算法设计与分析

Computer Algorithm Design & Analysis

2019.11

王多强

dqwang@mail.hust.edu.cn

群名称：2019-算法

群　号：835135560

Chapter 16
Greedy Algorithms

贪心算法

对于某些类型的最优化问题，使用动态规划算法略显复杂。

有没有更简单一点的方法?

- 贪心算法是一种选择。

什么是贪心算法?

贪心算法是这样一种方法：分步骤实施，它在每一步仅作出当时看起来最佳的选择，即局部最优的选择，并希望通过这样的选择最终能找到全局最优解。

如：食堂打菜：东一窗口之间不排队，西二二楼需要排队

- 经典问题：最小生成树问题的Prim算法、Kruskal算法，单源最短路径Dijkstra算法等，以及一些近似算法。

16.1 活动选择问题

1）问题描述

假定有一个活动的集合S含有n个活动$\{a_1, a_2, \cdots, a_n\}$，每个活动$a_i$都有一个开始时间$s_i$和结束时间$f_i$，$0 \leqslant s_i < f_i < \infty$。同时，这些活动都要使用同一资源(如演讲会场)，而这个资源在任何时刻只能供一个活动使用。

活动的兼容性：如果选择了活动a_i，则它在半开时间区间$[s_i, f_i)$内占用资源。若两个活动a_i和a_j满足$[s_i, f_i)$与$[s_j, f_j)$不重叠，则称它们是兼容的。

- 即，当$s_i \geqslant f_j$或$s_j \geqslant f_i$时，活动a_i与活动a_j兼容。

活动选择问题：就是对给定的包含n个活动的集合S，在已知每个活动开始时间和结束时间的条件下，从中选出最多可兼容活动的子集合，称为**最大兼容活动集合**。

不失一般性，设活动已经按照**结束时间单调递增**排序：

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n .$$

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，

并设**活动按结束时间的非减次序排列**：

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

按结束时间的非减序排列

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

其中 $\{a_1, a_4, a_8, a_{11}\}$、$\{a_2, a_4, a_9, a_{11}\}$ 是最大兼容活动集合。显然最大兼容活动集合不一定是唯一的。

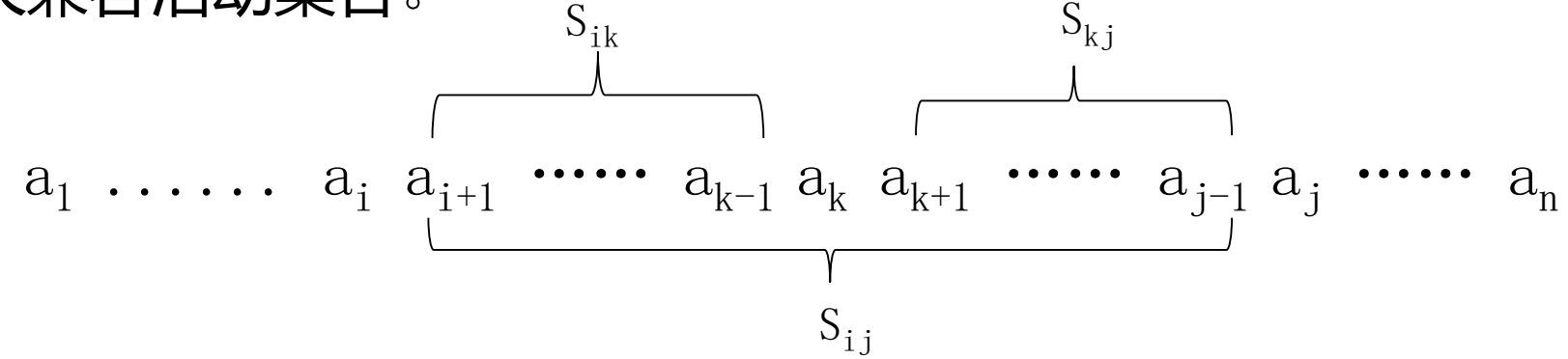
（1）活动选择问题的最优子结构

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。

问题和子问题的形式定义如下：

设A_{ij}是S_{ij}的一个最大兼容活动集，并设A_{ij}包含活动a_k，则有：A_{ik}表示A_{ij}中a_k开始之前的活动子集，A_{kj}表示A_{ij}中a_k结束之后的活动子集。

并得到两个子问题：**寻找S_{ik}的最大兼容活动集合**和**寻找S_{kj}的最大兼容活动集合**。



活动选择问题具有最优子结构性，即：

必有：A_{ik}是S_{ik}一个最大兼容活动子集，A_{kj}是S_{kj}一个最大兼容活动子集。而$A_{ij}= A_{ik}\cup\{a_k\}\cup A_{kj}$。—— 最优子结构性成立。

证明：

- 用剪切-粘贴法证明最优解A_{ij}必然包含两个子问题S_{ik}和S_{kj}的最优解。

设S_{kj}存在一个最大兼容活动集A_{kj}'，满足$|A_{kj}'|>|A_{kj}|$，则可以将A_{kj}'作为S_{ij}最优解的一部分。

这样就构造出一个兼容活动集合，其大小

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

与A_{ij}是最优解相矛盾。 得证。

活动选择问题的动态规划方法

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，可得递归式如下：

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

为了选择k，有：

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

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（自学）。

活动选择问题的贪心算法

贪心选择：在贪心算法的每一步所做的当前最优选择（局部最优选择）就叫做贪心选择。

活动选择问题的贪心选择：每次总选择具有最早结束时间的兼容活动加入到集合A中。

为什么？

直观上，按这种方法选择兼容活动可以为未安排的活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是使剩余的可安排时间段最大化，以便安排尽可能多的兼容活动。

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

结束时间递增

由于输入的活动已经按照结束时间的递增顺序排列好了，所以，首次选择的活动是a_1；

其后选择的是结束时间最早且开始时间不早于前面已选择的最后一个活动的结束时间的活动 （活动要兼容）。

- f_1是最早结束时间，不会有活动的结束时间早于a_1，因此所有与a_1兼容的活动都是在a_1结束之后开始。
- 令$S_k=\{a_i \in S: s_i \geq f_k\}$，即在$a_k$结束之后开始的任务集合。

  则在首次选择a_1后，S_1是接下来要求解的（唯一）子问题。
- **由最优子结构性得：**如果a_1在最优解中（确实在），那么**原问题的最优解由活动a_1及子问题S_1的最优子解构成。**
- 对S_1可以继续按照相同的方式求解。

算法正确吗？ 即按照上述的贪心选择方法选择的活动集是问题的最优解吗？

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。若$a_j=a_m$，则得证。否则，令 $A_k' = A_k-\{a_j\}\cup\{a_m\}$。

因为A_k中的活动都是不相交的，a_j是A_k中结束时间最早的活动，而a_m是S_k中结束时间最早的活动，所以$f_m \le f_j$。即A_k'中的活动也是不相交的。

由于$|A_k'|=|A_k|$，所以A_k' 也就是S_k的一个最大兼容活动子集，且包含a_m。 得证。

- 定理16.1告诉我们，选a_m不会错！
  - 从S_0开始，**反复选择结束时间最早的活动**，重复这一过程，直至不再有新的兼容活动。所得的子集就是最大兼容活动集合。

由于结束时间严格递增，故只需按照结束时间的单调递增顺序处理所有活动，每个活动考查且仅考查一次。

注：当输入的活动已按结束时间的递增顺序排列，贪心算法只需O(n)的时间即可选择出来n个活动的最大兼容活动集合。

如果所给出的活动未按结束时间的非减序排列，可以用O(nlogn)的时间重排。

活动选择问题的贪心算法

采用自顶向下的设计：**首先做出一个选择，然后求解剩下的子问题**。每次选择将问题转化成一个规模更小的问题。

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

这里，数组s、f分别表示n个活动的开始时间和结束时间。并假定n个活动已经按照结束时间单调递增排列好。对当前的k，算法返回S_k的一个最大兼容活动集。

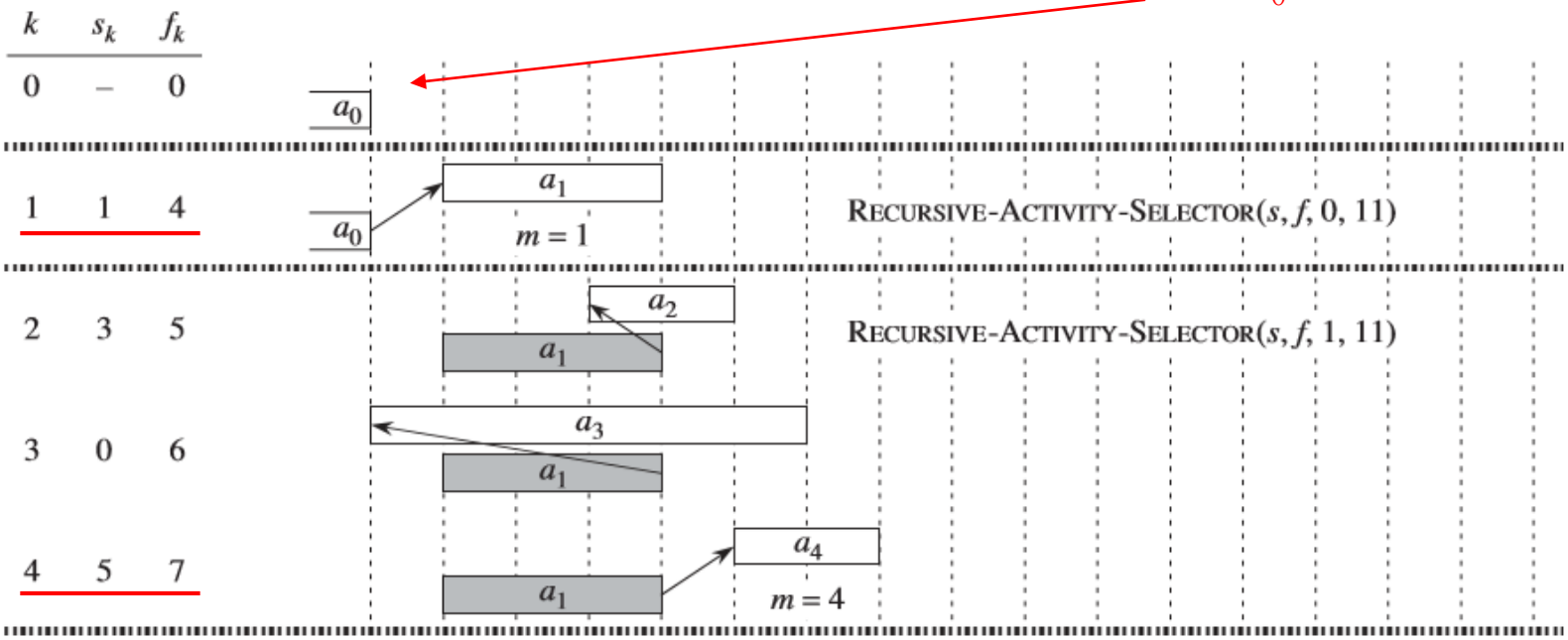
初次调用：RECURSIVE-ACTIVITY-SELECTOR(s,f,0,n)。

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

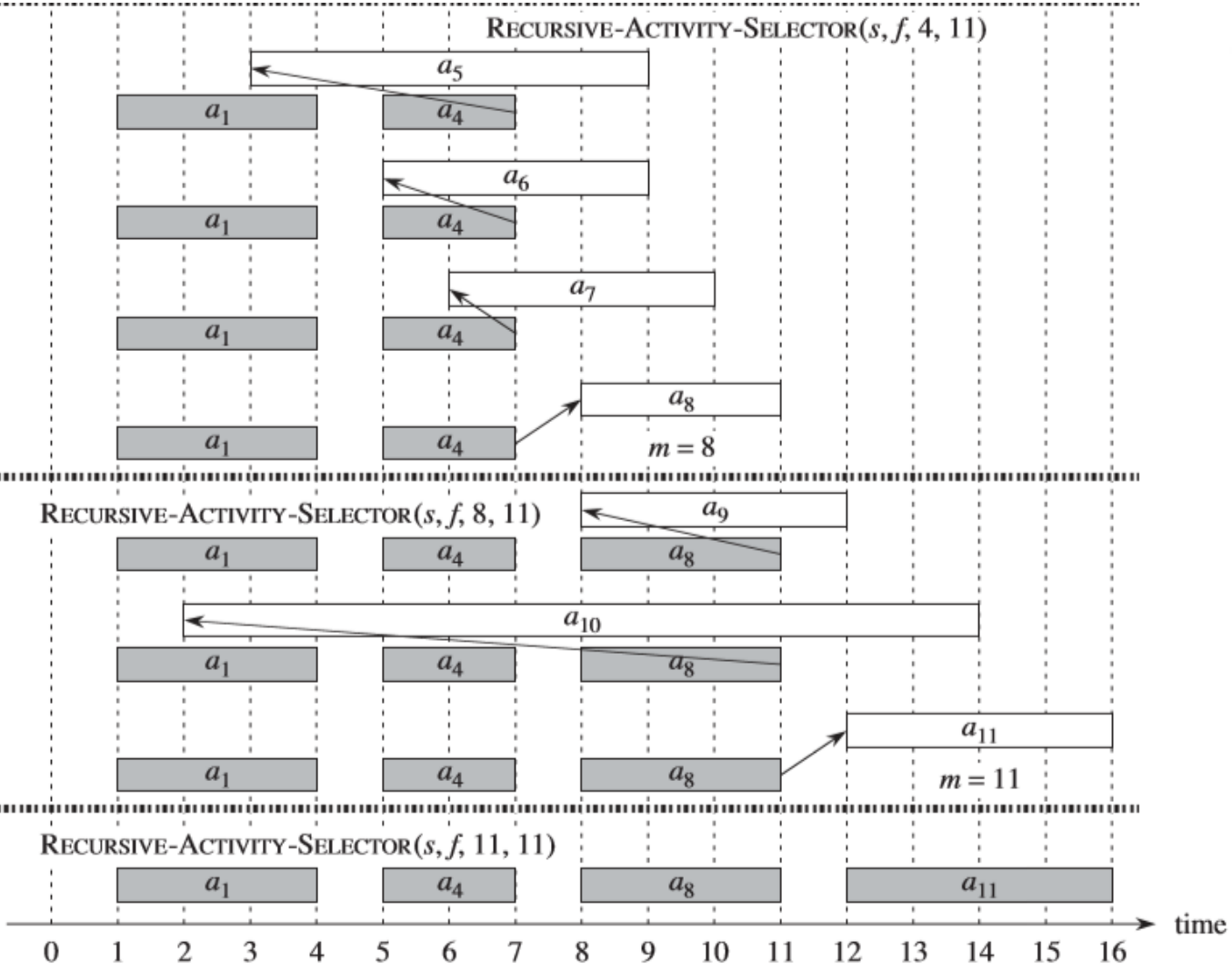
执行过程如图所示：

注：为了处理的方便，这里引入一个虚拟活动a_0，其结束时间$f_0=0$



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

k	s_k	f_k
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16



用迭代实现的贪心算法

注：上述的RECURSIVE-ACTIVITY-SELECTOR是一个“尾递归”过程，可以很容易地转换成迭代形式。

假定活动已经按照结束时间单调递增的顺序排列好

```
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] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

集合A用于收集选出的活动

k对应最后一个加入A的活动，f_k是A中活动的最大结束时间，若m的开始时间大于f_k，则m就是下一个被选中的活动。

算法的运行时间是O(n)。

16.2 贪心算法原理

- 贪心算法通过做出一系列选择来求问题的最优解 —— 即贪心选择：在每个决策点，它做出在当时看来是最佳的选择。
  - 这种启发式策略并不保证总能找到最优解，但对有些问题确实有效，相比动态规划算法，贪心算法简单和直接得多。
- 贪心算法通常采用自顶向下的设计，做出一个选择，然后求解剩下的子问题。**每次选择将问题转化成一个更小规模的问题**。

贪心求解的一般步骤：

1）确定问题的最优子结构；

2）将最优化问题转化为这样的形式：每次对其作出选择后，只剩下一个子问题需要求解；

3）证明作出贪心选择后，剩余的子问题满足：**其最优子解与前面的贪心选择组合即可得到原问题的最优解**（具有最优子结构）。

注：对应每个贪心算法，都有一个动态规划算法，但动态规划算法要繁琐的多。

如何用贪心算法求解?

- 贪心选择性质和最优子结构性是两个关键要素。
  - 如果能够证明问题具有这两个性质，则基本上就可以实施贪心策略。

1）贪心选择性质

贪心选择性质：可以通过做出局部最优选择（即贪心选择）来构造全局最优解的性质。

贪心选择性使得我们进行选择时，只需做出当前看起来最优的选择，而不用考虑子问题的解。

对比动态规划方法：

- 在动态规划方法中，每个步骤也都要进行一次选择，但这种选择通常**依赖于子问题的解**，这导致我们要先求解较小的子问题，然后才能计算较大的子问题。
- 在贪心方法中，我们总是做出当前看来最佳的选择，然后求解剩下的唯一一个子问题。尽管贪心算法进行选择时可能依赖之前做出的选择，但不依赖任何将来的选择或子问题的解。
- 动态规划要先求解子问题才能进行选择，贪心算法在进行第一次选择之前不需要求解任何子问题。
- 动态规划算法通常采用自底向上的方式完成计算，而贪心算法通常是自顶向下的，每一次选择，将给定的问题转换成一个更小的问题，然后继续求解小问题

- 如何证明每次贪心选择能生成全局最优解?

思路：类似动态规划的剪切-粘贴法。

通常先考查某个子问题的最优解，然后**用贪心选择替换某个其它选择**来修改此解，从而得到一个相似但更小的子问题，从而导出新解或矛盾。

参考定理16.1的证明。

2）最优子结构性

最优子结构性质是能否应用动态规划和贪心方法的关键要素。

- 贪心算法更为直接地使用最优子结构：

每次贪心选择后都得到一个子问题，而原问题的最优解就是贪心选择和子问题的最优解的组合。**能否成功，最优子结构性质是根本保证。**

分数背包问题

1.问题的描述

已知n种物品，各具有重量$(w_1, w_2, \cdots, w_n)$和效益值$(p_1, p_2, \cdots, p_n)$，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？

这里：

1）所有的$w_i>0$，$p_i>0$，$1\leqslant i\leqslant n$；

2）问题的解用向量$(x_1, x_2, \ldots, x_n)$表示，每个x_i表示物品i被放入背包的比例，$0\leqslant x_i\leqslant 1$。当物品i的一部分$x_i$放入背包，可得到$p_i x_i$的效益，同时会占用$x_i w_i$的重量。

问题分析：

① 装入背包的总重量不能超过M，即 $\sum_{1\le i\le n} w_i x_i \le M$。

② 如果当前问题实例的所有物品的总重量不超过M，即：

$\sum_{1\le i\le n} w_i \le M$，则只有把所有的物品都装入背包中才可获得**该实例**最大的效益值，此时所有的$x_i=1$，$1\leqslant i\leqslant n$。

③ 如果物品的总重量 $\sum_{1\le i\le n} w_i \ge M$，则将有物品可能无法全部装入背包。而此时，由于$0\leqslant x_i\leqslant 1$，所以可以把物品的全部或部分装入背包，最终背包中刚好装入重量为M的若干物品（可能是其一部分）。这种情况下，如果背包没有被装满，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：$\sum_{1\le i\le n} w_i x_i \le M$

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

目标函数：$\sum_{1\le i\le n} p_i x_i$

可 行 解：满足上述约束条件的任一$(x_1, x_2, \cdots, x_n)$ 都是问题的一个可行解。可行解可能有多个（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。最优解也可能有多个。

例5.1 设有三件物品和一个背包，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	

2. 贪心策略求解

讨论两个问题：

1）如何做贪心选择？

2）获得问题的贪心解后，如何证明贪心解是问题的最优解？

1）贪心选择的策略

这里讨论三种不同的**度量**。

策略1：以目标函数作为度量

思路：每装入一件物品，就使背包获得最大可能的效益增量。

规则：以目标函数作为度量，按效益值的非增次序将物品一件件地放入到背包；

注：如果正在考虑的物品放不进去，则只取其一部分装满背包。

此时，如果该物品的一部分不满足获得最大效益增量的度量，则在剩下的物品中选择可以获得最大效益增量的其它物品，将它或其一部分装入背包。

- 如：若背包剩余容量ΔM=2,而此时背包外还剩两件物品i,j，且有(p_i= 4，w_i=4) 和(p_j= 3，w_j=2)，则下一步应选择j而非i放入背包，因为

$$p_i/2 = 2 < p_j = 3$$

即虽然$p_i > p_j$，但物品j可以全部放入并带来3的效益值，而物品i只能放1/2，带来的2效益值。

策略1实例分析（例5.1，M=20，$(p_1, p_2, p_3) = (25, 24, 15)$，$(w_1, w_2, w_3) = (18, 15, 10)$）

∵ $p_1 > p_2 > p_3$

∴ 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，同时背包容量消耗掉$w_1 = 18$个单位，剩余空间ΔM=2。不足以放入物品2和3的全部。

然后考虑就ΔM=2而言，只能选择物品2或3的一部分装入背包。

- 物品2： 若 $x_2 = 2/15$， 则 $p_2 x_2 = 16/5 = 3.1$
- 物品3： 若 $x_3 = 2/10$， 则 $p_3 x_3 = 3$

故，为**使背包的效益有最大的增量**，应选择物品2的2/15装包，即$x_2 = 2/15$ 。

最后，ΔM=0，背包装满，物品3不能装，$x_3 = 0$ ，结束。

得到的解：$(x_1, x_2, x_3) = (1, 2/15, 0)$

$\sum p_i x_i = 28.2$ ，仅为次优解，非最优解。

分析：为什么以目标函数作为度量没能获得最优解？

尽管背包的效益值每次得到了最大的增加，但背包容量也过快地被消耗掉了，从而不能装入“更多”的物品。

策略2：以容量作为度量

思路：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。

规则：以容量作为度量，按物品重量的非降次序将物品装入到背包；如果正在考虑的物品放不进去，则只取其一部分装满背包即可；

策略2实例分析（例5.1，M=20，$(p_1, p_2, p_3) = (25, 24, 15)$，$(w_1, w_2, w_3) = (18, 15, 10)$）

∵ $w_3 < w_2 < w_1$

∴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，容量消耗$w_3 = 10$个单位，剩余容量ΔM=10。

然后考虑物品2。就ΔM=10而言有，只能选择物品2的10/15装入背包，即：

$$x_2 = 10/15 = 2/3$$

最后，ΔM=0，背包装满，物品1不能装入背包，$x_1 = 0$ 。

得到的解：$(x_1, x_2, x_3) = (0, 2/3, 1)$

$\sum p_i x_i = 31$，依旧是次优解，非最优解。

分析，为什么以容量作为度量又没能获得最优解？

尽管背包的容量每次消耗得最少，装入物品的“个数”多了，但效益值没能“最大程度”的增加。

策略3：最优度量的选择

思路：片面地考虑背包的效益增量和容量消耗都是不行的，应在**背包效益值的增长速率**和**背包容量的消耗速率**之间取得平衡。

进一步的考虑是，让背包发挥“最大的作用”，亦即，让其每一单位被占用的容量都尽可能地装进最大可能效益的物品。

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，使得每次装入后累计效益值与所用容量的比值有最多的增加（首次装入）和最小的减小（其后的装入）。

- 按物品的单位效益值（即p_i/w_i 值）的非降次序将物品装入到背包；
- 如果正在考虑的物品放不进去，则只取其一部分装满背包即可；

策略3实例分析（例5.1，M=20，$(p_1, p_2, p_3) = (25, 24, 15)$，$(w_1, w_2, w_3) = (18, 15, 10)$）

∵ $p_2/w_2 > p_3/w_3 > p_1/w_1$

∴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，容量消耗$w_2 = 15$个单位，剩余容量ΔM=5。

然后考虑物品3，就ΔM=5而言，只能放入物品3的5/10，即，

$$x_3 = 5/10 = 1/2$$

最后，ΔM=0，背包装满，物品1不能装入背包，$x_1 = 0$ 。

得到的解：$(x_1, x_2, x_3) = (0, 1, 1/2)$

$\sum p_i x_i = 31.5$，是问题的最优解吗？

3. 背包问题的贪心求解算法

```
procedure GREEDY－KNAPSACK(P,W,M,X,n)
  //P(1:n)和W(1:n)分别含有按P(i)/W(i)≥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，而X(1:n)是解向量//
    real P(1:n), W(1:n), X(1:n), M, cu;
    integer i,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 do
          if W(i) > cu then exit endif
          X(i) ←1
          cu ←cu-W(i)
    repeat
    if i≤n then X(i) ←cu/W(i)  endif
end GREEDY-KNAPSACK
```

4. 最优解的证明

策略3所得到的贪心解是问题的最优解吗？

最优解的含义：在满足约束条件的情况下，使目标函数取极大（小）值的可行解。

我们需要证明什么？

贪心解是可行解，故只需证明：贪心解可使目标函数取得极值。

这里需要注意的是：解的形式并不重要，重要的是贪心解和任意最优解一样，可使目标函数取得极值。

证明的基本思路：

1）设出问题最优解的一般形式。

2）将贪心解与“理想”最优解进行比较。

（2.1）如果这两个解相同，则显然贪心解就是最优解。

（2.2）如果这两个解不同，则肯定有不同的地方，即至少在一个分量x_i上存在不同。

（2.2.1）设法找出第一个不同的分量位置i；

（2.2.2）设法用贪心解的x_i替换最优解对应的分量

——消去不同的地方；

（2.2.3）作某些调整，以使得替换后的新解还是可行解，不违反任何约束条件。

（2.2.4）证明经过上述处理得到的新解在效益上与原最优解相同，至少不会降低，即依然“最优”。

（2.3）继续比较“新最优解”和贪心解，若还存在不同，则重复（2.2），反复进行代换，直到代换后产生的“最优解”与贪心解完全一样。

3）结论：在上述代换中，最优解的效益值没有受到任何损失，那么贪心解的效益值与代换前、后的最优解的效益值都是相同的。所以贪心解如同任意一个最优解一样，可使得目标函数取得极值。

从而得证：该贪心解即是问题的最优解。

定理16.1 如果$p_1/w_1 \geqslant p_2/w_2 \geqslant \cdots \geqslant p_n/w_n$，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$X=(x_1, x_2, \cdots, x_n)$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就是问题的最优解。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，

- $x_i=1 \quad 1 \leqslant i < j$,
- $0 \leqslant x_j < 1$
- $x_i=0 \quad j < i \leqslant n$

假设Y是问题的最优解：$Y=(y_1, y_2, \cdots, y_n)$，不失一般性，应有：$\sum w_i y_i = M \quad 0 \le y_i \le 1$

- 若X＝Y，则X就是最优解。否则，（2.1）
- X和Y至少在1个分量上存在不同。（2.2）

设k是使得$y_k \neq x_k$的最小下标，则有$y_k < x_k$。（2.2.1）

（为什么$y_k < x_k$？）

$y_k < x_k$：可分以下情况说明：

a）若k<j，则x_k=1。因为$y_k \neq x_k$，故只能有$y_k < x_k$。

b）若k=j，由于$\sum w_i x_i = M$，且对1≤i<j，有y_i=x_i=1，而对j<i≤n，有x_i=0；故此时若$y_k > x_k$，则将有$\sum w_i y_i > M$，与Y是可行解相矛盾。而$y_k \neq x_k$，所以也只能有$y_k < x_k$。

c）若k>j，则$\sum w_i y_i > M$，不能成立。

- 在Y中作以下调整：将y_k增加到x_k，　　（2.2.2）
- 因为$y_k < x_k$，为保持解的可行性，必须从$(y_{k+1}, \cdots, y_n)$中减去同样多的量 $\sum_{k<i\le n} w_i(y_i - z_i) = w_k(z_k - y_k)$。　　（2.2.3）

设调整后的解为$Z=(z_1, z_2, \cdots, z_n)$，其中$z_i = x_i$，$1 \leqslant i \leqslant k$，且Z的效益值有：

$$\sum_{1\le i\le n} p_i z_i = \sum_{1\le i\le n} p_i y_i + (z_k - y_k) w_k p_k / w_k - \sum_{k<i\le n} (y_i - z_i) w_i p_i / w_i$$

$$\geq \sum_{1\le i\le n} p_i y_i + (z_k - y_k) w_k p_k / w_k - \sum_{k<i\le n} (y_i - z_i) w_i p_k / w_k$$

$$= \sum_{1\le i\le n} p_i y_i + \left[(z_k - y_k) w_k - \sum_{k<i\le n} (y_i - z_i) w_i \right] p_k / w_k$$

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

差值＝0

$p_k/w_k > p_i/w_i$
$k<i\leqslant n$

由以上分析得，（2.3）

- 若 $\sum p_i z_i > \sum p_i y_i$ ，则Y将不是最优解；
- 若 $\sum p_i z_i = \sum p_i y_i$ ，则或者Z=X，则X就是最优解；
- 或者Z≠X，则重复以上替代过程，或者证明Y不是最优解，或者把Y转换成X，从而证明X是最优解。

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

计算$\sum p_i x_i$

参考答案：

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

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

$x_2=1$　　$x_1=1$　　$x_4=1$　　$x_3=2/15$　　$\sum w_i x_i=54$

$X=(1, 1, 2/15, 1)$　　$\sum p_i x_i=166/3$

16.3 Huffman编码

Huffman编码：最佳编码方案，通常可以节省20%~90%的空间。

Huffman编码问题是一个典型的贪心算法问题。

实例说明：

设要压缩一个有10万个字符的数据文件，文件中出现的所有字符和它们的出现频率如下：

	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

只有六个字符

分析：

	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

采用二进制字符编码（简称编码），每个字符用唯一的二进制串表示，称为码字。

1）定长编码：每个字符的编码长度一样。

- 如上例，考虑到有六个字符，可以用3位码字对每个字符编码，如表中的定长编码方案。
- 10万个字符需要用30万个二进制位来对文件编码。

2）变长编码：每个字符赋予不同长度的码字。

- 思路：赋予高频字符短码字，低频字符长码字，**字符的码字互不为前缀，这样才能唯一解码。**
- 如表中变长编码方案：a用1位的串0表示，b用3位的串101表示，f用4位的串1100表示等。
- 10万个字符仅需22.4万个二进制位，节约了25%的空间。

最优编码方案的设计

前缀码(Prefix code)：任何码字都不是其它码字的前缀。

文件编码过程：将文件中的每个字符的码字连接起来即可完成文件的编码过程。

如，设文件中包含3个字符：abc

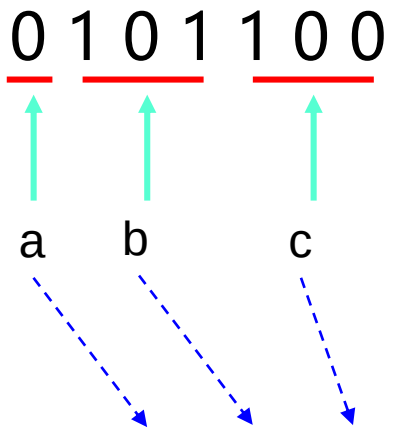
字符编码（前缀码）：

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

文件编码：0101100

文件解码过程：

利用前缀码的性质：由于没有码字是其它码字的前缀，所以编码文件的开始部分是没有歧义的，可以唯一地转换回原字符，然后对编码文件剩余部分重复解码过程，即可“解读”出原来的内容。



对每一个二进制位串，在码字表里都只有一个字符唯一地与之对应

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

编码树：一种为表示字符二进制编码而构造的二叉树。

叶子结点：对应给定的字符，每个字符对应一个叶子结点。

编码构造：字符的二进制码字由根结点到该字符叶子结点的简单路径表示：0代表转向左孩子，1代表转向右孩子。

如：

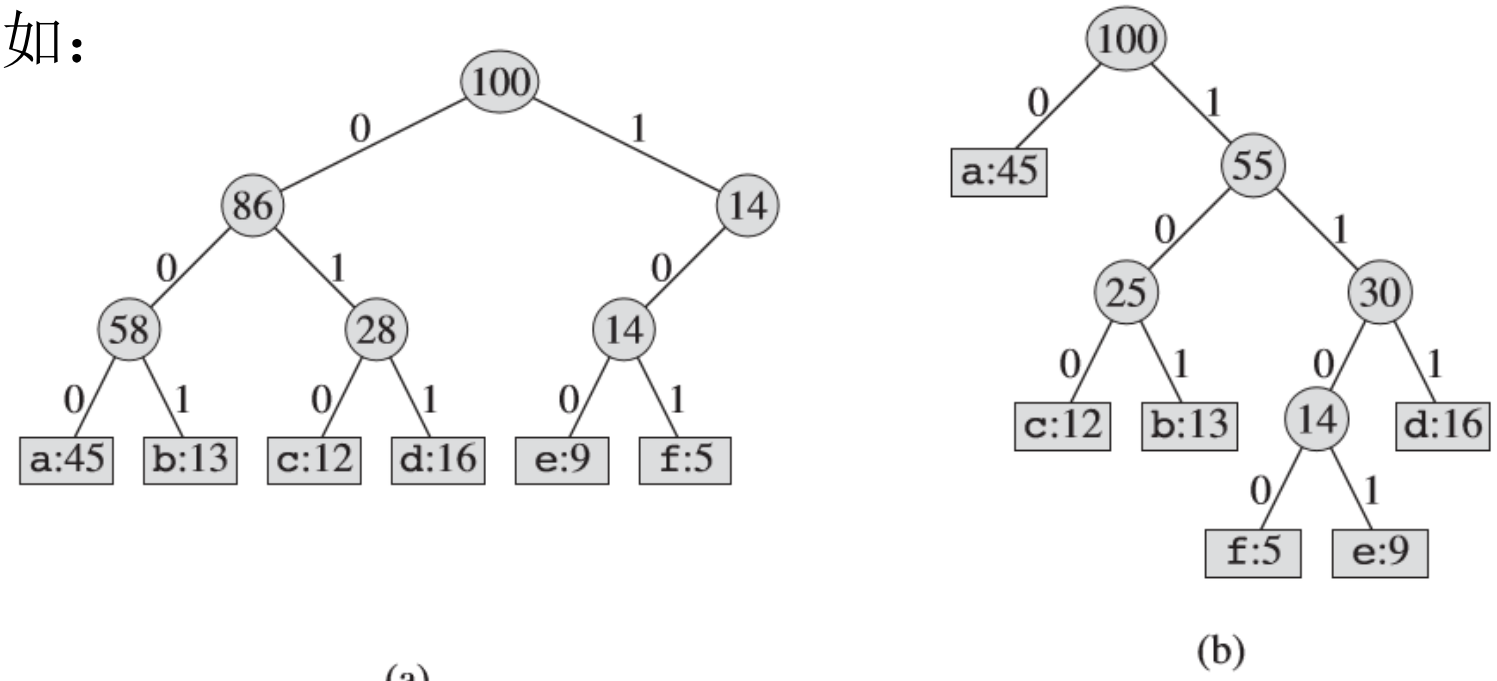
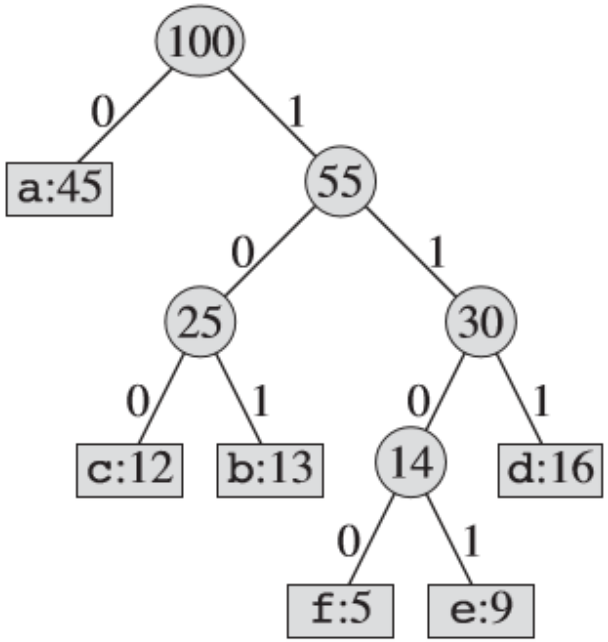


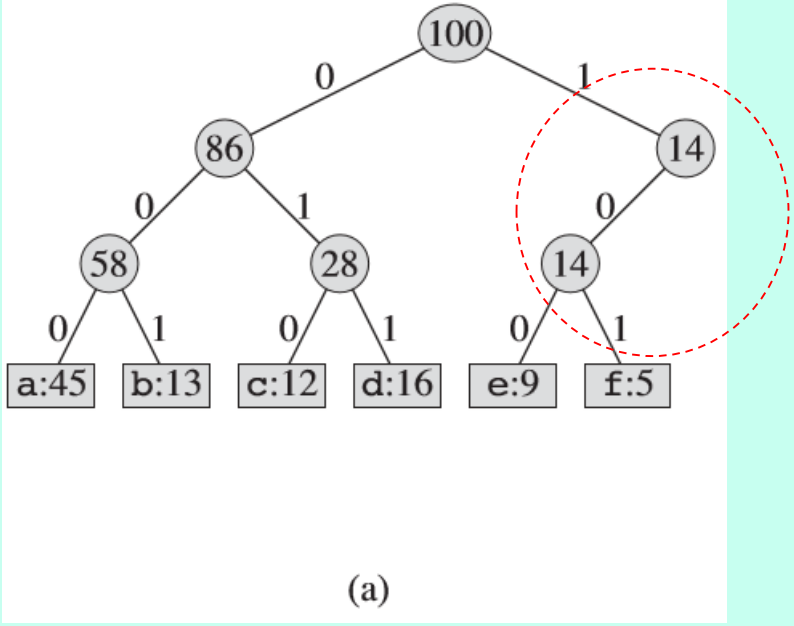
图 16-3 中编码方案的二叉树表示。每个叶结点标记了一个字符及其出现频率。每个内部结点标记了其子树中叶结点的频率之和。(a)对应定长编码 a=000，…，f=101 的二叉树。(b)对应最优前缀码 a=0，b=101，…，f=1100 的二叉树

一个文件的最优字符编码方案总对应一棵**满(full)**二叉树，即**每个非叶子结点都有两个孩子结点**。

如图(b)：



(b)



(a)

图(a)的定长编码实例不是最优的，因为它的编码树不是满二叉树：包含以10开头的码字，但不包含以11开头的码字。

最优编码方案

文件的最优编码方案对应一棵满二叉树（full binary tree）：

- 设C为字母表
  - 对字母表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（设所有字符的出现频率均为正数）。
  - 最优前缀码对应的树中恰好有|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。
- 令T表示一棵前缀编码树；
- 令$d_T(c)$表示c的叶子结点在树T中的深度（根到叶子结点的路径长度）。
  - $d_T(c)$也是字符c对应的码字的长度。

令B(T)表示采用编码方案T时**文件的编码长度**，则：

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

- 即文件要用B(T)个二进制位表示。

- 称B(T)为T的**代价**。
- **最优编码**：对给定的字符集和文件，使文件的编码长度最小的编码称为最优编码。
  - Huffman编码是一种最优编码。

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继续合并，这样执行|C|-1次“合并”后即可构造出一棵编码树——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 = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)
```

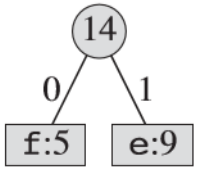
采用以freq为关键字的最小优先队列Q。提取两个最低频率的对象将之合并。

合并后，新对象的频率等于原来两个对象的频率之和。

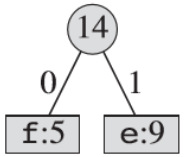
例：构造前面实例的Huffman编码

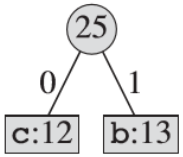
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

(a) f:5 e:9 c:12 b:13 d:16 a:45

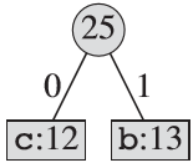
(b) c:12 b:13 

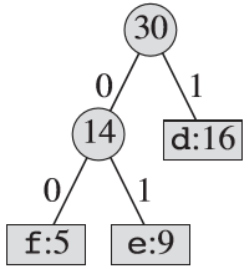
d:16 a:45

(c) 

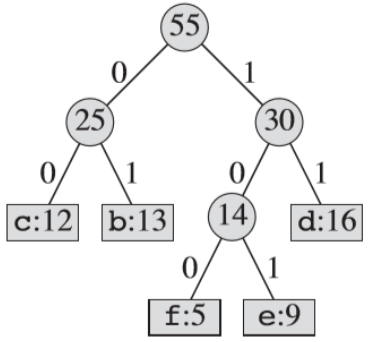
d:16 

a:45

(d) 

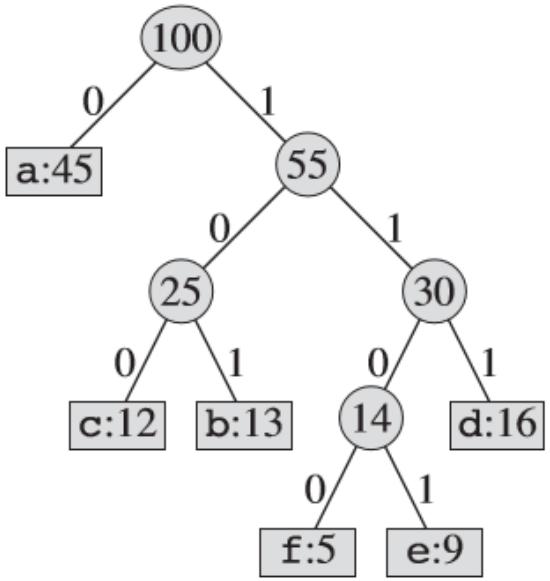


a:45

(e) a:45


- 每一步选择频率最低的两棵树进行合并。
- 叶子结点用矩形表示，每个叶子结点包含一个字符及其频率。
- 内结点用圆形结点表示，频率等于其孩子结点的频率之和。
- 内结点指向左孩子的边标记为0，指向右孩子的边标记为1。
- **一个字母的码字对应从根到其叶子结点的路径上的边的标签序列。**

  如 f 的码字是：1000，e的码字是1001



(b)

最后得到的前面实例的Huffman编码树

```
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 = EXTRACT-MIN(Q)
6      z.right = y = 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
```

时间分析

假设Q使用**最小二叉堆**实现，则

- 首先，Q的初始化花费O(n)的时间。
- 其次，循环的总代价是O($n$lg$n$)。
  - for循环共执行了n-1次，每次从堆中找出当前频率最小的两个结点及把合并得到的新结点插入到堆中均花费O(lg$n$)，所以循环的总代价是O($n$lg$n$).

所以，HUFFMAN的总运行时间O($n$lg$n$)。

注：如果将最小二叉堆换为van Emde Boas树（ Chp 20），可以将运行时间减少到O(nlglgn)

HUFFMAN算法的正确性

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∈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，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:

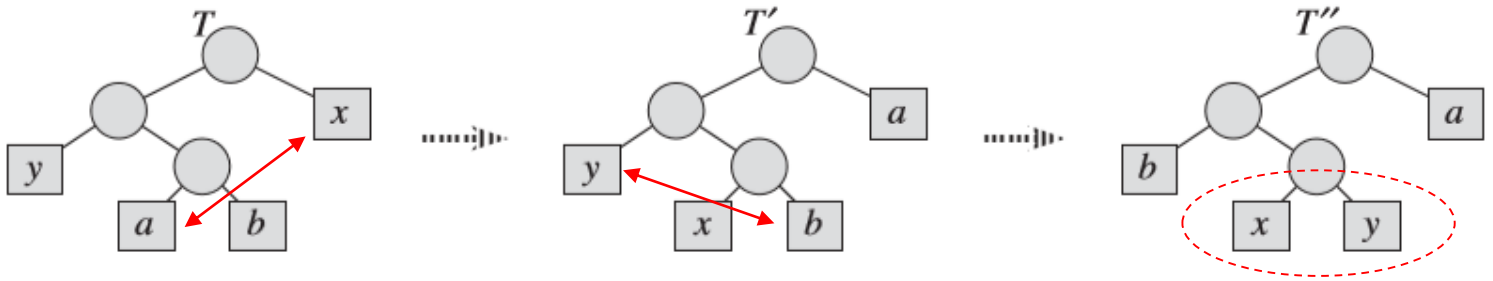
令T是一个最优前缀码所对应的编码树——满二叉树。

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≤b.freq且x.freq≤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≤a.freq且y.freq≤b.freq。

注：有可能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。
- 假定x.freq≠b.freq，即x≠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是深度最深的两个兄弟结点。如图所示：



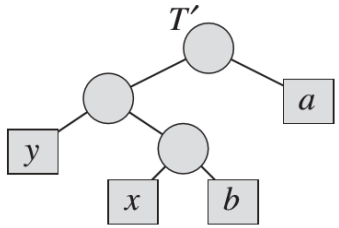
在最优树T中，叶子结点a和b是最深的叶子结点中的两个，并且是兄弟结点。叶子结点x和y为算法首先合并的两个叶子结点，它们可出现在T中的任意位置上。假设x≠b，叶子结点a和x交换得到树T’，然后交换叶子结点b和y得到树T”。

根据文件编码的计算公式，T和T’的**代价差**为：

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

$$\begin{aligned} & B(T) - B(T') \\ = & \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\ = & x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\ = & x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\ = & (a.freq - x.freq)(d_T(a) - d_T(x)) \\ \geq & 0 , \end{aligned}$$

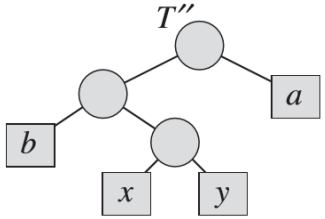
注，其中a.freq-x.freq和dT(a)-dT(x)均为非负值。



- B(T)-B(T’)≥0 表示从T到T’并没有增加代价。
- 类似地，从T’到T”，交换y和b也不会增加代价，即

  B(T’)-B(T”)≥0。

  因此， B(T”)≤ B(T)。



根据假设，T是最优的，因此B(T”)= B(T)，即得证：T”也是最优解，且x和y是其中深度最大的两个兄弟结点，x和y的码字长度相同，且只有最后一个二进制位不同。

得证。

引理 16.2 令C为一个字母表，其中每个字符c∈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，x和y的码字长度相同，且只有最后一个二进制位不同。

下面不失一般性，通过合并来构造最优树。

- 贪心选择：每次选择出现频率最低的两个字符。
  - 将一次合并操作的代价视为被合并的两项的频率之和，而编码树构造的总代价等于所有合并操作的代价之和。
  - 下面的引理表明：在所有的合并操作中，HUFFMAN选择是代价最小的方案：

引理 16.3 令C为一个给定的字母表，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。
- 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，即$C' = C - \{x, y\} \cup \{z\}$。
  - 类似C，也为C'定义freq，且$z.freq = x.freq + y.freq$。
- 令T'为字母表C'的任意一个最优前缀码对应的编码树。

则有：可以将T'中叶子结点z替换为一个以x和y为孩子的内部结点，得到树T，而T表示字母表C的一个最优前缀码。

证明:

对C中不是x和y的字符c，即c∈C-{x,y}，有

$$d_T(c)=d_{T'}(c),$$

亦有：c.freq • $d_T(c)$=c.freq • $d_{T'}(c)$。

而对于z和x、y，由于 $d_T(x)=d_T(y)=d_{T'}(z)+1$

故有：

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

从而可得：$B(T) = B(T') + x.freq + y.freq$

或等价地：$B(T') = B(T) - x.freq - y.freq$

下面用反证法证明T对应的前缀码是C的最优前缀码：

假定T对应的前缀码不是C的最优前缀码。则会存在最优前缀码树T″满足：B(T″)<B(T)。

不失一般性，由引理16.2有，T″包含兄弟结点x和y。

令T‴为将T″中x、y的父结点替换为叶结点z得到的树，其中z.freq=x.freq+y.freq。于是

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

这与T′对应C′的一个最优前缀码的假设矛盾。

因此，T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即可得。

贪心选择性：

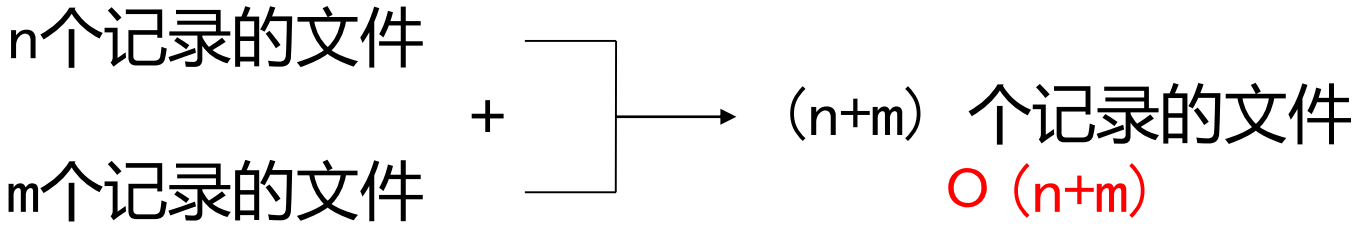
- 引理16.2说明首次选择频率最低的两个字符和选择其它可能的字符一样，都可以构造相应的最优编码树。
- 引理16.3说明首次贪心选择，选择出频率最低的两个字符x和y，合并后将z加入元素集合，可以构造包含z的最优编码树，而还原x和y，一样还是最优编码树。
- 所以贪心选择性成立。

16.4 最优归并模式问题

1. 问题的提出

1）归并问题

归并：如MergeSort中的Merge过程， 两个已知文件的归并，计算时间＝O(两个文件的元素总数)



2）归并模式

当有多个文件需要归并时，存在一个归并方式的选择，亦即寻找归并模式的问题。

已知n个文件，将它们归并成一个单一的文件，可有不同的归并模式。例：假定文件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：

① $X_1+X_2=Y_1+X_3=Y_2+X_4=Y_3$

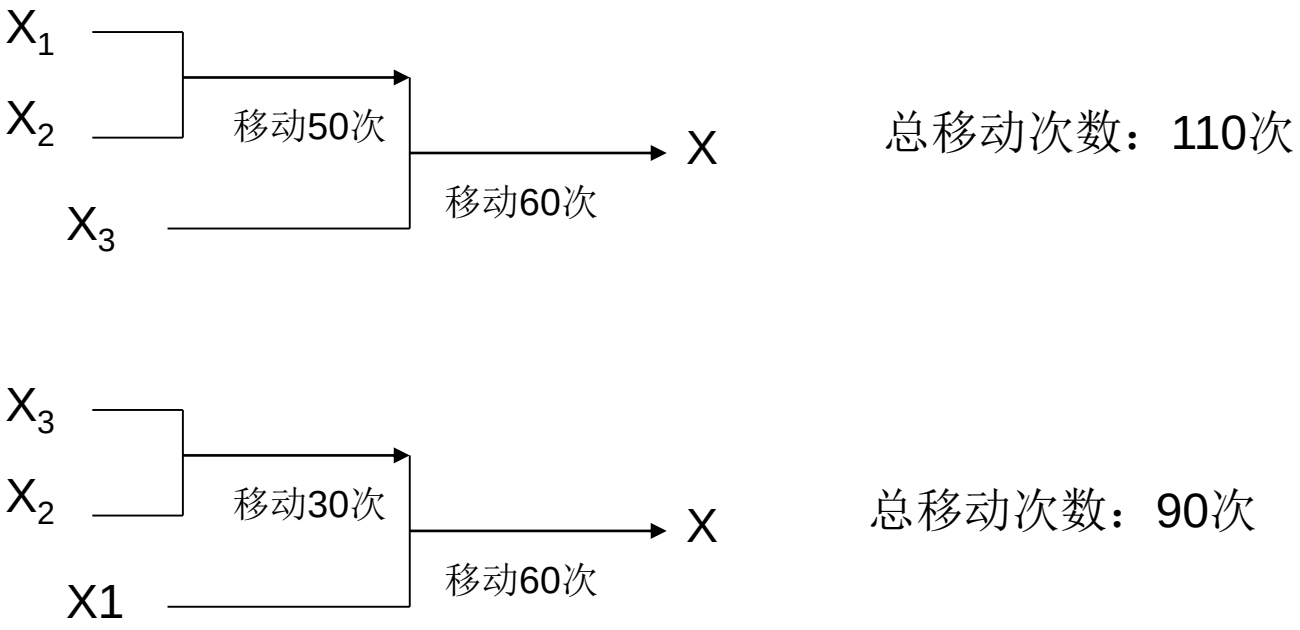
② $X_1+X_2=Y_1$

$Y_1+X_3=Y_2$ → Y_3

$X_3+X_4=Y_2$

不同的归并模式所需的计算代价可能是不同的。

例 已知X_1, X_2, X_3是分别为30、20、10个记录长度的已分类文件。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如下：



问题：采用何种归并模式才能使归并过程中元素的移动次数最少（或执行的速度最快）？

2.二路归并模式

每次仅作两个文件归并的归并模式称为二路归并模式。

该模式下，当有多个文件时，采用两两归并的模式，最终得到一个完整的记录文件。

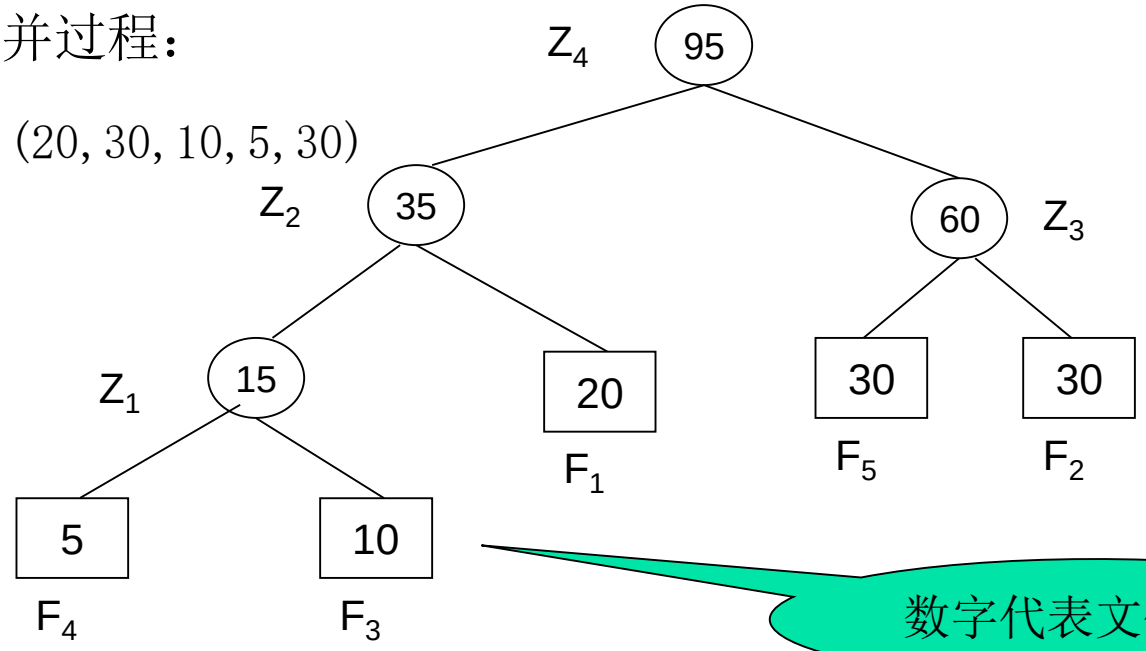
二路归并模式的归并过程可以用一棵二元树进行描述，称之为二元归并树。如下，

1）二元归并树的结构

- ◆外结点：代表n个原始文件。
- ◆内结点：代表归并过程中的中间文件，根代表最终的文件。
- ◆在两路归并模式下，每个内结点刚好有两个儿子，代表把它的两个儿子表示的文件归并成其本身所代表的文件。

如五个文件的归并过程：

$(F_1, F_2, F_3, F_4, F_5) = (20, 30, 10, 5, 30)$

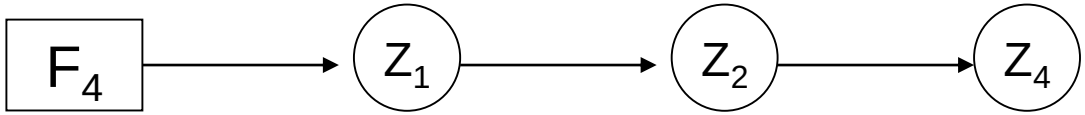


2) 最优二路归并模式

建模：

在二元归并树中，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＝该外部结点到根的距离，即根到该外部结点路径的长度。如上图中，

F_4 ：



则，F_4中的所有记录在整个归并过程中均需移动3次，总需$|F_4|$*3次移动。

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，则这棵树所代表的归并过程中元素的移动总量是：

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

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。

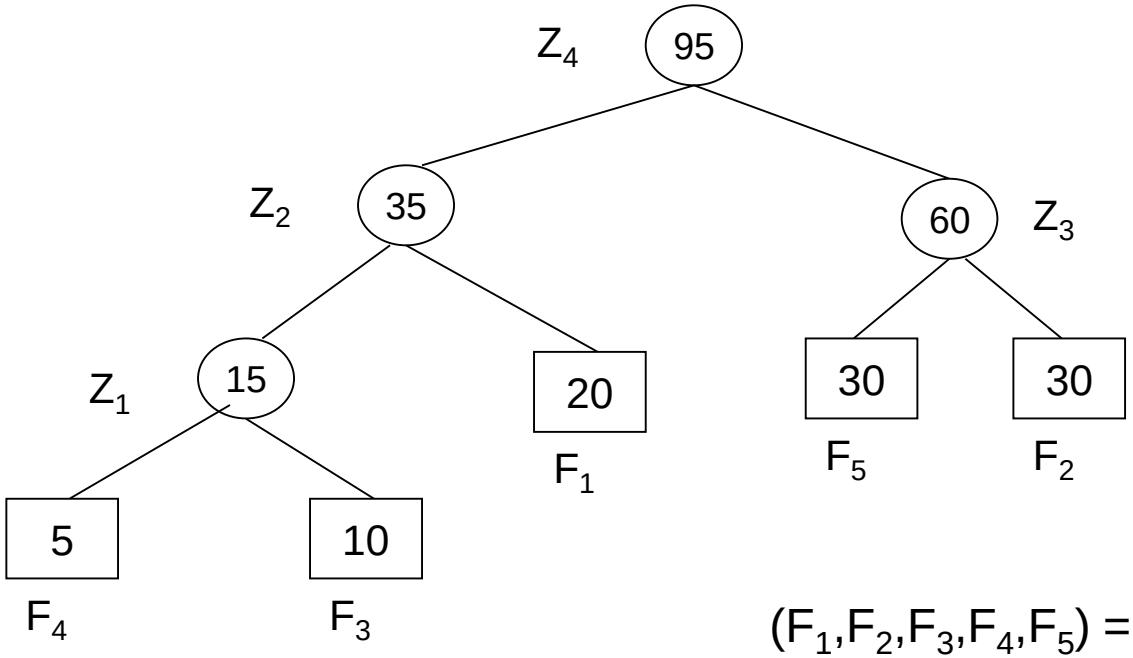
对已知的n个文件，怎么求取其相应的最优二路归并模式？

2. 贪心策略求解

1）度量标准的选择

这里以目标函数做为度量。为使其达到最小，每次归并应使目标函数有最小的增加。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，故得处理规则如下：每次选择长度最小的两个文件进行归并。

如：



$(F_1,F_2,F_3,F_4,F_5) = (20,30,10,5,30)$

2)最优二路归并模式算法

```
procedure TREE(L,n)
 //L是n个单结点的二元树表//
  for i←1 to n-1 do
      call GETNODE(T)              //构造一颗新树T//
      LCHILD(T) ←LEAST(L)          //从L中选当前根WEIGHT最小的树，并从中删除//
      RCHILD(T) ←LEAST(L)
      WEIGHT(T) ←WEIGHT(LCHILD(T))+WEIGHT(RCHILD(T))
      call INSERT(L,T)             //将归并的树T加入到表L中//
  repeat
  return (LEAST(L))                //此时，L中的树即为归并的结果//
end TREE
```

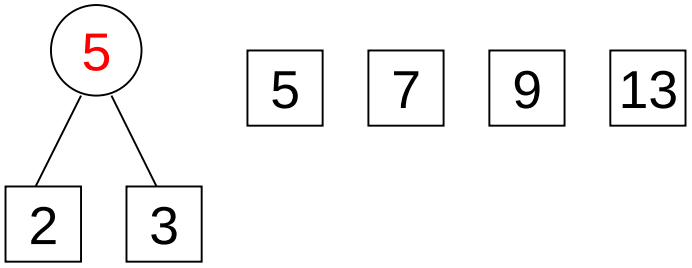
例5.4 已知六个初始文件，长度分别为：2,3,5,7,9,13。

采用算法TREE，各阶段的工作状态如图所示：

迭代　　L

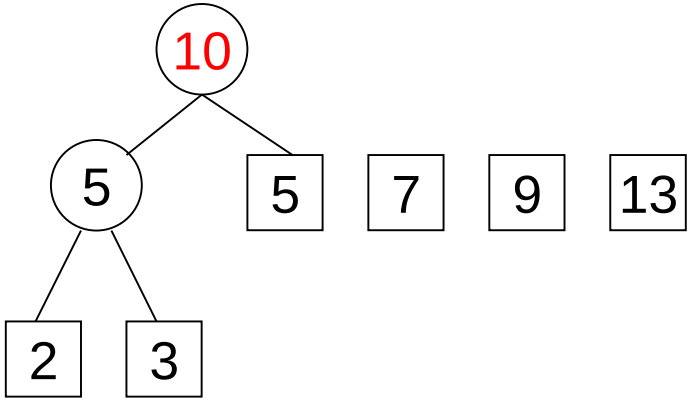
0　2 3 5 7 9 13

1



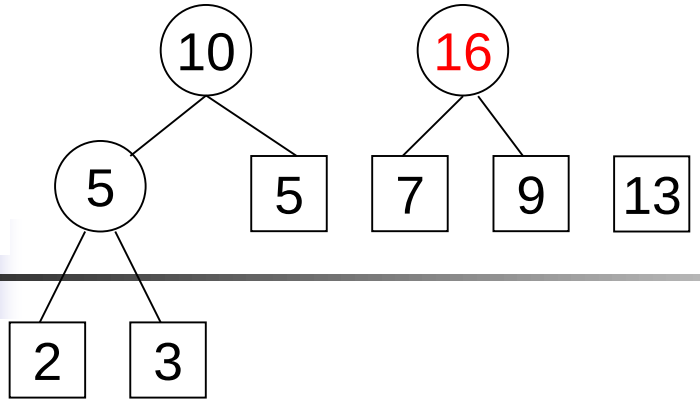
5 7 9 13

2

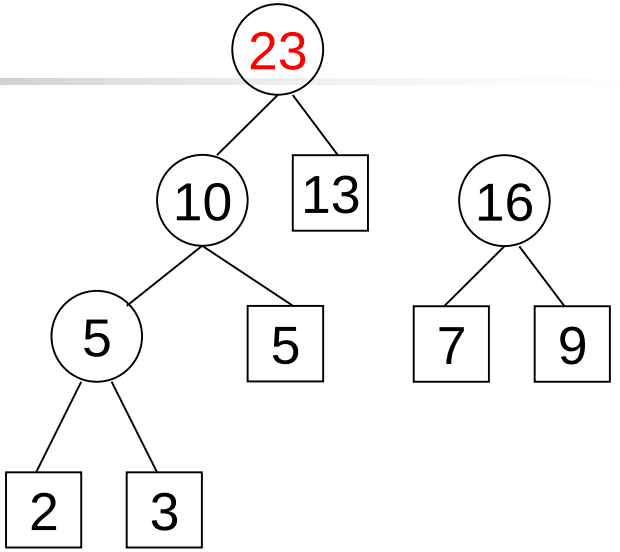


5 7 9 13

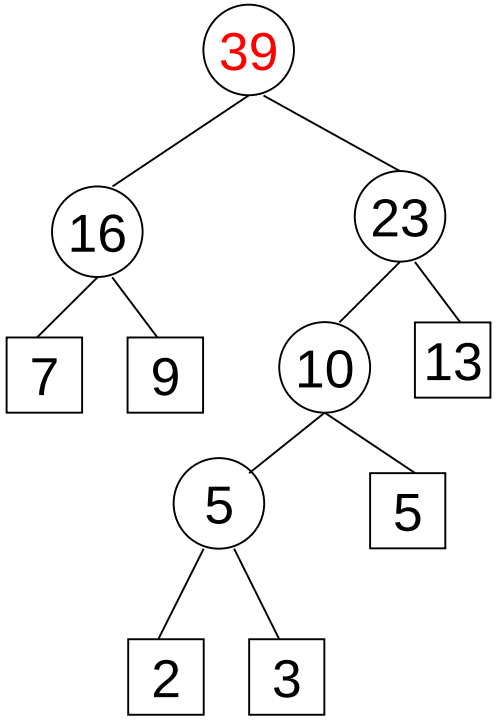
3



4



5



3)算法的时间分析

(1) 循环体：n-1次

(2) 若L以有序表表示

LEAST(L)：O(1)

INSERT(L,T)：O(n)

总时间：O(n^2)

(3) 若L以min-堆表示

LEAST(L)：O(logn)

INSERT(L,T)：O(logn)

总时间：O(nlogn)

3. 最优解的证明

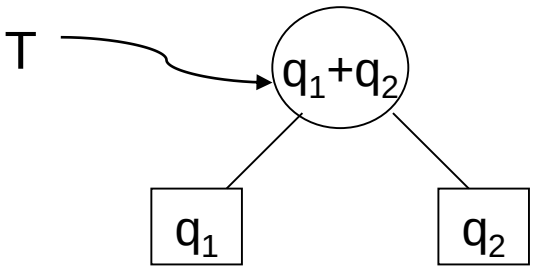
定理5.4 若L最初包含n≥1个单结点的树，这些树有WEIGHT值为$(q_1, q_2, \cdots, q_n)$，则算法TREE对于具有这些长度的n个文件生成一棵最优的二元归并树。

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 假定算法对所有的$(q_1, q_2, \cdots, q_m)$，$1 \leqslant m < n$，生成一棵最优二元归并树。

③ 对于n，不失一般性，假定$q_1 \leqslant q_2 \leqslant \cdots \leqslant q_n$，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对应的树构成的子树：



- 设T’是一棵关于$(q_1, q_2, \cdots, q_n)$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。

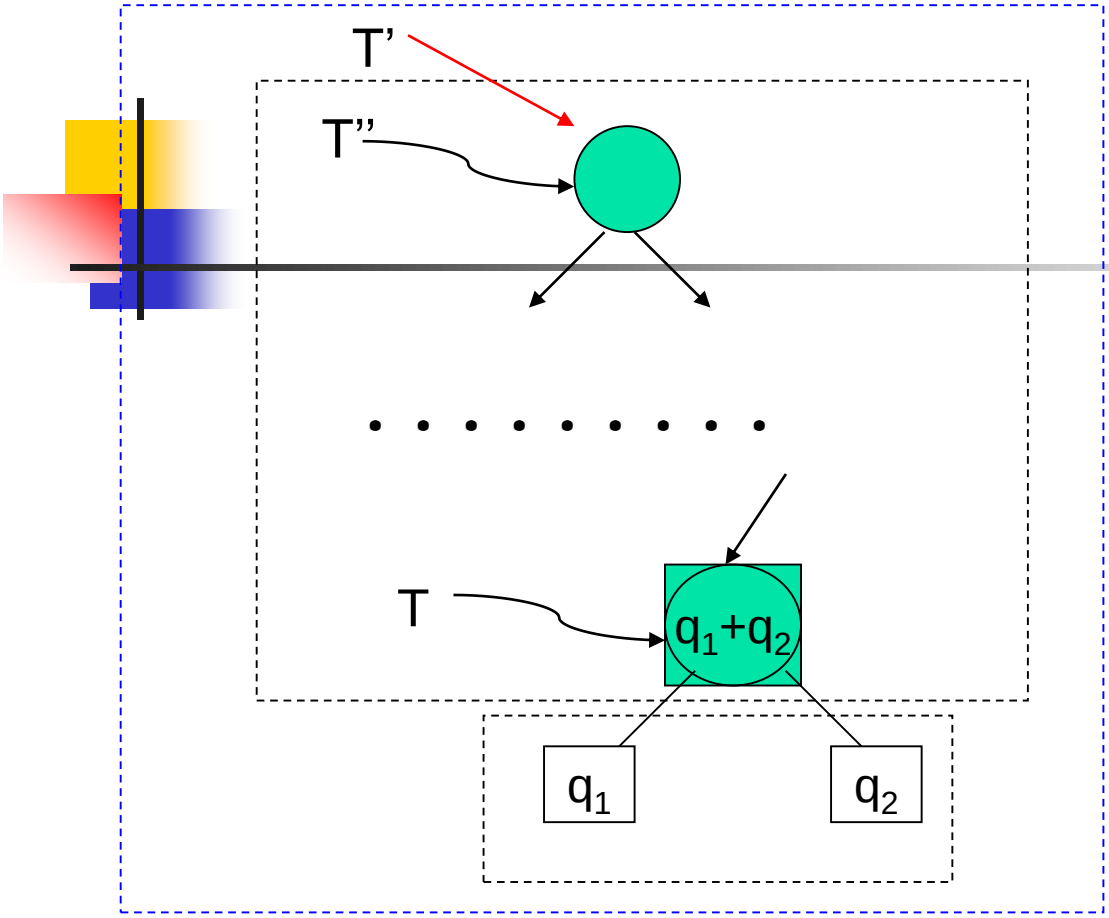
若P的两棵子树不是q_1和q_2，则用q_1和q_2代换P当前的子树而不会增加T’的带权外部路径长度。

故，T应是最优归并树中的子树。

则在T’中用一个权值为q_1+q_2的外部结点代换T，得到的应是一棵关于$(q_1+q_2, \cdots, q_n)$最优归并树T”。

而由归纳假设，在用权值为q_1+q_2的外部结点代换了T之后，过程TREE将针对$(q_1+q_2, \cdots, q_n)$得到一棵最优归并树。将T带入该树，根据以上讨论，将得到关于$(q_1, q_2, \cdots, q_n)$的最优归并树。

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



$F(T') = F(T'') + q_1 + q_2$

则，

$F_{min}(T') = \min(F(T'))$

$= \min(F(T'') + q_1 + q_2)$

$= \min(F(T'')) + q_1 + q_2$

$= F_{min}(T'') + q_1 + q_2$

4. 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：可以用一颗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。

★ 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 mod (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。

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，M=15，（$p_1, \cdots, p_7$）=(10, 5, 15, 7, 6, 18, 3)，（$w_1, \cdots, w_7$）=(2, 3, 5, 7, 1, 4, 1)。