

课程编号：20006026

算法分析与设计

主讲教师：刘 瑶

电子科技大学信息与软件工程学院

第4章：贪心算法

(Greedy Algorithm)

知识要点

∞ 理解贪心算法的概念和基本要素

- ⊕ 最优子结构性性质和贪心选择性性质
- ⊕ 理解贪心算法与动态规划算法的差异

∞ 贪心设计策略的典型例子

- ⊕ 活动安排问题
- ⊕ 最优装载问题
- ⊕ 单源最短路径
- ⊕ 多机调度问题

贪心算法的例子

☞ 找零钱问题

- ⊕ 假设有4种硬币，面值分别为：二角五分、一角、五分和一分
- ⊕ 现在要找给顾客六角三分钱，如何找使得给出的硬币个数最少？

☞ 问题的求解

- ⊕ 正解：选择2个两角五分的硬币、1个一角的硬币、3个一分的硬币。和其它找法相比，所拿出的硬币个数最少。
- ⊕ 求解过程
 - 首先选出1个面值不超过六角三分的最大硬币，即两角五分
 - 然后从六角三分中减去两角五分，剩下三角八分
 - 再选出1个面值不超过三角八分的最大硬币
 - 即又一个两角五分。如此一直做下去.....
 - 这里用到的方法就是贪心算法

贪心算法的例子

☞ 找零钱问题

- ⊕ 在这个例子中，找硬币算法得到的结果是整体最优解
- ⊕ 问题本身具有最优子结构性质，可以用动态规划算法求解
- ⊕ 用贪心算法更简单、更直接、且解题效率更高分

☞ 贪心算法不能保证全局最优

- ⊕ 找硬币问题和硬币面值的特殊性有关
- ⊕ 如果将硬币面值改为：一分、五分和一角一分，
- ⊕ 假设要找给顾客的是一角五分
- ⊕ 利用贪心算法，将找给顾客1个一角一分的硬币和4个一分硬币
- ⊕ 显然，3个五分硬币是最优的解法

贪心算法的基本思想

贪心算法的基本思想

- ⊕ 优化问题的算法往往包含一系列步骤，每一步都有一组选择
- ⊕ 贪心算法在每一步选择中都采取在**当前状态下最优**的选择
 - 目的是希望由此导出的果是最优的
- ⊕ 简言之：贪心算法在求解问题时并不着眼于整体最优
 - 它所作出的选择仅仅是当前看来是最优的
- ⊕ 贪心算法能否得到整体最优解？具体问题，具体分析

贪心算法在有最优子结构的问题中尤为有效

- ⊕ 最优子结构的意思是：局部最优解能决定全局最优解
 - 问题能够分解成子问题来解决
 - 子问题的最优解能递推到最终问题的最优解

贪心算法与动态规划的区别

❧ 动态规划算法

- ⊕ 每一步的最优解是由上一步的局部最优解进行选择得到的
- ⊕ 因此需要保存（之前求解的）所有子问题的最优解备查

❧ 贪心算法

- ⊕ 贪心策略：下一步的最优解是由上一步的最优解推导得到的
- ⊕ 当前最优解包含上一步的最优解，之前的最优解则不作保留
- ⊕ 因此在贪心算法中作出的每步决策都无法改变（不能回退）

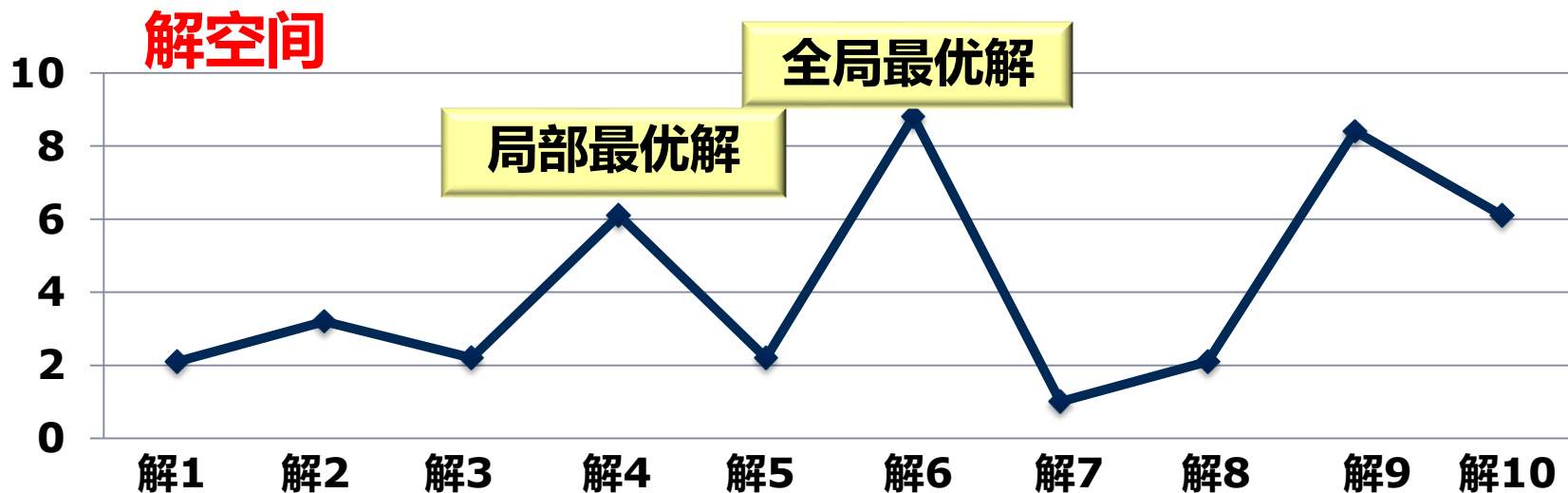
❧ 二者关系

- ⊕ 贪心算法本质上是一种（更快的）动态规划算法
- ⊕ 贪心法正确的条件：每一步的最优解一定包含上一步的最优解
- ⊕ 如果可以证明：在递归求解的每一步，按贪心选择策略选出的局部最优解，最终可导致全局最优解，则二者是等价的

贪心算法的基本思想

需要再次强调的是：贪心算法得到的结果不能保证全局最优

- 虽然贪心算法不能对所有问题都得到全局最优解
 - 但对许多问题它能产生整体最优解
 - 如单源最短路径和最小生成树问题等
- 在另一些情况下，贪心算法的结果是最优解的良好近似
- 在科研和工程实践中被广泛应用（在学习和实践中总结规律）



4.1 活动安排问题

(Activity-Selection Problem)

活动安排问题

问题定义

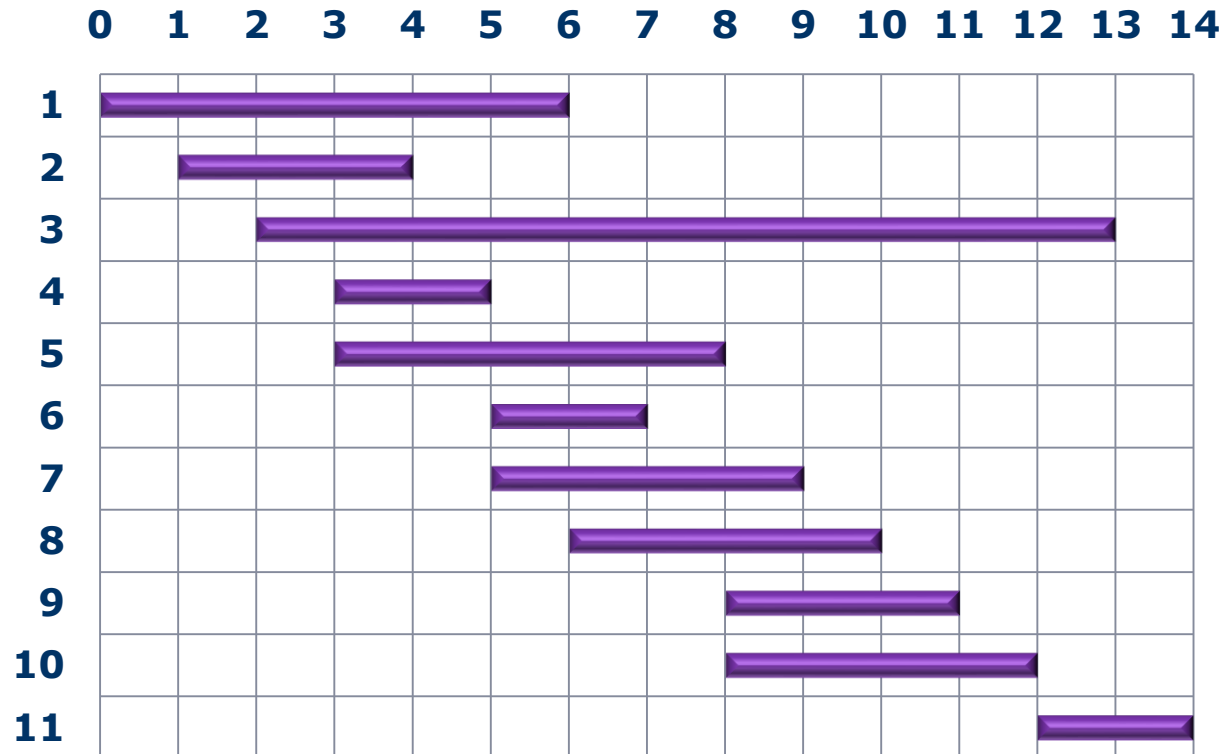
- ⊕ 设：有 n 个活动的集合 $E = \{1, 2, \dots, n\}$
- ⊕ 其中：每个活动都要求竞争使用**同一资源**（如演讲会场等），而在同一时间内**只有一个**活动能使用这一资源
 - 每个活动 i 都有一个请求使用该资源的起始时间 s_i
 - 每个活动 i 都有一个使用资源的结束时间 f_i ，且 $s_i < f_i$
 - 如果选择了活动 i ，则它在半开时间区间 $[s_i, f_i)$ 内占用资源
 - 若区间 $[s_i, f_i)$ 与 $[s_j, f_j)$ 不相交，则称活动 i 与活动 j 是**相容的**
 - 也就是说，当 $s_i \geq f_j$ 或 $s_j \geq f_i$ 时，活动 i 与活动 j 相容
- ⊕ 活动安排问题就是要在所给的活动集合中，选出最大的相容活动子集合，即使得**尽可能多**的活动能兼容地使用公共资源

求解活动安排问题

例：设待安排的11个活动如下：

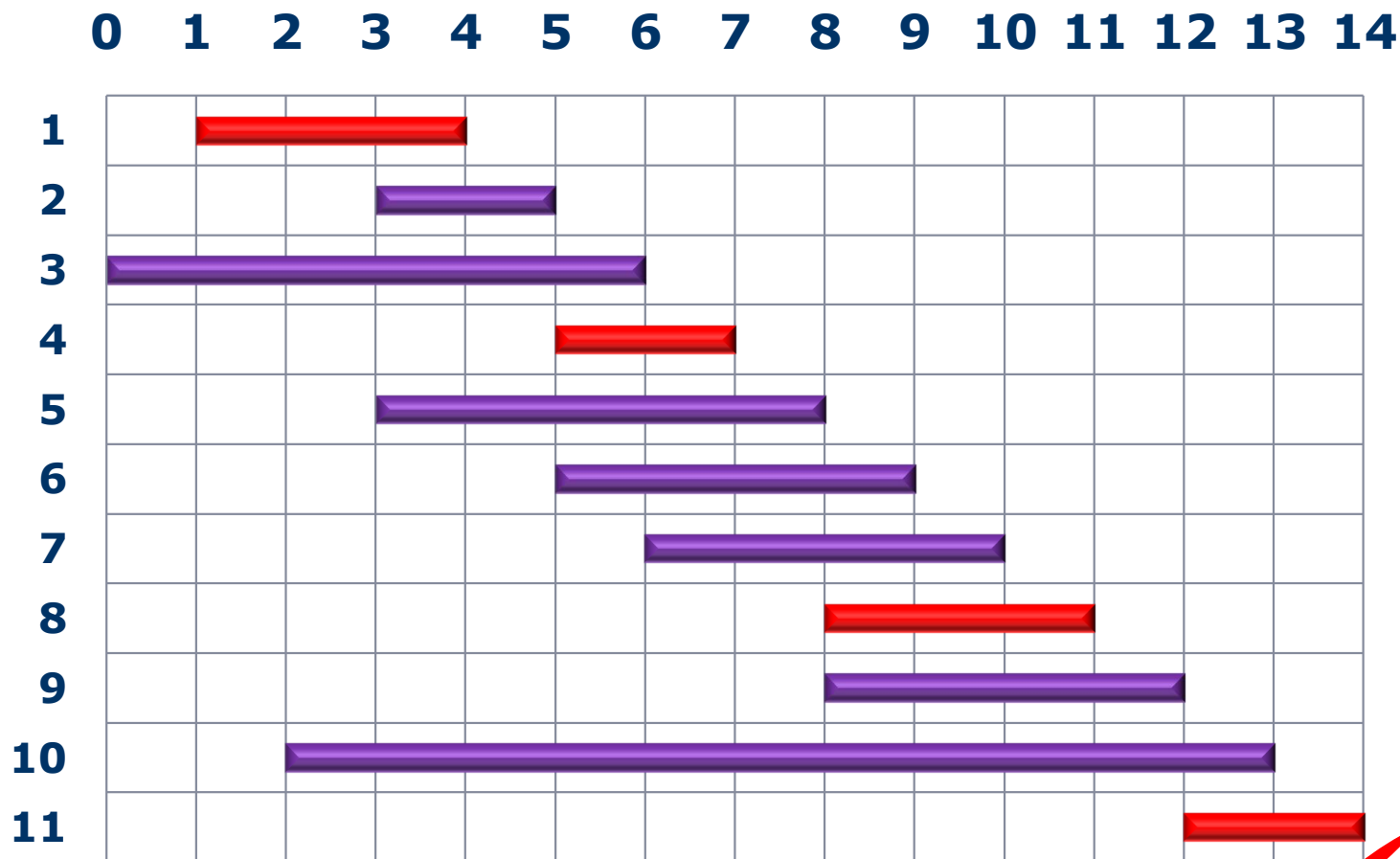
i	1	2	3	4	5	6	7	8	9	10	11
S[i]	0	1	2	3	3	5	5	6	8	8	12
F[i]	6	4	13	5	8	7	9	10	11	12	14

按开始时间非减序排序：



例：设待安排的11个活动按**结束时间**的非减序排列如下：

i	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	12
F[i]	4	5	6	7	8	9	10	11	12	13	14



再问：这种解法能否确保全局最优？



证明：按 $F[1:n]$ 递增顺序进行贪心选择可得全局最优解

证明思路

- ⊕ 证明活动安排问题有一个最优解以贪心选择开始
- ⊕ 用数学归纳法证明贪心算法的解是全局最优解

首先证明活动安排问题有一个最优解以贪心选择开始

- ⊕ 设： $E = \{1, \dots, n\}$ 为给定活动集合（按 $F[k]$ 非减序编号）
 - 显然活动1具有最早的完成时间
- ⊕ 设：集合A是该问题的一个最优解（元素按 $F[k]$ 非减序排列）
 - 不妨设A中的第一个活动是活动k
- ⊕ 若 $k=1$ ，则：A就是一个以贪心选择开始的最优解
- ⊕ 若 $k>1$ ，则设： $B = (A - \{k\}) \cup \{1\}$
 - 由于 $F[1] \leq F[k]$ ，且A中活动相容，故B中活动也相容
 - 由于B和A中包含的活动个数相同，故B也是最优的
- ⊕ 得证：**总存在一个以贪心选择开始的最优活动安排方案**

活动安排问题的最优子结构性质

- 设 $E = \{1, 2, \dots, n\}$ 为所给的活动集合，在做了贪心选择，即选择了活动1后，原问题就简化为对 E 中所有与活动1相容的活动进行活动安排的子问题。
- 即，若 A 是原问题的最优解，则 $A' = A - \{1\}$ 是活动安排问题 $E' = \{i \in E : s_i \geq f_1\}$ 的最优解。
 - 证明：如果能找到 E' 的一个解 B' ，它包含比 A' 更多的活动，则将活动1加入到 B' 中将产生 E 的一个解，它包含比 A 更多的活动。这与 A 的最优性矛盾！
- 因此：在做出贪心选择（活动1）之后，原问题 N 简化为：
子问题 N' ：对 E 中所有与活动1相容的活动进行安排

每一步所做的贪心选择都将问题简化为一个更小的与原问题具有相同形式的子问题。

活动安排问题

∞ 算法设计

- ⊕ 用**数组A[1:n]**来存储所选择的**活动**（设活动总数为n）
 - 约定：若活动i在集合A中，则 $A[i]=1$ ；否则 $A[i]=0$
- ⊕ 各活动的起始时间和结束时间存储于**数组S[1:n]**和**F[1:n]**中
 - 且：数组**F[1:n]**已按结束时间的非减序排列（递增）
- ⊕ 依次从**F[1:n]**中选择活动 i ，尝试加入集合A
 - 设：变量k记录A中最近一次加入的活动：由于F[1:n]有序，所以**F[k]**总是当前集合A中所有活动的最大结束时间
 - 然后依次检查活动 i 是否与当前已选择的所有活动相容
 - 若相容则将活动i加入集合A中；若不相容，则放弃活动i
 - 继续检查**F[1:n]**中下一个活动与集合A中活动的相容性
 - 直到所有活动均已检查完毕，程序结束

活动安排问题

∞ 算法设计（续）

⊕ 新增的活动 i 和当前集合 A 中所有活动相容的充分必要条件是

- $S[i] \geq F[k]$: 即活动 i 的开始时间不早于 k 的结束时间
 - k 为最近加入集合 A 的活动
 - 若条件满足, 则活动 i 取代 k 成为最近加入 A 的活动
- 若 $S[i] < F[k]$, 则放弃活动 i , 转而考虑下一个活动

⊕ 算法的直觉: 这种选择方式为后序活动预留尽可能多的时间

- 由于输入的活动按照其完成时间的非减序排列
- 所以每次总是选择具有最早完成时间的相容活动加入集合 A
- 贪心选择的意义在于: 使剩余的可安排时间段极大化, 以便安排尽可能多的相容活动

贪心算法求解活动安排问题

各活动的起始时间和结束时间存储于数组s[]和f[]中且按结束时间的非减序排列

■ 算法描述

```
Public static int greedySelector(int[ ] s, int[ ] f, boolean a[ ])
```

```
{  
    int n=s.length-1;  
    a[1]=true;  
    int j=1;  
    int count=1;  
    for(int i=2;i<=n;i++){  
        if(s[i]>=f[j]){  
            a[i]=true;  
            j=i;  
            count++;  
        }  
        else a[i]=false;  
    }  
    return count;  
}
```

时间复杂度

$O(n)$

算法用数组a[]存储所选择的活动，活动i在集合A中，当且仅当a[i]=true

变量j记录最近一次加入A的活动，由于输入的活动以其完成时间的非减序排列，所以总是当前集合A中所有活动的最大结束时间。算法开始选择活动1，并将j初始化为1

若当前活动i的结束时间f[i]大于或等于当前集合A中所有活动的最大结束时间f[j]，即s[i] ≥ f[j]，若活动i与之相容，则i成为最近加入集合A中的活动，并取代活动j的位置。

活动安排问题

∞ 算法分析

⊕ **复杂度分析：** 为使最多的活动能相容地使用公共资源

- 若活动已按结束时间的非减序排列：算法只需 $O(n)$ 的时间
- 若活动未按非减序排列：可以用 $O(n\log n)$ 的时间先排序

⊕ **全局最优解**

- 贪心算法并不总能求得问题的整体最优解
- 但对于活动安排问题，贪心算法却总能求得的整体最优解
- 即：它最终所确定的相容活动集合 A 的规模最大
- 这个结论可以用数学归纳法证明

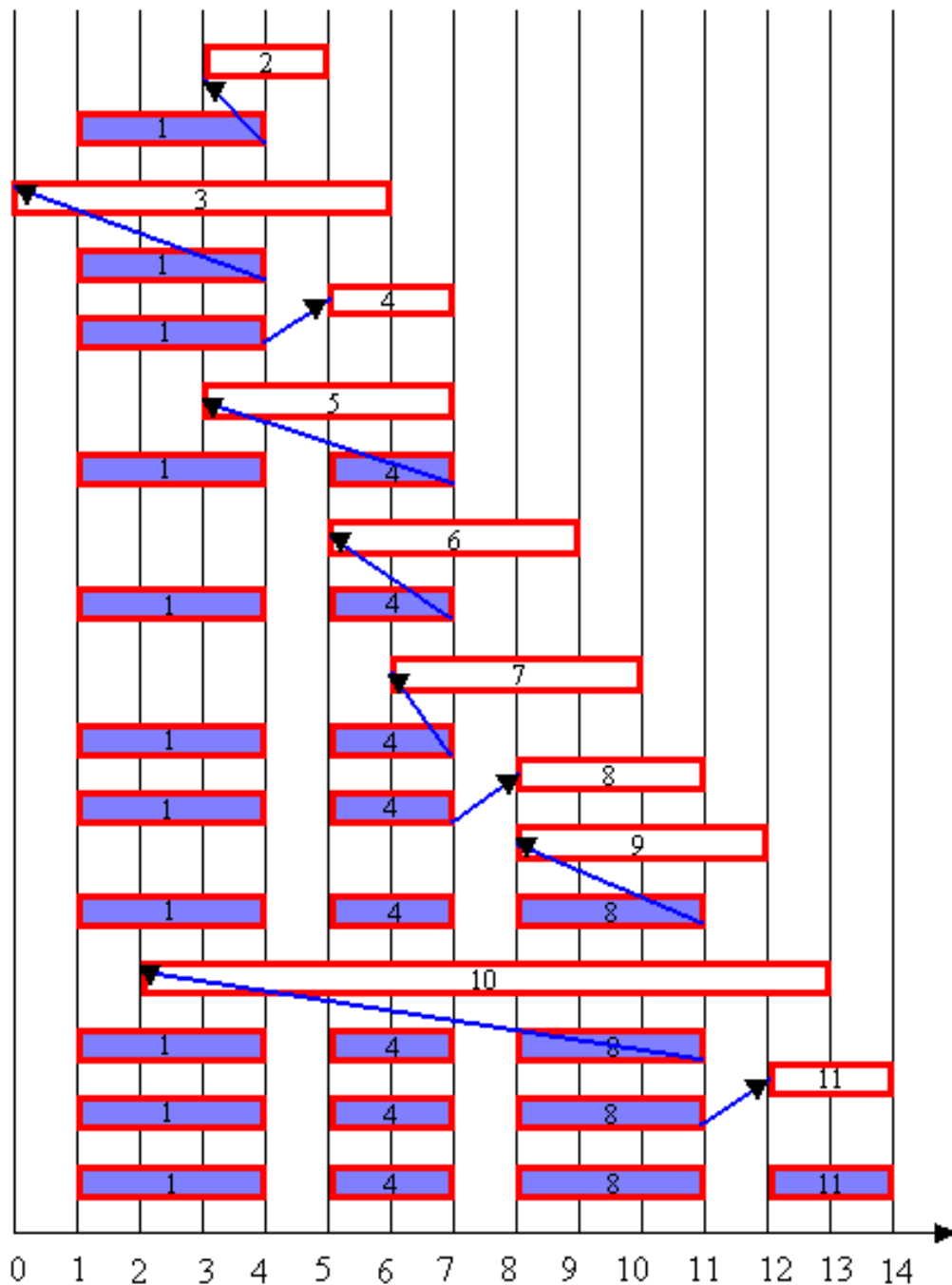
活动安排问题实例

例： 设待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下：

i	1	2	3	4	5	6	7	8	9	10	11
s[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

活动安排问题实例

算法 greedySelector 的计算过程如左图所示。图中每行相应于算法的一次迭代。阴影长条表示的活动是已选入集合A的活动，而空白长条表示的活动是当前正在检查相容性的活动。



4.2 贪心算法的基本要素

贪心算法的基本要素

❧ 应用贪心算法解决具体问题时需要考虑如下两个问题

⊕ 该问题是否可以采用贪心算法求解？

⊕ 采用贪心算法能否得到问题的最优解？

❧ 遗憾的是：对许多实际问题而言，很难给出肯定的回答 😞

❧ 可以用贪心算法求解的问题的一般特征

⊕ 从许多可以用贪心算法求解的问题中可以看到（经验）

⊕ 这类问题一般具有2个重要的性质：

- **贪心选择性质**和**最优子结构性质**

贪心算法的基本要素

∞ 贪心算法的基本要素1：贪心选择性质

- ⊕ 所求问题的整体最优解可以通过一系列局部最优的选择得到
 - 这是贪心算法可行的第一个**基本要素**
 - 对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终能够导致问题的整体最优解

∞ 贪心选择性质是贪心算法与动态规划算法的主要区别

- ⊕ 动态规划算法通常以**自底向上**的方式求解各子问题
- ⊕ 贪心算法则通常以**自顶向下**的方式进行
 - 迭代地做出相继的贪心选择
 - 每一次贪心选择就将所求问题简化为规模更小的子问题

贪心算法的基本要素

∞ 贪心算法的基本要素2：最优子结构性质

- ⊕ 当一个问题最优解包含其子问题的最优解时
- ⊕ 称此问题具有**最优子结构性质**
- ⊕ 这是一个问题可用动态规划算法或贪心算法求解的**关键特征**

∞ 以活动安排问题为例

- ⊕ 贪心选择性质
 - 总存在以贪心选择开始的最优活动安排方案
 - 贪心选择次数就是活动安排问题的全局最优解
- ⊕ 最优子结构性质
 - 每一步做出的贪心选择都将当前问题简化为一个规模更小的与原问题具有相同形式的子问题

贪心算法与动态规划算法

∞ **二者的共同点：** 都要求问题具有最优子结构性质

∞ **问题：** 如果一个问题具有最优子结构性质

⊕ 应选用贪心算法还是动态规划算法求解？

⊕ 能用动态规划求解的问题是否也能用贪心算法求解？

⊕ 以两个经典问题为例，通过比较来说明二者的主要差别

组合优化问题1：背包问题

∞ 0-1背包问题：

- ⊕ 给定 n 种物品和一个背包，背包的容量为 C
- ⊕ 设物品 i 的重量是 $W[i]$ ，其价值为 $V[i]$ 。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大？
- ⊕ 限制条件：在选择装入背包的物品时，对每种物品 i 只有2种选择，即装入背包或不装入背包；不能将物品 i 装入背包多次，也不能只装入部分的物品 i 。（因此称为0/1背包问题）

∞ 背包问题：

- ⊕ 与0-1背包问题类似，所不同的是在选择物品 i 装入背包时，可以选择物品 i 的一部分，而不一定要全部装入背包
- ⊕ 这两类问题都具有相似的最优子结构性质，但**背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解！**

两种背包问题的形式化表达

∞ 0/1背包问题:

- ⊕ 给定: $c > 0, w_i > 0, v_i > 0, 1 \leq i \leq n$
- ⊕ 要求找出一个n元0/1向量 (x_1, x_2, \dots, x_n)
- ⊕ 满足: $\sum_{i=1}^n w_i x_i \leq c$ ($x_i \in \{0, 1\}$)
- ⊕ 使得下式最大化: $\sum_{i=1}^n v_i x_i$

∞ 背包问题:

- ⊕ 给定: $c > 0, w_i > 0, v_i > 0, 1 \leq i \leq n$
- ⊕ 要求找出一个n元向量 (x_1, x_2, \dots, x_n)
- ⊕ 满足: $\sum_{i=1}^n w_i x_i \leq c$ ($0 \leq x_i \leq 1$)
- ⊕ 使得下式最大化: $\sum_{i=1}^n v_i x_i$

两种背包问题都具备最优子结构性质

∞ 0-1背包问题:

- ⊕ 设: A 是能装入背包的最大价值物品集合
- ⊕ 则: $A[k] = A - \{k\}$ 表示 $n-1$ 个物品 $(1, 2, \dots, k-1, k+1, \dots, n)$,
可装入容量为 $C - w[k]$ 的背包的最大价值物品集合

∞ 背包问题:

- ⊕ 若: 它的一个最优解 A 包含物品 k 的一部分
- ⊕ 设: w_k' 是从 A 中拿出所含物品 k 的部分重量
- ⊕ 则: 剩余的就是从 $n-1$ 个原有物品 $(1, 2, \dots, k-1, k+1, \dots, n)$
以及重量为 $(w_k - w_k')$ 的物品 k 当中,
可装入容量为 $C - w_k'$ 的背包的最大价值物品集合

采用贪心算法求解背包问题

贪心算法求解背包问题的基本步骤：

- ⊕ 首先计算每种物品单位重量的价值： V_i/W_i
- ⊕ 然后按照贪心选择策略
 - 将尽可能多的单位重量价值最高的物品装入背包
 - 若将这种物品全部装入后，背包内的物品总重量未超过C
 - 则选择单位重量价值次高的物品并尽可能多地装入背包
- ⊕ 依此策略一直地进行下去，直到背包装满为止

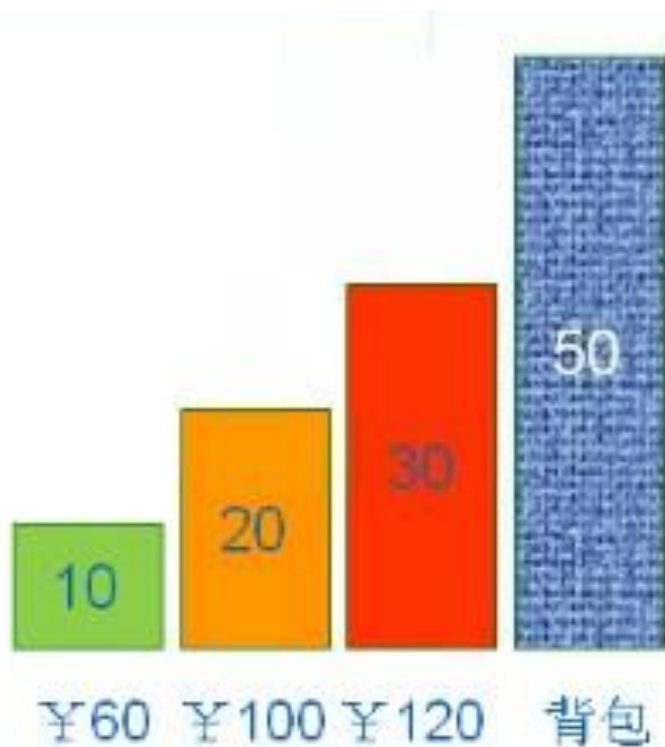
算法复杂度分析

- ⊕ 计算时间主要用于对各种物品按单位重量的价值排序
- ⊕ 因此算法的计算时间上界为： **$O(n \log n)$**

贪心算法与动态规划算法的差异

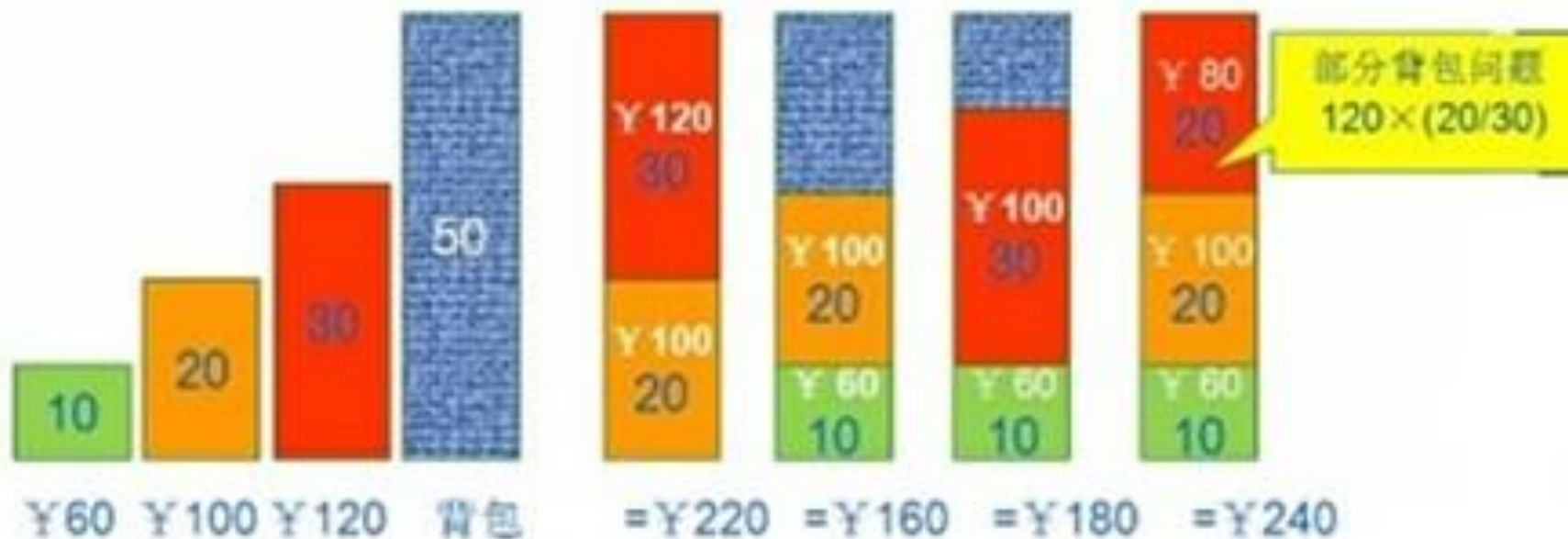
■ 例子

- 总共3件物品，背包容量50磅。物品1重10磅，价值60元，物品2重20磅，价值100元，物品3重30磅，价值120元。



贪心算法与动态规划算法的差异

- 物品1每磅价值6元，大于物品2的每磅价值5元和物品3的每磅价值4元。
- 对0-1背包问题，按照贪心策略的话就要物品1。然而从图中看出，最优解取的是物品2和3。选择物品1的可能解都是次优的。
- 对背包问题，选择物品1可以达到最优。



分析：贪心算法与动态规划算法的差异

∞ **思考：对于0-1背包问题，贪心选择为什么不能得到最优解**

- ⊕ 因为对该问题采用贪心选择策略无法确保最终能将背包装满
- ⊕ 部分闲置的背包空间降低了每公斤背包空间的价值

∞ **思考：求解0-1背包问题应采取什么样的思路？**

- ⊕ 对每个物品，应比较：选择和不选择该物品所形成的方案
- ⊕ 然后再作出最优选择（自底向上逐步求解）
- ⊕ 由此会导致出现许多互相重叠的子问题
 - 例如：装入物品 k 和 $k+1$ 时，都要考虑物品 n 是否装入
- ⊕ 重叠子问题性质正是该问题可用动态规划算法求解的重要特征
 - 事实上动态规划算法的确可以有效求解0/1背包问题

组合优化问题2：最优装载问题

∞ **问题描述：**有一批集装箱要装船

⊕ 其中：集装箱 i 的重量为 w_i ，轮船最大载重量为 c

⊕ 要求：在**不受体积限制**的情况下，将尽可能多的集装箱装船

∞ **问题形式化描述**

⊕ 给定： $c > 0$, $w_i > 0$, $v_i > 0$, $1 \leq i \leq n$

⊕ 要求：找出一个 n 元 0/1 向量 $\mathbf{x} = (x_1, x_2, \dots, x_n)$ 其中 $x_i \in \{0, 1\}$

⊕ 使得： $\sum_{i=1}^n w_i x_i \leq c$ ，而且 $\sum_{i=1}^n x_i$ 达到最大

∞ **问题分析**

⊕ 首先再看该问题是否满足贪心选择性质

⊕ 然后看该问题是否具有最优子结构性质

证明：最优装载问题满足贪心选择性质

设：集装箱已按重量从小到大排序

设： $\mathbf{X}=(x_1, x_2, \dots, x_n)$ 是最优装载问题的一个最优解

设： $k = \min\{ i \mid x_i=1, 1 \leq i \leq n \}$ (第一个非零元素的下标)

- 易知，如果给定的最优装载问题有解，则 $1 \leq k \leq n$
- 当 $k=1$ 时， \mathbf{X} 是一个满足贪心选择性质的最优解
- 当 $k>1$ 时，取 $y_1=1$; $y_k=0$; $y_i=x_i, 1 < i \leq n, i \neq k$, 则：

$$\sum_{i=1}^n w_i y_i = (w_1 - w_k) + \sum_{i=1}^n w_i x_i \leq \sum_{i=1}^n w_i x_i \leq c$$

- 因此： $\mathbf{Y}=(y_1, y_2, \dots, y_n)$ 是所给最优装载问题的可行解

由于： $\sum_{i=1}^n y_i = \sum_{i=1}^n x_i$ ，故： \mathbf{Y} 也是满足贪心选择性质的最优解

因此：最优装载问题满足贪心选择性质！

组合优化问题2：最优装载问题

∞ 最优装载问题具有最优子结构性质

- ⊕ 设 $X=(x_1, x_2, \dots, x_n)$ 是载重量为 c ，集装箱为 $\{1, 2, \dots, n\}$ 的最优装载问题满足贪心选择性质的的一个最优解
- ⊕ 由该问题的贪心选择性质易知： $x_1=1$
- ⊕ 且 $X'=(x_2, x_3, \dots, x_n)$ 是如下子问题的最优解
 - 载重量为 $c-w_1$ ，集装箱为 $\{2, 3, \dots, n\}$ 的最优装载问题
- ⊕ 因此：最优装载问题具有最优子结构性质！

∞ 算法描述：采用贪心算法求解

- ⊕ 贪心选择策略：重量最轻者先装船
- ⊕ 根据以上分析：由此可产生该装载问题的最优解

最优装载问题的贪心算法

```
void loading(int x[], int w[], int c, int n) {  
    int *R = (int *)malloc((n+1)*sizeof(int));  
    // 根据w从小到大排序, 数组R记录调整后的序号  
    sort(w, R, n);  
    for (int i = 1; i <= n; i++) x[i] = 0;  
    for (int i = 1; i <= n; i++) {  
        int id = R[i];  
        if (w[id] > c) break;  
        x[id] = 1; c -= w[id];  
    }  
}
```

时间复杂度: $O(n \log n)$

4.4 单源最短路径

(Single Source Shortest Paths)

最短路径问题

问题:

- 1. 两地之间是否有通路?**
- 2. 若存在多条通路, 哪条路最短?**

最短路径问题

☞ 单源最短路径

Single-Source Shortest Path
(Dijkstra算法)

- 所有顶点对间的最短路径问题

All-Pairs Shortest paths
(Floyd算法)

单源最短路径

☞ 在有向图中，寻找从某个源点到其余各个顶点或者每一对顶点之间的最短带权路径的运算，称为最短路径问题

☞ 单源最短路径问题

⊕ 给定：带权有向图 $G=(V,E)$

- 其中：每条边的权是非负实数

⊕ 给定顶点集合 V 中的一个顶点 v ，称为源点

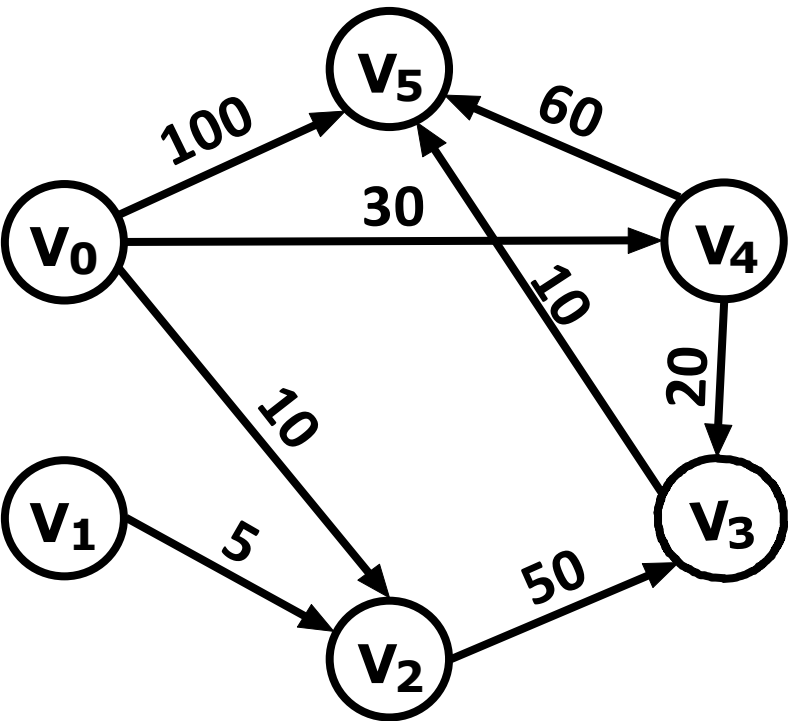
⊕ 求解：从源点 v 到 G 中其余各顶点之间的最短路径

- 这里路径长度是指各条边的权值之和

例：道路图：从沙河校区到清水河校区的最短路径？

例如：给定带权有向图G

- 从图中可见：从 v_0 到 v_1 没有路径
- 从 v_0 到 v_3 有两条不同的路径： (v_0, v_2, v_3) 和 (v_0, v_4, v_3)
- 前者长度为60，而后者长度为50
- 因此后者是从 v_0 到 v_3 的最短路径
- 从 v_0 到其余各顶点之间的最短路径参见下表



v_0 到各顶点的最短路径

源点	终点	最短路径	路径长度
v_0	v_1	----	----
	v_2	(v_0, v_2)	10
	v_3	(v_0, v_4, v_3)	50
	v_4	(v_0, v_4)	30
	v_5	(v_0, v_4, v_3, v_5)	60

迪杰斯特拉 (Dijkstra) 算法

∞ Dijkstra算法是求解单源最短路径问题的一种有效算法

∞ 算法基本思想（贪心算法的思想）：

⊕ 将图中所有顶点分成两组： S , $V-S$

- S ：已确定最短路径的顶点的集合
- $T=V-S$ ：尚未确定最短路径的顶点集合
- 初始时，集合 S 中仅包含源点 V_0 。
- 不断在集合 T 中做贪心选择扩充集合 S
- 直到 S 中包含了 V 中的所有顶点

迪杰斯特拉 (Dijkstra) 算法

∞ 算法设计思路:

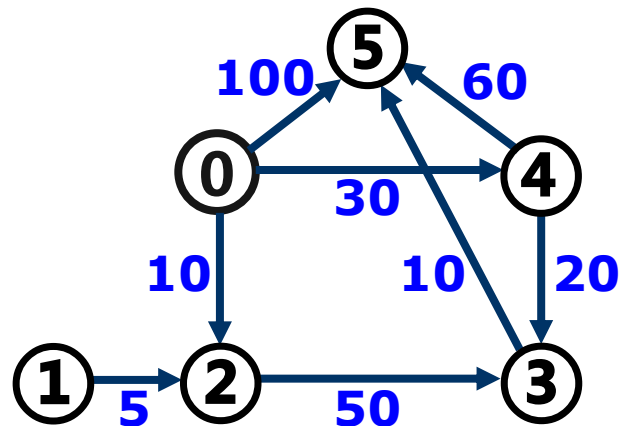
- ⊕ 初始时, S 仅包含源 v_0 ,
- ⊕ 定义 “特殊路径” :
 - ⊕ 从源 v_0 到 G 中某一顶点 u 且中间只经过 S 中顶点的路径称为从源到 u 的特殊路径。
 - ⊕ 用数组元素 $\text{dist}[u]$ 记录源 v_0 到 u 的**最短**特殊路径的长度
- ⊕ Dijkstra 算法每次从 T 中取出具有最短特殊路径长度的顶点 u , 将 u 添加到 S 中, 同时对数组 dist 作必要的修改。一旦 S 包含了所有 V 中顶点, dist 就记录了从源到其它所有顶点之间的最短路径长度。

Dijkstra算法流程

∞ Dijkstra算法的数据结构设计

- 使用带权邻接矩阵表示有向图G
- 辅助数组: $S[nvex]$ ($nvex$ 为图中顶点)
 - + 表示已找到从 V_0 出发的最短路径的终点的集合
- 辅助数组: $dist[nvex]$
 - + 存放当前找到的从 V_0 到每个 V_i 的最短路径长度
- 辅助数组: $prev[nvex]$ (存储最短路径)
 - + 数组元素为从 V_0 到各顶点的最短路径上该顶点的前一顶点的序号 (若从 V_0 到某终点无路径, 则用-1表示)

例子



	0	1	2	3	4	5
0	∞	∞	10	∞	30	100
1	∞	∞	5	∞	∞	∞
2	∞	∞	∞	50	∞	∞
3	∞	∞	∞	∞	∞	10
4	∞	∞	∞	20	∞	60
5	∞	∞	∞	∞	∞	∞

终点	从v ₀ 到各终点的最短路径和路径长度值 (dist)				
v ₁	∞	∞	∞	∞	∞
v ₂	10 (v ₀ ,v ₂)	X	X	X	X
v ₃	∞	60(v ₀ ,v ₂ ,v ₃)	50(v ₀ ,v ₄ ,v ₃)	X	X
v ₄	30 (v ₀ ,v ₄)	30(v ₀ ,v ₄)	X	X	X
v ₅	100(v ₀ ,v ₅)	100(v ₀ ,v ₅)	90(v ₀ ,v ₄ ,v ₅)	60(v ₀ ,v ₄ ,v ₃ ,v ₅)	X
v _i	v ₂	v ₄	v ₃	v ₅	

Dijkstra算法流程说明

∞ 算法伪代码:

1. 初始化条件:

- 令: $S = \{V_0\}$, $T = \{\text{其余顶点}\}$
- T 中顶点 V_i 对应的距离值 $\text{dist}[i]$ 为:
 - + 若存在 $\langle V_0, V_i \rangle$: $\text{dist}[i]$ 为 $\langle V_0, V_i \rangle$ 弧上的权值
 - + 若不存在 $\langle V_0, V_i \rangle$: $\text{dist}[i]$ 为 ∞

2. 从 T 中选取一个 dist 距离值最小的顶点 u 加入 S

3. 对 T 中每一个顶点 V_j 的距离值 $\text{dist}[j]$ 进行修改:

- 若增加 u 作中间顶点之后, 从 V_0 到 V_j 的距离值比不加 u 的路径要短, 则更新 V_j 距离值 (为较小的值)

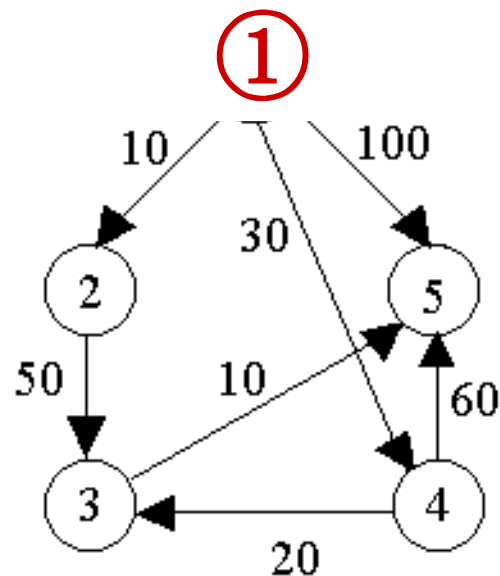
4. 重复上述步骤, 直到 S 中包含所有顶点 (即 $S=V$) 为止

```

template<class Type>
void Dijkstra(int n, int v, Type dist[], int prev[], Type **c) //v为源顶点的序号
{
    bool s[maxint];
    for(int i=1; i<=n; i++) {
        dist[i]=c[v][i]; //c[v][i]是边[v,i]的权
        s[i]=false;
        if (dist[i] == maxint) prev[i]=0; //prev[i]记录从源到顶点i的路径上
        else prev[i]=v; //i的前一个顶点, 初始为0或源v
    }
    dist[v]=0; s[v]=true;
    for (int i=1; i<n; i++) {
        int temp=maxint;
        int u=v;
        for(int j=1; j<=n; j++) //V-S中找最小的dist[j]
            if(!s[j] && (dist[j]<temp)) { u=j; temp=dist[j]; }
        s[u]=true;
        for(int j=1; j<=n; j++) //修改V-S中的dist[j]
            if((!s[j] && (c[u][j] < maxint )){
                Type newdist=dist[u]+c[u][j];
                if(newdist < dist[j]) { dist[j]=newdist; prev[j]=u; } //有改善
            }
    }
}

```

初始状态下，S中只有一个点（源点v1）。



s:

1	0	0	0	0
---	---	---	---	---

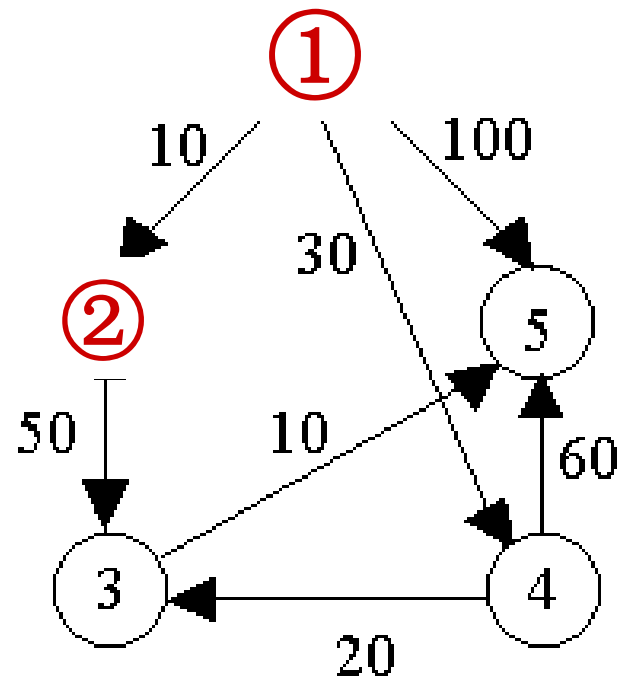
dist:

0	10	∞	30	100
---	----	----------	----	-----

prev:

-1	1	-1	1	1
----	---	----	---	---

第二步，将S外距离S最近的点v2加入S。更新相应信息。



s:

1	1	0	0	0
---	---	---	---	---

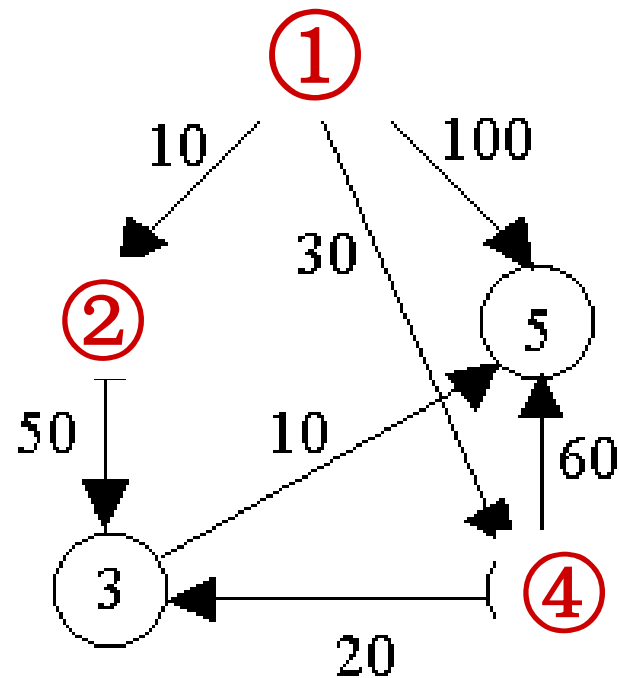
dist :

0	10	60	30	100
---	----	----	----	-----

prev:

-1	1	2	1	1
----	---	---	---	---

第三步，将S外距离S最近的点v4加入S。更新相应信息。



s:

1	1	0	1	0
---	---	---	---	---

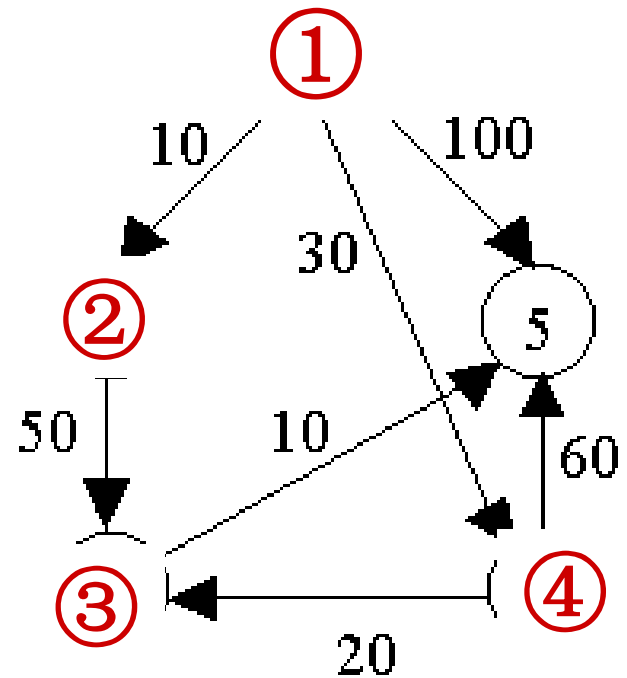
dist :

0	10	50	30	90
---	----	----	----	----

prev:

-1	1	4	1	4
----	---	---	---	---

第四步，将S外距离S最近的点v3加入S。更新相应信息。



s:

1	1	1	1	0
---	---	---	---	---

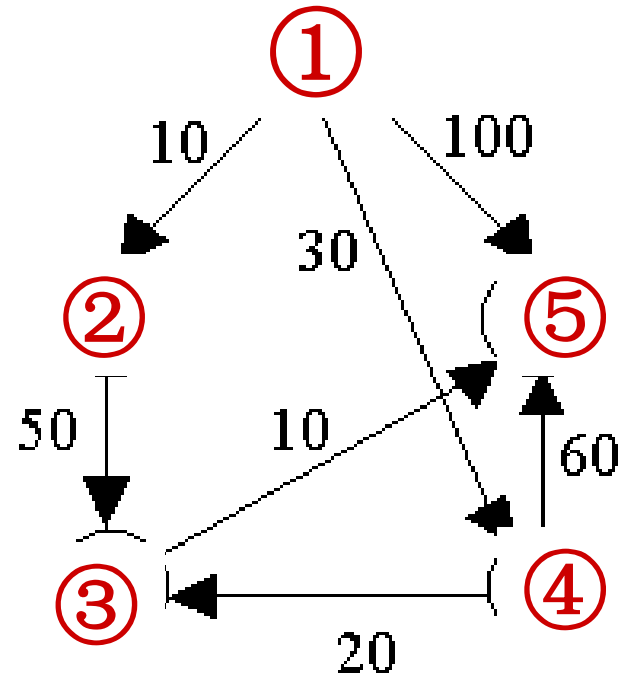
dist :

0	10	50	30	60
---	----	----	----	----

prev:

-1	1	4	1	3
----	---	---	---	---

第五步，将S外距离S最近的点v5加入S。更新相应信息。



s:

1	1	1	1	1
---	---	---	---	---

dist :

0	10	50	30	60
---	----	----	----	----

prev:

-1	1	4	1	3
----	---	---	---	---

单源最短路径

∞ 算法复杂性分析

⊕ 对于具有 n 个顶点和 e 条边的带权有向图 G

- 如果用带权邻接矩阵表示图 G
- 则：Dijkstra算法的循环体需要 $O(n)$ 时间
- 这个循环需要执行 $n-1$ 次，所以完成循环需时 $O(n^2)$

⊕ 算法的其余部分需要时间不超过 $O(n^2)$

单源最短路径

∞ 算法正确性分析

⊕ 贪心选择性质:

- 从 $V-S$ 中选择具有最短特殊路径的顶点 u , 则从源到 u 的最短路径长度 $\text{dist}[u]$ 。

• 最优子结构性质:

- 如果顶点序列 $\{V_i, \dots, V_k, \dots, V_s, \dots, V_j\}$ 是源 V_i 到顶点 V_j 的最短路径, 则 $\{V_k, \dots, V_s\}$ 是源 V_k 到顶点 V_s 的最短路径。

4.5 多机调度问题

(MultiProcessor Scheduling)

多机调度问题

问题定义

- ⊕ 设：有 n 个独立的作业 $\{1, 2, \dots, n\}$
- ⊕ 设：这 n 个作业由 m 台相同的机器进行加工处理
 - 作业 i 所需要的执行时间为： t_i
- ⊕ 约定：
 - 每个作业均可以在任何一个机器加工处理
 - 但作业未完成之前不容许中断处理
 - 作业也不能拆分为更小的子作业
- ⊕ 多机调度问题要求：给出一种作业调度方案，使所给的 n 个作业在**尽可能短的时间内**由 m 台机器加工处理完成

多机调度问题

∞ 问题分析

- ⊕ 这个问题是NP完全问题，到目前为止还没有十分有效的解法
- ⊕ 对于这一类问题（NP完全问题）
 - 用贪心选择策略有时可以设计出较好的近似算法
- ⊕ 具体说来：采用最长处理时间作业优先的贪心选择策略可以设计出解多机调度问题的较好的近似算法

多机调度问题

∞ 贪心算法求解多机调度问题

⊕ 贪心选择策略：**最长处理时间作业优先**

⊕ 当 $n \leq m$ 时

- 只要将机器 i 的 $[0, t_i]$ 时间区间分配给作业 i 即可
- 算法只需要 **$O(1)$** 时间

⊕ 当 $n > m$ 时



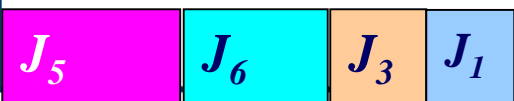
- 首先将 n 个作业依其所需的处理时间从大到小排序
- 然后依次顺序将作业分配给空闲的处理机
- 算法所需的计算时间为 **$O(n \log n)$**

多机调度问题

例如：

⊕ 设7个独立作业{1,2,3,4,5,6,7}，所需的处理时间分别为{2,14,4,16,6,5,3}，由3台机器 $M_1 \sim M_3$ 加工处理。

⊕ 排序： $J_4, J_2, J_5, J_6, J_3, J_7, J_1$

机器	调度方案	所用时间
M_1		16
M_2		$14+3=17$
M_3		$6+5+4+2=17$

```

template<class Type>
void Greedy(Type a[], int n, int m)
{  if(n<=m) {
    cout<<“为每个作业分配一台机器”<<endl;
    return;
  }
  Sort(a,n); //将n个作业依其所需的处理时间从小到大排序
  MinHeap<MachineNode>H(m);
  MachineNode x;
  for(int i=1; i<=m; i++) { //为机器建初始堆，堆顶为M1
    x.avail=0; //初始化：每个机器均没作业，其处理作业结束时间均为0
    x.ID=i;
    H.Insert(x);
  }
  for(int i=n; i>=1; i--) { //机器按处理作业结束时间从小到大排序
    H.DeleteMin(x);
    cout<<“将机器”<<x.ID<<“从”<<x.avail<<“到”
      <<(x.avail+a[i].time<<“的时间段分配给作业”<<a[i].ID<<endl;
    x.avail += a[i].time;
    H.Insert(x);
  }
}

```

三类常用算法小结

∞ **Divide-and-conquer**

- ⊕ Break up a problem into some sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

∞ **Dynamic programming**

- ⊕ Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

∞ **Greed**

- ⊕ Build up a solution incrementally, myopically optimizing some local criterion.

作业三：动态规划算法（期末之前提交）

∞ 给定如下矩阵链： $p = (10, 100, 5, 50, 30, 20, 60, 45, 50)$

- 问题（1）：请写出其最优完全加括号方式
- 问题（2）：请画出该问题的语法树和相应的最优三角剖分图

∞ 给定如下有序集： $s = \{k_1, k_2, \dots, k_5\}$ ($k_1 < k_2 < \dots < k_5$)

- 其对应元素的查找概率分布如下：

a_0	b_1	a_1	b_2	a_2	b_3	a_3	b_4	a_4	b_5	a_5
0.05	0.15	0.1	0.1	0.05	0.05	0.05	0.1	0.05	0.2	0.1

- 问题（1）：请描述求解最优二叉搜索树的动态规划算法
- 问题（2）：计算并填出最优值 $m(i, j)$ 、子树概率 $w_{i, j}$ 和根节点标识 $xr(s[i][j]=r)$ 的表格

作业三：贪心算法

已知有若干活动的起止时间如下表所示

i	1	2	3	4	5	6	7	8	9
s[i]	2	2	3	4	6	7	9	10	13
f[i]	3	4	5	7	8	11	12	15	17

- 问题：最多可以安排多少活动？为哪些？请编程实现输出

设字符a, b, c, d, e, f, g在某文件中出现的频率依次是：（选做）

- 31%, 12%, 5%, 2%, 10%, 18%, 22%
- 请画出哈夫曼树，给出各字符编码，并计算平均码长