



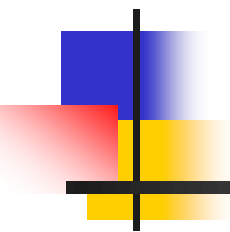
# 算法设计与分析

Computer Algorithm Design & Analysis

2024.11

王多强

QQ: 1097412466



# Chapter 16

## Greedy Algorithms

---

贪心算法

# 概述：

## 1、什么是贪心方法？

**贪心方法**也是一种求解**最优化问题**的方法，适用于这样一种情形的问题：

已知包含  $n$  个数据对象的**集合  $S$** ，问题的解是从  $S$  中选择某些数据对象构成一个**解集合  $J$** ，但需要满足一定的**约束条件**，同时有**目标函数**评价  $J$  集合的**优劣**。只满足约束条件的解称为**可行解**，而能够**使得目标函数取得极值**的可行解称为问题的**最优解**，问题的目标是求问题的最优解。

贪心方法就是按照上述规则求一类“**子集选择**”问题最优解的策略。具体思想如下：

## 贪心方法的基本思想：

开始时，将**解子集  $J$**  置空， $J \leftarrow \Phi$ ；并确定一个**选择元素的策略**。然后按照该选择策略，每次从**元素集合  $S$**  中选择一个元素：

- ◆ 如果当前元素  $i$  和  $J$  中已有的元素加在一起不违反约束条件，就将该元素并入  $J$  中： $J \leftarrow J \cup \{i\}$ 。**否则就舍去。**
- ◆ 继续考察后面的元素

当所有元素都处理完毕，**最终被计入到  $J$  中的元素就构成了问题的一个解。**

这个解可称为问题在该选择策略下的一个**贪心解**。

例，已知有100个任意正整数，在其中选择不超过10个不重复的奇数，使得它们的**和是所有选择中最大的**。

- ◆ **数据集S**： 由100个正整数组成的集合
- ◆ **约束条件**：
  - (1) 选择出来的**元素必须是奇数**；
  - (2) 选择出来的**元素不能重复**；
  - (3) 选择出来的**元素个数  $\leq 10$** ；

**目标函数**： 使选择出来的**元素之和达到最大**。

◆ 解决该问题的一个贪心算法可设计如下：

置解集合  $J \leftarrow \Phi$ 。

**选择策略：**按照**非增次序**选择  $S$  中的元素。这样每次选出的元素是当前  $S$  中剩余元素中的最大者。

**考察元素：**对当前选出的元素，看其是否满足约束条件：

- (1) 奇数；
- (2) 不能和  $J$  中现有的元素重复；
- (3)  $J$  中的元素个数  $< 10$ 。

以上条件都满足，就将该元素加入  $J$  集合，否则就丢弃；

然后继续选择下一个元素，直到已经选出了10个最大的奇数或者100个整数都考察完毕，算法结束。

## 2、设计贪心算法需要特别考虑的问题：

### (1) 贪心解是问题的可行解吗？ 是！

- ◆ 因为计入到解集合  $J$  中的元素都是满足约束条件的，所以最后获得的贪心解 ( $J$ ) 一定是可行解。

### (2) 贪心解一定是问题的最优解吗？ 否！

- ◆ 如果**选择策略**不正确，就可能无法获得最优解。

如前例，如果选择策略是“**按非降次序选择元素**”，则只要是“**奇数、不重复、个数不超过10**”的整数就会加入  $J$  集合，但这样选择，结果虽然“可行”，但显然不是最优解。

### (3) 不是所有问题都适合用贪心策略求解。

如**问题本身不适合用贪心策略**而用“贪心策略”求解也是得不到问题最优解的。

如上例中，若将**约束条件 (3)** 改为

**“J 中元素总和不能超过101”**，

然后对1~100中的奇数按照同样的规则处理就无法获得最优解：

- 算法选择的结果是： $\{99, 1\}$ ，和为100；
- 而和最大的选择是： $\{97, 3, 1\}$ （或其它选择）  
和是101。



**所以，用贪心方法求解问题的要点是：**

**① 先确定问题适不适合用贪心策略求解**

只有合适的问题才能用贪心方法求解。

**② 精心设计一个选择策略**

以期待贪心解是问题的最优解。

并且，通常要**证明贪心选择的正确性**，即该选择策略下能够获得问题的最优解，而不是“凭感觉”或“表面看起来像”。

——如果一个选择策略能够获得问题的最优解，则称这种选择策略是问题的**最优选择策略**。

**③ 实施求解，获得问题的最优解。**

### 3、贪心策略的核心思想：局部最优选择

贪心策略的一个基本原则是：按既定的**元素选择策略**一步步选择和考察元素，如果当前选出的元素  $i$  有  $J \cup \{i\}$  可行，就将  $i$  并入集合  $J$ ；如果不可行，就舍弃，**而且一旦舍弃，就不会“后悔”或“推倒重来”**。

如前面选择  $\{99, 1\}$  的问题，是不是发现选 99 不好的时候重新选择别的元素呢？**贪心策略“不走回头路”**。

所以**贪心选择**是“**只看眼前**”，认为**当前选出来的元素就是最好的**，但是不是“**全局最好**”在当前时刻无法判定，而且即使以后发现不好，也不再重新选择。

这种选择元素的思路称为“**局部最优选择**”。

——贪心策略就是用“**局部最优选择**”来构造**全局最优解**。

但显然有时**局部最优选择未必能产生全局最优解**。

## 16.1 活动选择问题

### 1、问题描述

假定有一个含有  $n$  个活动的集合  $S = \{a_1, a_2, \dots, a_n\}$ , 每个活动  $a_i$  都有一个开始时间  $s_i$  和结束时间  $f_i$ ,  $0 \leq s_i < f_i < \infty$ 。

这些活动**要使用同一资源** (如演讲会场), 而这个资源在任何时刻只能供一个活动使用。如果选择了活动  $a_i$ , 则它将在一个半开区间  $[s_i, f_i)$  内独占资源; 若要安排两个活动  $a_i$  和  $a_j$ , 则必须保证它们的时间不能重叠, 即  $s_i \geq f_j$  或  $s_j \geq f_i$  —— 如果两个活动时间不重叠, 则称它们是**兼容**的。

**活动选择问题**就是选择  $S$  的一个**最大兼容活动子集**  $A$ :  $A$  中的活动都是兼容的且对已知活动集  $S$ ,  $A$  包含的活动数量最多。

例：设有11个待安排的活动，它们的开始时间和结束时间如下表所示。

$i$	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

则  $\{a_3, a_9, a_{11}\}$ 、 $\{a_1, a_4, a_8, a_{11}\}$ 、 $\{a_2, a_4, a_9, a_{11}\}$  等都是兼容活动集合。

其中  $\{a_1, a_4, a_8, a_{11}\}$ 、 $\{a_2, a_4, a_9, a_{11}\}$  各包含四个活动，是该问题的最大兼容活动集合（注：最大兼容活动集合不一定是唯一的）。

## 分析：如何在资源受限的条件下安排尽量多的活动？

**一个基本的设计思路**是：尽量早地结束前面的活动，尽量多地 **“空出” 后面的时间** 以安排更多的活动。

根据这一思路，可以构造这样一个**贪心选择策略**：

- ◆ **按照活动结束时间的非降次序**选择作业，这样当前选择的作业就是**结束时间最早**的作业。
- ◆ 如果它能和前面选出的其它活动兼容，则将之加入贪心解集合  $A$ ，否则舍弃。

事实上，按照这样的顺序选择活动时，前面最后一个被计入  $A$  集合中的活动就是目前最晚结束的活动，判断当前活动是否和前面已选活动兼容，仅需比较当前活动开始时间  $s_i$  和前面最后一个被计入  $A$  集合中的活动的结束时间  $f_j$ ，只要  $s_i \geq f_j$ ，即兼容；否则就不兼容。

# 由此可以设计出活动选择问题的一个贪心算法：

GREEDY-ACTIVITY-SELECTOR( $s, f$ )

// 不失一般性，假设算法在开始之前已经**按结束时间的非降次序对所有活动排序**，

// 即已有  $f_1 \leq f_2 \leq f_3 \leq \dots \leq f_{n-1} \leq f_n$ 。

1  $n = s.length$

2  $A = \{a_1\}$  // 第一个活动是**结束时间最早**的活动，所以首先被计入解集  $A$  中

3  $k = 1$  //  $k$  指示最后一个加入  $A$  集合的活动（注：是**该活动在  $S$  集合中的下标**）

4 **for**  $m = 2$  **to**  $n$

5 **if**  $s[m] \geq f[k]$  //  $m$  指示当前正在考虑的活动，它是  $S$  集中剩余活动中结束时间  
// 最早的活动，比较  $s[m]$  和  $f[k]$  判断活动  $m$  是否可行。

6  $A = A \cup \{a_m\}$  // 活动  $m$  与之前计入  $A$  中的其它活动兼容，将  $a_m$  并入解集合  $A$

7  $k = m$  //  $k$  指示最后一个计入  $A$  中的活动

8 **return**  $A$

- ◆ 活动按结束时间排序：  $O(n \log n)$
- ◆ 选择最大兼容活动集合：  $O(n)$
- ◆ 如果考虑活动预排序，算法总的时间复杂度是  $O(n \log n)$ 。

如前例，11个待安排的活动如下。

$i$	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	9	9	10	11	12	14	16

活动均按结束时间的**非降序次序**排列

根据算法的处理策略有：

- ◆  $a_1$  是最早结束的活动，首先被计入活动集  $A = \{a_1\}$ ,  $k = 1$ ;
- ◆  $a_2$  是下一个被考察的活动，但  $s[2]=3 < f[1]=4$ , 与  $a_1$  不兼容，舍弃。
- ◆  $a_3$  是第三早结束的活动，但  $s[3]=0 < f[1]=4$ , 与  $a_1$  也不兼容，舍弃。
- ◆ 继续考察  $a_4$ ，由于  $s[4]=5 > f[1]=4$ , 与  $a_1$  兼容，所以  $a_4$  可被选中，并入集合  $A = A \cup \{a_4\}$ ,  $k = 4$ 。
- ◆ 再继续考察  $a_5$ 、 $a_6$ 、 $a_7$ ，与  $a_4$  均不兼容，而被舍弃。



$i$	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

- ◆ 再继续考察  $a_8$ ,  $s[8]=8 > f[4]=7$ , 与  $a_4$  兼容, 被选中, 并入集合  $A = A \cup \{a_8\}$ ,  $k = 8$ 。
  - ◆ 再继续考察剩余的活动, 有:  $a_9$ 、 $a_{10}$  被舍弃, 而  $a_{11}$  和  $a_8$  兼容, 被选中, 并入集合  $A = A \cup \{a_{11}\}$ ,  $k = 11$ 。
- 至此, 所有活动都考察完毕, 算法结束。得到的  $A = \{1, 4, 8, 11\}$ 。

问:  $A = \{1, 4, 8, 11\}$  是该问题的一个最大兼容活动集合吗?

通过“观察”, 确实是!

但一般情况下该如何证明一个选择策略是正确的呢?

(另: 最大兼容活动集不唯一, 可以改造算法来找其它最大兼容活动集。自己思考, 这里不做讨论)



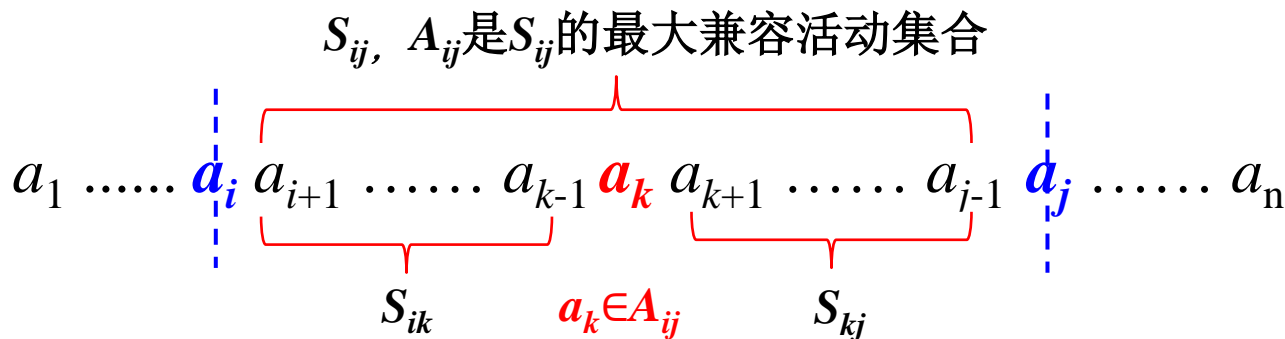
## 2、活动选择问题的最优子结构

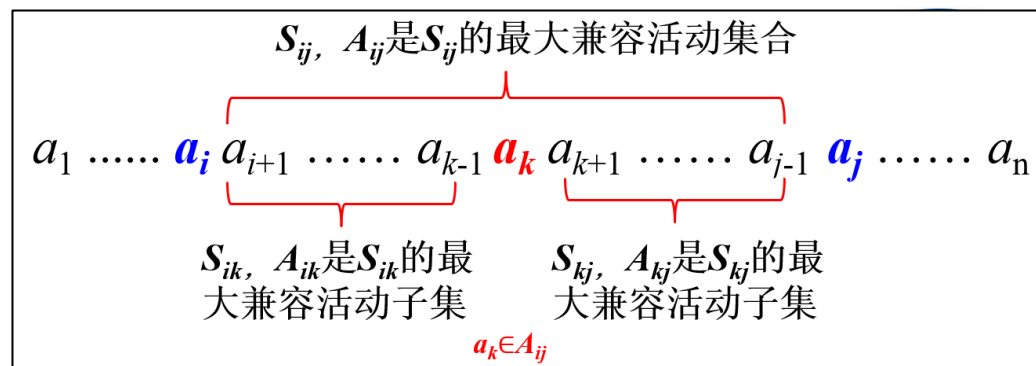
**和动态规划类似，适合用贪心策略求解的问题也要求问题具有最优子结构性。**

对于活动选择问题，其最优子结构描述如下：

令  $S_{ij}$  表示在  $a_i$  结束之后开始且在  $a_j$  开始之前结束的那些活动的集合（注： $S_{ij}$  中不包括  $a_i$  和  $a_j$ ）。

设  $A_{ij}$  是  $S_{ij}$  的一个最大兼容活动集（ $a_{i+1} \sim a_{j-1}$  范围内的最优解），并设  $A_{ij}$  包含活动  $a_k$ ，如图所示。

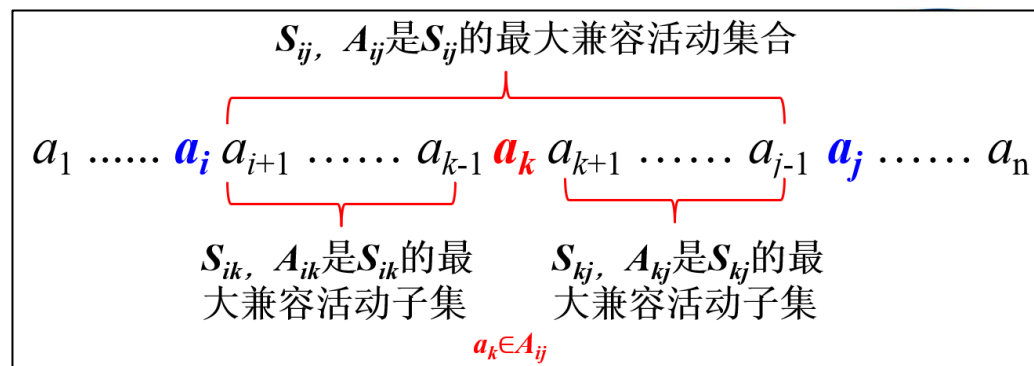




如图所示，相对  $a_k$  就有：

(1) 将  $A_{ij}$  被分成**左侧**和**右侧**两部分：

- ◆ **左侧子序列：** 记为  $A_{ik}$ ，其中活动必是兼容、可行的，并恰好是  $A_{ij}$  中所有**在  $a_i$  结束之后开始、 $a_k$  开始之前结束**的活动子集。
- ◆ **右侧子序列：** 记为  $A_{kj}$ ，其中活动也都是兼容、可行的，并恰好是  $A_{ij}$  中所有**在  $a_k$  结束之后开始、 $a_j$  开始之前结束**的活动子集。
- ◆  $A_{ik}$  和  $A_{kj}$  中均不包括  $a_k$ ：  $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$



(2)  $k$  位置也将  $S_{ij}$  分成**左侧**和**右侧**两部分:

- ◆ 记  $S_{ik}$  为  $S_{ij}$  中所有在  $a_i$  **结束之后开始**、 $a_k$  **开始之前结束** 的活动子集。它们未必都兼容，但都位于  $a_k$  的左侧。
- ◆ 记  $S_{kj}$  为  $S_{ij}$  中所有在  $a_k$  **结束之后开始**、 $a_j$  **开始之前结束** 的活动子集，它们也未必都兼容，但都位于  $a_k$  的右侧。
- ◆  $S_{ik}$  和  $S_{kj}$  中均不包括  $a_k$ :  $S_{ij} = S_{ik} \cup \{a_k\} \cup S_{kj}$

则相对原问题（求  $S_{ij}$  的最大兼容活动集合）就形成两个子问题：

- ① 求  $S_{ik}$  的最大兼容活动集合； ② 求  $S_{kj}$  的最大兼容活动集合。

## “活动选择问题具有最优子结构” 体现在：

因为  $A_{ij}$  是  $S_{ij}$  的一个最大兼容活动子集，所以**必有**  $A_{ij}$  的子序列  $A_{ik}$ 、 $A_{kj}$  分别是  $S_{ij}$  的子问题  $S_{ik}$ 、 $S_{kj}$  的一个最大兼容活动子集。

**证明：**（**剪切-粘贴法**）

假设  $S_{kj}$  存在一个最大兼容活动集  $A'_{kj}$ ，有  $|A'_{kj}| > |A_{kj}|$ ，则可以在  $A_{ij}$  中**用  $A'_{kj}$  替换  $A_{kj}$** ，这样就构造出一个新的兼容活动集合

$A'_{ij} = A_{ik} \cup \{a_k\} \cup A'_{kj}$ ，其大小为

$$|A'_{ij}| = |A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$$

这就与  $A_{ij}$  是  $S_{ij}$  的最优解相矛盾，故  $S_{kj}$  中不会存在比  $A_{kj}$  还大的兼容活动子集。

同理， $A_{ik}$  是  $S_{ik}$  的一个最大兼容活动子集。证毕。

## ◆ 活动选择问题的动态规划方法

由于活动选择问题具有**最优子结构性**，所以可以用**动态规划**方法求解：

令  $c[i, j]$  表示集合  $S_{ij}$  的最优解大小，即  $|A_{ij}|$  的值。

若以  $k$  划分  $[i, j]$  区间，可得：

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

为了选择  $k$ ，建立递推关系式（状态转移方程）：

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset . \end{cases}$$

进而可以设计**带备忘机制的递归算法**或**自底向上的填表算法**求解（自行设计）。

### 3、贪心算法正确性（即可以获得最优解）证明

根据**算法的预设**：所有活动已按照结束时间**非降次序**排序，  
所以活动  $a_1$  是最早结束的活动，故首先有： $A = \{a_1\}$ ， $k = 1$ 。

之后，依次考察其它活动，那些能够和  $A$  集合兼容的活动都将并入集合  $A$ ：任何活动  $i$ ，若  $A = A \cup \{a_i\}$ ，则置： $k = i$ 。

令  $S_k = \{a_i \in S : s_i \geq f_k\}$ ，即在  $a_k$  结束之后才开始的所有任务集合，亦即**所有和  $a_k$  兼容的活动**。则，

- ◆ 在**首次选择  $a_1$** 后， $S_1$  就是接下来要求解的子问题，并且是唯一子问题。

**注： $S_1$  中不包含任何和  $a_1$  不兼容的活动**

- ◆ 算法下一步要做的就是**在  $S_1$  中再选择结束之间最早的活动**加入  $A$  集合，并生成一个新的  $S_k$  作为下一步求解的子问题继续求解，而且这个  $S_k$  也是下一步待求解的唯一子问题。

这样依次进行下去。随着新的兼容活动被并入集合  $A$ ， $k$  也不断被更新而生成一系列  $S_k$ ，每个  $S_k$  都代表下一步要求解的子问题，而且是**下一步的唯一子问题**。

如前例，11个待安排的活动如下。

$i$	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	9	9	10	11	12	14	16

活动均按结束时间的**非降序次序**排列

根据上述思想有：

- ◆ **第一步：**  $a_1$  是最早结束的活动，首先进入活动集， $A = \{a_1\}$ ，并置  $k = 1$ ，得  $S_1 = \{a_4, a_6, a_7, a_8, a_9, a_{11}\}$ ， $a_4$  是  $S_1$  中结束时间最早的活动。

$i$	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

- ◆ **第二步：** 从  $S_1$  中选择结束时间最早的活动，即  $a_4$ ，并置  $A = \{a_1, a_4\}$ ， $k = 1$ ，得  $S_4 = \{a_8, a_9, a_{11}\}$ ， $a_8$  是  $S_4$  中结束时间最早的活动。

$i$	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



再继续:

$$S_4 = \{a_8, a_9, a_{11}\}$$

- ◆ **第三步:** 从  $S_4$  中选择结束时间最早的活动, 即  $a_8$ , 并入集合  $A$ ,  
 $A = \{a_1, a_4, a_8\}$ , 并置  $k = 8$ , 得  $S_8 = \{a_{11}\}$ ,  $a_{11}$  是  $S_8$  中结束  
 时间最早且唯一的活动。

$i$	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

- ◆ **第四步:** 最后从  $S_8$  中选择结束时间最早的活动  $a_{11}$ , 并入集合  $A$ ,  
 $A = \{a_1, a_4, a_8, a_{11}\}$ , 再置  $k = 11$ , 得  $S_{11} = \Phi$ , 算法结束。

$i$	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

## 上述过程就是：

从  $A = \{a_1\}$ ,  $k = 1$  开始，依次往后选择兼容活动：

设当前过程进行到这样的状态：前一步选择了活动  $a_k$  并加入集合  $A$ ，剩下的待下一步求解的唯一子问题是  $S_k$ 。

由于“**不走回头路**”，所以算法最终所期待的贪心解**从本步来看**，就只能是由当前的  $A \cup \{ \text{未来 } S_k \text{ 的子解} \}$  构成。

根据最优子结构性：**最优解由最优子解组合构成**。所以最终贪心解如果是问题最优解的话，则就本步而言，**当前的  $A$  应该是  $S_k$  之前子问题的最优子解**，而“**未来  $S_k$  的子解**”也应该是  $S_k$  的最优子解。

**但问题是：这样一步步选择，最终能得到最优解吗？**

## 这里只需回答一个问题：

对任意  $S_k$ （包括原始序列  $S$ ，可视为  $S_0$ ），选择其中结束时间最早的那个活动（包括  $a_1$ ），是不是最优选择？

对  $S_k$  内的活动而言，（由于算法预设所有活动已按照结束时间的非降次序排列，所以）其**第一个活动**就是结束时间最早的活动，设为  $a_m$ 。

① 选  $a_m$  是可行的吗？ **是**

因为  $S_k$  中的活动都和截止到目前为止已选出的其它活动兼容，包括  $a_m$ ，所以可以选  $a_m$ 。

② 选  $a_m$  是最优的吗？

如果选  $a_m$  是最优选择，后续就可以“同理”去构造  $S_m$  及其它后续子问题的最优子解。直至最终由这一系列子问题的最优子解合在一起构成原始问题的最优解。

这样做真的可以吗？

下面的**定理16.1**证明了对任意非空子问题  $S_k$ ，选择其中的结束时间最早的活动是一个最优选择。

**定理16.1** 考虑任意非空子问题  $S_k$ ，设  $a_m$  是  $S_k$  中结束时间最早的活动，则  $a_m$  必在  $S_k$  的某个最大兼容活动子集中。

**定理16.1** 考虑任意非空子问题  $S_k$ , 设  $a_m$  是  $S_k$  中结束时间最早的活动, 则  $a_m$  必在  $S_k$  的某个最大兼容活动子集中。

**证明:**

令  $A_k$  是  $S_k$  的一个最大兼容活动子集, 且  $a_j$  是  $A_k$  中结束最早的活动。

① 若  $a_j = a_m$ , 则得证。否则,

② 令  $A'_k = A_k - \{a_j\} \cup \{a_m\}$ , 则  $A'_k$  依然是  $S_k$  的一个最大兼容活动子集, 并且是一个包含  $a_m$  的最大兼容活动子集。

这是因为:  $a_j$  仅是  $A_k$  中结束最早的活动, 而  $a_m$  是整个  $S_k$  中结束时间最早的活动, 所以  $f_m \leq f_j$ 。那么在  $A_k$  中用  $a_m$  替换  $a_j$  后得到的新集合  $A'_k$ , 其中的所有活动也必是不相交的, 所以  $A'_k$  也是  $S_k$  的一个可行解, 且  $|A'_k| = |A_k|$ , 所以  $A'_k$  也是  $S_k$  的一个最大兼容活动子集。 ■ 证毕

## ◆ 定理16.1说明:

### ① 选择 $a_m$ 是可以构造 $S_k$ 最大兼容活动子集的。

这样, 根据最优子结构性, 选出  $a_m$  后, 只要设法找现在的  $S_m$  的最优子解, 然后和  $a_m$  合在一起就可以得到  $S_k$  的最优解。而对  $S_m$  的进一步求解, 就可以按照同样的方法: 找其中结束时间最早的活动加入兼容活动集即可。

这样至少说明可以通过选择  $a_m$  找到  $S_k$  的最优解。

### ② 给出了一种 “类似数学归纳法” 的思路来证明整个贪心选择策略的正确性:

首先, 对原始问题  $S_0 = S$  做 “初始选择”:  $A = \{a_1\}$ , 那么

**定理16.1**保证了  $a_1$  是初始状态下的一个最优选择。

注：

① 可能除了 $a_1$ ，还有别的选择也可以构成初始的最优选择，进而有其它形式的最优解，但我们只得其一即可，所以不失一般性，就选 $a_1$ ，定理16.1说明这样选可行。

② 之后那些不在 $S_1$ 中的活动都和 $a_1$ 不兼容，所以不可能和 $a_1$ 构成可行解，故不用考虑，而只需考虑还在 $S_1$ 中的活动所构成的子问题。

然后假设：该操作对**直到**某个 $S_k$ 之前的所有子问题都成立。

最后，定理**定理16.1**给出了推论证明：到 $S_k$ 时，选择 $S_k$ 中结束时间最早的活动 $a_m$ 可以构造 $S_k$ 的最优子解，而 $S_k$ 之前的选择出来的活动是到 $S_k$ 之前的最优子解，二者合在一起就构成全局最优解。所以**整个操作就是正确的**了。

## ◆ 从定理16.1得到的启发

对其它问题的贪心算法可以沿用同样的思路**证明其正确性**：

算法的每一步在做出一次**贪心选择**时，就会有当前选择的数据对象和剩下的唯一一个待继续求解的子问题。

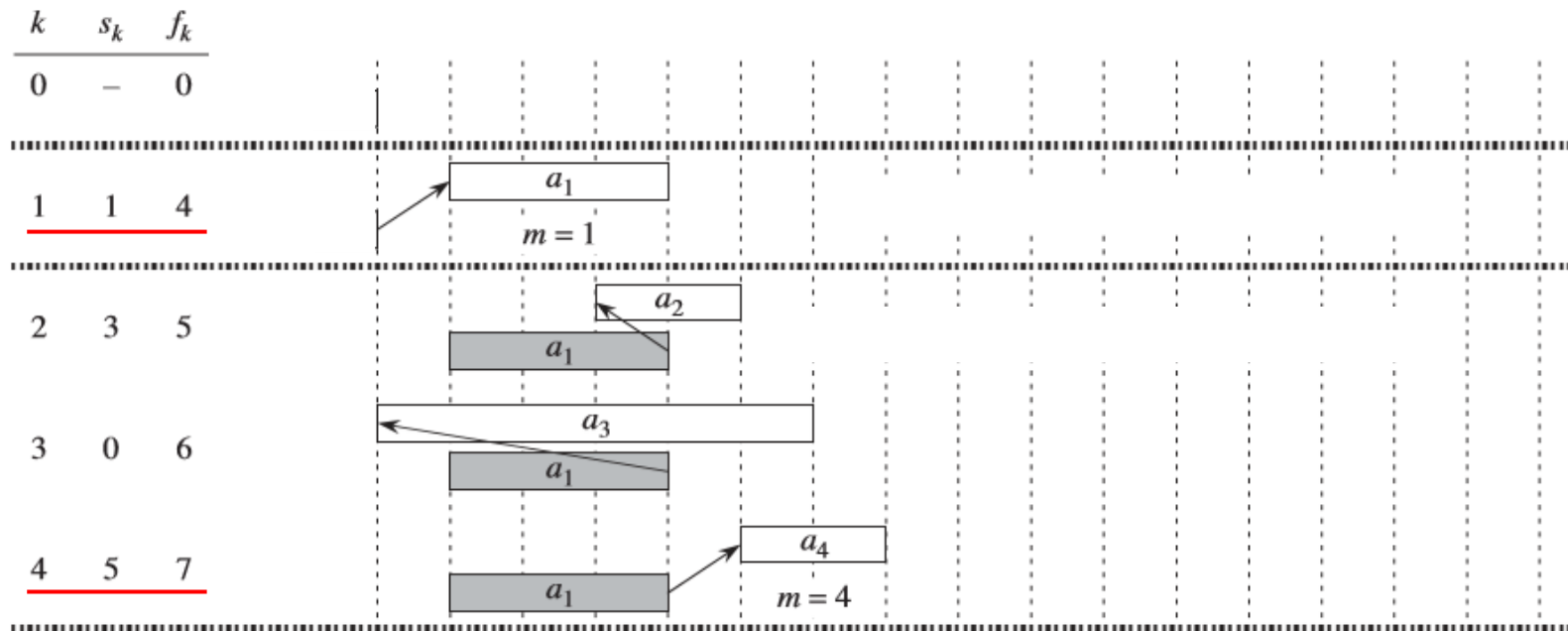
**沿用定理16.1的思路**：证明对当前非空子问题，在算法所用的**贪心选择策略**下所选的数据对象是当前子问题的**某个最优子解中的元素**，则可以证明整个贪心算法的正确性。



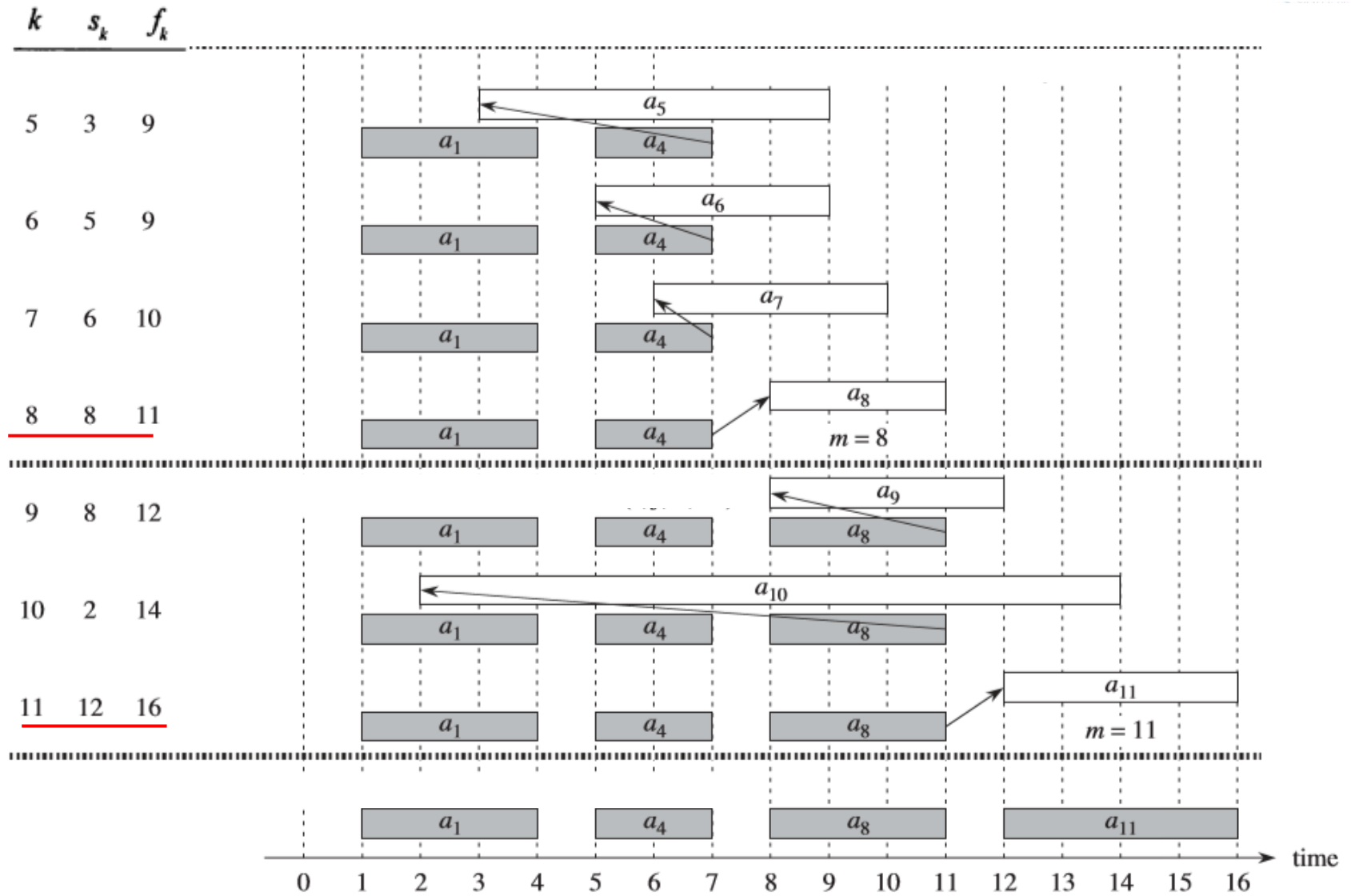
- ◆ 教材P240给出了上述实例**求解过程的图示**，请自行阅读。

$i$	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

执行过程如图所示：



$i$	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



### 3、贪心选择性质

贪心算法的核心思想是：**局部最优选择**。

如果一个问题可以**通过局部最优选择来构造全局最优解**，则称该问题具有**贪心选择性质**。

—— 贪心选择性使得在进行选择时，只需做出**当前看起来最优的选择**，而不用考虑以后子问题的解。

另一方面，能否通过**局部最优选择**构造**全局最优解**的关键是**最优选择策略**。但一个选择策略是不是“最优选择策略”，需要进行证明，而且是**case by case**的。

如定理16.1证明了如果按照活动结束时间的非降顺序依次选择兼容活动，则可以得到活动选择问题的最优解。但如果换一种选择策略，则整个证明过程就要重写。

◆ 但如果问题具有贪心选择性质，则可以“**方便求解**”：

只需按照**既定的选择策略**，一次次做出选择即可：只根据当前情况做出一次选择，只要当前的选择可行就保留，不可行就舍弃，“**不走回头路、不看将来**”（甚至就算错了也要做下去）。

——**可以看作是对最优子结构性最简单、最直接的应用。**

并且一次选择只需要做一次可行性判断，不需要更复杂的运算，所以**贪心算法通常具有更低的时间复杂度**。

**对比动态规划**：动态规划中，每一步需要在“大量”子问题的解中进行组合判断，从中选出当前最优的一种，计算复杂。而且动态规划求解“严重依赖子问题”，需要考虑所有**子问题的各种分解和组合的情况**，需要将较小子问题都求出来，才能计算较大问题，需要一个“**自下而上**”的复杂计算过程。

而贪心策略则认为**用选择策略选出的那个元素就是“最好”选择**，不用和其它元素比较，思路简单。而在一次贪心选择后，就只剩下**唯一**一个子问题留待后续求解，并且对这个子问题进行求解时，**几乎不用考虑前面曾处理过的所有子问题**，也不考虑将来的选择或子问题，只基于自己当前的情况做选择即可。

① 这里所说的 **“几乎不用考虑前面曾处理过的所有子问题”** 的含义是：

在基于自己的情况做一次贪心选择时，仅需将本步选出的数据对象和之前子问题的解（前面子问题选出并保留下来的数据对象）进行一次可行性判断，而**不需要关心以前的那些子问题是什么**。所以和之前的子问题关系“不大”，处理起来更简单。

## ② 贪心算法的求解是一个“自上而下”的过程。

贪心算法在进行第一次选择之前不需求解任何子问题。且第一次选择后，剩下的子问题不仅只有一个，而且**规模比原问题小**——至少去掉了第一次选择出来的那个元素。后续处理也只会把剩下的**待处理子问题的规模变得越来越小**，所以整个处理过程就是一个简单的“**自上而下**”的计算过程。

如活动选择问题：

- ◆ 原始问题：  $S_0 = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}\}$
- ◆ 第一个子问题：  $S_1 = \{a_4, a_6, a_7, a_8, a_9, a_{11}\}$
- ◆ 第二个子问题：  $S_4 = \{a_8, a_9, a_{11}\}$
- ◆ 第三个子问题：  $S_8 = \{a_{11}\}$
- ◆ 第四个子问题：  $S_{11} = \Phi$

## ③ 贪心选择性质和最优子结构性是贪心算法的两个关键要素。

# 16.2 分数背包问题

## 1、问题的描述

已知  $n$  件物品，各具有重量  $(w_1, w_2, \dots, w_n)$  和效益值  $(p_1, p_2, \dots, p_n)$ ， $(w_i > 0, p_i > 0)$ ，及一个可容纳  $M$  重量的背包。

**问：**怎样装包才能使在不超过背包容量的前提下，装入背包的物品的**总效益最大**？

问题的解用一个**向量**  $X = (x_1, x_2, \dots, x_n)$  表示，每个  $x_i$  表示物品  $i$  被放入背包的比例， $0 \leq x_i \leq 1$ 。当物品  $i$  的一部分  $x_i$  放入背包时，可得到  $p_i x_i$  的效益，同时会占用  $w_i x_i$  的容量。

$$\text{背包能够获得的总效益} = \sum_{1 \leq i \leq n} p_i x_i$$

## 问题分析:

① 装入背包的物品的总重量不能超过  $M$ :  $\sum_{1 \leq i \leq n} w_i x_i \leq M$ 。

② 如果当前问题实例的所有物品的总重量不超过  $M$ , 则显然只有把所有的物品都装入背包中才可获得最大效益, 此时应有所有的  $x_i = 1, 1 \leq i \leq n$ 。

③ 如果物品的总重量  $\sum_{1 \leq i \leq n} w_i \geq M$ , 则将有物品无法全部或部分装入。

此时, 由于  $0 \leq x_i \leq 1$ , 所以最终背包可以刚好装入总重量为  $M$  的若干物品 (全部或其中一部分)。而如果这种情况下背包没有装满, 则显然不能获得最大的效益值。



# 问题的形式化描述

◆ **约束条件**:  $\sum_{1 \leq i \leq n} w_i x_i \leq M$

$$0 \leq x_i \leq 1, \quad p_i > 0, \quad w_i > 0, \quad 1 \leq i \leq n$$

◆ **目标函数**:  $\max(\sum_{1 \leq i \leq n} p_i x_i)$

◆ **可行解**: 满足上述约束条件的任一  $(x_1, x_2, \dots, x_n)$  都是问题的一个可行解。显然可行解可能有多多个（甚至是无穷多个）。

◆ **最优解**: 能够使目标函数取最大值的可行解是问题的最优解。显然最优解也可能有多多个。

例 设有  $n = 3$  件物品和 1 个背包，背包容量  $M = 20$ ，物品效益值  $(p_1, p_2, p_3) = (25, 24, 15)$ ，重量  $(w_1, w_2, w_3) = (18, 15, 10)$ 。求该背包问题的解。

可能的可行解如下：

$(x_1, x_2, x_3)$	$\sum w_i x_i$	$\sum p_i x_i$	
① $(1/2, 1/3, 1/4)$	16.5	24.25	//没有装满背包//
② $(1, 2/15, 0)$	20	28.2	
③ $(0, 2/3, 1)$	20	31	
④ $(0, 1, 1/2)$	20	31.5	//最优解//

## 2. 贪心策略求解

### (1) 贪心选择的策略

有三种可能的考虑：

**策略1：**以目标函数作为度量，每装入一件物品都使背包获得最大可能的效益增量。

**具体方法：**按物品效益值的非增次序依次考察物品，每次选择效益值最大的物品装入背包，以获得最大效益增量。如果背包容量不能放下一件物品的全部，则选择可以获得最大效益增量的部分物品放入。

如上例,

$M = 20, (p_1, p_2, p_3) = (25, 24, 15), (w_1, w_2, w_3) = (18, 15, 10)$ 。

- 按物品效益值排序有:  $p_1 > p_2 > p_3$
- 物品装包过程:

首先将物品1放入背包,  $x_1 = 1$ , 背包获得  $p_1 = 25$  的效益增量, 但同时背包容量消耗掉  $w_1 = 18$  个单位, 然后剩余容量  $\Delta M = 2$ 。

由于剩余容量都不足以放入物品2和3的全部, 且就  $\Delta M = 2$  而言, 物品2的  $2/15$  可带来  $16/5$  的效益增量, 大于物品3的  $2/10$  可带来的效益增量 3。

所以为使背包有最大效益增量, 选择物品2的  $2/15$  装包, 即  $x_2 = 2/15$ 。最后,  $\Delta M = 0$ , 背包装满, 物品3不能装,  $x_3 = 0$ , 结束。

最后得到的解:  $(x_1, x_2, x_3) = (1, 2/15, 0)$ ,

背包总效益  $\sum p_i x_i = 28.2$ , 不是最优解。



**策略2：**以容量作为度量，让背包容量尽可能慢地被消耗，从而可以“尽可能多”地装入一些物品。

此种策略下每次选择重量最小的物品装入背包。

**具体方法：**按物品重量的非降次序依次考察物品，每次选择重量最小的物品装入背包。如果背包剩余容量放不下当前重量最小的物品的全部，则放入它的一部分，把背包装满即可。

如上例，

$$M = 20, (p_1, p_2, p_3) = (25, 24, 15), (w_1, w_2, w_3) = (18, 15, 10)。$$

- 按物品重量排序有：  $w_3 < w_2 < w_1$
- 物品装包过程：

首先将物品3放入背包，  $x_3 = 1$ ，背包获得  $p_3 = 15$  的效益增量，同时容量消耗  $w_3 = 10$  个单位，剩余容量  $\Delta M = 10$ 。

然后考虑物品2。就  $\Delta M = 10$  而言，只能选择物品2的  $10/15$  装入背包，所以取  $x_2 = 10/15 = 2/3$ 。

最后，  $\Delta M = 0$ ，背包装满，物品1不能装入背包，  $x_1 = 0$ 。

最后得到的解：  $(x_1, x_2, x_3) = (0, 2/3, 1)$ ，

背包总效益  $\sum p_i x_i = 31$ ，也不是最优解。

## 原因分析:

任何物品放入背包，在背包总效益获得增量的同时，容量也被消耗掉一部分。以上两种策略没有**综合考虑背包的效益增量和容量消耗之间的关系**：或者只考虑了效益增量，而忽略了容量消耗也快；或者只考虑到容量消耗慢，而忽略了效益增量也慢。**在效益增量和容量消耗间没有取得平衡**，故而没有获得问题的最优解。

(具体情况自行分析)。

### 策略3：最优度量标准

基于上述分析，**一种可能的考虑是**：让背包中放入的物品所占用的**每单位容量都能“发挥最大的作用”**——每单位容量都尽可能地装进最大可能效益的物品。

**具体策略**：以**已装入的物品的累计效益值与所用容量之比**为度量，使得每装入一件物品，背包已用容量的**“效益密度”**尽可能大。

**具体方法**：**按物品的  $p_i/w_i$ （即单位效益）的非降次序考察物品**。每次选择单位效益值最大的物品装包。如果背包剩余容量不能放下下一个物品的全部，则放入它的一部分，把背包装满。



如上例,

$M = 20, (p_1, p_2, p_3) = (25, 24, 15), (w_1, w_2, w_3) = (18, 15, 10)$ 。

- 按物品的**效益密度**非降次序排序有:  $p_2/w_2 > p_3/w_3 > p_1/w_1$ 。
- 物品装包过程:

首先将物品2放入背包,  $x_2 = 1$ , 背包获得  $p_2 = 24$  的效益增量, 同时容量消耗  $w_2 = 15$  个单位, 剩余容量  $\Delta M = 5$ 。

然后考虑物品3, 就  $\Delta M = 5$  而言, 只能选择物品3的  $5/10$ , 放入背包, 所以  $x_3 = 5/10 = 1/2$ ,

最后,  $\Delta M = 0$ , 物品1不能装入背包,  $x_1 = 0$ 。

最后得到的解是  $(x_1, x_2, x_3) = (0, 1, 1/2)$ ,

$$\text{背包总效益 } \sum p_i x_i = 31.5$$

是问题的**最优解**吗?

### (3) 背包问题的贪心算法

**GREEDY - KNAPSACK**( $P, W, M, X, n$ )

$X = 0$       //将解向量初始化为 0//

$cu = M$     // $cu$  是背包的**剩余容量**//

**for**  $i=1$  **to**  $n$

**if**  $W[i] > cu$

**break**    //剩余容量不足以放下  $i$  物品

$X[i] = 1$       //可以放下  $i$  物品的全部

$cu = cu - W[i]$     //容量消耗

**if**  $i \leq n$

$X[i] = cu/W[i]$     //最后剩余的容量放下  $i$  物品的一部分

**end** GREEDY-KNAPSACK

**说明:**

- ◆  $P[1..n]$ 和 $W[1:n]$ 分别含有按 $P[i]/W[i] \geq P[i+1]/W[i+1]$ 排序的  $n$  件物品的效益值和重量。
- ◆  $M$  是背包的容量大小。
- ◆  $X[1:n]$ 是解向量。

◆物品排序时间:  $O(n \log n)$

◆物品选择时间:  $O(n)$

◆如果预排序, 算法总的时间复杂度是  $O(n \log n)$ .

### 3. 背包问题贪心算法的最优解证明

如何证明贪心解是最优解？

—— 只要贪心解能和其它任意最优解一样，使目标函数取得极值，它就是最优解。

**定理16.2** 如果  $p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n$ ，则算法GREEDY-KNAPSACK 对于给定的背包问题实例生成一个最优解。

证明:

设  $X = (x_1, x_2, \dots, x_n)$  是GREEDY-KNAPSACK所生成的贪心解。

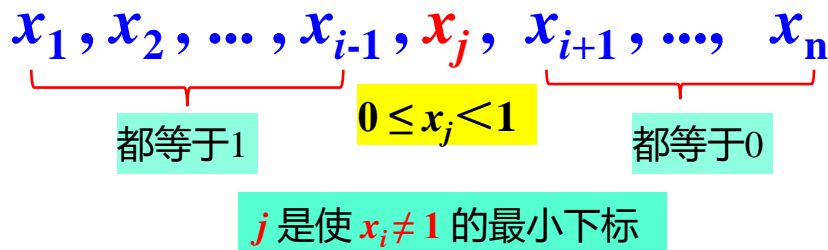
① 如果所有的  $x_i$  都等于 1, 则显然  $X$  是问题的最优解。否则,

② 设  $j$  是使  $x_i \neq 1$  的最小下标。由算法的执行过程可知,

◆  $x_i = 1 \quad 1 \leq i < j,$

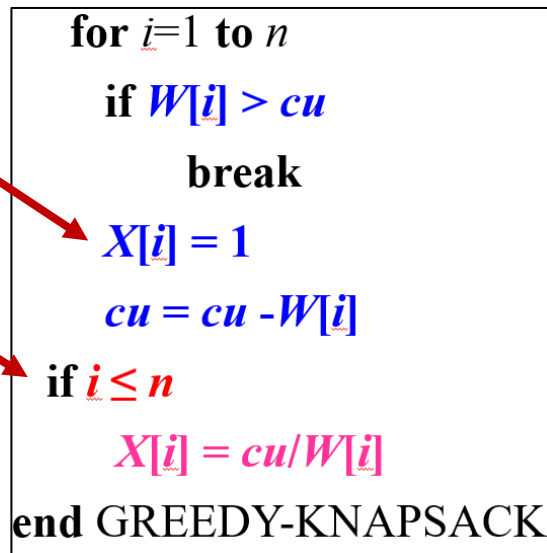
◆  $0 \leq x_j < 1,$

◆  $x_i = 0 \quad j < i \leq n$



```

for  $i=1$  to  $n$ 
  if  $W[i] > cu$ 
    break
   $X[i] = 1$ 
   $cu = cu - W[i]$ 
  if  $i \leq n$ 
     $X[i] = cu / W[i]$ 
end GREEDY-KNAPSACK
  
```



假设  $Y$  是问题的“理想”最优解：  $Y = (y_1, y_2, \dots, y_n)$  ,

不失一般性，应有：  $\sum w_i y_i = M$  ,  $0 \leq y_i \leq 1$

- ◆ 若  $X = Y$ ，毫无疑问， $X$  就是最优解。否则，
- ◆  $X$  和  $Y$  至少在 1 个分量上存在不同。

设  $k$  是使得  $y_k \neq x_k$  的最小下标。

$$\begin{array}{ccccccc} x_1 & x_2 & \dots & x_{k-1} & x_k & \dots & x_n \\ \uparrow & \uparrow & & \uparrow & \uparrow & & \\ = & = & & = & \neq & & \\ \downarrow & \downarrow & & \downarrow & \downarrow & & \\ y_1 & y_2 & \dots & y_{k-1} & y_k & \dots & y_n \end{array}$$

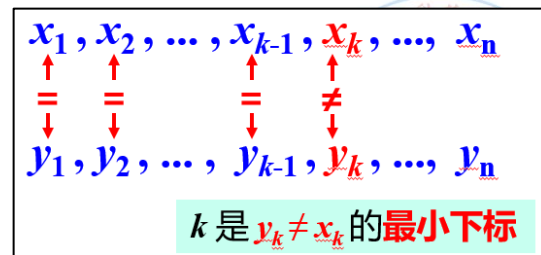
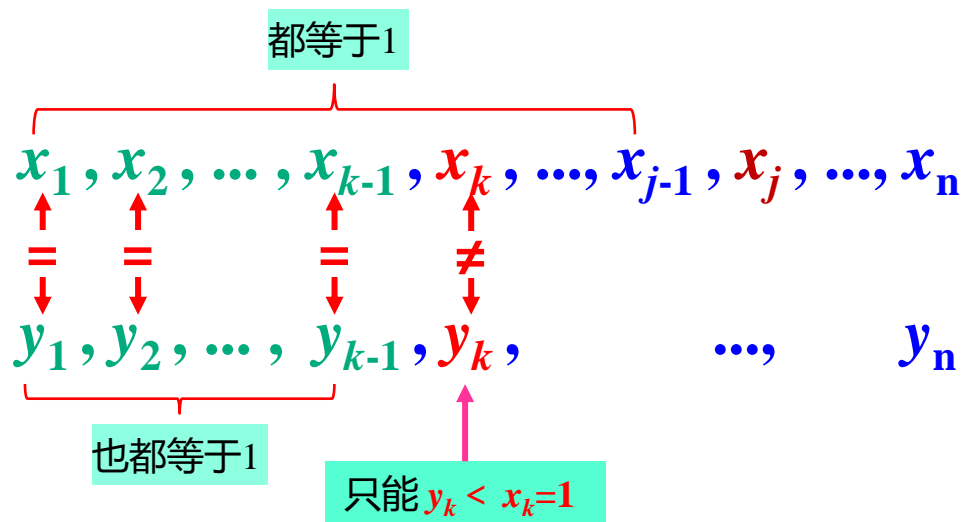
$k$  是  $y_k \neq x_k$  的最小下标

则有  $y_k < x_k$ 。（为什么？）

# 为什么 $y_k < x_k$ ?

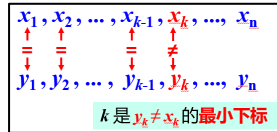
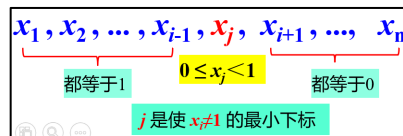
分以下情况说明:

(1) 若  $k < j$ :



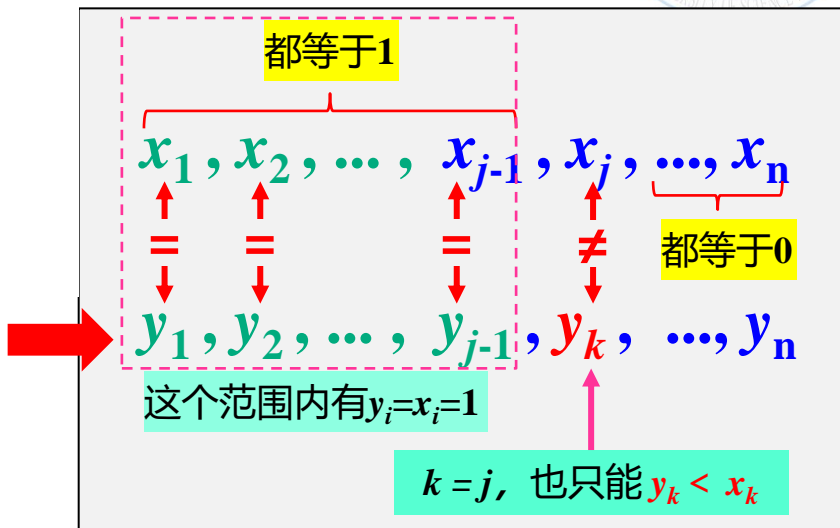
因为此处  $x_k = 1$ , 而  $y_k \neq x_k$  且  $0 \leq y_k \leq 1$ , 故只能  $y_k < 1$ ,  
所以  $y_k < x_k$ 。

(2) 若  $k=j$ :



如图所示:

- ① 对所有  $1 \leq i < j$ ,  $x_i = 1$ , 而  $j = k$ ,  
且  $k$  是  $y_k \neq x_k$  的最小下标, 所以  
对所有  $1 \leq i < k$ ,  $y_i = x_i = 1$ 。



- ② 虽然  $\sum_{1 \leq i \leq n} w_i x_i = M$ , 但对位置  $j$ , 由于所有  $j < i \leq n$ ,  $x_i = 0$ , 故事

实上是  $\sum_{1 \leq i < j} w_i x_i + w_j x_j = M$ ; 故此时若  $y_k > x_k$  (即  $y_k > x_j$ ), 就会

有  $\sum_{1 \leq i < j} w_i x_i + w_j y_k > M$ , 亦即  $\sum_{1 \leq i < k} w_i y_i + w_k y_k > M$ , 与  $Y$  是可行解相矛

盾。故只能  $y_k \leq x_k$ , 而  $y_k \neq x_k$ , 所以也只能有  $y_k < x_k$ 。

(3) 若  $k > j$ :

如图所示:

① 同 (2) 中的分析, 对位置  $j$ ,

由于对所有  $j < i \leq n$ ,  $x_i = 0$ ,

故实际有  $\sum_{1 \leq i \leq j} w_i x_i = M$ ; 故此时

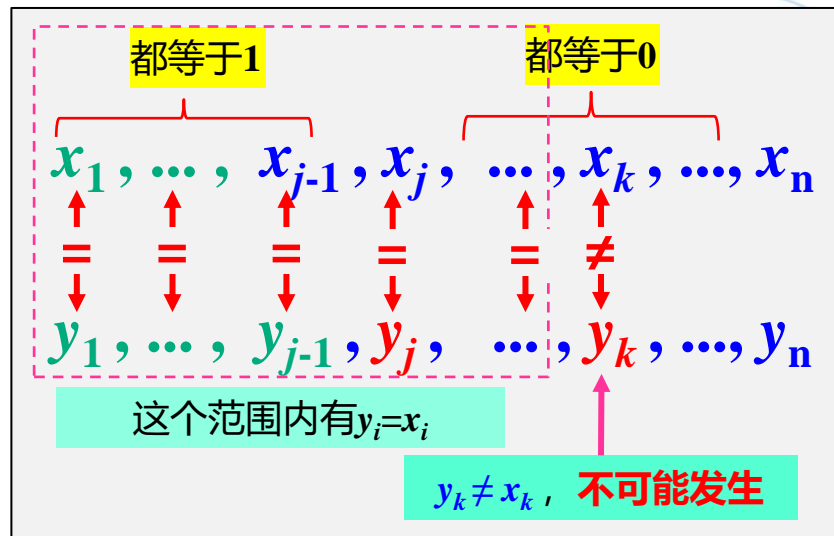
对  $1 \leq i \leq j$ , 因为  $j < k$ , 所以不仅有  $y_i = x_i$ , 还有  $\sum_{1 \leq i \leq j} w_i y_i = M$ 。

② 同时,  $j < k$  时  $x_k = 0$ , 若  $y_k > x_k$  就意味着  $y_k > 0$ , 也就意味着

$\sum_{1 \leq i \leq j} w_i y_i + w_k y_k > M$ , 与  $Y$  是可行解相矛盾。故只能  $y_k < x_k$ , 也

就是  $y_k < 0$ , 这又与  $0 \leq y_k \leq 1$  相矛盾。故这样的位置  $k$  事实上

是不存在的。所以  $k > j$  的情况不会发生。





上面证明了  $y_k < x_k$ , 下面继续最优解证明:

现在做一个调整: 在  $Y$  中将  $y_k$  增加到  $x_k$ 。目的: 将第一个不同点消去而使得在  $k$  下标处  $X$  和  $Y$  的分量取值一样。

但这也意味着对  $Y$  而言  $k$  物品装入背包的重量多了  $(x_k - y_k) w_k$ , 而造成“新解”不可行。

故, 为保持解的可行性, 再从  $(y_{k+1}, \dots, y_n)$  中减去同样多的重量 (可以通过减小任意  $y_j$  实现,  $k+1 \leq j \leq n$ )。

设经过上述调整以后得到的解为  $Z = (z_1, z_2, \dots, z_n)$ , 其中  $z_i = x_i$ ,  $1 \leq i \leq k$ 。则对  $Z$  有:

$$\textcircled{1} \quad \sum_{k < i \leq n} w_i (y_i - z_i) = w_k (z_k - y_k)$$

即从  $(y_{k+1}, \dots, y_n)$  中减去的重量等于在  $y_k$  处增加的重量。

## ② Z 的效益值有:

$$\begin{aligned}\sum_{1 \leq i \leq n} p_i z_i &= \sum_{1 \leq i \leq n} p_i y_i + (z_k - y_k) w_k p_k / w_k - \sum_{k < i \leq n} (y_i - z_i) w_i p_i / w_i \\ &\geq \sum_{1 \leq i \leq n} p_i y_i + (z_k - y_k) w_k p_k / w_k - \sum_{k < i \leq n} (y_i - z_i) w_i p_k / w_k\end{aligned}$$

对所有  $k < i \leq n$ ,  
 $p_k / w_k > p_i / w_i$

$$= \sum_{1 \leq i \leq n} p_i y_i + \underbrace{[(z_k - y_k) w_k - \sum_{k < i \leq n} (y_i - z_i) w_i] p_k / w_k}_{\text{差值} = 0}$$

$$= \sum_{1 \leq i \leq n} p_i y_i$$

即, 对Y经过上述调整后得到的“新解”Z, 不仅可行, 而且效益值不少于原来的最优解Y。

进而,

- ◆ 若  $\sum p_i z_i > \sum p_i y_i$  则  $Y$  就不是最优解 (矛盾, 所以不可能大于)。
- ◆ 若  $\sum p_i z_i = \sum p_i y_i$ , 则,
  - 或者  $Z = X$ , 则  $X$  就和  $Z$  ( $Y$ ) 一样具有最大效益值, 所以  $X$  就是最优解;
  - 或者  $Z \neq X$ , 则基于  $Z$  重复以上变换过程, 直至或者证明  $Y$  不是最优解 (矛盾, 不可能成立), 或者把  $Y$  转换成  $X$ , 从而证明  $X$  是最优解。

证毕 ■

## ◆ 证明贪心解是最优解的基本策略 (如定理16.1和定理16.2)

首先**设出最优解的一般形式**，然后**看贪心解和“最优解”有什么不同**：

- ① 如果二者完全一样，则毫无疑问，贪心解就是最优解；
- ② 如果二者不同，则一定存在不同的地方。于是：

设法**定义出二者之间第一个不同的地方** (如  $S_k$  中最早完成的作业  $a_m$ 、 $Y$  中  $y_i \neq x_i$  的最小下标  $k$ )，**然后设法消去这个不同** (如用  $a_j$  替换  $a_m$ ，用  $x_k$  替换  $y_k$ )；

**证明变换后“新形式的最优解”的最优性依然得以保持。**

重复这种变换，最终**将最优解变成和贪心解一样的形式**，也就说明贪心解和最优解一样，能够达到最优的要求 (可以使目标函数取得极值)，所以**贪心解也是一个最优解**。

## ◆ 分数背包问题和0-1背包问题

**分数背包问题**：解向量  $X$  中的  $x_i$  取值是  $0 \leq x_i \leq 1$ 。代表**物品可以分割**，可以恰好“装满”背包。

**0-1 背包问题**： $x_i$  只能取 0 或 1，代表物品要么装、要么不装，**不能被分割**，因而背包有可能装不满。

如  $M=30$ ,  $(p_1, p_2, p_3) = (25, 24, 15)$ ,  $(w_1, w_2, w_3) = (18, 15, 10)$ 。

0-1背包问题的解是  $X = (1, 0, 1)$ ，有2个单位的容量空闲，总效益是40。

## ◆ 0-1背包问题一般不能用贪心方法求解。

如上例，如果用和前面一样的贪心方法求解，得到的解是  $X = (0, 1, 1)$ ，有5个单位的容量空闲，总效益是39。

**对0-1背包问题可以设计动态规划算法求解** (查阅资料学习)。

## ◆ 贪心方法和动态规划的比较

(1) 二者都要求问题满足**最优子结构性**质。

因而，一个可以用贪心方法求解的问题，都会对应地有一个动态规划算法求解该问题。但显然，贪心方法简单一些，动态规划方法复杂一些。

(2) 贪心策略除要求问题具有最优子结构性质以外，还要求具有**贪心选择性质**，否则无法通过贪心选择正确求解。

贪心算法依赖“**最优选择策略**”，而一个策略是不是最优策略需要进行证明。

动态规划全面考虑所有子问题的组合情况，然后从中选出最优方案，所以“**自带证明**”，不用再单独证明。



(3) 贪心算法是“**自上向下**”的计算过程。每一次选择最多和前面的子问题的解有一定相关性，但不关心后续子问题，直接做出自己的选择即可。

动态规划是“**自下而上**”的计算过程。每一步计算都要考虑相关的所有子问题，必须先求解出“下面”的所有小问题，才能“向上”求解大问题。



课堂练习：利用贪心策略求解下列背包问题。

设,  $n = 4$ ,  $M = 54$ ,

$$(p_1, p_2, p_3, p_4) = (20, 16, 10, 18)$$

$$(w_1, w_2, w_3, w_4) = (16, 12, 15, 24)$$

求解向量  $X$  和  $\sum p_i x_i$ .

参考答案:

$$p_1/w_1=4/5, \quad p_2/w_2=4/3, \quad p_3/w_3=2/3, \quad p_4/w_4=3/4$$

$$p_2/w_2 > p_1/w_1 > p_4/w_4 > p_3/w_3$$

$$x_2=1 \quad x_1=1 \quad x_4=1 \quad x_3=2/15$$

$$X = (1, 1, 2/15, 1)$$

$$\sum w_i x_i = 54$$

$$\sum p_i x_i = 166/3$$



# 16.4 Huffman编码

注：也可用十六进制、十进制及其它形式表示编码，这里仅研究二进制编码。

## 1、编码的基本概念

◆ **字符编码**：用二进制数字表示字符。

如：ASCII码，格雷码、GB2312等。

◆ 字符的二进制数串表示称为字符的**码字**。

◆ 每个字符的二进制编码的位数称为该字符的**编码长度**。

■ **定长编码**：每个字符的编码长度一样。

如 8位的ASCII码，1~9 的四位二进制编码。

■ **变长编码**：每个字符的编码长度不同。

六个字符的定长和不定长编码						
	a	b	c	d	e	f
定长编码	000	001	010	011	100	101
变长编码	0	101	100	111	1101	1100

- ◆ **编码规则**：字符向编码的映射关系，是编码的**核心算法**。
  - 不同的编码方案有不同的编制规则（不同的算法）。
- ◆ **文件编码**：将一个文件中的所有字符用其**二进制编码**表示，最后得到整个文件的**二进制数串表示形式**的过程。
- ◆ **文件编码长度**：一个编码文件的总的**二进制位数**称为该文件的编码长度。
- ◆ **最优编码方案**：能够使文件编码长度最小的编码方案称**最优编码方案**。

◆ **解码过程**：将一个用**二进制数串**表示的**编码文件**转换为原始字符文件的过程。

如，设文件中包含3个连续的字符：**abc**

六个字符的定长和不定长编码						
	a	b	c	d	e	f
定长编码	000	001	010	011	100	101
变长编码	0	101	100	111	1101	1100

① 文件的定长编码：**000001010**  
↓ ↓ ↓  
解码： a b c

② 文件的不定长编码：**0101100**  
↓ ↓ ↓  
解码： a b c

◆ **前缀码**：任何字符的码字都不是其它字符码字的前缀。

如：

六个字符的前缀码和非前缀码编码						
	a	b	c	d	e	f
前缀码	0	101	100	111	1101	1100
非前缀码	0	01	10	001	0011	0110

由于前缀码中没有任何字符的码字是其它字符码字的前缀，所以在对编码文件解码时，可以**无歧义地将编码文件中的字符一一解读出来**，唯一转换成原字符文件。否则无法正确解读。

如，abc，

① 前缀码编码：0101100

解码：

↓ ↓ ↓  
a b c

唯一翻译

② 非前缀码编码：00110

解码：

abc? dc? ca? af?

## 2、最优编码方案

### (1) 编码树

- ◆ **编码树**：一种为表示字符二进制编码而构造的二叉树。

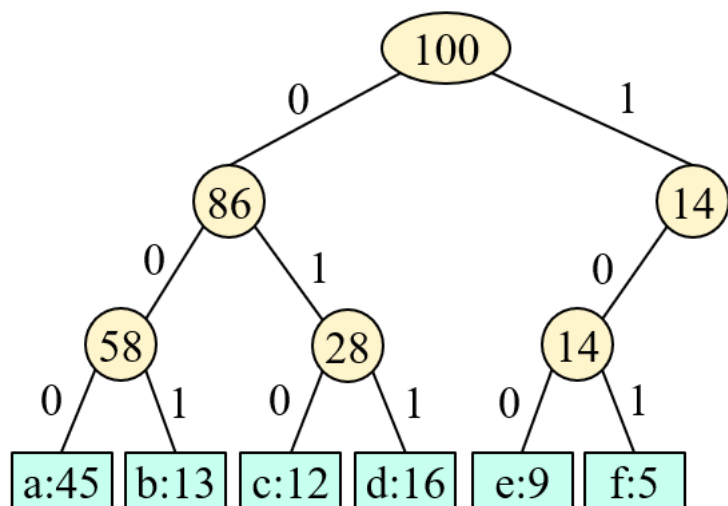
编码树中，**叶子结点**对应字符，内部结点按照某种规则组织。

- ◆ **编码构造**：从**根结点**开始，对每个内部结点至其儿子的**边**标上**0 或 1**，**0** 代表转向左孩子，**1** 代表转向右孩子。

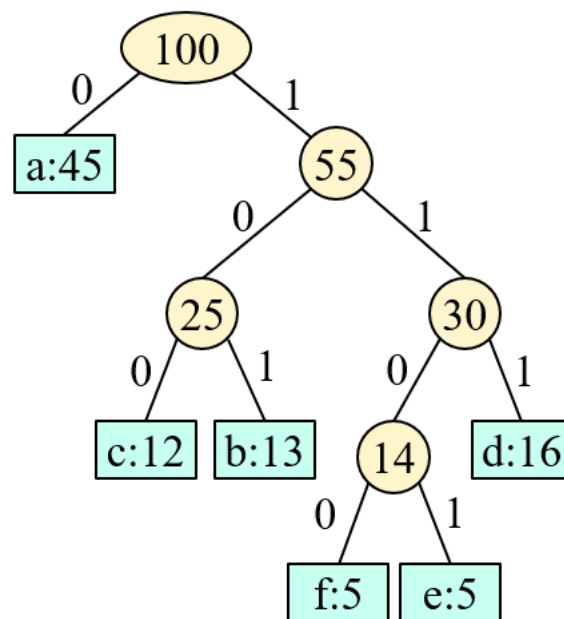
则一个字符的二进制编码字由**根结点到该字符叶子结点的简单路径上的0-1序列**表示。

如：对同样6个字符的两棵编码树

- ◆ **叶子结点**：代表**字符**，数字代表该字符出现的**频率**。
- ◆ **内部结点**：数字代表其子树中叶子结点的频率之和。



方案(a)



方案(b)

两棵编码树给出的两种6个字符的编码方案

	a	b	c	d	e	f
方案(a)	000	001	010	011	100	101
方案(b)	0	101	100	111	1101	1100

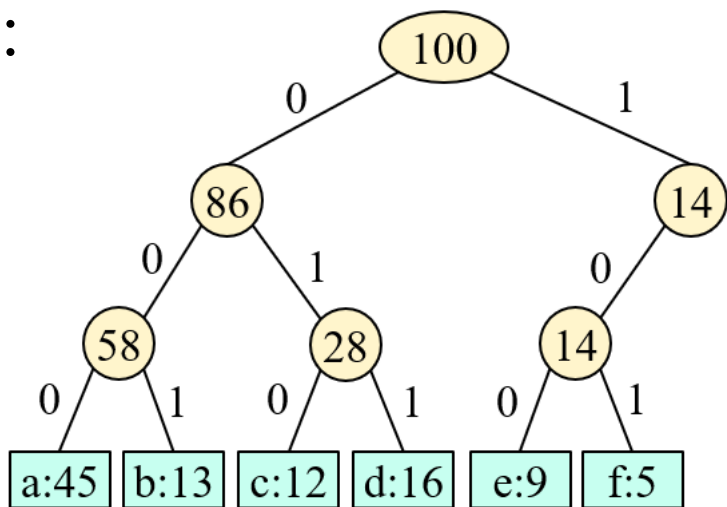
定长编码

不定长编码

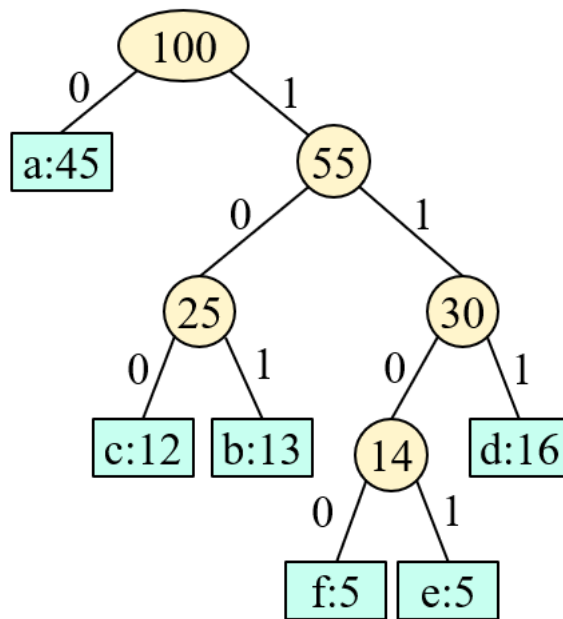
- ◆ **满二叉树**：如果二叉树中的每个**非叶子结点**都有两个孩子结点，则称这样的二叉树为**满二叉树**。

(注意和数据结构中满二叉树定义的区别)

如：



方案(a) 非满二叉树



方案(b) 满二叉树

- ◆ 一个文件的最优字符编码方案至少和一棵满二叉树对应。

设  $C$  为字母表（待编码字符表）。对  $C$  中的任意字符  $c$ ，令属性  $c.freq$  表示字符  $c$  在文件中出现的频率。不失一般性，设所有字符的出现频率均为正数。

令  $T$  表示一棵**前缀码编码树**；

令  $d_T(c)$  表示  $c$  对应的叶子结点在树  $T$  中的深度（即根到叶子结点的路径长度，或路径上的边数），则  $d_T(c)$  就是该编码树对应编码方案中**字符  $c$  的码字长度**。

再令  $B(T)$  表示采用编码方案  $T$  时的文件编码长度，则：

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

即编码文件的总二进制位数。



称  $B(T)$  为用  $T$  对文件进行编码的**代价**，则对给定的字符集  $C$  和文件，使  $B(T)$  最小的编码方案称为**最优编码方案**。

- ◆ **Huffman** 编码是一种最优变长编码。  
据研究，可以节省20%~90%的编码量。
- ◆ 最优前缀码对应的编码树中恰好有  $|C|$  个**叶子结点**，每个叶子结点对应字母表中的一个字符，且恰有  $|C| - 1$  个**内部结点**。

### 3、Huffman编码的贪心算法

算法HUFFMAN从  $|C|$  个叶子结点开始，每次选择**频率最低**的两个结点合并，将得到的新结点加入集合并继续合并。这样执行  $|C| - 1$  次合并后即可构造出一棵编码树，称为 *Huffman 树*。

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**  $\text{EXTRACT-MIN}(Q)$

从队列  $Q$  中提取两个**最低频率**的对象

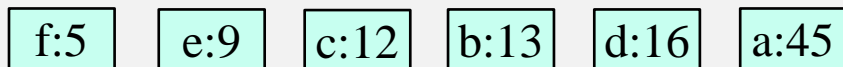
合并：新对象的频率等于原来两个对象的频率之和

将新对象插入结点集 $Q$

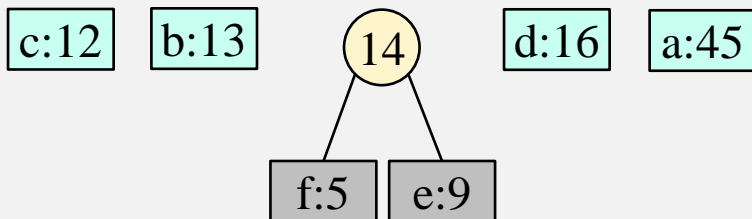
# 例：构造前面实例的Huffman编码

6个字符的频率						
	a	b	c	d	e	f
频率( $10^3$ )	45	13	12	16	9	5

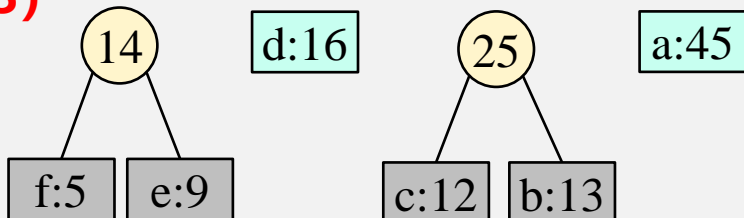
(1)



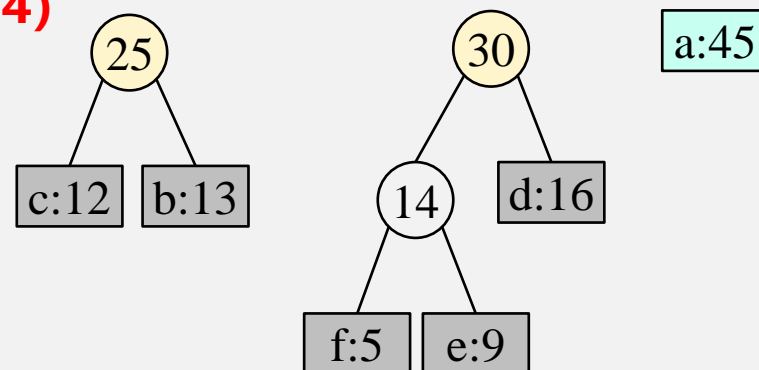
(2)



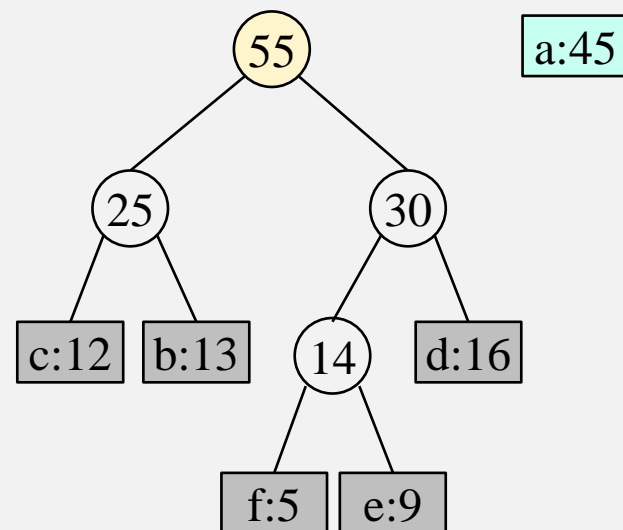
(3)



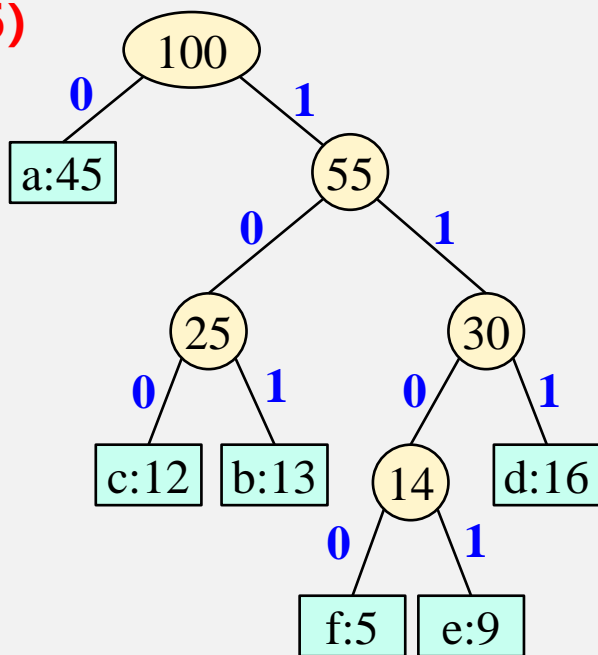
(4)



(5)



(6)



最后生成的Huffman树。并为边上**左0、右1**，然后得到各个字符的Huffman编码。

6个字符的频率和Huffman编码

	a	b	c	d	e	f
频率( $10^3$ )	45	13	12	16	9	5
Huffman编码	0	101	100	111	1101	1100

## 基本规则：

- ◆ 叶子结点用矩形表示，内结点用圆形结点表示，
- ◆ 每次选择**频率最低**的两棵子树合并，生成一棵新子树，新子树的频率等于其孩子结点的频率之和，然后参与后续合并。
- ◆ **字符的码字等于从根到其叶子结点的路径上边的0-1序列。**

# HUFFMAN算法的时间分析

◆ 可以使用**最小二叉堆**实现  $Q$  , 则

- 首先,  $Q$  的初始化 (建堆) 花费  $O(n)$  的时间。
- 其次, 循环的总代价是  $O(n \log n)$ 。

每次,  $\text{EXTRACT-MIN}(Q)$  从堆中找出当前频率最小的结点需要  $O(\log n)$  ,  $\text{INSERT}(Q, z)$  将新结点  $z$  插入到堆中花费  $O(\log n)$ ; **for** 循环共执行了  $n-1$  次, 所以循环的总代价是  $O(n \log n)$ 。

所以, HUFFMAN的总运行时间  **$O(n \log n)$** 。

◆ 可以用van Emde Boas树实现  $Q$  (Chp 20) , 可以将运行时间减少到  $O(n \log \log n)$

HUFFMAN(  $C$  )

1  $n = |C|$

2  $Q = C$

3 **for**  $i = 1$  **to**  $n - 1$

4     allocate a new node  $z$ .

5      $z.\text{left} = x = \text{EXTRACT-MIN}(Q)$

6      $z.\text{right} = y = \text{EXTRACT-MIN}(Q)$

7      $z.\text{freq} = x.\text{freq} + y.\text{freq}$

8     **INSERT**( $Q, z$ )

9 **return**  $\text{EXTRACT-MIN}(Q)$

## 4、HUFFMAN算法的正确性（最优编码方案证明）

由下面的引理16.2、16.3和定理16.4给出。

**引理 16.2** 令  $C$  为一个字母表，其中每个字符  $c \in C$  都有一个频率  $c.freq$ 。令  $x$  和  $y$  是  $C$  中频率最低的两个字符。那么存在  $C$  的一个最优前缀码，其中  $x$  和  $y$  的码字长度相同，且只有最后一个二进制位不同。

证明：

令  $T$  是一个最优前缀码所对应的编码树——满二叉树。

令  $a$  和  $b$  是  $T$  中深度最大的两个兄弟叶结点。

- ◆ 不是一般性，假设  $a.freq \leq b.freq$  且  $x.freq \leq y.freq$ 。
- ◆ 由于  $x$  和  $y$  是叶结点中频率最低的两个结点，所以应有

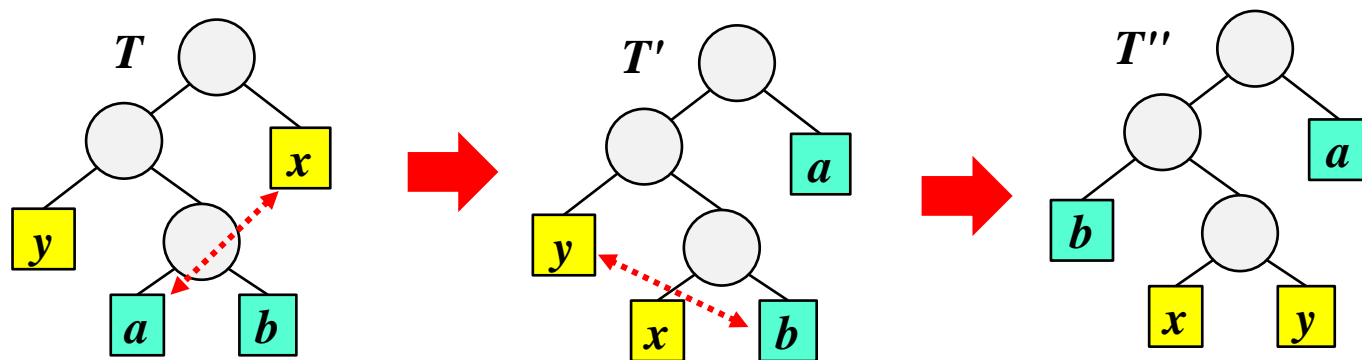
$$x.freq \leq a.freq \text{ 且 } y.freq \leq b.freq。$$

注：允许  $x.freq = a.freq$  或  $y.freq = b.freq$ 。

- 若  $x.freq = b.freq$ ，则有  $a.freq = b.freq = x.freq = y.freq$ ，此时引理显然成立。

此时，通过互换  $x$  和  $a$ 、 $y$  和  $b$ （或者  $x$  就是  $a$ 、 $y$  就是  $b$ ），不改变前缀码特性及树的编码代价，得到一棵新树。新树中  $x$  和  $y$  成为兄弟叶结点，所以**码字长度相同**，且**只有最后一个二进制位不同**。

- 若  $x.freq \neq b.freq$ ，一定有  $x \neq b$ 。则在  $T$  中交换  $x$  和  $a$ ，生成一棵新树  $T'$ ；然后再在  $T'$  中交换  $b$  和  $y$ ，生成另一棵新树  $T''$ ，那么在  $T''$  中  $x$  和  $y$  成为深度最深的两个兄弟结点。如图所示：



注：  $T$  是假设的最优编码树，并不保证  $x$  和  $y$  在  $T$  中是兄弟叶子结点，也不保证它们处在最深一层，它们可出现在  $T$  中的任意合理位置上。叶子结点  $a$  和  $b$  是特指的最深的叶子结点中的两个，并且是兄弟结点。假设  $x \neq b$ ，交换叶子结点  $a$  和  $x$  得到树  $T'$ ，然后交换叶子结点  $b$  和  $y$  得到树  $T''$ 。

根据文件编码长度  $B(T)$  的计算公式，计算  $T$  和  $T'$  的代价差：

$$B(T) - B(T')$$

$$= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c)$$

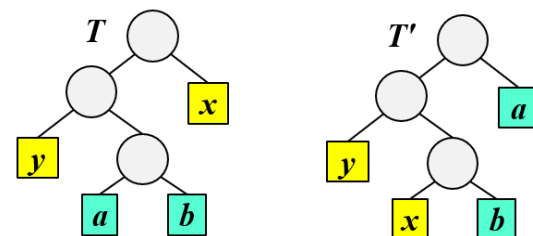
$$= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a)$$

$$= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x)$$

$$= (a.freq - x.freq)(d_T(a) - d_T(x))$$

$$\geq 0 \quad \text{注, } a.freq \geq x.freq, d_T(a) \geq d_T(x)$$

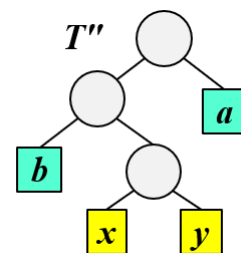
$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$



$B(T) - B(T') \geq 0$  表示从  $T$  到  $T'$  并没有增加代价。

类似地，从  $T'$  到  $T''$ ，同理可证：交换  $y$  和  $b$  也不没有增加代价，即  $B(T') - B(T'') \geq 0$ 。

因此， $B(T) \geq B(T'')$ ，即  $T''$  的代价不比  $T$  大。





根据假设,  **$T$  是最优的**, 因此  **$T''$**  的代价不可能比  $T$  更小,  
所以只能有  **$B(T'') = B(T)$** 。

即得证:  **$T''$  也是最优解**, 且  $x$  和  $y$  是  $T''$  中深度最大的两个兄弟结点, 二者的码字长度相同, 且只有最后一个二进制位不同。

引理得证■

### 引理16.2表明:

对字符集  $C$  一定存在一棵具有 “ **$x$  和  $y$  为兄弟结点**” 结构特征的最优编码树。

**引理 16.3** 令  $C$  为一个给定的字母表，其中每个字符  $c$  都有一个频率  $c.freq$ 。

- ◆ 令  $x$  和  $y$  是  $C$  中频率最低的两个字符。
- ◆ 令  $C'$  为  $C$  去掉字符  $x$  和  $y$ 、并加入一个新字符  $z$  后得到的字母表： $C' = C - \{x, y\} \cup \{z\}$ 。
- ◆ 类似  $C$ ，也为  $C'$  中的字符定义  $freq$ ，并令：  

$$z.freq = x.freq + y.freq$$
，而其它字符的频率不变。
- ◆ 设  $T'$  是字母表  $C'$  的任意一个最优前缀码对应的编码树。

**则有：**若将  $T'$  中叶子结点  $z$  替换为一个以  $x$  和  $y$  为孩子的内部结点，得到的新树  $T$  将是代表字母表  $C$  的最优编码的编码树。

## 证明:

由于从  $T'$  变换到  $T$ , 是对  $T'$  的叶子结点  $z$  做的替换: **只为  $z$  增加了  $x$  和  $y$  两个儿子**。而  $C$  中的其它字符  $c \in C - \{x, y\}$ , 它们在树  $T$  仍是叶子结点, 位置和深度与原来在  $T'$  中是一样的, 即

$$d_T(c) = d_{T'}(c), \quad c \in C - \{x, y\}$$

也就有:  $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$ 。

而对  $T$  中的  $x$ 、 $y$ , 由于它们是  $z$  的儿子, 它们在  $T$  中的**深度相同**但在  $T$  中的深度值比  $z$  在  $T'$  中的深度值大**1**, 即:

$$d_T(x) = d_T(y) = d_{T'}(z) + 1$$

故计算  $x$  和  $y$  在  $T$  中的成本贡献就有:

$$\begin{aligned} & x.freq \cdot d_T(x) + y.freq \cdot d_T(y) \\ &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

再考虑其它字符  $c \in C - \{x, y\}$ , 由于  $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$ ,

从而可得:  $B(T)$

$$\begin{aligned} &= \sum_{c \in C} c.freq \cdot d_T(c) \\ &= \sum_{c \in C - \{x, y\}} c.freq \cdot d_T(c) + x.freq \cdot d_T(x) + y.freq \cdot d_T(y) \\ &= \sum_{c \in C - \{x, y\}} c.freq \cdot d_{T'}(c) + \underline{x.freq \cdot d_T(x) + y.freq \cdot d_T(y)} \\ &\quad \quad \quad \downarrow \\ &= \sum_{c \in C - \{x, y\}} c.freq \cdot d_{T'}(c) + z.freq \cdot d_{T'}(z) + x.freq + y.freq \\ &= \sum_{c \in C'} c.freq \cdot d_{T'}(c) + x.freq + y.freq \\ &= B(T') + x.freq + y.freq \end{aligned}$$

亦即:  $B(T') = B(T) - x.freq - y.freq$

## 用反证法证明 $T$ 代表 $C$ 的最优（前缀码）编码：

假定  $T$  对应的编码方案不是  $C$  的最优编码方案。

则，设树  $T''$  是  $C$  的一个最优编码树，并且不失一般性，根据引理 16.2，设这个  $T''$  是恰好有 “ $x$  和  $y$  为兄弟结点” 结构特征的一棵最优编码树，且有： $B(T'') < B(T)$ 。

且，再令  $T'''$  是在  $T''$  中删去  $x$ 、 $y$  的叶子结点，并使其父结点成为叶结点  $z$  后得到的一棵树，且  $z.freq = x.freq + y.freq$ 。则用和前面推导类似的推导可得：（可将这里的  $T'''$  对应到前面的  $T'$ ） $T'''$  对应的字符集就是  $C'$

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') \end{aligned}$$

这与  $T'$  是  $C'$  的一棵最优前缀码编码树相矛盾。故假设不成立。  
因此， $T$  就必然表示字母表  $C$  的一个最优前缀码。 证毕。

**定理 16.4** 过程HUFFMAN会生成一个最优前缀码。

**证明：**由引理16.2和引理16.3可直接得出。

## 5、贪心选择性的确立：

**引理16.2**说明**首次选择频率最低**的两个字符和选择其它可能的字符一样，都可以构造出的最优编码树。

**引理16.3**说明首次选出频率最低的两个字符  $x$  和  $y$ 、合并后将  $z$  加入元素集合后，**接着可以继续构造包含  $z$  的最优编码树**。以此类推就可以得到字符集的最优编码树。

所以**贪心选择性成立**，过程HUFFMAN最终会生成一个关于字符集  $C$  和已知频率的最优前缀码。



# 16.5 最优归并模式问题

## 1、归并问题

(1) **两个文件的归并**：把两个分别包含  $n$  和  $m$  个记录的文件合并成一个包含所有  $n + m$  个记录的文件。

**具体方法**：把两个文件的记录依次挪到第三个文件里即可。

(事实上如同MergeSort里的Merge过程)

故，这样的两个文件的一次归并时间 =  $O(n+m)$

(2) **多个文件的归并**：把多个各包含若干个记录的文件合并成一个包含所有记录的文件。

**具体方法**：如何合并？

**基本策略**：**两两归并**。**计算代价**呢？

## 2、归并模式

当有**多个文件**需要归并时，存在一个**归并模式**的问题

（即第一步选择哪两个文件归并，然后再选择哪两个文件进行第二次归并，……，直到最后得到归并好的文件）。

对已知的  $n$  个文件，将它们归并成一个单一的文件会有**不同**的归并模式。

例：假定有4个文件  $X_1, X_2, X_3, X_4$ ，采用**两两归并**的方式，可能的归并模式有：

$$\textcircled{1} X_1 + X_2 = Y_1 + X_3 = Y_2 + X_4 = Y_3$$

$$\begin{array}{l} \textcircled{2} X_1 + X_2 = Y_1 \\ \quad \quad \quad + \\ X_3 + X_4 = Y_2 \end{array} \left. \vphantom{\begin{array}{l} X_1 + X_2 = Y_1 \\ + \\ X_3 + X_4 = Y_2 \end{array}} \right\} \rightarrow Y_3$$

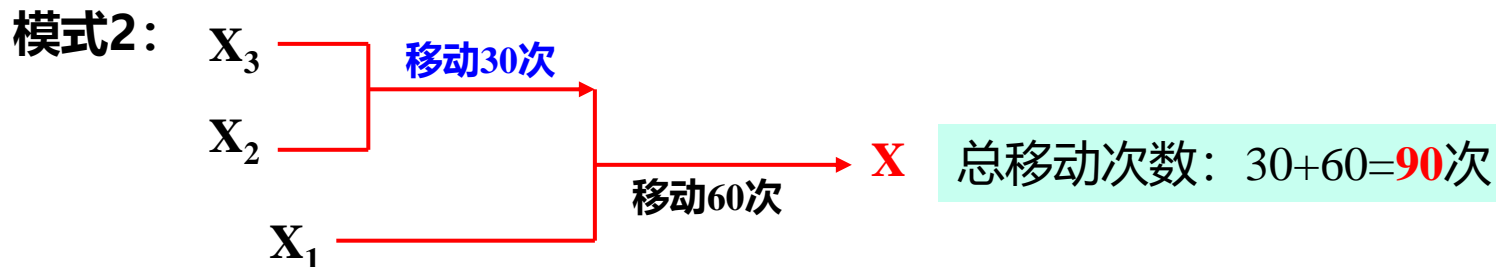
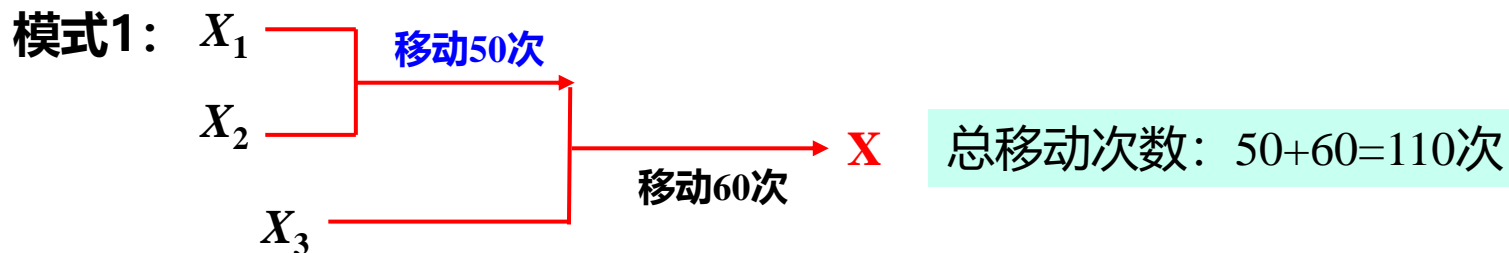
这里  $Y_1$ 、 $Y_2$  是归并过程中产生的中间文件， $Y_3$  是最终结果文件。



◆ 不同的归并模式所需的**计算代价**可能是不同的。

例，设  $X_1$ 、 $X_2$ 、 $X_3$  是**记录长度**分别为 30、20、10 的已分类文件。将这 3 个文件归并成长度为 60 的文件。

可能的归并过程和相应的**记录移动次数（代价）**如下：



### 3、二路归并模式

每次仅归并两个文件的归并模式称为**二路归并模式**。

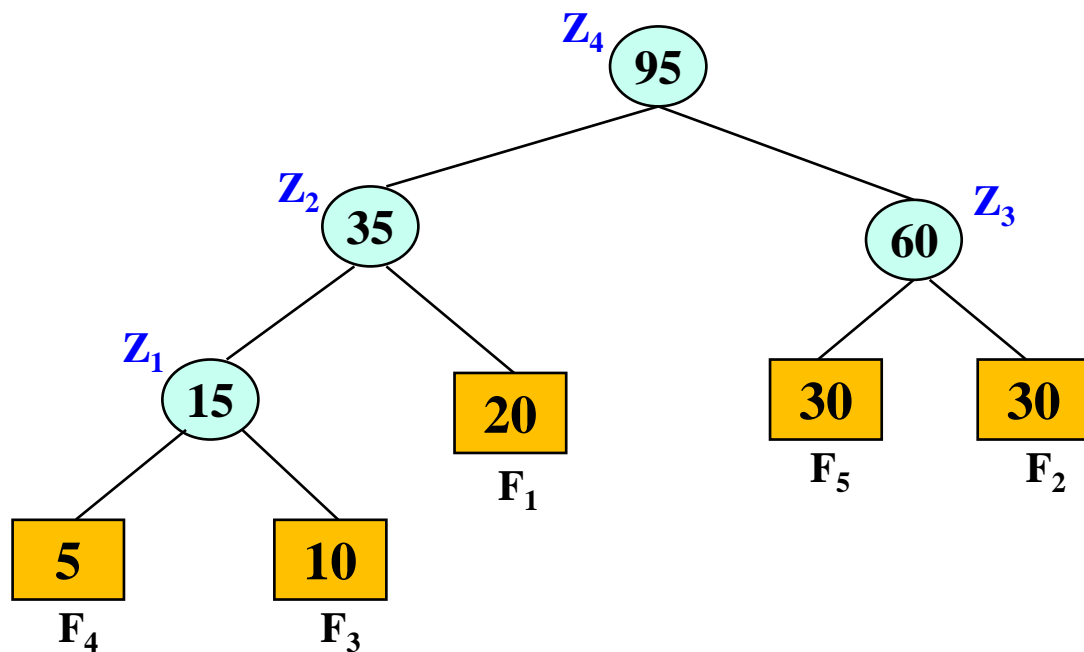
该模式下，当有多个文件时，采用**两两归并**的完成。

#### (1) 二元归并树

二路归并模式的**归并过程**可以用一棵**二元树**描述，称为**二元归并树**：

- ◆ **外部结点（叶子结点）**：代表  $n$  个**原始文件**。
- ◆ **内结点**：代表归并过程中产生的**中间文件**，根代表**最终文件**。
- ◆ 在两路归并模式下，**每个内结点刚好有两个儿子**，表示把它的两个儿子代表的文件归并成其本身所代表的文件。

如, 已知五个文件,  $(F_1, F_2, F_3, F_4, F_5) = (20, 30, 10, 5, 30)$ ,  $F_i$  表示  $i$  文件的长度。一棵**二元归并树**如下所示:

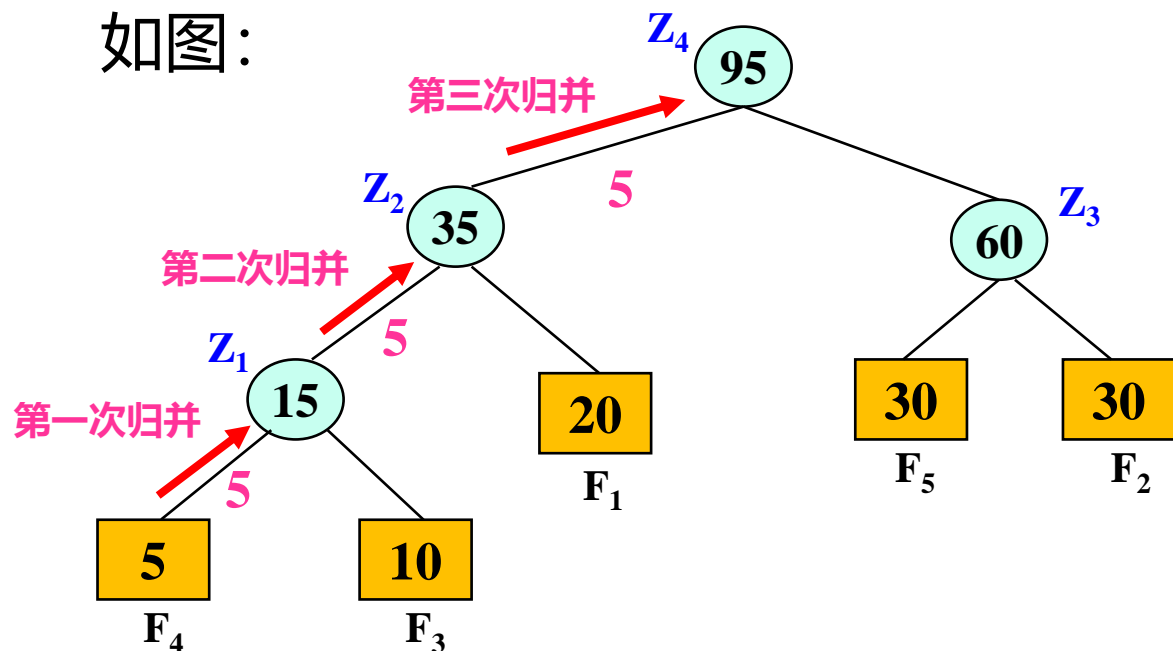


- ◆ **外部结点**代表  $n$  个**原始文件**, 用**矩形**表示, 其中的数字是原始文件的长度。
- ◆ **内结点**代表归并过程中产生的**中间文件**, 用**圆形**表示。根代表**最终的文件**。其中的数字代表将其两个儿子代表的文件合并成它自己后的文件长度。

## (2) 最优二路归并模式

归并树反映出：为得到根所表示的归并文件，每个外部结点所代表的文件中的**每个记录需要移动的次数等于该外部结点到根的距离**（根到该外部结点的路径的长度，即路径上边的数量）。

如图：



$F_4$  中的所有记录在整个归并过程中均移动 **3 次**。

记  $|F_i|$  为  $i$  文件的长度，则  $F_4$  中的记录在整个归并过程中总共发生的**记录移动次数**是  $|F_4| * 3 = 15$  次。

## ◆ 带权外部路径长度：

记  $d_i$  是由根到代表文件  $F_i$  的外部结点的距离,  $q_i$  是  $F_i$  的长度, 则一棵归并树所代表的归并过程中**元素的移动总量**是：

$$\sum_{1 \leq i \leq n} q_i d_i$$

称为这棵树的**带权外部路径长度**。

- ◆ **最优二路归并模式**：一棵具有**最小带权外部路径长度**的二元归并树所对应的二元归并模式称为**最优二路归并模式**。

对已知的  $n$  个文件, 怎么求取其相应的最优二路归并模式呢?

## 4、最优二路归并模式问题的贪心算法设计

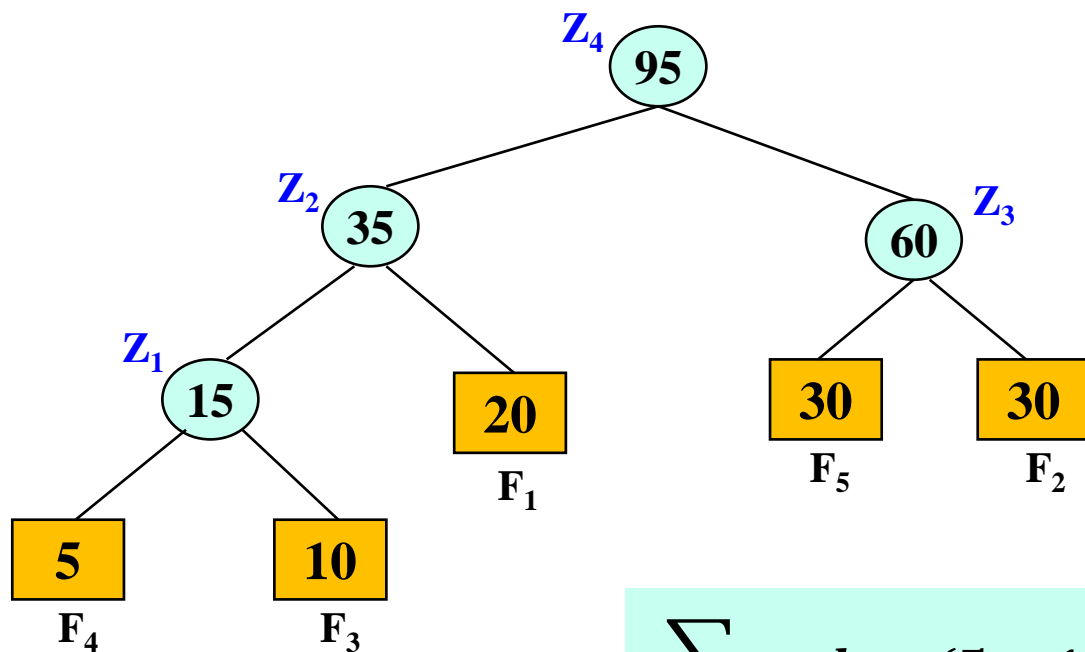
### (1) 贪心选择策略

这里以**目标函数**  $(\sum_{1 \leq i \leq n} q_i d_i)$  做为度量。

① **为使目标函数达到最小**，可能的思路是：**每次归并应使目标函数有最小的增加**。

② 由于任意两个文件的归并所需的元素移动次数与这两个文件的长度之和成正比，所以具体处理规则可以设计为：**每次选择长度最小的两个文件进行归并**。

如图中所展示的归并过程： $(F_1, F_2, F_3, F_4, F_5) = (20, 30, 10, 5, 30)$



$$\sum_{1 \leq i \leq n} q_i d_i = (5 + 10) \times 3 + (20 + 30 + 30) \times 2 = 205$$

注：每次归并后，参加归并的两个文件将从文件集合中删除，并将新得到的文件加入文件集合，该文件的长度等于原两个文件的长度之和。下一次归并将基于新的文件集合进行。

## (2) 最优二路归并模式算法

**MergeTree** ( $L, n$ ) // 开始的时候  $L$  是  $n$  个单结点的二元树表

**for**  $i = 1$  **to**  $n - 1$

$T = \text{GetTreeNode}()$  // 构造一棵新树 (根)  $T$

$T.\text{left} = \text{LeastExtract}(L)$  // 从  $L$  中选  $wight$  最小的树作为新树的左子树, 然后从  $L$  中删除

$T.\text{right} = \text{LeastExtract}(L)$  // 从  $L$  中选下一  $wight$  最小的树作为新树的右子树, 并从  $L$  中删除

$T.\text{weight} = T.\text{left}.\text{weight} + T.\text{right}.\text{weight}$  // 新树的  $\text{weight}$  等于其左右儿子的  $wight$  之和。

**INSERT**( $L, T$ ) // 将新树  $T$  加入到表  $L$  中

**return** ( $\text{LeastExtract}(L)$ ) // 最后,  $L$  中只有一棵树, 就是最后的归并结果, 返回

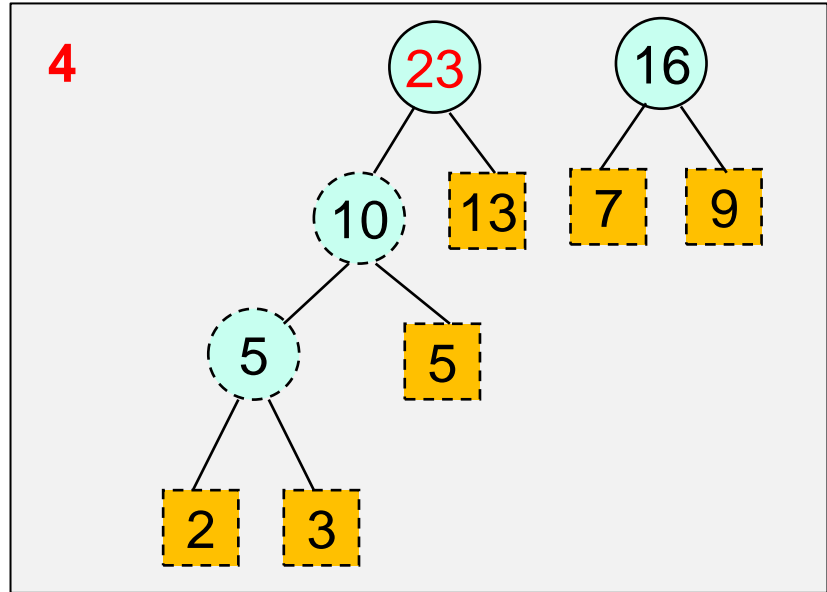
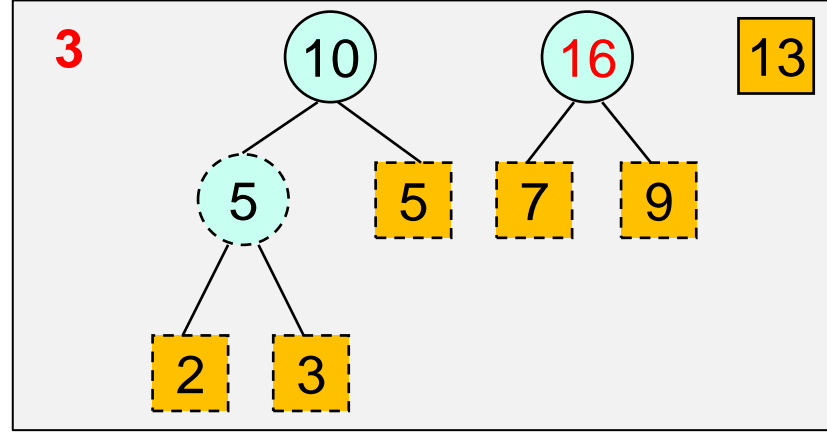
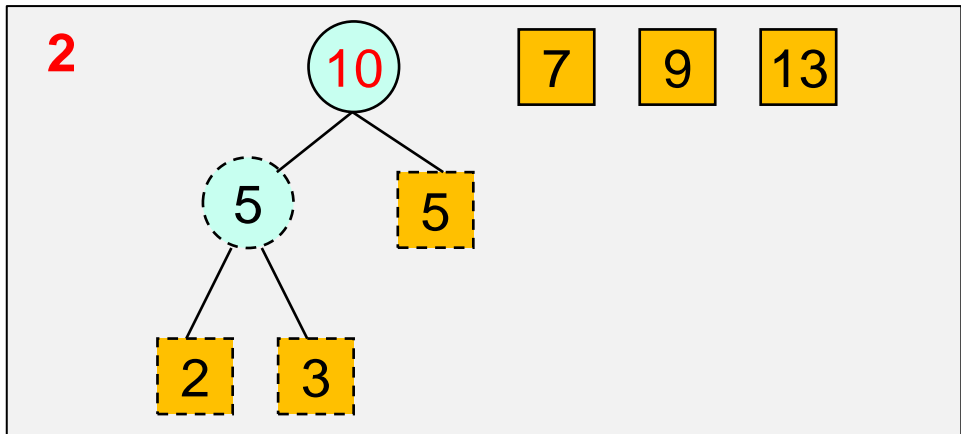
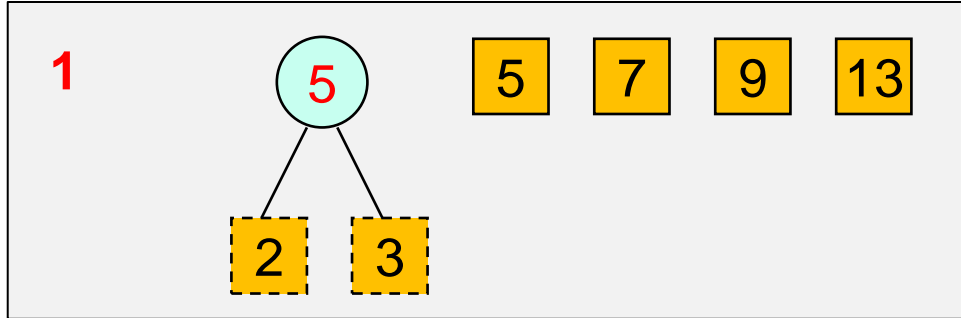
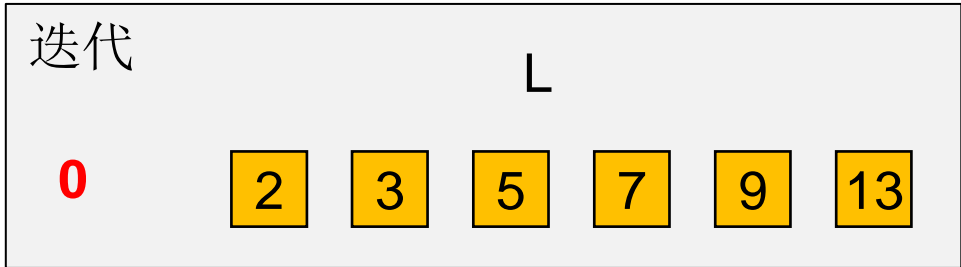
**end MergeTree**

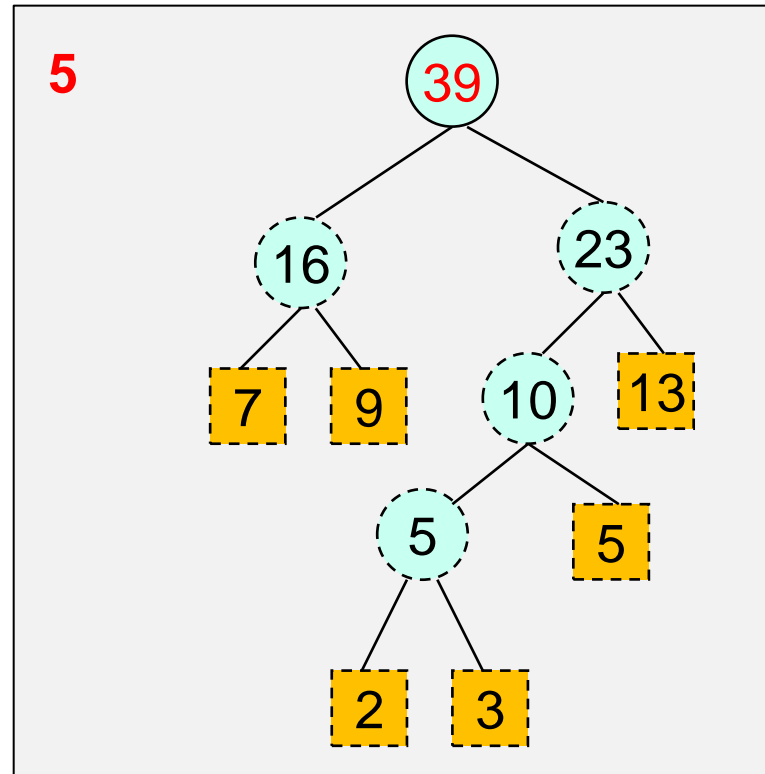
和Huffman树算法一致



**例** 已知六个初始文件，长度分别为：2, 3, 5, 7, 9, 13。采用算法

MergeTree，各阶段的工作状态如图所示示：





$$\sum_{1 \leq i \leq n} q_i d_i = 7 \times 2 + 9 \times 2 + 13 \times 2 + 5 \times 3 + 2 \times 4 + 3 \times 4$$

$$= 93$$

# 算法的时间分析

(1) 循环体:  $n - 1$  次

(2) 若  $L$  以有序表表示

LeastExtract( $L$ ):  $O(1)$

INSERT( $L, T$ ):  $O(n)$

总时间:  $O(n^2)$

(3) 若  $L$  以优先队列表示

LeastExtract( $L$ ):  $O(\log n)$

INSERT( $L, T$ ):  $O(\log n)$

总时间:  $O(n \log n)$

```
MergeTree ( $L, n$ )  
  for  $i = 1$  to  $n - 1$   
     $T = \text{GetTreeNode} ()$   
     $T.\text{left} = \text{LeastExtract}(L)$   
     $T.\text{right} = \text{LeastExtract}(L)$   
     $T.\text{weight} = T.\text{left}.\text{weight} + T.\text{right}.\text{weight}$   
    INSERT( $L, T$ )  
  return (LeastExtract( $L$ )  
end MergeTree
```

## 5、MergeTree 算法最优解的证明

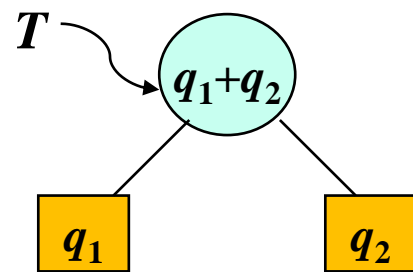
**定理16.5** 若  $L$  最初包含  $n \geq 1$  个单结点的树，这些树有 WEIGHT 值为  $(q_1, q_2, \dots, q_n)$ ，则算法 MergeTree 对于具有这些 WEIGHT 的  $n$  个文件生成一棵最优的二元归并树。

**证明：** 归纳法证明

① 当  $n = 1$  时，返回一棵没有内部结点的树。定理得证。

② 假设算法对所有  $(q_1, q_2, \dots, q_m)$ ， $1 \leq m < n$ ，都能生成一棵最优二元归并树。则对于  $n$  有：

③ 不失一般性，假定  $q_1 \leq q_2 \leq \dots \leq q_n$ ，则  $q_1$  和  $q_2$  将是在 *for* 循环的第一次迭代中选出的两个最小WEIGHT值，如图所示，设  $T$  是由  $q_1$  和  $q_2$  合并后构成的子树：



- ◆ 设  $T'$  是一棵关于  $(q_1, q_2, \dots, q_n)$  的最优二元归并树。
- ◆ 设  $P$  是  $T'$  中距离根最远的一个内部结点。

若  $P$  的两棵子树不是  $q_1$  和  $q_2$ , 则用  $q_1$  和  $q_2$  替换  $P$  当前的子树而不会增加  $T'$  的带权外部路径长度。

故,  $T$  可以是一棵最优归并树中的子树。 (引理16.2)

不失一般性, 设  $T'$  就是一棵包含  $T$  的最优归并树。

则在  $T'$  中用一个权值为  $q_1 + q_2$  的外部结点替换  $T$ , 得到的将是一棵关于  $(\dots, q_1 + q_2, \dots, q_n)$  的归并树  $T''$ 。

而由归纳假设, MergeTree 可以对  $(\dots, q_1 + q_2, \dots, q_n)$  生成一棵最优归并树, 记为  $T'''$ 。



可以证明  $T''$  的带权外部路径长度不会大于  $T'''$ ，所以  $T''$  也是关于  $(\dots q_1 + q_2, \dots, q_n)$  的一棵最优归并树。 (引理16.3)

最后，再将  $T$  带入该树，再作为权值为  $q_1 + q_2$  的那个结点的儿子结点。由于  $q_1$ 、 $q_2$  处增加了一层，所以此时的带权外部路径长度等于  $T''$  的带权外部路径长度  $+ q_1 + q_2$ 。而这个值是增加  $q_1$ 、 $q_2$  后带权外部路径长度的最小值，所以是关于  $(q_1, q_2, \dots, q_n)$  的最优归并树。 (引理16.3)

故，MergeTree 生成一棵关于  $(q_1, q_2, \dots, q_n)$  的最优归并树。

以上定理证明的详细过程请参考引理16.2、引理16.3、定理16.4。

## 6. $k$ 路归并模式

- ◆  **$k$  路归并模式**：每次归并 $k$ 个文件的归并模式。
- ◆  **$k$  元归并树**：可以用一棵  $k$  叉树描述  $k$  路归并过程，称为  $k$  元归并树。在  $k$  元归并树中，可能需要增加 “虚” 结点以补充不足的外部结点。

可以证明：

★ 如果一棵树的所有内部结点的度都为  $k$ ，则外部结点数  $n$  满足

$$n \bmod (k - 1) = 1$$

★ 对于满足  $n \bmod (k - 1) = 1$  的整数  $n$ ，存在一棵具有  $n$  个外部结点的  $k$  元树  $T$ ，且  $T$  中所有结点的度为  $k$ 。故，至多需要增加  $k-2$  个外部结点。

- ◆  **$k$  路最优归并模式的贪心规则**：每一步选取  $k$  个具有最小长度的文件进行归并。

## 本章作业

- ◆ 16.1-4、16.2-7、16.3-3、16-1
- ◆ 求以下背包问题的最优解：

$$n = 7, \quad M = 15,$$

$$(p_1, \dots, p_7) = (10, 5, 15, 7, 6, 18, 3),$$

$$(w_1, \dots, w_7) = (2, 3, 5, 7, 1, 4, 1)。$$