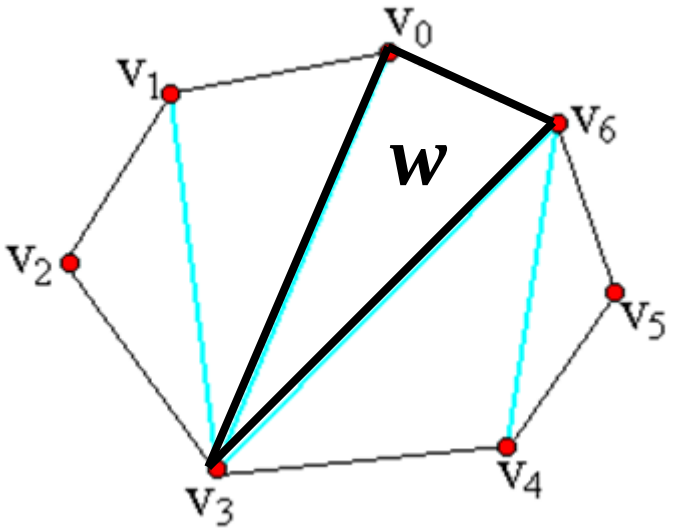
3.5 凸多边形最优三角剖分

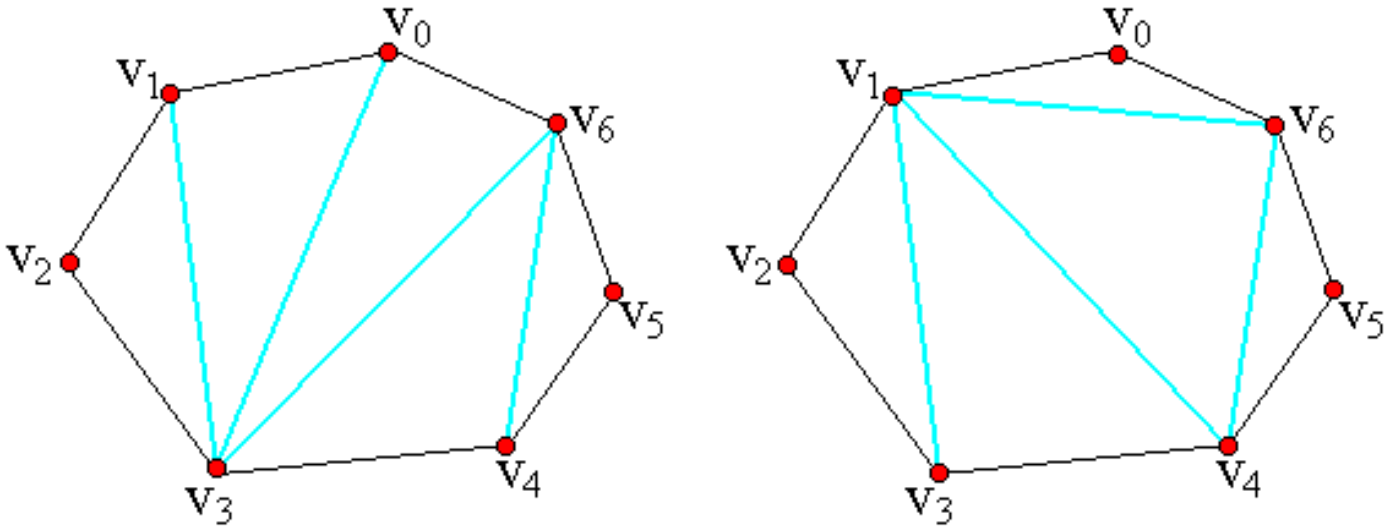
- 用多边形顶点的逆时针序列表示凸多边形，即$P=\{v_0,v_1,\cdots,v_{n-1}\}$表示具有n条边的凸多边形。
- 多边形的三角剖分是将多边形分割成互不相交的三角形的弦的集合T。

例如：$P=\{v_0,v_1,\cdots,v_6\}$，表示7边的凸多边形

右图为一个三角剖分



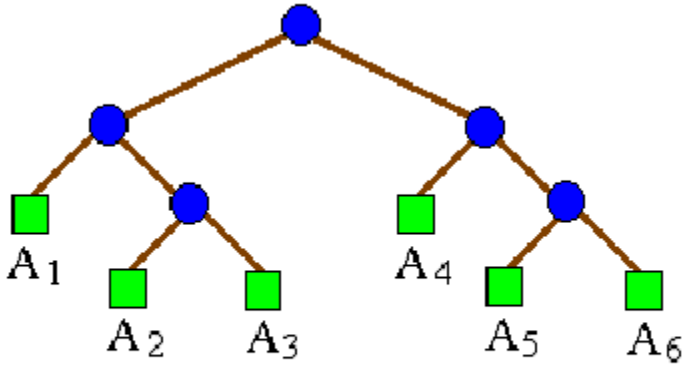
- 凸多边形最优三角剖分问题：
  - 输入:给定凸多边形P，以及定义在由多边形的边和弦组成的三角形上的权函数w。
  - 输出:要求确定该凸多边形的三角剖分，使得即该三角剖分中诸三角形上权之和为最小。



1、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。

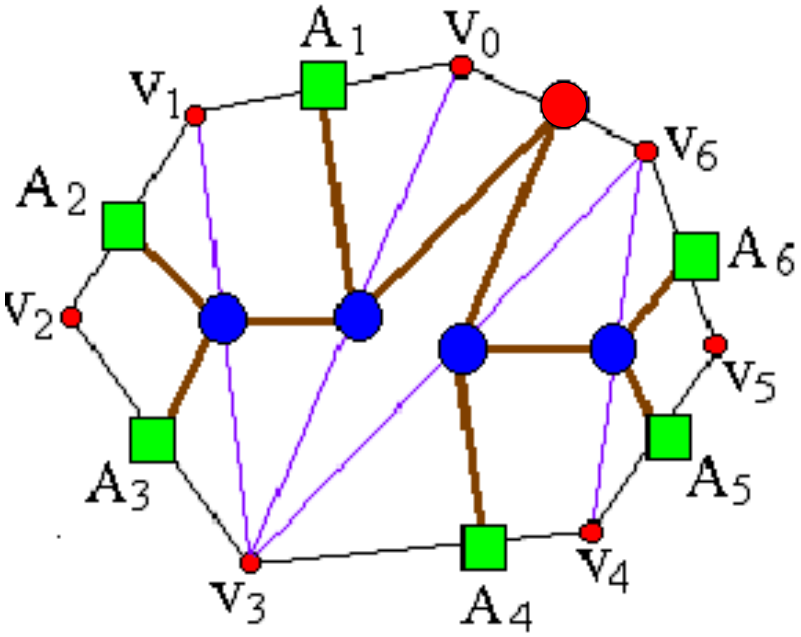
- 叶结点：矩阵



(a)

语法树表示凸多边形

- 凸多边形$\{v_0,v_1,\cdots v_{n-1}\}$的三角剖分也可以用语法树表示。
- 一个凸n多边形的三角剖分对应一棵有n-1个叶结点的语法树。



(b)

与矩阵连乘积问题的比较

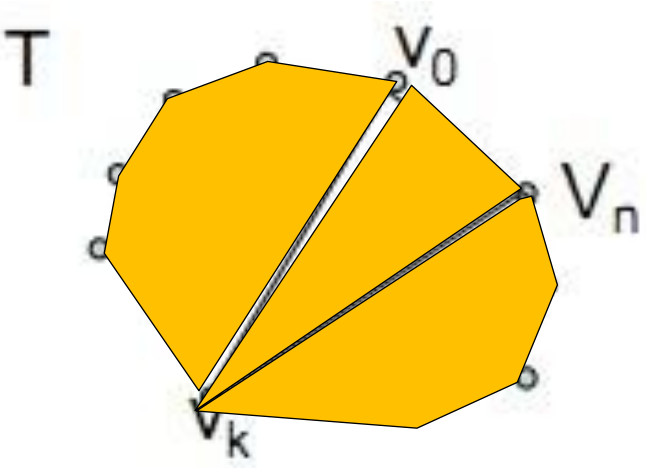
✓矩阵连乘积中的每个矩阵A_i对应于凸(n+1)边形中的一条边$v_{i-1}v_i$。

✓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]。

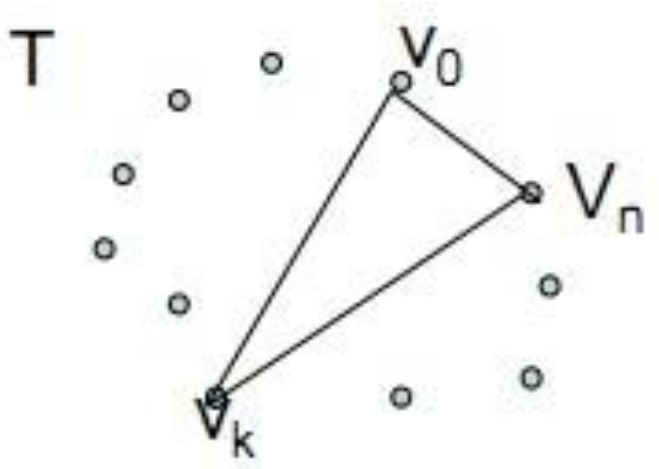
语法树	凸多边形
根节点	边v_0v_6
内节点	三角剖分的弦
叶子节点	除v_0v_6外的各边

2、最优子结构性质

- 分析：
  - 若凸(n+1)边形P=$\{v_0, v_1, \cdots, v_{n-1}\}$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$v_0v_kv_n$的权
    - 子多边形$\{v_0, v_1, \ldots, v_k\}$
    - 子多边形$\{v_k, v_{k+1}, \ldots, v_n\}$



- 由T所确定的这2个子多边形的三角剖分也是最优的。
- 反证法：
  - 因为若有$\{v_0,v_1,\cdots,v_k\}$或$\{v_k,v_{k+1},\cdots,v_n\}$的更小权的三角剖分，将权值替换现有三角剖分，将导致T不是最优三角剖分的矛盾。

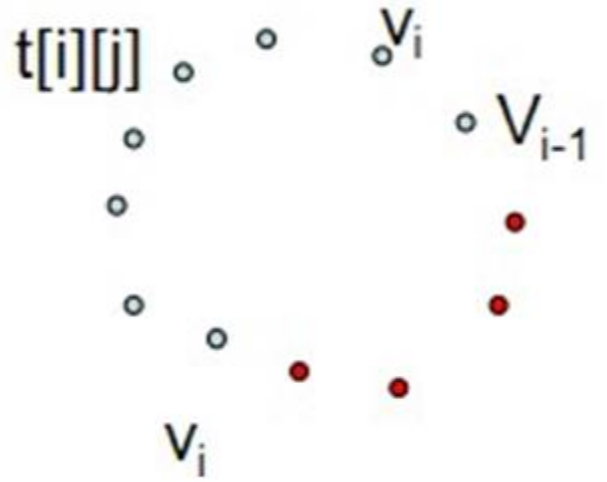


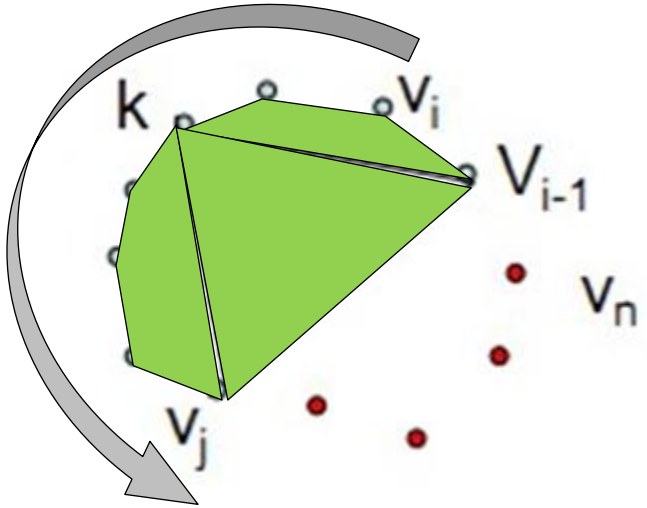
3、最优三角剖分的递归结构

◆定义t[i][j]，$1 \le i < j \le n$为凸子多边形$\{v_{i-1},v_i,\ldots,v_j\}$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$\{v_{i-1},v_i\}$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。





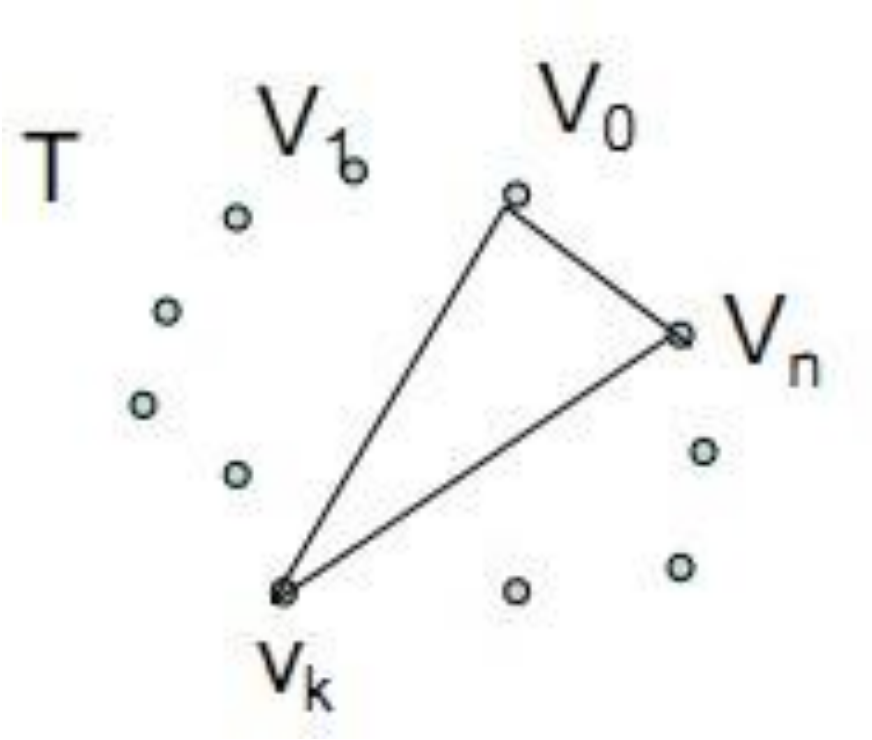
◆当j-i≥1时，凸子多边形至少有3个顶点。t[i][j]的值应为t[i][k]的值加上t[k+1][j]的值，再加上三角形$v_{i-1}v_kv_j$的权值，其中i≤k≤j-1。

◆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。

◆t[i][j]可递归地定义为：

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

- **凸(n+1)边形P的最优权值为t[1][n]。**



$$t[1][n]=\min\{\, t[1][k]+t[k+1][n]+w(v_0v_kv_n) \mid 1\le k<n\}$$

4. 计算最优值

```
1.  Void MinWeightTriangulation(int n, Type **t, int **s)
2.   {  for (int i = 1; i <= n; i++) t[i][i] = 0;
3.      for (int r = 2; r <= n; r++)
4.         for (int i = 1; i <= n – r +1; i++) {
5.            int j = i + r – 1;
6.            t[i][j] = t[i+1][j] + w(i–1, i, j);
7.            s[i][j] = i;
8.            for (int k = i + 1; k < j; k++) {
9.               int u = t[i][k] + t[k+1][j] + w(i–1, k, j);
10.              if (u < t[i][j]) {t[i][j] = u; s[i][j] = k;}
11. }}}
```

3.10 0-1背包问题

有编号分别为a, b, c, d, e的五件宝石原石，它们的重量分别是2,2,6,5,4克拉，它们的价值分别是6,3,5,4,6万元。

现在给你个承重为10克拉的背包，如何让背包里装入的物品具有最大的价值总和?

例如：a、b、e

物品名称	重量(克拉)	价值(万)
a	2	6
b	2	3
c	6	5
d	5	4
e	4	6

- 0-1背包问题是一类经典的组合优化问题
- 对0-1背包问题的研究可以广泛运用于资源分配、投资决策、货物装载等方面。

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?

对每种物品i只有两种选择，即装入背包或者不装入背包，不能将物品i重复装入，也不允许部分装入。因此，该问题被称为0-1背包问题。

0-1背包问题是一个特殊的整数规划问题。问题的解是找出一个n元的0,1向量$\{x_1, x_2, \ldots\ldots x_n\}$，使得

- ✓ 输入：$C>0$，$w_i>0$，$v_i>0$，$1 \leqslant i \leqslant n$
- ✓ 输出：$(x_1, x_2, \ldots, x_n)$，$x_i \in \{0, 1\}$，满足

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

x_i表示物品i是否装入背包

①最优子结构性质

- 设$(y_1, y_2,\ldots,y_n)$ 是所给问题的一个最优解，则 $(y_2,\ldots, y_n)$是下面相应**子问题**的一个最优解：

$$\max \sum_{i=2}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=2}^{n} w_i x_i \le c - w_1 y_1 \\ x_i \in \{0,1\}, 2 \le i \le n \end{cases}$$

- 反证法证明

如果不是最优解，进行替代，矛盾！

②递归关系

设所给0-1背包问题的子问题:

最优值$m(i,j)$是

$$\max \sum_{k=i}^{n} v_k x_k \qquad \begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, i \le k \le n \end{cases}$$

即$m(i,j)$是背包容量为j，可选择物品为i，$i+1$，…，n时0-1背包问题的最优值。

由0-1背包问题的最优子结构性质，可以建立计算m(*i*,*j*)的递归式如下。

选择了物品*i*的最优解

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

未选择物品*i*的最优解

容量不够装物品*i*的情况

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

只剩下最后一个物品*n*的简单情况

- 用二维数组m[i][j], 0≤j≤c, 存储m(i, j)的值。
- 求解0-1背包问题就是在二维数组m中填入相应的值。而m[1][c]中的值就是该背包问题的最优值。

- 二维数组m中最先填入物品n的最优值m(n, j):
  - 若$0 \le j < w_n$，m[n][j]=0;
  - 若$j \ge w_n$，m[n][j]=v_n。
- 然后从物品n–1到物品1逐个填入它们的最优值m(i, j):
  - 若$0 \le j < w_i$，m[i][j]=m[i+1][j];
  - 若$j \ge w_i$，m[i][j] = max{m[i+1][j], m[i+1][$j-w_i$]+v_i} 。

算法说明:

1. 处理最简单情况，即填好m(n,j)。当$j<w_n$时，m(n,j)=0； 当$j<w_n$时， m(n,j)=v[n];
2. 从第n-1行开始到第2行，一行一行的填写m。假设当前填写的是第i行
   1. 如果第i个物品不能放入背包，则m(i,j)=m(i+1,j)
   2. 否则，比较放与不放两种情况下哪种更优，取更优的值最为m(i,j)的值。
3. 考虑第一个物品，不能放的情况m(1,c)=m(2,c);
   否则m(1,c)=放的情况下和不放情况下的最大值。
4. 返回m(1,c)

}

```
void Traceback(T **m, int w[ ],
               int c, int n, int x[])  {
  // 计算x[i]，它表示i物品是否装入背包
  for (int i = 1; i < n; i++)
    if (m[i][c] ==m[i+1][c]) x[i] = 0;    //未装入
    else {   x[i] = 1;                    //装入了背包
             c -= w[i];   }
    x[n] = (m[n][c]) ? 1 : 0;
}
```

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。

存在的不足：

- 求物品重量w_i是整数
- 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。

算法改进

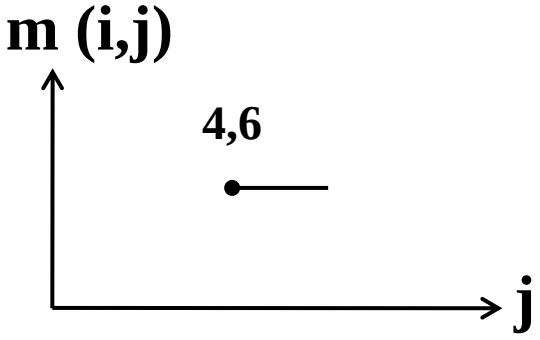
背包容量为实数时，
递归公式仍然成立

例子：n=5，c=10，w={2，2，6，5，4}，
v={6，3，5，4，6}由计算m(i,j)的递归式，

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

当i=5时，

$$m(5,j)=\begin{cases} 6 & j \geq 4 \\ 0 & 0 \leq j < 4 \end{cases}$$



该函数是关于变量j的阶梯状函数。

- 对于每一个确定的i，函数m(i, j)是关于变量j的阶梯状单调不减函数：
- 由0-1背包问题的递归式

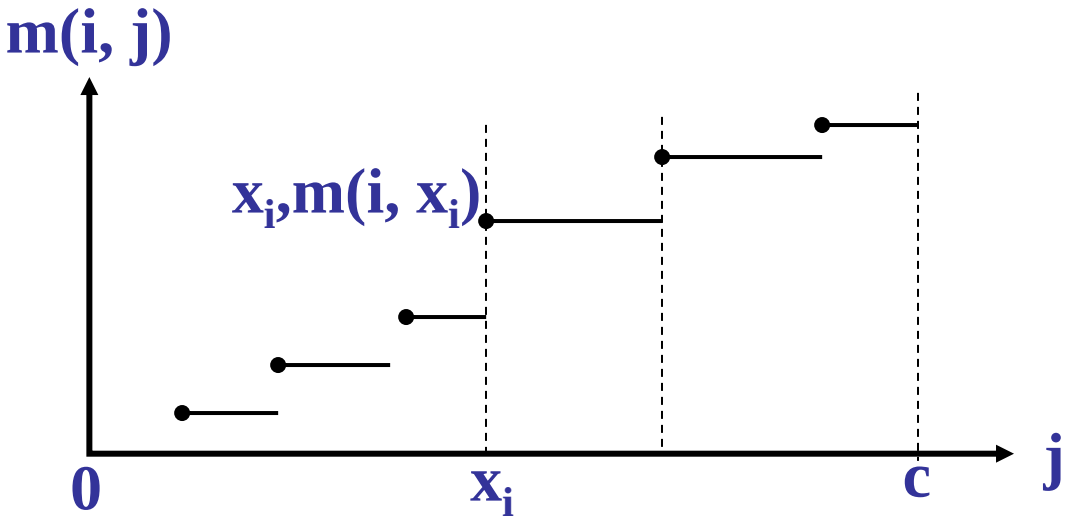
$$m(i,j)=\begin{cases}\max\{m(i+1,j),\ m(i+1,j-w_i)+v_i\} & j\geq w_i\\ m(i+1,j) & 0\leq j<w_i\end{cases}$$

可知每当j增大超过一个物品的重量w_i时，函数m(i, j)便有可能增加v_i，因而形成了关于变量j的阶梯状单调不减函数。

- 显然，对于j为连续的，w_i为实数时，m(i, j)依然是梯状单调不减函数。

阶跃函数m(i, j)的跳跃点

- 实际上在m(i, j)的计算中并不需要将j逐渐加1。因为m(i, j)有阶跃性。这样的点称为函数m(i, j)的跳跃点，可以表示为（j，m(i,j)）。



跳跃点决定了函数m(i, j)

- 在m(i, j)的计算中，只需要计算它的跳跃点。
- 在变量j是连续的情况下，可以对每个确定的i (1≤i≤n)，用一个表P[i]来保存m(i, j)的全部跳跃点。
- 对于每一个确定的实数 j，可以通过查表P[i]来确定m(i, j)的值。
- 由于m(i, j)是单调不减的阶梯函数，所以P[i]中的全部跳跃点的值也是递增排列的。

跳跃点的递推关系

- 跳跃点的递归计算式的推导
  - 初始时p[n+1]={(0，0)}。
  - 表p[i]可根据计算m(i，j)的递归式递归地由表p[i+1]计算

物重为0,价值为0

- **函数m(i,j)= max{m(i+1, j), m(i+1, j–w_i)+v_i}**
  - **函数m(i,j)的全部跳跃点**

$$p[i] = p[i+1] \cup q[i+1]$$

p[i+1]=m(i+1,j)的跳跃点集

q[i+1]=m(i+1, j–w_i)+v_i的跳跃点集

满足条件

- $(s,t)\in q[i+1]$当且仅当$w_i\le s\le c$且$(s-w_i,t-v_i)\in p[i+1]$。因此，容易由p[i+1]确定跳跃点集q[i+1]，
  - $q[i+1]=p[i+1]\oplus(w_i,v_i)$
    $=\{(j+w_i,m(i,j)+v_i)|(j,m(i,j))\in p[i+1]\}$

例子：n=5，c=10，w={2，2，6，5，4}，v={6，3，5，4，6},计算p[i]

i=6 p[6]={(0,0)}　　$q[6]=p[6]\oplus(w_5,v_5)=\{(4,6)\}$

i=5 p[5]={(0,0), (4,6)}　　q[5]={(5,4),(9,10)}

i=4 p[4]={(0,0), (4,6), (5,4), (9,10)}

不是真正的跳跃点

受控跳跃点

- 前面产生的跳跃点中有些可能并不是真正的跳跃点，而是所谓的受控跳跃点。
- 设(a, b)和(d, e)都是P[i]中的跳跃点，若a≤d且e≤b，其情形如下图所示：

显然(d, e)不是真正的跳跃点。

- (d, e)实际上受控于(a, b)，被称为受控跳跃点。
- 因此，将前面产生的跳跃点中的受控跳跃点删除，便可得到P[i]中的跳跃点。
- 由此便可得到计算表P[i]的算法。

- $p[i]= p[i+1] \cup q[i+1]$ –控制点
  - 在递归地由表p[i+1]计算表p[i]时
    - 先由p[i+1]计算出q[i+1]
    - 然后合并表p[i+1]和表q[i+1]
    - 清除其中的受控跳跃点得到表p[i]。
- $q[i+1]=p[i+1]\oplus(w_i,v_i)$

  $=\{(j+w_i,m(i,j)+v_i)|(j,m(i,j))\in p[i+1]$
- $p[n+1]=\{(0，\ 0)\}$。

计算公式

$p[i]=p[i+1] \cup q[i+1];$

$q[i+1]=p[i+1] \oplus (w_i,v_i)=\{ (j+w_i, m(i,j)+v_i) \mid (j, m(i,j)) \in p[i+1] \}$

初始时$p[6]=\{(0,0)\}$，$(w_5,v_5)=(4,6)$。因此，

$q[6]=p[6] \oplus (w_5,v_5)=\{(4,6)\}$。

$p[5]=\{(0,0),(4,6)\}$。

$q[5]=p[5] \oplus (w_4,v_4)=\{(5,4),(9,10)\}$。从跳跃点集$p[5]$与$q[5]$的并集$p[5] \cup q[5]=\{(0,0),(4,6),(5,4),(9,10)\}$中看到跳跃点(5,4)受控于跳跃点(4,6)。

n=5, c=10,w={2, 2, 6, 5, 4}，v={6, 3, 5, 4, 6}。
可写成 { (2, 6), (2, 3), (6, 5), (5, 4), (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)}

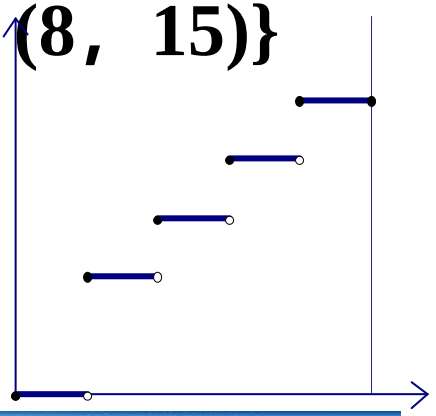
p[2]={ (0, 0)，(2, 3)，(4, 6)，(6, 9)，(9, 10)，(10, 11) }

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。



- ✓上述算法的主要计算量在于计算跳跃点集p[i] (1≤i≤n)。
- ✓分析：
  - ✓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,…,x_n的0/1赋值。

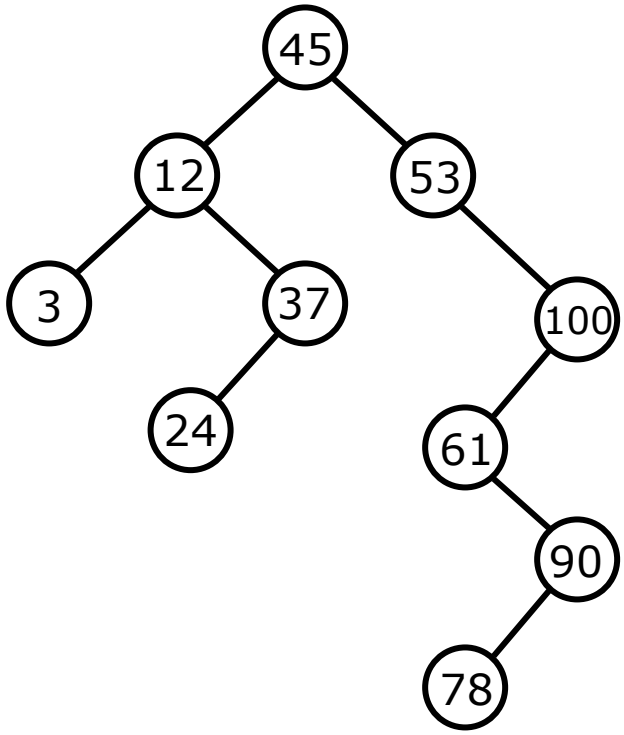
✓因此，p[i]中跳跃点个数不超过2^{n-i+1}。由此可见，算法计算跳跃点集p[i]所花费的计算时间为 $O\left(\sum_{i=2}^{n}|p[i+1]|\right)=O\left(\sum_{i=2}^{n}2^{n-i}\right)=O(2^n)$

✓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$w_i(1\leqslant i\leqslant n)$是整数时，$|p[i]|\leqslant c+1$，$(1\leqslant i\leqslant n)$。

✓在这种情况下，改进后算法的计算时间复杂性为$O(\min\{nc,2^n\})$。

3.11最优二叉搜索树



- 什么是二叉搜索树？

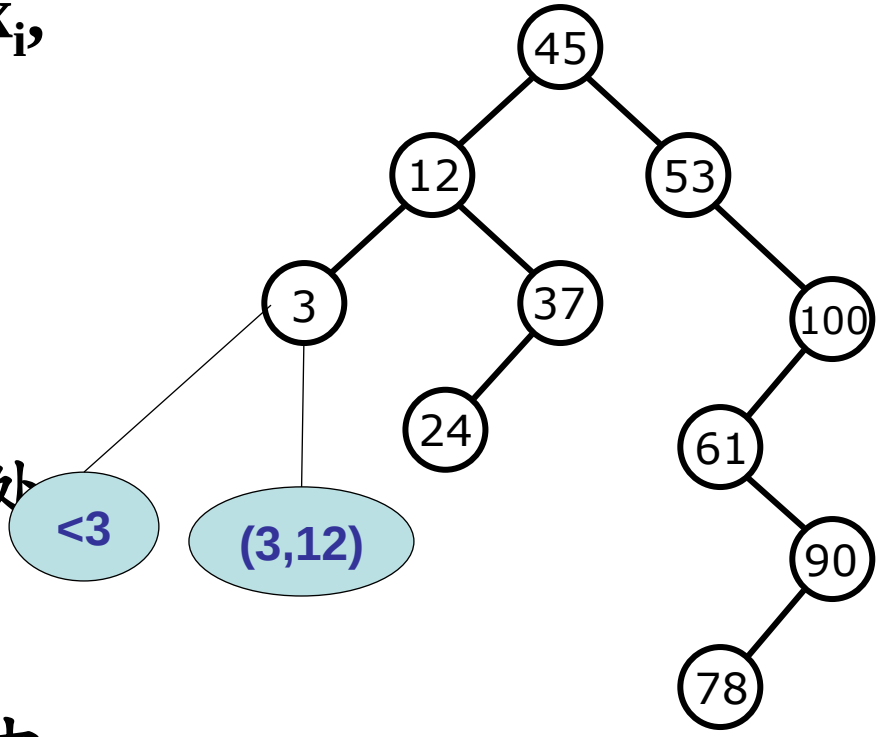
（1）若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值；

（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

二叉树中的叶顶点是形如(x_i, x_{i+1})的开区间。在二叉搜索树中搜索一个元素x，返回的结果为：

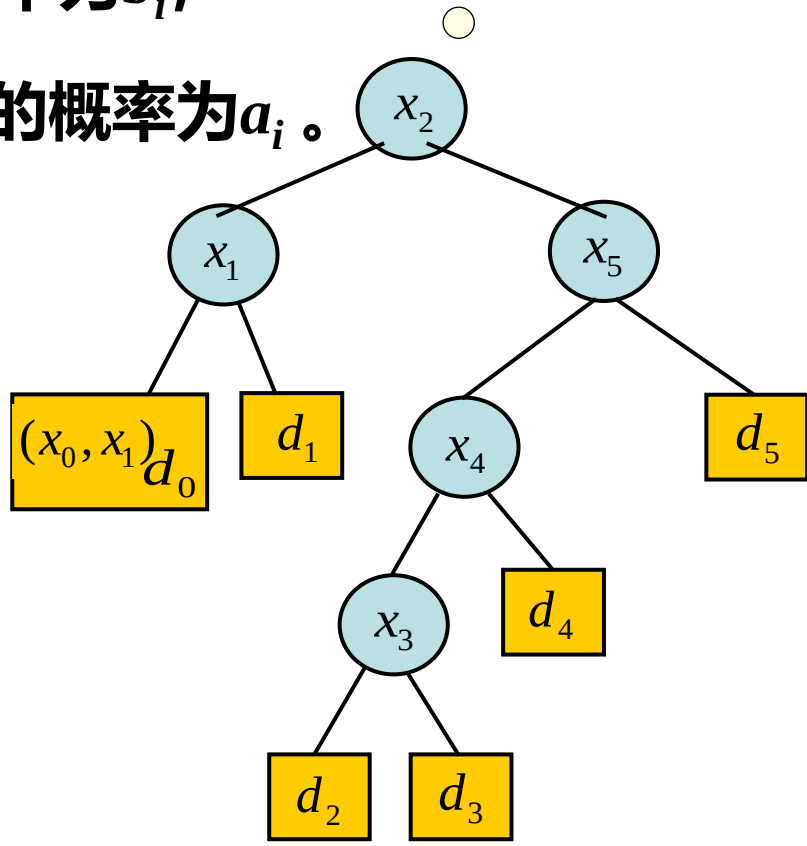


- （1）二叉树的内部顶点处找到：$x=x_i$;
- （2）在二叉树的叶顶点中确定：$x \in (x_i, x_{i+1})$.

伪关键字

- 假定在（1）情况出现(即$x=x_i$)的概率为b_i；
- 在（2）情况出现，即$x \in (x_i, x_{i+1})$.的概率为a_i。
- 这里约定$x_0=-\infty$, $x_{n+1}=+\infty$.
- 显然 $a_i \geq 0$，$0 \leq i \leq n$，$b_j \geq 0$; $1 \leq j \leq n$,

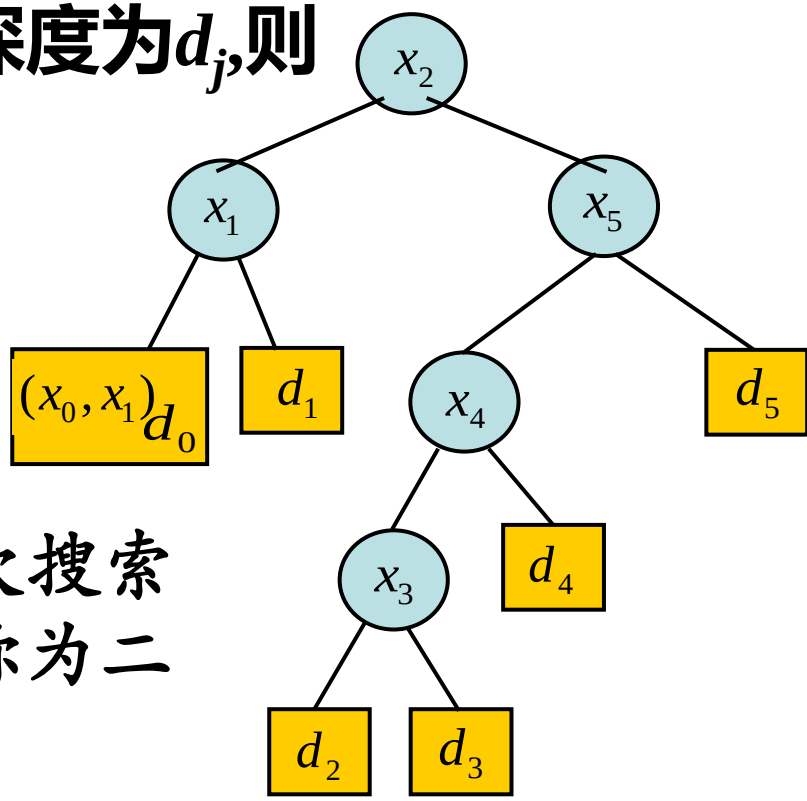
$$\sum_{i=0}^{n} a_i + \sum_{j=1}^{n} b_j = 1$$



$(a_0,b_1,a_1,\ldots\ldots b_n,a_n)$称为集合S的**存取概率分布**

在表示S的二叉搜索树T中，设存储元素 x_i 结点深度是c_i,叶结点（x_j, x_{j+1})的结点深度为d_j,则

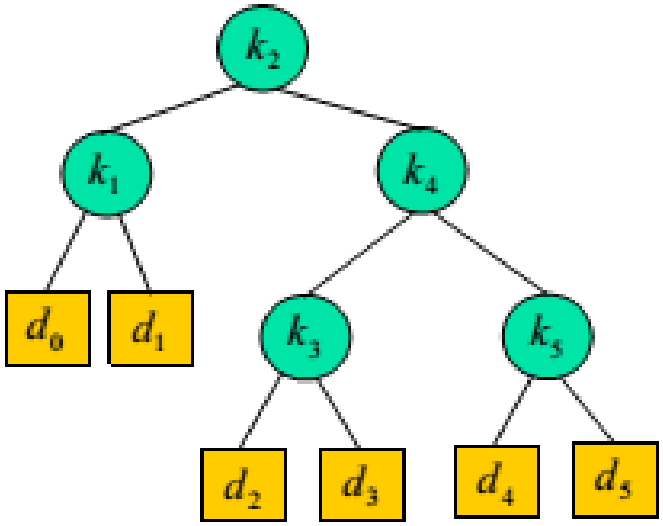
$$p = \sum_{i=1}^{n} b_i(1+c_i) + \sum_{j=0}^{n} a_j d_j$$



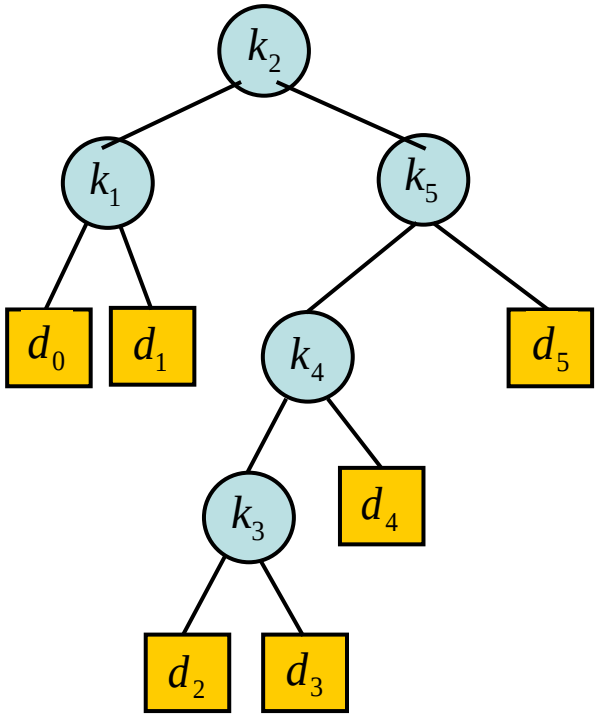
- 表示在二叉搜索树T中作一次搜索所需的平均比较次数。P又称为二叉搜索树T的平均路长。
- 在一般情况下，不同的二叉搜索树的平均路长是不同的。

最优二叉搜索树问题：

- 对于有序集S及其存取概率分布（$a_0, b_1, a_1, \cdots, b_n, a_n$），在所有表示有序集$S$的二叉搜索树中找出一棵具有期望搜索代价最小的二叉搜索树。



结点	深度c_i	概率a_i / b_i	贡献
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
合计			2.80



结点	深度	概率	贡献
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
合计			2.75

1、最优子结构性质

- 二叉搜索树T 的一棵含有顶点$x_i, \cdots, x_j$和叶顶点$(x_{i-1}, x_i), \cdots, (x_j, x_{j+1})$的子树可以看作是有序集$\{x_i, \cdots, x_j\}$关于全集为$\{x_{i-1}, x_{j+1}\}$的一棵二叉搜索树（T 自身可以看作是有序集)，其存取概率为

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

$w_{i,j}$是概率之和，$w_{i,j} = a_{i-1} + b_i + \ldots\ldots + b_j + a_j$

设Tij是有序集$\{x_i, \cdots, x_j\}$关于存储概率分布为$\{\overline{a_{i-1}}, \overline{b_i}, \ldots, \overline{b_j}, \overline{a_j}\}$的一棵最优二叉搜索树

- P_{ij}：平均路长；
- x_m：T_{ij}的根顶点存储的元素
- p_l和p_r：左子树T_l和右子树T_r的平均路长

$$w_{i,j}p_{i,j} = w_{i,j} + w_{i,m-1}p_l + w_{m+1,j}p_r \qquad i \le j$$

期望搜索代价

左子树的搜索概率

右子树的搜索概率

构造最优二叉搜索树时，可以选择先构造其左右子树，使其左右子树最优，然后构造整棵树

2、递归计算最优值

- 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$p_{1,n}$。根据二叉树的期望代价公式

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

  - 初始时$p_{i,i-1}=0$，$w_{1,n}=1$

- 记$m(i,j)=w_{i,j}p_{i,j}$，则$m(1,n)=w_{1,n}p_{1,n}=p_{1,n}$

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

$$m(i,i-1) = 0, \quad i = 1,2,\cdots,n$$

空子树

```
void OptimalBinarySearchTree ( int a, int b,int n, int **m, int **s,
int **w)
{    for( int i=0; i<=n; i++){  w[i+1][i]=a[i];   m[i+1][i]=0; }
      //初始化，构造没有内部节点时的情况
     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];
        }}
```

初始化空子树对角线赋值

S[i][j]中存放最优子树T(i,j)的根节点

取第k个结点作根

3、构造最优解

- 算法中OptimailBinarySearchTree中s[i][j]保存最优子树T(i, j)的根结点中元素。
  - s[1][n]=k时，x_k为所求二叉搜索树根结点
  - 左子树为T(1,k-1)
- i=s[1][k-1]表示T(1,k-1)根结点元素为x_i
- 依次类推，由s构造出的最优二叉搜索树时间复杂度为O(n)。

4、计算复杂性

- 算法中使用了三个数组，因此所需的空间为$O(n^2)$。
- 算法的主要计算量在于计算

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

- 对于固定r,它所需要的计算时间为$O(j-i+1)=O(r+1)$
- 因此，算法所耗费的总时间为$O(n^3)$。

课后作业

- P78 算法分析题 3-1,3-3,3-4
- 提交时间:11.21