动态规划

深圳大学计算机与软件学院
卢亚辉

Outline

- 计算Fibnacci数
- 计算二项式系数
- Warshall算法
- Floyd算法
- 0-1背包问题
- 带记忆功能的背包问题解法

0,1,1,2,3,5,8,13,21,34

它可以用一个简单的递推式和两个初始条件来定义：

$$\text{当 } n \ge 2 \text{ 时，} \quad F(n) = F(n-1) + F(n-2) \tag{8.1}$$

$$F(0) = 0, \quad F(1) = 1 \tag{8.2}$$

如果我们试图利用递推式（8.1）直接计算第 n 个斐波那契数 $F(n)$，可能必须对该函数的相同值重新计算好几遍（图 2.6 给出了一个具体的例子）。请注意， 计算 $F(n)$这个问题是以计算它的两个更小的交叠子问题 $F(n-1)$和 $F(n-2)$的形式来表达的。所以，我们可以简单地在一张一维表中填入 $n+1$ 个 $F(n)$的连续值；开始时，通过观察初始条件（8.2）可以填入 0 和 1；然后以（8.1）作为运算规则计算出其他所有的元素。显然，该数组的最后一个单元应该包含 $F(n)$。这个非常简单的算法只需要一个单循环就能完成，2.5 节给出了它的一个伪代码。

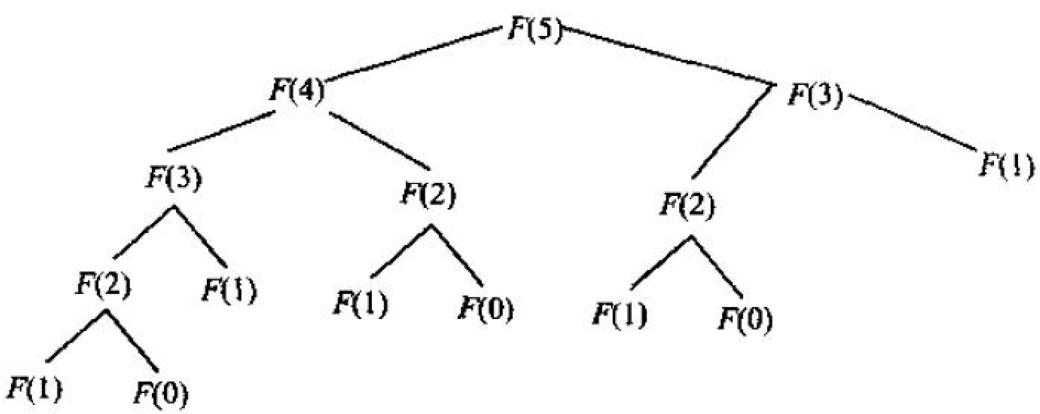


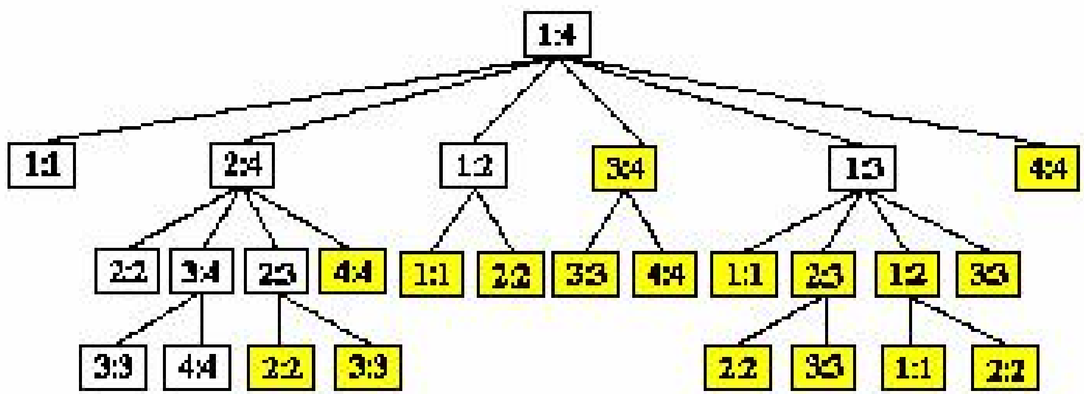
图 2.6 $n=5$ 时，计算斐波那契数的递归调用树

算法 *Fib*(n)

//根据定义，迭代计算第 n 个斐波那契数
//输入：一个非负整数 n
//输出：第 n 个斐波那契数
$F[0] \leftarrow 0$; $F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
  $F[i] \leftarrow F[i-1] + F[i-2]$
return $F(n)$

如果问题是由交叠的子问题所构成的，我们就可以用动态规划技术来解决它。一般来说，这样的子问题出现在对给定问题求解的递推关系中，这个递推关系中包含了相同类型的更小子问题的解。动态规划法建议，与其对交叠的子问题一次又一次地求解，还不如对每个较小的子问题只求解一次并把结果记录在表中，这样就可以从表中得出原始问题的解。

从顶至下的变种。但无论我们使用动态规划的经典从底至上版本还是它基于记忆功能的从顶至下版本，设计这样一种算法的关键步骤还是相同的，即，导出一个问题实例的递推关系，该递推关系包含该问题的更小（并且是交叠的）子实例的解。但像计算第 n 个斐波那契数这样，直接表现为公式（8.1）的形式，可以说是这个规则的一个极少例外。



0-1背包问题

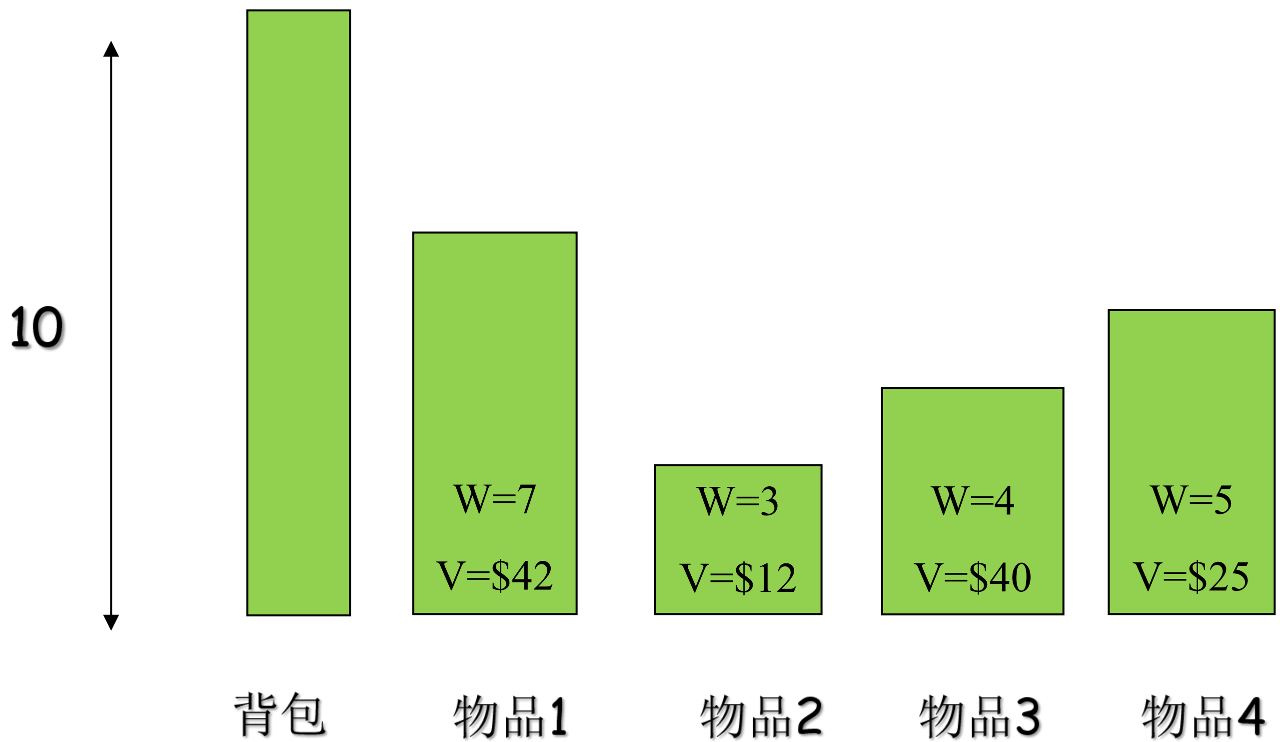
- 问题

设 $U=\{u_1, u_2, \ldots, u_n\}$ 是n个待放入容量为C的背包的物品集. 任意i: $n \geq i \geq 1$, 设s_i和v_i分别是第i种物品的重量和价值。每个物品要么整个放入背包，要么不放。且已知物品i放入背包能产生v_i的价值($n \geq i \geq 1$)。其中C和s_i、v_i（$n \geq \geq 1$）均为正整数。

如何装包才能获得最大价值?

15.1 背包问题

计算机科学中一个著名的问题。给定n个体积为$w_1,\ldots, w_n$、价值为$v_1,\ldots,v_n$的物品和一容量为W的背包，求这些物品中一个最有价值的子集。



动态规划求解背包问题

给定n个物品

整数容量:　w_1　w_2　…　w_n

价值:　v_1　v_2　…　v_n

具有整数W容量的背包

寻找最优价值的子集可以放入背包中

蛮力法：考虑所有物品的组合，逐一检查是否可以放入背包，在可行的组合中选择具有最优价值的组合。

效率：$\theta(2^n)$

$$V[i,j]=\begin{cases}\max\{V[i-1,j],v_i+V[i-1,j-w_i]\} & \text{如果} j-w_i\geq 0\\ V[i-1,j] & \text{如果} j-w_i<0\end{cases}\qquad (8.12)$$

当 $j\geq 0$ 时，$V[0,j]=0$；当 $i\geq 0$ 时，$V[i,0]=0$ （8.13）

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$V[i-1,j-w_i]$	$V[i-1,j]$	
w_i, v_i	i	0		$V[i,j]$	
	n	0			目标

图 8.12 用动态规划算法解背包问题的表格

为了设计一个动态规划算法，需要推导出一个递推关系，用较小子实例的解的形式来表示背包问题的实例的解。让我们来考虑一个由前 i 个物品（$1 \le i \le n$）定义的实例，物品的重量分别为 $w_1,\ldots,w_i$、价值分别为 $v_1,\ldots,v_i$，背包的承重量为 j（$1 \le j \le W$）。设 $V[i,j]$ 为该实例的最优解的物品总价值，也就是说，是能够放进承重量为 j 的背包中的前 i 个物品中最有价值子集的总价值。可以把前 i 个物品中能够放进承重量为 j 的背包中的子集分成两个类别：那些包括第 i 个物品的子集和那些不包括第 i 个物品的子集。然后有下面的结论：

1. 根据定义，在不包括第 i 个物品的子集中，最优子集的价值是 $V[i-1,j]$。
2. 在包括第 i 个物品的子集中（因此，$j-w_i \ge 0$），最优子集是由该物品和前 $i-1$ 个物品中能够放进承重量为 $j-w_i$ 的背包的最优子集组成。这种最优子集的总价值等于 $v_i+V[i-1,j-w_i]$。

因此，在前 i 个物品中最优解的总价值等于这两个价值中的较大值。当然，如果第 i 个物品不能放进背包，从前 i 个物品中选出的最优子集的总价值等于从前 $i-1$ 个物品中选出的最优子集的总价值。这个观察结果导致了下面这个递推式：

$$V[i,j]=\begin{cases}\max\{V[i-1,j],v_i+V[i-1,j-w_i]\} & \text{如果} j-w_i \ge 0 \\ V[i-1,j] & \text{如果} j-w_i < 0\end{cases} \qquad (8.12)$$

我们可以很容易地这样定义初始条件：

$$\text{当 } j \ge 0 \text{ 时，} V[0,j]=0\text{；当 } i \ge 0 \text{ 时，} V[i,0]=0 \qquad (8.13)$$

动态规划求解背包问题(例)

例：背包容量 $W = 5$

物品	体积	价值
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

容量 j

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

?

回溯求解最优子集

动态规划求解背包问题伪代码

```
Algorithm DPKnapsack(w[1..n], v[1..n], W)
    var V[0..n,0..W], P[1..n,1..W]: int
    for j := 0 to W do
        V[0,j] := 0
    for i := 0 to n do
        V[i,0] := 0
    for i := 1 to n do
        for j := 1 to W do
            if w[i] ≤ j and v[i] + V[i-1,j-w[i]] > V[i-1,j] then
                V[i,j] := v[i] + V[i-1,j-w[i]]; P[i,j] := j-w[i]
            else
                V[i,j] := V[i-1,j]; P[i,j] := j
    return V[n,W] 和最优子集
```

效率: $O(nW)$.

就像我们在本章开头所讨论的以及在接下来的几节中所阐述的，动态规划方法所涉及问题的解，满足一个用交叠的子问题来表示的递推关系。直接自顶向下对这样一个递推式

求解导致一个算法要不止一次地解公共的子问题，因此效率是非常低的（一般来说是指数级的，甚至更差）。另一方面，经典的动态规划方法是自底向上工作的：它用所有较小子问题的解填充表格，但是每个子问题只解一次。这种方法无法令人满意的一面是，在求解给定问题时，有些较小子问题的解常常不是必需的。由于这个缺点没有在自顶向下法中表现出来，所以我们很自然地希望把自顶向下和自底向上方法的优势结合起来。我们的目标是得到这么一种方法，它只对必要的子问题求解并且只解一次。这种方法是存在的，它是以记忆功能为基础的[BB96]。

该方法用自顶向下的方式对给定的问题求解，但还需要维护一个类似自底向上动态规划算法使用的表格。一开始的时候，用一种“null”符号初始化表中所有的单元，用来表明它们还没有被计算过（在这里可以使用一种称为**虚拟初始化**的技术——参见习题 7.1 的第 8 题）。然后，一旦需要计算一个新的值，该方法先检查表中相应的单元：如果该单元不是“null”，它就简单地从表中取值；否则，就使用递归调用进行计算，然后把返回的结果记录在表中。

算法 *MFKnapsack* (i,j)

```
//对背包问题实现记忆功能方法
//输入： 一个非负整数 i 指出先考虑的物品数量，一个非负整数 j 指出了背包的承重量
//输出：前 i 个物品的最优可行子集的价值
//注意：我们把输入数组 Weights[1..n]、Values[1..n]和表格 V[0..n,0..W]作为全局变量，除了行 0 和
  列 0 用 0 初始化以外，V 的所有单元都用-1 做初始化。
if V[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1), j),
                    Value[i] + MFKnapsack(i − 1, j − Weights[i]))
    V[i, j] ← value
return V[i, j]
```

		承重量 j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	–	12	12
$w_2 = 1,\ v_2 = 10$	2	0	–	12	22	–	22
$w_3 = 3,\ v_3 = 20$	3	0	–	–	22	–	32
$w_4 = 2,\ v_4 = 15$	4	0	–	–	–	–	37

图 8.14 用记忆功能算法解背包问题实例的一个例子

计算二项式系数

$C(n,k)$或者 $\binom{n}{k}$

是来自于一个n元素集合的k元素组合（子集）的数量（$0 \le k \le n$）。

$$(a+b)^n = C(n,0)a^n + \ldots + C(n,i)a^{n-i}b^i + \ldots + C(n,n)b^n$$

当$n > k > 0$时， $C(n,k) = C(n-1,k-1) + C(n-1,k)$ （8.3）

$$C(n,0) = C(n,n) = 1 \quad （8.4）$$

算法 *Binomial* (n,k)

```
//用动态规划算法计算 C(n,k)
//输入: 一对非负整数 n ≥ k ≥ 0
//输出: C(n,k)的值
for i ← 0 to n do
    for j ← 0  to min(i, k) do
        if j = 0  or  j=k
            C[i, j] ← 1
      ekse C[i, j] ← C[i − 1, j − 1] + C[i − 1,
 return C[n, k]
```

	0	1	2	…	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
⋮						
k	1					1
⋮						
$n-1$	1				$C(n-1, k-1)$	$C(n-1, k)$
n	1					$C(n, k)$

图 8.1 动态规划算法中，用来计算二项式系数 $C(n,k)$的表

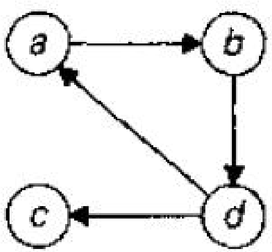
$$A(n,k)=\sum_{i=1}^{k}\sum_{j=1}^{i-1}1+\sum_{i=k+1}^{n}\sum_{j=1}^{k}1=\sum_{i=1}^{k}(i-1)+\sum_{i=k+1}^{n}k$$

$$=\frac{(k-1)k}{2}+k(n-k)\in\Theta(nk)$$

传递闭包问题

定义 一个 n 顶点有向图的传递闭包可以定义为一个 n 阶布尔矩阵 $T=\{t_{ij}\}$，如果从第 i 个顶点到第 j 个顶点之间存在一条有效的有向路径，矩阵第 i 行（$1\le i\le n$）第 j 列（$1\le j\le n$）的元素为 1；否则，t_{ij} 为 0。

作为例子，图 8.2 给出了一个有向图、该图的邻接矩阵及其传递闭包。



(a)

$$A=\begin{array}{c} \\ a\\ b\\ c\\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array}\\ \begin{bmatrix} 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0\\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b)

$$T=\begin{array}{c} \\ a\\ b\\ c\\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array}\\ \begin{bmatrix} 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1\\ 0 & 0 & 0 & 0\\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c)

图 8.2 （a）有向图；（b）它的邻接矩阵；（c）它的传递闭包

我们可以在深度优先查找和广度优先查找的帮助下生成有向图的传递闭包。从第 i 个顶点开始，无论采用哪种遍历方法，都能够得到通过第 i 个顶点访问到的所有顶点的信息，因此，传递闭包的第 i 行的相应列置为 1。这样的话，以每个顶点为起始点做一次这样的遍历就生成了整个图的传递闭包。

Warshall 算法

算法通过一系列 n 阶布尔矩阵来构造一个给定的 n 个顶点有向图的传递闭包：

$$R^{(0)},\dots,R^{(k-1)},R^{(k)},\dots,R^{(n)} \tag{8.5}$$

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ 或 } r_{ik}^{(k-1)} \text{ 和 } r_{kj}^{(k-1)} \tag{8.7}$$

算法 *Warshall* ($A[1..n,1..n]$)

效率仅仅属于 $\Theta(n^3)$。

```
//实现计算传递闭包的 Warshall 算法
//输入：包括 n 个顶点有向图的邻接矩阵 A

//输出：该有向图的传递闭包
R^(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k-1)[i, j] or R^(k-1)[i, k] and R^(k-1)[k, j]
return R^(n)
```

该算法的中心思想是，任何 $R^{(k)}$ 中的所有元素都可以通过它在序列（8.5）中的直接前趋 $R^{(k-1)}$ 计算得到。把矩阵 $R^{(k)}$ 中第 i 行第 j 列的元素 $r_{ij}^{(k)}$ 置为 1。这意味着存在一条从第 i 个顶点到第 j 个顶点的路径，路径中每一个中间顶点的编号都不大于 k：

$$v_i,\ \text{每个顶点编号都不大于} k \text{ 的一个中间顶点列表},\ v_j \tag{8.6}$$

对于这种路径，有两种可能的情况。在第一种情况下，路径的中间顶点列表中不包含第 k 个顶点。那么这条从 v_i 到 v_j 的路径中顶点的编号不会大于 $k-1$，所以 $r_{ij}^{(k-1)}$ 也等于 1。第二种可能性是路径（8.6）的中间顶点的确包含第 k 个顶点 v_k。在不失一般性的前提下，假设 v_k 在列表中只出现一次。（如果不是这种情形，我们只要简单地把路径中第一个 v_k 和最后一个 v_k 之间的顶点全部消去，就可以创建一条从 v_i 到 v_j 的新路径。）在做出上述说明以后，路径（8.6）可以改写成下面这种形式：

$$v_i,\ \text{编号} \le k-1 \text{ 的顶点},\ v_k,\ \text{编号} \le k-1 \text{ 的顶点},\ v_j$$

这个表现形式的第一部分意味着存在一条从 v_i 到 v_k 的路径，路径中每个中间顶点的编号都不大于 $k-1$（因此 $r_{ik}^{(k-1)}=1$），而第二部分意味着存在一条从 v_k 到 v_j 的路径，路径中每个中间顶点的编号也都不大于 $k-1$（因此 $r_{kj}^{(k-1)}=1$）。

我们刚才所证明的是，如果 $r_{ij}^{(k)}=1$，则要么 $r_{ij}^{(k-1)}=1$，要么 $r_{ik}^{(k-1)}=1$ 而且 $r_{kj}^{(k-1)}=1$。很容易看出，它的逆命题也成立。因此，对于如何从矩阵 $R^{(k-1)}$ 的元素中生成矩阵 $R^{(k)}$ 的元素，我们有下面的公式：

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ 或 } r_{ik}^{(k-1)} \text{ 和 } r_{kj}^{(k-1)} \tag{8.7}$$

- 如果一个元素 r_{ij} 在 $R^{(k-1)}$中是 1，它在 $R^{(k)}$中仍然是 1。

- 如果一个元素 r_{ij} 在 $R^{(k-1)}$中是 0，当且仅当矩阵中第 i 行第 k 列的元素和第 k 行第 j 列的元素都是 1，该元素在 $R^{(k)}$中才能变成 1（图 8.3 阐述了这个规则）。

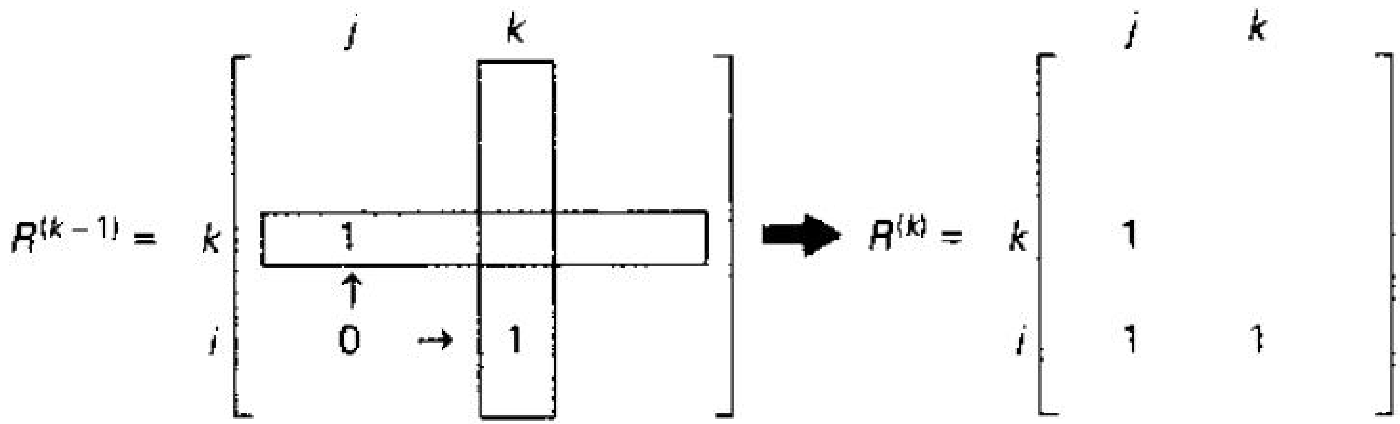
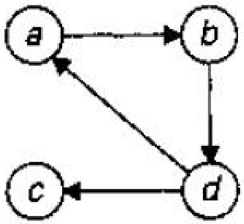


图 8.3　Warshall 算法中将 0 变成 1 的规则



$$R^{(0)} = \begin{array}{cc} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

第一个矩阵的说明：该矩阵反映了不包含中间顶点的路径（$R^{(0)}$就是邻接矩阵本身）；框起来的行和列用来计算$R^{(1)}$

$$R^{(1)} = \begin{array}{cc} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & \mathbf{1} & 1 & 0 \end{bmatrix} \end{array}$$

第二个矩阵的说明：该矩阵反映了包含编号不大于 1 的中间顶点（也就是 a）的路径（请注意有一条从 d 到 b 的新路径）；框起来的行和列用来计算 $R^{(2)}$

$$R^{(2)} = \begin{array}{cc} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & \mathbf{1} \end{bmatrix} \end{array}$$

第三个矩阵的说明：该矩阵反映了包含编号不大于 2 的中间顶点（也就是 a 和 b）的路径（请注意有两条新路径）；框起来的行和列用来计算 $R^{(3)}$

$$R^{(3)} = \begin{array}{cc} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

第四个矩阵的说明：该矩阵反映了包含编号不大于 3 的中间顶点（也就是 a，b，c）的路径（没有新路径）；框起来的行和列用来计算 $R^{(4)}$

$$R^{(4)} = \begin{array}{cc} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} \mathbf{1} & 1 & \mathbf{1} & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

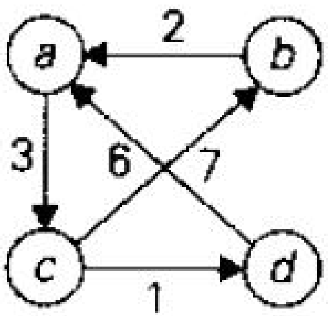
第五个矩阵的说明：该矩阵反映了包含编号不大于 4 的中间顶点（也就是 a，b，c、d）的路径（请注意有 5 条新路径）

图 8.4 对图中的有向图应用 Warshall 算法，新的路径用黑体字表示。

最短路问题

- Dijkstra算法
  - E.W.Dijkstra，1959
  - 算法描述：
    - 1）根据起始结点，为每一个结点标号
    - 2）寻找标号最小的新结点，并更新其它结点的标号
    - 3）重复上一步，直到找到目标结点
  - 只适用于处理非负权图
  - 时间复杂度O(ElogV)，O(V^2)

所有结点对间的最短路问题



(a)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

图 8.5 （a）有向图；（b）图的权重矩阵；（c）图的距离矩阵

Floyd算法

Floyd 算法通过一系列 n 阶矩阵来计算一个 n 顶点加权图的距离矩阵：

$$D^{(0)},\ldots,D^{(k-1)},D^{(k)},\ldots,D^{(n)} \tag{8.8}$$

$$\text{当 } k \ge 1, d_{ij}^{(0)} = w_{ij} \text{ 时，} \quad d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \tag{8.10}$$

算法 *Floyd* (W[1..n, 1..n])

```
    // 实现计算完全最短路径的 Floyd 算法
    // 输入：图的权重矩阵 W
    // 输出：包含最短路径长度的距离矩阵
    D ← W                       //如果可以改写 W 的话，这一步可以省略
   for  k ← 1  to n do
    for  i ← 1 to n do
      for  j ← 1  to n do
           D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
   return D
```

- 时间复杂度： $\Theta(n^3)$

  空间复杂度： $\Theta(n^2)$

和 Warshall 算法一样，每一个 $D^{(k)}$ 中的任何元素都可以通过它在序列（8.8）中的直接前趋 $D^{(k-1)}$ 计算得到。把 $d_{ij}^{(k)}$ 作为矩阵 $D^{(k)}$ 中第 i 行第 j 列的元素。这意味着 $d_{ij}^{(k)}$ 等于从第 i 个顶点到第 j 个顶点之间所有路径中一条最短路径的长度，并且路径的每一个中间顶点的编号不大于 k：

$$v_i，每个顶点编号都不大于 k 的一个中间顶点列表，v_j \tag{8.9}$$

我们可以把所有这种路径分成两个不相交的子集：一个子集中的路径不把第 k 个顶点作为中间顶点，另一个子集则反之。因为第一个子集中，路径所包含的中间顶点的编号不会大于 $k-1$，根据我们的矩阵定义，其中最短路径的长度为 $d_{ij}^{(k-1)}$。

在第二个子集中最短路径的长度是多少呢？如果图中不包含长度为负的回路，则可以把注意力集中在第二个子集的这种路径上——即顶点 v_k 只在中间顶点中出现过一次的路径（因为多次访问 v_k 只会增加路径的长度）。所有这种路径都具有下面的形式：

$$v_i，编号 \le k-1 的顶点，v_k，编号 \le k-1 的顶点，v_j \quad .$$

换句话说，每条这种路径都由两条路径构成：一条从 v_i 到 v_k 的路径，路径中每个中间顶点的编号都不大于 $k-1$；一条从 v_k 到 v_j 的路径，路径中每个中间顶点的编号也都不大于 $k-1$。参见图 8.6。

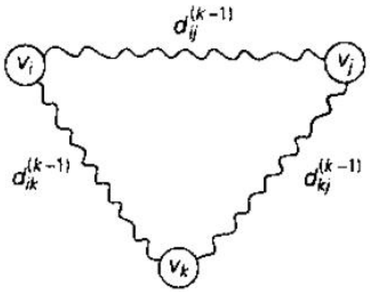
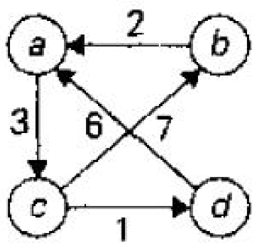


图 8.6 Floyd 算法的内在思想

因为从 v_i 到 v_k 的所有路径中，中间顶点编号不大于 $k-1$ 的最短路径长度等于 $d_{ik}^{(k-1)}$，而从 v_k 到 v_j 的所有路径中，中间顶点编号不大于 $k-1$ 的最短路径长度等于 $d_{kj}^{(k-1)}$，那么这些路径中，以第 k 个顶点为中间顶点的最短路径长度等于 $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$。把两个子集中最短路径的长度都考虑进来，我们得出了下面的递推式：

$$当 k \ge 1, d_{ij}^{(0)} = w_{ij} 时，\quad d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \tag{8.10}$$



$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

第一个矩阵的说明：不包含中间顶点的最短路径长度（$D^{(0)}$）就是权重矩阵本身）

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

第二个矩阵的说明：中间顶点（也就是 a）的编号不大于 1 的最短路径（请注意有两条新的最短路径，从 b 到 c；从 d 到 c）

$$D^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

第三个矩阵的说明：中间顶点（也就是 a 和 b）的编号不大于 2 的最短路径（请注意有一条从 c 到 a 的新的最短路径）

$$D^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

第四个矩阵的说明：中间顶点（也就是 a，b，c）的编号不大于 3 的最短路径（请注意有 4 条新的最短路径，从 a 到 b；从 a 到 d；从 b 到 d；从 d 到 b）

$$D^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

第五个矩阵的说明：中间顶点（也就是 a，b，c，d）的编号不大于 4 的最短路径（请注意有一条从 c 到 a 的新的最短路径）

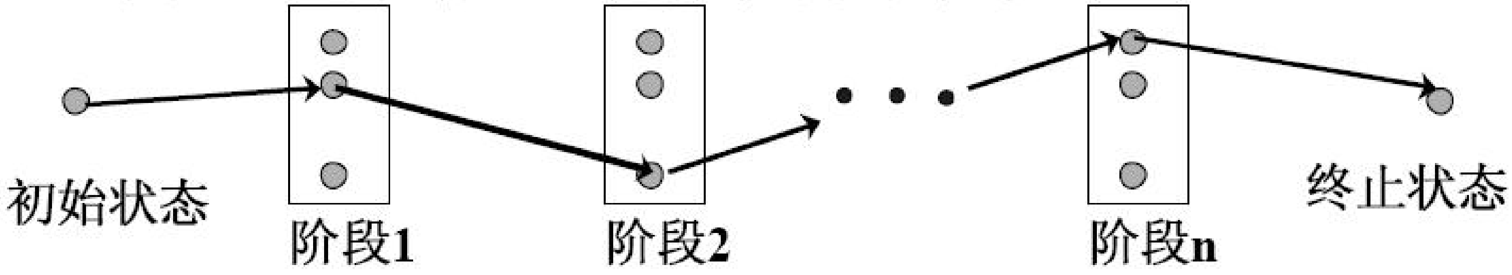
图 8.7　对给出的图应用 Floyd 算法。被改写的元素用黑体字表示

- 适用范围
  - 多阶段决策的最优化问题
  - 最优解满足最优性原理
  - 子问题的重叠性
- 基本思想
- 设计一个动态规划算法有四个步骤

动态规划的基本模式

- 适用问题
  - 多阶段决策的最优化问题

问题的求解过程分阶段来完成，在每阶段需要做出一个决策，形成一个决策序列。每个决策序列对应一个目标函数值。要求求出目标值最优（最大或最小）的决策序列，即最优决策序列（最优解），所对应的目标值为最优值。



动态规划的基本模式

- 最优性原理

  当问题的最优解包含有子问题的最优解时，称该问题满足最优性原理。

  因此，问题的最优解可在其子问题的最优解中寻找。这就决定了计算过程是：首先将问题分解为子问题，求得子问题的最优解，由此再构造得到原问题的最优解。

- 子问题重叠性

  原问题的子问题中由于可能有大量重复的子问题，而相同的子问题只求解一次，因而其效率往往高于枚举法。且子问题重复的越多，其效率越高。

动态规划的基本模式

- 动态规划法的基本思想

　　将原问题转化为若干个子问题来解决。但是每个子问题只计算一次。因此，一般需要将子问题的解保存起来以避免重复计算。

　　对所考虑的每个子问题都求出最优解，然后由子问题的最优解递推地构造原问题的最优解。

　　但是，在求解过程中由于需要将子问题的解保存下来以备将来使用，所以该方法往往需要较多的附加空间。

- 动态规划法的基本步骤

  1. 证明满足最优性原理

  实际上即是说明大问题的最优解可从子问题的最优解答中找。这就决定了计算是从下而上地进行根据子问题的最优解逐步构造出整个问题的最优解的过程。

  2. 递归定义最优解的值

  即找出如何由子问题的最优解得到原问题的最优解的关系式。

  3. 按自下而上的方式计算最优解的值

  4. 由计算的结果构造出一个最优解

- 动态规划与分治法的异同点
- 动态规划与减治法的异同点

- 捡麦穗：动态规划：一行一行看过去，每次拿到这一行最大的麦穗，与新的麦穗进行比较，拿最大的

15.4 最长公共子序列（LCS）

- 问题描述
- 如何求X、Y的LCS

  －LCS最优解结构特征（step1）

  －子问题的递归解（step2）

  －计算最优解值（step3）

  －构造一个LCS（step4）

问题描述（1）

- 子序列定义

  给定序列$X=(x_1,x_2,\ldots,x_m)$，序列$Z=(z_1,z_2,\ldots,z_k)$是X的一子序列，必须满足：若X的索引中存在一个严格增的序列$i_1,i_2,\ldots,i_k$，使得对所有的$j=1\sim k$，均有$x_{i_j}=z_j$

  例如，序列Z={B，C，D，B}是序列X={A，B，C，B，D，A，B}的子序列，相应的递增下标序列为{2，3，5，7}。

- 两个序列的公共子序列

  Z是X和Y的子序列，则Z是两者的公共子序列CS。

- 最长的公共子序列(LCS)

  在X和Y的CS中，长度最大者为一个最长公共子序列LCS。

问题描述（2）

- **Example:**

  In biological application, given two DNA sequences, for instance

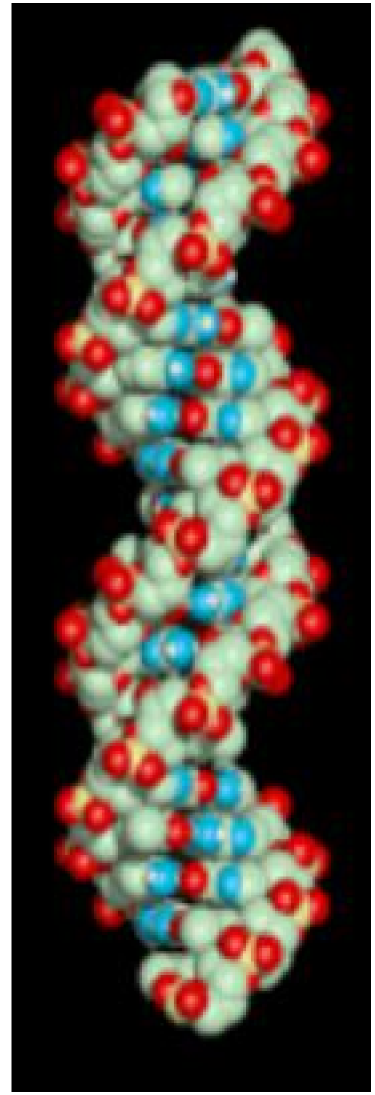
  S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

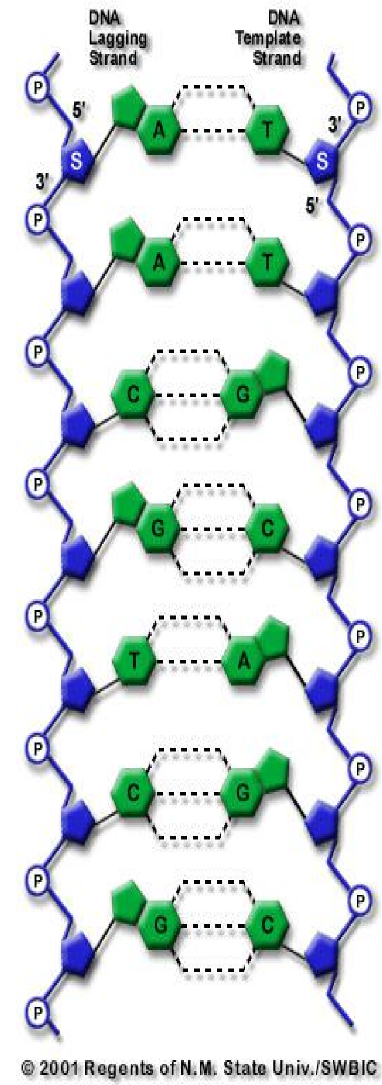
  and

  S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA,

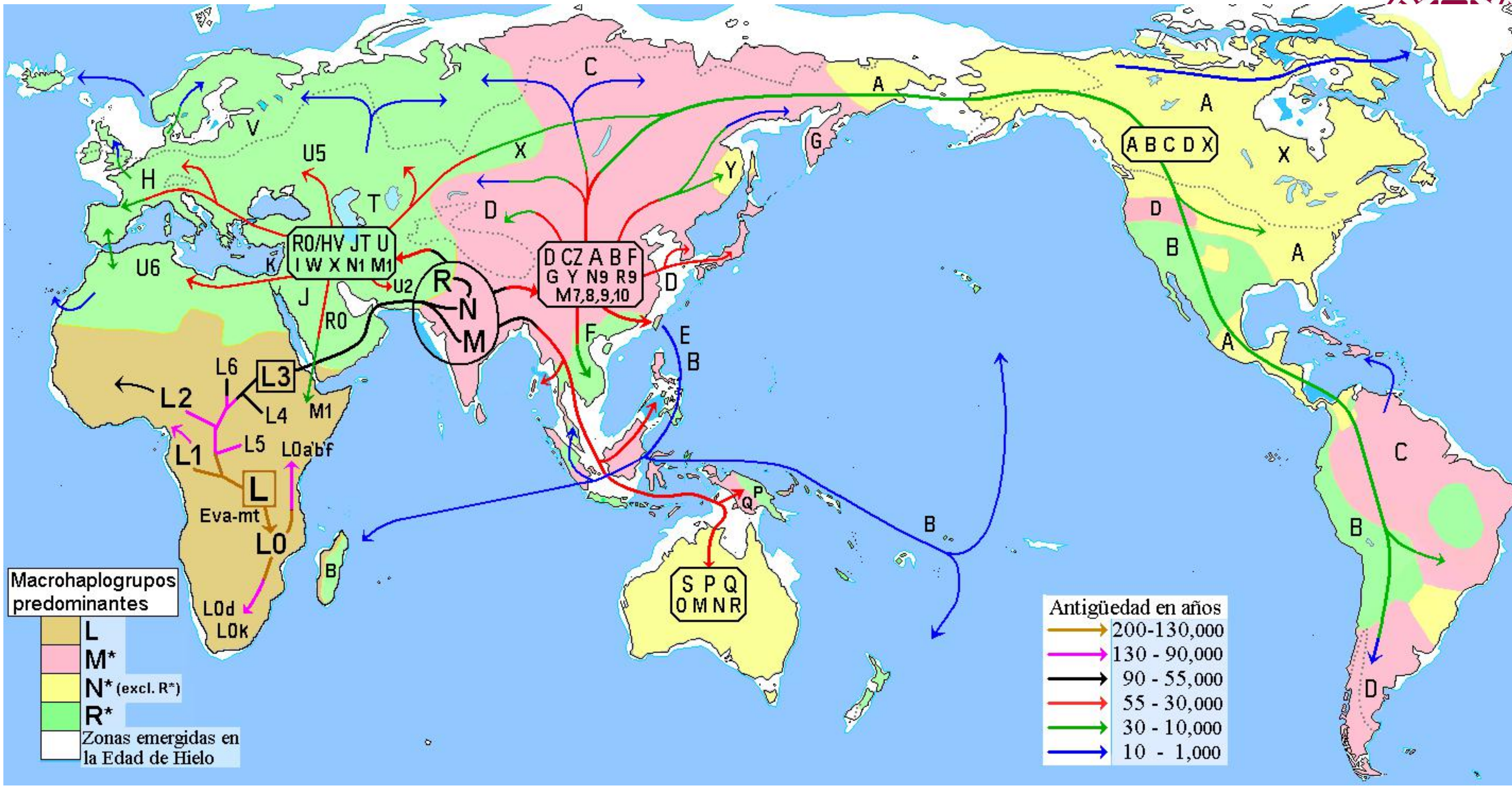
  how to compare them?

  We have various standards of similarity for distinct purposes. While, the LCS of S_1 and S_2 is S_3 = GTCGTCGGAAGCCGGCCGAA.





最长公共子串Longest Common Substring 下标连续的LCS。比如{CGGAA}



源出非洲模型（Out-of-Africa model），女性的粒线体DNA（mtDNA）和男性Y染色体的研究，研究人员得出结论，她们都是来自非洲的同一个女人的后代，这个女人称为粒线体夏娃。

中国人种基因图谱

http://baike.baidu.com/link?url=jg9BzIX1dHqHrrrQaYkUE05PB_eRtUQYJG2_vKlfk5XShXyiWKiHk9W5EuXd6jhzOLrsFsGALPv-x5N-SdifC_

求LCS的step1（1）

- **Step1: LCS最优解的结构特征**

定义X的i^{th}前缀：$X_i=(x_1,x_2,\ldots,x_i)$，$i=1\sim m$

$X_0=\varphi$　　φ为空集

- **Th15.1 (一个LCS的最优子结构)**

设序列$X=(x_1,x_2,\ldots,x_m)$和$Y=(y_1,y_2,\ldots,y_n)$，$Z=(z_1,z_2,\ldots,z_k)$是X和Y的任意一个LCS，则

(1)若$x_m=y_n$, ==> $z_k=x_m=y_n$且Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS;

(2)若$x_m<>y_n$且$z_k<>x_m$, ==> Z是X_{m-1}和Y的一个LCS;

(3)若$x_m<>y_n$且$z_k<>y_n$, ==> Z是X和Y_{n-1}的一个LCS;

注：由此可见，2个序列的最长公共子序列可由(1)(2)(3)算出，(2)(3)的解是对应子问题的最优解。因此，最长公共子序列问题具有**最优子结构性质**。

求LCS的step1（2）

- Th15.1 的证明

  (1)若$x_m=y_n$, ==> $z_k=x_m=y_n$且Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS;

  （应用反证法）

  先证：$z_k=x_m=y_n$。

  若z_k<>x_m (也有z_k<>y_n)，则将x_m加到Z后，于是获得X和Y的长度为k+1的CS，与Z是X和Y的LCS矛盾。

  ==> $z_k=x_m=y_n$

  再证：Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS。

  由Z的定义 ==> 前缀Z_{k-1}是X_{m-1}和Y_{n-1}的CS(长度为k-1)

  若Z_{k-1}不是X_{m-1}和Y_{n-1}的LCS，则存在一个X_{m-1}和Y_{n-1}的公共子序列W，W的长度>k-1，于是将z_k加入W之后，则产生的公共子序列长度>k，与Z是X和Y的LCS矛盾。

  ==> Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS

求LCS的step1（3）

- Th15.1 的证明

  (2)若$x_m <> y_n$且$z_k <> x_m$, ==> Z是X_{m-1}和Y的一个LCS;

  ∵ $z_k <> x_m$，则Z是 X_{m-1}和Y的一个CS

  下证：Z是X_{m-1}和Y的LCS

  （反证）若不然，则存在长度>k的CS序列W，

  显然，W也是X和Y的CS，但其长度>k，矛盾。

  (3)若$x_m <> y_n$且$z_k <> y_n$, ==> Z是X和Y_{n-1}的一个LCS;

  (3)与(2)对称，类似可证。

  综上，定理15.1证毕。 □

求LCS的step2

- Step2: 子问题的递归解
  - 定理15.1将X和Y的LCS分解为：

    (1)if $x_m=y_n$ then　　　//解一个子问题

    　　找X_{m-1}和Y_{n-1}的LCS；

    (2)if $x_m<>y_n$ then　　　//解二个子问题

    　　找X_{m-1}和Y的LCS 和 找X和Y_{n-1}的LCS；

    　　取两者中的最大的；
  - c[i,j]定义为X_i和Y_j的LCS长度，i=0~m, j=0~n；

$$c[i,j]=\begin{cases} 0 & i=0\ or\ j=0 \\ c[i-1,j-1]+1 & i,j>0\ and\ x_i=y_j \\ \max\{c[i,j-1],c[i-1,j]\} & i,j>0\ and\ x_i<>y_j \end{cases}$$

求LCS的step3（1）

- Step3: 计算最优解值
  - 数据结构设计

c[0..m, 0..n]　　//存放最优解值，计算时行优先

b[1..m, 1..n]　　//解矩阵，存放构造最优解信息

$$
b[i, j] = \begin{cases} \nwarrow & \text{如果}c[i, j]\text{由}c[i-1, j-1]\text{确定} \\ \uparrow & \text{如果}c[i, j]\text{由}c[i-1, j]\text{确定} \\ \leftarrow & \text{如果}c[i, j]\text{由}c[i, j-1]\text{确定} \end{cases}
$$

当构造解时，从b[m,n]出发，上溯至i=0或j=0止

上溯过程中，当b[i,j]包含“↖”时打印出xi(yj)

求LCS的step3（2）

- **Step3: 计算最优解值**

—算法

```
LCS_Length(X, Y)
{   m ←length[X];  n ←length[Y];
    for  i←0  to m  do  c[i,0] ←0;    //0列
    for  j←0  to n  do  c[0,j] ←0;    //0行
    for  i←1  to m  do
        for j←1  to n  do
            if xi=yj  then
             {   c[i, j]←c[i-1, j-1] +1; b[i, j] ← "↖ "; }
            else
                if c[i-1, j]>=c[i, j-1] then
                 {   c[i, j]←c[i-1, j]; b[i, j] ← "↑"; }   //由Xi-1和Yj确定
            else
                {  c[i, j]←c[i, j-1]; b[i, j] ← "←"; }   //由Xi和Yj-1确定
   return  b and  c;
  }
```

时间： θ (mn)

求LCS的step3（3）

Example

j / i	0 y_j	1 B	2 D	3 C	4 A	5 B	6 A
0 x_i	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5 D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6 A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7 B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

求LCS的step4

- Step4: 构造一个LCS
  - 算法

```
Print_LCS(b, X, i, j)
{   if i=0 or j=0 then return;
    if b[i,j]="↖ "  then
     {   Print_LCS(b, X, i-1, j-1);
         print  xi;
     }
    else
         if b[i,j]="↑" then Print_LCS(b, X, i-1, j);
         else  Print_LCS(b, X, i, j-1);
}
```

时间：θ(m+n)