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?不妨设w_i, v_i, C都是正整数

0-1背包问题是一个特殊的整数规划问题。

目标函数 $$\max \sum_{i=1}^{n} v_i x_i$$

约束条件 $$\begin{cases} \sum_{i=1}^{n} w_i x_i \le \mathrm{c} \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

0-1背包问题

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

最优子结构性质：

设（y1,.., yn）是所给0-1背包问题的一个最优解，则（y2, .., yn）是下面相应子问题的一个最优解：

$$\max \sum_{i=2}^{n} v_i x_i \qquad \begin{cases} \sum_{i=2}^{n} w_i x_1 \le c - w1y1 \\ x_i \in \{0,1\} \quad\quad 1 \le i \le n \end{cases}$$

否则，设（z2，…，zn）是上述子问题的一个最优解，而（y2, .., yn）不是它的最优解。由此可知$\sum_{i=2}^{n} v_i z_i > \sum_{i=2}^{n} v_i y_i$，且$w_1 y_1 + \sum_{i=2}^{n} w_i z_i \le c$。因此

$$v_1 y_1 + \sum_{i=2}^{n} v_i z_i > \sum_{i=l}^{n} v_i y_i \qquad\qquad w_1 y_1 + \sum_{i=2}^{n} w_i z_i \le c$$

这说明(y1, z2, …, zn)是所给0-1背包问题的一个更优解，从而(y1, …, yn)不是最优解。此为矛盾。

0-1背包问题

设0-1背包问题的子问题 $\max \sum_{k=i}^{n} v_k x_k$ $\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, i \le k \le n \end{cases}$

的最优值为m(i，j)。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。

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

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

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

```
void Knapsack(int v[], int w[], int c int n,  int[][] m){
        int jMax = min(w[n]-1, c);
        for(int j = 0; j <= jMax; j++)
         m[n][j] = 0;
         for(int j = w[n]; j <= c; j++)
         m[n][j] = v[n];
          for(int i = n-1; i>1; i++)
                jMax = min(w[i]-1, c);
          for(int j = 0; j <= jMax; j++)
                  m[i][j] = m[i+1][j];
                for(int j = w[n]; j <= c; j++)
                 m[i][j] = max(m[i+1][j], m[i+1][j-w[i]]+v[i]);
```

算法实现

Knapsack 续

```
        m[1][c] = m[2][c]; \\当i=1时，只用计算j=c的情况
                if(c>=w[1])
                    m[1][c] = max(m[1][c], m[2][c-w[1]]+v[1];
}
void Traceback(int m[ ][ ], int w[ ], int c, int n, int x[ ]){
        for(int i=1;i<n;i++)
                if(m[i][c] == m[i+1][c])
                        x[i] = 0;
                else
                        x[i] = 1;
                        c-=w[i];
                x[n] = (m[n][c])?1:0
}
```

用动态规划法求最优值

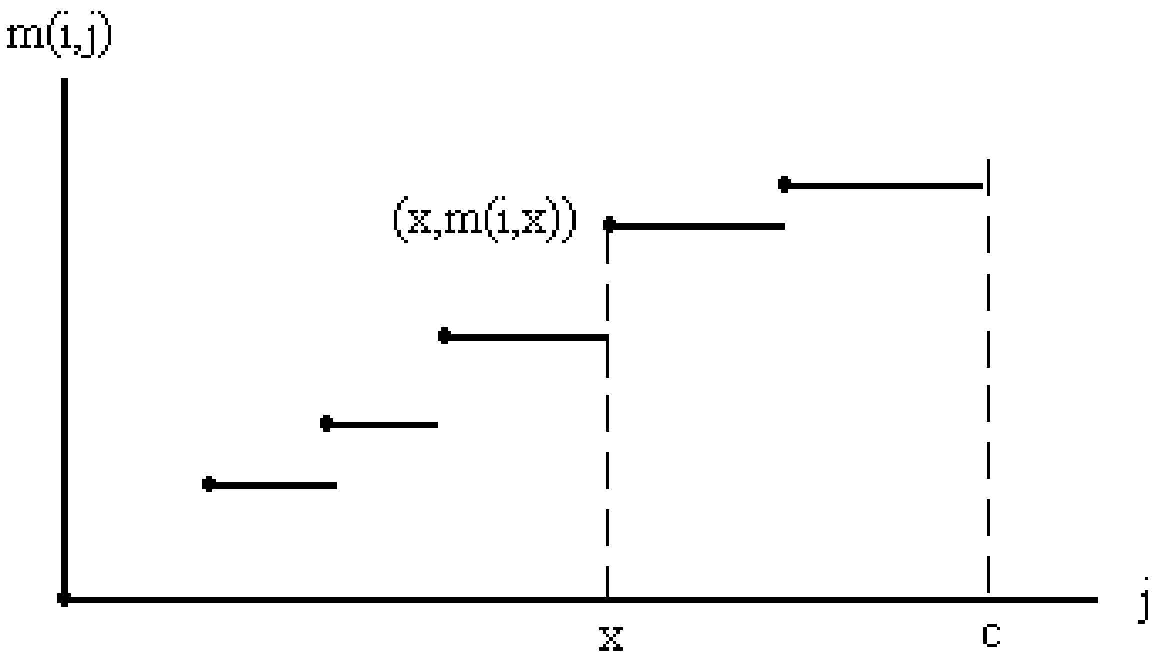
$$m(i,j)=\begin{cases}\max\{m(i+1,j),m(i+1,j-w_i)+v_i\} & j\ge w_i\\ m(i+1,j) & 0\le j<w_i\end{cases}$$

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。

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。



横坐标j是背包容量，纵坐标m(i, j)是最优价值。

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

- 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(j, m(i, j))。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)}。
- 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]⊕(wi,vi)={(j+wi,m(i,j)+vi)|(j,m(i,j))∈p[i+1]}

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

- 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。

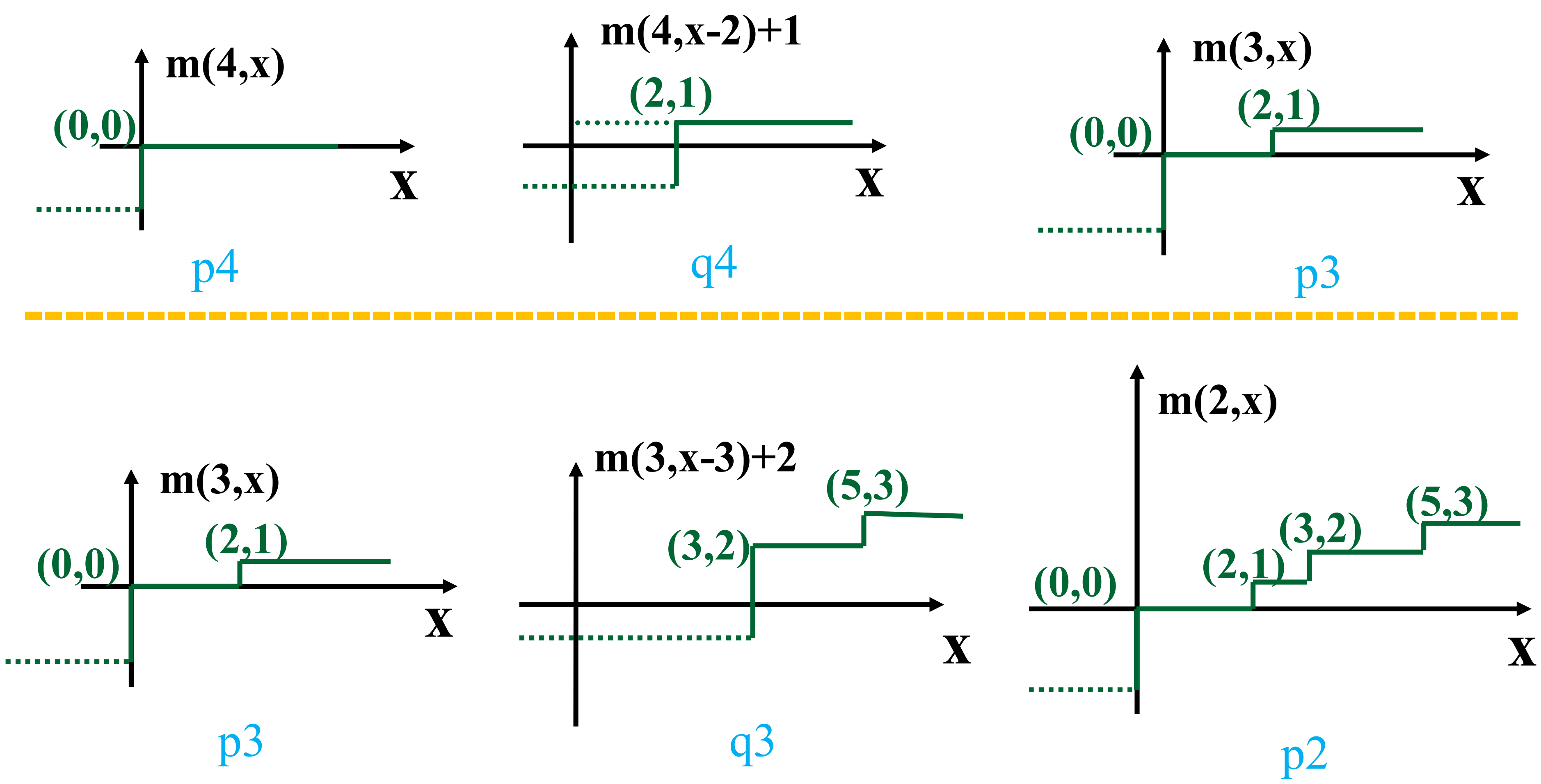
由此可见，在递归地由表p[i+1]计算表p[i]时，

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]。

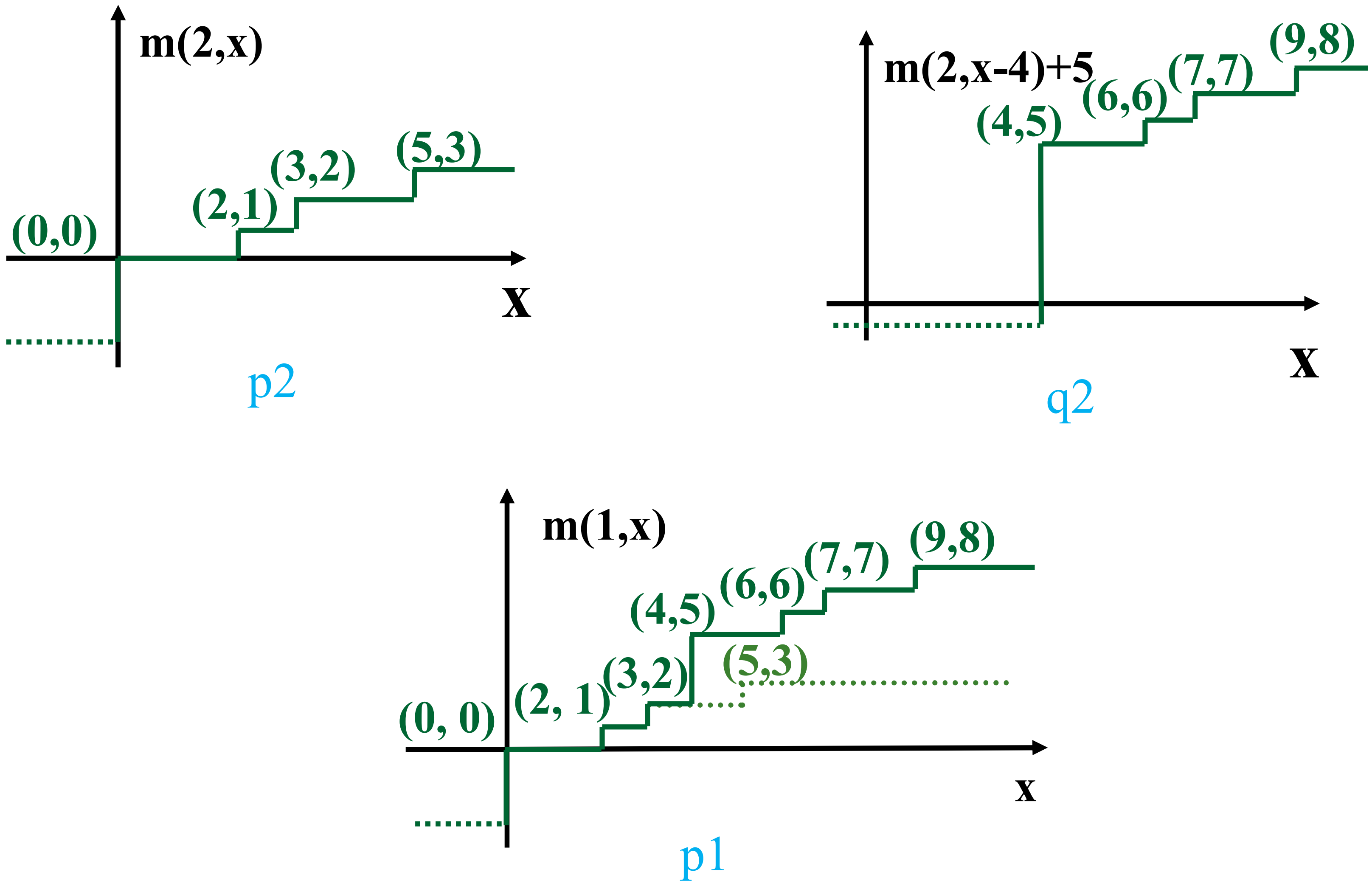
典型例子1

n=3，c=6，w={4，3，2}，v={5，2，1}。



典型例子1

n=3，c=6，w={4，3，2}，v={5，2，1}。



典型例子2

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)。从而，

q[6]=p[6]⊕(w5,v5)={(4,6)}。因此

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)}。

典型例子2

n=5，c=10，w={2，2，6，5，4}，v={6，3，5，4，6}

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]($1\le i\le n$)。由于q[i+1]=p[i+1]$\oplus$(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]|)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。故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

$$O\left(\sum_{i=2}^{n}|\,p[i+1]\,|\right)=O\left(\sum_{i=2}^{n}2^{n-i}\right)=O\left(2^{n}\right)$$

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

算法实现

Knapsack2代码的整体思路是依次生成i，i=n...1，的跳跃点。在从i的跳跃点生成i-1的跳跃点时，分别扫描处理p[i]和q[i]。p[i]一边扫描一边判断是否受控，不受控放入p[i-1]。q[i]一边生成一边判断是否受控，不受控放入p[i-1]。

```
int Knapsack2(int n, int c, int v[], int w[], int[][] p) {
        int[] head = new int[n + 2]; //指向物品i的跳跃点的集合起始位置
        head[n + 1] = 0;
        p[0][0] = 0; //p[j][0]和p[j][1]分别存储跳跃点的s和t
        p[0][1] = 0;
        int left = 0, right = 0, next = 1; //left和right记录第i个物品跳跃点
的左右边界
        head[n] = 1;                      //next指向当前处理的跳跃点
```

算法实现

Knapsack2 续

```
for (int i = n; i >= 1; i--) //for 1
        int k = left;
        for (int j = left; j <= right; j++) //for 2
              if (p[j][0] + w[i] > c)
                     break;
              int y = p[j][0] + w[i];
              int m = p[j][1] + v[i];
              while (k <= right && p[k][0] < y)
                     p[next][0] = p[k][0];
                     p[next++][1] = p[k++][1];
              if (k <= right && p[k][0] == y)
                     if (m < p[k][1])
                            m = p[k][1];
                     k++;
```

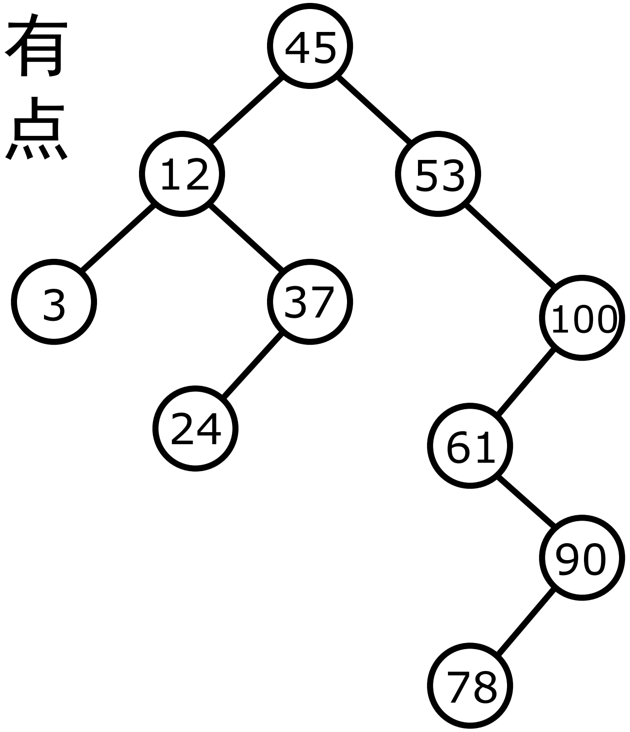
算法实现

Knapsack2 续

```
            if (m > p[next - 1][1])
                    p[next][0] = y;
                    p[next++][1] = m;
             while (k <= right && p[k][1] <= p[next - 1][1])
                     k++;
             } //end for 2
             while (k <= right)
                     p[next][0] = p[k][0];
                     p[next++][1] = p[k++][1];
              left = right + 1;
              right = next - 1;
              head[i - 1] = next;
        } //end for 1
        Traceback(n, w, v, p, head);
        return p[next-1][1];
}
```

最优二叉搜索树

- 设S={x1, x2, . . . , xn}是有序集。表示有序集S的二叉搜索树利用二叉树的结点存储有序集中的元素。



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；

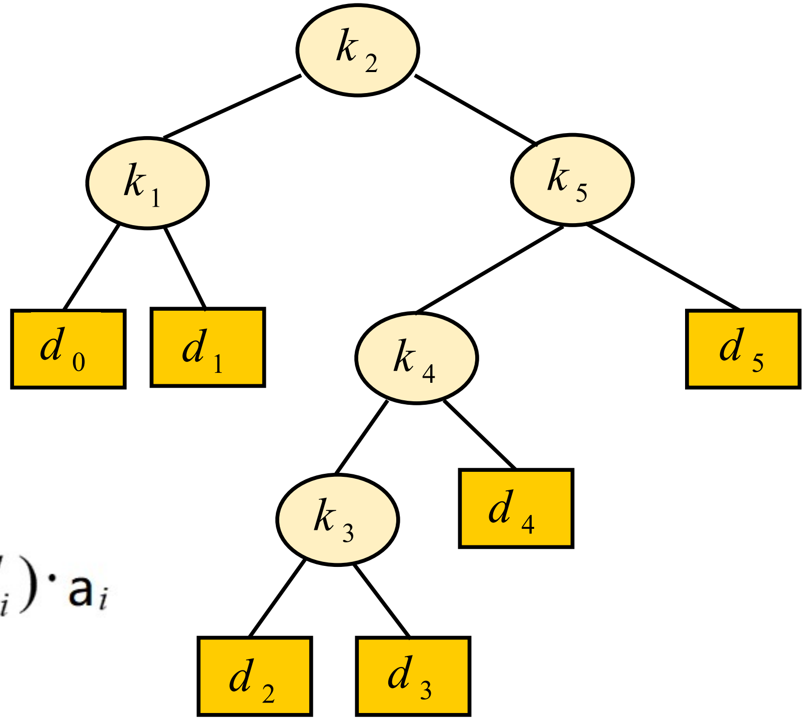
3）它的左、右子树也分别为二叉搜索树。

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

- 在二叉查找树中搜索一个元素x
  - ① 查找成功：在内结点中找到x=xi的概率为bi；
  - ② 查找失败：在叶结点中找到x=xi的概率为ai

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

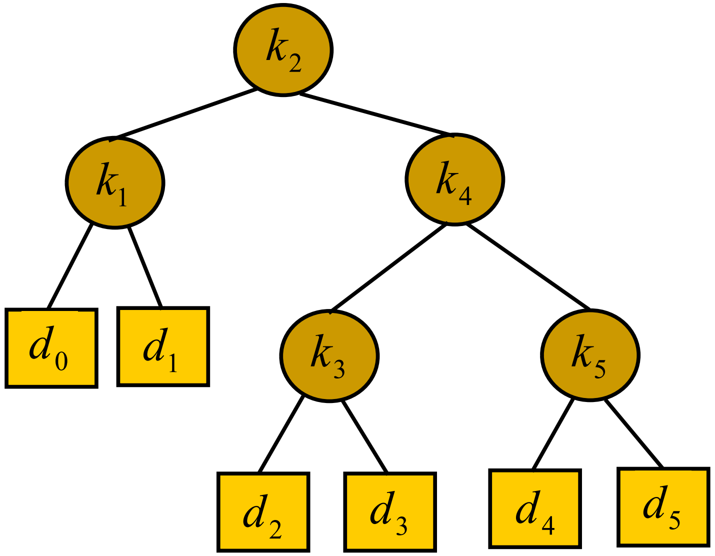


- 二叉查找树的期望耗费

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

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

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	0.10
d_1	2	0.10	0.20
d_2	3	0.05	0.15
d_3	3	0.05	0.15
d_4	3	0.05	0.15
d_5	3	0.10	0.30
Total			2.40

$$E(\text{search cost in } T)$$
$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot \mathrm{b}_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot \mathrm{a}_i$$

最优二叉搜索树

- 设S={x1, x2, …, xn}是有序集。在表示S的二叉搜索树T中，设存储xi的结点深度为c_i，叶节点（x_j，x_{j+1})的结点深度为d_j，则T的平均路长p=$\sum_{i=1}^{n} b_i(1+c_i)+\sum_{j=0}^{n} a_j d_j$。
- 最优二叉搜索树问题是对于有序集S及其存取概率分布(a0, b1, a1, …, bn, an)在所有表示S的二叉搜索树中找出一棵具有最小平均路长的二叉搜索树。
- 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

最优二叉搜索树

设最优二叉搜索树T_{ij}的平均路长为p_{ij}，初始时$p_{i,i-1}=0$；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}的递归式为：$w_{i,j}p_{i,j} = w_{i,j} + \min\limits_{i \le k \le j}\{w_{i,k-1}p_{i,k-1} + w_{k+1,j}p_{k+1,j}\}$

其中$w(i,j) = a_{i-1}+b_i+a_i+\ldots+b_j+a_j$

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:

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

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

算法实现

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

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

```
void OptimalTree(int a[ ], int b[ ], int n, int[][] m, int s[ ][ ],
int w[ ][ ]) {
       for(int i=0;i<=n;i++) //初始化构造无内部节点的情况
       w[i+1][i] = a[i];     m[i+1][i] = 0;
       for(int r=0; r<n; r++)
        for(int i=1; i<=n-r; i++)
              int j = i+r; //i, j之间距离为r时,首选i为根,其左子树为空,右子树为节点
              w[i][j] = w[i][j-1] + a[j] + b[j]; //当k=i时
              m[i][j] = m[i+1][j]; //实际还加了m[i][i-1]=0
              s[i][j] = i;
              for(int k=i+1;k<=j;k++) //遍历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];
}
```

最优二叉搜索树

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

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

上述算法时间复杂度为:

$$\sum_{r=0}^{n-1}\sum_{i=1}^{n-r} O(r+1) = O(n^3)$$

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(n²)的算法