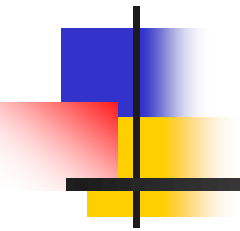


动态规划

引论

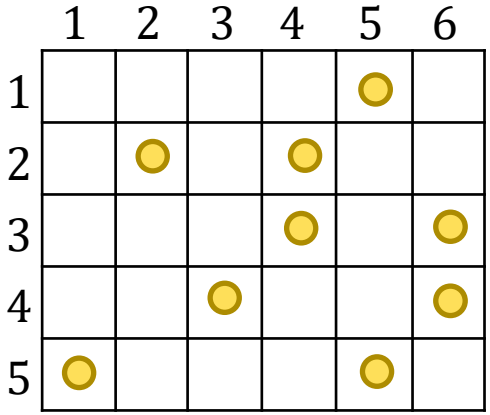
- 动态规划法在本课程介绍的算法设计方法中是最难的！
- 应用：
  1. 硬币收集问题
  2. 多段图最短路径
  3. 0/1背包问题
  4. 矩阵乘法链
  5. All-Pair 最短路径
  6. 不交叉网的子集
  7. 旅行商问题 (TSP)

硬币收集问题

- 在 n × m 格木板中放有一些硬币，每格的硬币数目最多为一个。
- 在木板左上方的一个机器人需要收集尽可能多的硬币并把它们带到右下方的单元格。
- 每一步，机器人可以从当前的位置向右移动一格或向下移动一格。
- 当机器人遇到一个有硬币的单元格时，就会将这枚硬币收集起来。
- 设计一个算法：
  - 找出机器人能找到的最大硬币数，
  - 并给出相应的路径。

硬币收集问题

- 木板上初始的硬币格局及对应矩阵 $C_{n \times m}$：



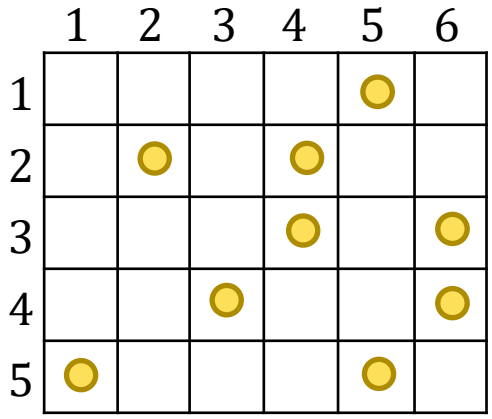
C=

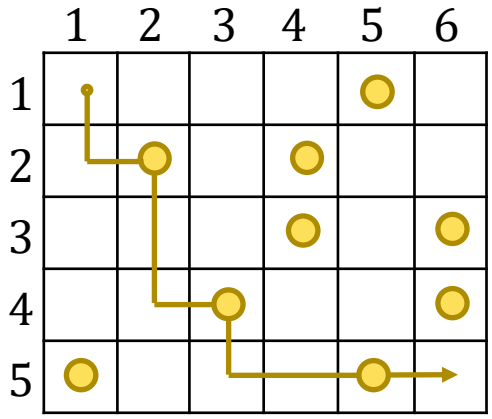
	1	2	3	4	5	6
1	0	0	0	0	1	0
2	0	1	0	1	0	0
3	0	0	0	1	0	1
4	0	0	1	0	0	1
5	1	0	0	0	1	0

- 矩阵 $C_{n \times m}$ 中元素 c_{ij} 是一个 0/1 变量。当单元格 (i, j) 有硬币时，$c_{ij}=1$， 否则为 0。

硬币收集问题

- 单源单点最长路径！
- 贪心算法：每次选令硬币数目最多的单元格。
- 木板上初始的硬币格局如图：

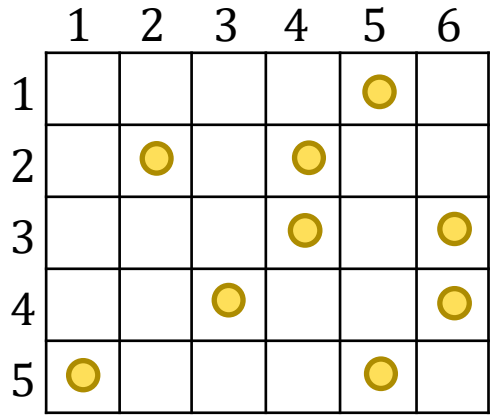




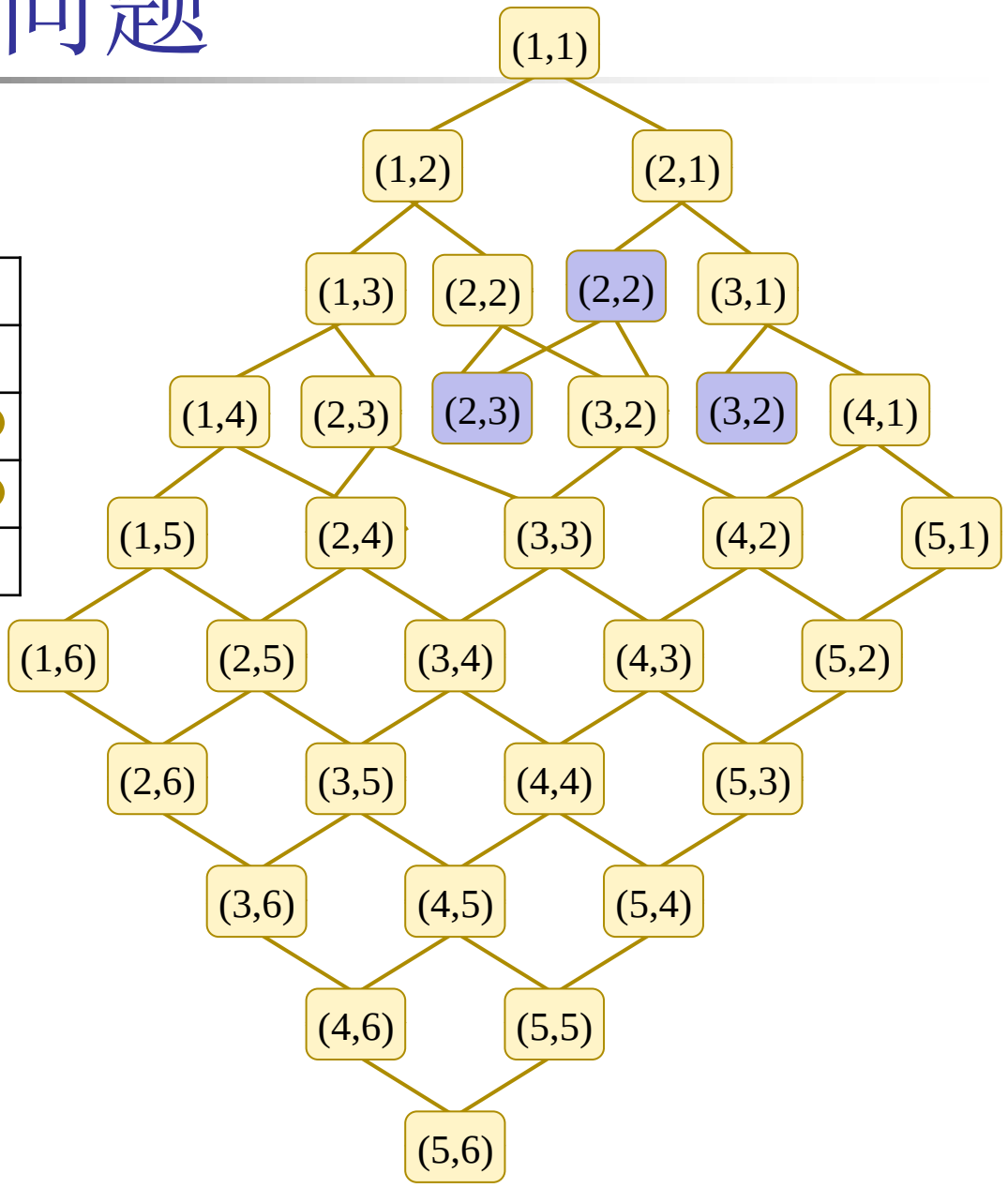
- 贪心解为 3 枚硬币，不是最优解！
- 时间复杂度为 $T(n, m) = \Theta(n + m)$

硬币收集问题

分治法：

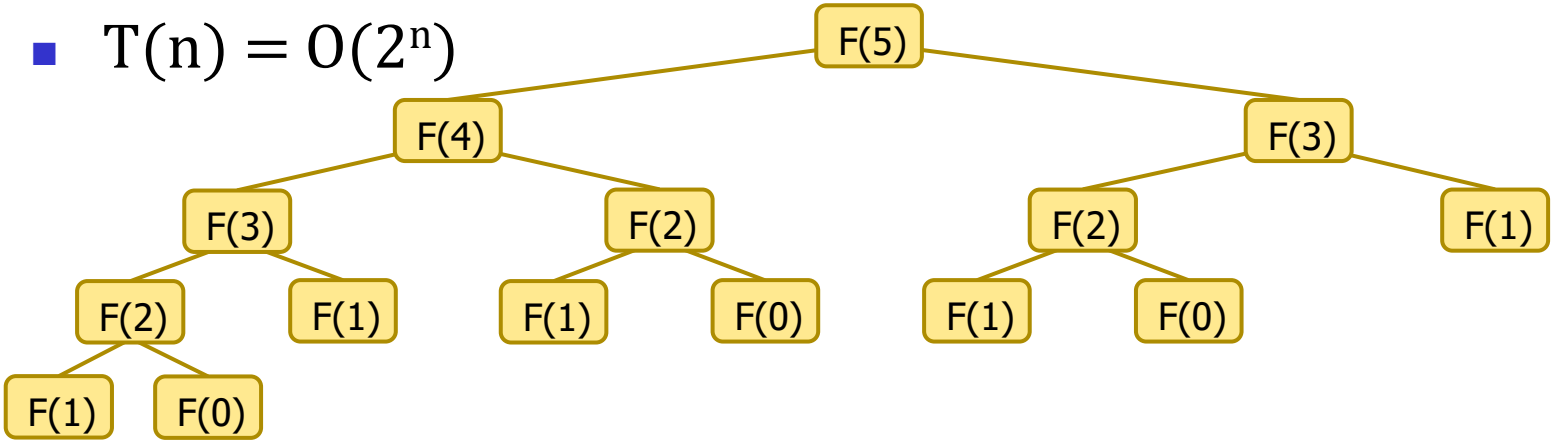


$$T(n, m) = T(n, m-1) + T(n-1, m)$$



斐波那契数列

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- 分治法求解需要对该函数的相同值重复计算好几遍：
- $F(n) = F(n-1) + F(n-2)$
- $T(n) = O(2^n)$



- 如果只存储斐波那契序列中最后两个元素的值，可以避免使用额外的数组来完成这个任务。

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)
0	1	1	2	3	5

硬币收集问题

- 对策：也设计一个表格，避免重复运算。
- F(i, j)：机器人到单元格 (i, j) 时收集到的最大硬币数。
  - $F(0, j) = 0; F(i, 0) = 0.$
  - 单元格 (i-1, j) 处的最大硬币数为 F(i-1, j)；
  - 单元格 (i, j-1) 处的最大硬币数为 F(i, j-1)。
- 单元格 (i, j) 可以经由上方格 (i-1, j) 或左侧格 (i, j-1) 到达。
- $F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij}, c_{ij} \in \{0, 1\}.$

硬币收集问题动态规划算法

- 通过公式 $F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij}$ 得到 F 矩阵：

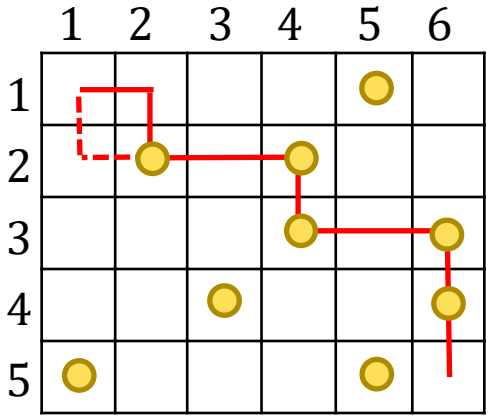
$C =$

	1	2	3	4	5	6
1	0	0	0	0	1	0
2	0	1	0	1	0	0
3	0	0	0	1	0	1
4	0	0	1	0	0	1
5	1	0	0	0	1	0

$F =$

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	5

- 沿最大值回溯得到最优路径：



- 计算每个单元格的值 $F(i, j)$ 花费常量时间，算法时间复杂性为 $T(n, m) = \Theta(nm)$.

硬币收集问题动态规划算法

- 得到的是最优解！
- **策略**：把可能的值填入表格，不着急做决定，待触底后回溯。

RobotCoinCollection(C[1..n, 1..m])

1. F[1, 1] ← C[1, 1]
2. **for** j ← 2 **to** m **do** F[1, j] ← F[1, j-1] + C[1, j]
3. **for** i ← 2 **to** n **do**
4. F[i, 1] ← F[i-1, 1] + C[i, 1]
5. **for** j ← 2 **to** m **do**
6. F[i, j] ← max(F[i-1, j], F[i, j-1]) + C[i, j]
7. **return** F[n, m]

动态规划

- **动态规划**是运筹学的一个分支。20世纪50年代初美国数学家**理查德·贝尔曼**等人在研究多阶段决策过程的优化问题时，提出了著名的最优性原理，把多阶段过程转化为一系列单阶段问题，逐个求解，创立了解决这类过程优化问题的新方法——动态规划。
- **多阶段决策问题**：求解的问题可以划分为一系列相互联系的阶段，在每个阶段都需要做出决策，且一个阶段决策的选择会影响下一个阶段的决策，从而影响整个过程的活动路线。求解的目标是选择各个阶段的决策使整个过程达到最优。
- 动态规划主要用于求解以时间划分阶段的动态过程的优化问题。

最优性原理

- Bellman**最优性原理**：

  An optimal policy has the property that whatever the initial state and initial decision are, then remaining decisions must constitute an optimal policy with regard to the state resulting from first decision.

  不论初始状态和初始决策如何，对于前面决策所造成的某一状态而言，其后各阶段的决策序列必须构成最优策略。

- **注**：对具有最优性原理性质的问题而言，如果有一个决策序列包含有非最优的决策子序列，则该决策序列一定不是最优的。
- 如果待求解问题的一个最优策略序列的子策略序列总是最优的，则称该问题满足最优性原理。

一些术语和概念

- 阶段：把所给的问题的求解过程恰当地划分为若干个相互联系的阶段。
- 状态：状态表示每个阶段开始时问题或系统所处的客观状态。状态既是该阶段的某个起点，又是前一个阶段的某个终点。通常一个阶段有若干个状态。
  - 状态的无后效性：如果某阶段状态给定后，则该阶段以后过程的发展不受该阶段以前各阶段状态的影响，也就是说状态具有马尔可夫性质。
  - 注：适于动态规划法求解的问题具有状态的无后效性。
- 策略：各个阶段的决策确定后，就组成了一个决策序列，该序列称之为一个策略。由某个阶段开始到终止阶段的过程称为子过程，其对应的某个策略称为子策略。

马尔可夫性质(Markov property)

- 马尔可夫性质因为俄国数学家安德雷·马尔可夫得名，是概率论中的一个概念。
- 当一个随机过程在给定现在状态及所有过去状态的情况下，其未来状态的条件概率分布仅依赖于当前状态；换句话说，在给定现在状态时，它与过去状态（即该过程的历史路径）是条件独立的，那么此随机过程即具有马尔可夫性质。
- 具有马尔可夫性质的过程通常称之为马尔可夫过程。

动态规划原理

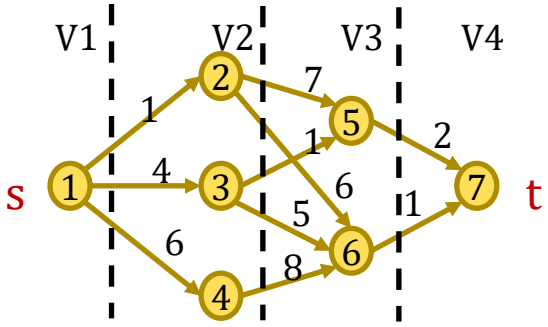
- 从算法设计的角度看, 动态规划是一种在各个不同大小(size)的子问题的优化值之间建立递归关系并求解的过程。
- 能用动态规划求解的问题必须满足优化原理: 优化解包含的子问题的解也是优化的。
- 利用优化原理，使用枚举法建立不同长度子问题的优化值之间的递归关系——动态规划方程.
- 动态规划得到的是精确解。
- 子问题的数目决定算法的复杂性。
- 实现时要尽可能消去递归。

动态规划

- 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。
- 一些与时间无关的静态规划，如线性规划、非线性规划等，可以人为地引进时间因素，把它视为多阶段决策过程，也可以用动态规划方法方便地求解。

多段图(multistage graph)

- 多段图G=(V, E)是一个带权有向图，它具有如下特性：
  - 图中的结点被划分成 $k \geq 2$ 个互不相交的子集合 V_i，$1 \leq i \leq k$。
  - V_1 和 V_k 分别只有一个结点，V_1 包含源点 s，V_k 包含汇点 t。
  - 对所有边 $<u, v> \in E$, 若 $u \in V_i$, 则$v \in V_{i+1}$, $1 \leq i \leq k$。
  - 每条边的权值为 $c(u, v)$。从 s 到 t 的路径长度是这条路径上边的权值之和。若$<u, v> \notin E$，则边上成本记 $c(u, v) = \infty$。
- 多段图问题是寻找从 s 到 t 的一条长度最短的路径。



多段图最短路径

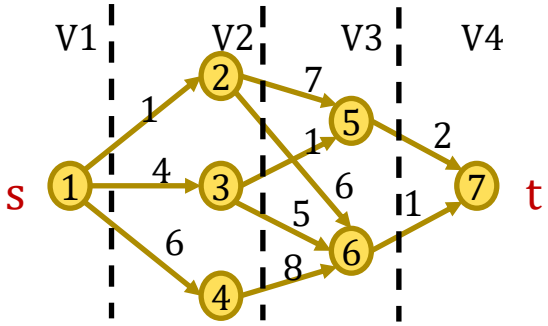
- 多段图问题满足优化原理:

  设 s, …, u, …, v, …, t 是一条由 s 到 t 的最短路径，则u, …, v, …, t 也是由 u 到 t 的最短路径。(反证即可)

- 设 c(i) 为结点 i 到汇点的最短路长度, A(i)为 i 的邻结点集合，有$c(t) = 0$，

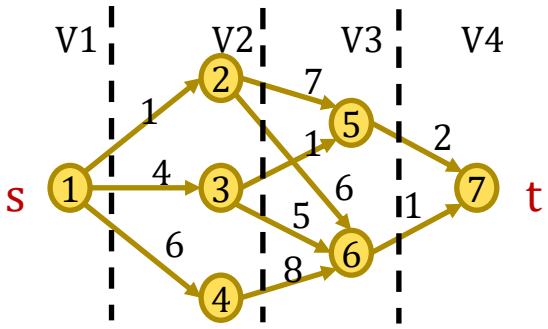
$$c(i) = \min_{j \in A(i)}\{c(j)+cost(i, j)\}$$

- c(i) 由 i 到汇点的子图决定, 和源点怎样走到 i 没有关系. (Markov性质)

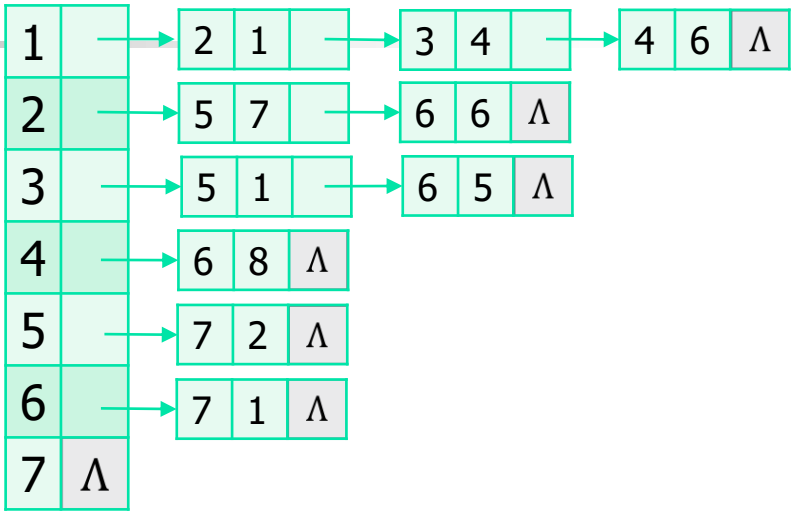


多段图最短路经

- 多段图问题满足优化原理: 最短路 $(1 \to 3 \to 5 \to 7)$ 上的子路径 $(3 \to 5 \to 7)$ 是 3 到目的节点 7 在子图上的最短路。
- 节点 1 到目的节点的最短路长度 $c(1)$ 可从 2, 3, 4 到目的节点的最短路长度 $c(i) + cost(1,i)$ 经枚举得到。
- 但 2, 3, 4 到目的节点的最短路长度 $c(2)$, $c(3)$, $c(4)$ 还不知道。我们须计算$c(2),c(3),c(4)$; 仍使用优化原理。
- $c(i) = \min_{j \in A(i)}\{c(j)+cost(i, j)\}$
- $c(7)=0$。



多段图最短路径



$c(i) = \min_{j \in A(i)}\{c(j)+cost(i, j)\}$

初始c(7)=0，

依次计算c(6), …,c(1)：

c(6)=1, c(5)=2,

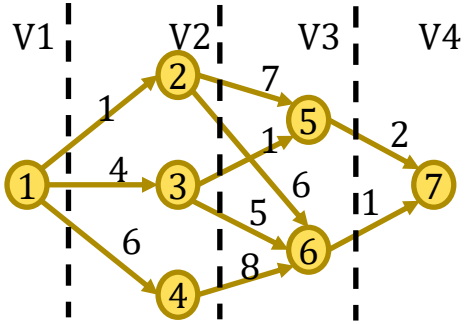
c(4)=8+c(6)=9

c(3) = min{c(5)+cost(3, 5), c(6)+cost(3, 6)}

= min{2+1, 1+5} = 3,

c(2) = min{c(5)+cost(2, 5), c(6)+cost(2, 6)}

= min{2+7, 1+6} = 7,



c(1) = min{c(2)+cost(1, 2), c(3)+cost(1, 3), c(4)+cost(1,4)}

= min{7+7, 3+4, 9+6} = 7。

回溯得最短路径 <1, 3, 5, 7>。递归还可从前向后.

动态规划方法的求解步骤:

1. 验证待求解的问题是否满足优化原理；找出最优解的性质，并刻画其结构特征；
2. 应用优化原理建立子问题优化解的值(优化值)之间的递归式，递归地定义最优值 (写出动态规划方程)
3. 以自底向上的方式计算出满足递归式的最优值。
4. 根据计算最优值时记录的信息回溯(traceback)，从优化值构造最优解。

注：

- 步骤 1、2、3 是动态规划算法的基本步骤。如果只需要求出最优值的情形，步骤 4 可以省略；
- 若需要求出问题的一个最优解，则必须执行步骤 4，步骤 3 中记录的信息是构造最优解的基础。

算法复杂性

- 直接用递归实现动态规划递归方程往往会引发大量重复计算，使得算法的计算量变得非常可观。最好使用迭代法实现动态规划算法。
- 迭代实现需要存贮所有子问题的优化解的值 f(i, y)，以便避免重复计算，所以算法往往需要较大的存储空间。
- 算法的复杂性来自子问题的数目，通常子问题的数目很大。

适用条件

动态规划方法的有效性依赖于问题本身所具有的两个重要的适用性质：

1.最优子结构：如果问题的最优解是由其子问题的最优解来构造，则称该问题具有最优子结构性质。

2.重叠子问题：在用递归算法自顶向下解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只解一次，而后将其解保存在一个表格中，在以后该子问题的求解时直接查表。

最优性原理判别举例

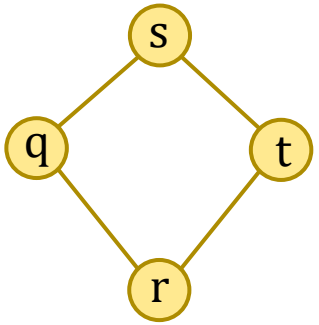
例 1：设 G 是一个有向加权图，则 G 从顶点 i 到顶点 j 之间的最短路径问题满足最优性原理。

证明：(反证)

- 设 i … p … q … j 是一条最短路径，但其中子路径 p … q … j 不是最优的。
- 假设最优的路径为p … q’ … j，则我们重新构造一条路径 i … p … q’ … j，显然该路径长度小于 i … p … q … j。
- 与 i … p … q … j 是顶点 i 到顶点 j 的最短路径相矛盾.
- 所以，原问题满足最优性原理。

最优性原理判别举例

例 2：最长路径问题不满足最优性原理。



证明：

- $q \to r \to t$ 是 q 到 t 的最长路径;
- q 到 r 的最长路径是$q \to s \to t \to r$;
- r 到 t 的最长路径是$r \to q \to s \to t$。
- q 到 r 和 r 到 t 的最长路径合起来不是 q 到 t 的最长路径。
- 所以，原问题并不满足最优性原理。

注：因为 q 到 r 和 r 到 t 的子问题都共享路径$s \to t$，组合成原问题解时，有重复的路径对原问题是不允许的。

最优性原理判别举例

例 3 ： 0-1背包问题Knap(1,n,c)满足最优性原理。

证明：

- 设 $(y_1, y_2, \cdots, y_n)$ 是 $Knap(1, n, c)$ 的一个最优解
- $(y_2, \cdots, y_n)$ 是 $Knap(2, n, c - w_1y_1)$ 子问题的一个最优解。
- 否则, 设 $(z_2, \cdots, z_n)$ 是 $Knap(2, n, c - w_1y_1)$ 的最优解, 则$(y_1, z_2, \cdots, z_n)$ 是 $Knap(1, n, c)$ 的一个更最优解。
- 与假设矛盾。

设计技巧

- 动态规划的设计技巧：阶段的划分，状态的表示和存储表的设计；
- 在动态规划的设计过程中，阶段的划分和状态的表示是其中重要的两步，这两步会直接影响该问题的计算复杂性和存储表设计，有时候阶段划分或状态表示的不合理还会使得动态规划法不适用。
- 记忆型递归——动态规划的变种：
  - 每个子问题的解对应一表项；
  - 每个表项初值为一特殊值，表示尚未填入；
  - 递归时，每一次遇到子问题进行计算并填表，以后查表取值；

如：后面将要讲到的矩阵乘法链的记忆型递归算法。

存在的问题

- 问题的阶段划分和状态表示，需要具体问题具体分析，没有一个清晰明朗的方法。
- 空间溢出的问题，是动态规划解决问题时一个普遍遇到的问题；
- 动态规划需要很大的空间以存储中间产生的结果，包含同一个子问题的所有问题共用一个子问题解，从而体现动态规划的优越性。但这是以牺牲空间为代价的，为了有效地访问已有结果，数据也不易压缩存储，因而空间矛盾是比较突出的。

动态规划 V.S. 贪心算法

- 贪心算法通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。
- 贪心算法是在解决子问题之前做出选择，然后希望做出的选择是正确的，每一个选择都不可撤回。
- 动态规划是在一系列选择之后，挑选一个能取得最优解的选择。
- 动态规划算法通常以自底向上的方式求解子问题。
- 一般情况下，贪心算法比动态规划算法快。但我们并不总能简单地判断出贪心算法是否有效。

动态规划 V.S. 分治算法

- 动态规划的思想实质: 分治思想和解决冗余。
  - 与分治法类似的是: 将原问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。
  - 与分治法不同的是: 经分解的子问题往往不是互相独立的。若用分治法来解，有些共同部分(子问题或子子问题)被重复计算了很多次。
- 动态规划法的基本思路: 用一个表来记录所有已解的子问题的答案，当其重复出现时查表即可避免重复求解。
- 具体的动态规划算法多种多样，但它们具有相同的填表方式。

0/1背包问题

- 假设你是个小偷，背着一个可装 4kg 东西的背包进入一个宿舍。可盗窃的物品有三件：平板 (1kg, 1500元)；笔记本 (4kg, 3000元)；吉他 (3kg, 2000元) 。为了让盗窃的商品价值最高，你该选择哪些商品？
- 贪心法可以给出近似解，但不是最优解。
- 枚举尝试各种可能的商品组合，可以找出价值最高的组合。
- 但是，速度太慢！2^n种可能，太危险了！☺

动态规划解法

平板 (1kg, 1500元)；笔记本(4kg, 3000元)；吉他(3kg, 2000元)。以最小重量单位做分隔做表格：

- 待选物品只有平板时：

	1	2	3	4
平板	1500	1500	1500	1500

- 待选物品为平板和笔记本时：

	1	2	3	4
平板	1500	1500	1500	1500
笔记本	1500	1500	1500	3000

- 吉他也可以选了：

	1	2	3	4
平板	1500	1500	1500	1500
笔记本	1500	1500	1500	3000
吉他	1500	1500	2000	3500

- $c_{ij} = \max \begin{cases} \text{上一个单元格的值 } c_{i-1,\,j} \\ \text{当前物品的价值} + \text{剩余空间的价值} \end{cases}$

增加一件物品

- 平板(1kg, 1500元)；笔记本(4kg, 3000元)；吉他(3kg, 2000元)。
- 又发现一个iPhone (1kg, 2000元)！

	1	2	3	4
平板	1500	1500	1500	1500
笔记本	1500	1500	1500	3000
吉他	1500	1500	2000	3500
iPhone	2000	3500	3500	4000

递增

最大值

递增

- 如果再发现一件物品相机(1kg, 1000)，要偷走吗？

换一个顺序

- 笔记本(4kg, 3000元); 吉他(3kg, 2000元); 平板(1kg, 1500元)。改变选择的顺序, 会不会影响结果？
- $c_{ij} = \max \begin{cases} \text{上一个单元格的值 } c_{i-1,\,j} \\ \text{当前物品的价值} + \text{剩余空间的价值} \end{cases}$
- 待选物品只有平板时：

	1	2	3	4
笔记本	0	0	0	3000

- 待选物品为平板和笔记本时：

	1	2	3	4
笔记本	0	0	0	3000
吉他	0	0	2000	3000

- 游戏本也可以选了：

	1	2	3	4
笔记本	0	0	0	3000
吉他	0	0	2000	3000
平板	1500	1500	2000	3500

同样的结果

背包增容1kg

- 如果背包增容1kg?

	1	2	3	4	5
平板	1500	1500	1500	1500	1500
笔记本	1500	1500	1500	3000	4500
吉他	1500	1500	2000	3500	4500

增加一件更小的物品

- 平板(1kg, 1500元)；笔记本(4kg, 3000元)；吉他(3kg, 2000元)。最小重量单位1kg。
- 又发现一条手链(0.5kg, 1000元)
- 最小重量单位: 0.5kg

	0.5	1	1.5	2	2.5	3	3.5	4
平板	0	1500	1500	1500	1500	1500	1500	1500
笔记本	0	1500	1500	1500	1500	1500	1500	3000
吉他	0	1500	1500	1500	1500	2000	2000	3500
手链	1000	1500	2500	2500	2500	2500	3000	3000

可以偷商品的一部分吗？

- 贪心算法处理连续背包问题可以得到最优解。
- 动态规划无法处理连续背包问题。
  - 原因就是没法画格！☹
  - 要么拿走整件，要么不拿！
- 动态规划最优解也可能导致背包没装满。
  - 假设除了平板、笔记本、吉他，你还可以偷一个模型 (3.5kg, 100万元)。
  - 当然偷模型，哪怕背包还剩0.5kg！☺☺☺

	0.5	1	1.5	2	2.5	3	3.5	4
平板	0	1500	1500	1500	1500	1500	1500	1500
笔记本	0	1500	1500	1500	1500	1500	1500	3000
吉他	0	1500	1500	1500	1500	2000	2000	3500
模型	0	1500	1500	1500	1500	2000	1M	1M

0/1背包问题

- **问题**：背包容量不足以装入所有物品，面临选择。
- **目标函数**：效益值最大(一组非负数之和)。
- **可行解**：物品 $1, \cdots, n$ 的一种放法 ($x_1, \cdots, x_n$ 的 0/1 赋值)；
- **优化解**：使得效益值最大的一种放法 ($z_1, \cdots, z_n$)。
- **优化原理**: 无论优化解 ($z_1, \cdots, z_n$) 是否放物品 1，相对剩余背包容量，优化解 ($z_2, \cdots, z_n$) 对物品 $2, \cdots, n$ 的放法也是优化解。(最优子结构性质)
- 背包问题满足优化原理，可用反证法证明。

优化值间的递归式

- 虽然我们不知道优化解是否放物品 1，但我们可以利用优化原理，从枚举“放”和“不放”两种情形建立优化值之间的递归式。

- 设 f(i, y) 为以背包容量 y，放物品 i, …, n 得到的优化效益值，按优化原理可列递归关系如下：

$$f(1, c) = \max\{f(2, c), f(2, c-w_1)+p_1\};$$

$$f(i, y) = \begin{cases} \max\{f(i+1, y), f(i+1, y-w_i)+v_i\} & y \geq w_i \\ f(i+1, y) & 0 \leq y < w_i \end{cases}$$

背包容量 y 不足以放下物品 i

- 边界条件

$$f(n, y) = \begin{cases} v_n & y \geq w_n \\ 0 & 0 \leq y < w_n \end{cases}$$

0/1背包问题的递归实现

```
1.  int f( int i, int y )
2.  { // 返回 f(i, y)
3.      if ( i == n ) {
4.          f[i][y] = (y < w[n] ? 0:v[n]);
5.          return f[i][y];
6.      }
7.      if ( y < w[i] ) f[i][y] = f(i+1, y);
8.      else
9.          f[i][y] = max( f[i+1][y], f[i+1][y-w[i]] + v[i] );
10.     return f[i][y];
11. }
```

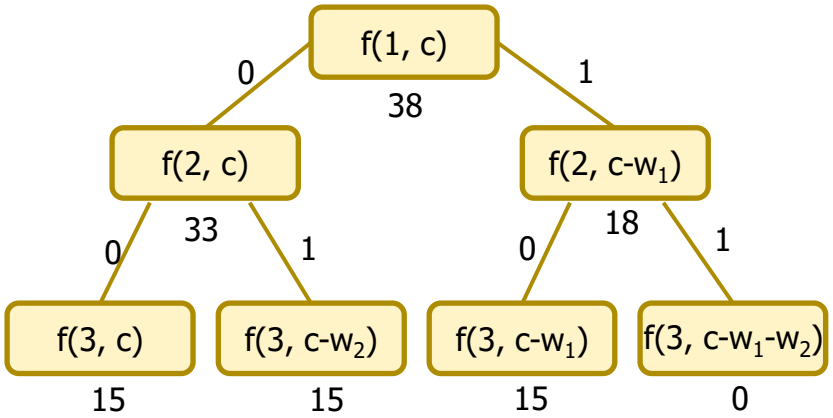
- 最坏时间复杂度: $T(n) = 2T(n-1)+c = \Theta(2^n)$

0/1背包问题的DP算法例

例：n=3, w=[100,14,10], v=[20,18,15], c= 116.

解:

$$f(3,y)=\begin{cases}15 & y\geq 10\\ 0 & 0\leq y<10\end{cases}$$



$$f(2,y)=\begin{cases}\max\{f(3,y),\ f(3,y-14)+18\} & y\geq 14\\ f(3,y) & 0\leq y<14\end{cases}$$

$f(1,c)=\max\{f(2,c),\ f(2,c-w_1)+20\}$;

最优值为38.

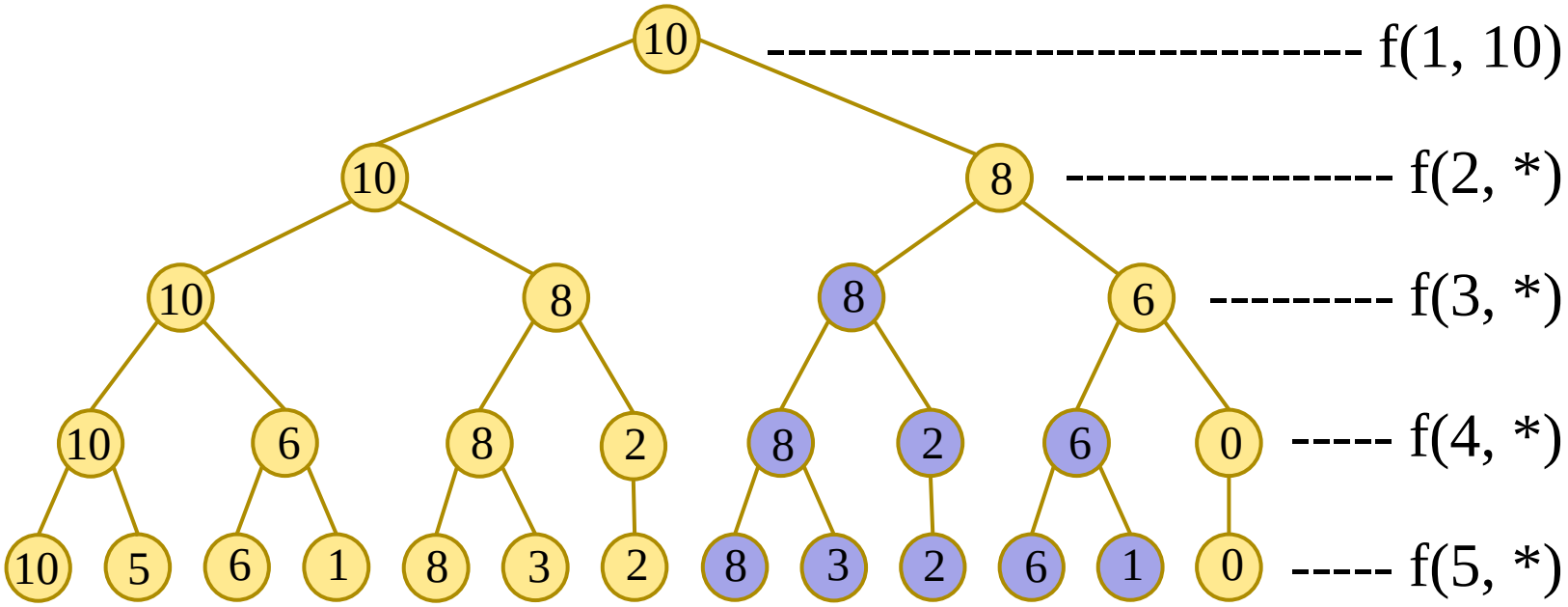
回溯：$f(1,c)\neq f(2,c)$, $x_1=1$; $f(2,c-w_1)\neq f(3,c-w_1)$, $x_2=1$; 此时r=16-14=2, 不足以放物品3, 所以$x_3=0$。

0/1背包问题的回溯

```
1.  template<class T>
2.  void Traceback(T  **f, int w[], int c, int n, int x[])
3.  {// 计算x
4.     for (int i = 1; i < n; i++)
5.        if  (f[i][c] == f[i+1][c]) x[i] = 0;
6.        else {
7.           x[i] = 1;
8.           c -= w[i];
9.        }
10.    x[n]=(f[n][c]) ? 1 : 0;
11. }
```

权取整数时0/1背包问题例

- 例：n = 5, p = [6, 3, 5, 4, 6], w = [2, 2, 6, 5, 4], c = 10, 求 f(1, 10)。
- 解：为了确定 f(1, 10), 调用函数 f(i, y)。递归调用关系如下的树型结构所示:



- 28 次递归调用中有 9 次 (蓝色节点) 是重复的。

权取整数时0/1背包问题例

- n=5,v=[6,3,5,4,6],w=[2,2,6,5,4] 且c=10 ,求f (1,10).
- 当物品重量为整数时, 可用二维数组 f[i][y] 来保存每个f(i, y)的值, 避免重复计算。

	y										
i	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11

- 算法时间复杂性 $\Theta(nc)$，二维数组需 $\Theta(nc)$ 空间。当背包容量 c 很大时, 例如 $c > 2^n$, 算法要求的存储空间为$\Omega(n2^n)$。
- 函数 Traceback 从 f[i][y] 产生优化的 x_i 值。复杂性为$\Theta(n)$。
- f(1,10)=max{f(2,10), f(2,10-2)+6}=15.

权取整数时0/1背包问题的迭代实现

```
1.  template  <class T>
2.  void Knapsack(T v[], int w[], int c, int n, T **f) {
3.      for (int y = 0; y <= MAX; y++)  f[n][y] = 0;
4.      for (int y = w[n]; y <= c; y++) f[n][y] = v[n];
5.      for (int i = n - 1; i > 1; i--)
6.      {
7.          for (int y = 0; y <= MAX; y++)
8.              f[i][y]=f[i+1][y];
9.          for (int y = w[i]; y <= c; y++)
10.             f[i][y] = max(f[i+1][y], f[i+1][y-w[i]]+ v[i]);
11.     }
12.     f[1][c] = f[2][c];
13.     if (c >= w[1])
14.         f[1][c] = max(f[1][c], f[2][c-w[1]] + v[1]);}
```

权取整数时0/1背包问题回溯算法

```
1.  template <class T>
2.  void Trackback(T **f, int w[], int c, int n, int x[])
3.  {
4.      for (int i = 1; i < n; i++)
5.          if (f[i][c] == f[i+1][c]) x[i] = 0;
6.      else
7.      {
8.              x[i] = 1;
9.              c -= w[i];
10.     }
11.     x[n] = f[n][c]?1:0;
12. }
```

权取整数时0/1背包问题

- 当物品重量为整数时, 可用二维数组 f[i][y] 来保存每个 f(i, y) 的值, 避免重复计算。
- 二维数组需 $\Theta(nc)$ 空间.
- Knapsack 的复杂性 $\Theta(nc)$。当背包容量 c 很大时, 例如$c > 2^n$, 算法要求的存储空间为$\Omega(n2^n)$。
- 函数 Traceback 从 f[i][y] 产生优化的 x_i 值。复杂性为 $\Theta(n)$。

权取整数时0/1背包问题例

- n=5,v=[6,3,5,4,6],w=[2,2,6,5,4] 且c=10 ,求f (1,10).
- f(1,10)=max{f(2,10), f(2,10-2)+6}=15.

	y										
i	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11

- 观察表格中 f(i, y)，发现里面的跳跃点很少。

元祖法

	y										
i	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11

将函数 f(i, y) 的跳跃点以元组 (y, f(i, y)) 形式存储于一个线性表 P(i) 中.

表 P(i) 中的元组 (y,f(i, y)) 按 y 的增序排列.

P(5)=[(0,0),(4,6)]; // 对应x5=0和x5=1

P(4)=[(0,0),(4,6),(9,10)]; // 对应00, 01和11

P(3)=[(0,0),(4,6),(9,10),(10,11)]; // 对应000, 001, 011, 110

P(2)=[(0,0),(2,3),(4,6),(6,9),(9,10),(10,11)].

//对应0000, 1000, 0001, 1001, 0011, 0110

元祖法

- n = 5, w = [2, 2, 6, 5, 4], v = [6, 3, 5, 4, 6], c = 10。

(w5, v5) = (4, 6)

P(5) = [(0, 0), (4, 6)];

(w_4, v_4) = (5, 4), Q = {(5, 4), (9, 10)}

P(4) = [(0, 0), (4, 6), (9, 10)];

(w_3, v_3) = (6, 5), Q = {(6, 5), (10, 11)}

P(3) = [(0, 0), (4, 6), (9, 10), (10, 11)]; (6, 5)

(w_2, v_2) = (2, 3), Q = {(2, 3), (6, 9)}

P(2) = [(0, 0), (2, 3), (4, 6), (6, 9), (9, 10), (10, 11)]

数对(a,b)和(c,d). 如果a ≤ c且b>d, 则称(a,b)受(c,d)支配.

- P(i) = P(i+1) + Q.
- P(i+1) 中包含 $x_i = 0$ 的数对, Q 包含 $x_i = 1$ 的数对.
- 合并 P(i+1) 和 Q 时, 删除受支配和重复数对, 得到 P(i) .

元祖法的时间复杂度

- P(i) 中的元组个数至多为 P(i+1) 中元组个数的 2 倍。初始 P(n) = 2，所以，P(i) 中的元组个数不超过 $2^{(n-i+1)}$。
- 计算 Q 需 O(|P(i+1)|) 的时间，合并 P(i+1) 和 Q 需要 O(|P(i＋1)| + |Q|) = O(2|P(i+1)|)的时间。
- 所以计算 P(i) 需 $O(2^{n-i+1})$ 时间.
- 计算所有 P(i) 时所需要的总时间 $O(2^n)$.
- 存在输入使算法最坏情形为 2^n 量级.

完全背包问题

- 有 n 种物品和一个容量为 C 的背包，每种物品都就可以选择任意多个，第 i 种物品的重量为 w_i，价值为 v_i，求解：选哪些物品放入背包，可以使得这些物品的价值最大，并且重量总和不超过背包容量？
- 0/1 背包问题中，每件物品最多选择一件，而在完全背包问题中，只要背包装得下，每件物品可以选择任意多件。
- 从每件物品的角度来说，与之相关的策略已经不再是选或者不选了，而是有取 0 件、取 1 件、取 2 件…，直到取 $[C/w_i]$ (向下取整) 件。
- 完全背包问题贪心算法仍然具有 0/1 背包问题的特点，即可能因为剩余空间无法得到最优收益。
- 反例：$C = 10$， $n = 2$， $W = (5, 7)$，$V = (5, 8)$。

完全背包问题递归公式

- g(i, c) 表示前 i 种物品放入一个容量为 c 的背包获得的最大价值。
- 对于第 i 种物品，我们有 k 种选择，$0 \le k \cdot w_i \le c$，即可以选择 0、1、2、…、k 个第 i 种物品。
- 递推表达式为：

  $$g(i, c) = \max\{g(i-1, c - w_i \cdot k) + v_i \cdot k\} \ (0 \le k \cdot w_i \le c)$$

- 边界条件：$g(0, c) = 0$；$g(i, 0) = 0$。

完全背包问题DP算法

```
if (i == 0 || c == 0){ // 初始条件
    result = 0;
} else if(w_i > c){ // 装不下该珠宝
    result = g(i-1, c);
} else { // 可以装下
    // 取 k 个物品 i，取其中使得总价值最大的 k
    for (int k = 0; k · w_i ≤ c; k++){
        int tmp2 = g(i-1, c - w_i · k) + v_i · k;
        if (tmp2 > result){
            result = tmp2; } } }
return result;
```

完全背包问题的最优性

反证法：

- 已知完全背包的解为 $(x_1, x_2, \cdots, x_n)$，x_i 表示第 i 件物品的选取数量。
- $g(x_1, x_2, \cdots, x_i)$ 为将前 i 种物品按照 $(x_1, x_2, \cdots, x_i)$ 方案放入容量为 c 的背包所取得的价值。
- 假设 $(x_1, x_2, \cdots, x_i)$ 不是子问题的最优解，则存在另一组解 $(y_1, y_2, \cdots, y_i)$，使得 $g(y_1, y_2, \cdots, y_i) > g(x_1, x_2, \cdots, x_i)$。
- 则 $g(y_1, y_2, \cdots, y_i, x_{i+1}, \cdots, x_n) > g(x_1, x_2, \cdots, x_n)$。
- 因此 $(x_1, x_2, \cdots, x_n)$ 不是原问题的最优解，与已知矛盾。
- 所以 $(x_1, x_2, \cdots, x_i)$ 必然是子问题的最优解。

完全背包问题的无后效性

- 后续子问题的解只与背包的剩余空间有关。
- 前 i 种物品如何选择，都不会影响后面物品的选择。
- 即子问题的任意解，都不会影响后续子问题的解。
- 满足无后效性。
- 因此，完全背包问题可以使用动态规划来解决。
- 递归公式就是我们需要的状态转移方程。

多重背包问题

- 有 n 种物品和一个容量为 C 的背包，第 i 种物品最多有 m_i 件可用，重量为 w_i，价值为 v_i。求解：选哪些物品放入背包，可以使得这些物品的价值最大，并且重量总和不超过背包容量。
- 与完全背包问题相比，只是对物品的件数加以限制：
  - 完全背包问题中，物品可以选择任意多件；
  - 多重背包中每种物品都有指定的数量限制。
- 多重背包问题也可以转化成 0/1 背包问题来求解：因为第 i 件物品最多选 m_i 件，于是可以把第 i 种物品转化为 m_i 件体积和价值相同的物品，然后再来求解这个 0/1 背包问题。

多重背包问题递归公式

- f(i, c) 表示前 i 种物品放入一个容量为 c 的背包获得的最大价值。
- 对于第 i 种物品，我们有 k 种选择，即可以选择0、1、2、…、m_i 个第 i 种物品 ($0 \le k \le m_i$ && $0 \le k \cdot w_i \le c$)，所以递推表达式为：

$$f(i, c) = \max\{f(i-1, c - w_i \cdot k) + v_i \cdot k\}$$

- 边界条件：$f(0, c) = 0$；$f(i, 0) = 0$。
- 递归公式与完全背包问题一模一样，只是 k 多了一个限制条件 $0 \le k \le m_i$。

多重背包问题例

- $n = 3, W = (3, 4, 5), V = (2, 3, 4), M = (4, 3, 2), C = 16$。

i \ c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	4	4	4	6	6	6	8	8	8	8	8
2	0	0	0	2	3	3	4	5	6	6	7	8	9	9	10	11	11
3	0	0	0	2	3	4	4	5	6	7	8	8	9	10	11	12	12

选2个物品2

选1个物品2, 1个物品3

- 在多重背包中，物品有数量限制，也可以把第 i 种物品转化为 m_i 件体积和价值相同的物品，然后再来求解。

多重背包问题递归算法

```
1.  int result = 0;
2.  if (i == 0 || c == 0){ // 初始条件
3.     result = 0; }
4.  else if(w[i] > c){ // 装不下该珠宝
5.     result = ks2(i-1, c); }
6.  else { // 可以装下，取k个物品，取其中使得价值最大的
7.     for (int k = 0; k ≤ m_i && k · w_i ≤ c; k++){
8.        int tmp2 = ks2(i-1, c - w[i] · k) + v[i] · k;
9.        if (tmp2 > result){
10.          result = tmp2;  } } }
11. results[i][c] = result;
12. return result;
```

多重背包问题DP算法

```
for (int i = 0; i < n; i++){
    // 考虑第i个物品
    // 分两种情况：1. m_i · w_i ≥ C，可以当做完全背包问题来处理
    if (m_i · w_i ≥ C) {
      for (int j = w_i; j ≤ c ; j++) {
        f [j] = max(f[j], f[j - w_i] + v_i); } }
    // 2. m_i · w_i < C, 需要在 f[j-w_i · k] + v_i · k中找到最大值, 0 ≤ k ≤ m_i
    else {
       for (int j = w_i; j ≤ c ; j++) {
          int k = 1;
          while (k ≤ m_i && j ≥ w_i · k ){
             f[j] = max(f[j], f[j - w_i · k] + v_i · k);
             k++; } } }
}
```

矩阵乘法链

- 问题:对矩阵乘法链 $M_1 \times \cdots \times M_q$，$M_i$ 的维数为 $r_i \times r_{i+1}$ $(1 \le i \le n)$，求优化的乘法顺序，使得计算该乘法链所用的乘法数最少。
- 长度为 q 的矩阵乘法链有指数量级 $\Omega(2^q)$ 的可能的乘法顺序。
- 矩阵 $A_{m \times n}$ 与 矩阵 $B_{n \times p}$ 相乘需要做 mnp 个元素乘法。
- 计算矩阵 A、B 和 C 的乘积有两种方式：(A × B) × C 和 A × (B × C)。
- 假定 $A_{100 \times 1}$、$B_{1 \times 100}$、C为100×1矩阵,
  - (A × B) × C 需乘法数为 100×1×100＋100×100×1 ＝ 20000;
  - 而 A × (B × C) 所需乘法数为 1×100×1＋100×1×1＝ 200。

优化原理

- **最优性原理**：用 $M(i, j)$ 表示链 $M_i \times \cdots \times M_j$ $(i \le j)$ 的乘积。假设优化的矩阵链乘法顺序最后计算乘积 $M(i, k) \times M(k+1, j)$，则计算 $M(i, j)$ 的优化乘法顺序在计算子链 $M(i, k)$ 和 $M(k+1, j)$ 时也是优化的。(反证可得)
- 考虑 5 个矩阵的乘法链，其行列数为 $r = (10, 5, 1, 10, 2, 10)$，即 M_1 为 10×5 的矩阵，等等。
- 优化的乘法顺序为 $(M_1 \times M_2) \times ((M_3 \times M_4) \times M_5))$。
- $(M_3 \times M_4) \times M_5$ 对子链 $M(3, 5) = M_3 \times M_4 \times M_5$ 也是优化的。

递归式

- 设 c(i, j) 为计算 M(i, j) 所需乘法次数的最小值(优化值), 根据优化原理,优化值之间满足:

$$c(i,j)=\begin{cases}0 & \text{if } j=i\\ r_i r_{i+1} r_{i+2} & \text{if } j=i+1\\ \min_{i\le k<j}\{c(i,k)+c(k+1,j)+r_i r_{k+1} r_{j+1}\} & \text{if } j>i+1\end{cases}$$

- 令 kay(i, j) 为达到最小值的 k，即断开位置。可回溯找到优化的乘法顺序。
- K 只有 j-i 种可能的取值，即 $k=i, i+1, \cdots, j-1$。我们只需检查所有可能情况，找到最优者即可。
- 保存计算的每个 c(i, j) 和 kay(i, j) 值，可避免大量重复计算。但需 $O(q^2)$ 的存储空间，q 为乘法链的长度。

矩阵乘法链递归方法时间复杂度

- $c(i,j)=\min_{i\le k<j}\{c(i,k)+c(k+1,j)+r_i r_{k+1} r_{j+1}\}$
- 计算 $c(i,j)$ 的时间代价由 i 与 j 之间的矩阵个数决定。
- 记 $s=j-i+1$，有

$$
\begin{aligned}
t(s) &= \sum_{1\le k<j} t(k-i+1) + \sum_{i\le k<j} t(j-k) + \Theta(s) \\
&= 2\sum_{1\le k<s} t(k) + \Theta(s) \\
&> 2t(s-1) + \Theta(s) \\
&= \Omega(2^s)
\end{aligned}
$$

- 时间复杂度很高的原因在于对 $c(i,j)$ 的重复计算。
- 采用迭代方法可以将复杂度降到 $O(q^3)$。

矩阵乘法链DP算法例

q = 5， r = (10, 5, 1, 10, 2, 10)

$c_{11} = c_{22} = c_{33} = c_{44} = c_{55} = 0$

$c_{12} = r_1r_2r_3 = 50$, $c_{23} = r_2r_3r_4 = 50$, $c_{34} = r_3r_4r_5 = 20$, $c_{45} = 200$.

$$c(i, j)=\min_{i\le k<j}\{c(i, k)+c(k+1, j)+r_i r_{k+1} r_{j+1}\}$$

$c_{13} = \min\{c_{11}+c_{23}+500, c_{12}+c_{33}+100\} = 150$;

$c_{24} = \min\{c_{22}+c_{34}+10, c_{23}+c_{44}+100\} = 30$;

$c_{35} = \min\{c_{33}+c_{45}+100, c_{34}+c_{55}+20\} = 40$;

$c_{14} = \min\{c_{11}+c_{24}+100, c_{12}+c_{34}+20, c_{13}+c_{44}+200\} = 90$;

$c_{25} = \min\{c_{22}+c_{35}+50, c_{23}+c_{45}+500, c_{24}+c_{55}+100\} = 90$;

$c_{15} = \min\{c_{11}+c_{25}+500, c_{12}+c_{35}+100,$

$c_{13}+c_{45}+1000, c_{14}+c_{55}+200\} = 190$.

- Traceback: 最优解$(M_1*M_2)*((M_3*M_4)*M_5)$.

矩阵乘法链DP算法例

q=5 和 r=(10, 5, 1, 10, 2, 10),

c_{12}=50, c_{23}=50,

c_{34}=20, c_{45}=200.

c_{13}=c_{12}+c_{33}+100=150;

c_{24}=c_{22}+c_{34}+10=30;

c_{35}=c_{34}+c_{55}+20=40;

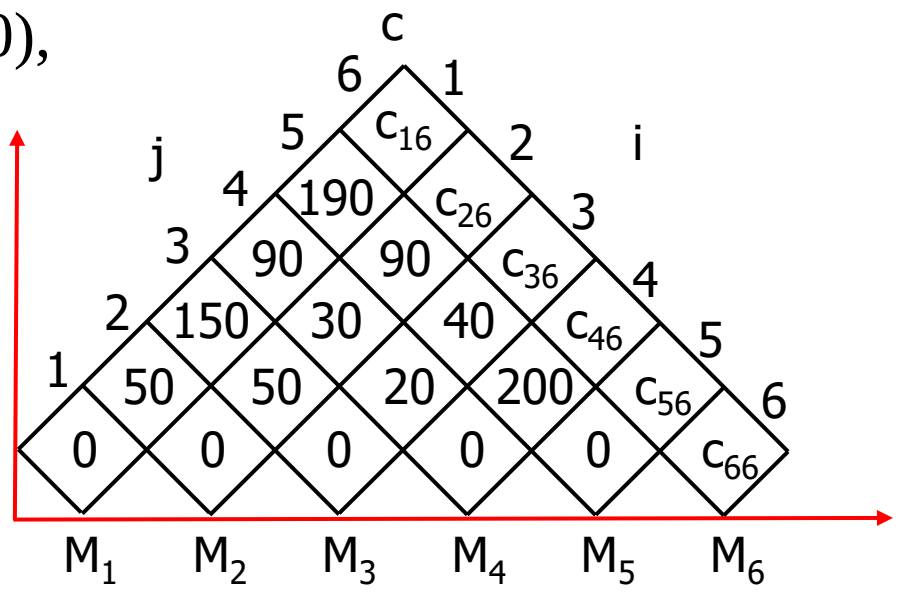
c_{14}=c_{12}+c_{34}+20=90;

c_{25}=c_{22}+c_{35}+50=90;

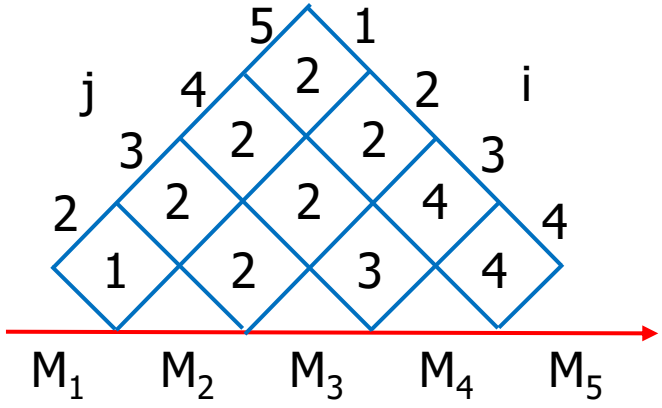
c_{15}=c_{12}+c_{35}+100=190.

回溯可找到优化的乘法顺序.

需 $O(q^2)$ 的存储空间保存计算的每个 c(i, j) 和 kay(i, j) 值.



kay: 达到最小值的 k, 即断开位置



矩阵乘法链迭代方法时间复杂度

- $c(i, j) = \min_{i \le k < j}\{c(i, k)+c(k+1, j)+r_i r_{k+1} r_{j+1}\}$
- 在迭代过程中，每个 $c(i, j)$ 只计算一次。
- $s = j - i = 1$ 时，计算每个 $c(i, j)$ 只需要 $\Theta(1)$ 的时间。这样的 $c(i, j)$ 共有 q 个。
- $s = j - i > 1$ 时，需要做 $\Theta(s)$ 次加法和比较。这样的 $c(i, j)$ 共有 $q-s+1$ 个。
- 总的计算量为

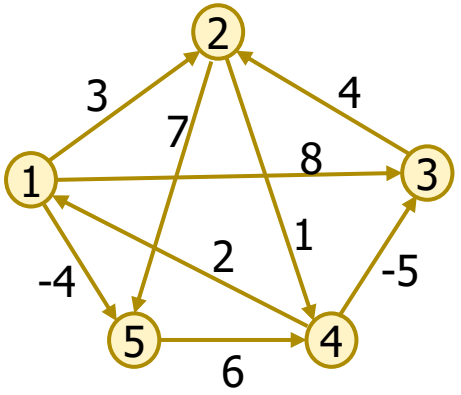
$$T(q) = \sum_{0 \le s < q-1} [s \cdot (q-s+1)] = O(q^3)$$

矩阵乘法链DP算法

```
void MatrixChain(int r[], int q, int **c, int **kay)
{for (int i=1; i<q; i++){
        c[i][i] = 0;
        c[i][i+1] = r[i]*r[i+1]*r[i+2];
        kay[i][i+1] = i;
}
    c[q][q] = 0;
    for (int s = 2; s< q; s++)
    for (int i = 1; i <= q – s; i++)
        {//k = i 时的最小项
            c[i][i+s]=c[i][i]+c[i+1][i+s]+r[i]*r[i+1]*r[i+s+1];
            kay[i][i+s] = i;
    for (int k=i+1; k<i+s; k++){
        int t=c[i][k] + c[k+1][i+s]+r[i]*r[k+1]*r[i+s+1];
        if (t < c[i][i+s]){//更小的最小项
            c[i][i+s] = t;
            kay[i][i+s] = k; }
        }
    }
}
```

All-Pair 最短路问题

- 现有一张城市地图，图中的顶点为城市，有向边代表两个城市间的连通关系，边上的权即为距离。



- 问题：为每一对可达的城市间设计一条公共汽车线路，要求线路的长度在所有可能的方案里是最短的。
- 对于每对顶点 (i, j)，定义从 i 到 j 的所有路径中，具有最小长度的路径为从 i 到 j 的最短路。
- 假定图上无负成本的环路，这时只需考虑简单路径：加上环路只会增加路径成本。

All-Pair 最短路问题

- 从城市 1 到 4 的路径有:

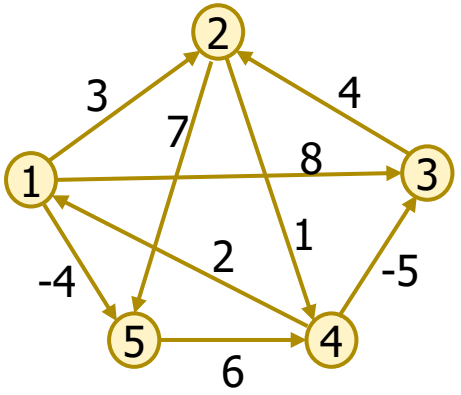
  p_1: 1, 2, 4 (4)

  p_2: 1, 2, 5, 4 (16)

  p_3: 1, 3, 2, 4 (13)

  p_4: 1, 3, 2, 5, 4 (25)

  p_5: 1, 5, 4 (2)



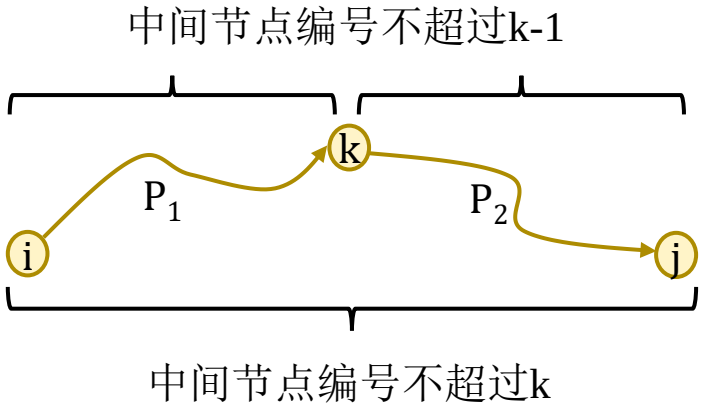
- 括号内的数字为路径长度，路径 p_5 最短。
- 用 $c_{ij}(k)$ 表示 节点 i 到 j 的中间节点编号不超过 k 的最短路长度，则

  $c_{14}(0) = \infty$，$c_{14}(1) = \infty$，$c_{14}(2) = 4$，

  $c_{14}(3) = 4$，$c_{14}(4) = 4$，$c_{14}(5) = 2$

All-Pair 最短路问题

- 我们发现以下事实：
  - $c_{ii}(k) = 0$ for all k；
  - $c_{ij}(0) = w_{ij}$ 或 ∞；
  - $c_{ik}(k) = c_{ik}(k-1)$；
  - $c_{kj}(k) = c_{kj}(k-1)$；
  - 如果最短路径不包含节点 k，则 $c_{ij}(k) = c_{ij}(k-1)$；
    否则，$c_{ij}(k) = c_{ik}(k-1) + c_{kj}(k-1)$。
  - $c_{ij}(n)$ 是在原来的图上 i 到 j 的最短路长度。



- 所以，我们建立了 $c_{ij}(k)$ 和 $c_{ij}(k-1)$ 之间的递归关系：

$$c_{ij}(k)=\min\{c_{ij}(k-1), c_{ik}(k-1)+c_{kj}(k-1)\}$$

- 利用迭代方法可以避免重复计算，降低复杂度。

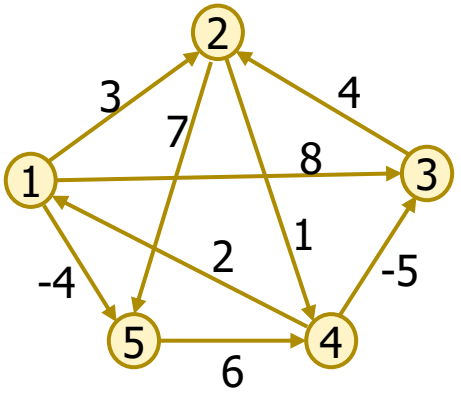
Floyd-Warshall算法

1. n=|V|
2. C(0)=W
3. for k=1 to n
4. let C(k)=[$c_{ij}(k)$] be a new matrix
5. for i=1 to n
6. for j=1 to n
7. $c_{ij}(k)=\min\{c_{ij}(k-1), c_{ik}(k-1)+c_{kj}(k-1)\}$
8. return C(n)

- 算法只需要使用一个矩阵，每次迭代更新。
- 因为 $c_{ii}(k) = 0 \ \forall\ k$, 所以矩阵 C(k) 的对角线元素为 0。
- C(k)的第k行、第k列上的元素不变。
- C(k)的非k行k列上的元素按递归式更新。
- 时间复杂度为$O(n^3)$.

Floyd-Warshall算法

$$C(0) = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} = W$$



$$C(1) = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$C(2) = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$C(3) = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Floyd-Warshall算法

$$C(4) = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$C(5) = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

- 矩阵C(5)中的元素 $c_{ij}(5)$ 就是节点 i 到 节点 j 的最短路长度。
- 怎么样找到节点 i 到 节点 j 之间的最短路？

构建一条最短路径

- 在 Floyd-Warshall 算法中，可以在计算矩阵 C(k) 的同时计算前驱矩阵 $\Pi(k) = [\pi_{ij}(k)]$。
- 令 $p_{ij}(k)$ 为节点 i 和 j 之间的中间节点编号不超过 k 的最短路径，$\pi_{ij}(k)$ 为路径 $p_{ij}(k)$ 上 j 的前驱节点。
- 如果 $0 < w_{ij} < \infty$，$\pi_{ij}(0) = i$，否则，$\pi_{ij}(0) = 0$。
- $c_{ij}(k) = \min(c_{ij}(k-1), c_{ik}(k-1) + c_{kj}(k-1))$。
  - 如果节点 k 不是路径 $p_{ij}(k)$ 上的中间节点，即 $c_{ij}(k) = c_{ij}(k-1)$，则 $\pi_{ij}(k) = \pi_{ij}(k-1)$;
  - 如果节点 k 是路径 $p_{ij}(k)$ 上的中间节点，即 $c_{ij}(k) < c_{ij}(k-1)$，则 $\pi_{ij}(k) = \pi_{kj}(k-1)$。
- 在矩阵 C(k) 中元素 $c_{ij}(k)$ 被更新的同时矩阵 $\Pi(k)$ 的元素 $\pi_{ij}(k)$ 也被更新。

Floyd-Warshall算法回溯

$$C(0)=\begin{pmatrix}0 & 3 & 8 & \infty & -4\\ \infty & 0 & \infty & 1 & 7\\ \infty & 4 & 0 & \infty & \infty\\ 2 & \infty & -5 & 0 & \infty\\ \infty & \infty & \infty & 6 & 0\end{pmatrix}\qquad \Pi(0)=\begin{pmatrix}0 & 1 & 1 & 0 & 1\\ 0 & 0 & 0 & 2 & 2\\ 0 & 3 & 0 & 0 & 0\\ 4 & 0 & 4 & 0 & 0\\ 0 & 0 & 0 & 5 & 0\end{pmatrix}$$

$$C(1)=\begin{pmatrix}0 & 3 & 8 & \infty & -4\\ \infty & 0 & \infty & 1 & 7\\ \infty & 4 & 0 & \infty & \infty\\ 2 & 5 & -5 & 0 & -2\\ \infty & \infty & \infty & 6 & 0\end{pmatrix}\qquad \Pi(1)=\begin{pmatrix}0 & 1 & 1 & 0 & 1\\ 0 & 0 & 0 & 2 & 2\\ 0 & 3 & 0 & 0 & 0\\ 4 & 1 & 4 & 0 & 1\\ 0 & 0 & 0 & 5 & 0\end{pmatrix}$$

$$C(2)=\begin{pmatrix}0 & 3 & 8 & 4 & -4\\ \infty & 0 & \infty & 1 & 7\\ \infty & 4 & 0 & 5 & 11\\ 2 & 5 & -5 & 0 & -2\\ \infty & \infty & \infty & 6 & 0\end{pmatrix}\qquad \Pi(2)=\begin{pmatrix}0 & 1 & 1 & 2 & 1\\ 0 & 0 & 0 & 2 & 2\\ 0 & 3 & 0 & 2 & 2\\ 4 & 1 & 4 & 0 & 1\\ 0 & 0 & 0 & 5 & 0\end{pmatrix}$$

Floyd-Warshall算法回溯

$$C(3)=\begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi(3)=\begin{pmatrix} 0 & 1 & 1 & 2 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 3 & 0 & 2 & 2 \\ 4 & 3 & 4 & 0 & 1 \\ 0 & 0 & 0 & 5 & 0 \end{pmatrix}$$

$$C(4)=\begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi(4)=\begin{pmatrix} 0 & 1 & 4 & 2 & 1 \\ 4 & 0 & 4 & 2 & 1 \\ 4 & 3 & 0 & 2 & 1 \\ 4 & 3 & 4 & 0 & 1 \\ 4 & 3 & 4 & 5 & 0 \end{pmatrix}$$

$$C(5)=\begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi(5)=\begin{pmatrix} 0 & 3 & 4 & 5 & 1 \\ 4 & 0 & 4 & 2 & 1 \\ 4 & 3 & 0 & 2 & 1 \\ 4 & 3 & 4 & 0 & 1 \\ 4 & 3 & 4 & 5 & 0 \end{pmatrix}$$

Robert W．Floyd(1936-2001)

- 罗伯特·弗洛伊德是美国计算机科学家，堆排序算法和Floyd-Warshall 算法的创始人之一。
- 1953年于芝加哥大学获得文学士学位。因经济不景气找工作比较困难，到西屋电气公司当计算机房夜班值班。
- 一段时间后，弗洛伊德对计算机产生了兴趣，决定弄懂它！于是开始自学，并回母校旁听相关课程，于1958年获得理科学士学位。
- 1962年到Computer Associates公司当分析员。
- 1962年与Warshall合作发布了Floyd-Warshall算法。
- 1965年应聘成为卡内基-梅隆大学的副教授。
- 1968年为斯坦福大学副教授，1970年被聘为教授。
- 1978年因为提出归纳断言法等贡献获得图灵奖！

Stephen Warshall(1935-2006)

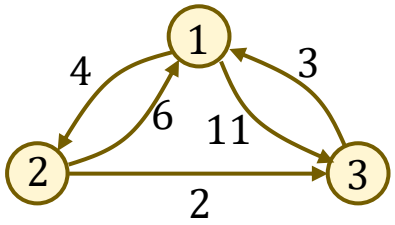
- 1956年于哈佛大学获得数学学士学位。
- 因为没有找到喜欢的专业，他没有继续读学位，只是在多所大学旁听感兴趣的研究生课程。主要是计算机科学与软件工程方面。
- 毕业后一直从事软件方面的工作。
- 1961年创办了Computer Associates公司。这个公司后来与其他公司合并为Applied Data Research (ADR) 的一部分。华沙一直担任理事到退休。
- 1962年聘请了弗洛伊德并与他合作发布了Floyd-Warshall算法。
- 学术活动：1971-1972年在一所法语学校教软件工程。

最短路径练习题

已知 $W=\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$，试画出对应的城市地图，并找出每对节点间的最短路径。

最短路径练习题答案

$$C(0)=\begin{pmatrix}0 & 4 & 11\\ 6 & 0 & 2\\ 3 & \infty & 0\end{pmatrix}\qquad \Pi(0)=\begin{pmatrix}0 & 1 & 1\\ 2 & 0 & 2\\ 3 & 0 & 0\end{pmatrix}$$



$$C(1)=\begin{pmatrix}0 & 4 & 11\\ 6 & 0 & 2\\ 3 & 7 & 0\end{pmatrix}\qquad \Pi(1)=\begin{pmatrix}0 & 1 & 1\\ 2 & 0 & 2\\ 3 & 1 & 0\end{pmatrix}$$

$$C(2)=\begin{pmatrix}0 & 4 & 6\\ 6 & 0 & 2\\ 3 & 7 & 0\end{pmatrix}\qquad \Pi(2)=\begin{pmatrix}0 & 1 & 2\\ 2 & 0 & 2\\ 3 & 1 & 0\end{pmatrix}$$

$$C(3)=\begin{pmatrix}0 & 4 & 6\\ 5 & 0 & 2\\ 3 & 7 & 0\end{pmatrix}\qquad \Pi(3)=\begin{pmatrix}0 & 1 & 2\\ 3 & 0 & 2\\ 3 & 1 & 0\end{pmatrix}$$

旅行商问题(TSP)

- 旅行商问题(Travelling salesman problem, TSP)：又称旅行推销员问题, 货郎担问题, 邮递员问题。给定一系列城市和每对城市之间的距离, 求访问每一座城市一次并回到起始城市的最短回路。
- TSP的研究历史很久，最早的描述是1759年欧拉研究的骑士环游问题，即对于国际象棋棋盘中的64个方格，走访64个方格一次且仅一次，并且最终返回到起始点。
- 问题的可行解是所有顶点的全排列。它是NP完全问题, 在交通运输、电路板线路设计以及物流配送等领域内有着广泛的应用。
- 1954年，乔治·丹捷格 (线性规划之父) 等人用线性规划解决了美国49个城市的巡回问题，取得了旅行商问题的历史性突破。

TSP问题用DP方法求解的可行性

- 问题：求图 $G=(V,E)$ 的最小成本周游路线。
- 设 $s, s_1, \cdots, s_p, s$ 是从 s 出发的一条路径长度最短的简单回路。
- 假设从 s 到下一个城市 s_1 的最短路径已经求出，则问题转化为求从 s_1 到 s 的最短路径。
- 显然 $s_1, s_2, \cdots, s_p, s$ 一定构成一条从 s_1 到 s 的最短路径。
- TSP问题满足最优性原理，可以用动态规划来求解。

TSP问题的DP求解

- 求图 $G = (V, E)$ 的最小成本周游路线。
- 节点 s 为出发点，S 为 V 的不含节点 s 的子集。
- $p(i, S)$ 为从节点 s_i 出发，经过 S 中的所有节点各一次，到达节点 s 的最短路径。
- $\delta(i, S)$ 为路径 $p(i, S)$ 的长度。
- 根据优化原理，有以下递归式

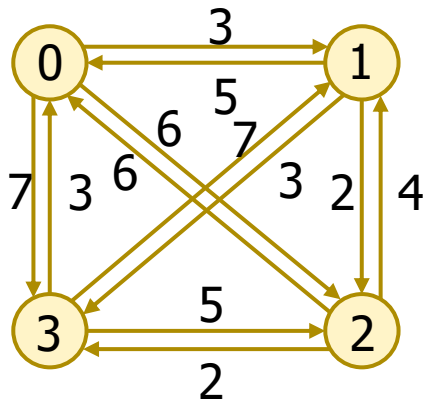
$$\delta(i, S) = \min_{j \in S}\{w_{ij} + \delta(j, S-\{s_j\})\}$$

- 初始: $\delta(i, \emptyset) = w_{i1}$，即从 s_i 不经过任何中间点就回到s。
- 从 $S = \emptyset$ 开始，依次对 $|S| = 1, 2, \cdots, n-1$ 计算。
- 原问题为 $\delta(s, V-\{s\})$。

TSP问题例

- 假设有四个城市, 0, 1, 2, 3, 它们之间的代价如下图.
- 也可以表示成二维表的形式：

$$C = \begin{pmatrix} 0 & 3 & 6 & 7 \\ 5 & 0 & 2 & 3 \\ 6 & 4 & 0 & 2 \\ 3 & 7 & 5 & 0 \end{pmatrix}$$

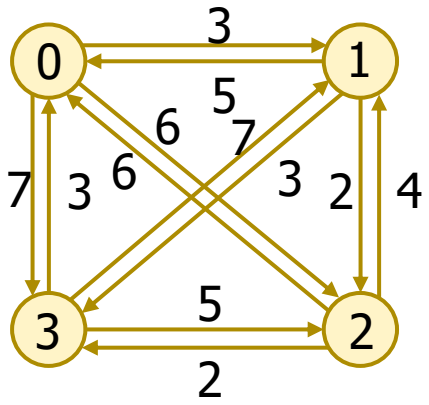


- 问题：从城市0出发，最后回到0。期间1, 2, 3都必须且只能经过一次，使代价最小。

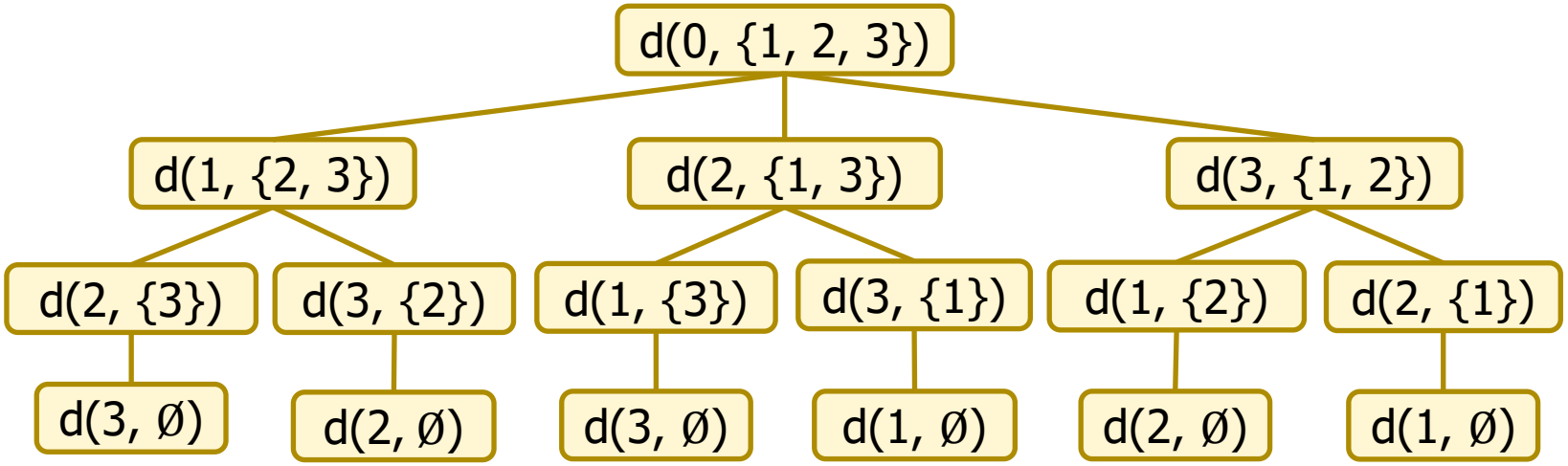
TSP问题的DP求解过程

- 假设出发城市是城市0.

$$\begin{pmatrix} 0 & 3 & 6 & 7 \\ 5 & 0 & 2 & 3 \\ 6 & 4 & 0 & 2 \\ 3 & 7 & 5 & 0 \end{pmatrix}$$



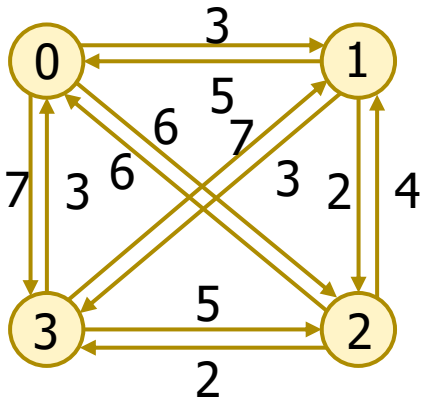
- 构造解空间树：



TSP问题的DP求解过程

- 假设出发城市是城市0.

$$\begin{pmatrix} 0 & 3 & 6 & 7 \\ 5 & 0 & 2 & 3 \\ 6 & 4 & 0 & 2 \\ 3 & 7 & 5 & 0 \end{pmatrix}$$



- DP表：$\delta(i, S) = \min_{j \in S}\{w_{ij} + \delta(j, S-\{s_j\})\}$

	Ø	{1}	{2}	{3}	{1, 2}	{1, 3}	{2, 3}	{1, 2, 3}
0								10
1	5		8	6			7	
2	6	9		5		10		
3	3	12	11		14			

$w_{21}+d(1, Ø) = 4+5=9$

$w_{31}+d(1, Ø) = 7+5=12$

$\min\{w_{31}+d(1, \{2\}), w_{32}+d(2, \{1\})\}=14$

DP法求解TSP问题

1. {
2. for (i=1; i<n; i++) d[i][0]=w[i][0];
3. for (j=1; j<2^{n-1}-1; j++)
4. for (i=1; i<n; i++)
5. {
6. if (I ∉ S_i)
7. {
8. for each k∈S_i, d[i][S_i]=min{c[i][k]+d[k][{S_i-k}]|k∈S_i};
9. }
10. }
11. for each k∈V[2^{n-1}-1], d[0][2^{n-1}-1]= min{c[0][k]+d[k][2^{n-1}-2]};
12. return: d[0][2^{n-1}-1];
13. }

DP法求解TSP问题的时间复杂度

- DP法求解TSP问题的递归式：

$$\delta(i, S) = \min_{j \in S}\{w_{ij} + \delta(j, S-\{s_j\})\}$$

- |S|=k的子问题个数为C(n-2,k).
- 在|S|=k时，求最小值需要做k-1次比较。
- 时间复杂度是$T(n)=\sum_{1\le k\le n-2}(k-1)C(n-2,k)=O(n2^n)$.

DP法求解TSP问题例

- 考虑有邻接矩阵如下:

$$\begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix}$$

- 计算过程如下:

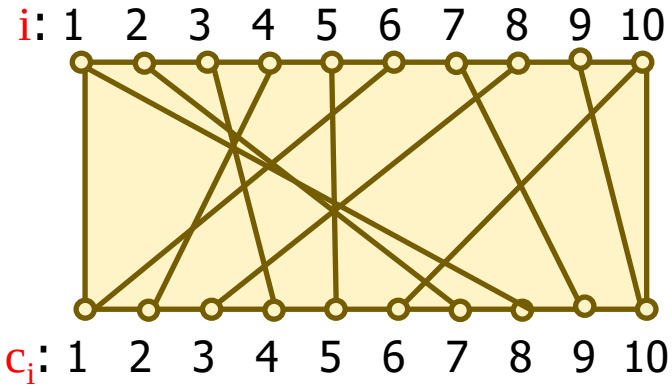
$$\delta(i, \varnothing)=w_{i1};\quad \delta(i,\{j\})=w_{ij}+w_{j1};$$

$$\delta(i, S) = \min_{j\in S}\{w_{ij}+ \delta(j, S-\{s_j\})\}$$

	Ø	{2}	{3}	{4}	{2, 3}	{2, 4}	{3, 4}	{2, 3, 4}
1								35
2	5		15	18			25	
3	6	18		20		25		
4	8	13	15		24			

不交叉网的子集

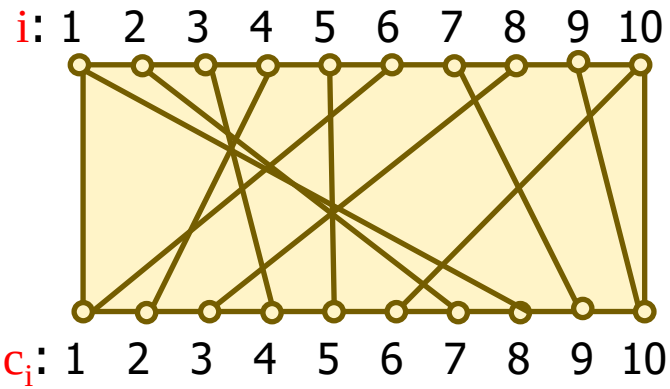
- 每个 i 有唯一一个子网 (i, c_i)。
- 满足 $i \le 5$ 且 $j \le 7$ 的不交叉网的子集有：
  - 满足上述条件的单个网构成的子集。
  - 子集 {(4, 2), (5, 5)} 和 {(3, 4), (5, 5)}。
- 最大子集不唯一，{(4, 2), (5, 5)} 是一个最大子集，{(3, 4), (5, 5)} 也是一个最大子集。



不交叉网的子集的递归关系

- 记 m_{ij} 为所有满足 $u \le i$ 且 $c_u \le j$ 的最大无交叉子集。
- s_{ij} 为 m_{ij} 的大小。
- 根据优化解中是否包含网 (i, c_i)，我们有以下递归关系:

$$s_{ij} = \begin{cases} 0 & \text{if } j = 0 \\ 0 & \text{if } i = 1, j < c_1 \\ 1 & \text{if } i = 1, j \ge c_1 \\ s_{i-1,j} & \text{if } i > 1, j < c_i \\ \max\{s_{i-1,j}, s_{i-1,c_i-1}+1\} & \text{if } i > 1, j \ge c_i \end{cases}$$

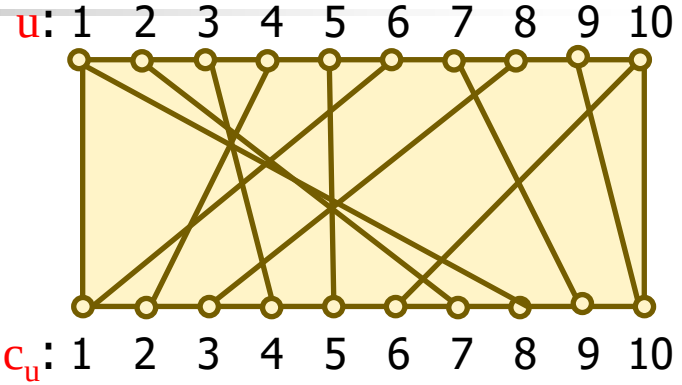


不交叉网的子集的DP表

$s_{ij} = \max\{s_{i-1,j}, s_{i-1,ci-1}+1\}$

if $i>1, j \geq c_i$

- $s_{1j}=0, j<8;$
- $s_{1j}=1, j \geq 8;$
- $s_{2j} = s_{1j}=0, j<7;$
- $s_{2j}=\max\{s_{1j}, s_{16}+1\}=1,$ $j \geq 7;$

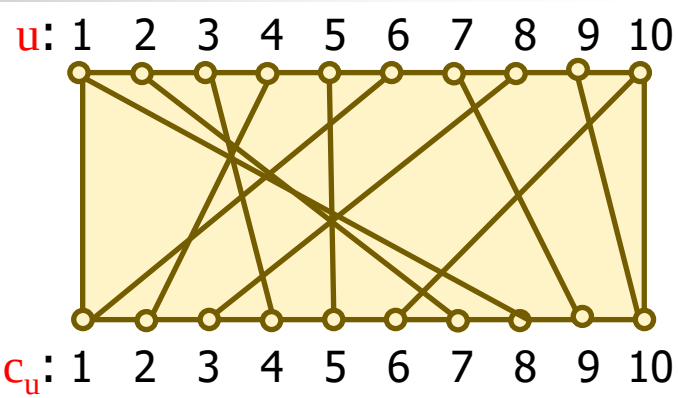


	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	1	1	1
3	0	0	0	1	1	1	1	1	1	1
4	0	1	1	1	1	1	1	1	1	1
5	0	1	1	1	2	2	2	2	2	2
6	1	1	1	1	2	2	2	2	2	2
7	1	1	1	1	2	2	2	2	3	3
8	1	1	2	2	2	2	2	2	3	3
9	1	1	2	2	2	2	2	2	3	4
10	1	1	2	2	2	3	3	3	3	4

不交叉网的子集的回溯

- 回溯：如果，$s_{ij} \neq s_{i-1,j}$，则 m_{ij} 包含(i, c_i)。

1. $j = 10, s_{9,10} \neq s_{8,10}$，$m_{9,10}$ 包含 $(9, 10)$;
2. $j = 10\text{-}1 = 9, s_{7,9} \neq s_{6,9}$，$m_{7,9}$ 包含 $(7, 9)$;
3. $j = 9\text{-}1 = 8. s_{5,8} \neq s_{4,8}$，$m_{5,8}$ 包含 $(5, 5)$;
4. $j = 5\text{-}1 = 4. s_{3,4} \neq s_{2,4}$，$m_{3,4}$包含$(3,4)$.



	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	1	1	1
3	0	0	0	1	1	1	1	1	1	1
4	0	1	1	1	1	1	1	1	1	1
5	0	1	1	1	2	2	2	2	2	2
6	1	1	1	1	2	2	2	2	2	2
7	1	1	1	1	2	2	2	2	3	3
8	1	1	2	2	2	2	2	2	3	3
9	1	1	2	2	2	2	2	2	3	4
10	1	1	2	2	2	3	3	3	3	4

最长公共子序列

- 观察序列 $X=\{A, B, C, B, D, A, B\}$ 和 $Z=\{B, C, D, B\}$，发现序列 Z 的元素是 X 序列的第 2, 3, 5, 7 元素。
- 给定两个序列 $X=\{x_1, x_2, \cdots, x_m\}$ 和 $Z=\{z_1, z_2, \cdots, z_k\}$。
- 如果存在一个严格递增下标序列 $\{i_1, i_2, \cdots, i_k\}$，使得对于所有 $j=1, 2, \cdots, k$ 有 $z_j=x_{i_j}$，则称序列 Z 是 X 的**子序列**.
- 给定 2 个序列 X 和 Y，当另一序列 Z 既是 X 的子序列又是 Y 的子序列时，称 Z 是序列 X 和 Y 的**公共子序列**。
- **问题**：给定 2 个序列 $X=\{x_1, x_2, \cdots, x_m\}$ 和 $Y=\{y_1, y_2, \cdots, y_n\}$，找出 X 和 Y 的最长公共子序列。

最长公共子序列的最优性

- 设序列 $X=\{x_1, x_2, \cdots, x_m\}$ 和 $Y=\{y_1, y_2, \cdots, y_n\}$ 的最长公共子序列为 $Z=\{z_1, z_2, \cdots, z_k\}$，则

  (1) 记 $X_i=\{x_1, x_2, \cdots, x_i\}$；$Y_j=\{y_1, y_2, \cdots, y_j\}$。

  (2) 若 $x_m=y_n$，则 $z_k=x_m=y_n$，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的最长公共子序列。

  (3) 若 $x_m \neq y_n$，有

  - 如果 $z_k \neq x_m$，则 Z 是 X_{m-1} 和 Y 的最长公共子序列。
  - 如果 $z_k \neq y_n$，则 Z 是 X 和 Y_{n-1} 的最长公共子序列。

- 由此可见，2 个序列的最长公共子序列包含了这 2 个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有**最优子结构性质**。

最长公共子序列问题的递归关系

- 用 c_{ij} 记录序列 X_i 和 Y_j 的最长公共子序列的长度。
- 当 $i=0$ 或 $j=0$ 时，X_i 和 Y_j 的最长公共子序列是 $\emptyset$，此时 $c_{ij}=0$。
- 当 $i,j>0$ 时
  - 如果 $x_i=y_j$，有 $c_{ij}=c_{i-1,j-1}+1$。
  - 如果 $x_i\neq y_j$，有 $c_{ij}=\max\{c_{i,j-1},c_{i-1,j}\}$。
- 由最长公共子序列问题的最优子结构性质可建立递归关系如下：

$$c_{ij}=\begin{cases}0 & i=0,\ j=0\\ c_{i-1,j-1}+1 & i,j>0;\ x_i=y_j\\ \max\{c_{i,j-1},c_{i-1,j}\} & i,j>0;\ x_i\neq y_j\end{cases}$$

最长公共子序列算法

```
void LCSLength(int m，int n，char *x，
char *y，int **c，int **b)
{
      int i，j;
      for (i = 1; i <= m; i++) c[i][0] = 0;
      for (i = 1; i <= n; i++) c[0][i] = 0;
      for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++) {
          if (x[i]==y[j]) {
              c[i][j]=c[i-1][j-1]+1; b[i][j]=1;}
          else if (c[i-1][j]>=c[i][j-1]) {
              c[i][j]=c[i-1][j]; b[i][j]=2;}
          else { c[i][j]=c[i][j-1]; b[i][j]=3; }
          }
}
```

构造最长公共子序列

```
void LCS(int i，int j，char *x，int **b)
{
    if (i ==0 || j==0) return;
    if (b[i][j]== 1)
    {
          LCS(i-1，j-1，x，b);
          cout<<x[i];
    }
    else if (b[i][j]== 2) LCS(i-1，j，x，b);
    else LCS(i，j-1，x，b);
}
```

由于在所考虑的子问题空间中，总共有θ(mn)个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

算法的改进

- 在算法 **lcsLength** 和 **lcs** 中，可进一步将数组 b 省去。
- 事实上，数组元素 c_{ij} 的值仅由 $c_{i-1,j-1}$，$c_{i-1,j}$ 和 $c_{i,j-1}$ 这 3 个数组元素的值所确定。
- 对于给定的数组元素 c_{ij}，可以不借助于数组 b 而仅借助于 c 本身在线性时间内确定 c_{ij} 的值是由 $c_{i-1,j-1}$，$c_{i-1,j}$ 和 $c_{i,j-1}$ 中哪一个值所确定的。

- 如果只需要计算最长公共子序列的长度，则算法的空间需求可大大减少。事实上，在计算c[i][j]时，只用到数组c的第i行和第i-1行。因此，用2行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至O(min(m,n))。

复习要求

- 根据优化原理列递归式
- 设计实现递归式的迭代算法(列表)
- 0/1背包问题

  矩阵乘法链

  多段图

  求各对点之间的最短路
- 要求会做实例；分析算法的复杂度

习题

1. 设 c(i) 为多段图上节点 1 到目的节点的最短路长度，试列出动态规划的递归式。并就课堂上的例子给出求解过程。

2. 0/1背包问题: $n = 4, c = 20, w = (10, 15, 6, 9), p = (2, 5, 8, 1)$。

   (1) 产生元组集合 P(1), P(2), P(3) 和该背包实例的解。

   (2) 证明当重量和效益值均为整数时动态规划算法的时间复杂度为 $O(\min\{2^n, n\Sigma_{1\le i\le n}p_i, nc\})$

   提示: $|P(i)| \le 1 + \Sigma_{i\le j\le n}p_j$

习题

3. 子集和数问题: 设 $S = \{s_1, s_2, ..., s_n\}$ 为 n 个正数的集合，试找出和数不超过 M 且最大的 S 的子集，该问题是 NP-难度问题，试用动态规划法设计一算法。
4. 设一个矩阵乘法链的行列数为 $r = (10, 20, 50, 1, 100)$，用动态规划算法给出优化的乘法顺序和优化的乘法数。
5. 补充例题 15、17 的计算过程。
6. 本章习题19。