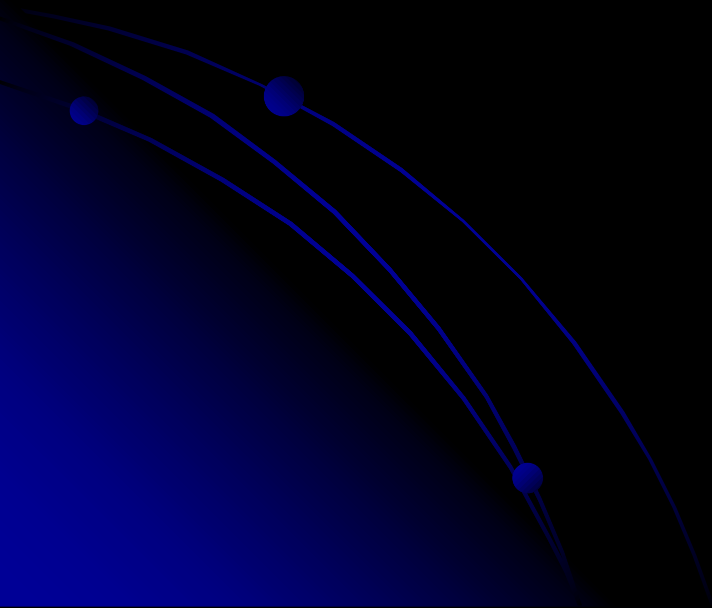
第6章　贪心法



学习要点：

- 理解贪心算法的概念
- 掌握贪心算法的基本要素

  （1）最优子结构性质

  （2）贪心选择性质
- 理解贪心算法的一般方法
- 通过应用范例学习贪心设计策略。

  （1）背包问题;

  （2）最优归并模式;

  （3）最小代价生成树;

- 章节内容

6.1 一般方法

6.2 背包问题

6.4 最优归并模式

6.5 最小代价生成树

6.1 贪心法的一般方法

- 可行解

——问题给定某些约束条件，满足约束条件的问题解，即称为可行解。

- 最优解

——问题给出目标函数衡量可行解的好坏，使目标函数取最大（或最小）值的可行解称为最优解。

贪心法求解最优化问题。

- 贪心法通过分步决策的方法求解问题，每一步决策产生n-元组解（$x_0, x_1, \ldots, x_{n-1}$）的一个分量。贪心法每一步上用作决策依据的选择准则被称为最优量度标准。
- 在选择解分量的过程中，添加新的解分量x_k后，形成的部分解（$x_0, x_1, \ldots, x_k$）不违反可行解约束条件。
- 每一次贪心选择都将所求问题简化为规模更小的子问题。

贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择，仅依赖以前的选择，但不依赖于以后的选择。

对于一个贪心算法，必须证明该算法的每一步上作出的选择，都必然最终导致问题的一个整体最优解。

贪心算法不能保证对所有问题都得到整体最优解。

对许多问题，如：一般背包问题、最佳合并模式问题、单源最短路径问题，最小生成树问题等，贪心算法确实能产生整体最优解。

一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

能用贪心算法求解的问题一般具有两个性质：**贪心选择性质**和**最优子结构性质**。

1、贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。

这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。

对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

2、最优子结构性质

一个问题的最优解包含其子问题的最优解，则称此问题具有最优子结构性质。

问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

程序6-1 贪心法算法框架

```
SolutionType Greedy(SType a[],int n)
{
    SolutionType solution=Ø;  //初始时，解向量不包含任何分量
    for (int i=0;i<n;i++){
        SType x=Select(a);    //问题的解用n元组(x0,x1,…,xn-1)表示
                              //遵循最优度量标准选择一个分量x
        if (Feasiable(solution,x))  //判断加入新分量x后部分解是否可行
            solution=Union(solution,x);  //形成新的部分解
    }
    return solution;          //返回生成的最优解
}
```

6.2 背包问题

给定n种物品和一个容量为M的背包。物品i的重量是w_i，其价值为p_i。应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

- 物品不能分割：在选择装入背包的物品时，对每种物品i只有2种选择，即装入背包或不装入背包。不能将物品i装入背包多次，也不能只装入部分的物品 —— 0/1背包问题
- 物品是可分割的。选择物品i装入背包时，可以选择物品i的一部分x_i（$0 \leqslant x_i \leqslant 1$）装入，而不一定要全部装入背包 —— 一般背包问题，简称背包问题

这2类问题都具有最优子结构性质，极为相似。

●但一般背包问题可以用贪心算法求解；

●而0-1背包问题却不能用贪心算法求解，而只能得到它的近似解。

一般背包问题用贪心选择可保证背包刚好能装满。
而对于0-1背包问题，贪心选择无法保证最终能将背包装满。部分闲置的背包空间使每公斤背包空间的价值降低了，因此不能得到最优解。

事实上，在考虑0-1背包问题时，应比较选择该物品和不选择该物品时所导致的最终方案，然后再作出最好选择。

由此导出许多互相重叠的子问题，这正是该问题可用动态规划算法求解的重要特征。

实际上，动态规划算法的确可以有效求解0-1背包问题。

贪心法求解背包问题

背包问题的解可表示成一个n-元组:$X=(x_0,x_1,\ldots x_{n-1})$, $0\le x_i\le 1$，$0\le i<n$. x_i为第i件物品装入背包中的那部分。

- 可行解的约束条件:

$$\sum_{i=0}^{n-1} w_i x_i \le M \quad wi>0,\ 0\le xi\le 1,\ 0\le i<n$$

- 最优解的目标函数:

(见P104　例6-1　表6-1)

$$\max\sum_{i=0}^{n-1} p_i x_i \quad pi>0,\ 0\le xi\le 1,\ 0\le i<n$$

用贪心法求解, 找出**最优量度标准**是至关重要的.

选择最优量度标准

- 标准1
  - 选取目标函数(总价值)作为量度标准,每次取价值最大的物品装包,不考虑重量.
  - 得到近似解,而不是最优解.
  - 原因:只考虑当前最大收益,背包载重消耗太快.
- 标准2
  - 选取重量作为量度标准,每次取重量最小的物体装包,不考虑收益.
  - 得到近似解,而不是最优解.
- 标准3
  - 选取单位重量价值最大的物品装包,即每次选p_i/w_i最大的物品装包.
  - 标准最合理,得到最优解.(正确性有待证明)

背包问题的贪心算法

基本步骤：

1、首先计算每种物品单位重量的价值P_i/W_i并按非增次序进行排序；

2、然后依贪心选择策略，选择单位重量价值最高的物品装入背包。依此策略一直地进行下去，将尽可能多的物品全部装入背包，直到将背包装满。

3、若装入某件物品时，不能全部装下，而背包内的物品总重量仍未达到W，则根据背包的剩余载重，选择单位重量价值次高的物品并尽可能多地装入背包。

具体算法可描述如下页：

数组w存放物品的重量，数组p存放物品的价值，数组x存放背包问题的最优解。m为背包载重量，u为背包剩余载重量。

程序6-2　背包问题的贪心算法（最优度量标准：单位重量价值p_i/w_i最大）

```
void GreedyKnapsack(float *x)
{  //前置条件：w[i]已按p[i]/w[i]的非增次序排序
    float u=m;                          //将背包剩余载重量u初始化为m
    for (int i=0;i<n;i++) x[i]=0;       //对解向量x初始化
    for (i=0;i<n;i++)       {           //按最优量度标准选择解分量xi
          if (w[i]>u) break;            //若当前物品i已无法全部装下，则跳出
```

如果不计按p[i]/w[i]的非增次序排列w[i]的时间，程序6-2的时间复杂度为O(n)。
算法Greedyknapsack的主要计算时间在于将各种物品按单位重量价值p[i]/w[i]从大到小排序，为O(nlogn)。
因此，算法的计算时间上界为O(nlogn）。
为了证明算法的正确性，还必须证明背包问题具有贪心选择性质。

最优解证明

定理6-1：如果$p_0/w_0 \geq p_1/w_1 \geq \ldots \geq p_{n-1}/w_{n-1}$,则程序6-2求得的背包问题的解是最优解。

证明：

设$X=(x_0,x_1,\ldots,x_{n-1})$,$0 \leq x_i \leq 1$,$0 \leq i < n$是贪心背包算法的解。则由贪心背包算法可知，解的形式一定为

$$X=(1,\ldots,1,x_j,0,\ldots,0), \quad 0 \leq x_j < 1$$

如果X不是最优解，另有可行解$Y=(y_0,y_1,\ldots,y_k,\ldots,y_{n-1})$是最优解，则应有：

$$\sum_{i=0}^{n-1} p_i y_i > \sum_{i=0}^{n-1} p_i x_i$$

设k是使得$y_k \neq x_k$的最小下标，必有$y_k < x_k$ (见P106)。

假定以x_k替换Y中的y_k，得到新的解$Z=(z_0,\ldots, z_{k-1}, z_k, z_{k+1},\ldots, z_{n-1})$。可知：替换前后 X、Y、Z的前k个分量相等$z_i=y_i=x_i=1$，$0 \le i \le k-1$，但替换后$z_k=x_k$。

为保证Z是可行解，应有：$\sum_{i=k+1}^{n-1} w_i(y_i - z_i) = w_k(z_k - y_k)$ 。

则：$$\sum_{i=0}^{n-1} p_i z_i = \sum_{i=0}^{n-1} p_i y_i - \sum_{i=0}^{n-1} (y_i - z_i)(w_i / w_i) p_i$$

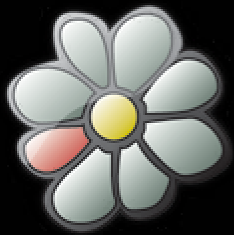
$$= \sum_{i=0}^{n-1} p_i y_i - (y_k - z_k)(w_k / w_k) p_k - \sum_{i=k+1}^{n-1} (y_i - z_i)(w_i / w_i) p_i$$

$$\ge \sum_{i=0}^{n-1} p_i y_i - (y_k - z_k) w_k (p_k / w_k) - \sum_{i=k+1}^{n-1} (y_i - z_i) w_i (p_k / w_k)$$

$$= \sum_{i=0}^{n-1} p_i y_i$$ （新的可行解Z的收益≥假定的最优解Y的收益）

因此假设不成立，证得贪心背包算法的解X即为最优解！

（重复这样的替换过程，最终可得到一个与X完全相等的解。）



课堂练习题：找换硬币

考虑用最少的硬币数来找n分钱的问题，假设每个硬币的值都是整数。

1）请给出一个贪心算法，使得所换硬币包括1角的，5分的， 2角5分的和1分的？

2）假设可换的硬币的单位是c的幂，也就是$c^0,c^1,\ldots,c^k$,其中整数$c>1,k\geq 1$。证明贪心算法总可以产生一个最优解。

3）对任意给定的硬币单位集合，用贪心法来求解，是否总能保证得到最优解（即：硬币数最少的最优硬币组合方案）？所给集合应当包括1分，以保证对任意n值均有解。

2）证明：（反证）

假设贪心法所得的解$X=(x_{n-1},x_{n-2},...,x_0)$并非最优解，而是另有$Y=(y_{n-1},y_{n-2},...,y_0)$为最优解。

则一定有：

$$\sum_{i=n-1}^{0} x_i c^i = \sum_{i=n-1}^{0} y_i c^i = M \qquad (1)$$

且易知：最优解中（除面值为C^{n-1}的硬币以外，其他）任何一种硬币的个数y_i一定小于C。

因为若某一种面值为C^{i-1}的硬币个数C，则将C个面值为C^{i-1}的硬币用一个C^i面值的硬币来取代，可获得比最优解中的硬币数更少的硬币数，即获得更优解。

证明（续）：因此

$$\sum_{i=k-1}^{0} y_i c^i < \sum_{i=k-1}^{0} (c-1)c^i = (c-1)\frac{c^k - 1}{c-1} = c^k - 1 < c^k \qquad (2)$$

即所有面值小于C^k的硬币面值之和，一定小于C^k。

从前向后依次比较X和Y中的解分量，若首个不相等的解分量下标为k，则定有$x_k > y_k$（因为在贪心法求解时，x_k已是当前情况下面值为C^k的硬币所能取的最大个数）。

又因为x_k和y_k均为整数，因此$x_k - y_k \geq 1$。

证明（续）：

由于（1）式中下标n-1~k+1的分量均相等，因此可将（1）式简化为：

$$\sum_{i=k}^{0} x_i c^i = \sum_{i=k}^{0} y_i c^i$$

即：$$\sum_{i=k-1}^{0} y_i c^i = \sum_{i=k-1}^{0} x_i c^i + (x_k - y_k)c^k > c^k \qquad (3)$$

显然（2）式和（3）式产生矛盾，因此假设不成立，贪心法求得的解就是最优解。得证！

3）对任意给定的硬币单位集合，用贪心法来求解，是否总能保证得到最优解（即：硬币数最少的最优硬币组合方案）？所给集合应当包括1分，以保证对任意n值均有解。

不能。

例如：硬币单位集合为（4,3,1），n=10。

贪心法求得的解为（2,0,2），总共需要4枚硬币。而最优解为（1,2,0），总共需要3枚硬币。

补充：
6.3 带时限的(单位时间)作业排序

有n个作业，每个作业都有一个截止期限$d_i>0$（d_i为整数）。每个作业运行时间为1个单位时间。每个作业若能够在截止期限内完成，可获得$p_i>0$的收益。

要求：

得到一种作业调度方案，给出作业的一个子集和该子集的一种排列，使子集中的作业都能如期完成，并且获得最大的收益。

最优量度标准：

按**收益的非增次序**选择作业，在**不违反截止时限**的前提下，加入部分解向量中。

程序6-3 带时限（单位时间）作业排序的贪心算法

```
void GreedyJob(int d[], Set X, int n)
{    //前置条件：p0≥p1 ≥ ......≥pn-1
     X={0}                  //将作业0选入X
     for (int j=1;j<n;j++)
        if (集合X∪{ j}中作业都能在给定的时限内完成)
              X=X∪{ j };    //可行性判定
}
```

问题一：如何证明求得的一定是最优解？

定理6-2 证明该贪心算法对于带时限（单位时间）作业排序问题将得到最优解。

证明：

设$X=(x_0,x_1,\ldots,x_k)$是贪心算法求得的解。并且X不是最优解，另有$Y=(y_0,y_1,\ldots,y_r)$是最优解。

则必有$X \neq Y$。

- 若X=Y无需证明。
- 若$X \subset Y$，则Y不是可行解，否则贪心法会将属于Y但不属于X的其他作业继续加入X。
- 若$Y \subset X$，则Y不是最优解，矛盾！

定理6-2 证明该贪心算法对于带时限（单位时间）作业排序问题将得到最优解。

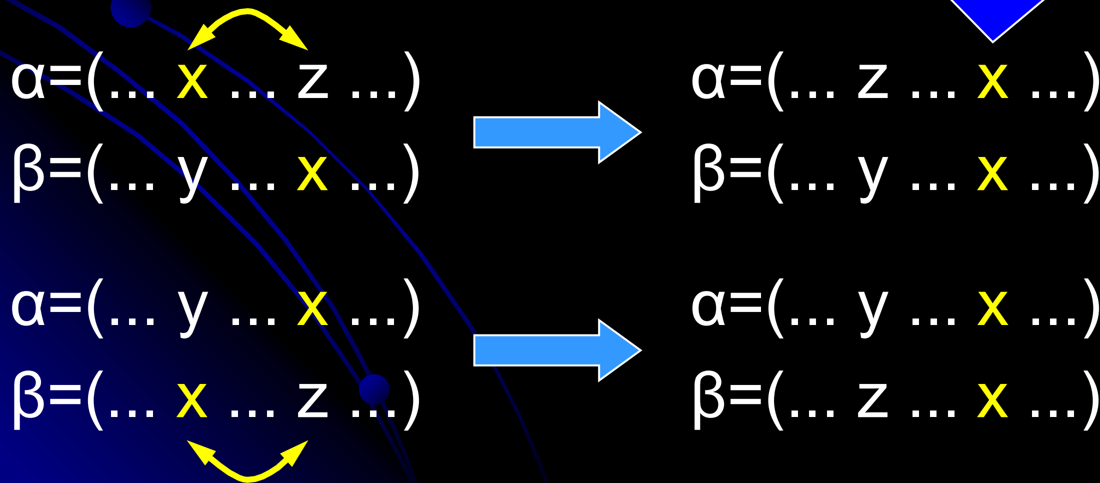
证明：

设$X=(x_0,x_1,...,x_k)$是贪心算法求得的解。并且X不是最优解，另有$Y=(y_0,y_1,...,y_r)$是最优解。

则必有X≠Y。

设α和β分别是X和Y的可行作业 … β的次序调度作业执行）。采用下述 … 业交换到相同的位置上：

同时，这种交换不会造成任何作业超限，因而不影响这两个排列的可行解性质。



将两者共有的作业向后移动，从而使相同作业在两种次序中处于相同的位置，因而在相同的时刻被调度。

证明（续）：

由于$X \not\subset Y$且$Y \not\subset X$，因此必定存在两个作业$a \in X$且$a \notin Y$，$b \in Y$且$b \notin X$。

进一步假定作业a是使得$a \in X$且$a \notin Y$的一个收益最大的作业。

而作业b在β中的位置，与a在α中的位置相同。

由贪心法可知：一定有$p_a \geq p_b$，否则作业b将被加入X中。

现用a取代序列β中的b，得到一个新序列，显然新序列仍然是可行的，且收益不低于Y。

与假设（Y是最优解）矛盾。贪心法求得的解一定是最优解得证。

程序6-3 带时限（单位时间）作业排序的贪心算法

```
void GreedyJob(int d[], Set X, int n)
{     //前置条件：p0≥p1 ≥ ……≥pn-1
      X={0}                    //将作业0选入X
      for (int j=1;j<n;j++)
          if (集合X∪{ j}中作业都能在给定的时限内完成)
                X=X∪{ j };    //可行性判定
}
```

问题二：可行性判定（判断作业j是否允许添加到部分解向量中）如何做？

定理6-3 $X=(x_0,x_1,\ldots,x_k)$是k个作业的集合，$\alpha=(\alpha_0, \alpha_1, \ldots, \alpha_k)$是X的一种特定排列，它使得$d\alpha_0 \le d\alpha_1 \le \ldots \le d\alpha_k$，其中$d\alpha_j$是作业$\alpha_j$的时限。X是一个可行解当且仅当X中的作业能够按α次序调度而不会有作业超期。

判断作业j是否允许添加到部分解向量中的具体做法：

设算法现考察作业j，x=(x[0],x[1],…,x[k])为当前已入选的作业向量，且d[x[0]] ≤d[x[1]]≤…≤d[x[k]]。

由于当前的部分解是可行的，故必有d[x[i]]≥i+1，0≤i≤k。

将作业j按时限的非减次序插入解向量中的某个位置，使得插入作业j后，由k+1个解分量组成的部分解向量仍按时限的非减次序排列。

不妨设作业j插入到下标r+1处，为了完成本次插入，作业x[r+1],…,x[k]在向量中的位置都必须依次后移一位，形成一个新的部分解分量。

为了保证添加作业j后的作业子集仍构成可行解，必须满足下列两点要求：

（1）d[x[i]]>i+1，r+1≤i≤k，否则作业x[r+1],…,x[k]的后移将导致其中某些作业超期；

（2）d[j]>r+1，否则作业j自己无法在时刻r+2前完成。

上述思想用于实现带时限的作业排序程序：

程序6-4 带时限的（单位时间）作业排序程序

```
int JS(int *d, int *x, int n)
{    //设p0≥p1 ≥... ≥pn-1
     int k=0;        //(x[0],...,x[k])是当前已入选的作业向量
     x[0]=0;
     for (int j=1;j<n;j++)
     {
       while( r>=0 && d[x[r]]>d[j]  && d[x[r]]>r+1) r--;  //搜索作业j的插入位置
       if ((r<0 || d[x[r]]<=d[j]) && d[j]>r+1)   //若条件不满足，选择下一个作业
       {   //已选作业中位置r+1至k的作业均可后移
           //且作业j自身也能在时限内完成
           for (int i=k; i>=r+1; i--)   x[i+1]=x[i];      //将x[r]以后的作业后移
           x[r+1]=j;            //作业j插入r+1位置处
           k++;
        }
     }
     return k;
}
```

该条件不满足说明是因为d[x[r]]<=r+1造成的，原作业r无法后移

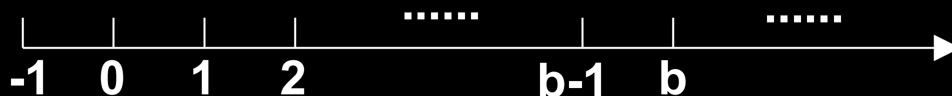
该条件不满足说明新作业j无法后移

时间复杂度：$O(n^2)$

改进方法：

按时间片调度作业，令$b=\{ n,\max\{d_i|0\le i\le n-1\} \}$为可行的作业调度方案所需的最大时间。它的值不会超过作业的最大时限，也不会超过作业总数n。

将b分成b个时间片，每个时间片为一个单位时间。



按时间片调度作业

具体做法（尽量推迟一个作业的执

●作业仍按收益的非增次序排列。

●为收益最大的作业0分配时间片

●为收益次大的作业1分配作业时

]，如果该时间片已分配，再考虑

依次向前寻找第一个空闲的时间片分配之。如果d_1之前的所有时间片均已分配，则作业1应该舍弃。

为了加快时间片的分配速度，可以将作业的时限值划分成为若干个子集，用并查集来实现。

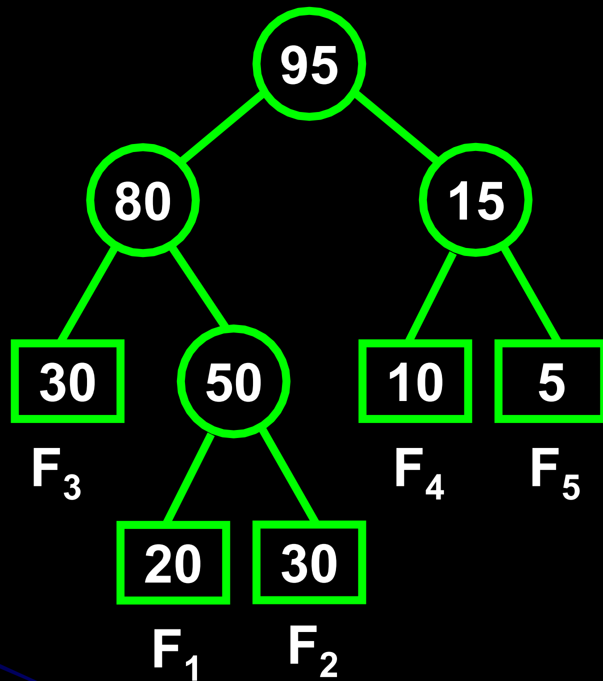
算法时间约为O(nlogn)。

6.4 最优归并模式

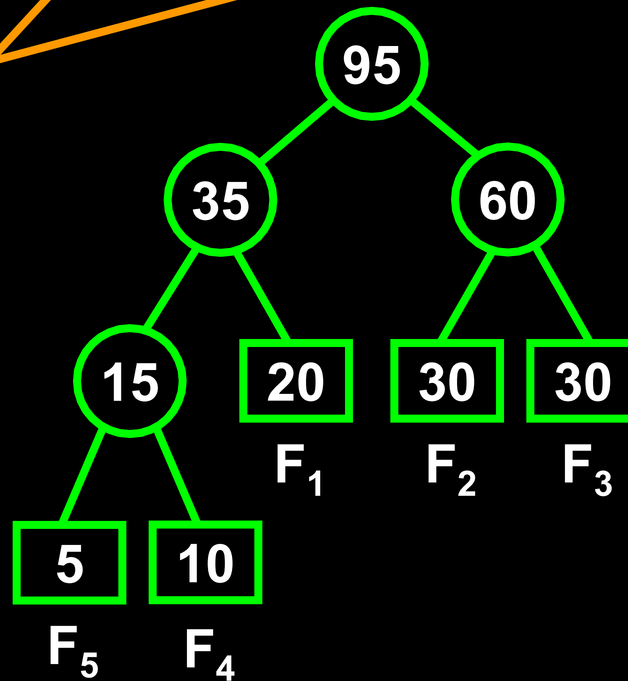
例6-4：将5个长度分别为(20,30,30,10,5)的有序文件(F_1,F_2,F_3,F_4,F_5)两两合并成一个有序文件。

- **两路合并外排序**：通过反复执行将两个有序子序列合并成一个有序文件的操作，将n个长度不等的有序子文件合并成一个有序文件。
- 整个合并过程中需要读/写记录数最少的合并方案为**最佳合并模式**。
- 每一种合并模式可以用**合并树**描述：

扩充二叉树：除叶子结点外，其余结点都必须有2个孩子。



(a) 一种合并模式



(b) 最佳合并模式

扩充二叉树的**带权外路径长度**：WPL= $\sum_{k=1}^{m} w_k l_k$

其中：m是叶子结点的个数，w_k是第k个叶子结点的权，l_k是从根到该叶子结点的路径长度。

两路合并最佳模式问题的贪心算法：

（最优度量标准——带权外路径长度最小）

- 设（$w_0,w_1,\ldots,w_{n-1}$)是n个有序文件的长度，以每个文件的长度作为根结点的权值，构造n个只有根的二叉树；
- 选择两棵根结点权值最小的树，作为左右子树构造一棵新二叉树，新树根的权值是两棵子树根的权值之和；
- 重复此过程，直到合并为一棵二叉树为止。

生成的两路合并树正是代表两路合并最佳模式的二叉合并树。

具体实现见下页：程序6-6

- 类BTNode<T>为二叉合并树结点；
- 优先权队列pq中的元素为HNode类的对象，包含两个数据成员：指针ptr（指向二叉树的根）和weight（存放该根的权值）；
- Append向优先权队列中添加新元素;
- Serve从优先权队列中取出具有最高优先权（即权值最小）的元素。

程序6-6　两路合并最佳模式的贪心算法

```
PrioQueue<HNode<T>> pq(2*n-1);          BTNode<T> *p;   HNode<T> a,b;
for (int i=0;i<n;i++){          //构造n棵只有根的二叉树
        p=new BTNode<T>(w[i]);              //新建一个二叉树结点
        a.ptr=p; a.weight=w[i];     //合并树对象a包含指向根(p)的指针和根(p)的权值
        pq.Append(a);    //将合并树对象a中指向根的指针和根的权值加入队列pq
}
for (i=1;i<n;i++){              //两两合并n-1次，将n棵树合并成一棵树
        pq.Serve(a);        pq.Serve(b);     //从pq中取出优先权最高(权值最小)的两棵树
        a.weight=a.weight+b.weight;
        p=new BTNode<T>(a.weight,a.ptr,b.ptr); //合并这两棵树，构造一棵新二叉树
        a.ptr=p;                                    //对象a中的指针重新指向新二叉树的根
        pq.Append(a);    //将合并树对象a中指向新根的指针和根的权值加入队列pq
}
pq.Serve(a);                    //取出生成的最佳合并树
return a.ptr;                   //a.ptr指向最佳合并树的根
```

算法正确性证明

定理6-4：设有n个权值$W=\{w_0,w_1,\ldots,w_{n-1}\}$作为外结点的权值，构造两路合并树的贪心算法将生成一棵具有最小带权外路径长度的二叉树。

证明：（归纳法）

假设：外结点数目k<n时,算法能生成有k个外结点的最佳两路合并树。证明k=n时贪心算法生成的两路合并树为最佳两路合并树：

若：k=n时,由权值$w_0 \le w_1 \le \ldots \le w_{n-1}$按贪心算法生成的两路合并树为$T_n$。假设$T_n$不是最优的，而另一棵两路合并树$T_n'$是最优的，即：$cost(T_n') < cost(T_n)$。

对树T_n'作调整：

将离根最远的内结点p的两个孩子w_i和w_j，与权值最小的两个外结点w_0和w_1交换，得到另一棵二叉合并树T_n''。必有$cost(T_n'') \le cost(T_n')$。

对T_n''和T_n，都用权值为w_0+w_1的外结点取代它们的根p。

（见图6-8）得到有n-1个外结点的两路合并树T_{n-1}''和T_{n-1}。

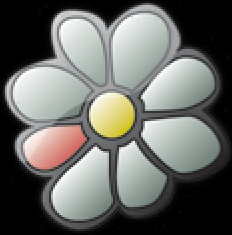
根据归纳法假设：$cost(T_{n-1}) \le cost(T_{n-1}'')$。

又因为：$cost(T_n)=cost(T_{n-1})+w_0+w_1$

和 $cost(T_n'')=cost(T_{n-1}'')+w_0+w_1$，

因此$\Rightarrow cost(T_n) \le cost(T_n'')$。

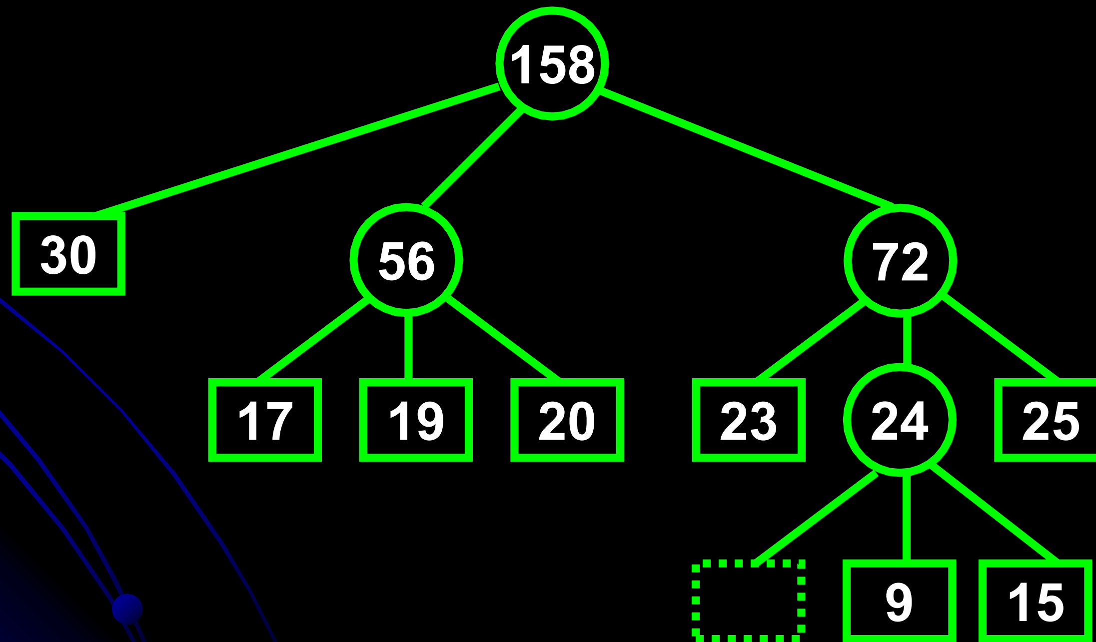
与假设矛盾，贪心法生成的两路合并树必是最佳两路合并树！



课堂练习题

画出W={9, 15, 17, 19, 20, 23, 25, 30}的三路最佳合并树。

由于n=8（权值的个数），k=3（三路合并）， n-1=7不是k-1=2 的整数倍时需补充（k-1)-(n-1)%(k-1)=1个零权值（虚结点）。



6.5 最小代价生成树

设G =(V,E)是无向连通带权图，即一个**网络**。E中每条边(v,w)的权为c[v][w]。

若极小连通子图G’包括图G中的所有顶点，并有尽可能少的边，则称G’为G的生成树。

生成树上各边的权值代表相应的代价。树中各条边的代价总和是生成树的代价。

图的生成树不唯一，采用不同的遍历方法，从不同的结点出发可得到不同的生成树。在G的所有生成树中，代价最小的生成树称为G的**最小生成树**。

网络的最小生成树在实际中有广泛应用。例如：在设计通信网络时，用图的顶点表示城市，用边(v,w)的权c[v][w]表示建立城市v和城市w之间的通信线路所需的费用，则最小生成树就给出了建立通信网络的最经济的方案。

一般情况下，用贪心法得到的是近似最优解，而不能保证得到最优解。但用贪心法计算最小生成树，却可以设计出保证得到最优解的构造最小生成树的有效算法。

本节介绍的构造最小生成树的Prim算法和Kruskal算法都可以看作是应用贪心算法设计策略的例子。尽管这2个算法做贪心选择的方式不同，它们都利用了下面的最小生成树性质：

定理6-5：设G=(V,E)是连通带权图，U是V的一个真子集。如果(u,v)∈E是所有u∈U，v∈V-U的边中权值c[u][v]最小者，那么一定存在G的一棵最小代价生成树T=(V,S),以(u,v)为其中一条边。这一性质也称为MST性质。

程序6-7　贪心法求带权无向图的最小代价生成树的粗略算法

```
ESetType SpanningTree(ESetType E,int n)
{          //G=(V,E)为无向图，E是图G的边集，n是图中结点数
           ESetType S=Ø;          //S为生成树上边的集合
           while (k<n-1 && E中尚有未检查的边){
                      e=select(E);     //按最优度量标准选择一条边
                      if (S∪e
                                 k++;     }          //k为生成树边集S中边的条数
           }
           return S;
}
```

若所有的边都考察完毕后，S集合中的边数k仍然小于n-1，就表明原图G不是连通图。

最简单的最优度量标准：选择使得迄今为止已入选S中的边代价之和增量最小的边。

Kruskal算法

Kruskal在1956年提出的最小代价生成树算法。这种算法对边数较少的带权图有较高效率。它的基本思想是：

- 将图G中的边按其权值由小到大排序。
- 然后作如下的贪心选择：

  在E中选取一条权值最小的边e=(u,v)，并将其从E中删除；

  - 若在S中加入边(u,v)后不形成回路，则将该边加入生成树S中(这要求u、v分属于两棵不同的子树）；
  - 若在S中加入边(u,v)后会形成回路，则舍弃该边，以后也不再考虑。

- 如此依次进行，直到选够（n-1）条边即得到最小生成树。

设G=(V,E)是一个连通带权图，V={1,2,...,n}。构造G的一棵最小生成树F=(U,S)的Kruskal算法的基本步骤是：

(S为正在构造的生成树边集)

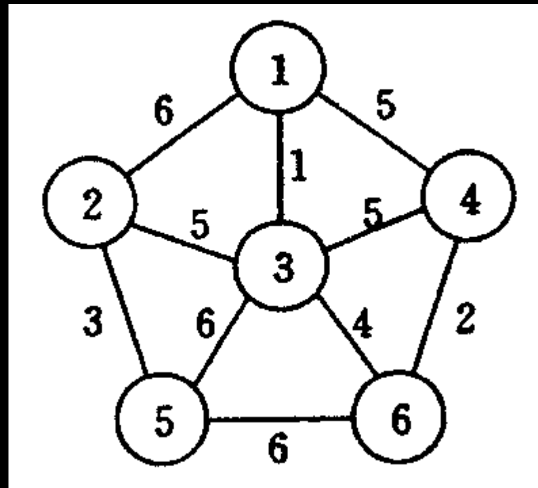
1、首先将G的n个顶点看成n个孤立的连通分支。

2、将所有的边按权从小到大排序。然后从第一条边开始，依边权递增的顺序查看每一条边，并按下述方法连接2个不同的连通分支：当查看到第k条边(u,v)时，

- 如果端点u和v分别是当前2个不同的连通分支T1和T2中的顶点时，就用边(u,v)将T1和T2连接成一个连通分支，然后继续查看第k+1条边；
- 如果端点u和v在当前的同一个连通分支中，就直接再查看第k+1条边。

3、这个过程一直进行到只剩下一个连通分支时为止，此时便是G的一棵最小代价生成树。

例如：对于下图中的连通带权图



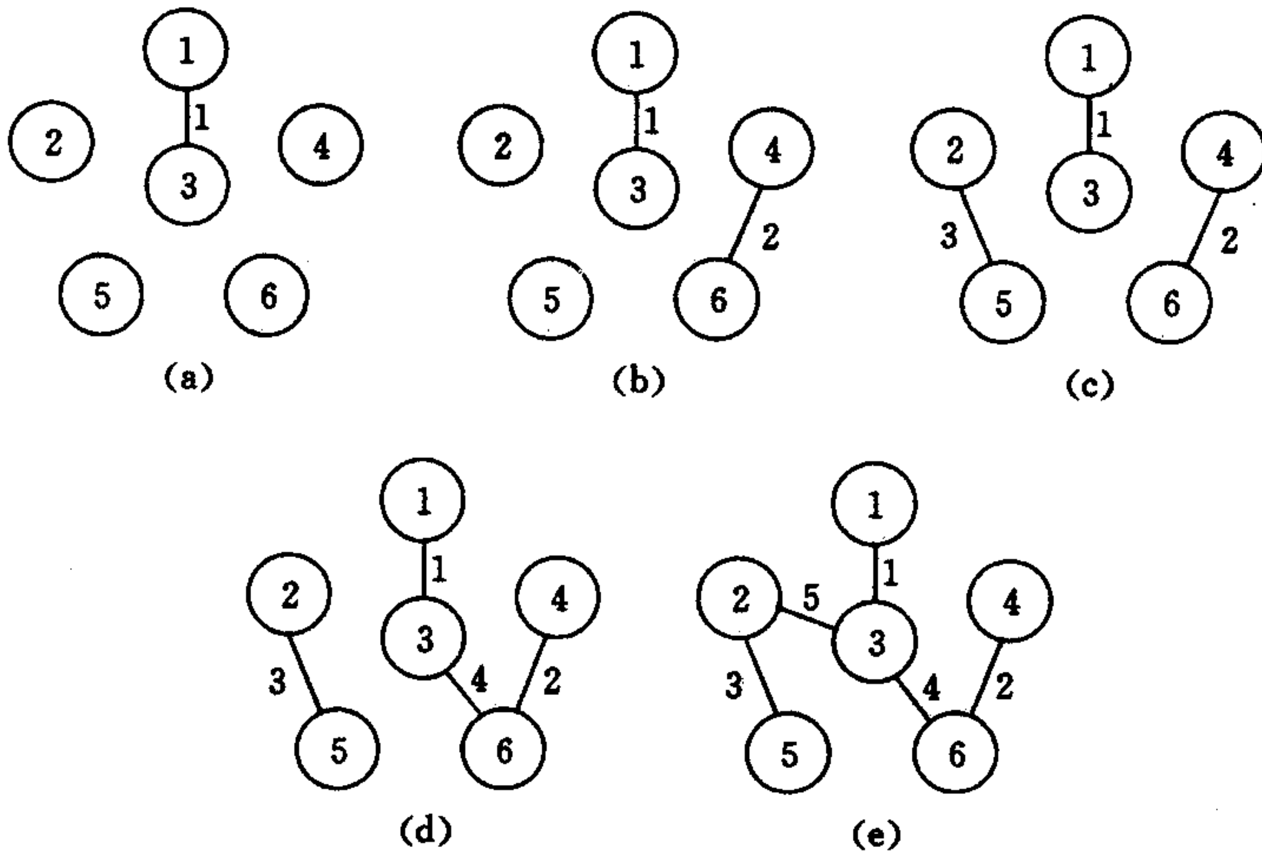
各边按权值排序为：

$d_{13}=1$　$d_{46}=2$　$d_{25}=3$　$d_{36}=4$　$d_{14}=5$

$d_{34}=5$　$d_{23}=5$　$d_{12}=6$　$d_{35}=6$　$d_{56}=6$

按Kruskal算法选边的过程如下图所示：

$d_{13}=1$
$d_{46}=2$
$d_{25}=3$
$d_{36}=4$
$d_{14}=5$
$d_{34}=5$
$d_{23}=5$
$d_{12}=6$
$d_{35}=6$
$d_{56}=6$



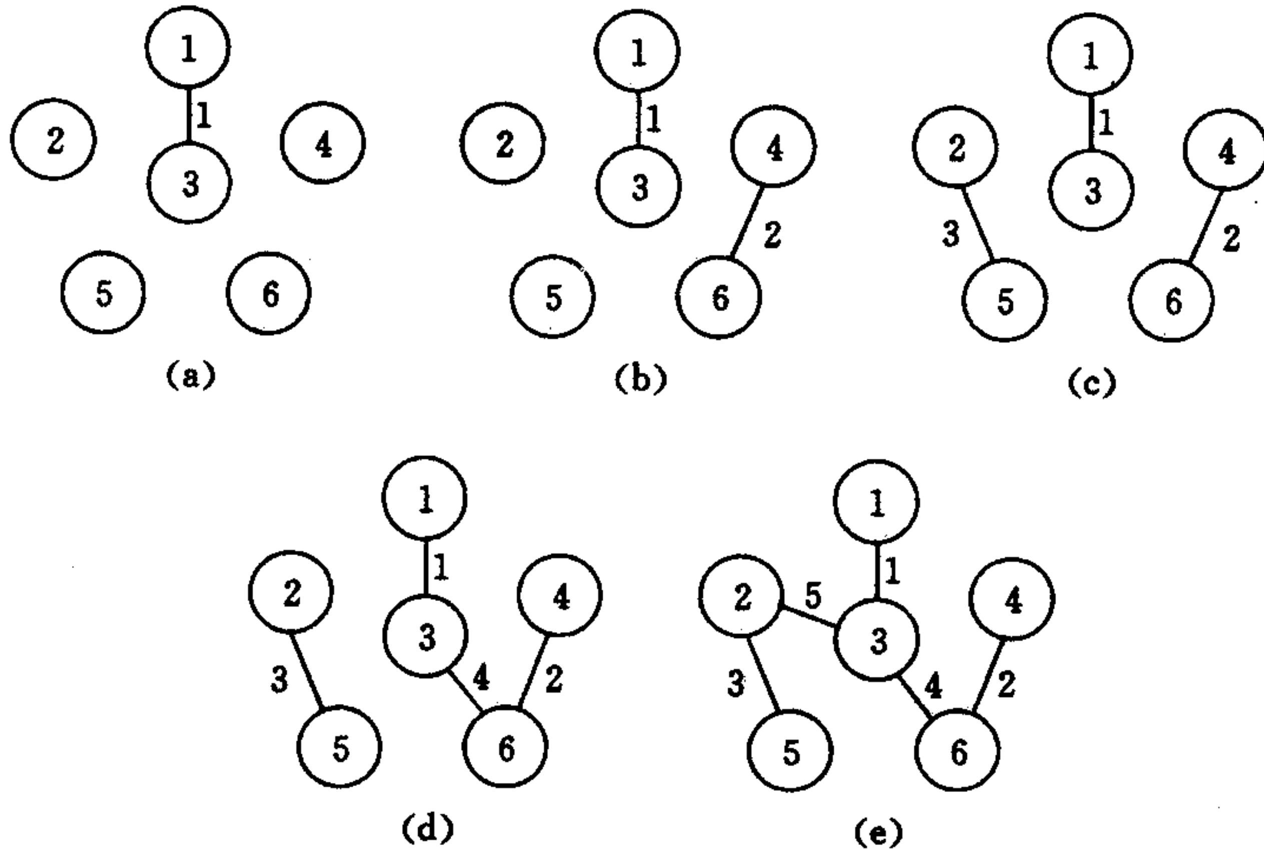
关于集合的一些基本运算可用于实现Kruskal算法：

- 对一个由连通分支组成的集合不断进行修改，需要用到抽象数据类型并查集UnionFind（每一个子集用树根的标号唯一标识，参见P112-6.3节）所支持的基本运算。
- 按权的递增顺序查看边，等价于用最小堆实现一个优先权队列。

程序6-9 Kruskal算法

```
void Kruskal(PrioQueue<eNode<T>>& pq)
{          //优先权队列pq中保存无向图边的集合。
           while (k<n-1 && !pq.IsEmpty()){
                      pq.Serve(x);    //用Serve函数从pq中取出最小代价的边x
                      u=s.Find(x.u);  v=s.Find(x.v);
                      //分别找边x的两个结点x.u和x.v所在的并查集的树根
                      if (u!=v){         //若结点u和v不在同一树中
                                 s.Union(u,v);    //合并u、v所在的两棵并查集树
                                 k++;              //边x加入生成树后边数加1
                                 cout<<“(“<<x.u<<“,”<<x.v<<“,”<<x.w<<“)”;    }
                      }                            //输出生成树上一条边x
           if (k<n-1) throw NonConnected; //若边数少于n-1，则原图非连通
}
```

$d_{13}=1$
$d_{46}=2$
$d_{25}=3$
$d_{36}=4$
$d_{14}=5$
$d_{34}=5$
$d_{23}=5$
$d_{12}=6$
$d_{35}=6$
$d_{56}=6$



程序6-9按边加入的顺序输出，可输出以下边集：

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

边的权值

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/465200032343011131