華中科技大學
计算机科学与技术学院
School of Computer Science &Technology，HUST

算法设计与分析

刘渝
Liu_yu@hust.edu.cn

2023秋季-华科-计算机
22级CS

算法分析与设计
第一十五章
动态规划

华中科技大学计算机科学与技术学院
School of Computer Science & Technology, HUST

最优化问题

这一类问题的可行解可能有很多个。每个解都有一个值，我们希望寻找具有最优值的解（最小值或最大值）

—— 这种找最优解的问题通常称为**最优化问题**

注：这里，我们称这个解为问题的一个最优解（an optimal solution），而不是the optimal solution，因为最优解也可能有多个

Example

给定一个“函数”F(X)，以及“自变量”X，X应满足的一定条件，求X为怎样的值时，F(X)取得最大值或最小值。

这里，F(X)称为“目标函数”，X应满足的条件称为“约束条件”。约束条件可用一个集合D表示为：X∈D。

求目标函数F(X)在约束条件X∈D下的最小值或最大值问题，就是一般最优问题的数学模型，可以用数学符号简洁地表示成：

Min F(X) 或 Max F(X)

分类

根据描述约束条件和目标函数的数学模型的特性和问题的求解方法的不同，可分为：线性规划、整数规划、非线性规划、动态规划等问题。而研究解决这些问题的科学一般就总称之为最优化理论和方法。

在运筹学领域有对最优化理论更深入的研究

本章介绍动态规划（**Dynamic Programming**，简称**DP**）

造表

动态规划(Dynamic Programming)与分治法有点相似：都是通过组合子问题的解来求解原问题。

分治法：要求将问题划分为互不相交的子问题，递归地求解子问题，然后将子问题的解组合成原问题的解。如果子问题有重叠，则递归求解中就会反复地求解这些公共子问题，造成算法效率的下降。

动态规划：与分治不同，适用于有子问题重叠的情况，即不同的子问题具有公共的子子问题。动态规划算法对每个这样的子子问题只求解一次，将其解保存在一个表格中，再次碰到时，无需重新计算，只从表中找到上次计算的结果加以引用即可，避免了对子问题的重复计算。

步骤

1. 刻画一个最优解的结构特征；

2. 递归地定义最优解的值；

3. 计算最优解的值；

4. 利用计算出的信息，构造一个最优解

步骤

注：

1）前三步是动态规划算法求解问题的基础。如果仅需要一个最优解的值，而非解本身，可以忽略步骤4。

2）如果要求解本身，就要做步骤4，这通常需要在执行步骤3的过程中维护一些额外的信息，以便用来构造一个最优解。

3）第三步通常采用**自底向上**的方法计算最优解；

目　录

钢条切割

Serling公司购买长钢条，将其切割为短钢条出售。不同的切割方案，收益是不同的，怎么切割才能有最大的收益呢?

- 假设，切割工序本身没有成本支出。
- 假定出售一段长度为i英寸的钢条的价格为p_i（i=1,2,...），下面是一个价格表P。

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

长度为i英寸的钢条可以带来p_i美元的收益

分析

如果长度为n英寸的钢条的价格p_n足够大，则可能完全不需要切割，出售整条钢条是最好的收益。

但由于每个p_i不同，可能切割后出售会更好一些。

思考：按价格从高到低分段出售，即先卖p_i最大的段，然后再卖p_i较小的段。是否可以获得最好的收益？（贪心策略）

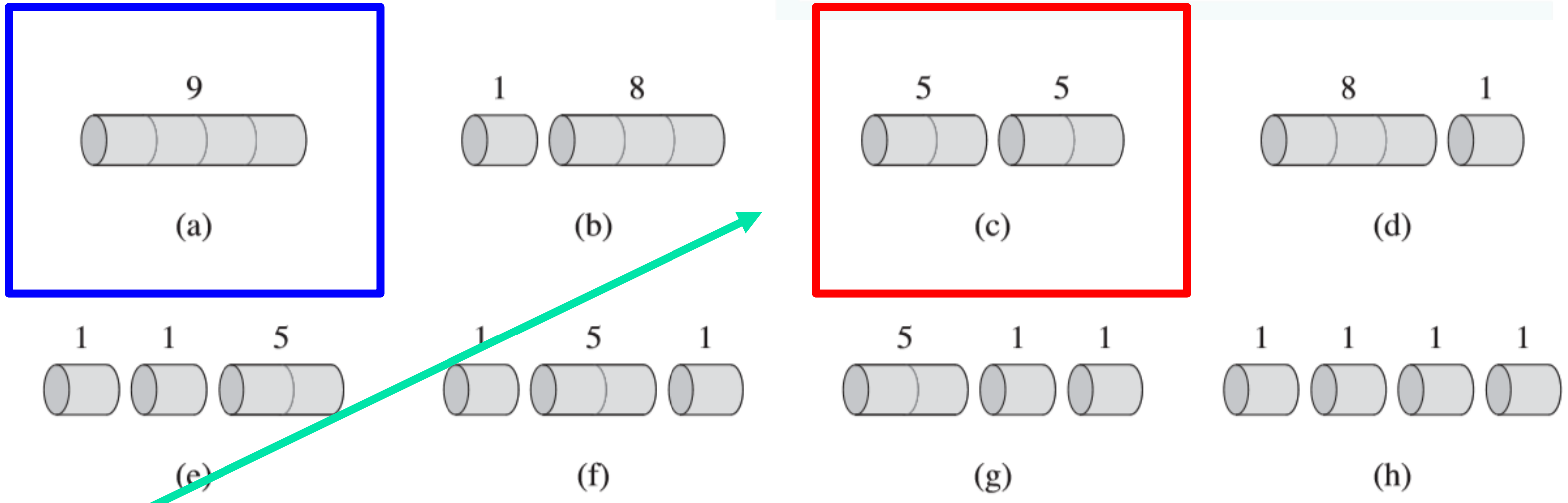
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

分析

考虑如下n=4的情况。

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

4英寸的钢条所有可能的切割方案有以下几种：



最优方案：方案c，将4英寸的钢条切割为两段，每段长2英寸，**此时可产生的收益为10**，为最优解。

分析

- 长度为n英寸的钢条共有2^{n-1}种不同的切割方案。
  - 每一英寸都可切割，共有n-1个切割点
- 如果一个**最优解**将总长度为n的钢条切割为k段，每段的长度为i_j（$1 \le j \le k$），则有：$n=i_1+i_2+\dots+i_k$

  得到的**最大收益**为：$r_n=p_{i1}+p_{i2}+\dots+p_{ik}$

分析

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

如，从价格表可得以下基本方案：

r_1=1，切割方案1=1（无切割）

r_2=5，切割方案2=2（无切割）

r_3=8，切割方案3=3（无切割）

r_4=10，切割方案4=2+2

r_5=13，切割方案5=2+3

r_6=17，切割方案6=6 （无切割）

r_7=18，切割方案7=1+6或7=2+2+3

r_8=22，切割方案8=2+6

r_9=25，切割方案9=3+6

r_{10}=30，切割方案10=10 （无切割）

分析

对于长度为n（n≥1）的钢条，设r_n是最优切割的收益，那么该如何获得该最优切割?

- 对**最优切割**，设某次切割在位置i，将钢条分成长度为i和n-i的两段，令r_i和r_{n-i}分别是这两段的最优子切割收益，则有：

$$r_n = r_i + r_{n-i}$$

- **一般情况**，任意切割点j都将钢条分为两段，长度分别为j和n-j，1≤j≤n。令r_j和r_{n-j}分别是这两段的最优切割收益，则该切割可获得的最好收益是：$r'_n = r_j + r_{n-j}$

分析

$$r_n = r_i + r_{n-i} \quad <--> \quad r'_n = r_j + r_{n-j}$$

- 这样的j有n种选择(包括不切割)，而最优切割是能够获得最终最大收益的切割方案，所以有：

$$r_n = \max_j \{ r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots r_j + r_{n-j}, \ldots, r_{n-1} + r_1, p_n \}$$

- 即，j位置切割后，将两段钢条看成两个独立的钢条切割问题实例。若分别获得两段钢条的最优切割收益r_j和r_{n-j}。

最优子结构

$$r_n = r_i + r_{n-i} \quad \text{<-->} \quad r'_n = r_j + r_{n-j}$$

- j位置切割后，将两段钢条看成两个独立的钢条切割问题实例。若分别获得两段钢条的最优切割收益r_j和r_{n-j}。
- **则原问题的解就可以通过组合这两个相关子问题的最优子解获得，而体现该问题最优解的一个重要性质：最优子结构性**

如果$r_n = r_i + r_{n-i}$是最优切割收益，则r_i、r_{n-i}是相应子问题的最优切割收益

朴素递归求解

简化：将钢条从左边切割下长度为i的一段，然后只对右边剩下的长度为n-i的一段继续进行切割（递归求解），但对左边的一段不再进行切割。

$$r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\}$$

即，此时，原问题的最优解只包含一个相关子问题（右端剩余部分）的解，而不是两个。

朴素递归求解 自顶向下

其中，p是价格数组，n是钢条长度。

若n=0，则收益为0。

CUT-ROD(p, n)

```
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

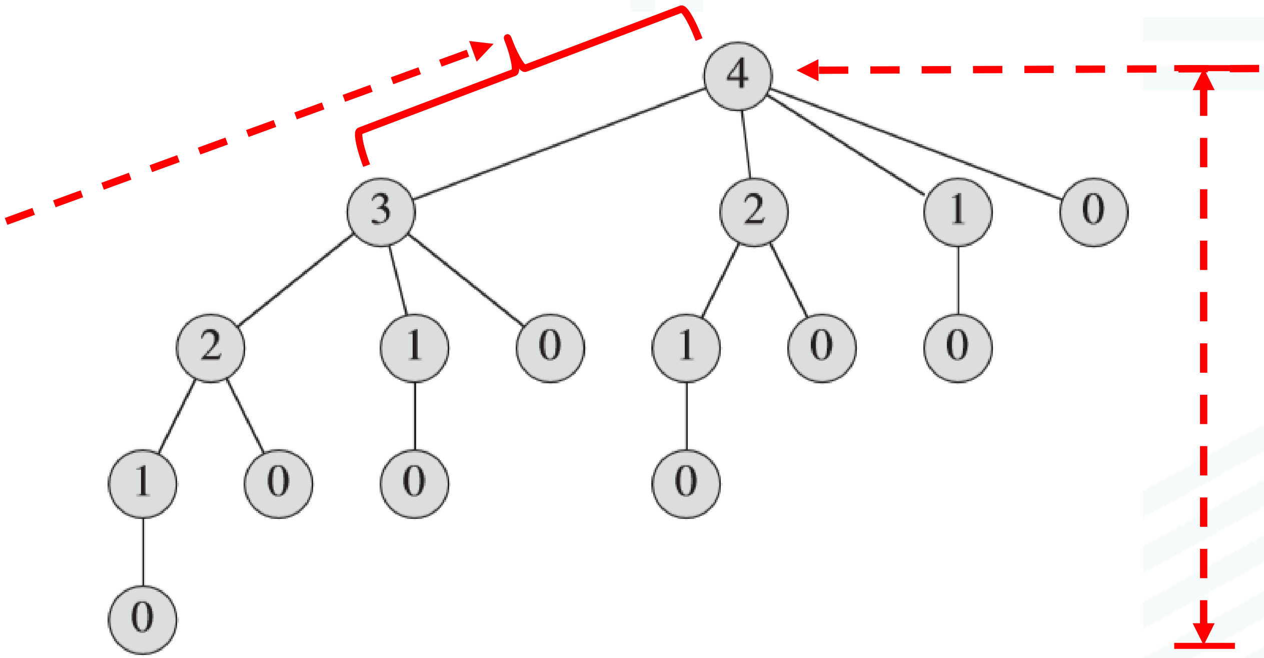
该过程的效率很差：

- 存在一些相同长度的子问题，在CUT-ROD的整个执行过程中反复地做一些计算（递归调用）。

朴素递归求解 自顶向下

如n=4， CUT-ROD的递归执行过程可以用递归调用树表示为：

从父结点s到子结点t的边表示从钢条左端切下长度为s-t的一段，然后继续递归求解剩余规模为t的子问题。



结点中的数字为对应子问题的规模

从根结点到叶结点的一条路径对应长度为n的钢条的2^{n-1}种切割方案之一。

一般来说，这棵递归调用树有2^n个结点，其中有2^{n-1}个叶结点。

朴素递归求解 自顶向下

令T(n)表示对规模为n的问题， CUT-ROD的调用次数，则有：

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) .$$

- 可以证明：

$$T(n) = 2^n$$

动态规划求解

动态规划方法将仔细安排求解顺序，对每个子问题只求解一次，并将结果保存下来。如果再次需要此子问题的解，只需查找保存的结果，而不必重新计算。

- 动态规划方法需要付出额外的空间保存子问题的解，是一种典型的时空权衡（time-memory trade-off）。

动态规划求解

- **可获得的改进**：动态规划方法节省了时间，可以将一个指数时间的解转化为一个多项式时间的解。如果子问题的数量是n的多项式函数，而且可以在多项式时间内求解出每个子问题，则动态规划方法的总运行时间就是多项式阶的。

动态规划求解

（1）带备忘的自顶向下法（top-down with memoization)

- 形式是**递归**的，但处理过程中会保存每个子问题的解。
- 过程每次被调用时，首先检查当前子问题是否已经被计算过并保存过子解。
  - 如果是，则直接返回以前保存的值；
  - 否则，按照一般方式计算该子问题并保存其解

备忘：造一个表，“记住”之前已经计算出来的结果。表的形式可以是一个数组或者是散列表

动态规划求解 带备忘 自顶向下

```
MEMOIZED-CUT-ROD(p, n)
    let r[0..n] be a new array
    for i = 0 to n
        r[i] = -∞
    return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

用辅助数组作为备忘表。

- 初始化为-∞;

动态规划求解 带备忘 自顶向下

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

查表：r[n]≥0表示已经保存过解，直接引用r[n]。

否则，实施计算，r[n]保存当前结果;

动态规划求解

(2) 自底向上法（bottom-up with method）

从最小子问题开始，按照最小子问题、较小子问题、…、较大子问题、原问题的顺序依次求解。一个较大子问题的解通过较小子问题解组合而成，直到原始问题得到答案。

自底向上求解：先求解较小的子问题，当求解一个较大子问题时，它所依赖的更小子问题都已求解完毕，结果已经保存，所以可以直接引用更小子问题的解并组合出它的解。

动态规划求解 自底向上

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

从规模为0的“最小”子问题开始依次求解

动态规划求解 对比

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

递归设计框架

自顶向下

自底向上

迭代设计框架

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

动态规划求解 对比

具有相同的渐近运行时间：$\Theta(n^2)$

通常，**自顶向下法和自底向上法具有相同的渐近运行时间。**

- 但由于自底向上法没有频繁的递归函数调用的开销，所以自底向上法的时间复杂性函数通常具有更小的系数。
- 而在某些特殊情况下，自顶向下法可能没有递归处理所有可能的子问题（剪枝）而减少工作量。

动态规划求解 对比

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

“硬”求解。整个过程对存在的重复子问题的计算，效率低下。

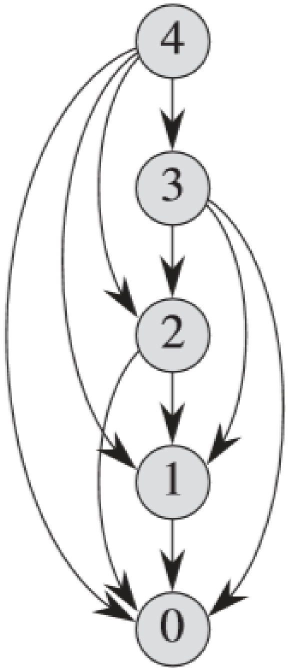
```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

在求解的过程中记录了子问题的解。当再次遇到相同子问题时，仅“引用”以前的计算结果，避免了重复计算。只有当碰到新的子问题时才递归计算。

子问题图

当思考用动态规划求解一个问题时，应该弄清楚所涉及的子问题与子问题之间的依赖关系，可用子问题图描述：

子问题图用于描述子问题与子问题之间的依赖关系

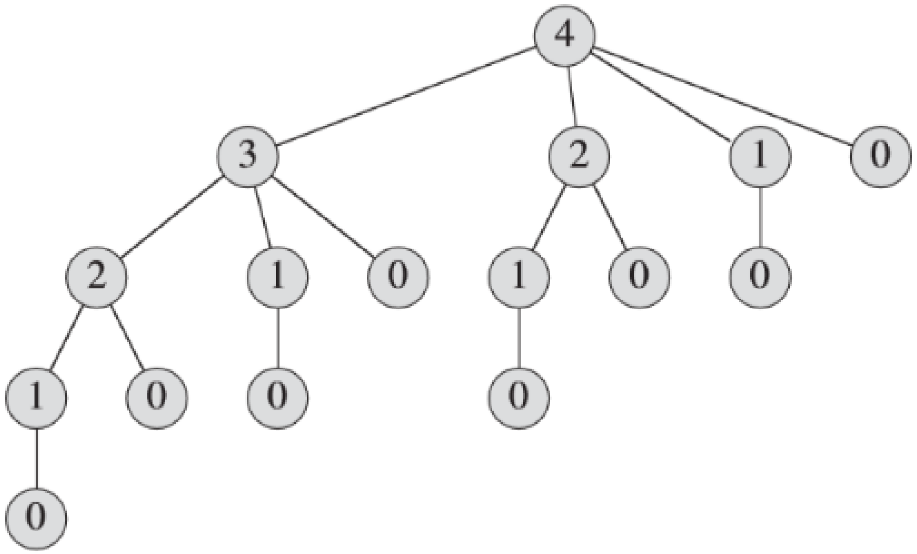


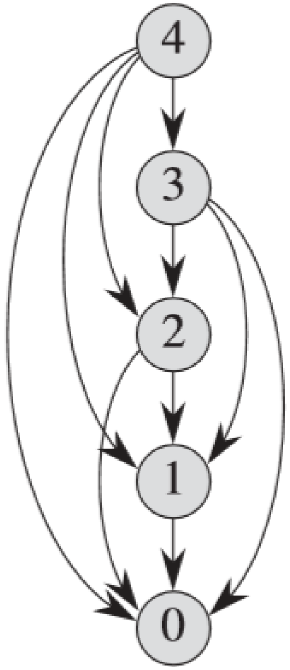
- 子问题图是一个有向图，每个顶点唯一地对应一个子问题。
- 若求子问题x的最优解时需要直接用到子问题y的最优解，则在子问题图中画一条从子问题x的顶点到子问题y的顶点的有向边。

n=4时，钢条切割问题的子问题图。顶点的标号给出了子问题的规模。有向边(x,y)表示当求解子问题x时需要子问题y的解。

子问题图

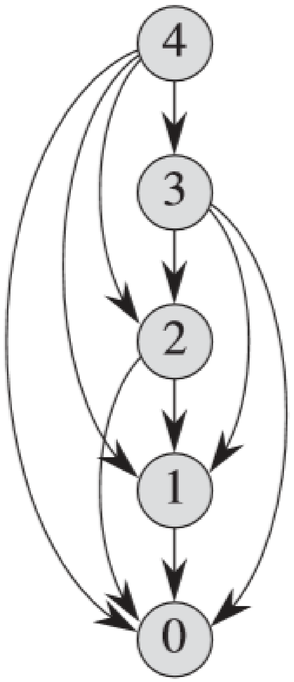
递归树中所有对应**相同子问题**的结点合并为子问题图中的一个单一顶点，相关的边都从父结点指向子结点





子问题图

自底向上的动态规划方法处理子问题图中顶点的顺序为

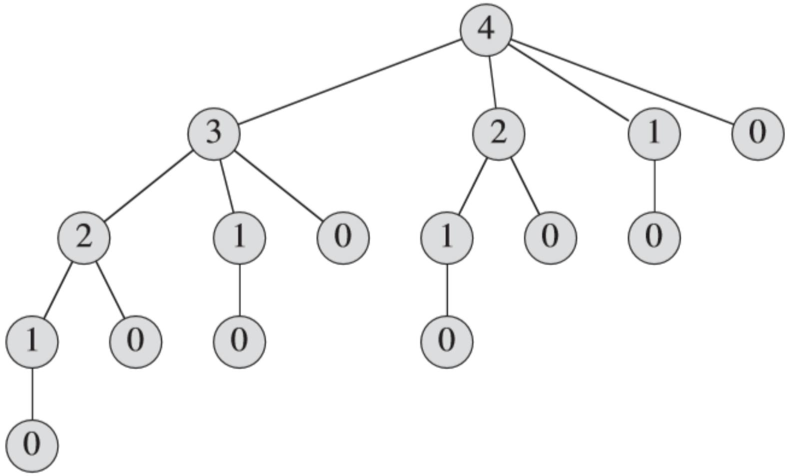


1）对一个给定的子问题x，在求解它之前先求解邻接至它的子问题。

2）对于任何子问题，仅当它依赖的所有子问题都求解完成了才会求解它 — 逆拓扑序，深度优先原则进行处理。

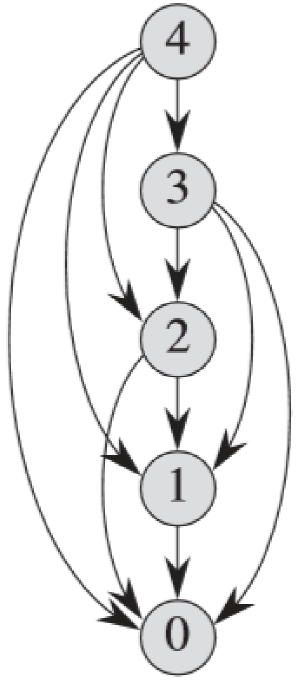
子问题图

基于子问题图估算算法的运行时间：



算法运行时间等于所有子问题求解的时间之和。子问题图中，子问题对应顶点，子问题的数目等于顶点数。一个子问题的求解时间与子问题图中对应顶点的“出度”成正比

因此，一般情况下，动态规划算法的运行时间与顶点和边的数量至少呈线性关系。



重构解

CUT-ROD算法给出了最优收益，但怎么切割的呢？扩展上述算法，在求出最优收益之后，求出**切割方案**

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

数组s用于记录切割点。

- 对长度为j的钢条，求出其最优收益，并记录获得最优收益时左侧第一个切割点的位置

输出

对已知价格表p和钢条长度n，下述过程能够计算出长度数组s[1..n]，并输出完整的最优切割方案：

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- PRINT-CUT-ROD-SOLUTION(p,7)：1，6
- PRINT-CUT-ROD-SOLUTION(p,10)：10

构建过程

根据问题的性质构造递推关系式，形成有效的计算过程

- 首先给出子问题的定义和递推关系式：$r_n = \max\limits_{1\le i\le n}\{p_i + r_{n-i}\}$
- 造好备忘表，不同的问题表的结构不同： r[0..n]

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

作业1：

華中科技大學
计算机科学与技术学院
School of Computer Science &Technology，HUST

- 算法设计题：
  - 15.1-3

矩阵链乘法

两个矩阵的乘运算：

已知A为$p \times r$的矩阵，B为$r \times q$的矩阵，则A与B的乘积是一个$p \times q$的矩阵，记为C：

$$C = A_{p\times r} \times B_{r\times q} = (c_{ij})_{p\times q},$$

其中，

$$c_{ij} = \sum_{1\le k\le r} a_{ik} b_{kj}, \quad i = 1,2,..p, \quad j = 1,2,\ldots, q$$

每个c_{ij}的计算需要r次乘法（另有r-1次加法，这里仅考虑元素的标量乘法），C中共有$p \times q$个元素，所以计算C共需要$p \times q \times r$次标量乘法运算。

原版伪代码

MATRIX-MULTIPLY(A, B)

if $A.columns \neq B.rows$
    error "incompatible dimensions"
else let C be a new $A.rows \times B.columns$ matrix
    for $i = 1$ **to** $A.rows$
        for $j = 1$ **to** $B.columns$
            $c_{ij} = 0$
            for $k = 1$ **to** $A.columns$
                $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
    return C

注：只有两个矩阵"相容"(compatible)才能相乘。

三重循环结构

问题分析

n个要连续相乘的矩阵构成一个矩阵链$<A_1,A_2,\ldots,A_n>$,要计算这n个矩阵的连乘乘积：$A_1A_2\ldots A_n$，称为**矩阵链乘**问题。

- 矩阵链乘不满足交换律：$A_1A_2A_3 \neq \mathbf{A_3A_2A_1}$
- 矩阵链乘满足**结合律**，所以矩阵链乘相当于在矩阵之间**加括号**。不同的加括号方案导出不同的矩阵链乘**计算模式**

问题分析

已知四个矩阵 A_1, A_2, A_3, A_4，根据不同的加括号方式，乘积$A_1A_2A_3A_4$有五种不同的计算模式：

$(A_1(A_2(A_3A_4)))$　　$(A_1((A_2A_3)A_4))$　　$((A_1A_2)(A_3A_4))$

$((A_1(A_2A_3))A_4)$　　$(((A_1A_2)A_3)A_4)$

问题分析

尽管不同的**计算模式**最后得到的结果是一样的，但计算过程中产生的代价是不同的!

- **问题：怎么求代价最小的计算模式——最优计算模式呢?**

问题分析

如，设有三个矩阵的链<A_1,A_2,A_3>，维数分别为10×100,100×5,5×50。

1）如果按**($(A_1A_2)A_3$)**的次序完成乘法，则A_1与A_2乘需要10×100×5 =5000次**标量乘法**运算，得一10×5的中间结果矩阵，再继续与A_3相乘，又需要10×5×50=2500次标量乘法运算，总共为**7500次**标量乘法运算。

2）如果按**($A_1(A_2A_3)$)**的次序完成乘法，则A_2与A_3乘需要100×5×50 =25000次标量乘法运算，得一100×50的中间结果矩阵，A_1与之再次相乘，又需要10×100×50=50000次标量乘法运算，总共为**75000次**标量乘法运算。

可见，上述两种计算模式的计算量**相差10倍**！

问题

给定n个矩阵的链，记为$<A_1,A_2,\ldots,A_n>$，其中$i=1,2,\ldots,n$，矩阵A_i的维数为$p_{i-1}\times p_i$。求一个完全“括号化方案”，使得计算乘积$A_1A_2\ldots A_n$所需的标量乘法次数最小。

令p(n)表示n个矩阵链乘时可供选择的括号化方案的数量。则有：

$$P(n)=\begin{cases}1 & \text{if } n=1\,,\\ \sum_{k=1}^{n-1}P(k)P(n-k) & \text{if } n\geq 2\,.\end{cases}$$

可以证明：$P(n)=\Omega(2^n)$

显然，**穷举**所有可能的括号化方案是不可取的。

最优子结构

用记号$\mathbf{A_{i,j}}$表示子问题$A_iA_{i+1}…A_j$通过加括号后得到的一个最优计算模式，且该计算模式下的最大区间恰好在A_k与A_{k+1}之间分开：

$$\overline{(A_iA_{i+1}\ldots A_k)}\,\overline{(A_{k+1}\ldots A_j)}$$

则必须有：$\overline{(A_iA_{i+1}\ldots A_k)}$ 必是“前缀子链”$A_iA_{i+1}…A_k$的一个最优的括号化子方案，记为$\mathbf{A_{i,k}}$；同理$\overline{(A_{k+1}A_{k+2}\ldots A_j)}$ 也必是“后缀子链”$A_{k+1}A_{k+2}…A_j$的一个最优的括号化子方案，记为$\mathbf{A_{k+1,j}}$。

证明：反证法

如若不然，设$A'_{i,k}$是$<A_i,A_{i+1},\ldots A_k>$一个代价更小的计算模式，则由$A'_{i,k}$和$A_{k+1,j}$构造计算过程$A'_{i,j}$，代价将比$A_{i,j}$小，这与$A_{i,j}$是最优链乘模式相矛盾。

对$A_{k+1,j}$亦然。

——这一性质称为（该问题的）**最优子结构性**：若$A_{i,j}$是最优的计算模式，则其中的$A_{i,k}$、 $A_{k+1,j}$也都是相应子问题的最优计算模式。

递归求解方案

整体的最优括号化方案可以通过寻找使最终标量乘法次数最小的两个最优括号化子方案得到。

形如：$(A_1A_{i+1}\ldots A_k)(A_{k+1}\ldots A_n)$

到哪里找这个使标量乘法次数最小的k呢?

递归求解方案

1）递推关系式

表的结构：i~j代表区间，但数据结构是二维数组。

令 m[i,j] 为计算矩阵链$A_{i,j}$所需的标量乘法运算次数（最小值）,则有

$$m[i,j]=\begin{cases}0 & \text{if } i=j\ ,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j\ .\end{cases}$$

递推方程

递归求解方案

1）递推关系式

- 对任意的k($i \le k < j$)分开的子乘积，$A_{i,k}$是一个$p_{i-1} \times p_k$的矩阵，$A_{k+1,j}$是一个$p_k \times p_j$的矩阵 。结果矩阵$A_{i,j}$是$A_{i,k}$和$A_{k+1,j}$最终相乘的结果。
- 对于某个k，要想使得m[i,j]最小，必然是：m[i,j]等于计算子乘积$A_{i,k}$最小代价m[i,k] + 计算子乘积$A_{k+1,j}$的最小代价m[k+1,j] + 这两个子矩阵最后相乘的代价$p_{i-1}p_kp_j$。而这样的k有j-i种可能性,取能使m[i,j]最小的k。
- m[1,n]是计算$A_{1,n}$的最小代价。

递归求解方案

2）构造解

再设**s[i,j]**，记录使m[i,j]取最小值的k，后面可以根据s求出最优链乘模式。

下述过程**MATRIX-CHAIN-ORDER**采用自底向上表格法计算n个矩阵链乘的最优模式。

递归求解方案

3）伪代码

```
MATRIX-CHAIN-ORDER(p)

n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

递归求解方案

3）伪代码

输入序列$p=<p_0,p_1,\ldots,p_n>$是n个矩阵的维数表示，矩阵A_i的维数是$p_{i-1}\times p_i$，$i=1,2,\ldots,n$。

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n

for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

辅助表m保存所有m[i,j]的代价，s[i,j]记录使 m[i,j] 时取得最优值时的分割点k

递归求解方案

3）伪代码

自底向上完成m[i,j]的计算：在m[i,i] =0的基础上，求出所有**m[i,j]**。最后算出m[1,n]。

m[1,n]是计算$A_{1,n}$的最小代价

MATRIX-CHAIN-ORDER(p)

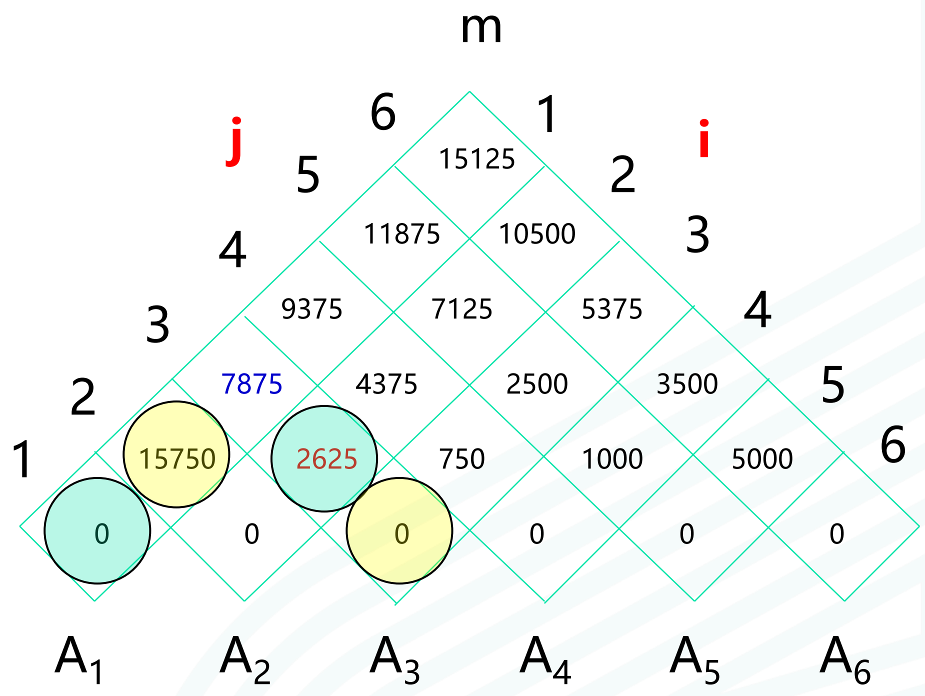
```
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                      // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j\,. \end{cases}$$

递归求解方案

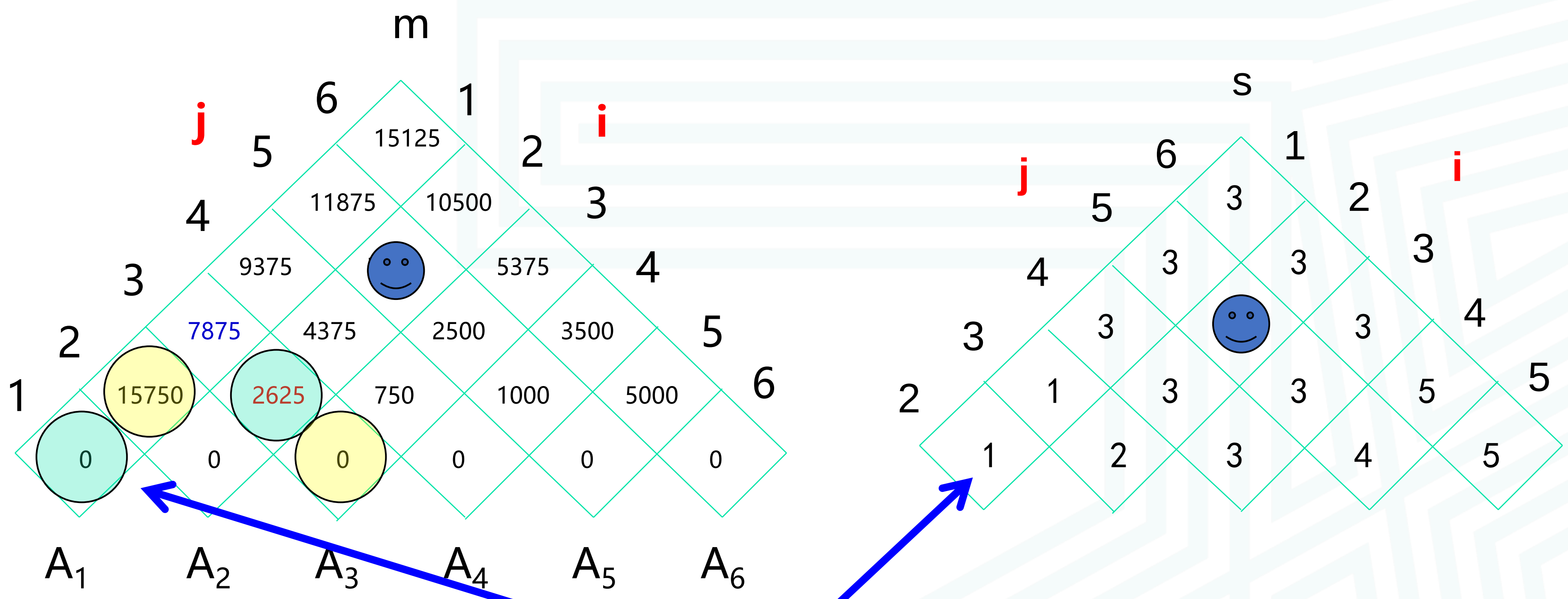
$$
\begin{aligned}
m[1,3] = \min\{&m[1,1]+m[2,3]+30\times35\times5,\\
&m[1,2]+m[3,3]+30\times15\times5\}\\
= \min\{&0+2625+5250，15750+0+2250\}\\
= 7875&
\end{aligned}
$$

矩阵	维数
A1	30×35
A2	35×15
A3	15×5
A4	5×10
A5	10×20
A6	20×25



递归求解方案

矩阵	维数
A1	30×35
A2	35×15
A3	15×5
A4	5×10
A5	10×20
A6	20×25



$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j\ . \end{cases}$$

$m[i,i] = 0$

$m[i,i+1] = p_{i-1}p_ip_{i+1}$

$s[i,i+1] = i$

时间复杂度

算法的主体由一个三层循环构成。最外层循环执行n-1次，内层两个循环都至多执行n-1次，所以MATRIX-CHAIN-ORDER的算法复杂度是$\Omega(n^3)$。

另，算法需要$\Theta(n^2)$的空间保存m和s。

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

递归求解方案

4）构造最优解

- s[i,j]记录了$A_iA_{i+1}\ldots A_j$的最优括号化方案中“首个”加括号的位置点k。
  - 基于s[i,j]，对$A_iA_{i+1}\ldots A_j$的括号化方案是：

$$(A_iA_{i+1}\ldots A_{s[i,j]})\quad(A_{s[i,j]+1}\ldots A_j)$$

- $A_{1\ldots n}$的最优方案中最后一次矩阵乘运算是：

$$(A_{1\ldots s[1,n]})\quad(A_{s[1,n]+1\ldots n})$$

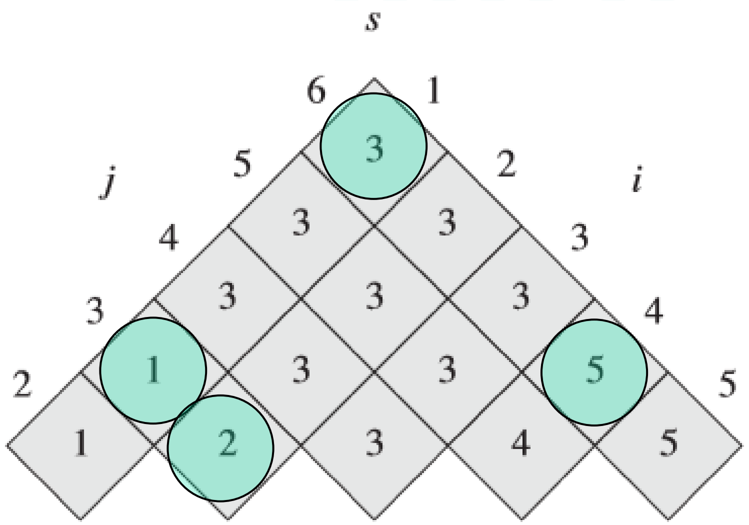
- 用递归的方法求出$A_{1\ldots s[1,n]}$、$A_{s[1,n]+1\ldots n}$及其它所有子问题的最优括号化方案。

最优计算模式

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

例： PRINT-OPTIMAL-PARENS(s,1,6)

$((A_1(A_2A_3))((A_4A_5)A_6))$



总结：

華中科技大學
计算机科学与技术学院
School of Computer Science &Technology，HUST

动态规划：与分治不同，适用于有子问题重叠的情况。

子问题重叠的解决方案：造表！

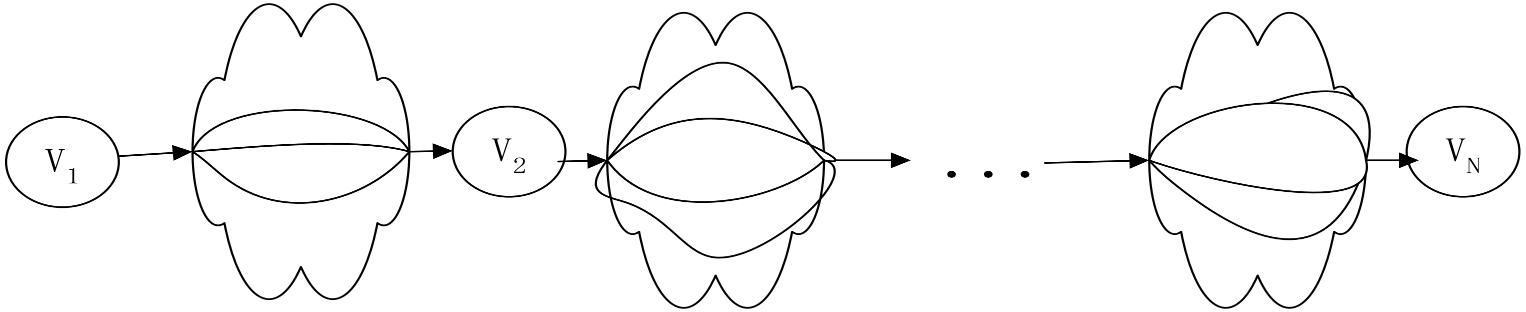
动态规划算法对重复的子问题的求解只做一次，并将其最优的解保存在一个表格中，再次碰到时，无需重新计算，只从表中找到上次计算的结果加以引用即可，避免了不必要的计算工作。

作业2：

- 算法设计与计算题：
  - 15.2-1

一般方法

动态规划(dynamic programming)是运筹学的一个分支，是求解决策过程(decision process)最优化的数学方法。



动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其它方法求解更为方便。

多阶段决策过程

- **阶段**：事件的发展过程分为若干个相互联系的阶段。事件的发展总是从初始状态开始，依次经过第一阶段、第二阶段、第三阶段、…，直至最后一个阶段结束。
- **阶段变量**：描述阶段的变量称为阶段变量。

多阶段决策过程

- **状　　态**：状态表示每个阶段的自然状况或客观条件，用一组变量的值表示。它既是当前阶段某途径的起点，也是前面阶段某途径的终点。
- **状态变量**：过程的状态通常可以用一个或一组数来描述，称为状态变量。
  - 如果用一组数表示，状态变量就是多维的，用向量表示。
- **状态集合**：当过程按所有可能不同的方式发展时，过程的各个段中，状态变量将在某一确定的范围内取值，状态变量取值的集合称为状态集合。

多阶段决策过程

- **决　　策**：在一个阶段的状态给定以后，从该状态演变到下一阶段某个状态的一种选择称为**决策**。也就是在两个阶段间选择发展途径的行为。
- 决策变量：描述决策的变量称**决策变量**。用一个数或一组数表示。不同的决策，决策变量对应着不同的数值

一般情况下，从i阶段发展到i+1阶段（$0 \le i < n$）可能有多种不同的途径，而事件必须从中选择一条途径往前进展。使过程从一个状态演变到下一状态。

多阶段决策过程

- **决　　策**：在一个阶段的状态给定以后，从该状态演变到下一阶段某个状态的一种选择称为**决策**。也就是在两个阶段间选择发展途径的行为。
- 决策变量：描述决策的变量称**决策变量**。用一个数或一组数表示。不同的决策，决策变量对应着不同的数值
- 决策序列：事件的发展过程之中需要经历n个阶段，需要做n次“决策”。这些“决策”就构成了事件整个发展过程的一个决策序列。
- 多阶段决策过程：具备上述性质的过程称为多阶段决策过程(multistep decision process)。

多阶段决策过程

一般情况下，从i阶段发展到i+1阶段（$0 \le i < n$）可能有多种不同的途径，而事件必须从中选择一条途径往前进展。使过程从一个状态演变到下一状态。

求解多阶段决策过程问题就是求取事件发展的决策序列。

多阶段决策过程

无后效性：对任意的阶段i，阶段i以后的行为仅依赖于i阶段的状态，而与i阶段之前过程是如何达到这种状态的方式无关，这种性质称为**无后效性**。

这一性质意味着过程的历史只能通过当前的状态去影响它的未来发展，而不能直接作用于未来的发展。如果某一阶段的状态已确定，则在这一阶段以后，过程的发展就不再受这阶段以前各阶段状态的直接影响。

因满足无后效性，故在每个阶段选择决策时只需考虑当前的状态而无须考虑过程的历史。各阶段的决策都确定时，整个过程也就确定了。

这是一个问题的解可由其直接子问题的解“组合”而成的基础。

多阶段决策过程

状态转移方程：阶段之间状态变量值的变化存在一定的关系。

如果给定i阶段的状态变量x(i)的值后，第i+1阶段的状态变量x(i+1)就可以完全确定，即x(i+1)的值随x(i)和第i阶段的决策u(i)的值变化而变化，可以把这一关系看成(x(i)，u(i))与x(i+1)的函数关系，表示为：

$$x(i+1)=T_i(x(i),u(i))$$

—— 这种从i阶段到i+1阶段的状态转移规律称为状态转移方程。

最优化问题

每一决策都附有一定的“成本”，决策序列的成本是序列中所有决策的成本之和。

设从阶段i到阶段i+1有p_i种不同的选择，则从阶段1至阶段n共有$p_1p_2……p_n$种不同的途径，每个途径对应一个决策序列。

问：这些途径里面，哪一个的成本最小?

——如何求取最优决策序列?

最优化问题

可行解：从问题的开始阶段到最后阶段每一个合法的决策序列都是问题的一个可行解。

目标函数：用来衡量可行解优劣的标准，通常以函数形式给出。

最优解：能够使目标函数取极值的可行解是最优解。

多阶段决策过程的最优化问题就是求能够获得问题最优解的决策序列——最优决策序列。

求解策略

记问题的决策序列为:（x_1，x_2，…，x_n）,其中，x_i表示第i阶段的决策：

$$S_0 \xrightarrow{x_1} S_1 \xrightarrow{x_2} S_2 \xrightarrow{x_3 \cdots} \cdots\cdots \xrightarrow{x_n} S_n$$

求解策略

1）枚举法

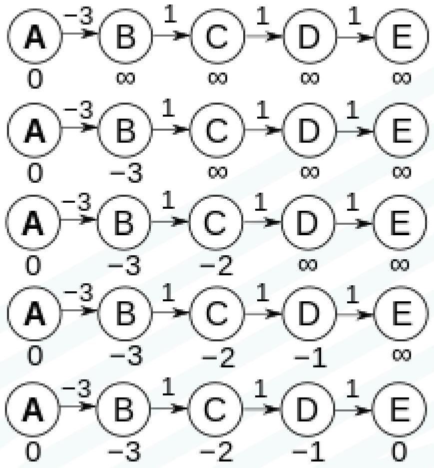
穷举可能的决策序列，从中选取可以获得最优解的决策序列：

若问题的决策序列由n次决策构成，每一阶段分别有p_1、p_2、…、p_n种选择，**则可能的决策序列将有$p_1p_2\ldots p_n$个**。

求解策略

2）动态规划

20世纪50年代初美国数学家R.E.Bellman等人在研究多阶段决策过程的优化问题时，提出了著名的最优化原理(principle of optimality)，从而把多阶段过程转化为一系列单阶段问题，创立了解决这类过程优化问题的新方法——**动态规划**。



求解策略

2）动态规划

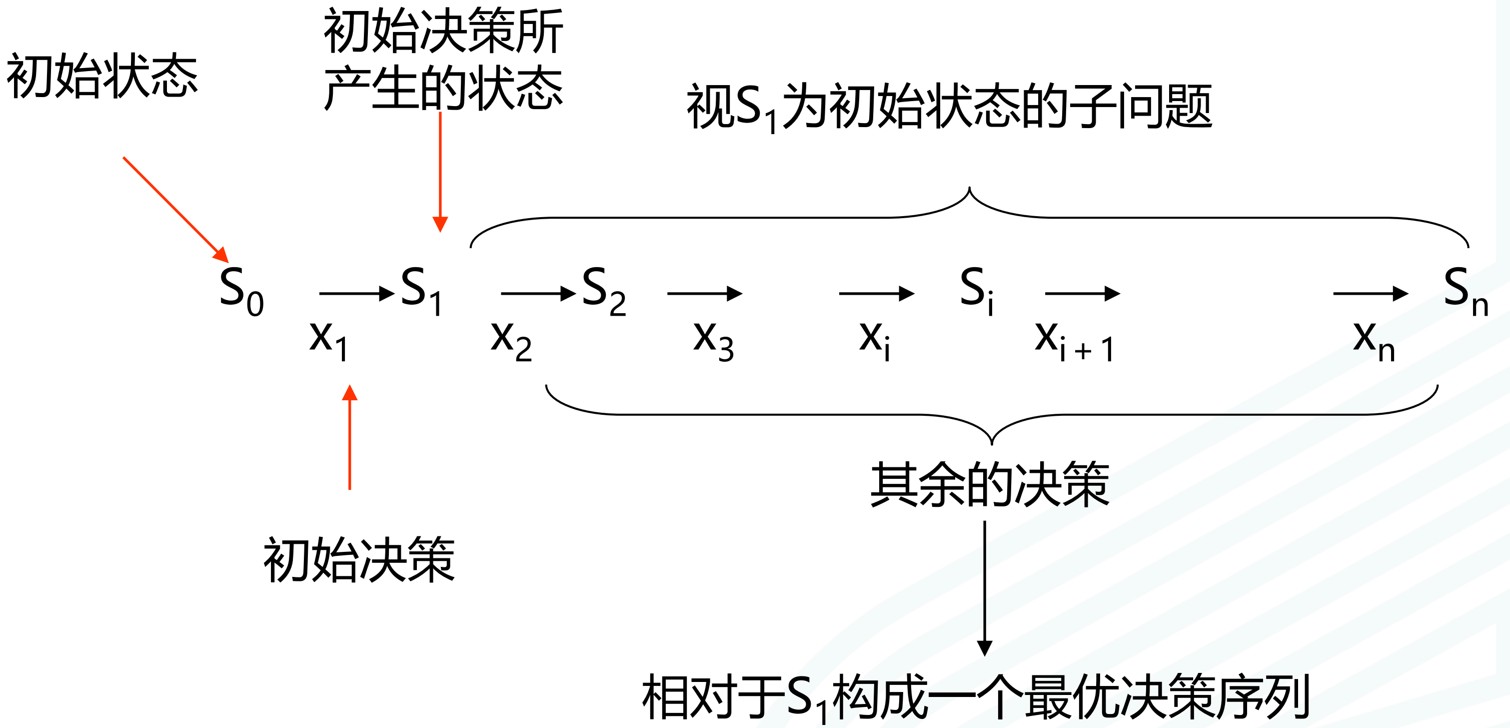
用动态规划策略求解，需要问题满足两个基本性质：

1、无后效性

2、最优化原理（最优子结构性）

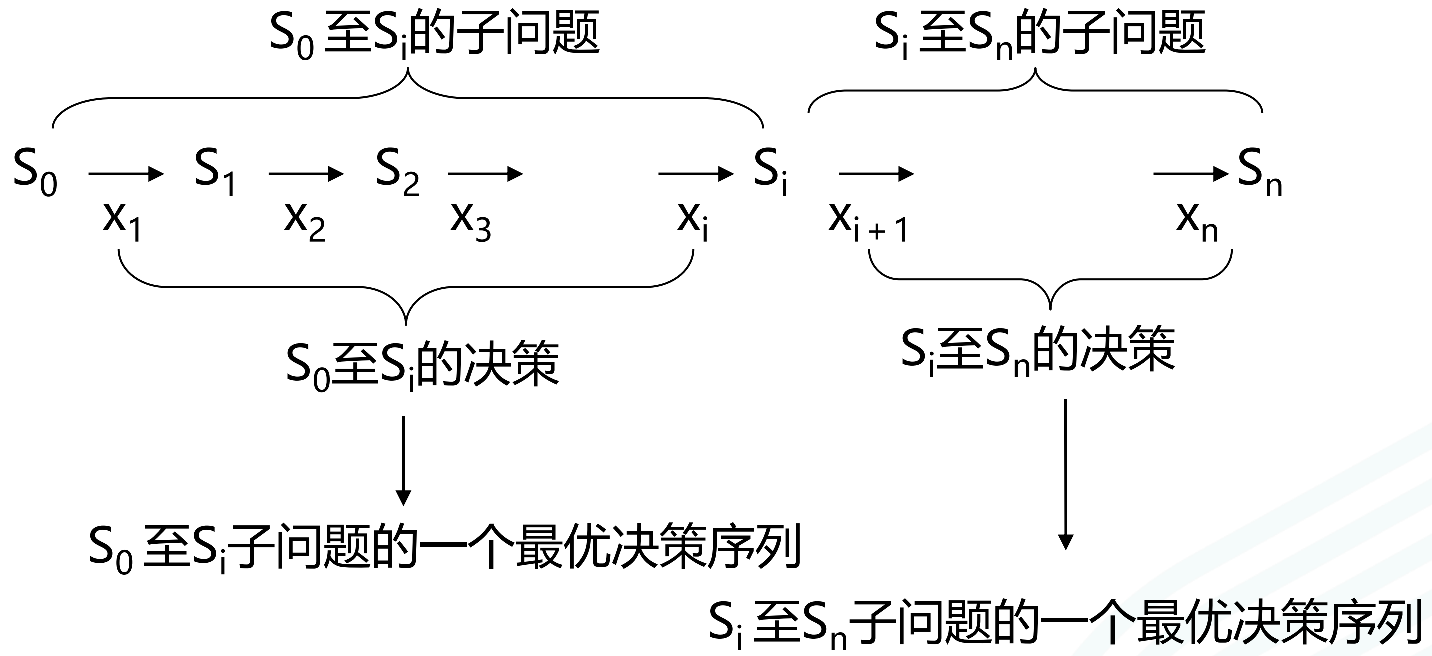
最优化原理

过程的最优决策序列具有如下性质：无论过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。



最优化原理

若全局是最优的，则局部亦是最优的



特征：如果整个序列是最优决策序列，则该序列中的任何一段子序列将是相对于该子序列所对应子问题的最优决策子序列（最优子结构）

Example

若$v_1v_2v_3$……v_n是从节点v_1到节点v_n的最短路径。则：

- v_2v_3……v_n是从v_2到v_n的最短子路径；
- v_3……v_n是从v_3到v_n的最短子路径；

……

推广：

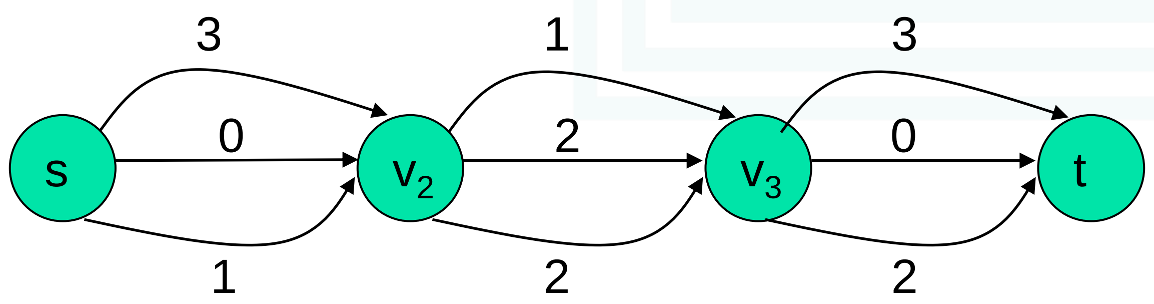
对$v_1v_2v_3$……v_n中的任意一段子路径：

v_pv_{p+1}……$v_q(p \le q, 1 \le p, q \le n)$,

将代表从v_p至v_q的最短子路径。

Example

求由s至t的一条路径，使得该路径的长度模4的余数（即Length(s,t) mod 4)最小。



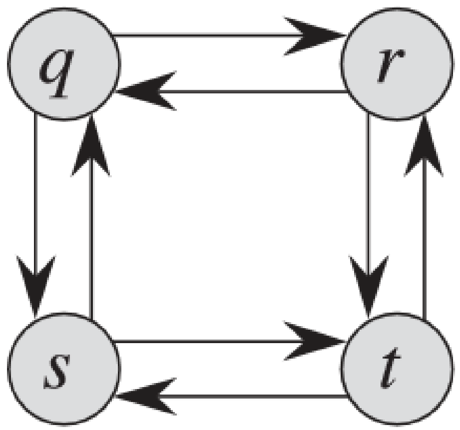
此时，问题的一个最优决策序列是：

$s\text{-}\underline{3}\text{->}v_2\text{-}\underline{2}\text{->}v_3\text{-}\underline{3}\text{->}t$

但最优子结构不成立：最优决策序列上的任一子决策序列相对于当前子问题不是最优的。

Example

最长简单路径问题：在图中找一条从结点q到结点t的边数最多的简单路径。



此例显示了无权有向图最长简单路径问题不具有最优子结构性质。路径 $q \to r \to t$ 是从 q 到 t 的一条最长简单路径，但 $q \to r$ 不是从 q 到 r 的一条最长简单路径，$r \to t$ 同样不是从 r 到 t 的一条最长简单路径

- 最长简单路径问题不具有最优子结构性

求解策略

第一步：证明问题满足最优性原理

所谓“问题满足最优性原理”即：问题的最优决策序列具有最优子结构性。

证明问题满足最优性原理是实施动态规划的必要条件：如果证明了问题满足最优性原理，则说明用动态规划方法有可能解决该问题。

第二步：获得问题状态的递推关系式（即状态转移方程）

能否求得各阶段间状态变换的递推关系式是解决问题的关键。

$f(x_1,x_2,\ldots,x_i) \to x_{i+1}$ 向后递推

或 $f(x_i,x_{i+1},\ldots,x_n) \to x_{i-1}$ 向前递推

回顾：

1）不管是钢管切割问题还是矩阵链乘问题，我们都首先讨论了问题最优解的结构特征，即证明问题的最优解具有最优子结构性：

- 钢管切割问题:若s(1,n)为最优切割方案，则第一次切割（假定切割点在位置k）后得到的两段：s(1,k)和s(k+1,n)也必是最优的子方案。
- 矩阵链乘问题:设$A_{1,n}$是最优括号化方案,“最大子括号”加在A_k后面，则$A_{1,k}$和$A_{k+1,n}$必是子矩阵链的最优括号化方案。

回顾：

2）状态转移方程

- 钢管切割问题： $r_n = \max\limits_{1\le i\le n}\{p_i + r_{n-i}\}$
- 矩阵链乘问题： $m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i\le k<j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j\ . \end{cases}$

求解策略

1）动态规划算法通常用于求解具有某种最优性质的问题。

- 在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。

2）动态规划是一种策略，不是一个具体的算法。

- 因此，动态规划求解不像搜索或数值计算，具有一个标准的数学表达式和明确清晰的解题方法。
- 动态规划针对的最优化问题，由于问题性质的不同，确定最优解的条件的不同，动态规划的设计对不同的问题，有各具特色的解题方法。

技术要点

- 用一个表（备忘）来记录所有已解的子问题的答案。
- 不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。
- 动态规划算法都具有类似的填表模式。

深度理解

利用查表，避免对重复子问题的重复求解，动态规划可以将原来具有指数级复杂度的搜索算法改进成了具有多项式时间的算法，这是动态规划算法的根本目的。

动态规划实质上是一种以空间换时间的技术，它在实现的过程中，需要存储过程中产生的各种状态（中间结果），所以它的空间复杂度要大于其它的算法。

子问题无关

能够用动态规划策略求解的问题，构成最优解的子问题间必须是无关的

所谓子问题无关，是指同一个原问题的一个子问题的解不影响另一个子问题的解，可以各自独立求解。

- 最长简单路径问题
  - 子问题间相关，不能用动态规划策略求解。
- 最短路径问题
  - 子问题不相关，满足最优子结构性，可以用动态规划问题求解。

满足最优子结构性证明

作为构成原问题最优解的组成部分，对应每个子问题的部分应是该子问题的最优解。

"剪切-粘贴"（cut-and-paste）技术：

本质上是**反证法证明**：假定原问题最优解中对应于某个子问题的部分解不是该子问题的最优解，而存在"更优的子解"，那么我们可以从原问题的解中"剪切"掉这一部分，而将"更优的子解"粘贴进去，从而得到一个比最优解"更优"的解，这与最初的解是原问题的最优解的前提假设相矛盾。因此，不可能存在"更优的子解"。

——所以，原问题的子问题的解必须是其最优解，最优子结构性成立。

代价

求解原问题的代价通常就是：

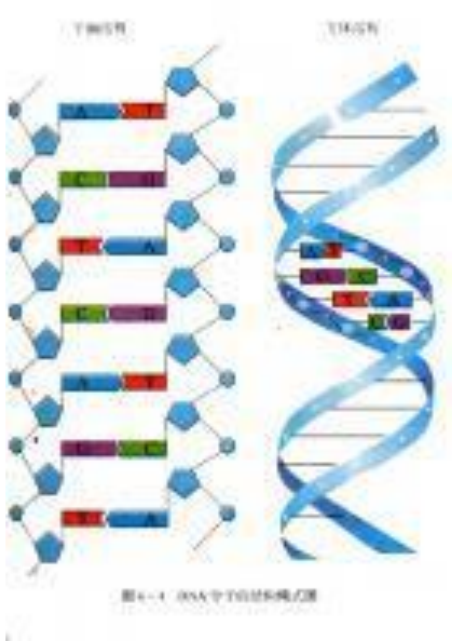
求子问题最优解的代价 + 此次选择直接产生的代价

如矩阵链乘算法，计算矩阵链$A_iA_{i+1}\ldots A_j$的最优括号化方案，若选择划分位置为A_k，则计算矩阵链$A_iA_{i+1}\ldots A_j$的最优括号化方案的代价就是子问题$A_i\ldots A_k$和$A_{k+1}\ldots A_j$的最优括号化方案的代价加上此次选择本身产生的代价$p_{i-1}p_kp_j$。

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j\,. \end{cases}$$

最长公共子序列

基因序列比对



一条DNA上碱基分子的排列被表示为有穷字符集{A,C,G,T}上的一个串进行表示

如：两个有机体的DNA分别为

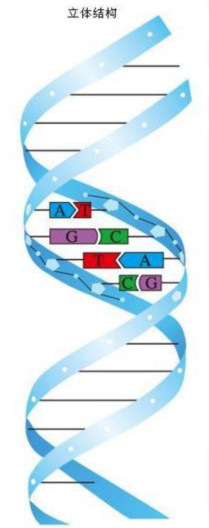
S_1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

S_2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

相似性度量

通过比较两个有机体的DNA来确定这两个有机体有多么“相似”。这在生物学上叫做“**基因序列比对**”

- 如果一个DNA螺旋是另一个DNA螺旋的子串，就说这两个DNA（串）相似。
- 当两个DNA螺旋互不为对方子串的时候，怎么度量呢?



方法一：如果将其中一个转换成另一个所需改变的工作量小，

则可称其相似（参见 ***Edit distance 15-5***）。

相似性度量

方法二：在S_1和S_2中找出第三个存在$\mathbf{S_3}$，使得S_3中的基以同样的先后顺序出现在S_1和S_2中，但不一定连续。

S_1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

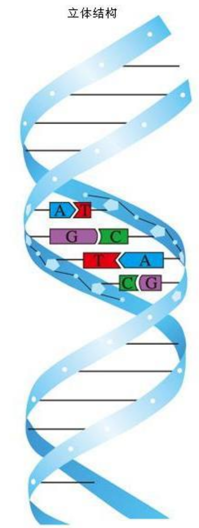
S_2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

S_3=GTCGTCGGAAGCCGGCCGAA

两个字符串的最长公共非连续子串，称为**最长公共子序列**

How?

以S_3的长度确定S_1和S_2的相似度。S_3越长，S_1和S_2的相似度越大，反之越小



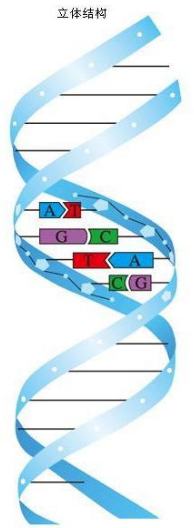
定义

1）子序列

对给定的两个序列$X=<x_1,\dots,x_n>$和$Z=<z_1,\dots,z_k>$,若存在一个X的严格递增下标序列$<i_1,\dots,i_k>$,使对所有的$j=1,\dots,k$,有 $x_{i_j}=z_j$,则称Z是X的一个子序列。

如：Z=<B, C, D, B>是X=<A, B, C, B, D, A, B>的一个子序列，

相应的下标序列为<2, 3, 5, 7>。



定义

2）公共子序列

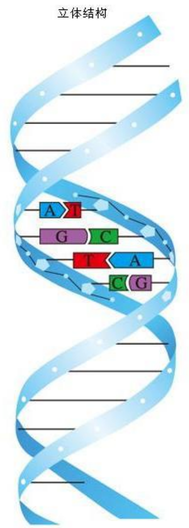
对给定的两个序列X和Y，若序列Z既是X的子序列，也是Y的子序列，则称Z是X和Y的**公共子序列**。

如，X=<A,B,C,B,D,A,B>,Y=<B,D,C,A,B,A>，则序列<B,C,A>是X和Y的一个公共子序列。

3）最长公共子序列

两个序列的长度最大的公共子序列称为它们的**最长公共子序列**。

如，<B,C,A>是上面X和Y的一个公共子序列，但不是X和Y的最长公共子序列。最长公共子序列是<B,C,B,A>。



求解

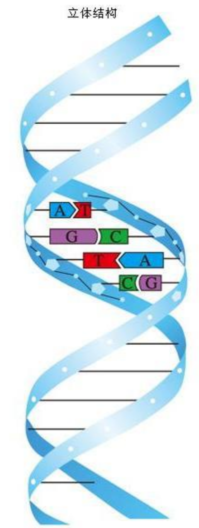
求（两个）序列的最长公共子序列

前缀：给定一个序列$X=<x_1,x_2,\ldots,x_m>$，对于$i=0,1,\ldots,m$, 定义X的第i个前缀为$X_i=<x_1,x_2,\ldots,x_i>$，即前i个元素构成的子序列。

如，$X=<A,B,C,B,D,A,B>$，则

$X_4=<A,B,C,B>$。

$X_0=\Phi$。



Longest-Common-Subsequence,LCS

最优子结构

LCS问题的最优子结构性

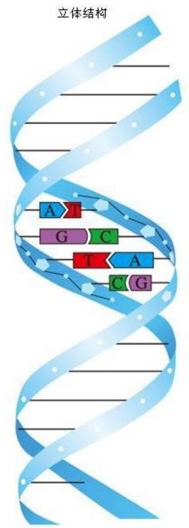
定理6.2 设有序列$X=<x_1,x_2,\ldots,x_m>$和$Y=<y_1,y_2,\ldots,y_n>$，并

设序列$Z=<z_1,z_2,\ldots,z_k>$为X和Y的任意一个**LCS**。

（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}的一个LCS。

（2）若$x_m \neq y_n$，则$z_k \neq x_m$蕴含Z是X_{m-1}和Y的一个LCS。

（3）若$x_m \neq y_n$，则$z_k \neq y_n$蕴含Z是X和Y_{n-1}的一个LCS。



Longest-Common-Subsequence,LCS

求解

求（两个）序列的最长公共子序列

LCS问题的最优子结构性

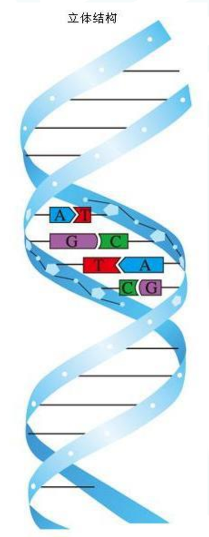
子问题的定义：从“X_m和Y_n的LCS”到

“X_{m-1}和Y_{n-1}的LCS”、

“X_{m-1}和Y_n的LCS”、

“X_m和Y_{n-1}的LCS”

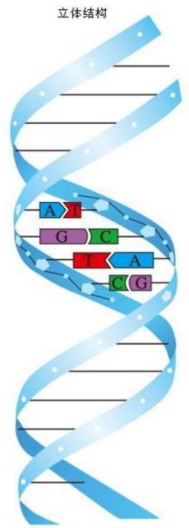
Longest-Common-Subsequence,LCS



证明

(1) 如果$z_k \neq x_m$，则可以添加x_m（也即y_n）到Z中，从而可以得到X和Y的一个长度为k+1的公共子序列。这与Z是X和Y的最长公共子序列的假设相矛盾，故必有$z_k = x_m = y_n$。

同时，如果X_{m-1}和Y_{n-1}有一个长度大于k-1的公共子序列W，则将x_m （也即y_n）添加到W上就会产生一个X和Y的长度大于k的公共子序列，与Z是X和Y的最长公共子序列的假设相矛盾，故Z_{k-1}必是X_{m-1}和Y_{n-1}的LCS。



Longest-Common-Subsequence,LCS

证明

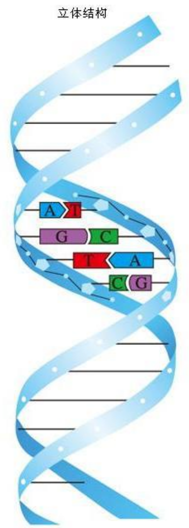
(2) 若$z_k \neq x_m$，设X_{m-1}和Y有一个长度大于k的公共子序列W，则W也应该是X_m和Y的一个公共子序列。这与Z是X和Y的一个LCS的假设相矛盾。故Z是X_{m-1}和Y的一个LCS。

(3) 同（2）。

(证毕)

子问题

上述定理说明，两个序列的一个LCS也包含了两个序列的前缀的LCS，即LCS问题具有最优子结构性质。



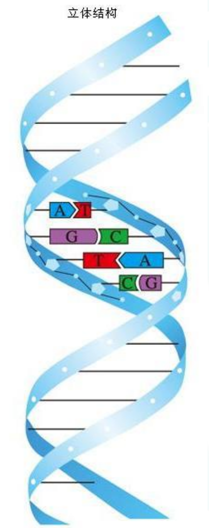
Longest-Common-Subsequence,LCS

递推关系式

记，c[i,j]为前缀序列X_i和Y_j的一个LCS的长度。则有

$$
c[i,j]=\begin{cases} 0 & \text{如果} i=0 \text{或} j=0 \\ c[i-1,j-1]+1 & \text{如果} i,j>0 \text{且} x_i = y_j \\ \max(c[i,j-1],c[i-1,j]) & \text{如果} i,j>0 \text{且} x_i \neq y_j \end{cases}
$$

c[i,j]　表结构，二维数组



Longest-Common-Subsequence,LCS

递推关系式

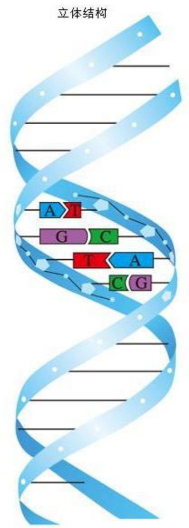
1）若i=0或j=0，即其中一个序列的长度为零，则二者的LCS的长度为0，LCS=Φ；

2）若$x_i=y_j$，则X_i和Y_j的LCS是在X_{i-1}和Y_{j-1}的LCS之后附加将x_i（也即y_j）得到的，所以

$$c[i,j]=c[i-1,j-1]+1;$$

3）若$x_i \neq y_j$，则X_i和Y_j的LCS的最后一个字符不会是x_i或y_j(不可能同时等于两者，或与两者都不同)，此时该LCS应等于X_{i-1}和Y_j的LCS与X_i和Y_{j-1}的LCS之中的长者。所以

$$c[i,j]=max(c[i-1,j],c[i,j-1]);$$



注：以上情况涵盖了X_m和Y_n的LCS的所有情况

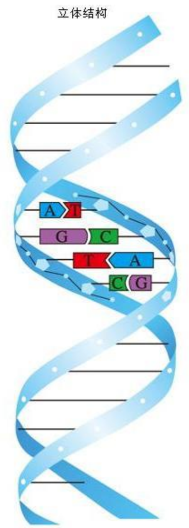
Longest-Common-Subsequence,LCS

求解

X_m和Y_n的LCS是基于X_{m-1}和Y_{n-1}的LCS、或X_{m-1}和Y_n的LCS、或X_m和Y_{n-1}的LCS求解的。

下述过程**LCS-LENGTH(X,Y)**求序列X=<$x_1,x_2,\ldots,x_m$>和Y=<$y_1,y_2,\ldots,y_n$>的LCS的长度，表**c[1..m,1..n]**中包含每一阶段的LCS长度，c[m,n]等于X和Y的LCS的长度。

同时，还设置了一个表**b[1..m,1..n]**，记录当前c[i,j]的计值情况，以此来构造该LCS。



Longest-Common-Subsequence,LCS

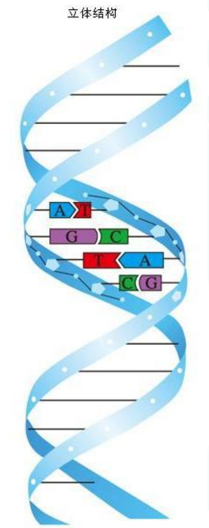
代码

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

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

i \ j		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

LCS-LENGTH的时间复杂度为O(mn)



Longest-Common-Subsequence, LCS

代码

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

j		0	1	2	3	4	5	6
i	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

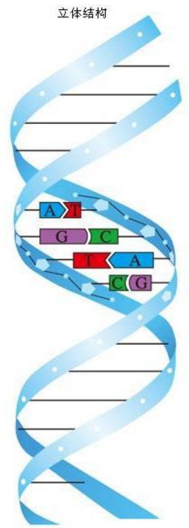
说明：

1）第i行和第j列中的方块包含了c[i,j]的值以及b[i,j]记录的箭头。

2）对于i,j＞0，项c[i,j]仅依赖于是否有$x_i=y_j$及项c[i-1,j]、c[i,j-1]、c[i-1,j-1]的值。

3）为了重构一个LCS，从右下角开始跟踪b[i,j]箭头即可，即如图所示中的蓝色方块给出的轨迹。

4）图中，c[7,6]=4，
LCS(X,Y)=<B,C,B,A>



Longest-Common-Subsequence,LCS

最长公共子序列

记录

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

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

$$c[i,j]=\begin{cases}0 & \text{如果}i=0\text{或}j=0\\ c[i-1,j-1]+1 & \text{如果}i,j>0\text{且}x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{如果}i,j>0\text{且}x_i\neq y_j\end{cases}$$

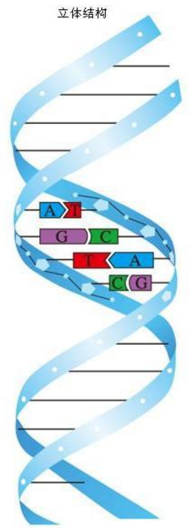
c[0,j] = 0　　c[i,0] = 0

i = 1, j = 1　　$x_1 \neq y_1$

c[1,1] = max{c[1,0], c[0,1]} = max{0,0}
= 0

b[1,1] = '↑'

$$\begin{aligned}&\textbf{elseif } c[i-1,j] \geq c[i,j-1]\\ &\quad c[i,j] = c[i-1,j]\\ &\quad b[i,j] = \text{“}\uparrow\text{”}\end{aligned}$$



Longest-Common-Subsequence, LCS

记录

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

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	B	0	↖ 1	←1	←1	↑ 1	↖ 2	←2
3	C	0	↑ 1	↑ 1	↖ 2	←2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

$$c[i,j]=\begin{cases}0 & \text{如果}i=0\text{或}j=0\\ c[i-1,j-1]+1 & \text{如果}i,j>0\text{且}x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{如果}i,j>0\text{且}x_i\neq y_j\end{cases}$$

c[0,j] = 0　　c[i,0] = 0

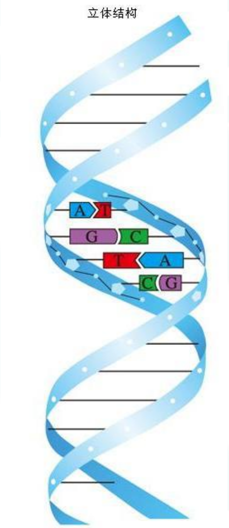
i = 1, j = 2　　$x_1 \neq y_2$

c[1,2] = max{c[1,1], c[0,2]} = max{0,0}
　　　= 0

b[1,2] = '↑'

$$\begin{aligned}&\textbf{elseif } c[i-1,j]\geq c[i,j-1]\\ &\quad c[i,j]=c[i-1,j]\\ &\quad b[i,j]=\text{“}\uparrow\text{”}\end{aligned}$$

Longest-Common-Subsequence,LCS



记录

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

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

$$c[i,j]=\begin{cases}0 & \text{如果}i=0\text{或}j=0\\ c[i-1,j-1]+1 & \text{如果}i,j>0\text{且}x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{如果}i,j>0\text{且}x_i\neq y_j\end{cases}$$

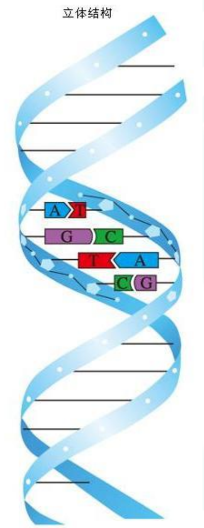
c[0,j] = 0 c[i,0] = 0

i = 1, j = 3 $x_1 \neq y_3$

c[1,3] = max{c[1,2], c[0,3]} = max{0,0}
= 0

b[1,3] = '↑'

$$\begin{aligned}&\textbf{elseif } c[i-1,j]\geq c[i,j-1]\\ &\quad c[i,j]=c[i-1,j]\\ &\quad b[i,j]=\text{“}\uparrow\text{”}\end{aligned}$$



Longest-Common-Subsequence,LCS

记录

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

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

$$c[i,j]=\begin{cases}0 & \text{如果}i=0\text{或}j=0\\ c[i-1,j-1]+1 & \text{如果}i,j>0\text{且}x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{如果}i,j>0\text{且}x_i\neq y_j\end{cases}$$

c[0,j] = 0　　c[i,0] = 0

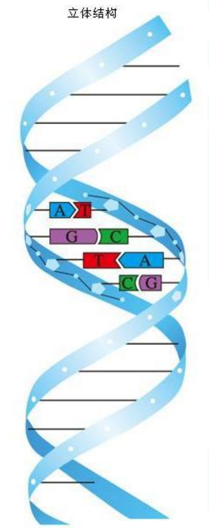
i = 1, j = 4　　$x_1=y_4$

c[1,4] = c[0,3]+1 = 0+1
= 1

b[1,4] = ‘↖’

$$\begin{aligned}&\textbf{if } x_i == y_j\\ &\quad c[i,j] = c[i-1,j-1]+1\\ &\quad b[i,j] = \text{“}\nwarrow\text{”}\end{aligned}$$

Longest-Common-Subsequence,LCS



记录

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

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

$$c[i,j]=\begin{cases}0 & \text{如果}i=0\text{或}j=0\\ c[i-1,j-1]+1 & \text{如果}i,j>0\text{且}x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{如果}i,j>0\text{且}x_i\neq y_j\end{cases}$$

c[0,j] = 0 c[i,0] = 0

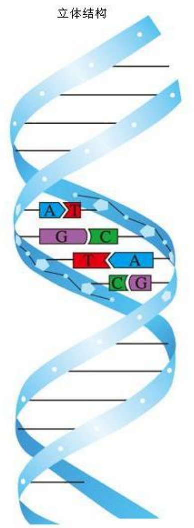
i = 1, j = 5 $x_1 \neq y_5$

c[1,5] = max{c[1,4], c[0,5]} = max{1,0} = 1

b[1,4] = ‘←’

$$\textbf{else } c[i,j] = c[i,j-1]$$
$$b[i,j] = \text{“}\leftarrow\text{”}$$

Longest-Common-Subsequence,LCS



记录

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

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

$$c[i,j]=\begin{cases}0 & 如果i=0或j=0\\ c[i-1,j-1]+1 & 如果i,j>0且x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & 如果i,j>0且x_i\neq y_j\end{cases}$$

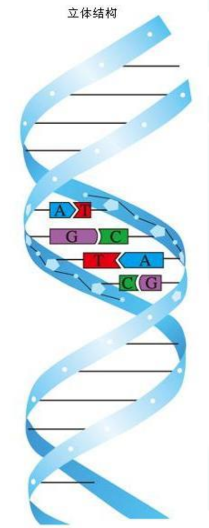
c[0,j] = 0　　c[i,0] = 0

i = 1, j = 6　　$x_1 = y_6$

c[1,6] = c[0,5]+1 = 0+1 = 1

b[1,4] = ‘↖’

$$\begin{aligned}&\textbf{if } x_i == y_j\\ &\quad c[i,j] = c[i-1,j-1]+1\\ &\quad b[i,j] = \text{“}\nwarrow\text{”}\end{aligned}$$



Longest-Common-Subsequence,LCS

记录

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

$$c[i,j]=\begin{cases}0 & \text{如果} i=0 \text{或} j=0 \\ c[i-1,j-1]+1 & \text{如果} i,j>0 \text{且} x_i=y_j \\ \max(c[i,j-1],c[i-1,j]) & \text{如果} i,j>0 \text{且} x_i\neq y_j\end{cases}$$

c[0,j] = 0   c[i,0] = 0

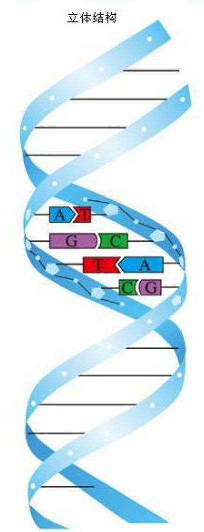
i = 2, j = 1 $x_2 = y_2$

c[2,1] = c[1,0]+1 = 0+1 = 1

b[2,1] = ‘↖’

if $x_i == y_j$
$c[i,j] = c[i-1,j-1]+1$
$b[i,j] = $ “↖”

Longest-Common-Subsequence,LCS



记录

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

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

$$c[i,j]=\begin{cases}0 & \text{如果}i=0\text{或}j=0\\ c[i-1,j-1]+1 & \text{如果}i,j>0\text{且}x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{如果}i,j>0\text{且}x_i\neq y_j\end{cases}$$

c[0,j] = 0　　c[i,0] = 0

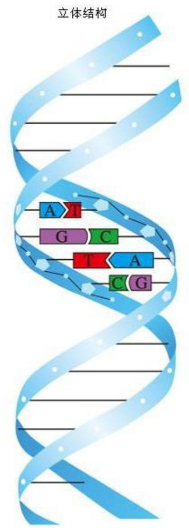
i = 2, j = 5　　$x_2 = y_5$

c[2,5] = c[1,4]+1 = 1+1 = 2

b[2,5] = ‘↖’

$$\begin{aligned}&\textbf{if } x_i == y_j\\ &\quad c[i,j] = c[i-1,j-1]+1\\ &\quad b[i,j] = \text{“↖”}\end{aligned}$$

Longest-Common-Subsequence,LCS



构造

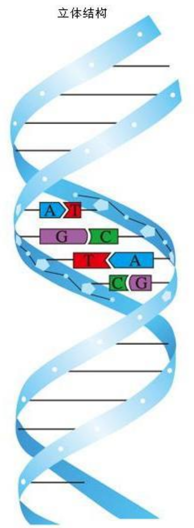
表b用来构造序列X=<A,B,C,B,D,A,B>,Y=<B,D,C,A,B,A>的一个LCS

反序，从b[m,n]处开始，沿箭头在表格中向上追踪。每当在表项b[i,j]中：

- 遇到一个“↖”时，意味着$x_i=y_j$是LCS的一个元素，下一步
  继续在b[i-1,j-1]中寻找上一个元素；
- 遇到“←”时，下一步到b[i,j-1]中寻找上一个元素；
- 遇到“↑”时，下一步到b[i-1,j]中寻找上一个元素。

i \ j		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Longest-Common-Subsequence,LCS



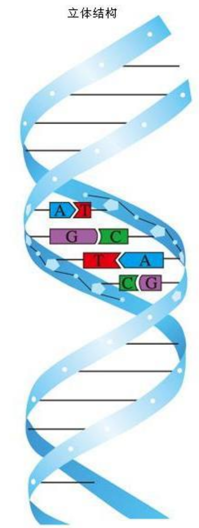
代码

过程PRINT-LCS按照上述规则输出X和Y的LCS

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
        return
if b[i, j] == "↖"
        PRINT-LCS(b, X, i − 1, j − 1)
        print x_i
elseif b[i, j] == "↑"
        PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

由于每一次循环使i或j减1，最终m=0，n=0，算法结束，所以PRINT-LCS的时间复杂度为O(m+n)

注意递归调用和print x_i的先后顺序



Longest-Common-Subsequence,LCS

打印

PRINT-LCS(b,X,7,6)

PRINT-LCS(b,X,6,6)　print A

PRINT-LCS(b,X,5,5)

PRINT-LCS(b,X,4,5)　print B

PRINT-LCS(b,X,3,4)

PRINT-LCS(b,X,3,3)　print C

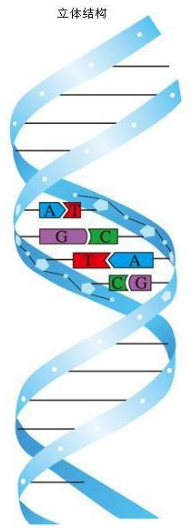
PRINT-LCS(b,X,2,2)

PRINT-LCS(b,X,2,1)　print B

PRINT-LCS(b,X,1,0)　结束

j		0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Longest-Common-Subsequence,LCS



思考：

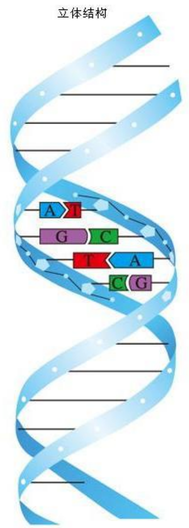
是否可以去掉表b？直接基于c求LCS！

改进

算法中，每个c[i,j]的计算仅需c的两行的数据：

正在被计算的一行和前面的一行

故可以仅用表c中的2*min(m,n)项以及O(1)的额外空间即可完成整个计算。



作业3：

- 计算题：
  - 15.4-1

最优二叉搜索树

二叉搜索树 T 是一棵二元树，它或者为空，或者其每个结点含有一个可以比较大小的数据元素，且有：

- T 的左子树的所有元素比根结点中的元素小；
- T 的右子树的所有元素比根结点中的元素大；
- T 的左子树和右子树也是二叉搜索树。

不失一般性，这里假设所有元素互异。

代码

```
SEARCH(T, X, i)
    //在二叉搜索树T中检索X。如果X不在T中，则置i=0；否则i有IDENT(i)=X//
    i←T
    while i≠0 do
      case
        :X < IDENT(i): i←LCHILD(i) //若X小于IDENT(i)，则在左子树中继续查找//
        :X = IDENT(i): return         //X等于IDENT(i)，则返回//
        :X > IDENT(i): i←RCHILD(i) //若X大于IDENT(i)，则在右子树中继续查找//
      endcase
    repeat
end SEARCH
```

最优二叉搜索树

场景

语言翻译，从英语到法语，对给定的单词，在单词表里找到该词

方法

创建一棵**二叉搜索树**，以英语单词作为关键字构建树

目标

尽快地找到英语单词，使“**总**”的搜索时间尽量少

思路

频繁使用的单词，如the，尽可能靠近根；不经常出现的单词可以离根远一些

- 引入新的因素：使用频率

二叉搜索树定义

给定一个n个关键字的已排序的序列$K=<k_1,k_2,\ldots,k_n>$（不失一般性，设$k_1<k_2<\ldots<k_n$），对每个关键字k_i，都有一个概率p_i表示其被搜索的频率。根据k_i和p_i构建一个二叉搜索树T，每个k_i对应树中的一个结点。

对搜索对象x，在T中可能找到、也可能找不到：

- 若x等于某个k_i，则一定可以在T中找到结点k_i，称为**成功搜索**。

- 成功搜索的情况一共有n种，分别是x恰好等于某个k_i。

二叉搜索树定义

- 若$x<k_1$、或 $x>k_n$、或 $k_i<x<k_{i+1}$ $(1\le i<n)$, 则在T中搜索x将失败，称为失败搜索
  - 为此引入外部结点$d_0,d_1,\ldots,d_n$，用来表示不在K中的值,称为伪关键字。
  - 伪关键字在T中对应外部结点，共有$n+1$个。

    ——**扩展二叉树**：内结点表示关键字k_i，外结点(叶子结点)表示d_i。
  - 这里每个**d_i代表一个区间**。
    - d_0表示所有小于k_1的值， d_n表示所有大于k_n的值，对于$i=1,\ldots,n-1$，d_i表示所有在k_i和k_{i+1}之间的值。
  - 每个d_i也有一个概率q_i，表示搜索对象x恰好落入区间d_i的频率。

Example

例：设有n=5个关键字的集合，每个k_i的概率p_i和相应d_i的概率q_i如表所示

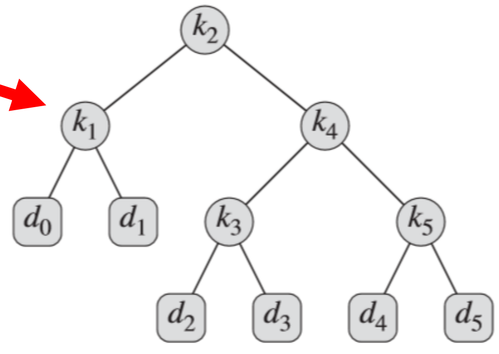
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

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

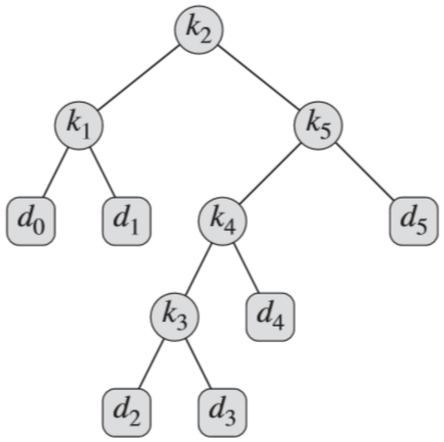
成功检索和不成功检索总共有2n+1种情况

概率p_i表示其被搜索的频率　　概率q_i表示搜索对象x恰好落入区间d_i的频率

每个k_i对应一个内结点，共有n个，用圆形结点表示，代表成功检索的位置。



(a)



(b)

每个d_i对应一个外部结点，有n+1个，用矩形框表示，代表失败检索的情况。

期望搜索代价

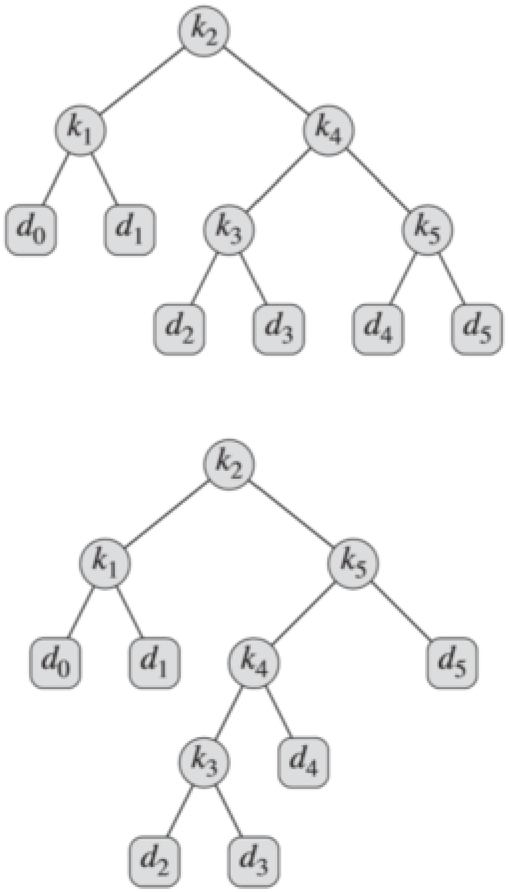
对于特定搜索对象，搜索过程是从根开始到某个结点的检索过程。成功搜索结束于内结点，不成功搜索结束于外部结点

记一次搜索的代价等于从根结点开始访问结点的数量(含内部结点和外部结点)

记$depth_T(i)$为结点i在T中的深度（根到i的路径上的边数）。

则从根结点开始访问结点i的数量等于$depth_T(i)+1$

期望搜索代价

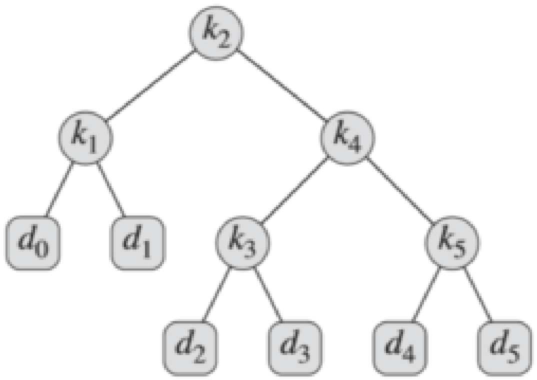


则二叉搜索树T的**期望代价**为：

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

即：加权平均代价，包括所有成功搜索的结点和失败搜索的结点。

期望搜索代价



n=5

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

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

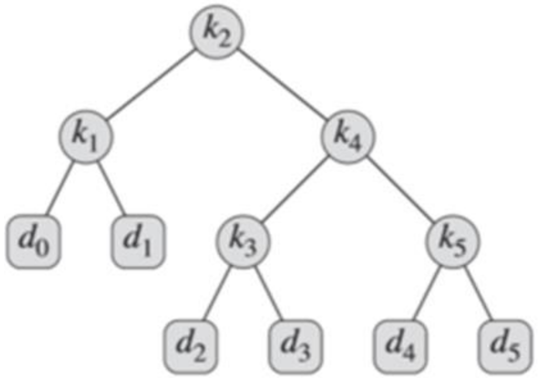
node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

期望搜索代价为2.80

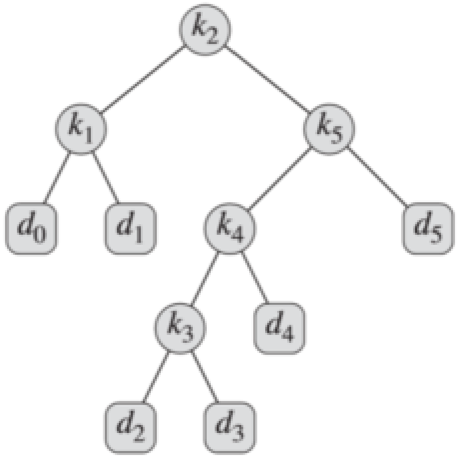
期望搜索代价

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

n=5



期望搜索代价为2.80



期望搜索代价为2.75

树b的期望代价小于树a。事实上，树b是当前问题实例的一棵最优二叉搜索树。

最优二叉搜索树定义

对于给定的关键字及其概率集合，期望搜索代价最小的二叉搜索树称为其最优二叉搜索树。

- 对给定的关键字和概率集合，怎么构造最优二叉搜索树?
- 关键问题：确定谁是树根
  - 树根不一定是概率最高的关键字；
  - 树也不一定是最矮的树；
  - 但该树的期望搜索代价必须是最小的。

枚举？No！

证明

（1）证明最优二叉搜索树的**最优子结构**

如果**T**是一棵相对于关键字$k_1,\ldots,k_n$和伪关键字d_0，…,d_n的最优二叉搜索树，则T中一棵包含关键字$k_i,\ldots,k_j$的子树T’必然是相对于关键字$k_i,\ldots,k_j$（和伪关键字$d_{i-1}, \ldots, d_j$）的最优二叉搜索子树。

证明

用剪切-粘贴法证明

对关键字$k_i,\ldots,k_j$和伪关键字$d_{i-1},\ldots,d_j$，如果存在子树T″，其期望搜索代价比T′低，那么将T′从T中删除，将T″粘贴到相应位置上，则可以得到一棵比T期望搜索代价更低的二叉搜索树，与T是最优的假设矛盾。

构造

（2）构造最优二叉搜索树

利用最优二叉搜索树的最优子结构性来构造最优二叉搜索树

分析

根在哪里?

对给定的关键字$k_i,\ldots,k_j$，若其最优二叉搜索（子）树的根结点是k_r（$i \le r \le j$），则k_r的左子树中包含关键字$k_i,\ldots,k_{r-1}$及伪关键字d_{i-1}，$\ldots,d_{r-1}$，右子树中将含关键字$k_{i+1},\ldots,k_j$及伪关键字$d_r,\ldots,d_j$。

策略

在 $i \le l \le j$ 的范围内检查所有可能的结点k_l

对每一个l，事先分别求出包含关键字$k_i,\ldots,k_{l-1}$和关键字$k_{l+1},\ldots,k_j$的最优二叉搜索子树，通过组合左、右子树找到具有最小期望成本的k_r，它就是包含关键字$k_i, \ldots,k_j$的最优二叉搜索（子）树的根。

计算

求解包含关键字$k_i,\ldots,k_j$的最优二叉搜索树，其中$1\le i, j\le n$ 且 $i\le j$

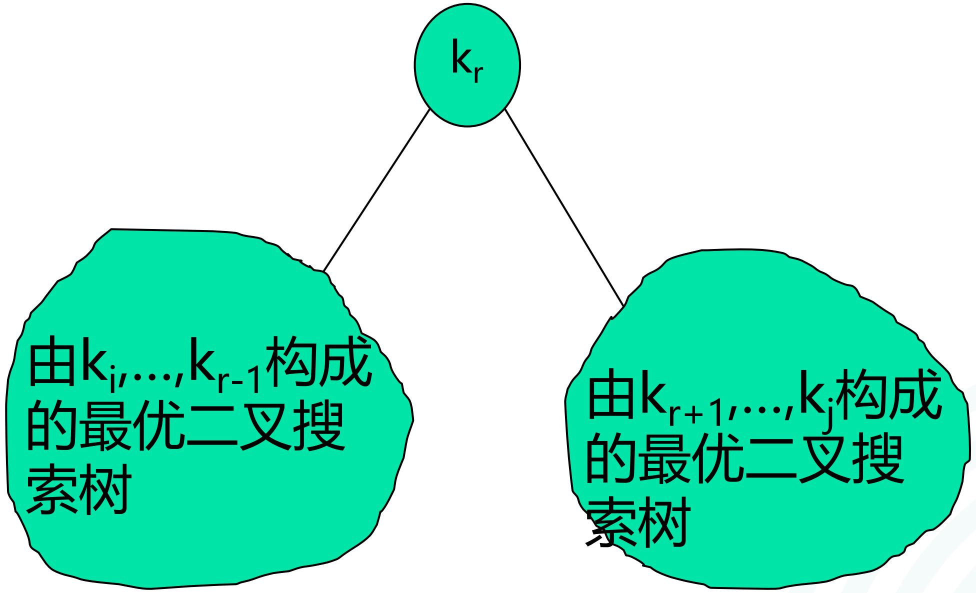
定义e[i,j]：为包含关键字$k_i,\ldots,k_j$的最优二叉搜索树的期望搜索代价

- e[1,n]为问题的最终解的期望搜索代价。

计算过程

（1）当$i \le j$时，从$k_i,\ldots,k_j$中选择出根结点k_r。

- 其左子树包含关键字$k_i,\ldots,k_{r-1}$且是最优二叉搜索子树；
- 其右子树包含关键字$k_{r+1},\ldots, k_j$且同样为最优二叉搜索子树。

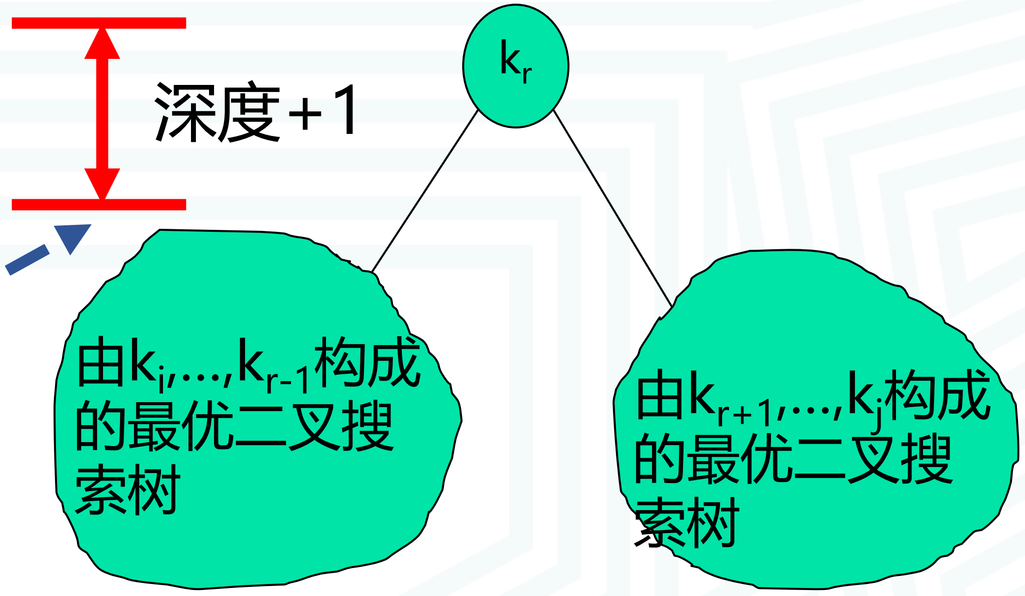


计算过程

（1）当i≤j时，从$k_i,\ldots,k_j$中选择出根结点k_r

当一棵树成为另一个结点的子树时，有以下变化：

- 子树的每个结点的深度增加1。
- 根据搜索代价期望值计算公式，子树对根为k_r的树的期望搜索代价的贡献是其期望搜索代价+其所含所有结点的概率之和。



对于包含关键字$k_i,\ldots,k_j$的子树，所有结点的概率之和为（包含外部结点）

$$w(i,j)=\sum_{l=i}^{j}p_l+\sum_{l=i-1}^{j}q_l \,.$$

计算过程

（1）当$i \le j$时，从$k_i,\ldots,k_j$中选择出根结点k_r

若k_r为包含关键字$k_i,\ldots,k_j$的最优二叉搜索树的根，则其期望搜索代价e[i,j]与左、右子树的期望搜索代价e[i,r-1]和e[r+1,j]的递推关系为：

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

其中，w(i,r-1)和w(r+1,j)是左右子树所有结点的概率之和。且有：

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

计算过程

（1）当i≤j时，从k_i,…,k_j中选择出根结点k_r

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

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

故有：

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j)\,.$$

因此，**求k_r的递归公式**为：

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

计算过程

（2）边界条件

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

存在e[i, i-1]和e[j+1, j]的边界情况。

- 此时，子树不包含实际的关键字，而只包含伪关键字d_{i-1}，其期望搜索代价仅为： $e[i,i-1]=q_{i-1}$.

计算过程

（3）构造最优二叉树

定义root[i,j]，保存计算e[i, j]时，使e[i, j]取得最小值的r，k_r即为相对于关键字$k_i,\ldots,k_j$的最优二叉搜索（子）树的树根。

在求出e[1,n]后，利用root即可构造出最终的最优二叉搜索树。

期望值

定义三个表（数组）：

- e[1..n+1,0..n]：用于记录所有e[i,j]的值。
  - 注：e[n+1,n]对应伪关键字d_n的子树；e[1,0]对应伪关键字d_0的子树
- root[1..n]：用于记录所有**最优二叉搜索子树**的根结点。
  - 包括整棵最优二叉搜索树的根。
- w[1..n+1,0..n]：用于保存子树的结点概率之和，且有

$$w[i,j] = w[i,j-1] + p_j + q_j \,.$$

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \,.$$

这样，每个w[i,j]的计算时间仅为Θ(1)。

伪代码

```
OPTIMAL-BST(p, q, n)
let e[1..n + 1, 0..n], w[1..n + 1, 0..n],
        and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i − 1] = q_{i−1}
    w[i, i − 1] = q_{i−1}
for l = 1 to n
    for i = 1 to n − l + 1
        j = i + l − 1
        e[i, j] = ∞
        w[i, j] = w[i, j − 1] + p_j + q_j
        for r = i to j
            t = e[i, r − 1] + e[r + 1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

l是区间长度

i、j是区间下标

自底向上的迭代计算

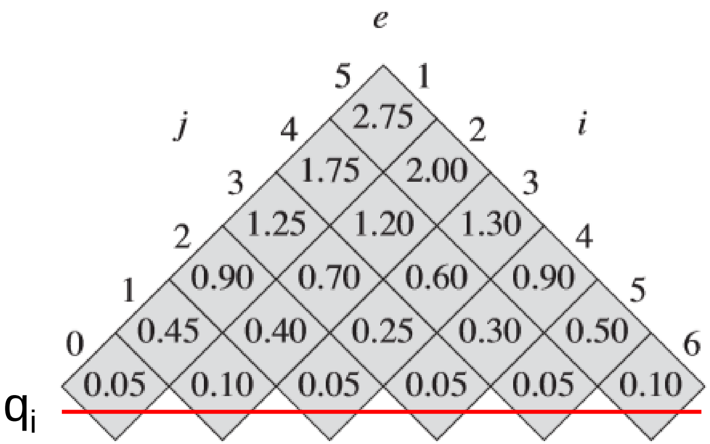
时间复杂度Θ(n^3)

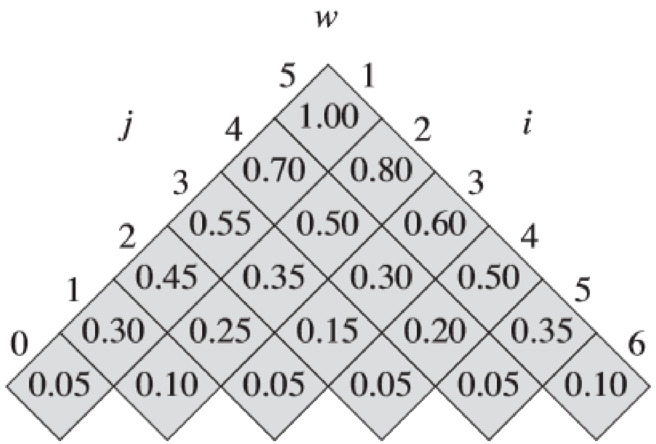
Example

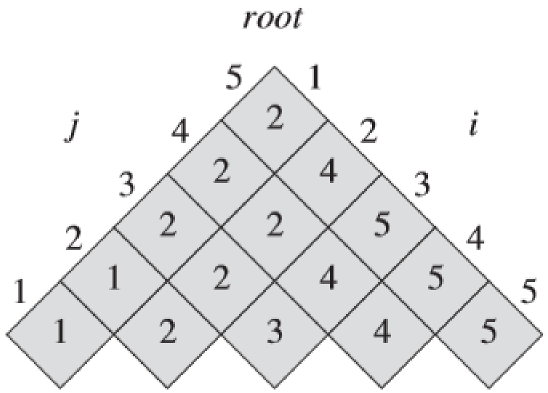
n=5,

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

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

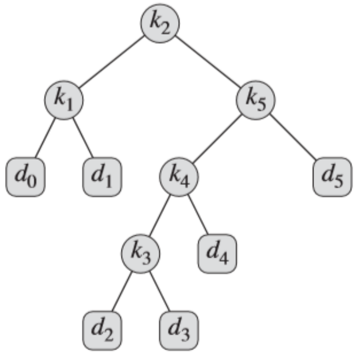






$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 , \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j . \end{cases}$$

$$w[i,j] = w[i,j-1] + p_j + q_j .$$

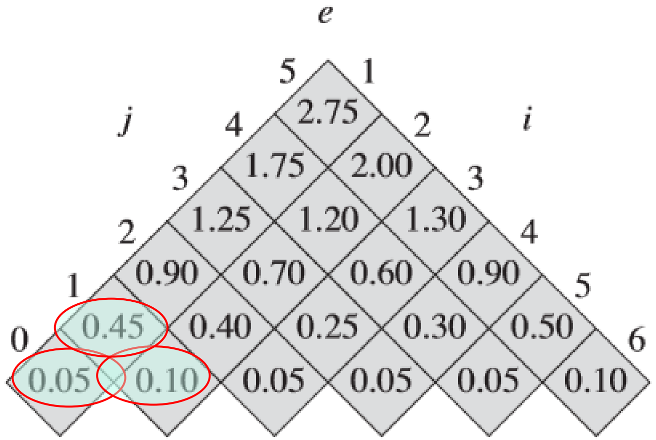


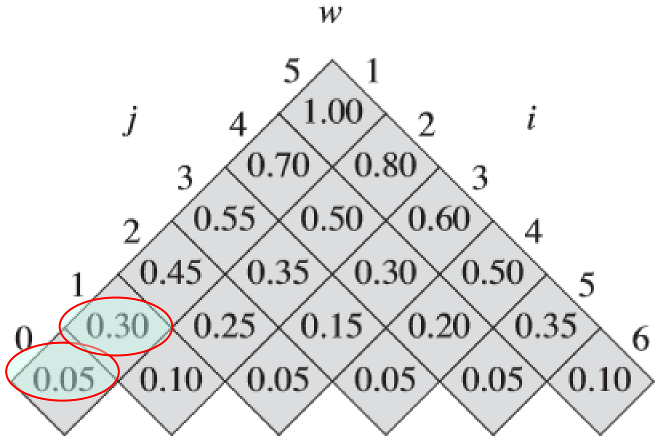
Example

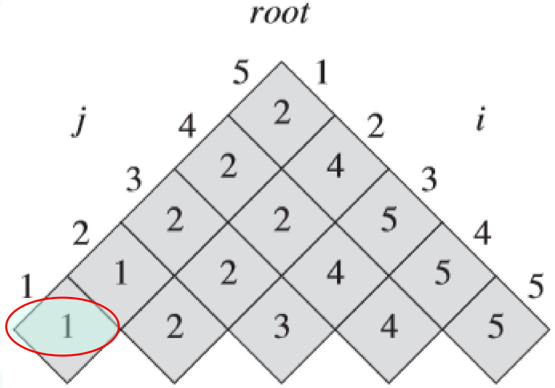
n=5,

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

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







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

$$w[i,j] = w[i,j-1] + p_j + q_j .$$

e[1,1] =

min{e[1,0]+e[2,1]+w[1,1]}

= 0.45

$e[1,0] = q_0 = 0.05$

$e[2,1] = q_1 = 0.10$

$w[1,0] = q_0 = 0.05$

$w[1,1] = w[1,0] + p_1+q_1 = 0.3$

$r[1,1] = 1$

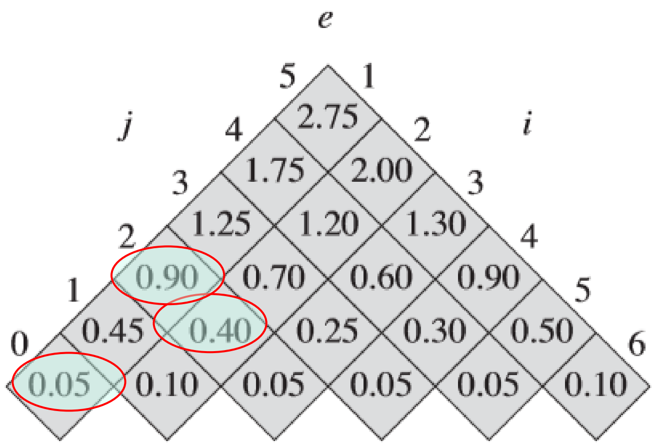
最优二叉搜索树

Example

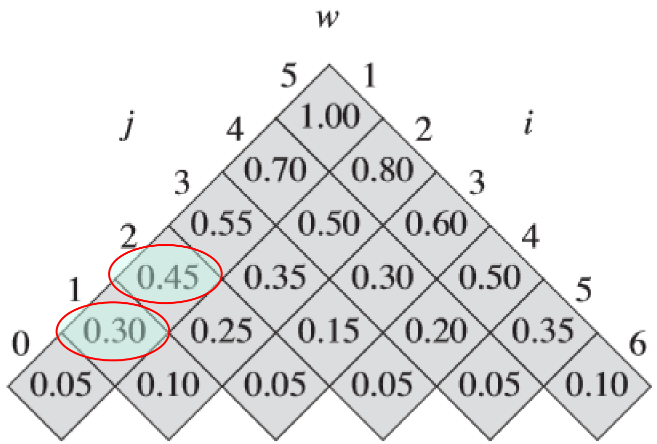
n=5,

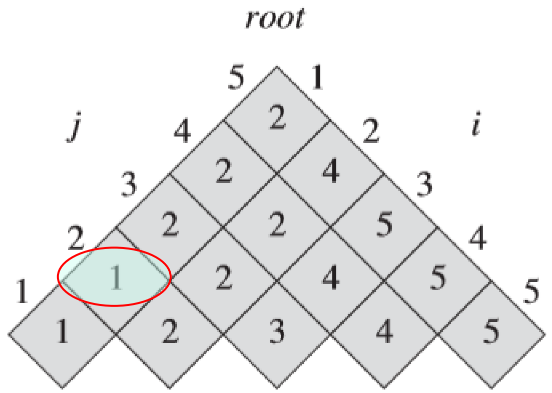
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

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



q_i





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

$e[1,0] = q_0 = 0.05$

$e[1,1] = 0.45$

$e[2,2] = 0.4$

$e[3,2] = q_2 = 0.05$

$$w[i,j] = w[i,j-1] + p_j + q_j .$$

$w[1,1] = 0.3$

$w[1,2] = w[1,1] + p_2+q_2 = 0.45$

e[1,2] =

min{e[1,0]+e[2,2]+w[1,2],

e[1,1]+e[3,2]+w[1,2]}=0.9

$r[1,2] = 1$

作业4：

根据PPT146页的命题写出计算以下数值的过程

（1）w[2,3]
e[2,3]
r[2,3]

（2）w[1,5]
e[1,5]
r[1,5]

- 计算题：
  - 15.5-2