第 5 章 贪心算法

5.1 概　　念

贪心算法与一般算法的不同之处在于它可以快速解决特定问题。贪心算法总是基于当前情况，不考虑整体最优解，只考虑下一步的最优解。所以贪心算法在解决问题时，总是自上而下、逐步迭代，每做一次贪心选择后，问题规模就会变小，直到问题的规模达到显而易见的程度。

虽然每一次贪心选择都能使局部达到最优，但由此产生的全局解决方案并不一定是全局的最优解。但是有时候可以通过各种方法来证明该贪心策略可以达到全局最优，或者达到近似的全局最优。贪心算法的核心问题是选择可以为问题生成最佳解决方案的最佳度量或特定的贪心策略。

5.1.1 贪心算法的基本要素

1. 贪心选择

贪心选择性质，是指通过一系列局部最优选择或贪心选择，可以得到一个问题的整体最优解。这是贪心算法可行的第一个基本要素。在贪心算法中，仅在当前状态下做出最好选择，即局部最优选择。然后再解出这个选择后产生的相应的子问题。贪心算法通常以自顶而下的方式进行，以迭代的方式做出相应的贪心选择，每做一次贪心选择就将所求问题简化为规模更小的子问题。

2. 最优子结构

当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。运用贪心策略在每一次转换时都取得了最优解。

5.1.2 贪心算法的基本思想

用局部解构造全局解，即从问题的某一个局部开始求解并向着总目标进行，以尽可能快地求得更好的解。当某个算法中的某一步不能再继续前进时，算法停止。贪心算法思想的本质就是分治，或者说，分治是贪心的基础。每次都形成局部最优解，即每次都处理出一个最好的方案。

5.1.3 贪心算法的实现过程

从问题的某一初始解出发，循环找到能朝给定总目标前进一步的可行解的一个最优解元素，直到达到总目标。由所有解元素构成的集合就是贪心算法的一个可行解。

5.1.4 适用条件

贪心算法的最终目的不是得到总问题的最优解，但对于某些问题，也可以总能求得整体的最优解，这需要解决特定条件的问题，因此贪心算法不一定能找到解决总问题的最优解。

只要能满足贪心算法的两个性质：贪心选择性质和最优子结构性质，贪心算法就可以求出问题的整体最优解。即使对于某些问题，贪心算法不能求得整体的最优解，也能求出近似整体最优解，如果对结果要求并不高时，贪心算法是一个很好的选择。

5.2 背包问题

5.2.1 问题描述

假设有一个最多能装 M kg 的背包，现在有 n 种物品，每件的重量分别是 $W_1,W_2,\cdots,W_n$，每件物品的价值分别为 $C_1,C_2,\cdots,C_n$，需要将物品放入背包中，要怎样放才能保证背包中物品的总价值最大。即背包问题如何选择装入背包的物品，使得装入背包的物品的总价值最大？但要注意在该背包问题中可以将物品的一部分装入背包。

问题分析如下。

首先对问题进行形式化表示。这里背包容量 $M>0$，每个物品重量 $W_i>0$，物品价值 $C_i>0$。要找出一个 n 元向量$(x_1,x_2,\cdots,x_n)$，其中，x_i 是物品 i 装入背包中的百分比，$0\leqslant x_i\leqslant 1$，$1\leqslant i\leqslant n$，要求：

$$\sum_{i=1}^{n} W_i x_i \leqslant M \tag{5-1}$$

求解目标是：

$$C=\sum_{i=1}^{n} C_i x_i \text{ 最大}$$

如果背包容量大于或等于要装入物品的总量，即 $\sum_{i=1}^{n} W_i \leqslant M$，则将所有物品全部装入背包中为最优解，这时：

$$x_i=1,\quad 1\leqslant i\leqslant n$$

最优值为：

$$C=\sum_{i=1}^{n} C_i \tag{5-2}$$

5.2.2 问题求解

用贪心算法解决背包问题的核心是如何选择贪心策略。下面以一个例子分析贪心策略的选择，考虑以下背包问题。

$$M=20,n=3$$
$$(C_1,C_2,C_3)=(25,24,15)$$
$$(W_1,W_2,W_3)=(18,15,10)$$

贪心策略 1：

考虑使背包中物品价值增长最快，每次选择价值最大的物品优先装入背包。

该策略首先按照物品价值从大到小的顺序进行排序：

$$(C_1,C_2,C_3)=(25,24,15)$$

优先把价值最大的物品放入背包。由于背包容量大于物品 1 重量，物品 1 全部放入背包；背包剩余容量为 2；然后查看排序序列中的下一个物品——物品 2，由于背包剩余容量小于物品 2 的重量，物品 2 只能部分放入背包，放入背包中的比例为 2/15，此时有：

$$x_1=1,\quad x_2=2/15,\quad x_3=0$$

$$C=\sum_{i=1}^{n}C_ix_i=25\times1+24\times2/15+15\times0=28.2$$

贪心策略 2：

考虑使背包剩余容量减少最慢，每次选择重量最轻的物品优先装入背包。

该策略首先按照物品重量从小到大的顺序进行排序：

$$(W_3,W_2,W_1)=(10,15,18)$$

优先把重量最轻的物品放入背包。由于背包容量大于物品 3 重量，物品 3 全部放入背包；背包剩余容量为 10；然后查看排序序列中的下一个物品——物品 2，由于背包剩余容量小于物品 2 的重量，物品 2 只能部分放入背包，放入背包中的比例为 10/15＝2/3，此时有：

$$x_1=0,\quad x_2=2/3,\quad x_3=1$$

$$C=\sum_{i=1}^{n}C_ix_i=25\times0+24\times2/3+15\times1=31$$

贪心策略 3：

考虑将单位价值最大的物品优先装入背包。

该策略首先按照物品单位价值从大到小的顺序进行排序：

$$(C_2/W_2,C_3/W_3,C_1/W_1)=(24/15,14/10,25/18)=(1.6,1.4,1.39)$$

优先把单位价值最大的物品装入背包。由于背包容量大于物品 2 的重量，物品 2 全部放入背包；背包剩余容量为 5；然后查看排序序列中的下一个物品——物品 3，由于背包剩余容量小于物品 3 的重量，物品 3 只能部分放入背包，放入背包中的比例为 5/10＝0.5，此时有：

$$x_1=0,\quad x_2=1,\quad x_3=0.5$$

$$C=\sum_{i=1}^{n}C_ix_i=25\times0+24\times1+15\times0.5=31.5$$

通过以上求解可知，不同贪心策略得到不同的解，贪心策略的选择决定了算法是否能够得到最优解。

5.2.3　算法实现

```
viodKnapsack(double c[ ],double w[ ],double M,int n){
    //C[1…n]存放 n 种物品的价值,W[1…n]存放 n 种物品的重量,
    //M 表示背包所能承受的重量,x[1…n]表示每种物品放进背包的比例
    double x[];                          //x[1…n]表示每种物品放进背包的比例
    按 C[i]/W[i]将序列 c 和 w 从大到小排列
    for(i=1; i<=n; i++) x[i]=0;          //初始化
```

```
        double cu=M; i=1;
        while(W[i]<=cu){
            x[i]=1; cu=cu-w[i]; i++;
        }
        x[i]=cu/w[i];
        printf(w);
        printf(x);
    }
```

5.2.4 算法分析

算法的时间复杂度包括两部分：排序时间和求解时间。排序时间为 $O(n\log_2 n)$，问题求解时间为 $O(n)$，因此算法总的时间复杂度为：

$$O(n\log_2 n) \tag{5-3}$$

对于背包问题选择贪心策略 1 和贪心策略 2 不一定能找到最优解，选择贪心策略 3 一定能找到最优解，下面给予证明。

证明：

设 $\frac{C_1}{W_1}>\frac{C_2}{W_2}>\cdots>\frac{C_n}{W_n}$，$x=(x_1,x_2,\cdots,x_n)$ 是贪心策略 3 产生的解。

如果 $M\geqslant\sum_{i=1}^{n}W_i$，即背包容量大于或等于物品重量总和，此时 $x=(1,1,\cdots,1)$，物品全部装入背包，x 一定是最优解。

如果 $M<\sum_{i=1}^{n}W_i$，则按照贪心策略 3 优先选择单位价值高的物品装入背包，这时一定存在某个正整数 $j(1\leqslant j\leqslant n)$，使得：

$$\begin{cases} x_1=x_2=\cdots=x_{j-1}=1 \\ 0\leqslant x_j\leqslant 1 \\ x_{j+1}=x_{j+2}=\cdots=x_n=0 \\ \sum_{i=1}^{n}W_ix_i=M \end{cases} \tag{5-4}$$

现假设 $x'=(x_1',x_2',\cdots,x_n')$ 是背包问题的最优解。

设存在一个 $k(1\leqslant k\leqslant n)$，对于 $(1\leqslant i\leqslant k-1)$，有 $x_i=x_i'$，$x_k\neq x_k'$

(1) 如果 $x_k>x_k'$：

由于 $\sum_{i=1}^{k-1}W_ix_i=\sum_{i=1}^{k-1}W_ix_i'$，所以

$$\sum_{i=1}^{k}W_ix_i>\sum_{i=1}^{k}W_ix_i' \tag{5-5}$$

而 $\sum_{i=1}^{n}W_ix_i=M$，所以 $\sum_{i=1}^{k}W_ix_i\leqslant M$，$\sum_{i=1}^{k}W_ix_i'<M$，因此 $x_{k+1}',x_{k+2}',\cdots,x_n'$ 必不全为 0。

因此可以增大 x_k' 的值，减少 $x_{k+1}',x_{k+2}',\cdots,x_n'$ 中某些项的值，同时保证 $\sum_{i=1}^{n}W_ix'_i=M$，

得到一个新解 x''。由于新解将 $x'_{k+1},x'_{k+2},\cdots,x'_n$ 中的容量调整到了 x'_k，$\frac{C_k}{W_k}>\frac{C_{k+1}}{W_{k+1}}>\frac{C_{k+2}}{W_{k+2}}>\cdots>\frac{C_n}{W_n}$，因此新解 x'' 优于最优解 x'，这与假设矛盾，因此满足 $x_i=x'_i(1\leqslant i\leqslant k-1)$，$x_k>x'_k$ 的最优解 x' 不存在。

（2）如果 $x_k<x'_k$：

如果 $x_k=0$，按 x_i 定义，一定有 $\sum_{i=1}^{k-1}W_ix_i=M$，又由于当 $1\leqslant i\leqslant k-1$ 时有 $x_i=x'_i$，所以 $\sum_{i=1}^{k-1}W_ix'_i=\sum_{i=1}^{k-1}W_ix_i=M$，则 $x'_k=0$，与假设 $x_k<x'_k$ 矛盾，所以 x_k 必大于 0。

如果 $x_k=1$，由于 x'_i 的取值范围为 0～1，所以这时 $x_k<x'_k$ 也不成立，因此 x_k 必小于 1。

如果 $0<x_k<1$，则必有 $x_1=x_2=\cdots=x_{k-1}=1$，$x_{k+1}=x_{k+2}=\cdots=x_n=0$。所以 $\sum_{i=1}^{k}W_ix_i=M$，如果 $x_k<x'_k$，则有 $\sum_{i=1}^{k}W_ix'_i>M$，背包内物品重量超过背包容量，因此命题也不成立。

因此，$x_k<x'_k$ 不成立。

因此，最优解 x' 不存在，x 即为问题最优解。

5.3　带时限的作业调度问题

5.3.1　问题描述

给定作业 $J_1,J_2,\cdots,J_m$，各作业的处理时间为 $t_1,t_2,\cdots,t_m$，各作业的完工时限为 $d_1,d_2,\cdots,d_m$，作业 J_i 若能在时限 d_i 之前完成，能获得利润 p_i。假设只有一台处理机为这批作业服务，处理机一次只能运行一个作业。

问题是：怎样选择作业子集 J，使 J 中的作业都能在各自的时限内完工，且获得的利润最大？

5.3.2　问题分析

问题是从作业集合中，选择一个子集合 $J=J_{i1},J_{i2},J_{i3},\cdots$，使得集合 J 中的作业都能在各自时限 d_i 内完成，使得收益 $\sum_{J_i\in J}p_i$ 最大，如图 5-1 所示。

设 f_i 是作业 J_i 的完工时刻，作业子集中 J 的作业 J_i 需满足 $f_i\leqslant d_i$。假设所有作业从时刻 0 开始运行，则必有 $d_i\geqslant t_i$。

为了简化问题，只讨论 $t_i=1$ 的情况，这时所有作业的执行时间都只占用一个单位时间，且 d_i 为正整数。

任务集合	J_1	J_2	…	J_m
执行时间	t_1	t_2	…	t_m
完工时限	d_1	d_2	…	d_m
任务收益	p_1	p_2	…	p_m

图 5-1　带时限的作业调度问题

例：$m=4$，$p=(115,70,85,100)$，$d=(2,1,2,1)$。

从时限 d 可知，一台处理机在时限内最多可处理两个作业，其利润表如表 5-1 所示。

表 5-1 作业调度表

序 号	作业子集	作业处理顺序	利 润
1	$\{J_1, J_2\}$	J_2, J_1	185
2	$\{J_1, J_3\}$	J_1, J_3 或 J_3, J_1	200
3	$\{J_1, J_4\}$	J_4, J_1	215
4	$\{J_2, J_3\}$	J_2, J_3	155
5	$\{J_2, J_4\}$	不可调度	
6	$\{J_3, J_4\}$	J_4, J_3	185

不同的作业选择有不同的收益。

5.3.3 以利润最大作为贪心策略

以利润最大作为贪心策略，可以保障利润大的作业优先进入加工队列中。

1. 算法过程

以利润为最优化量，按利润从大到小选择作业，其过程如下。

(1) 初始化集合：$J=\varnothing$，$\sum p_i=0$。

(2) 设子集 J 中已有 $k-1$ 个作业 $J_{i1}, J_{i2}, \cdots, J_{ik-1}$，它们是一个可完工的作业子序列，在剩余的作业中选择利润最大的作业 J_{ik} 加入 J 中。

(3) 判断作业 J_{ik} 加入后，判断 $J_{i1}, J_{i2}, \cdots, J_{ik-1}, J_{ik}$ 是否是一个在时限内可完工的子序列，如果不是，从集合 J 中删除作业 J_{ik}。

(4) 重复步骤(2)和(3)，直至完成所有作业的判断。

在上述步骤中，判断一个作业 J_{ik} 加入后，序列 $J_{i1}, J_{i2}, \cdots, J_{ik-1}, J_{ik}$ 是否是一个在时限内可完工的子序列，是一个难点。

如果用穷举法进行判断，其算法时间复杂度为 $1!+2!+\cdots+n!$，代价较高，可以采用如下方法判断。

将 J 中作业按时限非递减顺序排列，使得：

$$d_{i1} \leqslant d_{i2} \leqslant \cdots \leqslant d_{ik}$$

又因为每个作业需要执行一个单位时间，所以如果 $d_{ij} \geqslant j$ $(1 \leqslant j \leqslant k)$ 成立，则 J 是可完工的作业子序列。

2. 算法实现

```
void Js1( ){
    scanf(p,d,n);            //假设 p[1]>=p[2]>=….>=p[n]
    J[0]=0; D[0]=0;
    J[1]=1; k=1;             //J[i]表示第 i 个加入 J 的作业号,k 为 J 中已有的作业数
    for(i=2; i<=n; i++){
        r=k;
        while(D[J[r]]>D[i] && D[J[r]]!=r) r--;          //测试能否加入作业 i
```

```
            if(D[i]>r){
                for (h=k; h>=r+1; h--) J[r+1]=J[r];        //加入
                J[r+1]=i;
                k++;
            }
        }
    }
```

5.3.4　以最大时限作为贪心策略

以作业最大时限作为贪心策略，可以保障时限大的作业优先进入加工队列中，其过程如下。

(1) 令 $d=\max\limits_{1\leqslant i\leqslant n}d_i$，称 d 为该作业集的最大时限。由于每个作业执行时间为 1，因此最优调度集合 J 中最多包含 d 个作业。

(2) 初始化作业集合 J，此时作业集合 J 为空，如图 5-2 所示。

(3) 获取时限为 d 的作业集合，取集合中利润最大的作业放入时限为 d 的位置，如图 5-3 所示。

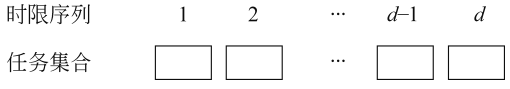

图 5-2　初始时作业集

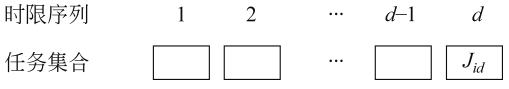

图 5-3　一次操作后作业集

(4) 获取时限大于或等于 $d-1$ 且未放入 J 中的作业集合。如果集合不为空，取集合中利润最大的作业放入时限为 $d-1$ 的位置。如果集合为空，则 $d-1$ 位置无作业可调度，如图 5-4 所示。

(5) 获取时限大于或等于 $d-2$ 且未放入 J 中的作业集合。如果集合不为空，取集合中利润最大的作业放入时限为 $d-2$ 的位置。如果集合为空，则 $d-2$ 位置无作业可调度。

(6) 重复上述过程，直至时限为 1。此时作业集合 J 如图 5-5 所示。

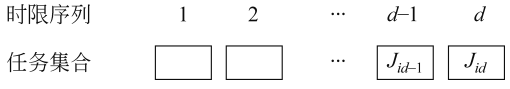

图 5-4　二次操作后作业集

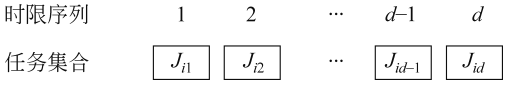

图 5-5　调度后作业集

该作业集合(作业序列)则为问题的解。

算法实现

```
void Js2( ){
    scanf(n,p[1..n], d[1..n]);
    d=max{d[1],d[2],…,d[n]};
    for(i=0; i<=d; i++) s[i]=∅;
    for(i=1; i<=n; i++) s[d[i]]=s[d[i]]∪{Ji};
                                    //将时限为 i 的作业放入集合 s[i]中
    k=0;
    for(i=d; i>=1; i--){
```

```
            if(s[i]≠∅){
                设 j 是 s[i]中利润最大的作业号;
                J[k+1]=j ;
                s[i]=s[i]-{ Ji};          //从时限为 i 的作业集合 s[i]中选入 J 中作业删除
                k++;
            }
            s[i-1]=s[i-1] ∪s[i];          //合并候选作业集
        }
    }
```

5.3.5 快速调度法

设 $b=\min(n,d)$，其中，n 为所有作业数，$d=\max\limits_{1\leqslant i\leqslant n} d_i$。

如果 $b=n<d$，将作业时限大于 b 的时限值设为 b，则任何最大利润的可完工子序列中作业个数必不大于 b，任何可完工的作业子序列中最多只包含一个时限为 1 的作业；最多包含两个时限不大于 2 的作业；……；最多包含 b 个时限不大于 b 的作业。

快速调度法是 5.3.3 节以利润最大作为贪心策略的一种改进贪心算法，以减少数据挪动的次数。

该算法用数 1,2,3,…,b 对应时间区间[0,1],[1,2],…,[$b-1$,b]，按利润 p_i 大小排出非递增序列 S，依次从 S 中取出 p_i，找出[0,1],[1,2],…,[d_{i-1},d_i]中还没有分配给任何作业的最大区间分配给作业 i，这样作业被安排到不能再推迟的那个时间内执行，以保证安排更多的作业。

例如，有 5 个作业 J_1,J_2,J_3,J_4,J_5，其时限分别为(2,2,1,1,5)，每个作业若能在规定的时限内完工，可获得的利润分别是(120,115,70,55,10)。此时 $b=5$，区间如图 5-6 所示。

这 5 个作业正好是按照非递增序列排序的，因此先安排作业 J_1，由于作业 J_1 的时限为 2，把它安排在区间[1,2]，此时如图 5-7 所示。

[0,1]	[1,2]	[2,3]	[3,4]	[4,5]

图 5-6 区间划分

区间	[0,1]	[1,2]	[2,3]	[3,4]	[4,5]
作业		J_1			

图 5-7 一次调度后

接着安排作业 J_2，作业 J_2 的时限为 2，由于区间[1,2]已安排了作业 J_1，所以作业 J_2 只能安排在区间[0,1]。

由于作业 J_3 和 J_4 的时限为 1，应该安排[0,1]，由于该区间已经安排了作业 J_2，所以作业 J_3、J_4 不能按时完工。

作业 J_5 时限为 5，安排在区间[4,5]，安排后的最优作业调度序列如图 5-8 所示。

区间	[0,1]	[1,2]	[2,3]	[3,4]	[4,5]
作业	J_2	J_1			J_5

图 5-8 调度后

算法实现

```
void Js3( ){
    read(n,p,d);                          //p[1]≥p[2]≥…≥p[n]
    b=min{n,max{d[i]}};
    for(i=1; i<=b; i++) J[i]=0;           //i 表示时间区间,0 表示尚未分配出去
    j=1;
    for(i=1; i<=n; i++){
        k=d[i];
        while(k>0)
            if(!J[k]) {J[k]=j; k=0; }
            else k--;
        j++;
    }
}
```

5.4　最佳合并顺序

5.4.1　问题描述

有 n 个有序子序列 $S_1,S_2,\cdots,S_n$，要求通过两两合并的方法把这 n 个子序列合并成一个有序的序列。试设计一个算法确定合并这个序列的最佳合并顺序，使所需的总比较次数最少。

5.4.2　问题分析

合并两个长度分别为 m_1、m_2 的子序列，在最坏情况下需要 m_1+m_2-1 次比较。

对给定的 n 个有序的子序列，若采用两两合并的方法，由于每个子序列的长度不同，不同的合并顺序所用的时间是不同的。

例如：

$$n=3,\quad |S_1|=80,\quad |S_2|=50,\quad |S_3|=40$$

合并方法如下。

(1) $(S_1+S_2)+S_3$，比较次数：(80+50−1)+(130+40−1)=298。

(2) $S_1+(S_2+S_3)$，比较次数：(50+40−1)+(90+80−1)=258。

(3) $(S_1+S_3)+S_2$，比较次数：(80+40−1)+(120+50−1)=288。

我们发现第(2)种合并次序需要的比较次数较少。

通过观察发现，对给定的 n 个序列，共需要 $n-1$ 次合并，要减少合并过程中的比较次数，就应减少中间结果的长度。因此，可以考虑按序列长度由小到大的顺序合并。

5.4.3　问题求解

假设有 5 个序列，这 5 个序列的长度分别为 50,70,80,90,100，图 5-9 表示两种不同合并顺序的二叉树，其中，每个叶子结点表示一个原始序列，每个内结点表示一次合并，如图 5-9 所示。

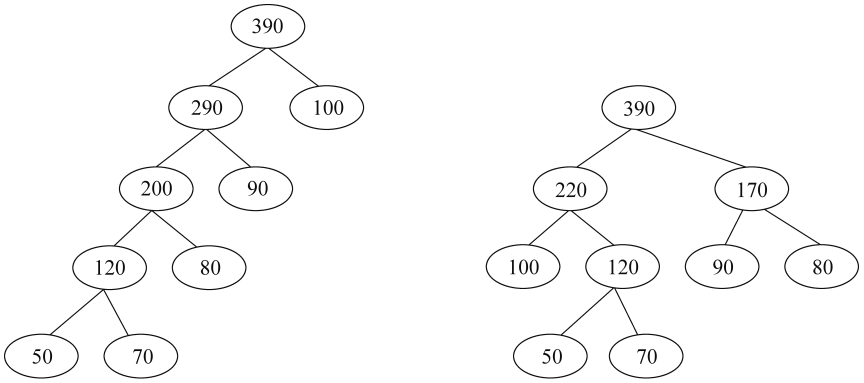

图 5-9　两种不同合并二叉树

二叉树中所有结点的度或为 0，或为 2。表示合并顺序的二叉树中总有 $n-1$ 个内结点，每个内结点表示一次合并，树中表示的总比较次数为各内结点中数值之和减去内结点数。由于内结点数固定为序列的长度减 1，因此可以用内结点中的数值之和表示比较次数。而内结点中的数值正好为各叶子的加权深度。因此，求最佳合并顺序问题等价于求加权深度最小的二叉树。

求加权深度最小的二叉树的过程如下。

(1) 为原始序列中的每一个序列建立一棵二叉树，这些二叉树构成一个树的集合 L，每棵二叉树的根即叶子。WEIGHT(T)是每棵二叉树的权，定义为每个序列的长度。

(2) 从 L 中找出两棵权值最小的树分别作为一棵树的左右儿子，构成一棵新的树，并从 L 中删去左右儿子。新树的权 WEIGHT(T)是它的左右儿子的权之和。

(3) 重复步骤(2)，直到 L 中只有一棵树为止，这棵树表示了合并顺序。

5.4.4　算法实现

```
TreeNode OMTree(int[] w){
    PriorityQueue<TreeNode> queue =
    new PriorityQueue<>(w.length,new Comparator<TreeNode>(){
        public int compare(TreeNode t1,TreeNode t2){
            return t1.val-t2.val;
        }
    });                                    //将所有结点存入优先权队列,按照权值递增排序
    for(int i=0; i<w.length; i++){
        queue.offer(new TreeNode(w[i]));
    }
    while( queue.size()>1 ){                                  //构造二叉树
        TreeNode node1=queue.poll();
        TreeNode node2=queue.poll();                          //弹出最小的两个结点
        TreeNode father=new TreeNode(node1+node2);            //构造父结点
        father.left =node1;
        father.right =node2;
```