



贪心法

Greedy Algorithm

湖南大学信息科学与工程学院



提纲

9.1 概述

9.2 活动选择问题

9.3 贪心算法原理

9.4 哈夫曼编码

9.5 (分数) 背包问题

9.6 多机调度问题

9.7 贪心算法与人工智能算法

9.1 概述-初识贪心算法

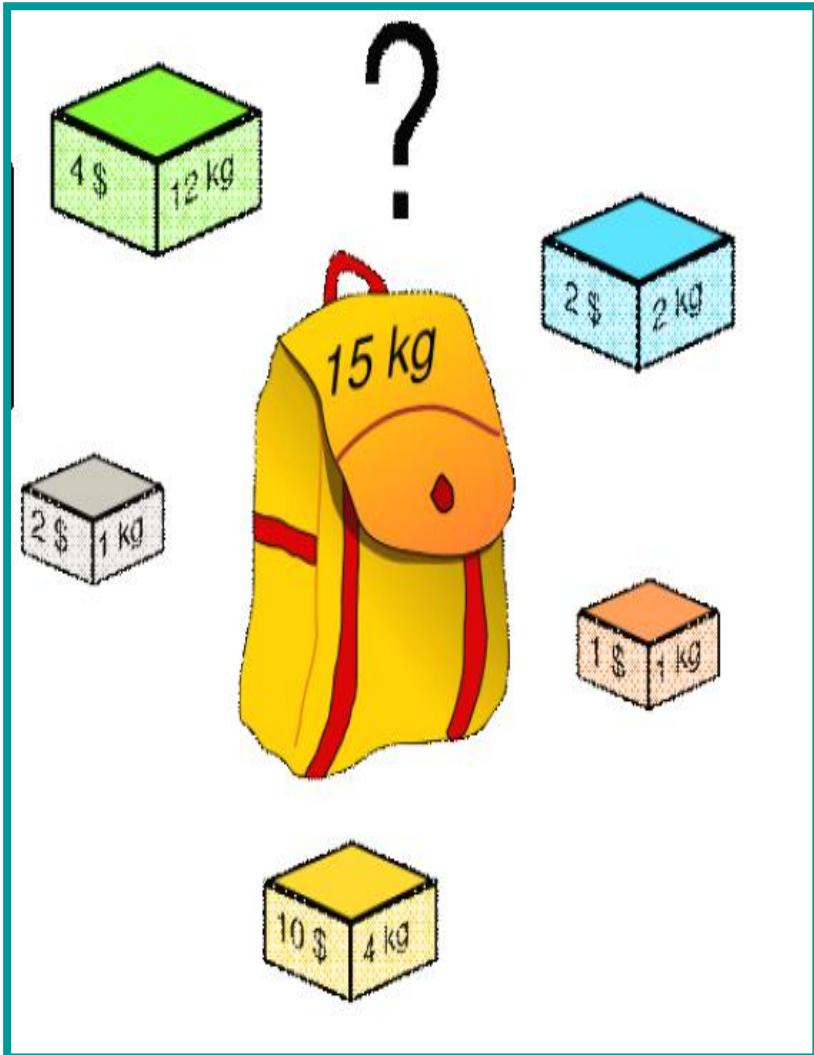


1元、2元、5元、10元、20元、50元、100元的纸币分别有若干张。

现在要用这些钱来支付K元，至少要用多少张纸币？



9.1 概述-初识贪心算法



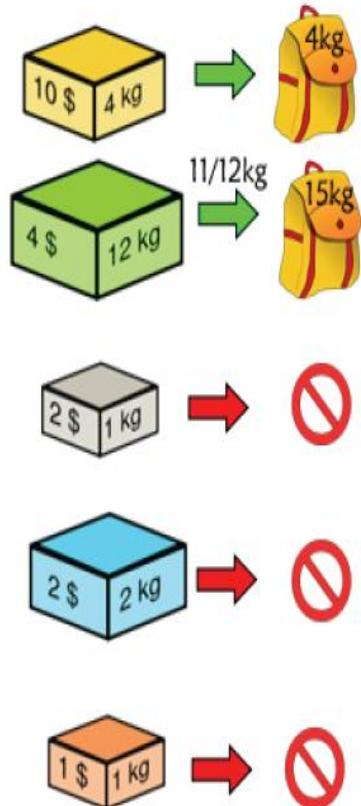
采用怎样的装包方法
才会使装入背包物品
的总收益最大?

9.1 概述-初识贪心算法



方案1：按物品价值降序装包

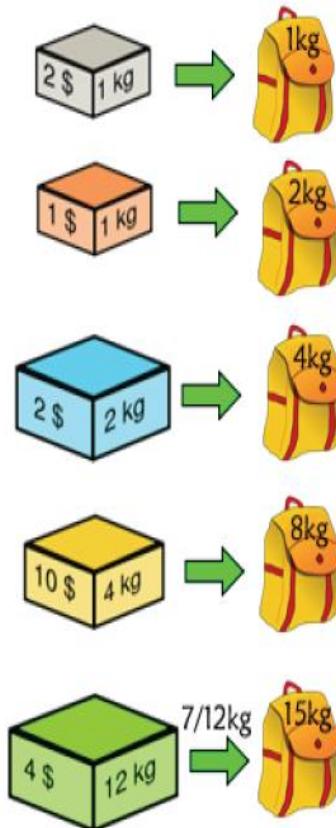
Max=15kg



总价值: 13.67 \$

方案2：按物品重量升序装包

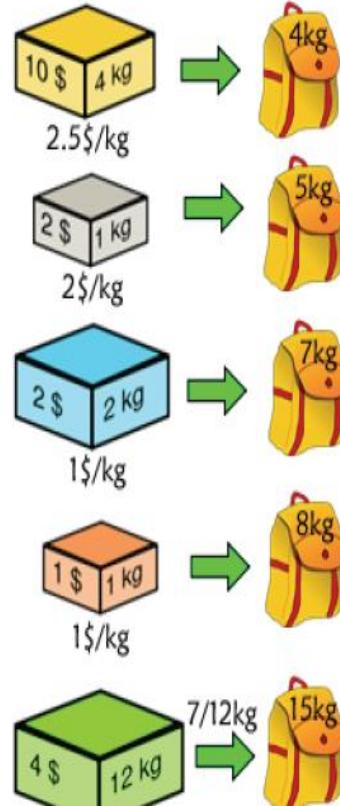
Max=15kg



总价值: 17.33 \$

方案3：按物品价值与重量比值的降序装包

Max=15kg

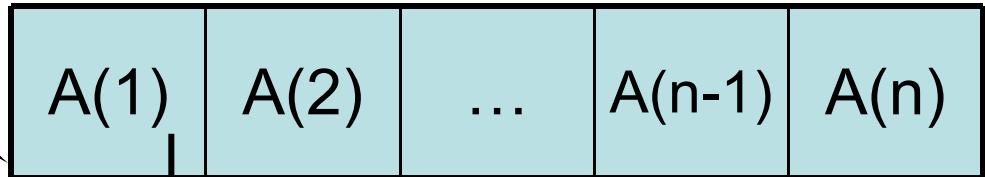


总价值: 17.33 \$



9.1 概述

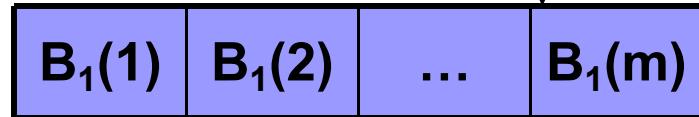
某一问题的n个输入



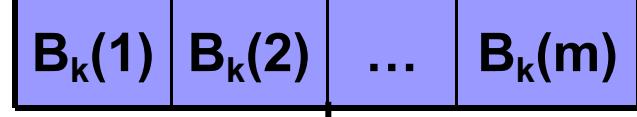
是A的一个子集

满足一定的条件

约束条件



...



该问题的一种解 (可行解)

取极值

目标函数

最优解

9.1 概述



贪心法: 将问题的求解过程看作是一系列选择,每次选择一个输入,每次选择都是当前状态下的最好选择(局部最优解).每作一次选择后,所求问题会简化为一个规模更小的子问题.从而通过每一步的最优解逐步达到整体的最优解。

[常见应用]背包问题,最小生成树,最短路径,作业调度等等

[算法优点]求解速度快,时间复杂性有较低的阶.

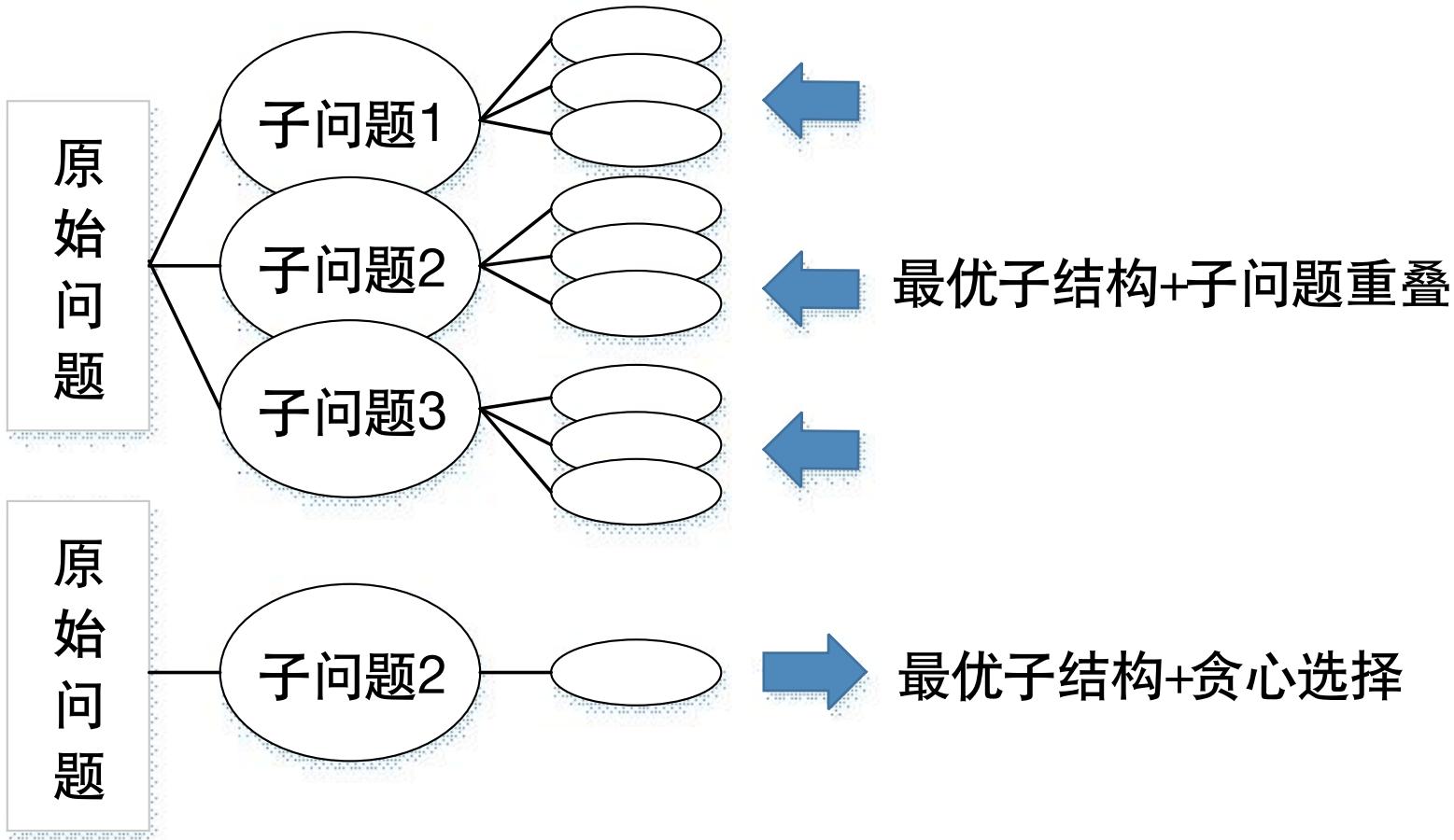
[算法缺点]需证明是最优解.



9.1 贪心算法原理

贪心算法 vs 动态规划法

动态规划法
贪心法





提纲

9.1 概述

9.2 活动选择问题

9.3 贪心算法原理

9.4 哈夫曼编码

9.5 (分数) 背包问题

9.6 多机调度问题

9.7 贪心算法与人工智能算法



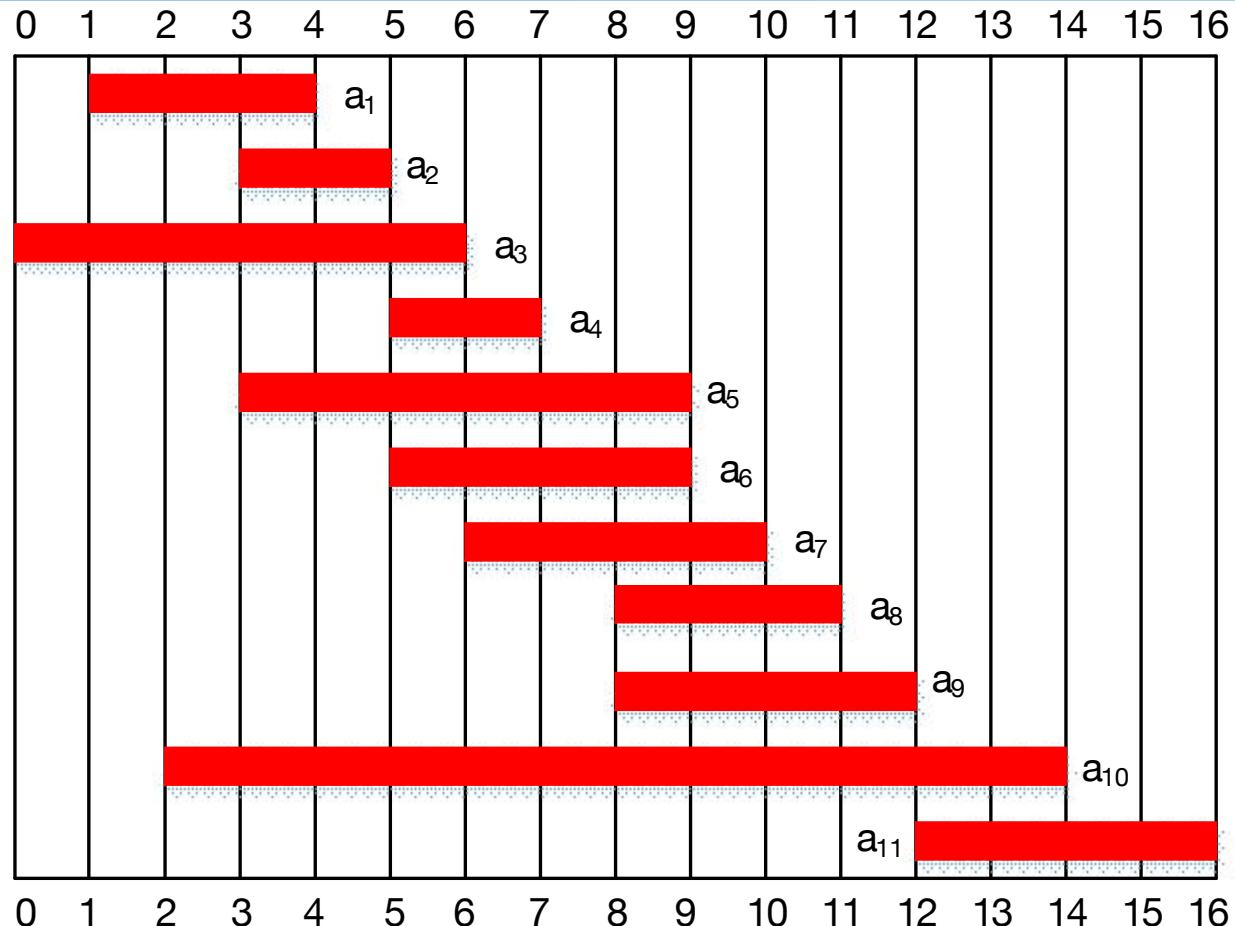
9.2 活动选择问题

问题描述:

假设我们存在这样一个活动集合 $S=\{a_1, a_2, a_3, a_4, \dots, a_n\}$, 其中每一个活动 a_i 都有一个开始时间 s_i 和结束时间 f_i 保证($0 \leq s_i < f_i$), 活动 a_i 进行时, 那么它占用的时间为 $[s_i, f_i]$. 现在这些活动占用一个共同的资源(教室), 就是这些活动会在某一时间段里面进行安排, 如果两个活动 a_i 和 a_j 的占用时间 $[s_i, f_i], [s_j, f_j]$ 不重叠, 那么就说明这两个活动是兼容的, 也就是说当 $s_j >= f_i$ 或者 $s_i >= f_j$ 那么活动 a_i, a_j 是兼容的。在活动选择问题中, 我们希望选出一个最大兼容活动集, 即在同一间教室能安排数量最多的活动。



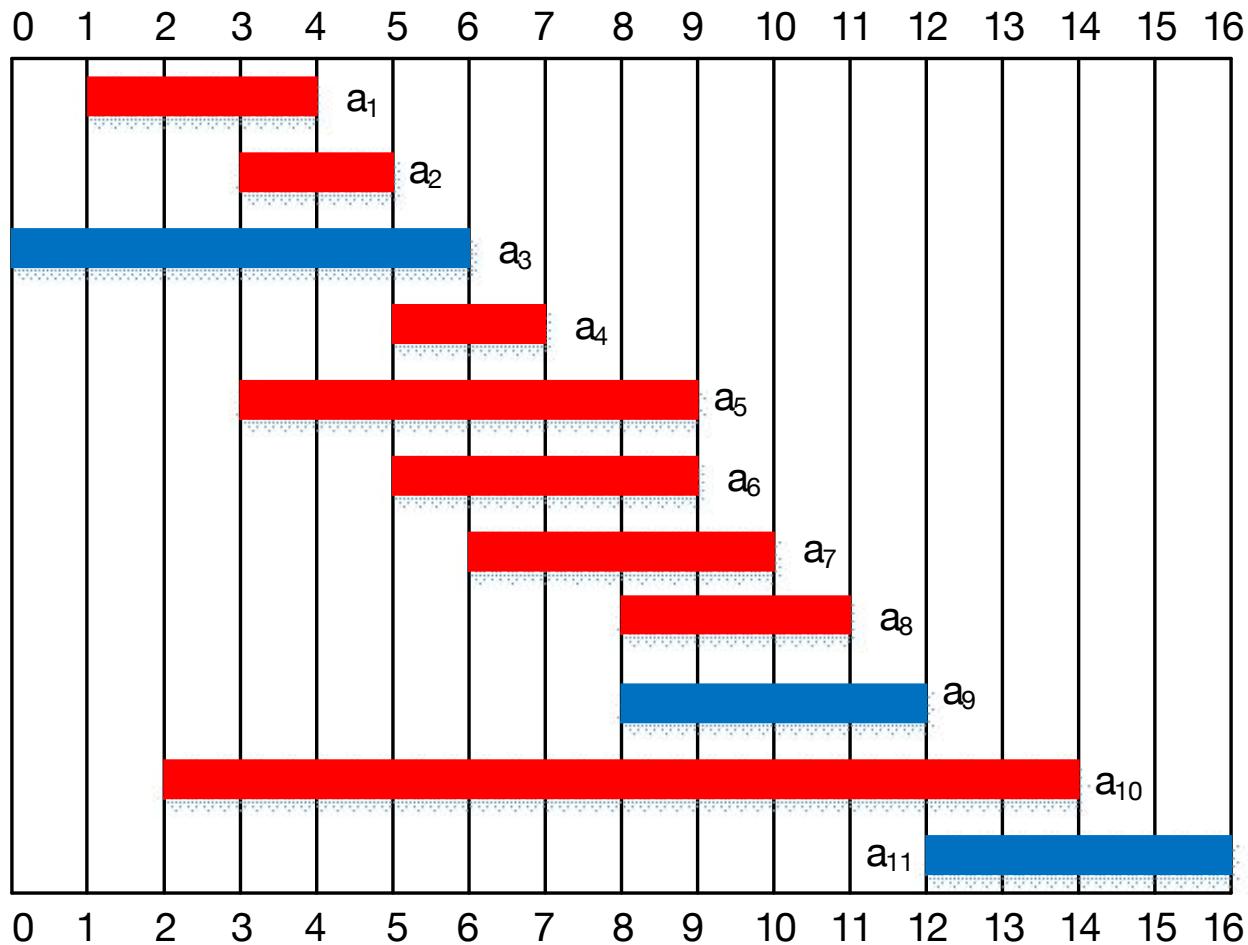
9.2 活动选择问题



| 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 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

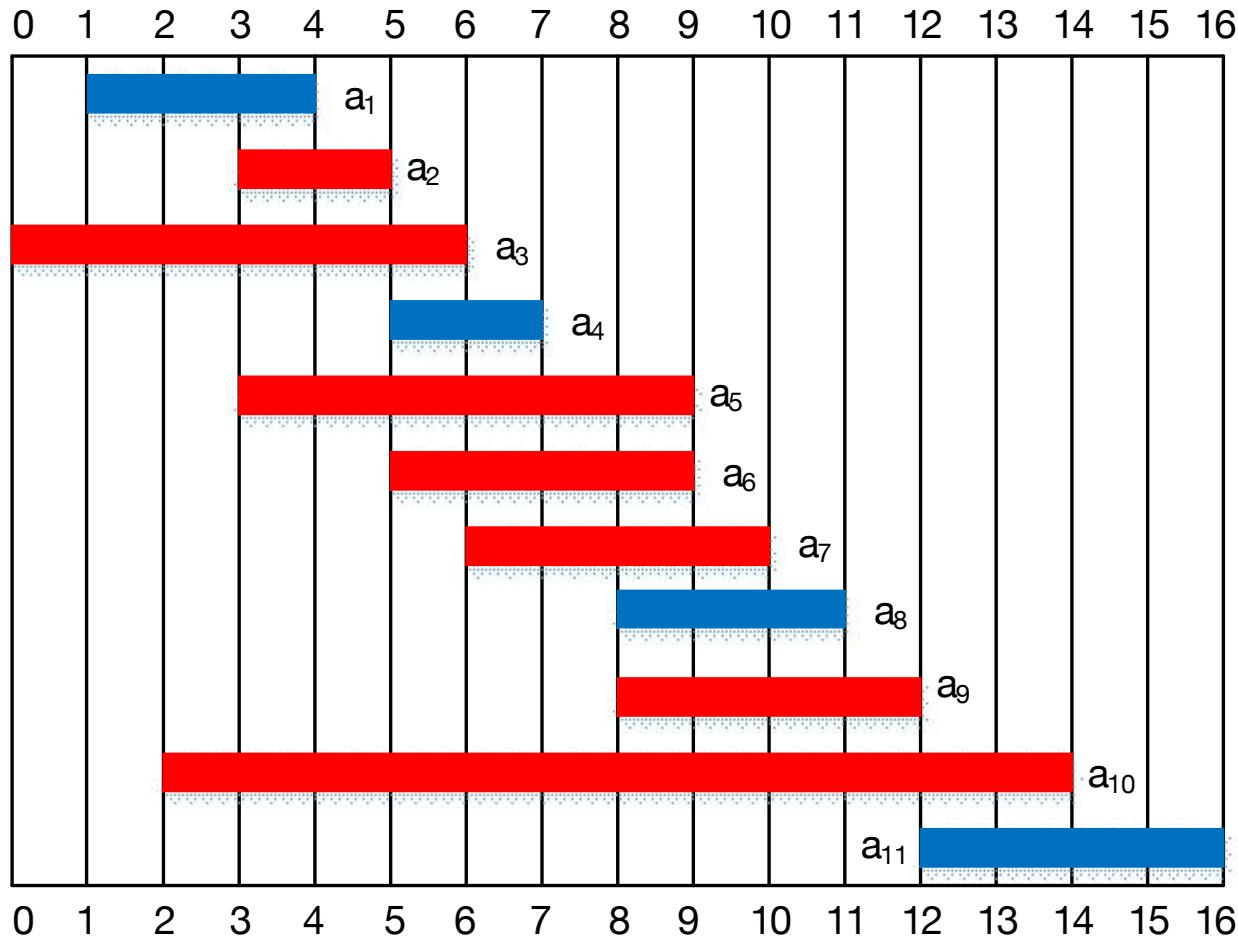


9.2 活动选择问题



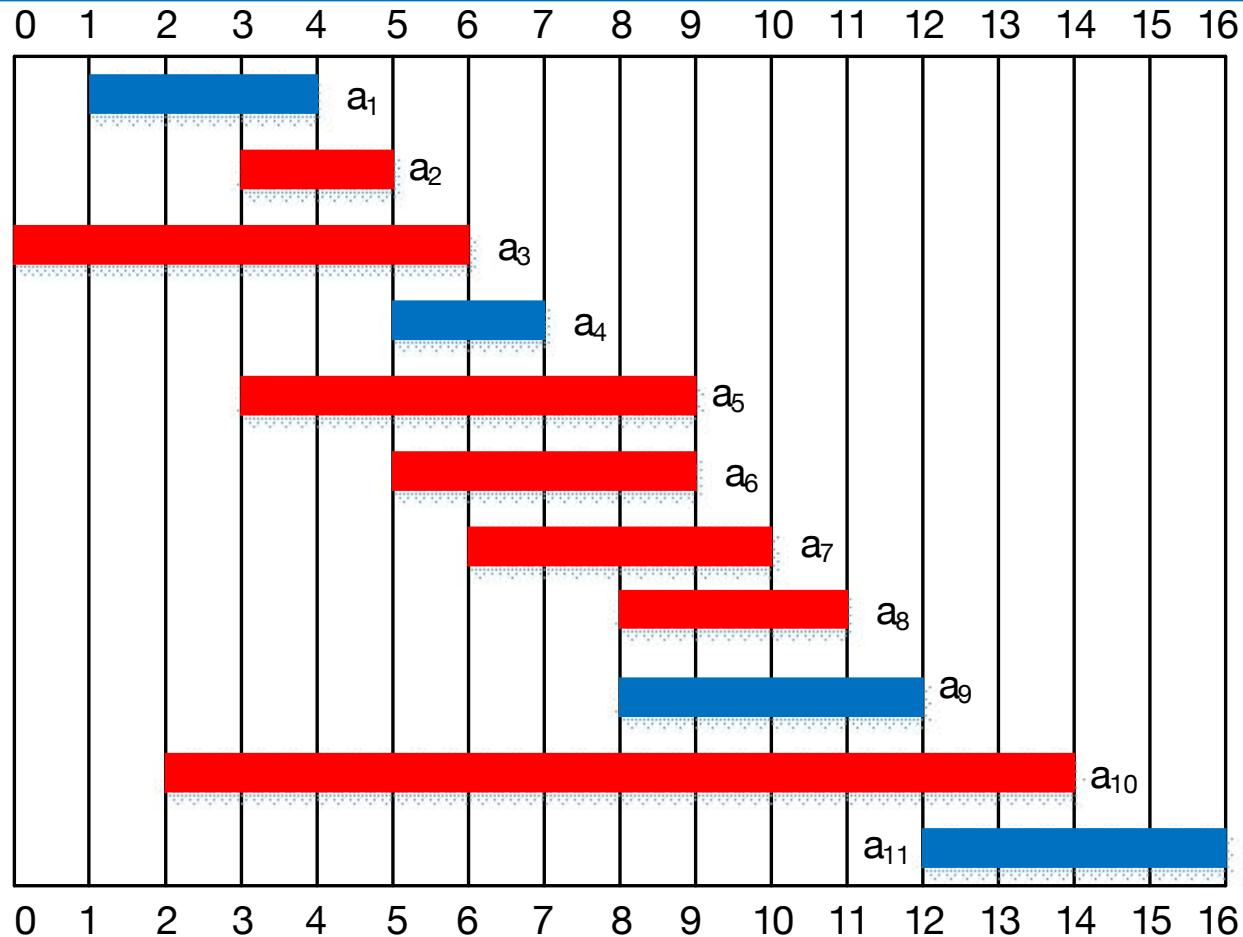


9.2 活动选择问题





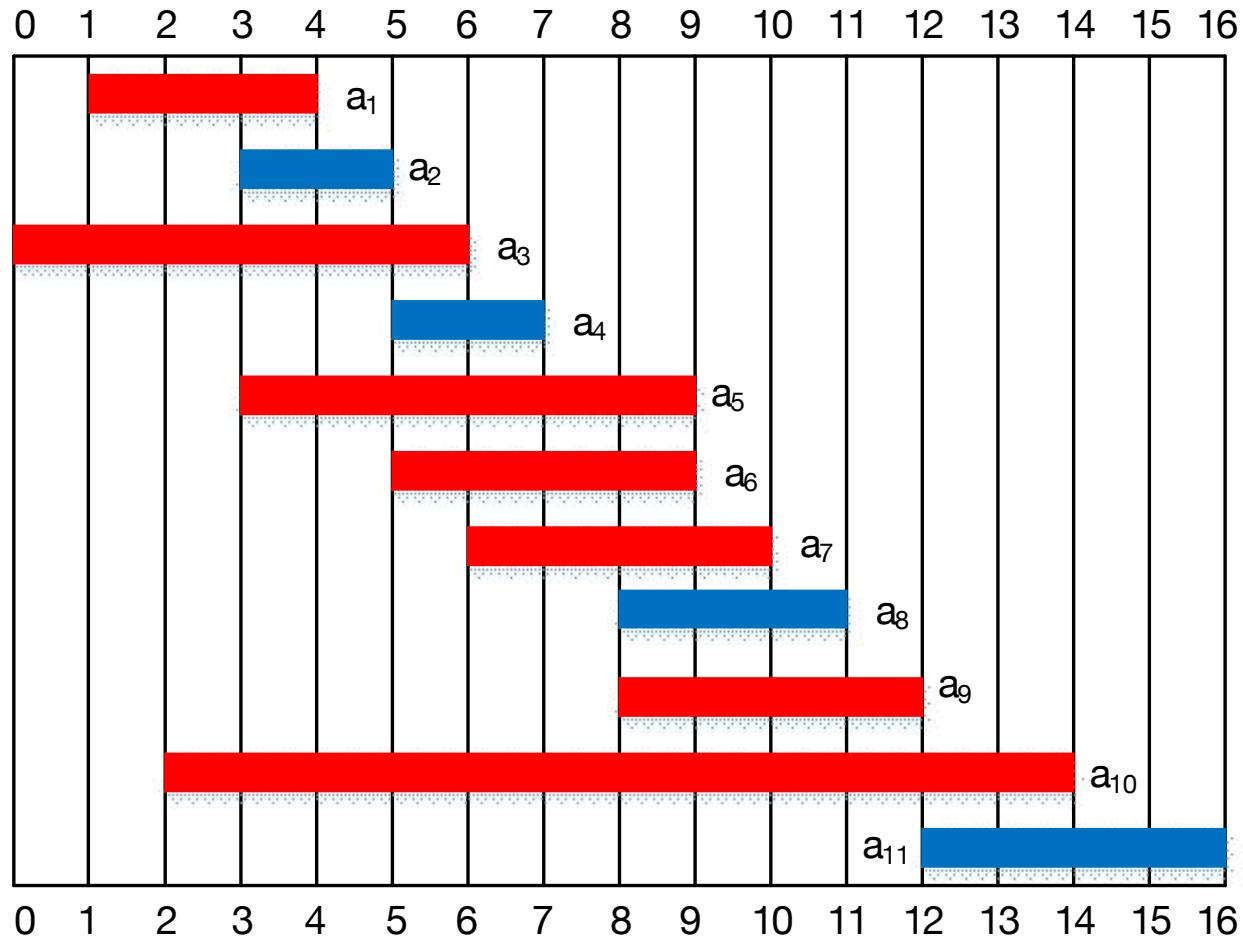
9.2 活动选择问题



| 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 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

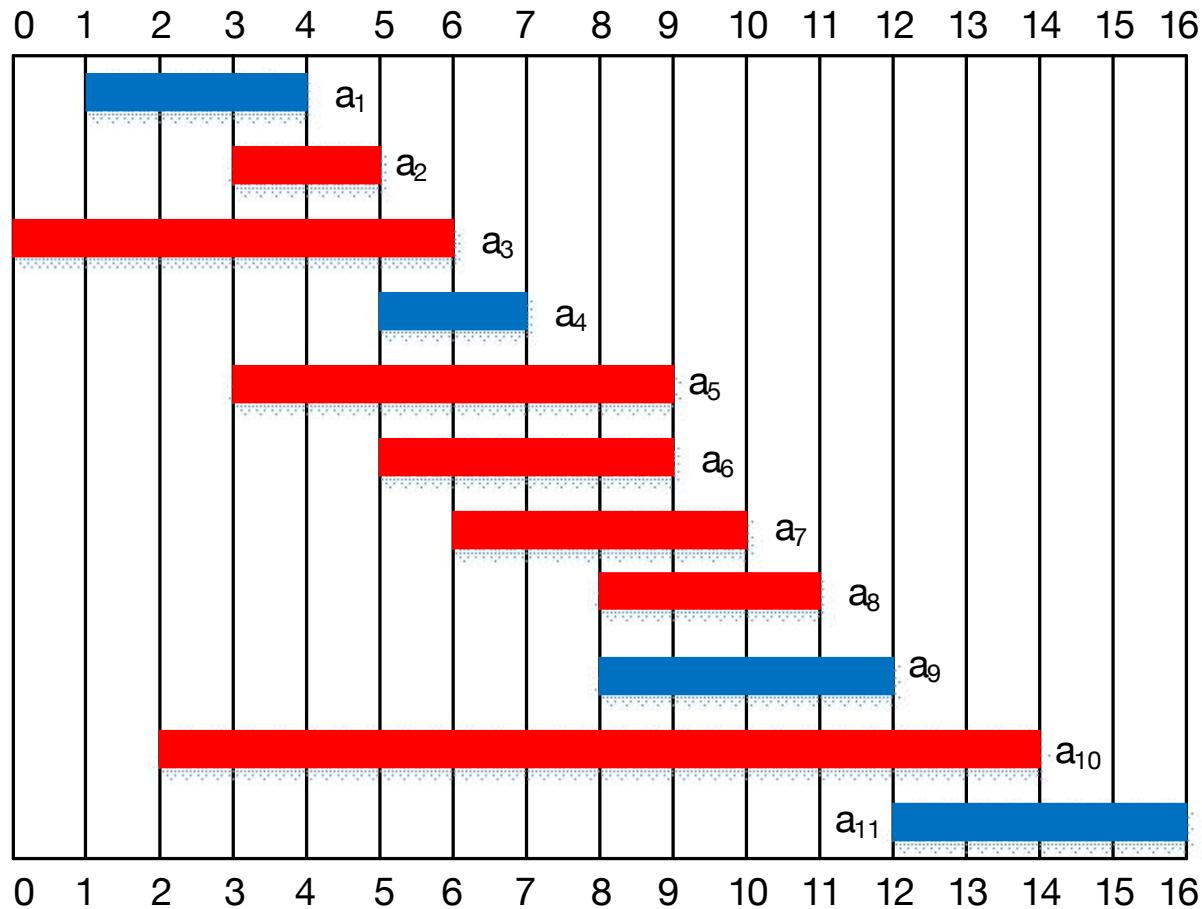


9.2 活动选择问题



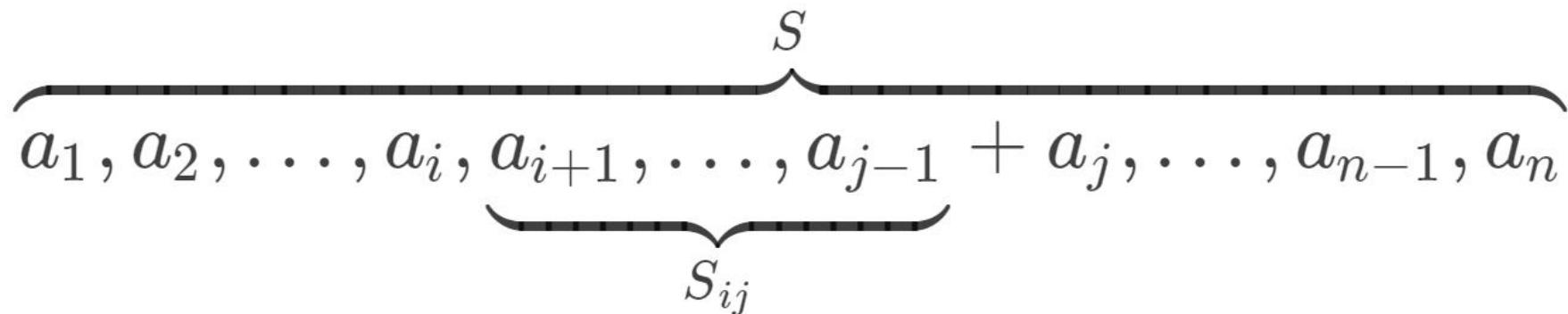


9.2 活动选择问题



| 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 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

9.2 活动选择问题-最优子结构



假设在有活动集合 S_{ij} 且其最大兼容子集为 A_{ij} , A_{ij} 之中包含活动 a_k

令 $A_{ik} = A_{ij} \cap S_{ik}$ 和 $A_{kj} = A_{ij} \cap S_{kj}$

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

S_{ij} 里面的最大活动兼容子集个数为

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$$

9.2 活动选择问题-最优子结构



最优子结构证明

通过一个活动 a_k 将问题 S_{ij} 分成两个子问题: S_{ik} 和 S_{kj}

S_{ij} 的最优解 A_{ij} 必然包含其子问题 S_{ik} 和 S_{kj} 的最优解:
 A_{ik} 和 A_{kj} 。

9.2 活动选择问题-动态规划法



递归式：

$$c[i, j] = c[i, k] + c[k, j] + 1$$

$$c[i, j] = \begin{cases} 0 & i = j - 1 \\ \max\{c[i, k] + c[k, j] + 1\} (i \leq k \leq j) & i >= j \end{cases}$$

9.2 活动选择问题-动态规划法



数据结构设计

Dyn_Prog_ActivitySelection.py

$c[i, j]$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|----|---|---|---|---|---|---|---|---|---|---|----|----|
| 0 | 0 | | | | | | | | | | | |
| 1 | | 0 | | | | | | | | | | |
| 2 | | | 0 | | | | | | | | | |
| 3 | | | | 0 | | | | | | | | |
| 4 | | | | | 0 | | | | | | | |
| 5 | | | | | | 0 | | | | | | |
| 6 | | | | | | | 0 | | | | | |
| 7 | | | | | | | | 0 | | | | |
| 8 | | | | | | | | | 0 | | | |
| 9 | | | | | | | | | | 0 | | |
| 10 | | | | | | | | | | | 0 | |
| 11 | | | | | | | | | | | | 0 |

9.2 活动选择问题-动态规划法



数据结构设计

A[i,j]

[empty, empty, empty, empty, [1], empty, [1], [1], [1, 4], [1, 4], empty, [1, 4, 8], [1, 4, 8, 11]]
[empty, empty, empty, empty, empty, empty, empty, empty, empty, [4], [4], empty, [4, 8], [4, 8, 11]]
[empty, empty, empty, empty, empty, empty, empty, empty, empty, [4], [4], empty, [4, 8], [4, 8, 11]]
[empty, empty, [7], [7, 11]]
[empty, empty, [8], [8, 11]]
[empty, empty, [8], [8, 11]]
[empty, empty, [11]]
[empty, empty, [11]]
[empty, empty, [11]]
[empty, empty, [11]]
[empty, empty, empty]
[empty, empty, empty]
[empty, empty, empty]

9.2 活动选择问题-动态规划法



代码实现

Dyn_Prog_ActivitySelection.py

```
[0, 0, 0, 0, 1, 0, 1, 1, 2, 2, 0, 3, 4]
[0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 2, 3]
[0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 2, 3]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

9.2 活动选择问题-动态规划法



代码实现

Dyn_Prog_ActivitySelection.py

```
[[], [], [], [1], [], [1], [1], [1, 4], [1, 4], [], [1, 4, 8], [1, 4, 8, 11]]  
[[], [], [], [], [], [], [4], [4], [], [4, 8], [4, 8, 11]]  
[[], [], [], [], [], [], [4], [4], [], [4, 8], [4, 8, 11]]  
[[], [], [], [], [], [], [], [7], [7, 11]]  
[[], [], [], [], [], [], [], [8], [8, 11]]  
[[], [], [], [], [], [], [], [], [11]]  
[[], [], [], [], [], [], [], [], [11]]  
[[], [], [], [], [], [], [], [], [11]]  
[[], [], [], [], [], [], [], [], [11]]  
[[], [], [], [], [], [], [], [], [11]]  
[[], [], [], [], [], [], [], [], [11]]  
[[], [], [], [], [], [], [], [], []]  
[[], [], [], [], [], [], [], [], []]
```

9.2 活动选择问题-动态规划法



代码实现

Dyn_Prog_ActivitySelection.py

```
s = [0,1,3,0,5,3,5, 6, 8, 8, 2,12,0]
f = [0,4,5,6,7,9,9,10,11,12,14,16,0]
```

```
s[0]=0
f[0]=0
s[n+1]=sys.maxsize
f[n+1]=sys.maxsize
c=[[0]*(n+2) for _ in range(n+2)]
A=[[[]for i in range(n+2)] for i in range(n+2)]
```

为什么引入虚拟活动 a_0 与 $a_{n+1}, f_0 = 0, s_{n+1} = \text{NaN}$ （无穷大）？

为什么 $c[n][n]$ 变成 $c[n+2][n+2]$ ， $A[n][n]$ 变成 $A[n+2][n+2]$ ？

作业!

9.2 活动选择问题-动态规划法



Dyn_Prog_ActivitySelection.py

作业!

算法时间复杂度分析! 写出详细分析过程。

9.2 活动选择问题-贪心算法



活动选择问题-贪心策略

输入: $S = \{1, 2, \dots, n\}$ 为 n 项活动的集合, s_i, f_i 分别为活动 i 的开始和结束时间, 活动 i 与 j 相容 $\Leftrightarrow s_i \geq f_j$ 或 $s_j \geq f_i$.
求: 最大的两两相容的活动集 A

实例

| 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 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

策略1: 排序使得 $s_1 \leq s_2 \leq \dots \leq s_n$, 从前向后挑选

策略2: 排序使得 $f_1 - s_1 \leq f_2 - s_2 \leq \dots \leq f_n - s_n$, 从前向后挑选

策略3: 排序使得 $f_1 \leq f_2 \leq \dots \leq f_n$, 从前向后挑选

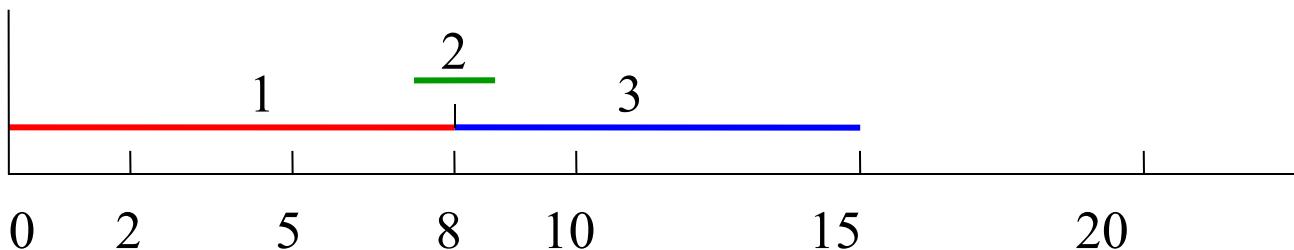
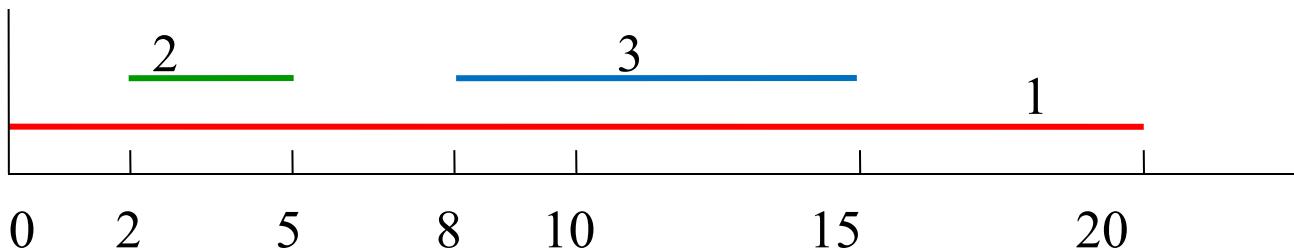
以上策略中的挑选都要注意满足相容性条件

9.2 活动选择问题-贪心算法



策略1: $S=\{1,2,3\}$, $s_1=0, f_1=20, s_2=2, f_2=5, s_3=8, f_3=15$

策略2: $S=\{1,2,3\}$, $s_1=0, f_1=8, s_2=7, f_2=9, s_3=8, f_3=15$



9.2 活动选择问题-贪心算法



贪心算法选择策略：

选择最早结束的活动。直观上认为每次选择最早结束的活动，后面的活动可选的时间就越充裕。

9.2 活动选择问题—贪心算法



我们的直觉是正确的吗？贪心选择——最早结束的活动——总是最优解的一部分吗？

定理16. 1考虑任意非空子问题 S_k ，令 a_m 是 S_k 中结束时间最早的活动，则 a_m 在 S_k 的某个最大兼容活动子集中。

证明：假设 A_k 是 S_k 的一个最大兼容活动子集，并且 a_j 为 A_k 中，结束时间最早的活动。

如果 $a_j = a_m$ ，则已得证；

如果 $a_m \neq a_j$ ，则令集合 $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ 。也就是将 A_k 中的元素 a_j 替换为 a_m 。因为 a_m 为 S_k 中结束时间最早的活动，所以 A'_k 中的活动也是相互兼容的，又因为 $|A_k| = |A'_k|$ ，所以 A'_k 也是 S_k 的最大兼容活动子集，所以得证。

9.2 活动选择问题-贪心算法



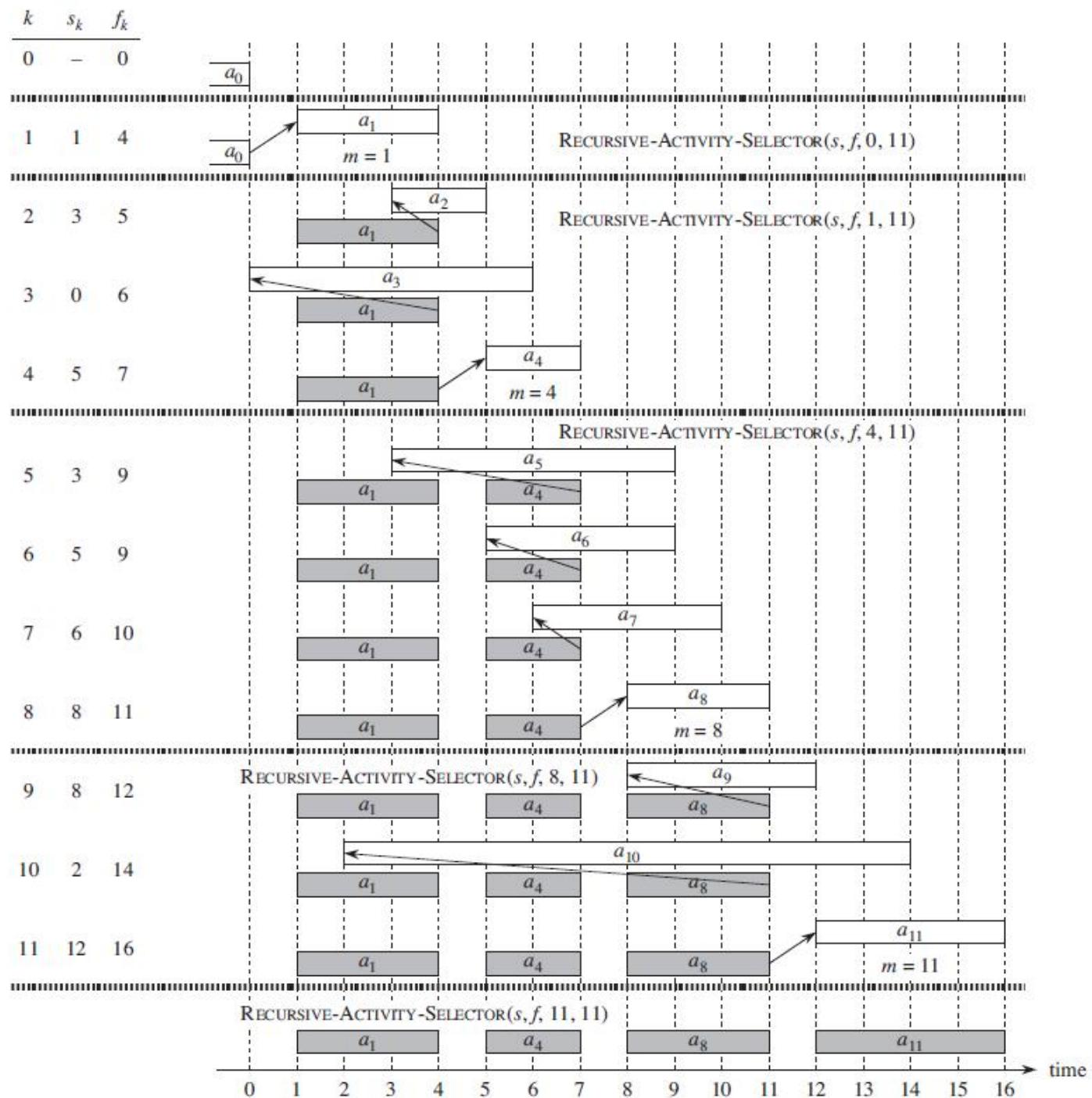
贪心算法选择策略3：选择最早结束活动，递归

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

```
1   $m = k + 1$ 
2  while  $m \leq n$  and  $s[m] < f[k]$       // find the first activity in  $S_k$  to finish
3       $m = m + 1$ 
4  if  $m \leq n$ 
5      return  $\{a_m\} \cup$  RECURSIVE-ACTIVITY-SELECTOR( $s, f, m, n$ )
6  else return  $\emptyset$ 
```

Greedy_ActivitySelection2.py

贪心算



9.2 活动选择问题-贪心算法



贪心算法选择策略3：选择最早结束活动，迭代

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $s[m] \geq f[k]$ 
6           $A = A \cup \{a_m\}$ 
7           $k = m$ 
8  return  $A$ 
```

Greedy_ActivitySelection3.py

$$f_k = \max\{f_i : a_i \in A\}$$

9.2 活动选择问题-贪心算法



最优解证明(算法正确性证明), 自学

关于贪心算法的正确性证明.docx

052贪心法的正确性证明.pdf

一个凭借直觉的算法——贪心算法及其应用：活动选择问题.pdf

9.2 活动选择问题



动态规划法和贪心算法 求解活动选择问题总结

动态规划算法

- 1、动态规划算法通常以自底向上的方式解各子问题，全局最优解中一定包含某个局部最优解，但不一定包含前一个局部最优解，因此需要记录之前的所有最优解
- 2、动态规划的关键是状态转移方程，即如何由以求出的局部最优解来推导全局最优解
- 3、边界条件：即最简单的，可以直接得出的局部最优解。

贪心算法

- 1、而贪心算法则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，每做一次贪心选择就将所求问题简化为规模更小的子问题。
- 2、贪心算法中，作出的每步贪心决策都无法改变，因为贪心策略是由上一步的最优解推导下一步的最优解，而上一步之前的最优解则不作保留。



提纲

9.1 概述

9.2 活动选择问题

9.3 贪心算法原理

9.4 哈夫曼编码

9.5 (分数) 背包问题

9.6 多机调度问题

9.7 贪心算法与人工智能算法

9.3 贪心算法原理



设计贪心算法经过下面几个步骤：

- ①确定问题的最优子结构
- ②基于问题的最优子结构设计一个递归算法
- ③证明我们做出的贪心选择，只剩下一个子问题
- ④证明贪心选择总是安全的
- ⑤递归算法实现贪心策略
- ⑥将贪心算法转化为迭代算法

9.3 贪心算法原理



贪心算法的设计步骤简化：

- ① 将最优化问题简化为这样的形式：最初一个选择以后，只剩下一个子问题需要求解！
- ② 证明在做出贪心选择以后，原问题总是存在最优解，即贪心选择总是安全的！
- ③ 证明在做出贪心选择以后，剩下的子问题满足性质：其最优解与做出选择的组合在一起得到原问题的最优解，即最优子结构。

9.3 贪心算法原理



贪心算法的设计步骤简化：

- ① 将最优化问题简化为这样的形式：最初一个选择以后，只剩下一个子问题需要求解！
- ② 证明在做出贪心选择以后，原问题总是存在最优解，即贪心选择总是安全的！
- ③ 证明在做出贪心选择以后，剩下的子问题满足性质：其最优解与做出选择的组合在一起得到原问题的最优解，即最优子结构。



9.3 贪心算法原理

贪心算法的求解过程：

- ① 候选集合Candidate: 为问题的可能解，即问题的最终解均来自于该候选集合；
- ② 解集合Answer: 为最终求解的完整解；
- ③ 解决函数Solution: 检验解集合是否已经构成该问题的完整解；
- ④ 选择函数Select: 贪心策略；
- ⑤ 可行函数Feasible: 检验解集合中新加入的候选对象是否可行。

9.3 贪心算法原理



贪心算法的求解过程代码框架：

```
Greedy(Candidate)
{
    Answer={};
    while(!Solution(Answer))//未构成完整解
    {
        object=Select(Candidate);
        if(Feasible(Answer,object)==true)
        {
            Answer=Answer+{object};
            Candidate=Candidate-{object};
        }
    }
    return Answer;
}
```



9.3 贪心算法原理

贪心算法的两个重要的性质：

- 贪心选择性质
- 最优子结构性质



9.3 贪心算法原理

贪心 vs 动态规划

➤ 01背包问题

一个正在抢劫商店的小偷发现了 n 个商品，第 i 个商品价值 v_i 元，重 w_i 磅， v_i 和 w_i 都是整数。小偷希望拿走价值尽可能高的商品，但是背包最做只能容纳 W 磅。他应该拿走哪些商品呢？

➤ 分数背包问题

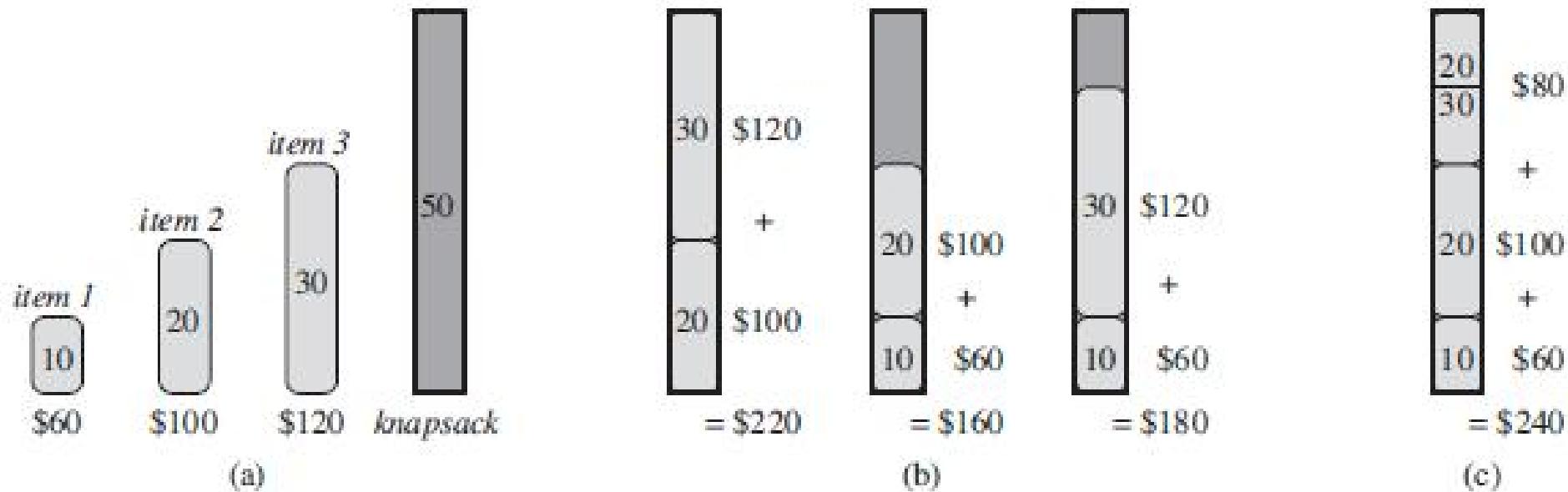
设定与0-1背包一样，但是对每个商品，小偷可以只拿走其一部分，而不是只能做出二元(0-1)选择。可以把0-1背包问题中的商品看做品质不一大小不同的金块，而分数背包问题中的商品更像是金砂。



9.3 贪心算法原理

贪心 vs 动态规划

01背包问题/分数背包问题



一个实例，说明贪心策略对 0-1 背包问题无效。(a) 小偷必须选择所示三个商品的一个子集，总重量不超过 50 磅。(b) 最优子集由商品 2 和商品 3 组成。虽然商品 1 有最大的每磅价值，但包含它的任何解都是次优的。(c) 对于分数背包问题，按每磅价值降序拿走商品会生成一个最优解



提纲

9.1 概述

9.2 活动选择问题

9.3 贪心算法原理

9.4 哈夫曼编码

9.5 (分数) 背包问题

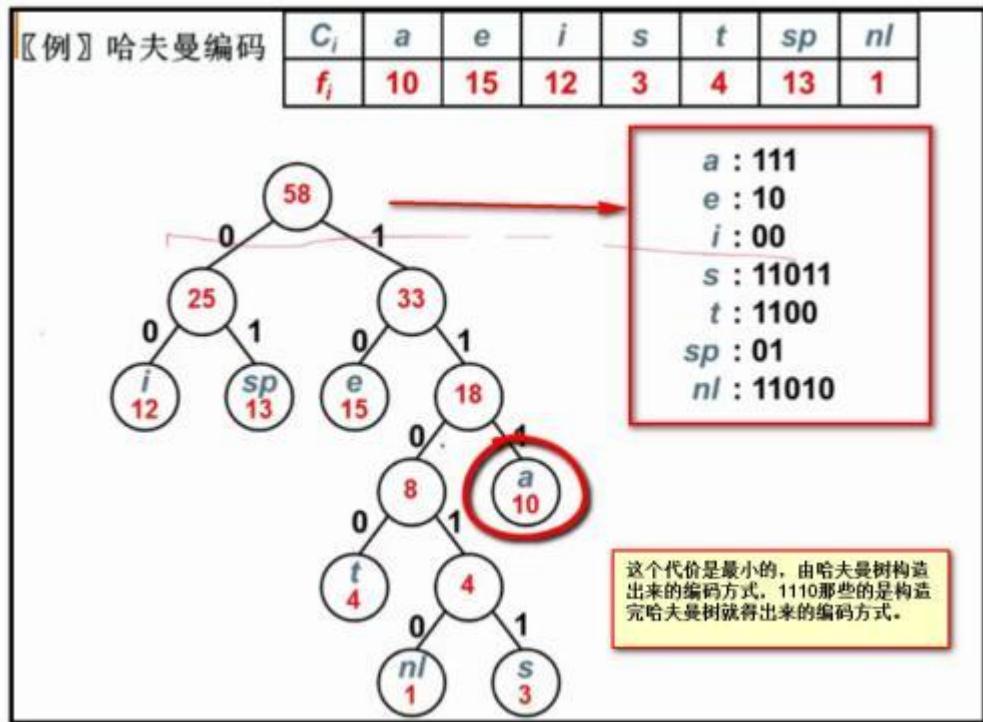
9.6 多机调度问题

9.7 贪心算法与人工智能算法



9.4 哈夫曼编码

一种字符编码方式，常用于数据文件压缩。压缩率通常在20%~90%。



采取可变长编码方式，对文件中出现次数多的字符采取比较短的编码，对于出现次数少的字符采取比较长的编码，可以有效地减小总的编码长度。



9.4 哈夫曼编码

例子

| | a | b | c | d | e | f |
|--------------------------|-----|-----|-----|-----|------|------|
| Frequency (in thousands) | 45 | 13 | 12 | 16 | 9 | 5 |
| Fixed-length codeword | 000 | 001 | 010 | 011 | 100 | 101 |
| Variable-length codeword | 0 | 101 | 100 | 111 | 1101 | 1100 |

9.4 哈夫曼编码



前缀码

设 $Q = \{a_1, a_2, \dots, a_m\}$ 是一个 0~1 序列集合。如果 Q 中没有一个序列是另一个序列的前缀，则称 Q 为前缀码。

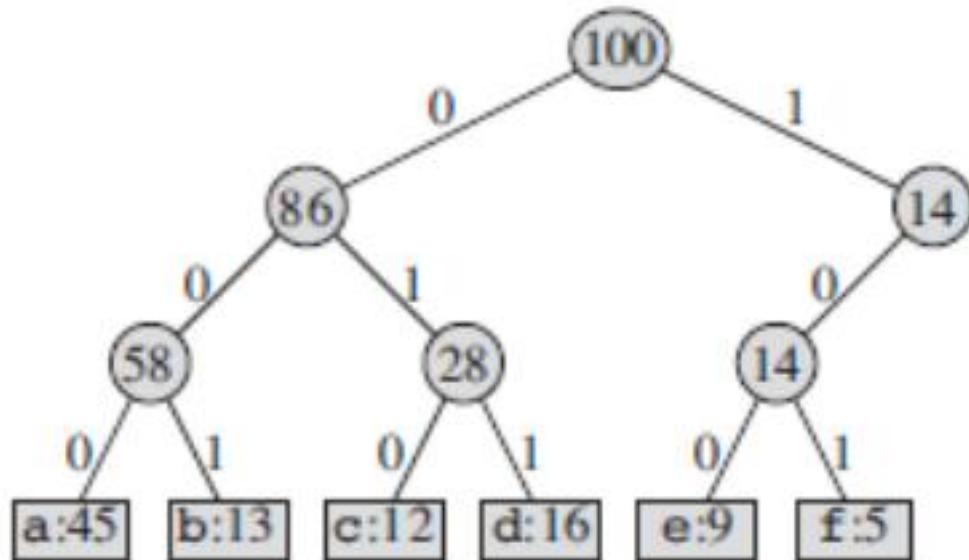
是指对字符集进行编码时，要求字符集中任一字符的编码都不是其它字符的编码的前缀。

前缀码的作用是简化解码的过程。由于没有码字是其他码字的前缀，编码文件的开始码字是无歧义的，我们可以简单地识别出开始码字，将其转换回原字符，然后对编码文件剩余部分重复这种解码过程。

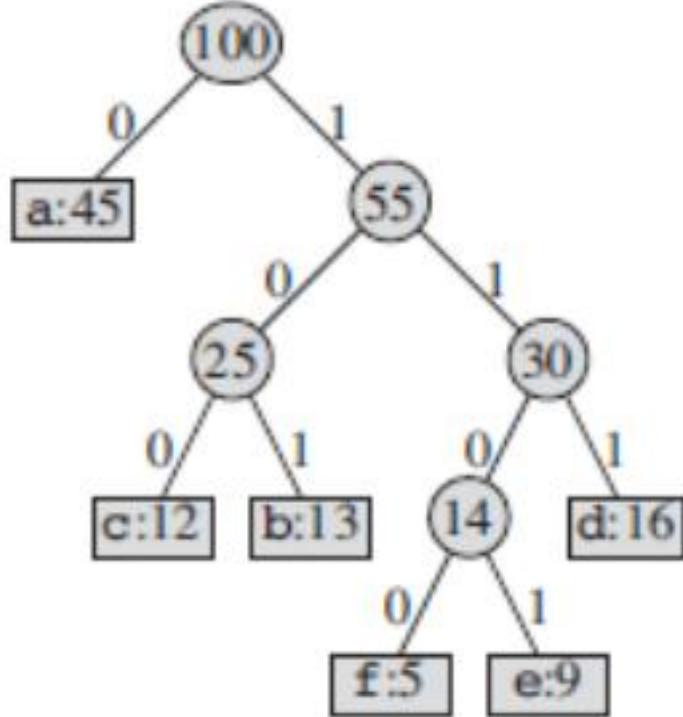


9.4 哈夫曼编码

编码的解码过程：



(a)



(b)



9.4 哈夫曼编码

计算编码文件所需要二进制位

给定一棵对应前缀码的树 T ，我们可以容易地计算出编码一个文件需要多少个二进制位。对于字母表 C 中的每个字符 c ，令属性 $c.\ freq$ 表示 c 在文件中出现的频率，令 $d_T(c)$ 表示 c 的叶结点在树中的深度。注意， $d_T(c)$ 也是字符 c 的码字的长度。则编码文件需要

$$B(T) = \sum_{c \in C} c.\ freq \cdot d_T(c) \quad (16.4)$$

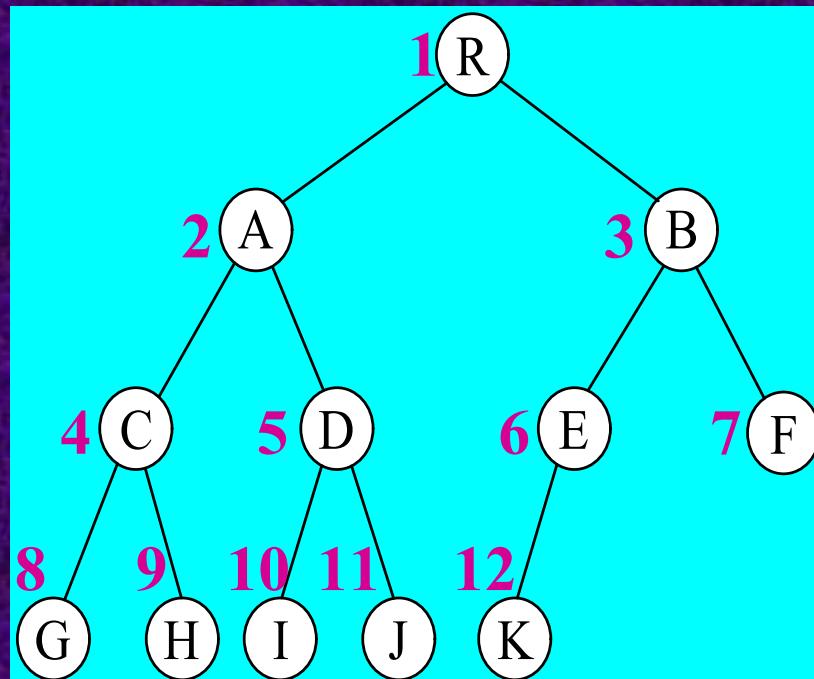
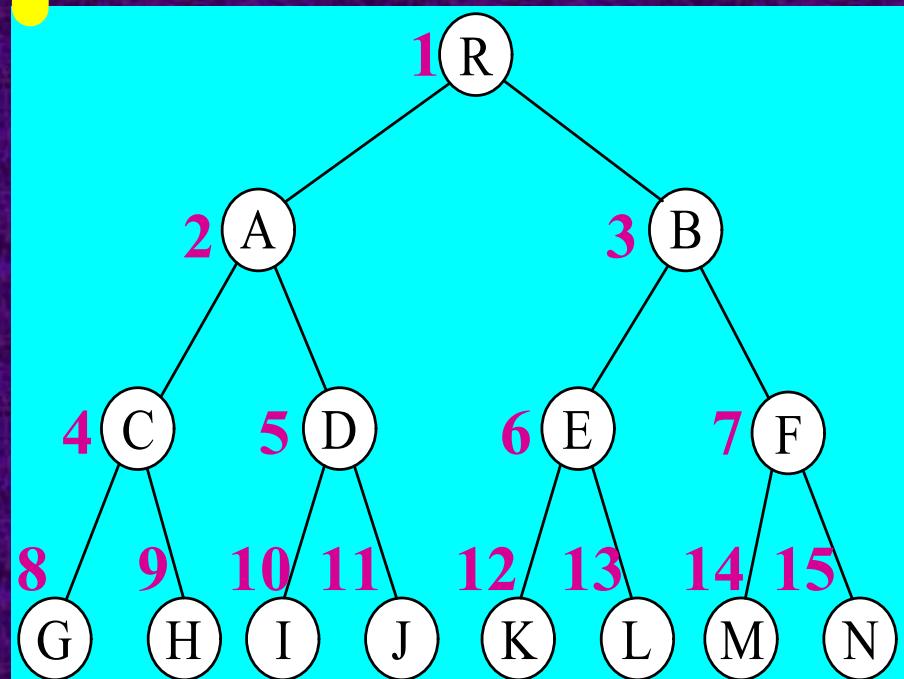
个二进制位，我们将 $B(T)$ 定义为 T 的代价。

■ 满二叉树

二叉树的所有分支结点都有左子树和右子树，并且所有叶子结点都在二叉树的最下一层；

■ 完全二叉树

深度为h的完全二叉树：前 $h-1$ 层为满二叉树，最后一层的结点必须从左向右连续出现。





9.4 哈夫曼编码

哈夫曼树（最优二叉树）

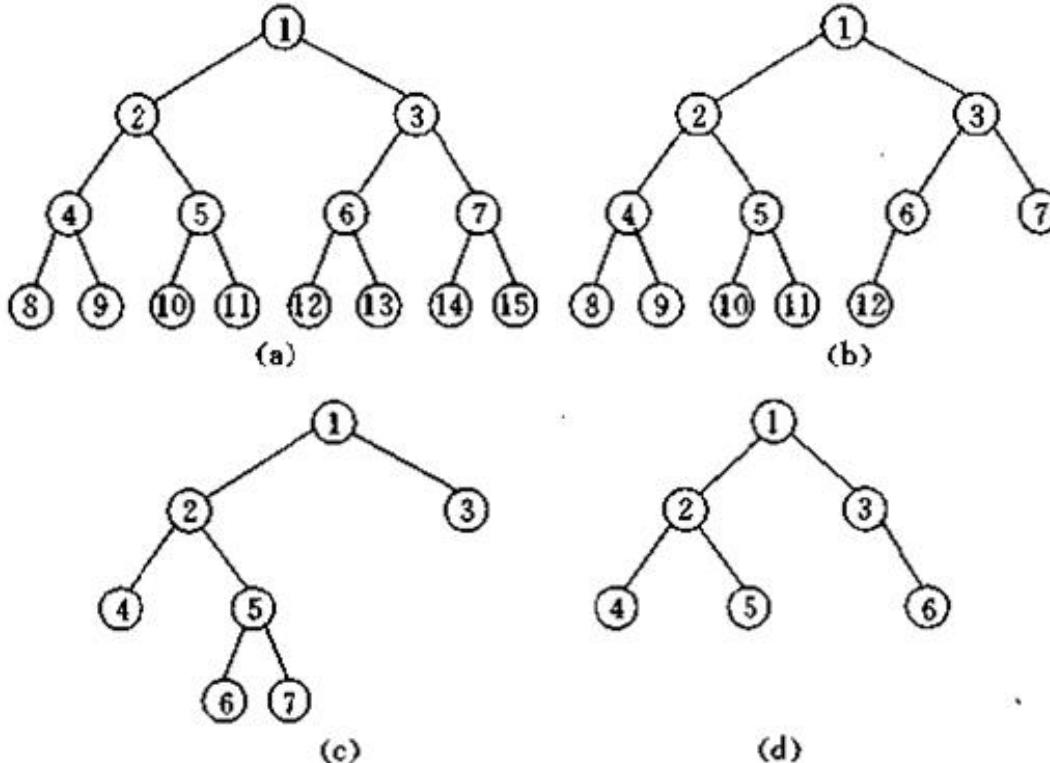


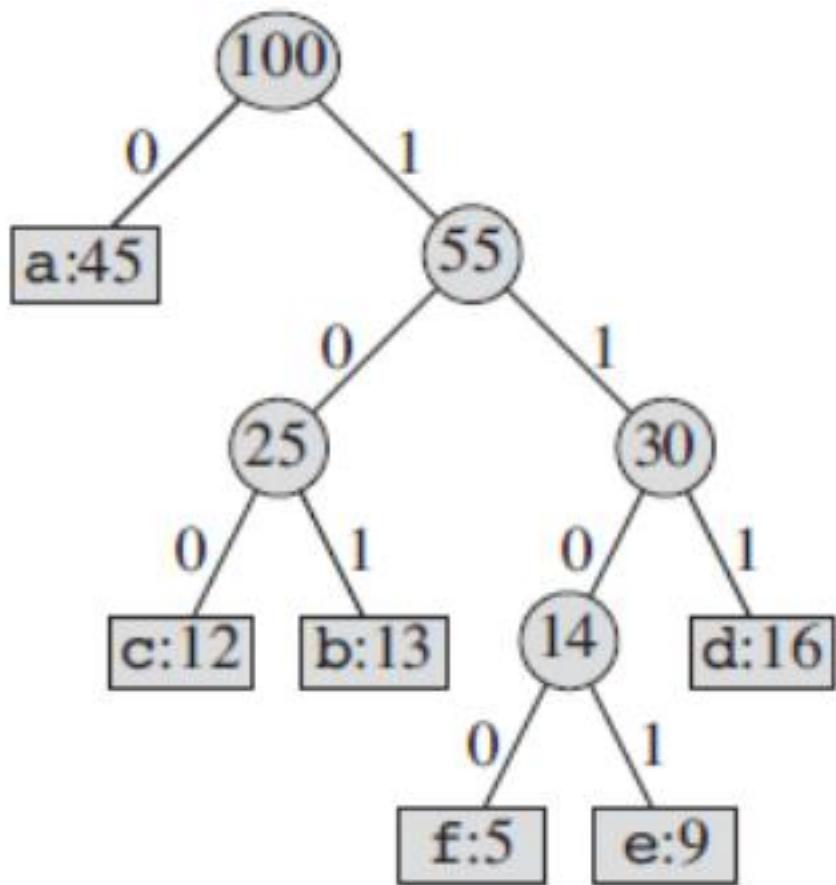
图 6.4 特殊形态的二叉树

(a) 满二叉树; (b) 完全二叉树; (c) 和 (d) 非完全二叉树。



9.4 哈夫曼编码

哈夫曼树的应用：哈夫曼编码





9.4 哈夫曼编码

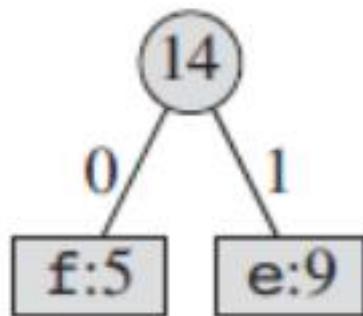
构造哈夫曼树算法的执行过程(贪心策略)

(a)

| | | | | | |
|-----|-----|------|------|------|------|
| f:5 | e:9 | c:12 | b:13 | d:16 | a:45 |
|-----|-----|------|------|------|------|

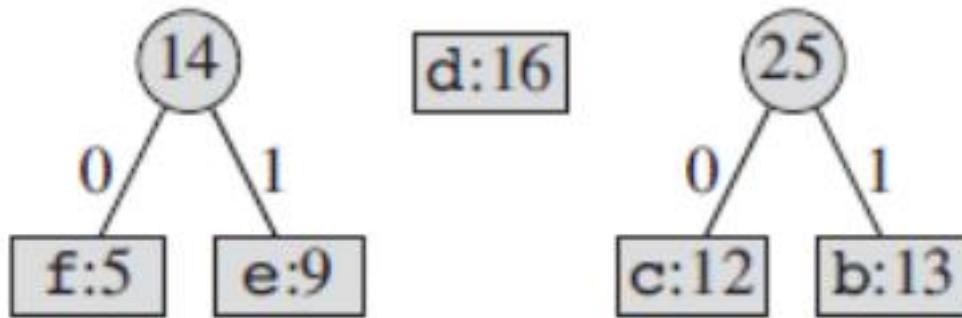
(b)

| | | | | |
|------|------|--|------|------|
| c:12 | b:13 | | d:16 | a:45 |
|------|------|--|------|------|



(c)

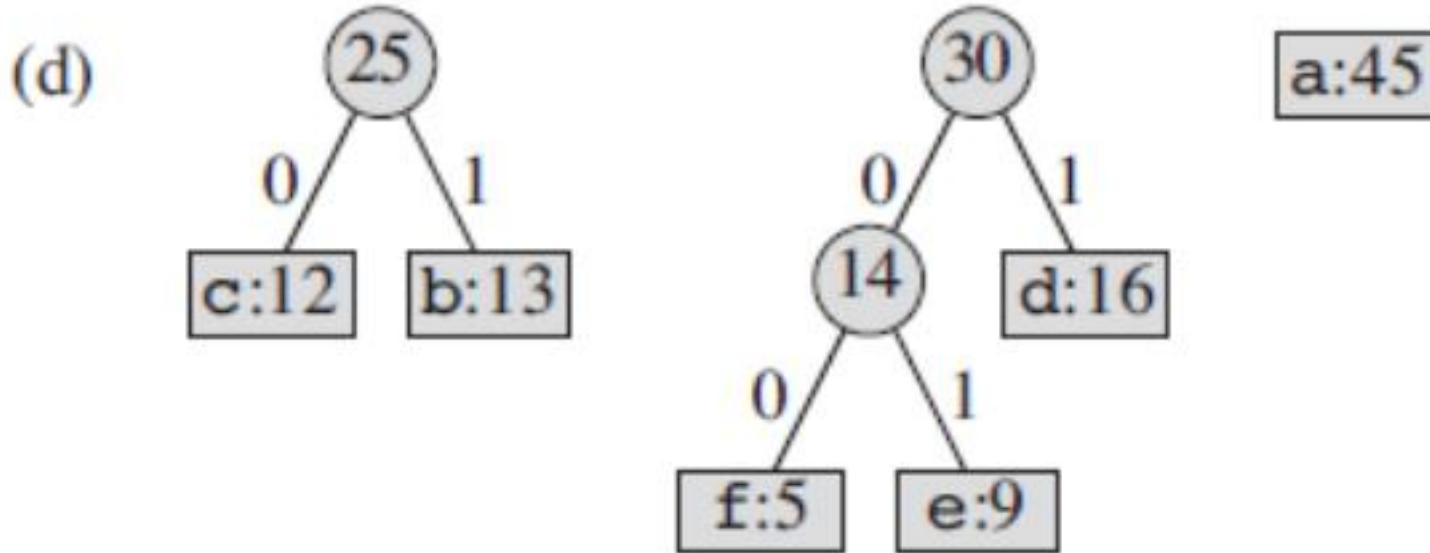
| | | | | | | |
|----|--|------|--|----|--|------|
| 14 | | d:16 | | 25 | | a:45 |
|----|--|------|--|----|--|------|





9.4 哈夫曼编码

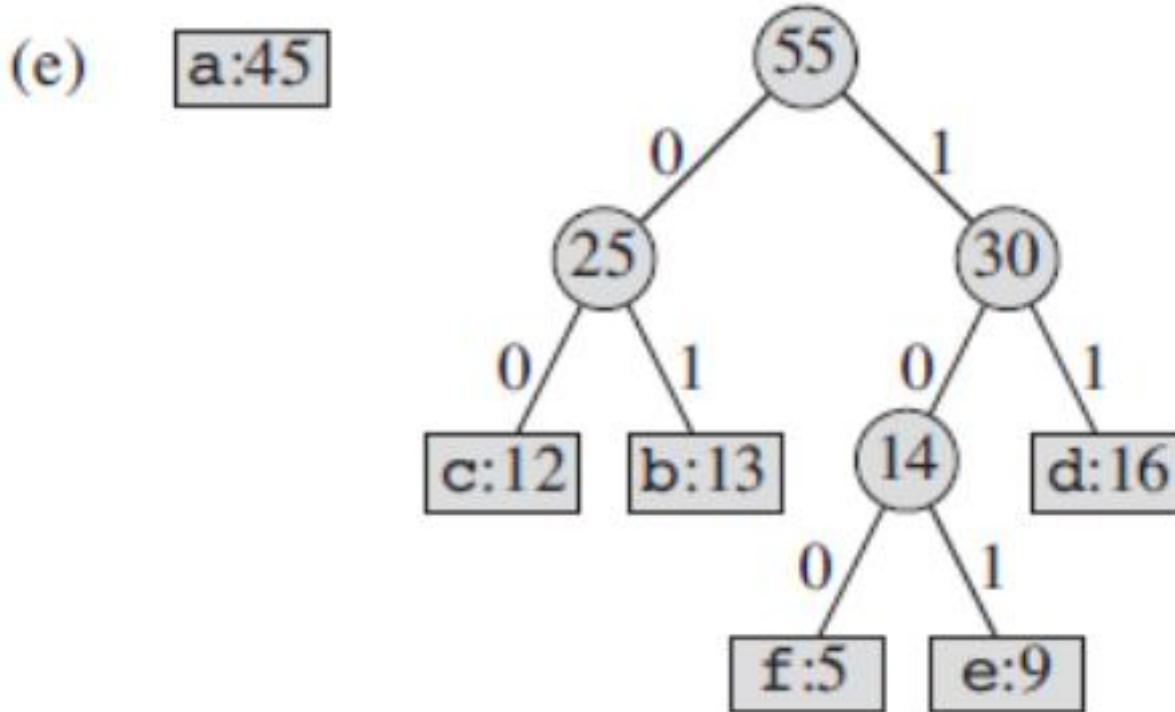
构造哈夫曼树算法的执行过程(贪心策略)





9.4 哈夫曼编码

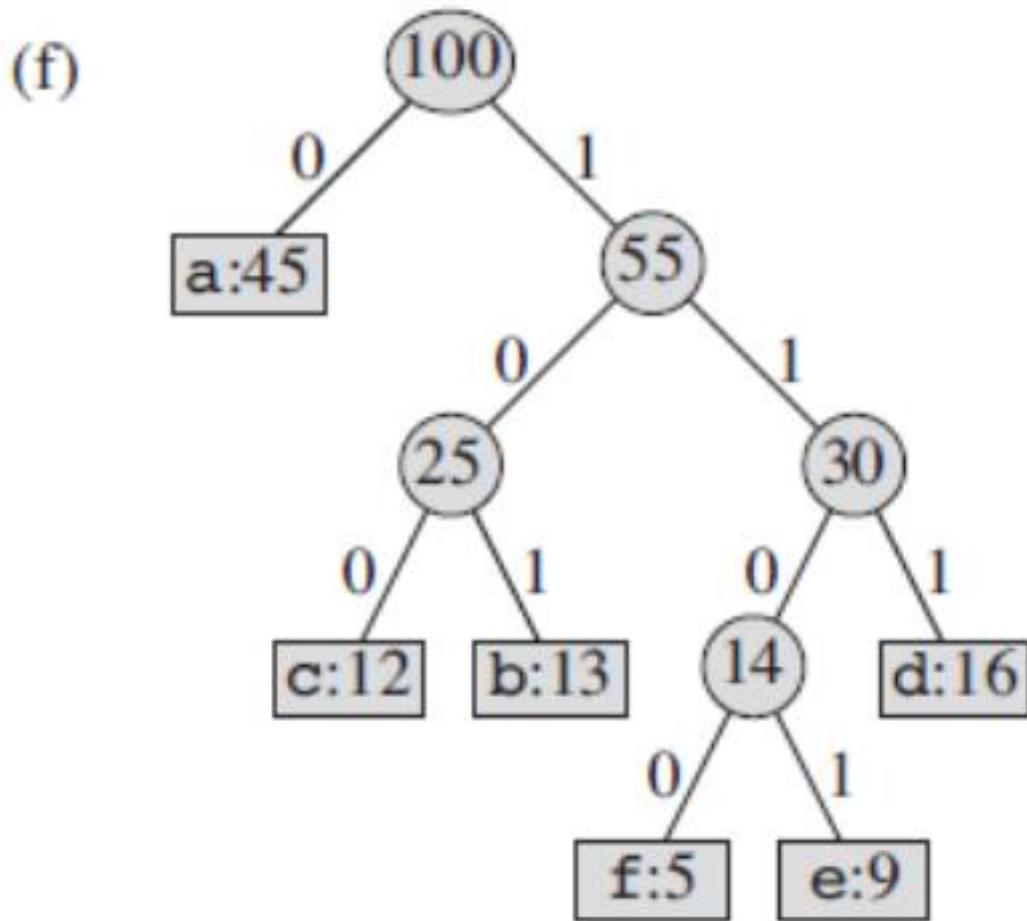
构造哈夫曼树算法的执行过程(贪心策略)





9.4 哈夫曼编码

构造哈夫曼树算法的执行过程(贪心策略)



堆的定义

- n 个元素的序列 $\{k_1, k_2, \dots, k_n\}$ 当且仅当满足如下关系时，称之为堆（heap）。若满足条件（1）则称大根堆（或最大堆）；若满足条件（2）则称小根堆（或最小堆）。

$$(1) \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad \text{或} \quad (2) \begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad (i = 1, 2, \dots, \lfloor n/2 \rfloor)$$

- 若将和此序列对应的一维数组（即以一维数组作此序列的存储结构）看成是一个完全二叉树，则堆实质上是满足如下性质的完全二叉树：树中任一非叶结点的关键字均不大于（或不小于）其左右孩子（若存在）结点的关键字。



9.4 哈夫曼编码

优先队列

定义：优先队列中元素出队列的顺序由元素的优先级决定。

从优先队列中删除元素是根据优先权高或低的次序，而不是元素进入队列的次序。

可以利用堆数据结构来高效地实现优先队列。

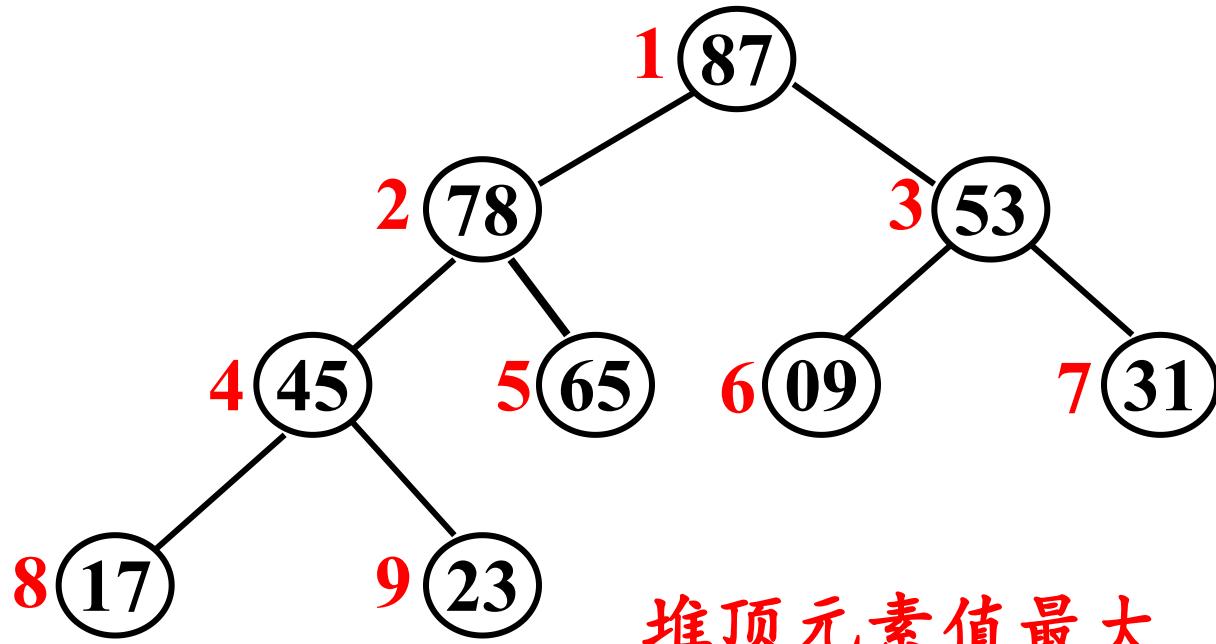
类型：最小优先队列（最小堆），最大优先队列（最大堆）



9.4 哈夫曼编码

最大优先队列 (最大堆)

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|----|----|----|----|----|----|----|----|
| 87 | 78 | 53 | 45 | 65 | 09 | 31 | 17 | 23 |

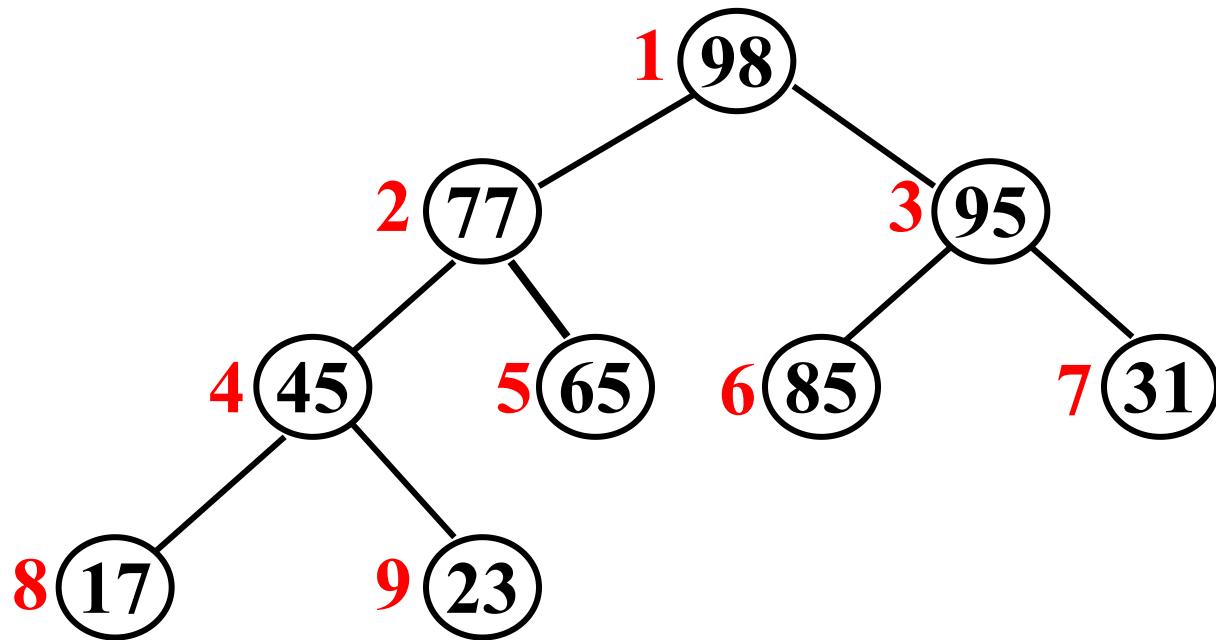




9.4 哈夫曼编码

练习：最大优先队列

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|----|----|----|----|----|----|----|----|
| 98 | 77 | 95 | 45 | 65 | 85 | 31 | 17 | 23 |



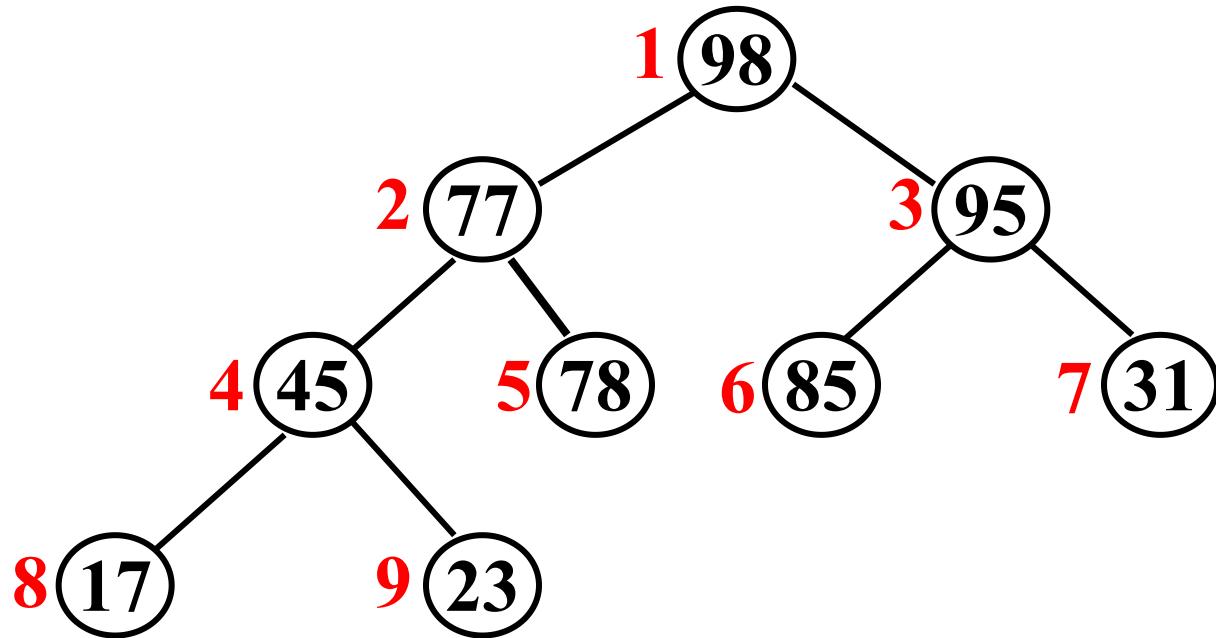
9.4 哈夫曼编码



练习：最大优先队列

1 2 3 4 5 6 7 8 9

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 98 | 77 | 95 | 45 | 78 | 85 | 31 | 17 | 23 |
|----|----|----|----|----|----|----|----|----|

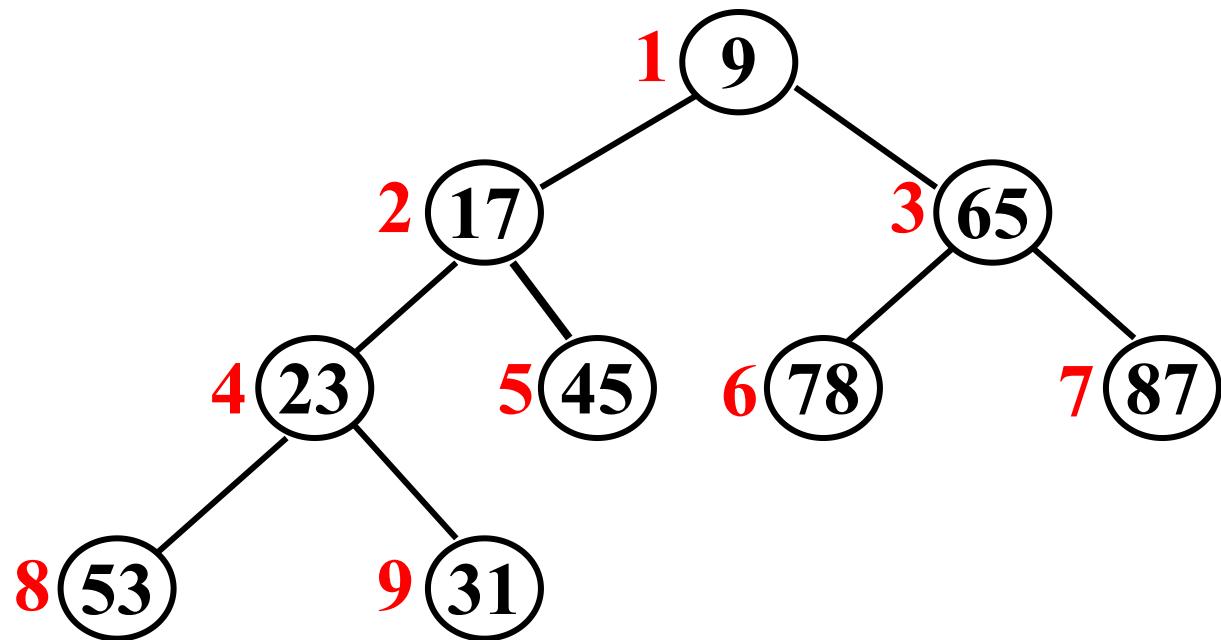


9.4 哈夫曼编码



最小优先队列（最小堆）

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|----|----|----|----|----|----|----|----|
| 9 | 17 | 65 | 23 | 45 | 78 | 87 | 53 | 31 |



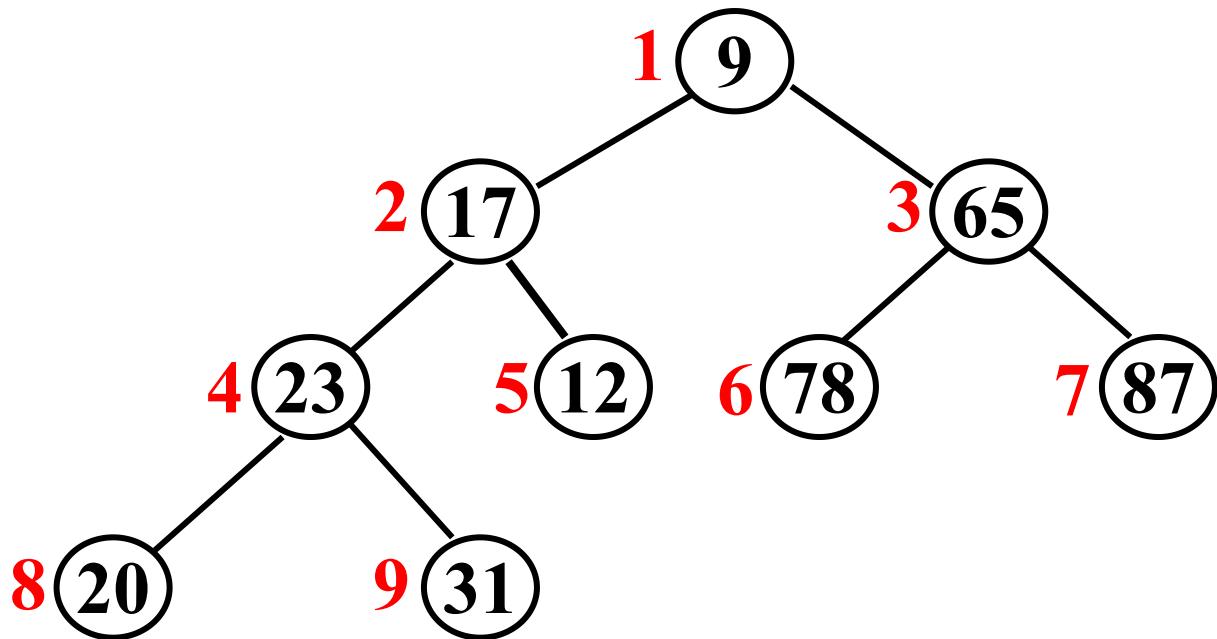
堆顶元素值最小

9.4 哈夫曼编码



练习：最小优先队列

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|----|----|----|----|----|----|----|----|
| 9 | 17 | 65 | 23 | 12 | 78 | 87 | 20 | 31 |

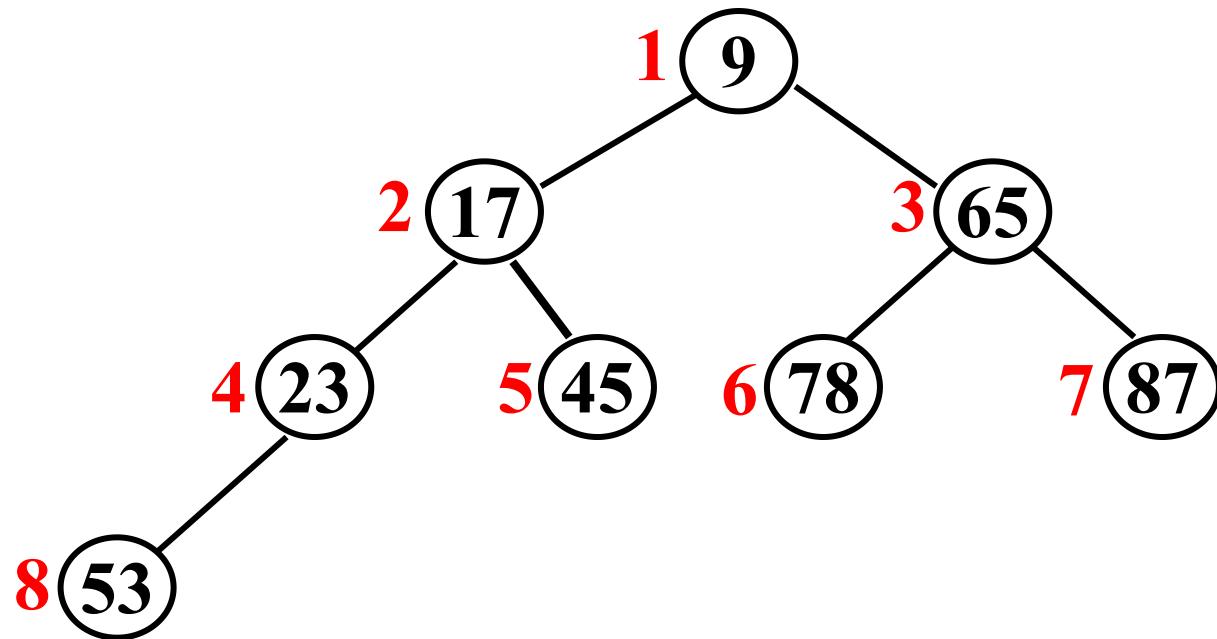


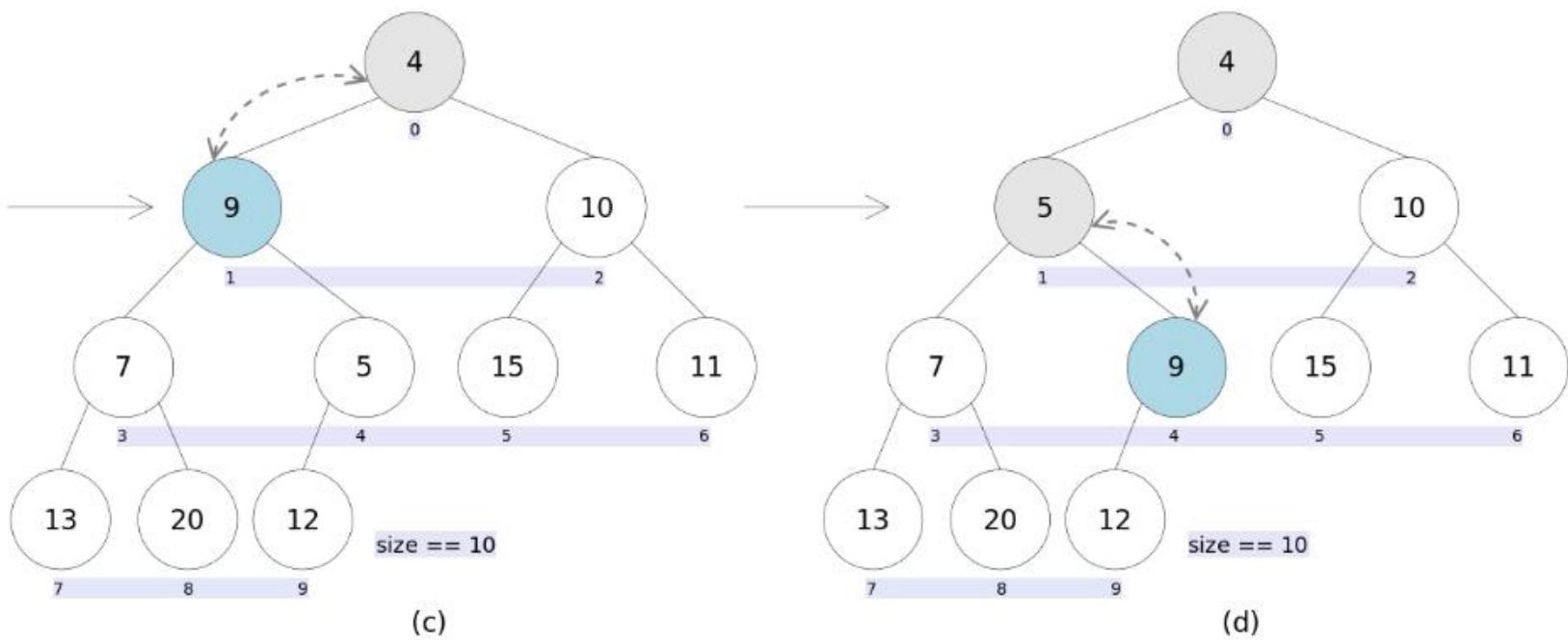
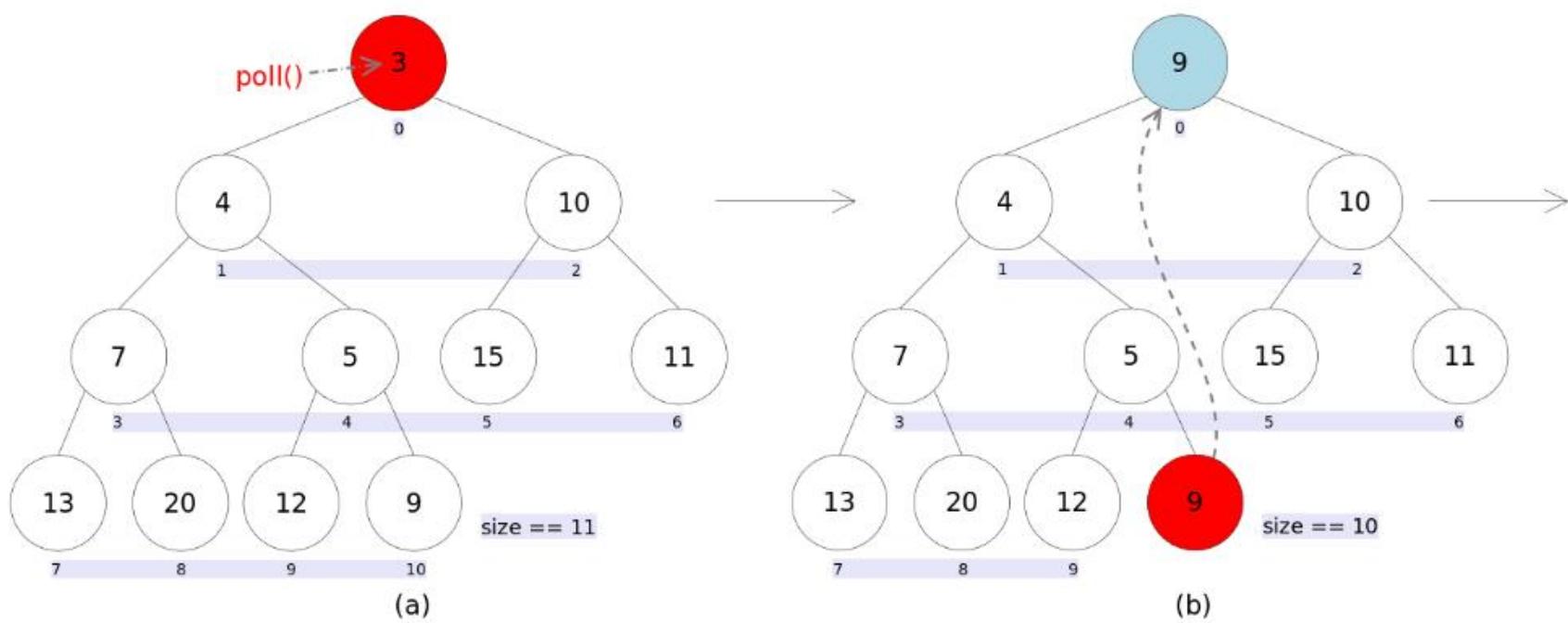


9.4 哈夫曼编码

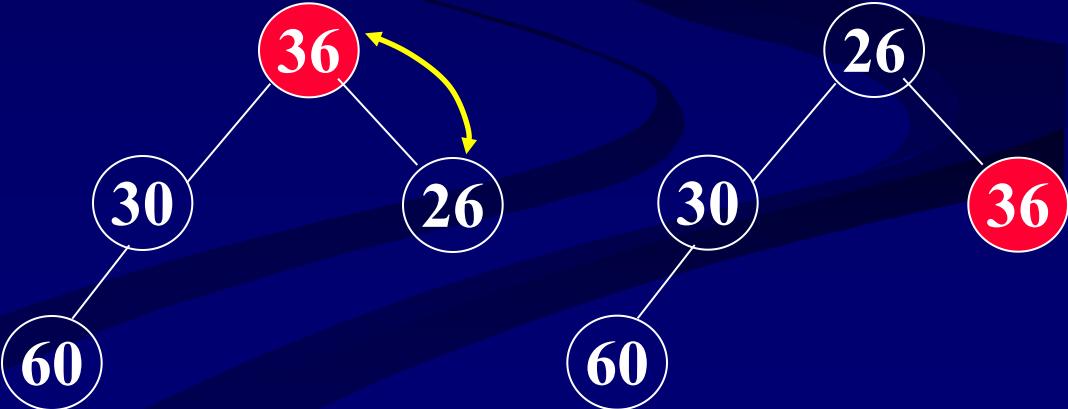
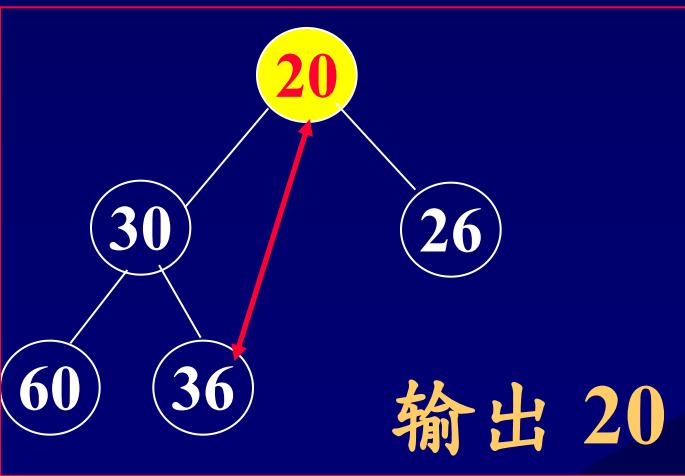
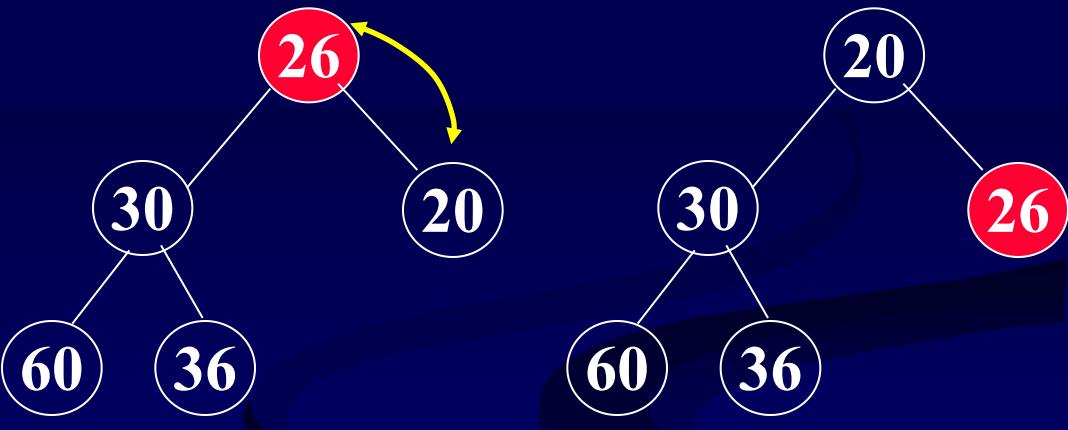
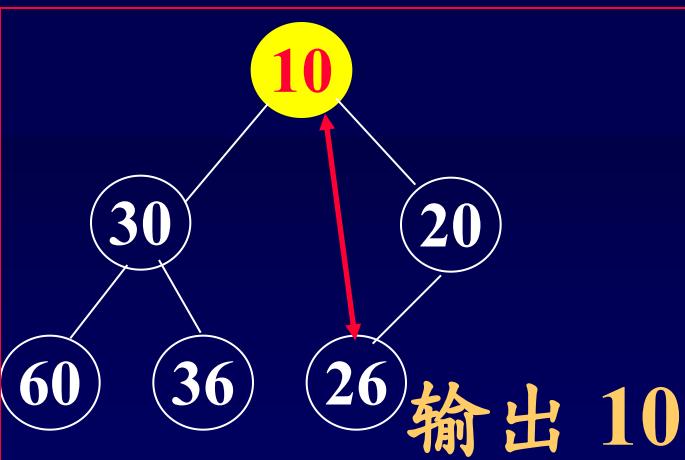
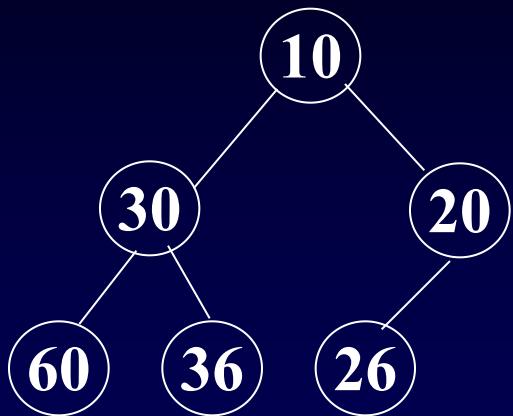
练习：最小优先队列练习

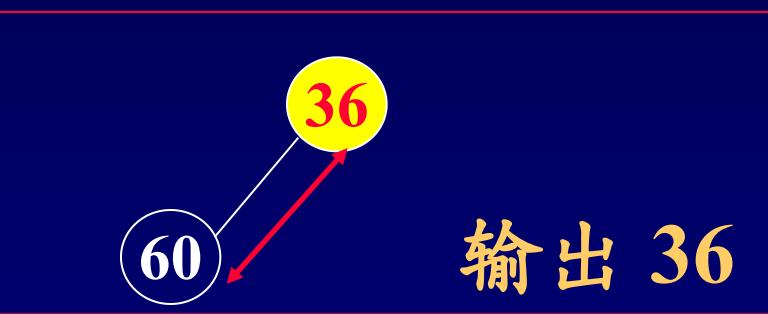
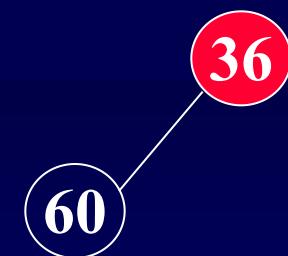
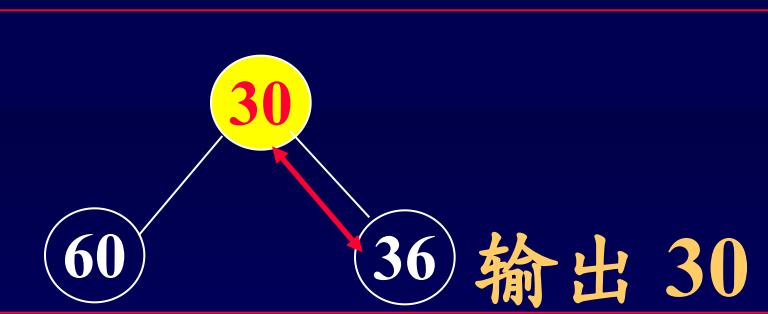
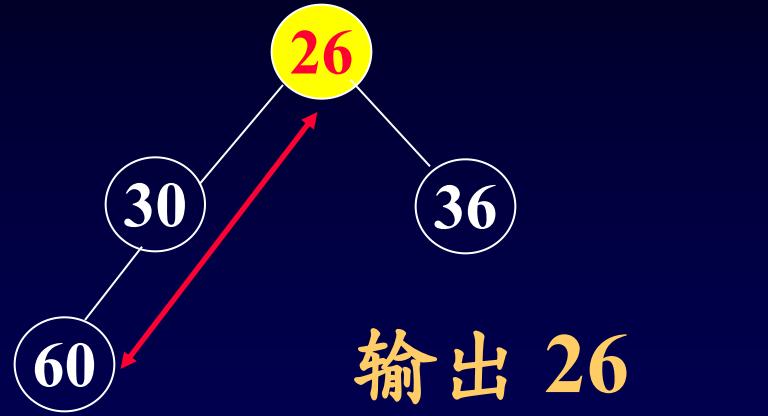
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|----|----|----|----|----|----|----|
| 9 | 17 | 65 | 23 | 45 | 78 | 87 | 53 |





最小优先队列输出序列





递增输出序列为： 10 20 26 30 36 60



9.4 哈夫曼编码

哈夫曼编码树实现伪代码

HUFFMAN(C)

```
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$     // return the root of the tree
```



9.4 哈夫曼编码

Python标准库模块之heapq

- 创建堆
- 访问堆内容
- 获取堆最大或最小值

9.4哈夫曼编码—heapq: 创建堆



```
1 import heapq
2
3 # 第一种
4 """
5 函数定义:
6 heapq.heappush(heap, item)
7     - Push the value item onto the heap, maintaining the heap invariant.
8 heapq.heappop(heap)
9     - Pop and return the smallest item from the heap, maintaining the heap invariant.
10    If the heap is empty, IndexError is raised. To access the smallest item without popping it,
11 """
12 nums = [2, 3, 5, 1, 54, 23, 132]
13 heap = []
14 for num in nums:
15     heapq.heappush(heap, num) # 加入堆
16
17 print(heap[0]) # 如果只是想获取最小值而不是弹出, 使用heap[0]
18 print([heapq.heappop(heap) for _ in range(len(nums))]) # 堆排序结果
19 # out: [1, 2, 3, 5, 23, 54, 132]
20
21
22 # 第二种
23 nums = [2, 3, 5, 1, 54, 23, 132]
24 heapq.heapify(nums)
25 print([heapq.heappop(nums) for _ in range(len(nums))]) # 堆排序结果
26 # out: [1, 2, 3, 5, 23, 54, 132]
```

9.4哈夫曼编码—heapq: 创建堆



```
1 """
2     函数定义:
3     heapq.merge(*iterables)
4         - Merge multiple sorted inputs into a single sorted output (for example, merge timestamp
5             - Similar to sorted(itertools.chain(*iterables)) but returns an iterable, does not pull
6 """
7 import heapq
8
9 num1 = [32, 3, 5, 34, 54, 23, 132]
10 num2 = [23, 2, 12, 656, 324, 23, 54]
11 num1 = sorted(num1)
12 num2 = sorted(num2)
13
14 res = heapq.merge(num1, num2)
15 print(list(res))
```

9.4 哈夫曼编码–heapq: 访问堆内容



```
1 import heapq
2 nums = [2, 43, 45, 23, 12]
3 heapq.heapify(nums)
4
5 print(heapq.heappop(nums))
6 # out: 2
7
8 # 如果需要所有堆排序后的元素
9 result = [heapq.heappop(nums) for _ in range(len(nums))]
10 print(result)
11 # out: [12, 23, 43, 45]
```

```
1 import heapq
2
3 nums = [1, 2, 4, 5, 3]
4 heapq.heapify(nums)
5
6 heapq.heapreplace(nums, 23)
7
8 print([heapq.heappop(nums) for _ in range(len(nums))])
9 # out: [2, 3, 4, 5, 23]
```

9.4 哈夫曼编码—`heapq`: 狱取堆最大或最小值



```
1 """
2     函数定义:
3     heapq.nlargest(n, iterable[, key])
4         - Return a list with the n largest elements from the dataset defined by iterable.
5         - key if provided, specifies a function of one argument that is used to extract a comparison
6         - Equivalent to: sorted(iterable, key=key, reverse=True)[:n]
7 """
8 import heapq
9
10 nums = [1, 3, 4, 5, 2]
11 print(heapq.nlargest(3, nums))
12 print(heapq.nsmallest(3, nums))
13
14 """
15     输出:
16     [5, 4, 3]
17     [1, 2, 3]
18 """
```

9.4 哈夫曼编码—heapq： 狱取堆最大或最小值



```
1 import heapq
2 from pprint import pprint
3 portfolio = [
4     {'name': 'IBM', 'shares': 100, 'price': 91.1},
5     {'name': 'AAPL', 'shares': 50, 'price': 543.22},
6     {'name': 'FB', 'shares': 200, 'price': 21.09},
7     {'name': 'HPQ', 'shares': 35, 'price': 31.75},
8     {'name': 'YHOO', 'shares': 45, 'price': 16.35},
9     {'name': 'ACME', 'shares': 75, 'price': 115.65}
10 ]
11 cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
12 expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
13 pprint(cheap)
14 pprint(expensive)
15 """
16 输出：
17 [{"name": "YHOO", "price": 16.35, "shares": 45},
18 {"name": "FB", "price": 21.09, "shares": 200},
19 {"name": "HPQ", "price": 31.75, "shares": 35}]
20 [{"name": "AAPL", "price": 543.22, "shares": 50},
21 {"name": "ACME", "price": 115.65, "shares": 75},
22 {"name": "IBM", "price": 91.1, "shares": 100}]
23 """
24 """
```



9.4 哈夫曼编码

HUFFMAN(C)

```
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8      INSERT( $Q, z$ )
9  return EXTRACT-MIN( $Q$ )    // return the root of the tree
```

算法时间复杂度分析
算法导论：P248



9.4哈夫曼编码

Greedy_HuffmanTree2.py



<https://blog.csdn.net/Campsisgranditora>



9.4 哈夫曼编码

算法的正确性证明(自学)

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Lemma 16.3

Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C , except that $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C' . Then the tree T , obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C .

Theorem 16.4

Procedure HUFFMAN produces an optimal prefix code.



提纲

9.1 概述

9.2 活动选择问题

9.3 贪心算法原理

9.4 哈夫曼编码

9.5 (分数) 背包问题

9.6 多机调度问题

9.7 贪心算法与人工智能算法



9.5 分数背包问题

三种贪心策略：

- 选择价值最大的物品
- 选择重量最轻的物品
- 选择单位重量价值最大的物品



9.5 分数背包问题

伪代码

1. 改变数组w和v的排列顺序，使其按单位重量价值 $v[i]/w[i]$ 降序排列；
2. 将数组x[n]初始化为0；
3. $i=1$ ；
4. 循环直到($w[i]>C$)
 - 4.1 将第*i*个物品放入背包： $x[i]=1$ ；
 - 4.2 $C=C-w[i]$ ；
 - 4.3 $i++$ ；
5. $x[i]=C/w[i]$; //物品*i*装入一部分



9.5 分数背包问题

代码实现

```
class goods:  
    def __init__(self, goods_id, weight=0, value=0):  
        self.id = goods_id  
        self.weight = weight  
        self.value = value
```

Greedy_FractionBag.py



9.5 分数背包问题

代码实现

```
def Bag(capacity=0, goods_set=[]):
    # 按单位价值量排序
    goods_set.sort(key=lambda obj: obj.value / obj.weight, reverse=True)
    result = []
    for a_goods in goods_set:
        if capacity < a_goods.weight:
            break
        result.append(a_goods)
        capacity -= a_goods.weight
    if len(result) < len(goods_set) and capacity != 0:
        result.append(goods(a_goods.id, capacity, a_goods.value * capacity / a_goods.weight))
    return result
```

Greedy_FractionBag.py



9.5 分数背包问题

代码实现

```
def main():
    some_goods = [goods(0, 2, 4), goods(1, 8, 6), goods(2, 5, 3), goods(3, 2, 8), goods(4, 1, 2)]
    res = Bag(6, some_goods)
    for obj in res:
        print('物品编号:' + str(obj.id) + ', 放入重量:' + str(obj.weight) + ', 放入的价值:' + str(obj.value))
        print('单位价值量为:' + str(obj.value / obj.weight))
```

Greedy_FractionBag.py



提纲

9.1 概述

9.2 活动选择问题

9.3 贪心算法原理

9.4 哈夫曼编码

9.5 (分数) 背包问题

9.6 多机调度问题

9.7 贪心算法与人工智能算法

9.6 多机调度问题描述



问题描述：设有 n 个独立的作业 $\{1, 2, \dots, n\}$ ，由 m 台相同的机器 $\{M_1, M_2, \dots, M_m\}$ 进行加工处理，作业 i 所需的处理时间为 t_i ($1 \leq i \leq n$)，每个作业均可在任何一台机器上加工处理，但不可间断、拆分。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。

9.6 多机调度问题-思路



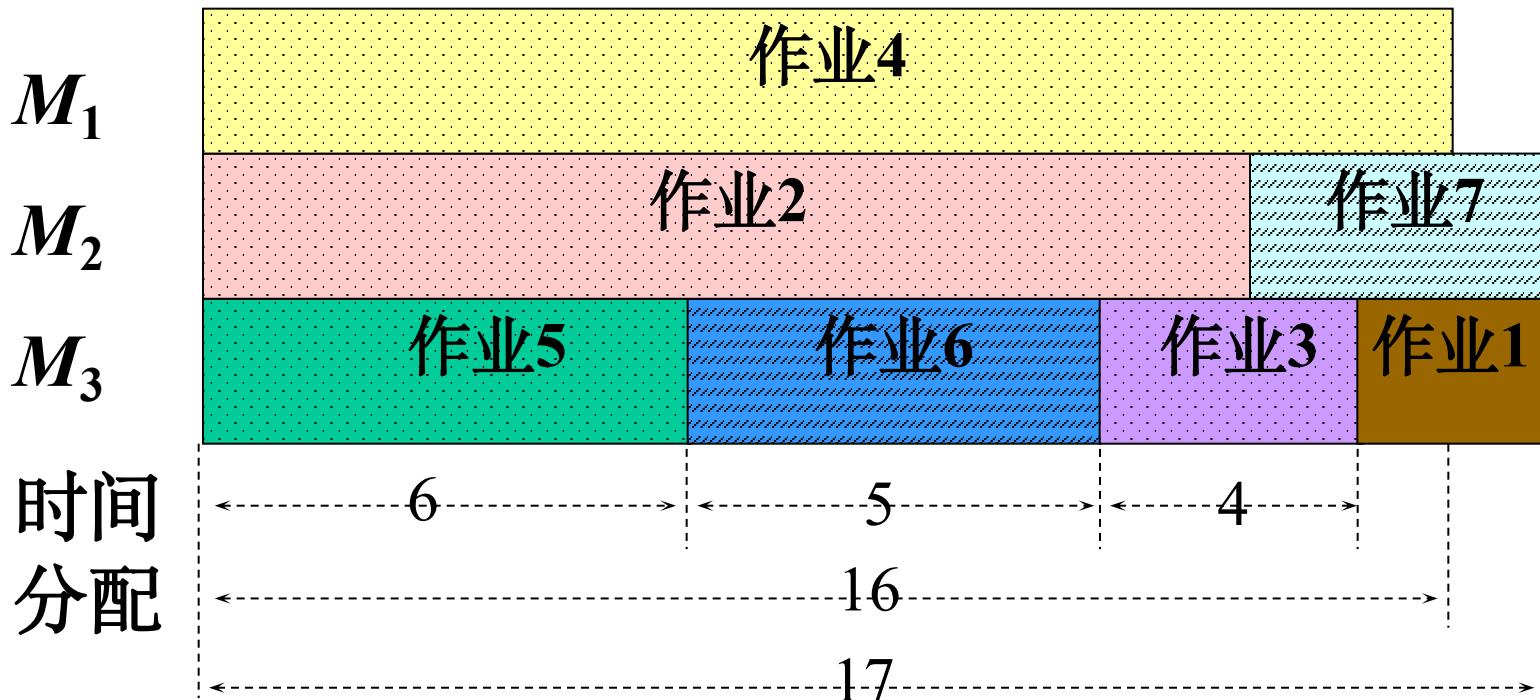
贪心法求解多机调度问题的贪心策略是最长处理时间作业优先，即把处理时间最长的作业分配给最先空闲的机器，这样可以保证处理时间长的作业优先处理，从而在整体上获得尽可能短的处理时间。

按照最长处理时间作业优先的贪心策略，当 $m \geq n$ 时，只要将机器*i*的 $[0, t_i)$ 时间区间分配给作业*i*即可；当 $m < n$ 时，首先将*n*个作业依其所需的处理时间从大到小排序，然后依此顺序将作业分配给空闲的处理机。

9.6 多机调度问题-实例



设7个独立作业 $\{1, 2, 3, 4, 5, 6, 7\}$ 由3台机器 $\{M_1, M_2, M_3\}$ 加工处理，各作业所需的处理时间为 $\{2, 14, 4, 16, 6, 5, 3\}$ 。





9.6 多机调度问题

设 n 个作业的处理时间存储在数组 $t[n]$ 中, m 台机器的空闲时间存储在数组 $d[m]$ 中, 集合数组 $S[m]$ 存储每台机器所处理的作业, 其中集合 $S[i]$ 表示机器 i 所处理的作业, 贪心法求解多机调度问题的算法如下:

伪代码

算法 7.9——多机调度问题

```
1. 将数组 t[n] 由大到小排序, 对应的作业序号存储在数组 p[n] 中;  
2. 将数组 d[m] 初始化为 0;  
3. for (i = 1; i <= m; i++)  
    3.1 S[i] = {p[i]}; // 将 m 个作业分配给 m 个机器  
    3.2 d[i] = t[i];  
4. for (i = m + 1; i <= n; i++)  
    4.1 j = 数组 d[m] 中最小值对应的下标; // j 为最先空闲的机器序号  
    4.2 S[j] = S[j] + {p[i]}; // 将作业 i 分配给最先空闲的机器 j  
    4.3 d[j] = d[j] + t[i]; // 机器 j 将在 d[j] 后空闲
```



提纲

9.1 概述

9.2 活动选择问题

9.3 贪心算法原理

9.4 哈夫曼编码

9.5 (分数) 背包问题

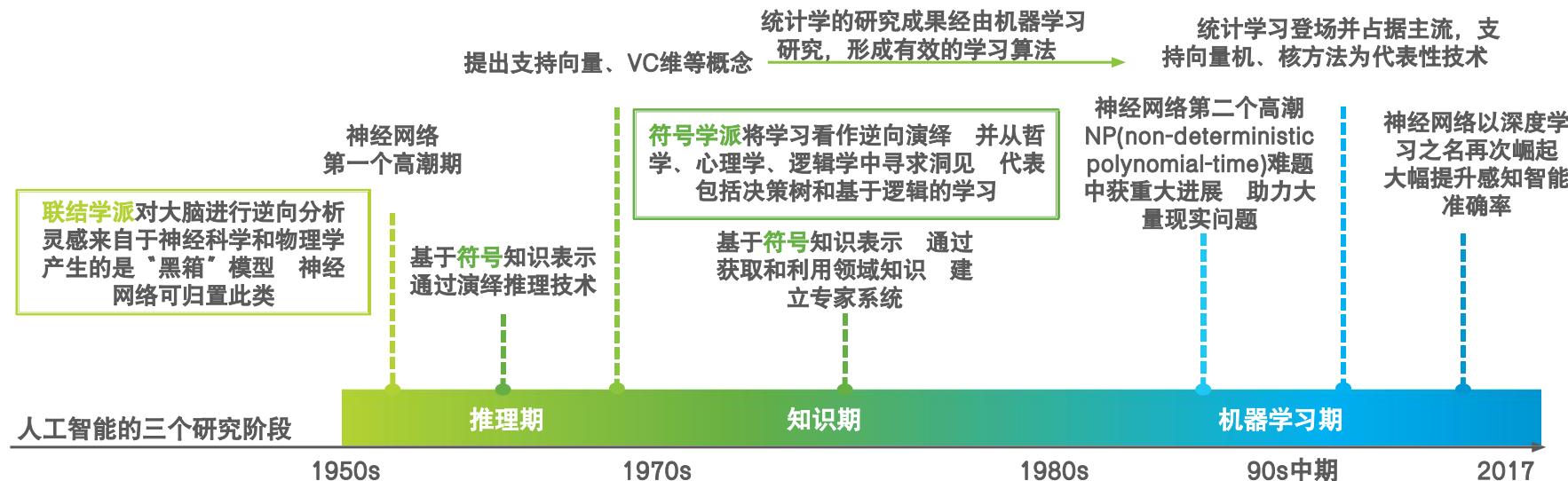
9.6 多机调度问题

9.7 贪心算法与人工智能算法



人工智能算法

- 从以“推理”为重点到以“知识”为重点，再到以“学习”为重点
- 机器可以自动“学习”的算法，即从数据中自动分析获得规律，并利用规律对未知数据进行预测的算法。目前，机器学习=“分类”
- 人工智能 > 机器学习 > 深度学习





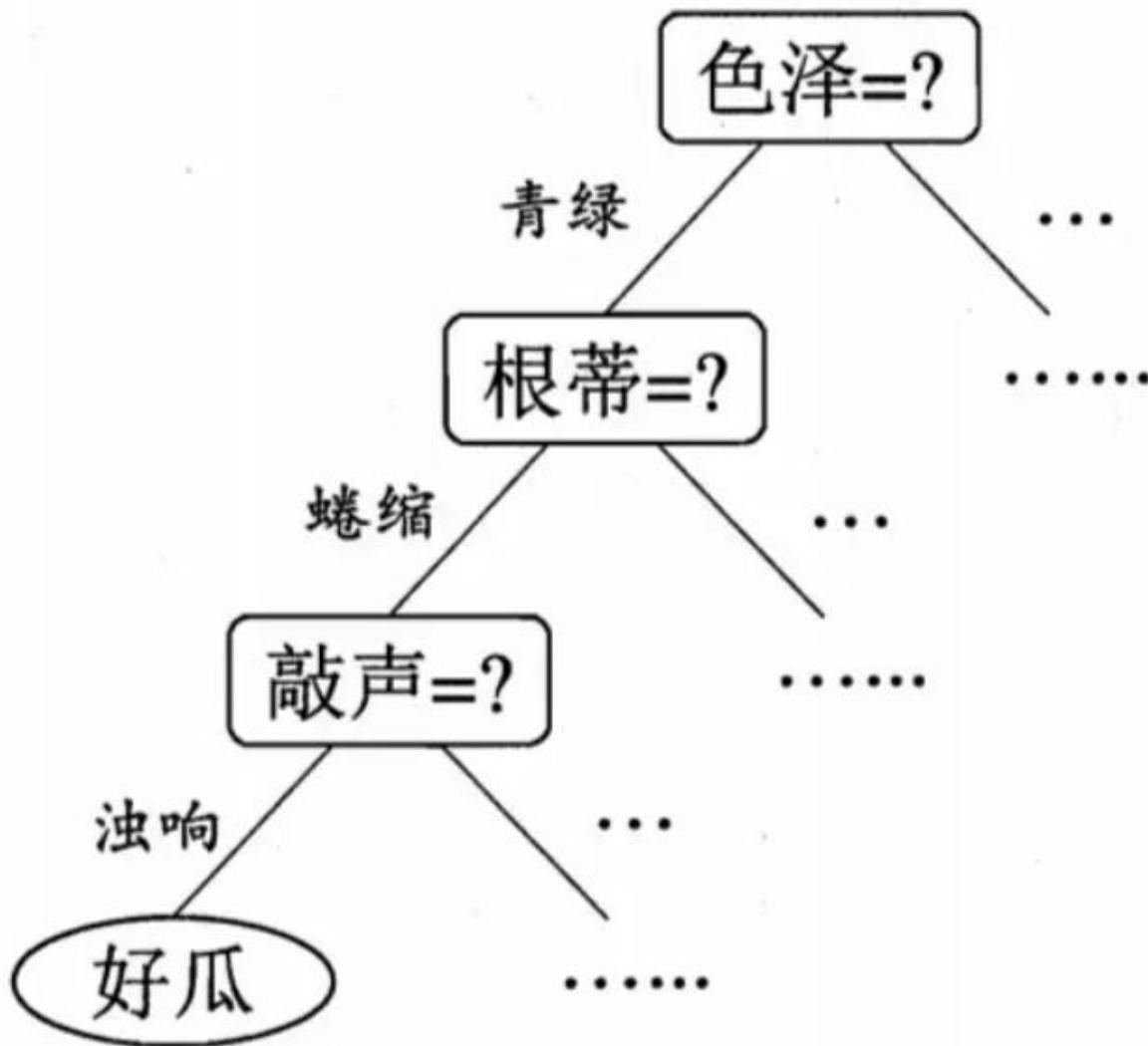
机器学习算法与贪心算法关系

现有机器学习算法利用输入数据多轮迭代地生成模型，每轮迭代中都是贪心地选择当前最优模型来生成。

为了进一步降低计算空间进而提高效率，在贪心的基础上，很多机器学习算法在执行过程中还加上了随机的思想。它们随机地在下一步可能的最优模型中选择若干分支。



机器学习典型算法：决策树





机器学习典型算法：决策树

一棵决策树的生成过程主要分为以下3个部分：

- 特征选择：特征选择是指从训练数据中众多的特征中选择一个特征作为当前节点的分裂标准，如何选择特征有着很多不同量化评估标准，从而衍生出不同的决策树算法。
- 决策树生成：根据选择的特征评估标准，从上至下递归地生成子节点，直到数据集不可分则停止决策树停止生长。树结构来说，递归结构是最容易理解的方式。
- 剪枝：决策树容易过拟合，一般来需要剪枝，缩小树结构规模、缓解过拟合。剪枝技术有预剪枝和后剪枝两种。

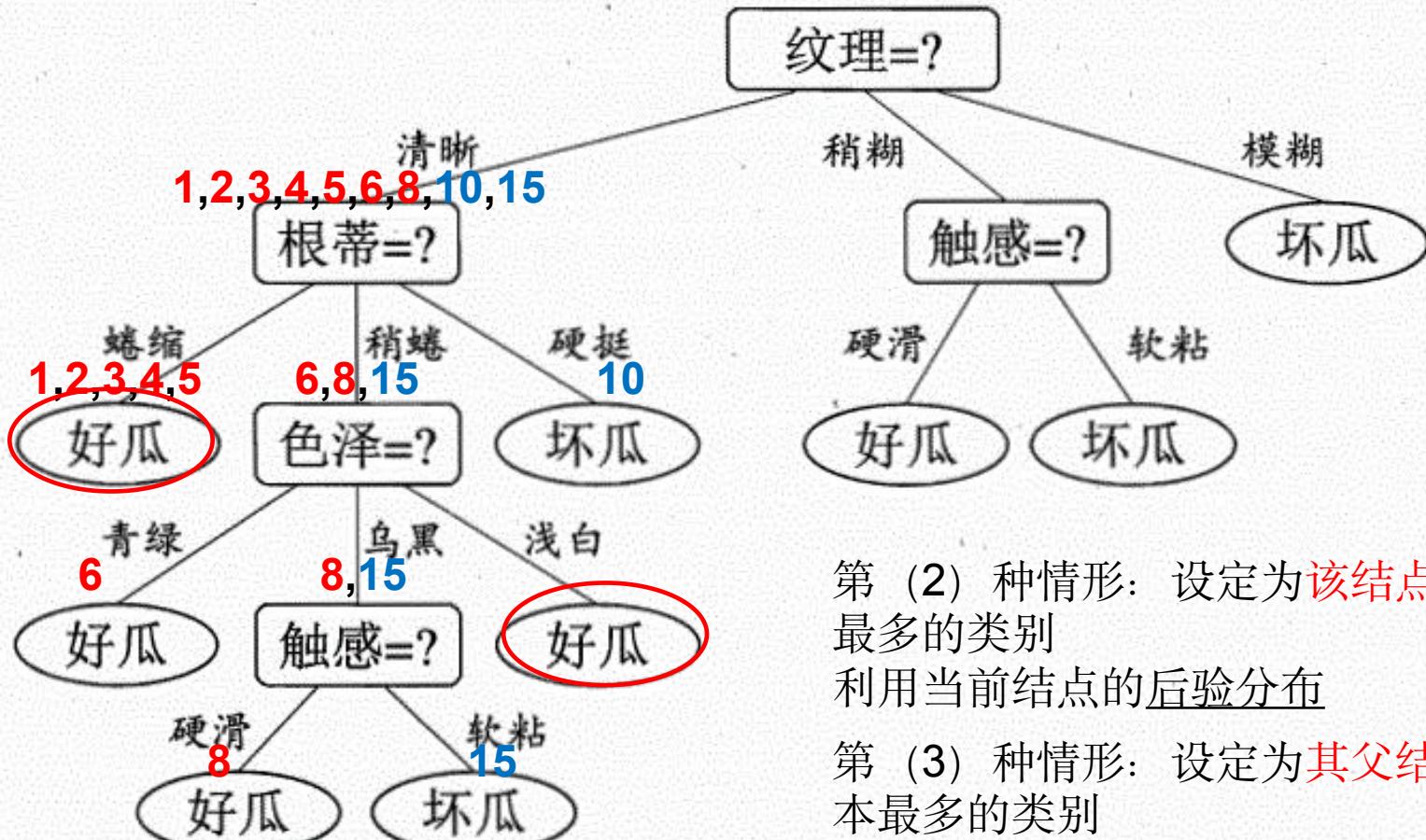


示例数据 (西瓜数据集2.0)

| 编号 | 色泽 | 根蒂 | 敲声 | 纹理 | 脐部 | 触感 | 好瓜 |
|----|----|----|----|----|----|----|----|
| 1 | 青绿 | 蜷缩 | 浊响 | 清晰 | 凹陷 | 硬滑 | 是 |
| 2 | 乌黑 | 蜷缩 | 沉闷 | 清晰 | 凹陷 | 硬滑 | 是 |
| 3 | 乌黑 | 蜷缩 | 浊响 | 清晰 | 凹陷 | 硬滑 | 是 |
| 4 | 青绿 | 蜷缩 | 沉闷 | 清晰 | 凹陷 | 硬滑 | 是 |
| 5 | 浅白 | 蜷缩 | 浊响 | 清晰 | 凹陷 | 硬滑 | 是 |
| 6 | 青绿 | 稍蜷 | 浊响 | 清晰 | 稍凹 | 软粘 | 是 |
| 7 | 乌黑 | 稍蜷 | 浊响 | 稍糊 | 稍凹 | 软粘 | 是 |
| 8 | 乌黑 | 稍蜷 | 浊响 | 清晰 | 稍凹 | 硬滑 | 是 |
| 9 | 乌黑 | 稍蜷 | 沉闷 | 稍糊 | 稍凹 | 硬滑 | 否 |
| 10 | 青绿 | 硬挺 | 清脆 | 清晰 | 平坦 | 软粘 | 否 |
| 11 | 浅白 | 硬挺 | 清脆 | 模糊 | 平坦 | 硬滑 | 否 |
| 12 | 浅白 | 蜷缩 | 浊响 | 模糊 | 平坦 | 软粘 | 否 |
| 13 | 青绿 | 稍蜷 | 浊响 | 稍糊 | 凹陷 | 硬滑 | 否 |
| 14 | 浅白 | 稍蜷 | 沉闷 | 稍糊 | 凹陷 | 硬滑 | 否 |
| 15 | 乌黑 | 稍蜷 | 浊响 | 清晰 | 稍凹 | 软粘 | 否 |
| 16 | 浅白 | 蜷缩 | 浊响 | 模糊 | 平坦 | 硬滑 | 否 |
| 17 | 青绿 | 蜷缩 | 沉闷 | 稍糊 | 稍凹 | 硬滑 | 否 |



示例决策树



第(2)种情形：设定为该结点所含样本最多的类别

利用当前结点的后验分布

第(3)种情形：设定为其父结点所含样本最多的类别

把父结点的样本分布作为当前结点的先验分布

图 4.4 在西瓜数据集 2.0 上基于信息增益生成的决策树



决策树典型算法

Hunt算法:

输入: 训练集 $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;
属性集 $A = \{a_1, a_2, \dots, a_d\}$.

过程: 函数 TreeGenerate(D, A)

1: 生成结点 node;

2: if D 中样本全属于同一类别 C then

3: 将 node 标记为 C 类叶结点; return

无需划分

4: end if

5: if $A = \emptyset$ OR D 中样本在 A 上取值相同 then

6: 将 node 标记为叶结点, 其类别标记为 D 中样本数最多的类; return

无法划分

7: end if

8: 从 A 中选择最优划分属性 a_* ;

9: for a_* 的每一个值 a_*^v do

10: 为 node 生成一个分支; 令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集;

11: if D_v 为空 then

12: 将分支结点标记为叶结点, 其类别标记为 D 中样本最多的类; return

不能划分

13: else

14: 以 TreeGenerate($D_v, A \setminus \{a_*\}$) 为分支结点

15: end if

16: end for

输出: 以 node 为根结点的一棵决策树

递归返回, 情形(1).

递归返回, 情形(2).

我们将在下一节讨论如何获得最优划分属性.

递归返回, 情形(3).

从 A 中去掉 a_* .



决策树算法

决策树学习的关键是算法的第8行：**贪心地**选择最优划分属性

什么样的划分属性是最优的？

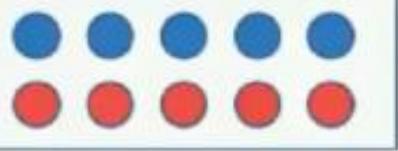
我们希望决策树的分支结点所包含的样本尽可能属于同一类别，即结点的“纯度”越来越高，可以高效地从根结点到达叶结点，得到决策结果。



决策树算法

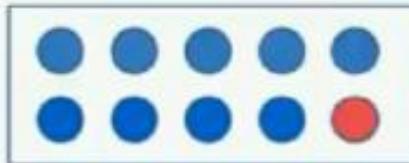
节点内尽量都是“同类”的数据！

用“不纯度” **impurity** 来度量



“不纯度” 很高

(纯度很低)



“不纯度” 很低

(纯度很高)

<https://blog.csdn.net/u12595>

不纯度 (impurity) -- GINI系数：

$$Gini(p) = \sum_{k=1}^K p_k(1-p_k) = 1 - \sum_{k=1}^K p_k^2$$



决策树算法

Gini计算示例

$$GINI = 1 - \sum_{i=1}^c p(i)^2$$

→

| | |
|----|---|
| C1 | 0 |
| C2 | 6 |

$$P(C1) = 0/6 = 0 \quad P(C2) = 6/6 = 1$$

$$Gini = 1 - P(C1)^2 - P(C2)^2 = 1 - 0 - 1 = 0$$

→

| | |
|----|---|
| C1 | 1 |
| C2 | 5 |

$$P(C1) = 1/6 \quad P(C2) = 5/6$$

$$Gini = 1 - (1/6)^2 - (5/6)^2 = 0.278$$

→

| | |
|----|---|
| C1 | 2 |
| C2 | 4 |

$$P(C1) = 2/6 \quad P(C2) = 4/6$$

$$Gini = 1 - (2/6)^2 - (4/6)^2 = 0.444$$



信息熵与信息增益

“信息熵” (information entropy) 是度量样本集合纯度最常用的一种指标。假定当前样本集合 D 中第 k 类样本所占的比例为 p_k ($k = 1, 2, \dots, |\mathcal{Y}|$), 则 D 的信息熵定义为

$$\text{Ent}(D) = - \sum_{k=1}^{|\mathcal{Y}|} p_k \log_2 p_k . \quad (4.1)$$

$\text{Ent}(D)$ 的值越小, 则 D 的纯度越高。对于二分类任务 $|\mathcal{Y}| = 2$

假定离散属性 a 有 V 个可能的取值 $\{a^1, a^2, \dots, a^V\}$, 若使用 a 来对样本集 D 进行划分, 则会产生 V 个分支结点, 其中第 v 个分支结点包含了 D 中所有在属性 a 上取值为 a^v 的样本, 记为 D^v . 我们可根据式(4.1) 计算出 D^v 的信息熵, 再考虑到不同的分支结点所包含的样本数不同, 给分支结点赋予权重 $|D^v|/|D|$, 即样本数越多的分支结点的影响越大, 于是可计算出用属性 a 对样本集 D 进行划分所获得的“信息增益” (information gain)

$$\text{Gain}(D, a) = \text{Ent}(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Ent}(D^v) . \quad (4.2)$$



信息熵示例

举例：求解划分根结点的最优划分属性

数据集包含17个训练样例：

$$\begin{aligned} \text{8个正例 (好瓜) 占 } p_1 &= \frac{8}{17} && \text{对于二分类任务 } |y| = 2 \\ \text{9个反例 (坏瓜) 占 } p_2 &= \frac{9}{17} \end{aligned}$$

以属性“色泽”为例计算其信息增益

根结点的信息熵：

$$\text{Ent}(D) = - \sum_{k=1}^2 p_k \log_2 p_k = - \left(\frac{8}{17} \log_2 \frac{8}{17} + \frac{9}{17} \log_2 \frac{9}{17} \right) = 0.998 .$$



信息增益示例

用“色泽”将根结点划分后获得3个分支结点的信息熵分别为：

$$\text{Ent}(D^1) = -\left(\frac{3}{6} \log_2 \frac{3}{6} + \frac{3}{6} \log_2 \frac{3}{6}\right) = 1.000,$$

$$\text{Ent}(D^2) = -\left(\frac{4}{6} \log_2 \frac{4}{6} + \frac{2}{6} \log_2 \frac{2}{6}\right) = 0.918,$$

$$\text{Ent}(D^3) = -\left(\frac{1}{5} \log_2 \frac{1}{5} + \frac{4}{5} \log_2 \frac{4}{5}\right) = 0.722,$$

属性“色泽”的信息增益为：

$$\begin{aligned}\text{Gain}(D, \text{色泽}) &= \text{Ent}(D) - \sum_{v=1}^3 \frac{|D^v|}{|D|} \text{Ent}(D^v) \\ &= 0.998 - \left(\frac{6}{17} \times 1.000 + \frac{6}{17} \times 0.918 + \frac{5}{17} \times 0.722 \right) \\ &= 0.109.\end{aligned}$$



基于信息增益的决策树示例

类似的，我们可计算出其他属性的信息增益：

$$\text{Gain}(D, \text{根蒂}) = 0.143; \quad \text{Gain}(D, \text{敲声}) = 0.141;$$

$$\text{Gain}(D, \text{纹理}) = 0.381; \quad \text{Gain}(D, \text{脐部}) = 0.289;$$

$$\text{Gain}(D, \text{触感}) = 0.006.$$

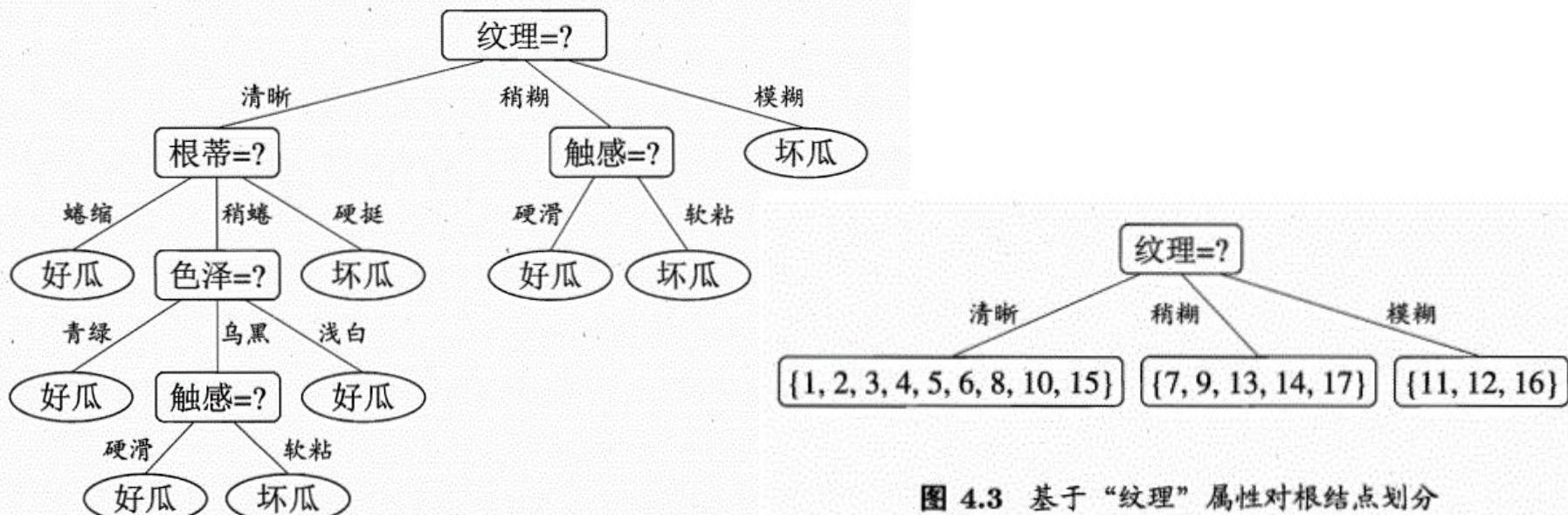


图 4.3 基于“纹理”属性对根结点划分

图 4.4 在西瓜数据集 2.0 上基于信息增益生成的决策树



信息增益的不足

若把“编号”也作为一个候选划分属性，则属性“编号”的信息增益为：

根结点的信息熵仍为： $Ent(D) = 0.998$

用“编号”将根结点划分后获得17个分支结点的信息熵均为：

$$Ent(D^1) = \dots = Ent(D^{17}) = -\left(\frac{1}{1} \log_2 \frac{1}{1} + \frac{0}{1} \log_2 \frac{0}{1}\right) = 0$$

则“编号”的信息增益为：

$$Gain(D, \text{编号}) = Ent(D) - \sum_{v=1}^{17} \frac{1}{17} Ent(D^v) = 0.998$$

远大于其他候选属性

信息增益准则对可取值数目较多的属性有所偏好



增益率

$$\text{Gain_ratio}(D, a) = \frac{\text{Gain}(D, a)}{\text{IV}(a)}, \quad (4.3)$$

其中

$$\text{IV}(a) = -\sum_{v=1}^V \frac{|D^v|}{|D|} \log_2 \frac{|D^v|}{|D|} \quad (4.4)$$

称为属性 a 的“固有值”(intrinsic value) [Quinlan, 1993]. 属性 a 的可能取值数目越多(即 V 越大), 则 $\text{IV}(a)$ 的值通常会越大.

增益率准则对可取值数目较少的属性有所偏好

著名的C4.5决策树算法综合了信息增益准则和信息率准则的特点: 先从候选划分属性中找出信息增益高于平均水平的属性, 再从中选择增益率最高的。

聚类算法典型——K均值算法



k-均值 (k-means) 算法，是一种得到最广泛使用的聚类算法

它将各个聚类子集内的所有数据样本的均值作为该聚类的代表点，其他点和其最近的代表点同一个类

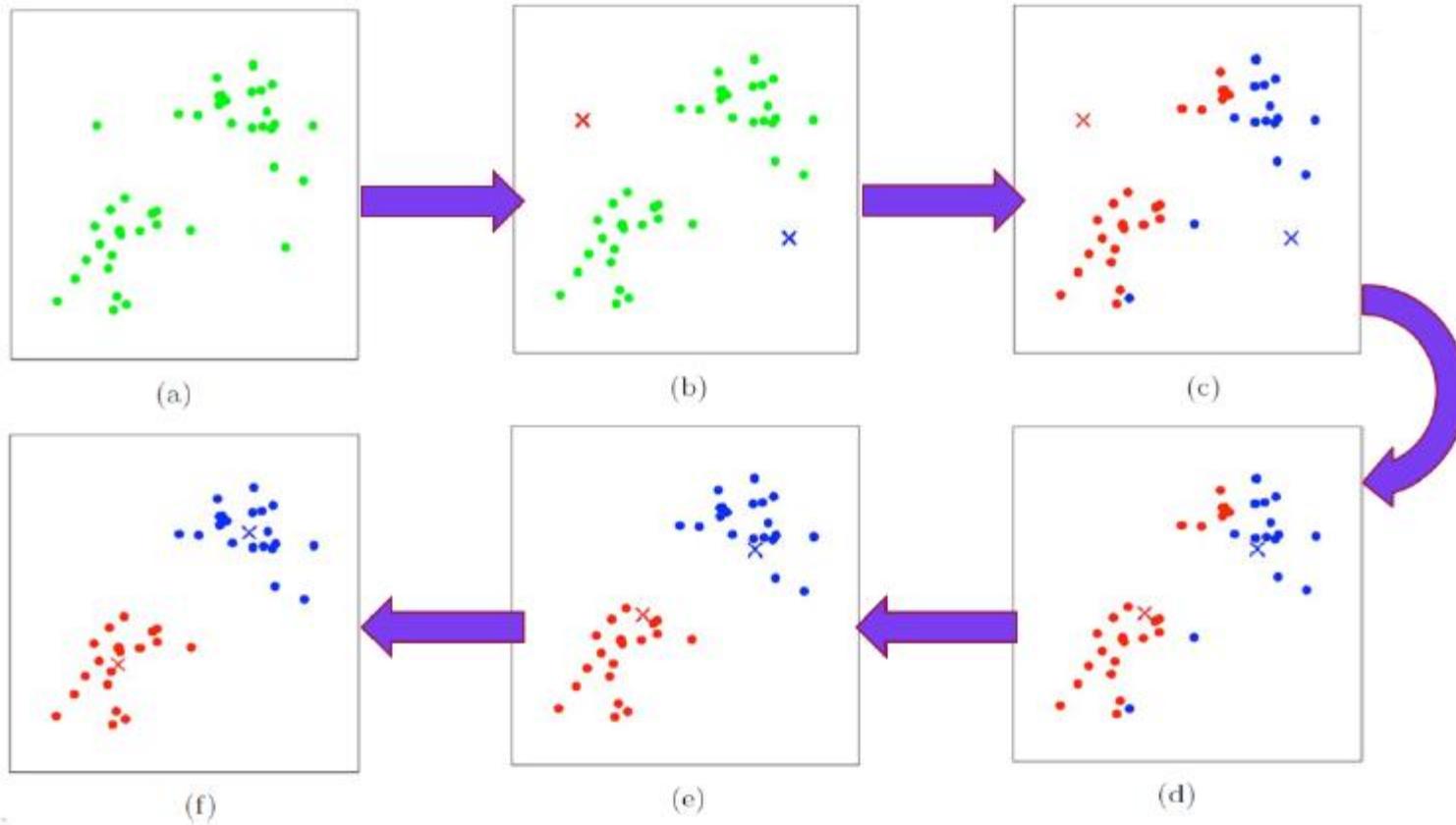
K均值算法过程



1. 为每个聚类随机确定一个实例作为初始聚类中心，这样就有 K 个初始聚类中心
2. 将每个实例按照最小距离原则分配到最邻近聚类（即贪心选择）
3. 使用每个聚类中实例均值作为新聚类中心
4. 重复步骤2.3直到聚类中心不再变化
5. 结束，得到 K 个聚类



算法运行实例





K均值算法距离定义

距离定义：典型定义为欧式距离：

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^d (x_{ik} - x_{jk})^2}$$



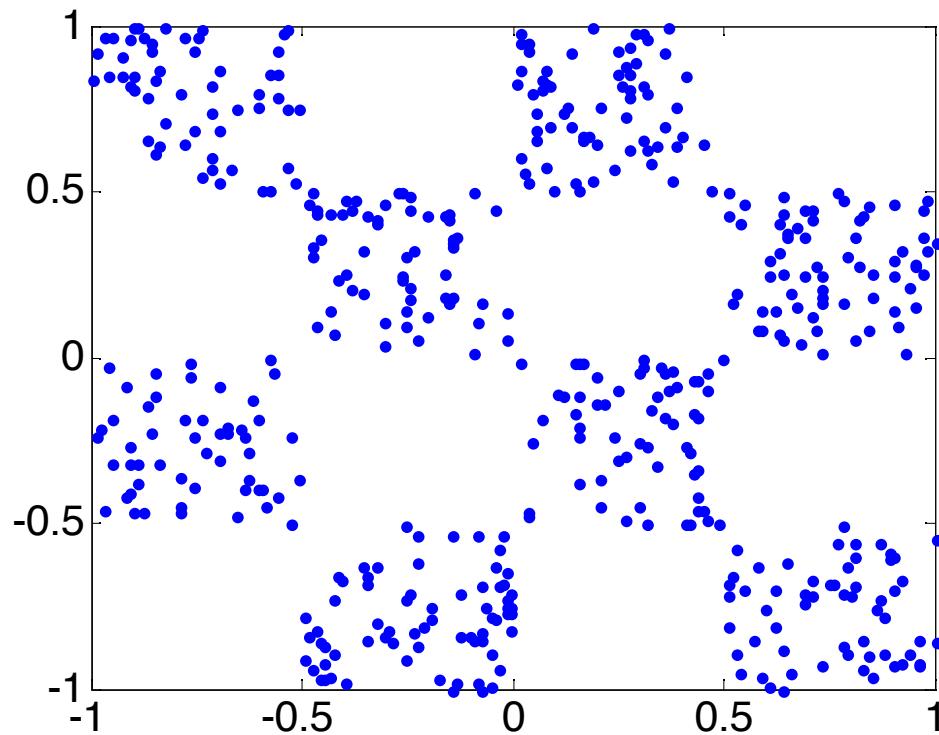
K均值算法特点

- 必须事先给出 k (要生成的簇的数目) ,而且对初值敏感;
- 对于“躁声” 和孤立点数据是敏感的, 少量的该类实例能够对结果产生极大的影响;
- 不能保证找到全局最优解, 只能确保局部最优解



初始中心的选取对算法的影响

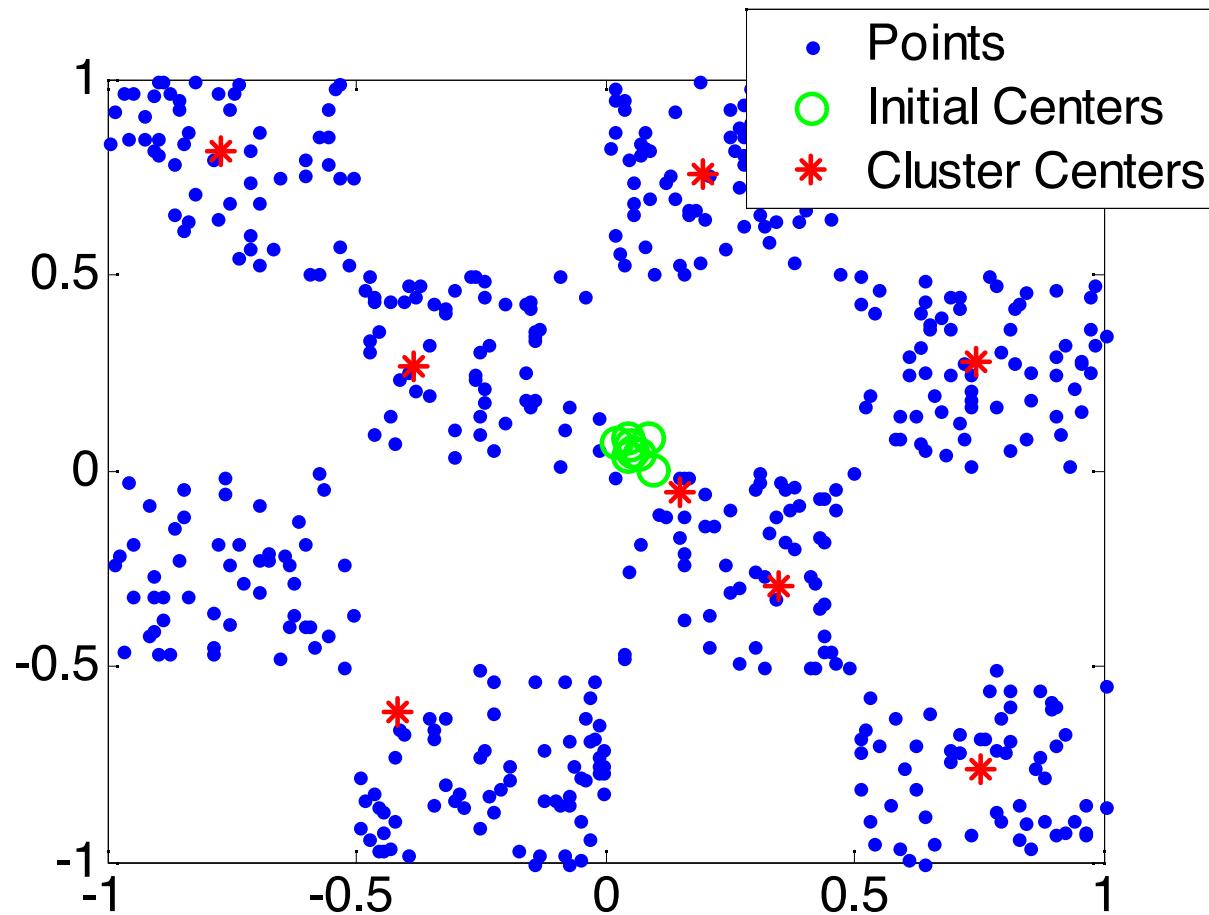
棋盘格数据集(Checkerboard data set)





初始中心的选取对算法的影响

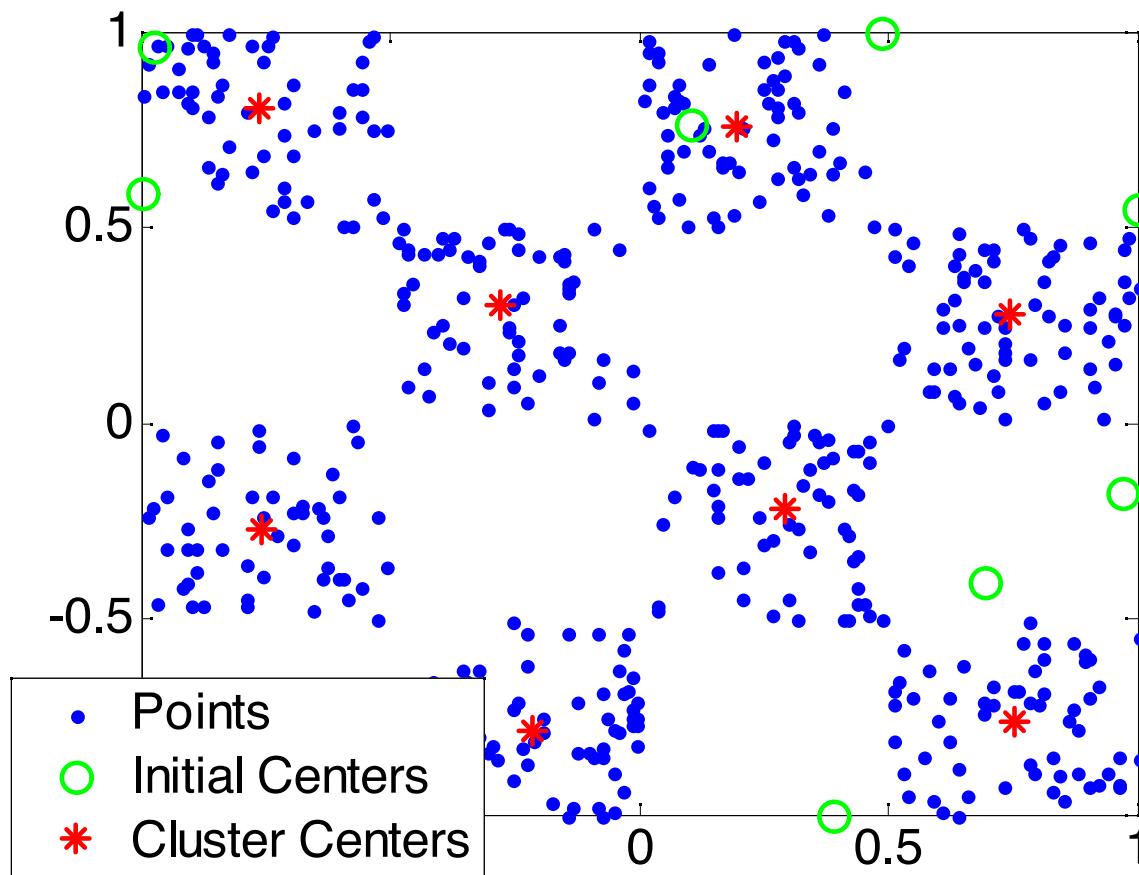
初始聚类中心均在中心附近





初始中心的选取对算法的影响

初始聚类中心在平面内随机选取



9.6 机器学习参考书目



《机器学习》，周志华著，清华大学出版社
《统计学习方法》，李航著，清华大学出版社
社