算法分析与设计

主讲教师：刘峤

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**
  - 最优子结构性质和贪心选择性质
  - 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；最小生成树
  - 哈夫曼编码；多机调度问题

贪心算法的例子：找零钱问题

- 找零钱问题
  - 假设有4种面值的硬币：二角五分、一角、五分和一分
  - 如果要找给顾客六角三分钱，怎样使找出的硬币个数最少？
  - 直觉：选择2个两角五分、1个一角、3个一分
- 问题的求解过程
  - 首先选出1个面值不超过六角三分的最大硬币(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(两角五分)
  - 如此一直做下去……直到找出的硬币总额为六角三分
  - 这里用到的方法就是贪心算法
  - 在这个例子中，贪心算法得到的结果是整体最优解

贪心算法的例子：找零钱问题

- 是否可以用动态规划算法求解？
  - 首先：看问题本身是否具有最优子结构性质
    - 提示：若m[38]+1是问题63的最优解？
  - 然后：设定问题的最优解（设给定金额为n）
    - m[i]表示凑出面值 i 所需最少的硬币数量
    - 原问题的最优解为：m[n]
  - 接着：列出递归求解最优值的表达式

$$m[i] = \min_{0 \le k \le 3} \{ m[i - coin[k]] + 1 \}$$

动态规划法求解找零钱问题

```
void change(int C[], int M[], int S[], int m, int n){
    int i, k; M[0] = 0; S[0] = 0;
    for(i = 1; i <= m; i++)  M[i] = INT_MAX;
    for(i = 1; i <= m; i++){
        for(k = 0; k < n; k++){
            if(C[k] <= i && (M[i - C[k]] + 1) < M[i]){
                M[i] = M[i - C[k]] + 1;
                S[i] = k;
            }
        }
    }
}
```

算法复杂度？ O(mn)

找零钱问题小结

- 问题具有最优子结构性质：因此可用动态规划求解
  - 然而：用贪心算法更简单，而且计算效率更高
- 在这个例子中采用贪心算法得到的结果是**全局最优解**
  - 然而：贪心算法并不能总是保证得到全局最优解
  - 思考：什么情况下不能用贪心算法求解找零钱问题？
    - 找零钱问题的解和硬币面值的设定有关
    - 如果将硬币面值改为：一分、五分和一角一分，
    - 假设要找给顾客一角五分？
    - 贪心算法：1个一角一分的硬币和4个一分硬币
    - 然而问题的全局最优解为：3个五分硬币

贪心算法的基本思想

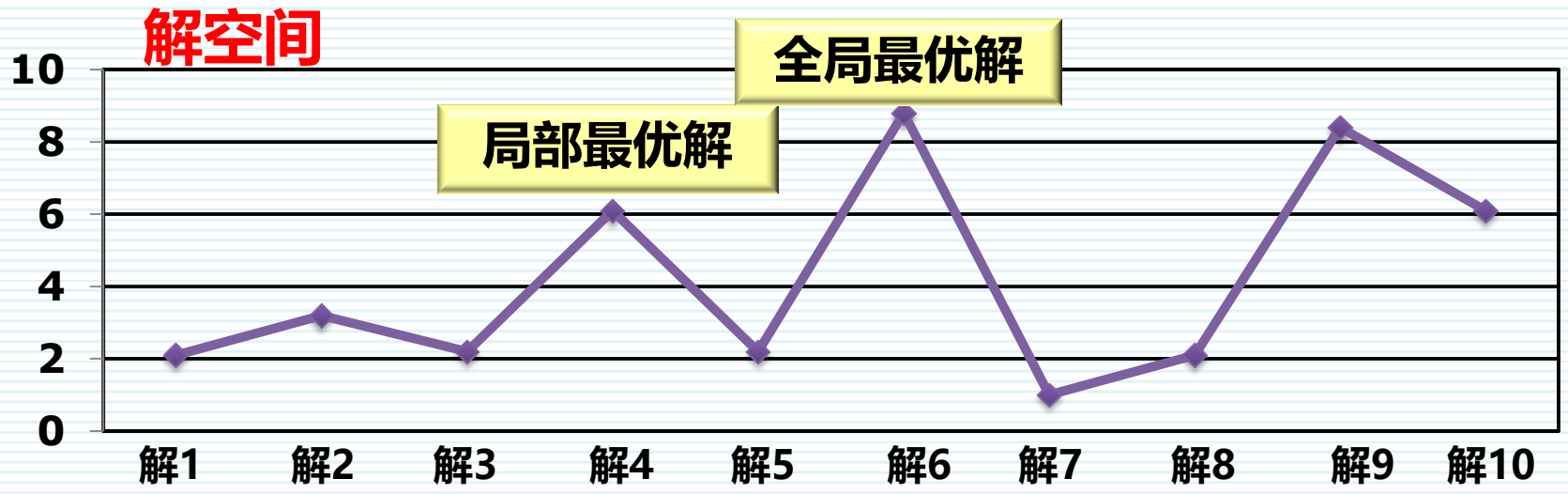
- 回顾我们目前已经见过的优化问题
  - 矩阵链加括号，最大子段和，背包问题，找零问题……
  - 优化问题的算法往往包含一系列步骤
    - 求解过程的每一步都面临一组选择
- 贪心算法的基本思想
  - 贪心算法在求解问题时并不着眼于整体最优
    - 在每一步选择中都采取在当前状态下最优的选择
  - 贪心算法能否得到整体最优解？……具体问题，具体分析
- 贪心算法在有最优子结构的问题中尤为有效
  - 问题能够分解成子问题来解决
  - 局部最优解能决定全局最优解

贪心算法与动态规划的区别

- 动态规划算法
  - 每一步的最优解通过对上一步的局部最优解进行选择得到
    - 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：下一步的最优解直接由上一步的最优解推导得到
  - 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）
- 贪心算法本质上是一种动态规划算法（更快捷）
  - 算法正确的条件：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

- 再次强调：贪心算法得到的结果不能保证全局最优
  - 但是：有不少问题能采用贪心算法得到全局最优解
    - 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



1．活动安排问题

（ Activity-Selection Problem ）

活动安排问题

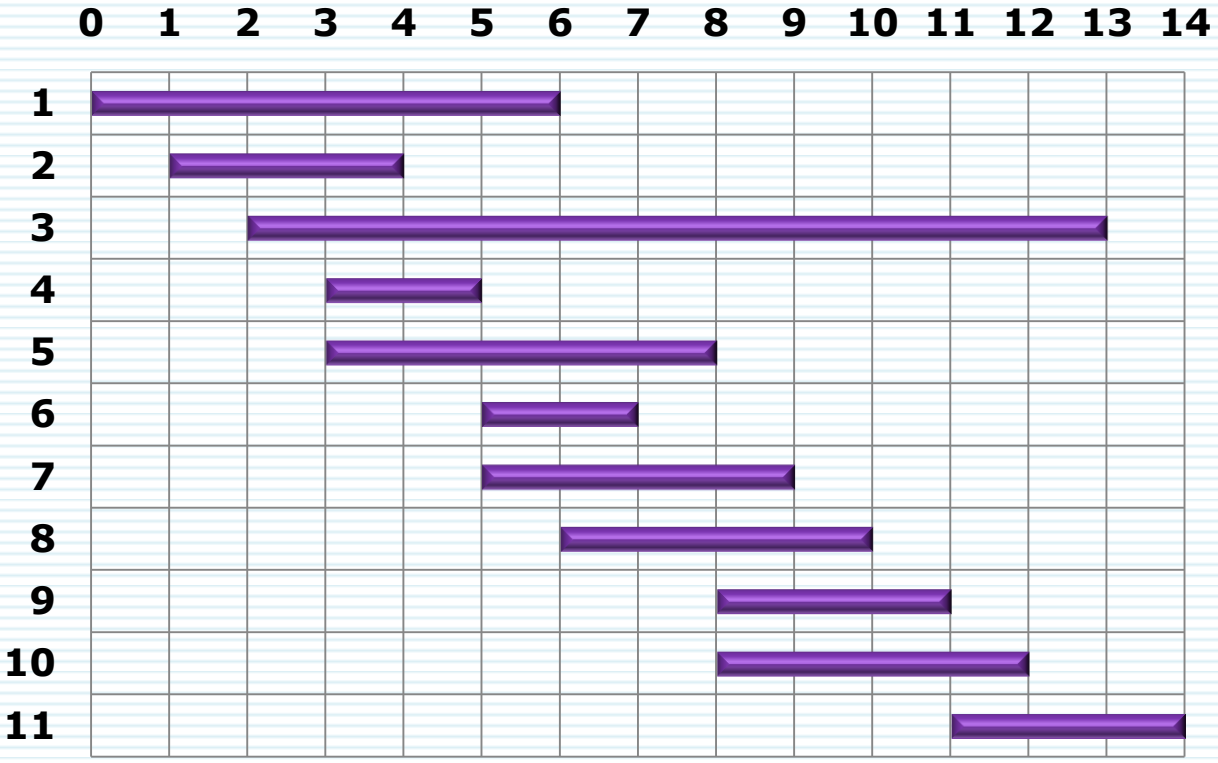
- 设：有n个活动的集合 **A={1,2,…,n}**
  - 其中：每个活动都要求竞争使用同一资源（如演讲会场等）
  - 而在同一时间内只有一个活动能使用这一资源
    - 每个活动 **i** 都有一个请求使用该资源的起始时间 s_i
    - 每个活动 **i** 都有一个使用资源的结束时间 f_i，且 $s_i < f_i$
  - 如果选择了活动 i，则它在半开时间区间$[s_i, f_i)$内占用资源
  - 活动 i 与活动 j **相容**：区间$[s_i, f_i)$与$[s_j, f_j)$不相交
    - 即：当 $s_i \geq f_j$ 或 $s_j \geq f_i$ 时，活动 i 与 j 相容
- 活动安排问题：在给定活动集合中选出**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

设：待安排的11个活动如下：

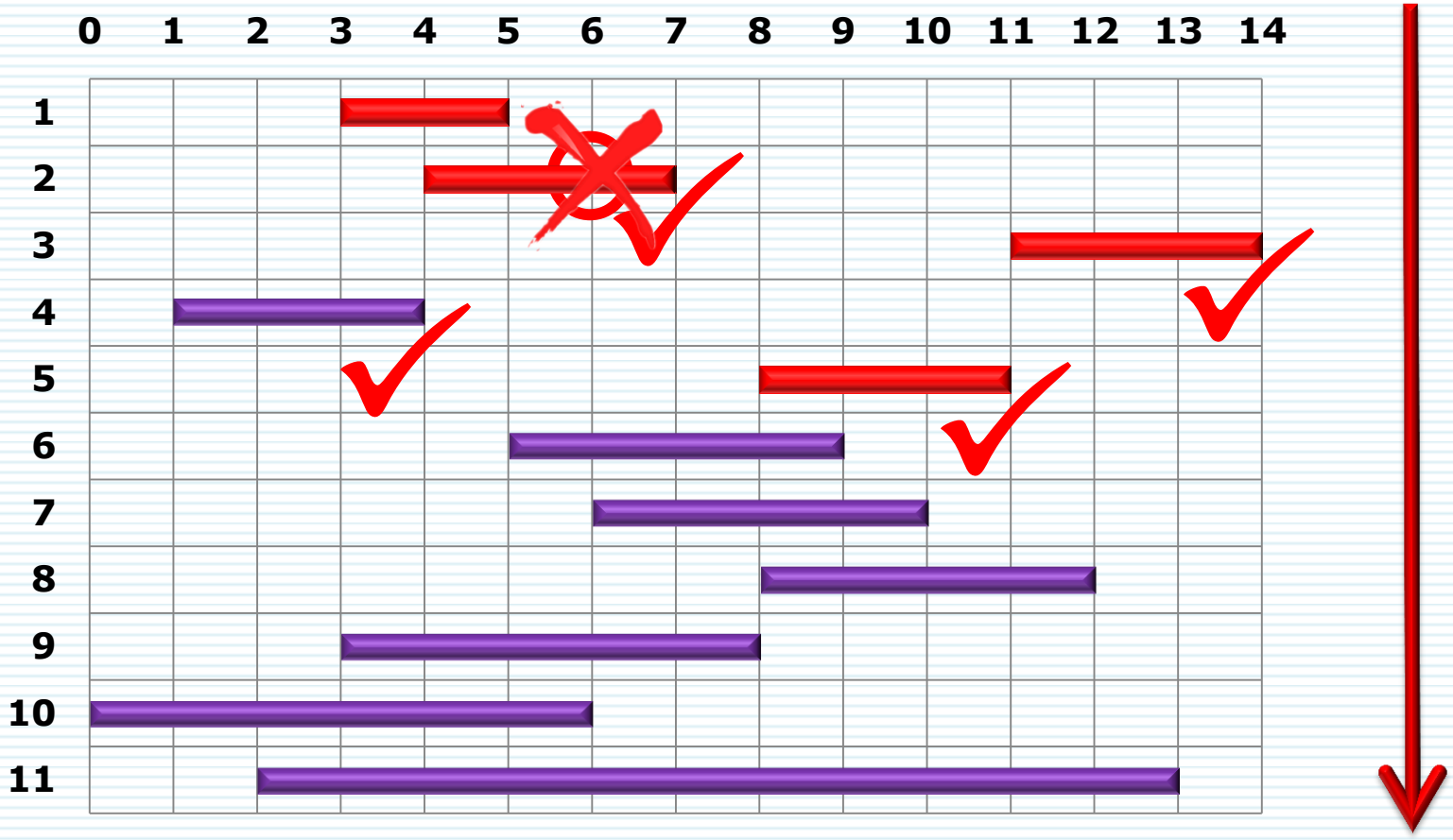
i	1	2	3	4	5	6	7	8	9	10	11
S[i]	0	1	2	3	3	5	5	6	8	8	11
F[i]	6	4	13	5	8	7	9	10	11	12	14

按活动开始时间 S[i] 排序：



设：待安排的11个活动按持续时间的非递减顺序排列如下：

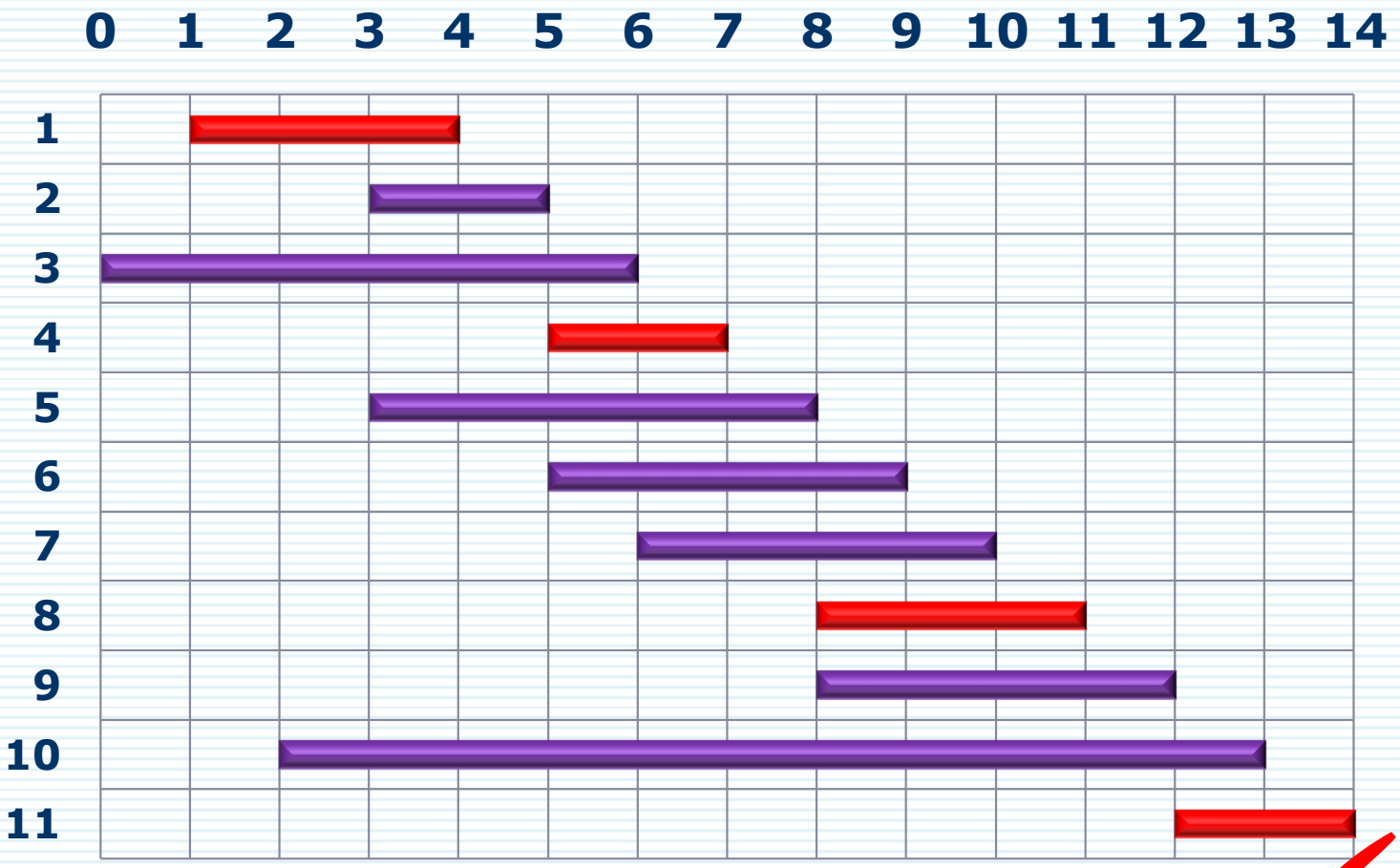
i	1	2	3	4	5	6	7	8	9	10	11
S[i]	3	5	11	1	8	5	6	8	3	0	2
F[i]	5	7	14	4	11	9	10	12	8	6	13



问题：这种解法能否确保全局最优？

例：设待安排的11个活动按结束时间的非递减顺序排列如下：

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



问：这种解法能否确保全局最优？

贪心算法求解活动安排问题

- 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
  - 用数学归纳法证明贪心算法的解是全局最优解
  - 首先需要证明活动安排问题**有一个**最优解以贪心选择开始
- 证明：活动安排问题有一个最优解以贪心选择开始
  - 设：A={1，…，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B是该问题的一个最优解（元素按F[k]非减序排列）
    - 不妨设：B中的第一个活动是活动 k

贪心算法求解活动安排问题

- 证明：活动安排问题有一个最优解以贪心选择开始
  - 若 k=1 则：B 就是一个以贪心选择开始的最优解
  - 若 k>1 ？
    - 设：C=(B-{k})∪{1}
    - 由于：F[1]≤F[k]
      - 且B中活动相容，故C中活动也相容
    - 由于C和B中包含的活动个数相同，故C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
- 据此可利用数学归纳法证明：贪心算法的解是全局最优解

贪心算法求解活动安排问题

- 数学归纳法证明：贪心算法的解是全局最优解
  - 在做出贪心选择（A_1）之后，原问题简化为：
    - 子问题 **A′**：对A中所有与 A_1 相容的活动进行安排
    - 显然： B′=B-{1}是活动安排问题 A′ 的一个最优解
      - 证明：若存在A′的最优解C′，包含比B'更多的活动
      - 则：C′+{1}将包含比B更多的活动，矛盾
    - 因此：每一步做出的贪心选择都将当前问题简化为
      - 规模更小且与原问题具有相同形式的子问题
  - 对贪心选择次数进行数学归纳：对于任意活动安排问题
    - 必然存在一个以贪心选择开始的最优活动安排方案
    - 因此贪心选择次数，就是全局最优解

贪心算法求解活动安排问题

- 算法设计：设活动集合A中的活动总数为n
  - 设：数组B[1:n]存储所选择的活动
    - 若活动i在集合B中，则B[i]=1；否则B[i]=0
  - 设：各活动的起止时间分别存储于数组S[1:n]和F[1:n]中
    - 且数组F[1:n]按活动结束时间的非递减顺序排列
  - 按下标顺序依次从F[1:n]中选择活动 i 尝试加入集合B
    - 设：变量k记录B中最近一次加入的活动
    - 思考：k与B中已有活动的关系？（提示： F[1:n]有序）
    - F[k]总是集合B中当前所有活动的最晚结束时间

贪心算法求解活动安排问题

- 算法设计（续）
  - 按下标顺序依次从F[1:n]中选择活动 i 尝试加入集合B
    - 设：变量k记录B中最近一次加入的活动
  - 然后依次检查活动 i 是否与B中已有的所有活动相容
    - 活动 i 与B中所有活动相容的充要条件是： S[i]≥F[k]
      - 即：活动 i 的开始时间不早于k的结束时间
    - 若相容：则活动 i 取代 k 成为最近加入B的活动
    - 若S[i]<F[k]：则放弃活动 i
  - 继续检查F[1:n]中下一个活动与集合B中活动的相容性
  - 直到所有活动均已检查完毕，程序结束

贪心算法求解活动安排问题

```
void greedy(int S[], int F[], int B[], int n){
    B[0] = 1;       // 贪心选择：A1加入集合B
    int k = 1;      // k记录最近一次加入B的活动
    for (int i = 1; i < n; i++){
        if(S[i] >= F[k]){
            B[i] = 1;  k = i;
        }
        else{
            B[i] = 0;
        }
    }
}
```

算法复杂度？ O(n)

活动安排问题小结

- 算法分析
  - 算法复杂度：为使最多的活动相容地使用公共资源
    - 对所有活动按结束时间的非递减顺序排序：O(nlogn)
    - 采用贪心选择策略进行活动安排：O(n)
- 思考：本例中的贪心选择策略是什么？意义何在？
  - 贪心选择策略：按 F_i 递增顺序选择最早结束的相容活动
  - 意义：这种方式可以为后序活动的预留尽可能多的时间
    - 使剩余时间段极大化以便安排尽可能多的相容活动
- 思考：对活动安排问题采用贪心算法为何能求得全局最优解？
  - 对贪心选择次数的全局最优性采用数学归纳法证明
    - 活动安排问题具有贪心选择性质！

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 ☹
  - 经验：可用贪心算法求解的问题一般具有如下特征
    - **最优子结构性质**
    - **贪心选择性质**
  - 绝大多数可用贪心算法求解的问题都具有这两个性质

贪心选择性质

- 贪心选择性质是贪心算法可行的第一个基本要素
- 贪心选择性质的定义
  - 所求问题的全局最优解可以通过一系列局部最优的选择得到
- 要想确定一个问题是否具有贪心选择性质
  - 必须证明每一步的贪心选择最终能够导致问题的全局最优解
- 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迭代地做出贪心选择
    - 每一次贪心选择就将所求问题简化为规模更小的子问题

最优子结构性质

- 最优子结构性质是贪心算法可行的第二个基本要素
- 回顾：什么是最优子结构性质？
  - 所求问题的全局最优解包含其子问题的最优解
- 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
  - 二者的共同点：都要求问题具有最优子结构性质
  - 问题：是否能用动态规划求解的问题也能用贪心算法求解?
    - 通过比较两个经典的组合优化来说明二者的主要差别
    - 背包问题和最优装载问题

小结：贪心算法的基本要素

- 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

2. 背包问题

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：$c>0$, $w_i>0$, $v_i>0$, $1\le i\le n$
  - 要求找出一个n元**0/1向量**$(x_1, x_2, \ldots, x_n)$
  - 满足：$\sum_{i=1}^{n} w_i x_i \le c \quad (x_i \in \{0,1\})$
  - 使得下式最大化：$\sum_{i=1}^{n} v_i x_i$
- **背包问题:**
  - 给定：$c>0$, $w_i>0$, $v_i>0$, $1\le i\le n$
  - 要求找出一个n元**实数向量**$(x_1, x_2, \ldots, x_n)$
  - 满足：$\sum_{i=1}^{n} w_i x_i \le c \quad (0 \le x_i \le 1)$
  - 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是能装入背包并使价值总和最大的物品集合
  - 则：A[k]=A-{k}表示n-1个物品(1,2,…, k-1, k+1,…,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
  - 则：从A中拿出所含物品k的那部分重量（w_k'）
  - 剩余的就是从n-1个原有物品(1, 2,…, k-1, k+1,…, 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（nlogn）

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，每磅价值6元
  - 物品2重20磅，价值100元，每磅价值5元
  - 物品3重30磅，价值120元，每磅价值4元
- 对于0-1背包问题而言
  - 按照贪心策略，应首先选择物品1
  - 显然最优解是选取物品2和3，包含物品1的可能解都是次优的
- 对于部分背包问题而言
  - 按照贪心策略，首先选择物品1可以达到全局最优
  - 最优解：物品1（10磅）+物品2（20磅）+物品3（20磅）

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

3. 最优装载问题

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$\mathbf{w_i}$，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：$c>0$, $w_i>0$, $v_i>0$, $1\le i\le n$
  - 要求：找出一个n元0/1向量$\mathbf{x}=(x_1,x_2,\ldots,x_n)$ 其中 $x_i\in\{0,1\}$
  - 使得：$\sum_{i=1}^{n} w_i x_i \le c$，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：$X=(x_1, x_2,\ldots, x_n)$是最优装载问题的一个最优解

设：$k = \min\{ i \mid x_i=1, 1 \le i \le n \}$ (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$1 \le k \le n$
- 当k=1时， X 是一个满足贪心选择性质的最优解
- 当k>1时，取 $y_1=1; y_k=0; y_i=x_i, 1 < i \le n, i \ne k$, 则：

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

- 因此： $Y=(y_1, y_2, \ldots, y_n)$ 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：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！

最优装载问题

- 最优装载问题具有最优子结构性质
  - 设：$X=(x_1,x_2,\ldots,x_n)$是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：$X'=(x_2, x_3,\ldots,x_n)$ 是如下子问题的最优解
    - 载重量为$c-w_1$，集装箱为$\{2,3,\ldots,n\}$的最优装载问题
  - 因此：最优装载问题具有最优子结构性质！
- 贪心算法描述
  - 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));
    sort(w, R, n); // 根据w递增排序，R记录调整后的序号
    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];
    } // free(R);
}
```

时间复杂度：O(nlogn)

4. 哈夫曼编码

（ Huffman Code ）

哈夫曼编码

- 哈夫曼编码
  - 是广泛应用于数据文件压缩的一种十分有效的编码方法
  - 其压缩率通常在20%~90%之间
- 哈夫曼编码算法
  - 使用字符在文件中出现的频率表作为输入
  - 目标是：构建一个用0/1位串表示各字符的最优表示方式
  - 基本思路是
    - 为出现频率较高的字符赋予较短的编码
    - 为出现频率较低的字符赋予较长的编码
    - 由此实现对文件总编码长度的压缩

等长编码

例如：需将文字“ABACCDA”转换成电文

分析：文字中有四种字符，用2位二进制便可分辨

编码方案	A	B	C	D
等长编码	**00**	**01**	**10**	**11**

则上述文字的电文为：00010010101100　　共14位

译码时：只需每2位一译即可

特点：等长等频率编码，译码容易，但电文不一定最短

不等长编码

例如：需将文字“ABACCDA”转换成电文

编码方案2	A	B	C	D
不等长编码	**0**	**00**	**1**	**01**

采用不等长编码，让出现次数多的字符用短码

则ABACCDA文字的电文为：000011010　　共9位

但无法译码：既可译为BBCCACA，也可译为AAAACCDA等

前缀码

例如：需将文字“ABACCDA”转换成电文

编码方案3

前缀码

A	B	C	D
0	110	10	111

采用不等长编码

- 出现次数多的字符用短码
- 且任一编码不能是另一编码的前缀

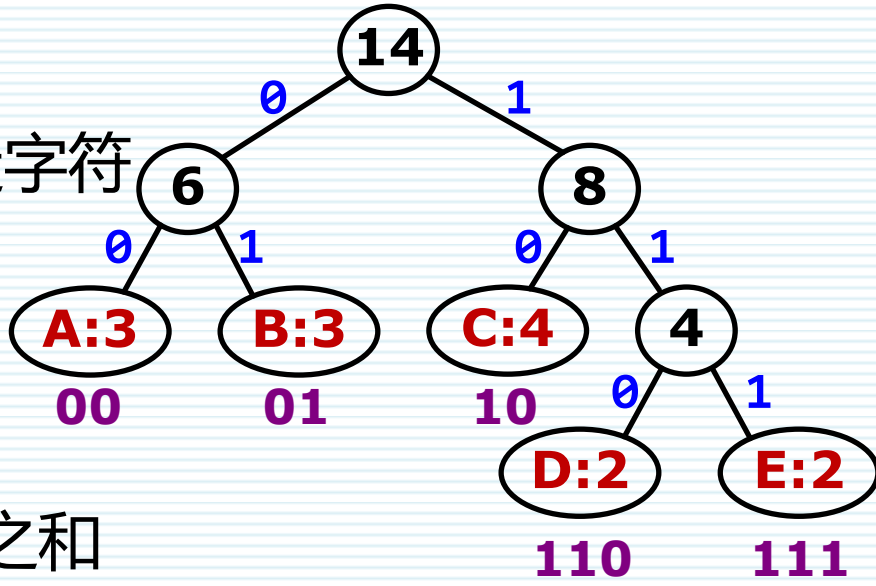
则ABACCDA文字的电文为：01100101011110　　共13位

前缀码（prefix code）

- 前缀码：对每一个字符规定一个0/1串作为其代码
  - 要求：**任一字符的代码都不是其他字符代码的前缀**
  - 这种编码称为前缀码（标准书面语， prefix-free code）
- 为什么要关注前缀码
  - 已经证明：通过字符编码获得的最优数据压缩方式总可用某种前缀编码来表达，因此算法设计时考虑前缀码不失一般性
  - 编码的前缀性质可以简化编解码方式
    - 编码：只要将文件中表示每个字符的编码并置起来即可
    - 解码：只需对第一个编码进行解码，然后迭代进行解码
      - 由于是前缀码，因此被编码文件的起始编码是确定的

前缀码的二叉树表示

- 前缀码可以采用二叉树进行表示
  - 利用二叉树的性质，可以很方便地对前缀码进行解码
- 前缀码二叉树的数据结构
  - 二叉树的叶节点表示一个特定字符
    - 出现的频率（即权重）
  - 二叉树的内节点表示
    - 其子树中所有叶子的频率之和
  - 字符的编码为从根至该字符的路径
    - 路径上的字符0表示：转向左子节点
    - 路径上的字符1表示：转向右子节点



最优前缀码

- 平均编码长度
  - 设：字母表A中的某个字符c在文件中出现的频率为：f(c)
  - 对于给定的编码方案，设对应的二叉树表示为T
  - 则：字符c在T中的深度 $d_T(c)$ 就是该字符的编码长度
  - 该编码方案的平均码长定义为：$B(T)=\sum_{c\in A} f(c)\cdot d_T(c)$
  - 即：编码该文件需要的位（bit）数，也称为树T的代价
- 最优前缀码
  - 使平均编码长度达到最小的前缀编码方案
  - 称为给定字符集A的最优前缀码

最优前缀码的性质

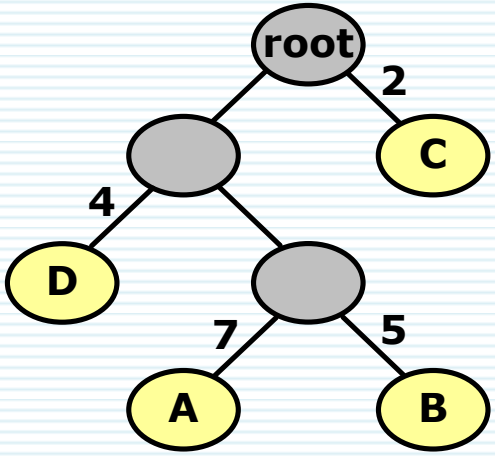
- 表示最优前缀码的二叉树总是一棵完全二叉树
  - 即：树中任何一个内节点都有2个子节点
- 如果A是包含待编码字符的字母表
  - 则：表示最优前缀编码的树T中恰有|A|片叶子
    - 每个叶节点表示字母表中的一个字母
  - 表示最优前缀编码的树T中共有|A|-1个内节点
- 若给定对应一种前缀编码的二叉树T
  - 容易计算出编码一个文件所需要的位数
  - 对所有叶节点求和：叶节点深度 x 叶节点在文件中的频率

哈夫曼编码

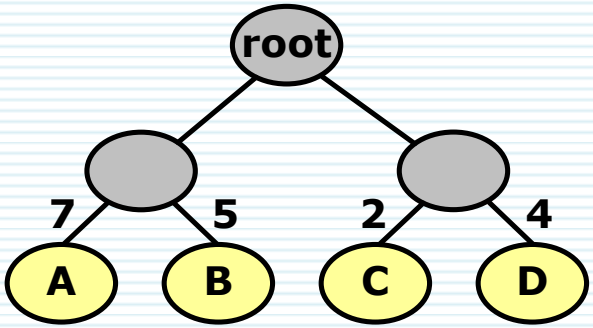
- 哈夫曼提出了一种构造最优前缀码的贪心算法
  - 由此产生的编码方案称为哈夫曼编码
- 哈夫曼算法
  - 以自底向上的方式构造表示最优前缀码的二叉树T
    - 这棵树通常被称为哈夫曼树
  - 算法从|A|个节点开始，选择权重最小的节点进行合并
  - 执行|A|-1次的合并运算后，产生最终所要求的树T

哈夫曼树（Huffman Tree）

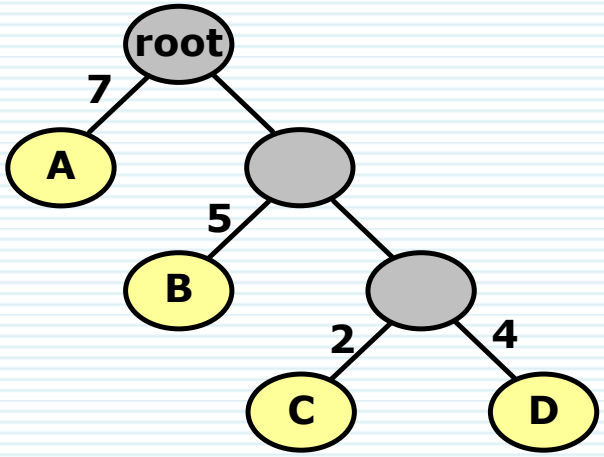
- 哈夫曼树的定义：平均码长（带权路径长度）最短的二叉树
  - 假设给定n个权值：{f(1), …, f(n) }
  - 据此构造一棵有n个叶结点的二叉树 $B(T)=\sum_{c\in A} f(c)\cdot d_T(c)$
  - 其中B(T)最小的二叉树称为Huffman树



B(T) = 46



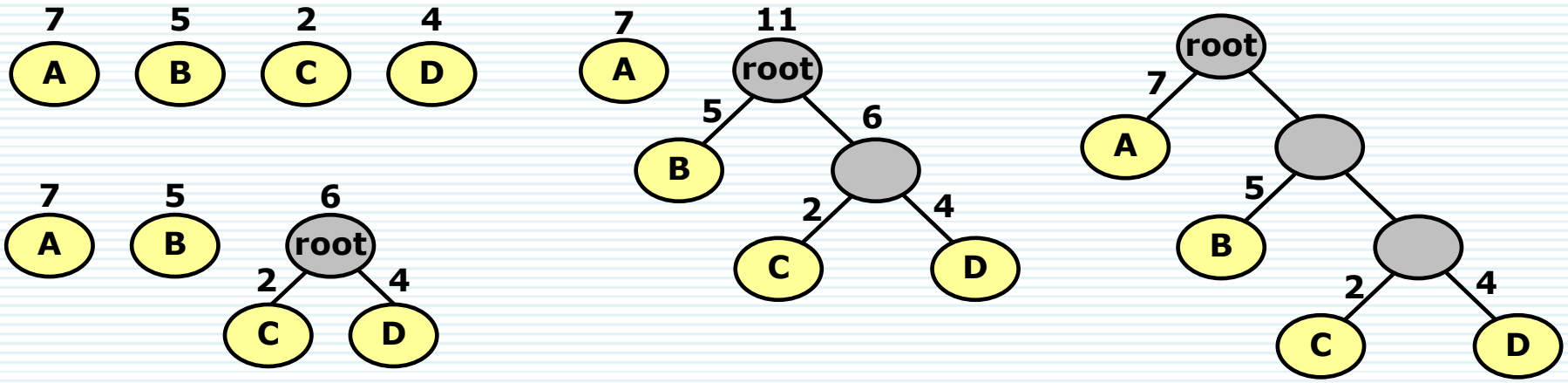
B(T) = 36



B(T) = 35

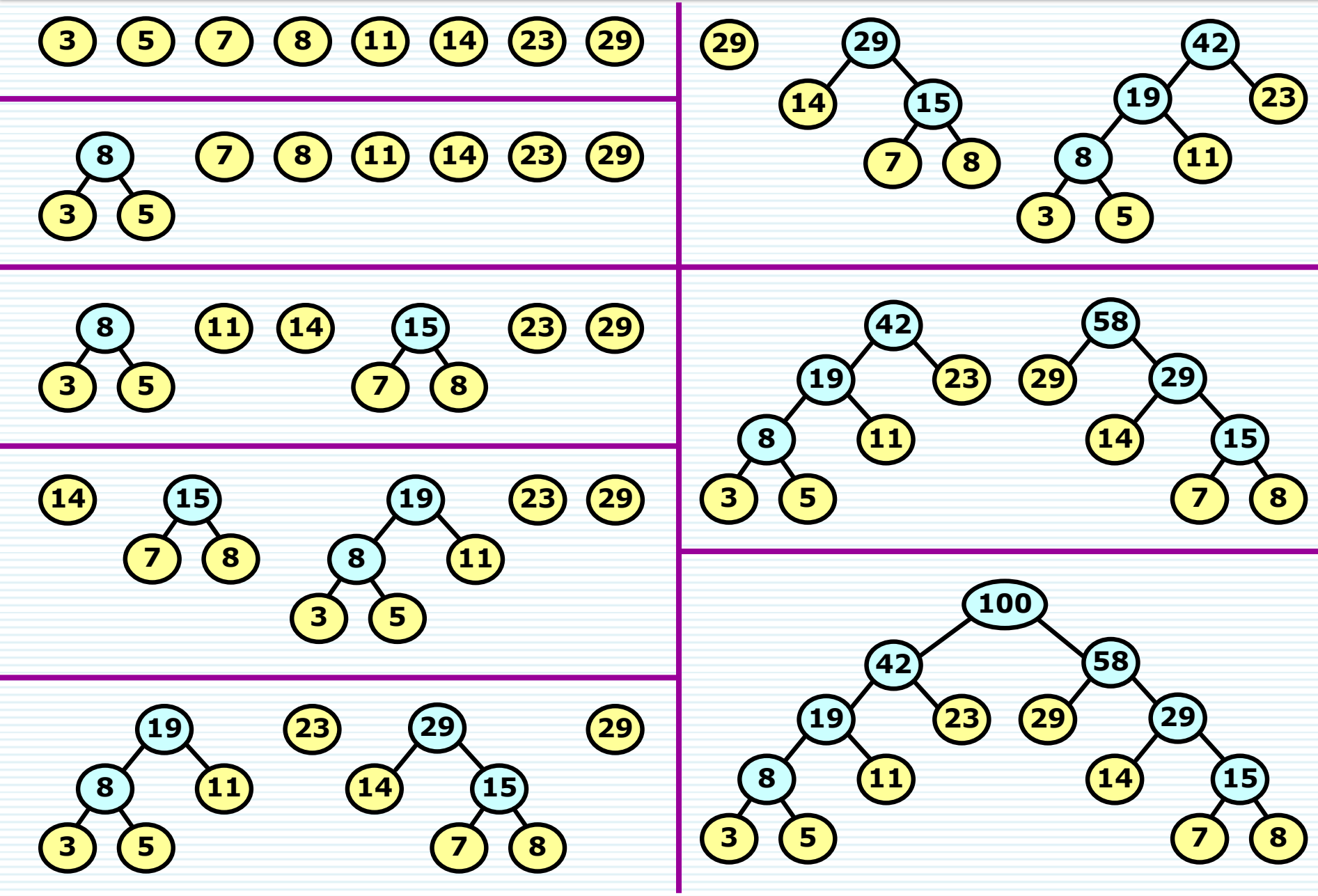
Huffman树

Huffman树的构造方法：Huffman算法



1. 根据给定的n个权值：$\{w_1,w_2,\ldots\ldots w_n\}$，构造n棵只含根结点的二叉树，令每棵树的权值为相应的结点权值（w_j）
2. 在森林中选取两棵根结点权值最小的树作为左右子树，构造一棵新的二叉树，新树根节点权值为其左右子树根结点权值之和
3. 在森林中删除这两棵树，同时将新得到的二叉树加入森林中
4. 重复上述两步直到森林中只含一棵树为止，这棵树即哈夫曼树

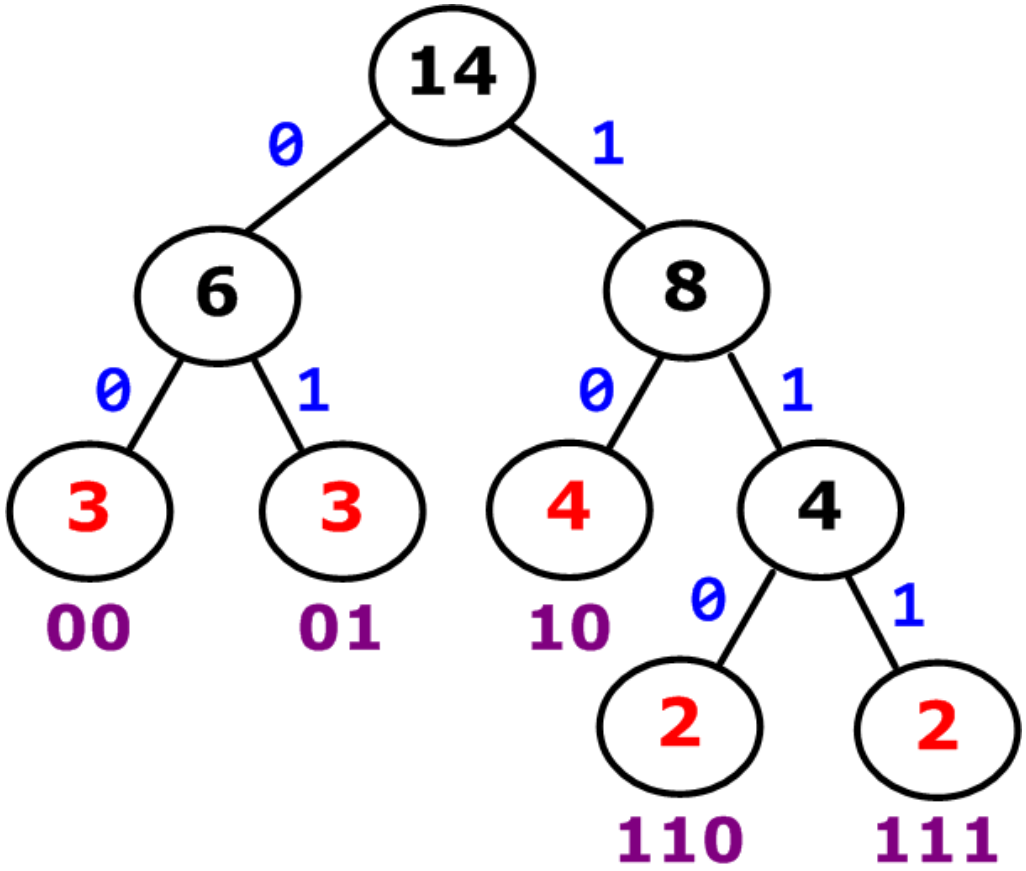
Huffman算法示例：w={5, 29, 7, 8, 14, 23, 3, 11}



Huffman编码方法

- 设：字母表中有n种字符
  - 每种字符在电文中出现的次数为 f(i)，其编码长度为d(i)
  - 则整个电文总长度为Σf(i)d(i)　(i=1,2,．．．n)
- 要使Σf(i)d(i)最小（电文长度最短）
  - 以字符出现的次数为权值构造一棵Huffman树
    - 规定左分支编码为0，右分支编码为1
  - 则字符的编码为：从根节点出发到该字符所在的叶结点
    - 经过的路径上的分支编号构成的序列
- 用Huffman树编出来的码，称为Huffman编码

Huffman编码



A:3 00

B:3 01

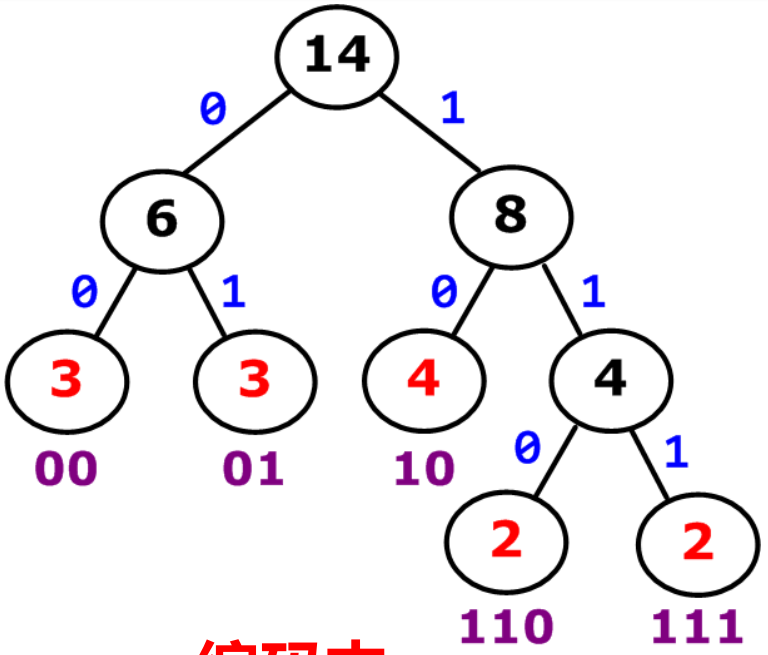
C:4 10

D:2 110

E:2 111

Huffman编码的译码操作

- 从待译码电文中逐位读取编码
  - 从Huffman树根开始
    - 若编码是0： 则沿lchild下行
    - 若编码是1： 则沿rchild下行
    - 到达叶结点：则译出一个字符
  - 重复上述步骤，直到电文结束



编码本

T : 3	00
; : 3	01
A : 4	10
C : 2	110
S : 2	111

明文是：CAS;CAT

编码为：11010111011101000

密文是：1101000

译文为：CAT

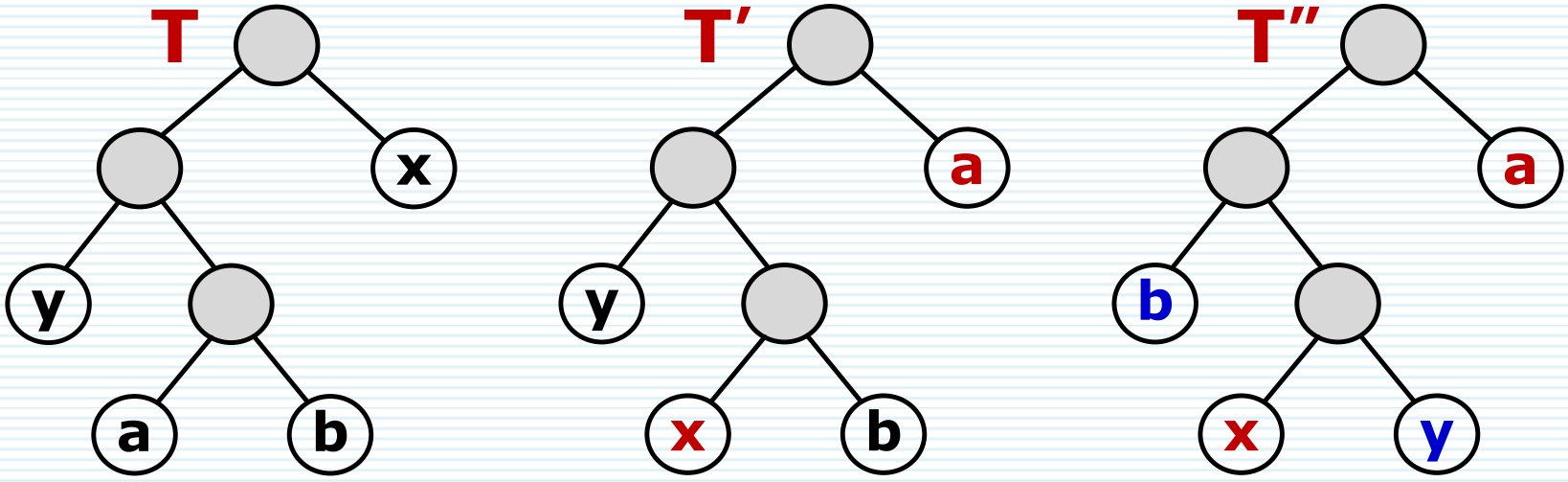
哈夫曼算法的正确性证明

- 思考：如何证明哈夫曼算法的正确性？
  - 证明最优前缀码问题具有贪心选择性质和最优子结构性质
- 思考：什么是最优前缀码的贪心选择性质？
  - 可以通过迭代进行贪心选择来完成最优二叉树的构造
    - 每一步选择两个频率最低的子树进行合并
- 问题：什么是最优前缀码的最优子结构性质？
  - 设：T是字符集A的最优前缀码对应的完全二叉树
  - 设：x和y是T中的两个兄弟叶节点（z是其父节点）
  - 令：$T'=T-\{x,y\}$ ； $A'=(A-\{x,y\})\cup\{z\}$
  - 则：T'为字符集A'的一个最优前缀码

哈夫曼算法的正确性证明

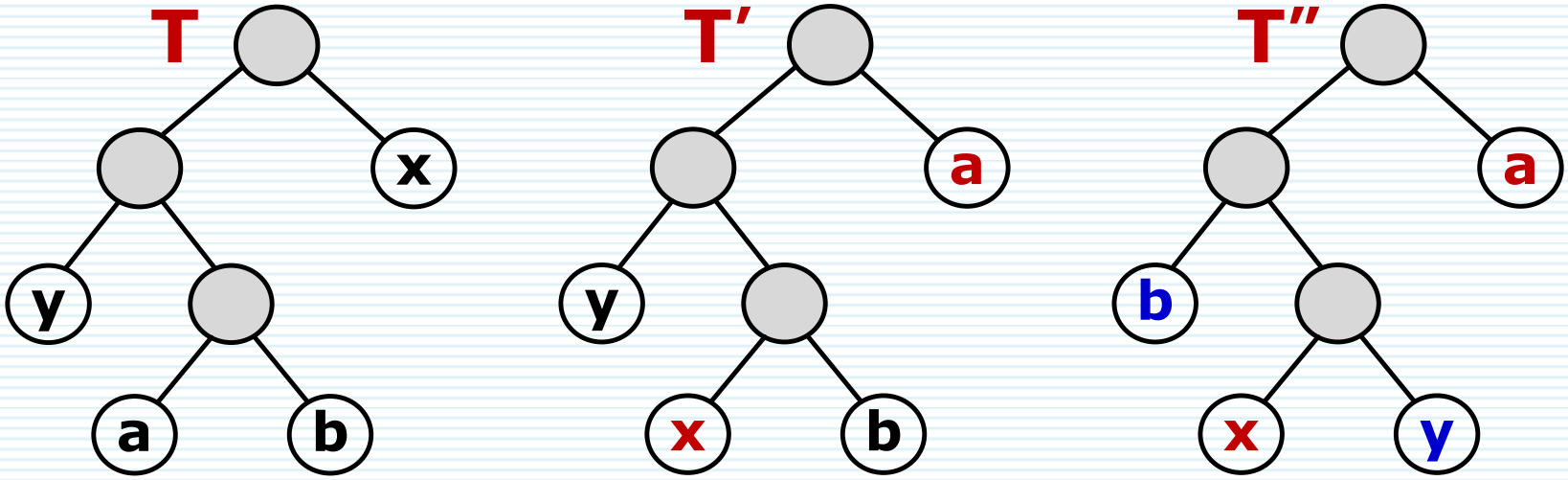
- 引理1
  - 设：字符集**A**中字符**c**的频率为**f(c)**
  - 设：**x**和**y**是**A**中具有最小频率的两个字符
  - 则：存在**A**的最优前缀编码使得
    - **x**和**y**的编码长度相同，且仅最后一位编码不同
- 证明思路
  - 设：树**T**表示**A**的任意一种最优前缀码
  - 需证明：对**T**做适当修改后可以得到一棵新的二叉树**T''**
    - 使字符**x**和**y**在树**T''**中成为具有最大深度的兄弟叶节点
    - 同时**T''**所表示的前缀码也是**A**的一个最优前缀码
  - 则字符**x**和**y**在**T’’**中的编码等长，并且仅最后一位不同

证明引理1



- 设：**x**和**y**是**A**中具有最小频率的两个字符
- 设：**a**和**b**为树**T**中具有最大深度的兄弟叶子节点
- 不失一般性，假设：f(a)≤f(b), f(x)≤f(y)
  - 由前提假设可知：f(x)≤f(y)≤f(a)≤f(b)
  - 交换**a**和**x**在树**T**中的位置产生树**T′**
  - 交换**b**和**y**在树**T**中的位置产生树**T″**

证明引理1



- 树T和树T′所表示的前缀码的平均码长之差为

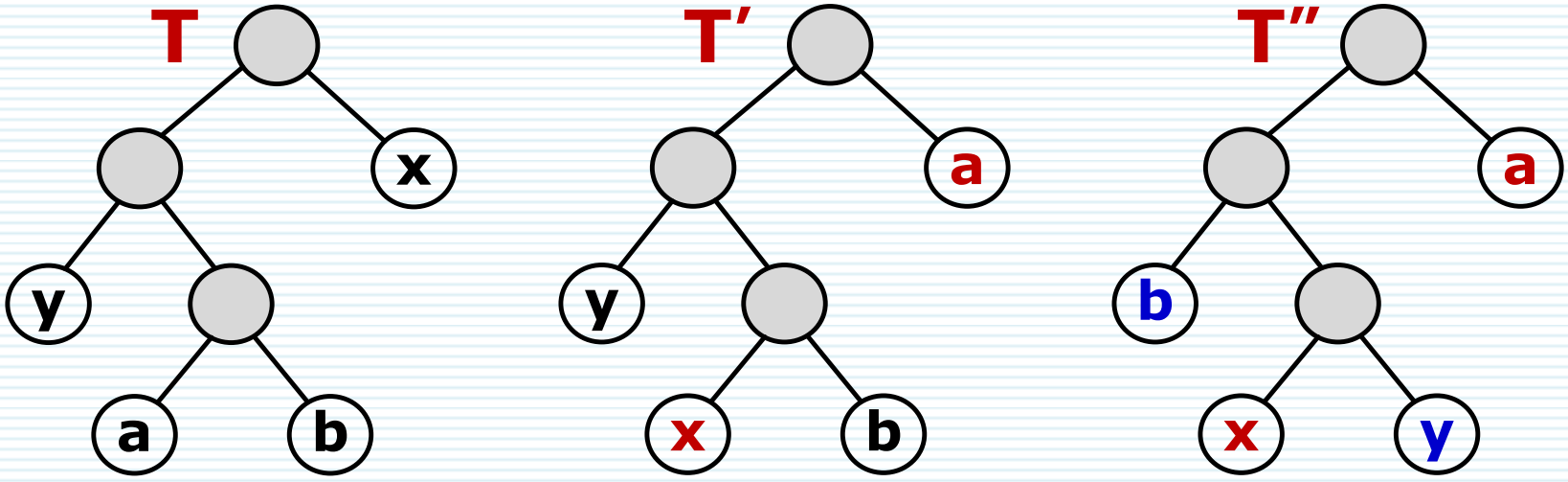
$$B(T)-B(T')=\sum_{c\in A} f(c)\cdot d_T(c)-\sum_{c\in A} f(c)\cdot d_{T'}(c)$$

$$=f(x)\cdot d_T(x)+f(a)\cdot d_T(a)-f(x)\cdot d_{T'}(x)-f(a)\cdot d_{T'}(a)$$

$$=f(x)\cdot d_T(x)+f(a)\cdot d_T(a)-f(x)\cdot d_T(a)-f(a)\cdot d_T(x)$$

$$=\big(f(a)-f(x)\big)\cdot\big(d_T(a)-d_T(x)\big)\geq 0$$

证明引理1



- 同理可证在T'中交换y与b的位置也不增加平均码长
  - 即： $B(T') - B(T'') \geq 0 \Rightarrow B(T) \geq B(T'')$
  - 根据假设：T为最优前缀码，即 B(T) ≤ B(T″)
  - 因此：B(T)=B(T″)，引理得证
  - 存在A的最优前缀码使x和y编码长度相同，仅最后一位不同

证明：最优前缀码问题具有贪心选择性质

- 引理1：设：x和y是A中具有最小频率的两个字符
  - 则：存在A的最优前缀编码使得
    - x和y的编码长度相同且仅最后一位编码不同

- 证明：由引理1可知
  - 采用合并的方式构造一棵最优树的过程
    - 可以（贪心地）从合并两个频率最低的字符开始
    - 并且可通过迭代进行贪心选择来完成最优二叉树的构造
  - 由此证明了构造哈夫曼编码的问题具有贪心选择性质

证明：最优前缀码问题具有最优子结构性质

- 引理2：设**T**是字符集**A**的最优前缀码对应的完全二叉树
  - A中字符c的频率为f(c)
  - 设：x和y是T中的两个兄弟叶节点，z是它们的父节点
  - 若：将z看作是具有频率 f(z)=f(x)+f(y) 的字符
  - 设：**T′**=T-{x,y} ； **A′**=(A-{x,y})∪{z}
  - 则：树 **T′** 表示字符集 **A′** 的一个最优前缀码

- 思考：为什么要引入引理2？
  - 已知：**T**是**A**的最优解，且**T′** ⊂ **T**，**A′** ⊂ **A**
  - 若引理2成立，则最优子结构性质成立

证明：最优前缀码问题具有最优子结构性质

- 引理2：设**T**是字符集**A**的最优前缀码对应的完全二叉树
  - A中字符c的频率为 f(c)
  - 设：x和y是T中的两个兄弟叶节点，**z**是它们的父节点
  - 若：将z看作是具有频率 f(z)=f(x)+f(y) 的字符
  - 设：**T′**=T-{x,y} ； **A′**=(A-{x,y})∪{z}
  - 则：树 **T′** 表示字符集 **A′** 的一个最优前缀码

- 思考：如何证明引理2？
  - 反证法：假设 T′ 不是A′ 的最优前缀码
  - 则不妨设：存在 A′ 的前缀码 T″ 使得：B(T″)<B(T′)
  - 将x和y加入树T″中作为z的左右孩子节点
  - 则有： $B(T'')+f(x)+f(y) < B(T')+f(x)+f(y)$

证明引理2

- 首先证明T的平均码长：**B(T) = B(T′)+ f(x) + f(y)**
  - 对于任意 c∈A-{x,y} 有：$d_T(c) = d_{T'}(c)$
    - 显然：$f(c)\, d_T(c) = f(c)\, d_{T'}(c)$
  - 由题意知：$f(z) = f(x) + f(y)$
  - 由题意知：$d_T(x) = d_T(y) = d_{T'}(z)+1$

$$f(x)\cdot d_T(x)+ f(y)\cdot d_T(y) = \big(f(x)+ f(y)\big)\cdot\big(d_{T'}(z)+1\big)$$
$$= f(x)+ f(y)+ f(z)\cdot d_{T'}(z)$$

  - 由此可知：**B(T) = B(T′) + f(x) + f(y)**

最优前缀码问题的最优子结构性质

- 接下来采用反证法证明引理2
  - 假设：**T′**不是**A′** 的最优前缀码
  - 不妨设：存在**T′′**表示**A′**的前缀码，使得B(T′′)<B(T′)
  - 由于： **z** 可以被看作 **A′** 的一个字符
    - 因此：**z** 是 **T′′** 的一个叶节点
  - 若将**x**和**y**加入树 **T′′** 中作为**z**的左右孩子节点
  - 则可以得到表示字符集A的前缀码的二叉树T′′′，且有：

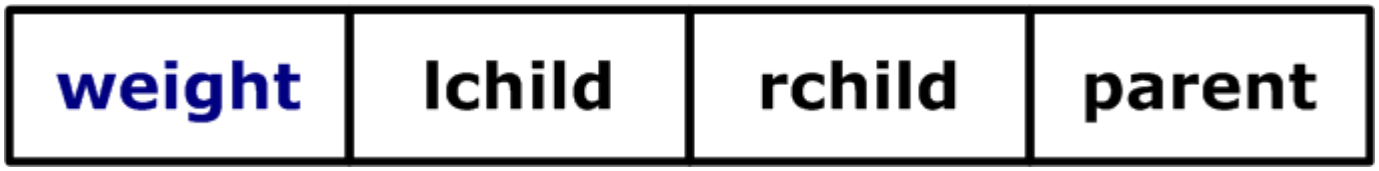
$$B(T''') = B(T'') + f(x) + f(y) < B(T') + f(x) + f(y) = B(T)$$

  - 与**T**是最优前缀码矛盾，故子问题A′的解T′是最优的
  - 因此哈夫曼算法具有最优子结构性质

Huffman算法

- Huffman算法流程
  - 首先将n个叶结点存入大小为2n-1的数组
    - 结点的父指针均置为-1（表示该结点为根结点）
  - 对数组中元素进行循环处理（直至数组被填满）
    - 从现有子树的根结点中选择两个权重最小者
    - 构造新子树的根结点加入数组
    - 更新新根节点的孩子指针和权重值
    - 同时修改选中结点的父指针：指向新的根结点

Huffman算法



- Huffman树的结点数据结构
  - 权值
    - 叶节点权值已知
    - 中间结点的权值在子树合并时通过计算得到
  - 左右孩子指针：左右子树根节点的下标
  - 父结点指针
    - 记录当前结点隶属于哪个中间结点
    - 根节点的父指针为-1

Huffman：子树选择算法

```
void select_subtree(Hnode* pht, int n, int *subA, int *subB){
    int  id, idxa = -1, idxb = -1;
    int  wa = INT_MAX,  wb = INT_MAX;  // wa最小值 wb次小值
    for(id = 0; id <= n; id++){
        if(pht[id].parent == -1){
            if( pht[id].weight < wa ){
                idxb = idxa;  idxa = id;  wa = pht[id].weight;
            }
            else if(pht[id].weight < wb ){ //
                idxb = id;  wb = pht[id].weight;
            }
        }
    }
    *subA = idxa;    *subB = idxb;     return;
}
```

算法复杂度：$O(n^2)$

构建Huffman树

```
Hnode* create_htree( int weights[],  int n ){
    Hnode* pht;  int i,  subA,  subB,  ntotal = 0;
    ntotal = (2 * n) - 1;                  // Huffman树的结点总数
    pht = (HNode *) malloc( sizeof( HNode ) * ntotal );
    for( i = 0; i < ntotal; ++i ){  // HTree初始化
        pht[i].weight = (i < n) ? weights[i] : 0;
        pht[i].lchild = -1; pht[i].rchild = -1; pht[i].parent = -1;
    }
```

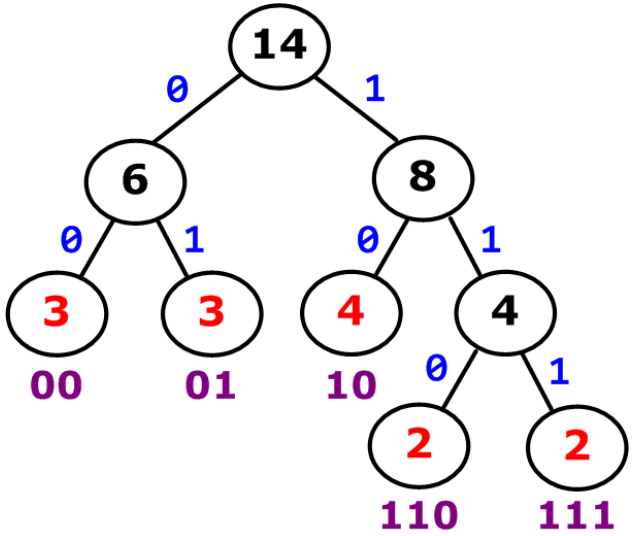
```
    for( i = n; i < ntotal; ++i ){  // 构建Huffman树
        select_subtree( pht,  (i-1),  &subA, &subB );
        pht[subA].parent = i; pht[subB].parent = i;
        pht[i].lchild = subA; pht[i].rchild = subB;
        pht[i].weight = pht[subA].weight + pht[subB].weight;
    }
    return pht;
}
```

哈夫曼编码方法

- 通过回溯生成字符的哈夫曼编码（编码本）

1. 选择哈夫曼树的某个叶结点（设其下标为 idxa）
2. 利用parent指针找到其父结点（设其下标为 idxb ）
3. 利用父结点的孩子指针域判断该结点是左孩子还是右孩子
   - 若该结点是左孩子（ lchild== idxa），则生成代码0
   - 若该结点是右孩子（ rchild== idxb），则生成代码1
4. 重复步骤(2)~(3) 直至回溯到根节点，得到一个0/1序列
   - 思考：这个0/1序列是否为该字符的Huffman编码？
   - 该序列是Huffman编码的逆序：将其反序得到字符编码
5. 重复步骤(1)~(4)，实现对全部叶节点的编码

哈夫曼编码的存储结构



A:4	10	0	'A'	"10"
B:3	01	1	'B'	"01"
C:3	00	2	'C'	"00"
D:2	110	3	'D'	"110"
E:2	111	4	'E'	"111"

```
#define LEN 100                 // 待编码字符个数
typedef struct{
    char ch;                    // 存储字符
    char code[LEN+1];           // 存放编码
}TCode;
TCode CodeBook[LEN];            // 编码本
```

哈夫曼编码算法

```
// 根据哈夫曼树pht求哈夫曼编码表book
void encoding (PHT pht, TCode *book, int n){
    int i, c, p, start;     // start表示编码在cd中的起始位置
    char cd[n+1];   cd[n]='\0';  // 临时存放编码
    for(i = 0, i < n, i++){          // 依次求叶子pht[i]的编码
        book[i].ch = pht[i].ch;  // 读入叶结点pht[i]对应的字符
        start = LEN;   c = i;        // 从叶结点pht[i]开始上溯
        while( p = pht[c].parent > 0){
            if(pht[p].lchild == c){ cd[--start]='0'; }
            else{ cd[--start] ='1; }
            c = p; } // 继续上溯直到根节点
        strcpy(book[i].code, &cd[start]); // 复制编码位串
} }
```

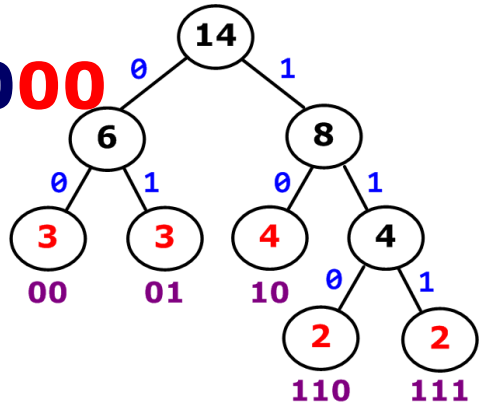
Huffman编码的译码操作

明文是：CAS;CAT

编码为：11010111011101000

密文是：1101000

译文为：CAT



T : 3	00
; : 3	01
A : 4	10
C : 2	110
S : 2	111

- 从待译码电文中逐位读取编码
  - 从Huffman树根开始
    - 若编码是**'0'**： 则沿lchild下行
    - 若编码是**'1'**： 则沿rchild下行
  - 若到达叶结点：则译出一个字符
  - 重复上述步骤，直到电文结束

哈夫曼解码算法

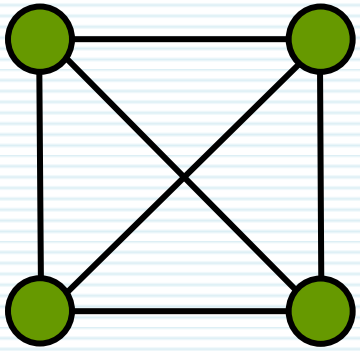
```
void decoding(PHT pht, char* codes, int n){
   int  i = 0, p = 2*n - 2;       // 从根结点开始
   while(codes[i]!='\0'){     // 当要解码的串没有结束时
      while(pht[p].lchild != -1 && pht[p].rchild != -1){
         if (codes[i]=='0') p = pht[p].lchild;
         else p = pht[p].rchild;
         i++;
      }
      printf("%c", pht[p].ch);  p = 2*n-2;
   }
   printf("\n");
}
```

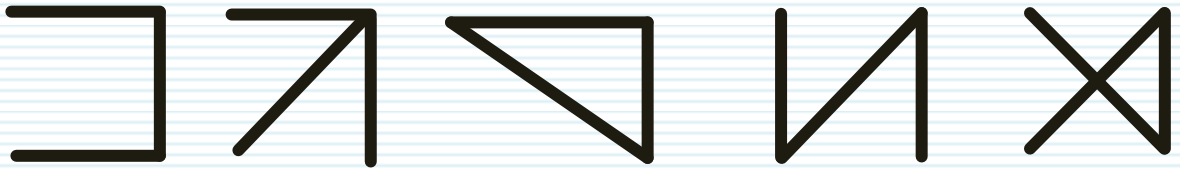
5. 最小生成树

（Minimum Spanning Tree）

最小生成树

- 设有无向连通图G=(V，{E})及其子图G′=(V，{T})
- 若子图G′满足如下三个条件：
  1. V(G′)=V(G) （顶点个数相同）
  2. G′是连通的
  3. G′中无回路





- 则称子图G′是图G的生成树

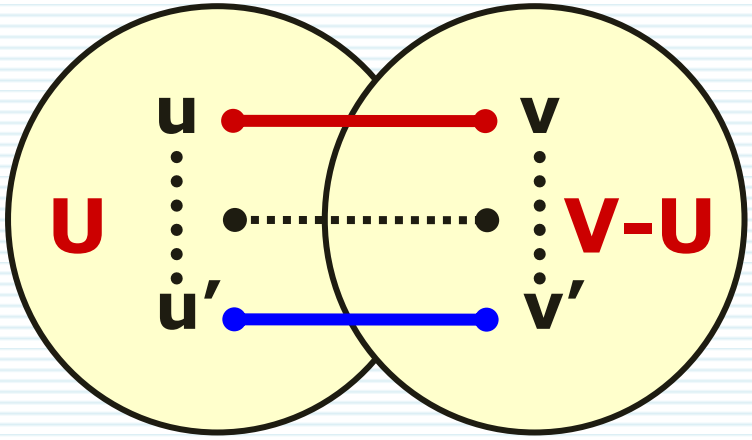
G的生成树

- 性质：对于具有n个顶点的无向连通图G而言
  1. 其任一生成树（G′）恰好包含含n-1条边
  2. 生成树不一定唯一

最小生成树

- 生成树的代价
  - 对图中每条边赋予一个权值（代价）则构成一个网
  - 定义：网的生成树的代价为图中各边的权值之和
- 最小生成树
  - 网的所有可能的生成树中代价最小者称为最小生成树
    - 简称 MST：Minimum Spanning Tree
  - 最小生成树也不一定唯一
- 典型应用：通信网络规划，交通规划

MST性质

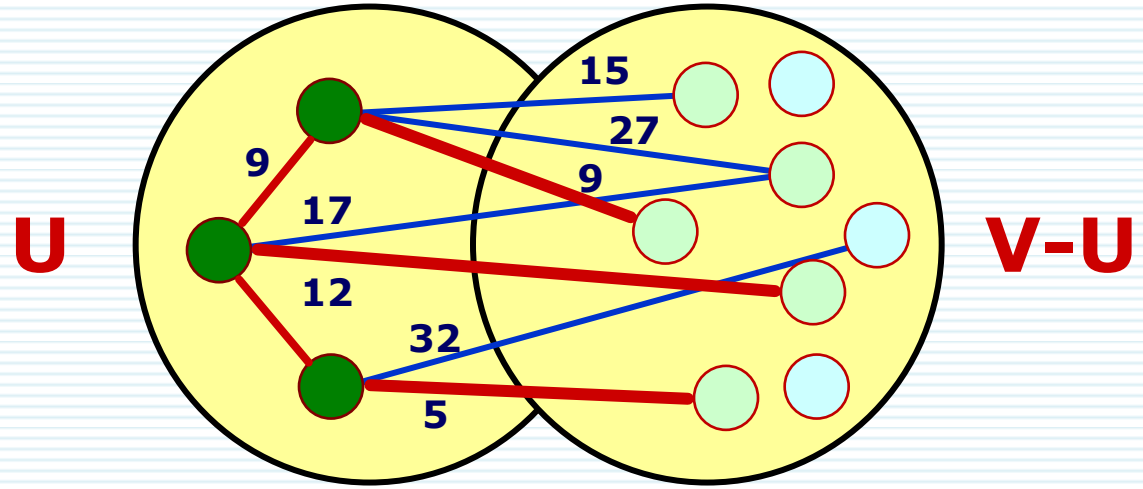


- 最小生成树（MST）的性质
  - 设：N=（V，{E}）是一个连通网
    - 将顶点分为两个不相交的非空子集：U和V-U
  - 若：(u, v)是一条具有最小权值的边（其中u∈U, v∈V-U）
    - 即：**(u, v) = min{ cost(x,y) | x∈U, y∈V-U }**
  - 则：必存在一棵包含边（u，v）的最小生成树

利用MST性质求解最小生成树问题

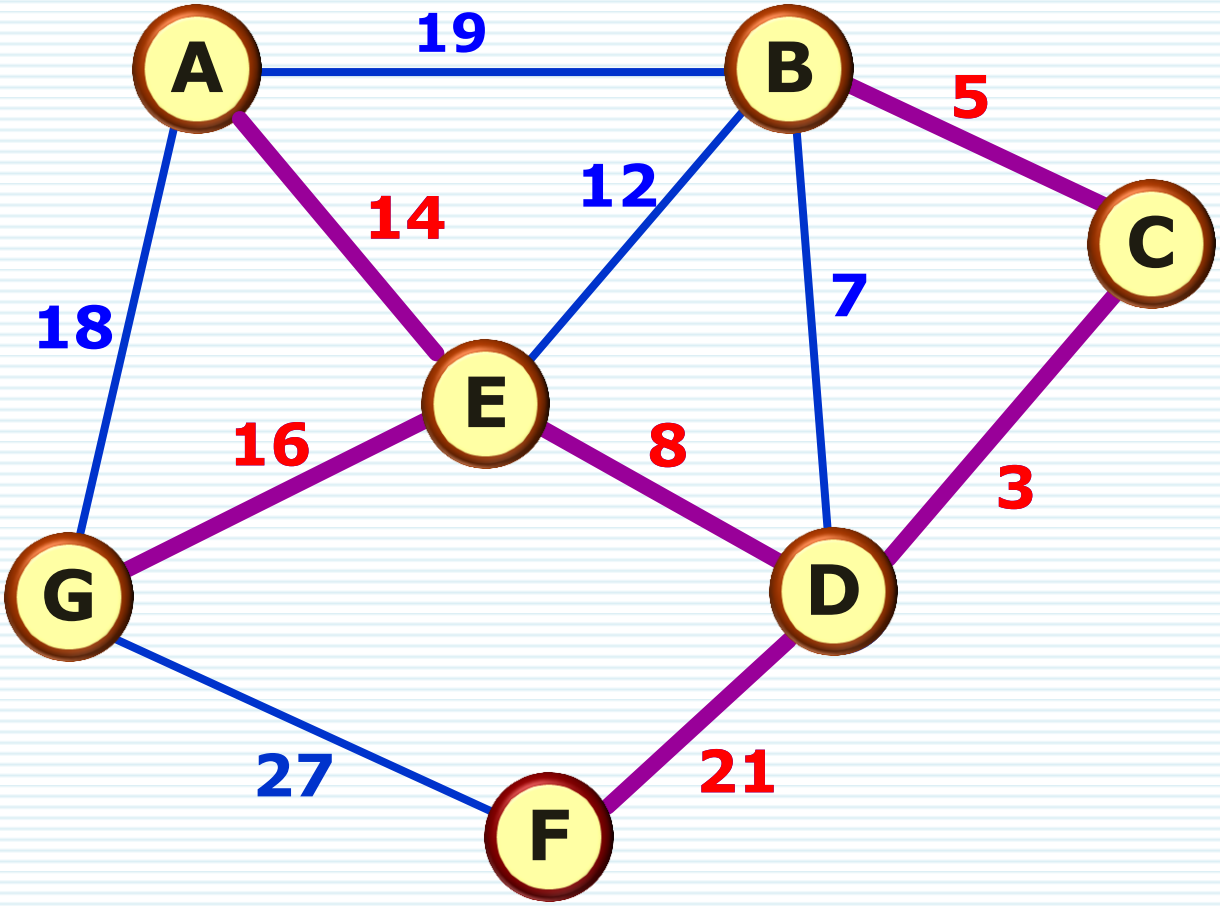
- MST性质
  - 若：**(u, v) = min{ cost(x,y) | x∈U, y∈V-U }**
    - 则必定存在一棵包含边（u，v）的最小生成树
  - MST性质实际上揭示了MST问题的贪心选择性质
- 因此可以利用贪心算法设计策略求解
  - 第一种贪心选择策略：子树生长法（断集法）
    - Prim算法
  - 第二种贪心选择策略：短边优先法（避圈法）
    - Kruskal算法

Prim算法



- Prim算法设计思想
  - 在生成树的构造过程中，图中 n 个顶点分属两个集合：
    - 已加入到生成树中的顶点集：U
    - 尚未加入到生成树中的顶点集：V-U
  - 在所有连通U和V-U的边中选取权值最小的边加入MST中

Prim算法



生成树代价 = 14+8+3+5+16+21 = 67

Prim算法的正确性

- 思考：如何证明Prim算法的正确性？
  - 贪心选择性质？ **MST性质即贪心选择性质**
  - 最优子结构性质？ **即证明每次贪心选择均产生部分最优解**
- 只要证明如下等价命题
  - 对 ∀ k<n：均存在一棵MST包含算法前k步选择的边
  - 在此基础上通过数学归纳可以证明Prim算法的正确性
- 思考：如何证明该等价命题？
  - 反证法：

Prim算法的正确性

- 等价命题：对 ∀ k<n均存在一棵MST包含算法前k步选择的边
  - 首先考察 k = 1 的情况
    - 设：$e_i = (v_0, v_i)$是所有关联 v_0 的边中权重最小的
    - 目标是证明：存在一棵包含边 e_i 的最小生成树
    - 设：T为一棵不包含 e_i 的最小生成树
    - 由MST的定义可知：$\mathbf{T \cup \{ e_i \}}$ 必然含有一条回路
    - 设：其中关联 v_0 的另一条边为 $e_j = (v_0, v_j)$
    - 令：$\mathbf{T' = (T - e_j) \cup \{ e_i \}}$
    - 则：**T'** 也是生成树，且 $\mathbf{W(T') \le W(T)}$ □

Prim算法的正确性

- 数学归纳
  - 假设：算法进行了k-1步
    - 生成树的边集为$\{e_1, e_2, \ldots, e_{k-1}\}$
    - 这些边的 k 个顶点构成顶点集合 U
  - 由归纳假设：存在G的一棵最小生成树T包含这些边
  - 设：算法第 k 步选择了顶点 v_{k+1}
    - 由MST性质知：v_{k+1} 到 U 中顶点的边权重最小
  - 设：这条边为 $e_k=(v_{k+1}, v_x)$，并假设T不含有 e_k
    - 则：将 e_k 加入 T 中将会形成回路
    - 这条回路有另外一条边 e，连接 U 与 V-U 中的顶点

Prim算法的正确性

- 数学归纳
  - 设：算法第 k 步选择了顶点 vk+1
  - 设：这条边为 $e_k=(v_{k+1}, v_x)$，并假设T不含有 e_k
    - 则：将 e_k 加入 T 中将会形成回路
    - 这条回路有另外一条边 e，连接 U 与 V-U 中的顶点
  - 令：$T' = (T - e) \cup \{e_k\}$
    - 则：T'是G的一棵生成树，包含 $\{e_1, e_2, \ldots, e_k\}$
    - 且：$W(T') \leq W(T)$ □

Kruskal算法

- Kruskal算法设计思想
  - 逐步向森林T中添加不与T中的边构成回路的当前最小代价边
  - 算法特点：以最小代价边主
- Kruskal算法思路
  - 先构造一个只含 n 个顶点的子图 T
  - 然后从权值最小的边 e_i 开始考察
    - 若添加 e_i 不使T中产生回路，则在T中加上这条边
  - 如此重复，直至加上 n-1 条边为止

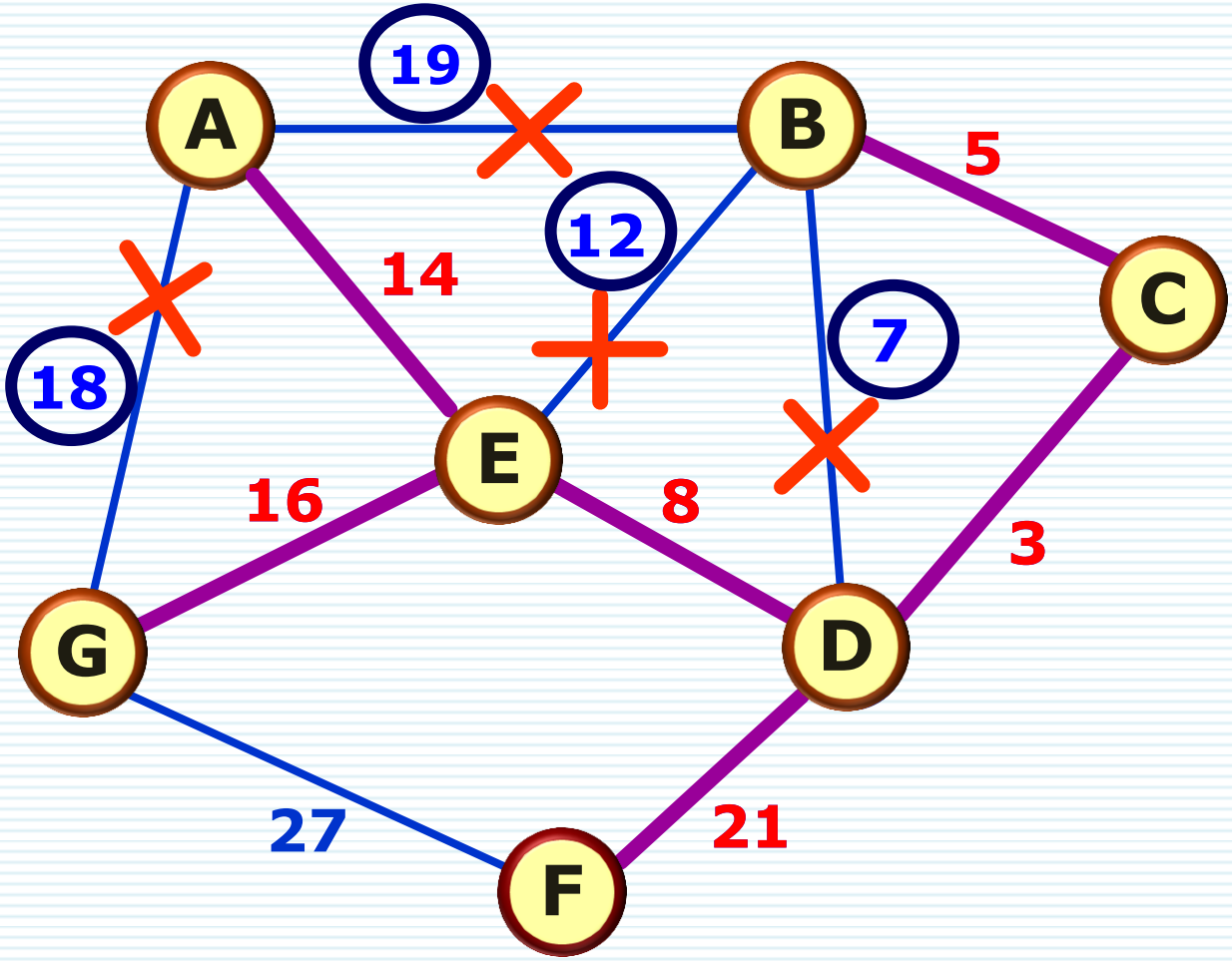
Kruskal算法

- Kruskal算法流程

  - 设 G ={V，{E}}为给定的连通网
  - 将生成树T的初始状态置为 T={V，{Φ}}
  - 当T中边数小于n-1时，重复下列步骤：

    1. 从E中选取代价最小的边（v, u）
    2. 若顶点v和u落在T中不同的连同分量上
       - 则：将其加入生成树T中，并从E中将其删除
    3. 否则：从E中将其删除，选择下一条代价最小的边

Kruskal算法



生成树代价 = 14+8+3+5+16+21 = 67

Kruskal算法的正确性

- 思考：如何证明？ **等价命题是什么？**
- 等价命题： Kruskal算法对于任意n阶图能得到一棵最小生成树
- 证明等价命题
  - 设n = 2，只有一条边，命题显然为真
  - 假设对n个顶点的图而言算法正确，考虑n+1个顶点的图G
    - 设：G中最小权边为 $e=(v_i, v_j)$
    - 从G中短接 v_i 和 v_j，得到图G'
    - 由归纳假设：若T'表示G'的MST，令：T = T'∪{e}
    - 目标是证明：T是关于G的一棵最小生成树

Kruskal算法的正确性

- 等价命题： Kruskal算法对于任意n阶图能得到一棵最小生成树
- 证明：设：G中最小权边为 $e=(v_i, v_j)$
  - 从G中短接 v_i 和 v_j，得到图G’， T’表示G’的MST
  - 令：T = T’∪{e} 需证明：T是关于G的一棵最小生成树

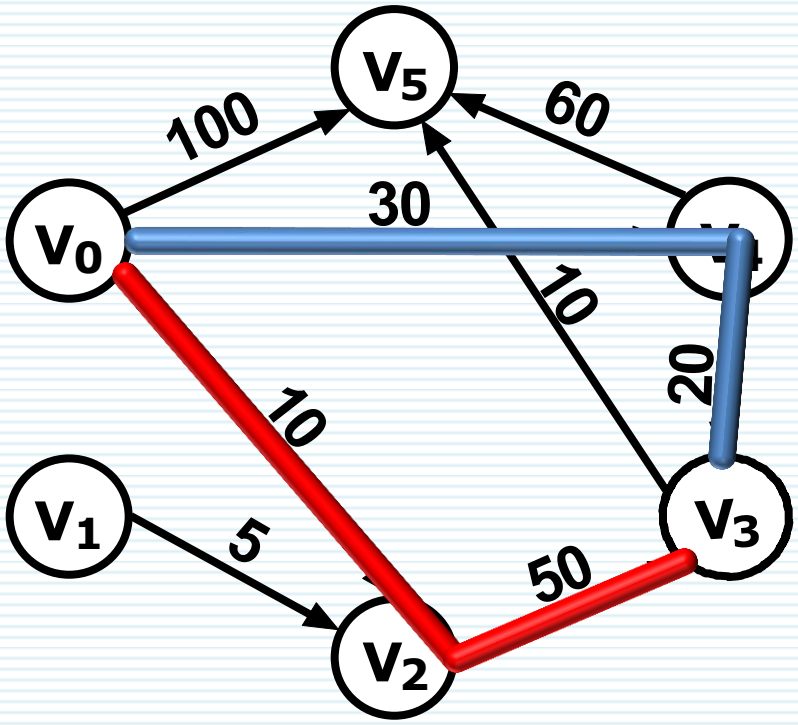
- 反证：设存在G的最小生成树T*，使得W(T*)<W(T)
  - 若：e∈T*，则短接e得到G'的生成树T*-{e}
    - 且W(T*-{e})<W(T')，与T'的最优性矛盾
  - 若：e不属于T*，则在T*加入边e，将形成回路
    - 去掉回路中任意一条其他边
    - 所得生成树的权小于W(T*)，与W(T*)的最优性矛盾□

6. 单源最短路径

(Single Source Shortest Paths)

单源最短路径

- 最短路径问题
  - 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



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

例如：在带权有向图G中求出v_0到其余各顶点之间的最短路径

- 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

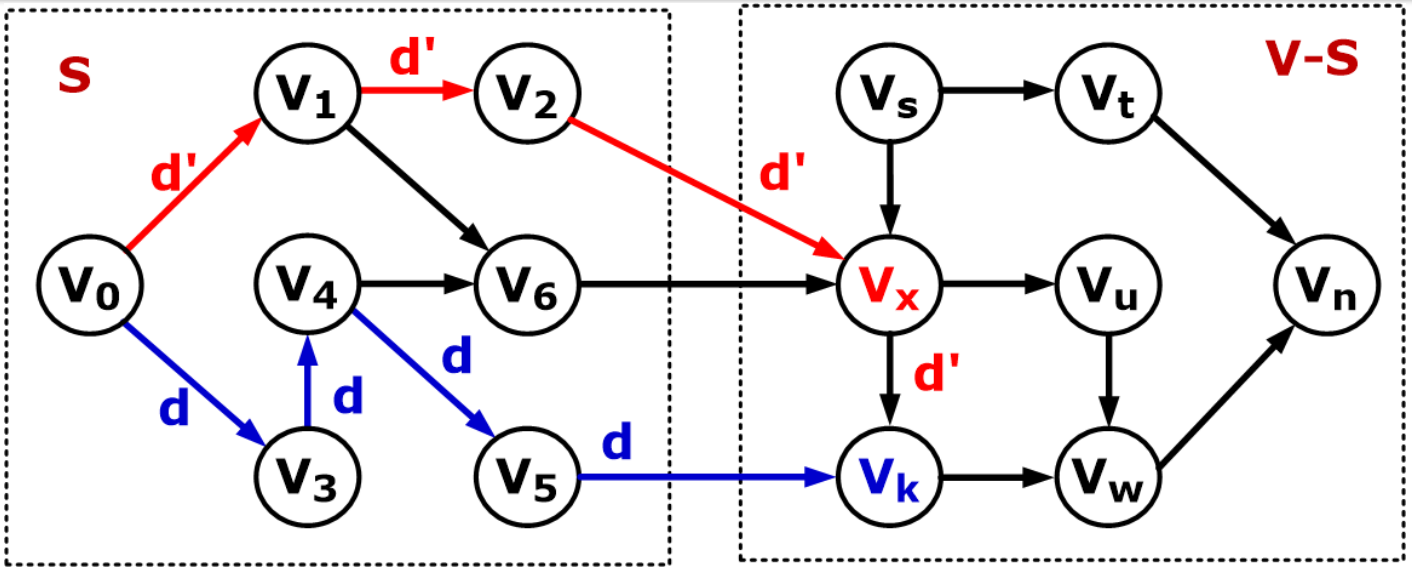
迪杰斯特拉（Dijkstra）算法

- Dijkstra算法是求解**单源最短路径问题**的一种有效算法
- 算法基本思路
  - 按路径长度递增的次序产生到各顶点的最短路径
  - 设置顶点集合S={V_0}并不断地做贪心选择来扩充S
- 贪心选择策略
  - 顶点V_k 属于S当且仅当从源到V_k 的最短路径长度已知
  - 可以证明：v_0 到 T=V-S 中顶点v_k的最短路径
    - 或者是从v_0 到v_k 的直接路径的权值
    - 或者是从v_0经S中顶点到v_k的路径权值之和

迪杰斯特拉（Dijkstra）算法

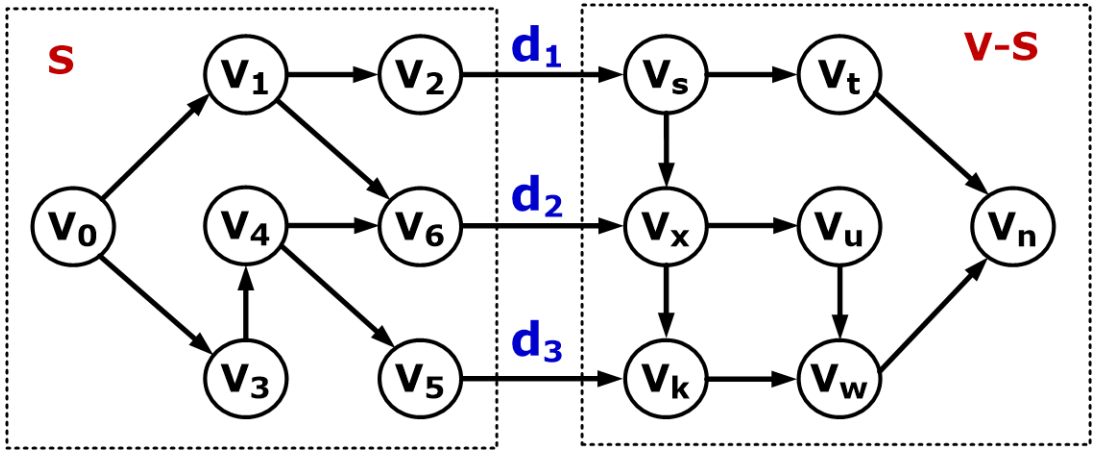
- 把图G的顶点V分成两组
  - S：已求出最短路径(SP)的顶点的集合
  - T=V-S：尚未确定最短路径的顶点集合
- 将T中顶点按最短路径递增的次序加入到S中并确保
  - 从v_0到S中任意顶点的SP ≤ 从V_0到T中任意顶点的SP
- 定义：从源v_0到顶点v_k的**特殊路径**
  - 从源v_0到v_k并且中间只经过S中顶点的路径
  - 由此：每个顶点对应一个距离值
    - S中顶点：从v_0到此顶点的最短路径长度
    - T中顶点：从v_0到此顶点的最短特殊路径长度

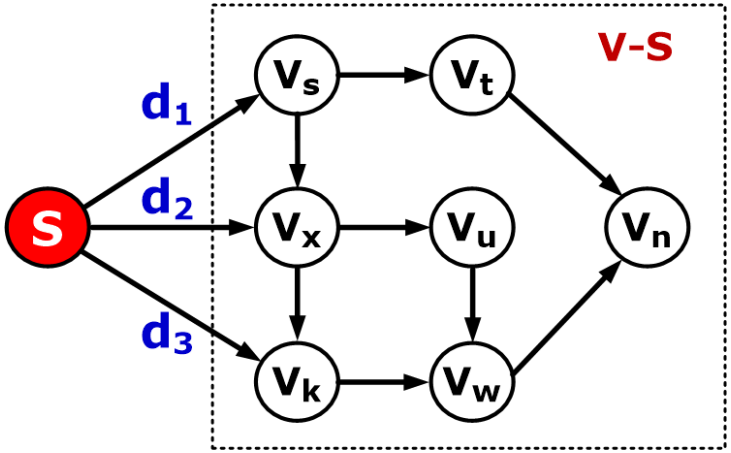
Dijkstra算法的贪心选择策略的有效性



- 贪心选择依据：v_0 到 T=V-S 中顶点v_k的最短路径（**d**）
  - 或者是从v_0 到v_k 的直接路径的权值
  - 或者是从v_0 经S中顶点到v_k的路径权值之和
- 问题：贪心选择的依据是否正确？
  - 反证：假设从v_0 到v_k 的最短路径**d'**经过T中的顶点v_x

Dijkstra算法的贪心选择策略的有效性



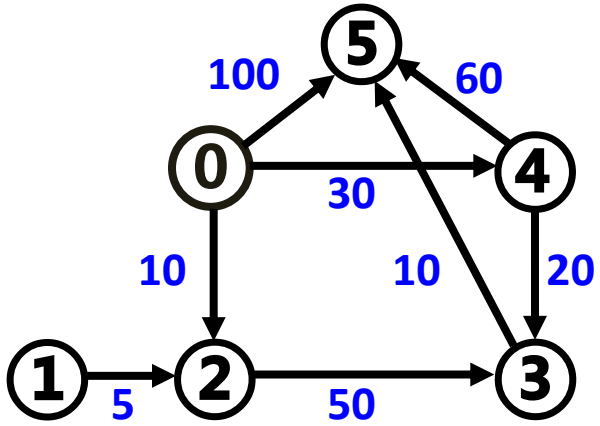


- 贪心选择策略：
  - 按路径长度递增的次序产生到各顶点的最短路径
- 问题：这种策略能否保证得到全局最优解？
  - 设图中标出的路径长度为从v_0到相应顶点的特殊路径长度
  - 不妨设：$d_1 \le d_2 \le d_3$
  - 则：从v_0出发到V-S的任意顶点的最短路径长度 $\ge d_1$

Dijkstra算法流程

1. 初始化
   - 令： S={v_0}，T={其余顶点}
   - T中顶点 v_i 与v_0的距离值 D_i 定义为
     - 若存在<v_0, v_i>： D_i 为弧<v_0, v_i>上的权值
     - 若不存在<v_0, v_i>： D_i 为∞
2. 从T=V-S中选取一个与v_0的距离值最小的顶点v_w加入S
   - 同时更新T中顶点的距离值：若增加v_w作中间顶点之后
     - 从v_0 经 v_w 到 v_i 的距离值比之前的特殊路径短
     - 则更新 V_i 距离值（为较小的值）
3. 重复上述步骤，直到S中包含所有顶点（即S=V）为止

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

- Dijkstra算法的数据结构设计
  - 使用带权邻接矩阵表示有向图G
  - 数组**S[n]**：顶点集合 (n 为图中顶点数)
    - 记录已找到从V_0出发的最短路径的顶点
  - 数组**dist[n]**
    - 存放各顶点距离V_0的当前最短路径长度
  - 辅助数组：**path[n]**(存储最短路径)
    - path[i]表示从V_0到V_i的SP上，V_i 的前序顶点的序号
    - 若从V_0到某顶点V_i无路径，则path[i]=-1

Dijkstra算法的存储结构

```
#define NV 6                    // 图中顶点总数

typedef  struct {

    int  vex[NV];               // 顶点数组

    int  arc[NV][NV];           // 邻接矩阵

}TGraph;
```

Dijkstra算法

```
// 求有向网G的v0顶点到其余顶点的最短路径
void dijkstra(TGraph G, int v0, int pat
    int S[NV] = {0};   S[v0] = 1;    //
    for(int i = 0; i < NV; i++) {
        dist[i] = G.arc[v0][i]; // v0到其他顶点的当前最短距离
        if( dist[i] < INT_MAX ) path[i] = v0;  // 记录前驱
        else path[i] = -1;
    }
```

```
typedef  struct {
    int  vex[NV];
    int  arc[NV][NV];
}TGraph;
```

Dijkstra算法

```
for(int  i = 0; i < NV; ++i) {
    if( i != v0 ){
        int min = INT_MAX,  v = -1;
        for( int k = 0; k < NV; k++){  // 找出最小的dist[k]
            if( S[k]==0 && dist[k] < min) {
                v = k;  min = dist[k];   } }
        if(v == -1) break;  // 已无顶点可加入S中
        S[v] = 1;                  // 将顶点v并入集合S
        for(int k = 0; k < NV; k++){
        if(S[k]==0 && min < (dist[k] - G.arc[v][k]) ){
            dist[k] = min + G.arc[v][k];   path[k] = v; }
        }
}}}
```

单源最短路径

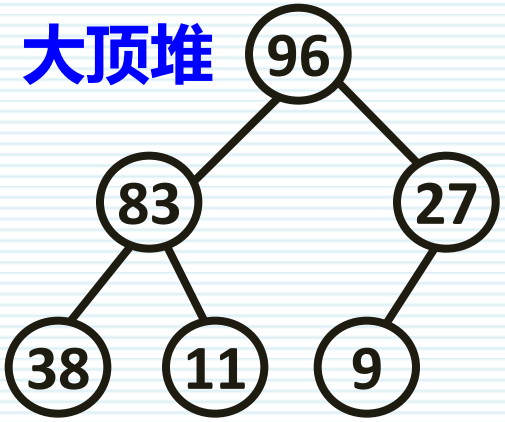
- 算法复杂性分析
  - 对于有n个顶点和e条边的带权有向图G
    - 采用带权邻接矩阵表示图G
    - 在dist[]数组中查找最小值需时：O(n)
    - 需对n-1的顶点执行上述操作
    - 算法的其余部分需时不超过$O(n^2)$
    - Dijkstra算法的复杂度为：$O(n^2)$
  - 算法改进？
    - 采用基于堆的优先队列：O(nlogn)

堆排序

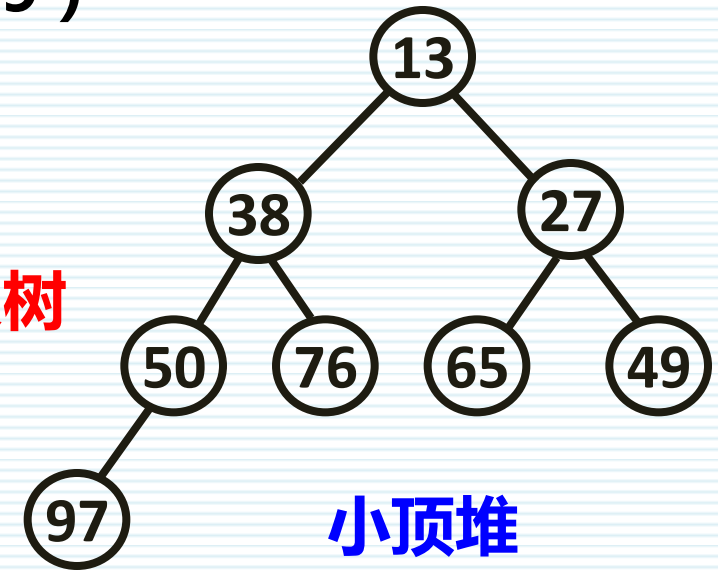
- n个元素（k_i）的序列，当且仅当满足下列关系时，称之为**堆**

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

- 例 （96，83，27，38，11，9）

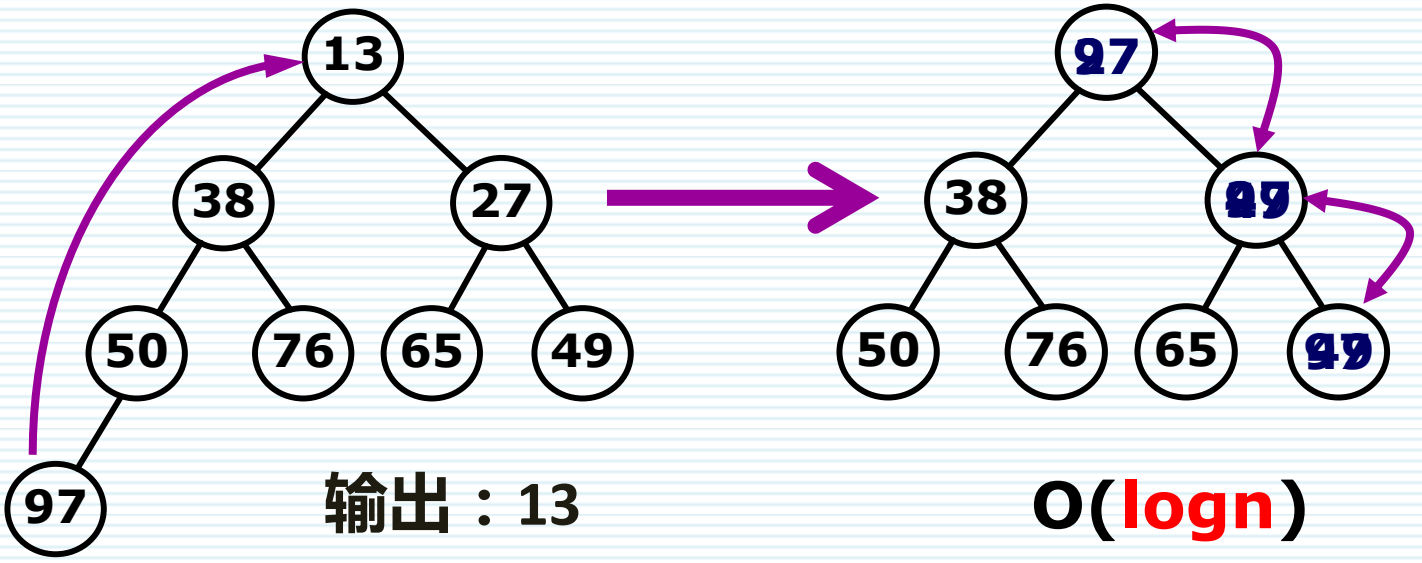


完全二叉树



- 例：（13，38，27，50，76，65，49，97）

输出堆顶元素后调整剩余元素成为一个新堆



- 解决方案（以小顶堆为例）
  - 输出堆顶元素之后，以堆中最后一个元素替代之
  - 比较根结点与左右子树根结点的值并与其中小者进行交换
  - 重复上述操作直至叶结点，将得到新的堆
- 称这个从堆顶至叶结点的调整过程为“筛选”（**Sift**）

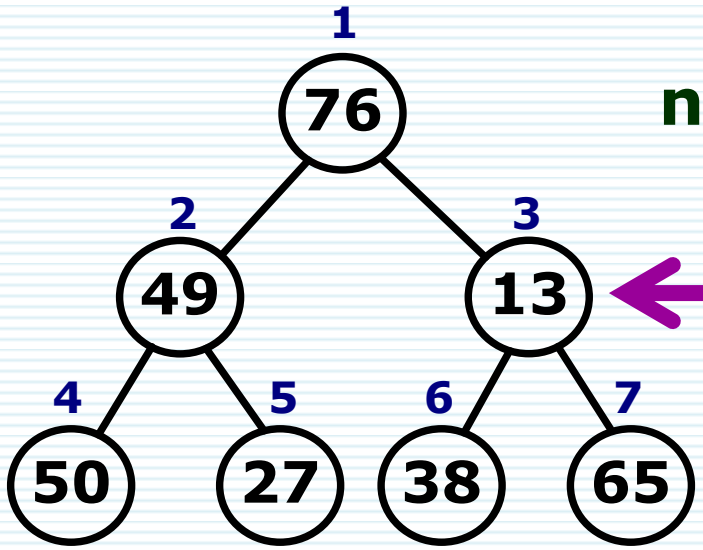
堆排序的筛选算法

```
// p是长度为n+1的数组（p[1:n]为堆元素序列）
void sift (int *p, int r, int n){  // r为指定的堆顶元素下标
    int k = 2 * r, p[0]= p[r];
    while ( k <= n ){
        if ( ( k < n) && p[k + 1] < p[k])  k++;
        if ( p[k] >= p[0]) { break; }
        p[r] = p[k];   r = k;
        k = 2 * r;
    }
    p[r] = p[0];     return;
}
```

时间复杂度

$T(n) = O(\log n)$

堆排序算法



n= 7　⌊n/2⌋= 3

从此处开始筛选

筛选顺序：3 → 2 → 1

- 如何由n个元素构成的无序序列构建一个堆？
  - 从无序序列的第⌊n/2⌋个元素起
  - 至第一个元素止，进行反复筛选
- 无序序列的第⌊n/2⌋个元素是什么意思？
  - 即：该序列对应的完全二叉树的最后一个非叶结点

堆排序的建堆算法

```
// p是长度为n+1的数组（p[1:n]为堆元素序列）

void build_heap (int *p, int n) {

    for( int i = n/2;  i>=1;  --i){

        sift (pbt,  i,  n);

    }

}
```

时间复杂度

$T(n) = O(nlogn)$

堆排序算法

```
void heap_sort(int *p, int n) {
    int i, p[0];
    for( i = n;  i>=2;  --i){
        p[0] = p[1];        // 保存堆顶元素
        p[1] = p[i];        // 将队尾元素交换到堆顶
        p[i] = p[0];        // p[i] 用于保存排序结果
        sift (p, 1, i-1);
    }
}
```

7. 多机调度问题

（ MultiProcessor Scheduling ）

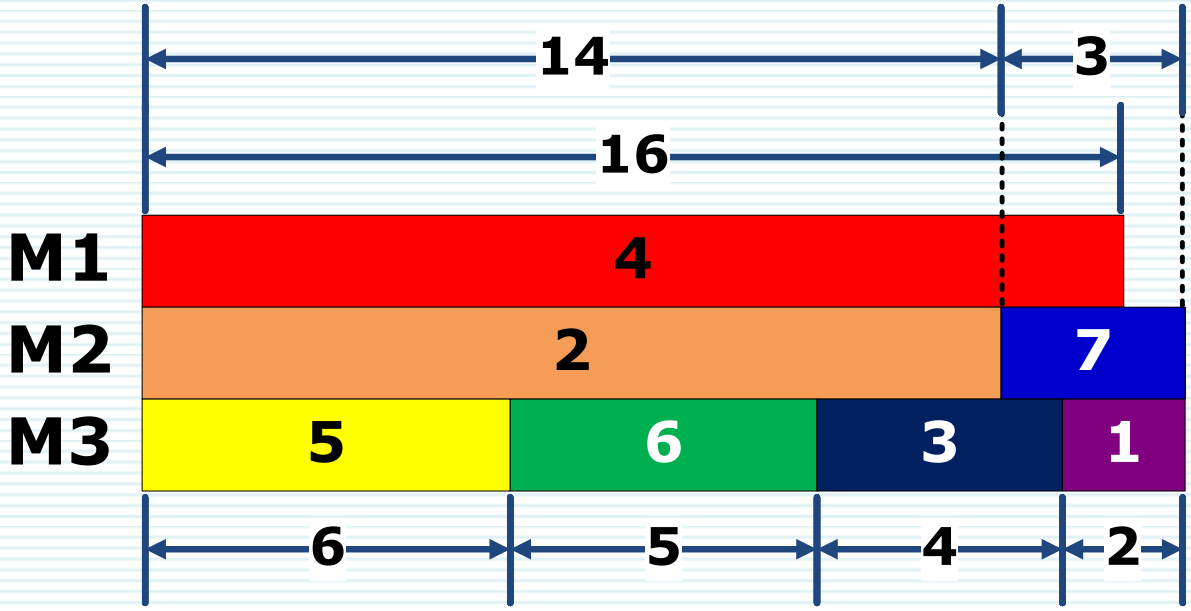
多机调度问题

- 问题定义：有n个独立的作业 {1,2…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n个作业在尽可能短的时间内由m台机器完成处理

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 $n \le m$ 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机器
    - 算法所需的计算时间为O(nlogn)

多机调度问题



- 设：有7个独立作业{1,2,3,4,5,6,7}交由3台机器处理
  - 各作业所需的处理时间分别为：{2, 14, 4, 16, 6, 5, 3}
  - 上述贪心算法产生的作业调度如图所示
    - 所需的加工时间总长度为17

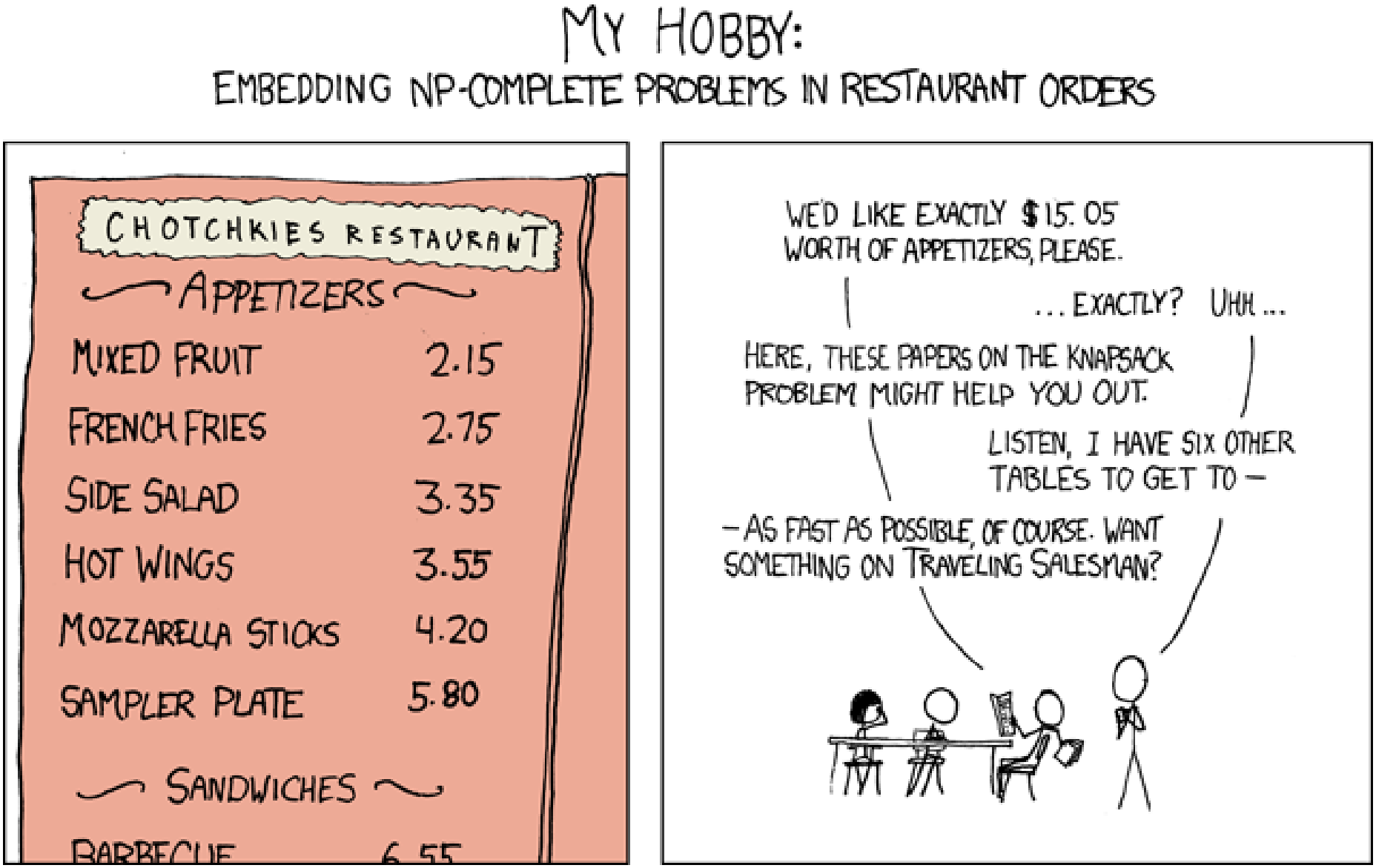
多机调度问题

- 多机调度问题小结
  - 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**
    - 即：采用**最长处理时间作业优先**的贪心选择策略

NP完全性问题简介

（ Introduction to NP-Complete ）

看似简单的问题未必简单：NP-Complete



算法理论的研究对象：两类抽象问题

- 优化问题（也称为极值问题）
  - 一个优化问题通常可以用以下四个部分来描述
    - 实例集合：若干实例 I 组成集合D
      - 其中每一个实例 I 含有一个问题所有输入的数据信息
    - 可行解集：每一个实例 I 有一个解集合 S(I)
      - 其中的每一个解都满足问题的条件，称为可行解
    - 目标函数：映射$c(\sigma)$: $S(I) \rightarrow \Re$
    - 最优化：求最优解 $\sigma_{opt}(I) \in S(I)$，使得对任意一个可行解 $\sigma \in S(I)$，都有$c(\sigma_{opt}(I)) \geq c(\sigma)$ 或者$c(\sigma_{opt}(I)) \leq c(\sigma)$
  - 一个优化问题也可以视为一个判定问题

算法理论的研究对象：两类抽象问题

- 判定问题（也称为识别问题）
  - 仅有两种可能的答案：“是”或者“否”
  - 可以将一个判定问题视为一个函数
    - 它将问题的输入集合 I 映射到问题解的集合{0 1}
  - 以路径判断问题为例：
    - 给定一个图G=(V, E) 和顶点集V中的两个顶点u, v
    - 判断 G 中是否存在一条 u 和 v 之间的路径
    - 如果用 i=<G, u, v>表示该问题的一个输入
      - 则：函数PATH(i)=1 （当u和v之间存在路径）
      - 则：函数PATH(i)=0 （当u和v之间不存在路径）

算法理论的研究对象：非确定性

- 非确定性是算法理论研究领域的一个基本问题
- 非确定性的定义
  - 给定某种计算机模型（自动机）
  - 若在任意时刻，对于自动机的当前状态和输入
    - 自动机有多个动作可供选择，则称机器为非确定性的
  - 若在任意时刻，对于自动机的当前状态和输入
    - 自动机的动作可唯一确定，则称机器为确定性的
  - 核心课题：非确定性能否提高机器的计算能力

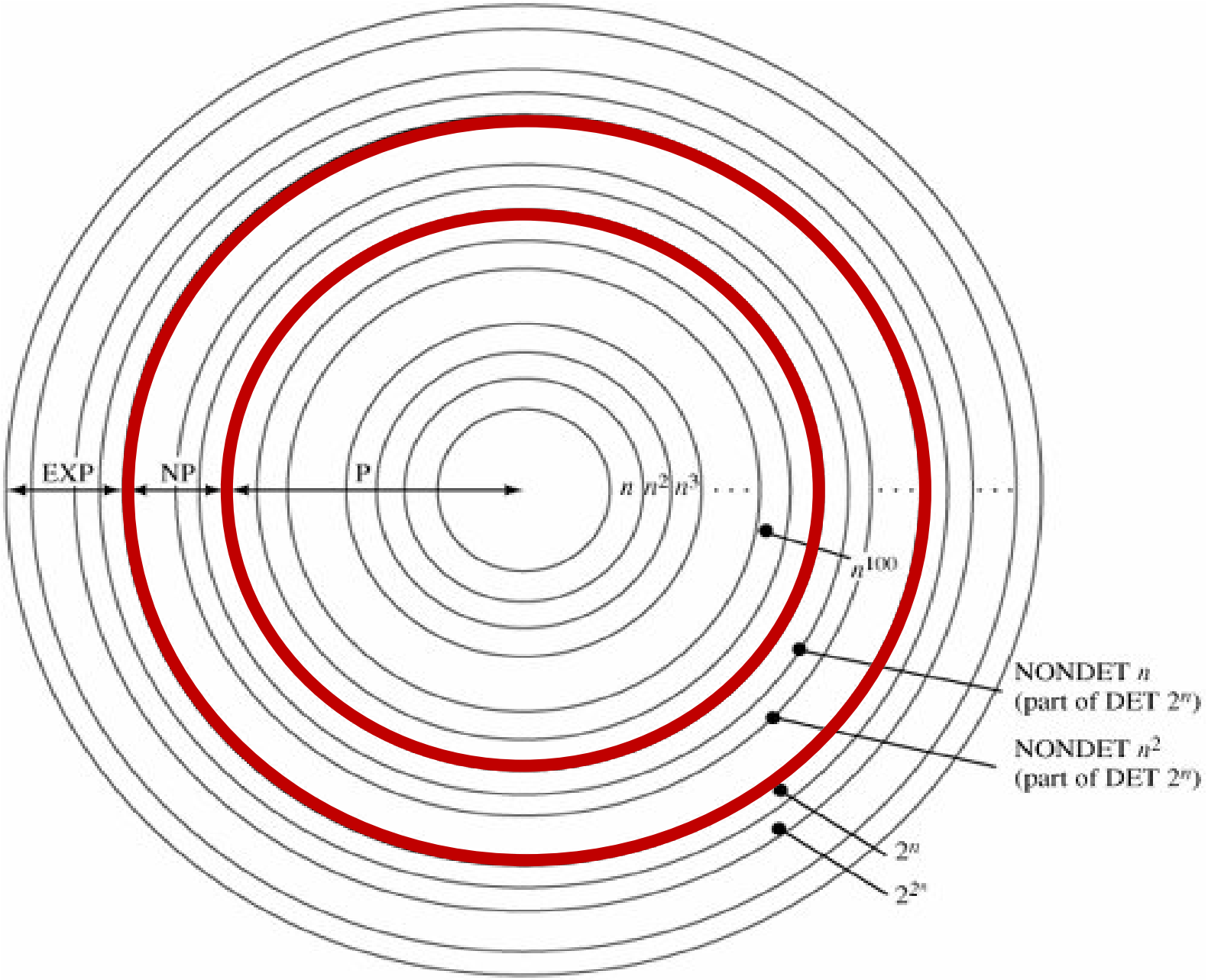
P和NP

- P和NP都是问题的集合
  - P是所有可在**多项式时间**内用**确定算法求解**的判定问题的集合
    - 对于一个问题X，若存在一个算法Xsolver
    - 能在$O(n^k)$时间内求解（k为某个常数）
    - 那么就称这个问题属于P
  - NP是所有可用**多项式时间**算法**验证其猜测准确性**的问题的集合
    - 对于一个问题X，若存在一个算法Xchecker
    - 能在多项式时间复杂度内给出验证结果
    - 那么就称这个问题属于NP

P和NP

- 显然： $P \subseteq NP$
- 数学的世纪难题，计算机科学领域的顶级难题： **P = NP ?**
  - 目前的研究结果倾向于认为：P!=NP
  - 即：有些不可快速计算的问题就是难处理的问题
  - 鉴于不能在多项式时间内给出解的算法的时间代价增长过快
  - 通常将可以由多项式时间的算法解决的问题看作是易处理的

计算复杂性的层次结构


EXP
NP
P
n n^2 n^3 . . .
n^{100}
NONDET n
(part of DET 2^n)
NONDET n^2
(part of DET 2^n)
2^n
2^{2^n}

NPC：NP-Complete

- NP-Complete的非形式化定义
  - 如果一个问题属于NP，且该问题与NP中的任何问题是一样难的
  - 则称该问题属于NP完全的（NPC，NP-Complete）
- 研究意义：NPC问题是20世纪的最伟大的发现之一
  - 1971年，Cook发现所有的NP问题都可以规约到SAT问题
    - SAT：SATISFIABLITY　布尔逻辑的可满足性问题
  - 1972年，Karp证明了21种问题是NP完全的
  - 直接推论：如果任何一个NPC问题可以在多项式时间内解决
    - 则NP中的所有问题都有一个多项式时间的算法
  - 迄今尚未发现任何一个NPC问题的多项式时间解决方案

NP完全性的证明

- 如何证明一个问题属于NPC类？
  - 证明一个问题是NP完全问题时（目的是证其困难）
    - 不是要证明存在某个有效的算法
    - 而是要证明不太可能存在一个有效的算法
  - 证明的方法依赖于三个关键概念：
    - 判定问题：NP完全性只适用于判定问题
    - 规约： NP完全性的定义和证明方法
    - 第一个NP完全问题：应用规约技术的前提
      - 已知一个NPC问题
      - 才能通过规约的方法证明另一个问题也是NPC的

问题的规约

- 对于给定的判定问题A，希望在多项式时间内解决该问题
  - 称某一特定问题的输入为该问题的一个实例（instance）
  - 假设有另一个不同的判定问题B可以在多项式时间内求解
  - 假设有如下过程
    - 可以将A的任意实例α转化为B的实例β
    - 转化操作需要多项式时间
    - 两个实例的答案相同（ α的答案为真 **iff** β的答案为真）
  - 称该过程为多项式时间的规约算法（reduction algorithm）

问题的规约

- 规约算法提供了一种在多项式时间内解决问题A的方法
  1. 首先利用规约算法将A的实例α转化为B的实例β
  2. 然后对实例β运行B的多项式时间判定算法
  3. 最后将β的答案作为α的答案
     - α的答案为真 **iff** β的答案为真
- 由于每一步只需多项式时间，因此判定α只需多项式时间
- 小结：通过将对问题A的求解规约为对问题B的求解
  - 就可以利用B的“易求解性”来证明A的“易求解性”
  - 思考：如果想证明某一问题是NP完全的？

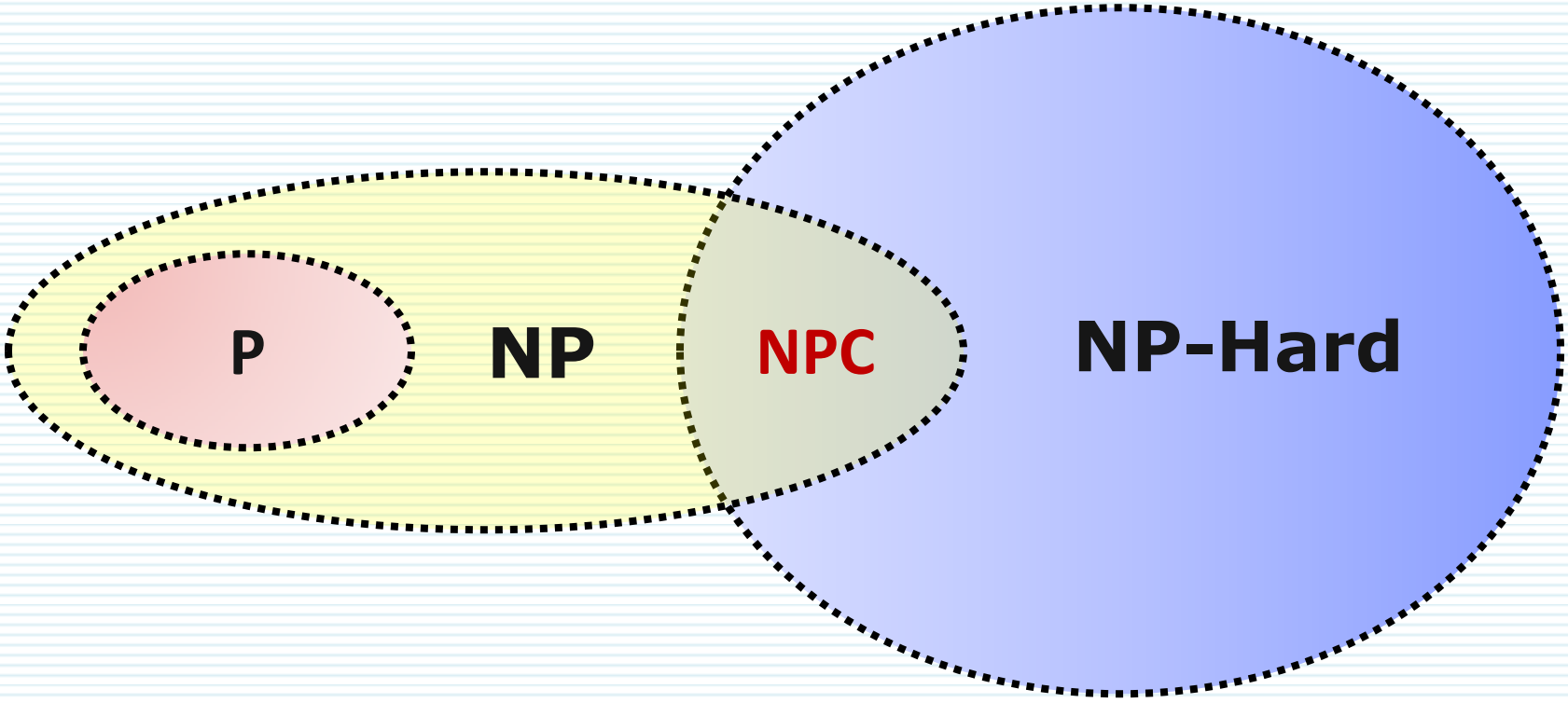
问题的规约

- 证明某一问题是NPC的思路恰恰与前面的思路相反
  - 利用规约表明对特定问题而言不存在多项式时间的算法
- 设：已知判定问题A不可能存在多项式时间的确定算法
  - 并设有一个多项式时间的规约将A的实例转化为B的实例
  - 则可以利用反证法证明B不可能存在多项式时间的算法
    - 反假设：B有一个多项式时间的算法
    - 根据规约算法：可以在多项式时间内解决问题A
    - 显然与已知矛盾（判定问题A没有多项式时间的算法）
  - 注意：无法假设问题A绝对没有多项式时间的算法

NPC和NPH

- NP-Complete的形式化定义
  - 如果一个判定问题A属于NP类
  - 而且NP中的任何问题均可在多项式时间内规约到A
  - 则称问题A是NP完全的（NP-Complete）
  - 判断A是否属于NP可以看其解是否可在多项式时间内被验证
- NP-Hard的形式化定义
  - 如果一个问题B满足上述条件2，则称之为NP-Hard问题
  - 也就是说：无论问题B是否属于NP类（是否满足条件1）
    - 若某一NPC问题可在多项式时间内规约到B
    - 则称问题B是NPH问题（NP-Hard）

几类问题之间的关系



一些经典的NP问题

- **SAT问题**
  - 对于输入的包含n个布尔变量的逻辑表达式
  - 求解使表达式为真的变量值组合
- **背包问题**
  - 给定背包容量C和n件物品及其重量
  - 求解物品选取方案，使得选出的物品重量之和恰好为C
- **n皇后问题**
  - 对于给定的nxn国际象棋棋盘
  - 要求输出一个在棋盘上放置n个互不攻击的皇后的方案
- **精确覆盖问题**
  - 对于输入的0/1矩阵，要求输出矩阵的若干个行号
  - 使得输入的0/1矩阵只保留输出的行后每列正好有一个1

一些经典的NP问题

- 旅行商问题（最优）：经典NPC问题
  - 对于输入的包含n个点的带权完全图
  - 要求输出一条遍历了所有顶点的总权值和最小的路径
- 旅行商问题： NP-Hard问题
  - 对于输入的包含n个点的带权完全图和一个正实数c
  - 要求输出一条遍历了所有顶点的总权值和不超过c的路径

NP完全性小结

- 一个判定问题A是NP完全的
  - 如果问题A属于NP类
  - 而且NP中的任何问题均可在多项式时间内规约到A
- 研究NP完全问题的意义
  - 如果任何一个NPC问题可以在多项式时间内解决
    - 则NP中的所有问题都有一个多项式时间的算法
  - 要成为一名优秀的算法设计者，熟悉这类问题是非常重要的
    - 很多有趣的自然问题并不比图的搜索等问题更困难
    - 目前已经证明的NP完全问题高达上千种
    - 如果问题是NPC的，更好的做法是采用近似算法求解

搜索算法简介

- 穷举搜索（ brute-force ）
- 盲目搜索（blind search）
  - 深度优先（DFS）：回溯法
  - 广度优先搜索（BFS）：分支限界法
  - 博弈树搜索（game-tree）：α-β剪枝算法
- 启发式搜索（heuristic search）
  - A*算法：IDA*算法， B*，局部择优搜索法
  - 仿生算法：蚁群算法，蜂群算法，禁忌算法，粒子群算法
  - 进化计算：遗传算法（ 1975 ）
  - 随机搜索：将随机过程引入搜索
    - 随机梯度下降算法，随机爬山算法
    - 模拟退火算法（ 1983 ），量子退火算法

算法之美

- 大道至简：简单就是美
  - 爱因斯坦质能方程：$E=mc^2$ （1905）
  - 冯·诺依曼体系结构（1946）
    - 将指令和数据同时存放在存储器中
    - 计算机组成：控制器、运算器、存储器、输入、输出设备
  - 递归：PageRank algorithm（Larry Page，1998）

$$PR(p_i)=\frac{1-\alpha}{n}+\alpha\sum_{p_j\in N(p_i)}\frac{PR(p_j)}{D(p_j)}$$

  - 动态规划：Viterbi algorithm（Andrew Viterbi，1967）

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.


UESTC
1956