第四章 贪心算法

§1. 贪心算法基本思想

找零钱 假如售货员需要找给小孩 67 美分的零钱。现在，售货员手中只有 25 美分、10 美分、5 美分和 1 美分的硬币。在小孩的催促下，售货员想尽快将钱找给小孩。她的做法是：先找不大于 67 美分的最大硬币 25 美分硬币，再找不大于 67－25＝42 美分的最大硬币 25 美分硬币，再找不大于 42－25＝17 美分的最大硬币 10 美分硬币，再找不大于 17－10＝7 美分的最大硬币 5 美分硬币，最后售货员再找出两个 1 美分的硬币。至此，售货员共找给小孩 6 枚硬币。售货员的原则是拿尽可能少的硬币找给小孩。

装载问题 有一艘大船用来装载货物。假设有 n 个货箱，它们的体积相同，重量分别是 $w_1, w_2, \cdots, w_n$，货船的最大载重量是 c。目标是在船上装更多个数的货箱该怎样装？当然，最简单的做法是“捡重量轻的箱子先往上装”，这是一种贪心的想法。如果用 $x_i = 1$ 表示装第 i 个货箱，而 $x_i = 0$ 表示不装第 i 个货箱，则上述问题是解优化问题：

求 $x_1, x_2, \cdots, x_n$，使得

$$\max \sum_{i=1}^{n} x_i \tag{4.1.1}$$

满足条件 $\sum_{i=1}^{n} w_i x_i \le c, \quad x_i = 0,1$ (4.1.2)

贪心算法，顾名思义，是*在决策中总是做出在当前看来是最好的选择*。例如找零钱问题中，售货员每捡一个硬币都想着使自己手中的钱尽快达到需要找钱的总数。在装载问题中，每装一个货箱都想着在不超重的前提下让船装更多的箱子。但是贪心方法并未考虑整体最优，它所做出的选择只是在某种意义上的局部最优。当然，在采用贪心算法时未必不希望结果是整体最优的。事实上，有相当一部分问题，采用贪心算法能够达到整体最优，如前面的找零钱问题以及后面将要讲到的单点源最短路径问题、最小生成树问题、工件排序问题等。为了更好理解贪心算法，我们将装载问题稍加推广，考虑可分割的背包问题。

背包问题 已知容量为 M 的背包和 n 件物品。第 i 件物品的重量为 w_i，价值是 p_i。因而将物品 i 的一部分 x_i 放进背包即获得 $p_i x_i$ 的价值。问题是：怎样装包使所获得的价值最大？即是如下的优化问题：

$$\max \sum_{1\le i\le n} p_i x_i \tag{4.1.3}$$

$$\sum_{1\le i\le n} w_i x_i \le M \tag{4.1.4}$$
$$0 \le x_i \le 1,\ p_i > 0, w_i > 0,\ 1 \le i \le n$$

采用贪心算法，有几种原则可循：a）.每次捡最轻的物品装；b）.每次捡当前价值最大的物品装；c）.每次装包时既考虑物品的重量又考虑物品的价值，也就是说每次捡单位价值 p_i/w_i 最大的物品装。按原则 a)来装只考虑到多装些物品，但由于单位价值未必高，总价值可能达不到最大；按原则 b)来装，每次选择的价值最大，但同时也可能占用了较大的空间，装的物品少，未必能够达到总价值最大。比较合理的原则是 c)。事实上，按照原则 c）来装，确实能够达到总价值最大。

程序 4-1-1 背包问题贪心算法

```
proc GreedyKnapsack (p, w, M, x, n) //价值数组 p[1..n]、重量数组
  // w[1..n]，它们元素的排列顺序满足 p[i]/w[i]≥p[i+1]/w[i+1];
  // M 是背包容量，x 是解向量
  float p[1..n], w[1..n], x[1..n], M, rc;
  integer i, n;
  x:= 0;  // 将解向量初始化为零
  rc:= M; // 背包的剩余容量初始化为 M
  for i to n do
    if w[i] ≤ rc then
      x[i]:=1; rc:=rc-w[i];
    else break;
    end{if}
  end{for}
  if i≤n then
    x[i]:=rc/w[i];
  end{if}
end{GreedyKnapsack}
```

定理 4.1.1 如果 $p[1]/w[1] \ge p[2]/w[2] \ge \cdots \ge p[n]/w[n]$，则 GreedyKnapsack 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 GreedyKnapsack 所生成的解，但不是最优解。

因而必有某个x_i不为 1。不妨设x_j是第一个这样的分量。于是，当$1 \le i < j$时，$x_i = 1$；当$i = j$时，$0 \le x_i < 1$；当$j < i \le n$时，$x_i = 0$，而且$\sum w_i x_i = M$。因为x不是最优解，必存在解向量$y = (y_1, y_2, \cdots, y_n)$，使得$\sum p_i y_i > \sum p_i x_i$。设$k$是使得$y_k \ne x_k$的最小下标，**则 $y_k < x_k$**. 这是因为，当$k < j$时，$x_k = 1$, 上述不等式自然成立；当$k \ge j$时，若 $x_k < y_k$，则由$x_1 = y_1, \cdots, x_{k-1} = y_{k-1}, x_{k+1} = 0, \cdots x_n = 0$, 可推出

$$\sum_{i=1}^{k-1} w_i y_i + w_k y_k + \sum_{i=k+1}^{n} w_i y_i > \sum_{i=1}^{k-1} w_i x_i + w_k x_k = \sum_{i=1}^{n} w_i x_i = M$$

y不是可行解，矛盾。

因y是比x更优的解，不失一般性，可以假定$\sum_{i=1}^{n} w_i y_i = M$。于是

$$\sum_{i=1}^{k-1} w_i y_i + w_k y_k + \sum_{i=k+1}^{n} w_i y_i = \sum_{i=1}^{k-1} w_i x_i + w_k x_k \quad (+ \sum_{i=k+1}^{j} w_i x_i \text{ 当 } k < j \text{ 时})。$$

再由$y_k < x_k$，有$\sum_{i=k+1}^{n} w_i y_i \ge w_k (x_k - y_k) > 0$。现在取新的向量$z = (z_1, z_2, \cdots, z_n)$满足

$$z_1 = y_1, \cdots, z_{k-1} = y_{k-1}, z_k = x_k, \quad 0 \le z_{k+1} \le y_{k+1}, \cdots, 0 \le z_n \le y_n \quad ,$$

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

这样的向量z是存在的，而且是背包问题的可行解，因为

$$\begin{aligned} \sum_{1 \le i \le n} w_i z_i &= \sum_{1 \le i \le k-1} w_i y_i + w_k z_k + \sum_{k+1 \le i \le n} w_i z_i \\ &= \sum_{1 \le i \le k-1} w_i y_i + \sum_{k \le i \le n} w_i y_i \\ &= \sum_{1 \le i \le n} w_i y_i \le M \end{aligned}$$

至此，我们找到一个新的解向量z。以下证明它的总价值不小于y的总价值：

$$\begin{aligned} \sum_{1 \le i \le n} p_i z_i &= \sum_{1 \le i \le n} p_i y_i + (z_k - y_k) w_k p_k / w_k - \sum_{k+1 \le i \le n} (y_i - z_i) w_i p_i / w_i \\ &\ge \sum_{1 \le i \le n} p_i y_i + \left((z_k - y_k) w_k - \sum_{k+1 \le i \le n} (y_i - z_i) w_i \right) p_k / w_k \\ &= \sum_{1 \le i \le n} p_i y_i \end{aligned}$$

中间的不等式是由于当$i>k$时有$p[k]/w[k]\geq p[i]/w[i]$而得。但是z与x的第一个不同分量的位置比y与x的第一个不同的分量的位置至少向后推迟了一个。以z代替y进行上面的讨论，我们又可以找到新的解向量z'，如此等等，由于分量的个数n有限，这样的过程必到某一步停止，最后找到解向量$y*$,它和x有相同的分量，又比x有更大的总价值，矛盾。这个矛盾源于x不是最优解的假设。证毕

贪心算法主要用于处理优化问题。每个优化问题都是由目标函数和约束条件组成。满足约束条件的解称为可行解，而那些使得目标函数取得最大（最小）值的可行解称为最优解。如背包问题是一个优化问题，式(4.1.3)中的函数是目标函数，而(4.1.4)式描述的要求是约束条件，这里优化是使目标函数取最大值。

贪心算法在每一步的决策中虽然没有完全顾忌到问题整体优化，但在局部择优中是朝着整体优化的方向发展的。为此，贪心算法首先要确定一个度量准则（称为贪心准则），每一步都是按这个准则选取优化方案。如背包问题的贪心准则是***选取单位价值 p/w 最大物品；***而装载问题的贪心的准则是***选取最轻的货箱；***找零钱问题所用的贪心准则是***选取面值最大的硬币。***对于一个给定的问题，初看起来，往往有若干种贪心准则可选，但在实际上，其中的多数都不能使贪心算法达到问题的最优解。如背包问题的下面实例：

n=3, M=20, p=(25, 24, 15), w=(18,15,10)

如果以价值最大为贪心准则,则贪心算法的执行过程是:首先考虑将物品 1 装包，此时获得效益值 25，包的剩余容量是 2。然后考虑将物品 2 装包，但物品 2 的重量 15 超出包的剩余容量，只能装入该种物品的 2/15，此时获得的总效益值为

25+24×2/15=28.2。

这样得到的可行解（1，2/15，0）并不是最优解。

事实上，如果以单位价值最大为贪心准则，则贪心算法的执行过程是：先计算出各个物品的单位价值(25/18, 24/15, 15/10)=(1.389, 1.6, 1.5)。首先考虑单位价值大的物品装包，即将物品 2 装包，此时获得效益值 24，背包的剩余容量是 5。然后考虑装物品 3，由于物品 3 的重量超出背包的剩余容量，只能装入该物品 5/10 =1/2, 至此背包已经装满,所得的总的效益值为 24+15/2=31.5。比前面的装法的效益值大。实践证明，选择能产生最优解的贪心准则是设计贪心算法的核心问题。以下给出贪心算法流程的伪代码。

程序 4-1-2 贪心算法抽象化控制流程

```
proc Greedy(A, n) // A[1:n]代表那个输入
   solution={}; //解向量初始化为空集
```

```
    for i to n do
        x:=Select(A);
        if Feasible(solution, x) then
            solution:=Union(solution, x);
        end{if}
    end{for}
    return(solution);
end{Greedy}
```

这里 Select(A)是按照贪心准则选取 A 中的输入项；Feasible(solution, x)是判断已知的解的部分 solution 与新选取的 x 的结合 Union(solution, x)是否是可行解。过程 Greedy 描述了用贪心策略设计算法的主要工作和基本控制流程。一旦给出一个特定的问题，就可将 Select，Feasible 和 Union 具体化并付诸实现。

§2. 调度问题

● 活动安排问题

我们首先从活动安排这一简单问题入手。该问题要求高效安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能够兼容地使用公共资源。

问题：*已知 n 个活动 E={1, 2, …, n}，要求使用同一资源，第 k 个活动要求的开始和结束时间为 s_k, f_k, 其中 $s_k < f_k$, k=1, 2, …, n . 活动 k 与活动 j 称为相容的如果 $s_k > f_j$ 或者 $s_j > f_k$。活动安排问题就是要在所给的活动集合中选出最大(活动个数最多)的相容活动子集。*

解决这个问题的基本思路是在安排时应该将结束时间早的活动尽量往前安排，好给后面的活动安排留出更多的时间，从而达到安排最多活动的目的。据此，贪心准则应当是：*在未安排的活动中挑选结束时间最早的活动安排*。在贪心算法中，将各项活动的开始时间和结束时间分别用两个数组 s 和 f 存储，并使得数组中元素的顺序按结束时间非降排列：$f_1 \leq f_2 \leq \cdots \leq f_n$。

算法 4-2-1 活动安排贪心算法伪代码

```
proc GreedyAction(s, f, n) // s[1..n]、f[1..n]分别代表 n 项活动的
    //起始时间和结束时间, 并且满足 f[1]≤ f[2]≤…≤ f[n]
    j:=1; solution:={1}; //解向量初始化
    for i from 2 to n do
```

```
    if si≥fj then
      solution:=solution ∪ {i}; // 将 j 加入解中
      j:=i;
    end{if}
  end{for}
  return(solution);
end{GreedyAction}
```

例 4.2.1 待安排的 11 个活动的开始时间和结束时间按结束时间的非减次序排列如下：

表 1 一个作业安排表

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

解集合为 {1,4,8,11}.容易证明算法 GreedyAction 所得到的解是最优解。

● 带期限的单机作业调度问题

为使问题简化，我们假定完成每项作业所用的时间都是一样的，如都是 1。带期限的单机作业安排问题陈述如下：

已知 n 项作业 E={1, 2, … ,n}，要求使用同台机器完成（该台机器在同一时刻至多进行一个作业），而且每项作业需要的时间都是 1。第 k 项作业要求在时刻 f_k 之前完成，而且完成这项作业将获得效益 p_k，k=1, 2, … , n 。作业集 E 的子集称为相容的，如果其中的作业可以被安排由一台机器完成。带限期单机作业安排问题就是要在所给的作业集合中选出总效益值最大的相容子集。

这个问题可以考虑用贪心算法求解。容易想到贪心准则应该是：

尽量选取效益值大的作业安排

用两个数组 f 和 p 分别存放作业的期限值和效益值，并使得数组 p 中元素按照不增的顺序排列。按照上述贪心准则，算法由前向后逐个考察各个作业是否可以加入到解集 J 中即可。J 初始为空集，按照相容性要求，作业 i 可以加入 J 只要当前解集 J 中期限值不大于 f_i 的作业少于 f_i 个。由该算法取得的解集 J 具有性质：**若作业 k 不属于 J，则 J 中恰有 f_k 个期限不超过 f_k、而效益值不小于 p_k 的作业。**该算法在程序 4-2-2 中实现。可以证明算法 GreedyJob 获得的解集是带期限单机作业调度问题的最优解。下面的伪代码给出带期限作业调度问题贪心算法的概略描述，其中没有给出相容性检验的具体步骤。

程序 4-2-2 带限期作业调度的贪心算法（概略）

```
proc GreedyJob(f, p, n) //f[1..n]和p[1..n]分别代表各项作业的
   //限期和效益值，而且n项作业的排序满足：p1 ≥ p2 ≥ … ≥ pn
   local J;
   J:={1};//初始化解集
   for i from 2 to n do
      if J ∪ {i} 中的作业是相容的 then //此步验证需要认真设计
        J:= J ∪ {i}; // 将i加入解中
      end{if}
   end{for}
end{GreedyJob}
```

定理 4.2.1 算法 GreedyJob 对于单机作业调度问题总是得到最优解。

证明：假设贪心算法所选择的作业集 J 不是最优解，则一定有相容作业集 I，其产生更大的效益值。假定 I 是具有最大效益值的相容作业集中使得|I∩J|最大者，往证 $I = J$ 。

反证法：若 $I = J$ 不成立，则这两个作业集 I 和 J 之间没有包含关系。这是因为算法 GreedyJob 的特性和假定 I 产生的效益值比 J 的效益值更大。假设 a 是 J\I 中具有最大效益的作业，即 **J 中比 a 具有更大效益的作业(如果有的话)都应该在 I 中**。如果作业 $b \in I \setminus J$ 且 $p_b > p_a$，那么由算法中对 J 中作业的选取办法（相容性要求），J 中至少有 f_b 个效益值 $\geq p_b$ 的作业，其期限值 $\leq f_b$。这些作业一定在 I 中，因而 I 中至少有 f_b+1 个作业，其期限值 $\leq f_b$，这与 I 的相容性矛盾。所以，$I \setminus J$ **中作业的效益值均不超过** p_a。

称区间 $[k-1,k]$ 为时间片 k，相容作业集 I 的一个调度表就是指定 I 中各个作业的加工时间片。如果 I 有一个调度表 S 将 f_a 时刻前的时间片安排的都是 $I \cap J$ 中的作业，且 $I \setminus J$ 中最早被安排的是作业 b，在时间片 k 上，则 $k > f_a$。前 $k-1$ 个时间片上安排的都是 J 中的作业，这些作业的集 A_1 再添上作业 a 得到 J 中 k 个作业。由作业集 J 的相容性，A_1 中至少有一个作业其期限值 $\geq k$。将这个作业与作业 b 交换安排的时间片，得到新的调度表 S_1。如果在调度表 S_1 中作业 b 安

排在时间片k_1，$k_1 > f_a$，则同理，可以将作业b再向前移，得到新的调度表S_2。如此做下去，必得到I的调度表S'，其在f_a时刻前的时间片安排有$I \setminus J$中作业b。令$I' = (I \setminus \{b\}) \cup \{a\}$，则$I'$也是相容的作业集。而且，由于$p_b \le p_a$，$I'$的效益总值不小于$I$的效益总值，因而$I'$具有最大的效益总值，但$|I' \cap J| > |I \cap J|$，与$I$的选取相悖。因而，$J = I$，$J$是最优解。 证毕

算法 GreedyJob 能够找到达到最大效益值的相容作业集，若按照作业期限的先后排序，就产生一个作业调度表。可以在算法中加建调度表的步骤：把被选中的作业安排在其期限允许的、而且期限晚于它的已安排作业之前的时间片上。如果用 J(r)表示安排在第 r 个时间片上的解集 J 中的作业，D(i)表示作业 i 的期限值，则相容性条件要求 D(J(r))≥r。根据上面分析，可以如下设计算法：

首先将作业 1 存入解数组 J 中，然后依次处理作业 2 到作业 n。假设已经处理了前 i-1 个作业，其中有 k 个作业 J(1),…,J(k)被选入 J 中，对它们的安排满足 D(J(1))≤ …≤ D(J(k))。为了检验作业集 J∪{i}的相容性，只需看能否找到按期限的非降次序插入作业 i 的适当位置，即，使得作业 i 在此处插入后有 D(J(r))≥r，1≤r≤k+1。找插入位置可以将作业 i 与 J 中作业依 D(J(k)),D(J(k-1)),…,D(J(1))的次序逐个比较。如果 D(i)≥D(J(j))>j 则选取作业 i 并将作业 i 插到作业 J(j)后面的位置上；如果 D(i)=D(J(j))=j，则作业 i 不能选入;如果 D(i)<D(J(j))且 D(i)≤j,则作业 i 不能选入;如果 D(i)<D(J(j))且 D(i)>j,则将作业 i 与作业 J(j-1)比较。如此下去，直到确定作业 i 不能被选入或可以选入并插入到 J 中适当位置为止。这样得到的作业集是相容的，而且作业是按期限的非降次序安排的。程序 4-2-3 给出了单机带期限作业调度贪心算法的伪代码。

程序 4-2-3 带期限作业调度问题贪心算法

```
proc GreedyJob(D,J,n,k)
 //D(1),…,D(n)是期限值，作业已按 p1 ≥ p2 ≥ … ≥ pn 排序。J(i)是
 //最优解中的第 i 个作业。终止时，D(J(i))≤ D(J(i+1))，1≤i≤k
 integer D[0..n],J[0..n],i,k,n,r
 D(0):=0; J(0):=0;  //初始化
 k:=1; J(1):=1;  //计入作业 1，k 表示当前选择的作业个数
 for i from 2 to n do
 //按 p 的非增次序考虑作业，检查作业 i 可否被选入，并找出插入位置
    r:=k;
    while D(J(r))>D(i) and D(J(r))<>r do
```

```
        r:=r-1;
      end{while};
      //期限不晚于D(i)的作业个数小于D(i)时
      if D(J(r))≤ D(i) and D(i)>r then
        for j from k to r+1 by -1 do //给作业i腾出位置
          J(j+1):=J(j);
        end{for};
       J(r+1):=i; k:=k+1;
      end{if};
    end{for};
  end{GreedyJob}
```

例 4.2.2 设 n=7, $(p_1, p_2, \cdots, p_n)=(35,30,25,20,15,10,5)$,

$(d_1, d_2, \cdots, d_n)=(4,2,4,3,4,8,3)$,

算法 GreedyJob 的执行过程可描述如下：

作业　1；　2，1；　2，1，3；　2，4，1，3；　2，4，1，3，6；

期限值　4；　2，4；　2，4，4；　2，3，4，4；　2，3，4，4，8；

算法的关键操作是比较和相应移动作业的位置，都是围绕着判断作业集 J∪{i}的相容性。考察 J 外的作业 i 该加入 J 时，作业至多和 J 中的每个作业都比较一次。因而，J ∪ {i}的相容性判断最坏情况下需要|J|次比较。上述算法在最坏情况下的时间复杂度为 $O(n^2)$。

为了避免上述算法反复移动作业的排序位置，可以考虑在算法中建立作业调度表，而且采取“将作业尽量安排在期限允许的最后面空闲时间片上”的策略。显然，所需要的时间片数不会超过 q:=min{n,max{D(i)|i=1,…,n}}。为了检验作业集 J 添加作业 i 是否还是相容的，只需检查时刻 k:=min{n,D(i)}前面是否还有空闲的时间片即可。例 4.2.2 按上述算法执行的情况如下：

作业　1；　2，1；　2，3，1；　4，2，3，1；　4，2，3，1，6；

期限值　4；　2，4；　2，4，4；　3，2，4，4；　3，2，4，4，8；

时间片　4；　2，4；　2，3，4；　1，2，3，4；　1，2，3，4，7；

其中，时间片 k 指区间[k-1,k]。

将时间片作为集合的元素，采用集合的并 Union 和查找 Find（参考附录“数据结构”）技术实现上述算法，可使时间复杂度达到 $O(n\cdot\beta(2n,n))$，其中 $\beta(2n,n)$

是 Ackermann 函数的逆函数。

程序 4-2-4 带期限作业调度快速算法

```
proc FastJob(D,J,n,q,k)
  //找最优解 J=J(1),…,J(k).假定 p1≥ p2≥ … ≥ pn.
  //q:=min{n,max{D(i)|i=1,2,…,n}}.
  integer q,D(n),J(n),F(0..q),P(0..q)
  for i from 0 to q do  //将集合树置初值
    F(i):=i; P(i):=-1;
  end{for}
  k:=0 //初始化
  for i to n do  //使用贪心规则
    j:=Find(min{n,D(i)});
    if F(j)≠0 then
      k:=k+1; J(k):=i;  //选择作业 i
      s:=Find(F(j)-1); Union(s,j);
      F(j):=F(s); P(s):=P(s)+P(j);
    end{if}
  end{for}
end{FastJob}
```

算法中 P(i)将作业链接到它的集合树上，同时指出该树上作业的个数。为了便于描述算法，还引进了 0 时间片[-1,0]。例 4.2.2 的执行情况在本章附页“快速作业调度算法”中描绘出来。

● 多机调度问题

设有 n 项独立的作业{1,2,…, n}，由 m 台相同的机器加工处理。作业 i 所需要的处理时间为 t_i。约定：任何一项作业可在任何一台机器上处理，但未完工前不准中断处理；任何作业不能拆分更小的子作业分段处理。*多机调度问题要求给出一种调度方案，使所给的 n 个作业在尽可能短的时间内由 m 台机器处理完。*

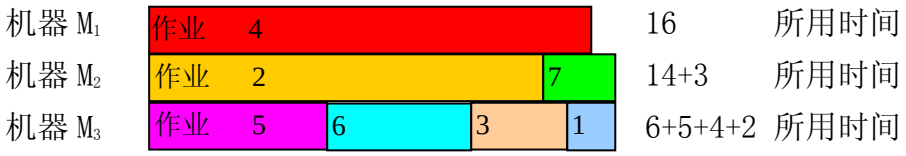
这是一个 NP 完全问题，到目前为止还没有一个有效的解法。利用贪心策略，有时可以设计出较好的近似解。可以采用贪心准则：*需要长时间处理的作业优先处理。*

例 4.2.3 设有 7 项独立的作业{1,2,3,4,5,6,7}，要由三台机器 M_1, M_2 , M_3 处理。各个作业所需要的处理时间分别为{2,14,4,16,6,5,3}.将作业标号按它们所需时间的长短排列：[4, 2, 5, 6, 3, 7,1].首先将作业 4 安排给机器 M_1 处理，

此时，机器 M_1 被占用的时间是 16，而机器 M_2 ，M_3 被占用的时间都是零，所以，应将作业 2 安排给机器 M_2 来处理，作业 5 应安排给机器 M_3 来处理，至此，机器 M_1，M_2 ，M_3 被占用的时间分别是 16、14、6。接下去应安排作业 6，因为机器 M_3 最早空闲，所以作业 6 应安排给机器 M_3。此时，机器 M_1，M_2 ，M_3 分别在 16、14、11 时刻开始空闲。所以下面将要安排的作业 3 安排给机器 M_3。此时，机器 M_1，M_2，M_3 分别在 16、14、15 时刻开始空闲。即将安排的作业 7 应安排给机器 M_2，此时，机器 M_1，M_2 ，M_3 分别在时刻 16、17、15 开始空闲。即将安排的作业 1 应安排给机器 M_3。如此全部作业均已安排完毕，所需要的时间是 17。这里采用的调度方案是：

将需要时间最长的未被安排作业首先安排给能够最早空闲下来的机器处理。

下面的图描述了上述例子的算法执行过程。



§3. 最优生成树问题

考虑无向图 G=(V, E)，不妨假定该图代表城市间的交通情况，顶点代表城市，边代表连接两个城市的可能的交通线路。现在将每条边赋予一个权值，这些权可以代表建造该条线路的成本、交通线的长度或其它信息，这样得到一个赋权图。在实际问题中，人们往往希望在结构上选择一组交通线，它们**连通所有的城市并且具有最小的建造成本。**这个问题相当于求取连通赋权图的具有最小权值的生成树。我们称之为最优生成树。贪心方法可以很好地解决此类问题。在此介绍两种算法：Prim 算法和 Kruskal 算法。

Prim 算法的基本思想：

1). 选择图 G 的一条权值最小的边 e_1，形成一棵两点一边的子树；

2). 假设 G 的一棵子树 T 已经确定；

3). 选择 G 的不在 T 中的具有最小权值的边 e，使得 $T\cup\{e\}$ 仍是 G 的一棵子树。

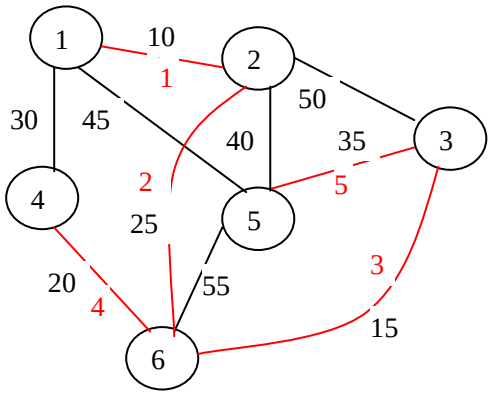
Kruskal 算法的基本思想：

1). 选择图 G 的一条权值最小的边 e_1；

2). 假设已经选好 G 的一组边 $L=\{e_1, e_2, \cdots, e_k\}$；

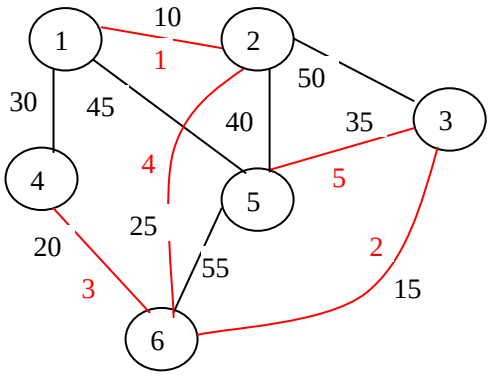
3). 选择 G 的不在 L 中的具有最小权值的边 e_{k+1}，使得 $L\cup\{e_{k+1}\}$ 诱导出的 G 的子图不含 G 的圈。下面两个图给出 Prim 算法和 Kruskal 算法的最优生成树产生过程。

Prim
算 法



(1,2)
(2,6)
(6,3)
(6,4)
(3,5)

Kruskal
算法



(1,2)
(6,3)
(6,4)
(6,2)
(5,3)

程序 4-3-1 Prim 最小生成树算法

```
proc PrimTree(E,COST,n,T,mincost) //E 是图 G 的边集，COST 是 G 的带权
    //邻接矩阵。计算一棵最小生成树 T 并把它作为一个集合存放到二维
    //数组 T[1..n-1,1..2]中，这棵树的权值赋给 mincost
1     real COST[1..n,1..n],mincost;
2     integer NEAR[1..n],n,i,j,k,s,T[1..n-1,1..2];
3     (k,s):= 权值最小的边;
4     mincost:=COST[k,s];
5     (T[1,1],T[1,2]):=(k,s); //初始子树
6     for i to n do //将 NEAR 赋初值（关于初始子树 T）
7        if COST[i,s]<COST[i,k] then
8          NEAR(i)=s;
9        else NEAR(i)=k;
10       end{if}
11    end{for}
12    NEAR(k):=0,NEAR(s):=0;
```

```
13  for i from 2 to n-1 do  //寻找 T 的其余 n-2 条边
14    choose an index j such that
          NEAR(j)≠0 and COST[j,NEAR(j)] is of minimum value;
15    (T[i,1],T[i,2]):=(j,NEAR(j)); // 加入边（j，NEAR(j)）
16    mincost:=mincost+COST[j,NEAR(j)];
17    NEAR(j):=0;
18    for t to n do  //更新 NEAR（关于当前子树 T）
19      if NEAR(t)≠0 and COST[t,NEAR(t)]>COST[t,j] then
20         NEAR(t):=j;
21      end{if}
22    end{for}
23  end{for}
24  if mincost ≥ ∞ then
25    print( 'no spanning tree' );
26  end{if}
27 end{PrimTree}
```

这里，树 T 的元素为(T[i,1],T[i,2]),其中 T[i,1],T[i,2]代表第 i 条边的两个端点。NEAR(j)表示顶点 j 的以最小权的边邻接 T 的一个顶点。

PrimTree 算法的时间复杂度分析：

语句 3 需要 O(|E|)的时间；语句 6 至语句 11 的循环体需要时间为 O(n)；

语句 18 至 22 的循环体需要时间 O(n)；

语句 14 至 17 需要时间 O(n)；

语句 13 至 23 的循环体共需要时间 $O(n^2)$。

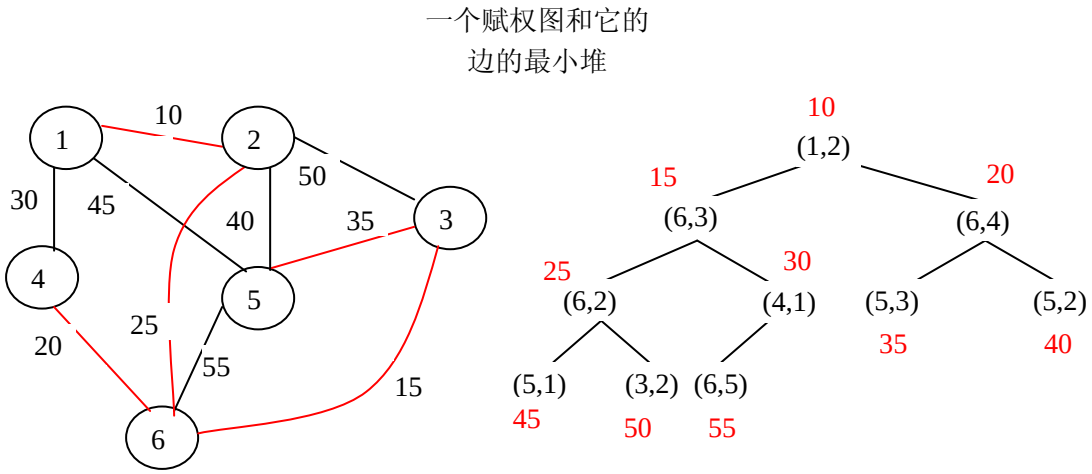
所以,PrimTree 算法的时间复杂度为 $O(n^2)$.

为叙述 Kruskal 算法，我们引进数据结构 min-堆：它是一个完全二叉树，其中每个结点都有一个权值，而且该二叉树满足：每个结点的权值都不大于其儿子的权值。min-堆的根具有最小的权值。

程序 4-3-2 Kruskal 最优生成树算法伪代码

```
proc KruskalTree(E,COST,n,T,mincost)//说明同算法 PrimTree
1  real mincost, COST[1..n,1..n];
2  integer Parent[1..n],T[1..n-1],n;
3  以带权的边为元素构造一个 min-堆;
4  Parent:=-1; //每个顶点都在不同的集合中;
```

```
5   i:= 0; mincost:= 0;
6   while i<n-1 and min-堆非空 do
7       从堆中删去最小权边（u，v）并重新构造 min-堆
8       j:= Find(u); k:= Find(v);
9       if j≠k then  //保证不出现圈
10        i=:i+1; T[i,1]:=u; T[i,2]:=v;
11        mincost:=mincost+COST[u,v];
12        Union(j,k); //把两个子树联合起来
13     end{if}
14  end{while}
15  if i≠n-1 then
16    print( 'no spanning tree' );
17  end{if}
18  return;
19 end{KruskalTree}
```



定理 4.3.1 Kruskal 算法对于每一个无向连通赋权图产生一棵最优树。

证明：设 T 是用 Kruskal 算法产生的 G 的一棵生成树，而 T' 是 G 的一棵最优生成树且使得|E(T')∩E(T)|最大。用 E(T)表示树 T 的边集，w(e)表示边 e 的权值，而边集 E 的权值之和用 w(E)表示。以下证明 E(T))=E(T')．反假设 E(T)≠E(T')，因为|E(T)|=|E(T')|，所以 E(T)与 E(T')没有包含关系。设 e 是 E(T)\E(T')中权值最小的边。将 e 添加到 T'中即得到 T'的一个圈：e，e_1，e_2，…，e_k。因为 T 是树，诸 e_i 中至少有一个不属于 E(T)。不妨设 e_i 不属于 E(T)，则必然有 w(e)≤w(e_i)。否则，由 w(e)>w(e_i)以及 E(T)中比 e 权值小的边都在 T'中，e_i 同这

些边一起不含有圈，因而，按 Kruskal 算法，e_i将被选到 E(T)中，矛盾。在 T′中去掉边 e_i，换上边 e，得到 G 的一棵新的生成树 T″,这棵树有两个特点：

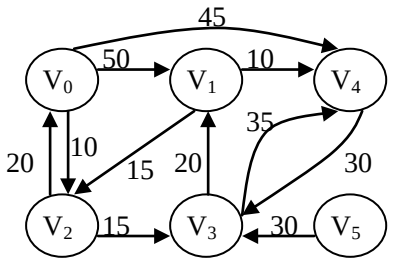
a). T″的权值不大于 T′的权值，因而与 T′有相等的权值；

b).|E(T″)∩E(T)| > |E(T′)∩E(T)|.

a)说明 T″也是一棵最优树，而 b)说明与 T′取法相悖。因此 E(T) = E(T′)，T 是最优生成树。　　证毕

§4 单点源最短路径问题

问题：已知一个赋权有向图 G=(V,E，w),求由 G 中某个指定的顶点 v_0出发到其它各个顶点的最短路径。



路径		长度
(1)	v_0v_2	10
(2)	$v_0v_2v_3$	25
(3)	$v_0v_2v_3v_1$	45
(4)	v_0v_4	45

从 v_0到其它各顶点的最短距离

对于一般的单点源最短路径问题，我们采用逐条构造最短路径的办法， 用

迄今已生成的所有路径长度之和为最小

作为贪心准则，这要求，每一条已被生成的单独路径都必须具有最小长度。假定已经构造了 k 条最短路径,则下面要构造的路径应该是下一条最短长度的最短度路径。现记这 k 条最短路径的终点之集为 S，为陈述方便，也将 v_0放于 S 中。如果 V\S 不是空集，则从 v_0到 V\S 中顶点的最短路径中应该有一条最短的，比如是 v_0到 v_{k+1}的最短路径 P：

$$P=v_0u_1\cdots u_{s-1}u_sv_{k+1} \qquad (4.4.1)$$

显然，$P_1=v_0u_1\cdots u_{s-1}u_s$应是 v_0到 u_s的最短路径，因而由 S 的定义和选定的贪心准则，u_s应属于 S。同理，路径 P 上其它顶点 u_i也都在 S 中。所以，由 v_0出发的新的最短路径一定是某个已有的最短路径向前延伸一步。如果用 Dist(v_i)记从 v_0到 S 中顶点 v_i的最短路径的长度，而图 G 中的顶点 w 依其属于 S 与否分别记之为 S(w)=1 或 S(w)=0，则从 v_0出发，新的最短路径的长度应该是

$$D(S)=\min_{S(u)=1,S(w)=0}\{Dist(u)+COST(u,w)\} \qquad (4.4.2)$$

满足(4.4.2)式的顶点 w 被选择加入 S，新的最短路径就是从 v_0出发到 w 的最短路径,而此时的 Dist(w)=D(S)，S 被更新为 $S'=S\cup\{w\}$,后者可以由更新 w 的（集

合）特征值来实现：S(w)=1(原来 S(w)=0).上述算法思想是 Dijkstra(迪杰斯特)提出的。

程序 4-4-1 Dijkstra 最短路径算法伪代码

```
proc DijkstraPaths(v,COST,n,Dist,Parent)//G 是具有 n 个顶点
   //{1,2,…,n}的有向图， v 是 G 中取定的顶点，COST 是 G 的邻接矩阵,
   //Dist 表示 v 到各点的最短路径之长度，Parent 表示各顶点在最短路径
   //上的前继。
    bool S[1..n]; float COST[1..n,1..n],Dist[1..n];
    integer u,v,n,num,i,w;
1   for i to n do     //将集合 S 初始化为空
2      S[i]:=0; Parent[i]:=v; Dist[i]:=COST[v,i];
3   end{for}
4   S[v]:=1; Dist[v]:=0; Parent[v]:=-1;//首先将节点 v 记入 S
5   for i to n-1 do   //确定由节点 v 出发的 n-1 条最短路
6      选取顶点 w 使得 Dist(w) = min{Dist(u)}
                              S(u)=0
7      S[w]:=1;
8      while S[u]=0 do
         //修改 v 通过 S 到达 S 以外的结点 u 的最小距离值
9        if Dist[u]> Dist[w]+COST[w,u] then
10          Dist[u]:=Dist[w]+COST[w,u];
11          Parent[u]:=w;
12       end{if}
13     end{while}
14  end{for}
15 End{DijkstraPath}
```

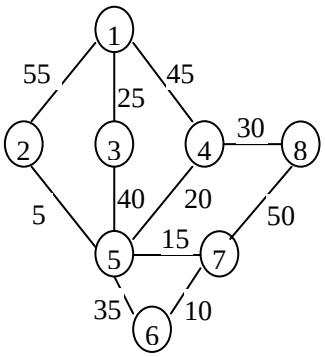
这里需要注意语句 6 和 10。根据(4.4.2)式，被选择的新的最短路径的长度应该等于语句 6 中的 $\min_{S(u)=0}\{Dist(u)\}$，这可由语句 10 中的 Dist 值更新语句看出。假设 S 变成 S′=S∪{w}后，下一次被选中的节点是 u，如果 Dist(u)在 S 变成 S′=S∪{w}后值发生了变化，则必然等于 Dist(w)+ COST(w,u)。事实上，假如在 w,u 的中间还有顶点 p，使得 w 通过 p 到 u 的距离比 COST(w,u)还小，则 v 到 p 的距离必然比 v 到 u 的距离更短，这与 u 被选中矛盾。所以，COST(w,u)是从 w 到 u 的最短路径的长度，进而 Dist(w)+ COST(w,u)是 v 到 u 的最短路径长度。因此 S 变

成 S′=S∪{w}后，S′之外各顶点 Dist 值的更新采用

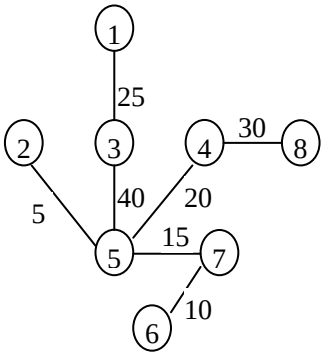
Dist(u)=min{Dist(u),Dist(w)+ COST(w,u)}

必将使下一步选点得到 v 到 S′之外各顶点最短路径之最小的顶点，这保证了算法 DijstraPath 的正确性。对于每个不是 v 的顶点 u，可以通过回溯 Parent(u)得到由 v 到达 u 的最短路径。

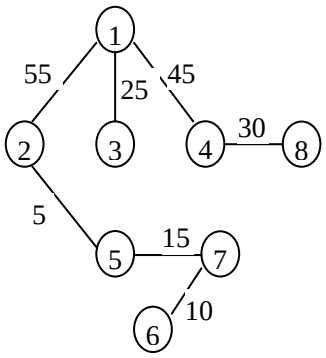
算法的时间复杂度为$O(n^2)$。这是因为，在 for 循环中，求得元素 w 需要做$\Theta(n-i-1)$比较；而在 while 循环中考虑更新 Dist 的值时，也需要$\Theta(n-i-1)$操作，因而总的操作次数应该是$O(n^2)$。



赋权连通图 G



G 的最优生成树



G 的一棵单点源
最短路径生成树

所有由 v 到达各顶点的最短路径并起来构成图 G 的一棵生成树，称为单点源最短路径树。上页的图给出了一个连通赋权图 G 及它的两棵生成树：最优生成树和单点源最短路径树。

§5 Huffman 编码

哈夫曼编码是用于数据文件压缩的一个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来建立一个 0,1 串，以表示各个字符的最优表示方式。下表给出的是具有 100,000 个字符文件中出现的 6 个不同的字符的出现频率统计。

表 4-5-1　字符出现频率表

不同的字符	a	b	c	d	e	f
频率（千次）	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

如果采用定长码，则每个字符至少需用三位，该文件总共需要 300,000 位；

如果采用上面所列变长码，则该文件需用

（45×1＋13×3＋12×3＋16×3＋9×4＋5×4）×1000＝224,000 位，

总码长减少了 25%。

一般的编码都是沿一个方向连续地写下 0,1 字串，比如从左到右、从上到下。解码时也是以同样的方向逐字地搜索。为了使解码唯一，必须使用一个规则。**前缀码**即是一种规则，它要求表示任何一个字符的 0,1 字串都不能是表示另一个字符的 0,1 字串的前部。比如，若用 01 表示字符 a，则表示其它字符的字串不能是具有形式：01…。

可以用一棵二叉树来表示前缀编码。用树叶代表给定的字符，并将给定字符的前缀码看作是从树根到代表该字符的树叶的一条道路。代码中每一位的 0 或 1 分别作为指示某节点到左儿子或右儿子的“路标”。

在一般情况下，若 C 是编码字符集，则表示其最优前缀码的二叉树中恰有 |C| 个叶结点，每个叶结点对应于一个字符。这样的二叉树的每个不是叶的结点恰有两个子结点，因而，该二叉树恰有 |C|-1 个内部结点。给定编码字符集 C 及其频率分布 f，即 C 中任一字符 c 以频率 f(c)在数据文件中出现。C 的一个前缀编码方案对应于一棵二叉树 T。字符 c 在树中的深度记为 $d_T(c)$。$d_T(c)$也是字符 c 的前缀码长。该编码方案的平均码长定义为

$$B(T)=\sum_{c\in C} f(c)d_T(c) \qquad (4.5.1)$$

使平均码长达到最小的前缀编码方案称为 C 的一个最优编码。

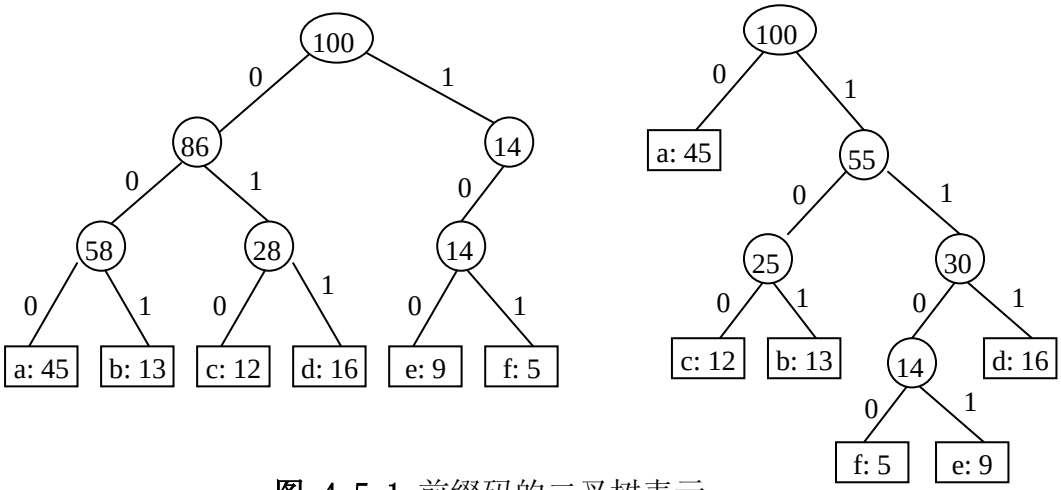


图 4-5-1 前缀码的二叉树表示

哈夫曼提出了一种构造最优前缀编码的贪心算法，称为哈夫曼编码。该算法以自底向上的方式构造表示最优前缀编码的二叉树 T。算法以 |C| 个叶结点开始，执行 |C|-1 次“合并”运算后产生最终所要求的二叉树 T。

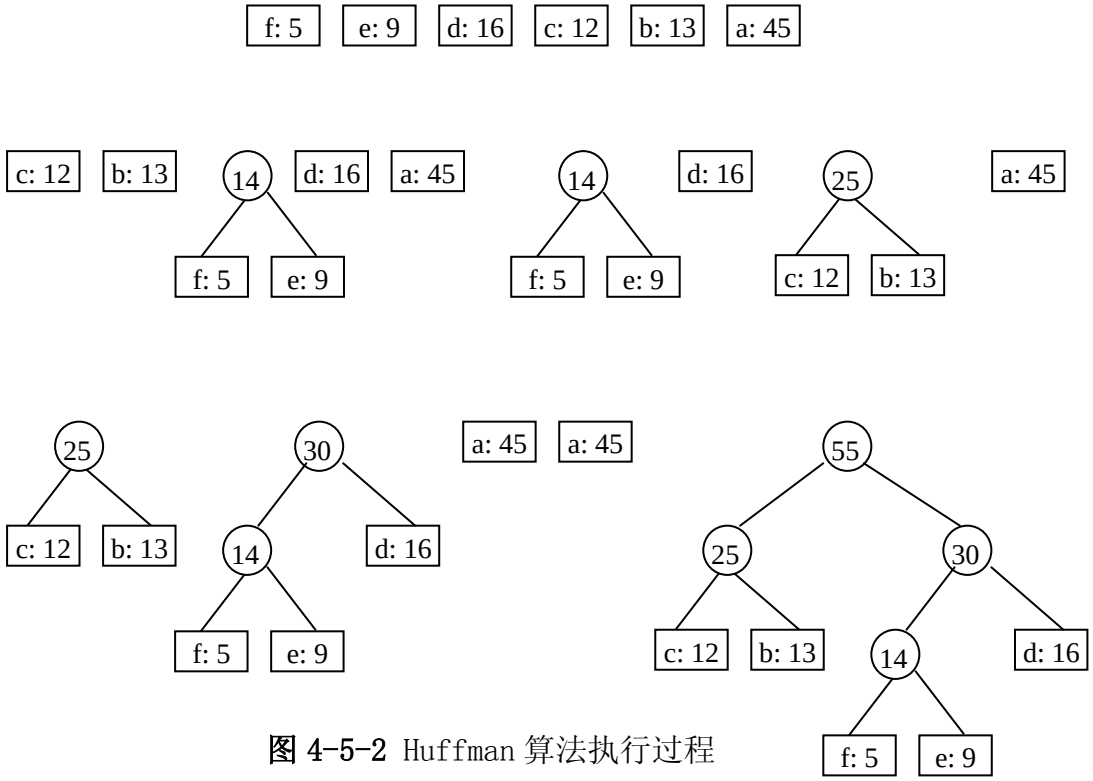


图 4-5-2 Huffman 算法执行过程

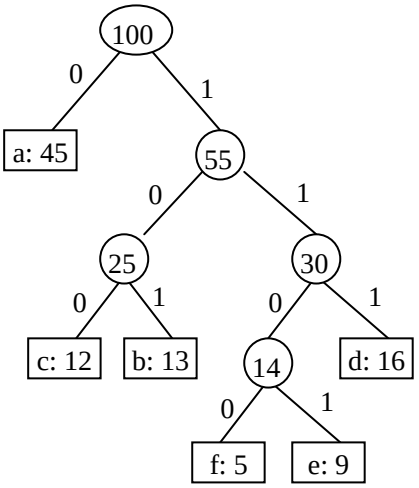


图 4-5-3 Huffman 编码树

Huffman 编码首先用字符集 C 中的每一个字符 c 的频率 f(c)初始化一个优先队列 Q，然后不断地从优先队列 Q 中取出具有最小频率的两棵树 x 和 y，将它们合并成一棵新树 z，z 的频率是 x 与 y 的频率之和。新树 z 以 x 为其左儿子，y 为其右儿子（当然，也可以 y 为左儿子，x 为右儿子）。经过至多 n-1 次合并后（假定 |C| = n），优先队列中只剩下一棵树，即是所求的 Huffman 树。

设计 Huffman 算法可以用最小堆实现优先队列 Q。初始化最小堆需要 $O(n)$ 计

算时间。由于 DeleteMin 和 Insert 只需要$O(\log n)$时间，$n-1$次的合并共需要$O(n\log n)$计算时间。因此，关于n个字符的 Huffman 编码的计算时间为$O(n\log n)$。

定理 4.5.1 Huffman 编码产生最优前缀编码方案。

证明： 设 T 是一棵表示最优前缀编码的二叉树，编码的字符集为 C。我们先证明：对于频率最小的两个字符 x,y，可以将它们调换到最深叶顶点的位置而使新树 T′的平均码长不增加。事实上，假设 b,c 是两个最深的叶顶点，它们是兄弟，具有同样的深度。不妨设 f(b)≤f(c),f(x)≤f(y)。在树 T 中将节点 b 与节点 x 互换，得到一个新树 T′。此时

$$B(T)-B(T') = f(b)d_T(b)+f(x)d_T(x)-f(b)d_T(x)-f(x)d_T(b)$$
$$= (f(b)-f(x))(d_T(b)-d_T(x)) \geq 0$$

如果在树 T′中将节点 y 与节点 c 互换，得到一棵新树 T″，同理可证 B(T′)≥B(T″)，但 T 是最有编码树，所以 B(T″)＝B(T)，说明 T″也是最优编码树。其次，如果令 T″中节点 x,y 的父节点代表一个新的字符 z，出现频率为

$$f(z)=f(x)+f(y),$$

则 T″去掉节点 x,y 后得到一个以{C\{x,y})∪{z}为字符集的最优前缀编码树。对于这棵树叶可以象对待树 T 那样进行调整，使得新树将具有最小频率的两个字符放在最深的叶子节点位置。如此继续下去，最后得到一棵只有一个节点的树，它是只有一个字符，出现率为 100%的最优编码树。以这棵树为根逐步将上述过程中摘掉的叶节点恢复出来，得到的就是 Huffman 树。所以 Huffman 树是最优树，即平均码长最短的树。 证毕

习题 四

1. 设有n个顾客同时等待一项服务。顾客i需要的服务时间为$t_i, 1 \leq i \leq n$。应该如何安排n个顾客的服务次序才能使总的等待时间达到最小？总的等待时间是各顾客等待服务的时间的总和。试给出你的做法的理由（证明）。

2. 字符$a \sim h$出现的频率分布恰好是前 8 个 Fibonacci 数，它们的 Huffman 编码是什么？将结果推广到n个字符的频率分布恰好是前n个 Fibonacci 数的情形。Fibonacci 数的定义为：$F_0=1, F_1=1, F_n=F_{n-2}+F_{n-1}\ if\ n>1$

3. 设$p_1, p_2, \cdots, p_n$是准备存放到长为 L 的磁带上的 n 个程序，程序p_i需要的

带长为a_i。设$\sum_{i=1}^{n} a_i > L$，要求选取一个能放在带上的程序的最大子集合（即其中含有最多个数的程序）Q。构造Q的一种贪心策略是按a_i的非降次序将程序计入集合。

1) 证明这一策略总能找到最大子集Q，使得$\sum_{p_i \in Q} a_i \le L$。

2) 设Q是使用上述贪心算法得到的子集合，磁带的利用率可以小到何种程度？

3) 试说明 1)中提到的设计策略不一定得到使$\sum_{p_i \in Q} a_i / L$取最大值的子集合。

4. 写出 Huffman 编码的伪代码，并编程实现。

5. 已知n种货币$c_1, c_2, \cdots, c_n$和有关兑换率的$n \times n$表R，其中$R[i, j]$是一个单位的货币c_i可以买到的货币c_j的单位数。

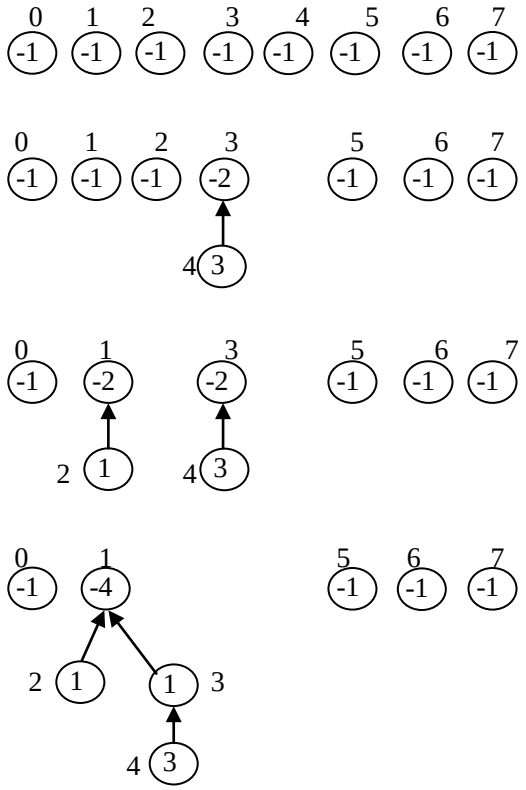
1）试设计一个算法，用以确定是否存在一货币序列$c_{i_1}, c_{i_2}, \cdots, c_{i_k}$使得：

$$R[i_1, i_2]R[i_2, i_3]\cdots R[i_k, i_1] > 1$$

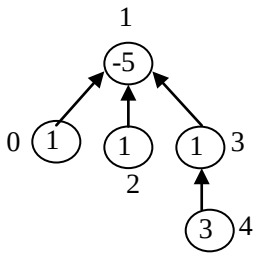
2） 设计一个算法打印出满足 1）中条件的所有序列，并分析算法的计算时间。

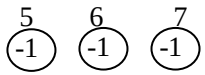
n=7, D=[4,2,4,3,4,8,3] 快速作业调度程序执行过程

作业	F[0]	F[1]	F[2]	F[3]	F[4]	F[5]	F[6]	F[7]	
无	0	1	2	3	4	5	6	7	考虑作业 1
{1}	0	1	2	3	3	5	6	7	考虑作业 2
{1,2}	0	1	1	3	3	5	6	7	考虑作业 3
{1,2,3}	0	1	1	1	3	5	6	7	考虑作业 4



{1,2,3,4}　　0　0　1　　1　　3　5　　6　　7



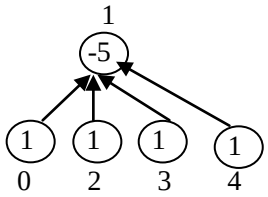


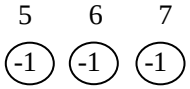
考虑作业 5

Find(min{7,D[5]})=Find(4)=1

F[1]=0,　作业 5 不被选择

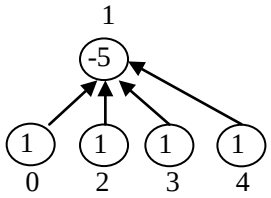
{1,2,3,4}　　0,　0,　1,　1,　3　5　6　　7

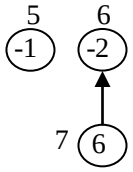




考虑作业 6

{1,2,3,4,6}　0,　0,　1,　1　3　5　6　　6





考虑作业 7

Find(min{7,D[7]})=Find(3)=1

F[1]=0,　作业 7 不被选择

最优解 J={1, 2, 3, 4, 6}