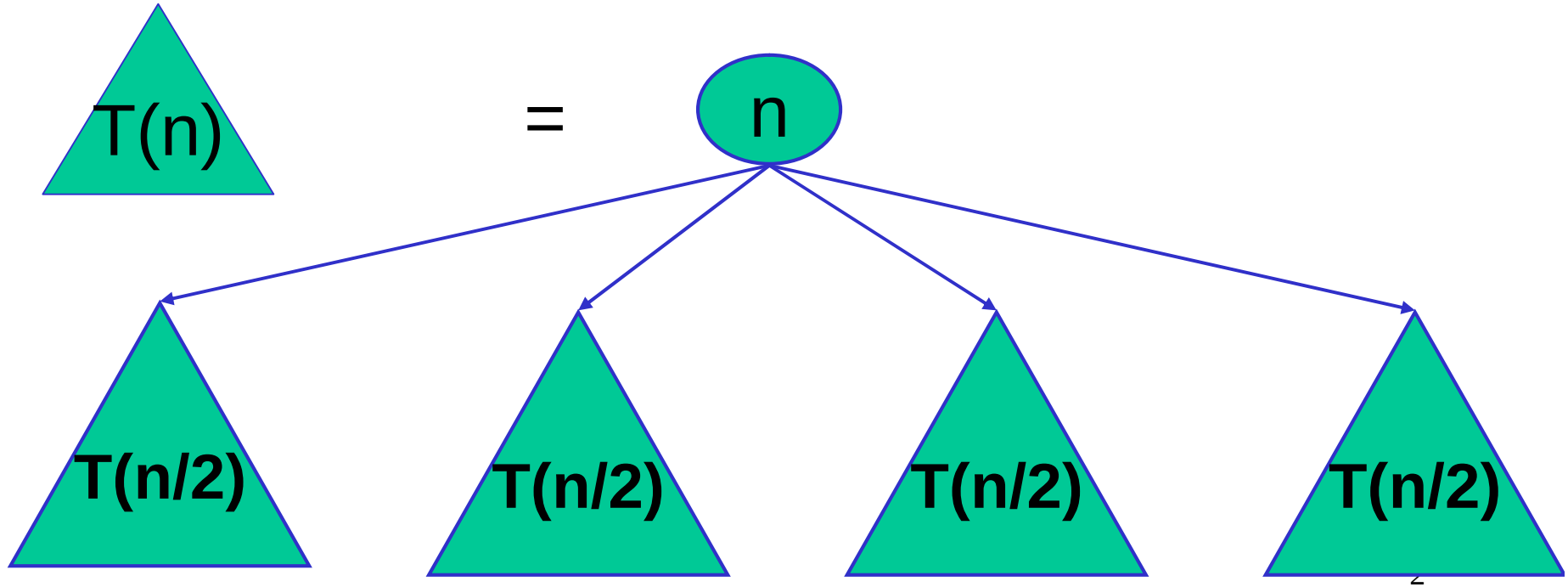
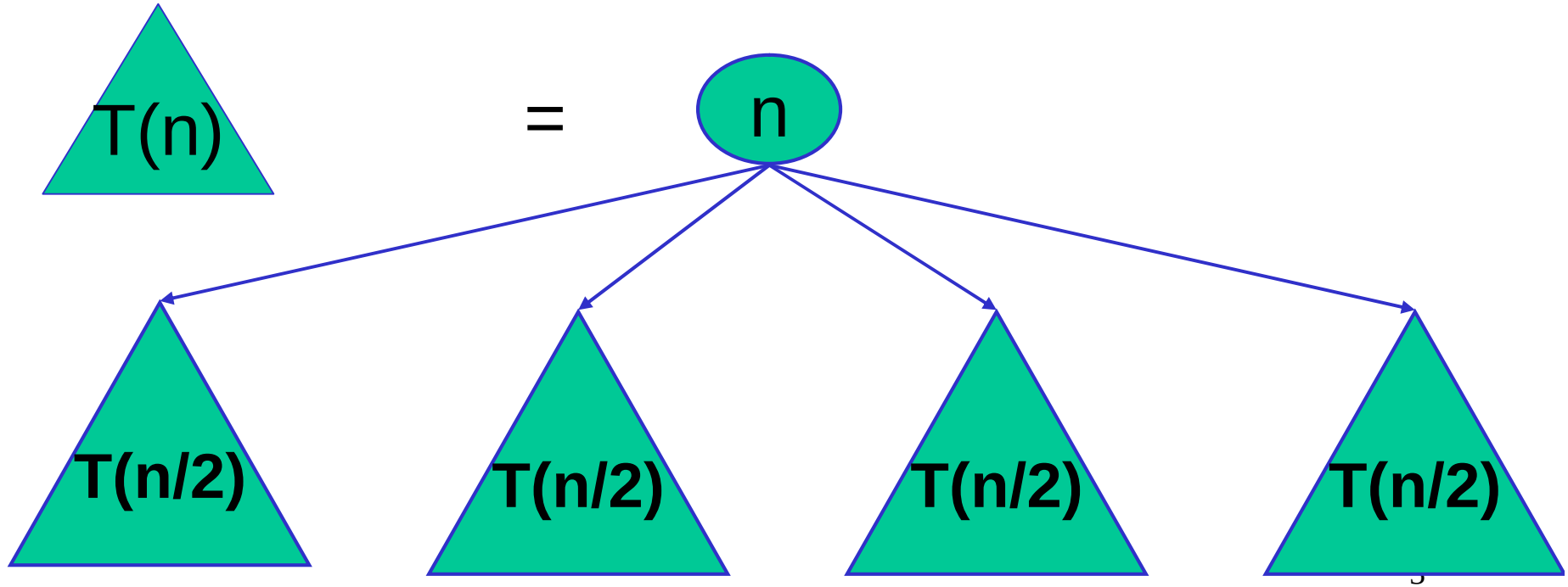
第3章　动态规划

- 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



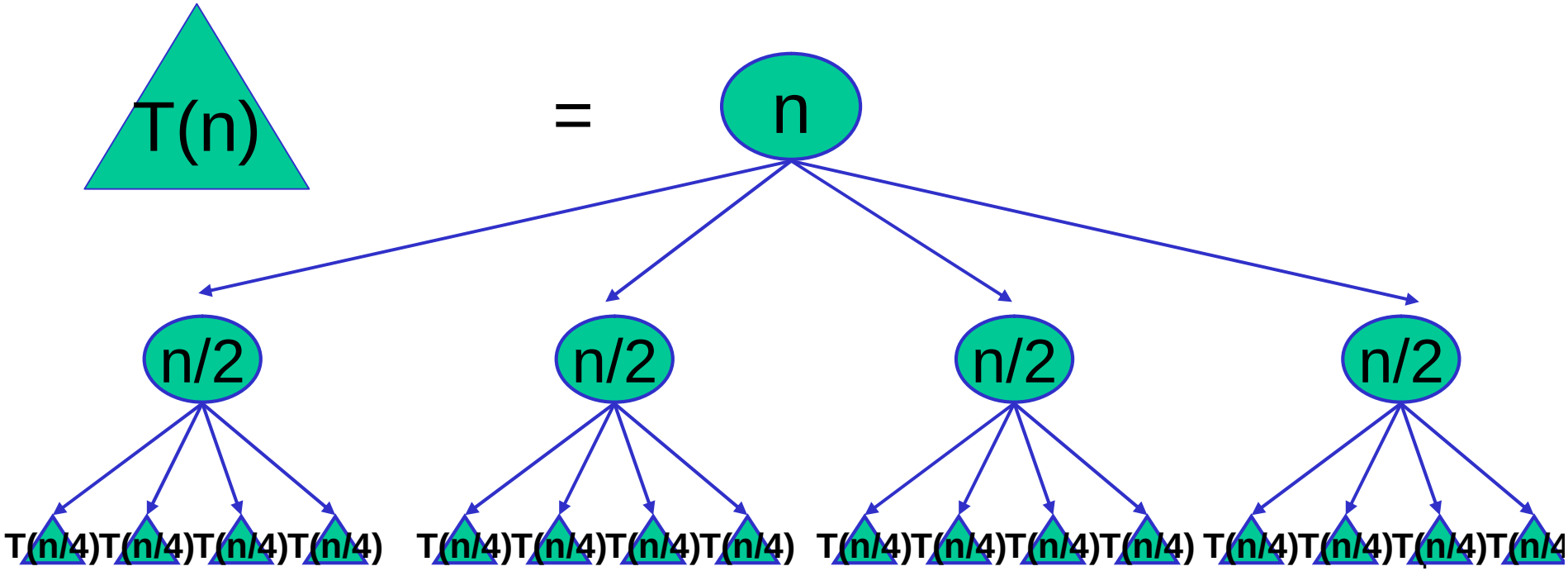
算法总体思想

- 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。

Those who cannot remember the past are doomed to repeat it.

-----George Santayana, The life of Reason, Book I: Introduction and Reason in Common Sense (1905)

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:

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

- 总共有五中完全加括号的方式

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

  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

$$A_1 A_2 ... A_n$$

- 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。

◆**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

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

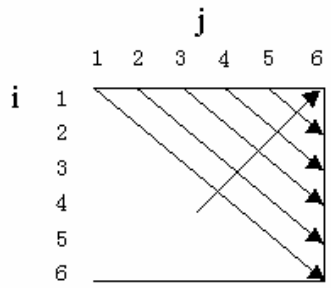
```
public static void matrixChain(int [] p, int [][] m, int [][] s)
{
   int n=p.length-1;
   for (int i = 1; i <= n; i++) m[i][i] = 0;
   for (int r = 2; r <= n; r++)
      for (int i = 1; i <= n - r+1; i++) {
         int j=i+r-1;
         m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
         s[i][j] =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

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

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$O(n^2)$。



(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]

动态规划算法的基本要素

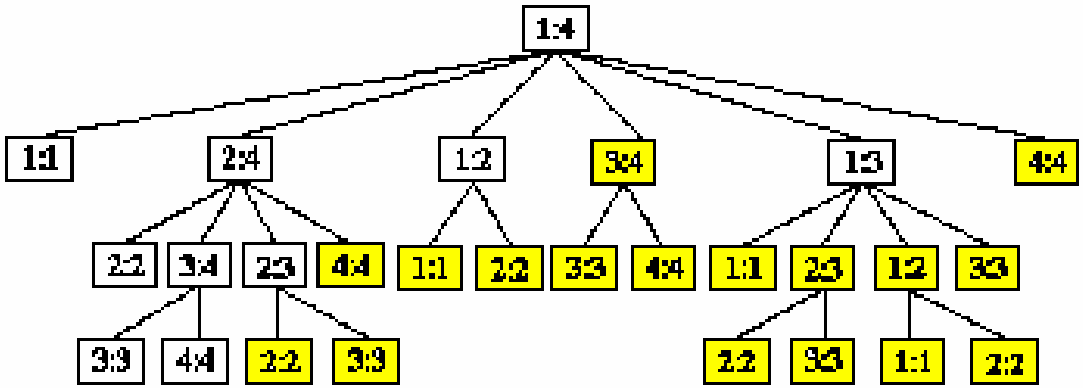
一、最优子结构

- 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。

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。



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。

```
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，$X_i=\{x_1,x_2,\ldots,x_i\}$；$Yj=\{y_1,y_2,\ldots,y_j\}$。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

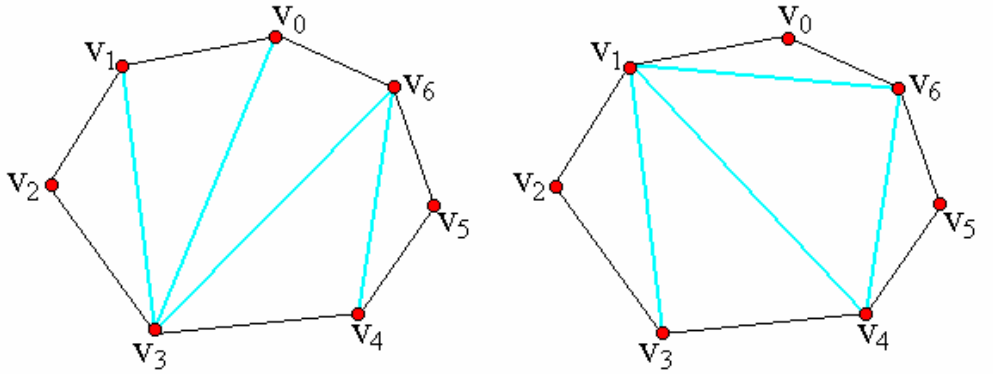
```
Algorithm lcs(int i,int j,char [] x,int [][] b)
  {
    if (i ==0 || j==0) return;
    if (b[i][j]== 1){
      lcs(i-1,j-1,x,b);
      System.out.print(x[i]);
      }
    else if (b[i][j]== 2) lcs(i-1,j,x,b);
      else lcs(i,j-1,x,b);
  }
```

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。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))。

凸多边形最优三角剖分

- 用多边形顶点的逆时针序列表示凸多边形，即$P=\{v_0,v_1,\ldots,v_{n-1}\}$表示具有n条边的凸多边形。
- 若v_i与v_j是多边形上不相邻的2个顶点，则线段v_iv_j称为多边形的一条弦。弦将多边形分割成2个多边形$\{v_i,v_{i+1},\ldots,v_j\}$和$\{v_j,v_{j+1},\ldots v_i\}$。
- **多边形的三角剖分**是将多边形分割成互不相交的三角形的弦的集合T。
- 给定凸多边形P，以及定义在由多边形的边和弦组成的三角形上的权函数w。要求确定该凸多边形的三角剖分，使得即该三角剖分中诸三角形上权之和为最小。

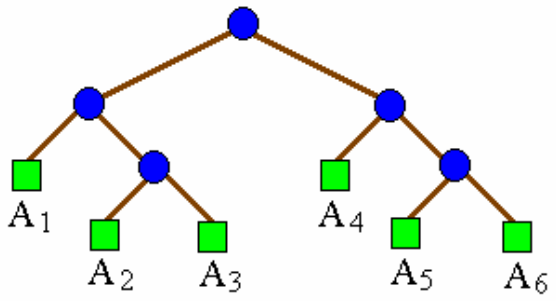


三角剖分的结构及其相关问题

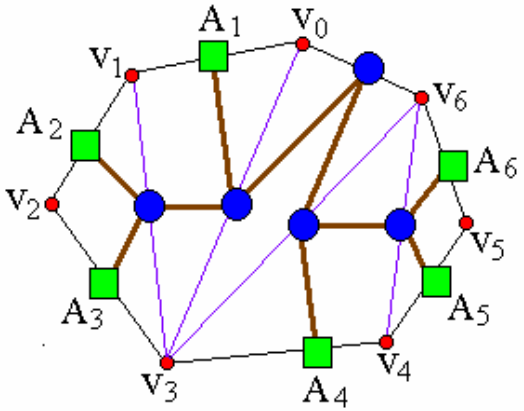
•一个表达式的完全加括号方式相应于一棵完全二叉树，称为表达式的语法树。例如，完全加括号的矩阵连乘积$((A_1(A_2A_3))(A_4(A_5A_6)))$所相应的语法树如图 (a)所示。

•凸多边形$\{v_0,v_1,\ldots v_{n-1}\}$的三角剖分也可以用语法树表示。例如，图 (b)中凸多边形的三角剖分可用图 (a)所示的语法树表示。

•矩阵连乘积中的每个矩阵A_i对应于凸(n+1)边形中的一条边$v_{i-1}v_i$。三角剖分中的一条弦v_iv_j，i<j，对应于矩阵连乘积A[i+1:j]。



(a)



(b)

最优子结构性质

- 凸多边形的最优三角剖分问题有最优子结构性质。
- 事实上，若凸(n+1)边形$P=\{v_0,v_1,\ldots,v_{n-1}\}$的最优三角剖分T包含三角形$v_0v_kv_n$，$1 \leqslant k \leqslant n-1$，则T的权为3个部分权的和：三角形v0vkvn的权，子多边形$\{v_0,v_1,\ldots,v_k\}$和$\{v_k,v_{k+1},\ldots,v_n\}$的权之和。可以断言，由T所确定的这2个子多边形的三角剖分也是最优的。因为若有$\{v_0,v_1,\ldots,v_k\}$或$\{v_k,v_{k+1},\ldots,v_n\}$的更小权的三角剖分将导致T不是最优三角剖分的矛盾。

最优三角剖分的递归结构

•定义t[i][j]，1≤i<j≤n为凸子多边形{vi-1,vi,…,vj}的最优三角剖分所对应的权函数值，即其最优值。为方便起见，设退化的多边形{vi-1,vi}具有权值0。据此定义，要计算的凸(n+1)边形P的最优权值为t[1][n]。

•t[i][j]的值可以利用最优子结构性质递归地计算。当j-i≥1时，凸子多边形至少有3个顶点。由最优子结构性质，t[i][j]的值应为t[i][k]的值加上t[k+1][j]的值，再加上三角形$v_{i-1}v_kv_j$的权值，其中i≤k≤j-1。由于在计算时还不知道k的确切位置，而k的所有可能位置只有j-i个，因此可以在这j-i个位置中选出使t[i][j]值达到最小的位置。由此，t[i][j]可递归地定义为：

$$t[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j\end{cases}$$

多边形游戏

多边形游戏是一个单人玩的游戏，开始时有一个由n个顶点构成的多边形。每个顶点被赋予一个整数值，每条边被赋予一个运算符“+”或“*”。所有边依次用整数从1到n编号。

游戏第1步，将一条边删除。

随后n-1步按以下方式操作:

(1)选择一条边E以及由E连接着的2个顶点V1和V2;

(2)用一个新的顶点取代边E以及由E连接着的2个顶点V1和V2。将由顶点V1和V2的整数值通过边E上的运算得到的结果赋予新顶点。

最后，所有边都被删除，游戏结束。游戏的得分就是所剩顶点上的整数值。

问题:对于给定的多边形，计算最高得分。

最优子结构性质

- 在所给多边形中，从顶点i(1≤i≤n)开始，长度为j(链中有j个顶点)的顺时针链p(i，j) 可表示为v[i]，op[i+1]，…，v[i+j-1]。
- 如果这条链的最后一次合并运算在op[i+s]处发生(1≤s≤j-1)，则可在op[i+s]处将链分割为2个子链p(i，s)和p(i+s，j-s)。
- 设m1是对子链p(i，s)的任意一种合并方式得到的值，而a和b分别是在所有可能的合并中得到的最小值和最大值。m2是p(i+s，j-s)的任意一种合并方式得到的值，而c和d分别是在所有可能的合并中得到的最小值和最大值。依此定义有

a≤m1≤b，c≤m2≤d

(1)当op[i+s]='+'时，显然有a+c≤m≤b+d

(2)当op[i+s]='*'时，有min{ac，ad，bc，bd}≤m≤max{ac，ad，bc，bd}

- 换句话说，主链的最大值和最小值可由子链的最大值和最小值得到。

图像压缩

图像的变位压缩存储格式将所给的象素点序列{p1,p2,…,pn},0⩽pi⩽255分割成m个连续段S1,S2,…,Sm。第i个象素段Si中(1⩽i⩽m)，有l[i]个象素,且该段中每个象素都只用b[i]位表示。设 $t[i]=\sum_{k=1}^{i-1} l[k]$ 则第i个象素段Si为

设 $h_i=\left\lceil \log\left(\max_{t[i]+1\le k\le t[i]+l[i]} p_k+1\right)\right\rceil$，则hi≤b[i]≤8。因此需要用3位表示b[i],如果限制1≤l[i]≤255，则需要用8位表示l[i]。因此，第i个象素段所需的存储空间为l[i]*b[i]+11位。按此格式存储象素序列{p1,p2,…,pn}，需要 $\sum_{i=1}^{m} l[i]*b[i]+11m$ 位的存储空间。

图像压缩问题要求确定象素序列{p1, p2, …, pn}的最优分段，使得依此分段所需的存储空间最少。每个分段的长度不超过256位。

图像压缩

设l[i]，b[i]，是{p1,p2,…,pn}的最优分段。显而易见，l[1]，b[1]是$\{p_1,\ldots,p_{l[1]}\}$的最优分段，且l[i]，b[i]，是$\{p_{l[1]+1},\ldots,p_n\}$的最优分段。即图像压缩问题满足最优子结构性质。

设s[i]，1≤i≤n，是象素序列{p1,…,pn}的最优分段所需的存储位数。由最优子结构性质易知:

$$s[i]=\min_{1\le k\le \min\{i,256\}}\{s[i-k]+k*\mathrm{bmax}(i-k+1,i)\}+11$$

其中 $\mathrm{bmax}(i,j)=\left\lceil \log\left(\max_{i\le k\le j}\{p_k\}+1\right)\right\rceil$

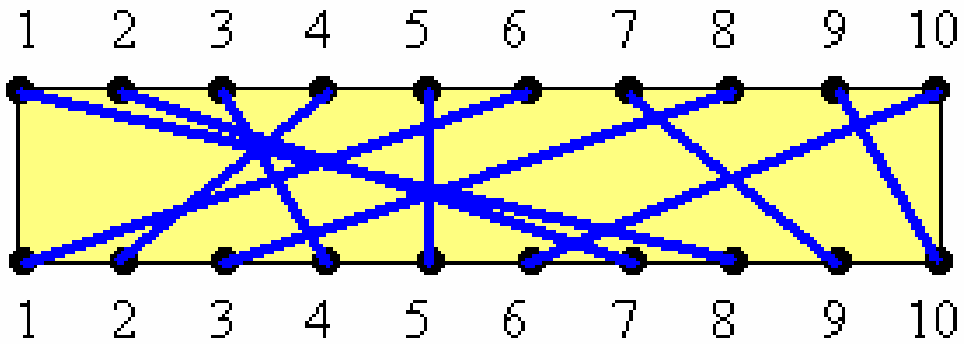
算法复杂度分析:

由于算法**compress**中对k的循环次数不超这256，故对每一个确定的i，可在时间O(1)内完成的计算。因此整个算法所需的计算时间为O(n)。

电路布线

在一块电路板的上、下2端分别有n个接线柱。根据电路设计，要求用导线(i, π(i))将上端接线柱与下端接线柱相连，如图所示。其中π(i)是{1,2,…,n}的一个排列。导线(i, π(i))称为该电路板上的第i条连线。对于任何1≤i<j≤n，第i条连线和第j条连线相交的充分且必要的条件是π(i)>π(j)。

电路布线问题要确定将哪些连线安排在第一层上，使得该层上有尽可能多的连线。换句话说，该问题要求确定导线集Nets={(i, π(i)),1≤i≤n}的最大不相交子集。



电路布线

记 $N(i,j)=\{t \mid (t,\pi(t)) \in Nets, t \le i, \pi(t) \le j\}$。N(i,j)的最大不相交子集为MNS(i,j)。Size(i,j)=|MNS(i,j)|。

(1)当i=1时，$MNS(1,j)=N(1,j)=\begin{cases}\varnothing & j<\pi(1)\\ \{(1,\pi(1))\} & j\ge\pi(1)\end{cases}$

(2)

2.1

2.2 i,j)

1,π(i)-1)的最大不相交子集。

2.3 若 $(i,\pi(i)) \notin N(i,j)$，则对任意(t,π(t)) ∈ MNS(i,j)有t<i。从而 $MNS(i,j) \subseteq N(i-1,j)$，因此，Size(i,j)⩽Size(i-1,j)。另一方面 $MNS(i-1,j) \subseteq N(i,j)$，故又有Size(i,j)⩾Size(i-1,j)，从而Size(i,j)=Size(i-1,j)。

(1)当i=1时 $Size(1,j)=\begin{cases}0 & j<\pi(1)\\ 1 & j\ge\pi(1)\end{cases}$

(2)当i>1时

$$Size(i,j)=\begin{cases}Size(i-1,j) & j<\pi(i)\\ \max\{Size(i-1,j),Size(i-1,\pi(i)-1)+1\} & j\ge\pi(i)\end{cases}$$

流水作业调度

n个作业{1，2，…，n}要在由2台机器M1和M2组成的流水线上完成加工。每个作业加工的顺序都是先在M1上加工，然后在M2上加工。M1和M2加工作业i所需的时间分别为a_i和b_i。

流水作业调度问题要求确定这n个作业的最优加工顺序，使得从第一个作业在机器M1上开始加工，到最后一个作业在机器M2上加工完成所需的时间最少。

分析：

- 直观上，一个最优调度应使机器M1没有空闲时间，且机器M2的空闲时间最少。在一般情况下，机器M2上会有机器空闲和作业积压2种情况。
- 设全部作业的集合为N={1，2，…，n}。$S \subseteq N$是N的作业子集。在一般情况下，机器M1开始加工S中作业时，机器M2还在加工其他作业，要等时间t后才可利用。将这种情况下完成S中作业所需的最短时间记为T(S,t)。流水作业调度问题的最优值为T(N,0)。

流水作业调度

设π是所给n个流水作业的一个最优调度，它所需的加工时间为$a_{\pi(1)}+T'$。其中T'是在机器M2的等待时间为$b_{\pi(1)}$时，安排作业π(2)，…，π(n)所需的时间。

记S=N-{π(1)}，则有$T'=T(S,b_{\pi(1)})$。

证明：事实上，由T的定义知$T'\geq T(S,b_{\pi(1)})$。若$T'>T(S,b_{\pi(1)})$，设π'是作业集S在机器M2的等待时间为$b_{\pi(1)}$情况下的一个最优调度。则π(1)，π'(2)，…，π'(n)是N的一个调度，且该调度所需的时间为$a_{\pi(1)}+T(S,b_{\pi(1)})<a_{\pi(1)}+T'$。这与π是N的最优调度矛盾。故$T'\leq T(S,b_{\pi(1)})$。从而$T'=T(S,b_{\pi(1)})$。这就证明了流水作业调度问题具有最优子结构的性质。

由流水作业调度问题的最优子结构性质可知，

$$T(N,0)=\min_{1\leq i\leq n}\{a_i+T(N-\{i\},b_i)\}$$

$$T(S,t)=\min_{i\in S}\{a_i+T(S-\{i\},b_i+\max\{t-a_i,0\})\}$$

Johnson不等式

对递归式的深入分析表明，算法可进一步得到简化。

设π是作业集S在机器M2的等待时间为t时的任一最优调度。若π(1)=i, π(2)=j。则由动态规划递归式可得:

$T(S,t)=a_i+T(S-\{i\},b_i+\max\{t-a_i,0\})=a_i+a_j+T(S-\{i,j\},t_{ij})$

其中，$t_{ij}=b_j+\max\{b_i+\max\{t-a_i,0\}-a_j,0\}$

$$=b_j+b_i-a_j+\max\{\max\{t-a_i,0\},a_j-b_i\}$$

$$=b_j+b_i-a_j+\max\{t-a_i,a_j-b_i,0\}$$

$$=b_j+b_i-a_j-a_i+\max\{t,a_i+a_j-b_i,a_i\}$$

如果作业i和j满足$\min\{b_i, a_j\}\geqslant\min\{b_j, a_i\}$，则称作业i和j满足Johnson不等式。

流水作业调度的Johnson法则

$t_{ij} = b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_i, a_i\}$

交换作业i和作业j的加工顺序，得到作业集S的另一调度，它所需的加工时间为$T'(S,t)=a_i+a_j+T(S-\{i,j\},t_{ji})$

其中$t_{ji} = b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_j, a_j\}$

当作业i和j满足Johnson不等式时，有

$$\max\{-b_i,-a_j\} \le \max\{-b_j,-a_i\}$$

$$a_i + a_j + \max\{-b_i,-a_j\} \le a_i + a_j + \max\{-b_j,-a_i\}$$

$$\max\{a_i + a_j - b_i, a_i\} \le \max\{a_i + a_j - b_j, a_j\}$$

$$\max\{t, a_i + a_j - b_i, a_i\} \le \max\{t, a_i + a_j - b_j, a_j\}$$

由此可见当作业i和作业j不满足Johnson不等式时，交换它们的加工顺序后，不增加加工时间。对于流水作业调度问题，必存在最优调度π，使得作业π(i)和π(i+1)满足Johnson不等式。进一步还可以证明，调度满足Johnson法则当且仅当对任意i<j有

$$\min\{b_{\pi(i)}, a_{\pi(j)}\} \ge \min\{b_{\pi(j)}, a_{\pi(i)}\}$$

由此可知，**所有满足Johnson法则的调度均为最优调度。**

算法描述

流水作业调度问题的Johnson算法

(1)令 $N_1=\{i \mid a_i < b_i\}, N_2=\{i \mid a_i \geq b_i\}$;

(2)将N_1中作业依a_i的非减序排序；将N_2中作业依b_i的非增序排序;

(3)N_1中作业接N_2中作业构成满足Johnson法则的最优调度。

算法复杂度分析：

算法的主要计算时间花在对作业集的排序。因此，在最坏情况下算法所需的计算时间为O(nlogn)。所需的空间为O(n)。

0-1背包问题

给定n种物品和一背包。物品i的重量是w_i，其价值为v_i，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

0-1背包问题是一个特殊的整数规划问题。

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

0-1背包问题

设所给0-1背包问题的子问题

$$\max \sum_{k=i}^{n} v_k x_k$$

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \end{cases}$$

算法复杂度分析：

从m(i，j)的递归式容易看出，算法需要O(nc)计算时间。当背包容量c很大时，算法需要的计算时间较多。例如，当$c>2^n$时，算法需要$\Omega(n2^n)$计算时间。

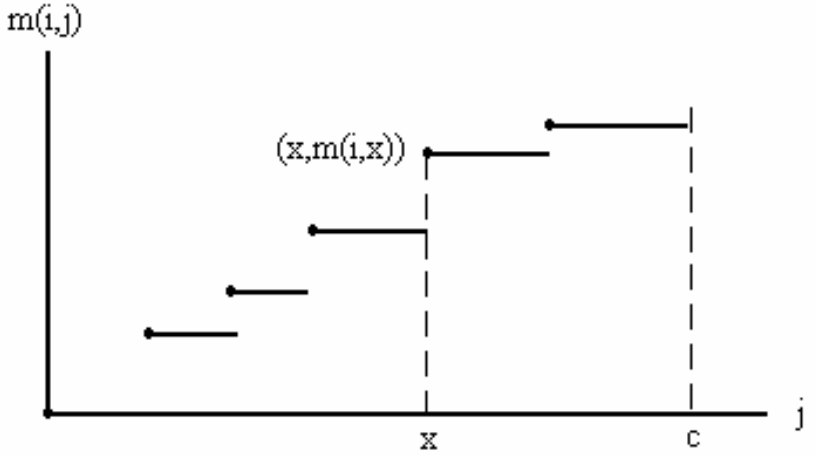
的 品为i，i+ 最优子结构性质，可以建立计算m(i，j)的递归式如下。

$$m(i,j)=\begin{cases}\max\{m(i+1,j),m(i+1,j-w_i)+v_i\} & j \ge w_i \\ m(i+1,j) & 0 \le j < w_i\end{cases}$$

$$m(n,j)=\begin{cases}v_n & j \ge w_n \\ 0 & 0 \le j < w_n\end{cases}$$

算法改进

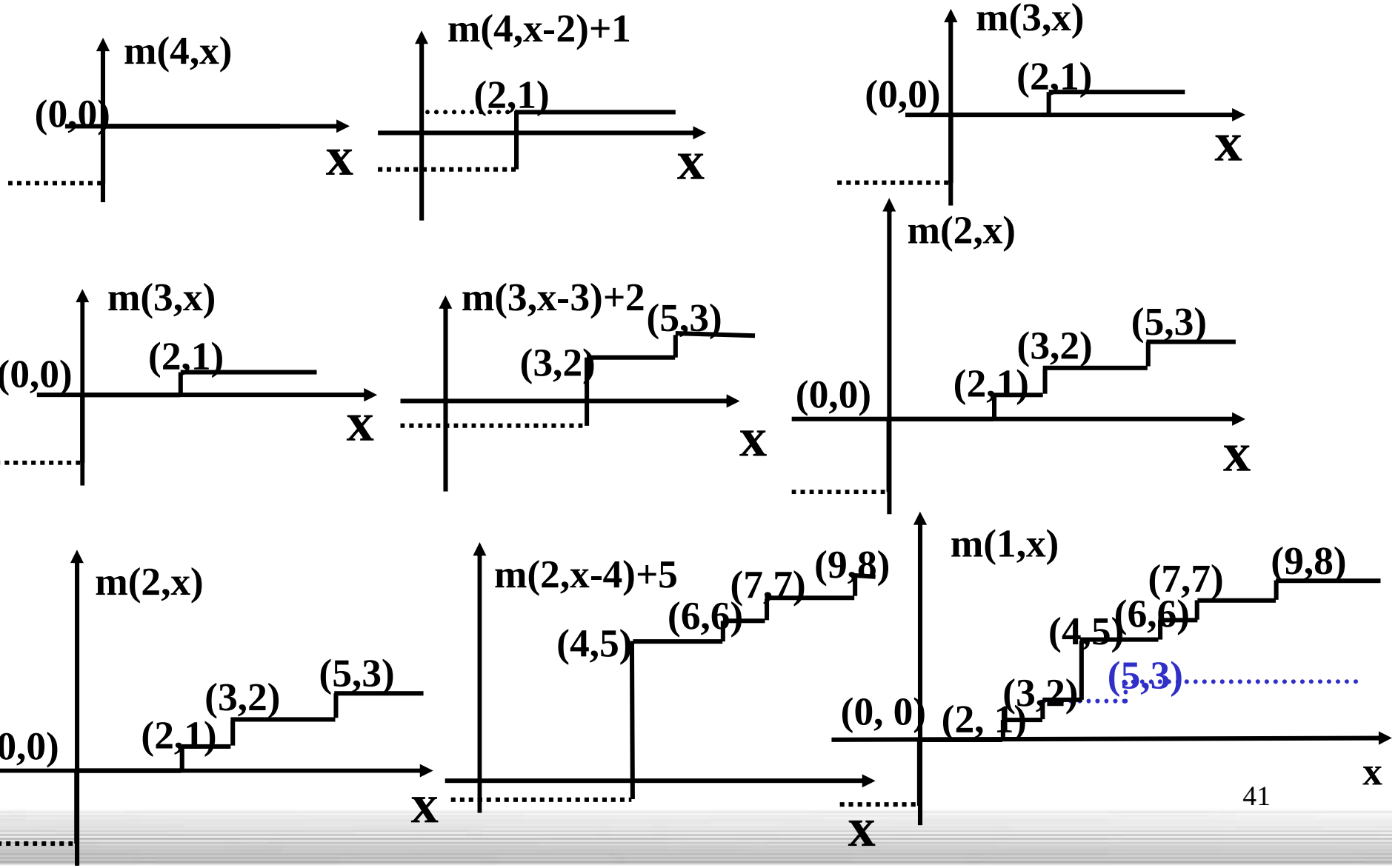
由m(i,j)的递归式容易证明，在一般情况下，对每一个确定的i(1≤i≤n)，函数m(i,j)是关于变量j的阶梯状单调不减函数。跳跃点是这一类函数的描述特征。在一般情况下，函数m(i,j)由其全部跳跃点惟一确定。如图所示。



对每一个确定的i(1≤i≤n)，用一个表p[i]存储函数m(i，j)的全部跳跃点。表p[i]可依计算m(i，j)的递归式递归地由表p[i+1]计算，初始时p[n+1]={(0，0)}。

典型例子（一）

n=3，c=6，w={4，3，2}，v={5，2，1}。



算法改进

- 函数m(i,j)是由函数m(i+1,j)与函数m(i+1,j-wi)+vi作max运算得到的。因此，函数m(i,j)的全部跳跃点包含于函数m(i+1，j)的跳跃点集p[i+1]与函数m(i+1，j-wi)+vi的跳跃点集q[i+1]的并集中。易知，(s,t)∈q[i+1]当且仅当wi≤s≤c且(s-wi,t-vi)∈p[i+1]。因此，容易由p[i+1]确定跳跃点集q[i+1]如下

$$q[i+1]=p[i+1]\oplus(wi,vi)=\{(j+wi,m(i,j)+vi)|(j,m(i,j))\in p[i+1]\}$$

- 另一方面，设(a，b)和(c，d)是p[i+1]∪q[i+1]中的2个跳跃点，则当c≥a且d<b时，(c，d)受控于(a，b)，从而(c，d)不是p[i]中的跳跃点。除受控跳跃点外，p[i+1]∪q[i+1]中的其他跳跃点均为p[i]中的跳跃点。
- 由此可见，在递归地由表p[i+1]计算表p[i]时，可先由p[i+1]计算出q[i+1]，然后合并表p[i+1]和表q[i+1]，并清除其中的受控跳跃点得到表p[i]。

典型例子（二）

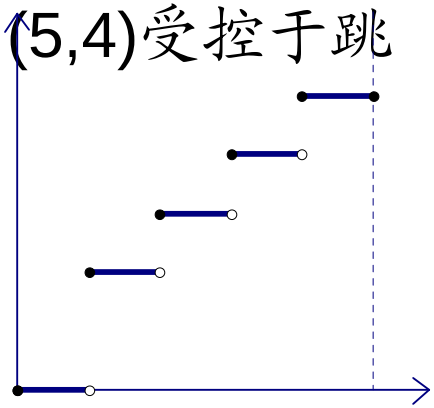
n=5，c=10，w={2，2，6，5，4}，v={6，3，5，4，6}。

初始时p[6]={(0,0)}，(w5,v5)=(4,6)。因此，

q[6]=p[6]⊕(w5,v5)={(4,6)}。

p[5]={(0,0),(4,6)}。

q[5]=p[5]⊕(w4,v4)={(5,4),(9,10)}。从跳跃点集p[5]与q[5]的并集p[5]∪q[5]={(0,0),(4,6),(5,4),(9,10)}中看到跳跃点(5,4)受控于跳跃点(4,6)。将受控跳跃点(5,4)清除后，得到



p[4]={(0,0),(4,6),(9,10)}

q[4]=p[4]⊕(6，5)={(6，5)，(10，11)}

p[3]={(0，0)，(4，6)，(9，10)，(10，11)}

q[3]=p[3]⊕(2，3)={(2，3)，(6，9)}

p[2]={(0，0)，(2，3)，(4，6)，(6，9)，(9，10)，(10，11)}

q[2]=p[2]⊕(2，6)={(2，6)，(4，9)，(6，12)，(8，15)}

p[1]={(0，0)，(2，6)，(4，9)，(6，12)，(8，15)}

p[1]的最后的那个跳跃点(8,15)给出所求的最优值为

m(1,c)=15

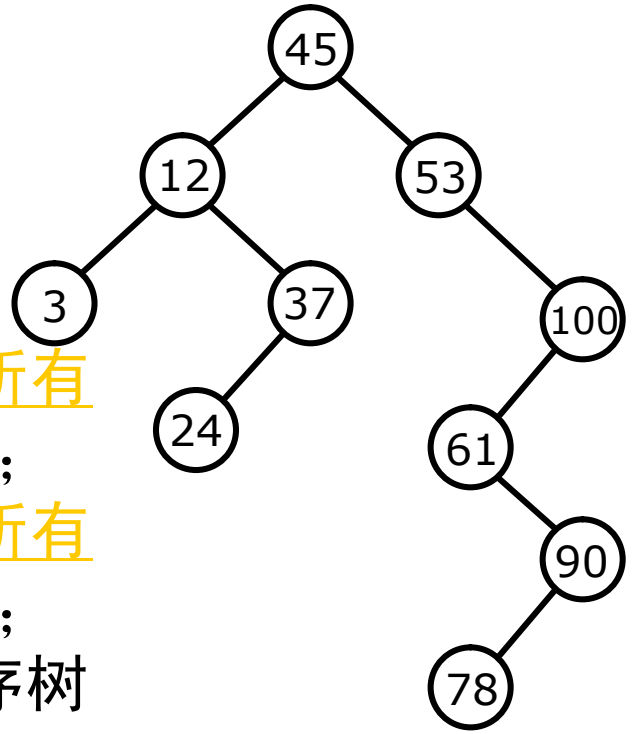
算法复杂度分析

上述算法的主要计算量在于计算跳跃点集p[i](1≤i≤n)。由于q[i+1]=p[i+1]⊕(w_i，v_i)，故计算q[i+1]需要O(|p[i+1]|)计算时间。合并p[i+1]和q[i+1]并清除受控跳跃点也需要O(|p[i+1]|)计算时间。从跳跃点集p[i]的定义可以看出，p[i]中的跳跃点相应于$x_i,\ldots,x_n$的0/1赋值。因此，p[i]中跳跃点个数不超过2^{n-i+1}。由此可见，算法计算跳跃点集p[i]所花费的计算时间为 $O\left(\sum_{i=2}^{n}|p[i+1]|\right)=O\left(\sum_{i=2}^{n}2^{n-i}\right)=O(2^n)$

从而，改进后算法的计算时间复杂性为$O(2^n)$。当所给物品的重量w_i(1≤i≤n)是整数时，|p[i]|≤c+1，(1≤i≤n)。在这种情况下，改进后算法的计算时间复杂性为O(min{nc,2^n})。

最优二叉搜索树

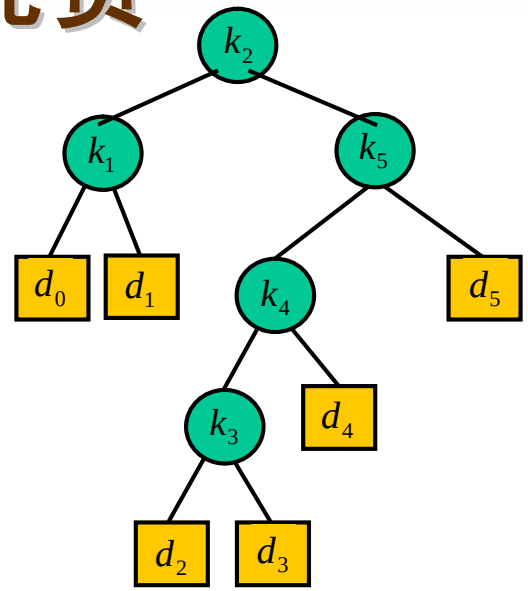
- 什么是二叉搜索树？



（1）若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
（2）若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
（3　它的左、右子树也分别为二叉排序树

在随机的情况下，二叉查找树的平均查找长度和$\log n$是等数量级的

二叉查找树的期望耗费



- 查找成功与不成功的概率

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

- 二查找树的期望耗费

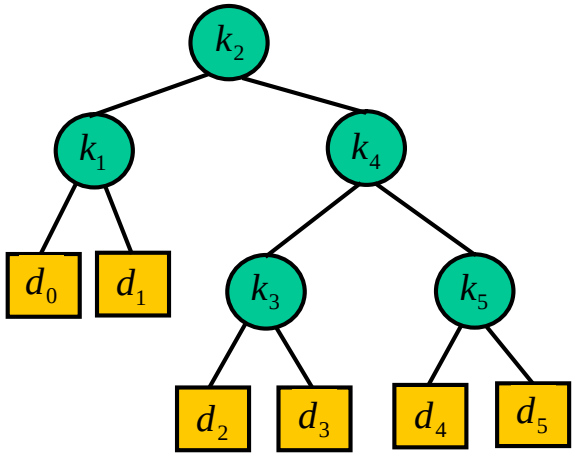
$$E(\text{search cost in } T)$$

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i$$

- 有 n个节点的二叉树的个数为：$\Omega(4^n / n^{3/2})$
- 穷举搜索法的时间复杂度为指数级

二叉查找树的期望耗费示例



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

最优二叉搜索树

最优二叉搜索树T_{ij}的平均路长为p_{ij}，则所求的最优值为$p_{1,n}$。由最优二叉搜索树问题的最优子结构性质可建立计算p_{ij}的递归式如下

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

记$w_{i,j}p_{i,j}$为m(i,j)，则$m(1,n)=w_{1,n}p_{1,n}=p_{1,n}$为所求的最优值。计算m(i,j)的递归式为

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

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

注意到，

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

可以得到$O(n^2)$的算法