动态规划算法的应用

汇报人：陈莹蔚

应用

01 最长公共子序列（LCS）

02 图像压缩

03 最大子段和

04 最优二分检索树

01

最长公共子序列（LCS）

子序列定义

X 的子序列 Z ： 设序列 $X, Z,$

$$X =< x_1, x_2, \ldots, x_m >$$

$$Z =< z_1, z_2, \ldots, z_k >$$

要求子序列顺序一致、无需连续

若存在 X 的元素构成的严格递增序列 $< x_{i_1}, x_{i_2}, \ldots, x_{i_k} >$

使得 $z_j = x_{i_j}, j = 1,2,\ldots,k,$ 则称 Z 是 X 的子序列

X 与 Y 的公共子序列 Z ： Z 是 X 和 Y 的子序列

最长公共子序列（LCS）

问题目标：
寻找两个序列的所有**公共子序列**中长度**最长**的那个。

示例：
X = <A, B, C, B, D, A, B>
Y= <B, D, C, A, B, A>

其最长公共子序列为

<B,C,B,A>（长度4）。

蛮力算法分析

蛮力算法思路

找出序列X的每个子序列X'

并且把X'和Y进行比较

看看X'是否也是Y的子序列。

如果是，X'就是X与Y的公共子序列。

当所有的比较完成后

就找到了X与Y的最长公共子序列。

子序列数量分析

在选择X'时，

每个X的元素有两种可能的选择：

属于X'还是不属于X'。

假设序列X的长度为m，

那么根据排列组合原理，

X有2^m个子序列。

蛮力算法分析

时间复杂度分析

如果检查X'与Y需要O(n)时间（n为序列Y的长度），

由于有2^m个子序列需要检查，

那么整个算法需要$O(n2^m)$时间，

这是一个指数时间的算法，

随着序列长度的增加，计算量会急剧增大。

子问题划分及依赖关系

子问题边界： X和Y起始位置为1，X的终止位置是 i， Y的终止位置是j，记作

$$X_i=<x_1,x_2,\ldots,x_i>,\quad Y_j=<y_1,y_2,\ldots,y_j>$$

依赖关系：

$$X=<x_1,x_2,\ldots,x_m>,\ Y=<y_1,y_2,\ldots,y_n>,\ Z=<z_1,z_2,\ldots,z_k>,$$

Z 为 X 和 Y 的 LCS，那么

(1) 若$x_m=y_n \Rightarrow z_k=x_m=y_n$,且$Z_{k-1}$是$X_{m-1}$与$Y_{n-1}$的 LCS;

(2) 若$x_m\neq y_n, z_k\neq x_m \Rightarrow Z$是$X_{m-1}$与$Y$的 LCS;

(3) 若$x_m\neq y_n, z_k\neq y_n \Rightarrow Z$是$X$与$Y_{n-1}$的 LCS.

动态规划算法思路

优化函数递推关系推导

设C[i, j]表示X_i与Y_j的最长公共子序列的**长度**。

根据上述分析，得到优化函数的递推关系：

递推方程

$$C[i,j]=\begin{cases}0 & \text{若}i=0\text{或}j=0\\ C[i-1,j-1]+1 & \text{若}i,j>0,x_i=y_j\\ \max\{C[i,j-1],C[i-1,j]\} & \text{若}i,j>0,x_i\neq y_j\end{cases}$$

表格：LCS递推关系与标记函数对应关系

情况	条件	递推关系	标记函数 $B[i,j]$
情况1	$x_i = y_j$	$C[i,j] = C[i-1,j-1] + 1$	$\backslash$
情况2	$x_i \neq y_j$ 且 $C[i-1,j] \geq C[i,j-1]$	$C[i,j] = C[i-1,j]$	$\uparrow$
情况3	$x_i \neq y_j$ 且 $C[i,j-1] > C[i-1,j]$	$C[i,j] = C[i,j-1]$	$\leftarrow$
初始化	$i = 0$ 或 $j = 0$	$C[i,j] = 0$	无

算法3.4 LCS(*X*,*Y*,*m*,*n*)

```
1. for i←1 to m do          //行1-4边界情况
2.     C[i,0]←0
3. for i←1 to n do
4.     C[0,i]←0
5. for i←1 to m do
6.     for j←1 to n do
7.         if X[i]=Y[j]
8.         then C[i,j]←C[i–1,j–1]+1
9.              B[i,j]←'↖'
10.        else if C[i–1,j] ≥ C[i,j–1]
11.             then C[i,j]←C[i–1,j]
12.                  B[i,j]←'↑'
13.             else C[i,j]←C[i,j–1]
14.                  B[i,j]←'←'
```

O(m)

O(n)

O(mn)

O(1)

时间复杂度：O（mn）

追踪解的算法

算法 Structure Sequence(*B*, *i*, *j*)

输入：*B*[*i*,*j*]

输出：*X*与*Y*的最长公共子序列

```
1. if i=0 or j=0 then return //一个序列为空
2. if B[i,j] = "↖"
3. then 输出X[i]
4.      Structure Sequence(B, i－1, j－1)
5. else if B[i,j]= "↑" then Structure Sequence (B, i－1, j)
6.      else Structure Sequence (B, i, j－1)
```

时间复杂度：O(m+n),算法从i=m,j=n开始

输入：X=<A,B,C,B,D,A,B>, Y=<B,D,C,A,B,A>,

标记函数：

每次"↖"出现，输出该元素。

	1	2	3	4	5	6
1	$B[1,1]=\uparrow$	$B[1,2]=\uparrow$	$B[1,3]=\uparrow$	$B[1,4]=\nwarrow$	$B[1,5]=\leftarrow$	$B[1,6]=\nwarrow$
2	$B[2,1]=\nwarrow$	$B[2,2]=\leftarrow$	$B[2,3]=\leftarrow$	$B[2,4]=\uparrow$	$B[2,5]=\nwarrow$	$B[2,6]=\leftarrow$
3	$B[3,1]=\uparrow$	$B[3,2]=\uparrow$	$B[3,3]=\nwarrow$	$B[3,4]=\leftarrow$	$B[3,5]=\uparrow$	$B[3,6]=\uparrow$
4	$B[4,1]=\uparrow$	$B[4,2]=\uparrow$	$B[4,3]=\uparrow$	$B[4,4]=\uparrow$	$B[4,5]=\nwarrow$	$B[4,6]=\leftarrow$
5	$B[5,1]=\uparrow$	$B[5,2]=\uparrow$	$B[5,3]=\uparrow$	$B[5,4]=\uparrow$	$B[5,5]=\uparrow$	$B[5,6]=\uparrow$
6	$B[6,1]=\uparrow$	$B[6,2]=\uparrow$	$B[6,3]=\uparrow$	$B[6,4]=\nwarrow$	$B[6,5]=\uparrow$	$B[6,6]=\nwarrow$
7	$B[7,1]=\uparrow$	$B[7,2]=\uparrow$	$B[7,3]=\uparrow$	$B[7,4]=\uparrow$	$B[7,5]=\uparrow$	$B[7,6]=\uparrow$

解：$X[2]$,$X[3]$, $X[4]$, $X[6]$, 即 B, C, B, A

总结

动态规划算法将求解最长公共子序列问题的蛮力算法的$O(n2^m)$时间降低到$O(mn)$时间，大大提高了算法效率。

02

图像压缩

变位压缩技术

01 概念

每个像素灰度值范围为 0～255，通常用 8 位二进制表示。

传统图像存储方式下，一幅有n个像素的图像，需8*n个二进制位存储。

这种方式浪费空间，因为很多图像中连续区域像素灰度值接近。

于是，采用**变位压缩技术**，对灰度值较小的段的像素采用较少位数（如2位），对灰度值较大的段的像素采用较多位数（如8位），以此减少空间占用。

02 变位压缩带来的读取问题

虽然变位压缩技术节省了空间，但读取图像时出现新问题。

在每个像素8位的存储方法中，读取图像时每8位就是一个像素的灰度值，不会出错。

对于分段压缩的图像，由于存储的是长长的0 - 1序列，需明确**每段的划分位置**及每段像素**占用的二进制位数。**

变位压缩存储格式

L[i]： 第 i 段中包含的像素个数。
b[i]： 第 i 段中每个像素的二进制存储位数。

取值范围：

$1 \leq b[i] \leq 8$（每个像素灰度值范围为 0～255，最多需要 8 位）。

计算方法：

$$b[i] = \lceil \log_2(\max_{p_k \in S_i} p_k + 1) \rceil$$

- $\max_{p_k \in S_i} p_k$：第 i 段中的最大灰度值。
- 例如，若段内最大灰度值为 15，则 $b[i] = \lceil \log_2(15+1) \rceil = 4$。

每段需要额外 11 位存储段头信息：
8 位：段内像素个数 L[i]（最多 256 个像素）+ 3 位：每个像素的存储位数 b[i]（最多 8 位）。

第i段占用空间：L[i] x b[i] +11

问题描述

输入：像素灰度值序列 P=<p1,p2,…,pn>
目标：将 P分段存储，使得总存储位数最少。

S[i]：前 i 个像素的最优存储位数。

$$b[i-j+1,i]=\left\lceil \log(\max_{p_k\in S_m} p_k+1)\right\rceil \le 8$$

$$s[i]=\min_{1\le j\le \min\{i,256\}}\{s[i-j]+j\times b[i-j+1,i]+11\}$$

对于每个 i（从 1 到 n），尝试所有可能的分段长度 j（从 1 到 min(256,i)）：
计算段内最大灰度值的二进制位数b[i−j+1,i]，更新最小的s[i]为新s[i].

```
Compress (P,n)                    //计算最小位数S[n]
1. Lmax←256; header←11; S[0]←0   //最大段长Lmax, 头header
2. for i←1 to n do
3.     b[i]←length(P[i])      //b[i]是第i个灰度P[i]的二进制位数
4.     bmax←b[i]               //3-6行分法的最后一段只有P[i]自己
5.     S[i]←S[i–1]+bmax
6.     l[i]←1
7.     for j←2 to min{i,Lmax} do   //最后段含 j 个像素
8.         if bmax<b[i–j+1]        //统一段内表示像素的二进制位数
9.         then bmax←b[i–j+1]
10.        if S[i]>S[i–j]+j*bmax
11.        then S[i]←S[i–j]+j*bmax
12.                l[i]←j
13.    S[i]←S[i]+header
```

时间复杂度 $T(n)=O(n)$

1. 输入像素序列：$P =< 10, 12, 15, 255, 1, 2 >$。
2. 计算前两步：
   - 第1个像素：$S[1] = 15,\ l[1] = 1$。
   - 第2个像素：$S[2] = 19,\ l[2] = 2$。
   - $S[3] = 23,\ l[3] = 3$。
   - $S[4] = 42,\ l[4] = 1$。
   - $S[5] = 50,\ l[5] = 2$。
   - $S[6] = 57,\ l[6] = 2$。

总位数：57

分段长度 $j = 1$：

- 段内最大灰度值 = 12 → $b = 4$。
- 当前位数 = $S[1] + 1 \times 4 + 11 = 15 + 4 + 11 = 30$。

分段长度 $j = 2$：

- 段内最大灰度值 = max(10,12) = 12 → $b = 4$。
- 当前位数 = $S[0] + 2 \times 4 + 11 = 0 + 8 + 11 = 19$。
- 更新 $S[2] = 19,\ l[2] = 2$。

输出了前 i 个像素的最小存储位数，但没有记录如何分段，因此我们还需要追踪解，得到具体分段方案。

追踪解Traceback(n,l)　　时间复杂度：O(n)

1. j←1：从第1段开始追踪。
   n：当前剩余的像素数量，初始为总长度 n。
2. 当 n≠0 时，继续追踪。
3. C[j]←l[n]
4. 将第 j 段的长度设为 n←n−l[n]：更新剩余的像素数量。
5. j←j+1：准备追踪下一段。

假设输入像素序列为 $P =< 10, 12, 15, 255, 1, 2 >$，动态规划结果如下：

- $l[1] = 1,\ l[2] = 2,\ l[3] = 3,\ l[4] = 1,\ l[5] = 2,\ l[6] = 2$。

1. **初始化**：
   - $j = 1,\ n = 6$。
2. **第1段**：
   - $C[1] = l[6] = 2$（最后一段长度为2）。
   - 更新 $n = 6 - 2 = 4$。
   - 更新 $j = 2$。
3. **第2段**：
   - $C[2] = l[4] = 1$（倒数第二段长度为1）。
   - 更新 $n = 4 - 1 = 3$。
   - 更新 $j = 3$。
4. **第3段**：
   - $C[3] = l[3] = 3$（倒数第三段长度为3）。
   - 更新 $n = 3 - 3 = 0$。
   - 算法结束。

最终结果：

- 数组 $C = [2, 1, 3]$，表示最优分段为：
  - 第1段：$< 1, 2 >$（长度2）。
  - 第2段：$< 255 >$（长度1）。
  - 第3段：$< 10, 12, 15 >$（长度3）。

03

最大子段和

最大子段和

问题描述：

给定一个包含 n 个整数（可能为负数）的序列 $(a_1, a_2, \ldots, a_n)$，找到连续子序列，使得其和最大。

数学表达：

$$\max\left\{0, \max_{1\le i\le j\le n}\sum_{k=i}^{j} a_k\right\}$$

示例：

序列：$(-2, 11, -4, 13, -5, -2)$

最大子段和：$11 + (-4) + 13 = 20$。

算法1：顺序求和+比较
算法2：分治策略
算法3：动态规划

算法1：顺序求和+比较

核心思想：枚举所有可能的子序列，计算其和，并记录最大值。

```
算法 Enumerate
输入：数组 A[1..n]
输出：sum, first, last
1. sum ← 0
2. for i ← 1 to n do
3.   for j ← i to n do
4.     thissum ← 0
5.     for k ← i to j do
6.       thissum ← thissum + A[k]
7.     if thissum > sum then
8.       sum ← thissum
9.       first ← i
10.      last ← j
```

时间复杂度：O(n^3)。

优点：简单直观。

缺点：效率低，不适用于大规模数据。

算法2：分治策略

核心思想：
将序列分成左右两部分，分别求解左右部分的最大子段和，再考虑跨越中点的最大子段和。

算法 MaxSubSum(A, left, right)

输入：数组A，$left$，$right$分别是A的左、右边界
输出：A的最大子段和sum及其子段的前后边界

1. if $|A|$=1 then 输出元素值（当值为负时输出0）
2. $center \leftarrow \lfloor (left+right)/2 \rfloor$
3. $leftsum \leftarrow$ MaxSubSum(A,$left$,$center$)　　//子问题A_1
4. $righsum \leftarrow$ MaxSubSum(A,$center$+1,$right$)　//子问题A_2
5. $S_1 \leftarrow$从A[$center$]向左的最大和　//从$center$向左的最大和
6. $S_2 \leftarrow$从A[$center$+1]向右的最大和　//从$center$+1向右的最大和
7. $sum \leftarrow S_1+S_2$
8. if $leftsum > sum$ then $sum \leftarrow leftsum$
9. if $rightsum > sum$ then $sum \leftarrow rightsum$

T(n)=2T(n/2)+O(n),T(c)=O(1)　　T(n)=O(nlogn)

算法3：动态规划

核心思想：
通过状态转移方程，逐步计算以每个位置结尾的最大子段和，最终得到全局最大值。

令$C[i]$ 是 $A[1..i]$中必须包含元素$A[i]$的最大子段和

$$C[i] = \max_{1 \le k \le i} \{ \sum_{j=k}^{i} A[j] \}$$

递推公式：
C[i]=max{C[i−1]+A[i],A[i]}
如果 C[i−1]>0，则将 A[i] 加入当前子段。
如果 C[i−1]≤0，则重新开始一个子段，从A[i] 开始。
初始化：C[0]=0

伪代码

```
算法 MaxSum(A, n)
输入：数组 A，数组长度 n
输出：最大子段和 sum，子段的最后位置 c
1. sum ← 0
2. b ← 0                                        // b 是前一个最大子段和
3. for i ← 1 to n do
4.   if b > 0 then
5.     b ← b + A[i]                             // 将 A[i] 加入当前子段
6.   else
7.     b ← A[i]                                 // 重新开始一个子段
8.   if b > sum then
9.     sum ← b                                  // 更新最大子段和
10.    c ← i  // 记录最大子段的结束位置
11. return sum, c
```

时间复杂度：O(n) (只有一个for循环)

如果想要确定子段开始位置呢?

```
算法 MaxSum(A, n)
输入：数组 A，数组长度 n
输出：最大子段和 sum，子段的最后位置 c
1. sum ← 0
2. b ← 0                                    // b 是前一个最大子段和
3. current_start ← 1                        // 当前子段的起始位置
4. s ← 1                                    // 最大子段的起始位置
5. c ← 1                                    // 最大子段的结束位置
6. for i ← 1 to n do
7.    if b > 0 then
8.       b ← b + A[i]                       // 将 A[i] 加入当前子段
9.    else
10.      b ← A[i]                           // 重新开始一个子段
11. current_start ← i                       // 重置子段时更新起始位置
12.   if b > sum then
13.      sum ← b                            // 更新最大子段和
14.      c ← i                              // 记录最大子段的结束位置
15.      s ← `current_start                 // 更新最大子段的起始位置
16. return sum, c, s
```

04 最优二分检索树

二分检索树

- 其每个节点的左子树只包含小于当前节点的数，右子树只包含大于当前节点的数，且左右子树也都是二叉搜索树。
- 检索时，从给定整数x与树根的比较开始。若x等于树根，则检索停止；若x小于树根，则进入左子树继续与左儿子比较；若x大于树根，则进入右子树与右儿子比较，如此递归进行，直到找到目标元素或确认元素不在树中。
- 树深度与算法时间复杂度的关系
  - 算法的最坏情况对应二分检索树中从根出发的最长路径，即树的深度代表了算法最坏情况下的时间复杂度。例如，某棵二叉树树深为3，则在最坏情况下，算法最多需要进行3次比较。

问题与定义

- **输入**：
  - 数据集 $S =< x_1, x_2, \ldots, x_n >$，有序关键字。
  - 存取概率分布 $P =< a_0, b_1, a_1, b_2, \ldots, b_n, a_n >$，其中：
    - b_i：数据结点 x_i 的存取概率。
    - a_j：空隙区间 (x_j, x_{j+1}) 的存取概率。
- **目标**：构造一棵二叉搜索树，最小化平均比较次数：

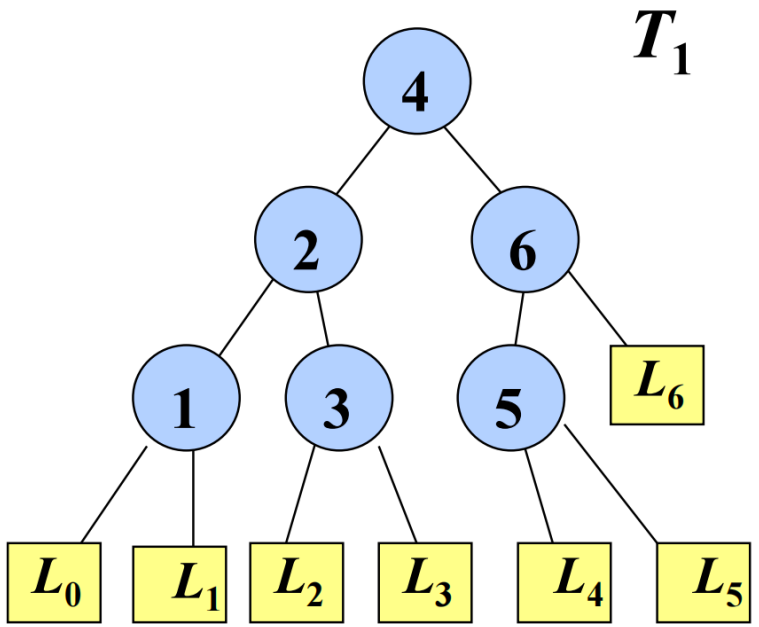
$$t = \sum_{i=1}^{n} b_i(1 + d(x_i)) + \sum_{j=0}^{n} a_j d(L_j)$$

其中 $d(x_i)$ 是结点 x_i 的深度，$d(L_j)$ 是空隙的深度。

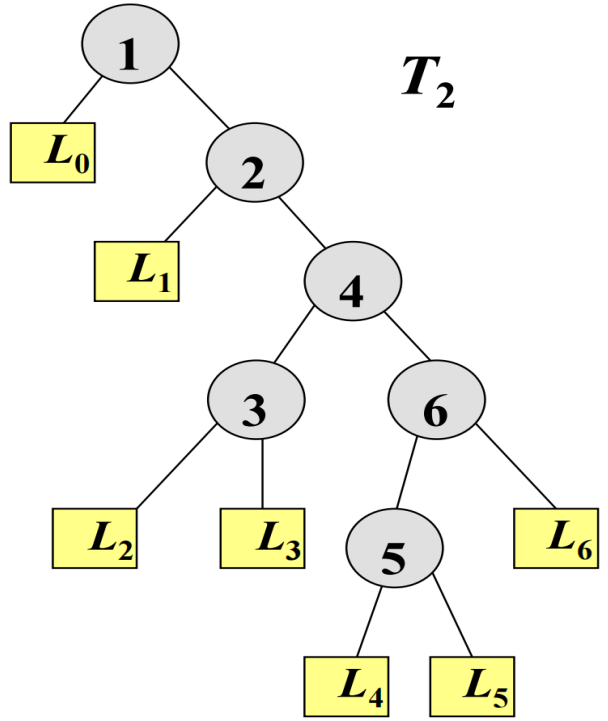
不同二分检索树的平均比较次数,m(Tree)

S = < 1, 2, 3, 4, 5, 6 >

P = < 0.04, 0.1, 0.01, 0.2, 0.05, 0.2, 0.02, 0.1, 0.02, 0.1, 0.07, 0.05, 0.04 >



m(T1)=[1*0.1+2*(0.2+0.05)+3*(0.1+0.2+0.1)]
+[3*(0.04+0.01+0.05+0.02+0.02+0.07)+2*0.04]
= 1.8+0.71=2.51



m(T2)=3.25

子问题划分

以 xk 作为根，划分成两个子问题：

S[i,k−1], P[i,k−1]

S[k+1,j], P[k+1,j]

例: 以B为根，划分成以下子问题：

S[1,1]=<A>，P[1,1]=<0.04, 0.1, 0.02>

S[3,5]=<C,D,E>, P[3,5]=<0.02, 0.1, 0.05, 0.2, 0.06,0.1,0.01>

递推方程

m[i][j]：区间 S[i..j] 构建最优 BST 的最小平均比较次数。

w[i][j]：区间 S[i..j] 的总概率（包括数据结点和空隙）。

$$w[i,j]=\sum_{p=i-1}^{j}a_p+\sum_{q=i}^{j}b_q$$

S = < 1, 2, 3, 4, 5, 6 >
P = < 0.04, 0.1, 0.01, 0.2, 0.05, 0.2, 0.02, 0.1, 0.02, 0.1, 0.07, 0.05, 0.04 >

w[2][4]=0.01+0.2+0.05+0.2+0.02+0.1+0.02

$$m[i,j]=\min_{i\le k\le j}\{m[i,k-1]+m[k+1,j]+w[i,j]\},\quad 1\le i\le j\le n$$

$$m[i,i-1]=0,\quad i=1,2,...,n$$

例子

$$m[1,1]=0.16,\quad m[2,2]=0.34,\quad m[3,3]=0.17,\quad m[4,4]=0.31,\quad m[5,5]=0.17$$

当i=j时，m[i][j]=min(m[i][i-1] + m[j+1][j] + w[i][j])=w[i][j]

$$m[1,2]=\min\{m[2,2],\ m[1,1]\}+0.48=0.64\quad k=2$$
$$m[2,3]=\min\{m[3,3],\ m[2,2]\}+0.49=0.66\quad k=2$$
$$m[3,4]=\min\{m[4,4],\ m[3,3]\}+0.43=0.60\quad k=4$$
$$m[4,5]=\min\{m[5,5],\ m[4,4]\}+0.42=0.59\quad k=4$$

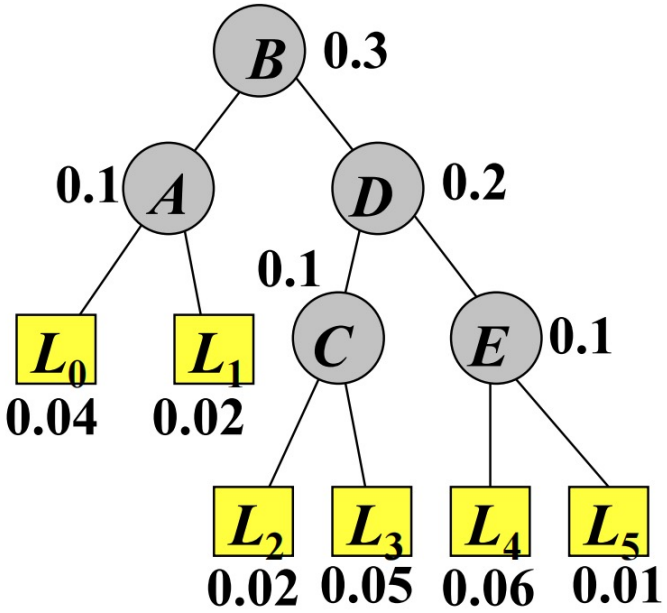
$$m[3,5]=\min\{m[4,5],\ m[3,3]+m[5,5],\ m[3,4]\}+0.54=0.88\quad k=4$$

计算规模为 5 的问题，也是原始问题.

$$m[1,5]=\min\{m[2,5],m[1,1]+m[3,5],m[1,2]+m[4,5],$$
$$m[1,3]+m[5,5],m[1,4]\}+1=2.04\quad k=2$$

追踪解

1. m[1, 5]=2.04,k=2,可知B为根结点时该树的平均比较次数最小，因此B为根结点。
2. B为根结点，左子树只有可能是A。
3. 右子树对应于输入S[3, 5]，计算m[3, 5]=0.88时，k=4,因此D为左子树的根节点，这样就构造出最优二分检索树。



复杂度分析

时间复杂度：O（n^3）

- **子问题数量**：共有 $O(n^2)$ 个子问题（每个 $[i,j]$ 对应一个子问题）。
- **每个子问题的计算代价**：每个子问题需要遍历 $O(n)$ 个可能的根节点 k。
- **递推公式中的加法操作**：计算 $m[i][j]$ 时，需要比较所有 k 的取值，导致每个子问题的计算时间为 $O(n)$。

空间复杂度：O（n^2）

- **存储二维表 $m[i][j]$**：需要 $O(n^2)$ 的空间。
- **预计算 $w[i][j]$**：可通过动态规划或前缀和优化为 $O(n^2)$，不影响总空间复杂度。

谢谢