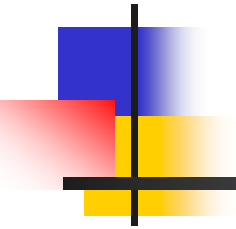


算法设计与分析

第3章　动态规划
(2)

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。

学习要点

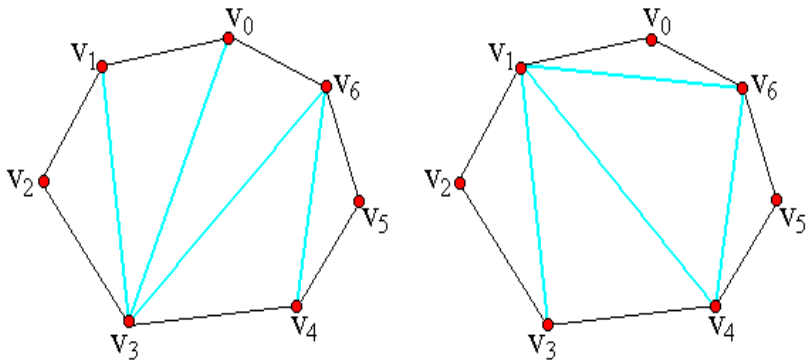
- 通过应用范例学习动态规划算法设计策略。
  （1）矩阵连乘问题;
  （2）最长公共子序列;
  （3）凸多边形最优三角剖分;
  （4）0/1背包问题;
  （5）最优二叉搜索树。

凸多边形最优三角剖分

- 应用
  - Catalan数
  - 多边形三角剖分是数字城市研究许多工作的前提
    - 城市景观三维重建中的三角剖分算法
  - 基于图像特征和三角剖分的水印算法
  - 基于三角剖分的小脑模型在增强学习中的应用
  - 传感网中的动态Delauanay三角剖分算法
  - ……

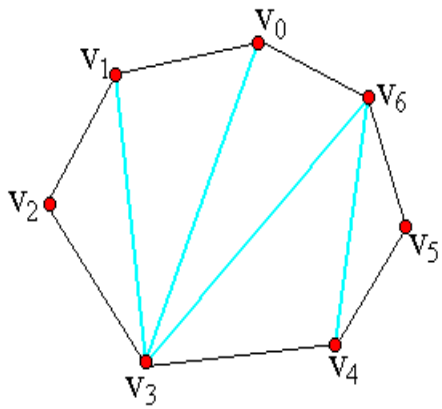
凸多边形最优三角剖分

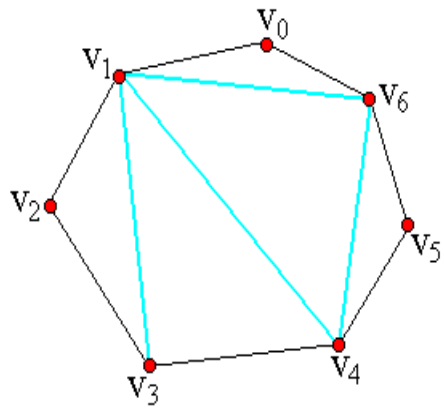
- 多边形的三角剖分是将多边形分割成互不相交的三角形的弦的集合T。
  - 输入:给定凸多边形P，以及定义在由多边形的边和弦组成的三角形上的权函数w。
  - 输出:要求确定该凸多边形的三角剖分，使得即该三角剖分中诸三角形上权之和为最小。



凸多边形最优三角剖分

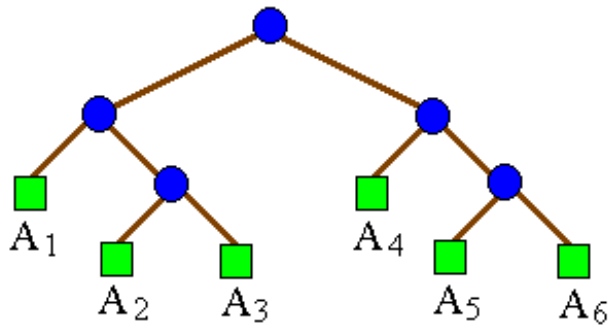
- 用多边形顶点的逆时针序列表示凸多边形，即$P=\{v_0,v_1,\ldots,v_{n-1}\}$表示具有n条边的凸多边形。
- 若v_i与v_j是多边形上不相邻的2个顶点，则线段v_iv_j称为多边形的一条弦。
  - 弦将多边形分割成2个多边形$\{v_i,v_{i+1},\ldots,v_j\}$和$\{v_j,v_{j+1},\ldots v_i\}$。





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。
  - 叶结点: 表达式中一个原子
- 例如，完全加括号的矩阵连乘积 $((A_1(A_2A_3))(A_4(A_5A_6)))$ 相应的语法树为
  - 叶结点: 一个矩阵

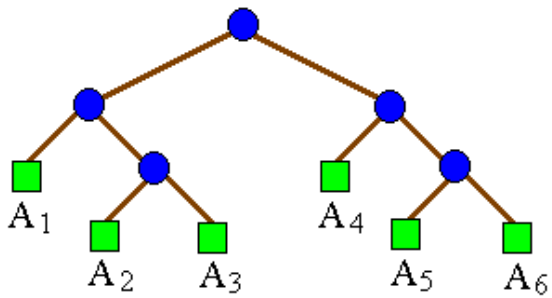


(a)

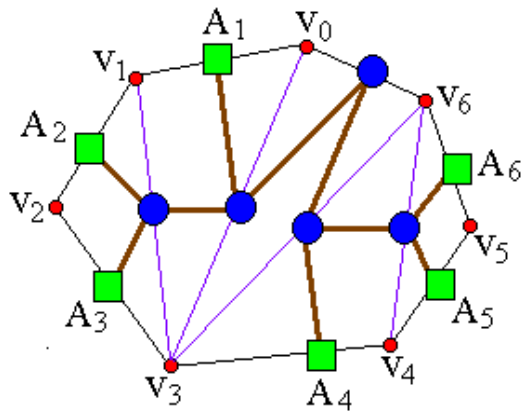
三角剖分的结构及其相关问题

- 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,如图。
  - 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
- 与矩阵连乘积问题的比较(P59)
  - 每个矩阵A_i对应于凸(n+1)边形中的一条边$v_{i-1}v_i$
  - 矩阵连乘积A[i+1:j]对应于三角剖分中的一条弦v_iv_j，$i<j$

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



(a)

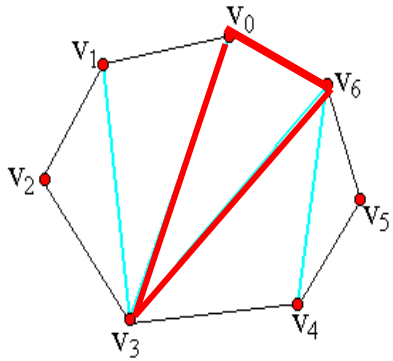


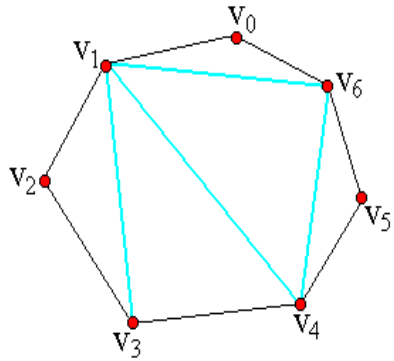
(b)

步骤1：最优子结构性质

- 最优三角剖分问题
  - 输入：多边形P 和代价函数W
  - 输出：求P 的三角剖分T，使得代价 $\Sigma s \in STW(s)$最小，其中ST 是T 所对应的三角形集合

步骤1：最优子





- 分析
  - 若凸(n+1)边形P={v_0,v_1,…,v_n}的最优三角剖分T包含三角形$v_0v_kv_n$，1≤k≤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,…,v_k}和{v_k,v_{k+1},…,v_n}的权之和。
  - 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,…,v_k}或{v_k,v_{k+1},…,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。

步骤2：最优三角剖分的递归结构

- t[i][j] (1⩽i<j⩽n) 为凸子多边形{v_{i-1},v_i,…,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]。

只有一条线段

➢ 当j-i⩾1时，凸子多边形至少有3个顶点(v_{i-1},v_i,v_j)。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，而k的所有可能位置只有j-i个，由此，t[i][j]可递归地定义为：

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

0-1背包问题

- 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。
  - 处理机和数据库在分布式计算机系统上的分配问题
  - 项目选择的货物装载问题
  - 削减库存问题等

0-1背包问题

- 研究
  - 在量子计算机上求解0/1背包问题
    - 计算机学报，中国科学技术大学计算机科学与技术系国家高性能计算中心，1999，12：1314～1316
  - 二次背包问题的一种快速解法
    - 计算机学报，国防科学技术大学计算机学院 长沙 2004，9：1162 ～ 1169
  - 多维背包问题的一个蚁群优化算法
    - 计算机学报，哈尔滨工业大学计算机科学与技术学院 ，2008，5：810 ～ 819

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?

✓ 输入：C>0，w_i>0，v_i>0，1≤ i≤n

✓ 输出：(x_1，x_2，…，x_n)，x_i∈{0，1}，满足

$$\max \sum_{i=1}^{n} v_i x_i \ , \ \sum_{i=1}^{n} w_i x_i \le C$$

$$x_i \in \{0,1\}, 1 \le i \le n$$

0-1背包问题 — ①最优子结构性质

- 0-1背包问题具有最优子结构性质
  - 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

0-1背包问题 — ②递归关系

- 设所给0-1背包问题的子问题

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

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。

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

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

通过m(i+1，x)来计算m(i，x)

0-1背包问题

- 算法复杂度分析:
  - 动态规划的本质是把所有前面已知的结果建成一个大表格,表格是迭代构造的.

n个物品
C最大容量

  - 从m(i，j)的递归式容易看出，表格有nc项，每一项由其他两项在常数式时间内计算得到.算法需要O(nc)计算时间。
    - 要求物品重量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。

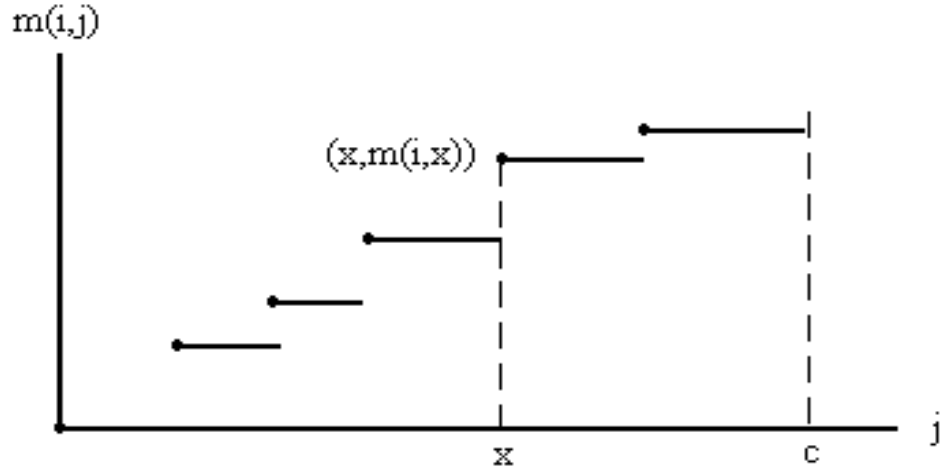
0-1背包问题－算法改进

- 例
  - n =5, c=10, w={2,2,6,5,4}, v={6,3,5,4,6}
  - 当i=5时

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

  - 当i=4时?

0-1背包问题－算法改进



- 分析:
  - 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。
    - 表示：跳跃点坐标(s,t)=(j, m(i,j))
  - 在一般情况下，函数m(i,j)由其全部跳跃点唯一确定。
  - j是连续变量时，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。

i：第i个可选择物品

0-1背包问题－算法改进

- 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

0-1背包问题 — 算法改进

- 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] ∪ q[i+1]
  - p[i+1]是函数m(i+1, j)的跳跃点集
  - q[i+1]是函数m(i+1, j-w_i)+v_i的跳跃点集
- (s,t)∈q[i+1]当且仅当$w_i \le s \le c$且(s-w_i,t-v_i) ∈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]⊕(w_i,v_i)

    ={(j+w_i,m(i,j)+v_i)|(j,m(i,j))∈p[i+1]}
- 控制点
  - 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。

(容量, 最优值)

意义：装载量多，价值却少，肯定不是最优解

0-1背包问题－算法改进

- p[i]= p[i+1] ∪ 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)}。

0-1背包问题 — 举例

- **n=5，c=10，w={2，2，6，5，4}，v={6，3，5，4，6}。**
- **$q[i+1]=p[i+1]\oplus(w_i,v_i)=\{(j+w_i,m(i,j)+v_i)|(j,m(i,j))\in p[i+1]$**
- **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)**

① 初始，$p[6]=\{(0,0)\}$，$(w_5,v_5)=(4,6)$。
因此，$q[6]=p[6]\oplus(w_5,v_5)=\{(4,6)\}$。

② $p[5]=\{(0,0),(4,6)\}$，　$q[5]=p[5]\oplus(w_4,v_4)=\{(5,4),(9,10)\}$
$p[5]\cup q[5]=\{(0,0),(4,6),(5,4),(9,10)\}$
跳跃点(5,4)受控于跳跃点(4,6)，将受控跳跃点(5,4)清除

③ $p[4]=\{(0,0),(4,6),(9,10)\}$
$q[4]=p[4]\oplus(6,5)=\{(6,5),(10,11)\}$

0-1背包问题－举例

- **n=5，c=10，w={2，2，6，5，4}，v={6，3，5，4，6}。**
- **q[i+1]=p[i+1]⊕(wi,vi)={(j+wi,m(i,j)+vi)|(j,m(i,j))∈p[i+1]**
- **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)**

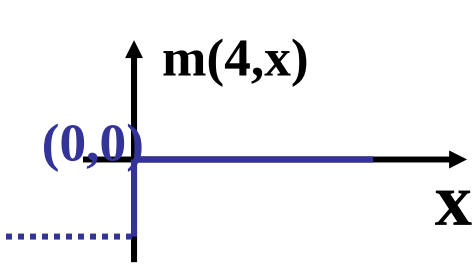
④ p[3]={(0，0)，(4，6)，(9，10)，(10，11)}
q[3]=p[3]⊕(2，3)={(2，3)，(6，9)}

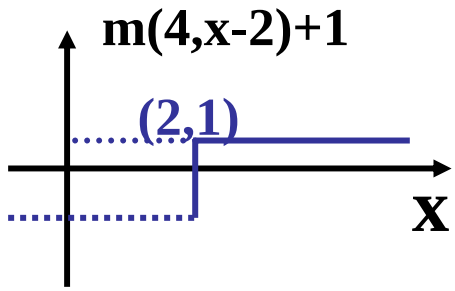
⑤ p[2]={(0, 0)，(2, 3)，(4, 6)，(6, 9)，(9, 10)，(10, 11)}
q[2]=p[2]⊕(2，6)={(2，6)，(4，9)，(6，12)，(8，15)}

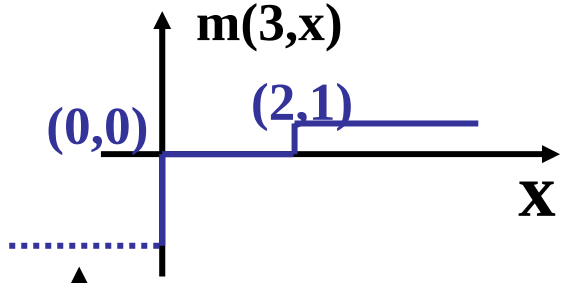
⑥ p[1]={(0，0)，(2，6)，(4，9)，(6，12)，**(8，15)**}

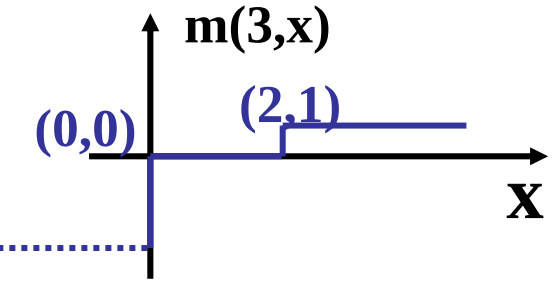
⑦ p[1]的最后的那个跳跃点(8,15)给出所求的最优值为m(1,c)=15。

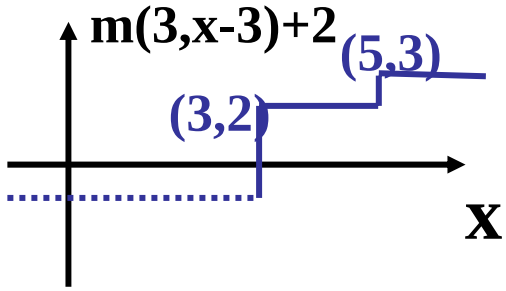
n=3，c=6，w={4，3，2}，v={5，2，1}。

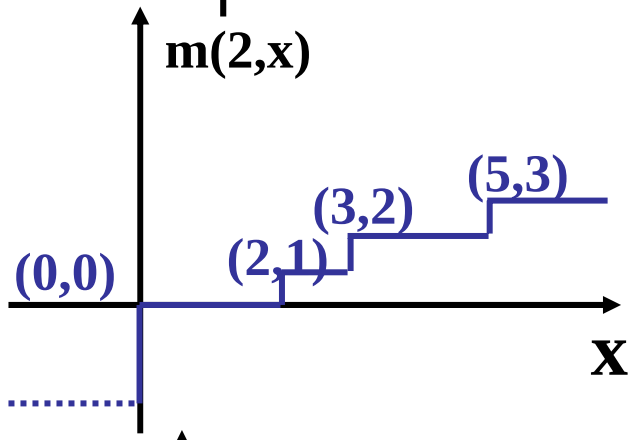


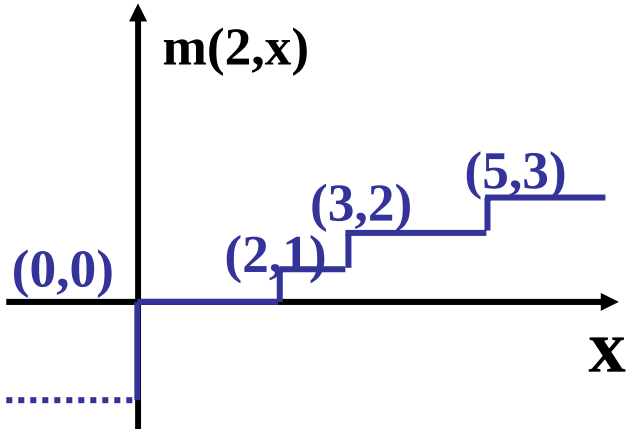


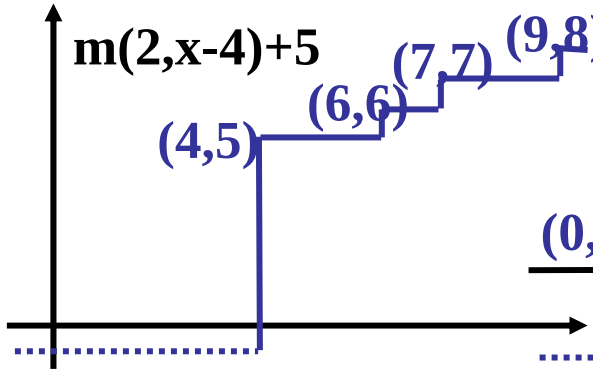


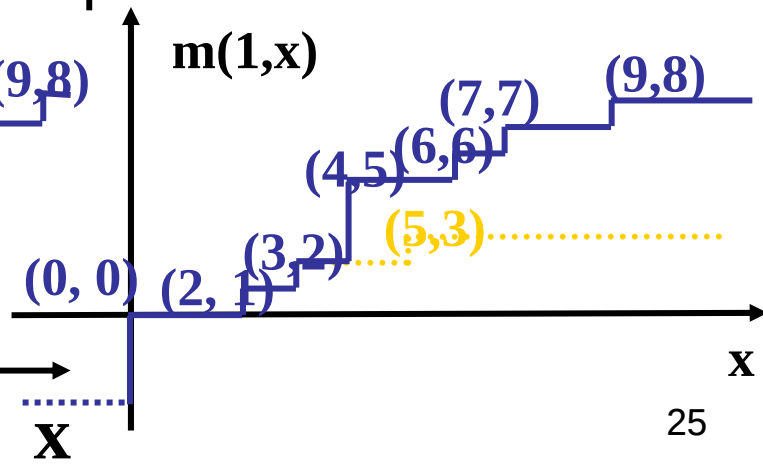












0-1背包问题－算法复杂度分析

- 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$x_i,\ldots,x_n$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

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

0-1背包问题－算法复杂度分析

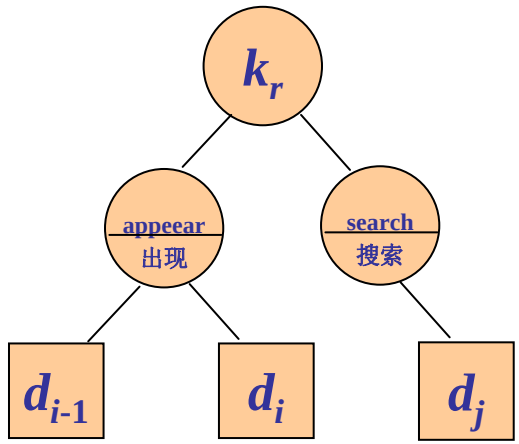
- 当所给物品的重量是整数时，（|p[i]|）$\leqslant c+1, 1 \leqslant i \leqslant n$)

- 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

- 设计程序将英文文档翻译成中文
  - 例如自动翻译机



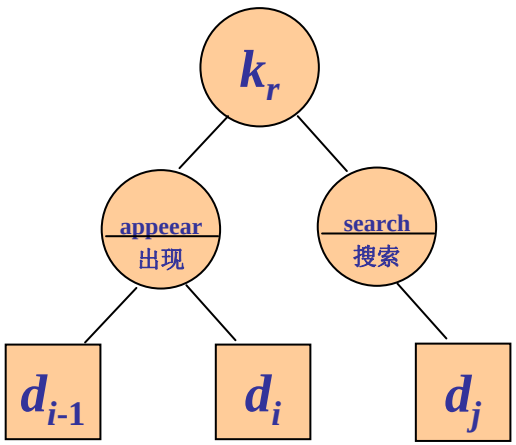
最优二叉搜索树

- 自动翻译机
  - 使用线性表操作每次运行需要O(n)搜索时间

生词表	
字段1	字段2
aggregate analysis	综合分析;总体分析
amortized	分期,分摊
arbitrary	任意的, 武断的, 独裁的, 专断的
auxiliary	辅助的, 补助的
binomial	二项的, 二项式的
bog	沼泽, 陷于泥沼
contemporary	当代的, 同时代的
convention	归约, 常规
crucial	至关紧要的
disjoint	不相交的,不相接的,分离的
distinct	清楚的, 明显的, 截然不同的, 独特的

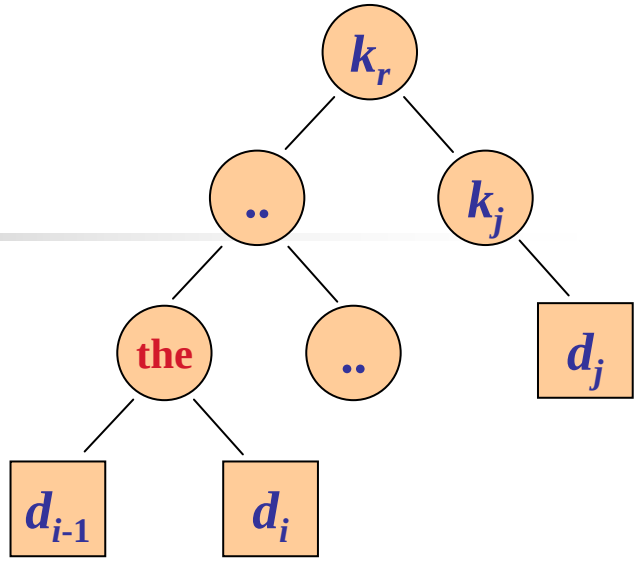
最优二叉搜索树

- 查找操作: build a BST with
  - n English words as keys
  - Chineses equivalents as satellite data （从属数据）
- 对于课文中出现的每个单词，都需要搜索该二叉树，如何使得总的搜索次数最少？
- 对于任何一个单词的搜索，使用二分搜索法的时间为O(lg n)
- However, Words appear with different frequencies…?



生词表	
字段1	字段2
aggregate analysis	综合分析;总体分析
amortized	分期,分摊
arbitrary	任意的, 武断的, 独裁的, 专断的
auxiliary	辅助的, 补助的
binomial	二项的, 二项式的
…	…

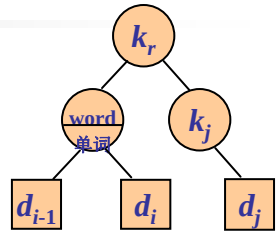
最优二叉搜索树



- 单词出现的频率并不同。

- 高频出现的单词出现在远离树根的节点，很少用的词在接近根的节点
  - “the” (frequently used) appears far from the root; mycophagist” (rarely used) appears near the root.
- 这种组织方式会减慢翻译速度, 因为：在二叉树中搜索 key 时，被访问的节点数 = 1+被搜索 key 的节点深度。
- 希望高频出现的单词接近树根

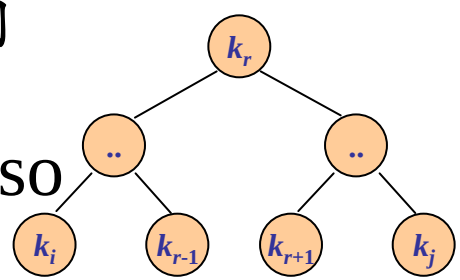
最优二叉搜索树



- 文中有些英语单词没有对应的汉语译文，即这些英语单词不出现在二叉搜索树中

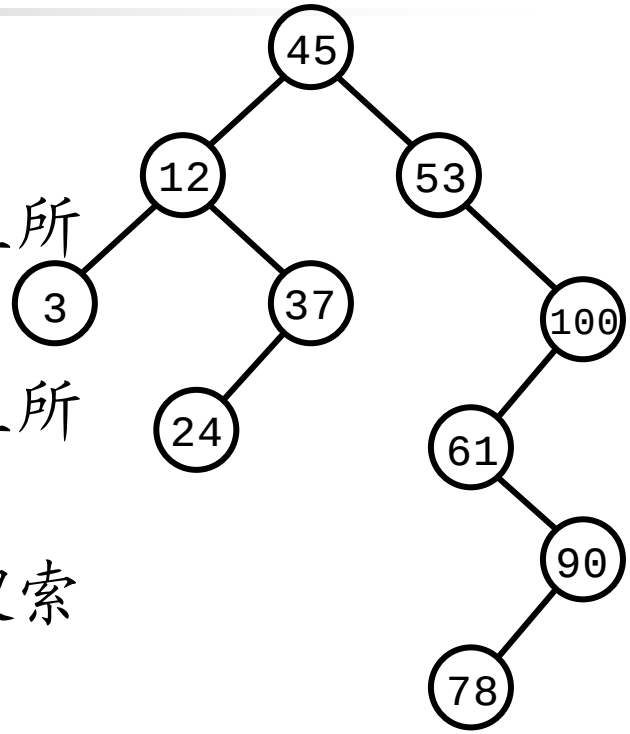
怎么办？

- 设已知每个单词出现的概率，如何组织一颗二叉搜索树，使得在所有搜索中，被访问的节点的总数最少？（ How do we organize a BST so as to minimize the number of nodes visited in all searches, given that we know how often each word occurs? ）



最优二叉搜索树

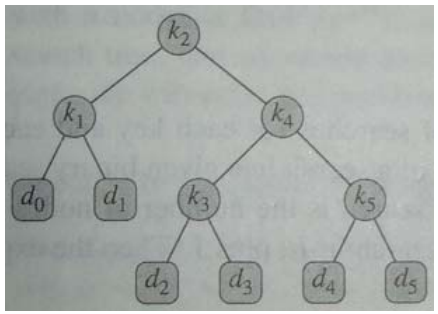
- 二叉搜索树
  - 1）若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值;
  - 2）若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值;
  - 3）它的左、右子树也分别为二叉搜索树
  - 4）二叉搜索树叶节点是形如（x_i,x_{i+1}）的开区间，表示为虚拟键d_i



➢在随机的情况下，二叉搜索树的平均查找长度和 logn 是等数量级的

最优二叉搜索树

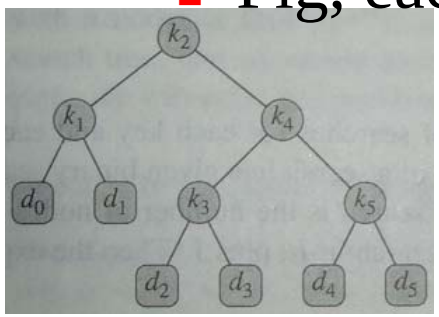
- S={x_1,x_2, …,x_n}是一个有序集，用二叉树的结点来存储S中的元素得到有序集S的二叉搜索树。搜索结果有两种:
  - 找到x=x_i, 概率为p_i （查找成功）
  - 在叶结点确定x ∈ (x_i,x_i+1), 概率为q_i（查找不成功）
- (p_0,q_1,p_1,q_1,…,q_n,p_n)称为S的存取概率分布

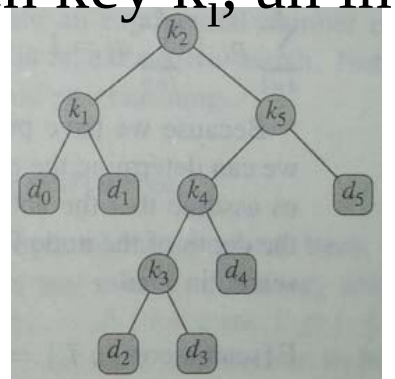


i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

最优二叉搜索树

- 给定有序集K = <$k_1, k_2, ..., k_n$> ， ($k_1 < k_2 < \cdots < k_n$), how to build a BST?
  - 对每个key k_i , 搜索概率p_i
  - 对某些不在K中的值, n+1 “虚拟键" $d_0, d_1, ..., d_n$
  - d_0 represents all values < k_1 ; d_n represents all values > k_n
  - for $1 \leqslant i \leqslant n-1$, d_i : $k_i < d_i < k_{i+1}$. 搜索概率q_i
  - Fig, each key k_i, an internal node; d_i , a leaf





i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

最优二叉搜索树

- 每次搜索要么成功，要么不成功,有

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

- 已知每个关键字(key)和虚拟键(dummy key)被搜索的概率,可以确定在给定 BST T内的一次搜索的期望代价.
  - 假设一次搜索的实际代价为检查的结点个数，则二叉搜索树T的一次搜索的期望代价为

$$\begin{aligned} \text{E[search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i \ , \end{aligned}$$

其中depth_T 表示一个节点在树T中的深度.

36

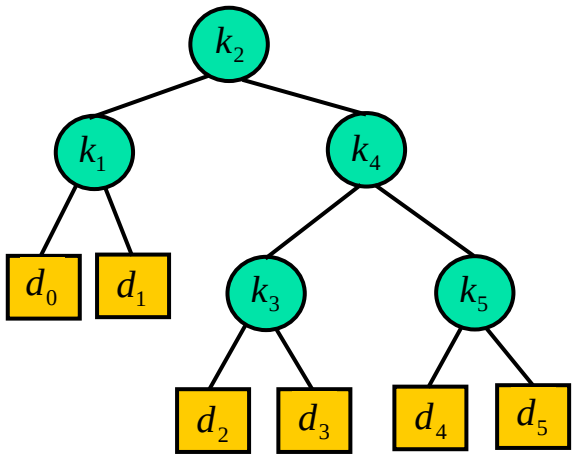
最优二叉搜索树

- 可以逐个结点(node by node)计算期望的搜索代价:

E[search cost in T]

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

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



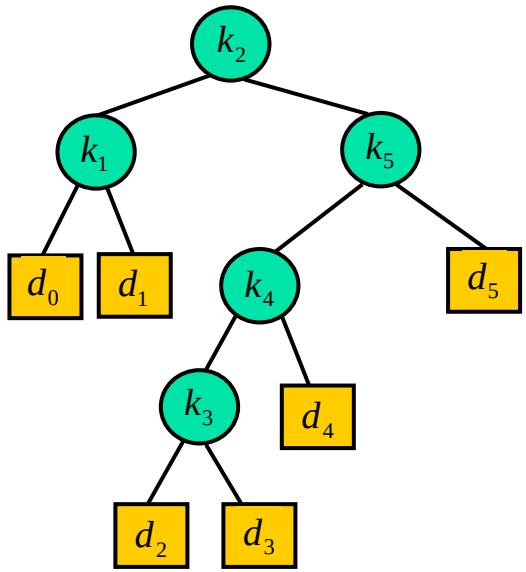
node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

最优二叉搜索树

E[search cost in T]

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

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



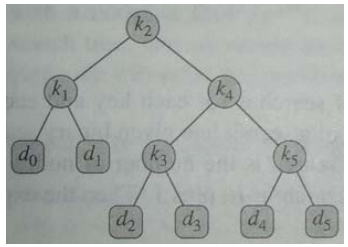
node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

最优二叉搜索树

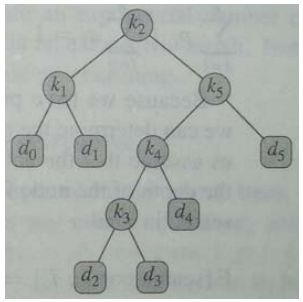
- 问题描述：
  - 对于有序集S及其存取概率分布，在所有表示有序集S的二叉搜索树中找出一颗具有最小平均路长的二叉搜索树。

最优二叉搜索树

- 最优 BST : for a given set of probabilities, our goal is to construct a BST whose **expected search cost is the smallest**.



(a) cost: 2.80



(b) cost: 2.75

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

- Figure (b) shows an optimal BST's expected cost is 2.75
  - An optimal BST is not necessarily a tree whose overall height is smallest. （不一定要求树的高度最小）
  - Nor can we necessarily construct an optimal BST by always putting the key with the greatest probability at the root. (The lowest expected cost of any BST with k_5 (the greatest probability) at the root is 2.85.)（不一定将概率最大的 key 放在树根，如…）

40

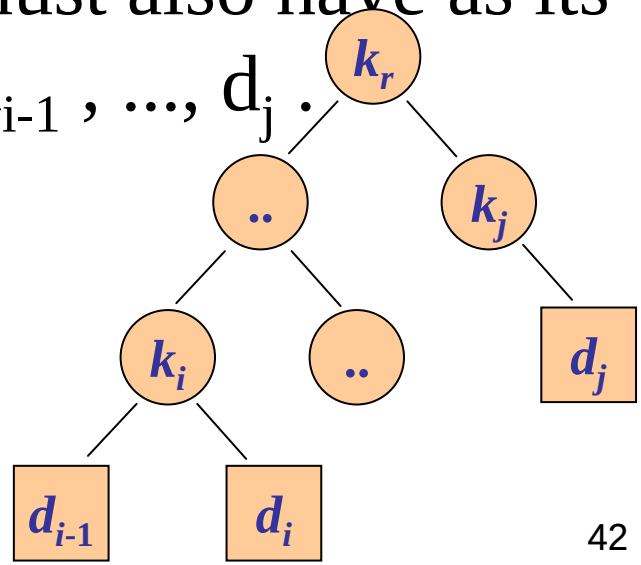
最优二叉搜索树

- 穷举检查所有的可能性的算法效率很低.
  - Matrix-chain multiplication; Scheduling assembly lines; LCS
- 为了构造 a BST，对具有 n 个节点的任何二叉树，将其节点分别标记为$k_1, k_2, \ldots, k_n$, 然后附加 dummy keys 作为树叶。则有 $\Omega(4^n/n^{3/2})$种二叉树。 （We can label the nodes of any n-node binary tree with the keys $k_1, k_2, \ldots, k_n$ to construct a BST, and then add in the dummy keys as leaves. We saw that the # of binary trees with n nodes is $\Omega(4^n/n^{3/2})$ ）
  - 穷举搜索法的时间复杂度为指数级
- Not surprisingly, we will solve this problem with **dynamic programming**. ?

步骤1:最优子结构性质

子问题

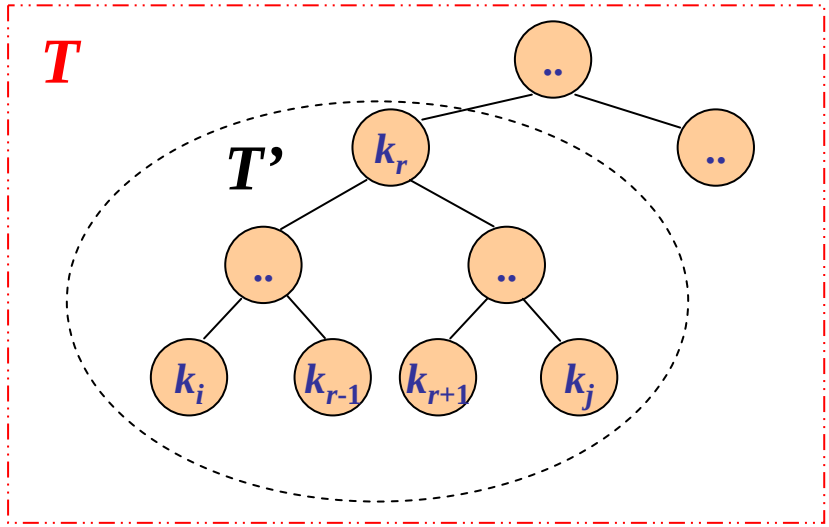
- Start with an observation about **subtrees**.
- Consider any subtree of a BST
  - It must contain keys in a contiguous range k_i , ..., k_j , for some $1 \leqslant i \leqslant j \leqslant n$.
  - In addition, the subtree must also have as its leaves the dummy keys d_{i-1} , ..., d_j .
- Optimal substructure?



42

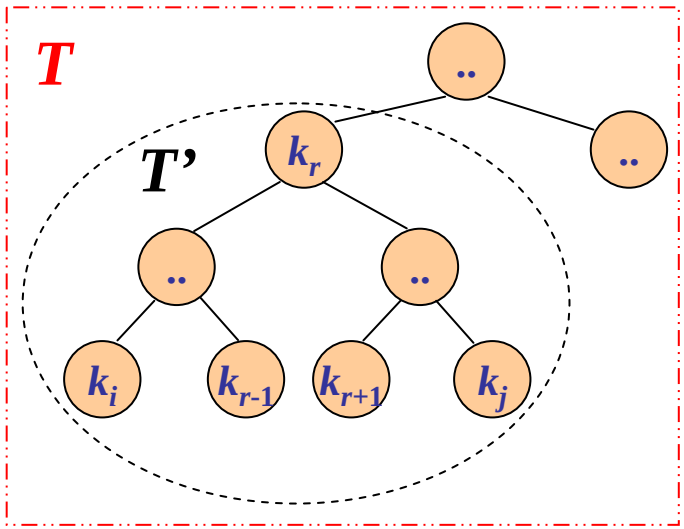
步骤1:最优子结构性质

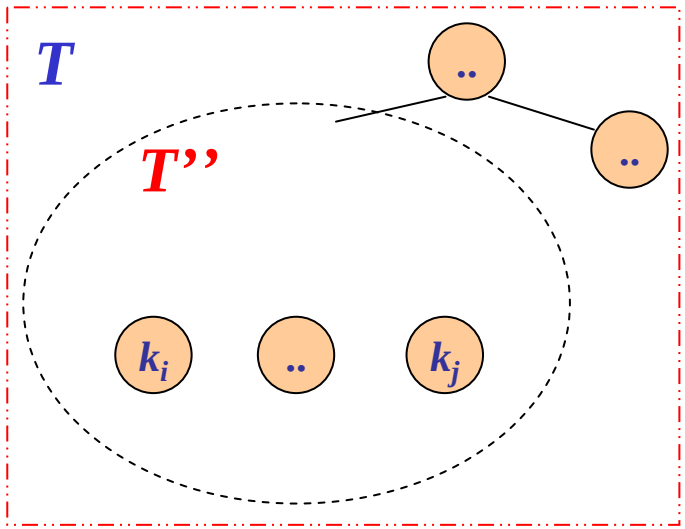
- 最优子结构
  - 设 T' 为最优BST T 的一个子树， T' 包含 keys $k_i, ..., k_j$, 那么T' 是子问题〔关于keys $k_i, ..., k_j$ 和dummy keys $d_{i-1}, ..., d_j$〕的最优BST



步骤1:最优子结构性质

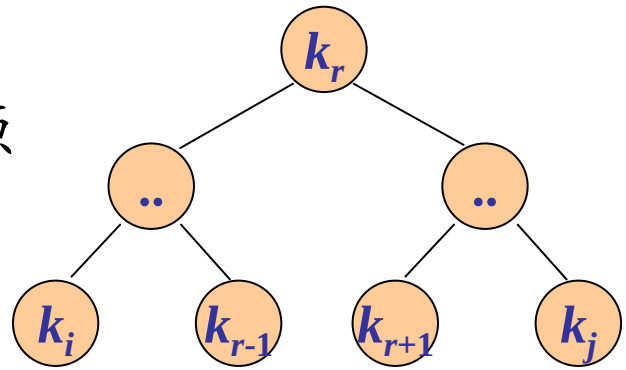
- 剪贴思想（Cut-and-paste）.
  - If there were a subtree *T*" whose expected cost is lower than that of *T*' , then we could cut *T*' out of *T* and paste in *T*" , resulting in a binary search tree of lower expected cost than *T*, thus contradicting the optimality of *T*.





步骤1:最优子结构性质

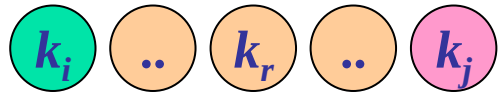
- 使用最优子结构, 可以根据子问题的最优解来构造原问题的一个最优解.
- Given keys k_i , ..., k_j , one of these keys, say(假设) **k_r** ($i \leqslant r \leqslant j$), will be the root of an optimal subtree.
  - The left subtree of the root k_r will contain the keys k_i , ..., k_{r-1} (and dummy keys d_{i-1} , ..., d_{r-1}); the right subtree will contain the keys k_{r+1} , ..., k_j (and dummy keys d_r , ..., d_j).
- As long as we examine all candidate roots k_r , where $i \leqslant r \leqslant j$, and we determine all optimal BST containing k_i , ..., k_{r-1} and those containing k_{r+1} , ..., k_j , we will find an optimal BST.（检查所有的 k_r和相应的子问题的最优BST，我们将找到原问题的最优BST）

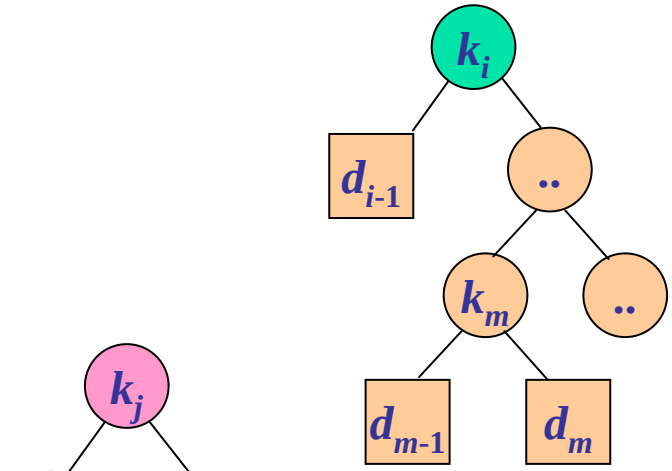


矩阵链乘法的分割点k

步骤1:最优子结构性质

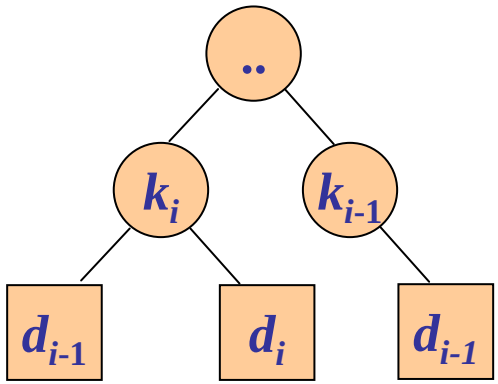
- A detail, “empty” subtrees
- Suppose that in a subtree with keys k_i , ..., k_j ,
  - We select k_i as the root, left subtree of k_i contains no keys. Bear in mind, however, that subtrees also contain dummy keys d_{i-1} .
  - Symmetrically, if we select k_j as the root, right subtree of k_j contains the keys k_{j+1} , ..., k_j ; this right subtree contains no actual keys, but it does contain the dummy key d_j .

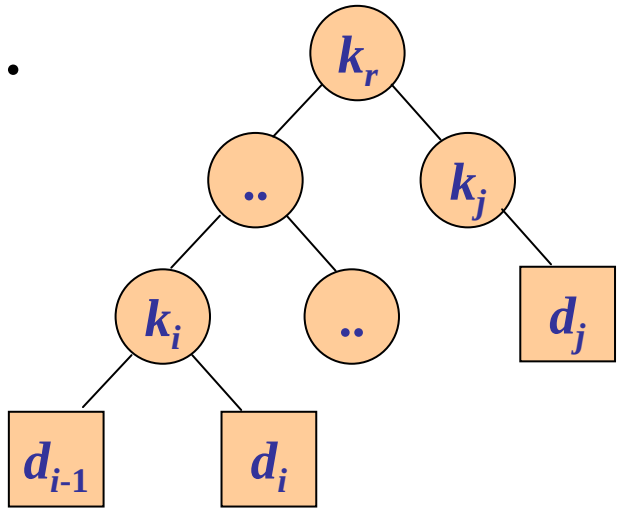




步骤2: 递归计算最优值

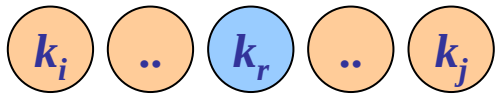
- 子问题 : finding an optimal BST containing the keys k_i , ..., k_j , where $i \geq 1$, $j \leq n$, and $j \geq i-1$. (when $j=i-1$, there are no actual keys, 只有虚拟键（ dummy key） d_{i-1} .)
- $e[i, j]$: 搜索一棵包含keys k_i , ..., k_j的最优BST的期望代价.
- Ultimately, wish to compute $e[1, n]$.
- when $j=i-1$, only d_{i-1} , $e[i, i-1] = q_{i-1}$.
- When $j \geq i$?

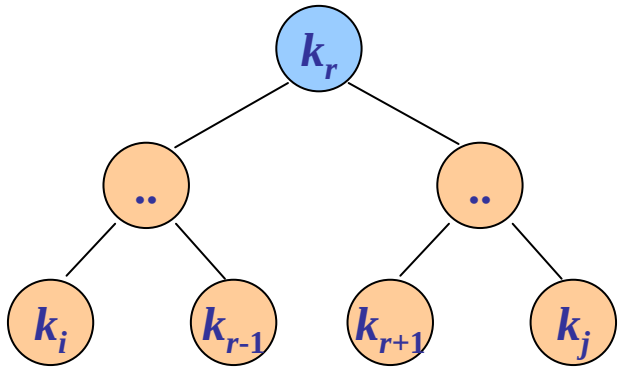




步骤2：递归计算最优值

- When $j \geqslant i$, select a root k_r from among k_i , ..., k_j , and then make an optimal BST with keys k_i , ..., k_{r-1} its left subtree and an optimal BST with keys k_{r+1} , ..., k_j its right subtree.



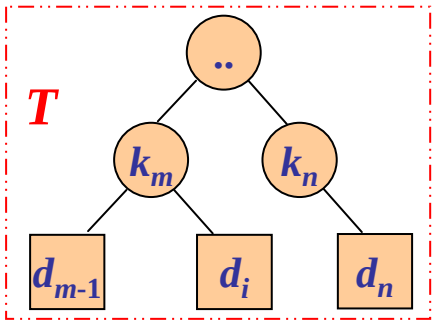


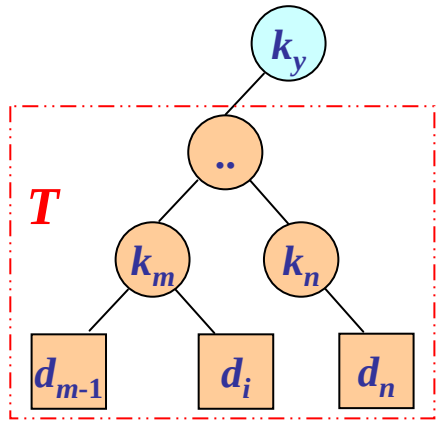
48

步骤2：递归计算最优值

- 当一棵树成为一个结点的子树时，期望搜索代价如何变化？
  - **The depth of each node in the subtree increases by 1, the expected search cost of this subtree increases by the sum of all the probabilities in the subtree. the sum of probabilities** （子树中每个节点的深度增加 1，该子树的期望搜索的 cost 的增量为该子树所有节点的搜索概率之和，…）

增量：$w[i,j]=\sum_{l=i}^{j} p_l+\sum_{l=i-1}^{j} q_l$





$$E_T = e[m,n]$$

$$E_T = \sum_{x=m}^{n}(\text{depth}(k_x)+1+1)\cdot p_i+\sum_{x=m-1}^{n}(\text{depth}(d_x)+1+1)\cdot q_x$$

$$=\sum_{x=m}^{n}(\text{depth}(k_x)+1)\cdot p_i+\sum_{x=m-1}^{n}(\text{depth}(d_x)+1)\cdot q_x+\sum_{x=m}^{n} p_i+\sum_{x=m-1}^{n} q_x$$

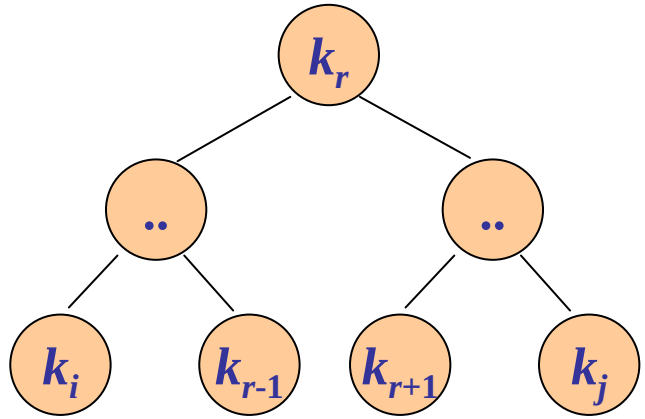
$$= e[m,n]+w[m,n]$$

步骤2: 递归计算最优值

- **Thus, if k_r is the root of an optimal subtree containing keys k_i , ..., k_j , we have**

  $e[i, j] = p_r + (e[i, r\text{-}1] + w[i, r\text{-}1])$

  $+ (e[r+1, j] + w[r+1, j])$?



Noting that $w[i, j] = w[i, r\text{-}1] + p_r + w[r+1, j]$

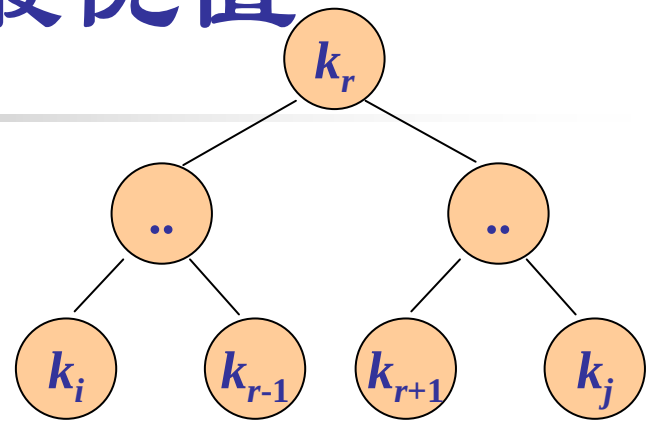
$$\left(w[i, r-1] = \sum_{l=i}^{r-1} p_l + \sum_{l=i-1}^{r-1} q_l \quad , \quad w[r+1, j] = \sum_{l=r+1}^{j} p_l + \sum_{l=r}^{j} q_l \right)$$

We rewrite $e[i, j]$ as

$$e[i, j] = e[i, r\text{-}1] + e[r+1, j] + w[i, j]$$

该递归公式假设知道采用哪个node k_r 作为根

步骤2：递归计算最优值



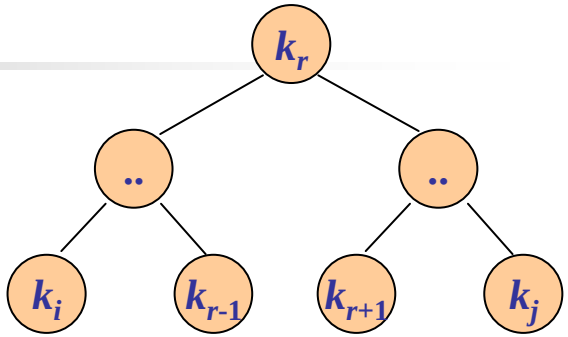
- **Choose k_r as the root that gives the lowest expected search cost, giving us our final recursive formulation:**

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w[i, j]\} & \text{if } i \le j. \end{cases}$$

- **$e[i, j]$ give the expected search costs in optimal BST.**
- **To help us keep track of the structure of optimal BST, define *root*$[i, j]$, for $1 \leqslant i \leqslant j \leqslant n$, to be the index r for which k_r is the root of an optimal BST containing keys k_i , ..., k_j .**

步骤3:计算期望搜索代价

$A_i .. A_r .. A_j$



$$e[i,j]=\begin{cases} q_{i-1} & \text{if } j=i-1, \\ \min_{i\le r\le j}\{e[i,r-1]+e[r+1,j]+w[i,j]\} & \text{if } i\le j. \end{cases}$$

- **Similarity: optimal BST and matrix-chain multiplication.**
- **A direct, recursive implementation would be as inefficient.**
- **Store the *e*[*i*, *j*] values in a table *e*[1.. *n*+1, 0.. *n*] .**
  - **The first index runs to *n*+1, in order to have a subtree containing only d_n , need to compute and store *e*[*n*+1, *n*] .**
  - **The second index starts from 0, in order to have a subtree containing only d_0 , need to compute and store *e*[1, 0] .**
- **Use a table *root*[*i*, *j*] , for recording the root of the subtree containing keys k_i , ..., k_j , only for $1 \leqslant i \leqslant j \leqslant n$.**

步骤3:计算期望搜索代价

- **Other table for efficiency.**

  $e[i, j] = e[i, r-1] + e[r+1, j] + w[i, j]$

- **Rather than compute the value of $w[i, j]$ every time we are computing $e[i, j]$—which would take $\Theta(j-i)$ additions—we store these values in a table $w[1.. n+1, 0.. n]$.**

  （无需每次计算 $e[i, j]$ 时都计算 $w[i, j]$，…）

  - **For the base case, we compute $w[i, i-1] = q_{i-1}$ for $1 \leqslant i \leqslant n$.**

- **For $j \geqslant i$,**

$$w[i, j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l = w[i, j-1] + p_j + q_j$$

- **Thus, compute the $\Theta(n^2)$ values of $w[i, j]$ in $\Theta(1)$ time each.**
- **输入: the probabilities p_1, ..., p_n and q_0, ..., q_n and the size n**
- **输出:表e和表root**

步骤3:计算期望搜索代价

$$e[i,j]=\begin{cases} q_{i-1} & \text{if } j=i-1,\\ \min\limits_{i\le r\le j}\{e[i,r-1]+e[r+1,j]+w[i,j]\} & \text{if } i\le j. \end{cases}$$

$$w[i,j]=\sum_{l=i}^{j}p_l+\sum_{l=i-1}^{j}q_l=w[i,j-1]+p_j+q_j$$

```
OPTIMAL-BST(p, q, n)
1  for i←1 to n+1
2      do e[i, i-1]←q_{i-1}
3         w[i, i-1]←q_{i-1}
4  for l←1 to n
5      do for i←1 to n-l+1   // ?1
6             do j←i+l-1     // ?2
7                e[i, j]←∞
8                w[i, j]←w[i, j-1]+p_j+q_j
9                for r←i to j
10                   do t←e[i, r-1]+e[r+1, j]+w[i, j]
11                      if t<e[i, j]
12                         then e[i, j]←t
13                              root[i, j]←r
14 return e and root
```

二叉树的键的个数
(决定子问题的规模)

子问题的起始(i)和终止(j)位置

?1

$k_1, k_2, \ldots, k_n$

e[i, j]:

l 个元素的 Opti-BST 的 cost

i = 1, *j* = *l*,

i = 2, *j* = *l*+1,

…

i = *x*, *j* = *n*,

n-*x*+1=*l* => *x*=*n*-*l*+1

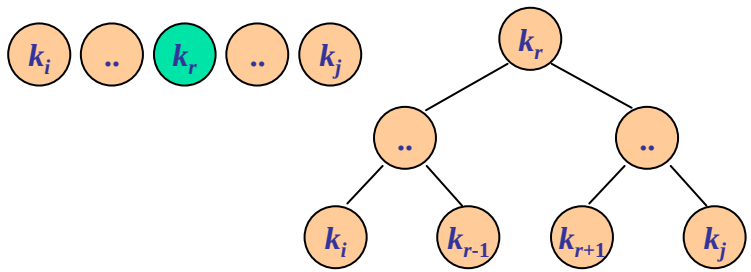
?2

j-*i*+1 = *i*+*l*-1-*i*+1 = *l*

步骤3:计算期望搜索代价

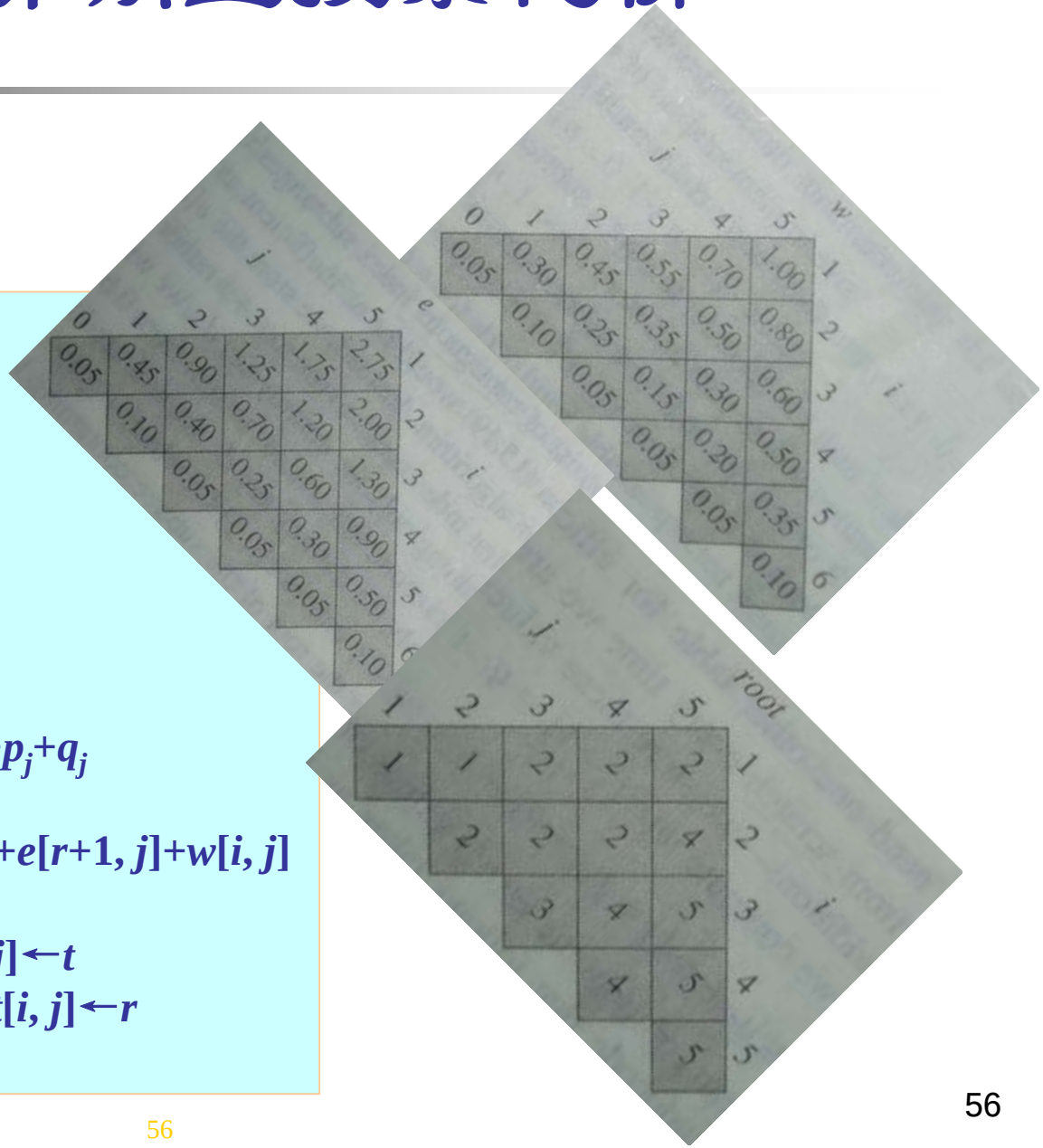
```
OPTIMAL-BST(p, q, n)
1  for i←1 to n+1
2       do e[i, i-1]←q_{i-1}
3          w[i, i-1]←q_{i-1}
4  for l←1 to n          // 求l 个元素的Opti-BST
5       do for i←1 to n-l+1
6             do j←i+l-1
7                e[i, j]←∞
8                w[i, j]←w[i, j-1]+p_j+q_j
9                for r←i to j
10                  do t←e[i, r-1]+e[r+1, j]+w[i, j]
11                     if t<e[i, j]
12                        then e[i, j]←t
13                             root[i, j]←r
14  return e and root
```

Innermost for loop, in lines 9–13, tries each candidate index r to determine which key k_r to use as the root of an optimal BSTcontaining keys k_i , ..., k_j .（对包含 k_i , ..., k_j 的最优 BST，尝试每一个 k_r 作为树根，…）



步骤3:计算期望搜索代价

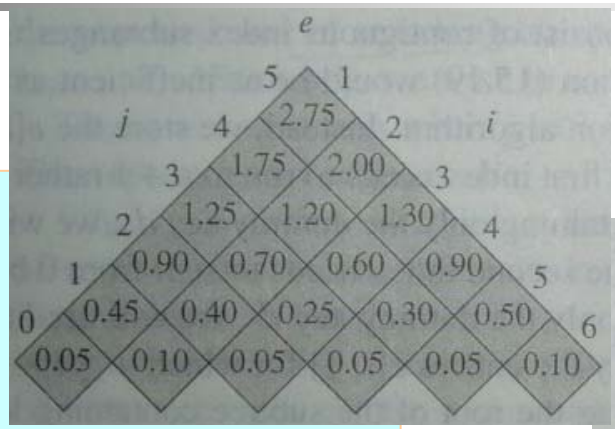
```
OPTIMAL-BST(p, q, n)
1  for i←1 to n+1
2       do e[i, i-1]←q_{i-1}
3          w[i, i-1]←q_{i-1}
4  for l←1 to n
5       do for i←1 to n-l+1
6             do j←i+l-1
7                e[i, j]←∞
8                w[i, j]←w[i, j-1]+p_j+q_j
9                for r←i to j
10                  do t←e[i, r-1]+e[r+1, j]+w[i, j]
11                     if t<e[i, j]
12                        then e[i, j]←t
13                             root[i, j]←r
14 return e and root
```

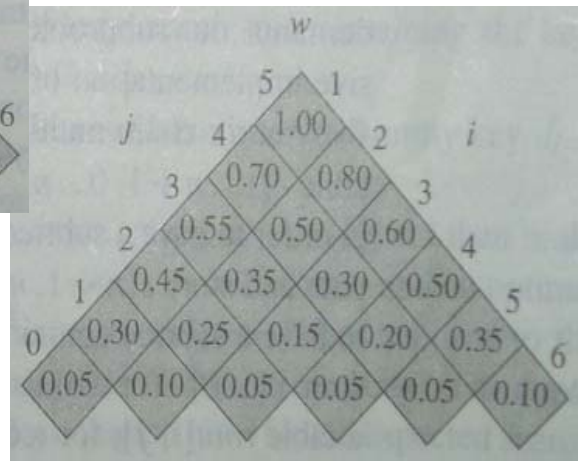


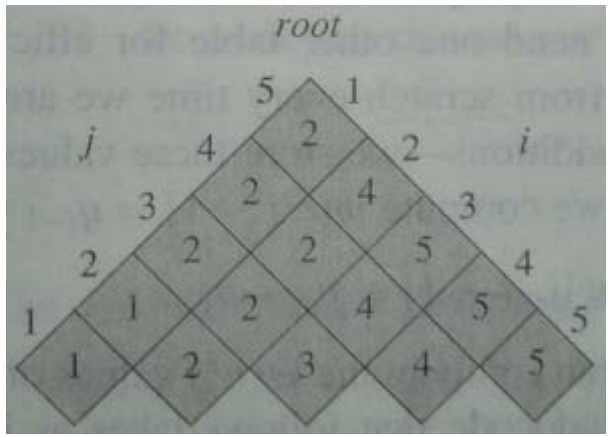
56

步骤3:计算期望搜索代价

```
OPTIMAL-BST(p, q, n)
1  for i←1 to n+1
2       do e[i, i-1]←q_{i-1}
3          w[i, i-1]←q_{i-1}
4  for l←1 to n
5       do for i←1 to n-l+1
6             do j←i+l-1
7                e[i, j]←∞
8                w[i, j]←w[i, j-1]+p_j+q_j
9                for r←i to j
10                  do t←e[i, r-1]+e[r+1, j]+w[i, j]
11                     if t<e[i, j]
12                        then e[i, j]←t
13                             root[i, j]←r
14 return e and root
```







57

步骤3:计算期望搜索代价

```
OPTIMAL-BST(p, q, n)
1  for i←1 to n+1
2       do e[i, i-1]←q_{i-1}
3          w[i, i-1]←q_{i-1}
4  for l←1 to n
5       do for i←1 to n-l+1     // n-l+1 times
6             do j←i+l-1
7                e[i, j]←∞
8                w[i, j]←w[i, j-1]+p_j+q_j
9                for r←i to j    // j-i+1=i+l-1-i+1=l
10                  do t←e[i, r-1]+e[r+1, j]+w[i, j]
11                     if t<e[i, j]
12                        then e[i, j]←t
13                             root[i, j]←r
14 return e and root
```

运行时间: $\Theta(n^3)$

Proof:

$O(n^3)$: for loops are nested three deep and each loop index takes on at most n values.

$\Omega(n^3)$:

$$\sum_{l=1}^{n}(n-l+1)l = \sum_{l=1}^{n}(n+1)l - \sum_{l=1}^{n}l^2$$

$$= \frac{(n+1)(n+1)n}{2} - \frac{n(n+1)(2n+1)}{6}$$

$$= \frac{n(n+1)(n+2)}{6} = \Omega(n^3)$$

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

分治法与动态规划法的比较

	分治法	动态规划法
相同点	通过合并子问题的解以得到原问题的解.	
不同点	1) 把原问题划分为独立的子问题（partition the problem into independent subproblems） 2) 递归地解这些子问题（solve the subproblems Recursively）， 3) 将各子问题的解合并得到原问题的解（combine their solutions to solve the original problem）.	子问题不独立，不同子问题共享相同的子子问题（ the subproblems are not indepe- ndent, that is, when subproblems share subsubproblems）
	做很多不必要的工作, 重复求解公共子子问题.	只求解公共子子问题一次，并用一张表来保存所有的答案,避免大量的重复计算.

分治法与动态规划法的比较

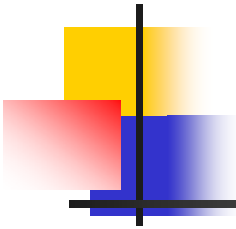
- 动态规划算法常用于优化问题 (optimization problems).
- 优化问题
  - There can be many possible solutions,
  - Each solution has a value,
  - We wish to find a solution with the optimal (min or max) value.
- 称问题的某个解为一个最优解，而不是单纯地称为最优解，因为可能有多个解能得出问题的最优值

分治法与动态规划法的比较

- 动态规划法包含以下四步(CRBC).
  - 1.找出最优解的性质，并刻划其结构特征(Characterize the structure of an optimal solution).
  - 2.递归地定义最优值(Recursively define the value of an optimal solution).
  - 3.以自底向上的方式计算出最优值(Compute the value of an optimal solution in a bottom-up fashion).
  - 4.根据计算最优值时得到的信息，构造最优解(Construct an optimal solution from computed information).
- 当仅需求最优值时step 4 通常可以省略。为了执行step 4〔更容易地构造最优解〕，通常在执行step 3 时记录必要的信息

总结

- 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。



总结

- 算法的共性
  - 表格化
  - 自底向上
  - 基于递归式