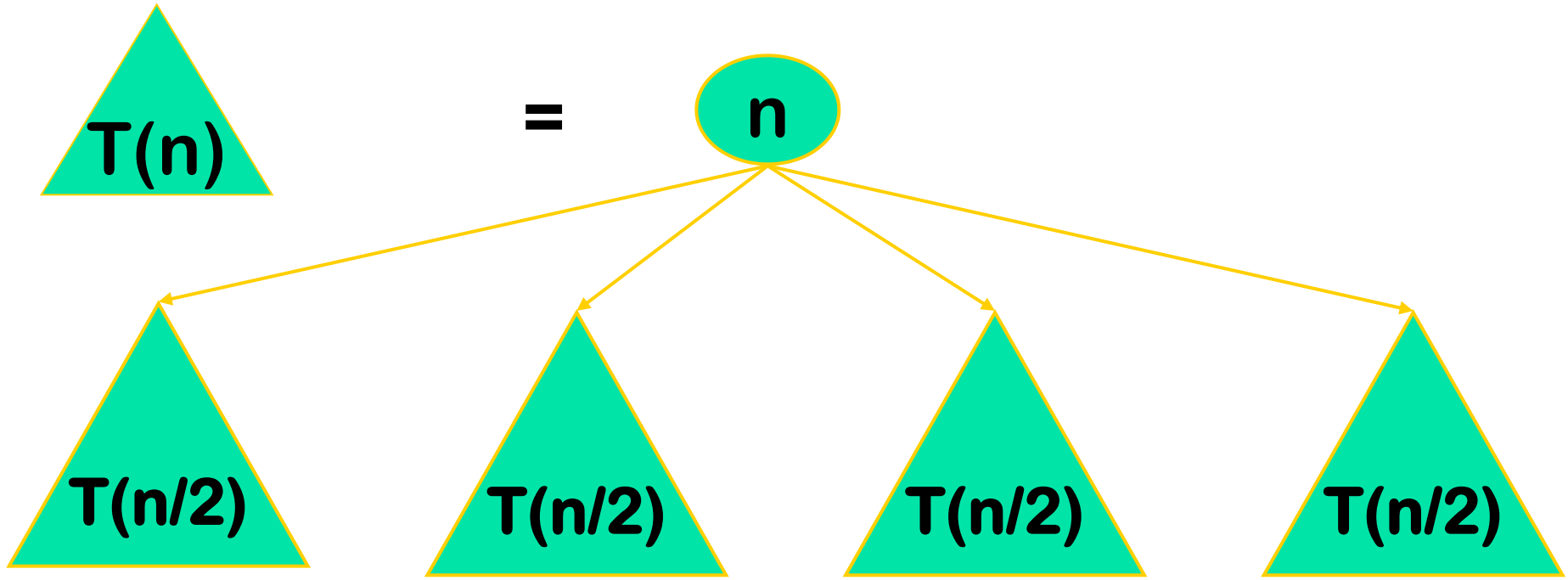
学习要点:

- 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，并刻划其结构特征。
- (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）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。

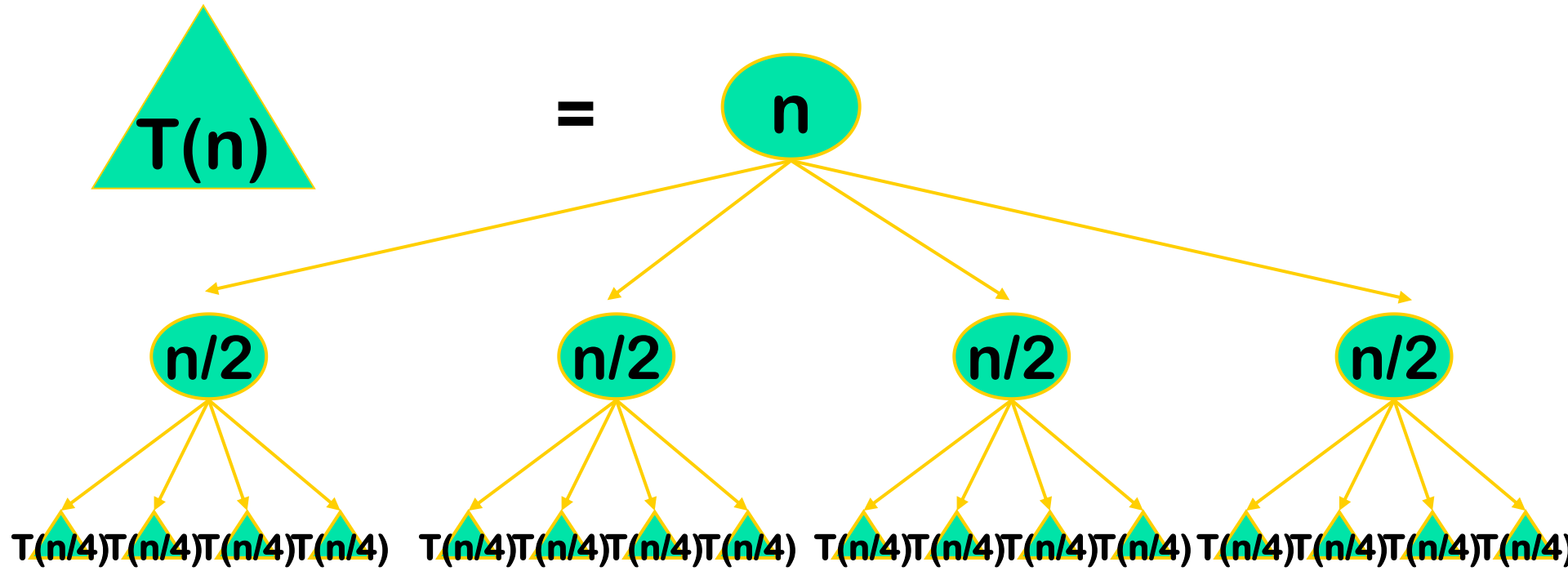
算法总体思想

- 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



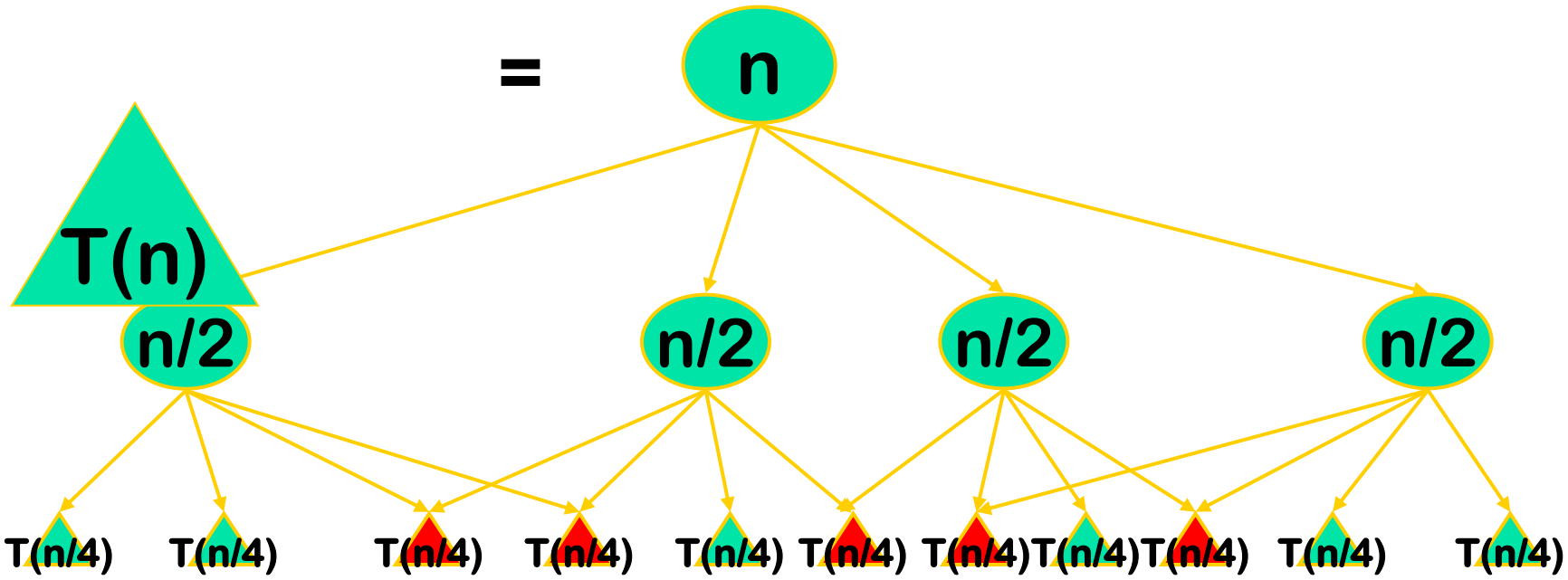
算法总体思想

- 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 $A, B, C, D,$ 它们的维数分别是：

  $A=50\times10$ $B=10\times40$ $C=40\times30$ $D=30\times5$

- 总共有五种完全加括号的方式

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

  $(((AB)C)D)$ $((A(BC))D)$

  16000, 10500, 36000, 87500, 34500

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，$i = 1, 2, ..., n-1$ 。考察这n个矩阵的连乘积

$$A_1 A_2 \ldots A_n$$

- 由于矩阵乘法满足结合律，所以计算矩阵的连乘可以有许多不同的计算次序。这种计算次序可以用加括号的方式来确定。
- 若一个矩阵连乘积的计算次序完全确定，也就是说该连乘积已完全加括号，则可以依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积

矩阵连乘问题

给定n个矩阵{$A_1,A_2,\ldots,A_n$}，其中Ai与Ai+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。

◆**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：(A1…Ak)(Ak+1…An)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

◆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

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

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

- 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]=Ai，因此，m[i,i]=0，i=1,2,…,n
- 当i<j时，

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

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：

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

k 的位置只有 $j-i$ 种可能

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

用动态规划法求最优解

```
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;}
        }
    }
```

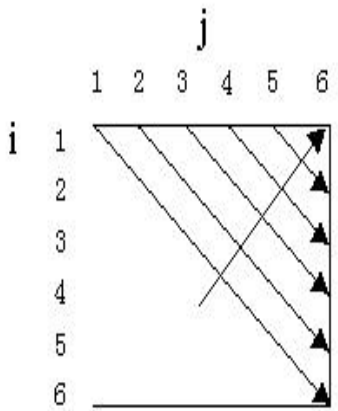
算法复杂度分析：

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$O(n^2)$。

用动态规划法求最优解:例子

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

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5 = 0+2500+35\quad 15\quad 20 = 13000 \\ m[2][3]+m[4][5]+p_1p_3p_5 = 2625+1000+35\quad 5\quad 20 = 7125 \\ m[2][4]+m[5][5]+p_1p_4p_5 = 4375+0+35\quad 10\quad 20 = 11375 \end{cases}$$



(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]

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）

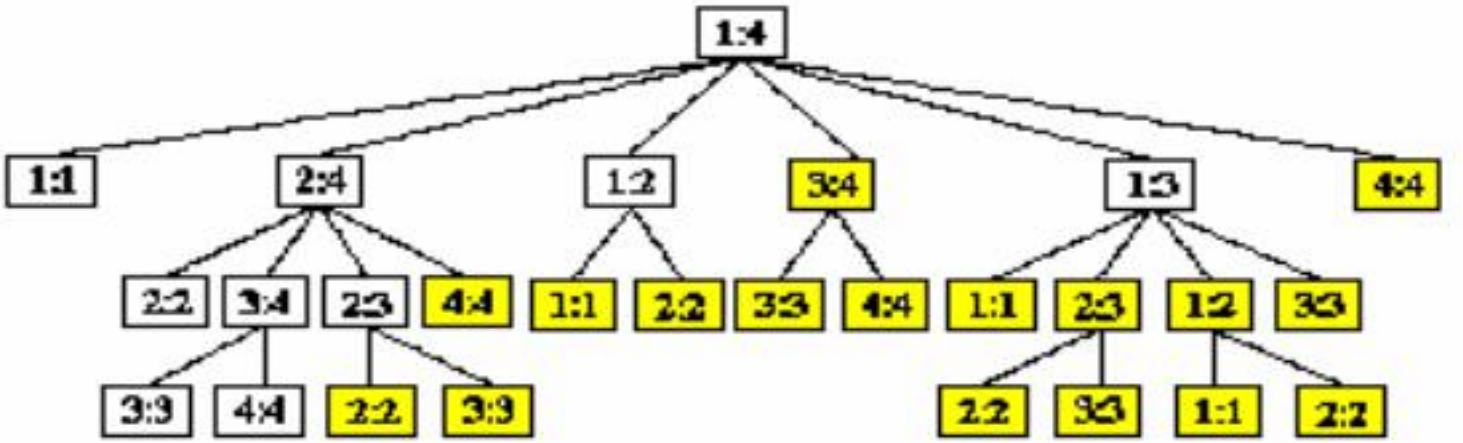
动态规划算法的基本要素

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、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。

```
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;
}
```

例2 最长公共子序列的结构

- 求序列X ={$x_1x_2 \ldots x_m$}和Y={$y_1y_2 \ldots y_n$}的最长公共的子序列。
- 如

X=abcabcdeabcdefg

Y=abdcabcdeabgcdefg

1 最长公共子序列的结构

- **穷举法求解**

  对序列X $=\{x_1x_2\ldots x_m\}$的所有子序列，检查它是否也是Y的子序列。时间复杂性O(n2^m)。

- **最优子结构性质**

  设序列$\boldsymbol{X}=\{x_1x_2\ldots x_m\}$和$\boldsymbol{Y}=\{y_1y_2\ldots y_n\}$的最长公共子序列为$\boldsymbol{Z}=\{z_1z_2\ldots z_k\}$，则有：

（1）若有$x_m=y_n$，则$z_k=x_m=y_n$，且$\boldsymbol{Z}_{k-1}$是$\boldsymbol{X}_{m-1}$和$\boldsymbol{Y}_{n-1}$的最长公共子序列。

（2）若有$x_m\neq y_n$，且$z_k\neq x_m$，且$\boldsymbol{Z}_{k-1}$是$\boldsymbol{X}_{m-1}$和$\boldsymbol{Y}_n$的最长公共子序列。

（3）若有$x_m\neq y_n$，且$z_k\neq y_n$，且$\boldsymbol{Z}_{k-1}$是$\boldsymbol{X}_m$和$\boldsymbol{Y}_{n-1}$的最长公共子序列。

其中$\boldsymbol{X}_{m-1}=\{x_1x_2\ldots x_{m-1}\}$，$\boldsymbol{Y}_{n-1}=\{y_1y_2\ldots y_{n-1}\}$和$\boldsymbol{Z}_{k-1}=\{z_1z_2\ldots z_{k-1}\}$。

2 子问题的递归结构

- **记号**

  设序列$\boldsymbol{X}=\{x_1x_2\ldots x_m\}$和$\boldsymbol{Y}=\{y_1y_2\ldots y_n\}$，用$\mathbf{c}[i][j]$表示序列$\boldsymbol{X_i}=\{x_1x_2\ldots x_i\}$和$\boldsymbol{Y_j}=\{y_1y_2\ldots y_j\}$的最长公共子序列。

- **递归描述**

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

3 计算最优值

- 直接使用递归算法实现

```
int LCS(int i, int j, char *x,char *y)  {
   if (i==0&&j==0) return 0;
   if ((i>0 && j>0)&& x[i]==y[j])
         return LCS(i-1, j-1, x,y)+1 ;
   int u= LCS(i-1, j, x,y);
   int v= LCS(i, j-1, x,y);
   if (u>v)   return   u;
   else return v;
}
```

本算法的复杂性是指数级的。

动态规划算法计算最优值

```
int LCSLength(int m, int n, char *x,char *y,int **c,int **s) {
  int i,j;
  for(i=1;i<=m;i++) c[i][0]= 0;
  for(i=1;i<=m;i++) c[0][i]= 0;
  for(i=1;i<=m;i++)
     for(j=1;i<=n;i++)  {
        if (x[i]==y[j]) {
          c[i][j]= c[i-1][j-1]+1;  s[i][j]="1";
          }
        else   if (c[i-1][ j]>=c[i][j-1]) {
                  c[i][j]= c[i-1][j];  s[i][j]="2";
                  }
        else       {
                   c[i][j]= c[i][j-1];  s[i][j]="3";
                  }
      }
}
```

5. 算法改进

- 可以省略数组s
  （课堂练习）
- 可以用两行的数组代替m×n的数组c
  （课后练习之一）

3.4 最大子段和

```
int MaxSum(int n, int *a, int *besti, int *bestj) {
    int sum=0;
   for(i=1;i<=n;i++)
        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;
}
```

最大子段和简单算法的改进

- 利用部分和

$$a_i + a_{i+1} + a_{i+2} + \ldots + a_{j-1} + a_j$$

$$= (a_i + a_{i+1} + a_{i+2} + \ldots + a_{j-1}) + a_j$$

最大子段和简单算法的改进

```
int MaxSum(int n, int *a, int *besti, int *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;
                   }
           }
        return sum;
}
```

2. 最大子段和问题的分治算法

- 将给定的整数序列对分为两段a[1：*n*/2]和a[*n*/2：n]，分别求这两段中的最大子段和，记为S [1：*n*/2] 和S[*n*/2：n]，那么， S[1：n]有三种情况：
  - S[1：n] =S [1：*n*/2]
  - S[1：n] = S[*n*/2：n]
  - S[1：n] 由a[1：*n*/2]的后部元素与a[*n*/2：n]的前部元素求和得到。

最大子段和的分治算法

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

int MaxSum(int n, int *a)  {
   return MAxSubSum(a,1,n);
}
```

分治算法的算法复杂性

- 该算法的复杂性T(n)满足典型的分治算法递归式

$$T(n)=\begin{cases}O(1) & n\le 1\\ 2T(n/2)+O(n) & n>1\end{cases}$$

- 解递归方程得

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

最大子段和的动态规划算法

- 记b[j]表示a[1],…,a[j]的下列子序列和的最大值

  a[1]+,…,　　+a[j]

  a[2]+,…,　　+a[j]

  a[3]+,…,　　+a[j]

  … …

  a[j]

- 注意b[j]有如下性质

  如果b[j-1]>0，则b[j]= b[j-1]+a[j],否则b[j]=a[j]，于是

  b[j]= max{b[j-1]+a[j],a[j]}

最大子段和的动态规划算法

- 最大子段和表示为

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

$$= \max_{1 \le j \le n} b[j]$$

- 根据b[j]的性质，易知最大子段和问题具有最优子结构性质。

最大子段和的动态规划算法

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

3.11 最优二叉搜索树

- 有序集

  $S=\{x_1,\ldots,x_n\}$且$x_1<\ldots<x_n$是n个元素的有序集。

- 二叉搜索树

  用一棵二叉树的结点来存储有序集S中的元素，它具有如下性质：

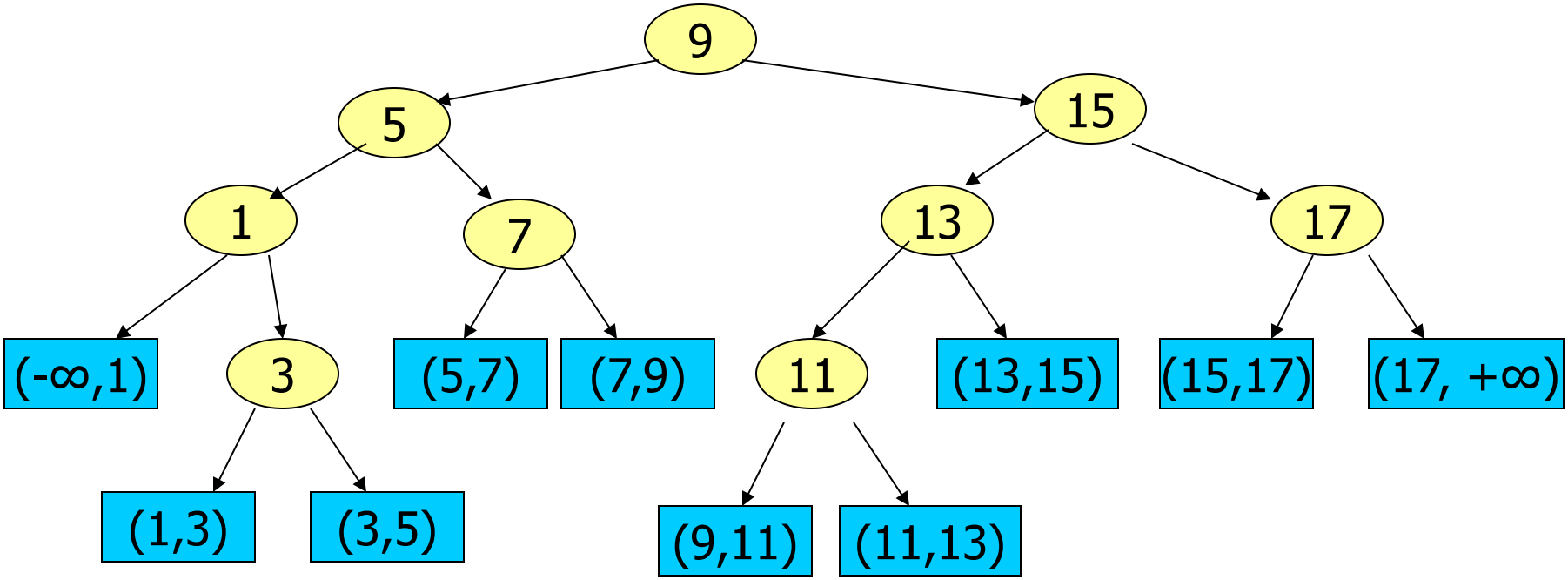
  - 每个结点存储的元素x大于其左子树中任一结点所存储的元素，且
  - 小于其右子树中任一结点所存储的元素。
  - 叶结点是形如(x_i, x_{i+1})的开区间。

- 搜索结果

  - 在二叉搜索树的内结点中的元素$x_i = x$。
  - 在二叉搜索树的叶结点中确定x在(x_i, x_{i+1})中。

二叉搜索树举例

- 例：表示S={1,3,5,7,9,11,13,15,17}的一棵二叉搜索树



若搜索x=13，则返回找到。

若搜索x=12，则返回12在(11,13)中(未找到)。

最优二叉搜索树有关概念

- 存取概率
  - “在二叉搜索树的内结点中的元素$x_i = x$”的概率为b_i，$b_i \geq 0$。
  - “在二叉搜索树的叶结点中确定x在(x_i, x_{i+1})中”的概率为a_i，$a_i \geq 0$。
  - 且有 $\sum_{i=0}^{n} a_i + \sum_{i=0}^{n} b_i = 1$
- 平均路径长

在s的二叉搜索树T中，设存储元素x_i的结点深度为c_i；叶结点(x_i, x_{i+1})的深度为d_i，则

$$p = \sum_{i=1}^{n} b_i (1 + c_i) + \sum_{j=0}^{n} a_j d_j$$

表示在T中作一次搜索所需的平均比较次数。P又称为平均路径长。

最优二叉搜索树有关概念

- 最优二叉搜索树

一般情况下， 表示有序集S=$\{x_1,\ldots, x_n\}$的不同的二叉搜索树的平均路径是不同的。

在给定有序集S=$\{x_1,\ldots, x_n\}$及其存取概率分布

$(a_0, b_1, a_1,\ldots, b_n,a_n)$,所有不同的二叉搜索树中平均路径最小者称为最优二叉搜索树。

二叉搜索树举例-续

- 例：表示S={1,3,5,7,9,11,13,15,17}的一棵二叉搜索树T中，设各元素的存取概率如下表

1	3	5	7	9	11	13	15	17
0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06

(-,1)	(1,3)	(3,5)	(5,7)	(7,9)	(9,11)	(11,13)	(13,15)	(15,17)	(17,+)
0.1	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04

$$\text{平均路径长}=0.06\times(1+2+2+3+3+3+3+4+4)+0.1\times 3+$$
$$0.04\times(4+4+3+3+4+4+3+3+3)$$
$$=3.04$$

1. 最优子结构性质

- 二叉搜索树的子树也是一棵二叉搜索树

二叉搜索树T的一棵含有结点$\{x_i,\ldots, x_j\}$及叶结点(x_{i-1}, x_i) , …, (x_j, x_{j+1})的子树可以看成有序集$\{x_i,\ldots, x_j\}$关于全集$\{x_{i-1},\ldots, x_j\}$的一棵二叉搜索树，其存取概率为条件概率

$$\bar{b}_k = b_k / w_{ij}, i \quad k \quad j, \bar{a}_h = a_h / w_{ij}, i-1 \quad h \quad j$$

其中

$$w_{ij} = a_{i-1} + b_i + \ldots + b_j + a_j$$

1. 最优子结构性质

- 设T_{ij}是有序集$\{x_i,\ldots,x_j\}$关于存取概率$\{\overline{a}_{i-1},\overline{b}_i,\ldots,\overline{b}_j,\overline{a}_j\}$的一棵最优二叉搜索树，其平均路径长为$p_{ij}$

  其根结点存储元素x_m。其左右子树为T_l和T_r的平均路长为P_l和P_r。由于左右子树为T_l和T_r中结点的深度是树T_{ij}的深度减1，于是有：

$$w_{i,j}p_{ij} = w_{i,j} + w_{i,m-1}p_l + w_{m+1,j}p_r$$

- 于是左子树为T_l是最优二叉搜索树，否则存在更另一棵二叉树T'_l，它的平均路长为P'_l更小，这样的话，用这棵子树为代替T_l得到的T_{ij}，其平均路长更小，矛盾。
- 同理，右子树为T_r也是最优二叉搜索树。

2. 递归计算最优值

- 设最优二叉搜索树$\mathbf{T}_{ij}$的平均路径长为$\boldsymbol{p}_{ij}$，则所要求的最优值为$\boldsymbol{p}_{1,n}$，根据最优子结构性质，有如下递归计算式：

$$w_{i,j}p_{ij} = w_{i,j} + \min_{i\ k\ j}\{w_{i,k-1}p_{i,k-1} + w_{k+1,j}p_{k+1,j}\}, i \le j$$

- 初始值$\boldsymbol{p}_{i,l\text{-}1}=0,\ 0\le \boldsymbol{i}\le \boldsymbol{n}$

$\boldsymbol{w}_{1,n}=1$

- 记$\boldsymbol{w}_{i,j}\ \boldsymbol{p}_{i,j}$为$\boldsymbol{m}_{i,j}$,则$\boldsymbol{m}_{1,n}=\boldsymbol{w}_{1,n}\ \boldsymbol{p}_{1,n}=\boldsymbol{p}_{1,n}$即为所要计算的最优值。计算的递归式变为

$$m(i,j) = w_{i,j} + \min_{i\ k\ j}\{m(i,k-1) + m(i,k-1)\}, i \le j$$

$$m(i,i-1) = 0, 1 \quad i \quad n$$

2. 递归计算最优值动态规划算法

```
void OBSTa(int *a,int *b,int n int **m,int **s,int **w) {
  for (int i=0;i<=n;i++) {   w[i+1][i]=0;      m[i+1][i]=0;    }
  for (int r=0;r<n;r++)
     for (int i=1;i<=n-r;i++) {
       int j=i+r;
       w[i][j]=w[i][j-1]+a[j]+b[j];     m[i][j]= m[i+1][j];     s[i][j]=i;
       for (int k=i+1;k<=j;k++) {
              int t=m[i][k-1]+m[k+1][j];
              if (t<m[i][j])  m[i][j]=t;
              s[i][j]=k;
        }
      m[i][j]+=w[i][j];
   }
}
```

3.构造最优解

```
struct TreeNode {
int data;
TreeNode * leftchild,*rightchild
}
TreeNode  *MakeOBST(int i, int j, int *a, int **s) {
    if (i==j)       return MakeNewNode(a[i]);
    else  {
        tnode=MakeNewNode(a[s[i][j]]);
        tnode.leftchild=MakeOBST(i, s[i][j]-1 , int *a, int **s);
        tnode.rightchild=MakeOBST(s[i][j]+1,j , int *a, int **s);
       return tnode;
    }
}
```

4.算法复杂性与改进

- 算法复杂性 $\mathbf{O}(n^3)$

第一个循环：$O(n)$

第二个循环：r从0到n, $O(n)$

第三个循环：i从1到$n-r$,$O(n)$

第四个循环：k从$i+1$到j, $O(n)$

因此，算法复杂为$\mathbf{O}(n^3)$

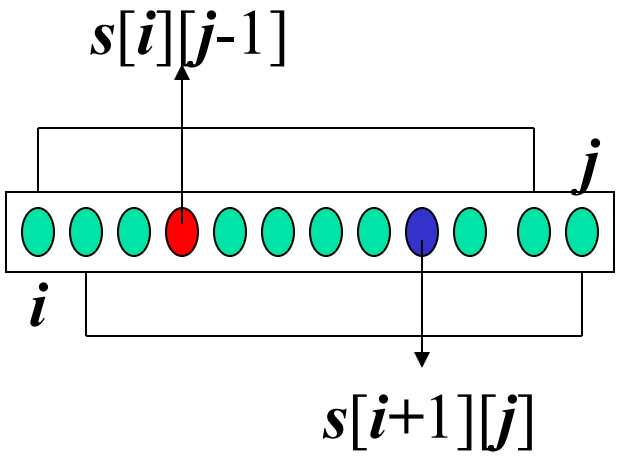
- 算法改进

$$\min_{i\ k\ j}\{m(i,k-1)+m(i,k-1)\}$$

$$= \min_{s[i][j-1]\ k\ s[i+1][j]}\{m(i,k-1)+m(i,k-1)\}$$

改进的算法

```
void OBSTb(int *a,int *b,int n int **m,int **s,int **w) {
  for (int i=0;i<=n;i++) {   w[i+1][i]=0;      m[i+1][i]=0; s[i+1][i]=0;   }
  for (int r=0;r<n;r++)
     for (int i=1;i<=n-r;i++) {
       int j=i+r;
       i1= s[i][j-1]>i? s[i][j-1]:i;
       j1= s[i+1][j]<j? s[i+1][j]:j;
       w[i][j]=w[i][j-1]+a[j]+b[j];     m[i][j]= m[i][i1-1]+ m[i1-1][j];   s[i][j]=i1;
      for (int k=i1+1;k<=j1;k++) {
           int t=m[i][k-1]+m[k+1][j];
           if (t<m[i][j])  m[i][j]=t;
           s[i][j]=k;
      }
     m[i][j]+=w[i][j];
  }
}
```



3.12 动态规划的加速原理

- 动态规划递归通式

$$m(i,j)=w(i,j)+\min_{i<k\le j}\{m(i,k-1)+m(k,j)\}\quad 1\le i<j\le n$$

- 在最优二分搜索树问题中

$$m(i,j)=w_{i,j}+\min_{i\ k\ j}\{m(i,k-1)+m(k,j)\},i\le j$$

其中 $w_{ij}=a_{i-1}+b_i+\ldots+b_j+a_j$

- 在矩阵连乘问题中

$$m[i][j]=\min_{i\le k<j}\{m[i][k]+m[k+1][j]+p_{i-1}p_kp_j\}$$

其中 $w(i,j)=p_{i-1}p_kp_j$

1. 通用动态规划算法模式

```
void DynamicProgramming(int ,int n int **m,int **s,int **w) {
  for (int i=0;i<=n;i++) {  m[i][i]=0;   s[i][i]=0;   }
  for (int r=1;r<n;r++)
    for (int i=1;i<=n-r;i++) {
      int j=i+r;
      w[i][j]=weight[i][j];    m[i][j]= m[i+1][j];    s[i][j]=i;
      for (int k=i+1;k<=j;k++) {
        int t=m[i][k]+m[k+1][j];
        if (t<m[i][j]) { m[i][j]=t; s[i][j]=k;  }
      }
    m[i][j]+=w[i][j];
    }
}
```

链长控制

最基本元素的计算

链头位置控制

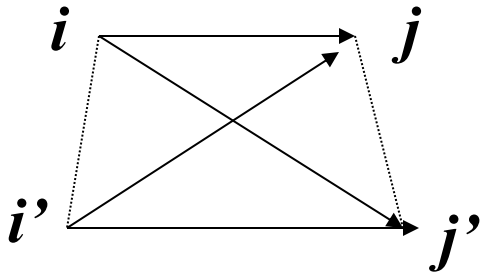
链尾位置控制

链内分裂位置控制

2. 动态规划算法的加速条件

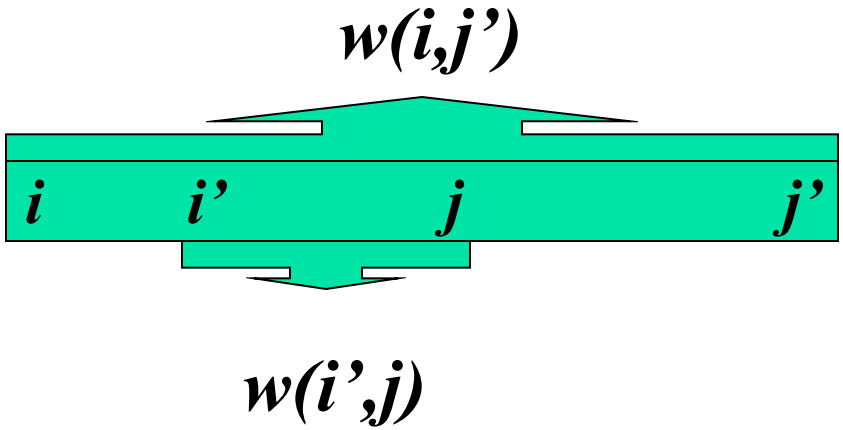
- w的四边形不等式

$$w(i,j)+w(i',j')\le w(i',j)+w(i,j'),\quad i\le i'<j\le j'$$



- w的单调性定义

$$w(i',j)\quad w(i,j'),\quad i\quad i'<j\quad j'$$



动态规划算法的加速条件

- 性质1

  如果***w***的单调性且满足四边形不等式，则***m***也满足四边形不等式，即有：

$$m(i,j)+m(i',j')\le m(i',j)+m(i,j'),\quad i\le i'<j\le j'$$

- 性质2

  如果***m***满足四边形不等式，则***s***满足单调性，即有：

$$s(i',j)\quad s(i,j'),\quad i\quad i'<j\quad j'$$

- 性质2是动态规划算法加速的关键条件

动态规划的加速算法

```
void SpeedDynamicProgramming(int ,int n int **m,int **s,int **w) {
  for (int i=0;i<=n;i++) {   m[i][i]=0;     s[i][i]=0;    }
  for (int r=1;r<n;r++)
     for (int i=1;i<=n-r;i++) {
       int j=i+r;
         i1=(s[i][j-1]>i? s[i][j-1]:i);
         j1=(s[i+1][j]<j? s[i+1][j]:j-1);
       w[i][j]=weight[i][j];     m[i][j]= m[i+1][j];     s[i][j]=i1;
      for (int k=i1+1;k<=j1;k++) {
          int t=m[i][k]+m[k+1][j];
          if (t<m[i][j])  {  m[i][j]=t;  s[i][j]=k;   }
     }
    m[i][j]+=w[i][j];
    }
```

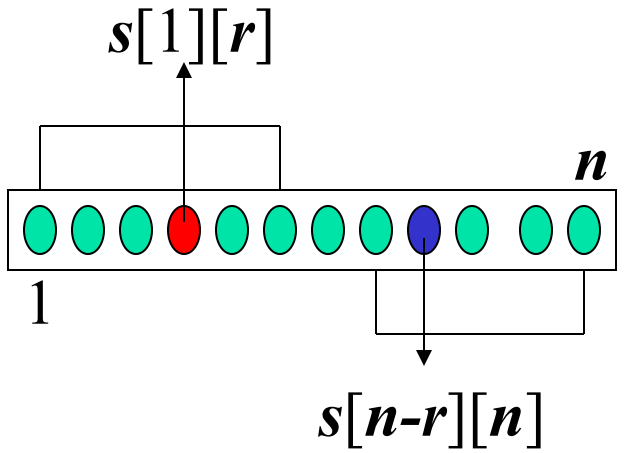
动态规划算法的加速分析

$$O(\sum_{r=0}^{n-1}\sum_{i=1}^{n-r}(1+s(i+1,i+r)-s(i,i+r-1)))$$

$$=O(\sum_{r=0}^{n-1}(n-r+s(n-r,n)-s(1,r))$$

$$=O(\sum_{r=0}^{n-1}n)$$

$$=O(n^2)$$



性质1证明

- 对四边形不等式中的“长度”***l=j'-i*** 应用数学归纳法

- 长度为1时，有***i=i'***或***j=j'***。

  若***i=i'***，则四边形不等式左边 $m(i,j)+m(i',j')$

  变为 $m(i,j)+m(i,j')$

  而右边 $m(i',j)+m(i,j')$

  也变为 $m(i,j)+m(i,j')$

  ，即四边形不等式等号成立。

- 长度为>1时，分两种情况讨论：

  情况1：$i<i'=j<j'$

  情况2：$i<i'<j<j'$

性质1证明（续）

- 情况1: $i < i' = j < j'$

四边形不等式

$$m(i,j)+m(i',j') \le m(i',j)+m(i,j')$$

变化为

$$m(i,j)+m(j,j') \le m(j,j)+m(i,j')$$

注意到 $m(j,j)=0$

于是四边形不等式化简为反三角不等式：

$$m(i,j)+m(j,j') \le m(i,j')$$

设 k 是最优分开位置

$$k=\max\{t \mid m(i,j')=m(i,t-1)+m(t,j')+w(i,j')\}$$

性质1证明（续）

- 情况1.1：$k \le j$

此时有

$$m(i, j') = w(i, j') + m(i, k-1) + m(k, j')$$

因此

$$\underline{m(i, j)} + m(j, j')$$

$$= \underline{w(i, j) + m(i, k-1) + m(k, j)} + m(j, j')$$

$$\le \underline{w(i, j')} + m(i, k-1) + \underline{m(k, j) + m(j, j')}$$

$$\le w(i, j') + m(i, k-1) + \underline{m(k, j')}$$

$$= m(i, j')$$

利用了 $m(i, j) = w(i, j) + m(i, k-1) + m(k, j)$

$w(i, j) \le w(i, j')$

$m(k, j) + m(j, j') \le m(k, j')$

性质1证明（续）

- 情况1.2：$k > j$

此时有

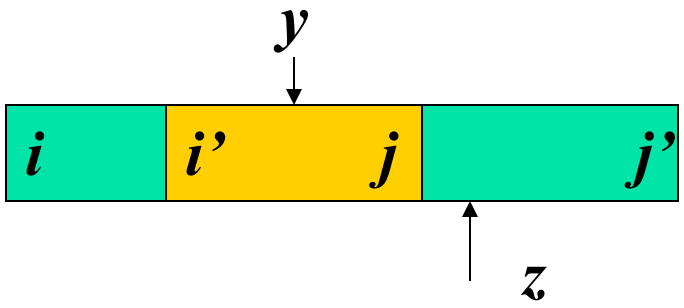
$$m(i, j') = w(i, j') + m(i, k-1) + m(k, j')$$

因此

$$\begin{aligned} & m(i, j) + \underline{m(j, j')} \\ & = m(i, j) + \underline{w(j, j') + m(j, k-1) + m(k, j')} \\ & = \underline{w(i, j')} + m(i, j) + \underline{m(j, k-1) + m(k, j')} \\ & \le w(i, j') + \underline{m(i, k-1)} + m(k, j') \\ & = m(i, j') \end{aligned}$$

性质1证明（续）

- 情况2：$i < i' < j < j'$



设

$$y = \max\{t \mid m(i', j) = m(i', t-1) + m(t, j') + w(i, j)\}$$

$$z = \max\{t \mid m(i, j') = m(i, t-1) + m(t, j') + w(i, j)\}$$

还要分两种情况：$z \le y$ 或 $z > y$

性质1证明（续）

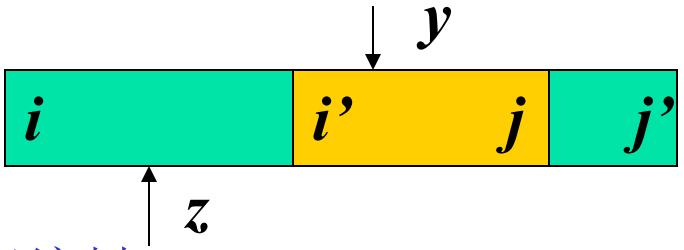
- 情况2.1：$z \le y$

因为 z y j 且 $i < z$ 。因此有

$$
\begin{aligned}
& m(i,j) + m(i',j') \\
&= w(i,j) + m(i,z-1) + m(z,j) \\
&\quad + w(i',j') + m(i',y-1) + m(y,j) \\
&\le w(i,j') + w(i',j) + m(i',y-1) + m(i,z-1) \\
&\quad + m(z,j) + m(y,j') \\
&\le w(i,j') + w(i',j) + m(i',y-1) + m(i,z-1) \\
&\quad + m(z,j') + m(y,j) \\
&= m(i,j') + m(i',j)
\end{aligned}
$$

***w*对(*i,i',j,j'*)应用一次四边形式**

***m*对(*z,y,j,j'*)应用一次四边形式**



性质1证明（续）

- 情况2.2：$z > y$

  因为 $i' \le y < z$ 且 $z < j'$ 。因此有

$$
\begin{aligned}
m(i,j) &= w(i,j) + m(y,j) + w(i',j') + m(\cdots) \\
&= w(i,j) + w(i',j') + + m(i,y-1) + m(y,j) + m(\cdots) \\
&\le w(i,j') + w(i',j) + m(i',z-1) + m(i,y-1) + m(y,j) + m(z,j') \\
&\le w(i,j') + w(i',j) + m(i',z-1) + m(i,y-1) + m(y,j') + m(z,j) \\
&= m(i,j') + m(i',j)
\end{aligned}
$$

w对(i,i',j,j')应用一次四边形式

m对(i',y,z,j')应用一次四边形式