第 1 章　贪婪算法

虽然设计一个好的求解算法更像是一门艺术，而不像是技术，但仍然存在一些行之有效的能够用于解决许多问题的算法设计方法，你可以使用这些方法来设计算法，并观察这些算法是如何工作的。一般情况下，为了获得较好的性能，必须对算法进行细致的调整。但是在某些情况下，算法经过调整之后性能仍无法达到要求，这时就必须寻求另外的方法来求解该问题。

本章首先引入最优化的概念，然后介绍一种直观的问题求解方法：贪婪算法。最后，应用该算法给出货箱装船问题、背包问题、拓扑排序问题、二分覆盖问题、最短路径问题、最小代价生成树等问题的求解方案。

1.1 最优化问题

本章及后续章节中的许多例子都是最优化问题（ optimization problem），每个最优化问题都包含一组限制条件（ constraint）和一个优化函数（ optimization function），符合限制条件的问题求解方案称为可行解（ feasible solution），使优化函数取得最佳值的可行解称为最优解（optimal solution）。

例 1-1 [渴婴问题] 有一个非常渴的、聪明的小婴儿，她可能得到的东西包括一杯水、一桶牛奶、多罐不同种类的果汁、许多不同的装在瓶子或罐子中的苏打水，即婴儿可得到 n 种不同的饮料。根据以前关于这 n 种饮料的不同体验，此婴儿知道这其中某些饮料更合自己的胃口，因此，婴儿采取如下方法为每一种饮料赋予一个满意度值：饮用 1 盎司第 i 种饮料，对它作出相对评价，将一个数值 s_i 作为满意度赋予第 i 种饮料。

通常，这个婴儿都会尽量饮用具有最大满意度值的饮料来最大限度地满足她解渴的需要，但是不幸的是：具有最大满意度值的饮料有时并没有足够的量来满足此婴儿解渴的需要。设 a_i 是第 i 种饮料的总量（以盎司为单位），而此婴儿需要 t 盎司的饮料来解渴，那么，需要饮用 n 种不同的饮料各多少量才能满足婴儿解渴的需求呢？

设各种饮料的满意度已知。令 x_i 为婴儿将要饮用的第 i 种饮料的量，则需要解决的问题是：找到一组实数 x_i（$1 \leqslant i \leqslant n$），使 $\sum_{i=1}^{n} s_i x_i$ 最大，并满足：$\sum_{i=1}^{n} x_i = t$ 及 $0 \leqslant x_i \leqslant a_i$。

需要指出的是：如果 $\sum_{i=1}^{n} a_i < t$，则不可能找到问题的求解方案，因为即使喝光所有的饮料也不能使婴儿解渴。

对上述问题精确的数学描述明确地指出了程序必须完成的工作，根据这些数学公式，可以对输入/ 输出作如下形式的描述：

输入：n，t，s_i，a_i（其中 $1 \leqslant i \leqslant n$，$n$ 为整数，t、s_i、a_i 为正实数）。

输出：实数 x_i（$1 \leqslant i \leqslant n$），使 $\sum_{i=1}^{n} s_i x_i$ 最大且 $\sum_{i=1}^{n} x_i = t$（$0 \leqslant x_i \leqslant a_i$）。如果 $\sum_{i=1}^{n} a_i < t$，则输出适当的错误信息。

在这个问题中，限制条件是 $\sum_{i=1}^{n} x_i = t$ 且 $0 \leqslant x_i \leqslant a_i$，$1 \leqslant i \leqslant n$。而优化函数是 $\sum_{i=1}^{n} s_i x_i$。任何满足限制条件的一组实数 x_i 都是可行解，而使 $\sum_{i=1}^{n} s_i x_i$ 最大的可行解是最优解。

例 1-2 [装载问题] 有一艘大船准备用来装载货物。所有待装货物都装在货箱中且所有货箱的大小都一样，但货箱的重量都各不相同。设第 i 个货箱的重量为 w_i（$1 \leqslant i \leqslant n$），而货船的最大载重量为 c，我们的目的是在货船上装入最多的货物。

这个问题可以作为最优化问题进行描述：设存在一组变量 x_i，其可能取值为 0 或 1。如 x_i 为 0，则货箱 i 将不被装上船；如 x_i 为 1，则货箱 i 将被装上船。我们的目的是找到一组 x_i，使它满足限制条件 $\sum_{i=1}^{n} w_i x_i \leqslant c$ 且 $x_i \in \{0, 1\}, 1 \leqslant i \leqslant n$。相应的优化函数是 $\sum_{i=1}^{n} x_i$。

满足限制条件的每一组 x_i 都是一个可行解，能使 $\sum_{i=1}^{n} x_i$ 取得最大值的方案是最优解。

例 1-3 [最小代价通讯网络] 城市及城市之间所有可能的通信连接可被视作一个无向图，图的每条边都被赋予一个权值，权值表示建成由这条边所表示的通信连接所要付出的代价。包含图中所有顶点（城市）的连通子图都是一个可行解。设所有的权值都非负，则所有可能的可行解都可表示成无向图的一组生成树，而最优解是其中具有最小代价的生成树。

在这个问题中，需要选择一个无向图中的边集合的子集，这个子集必须满足如下限制条件：所有的边构成一个生成树。而优化函数是子集中所有边的权值之和。

1.2 算法思想

在贪婪算法（greedy method）中采用逐步构造最优解的方法。在每个阶段，都作出一个看上去最优的决策（在一定的标准下）。决策一旦作出，就不可再更改。作出贪婪决策的依据称为贪婪准则（greedy criterion）。

例 1-4 [找零钱] 一个小孩买了价值少于 1 美元的糖，并将 1 美元的钱交给售货员。售货员希望用数目最少的硬币找给小孩。假设提供了数目不限的面值为 25 美分、10 美分、5 美分、及 1 美分的硬币。售货员分步骤组成要找的零钱数，每次加入一个硬币。选择硬币时所采用的贪婪准则如下：每一次选择应使零钱数尽量增大。为保证解法的可行性（即：所给的零钱等于要找的零钱数），所选择的硬币不应使零钱总数超过最终所需的数目。

假设需要找给小孩 67 美分，首先入选的是两枚 25 美分的硬币，第三枚入选的不能是 25 美分的硬币，否则硬币的选择将不可行（零钱总数超过 67 美分），第三枚应选择 10 美分的硬币，然后是 5 美分的，最后加入两个 1 美分的硬币。

贪婪算法有种直觉的倾向，在找零钱时，直觉告诉我们应使找出的硬币数目最少（至少是接近最少的数目）。可以证明采用上述贪婪算法找零钱时所用的硬币数目的确最少（见练习 1）。

例 1-5 [机器调度] 现有 n 件任务和无限多台的机器，任务可以在机器上得到处理。每件任务的开始时间为 s_i，完成时间为 f_i，$s_i<f_i$。$[s_i, f_i]$ 为处理任务 i 的时间范围。两个任务 i，j 重指两个任务的时间范围区间有重叠，而并非是指 i，j 的起点或终点重合。例如：区间[1，4]与区间[2，4]重叠，而与区间[4，7]不重叠。一个可行的任务分配是指在分配中没有两件重叠的任务分配给同一台机器。因此，在可行的分配中每台机器在任何时刻最多只处理一个任务。最优分配是指使用的机器最少的可行分配方案。

假设有 $n=7$ 件任务，标号为 a 到 g。它们的开始与完成时间如图 13-1a 所示。若将任务 a 分给机器 $M1$，任务 b 分给机器 $M2$，...，任务 g 分给机器 $M7$，这种分配是可行的分配，共使用了七台机器。但它不是最优分配，因为有其他分配方案可使利用的机器数目更少，例如：可以将任务 a、b、d 分配给同一台机器，则机器的数目降为五台。

一种获得最优分配的贪婪方法是逐步分配任务。每步分配一件任务，且按任务开始时间的非递减次序进行分配。若已经至少有一件任务分配给某台机器，则称这台机器是旧的；若机器非旧，则它是新的。在选择机器时，采用以下贪婪准则：根据欲分配任务的开始时间，若此时有旧的机器可用，则将任务分给旧的机器。否则，将任务分配给一台新的机器。

根据例子中的数据，贪婪算法共分为 n = 7 步，任务分配的顺序为 a、f、b、c、g、e、d。第一步没有旧机器，因此将 a 分配给一台新机器（比如 $M1$）。这台机器在 0 到 2 时刻处于忙状态。在第二步，考虑任务 f。由于当 f 启动时旧机器仍处于忙状态，因此将 f 分配给一台新机器(设为 $M2$)。第三步考虑任务 b，由于旧机器 $M1$ 在 $S_b=3$ 时刻已处于闲状态，因此将 b 分配给 $M1$ 执行，$M1$ 下一次可用时刻变成 $f_b=7$，$M2$ 的可用时刻变成 $f_f=5$。第四步，考虑任务 c。由于没有旧机器在 $S_c=4$ 时刻可用，因此将 c 分配给一台新机器（$M3$），这台机器下一次可用时间为 $f_c=7$。第五步考虑任务 g，将其分配给机器 $M2$，第六步将任务 e 分配给机器 $M1$，最后在第七步，任务

2 分配给机器 $M3$。（注意：任务 d 也可分配给机器 $M2$）。

上述贪婪算法能导致最优机器分配的证明留作练习（练习 7）。可按如下方式实现一个复杂性为 O (nl o gn)的贪婪算法：首先采用一个复杂性为 O (nl o gn)的排序算法（如堆排序）按 S_i 的递增次序排列各个任务，然后使用一个关于旧机器可用时间的最小堆。

例 1-6 [最短路径] 给出一个有向网络，路径的长度定义为路径所经过的各边的耗费之和。要求找一条从初始顶点 s 到达目的顶点 d 的最短路径。

贪婪算法分步构造这条路径，每一步在路径中加入一个顶点。假设当前路径已到达顶点 q，且顶点 q 并不是目的顶点 d。加入下一个顶点所采用的贪婪准则为：选择离 q 最近且目前不在路径中的顶点。

这种贪婪算法并不一定能获得最短路径。例如，假设在图 1 3 - 2 中希望构造从顶点 1 到顶点 5 的最短路径，利用上述贪婪算法，从顶点 1 开始并寻找目前不在路径中的离顶点 1 最近的顶点。到达顶点 3，长度仅为 2 个单位，从顶点 3 可以到达的最近顶点为 4，从顶点 4 到达顶点 2，最后到达目的顶点 5。所建立的路径为 1 , 3 , 4 , 2 , 5，其长度为 1 0。这条路径并不是有向图中从 1 到 5 的最短路径。事实上，有几条更短的路径存在，例如路径 1，4，5 的长度为 6。

根据上面三个例子，回想一下前几章所考察的一些应用，其中有几种算法也是贪婪算法。例如，霍夫曼树算法，利用 n- 1 步来建立最小加权外部路径的二叉树，每一步都将两棵二叉树合并为一棵，算法中所使用的贪婪准则为：从可用的二叉树中选出权重最小的两棵。L P T 调度规则也是一种贪婪算法，它用 n 步来调度 n 个作业。首先将作业按时间长短排序，然后在每一步中为一个任务分配一台机器。选择机器所利用的贪婪准则为：使目前的调度时间最短。将新作业调度到最先完成的机器上（即最先空闲的机器）。

注意到在机器调度问题中，贪婪算法并不能保证最优，然而，那是一种直觉的倾向且一般情况下结果总是非常接近最优值。它利用的规则就是在实际环境中希望人工机器调度所采用的规则。算法并不保证得到最优结果，但通常所得结果与最优解相差无几，这种算法也称为启发式方法（ h e u r i s t i c s)。因此L P T 方法是一种启发式机器调度方法。定理 9 - 2 陈述了 L P T 调度的完成时间与最佳调度的完成时间之间的关系，因此 L P T 启发式方法具有限定性能（ bounded performance ）。具有限定性能的启发式方法称为近似算法（ a p p r o x i m a t i o n a l g o r i t h m）。

本章的其余部分将介绍几种贪婪算法的应用。在有些应用中，贪婪算法所产生的结果总是最优的解决方案。但对其他一些应用，生成的算法只是一种启发式方法，可能是也可能不是近似算法。

练习

1. 证明找零钱问题（例 1 3 - 4）的贪婪算法总能产生具有最少硬币数的零钱。

2. 考虑例 1 3 - 4 的找零钱问题，假设售货员只有有限的 2 5 美分， 1 0 美分， 5 美分和 1 美分的硬币，给出一种找零钱的贪婪算法。这种方法总能找出具有最少硬币数的零钱吗？证明结论。

3. 扩充例 1 3 - 4 的算法，假定售货员除硬币外还有 50, 20, 10, 5, 和 1 美元的纸币，顾客买价格为 x 美元和 y 美分的商品时所付的款为 u 美元和 v 美分。算法总能找出具有最少纸币与硬币数目的零钱吗？证明结论。

4. 编写一个 C + +程序实现例 1 3 - 4 的找零钱算法。假设售货员具有面值为 1 0 0，2 0，1 0，5 和 1 美元的纸币和各种硬币。程序可包括输入模块(即输入所买商品的价格及顾客所付的钱数)，输出模块（输出零钱的数目及要找的各种货币的数目）和计算模块（计算怎样给出零钱）。

5. 假设某个国家所使用硬币的币值为 1 4 , 2 , 5 和 1 分，则例 1 3 - 4 的贪婪算法总能产生具有最少硬币数的零钱吗？证明结论。

6. 1) 证明例 1 3 - 5 的贪婪算法总能找到最优任务分配方案。

2) 实现这种算法，使其复杂性为 O (nl o gn)（提示：根据完成时间建立最小堆）。

*7. 考察例 1 3 - 5 的机器调度问题。假定仅有一台机器可用，那么将选择最大数量的任务在这台机器上执行。例如，所选择的最大任务集合为{*a*,*b*,*e*}。解决这种任务选择问题的贪婪算法可按步骤选择任务，每步选择一个任务，其贪婪准则如下：从剩下的任务中选择具有最小的完成时间且不会与现有任务重叠的任务。

1) 证明上述贪婪算法能够获得最优选择。

2) 实现该算法，其复杂性应为 O(*n*l o g*n*)。（提示：采用一个完成时间的最小堆）

1.3 应用

1.3.1 货箱装船

这个问题来自例 1 - 2。船可以分步装载，每步装一个货箱，且需要考虑装载哪一个货箱。根据这种思想可利用如下贪婪准则：从剩下的货箱中，选择重量最小的货箱。这种选择次序可以保证所选的货箱总重量最小，从而可以装载更多的货箱。根据这种贪婪策略，首先选择最轻的货箱，然后选次轻的货箱，如此下去直到所有货箱均装上船或船上不能再容纳其他任何一个货箱。

例 1-7 假设 n =8, [w_1, ... w_8]=[100,200,50,90,150,50,20,80], c= 4 0 0。利用贪婪算法时，所考察货箱的顺序为 7 , 3 , 6 , 8 , 4 , 1 , 5 , 2。货箱 7 , 3 , 6 , 8 , 4 , 1 的总重量为 3 9 0 个单位且已被装载，剩下的装载能力为 1 0 个单位，小于剩下的任何一个货箱。在这种贪婪解决算法中得到[x_1, ..., x_8] = [1 , 0 , 1 , 1 , 0 , 1 , 1 , 1]且$\sum x_i$= 6。

定理 1-1 利用贪婪算法能产生最佳装载。

证明可以采用如下方式来证明贪婪算法的最优性：令 x = [x_1, ..., x_n]为用贪婪算法获得的解，令 y =[y_1, ..., y_n]为任意一个可行解，只需证明 $\sum_{i=1}^{n} x_i \geqslant \sum_{i=1}^{n} y_i$ 。不失一般性，可以假设货箱都排好了序：即 $w_i \leqslant w_{i+1}$（$1 \leqslant i \leqslant n$）。然后分几步将 y 转化为 x，转换过程中每一步都产生一个可行的新 y，且 $\sum_{i=1}^{n} y_i$ 大于等于未转化前的值，最后便可证明 $\sum_{i=1}^{n} x_i \geqslant \sum_{i=1}^{n} y_i$ 。

根据贪婪算法的工作过程，可知在[0, n] 的范围内有一个 k，使得 x_i=1, $i \leqslant k$ 且 x_i=0, i>k。寻找[1 ,n]范围内最小的整数 j，使得 $x_j \neq y_j$ 。若没有这样的 j 存在，则 $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$ 。如果有这样的 j 存在，则 $j \leqslant k$，否则 y 就不是一个可行解，因为 $x_j \neq y_j$ ，x_j= 1 且 y_j= 0。令 y_j= 1，若结果得到的 y 不是可行解，则在[j+ 1 ,n]范围内必有一个 l 使得 y_l= 1。令 y_l= 0，由于 $w_j \leqslant w_l$ ，则得到的 y 是可行的。而且，得到的新 y 至少与原来的 y 具有相同数目的 1。

经过数次这种转化，可将 y 转化为 x。由于每次转化产生的新 y 至少与前一个 y 具有相同数目的 1，因此 x 至少与初始的 y 具有相同的数目 1。货箱装载算法的 C + +代码实现见程序 1 3 - 1。由于贪婪算法按货箱重量递增的顺序装载，程序 1 3 - 1 首先利用间接寻址排序函数 I n d i r e c t S o r t 对货箱重量进行排序（见 3 . 5 节间接寻址的定义），随后货箱便可按重量递增的顺序装载。由于间接寻址排序所需的时间为 O (*n*l o g*n*)（也可利用 9 . 5 . 1 节的堆排序及第 2 章的归并排序），算法其余部分所需时间为 O (*n*)，因此程序 1 3 - 1 的总的复杂性为 O (*n*l o g*n*)。

程序 13-1 货箱装船

```
template<class T>
void ContainerLoading(int x[], T w[], T c, int n)
{// 货箱装船问题的贪婪算法
// x[i] = 1 当且仅当货箱 i 被装载，  1<=i<=n
// c 是船的容量, w 是货箱的重量
// 对重量按间接寻址方式排序
// t 是间接寻址表
int *t = new int [n+1];
I n d i r e c t S o r t ( w, t, n);
```

```
// 此时, w[t[i]] <= w[t[i+1]], 1<=i<n
// 初始化 x
for (int i = 1; i <= n; i++)
x[i] = 0;
// 按重量次序选择物品
for (i = 1; i <= n && w[t[i]] <= c; i++) {
x[t[i]] = 1;
c -= w[t[i]];} // 剩余容量
delete [] t;
}
```

1.3.2 0/1 背包问题

在 0 / 1 背包问题中，需对容量为 c 的背包进行装载。从 n 个物品中选取装入背包的物品，每件物品 i 的重量为 w_i，价值为 p_i。对于可行的背包装载，背包中物品的总重量不能超过背包的容量，最佳装载是指所装入的物品价值最高，即 $\sum_{i=1}^{n} p_i x_i$ 取得最大值。约束条件为 $\sum_{i=1}^{n} w_i x_i \leqslant c$ 和 $x_i \in [0, 1]$ $(1 \leqslant i \leqslant n)$。

在这个表达式中，需求出 x_i 的值。$x_i = 1$ 表示物品 i 装入背包中，$x_i = 0$ 表示物品 i 不装入背包。0 / 1 背包问题是一个一般化的货箱装载问题，即每个货箱所获得的价值不同。货箱装载问题转化为背包问题的形式为：船作为背包，货箱作为可装入背包的物品。

例 1-8 在杂货店比赛中你获得了第一名，奖品是一车免费杂货。店中有 n 种不同的货物。规则规定从每种货物中最多只能拿一件，车子的容量为 c，物品 i 需占用 w_i 的空间，价值为 p_i。你的目标是使车中装载的物品价值最大。当然，所装货物不能超过车的容量，且同一种物品不得拿走多件。这个问题可仿照 0 / 1 背包问题进行建模，其中车对应于背包，货物对应于物品。

0 / 1 背包问题有好几种贪婪策略，每个贪婪策略都采用多步过程来完成背包的装入。在每一步过程中利用贪婪准则选择一个物品装入背包。一种贪婪准则为：从剩余的物品中，选出可以装入背包的价值最大的物品，利用这种规则，价值最大的物品首先被装入（假设有足够容量），然后是下一个价值最大的物品，如此继续下去。这种策略不能保证得到最优解。例如，考虑 n=2, w=[100,10,10], p =[20,15,15], c = 1 0 5。当利用价值贪婪准则时，获得的解为 x= [1 , 0 , 0]，这种方案的总价值为 2 0。而最优解为[0 , 1 , 1]，其总价值为 3 0。

另一种方案是重量贪婪准则是：从剩下的物品中选择可装入背包的重量最小的物品。虽然这种规则对于前面的例子能产生最优解，但在一般情况下则不一定能得到最优解。考虑 n= 2 ,w=[10,20], p=[5,100], c= 2 5。当利用重量贪婪策略时，获得的解为 x =[1,0], 比最优解[0 , 1] 要差。

还可以利用另一方案，价值密度 p_i/w_i 贪婪算法，这种选择准则为：从剩余物品中选择可装入包的 p_i/w_i 值最大的物品，这种策略也不能保证得到最优解。利用此策略试解 n= 3 ,w=[20,15,15], p=[40,25,25], c=30 时的最优解。

我们不必因所考察的几个贪婪算法都不能保证得到最优解而沮丧， 0 / 1 背包问题是一个 NP-复杂问题。对于这类问题，也许根本就不可能找到具有多项式时间的算法。虽然按 p_i/w_i 非递（增）减的次序装入物品不能保证得到最优解，但它是一个直觉上近似的解。我们希望它是一个好的启发式算法，且大多数时候能很好地接近最后算法。在 6 0 0 个随机产生的背包问题中，用这种启发式贪婪算法来解有 2 3 9 题为最优解。有 5 8 3 个例子与最优解相差 1 0 %，所有 6 0 0 个答案与最优解之差全在 2 5 %以内。该算法能在 O (nl o gn)时间内获得如此好的性能。我们也许会问，是否存在一个 x (x<1 0 0)，使得贪婪启发法的结果与最优值相差在 x%以内。答案是否定的。为说明这一点，考虑例子 n =2, w = [1 ,y], p= [1 0 , 9y], 和 c= y。贪婪算法结果为 x=[1,0], 这

种方案的值为 10。对于 $y \geqslant 10/9$，最优解的值为 9y。因此，贪婪算法的值与最优解的差对最优解的比例为((9y - 10)/9y* 100)%，对于大的 y，这个值趋近于 100%。但是可以建立贪婪启发式方法来提供解，使解的结果与最优解的值之差在最优值的 x% (x<100) 之内。首先将最多 k 件物品放入背包，如果这 k 件物品重量大于 c，则放弃它。否则，剩余的容量用来考虑将剩余物品按 p_i/w_i 递减的顺序装入。通过考虑由启发法产生的解法中最多为 k 件物品的所有可能的子集来得到最优解。

例 13-9 考虑 n =4, w=[2,4,6,7], p=[6,10,12,13], c = 11。当 k= 0 时，背包按物品价值密度非递减顺序装入，首先将物品 1 放入背包，然后是物品 2，背包剩下的容量为 5 个单元，剩下的物品没有一个合适的，因此解为 x = [1, 1, 0, 0]。此解获得的价值为 16。

现在考虑 k = 1 时的贪婪启发法。最初的子集为{1}, {2}, {3}, {4}。子集{1}, {2}产生与 k= 0 时相同的结果，考虑子集{3}，置 x_3 为 1。此时还剩 5 个单位的容量，按价值密度非递增顺序来考虑如何利用这 5 个单位的容量。首先考虑物品 1，它适合，因此取 x_1 为 1，这时仅剩下 3 个单位容量了，且剩余物品没有能够加入背包中的物品。通过子集{3}开始求解得结果为 x = [1, 0, 1, 0]，获得的价值为 18。若从子集{4}开始，产生的解为 x = [1, 0, 0, 1]，获得的价值为 19。考虑子集大小为 0 和 1 时获得的最优解为[1, 0, 0, 1]。这个解是通过 k= 1 的贪婪启发式算法得到的。

若 k= 2, 除了考虑 k< 2 的子集，还必需考虑子集{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}和{3, 4}。首先从最后一个子集开始，它是不可行的，故将其抛弃，剩下的子集经求解分别得到如下结果：[1, 1, 0, 0], [1, 0, 1, 0], [1, 0, 0, 1], [0, 1, 1, 0]和[0, 1, 0, 1]，这些结果中最后一个价值为 23，它的值比 k= 0 和 k= 1 时获得的解要高，这个答案即为启发式方法产生的结果。

这种修改后的贪婪启发方法称为 k 阶优化方法（k-optimal）。也就是，若从答案中取出 k 件物品，并放入另外 k 件，获得的结果不会比原来的好，而且用这种方式获得的值在最优值的(100/(k+1))%以内。当 k= 1 时，保证最终结果在最佳值的 50%以内；当 k= 2 时，则在 33.33%以内等等，这种启发式方法的执行时间随 k 的增大而增加，需要测试的子集数目为 O(n^k)，每一个子集所需时间为 O(n)，因此当 k >0 时总的时间开销为 O(n^{k+1})。实际观察到的性能要好得多。

1.3.3 拓扑排序

一个复杂的工程通常可以分解成一组小任务的集合，完成这些小任务意味着整个工程的完成。例如，汽车装配工程可分解为以下任务：将底盘放上装配线，装轴，将座位装在底盘上，上漆，装刹车，装门等等。任务之间具有先后关系，例如在装轴之前必须先将底板放上装配线。任务的先后顺序可用有向图表示——称为顶点活动（ Activity On Vertex, AOV）网络。有向图的顶点代表任务，有向边$\langle i,j\rangle$ 表示先后关系：任务 j 开始前任务 i 必须完成。图 1 - 4 显示了六个任务的工程，边（ 1, 4）表示任务 1 在任务 4 开始前完成，同样边（ 4, 6）表示任务 4 在任务 6 开始前完成，边（1, 4）与（4, 6）合起来可知任务 1 在任务 6 开始前完成，即前后关系是传递的。由此可知，边（1, 4）是多余的，因为边（1, 3）和（3, 4）已暗示了这种关系。

在很多条件下，任务的执行是连续进行的，例如汽车装配问题或平时购买的标有“需要装配”的消费品（自行车、小孩的秋千装置，割草机等等）。我们可根据所建议的顺序来装配。在由任务建立的有向图中，边（ i,j）表示在装配序列中任务 i 在任务 j 的前面，具有这种性质的序列称为拓扑序列（topological orders 或 topological sequences)。根据任务的有向图建立拓扑序列的过程称为拓扑排序（topological sorting）。图 1 - 4 的任务有向图有多种拓扑序列，其中的三种为 123456，132456 和 215346，序列 142356 就不是拓扑序列，因为在这个序列中任务 4 在 3 的前面，而任务有向图中的边为（ 3, 4），这种序列与边（ 3, 4）及其他边所指示的序列

相矛盾。可用贪婪算法来建立拓扑序列。算法按从左到右的步骤构造拓扑序列，每一步在排好的序列中加入一个顶点。利用如下贪婪准则来选择顶点：从剩下的顶点中，选择顶点 w，使得 w 不存在这样的入边（ v,w），其中顶点 v 不在已排好的序列结构中出现。注意到如果加入的顶点 w 违背了这个准则（即有向图中存在边（ v,w）且 v 不在已构造的序列中），则无法完成拓扑排序，因为顶点 v 必须跟随在顶点 w 之后。贪婪算法的伪代码如图 13-5 所示。while 循环的每次迭代代表贪婪算法的一个步骤。

现在用贪婪算法来求解图 1-4 的有向图。首先从一个空序列 V 开始，第一步选择 V 的第一个顶点。此时，在有向图中有两个候选顶点 1 和 2，若选择顶点 2，则序列 V＝2，第一步完成。第二步选择 V 的第二个顶点，根据贪婪准则可知候选顶点为 1 和 5，若选择 5，则 V＝25。下一步，顶点 1 是唯一的候选，因此 V＝251。第四步，顶点 3 是唯一的候选，因此把顶点 3 加入 V 得到 V＝2513。在最后两步分别加入顶点 4 和 6 ，得 V＝251346。

1. 贪婪算法的正确性

为保证贪婪算法算的正确性，需要证明： 1) 当算法失败时，有向图没有拓扑序列； 2) 若算法没有失败，V 即是拓扑序列。2) 即是用贪婪准则来选取下一个顶点的直接结果， 1) 的证明见定理 13-2，它证明了若算法失败，则有向图中有环路。若有向图中包含环 $q_j q_{j+1}..q_k q_j$，则它没有拓扑序列，因为该序列暗示了 q_j 一定要在 q_j 开始前完成。

定理 1-2 如果图 13-5 算法失败，则有向图含有环路。

证明注意到当失败时| V |<n, 且没有候选顶点能加入 V 中，因此至少有一个顶点 q_1 不在 V 中，有向图中必包含边（ q_2, q_1）且 q_2 不在 V 中，否则， q_1 是可加入 V 的候选顶点。同样，必有边（q_3, q_2）使得 q_3 不在 V 中，若 $q_3 = q_1$ 则 $q_1 q_2 q_3$ 是有向图中的一个环路；若 $q_3 \neq q_1$，则必存在 q_4 使（q_4, q_3）是有向图的边且 q_4 不在 V 中，否则，q_3 便是 V 的一个候选顶点。若 q_4 为 q_1, q_2, q_3 中的任何一个，则又可知有向图含有环，因为有向图具有有限个顶点数 n，继续利用上述方法，最后总能找到一个环路。

2. 数据结构的选择

为将图 1-5 用 C++代码来实现，必须考虑序列 V 的描述方法，以及如何找出可加入 V 的候选顶点。一种高效的实现方法是将序列 V 用一维数组 v 来描述的，用一个栈来保存可加入 V 的候选顶点。另有一个一维数组 InDegree，InDegree[j]表示与顶点 j 相连的节点 i 的数目，其中顶点 i 不是 V 中的成员，它们之间的有向图的边表示为（ i,j）。当 InDegree[j] 变为 0 时表示 j 成为一个候选节点。序列 V 初始时为空。InDegree[j]为顶点 j 的入度。每次向 V 中加入一个顶点时，所有与新加入顶点邻接的顶点 j，其 InDegree[j]减 1。对于有向图 1-4，开始时 InDegree[1:6]=[0,0,1,3,1,3]。由于顶点 1 和 2 的 InDegree 值为 0，因此它们是可加入 V 的候选顶点，由此，顶点 1 和 2 首先入栈。每一步，从栈中取出一个顶点将其加入 V，同时减去与其邻接的顶点的 InDegree 值。若在第一步时从栈中取出顶点 2 并将其加入 V，便得到了 v[0]=2，和 InDegree[1:6]=[0,0,1,2,0,3]。由于 InDegree[5]刚刚变为 0，因此将顶点 5 入栈。

程序 13-2 给出了相应的 C++代码，这个代码被定义为 Network 的一个成员函数。而且，它对于有无加权的有向图均适用。但若用于无向图（不论其有无加权）将会得到错误的结果，因为拓扑排序是针对有向图来定义的。为解决这个问题，利用同样的模板来定义成员函数 AdjacencyGraph, AdjacencyWGraph，LinkedGraph 和 LinkedWGraph。这些函数可重载 Network 中的函数并可输出错误信息。如果找到拓扑序列，则 Topological 函数返回 true; 若输入的有向图无拓扑序列则返回 false。当找到拓扑序列时，将其返回到 v[0:n-1]中。

3. Network:Topological 的复杂性

第一和第三个 for 循环的时间开销为(n)。若使用（耗费）邻接矩阵,则第二个 for 循环所用的时间为(n^2); 若使用邻接链表,则所用时间为(n+e)。在两个嵌套的 while 循环中，外层循环需执

行 n 次，每次将顶点 w 加入到 v 中，并初始化内层 while 循环。使用邻接矩阵时，内层 while 循环对于每个顶点 w 需花费(n)的时间；若利用邻接链表，则这个循环需花费 d_{wout} 的时间，因此，内层 while 循环的时间开销为(n_2)或$(n+e)$。所以，若利用邻接矩阵，程序 13-2 的时间复杂性为(n_2)，若利用邻接链表则为$(n+e)$。

程序 13-2 拓扑排序

```
bool Network::Topological(int v[])
{// 计算有向图中顶点的拓扑次序
// 如果找到了一个拓扑次序，则返回 true，此时，在 v[0:n-1]中记录拓扑次序
// 如果不存在拓扑次序，则返回 false
int n = Vertices();
// 计算入度
int *InDegree = new int [n+1];
InitializePos(); // 图遍历器数组
for (int i = 1; i <= n; i++) // 初始化
InDegree[i] = 0;
for (i = 1; i <= n; i++) {// 从 i 出发的边
int u = Begin(i);
while (u) {
InDegree[u]++;
u = NextVertex(i);}
}
// 把入度为 0 的顶点压入堆栈
LinkedStack<int> S;
for (i = 1; i <= n; i++)
if (!InDegree[i]) S.Add(i);
// 产生拓扑次序
i = 0; // 数组 v 的游标
while (!S.IsEmpty()) {// 从堆栈中选择
int w; // 下一个顶点
S.Delete(w);
v[i++] = w;
int u = Begin(w);
while (u) {// 修改入度
InDegree[u]--;
if (!InDegree[u]) S.Add(u);
u = NextVertex(w);}
}
DeactivatePos();
delete [] InDegree;
return (i == n);
}
```

1.3.4 二分覆盖

二分图是一个无向图，它的 n 个顶点可二分为集合 A 和集合 B，且同一集合中的任意两个

顶点在图中无边相连（即任何一条边都是一个顶点在集合 A 中，另一个在集合 B 中）。当且仅当 B 中的每个顶点至少与 A 中一个顶点相连时，A 的一个子集 A' 覆盖集合 B（或简单地说，A' 是一个覆盖）。覆盖 A' 的大小即为 A' 中的顶点数目。当且仅当 A' 是覆盖 B 的子集中最小的时，A' 为最小覆盖。

例 1-10 考察如图 1 - 6 所示的具有 17 个顶点的二分图，A={1, 2, 3, 16, 17}和 B={4, 5, 6, 7, 8, 9,10, 11, 12, 13, 14, 15}，子集 A'={1, 16, 17}是 B 的最小覆盖。在二分图中寻找最小覆盖的问题为二分覆盖（bipartite-cover）问题。在例 12 - 3 中说明了最小覆盖是很有用的，因为它能解决“在会议中使用最少的翻译人员进行翻译”这一类的问题。

二分覆盖问题类似于集合覆盖（set-cover）问题。在集合覆盖问题中给出了 k 个集合 S= {S_1, S_2, ., S_k}，每个集合 S_i 中的元素均是全集 U 中的成员。当且仅当$\bigcup_{i\leq}S_i$=U 时，S 的子集 S' 覆盖 U，S'中的集合数目即为覆盖的大小。当且仅当没有能覆盖 U 的更小的集合时，称 S′ 为最小覆盖。可以将集合覆盖问题转化为二分覆盖问题（反之亦然），即用 A 的顶点来表示 S_1, ., S_k，B 中的顶点代表 U 中的元素。当且仅当 S 的相应集合中包含 U 中的对应元素时，在 A 与 B 的顶点之间存在一条边。

例 1 - 11 令 S= {S_1，. . .，S_5}, U= {4，5，. . .，15}, S_1= {4，6，7，8，9，13}，S_2= {4，5，6，8}，S_3= {8，10，12，14，15}，S_4= {5，6，8，12，14，15}，S_5= {4，9，10，11}。S'= {S_1，S_4，S_5}是一个大小为 3 的覆盖，没有更小的覆盖，S' 即为最小覆盖。这个集合覆盖问题可映射为图 1-6 的二分图，即用顶点 1，2，3，16 和 17 分别表示集合 S_1，S_2，S_3，S_4 和 S_5，顶点 j 表示集合中的元素 j，4⩽j⩽15。

集合覆盖问题为 NP-复杂问题。由于集合覆盖与二分覆盖是同一类问题，二分覆盖问题也是 NP-复杂问题。因此可能无法找到一个快速的算法来解决它，但是可以利用贪婪算法寻找一种快速启发式方法。一种可能是分步建立覆盖 A'，每一步选择 A 中的一个顶点加入覆盖。顶点的选择利用贪婪准则：从 A 中选取能覆盖 B 中还未被覆盖的元素数目最多的顶点。

例 1-12 考察图 1 - 6 所示的二分图，初始化 A'= 且 B 中没有顶点被覆盖，顶点 1 和 16 均能覆盖 B 中的六个顶点，顶点 3 覆盖五个，顶点 2 和 17 分别覆盖四个。因此，在第一步往 A' 中加入顶点 1 或 16，若加入顶点 16，则它覆盖的顶点为{5, 6, 8, 12, 14, 15}，未覆盖的顶点为{4, 7, 9, 10, 11, 13}。顶点 1 能覆盖其中四个顶点（{4, 7, 9, 13}），顶点 2 覆盖一个({4})，顶点 3 覆盖一个（{10}），顶点 16 覆盖零个，顶点 17 覆盖四个{4, 9, 10, 11}。下一步可选择 1 或 17 加入 A'。若选择顶点 1，则顶点{10, 11} 仍然未被覆盖，此时顶点 1，2，16 不覆盖其中任意一个，顶点 3 覆盖一个，顶点 17 覆盖两个，因此选择顶点 17，至此所有顶点已被覆盖，得 A'= {16, 1, 17}。

图 1 - 7 给出了贪婪覆盖启发式方法的伪代码，可以证明： 1) 当且仅当初始的二分图没有覆盖时，算法找不到覆盖； 2) 启发式方法可能找不到二分图的最小覆盖。

1. 数据结构的选取及复杂性分析

为实现图 13 - 7 的算法，需要选择 A' 的描述方法及考虑如何记录 A 中节点所能覆盖的 B 中未覆盖节点的数目。由于对集合 A' 仅使用加法运算，则可用一维整型数组 C 来描述 A'，用 m 来记录 A' 中元素个数。将 A' 中的成员记录在 C[0 :m-1] 中。对于 A 中顶点 i，令 New_i 为 i 所能覆盖的 B 中未覆盖的顶点数目。逐步选择 New_i 值最大的顶点。由于一些原来未被覆盖的顶点现在被覆盖了，因此还要修改各 New_i 值。在这种更新中，检查 B 中最近一次被 V 覆盖的顶点，令 j 为这样的一个顶点，则 A 中所有覆盖 j 的顶点的 New_i 值均减 1。

例 1-13 考察图 1 - 6，初始时(New_1, New_2, New_3, New_{16}, New_{17}) = (6, 4, 5, 6, 4)。假设在例 1 - 12 中，第一步选择顶点 16，为更新 New_i 的值检查 B 中所有最近被覆盖的顶点，这些顶点为 5, 6, 8, 12, 14 和 15。当检查顶点 5 时，将顶点 2 和 16 的 New_i 值分别减 1，因为顶

点 5 不再是被顶点 2 和 16 覆盖的未覆盖节点；当检查顶点 6 时，顶点 1，2，和 16 的相应值分别减 1；同样，检查顶点 8 时，1，2，3 和 16 的值分别减 1；当检查完所有最近被覆盖的顶点，得到的 New_i 值为（4，1，0，4）。下一步选择顶点 1，最新被覆盖的顶点为 4，7，9 和 13；检查顶点 4 时，New_1，New_2，和 New_{17} 的值减 1；检查顶点 7 时，New_1 的值减 1，因为顶点 1 是覆盖 7 的唯一顶点。

为了实现顶点选取的过程，需要知道 New_i 的值及已被覆盖的顶点。可利用一个二维数组来达到这个目的，New 是一个整型数组，New[i] 即等于 New_i，且 cov 为一个布尔数组。若顶点 i 未被覆盖则 cov[i]等于 false，否则 cov[i]为 true。现将图 1-7 的伪代码进行细化得到图 1-8。

```
m=0; //当前覆盖的大小
对于 A 中的所有 i，New[i]=Degree[i]
对于 B 中的所有 i，Cov[i]=false
while (对于 A 中的某些 i,New[i]>0) {
    设 v 是具有最大的 New[i]的顶点;
    C[m++]=v;
    for ( 所有邻接于 v 的顶点 j) {
      if (!Cov[j]) {
        Cov[j]= true;
        对于所有邻接于 j 的顶点，使其 New[k]减 1
      } } }
if (有些顶点未被覆盖) 失败
else 找到一个覆盖
```

图 1-8　图 1-7 的细化

更新 New 的时间为 O (e)，其中 e 为二分图中边的数目。若使用邻接矩阵，则需花(n_2) 的时间来寻找图中的边，若用邻接链表，则需(n+e) 的时间。实际更新时间根据描述方法的不同为 O (n_2) 或 O (n+e)。逐步选择顶点所需时间为($SizeOfA$)，其中 $SizeOfA$=| A |。因为 A 的所有顶点都有可能被选择，因此所需步骤数为 O ($SizeOfA$)，覆盖算法总的复杂性为 O ($SizeOfA_2$+n_2) = O (n_2)或 O ($SizeOfA_2$+n + e)。

2. 降低复杂性

通过使用有序数组 New_i、最大堆或最大选择树（max selection tree）可将每步选取顶点 v 的复杂性降为(1)。但利用有序数组，在每步的最后需对 New_i 值进行重新排序。若使用箱子排序，则这种排序所需时间为($SizeOfB$) ($SizeOfB$ =|B|) （见 3.8.1 节箱子排序）。由于一般 $SizeOfB$ 比 $SizeOfA$ 大得多，因此有序数组并不总能提高性能。

如果利用最大堆，则每一步都需要重建堆来记录 New 值的变化，可以在每次 New 值减 1 时进行重建。这种减法操作可引起被减的 New 值最多在堆中向下移一层，因此这种重建对于每次 New 值减 1 需(1)的时间，总共的减操作数目为 O (e)。因此在算法的所有步骤中，维持最大堆仅需 O (e)的时间，因而利用最大堆时覆盖算法的总复杂性为 O (n_2)或 O (n+e)。

若利用最大选择树，每次更新 New 值时需要重建选择树，所需时间为(log $SizeOfA$)。重建的最好时机是在每步结束时，而不是在每次 New 值减 1 时，需要重建的次数为 O (e)，因此总的重建时间为 O (e log $SizeOfA$)，这个时间比最大堆的重建时间长一些。然而，通过维持具有相同 New 值的顶点箱子，也可获得和利用最大堆时相同的时间限制。由于 New 的取值范围为 0 到 $SizeOfB$，需要 $SizeOfB$+ 1 个箱子，箱子 i 是一个双向链表，链接所有 New 值为 i 的顶点。在某一步结束时，假如 New[6]从 12 变到 4，则需要将它从第 12 个箱子移到第 4 个箱子。利用模拟指针及一个节点数组 node（其中 node[i]代表顶点 i，node[i].left 和

node[i].right为双向链表指针），可将顶点6从第12个箱子移到第4个箱子，从第12个箱子中删除node[0]并将其插入第4个箱子。利用这种箱子模式，可得覆盖启发式算法的复杂性为O(n^2)或O($n+e$)。（取决于利用邻接矩阵还是线性表来描述图）。

3. 双向链接箱子的实现

为了实现上述双向链接箱子，图1-9定义了类Undirected的私有成员。NodeType是一个具有私有整型成员left和right的类，它的数据类型是双向链表节点，程序13-3给出了Undirected的私有成员的代码。

```
void CreateBins (int b, int n)
创建 b 个空箱子和 n 个节点
void DestroyBins() { delete [] node;
delete [] bin;}
void InsertBins(int b, int v)
在箱子 b 中添加顶点 v
void MoveBins(int bMax, int ToBin, int v)
从当前箱子中移动顶点 v 到箱子 ToBin
int *bin;
bin[i]指向代表该箱子的双向链表的首节点
NodeType *node;
node[i]代表存储顶点 i 的节点
```

图1-9 实现双向链接箱子所需的Undirected私有成员

程序13-3 箱子函数的定义

```
void Undirected::CreateBins(int b, int n)
{// 创建 b 个空箱子和 n 个节点
node = new NodeType [n+1];
bin = new int [b+1];
// 将箱子置空
for (int i = 1; i <= b; i++)
bin[i] = 0;
}
void Undirected::InsertBins(int b, int v)
{// 若 b 不为 0，则将 v 插入箱子 b
if (!b) return; // b 为 0，不插入
node[v].left = b; // 添加在左端
if (bin[b]) node[bin[b]].left = v;
node[v].right = bin[b];
bin[b] = v;
}
void Undirected::MoveBins(int bMax, int ToBin, int v)
{// 将顶点 v 从其当前所在箱子移动到 ToBin.
// v 的左、右节点
int l = node[v].left;
int r = node[v].right;
```

```
// 从当前箱子中删除
if (r) node[r].left = node[v].left;
if (l > bMax || bin[l] != v) // 不是最左节点
node[l].right = r;
else bin[l] = r; // 箱子 l 的最左边
// 添加到箱子 ToBin
InsertBins( ToBin, v);
}
```

函数 CreateBins 动态分配两个数组： node 和 bin，node[*i*]表示顶点 *i*, bin[*i*]指向其 New 值为 *i* 的双向链表的顶点, for 循环将所有双向链表置为空。如果 b≠0，函数 InsertBins 将顶点 v 插入箱子 b 中。因为 b 是顶点 v 的 New 值，b＝0 意味着顶点 v 不能覆盖 B 中当前还未被覆盖的任何顶点，所以，在建立覆盖时这个箱子没有用处，故可以将其舍去。当 $b \neq 0$ 时，顶点 n 加入 New 值为 b 的双向链表箱子的最前面,这种加入方式需要将 node[v] 加入 bin[b] 中第一个节点的左边。由于表的最左节点应指向它所属的箱子，因此将它的 node[v].left 置为 b。若箱子不空，则当前第一个节点的 left 指针被置为指向新节点。node[v] 的右指针被置为 bin[b]，其值可能为 0 或指向上一个首节点的指针。最后， bin[b]被更新为指向表中新的第一个节点。MoveBins 将顶点 v 从它在双向链表中的当前位置移到 New 值为 ToBin 的位置上。其中存在 bMax，使得对所有的箱子 bin[*j*]都有：如 j>bMax，则 bin[j]为空。代码首先确定 node[v]在当前双向链表中的左右节点，接着从双链表中取出 node[v]，并利用 InsertBins 函数将其重新插入到 bin[ToBin]中。

4. Undirected::BipartiteCover 的实现

函数的输入参数 L 用于分配图中的顶点（分配到集合 *A* 或 *B*）。L[*i*]＝1 表示顶点 i 在集合 *A* 中，L[*i*]＝2 则表示顶点在 B 中。函数有两个输出参数： C 和 m，m 为所建立的覆盖的大小，C[0, m - 1]是 *A* 中形成覆盖的顶点。若二分图没有覆盖，函数返回 false; 否则返回 true。完整的代码见程序 13 - 4。

程序 13-4 构造贪婪覆盖

```
bool Undirected::BipartiteCover(int L[], int C[], int& m)
{// 寻找一个二分图覆盖
// L 是输入顶点的标号, L[i] = 1 当且仅当 i 在 A 中
// C 是一个记录覆盖的输出数组
// 如果图中不存在覆盖，则返回 false
// 如果图中有一个覆盖，则返回 true;
// 在 m 中返回覆盖的大小; 在 C[0:m - 1]中返回覆盖
int n = Vertices( );
// 插件结构
int SizeOfA = 0;
for (int i = 1; i <= n; i++) // 确定集合 A 的大小
if (L[i] == 1) SizeOfA++;
int SizeOfB = n - SizeOfA;
CreateBins(SizeOfB, n);
int *New = new int [n+1]; //顶点 i 覆盖了 B 中 New[i]个未被覆盖的顶点
bool *Change = new bool [n+1]; // Change[i]为 true 当且仅当 New[i] 已改变
bool *Cov = new bool [n+1]; // Cov[i] 为 true 当且仅当顶点 i 被覆盖
InitializePos( );
```

```
LinkedStack<int> S;
// 初始化
for (i = 1; i <= n; i++) {
Cov[i] = Change[i] = false;
if (L[i] == 1) {// i 在 A 中
New[i] = Degree(i); // i 覆盖了这么多
InsertBins(New[i], i);}}
// 构造覆盖
int covered = 0, // 被覆盖的顶点
MaxBin = SizeOfB; // 可能非空的最大箱子
m = 0; // C 的游标
while (MaxBin > 0) { // 搜索所有箱子
// 选择一个顶点
if (bin[MaxBin]) { // 箱子不空
int v = bin[MaxBin]; // 第一个顶点
C[m++] = v; // 把 v 加入覆盖
// 标记新覆盖的顶点
int j = Begin(v), k;
while (j) {
if (!Cov[j]) {// j 尚未被覆盖
Cov[j] = true;
c o v e r e d + + ;
// 修改 N e w
k = Begin(j);
while (k) {
New[k]--; // j 不计入在内
if (!Change[k]) {
S.Add(k); // 仅入栈一次
Change[k] = true;}
k = NextVe r t e x ( j ) ; }
}
j = NextVe r t e x ( v ) ; }
// 更新箱子
while (!S.IsEmpty()) {
S . D e l e t e ( k ) ;
Change[k] = false;
MoveBins(SizeOfB, New[k], k);}
}
else MaxBin--;
}
D e a c t i v a t e P o s ( ) ;
D e s t r o y B i n s ( ) ;
delete [] New;
delete [] Change;
```

```
delete [] Cov;
return (covered == SizeOfB);
}
```

程序 13-4 首先计算出集合 *A* 和 *B* 的大小、初始化必要的双向链表结构、创建三个数组、初始化图遍历器、并创建一个栈。然后将数组 Cov 和 Change 初始化为 false，并将 *A* 中的顶点根据它们覆盖 *B* 中顶点的数目插入到相应的双向链表中。

为了构造覆盖，首先按 SizeOfB 递减至 1 的顺序检查双向链表。当发现一个非空的表时，就将其第一个顶点 v 加入到覆盖中，这种策略即为选择具有最大 New 值的顶点。将所选择的顶点加入覆盖数组 C 并检查 B 中所有与它邻接的顶点。若顶点 j 与 v 邻接且还未被覆盖，则将 Cov[j]置为 true，表示顶点 j 现在已被覆盖，同时将已被覆盖的 *B* 中的顶点数目加 1。由于 j 是最近被覆盖的，所有 *A* 中与 j 邻接的顶点的 New 值减 1。下一个 while 循环降低这些 New 值并将 New 值被降低的顶点保存在一个栈中。当所有与顶点 v 邻接的顶点的 Cov 值更新完毕后，New 值反映了 *A* 中每个顶点所能覆盖的新的顶点数，然而 A 中的顶点由于 New 值被更新，处于错误的双向链表中，下一个 while 循环则将这些顶点移到正确的表中。

1.3.5 单源最短路径

在这个问题中，给出有向图 *G*，它的每条边都有一个非负的长度（耗费） a[*i*][*j*]，路径的长度即为此路径所经过的边的长度之和。对于给定的源顶点 s，需找出从它到图中其他任意顶点（称为目的）的最短路径。图 13-10a 给出了一个具有五个顶点的有向图，各边上的数即为长度。假设源顶点 s 为 1，从顶点 1 出发的最短路径按路径长度顺序列在图 13-10b 中，每条路径前面的数字为路径的长度。

利用 E. Dijkstra 发明的贪婪算法可以解决最短路径问题，它通过分步方法求出最短路径。每一步产生一个到达新的目的顶点的最短路径。下一步所能达到的目的顶点通过如下贪婪准则选取：在还未产生最短路径的顶点中，选择路径长度最短的目的顶点。也就是说， Dijkstra 的方法按路径长度顺序产生最短路径。

首先最初产生从 s 到它自身的路径，这条路径没有边，其长度为 0。在贪婪算法的每一步中，产生下一个最短路径。一种方法是在目前已产生的最短路径中加入一条可行的最短的边，结果产生的新路径是原先产生的最短路径加上一条边。这种策略并不总是起作用。另一种方法是在目前产生的每一条最短路径中，考虑加入一条最短的边，再从所有这些边中先选择最短的，这种策略即是 Dijkstra 算法。

可以验证按长度顺序产生最短路径时，下一条最短路径总是由一条已产生的最短路径加上一条边形成。实际上，下一条最短路径总是由已产生的最短路径再扩充一条最短的边得到的，且这条路径所到达的顶点其最短路径还未产生。例如在图 13-10 中，b 中第二条路径是第一条路径扩充一条边形成的；第三条路径则是第二条路径扩充一条边；第四条路径是第一条路径扩充一条边；第五条路径是第三条路径扩充一条边。

通过上述观察可用一种简便的方法来存储最短路径。可以利用数组 p，p[i]给出从 s 到达 i 的路径中顶点 i 前面的那个顶点。在本例中 p[1:5]=[0,1,1,3,4]。从 s 到顶点 i 的路径可反向创建。从 i 出发按 p[i],p[p[i]],p[p[p[i]]], .的顺序，直到到达顶点 s 或 0。在本例中，如果从 i=5 开始，则顶点序列为 p[i]=4, p[4]=3, p[3]=1=s，因此路径为 1,3,4,5。

为能方便地按长度递增的顺序产生最短路径，定义 d[i]为在已产生的最短路径中加入一条最短边的长度，从而使得扩充的路径到达顶点 i。最初，仅有从 s 到 s 的一条长度为 0 的路径，这时对于每个顶点 i，d[i]等于 a[s][i]（a 是有向图的长度邻接矩阵）。为产生下一条路径，需要选择还未产生最短路径的下一个节点，在这些节点中 d 值最小的即为下一条路径的终点。当获得一条新的最短路径后，由于新的最短路径可能会产生更小的 d 值，因此有些顶点的 d 值可能

会发生变化。

综上所述，可以得到图 1 3 - 11 所示的伪代码， 1) 将与 s 邻接的所有顶点的 p 初始化为 s，这个初始化用于记录当前可用的最好信息。也就是说，从 s 到 i 的最短路径，即是由 s 到它自身那条路径再扩充一条边得到。当找到更短的路径时， p [i]值将被更新。若产生了下一条最短路径，需要根据路径的扩充边来更新 d 的值。

1) 初始化 $d[i]=a[s][i]$（$1 \leqslant i \leqslant n$），

对于邻接于 s 的所有顶点 i，置 $p[i]=s$，对于其余的顶点置 $p[i]=0$；

对于 $p[i] \neq 0$ 的所有顶点建立 L 表。

2) 若 L 为空，终止，否则转至 3)。

3) 从 L 中删除 d 值最小的顶点。

4) 对于与 i 邻接的所有还未到达的顶点 j，更新 $d[j]$值为 $min\{d[j], d[i]+a[i][j]\}$；若 $d[j]$发生了变化且 j 还未在 L 中，则置 $p[j]=1$，并将 j 加入 L，转至 2。

图 1 - 11 最短路径算法的描述

1. 数据结构的选择

我们需要为未到达的顶点列表 L 选择一个数据结构。从 L 中可以选出 d 值最小的顶点。如果 L 用最小堆（见 9 . 3 节）来维护，则这种选取可在对数时间内完成。由于 3) 的执行次数为 O (n)，所以所需时间为 O (n l o g n)。由于扩充一条边产生新的最短路径时，可能使未到达的顶点产生更小的 d 值，所以在 4) 中可能需要改变一些 d 值。虽然算法中的减操作并不是标准的最小堆操作，但它能在对数时间内完成。由于执行减操作的总次数为： O(有向图中的边数)= O (n_2)，因此执行减操作的总时间为 O (n_2l o g n)。

若 L 用无序的链表来维护，则 3) 与 4) 花费的时间为 O (n_2), 3) 的每次执行需 O(|L |) =O(n) 的时间，每次减操作需(1)的时间（需要减去 d[j] 的值，但链表不用改变）。利用无序链表将图 1 - 11 的伪代码细化为程序 1 3 - 5，其中使用了 C h a i n (见程序 3 - 8)和 C h a i n I t e r a t o r 类（见程序 3 - 1 8 ）。

程序 13-5 最短路径程序

```
template<class T>
void AdjacencyWDigraph<T>::ShortestPaths(int s, T d[], int p[])
{// 寻找从顶点 s 出发的最短路径，在 d 中返回最短距离
// 在 p 中返回前继顶点
if (s < 1 || s > n) throw OutOfBounds();
Chain<int> L; // 路径可到达顶点的列表
ChainIterator<int> I;
// 初始化 d, p, L
for (int i = 1; i <= n; i++){
d[i] = a[s][i];
if (d[i] == NoEdge) p[i] = 0;
else {p[i] = s;
L . I n s e r t ( 0 , i ) ; }
}
// 更新 d, p
while (!L.IsEmpty()) {// 寻找具有最小 d 的顶点 v
int *v = I.Initialize(L);
int *w = I.Next();
```

```
while (w) {
if (d[*w] < d[*v]) v = w;
w = I.Next();}
// 从 L 中删除通向顶点 v 的下一最短路径并更新 d
int i = *v;
L . D e l e t e ( * v ) ;
for (int j = 1; j <= n; j++) {
if (a[i][j] != NoEdge && (!p[j] ||
d[j] > d[i] + a[i][j])) {
// 减小 d [ j ]
d[j] = d[i] + a[i][j];
// 将 j 加入 L
if (!p[j]) L.Insert(0,j);
p[j] = i;}
}
}
}
```

若 N o E d g e 足够大，使得没有最短路径的长度大于或等于 N o E d g e，则最后一个 for 循环的 i f 条件可简化为：if (d[j] > d[i] + a[i][j]) NoEdge 的值应在能使 d[j]+a[i][j] 不会产生溢出的范围内。

2. 复杂性分析

程序 1 3 - 5 的复杂性是 O (n_2)，任何最短路径算法必须至少对每条边检查一次，因为任何一条边都有可能在最短路径中。因此这种算法的最小可能时间为 O (e)。由于使用耗费邻接矩阵来描述图，仅决定哪条边在有向图中就需 O (n_2)的时间。因此，采用这种描述方法的算法需花费 O (n_2)的时间。不过程序 1 3 - 5 作了优化（常数因子级）。即使改变邻接表，也只会使最后一个 f o r 循环的总时间降为 O (e)（因为只有与 i 邻接的顶点的 d 值改变）。从 L 中选择及删除最小距离的顶点所需总时间仍然是 O(n_2)。

1.3.6 最小耗费生成树

在例 1 - 2 及 1 - 3 中已考察过这个问题。因为具有 *n* 个顶点的无向网络 *G* 的每个生成树刚好具有 *n*-1 条边，所以问题是用某种方法选择 *n*-1 条边使它们形成 *G* 的最小生成树。至少可以采用三种不同的贪婪策略来选择这 *n*-1 条边。这三种求解最小生成树的贪婪算法策略是： K r u s k a l 算法，P r i m 算法和 S o l l i n 算法。

1. Kruskal 算法

(1) 算法思想

K r u s k a l 算法每次选择 *n*- 1 条边，所使用的贪婪准则是：从剩下的边中选择一条不会产生环路的具有最小耗费的边加入已选择的边的集合中。注意到所选取的边若产生环路则不可能形成一棵生成树。K r u s k a l 算法分 *e* 步，其中 *e* 是网络中边的数目。按耗费递增的顺序来考虑这 *e* 条边，每次考虑一条边。当考虑某条边时，若将其加入到已选边的集合中会出现环路，则将其抛弃，否则，将它选入。

考察图 1-12a 中的网络。初始时没有任何边被选择。图 13-12b 显示了各节点的当前状态。边（ 1 , 6 ）是最先选入的边，它被加入到欲构建的生成树中，得到图 1 3 - 1 2 c。下一步选择边（ 3，4 ）并将其加入树中（如图 1 3 - 1 2 d 所示）。然后考虑边(2，7)，将它加入树中并不会产生环路，于是便得到图 1 3 - 1 2 e。下一步考虑边（ 2，3 ）并将其加入树中（如图 1 3 - 1 2 f

所示）。在其余还未考虑的边中，（7，4）具有最小耗费，因此先考虑它，将它加入正在创建的树中会产生环路，所以将其丢弃。此后将边（ 5，4）加入树中，得到的树如图 13-12g 所示。下一步考虑边（ 7，5），由于会产生环路，将其丢弃。最后考虑边（ 6，5）并将其加入树中，产生了一棵生成树，其耗费为 99。图 1-13 给出了 Kruskal 算法的伪代码。

```
//在一个具有 n 个顶点的网络中找到一棵最小生成树
令 T 为所选边的集合，初始化 T=
令 E 为网络中边的集合
while(E≠ )&&(|T|≠n-1){
令(u,v)为 E 中代价最小的边
E=E-{(u,v)}//从 E 中删除边
if((u,v)加入 T 中不会产生环路)将（ u,v）加入 T
}
if(|T|==n-1) T 是最小耗费生成树
else 网络不是互连的，不能找到生成树
```

图 13-13 Kruskao 算法的伪代码

(2) 正确性证明

利用前述装载问题所用的转化技术可以证明图 13-13 的贪婪算法总能建立一棵最小耗费生成树。需要证明以下两点： 1) 只要存在生成树，Kruskal 算法总能产生一棵生成树； 2) 产生的生成树具有最小耗费。令 G 为任意加权无向图（即 G 是一个无向网络）。从 12.11.3 节可知当且仅当一个无向图连通时它有生成树。而且在 Kruskal 算法中被拒绝（丢弃）的边是那些会产生环路的边。删除连通图环路中的一条边所形成的图仍是连通图，因此如果 G 在开始时是连通的，则 T 与 E 中的边总能形成一个连通图。也就是若 G 开始时是连通的，算法不会终止于 $E=$ 和 $|T|<n-1$。

现在来证明所建立的生成树 T 具有最小耗费。由于 G 具有有限棵生成树，所以它至少具有一棵最小生成树。令 U 为这样的一棵最小生成树， T 与 U 都刚好有 $n-1$ 条边。如果 $T=U$, 则 T 就具有最小耗费，那么不必再证明下去。因此假设 $T\neq U$，令 $k(k>0)$ 为在 T 中而不在 U 中的边的个数，当然 k 也是在 U 中而不在 T 中的边的数目。

通过把 U 变换为 T 来证明 U 与 T 具有相同的耗费，这种转化可在 k 步内完成。每一步使在 T 而不在 U 中的边的数目刚好减 1。而且 U 的耗费不会因为转化而改变。经过 k 步的转化得到的 U 将与原来的 U 具有相同的耗费，且转化后 U 中的边就是 T 中的边。由此可知， T 具有最小耗费。每步转化包括从 T 中移一条边 e 到 U 中，并从 U 中移出一条边 f。边 e 与 f 的选取按如下方式进行:

1) 令 e 是在 T 中而不在 U 中的具有最小耗费的边。由于 $k>0$，这条边肯定存在。

2) 当把 e 加入 U 时，则会形成唯一的一条环路。令 f 为这条环路上不在 T 中的任意一条边。由于 T 中不含环路，因此所形成的环路中至少有一条边不在 T 中。

从 e 与 f 的选择方法中可以看出， $V=U+\{e\}-\{f\}$ 是一棵生成树，且 T 中恰有 $k-1$ 条边不在 V 中出现。现在来证明 V 的耗费与 U 的相同。显然，V 的耗费等于 U 的耗费加上边 e 的耗费再减去边 f 的耗费。若 e 的耗费比 f 的小，则生成树 V 的耗费比 U 的耗费小，这是不可能的。如果 e 的耗费高于 f，在 Kruskal 算法中 f 会在 e 之前被考虑。由于 f 不在 T 中，Kruskal 算法在考虑 f 能否加入 T 时已将 f 丢弃，因此 f 和 T 中耗费小于或等于 f 的边共同形成环路。通过选择 e，所有这些边均在 U 中，因此 U 肯定含有环路，但是实际上这不可能，因为 U 是一棵生成树。e 的代价高于 f 的假设将会导致矛盾。剩下的唯一的可能是 e 与 f 具有相同的耗费，

由此可知 *V* 与 *U* 的耗费相同。

(3) 数据结构的选择及复杂性分析

为了按耗费非递减的顺序选择边，可以建立最小堆并根据需要从堆中一条一条地取出各边。当图中有 *e* 条边时，需花(e) 的时间初始化堆及 O(log*e*) 的时间来选取每一条边。边的集合 *T* 与 *G* 中的顶点一起定义了一个由至多 *n* 个连通子图构成的图。用顶点集合来描述每个子图，这些顶点集合没有公共顶点。为了确定边（ *u*,*v*）是否会产生环路，仅需检查 *u*,*v* 是否在同一个顶点集中（即处于同一子图）。如果是，则会形成一个环路；如果不是，则不会产生环路。因此对于顶点集使用两个 Find 操作就足够了。当一条边包含在 *T* 中时，2 个子图将被合并成一个子图，即对两个集合执行 Union 操作。集合的 Find 和 Union 操作可利用 8.10.2 节的树（以及加权规则和路径压缩）来高效地执行。Find 操作的次数最多为 2*e*，Union 操作的次数最多为 *n*-1（若网络是连通的，则刚好是 *n*-1 次）。加上树的初始化时间，算法中这部分的复杂性只比 O(*n*+*e*) 稍大一点。

对集合 *T* 所执行的唯一操作是向 *T* 中添加一条新边。*T* 可用数组 *t* 来实现。添加操作在数组的一端进行，因为最多可在 *T* 中加入 *n*-1 条边，因此对 *T* 的操作总时间为 O(*n*)。

总结上述各个部分的执行时间，可得图 13-13 算法的渐进复杂性为 O(*n*+*e*log*e*)。

(4) 实现

利用上述数据结构，图 1-13 可用 C++代码来实现。首先定义 EdgeNode 类（见程序 13-6），它是最小堆的元素及生成树数组 *t* 的数据类型。

程序 13-6 Kruskal 算法所需要的数据类型

```
template <class T>
class EdgeNode {
public:
operator T() const {return weight;}
private:
T weight;//边的高度
int u, v;//边的端点
};
```

为了更简单地使用 8.10.2 节的查找和合并策略，定义了 UnionFind 类，它的构造函数是程序 8-16 的初始化函数，Union 是程序 8-16 的加权合并函数，Find 是程序 8-17 的路径压缩搜索函数。

为了编写与网络描述无关的代码，还定义了一个新的类 UNetWork，它包含了应用于无向网络的所有函数。这个类与 Undirected 类的差别在于 Undirected 类中的函数不要求加权边，而 UNetWork 要求边必须带有权值。UNetWork 中的成员需要利用 Network 类中定义的诸如 Begin 和 NextVertex 的遍历函数。不过，新的遍历函数不仅需要返回下一个邻接的顶点，而且要返回到达这个顶点的边的权值。这些遍历函数以及有向和无向加权网络的其他函数一起构成了 WNetwork 类（见程序 13-7）。

程序 13-7 WNetwork 类

```
template<class T>
class WNetwork : virtual public Network
{
public :
virtual void First(int i, int& j, T& c)=0;
virtual void Next(int i, int& j, T& c)=0;
};
```

象Begin和NextVertex一样，可在AdjacencyWDigraph及LinkedWDigraph类中加入函数First与Next。现在AdjacencyWDigraph及LinkedWDigraph类都需要从WNetWork中派生而来。由于AdjacencyWGraph类和LinkedWGraph类需要访问UNetwork的成员，所以这两个类还必须从UNetWork中派生而来。UNetWork::Kruskal的代码见程序13-8，它要求将Edges() 定义为NetWork 类的虚成员，并且把UNetWork定义为EdgeNode的友元）。如果没有生成树，函数返回false，否则返回true。注意当返回true 时，在数组t 中返回最小耗费生成树。

程序13-8 Kruskal算法的C++代码

```
template<class T>
bool UNetwork<T>::Kruskal(EdgeNode<T> t[])
{// 使用Kruskal算法寻找最小耗费生成树
// 如果不连通则返回false
// 如果连通，则在t[0:n-2]中返回最小生成树
int n = Vertices();
int e = Edges();
//设置网络边的数组
InitializePos(); // 图遍历器
EdgeNode<T> *E = new EdgeNode<T> [e+1];
int k = 0; // E的游标
for (int i = 1; i <= n; i++) { // 使所有边附属于i
int j;
T c;
First(i, j, c);
while (j) { // j 邻接自i
if (i < j) {// 添加到达E的边
E[++k].weight = c;
E[k].u = i;
E[k].v = j;}
Next(i, j, c);
}
}
// 把边放入最小堆
MinHeap<EdgeNode<T> > H(1);
H.Initialize(E, e, e);
UnionFind U(n); // 合并/搜索结构
// 根据耗费的次序来抽取边
k = 0; // 此时作为t的游标
while (e && k < n - 1) {
// 生成树未完成，尚有剩余边
EdgeNode<T> x;
H.DeleteMin(x); // 最小耗费边
e--;
int a = U.Find(x.u);
int b = U.Find(x.v);
```

```
if (a != b) {// 选择边
t[k++] = x;
U.Union(a,b);}
}
DeactivatePos();
H.Deactivate();
return (k == n - 1);
}
```

2. Prim 算法

与 Kruskal 算法类似，Prim 算法通过每次选择多条边来创建最小生成树。选择下一条边的贪婪准则是：从剩余的边中，选择一条耗费最小的边，并且它的加入应使所有入选的边仍是一棵树。最终，在所有步骤中选择的边形成一棵树。相反，在 Kruskal 算法中所有入选的边集合最终形成一个森林。

Prim 算法从具有一个单一顶点的树 T 开始，这个顶点可以是原图中任意一个顶点。然后往 T 中加入一条代价最小的边（ u, v）使 $T \cup \{(u, v)\}$仍是一棵树，这种加边的步骤反复循环直到 T 中包含 n-1 条边。注意对于边（ u, v），u、v 中正好有一个顶点位于 T 中。Prim 算法的伪代码如图 1 -1 4 所示。在伪代码中也包含了所输入的图不是连通图的可能，在这种情况下没有生成树。图 1 - 1 5 显示了对图 1-12a 使用 Prim 算法的过程。把图 1 - 1 4 的伪代码细化为 C＋＋程序及其正确性的证明留作练习（练习 3 1）。

```
//假设网络中至少具有一个顶点
设 T 为所选择的边的集合，初始化 T=
设 TV 为已在树中的顶点的集合，置 TV={1}
令 E 为网络中边的集合
while(E<>) && (|T|<>n-1) {
令（u,v）为最小代价边，其中 u TV, v TV
if（没有这种边） break
E=E-{(u,v)} //从 E 中删除此边
在 T 中加入边（ u,v）
}
if(|T|==n-1) T 是一棵最小生成树
else 网络是不连通的，没有最小生成树
```

图 13-14 Prim 最小生成树算法

如果根据每个不在 TV 中的顶点 v 选择一个顶点 $near(v)$，使得 $near(v) \in TV$ 且 $cost(v, near(v))$的值是所有这样的 $near(v)$ 节点中最小的，则实现 Prim 算法的时间复杂性为 $O(n_2)$。下一条添加到 T 中的边是这样的边：其 $cost(v, near(v))$ 最小，且 v TV。

3. Sollin 算法

Sollin 算法每步选择若干条边。在每步开始时，所选择的边及图中的 n 个顶点形成一个生成树的森林。在每一步中为森林中的每棵树选择一条边，这条边刚好有一个顶点在树中且边的代价最小。将所选择的边加入要创建的生成树中。注意一个森林中的两棵树可选择同一条边，因此必须多次复制同一条边。当有多条边具有相同的耗费时，两棵树可选择与它们相连的不同的边，在这种情况下，必须丢弃其中的一条边。开始时，所选择的边的集合为空。若某一步结束时仅剩下一棵树或没有剩余的边可供选择时算法终止。

图 1 - 6 给出了初始状态为图 1-12a 时，使用 S o l l i n 算法的步骤。初始入选边数为 0 时的情形如图 13-12a 时,森林中的每棵树均是单个顶点。顶点 1，2，.，7 所选择的边分别是(1.6), (2,7),(3,4), (4,3), (5,4), (6,1), (7,2)，其中不同的边是(1 , 6)，(2 , 7)，(3,4) 和(5 , 4)，将这些边加入入选边的集合后所得到的结果如图 1 3 - 1 6 a 所示。下一步具有顶点集{ 1 , 6 }的树选择边(6 , 5)，剩下的两棵树选择边(2 , 3)，加入这两条边后已形成一棵生成树，构建好的生成树见图 1 3 - 6 b。S o l l i n 算法的 C + +程序实现及其正确性证明留作练习（练习 3 2)。

练习

8. 针对装载问题，扩充贪婪算法，考虑有两条船的情况，算法总能产生最优解吗？

9. 已知 n 个任务的执行序列。假设任务 i 需要 t_i 个时间单位。若任务完成的顺序为 1，2，.，n，则任务 i 完成的时间为 $c_i=\sum_{j=1}^{i}t_j$ 。任务的平均完成时间（Av e rge Completion Time, ACT）为 $1-n_n\sum_{i=1}^{n}c_i$ 。

1) 考虑有四个任务的情况，每个任务所需时间分别是（ 4，2，8，1）。若任务的顺序为 1，2，3，4，则 A C T 是多少？

2) 若任务顺序为 2，1，4，3，则 A C T 是多少？

3) 创建具有最小 A C T 的任务序列的贪婪算法分 n 步来构造该任务序列，在每一步中，从剩下的任务里选择时间最小的任务。对于 1)，利用这种策略获得的任务顺序为 4，2，1，3，这种顺序的 A C T 是多少？

4) 写一个 C + +程序实现 3) 中的贪婪策略，程序的复杂性应为 O (nl o gn)，试证明之。

5) 证明利用 3) 中的贪婪算法获得的任务顺序具有最小的 A C T。

10. 若有两个工人执行练习 9 中的 n 个任务，需将任务分配给他们，同时他们具有自己的任务执行顺序。任务完成时间及 A C T 的定义同练习 9。使 A C T 最小化的一种可行的贪婪算法是：两个工人轮流选择任务，每次从剩余的任务中选择时间最小的任务。每个人按照自己所选任务的顺序执行任务。对于练习 9 中的例子，假定工人 1 首先选择任务 4，然后工人 2 选择任务 2，工人 1 选择任务 1，最后工人 2 选择任务 3。

1) 利用 C + +程序实现这种策略，其时间复杂性为多少？

2) 上述的贪婪策略总能获得最小的 A C T 吗？证明结论。

11. 1) 考虑有 m 个人可以执行任务，扩充练习 1 0 中的贪婪算法。

2) 算法能保证获得最优解吗？证明结论。

3) 用 C + +程序实现此算法，其复杂性是多少？

12. 考虑例 4 - 4 的堆栈折叠问题。

1) 设计一个贪婪算法，将堆栈折叠为最小数目的子堆栈，使得每个子堆栈的高度均不超过 H。

2) 算法总能保证得到数目最少的子堆栈吗？证明结论。

3) 用 C + +代码实现 1) 的算法。

4) 代码的时间复杂性是多少？

13. 编写 C + +程序实现 0 / 1 背包问题，使用如下启发式方法：按价值密度非递减的顺序打包。

14. 根据 k= 1 的性能受限启发式方法编写一个 C + +程序来实现 0 / 1 背包问题。

15. 对于 k= 1 的情况证明用性能受限的启发式方法求解 0 / 1 背包问题会发生边界错误。

16. 根据 k= 2 的性能受限启发式方法编写一个 C + +程序来实现 0 / 1 背包问题。

17. 考虑 $0\leqslant x_i\leqslant 1$ 而不是 $x_i\in\{0,1\}$的连续背包问题。一种可行的贪婪策略是：按价值密度非递减的顺序检查物品，若剩余容量能容下正在考察的物品，将其装入；否则，往背包中装入此物品的一部分。

1) 对于 n=3, w=[100,10,10], p= [2 0 , 1 5 , 1 5]及 c= 1 0 5 ,上述装入方法获得的结果是什么？

2) 证明这种贪婪算法总能获得最优解。

3) 用一个 C + +程序实现此算法。

18. 例 13-1 的渴婴问题是练习 17 中连续背包问题的一般化，将练习 17 的贪婪算法用于渴婴问题，算法能保证总能得到最优解吗？证明结论。

19. 1) 证明当且仅当二分图没有覆盖时，图 13-7 的算法找不到覆盖。

2) 给出一个具有覆盖的二分图，使得图 13-7 的算法找不到最小覆盖。

20. 当第一步选择了顶点 1 时，给出图 13-7 的工作过程。

21. 对于二分图覆盖问题设计另外一种贪婪启发式方法，可使用如下贪婪准则：如果 *B* 中的某一个顶点仅被 *A* 中一个顶点覆盖，选择 *A* 中这个顶点；否则，从 *A* 中选择一个顶点，使得它所覆盖的未被覆盖的顶点数目最多。

1) 给出这种贪婪算法的伪代码。

2) 编写一个 C++函数作为 Undirected 类的成员来实现上述贪婪算法。

3) 函数的复杂性是多少？

4) 验证代码的正确性。

22. 令 *G* 为无向图，*S* 为 *G* 中顶点的子集，当且仅当 *S* 中的任意两个顶点都有一条边相连时，*S* 为完备子图（clique），完备子图的大小即 *S* 中的顶点数目。最大完备子图（ maximum clique）即具有最大项点数目的完备子图。在图中寻找最大完备子图的问题（即最大完备子图问题）是一个 NP-复杂问题。

1) 给出最大完备子图问题的一种可行的贪婪算法及其伪代码。

2) 给出一个能用 1) 中的启发式算法求解最大完备子图的图例，以及不能用该算法求解的一个图例。

3) 将 1) 中的启发式算法实现为 Undirected::Clique(int C, int m) 共享成员，其中最大完备子图的大小返回到 *m* 中，最大完备子图的顶点返回到 C 中。

4) 代码的复杂性是多少？

23. 令 *G* 为一无向图，*S* 为 *G* 中顶点的子集，当且仅当 *S* 中任意两个顶点都无边相连时， *S* 为无关集（independent set）。最大无关集即是顶点数目最多的无关集。在一幅图中寻找最大无关集是一个 NP-复杂问题。按练习 22 的要求解决最大无关集问题。

24. 对无向图 *G* 着色的方法是：为 *G* 中的顶点编号（{1,2,.}），使得由一条边相连的两个顶点具有不同的编号。在图的着色问题中，要求利用最少的相互不同的颜色（编号）来给图 *G* 着色。图的着色问题也是一个 NP-复杂问题。按练习 22 的要求解决图着色问题。

25. 证明当按路径长度的顺序产生一条最短路径时，所产生的下一条最短路径总是由已产生的一条最短路径扩充一条边得到。

26. 证明对于具有一条或多条具有负长度的边，图 13-11 的贪婪算法不一定能正确地计算出最短路径的长度。

27. 编写一个 Path(p,s,i)函数，利用函数 ShortestPaths 计算出的 p 值，输出从顶点 s 到顶点 i 的一条最短路径。函数的复杂性是多少？

28. 若把有向图作为 LinkedwDigraph 类的一个成员，重写程序 13-5，函数应作为该类的一个成员。函数的复杂性是多少？

29. 若把有向图作为 LinkedWDigraph 类的一个成员且仅有 O(*n*)条边，重写程序 13-5，*L* 用最小堆来实现。函数的复杂性是多少？

30. 从 Network 类（见程序 12-15）派生出一个新的模板类 DNetwork（有向网络），这个类仅包含应用于有向网络的所有函数。为该类定义一个 ShortestPaths 函数，使得它与有向网络的描述形式无关，尤其适用于耗费邻接矩阵及邻接链表描述方法。在函数的实现过程中可利用原来的遍历函数，也可根据需要定义新的遍历函数。函数的复杂性应为 $O(n^2)$，其中 *n* 是顶点的数目，试证明之。

*31. 1) 给出 Prim 算法（如图 13-14 所示）的一种正确性证明。

2) 将图 13-14 细化为一个 C++程序 UNetwork::Prim，其复杂性应为 O(n_2)。
3) 证明程序的复杂性确实是 O(n_2)。
*32. 1) 证明对于任意连通无向图，Sollin 算法总能找到一个最小耗费生成树。
2) 在 Sollin 算法中，最大的步骤数是多少？试用图中顶点数 n 来表示。
3) 编写一个 C++程序 UNetwork::Sollin，使用 Sollin 算法找到一棵最小生成树。
4) 程序的复杂性是多少？
*33. 令 T 为一棵每条边均带有长度的树（不一定是二叉树）。令 S 为 T 中顶点的子集，并令 T/S 为从 T 中删除 S 中的顶点所得到的森林。我们希望能找到具有最小走势的子集 S，使得 T/S 中没有从根到叶的距离大于 d 的森林。
1) 给出一种寻找最小走势子集 S 的贪婪算法（提示：从叶节点开始向根移动）。
2) 证明算法的正确性。
3) 算法的复杂性是多少？如果它不是 T 中顶点数的线性函数，则重新设计算法，使其复杂性是线性的。
*34. 令 T/S 表示将 S 中的每个顶点复制两份而获得的森林，其中父节点的指针指向一个复本，而另一复本的指针指向其儿子。针对这种情况再做练习 33。

（ 说明：本资料是根据《数据结构、算法与应用》（美，Sartaj Sahni 著）一书第 13-17 章编辑、改写的。考虑到因特网传输速度等因素，大部分插图和公式不得不被删除。对于内容不连贯之处，请网友或读者参阅该书，敬请原谅。 ）

第 2 章 分而治之算法

君主和殖民者们所成功运用的分而治之策略也可以运用到高效率的计算机算法的设计过程中。本章将首先介绍怎样在算法设计领域应用这一古老的策略，然后将利用这一策略解决如下问题：最小最大问题、矩阵乘法、残缺棋盘、排序、选择和一个计算几何问题——找出二维空间中距离最近的两个点。

本章给出了用来分析分而治之算法复杂性的数学方法，并通过推导最小最大问题和排序问题的复杂性下限来证明分而治之算法对于求解这两种问题是最优的（因为算法的复杂性与下限一致）。

2.1 算法思想

分而治之方法与软件设计的模块化方法非常相似。为了解决一个大的问题，可以： 1) 把它分成两个或多个更小的问题； 2) 分别解决每个小问题； 3) 把各小问题的解答组合起来，即可得到原问题的解答。小问题通常与原问题相似，可以递归地使用分而治之策略来解决。

例 2-1 [找出伪币] 给你一个装有 16 个硬币的袋子。16 个硬币中有一个是伪造的，并且那个伪造的硬币比真的硬币要轻一些。你的任务是找出这个伪造的硬币。为了帮助你完成这一任务，将提供一台可用来比较两组硬币重量的仪器，利用这台仪器，可以知道两组硬币的重量是否相同。比较硬币 1 与硬币 2 的重量。假如硬币 1 比硬币 2 轻，则硬币 1 是伪造的；假如硬币 2 比硬币 1 轻，则硬币 2 是伪造的。这样就完成了任务。假如两硬币重量相等，则比较硬币 3 和硬币 4。同样，假如有一个硬币轻一些，则寻找伪币的任务完成。假如两硬币重量相等，则继续比较硬币 5 和硬币 6。按照这种方式，可以最多通过 8 次比较来判断伪币的存在并找出这一伪币。

另外一种方法就是利用分而治之方法。假如把 16 硬币的例子看成一个大的问题。第一步，

把这一问题分成两个小问题。随机选择 8 个硬币作为第一组称为 *A* 组，剩下的 8 个硬币作为第二组称为 *B* 组。这样，就把 16 个硬币的问题分成两个 8 硬币的问题来解决。第二步，判断 *A* 和 *B* 组中是否有伪币。可以利用仪器来比较 *A* 组硬币和 *B* 组硬币的重量。假如两组硬币重量相等，则可以判断伪币不存在。假如两组硬币重量不相等，则存在伪币，并且可以判断它位于较轻的那一组硬币中。最后，在第三步中，用第二步的结果得出原先 16 个硬币问题的答案。若仅仅判断硬币是否存在，则第三步非常简单。无论 *A* 组还是 *B* 组中有伪币，都可以推断这 16 个硬币中存在伪币。因此，仅仅通过一次重量的比较，就可以判断伪币是否存在。

现在假设需要识别出这一伪币。把两个或三个硬币的情况作为不可再分的小问题。注意如果只有一个硬币，那么不能判断出它是否就是伪币。在一个小问题中，通过将一个硬币分别与其他两个硬币比较，最多比较两次就可以找到伪币。这样，16 硬币的问题就被分为两个 8 硬币（*A* 组和 *B* 组）的问题。通过比较这两组硬币的重量，可以判断伪币是否存在。如果没有伪币，则算法终止。否则，继续划分这两组硬币来寻找伪币。假设 *B* 是轻的那一组，因此再把它分成两组，每组有 4 个硬币。称其中一组为 *B*1，另一组为 *B*2。比较这两组，肯定有一组轻一些。如果 *B*1 轻，则伪币在 *B*1 中，再将 *B*1 又分成两组，每组有两个硬币，称其中一组为 *B*1*a*，另一组为 *B*1*b*。比较这两组，可以得到一个较轻的组。由于这个组只有两个硬币，因此不必再细分。比较组中两个硬币的重量，可以立即知道哪一个硬币轻一些。较轻的硬币就是所要找的伪币。

例 2-2 [金块问题] 有一个老板有一袋金块。每个月将有两名雇员会因其优异的表现分别被奖励一个金块。按规矩，排名第一的雇员将得到袋中最重的金块，排名第二的雇员将得到袋中最轻的金块。根据这种方式，除非有新的金块加入袋中，否则第一名雇员所得到的金块总是比第二名雇员所得到的金块重。如果有新的金块周期性的加入袋中，则每个月都必须找出最轻和最重的金块。假设有一台比较重量的仪器，我们希望用最少的比较次数找出最轻和最重的金块。

假设袋中有 *n* 个金块。可以用函数 Max（程序 1-31）通过 *n*-1 次比较找到最重的金块。找到最重的金块后，可以从余下的 *n*-1 个金块中用类似的方法通过 *n*-2 次比较找出最轻的金块。这样，比较的总次数为 2*n*-3。程序 2-26 和 2-27 是另外两种方法，前者需要进行 2*n*-2 次比较，后者最多需要进行 2*n*-2 次比较。

下面用分而治之方法对这个问题进行求解。当 *n* 很小时，比如说， $n \leqslant 2$，识别出最重和最轻的金块，一次比较就足够了。当 *n* 较大时（$n > 2$），第一步，把这袋金块平分成两个小袋 *A* 和 *B*。第二步，分别找出在 *A* 和 *B* 中最重和最轻的金块。设 *A* 中最重和最轻的金块分别为 H_A 与 L_A，以此类推，*B* 中最重和最轻的金块分别为 H_B 和 L_B。第三步，通过比较 H_A 和 H_B，可以找到所有金块中最重的；通过比较 L_A 和 L_B，可以找到所有金块中最轻的。在第二步中，若 $n > 2$，则递归地应用分而治之方法。

假设 $n = 8$。这个袋子被平分为各有 4 个金块的两个袋子 *A* 和 *B*。为了在 *A* 中找出最重和最轻的金块，*A* 中的 4 个金块被分成两组 *A*1 和 *A*2。每一组有两个金块，可以用一次比较在 *A* 中找出较重的金块 H_{A1} 和较轻的金块 L_{A1}。经过另外一次比较，又能找出 H_{A2} 和 L_{A2}。现在通过比较 H_{A1} 和 H_{A2}，能找出 H_A；通过 L_{A1} 和 L_{A2} 的比较找出 L_A。这样，通过 4 次比较可以找到 H_A 和 L_A。同样需要另外 4 次比较来确定 H_B 和 L_B。通过比较 H_A 和 H_B（L_A 和 L_B），就能找出所有金块中最重和最轻的。因此，当 $n = 8$ 时，这种分而治之的方法需要 10 次比较。如果使用程序 1-31，则需要 13 次比较。如果使用程序 2-26 和 2-27，则最多需要 14 次比较。设 $c(n)$ 为使用分而治之方法所需要的比较次数。为了简便，假设 *n* 是 2 的幂。当 $n = 2$ 时，$c(n) = 1$。对于较大的 *n*，$c(n) = 2c(n/2) + 2$。当 *n* 是 2 的幂时，使用迭代方法（见例 2-20）可知
$c(n) = 3n/2 - 2$。在本例中，使用分而治之方法比逐个比较的方法少用了 25% 的比较次数。

例 2-3 [矩阵乘法] 两个 $n \times n$ 阶的矩阵 *A* 与 *B* 的乘积是另一个 $n \times n$ 阶矩阵 *C*，*C* 可表示为假如每一个 $C(i, j)$ 都用此公式计算，则计算 *C* 所需要的操作次数为 $n_3 m + n_2 (n-1) a$，其中 *m* 表示一次乘法，*a* 表示一次加法或减法。

为了得到两个矩阵相乘的分而治之算法，需要： 1) 定义一个小问题，并指明小问题是如何进行乘法运算的； 2) 确定如何把一个大的问题划分成较小的问题，并指明如何对这些较小的问题进行乘法运算； 3) 最后指出如何根据小问题的结果得到大问题的结果。为了使讨论简便，假设 n 是 2 的幂（也就是说， n 是 1，2，4，8，16，.）。

首先，假设 $n=1$ 时是一个小问题，$n>1$ 时为一个大问题。后面将根据需要随时修改这个假设。对于 1×1 阶的小矩阵，可以通过将两矩阵中的两个元素直接相乘而得到结果。

考察一个 $n>1$ 的大问题。可以将这样的矩阵分成 4 个 $n/2 \times n/2$ 阶的矩阵 A_1，A_2，A_3，和 A_4。当 n 大于 1 且 n 是 2 的幂时，$n/2$ 也是 2 的幂。因此较小矩阵也满足前面对矩阵大小的假设。矩阵 B_i 和 C_i 的定义与此类似.

根据上述公式，经过 8 次 $n/2 \times n/2$ 阶矩阵乘法和 4 次 $n/2 \times n/2$ 阶矩阵的加法，就可以计算出 A 与 B 的乘积。因此，这些公式能帮助我们实现分而治之算法。在算法的第二步，将递归使用分而治之算法把 8 个小矩阵再细分（见程序 2-19）。算法的复杂性为(n_3)，此复杂性与程序 2-24 直接使用公式（2-1）所得到的复杂性是一样的。事实上，由于矩阵分割和再组合所花费的额外开销，使用分而治之算法得出结果的时间将比用程序 2-24 还要长。

为了得到更快的算法，需要简化矩阵分割和再组合这两个步骤。一种方案是使用 Strassen 方法得到 7 个小矩阵。这 7 个小矩阵为矩阵 $D, E, ., J$，矩阵 D 到 J 可以通过 7 次矩阵乘法， 6 次矩阵加法，和 4 次矩阵减法计算得出。前述的 4 个小矩阵可以由矩阵 D 到 J 通过 6 次矩阵加法和两次矩阵减法得出.

用上述方案来解决 $n=2$ 的矩阵乘法。将某矩阵 A 和 B 相乘得结果 C，如下所示:

因为 $n>1$，所以将 A、B 两矩阵分别划分为 4 个小矩阵，每个矩阵为 1×1 阶，仅包含一个元素。1×1 阶矩阵的乘法为小问题，因此可以直接进行运算。利用计算 $D \sim J$ 的公式，得:

$D=1(6-8)=-2$

$E=4(7-5)=8$

$F=(3+4)5=35$

$G=(1+2)8=24$

$H=(3-1)(5+6)=22$

$I=(2-4)(7+8)=-30$

$J=(1+4)(5+8)=65$

根据以上结果可得:

对于上面这个 2×2 的例子，使用分而治之算法需要 7 次乘法和 18 次加/减法运算。而直接使用公式（2-1），则需要 8 次乘法和 7 次加/减法。要想使分而治之算法更快一些，则一次乘法所花费的时间必须比 11 次加/减法的时间要长。

假定 Strassen 矩阵分割方案仅用于 $n \geqslant 8$ 的矩阵乘法，而对于 $n<8$ 的矩阵乘法则直接利用公式（2-1）进行计算。则 $n=8$ 时，8×8 矩阵相乘需要 7 次 4×4 矩阵乘法和 18 次 4×4 矩阵加/减法。每次矩阵乘法需花费 $64m+48a$ 次操作，每次矩阵加法或减法需花费 $16a$ 次操作。因此总的操作次数为 $7(64m+48a)+18(16a)=448m+624a$。而使用直接计算方法，则需要 $512m+448a$ 次操作。要使 Strassen 方法比直接计算方法快，至少要求 512-448 次乘法的开销比 624-448 次加/减法的开销大。或者说一次乘法的开销应该大于近似 2.75 次加/减法的开销。

假定 $n<16$ 的矩阵是一个“小”问题，Strassen 的分解方案仅仅用于 $n \geqslant 16$ 的情况，对于 $n<16$ 的矩阵相乘，直接利用公式（ 2-1）。则当 $n=16$ 时使用分而治之算法需要 $7(512m+448a)+18(64a)=3584m+4288a$ 次操作。直接计算时需要 $4096m+3840a$ 次操作。若一次乘法的开销与一次加/减法的开销相同，则 Strassen 方法需要 7872 次操作及用于问题分解的额外时间，而直接计算方法则需要 7936 次操作加上程序中执行 for 循环以及其他语句所

花费的时间。即使直接计算方法所需要的操作次数比 Strassen 方法少，但由于直接计算方法需要更多的额外开销，因此它也不见得会比 Strassen 方法快。

n 的值越大，Strassen 方法与直接计算方法所用的操作次数的差异就越大，因此对于足够大的 n，Strassen 方法将更快。设 $t(n)$ 表示使用 Strassen 分而治之方法所需的时间。因为大的矩阵会被递归地分割成小矩阵直到每个矩阵的大小小于或等于 k（k 至少为 8，也许更大，具体值由计算机的性能决定）．用迭代方法计算，可得 $t(n) = (n^{\log_{27}})$。因为 $\log_{27} \approx 2.81$，所以与直接计算方法的复杂性(n_3)相比，分而治之矩阵乘法算法有较大的改进。

注意事项

分而治之方法很自然地导致了递归算法的使用。在许多例子里，这些递归算法在递归程序中得到了很好的运用。实际上，在许多情况下，所有为了得到一个非递归程序的企图都会导致采用一个模拟递归栈。不过在有些情况下，不使用这样的递归栈而采用一个非递归程序来完成分而治之算法也是可能的，并且在这种方式下，程序得到结果的速度会比递归方式更快。解决金块问题的分而治之算法（例 2-2）和归并排序方法（ 2.3 节）就可以不利用递归而通过一个非递归程序来更快地完成。

例 2-4 [金块问题] 用例 2-2 的算法寻找 8 个金块中最轻和最重金块的工作可以用二叉树来表示。这棵树的叶子分别表示 8 个金块（$a, b, ., h$），每个阴影节点表示一个包含其子树中所有叶子的问题。因此，根节点 A 表示寻找 8 个金块中最轻、最重金块的问题，而节点 B 表示找出 a,b,c 和 d 这 4 个金块中最轻和最重金块的问题。算法从根节点开始。由根节点表示的 8 金块问题被划分成由节点 B 和 C 所表示的两个 4 金块问题。在 B 节点，4 金块问题被划分成由 D 和 E 所表示的 2 金块问题。可通过比较金块 a 和 b 哪一个较重来解决 D 节点所表示的 2 金块问题。在解决了 D 和 E 所表示的问题之后，可以通过比较 D 和 E 中所找到的轻金块和重金块来解决 B 表示的问题。接着在 F，G 和 C 上重复这一过程，最后解决问题 A。

可以将递归的分而治之算法划分成以下的步骤:

1) 从图 2-2 中的二叉树由根至叶的过程中把一个大问题划分成许多个小问题，小问题的大小为 1 或 2。

2) 比较每个大小为 2 的问题中的金块，确定哪一个较重和哪一个较轻。在节点 D、E、F 和 G 上完成这种比较。大小为 1 的问题中只有一个金块，它既是最轻的金块也是最重的金块。

3) 对较轻的金块进行比较以确定哪一个金块最轻，对较重的金块进行比较以确定哪一个金块最重。对于节点 A 到 C 执行这种比较。

根据上述步骤，可以得出程序 14-1 的非递归代码。该程序用于寻找到数组 w[0:n-1]中的最小数和最大数，若 n<1，则程序返回 false，否则返回 true。

当 n≥1 时，程序 14-1 给 Min 和 Max 置初值以使 w[Min]是最小的重量，w[Max]为最大的重量。

首先处理 n≤1 的情况。若 n>1 且为奇数，第一个重量 w[0]将成为最小值和最大值的候选值，因此将有偶数个重量值 w[1:n-1]参与 for 循环。当 n 是偶数时，首先将两个重量值放在 for 循环外进行比较，较小和较大的重量值分别置为 Min 和 Max，因此也有偶数个重量值 w[2:n-1]参与 for 循环。

在 for 循环中，外层 if 通过比较确定(w[i],w[i+1])中的较大和较小者。此工作与前面提到的分而治之算法步骤中的 2) 相对应，而内层的 if 负责找出较小重量值和较大重量值中的最小值和

最大值，这个工作对应于 3)。for 循环将每一对重量值中较小值和较大值分别与当前的最小值 w[Min]和最大值 w[Max]进行比较，根据比较结果来修改 Min 和 Max（如果必要）。

下面进行复杂性分析。注意到当 n 为偶数时，在 for 循环外部将执行一次比较而在 for 循环内部执行 3(n/2-1)次比较，比较的总次数为 3n/2-2。当 n 为奇数时，for 循环外部没有

执行比较，而内部执行了 3(n-1)/2 次比较。因此无论 n 为奇数或偶数，当 n>0 时，比较的总次数为 ⌈3n/2⌉-2 次。

程序 14-1 找出最小值和最大值的非递归程序

```
template<class T>
bool MinMax(T w[], int n, T& Min, T& Max)
{// 寻找 w [ 0 : n - 1 ]中的最小和最大值
// 如果少于一个元素，则返回 f a l s e
// 特殊情形:  n <= 1
if (n < 1) return false;
if (n == 1) {Min = Max = 0;
return true;}
/ /对 Min 和 M a x 进行初始化
int s; // 循环起点
if (n % 2) {// n 为奇数
Min = Max = 0;
s = 1;}
else {// n 为偶数，比较第一对
if (w[0] > w[1]) {
Min = 1;
Max = 0;}
else {Min = 0;
Max = 1;}
s = 2;}
// 比较余下的数对
for (int i = s; i < n; i += 2) {
// 寻找 w[i] 和 w [ i + 1 ]中的较大者
// 然后将较大者与 w [ M a x ]进行比较
// 将较小者与 w [ M i n ]进行比较
if (w[i] > w[i+1]) {
if (w[i] > w[Max]) Max = i;
if (w[i+1] < w[Min]) Min = i + 1;}
else {
if (w[i+1] > w[Max]) Max = i + 1;
if (w[i] < w[Min]) Min = i;}
}
return true;
}
```

练习

1. 将例 1 4 - 1 的分而治之算法扩充到 n> 1 个硬币的情形。需要进行多少次重量的比较?

2. 考虑例 1 4 - 1 的伪币问题。假设把条件“伪币比真币轻”改为“伪币与真币的重量不同”，同样假定袋中有 n 个硬币。

1) 给出相应分而治之算法的形式化描述，该算法可输出信息“不存在伪币”或找出伪币。算法应递归地将大的问题划分成两个较小的问题。需要多少次比较才能找到伪币（如果存在伪币）?

2) 重复 1) ，但把大问题划分为三个较小问题。

3. 1) 编写一个 C++ 程序，实现例 14-2 中寻找 n 个元素中最大值和最小值的两种方案。使用递归来完成分而治之方案。

2) 程序 2-26 和 2-27 是另外两个寻找 n 个元素中最大值和最小值的代码。试分别计算出每段程序所需要的最少和最大比较次数。

3) 在 n 分别等于 100，1000 或 10 000 的情况下，比较 1）、2）中的程序和程序 14-1 的运行时间。对于程序 2-27，使用平均时间和最坏情况下的时间。1）中的程序和程序 2-26 应具有相同的平均时间和最坏情况下的时间。

4) 注意到如果比较操作的开销不是很高，分而治之算法在最坏情况下不会比其他算法优越，为什么？它的平均时间优于程序 2-27 吗？为什么？

4. 证明直接运用公式（14-2）~（14-5）得出结果的矩阵乘法的分而治之算法的复杂性为(n^3)。因此相应的分而治之程序将比程序 2-24 要慢。

5. 用迭代的方法来证明公式（14-6）的递归值为$(n^{\log 7})$。

*6. 编写 Strassen 矩阵乘法程序。利用不同的 k 值（见公式（14-6））进行实验，以确定 k 为何值时程序性能最佳。比较程序及程序 2-24 的运行时间。可取 n 为 2 的幂来进行比较。

7. 当 n 不是 2 的幂时，可以通过增加矩阵的行和列来得到一个大小为 2 的幂的矩阵。假设使用最少的行数和列数将矩阵扩充为 m 阶矩阵，其中 m 为 2 的幂。

1) 求 m/n。

2) 可使用哪些矩阵项组成新的行和列，以使新矩阵 A' 和 B' 相乘时，原来的矩阵 A 和 B 相乘的结果会出现在 C' 的左上角？

3) 使用 Strassen 方法计算 $A'*B'$ 所需要的时间为$(m^{2.81})$。给出以 n 为变量的运行时间表达式。

2.2 应用

2.2.1 残缺棋盘

残缺棋盘（defective chessboard）是一个有 $2^k \times 2^k$ 个方格的棋盘，其中恰有一个方格残缺。图 2-3 给出 $k \leqslant 2$ 时各种可能的残缺棋盘，其中残缺的方格用阴影表示。注意当 $k=0$ 时，仅存在一种可能的残缺棋盘（如图 14-3a 所示）。事实上，对于任意 k，恰好存在 2^{2k} 种不同的残缺棋盘。

残缺棋盘的问题要求用三格板（trominoes）覆盖残缺棋盘（如图 14-4 所示）。在此覆盖中，两个三格板不能重叠，三格板不能覆盖残缺方格，但必须覆盖其他所有的方格。在这种限制条件下，所需要的三格板总数为 $(2^{2k}-1)/3$。可以验证 $(2^{2k}-1)/3$ 是一个整数。k 为 0 的残缺棋盘很容易被覆盖，因为它没有非残缺的方格，用于覆盖的三格板的数目为 0。当 $k=1$ 时，正好存在 3 个非残缺的方格，并且这三个方格可用图 14-4 中的某一方向的三格板来覆盖。

用分而治之方法可以很好地解决残缺棋盘问题。这一方法可将覆盖 $2^k \times 2^k$ 残缺棋盘的问题转化为覆盖较小残缺棋盘的问题。$2^k \times 2^k$ 棋盘一个很自然的划分方法就是将它划分为如图 14-5a 所示的 4 个 $2^{k-1} \times 2^{k-1}$ 棋盘。注意到当完成这种划分后， 4 个小棋盘中仅仅有一个棋盘存在残缺方格（因为原来的 $2^k \times 2^k$ 棋盘仅仅有一个残缺方格）。首先覆盖其中包含残缺方格的 $2^{k-1} \times 2^{k-1}$ 残缺棋盘，然后把剩下的 3 个小棋盘转变为残缺棋盘，为此将一个三格板放在由这 3 个小棋盘形成的角上，如图 14-5b 所示，其中原 $2^k \times 2^k$ 棋盘中的残缺方格落入左上角的 $2^{k-1} \times 2^{k-1}$ 棋盘。可以采用这种分割技术递归地覆盖 $2^k \times 2^k$ 残缺棋盘。当棋盘的大小减为 1×1 时，递归过程终止。此时 1×1 的棋盘中仅仅包含一个方格且此方格残缺，所以无需放置三格板。

可以将上述分而治之算法编写成一个递归的 C++ 函数 TileBoard(见程序 14-2)。该函数定义了一个全局的二维整数数组变量 Board 来表示棋盘。Board[0][0]表示棋盘中左上角的方格。该函数还定义了一个全局整数变量 tile，其初始值为 0。函数的输入参数如下:

• tr 棋盘中左上角方格所在行。
• tc 棋盘中左上角方格所在列。
• dr 残缺方块所在行。
• dl 残缺方块所在列。
• size 棋盘的行数或列数。

TileBoard 函数的调用格式为 TileBoard(0,0, dr, dc,size)，其中 size=2_k。覆盖残缺棋盘所需要的三格板数目为(size^2-1)/3。函数 TileBoard 用整数 1 到(size^2-1)/3 来表示这些三格板，并用三格板的标号来标记被该三格板覆盖的非残缺方格。

令 $t(k)$ 为函数 TileBoard 覆盖一个 $2_k \times 2_k$ 残缺棋盘所需要的时间。当 $k=0$ 时，size 等于 1，覆盖它将花费常数时间 d。当 $k>0$ 时，将进行 4 次递归的函数调用，这些调用需花费的时间为 $4t(k-1)$。除了这些时间外， if 条件测试和覆盖 3 个非残缺方格也需要时间，假设用常数 c 表示这些额外时间。可以得到以下递归表达式:

程序 14-2 覆盖残缺棋盘

```
void TileBoard(int tr, int tc, int dr, int dc, int size)
{// 覆盖残缺棋盘
if (size == 1) return;
int t = tile++, // 所使用的三格板的数目
s = size/2; // 象限大小
//覆盖左上象限
if (dr < tr + s && dc < tc + s)
// 残缺方格位于本象限
TileBoard(tr, tc, dr, dc, s);
else {// 本象限中没有残缺方格
// 把三格板 t 放在右下角
Board[tr + s - 1][tc + s - 1] = t;
// 覆盖其余部分
TileBoard(tr, tc, tr+s-1, tc+s-1, s);}
//覆盖右上象限
if (dr < tr + s && dc >= tc + s)
// 残缺方格位于本象限
TileBoard(tr, tc+s, dr, dc, s);
else {// 本象限中没有残缺方格
// 把三格板 t 放在左下角
Board[tr + s - 1][tc + s] = t;
// 覆盖其余部分
TileBoard(tr, tc+s, tr+s-1, tc+s, s);}
//覆盖左下象限
if (dr >= tr + s && dc < tc + s)
// 残缺方格位于本象限
TileBoard(tr+s, tc, dr, dc, s);
else {// 把三格板 t 放在右上角
Board[tr + s][tc + s - 1] = t;
// 覆盖其余部分
TileBoard(tr+s, tc, tr+s, tc+s-1, s);}
```

```
// 覆盖右下象限
if (dr >= tr + s && dc >= tc + s)
// 残缺方格位于本象限
TileBoard(tr+s, tc+s, dr, dc, s);
else {// 把三格板 t 放在左上角
Board[tr + s][tc + s] = t;
// 覆盖其余部分
TileBoard(tr+s, tc+s, tr+s, tc+s, s);}
}
void OutputBoard(int size)
{
for (int i = 0; i < size; i++) {
for (int j = 0; j < size; j++)
cout << setw (5) << Board[i][j];
cout << endl;
}
}
```

可以用迭代的方法来计算这个表达式（见例 2-20），可得 $t(k)=(4_k)=$（所需的三格板的数目）。由于必须花费至少(1)的时间来放置每一块三格表，因此不可能得到一个比分而治之算法更快的算法。

2.2.2 归并排序

可以运用分而治之方法来解决排序问题，该问题是将 n 个元素排成非递减顺序。分而治之方法通常用以下的步骤来进行排序算法：若 n 为 1，算法终止；否则，将这一元素集合分割成两个或更多个子集合，对每一个子集合分别排序，然后将排好序的子集合归并为一个集合。

假设仅将 n 个元素的集合分成两个子集合。现在需要确定如何进行子集合的划分。一种可能性就是把前面 n-1 个元素放到第一个子集中（称为 A），最后一个元素放到第二个子集里（称为 B）。按照这种方式对 A 递归地进行排序。由于 B 仅含一个元素，所以它已经排序完毕，在 A 排完序后，只需要用程序 2-10 中的函数 insert 将 A 和 B 合并起来。把这种排序算法与 InsertionSort（见程序 2-15）进行比较，可以发现这种排序算法实际上就是插入排序的递归算法。该算法的复杂性为 $O(n_2)$。把 n 个元素划分成两个子集合的另一种方法是将含有最大值的元素放入 B，剩下的放入 A 中。然后 A 被递归排序。为了合并排序后的 A 和 B，只需要将 B 添加到 A 中即可。假如用函数 Max（见程序 1-31）来找出最大元素，这种排序算法实际上就是 SelectionSort（见程序 2-7）的递归算法。

假如用冒泡过程（见程序 2-8）来寻找最大元素并把它移到最右边的位置，这种排序算法就是 BubbleSort（见程序 2-9）的递归算法。这两种递归排序算法的复杂性均为(n_2)。若一旦发现 A 已经被排好序就终止对 A 进行递归分割，则算法的复杂性为 $O(n_2)$（见例 2-16 和 2-17）。

上述分割方案将 n 个元素分成两个极不平衡的集合 A 和 B。A 有 n-1 个元素，而 B 仅含一个元素。下面来看一看采用平衡分割法会发生什么情况： A 集合中含有 n/k 个元素，B 中包含其余的元素。递归地使用分而治之方法对 A 和 B 进行排序。然后采用一个被称之为归并（merge）的过程，将已排好序的 A 和 B 合并成一个集合。

例 2-5 考虑 8 个元素，值分别为[10，4，6，3，8，2，5，7]。如果选定 $k=2$，则[10,4,6,3]和[8,2,5,7]将被分别独立地排序。结果分别为[3,4,6,10]和[2,5,7,8]。从两个序列的头部开始归并这两个已排序的序列。元素 2 比 3 更小，被移到结果序列；3 与 5 进行比较，

3 被移入结果序列；4 与 5 比较，4 被放入结果序列；5 和 6 比较，.。如果选择 *k*= 4，则序列[1 0 , 4]和[6 , 3 , 8 , 2 , 5 , 7]将被排序。排序结果分别为[4 , 1 0]和[2 , 3 , 5 , 6 , 7 , 8]。当这两个排好序的序列被归并后，即可得所需要的排序序列。

图 2 - 6 给出了分而治之排序算法的伪代码。算法中子集合的数目为 2，*A* 中含有 *n*/*k* 个元素。

```
template<class T>
void sort( T E, int n)
{ //对 E 中的 n 个元素进行排序， k 为全局变量
if (n >= k) {
  i = n/k;
  j = n-i;
  令 A 包含 E 中的前 i 个元素
  令 B 包含 E 中余下的 j 个元素
  s o r t ( A , i ) ;
  s o r t ( B , j ) ;
  m e rge(A,B,E,i,j,); //把 A 和 B 合并到 E
  }
else 使用插入排序算法对 E 进行排序
}
```

图 14-6 分而治之排序算法的伪代码

从对归并过程的简略描述中，可以明显地看出归并 *n* 个元素所需要的时间为 O (*n*)。设 *t* (*n*) 为分而治之排序算法（如图 1 4 - 6 所示）在最坏情况下所需花费的时间，则有以下递推公式:

其中 *c* 和 *d* 为常数。当 *n* / *k* ≈ *n*-*n* / *k* 时，*t* (*n*) 的值最小。因此当 *k*= 2 时，也就是说，当两个子集合所包含的元素个数近似相等时， *t* (*n*) 最小，即当所划分的子集合大小接近时，分而治之算法通常具有最佳性能。

可以用迭代方法来计算这一递推方式，结果为 *t*(*n*)= (*n*l o g*n*)。虽然这个结果是在 *n* 为 2 的幂时得到的，但对于所有的 *n*，这一结果也是有效的，因为 *t*(*n*) 是 *n* 的非递减函数。*t*(*n*) =(*n*l o g*n*) 给出了归并排序的最好和最坏情况下的复杂性。由于最好和最坏情况下的复杂性是一样的，因此归并排序的平均复杂性为 *t* (*n*)= (*n*l o g*n*)。

图 2 - 6 中 *k*= 2 的排序方法被称为归并排序（ m e rge sort ），或更精确地说是二路归并排序（two-way merge sort）。下面根据图 1 4 - 6 中 *k*= 2 的情况（归并排序）来编写对 *n* 个元素进行排序的 C＋+函数。一种最简单的方法就是将元素存储在链表中（即作为类 c h a i n 的成员（程序 3 -8））。在这种情况下，通过移到第 *n*/ 2 个节点并打断此链，可将 *E* 分成两个大致相等的链表。

归并过程应能将两个已排序的链表归并在一起。如果希望把所得到 C＋+程序与堆排序和插入排序进行性能比较，那么就不能使用链表来实现归并排序，因为后两种排序方法中都没有使用链表。为了能与前面讨论过的排序函数作比较，归并排序函数必须用一个数组 a 来存储元素集合 E，并在 a 中返回排序后的元素序列。为此按照下述过程来对图 1 4 - 6 的伪代码进行细化: 当集合 E 被化分成两个子集合时，可以不必把两个子集合的元素分别复制到 *A* 和 *B* 中，只需简单地在集合 E 中保持两个子集合的左右边界即可。接下来对 a 中的初始序列进行排序，并将所得到的排序序列归并到一个新数组 b 中，最后将它们复制到 a 中。图 1 4 - 6 的改进版见图 1 4 - 7。

```
template<class T>
M e rgeSort( T a[], int left, int right)
```

```
{ //对 a[left:right]中的元素进行排序
if (left < right) {//至少两个元素
   int i = (left + right)/2; //中心位置
   MergeSort(a, left, i);
   MergeSort(a, i+1, right);
   Merge(a, b, left, i, right); //从 a 合并到 b
   Copy(b, a, left, right); //结果放回 a
   }
}
```

图 14-7 分而治之排序算法的改进

可以从很多方面来改进图 14-7 的性能，例如，可以容易地消除递归。如果仔细地检查图 14-7 中的程序，就会发现其中的递归只是简单地重复分割元素序列，直到序列的长度变成 1 为止。当序列的长度变为 1 时即可进行归并操作，这个过程可以用 n 为 2 的幂来很好地描述。长度为 1 的序列被归并为长度为 2 的有序序列;长度为 2 的序列接着被归并为长度为 4 的有序序列;这个过程不断地重复直到归并为长度为 n 的序列。图 14-8 给出 n= 8 时的归并(和复制)过程，方括号表示一个已排序序列的首和尾。

```
初始序列[8] [4] [5] [6] [2] [1] [7] [3]
归并到 b [4 8] [5 6] [1 2] [3 7]
复制到 a [4 8] [5 6] [1 2] [3 7]
归并到 b [4 5 6 8] [1 2 3 7]
复制到 a [4 5 6 8] [1 2 3 7]
归并到 b [1 2 3 4 5 6 7 8]
复制到 a [1 2 3 4 5 6 7 8]
```

图 14-8 归并排序的例子

另一种二路归并排序算法是这样的:首先将每两个相邻的大小为 1 的子序列归并，然后对上一次归并所得到的大小为 2 的子序列进行相邻归并，如此反复，直至最后归并到一个序列，归并过程完成。通过轮流地将元素从 a 归并到 b 并从 b 归并到 a，可以虚拟地消除复制过程。二路归并排序算法见程序 14-3。

程序 14-3 二路归并排序

```
template<class T>
void MergeSort(T a[], int n)
{// 使用归并排序算法对 a[0:n-1] 进行排序
T *b = new T [n];
int s = 1; // 段的大小
while (s < n) {
MergePass(a, b, s, n); // 从 a 归并到 b
s += s;
MergePass(b, a, s, n); // 从 b 归并到 a
s += s;
}
}
```

为了完成排序代码，首先需要完成函数MergePass。函数MergePass（见程序14-4）仅用来确定欲归并子序列的左端和右端，实际的归并工作由函数Merge(见程序14-5)来完成。函数Merge要求针对类型*T*定义一个操作符<=。如果需要排序的数据类型是用户自定义类型，则必须重载操作符<=。这种设计方法允许我们按元素的任一个域进行排序。重载操作符<=的目的是用来比较需要排序的域。

程序 14-4 MergePass 函数

```
template<class T>
void MergePass(T x[], T y[], int s, int n)
{// 归并大小为 s 的相邻段
int i = 0;
while (i <= n - 2 * s) {
// 归并两个大小为 s 的相邻段
Merge(x, y, i, i+s-1, i+2*s-1);
i = i + 2 * s;
}
// 剩下不足 2 个元素
if (i + s < n) Merge(x, y, i, i+s-1, n-1);
else for (int j = i; j <= n-1; j++)
// 把最后一段复制到 y
y[j] = x[j];
}
```

程序 14-5 Merge 函数

```
template<class T>
void Merge(T c[], T d[], int l, int m, int r)
{// 把 c[l:m]] 和 c[m:r] 归并到 d [ l : r ] .
int i = l, // 第一段的游标
j = m+1, // 第二段的游标
k = l; // 结果的游标
/ /只要在段中存在 i 和 j，则不断进行归并
while ((i <= m) && (j <= r))
if (c[i] <= c[j]) d[k++] = c[i++];
else d[k++] = c[j++];
// 考虑余下的部分
if (i > m) for (int q = j; q <= r; q++)
d[k++] = c[q];
else for (int q = i; q <= m; q++)
d[k++] = c[q];
}
```

自然归并排序（natural merge sort）是基本归并排序（见程序14-3）的一种变化。它首先对输入序列中已经存在的有序子序列进行归并。例如，元素序列[4，8，3，7，1，5，6，2]中包含有序的子序列[4，8]，[3，7]，[1，5，6]和[2]，这些子序列是按从左至右的顺序对元素表进行扫描而产生的，若位置 *i* 的元素比位置 *i*+1 的元素大，则从位置 *i* 进行分割。对于上面这个元素序列，可找到四个子序列，子序列1和子序列2归并可得[3,4,7,8]，子序列3和子序列4归并可得[1,2,5,6]，最后，归并这两个子序列得到[1,2,3,4,5,6,7,8]。因此，对

于上述元素序列，仅仅使用了两趟归并，而程序 14-3 从大小为 1 的子序列开始，需使用三趟归并。作为一个极端的例子，假设输入的元素序列已经排好序并有 n 个元素，自然归并排序法将准确地识别该序列不必进行归并排序，但程序 14-3 仍需要进行 $[\log_2 n]$ 趟归并。因此自然归并排序将在 (n) 的时间内完成排序。而程序 14-3 将花费 $(n \log n)$ 的时间。

2.2.3 快速排序

分而治之方法还可以用于实现另一种完全不同的排序方法，这种排序法称为快速排序（quick sort）。在这种方法中，n 个元素被分成三段（组）：左段 *left*，右段 *right* 和中段 *middle*。中段仅包含一个元素。左段中各元素都小于等于中段元素，右段中各元素都大于等于中段元素。因此 *left* 和 *right* 中的元素可以独立排序，并且不必对 *left* 和 *right* 的排序结果进行合并。*middle* 中的元素被称为支点（pivot）。图 14-9 中给出了快速排序的伪代码。

```
//使用快速排序方法对 a[0:n-1]排序
从 a[0:n-1]中选择一个元素作为 middle，该元素为支点
把余下的元素分割为两段 left 和 right，使得 left 中的元素都小于等于支点，而 right 中的元素都大于等于支点
递归地使用快速排序方法对 left 进行排序
递归地使用快速排序方法对 right 进行排序
所得结果为 left + middle + right
```

图 14-9 快速排序的伪代码

考察元素序列[4, 8, 3, 7, 1, 5, 6, 2]。假设选择元素 6 作为支点，则 6 位于 *middle*；4，3，1，5，2 位于 *left*；8，7 位于 *right*。当 *left* 排好序后，所得结果为 1，2，3，4，5；当 *right* 排好序后，所得结果为 7，8。把 *right* 中的元素放在支点元素之后，*left* 中的元素放在支点元素之前，即可得到最终的结果[1, 2, 3, 4, 5, 6, 7, 8]。

把元素序列划分为 *left*、*middle* 和 *right* 可以就地进行（见程序 14-6）。在程序 14-6 中，支点总是取位置 1 中的元素。也可以采用其他选择方式来提高排序性能，本章稍后部分将给出这样一种选择。

程序 14-6 快速排序

```
template<class T>
void QuickSort(T*a, int n)
{// 对 a[0:n-1] 进行快速排序
{// 要求 a[n] 必需有最大关键值
quickSort(a, 0, n-1);
template<class T>
void quickSort(T a[], int l, int r)
{// 排序 a[l : r]，  a[r+1] 有大值
if (l >= r) return;
int i = l, // 从左至右的游标
j = r + 1; // 从右到左的游标
T pivot = a[l];
// 把左侧>= pivot 的元素与右侧<= pivot 的元素进行交换
while (true) {
do {// 在左侧寻找>= pivot 的元素
i = i + 1;
```

```
} while (a[i] < pivot);
do {// 在右侧寻找<= pivot 的元素
j = j - 1;
} while (a[j] > pivot);
if (i >= j) break; // 未发现交换对象
Swap(a[i], a[j]);
}
// 设置 p i v o t
a[l] = a[j];
a[j] = pivot;
quickSort(a, l, j-1); // 对左段排序
quickSort(a, j+1, r); // 对右段排序
}
```

若把程序 14-6 中 do-while 条件内的<号和>号分别修改为<=和>=，程序 14-6 仍然正确。实验结果表明使用程序 14-6 的快速排序代码可以得到比较好的平均性能。为了消除程序中的递归，必须引入堆栈。不过，消除最后一个递归调用不须使用堆栈。消除递归调用的工作留作练习（练习 13）。程序 14-6 所需要的递归栈空间为 O(*n*)。若使用堆栈来模拟递归，则可以把这个空间减少为 O(log*n*)。在模拟过程中，首先对 *left* 和 *right* 中较小者进行排序，把较大者的边界放入堆栈中。在最坏情况下 *left* 总是为空，快速排序所需的计算时间为(n^2)。在最好情况下，*left* 和 *right* 中的元素数目大致相同，快速排序的复杂性为(*n*log*n*)。令人吃惊的是，快速排序的平均复杂性也是(*n*log*n*)。

定理 2-1 快速排序的平均复杂性为(*n*log*n*)。

证明用 *t*(*n*) 代表对含有 *n* 个元素的数组进行排序的平均时间。当 $n \leqslant 1$ 时，$t(n) \leqslant d$，*d* 为某一常数。当 *n*<1 时，用 *s* 表示左段所含元素的个数。由于在中段中有一个支点元素，因此右段中元素的个数为 *n*-*s*-1。所以左段和右段的平均排序时间分别为 *t*(*s*), *t*(*n*-*s*-1)。分割数组中元素所需要的时间用 *cn* 表示，其中 *c* 是一个常数。因为 *s* 有同等机会取 0~*n*-1 中的任何一个值.

如对（2-8）式中的 *n* 使用归纳法，可得到 $t(n) \leqslant kn\log_e n$，其中 *n*>1 且 *k*=2(*c*+*d*)，*e*~2.718 为自然对数的基底。在归纳开始时首先验证 *n*=2 时公式的正确性。根据公式（14-8），可以得到 $t(2) \leqslant 2c+2d \leqslant k n\log_e 2$。在归纳假设部分，假定 $t(n) \leqslant kn\log_e n$（当 $2 \leqslant n < m$ 时，*m* 是任意一个比 2 大的整数 = .

图 14-10 对本书中所讨论的算法在平均条件下和最坏条件下的复杂性进行了比较。

方法	最坏复杂性	平均复杂性
冒泡排序	n^2	n^2
计数排序	n^2	n^2
插入排序	n^2	n^2
选择排序	n^2	n^2
堆排序	*n*log*n*	*n*log*n*
归并排序	*n*log*n*	*n*log*n*
快速排序	n^2	*n*log*n*

图 14-10 各种排序算法的比较

中值快速排序（ median-of-three quick sort）是程序 14-6 的一种变化，这种算法有更好的平均性能。注意到在程序 14-6 中总是选择 a[1]做为支点，而在这种快速排序算法中，可以不

必使用 a[1]做为支点，而是取{a[1],a[(1+r)/2],a[r]} 中大小居中的那个元素作为支点。例如，假如有三个元素，大小分别为 5，9，7，那么取 7 为支点。为了实现中值快速排序算法，一种最简单的方式就是首先选出中值元素并与 a[1] 进行交换，然后利用程序 14-6 完成排序。如果 a[r] 是被选出的中值元素，那么将 a[1] 与 a[r] 进行交换，然后将 a[1]（即原来的 a[r]）赋值给程序 14-6 中的变量 pivot，之后继续执行程序 14-6 中的其余代码。

图 2-11 中分别给出了根据实验所得到的归并排序、堆排序、插入排序、快速排序的平均时间。对于每一个不同的 *n*，都随机产生了至少 100 组整数。随机整数的产生是通过反复调用 stdlib.h 库中的 random 函数来实现的。如果对一组整数进行排序的时间少于 10 个时钟滴答，则继续对其他组整数进行排序，直到所用的时间不低于 10 个时钟滴答。在图 2-11 中的数据包含产生随机整数的时间。对于每一个 *n*，在各种排序法中用于产生随机整数及其他开销的时间是相同的。因此，图 2-11 中的数据对于比较各种排序算法是很有用的。

对于足够大的 *n*，快速排序算法要比其他算法效率更高。从图中可以看到快速排序曲线与插入排序曲线的交点横坐标比 20 略小，可通过实验来确定这个交点横坐标的精确值。可以分别用 n=15,16,17,18,19 进行实验，以寻找精确的交点。令精确的交点横坐标为 nBreak。当 n≤nBreak 时，插入排序的平均性能最佳。当 n>nBreak 时，快速排序性能最佳。当 n>nBreak 时，把插入排序与快速排序组合为一个排序函数，可以提高快速排序的性能，实现方法是把程序 14-6 中的以下语句：

```
if(l >= r)return;
```

替换为

```
if (r-1<nBreak) {InsertionSort(a,l,r); return;}
```

这里 InsertionSort(a,l,r)用来对 a[1:r]进行插入排序。测量修改后的快速排序算法的性能留作练习（练习 20）。用更小的值替换 nBreak 有可能使性能进一步提高（见练习 20）。

大多数实验表明，当 n>*c* 时（*c* 为某一常数），在最坏情况下归并排序的性能也是最佳的。而当 n≤*c* 时，在最坏情况下插入排序的性能最佳。通过将插入排序与归并排序混合使用，可以提高归并排序的性能（练习 21）。

2.2.4 选择

对于给定的 n 个元素的数组 a[0:n-1]，要求从中找出第 k 小的元素。当 a[0:n-1]被排序时，该元素就是 a[k-1]。假设 n=8，每个元素有两个域 key 和 ID，其中 key 是一个整数，ID 是一个字符。假设这 8 个元素为[(12,*a*)，(4,*b*)，(5,*c*)，(4,*d*)，(5,*e*)，(10,*f*)，(2,*g*)，(20,*h*)]，排序后得到数组[(2,*g*)，(4,*d*)，(4,*b*)，(5,*c*)，(5,*e*)，(10,*f*)，(12,*a*)，(20,*h*)]。如果 k=1，返回 ID 为 *g* 的元素；如果 k=8，返回 ID 为 *h* 的元素；如果 k=6，返回是 ID 为 *f* 的元素；如果 k=2，返回 ID 为 *d* 的元素。实际上，对最后一种情况，所得到的结果可能不唯一，因为排序过程中既可能将 ID 为 *d* 的元素排在 a[1]，也可能将 ID 为 *b* 的元素排在 a[1]，原因是它们具有相同大小的 key，因而两个元素中的任何一个都有可能被返回。但是无论如何，如果一个元素在 k=2 时被返回，另一个就必须在 k=3 时被返回。

选择问题的一个应用就是寻找中值元素，此时 k=[n/2]。中值是一个很有用的统计量，例如中间工资，中间年龄，中间重量。其他 k 值也是有用的。例如，通过寻找第 n/4,n/2 和 3n/4 这三个元素,可将人口划分为 4 份。

选择问题可在 O(nlogn)时间内解决，方法是首先对这 n 个元素进行排序（如使用堆排序式或归并排序），然后取出 a[k-1]中的元素。若使用快速排序（如图 14-11 所示），可以获得更好的平均性能，尽管该算法有一个比较差的渐近复杂性 $O(n_2)$。

可以通过修写程序 14-6 来解决选择问题。如果在执行两个 while 循环后支点元素 a[l]被交换到 a[j],那么 a[l]是 a[l:j]中的第 j-l+1 个元素。如果要寻找的第 k 个元素在 a[l:r]中，并且 j-l+1 等于 k，则答案就是 a[l]；如果 j-l+1<k，那么寻找的元素是 *right* 中的第

k - j + l - 1 个元素，否则要寻找的元素是 *left* 中的第 k 个元素。因此，只需进行 0 次或 1 次递归调用。新代码见程序 14-7。Select 中的递归调用可用 for 或 while 循环来替代（练习 25）。

程序 14-7 寻找第 k 个元素

```
template<class T>
T Select(T a[], int n, int k)
{// 返回 a [ 0 : n - 1 ]中第 k 小的元素
// 假定 a[n] 是一个伪最大元素
if (k < 1 || k > n) throw OutOfBounds();
return select(a, 0, n-1, k);
}
template<class T>
T select(T a[], int l, int r, int k)
{// 在 a [ l : r ]中选择第 k 小的元素
if (l >= r) return a[l];
int i = l, // 从左至右的游标
j = r + 1; // 从右到左的游标
T pivot = a[l];
// 把左侧>= pivot 的元素与右侧<= pivot 的元素进行交换
while (true) {
do {// 在左侧寻找>= pivot 的元素
i = i + 1;
} while (a[i] < pivot);
do {// 在右侧寻找<= pivot 的元素
j = j - 1;
} while (a[j] > pivot);
if (i >= j) break; // 未发现交换对象
Swap(a[i], a[j]);
}
if (j - l + 1 == k) return pivot;
// 设置 p i v o t
a[l] = a[j];
a[j] = pivot;
// 对一个段进行递归调用
if (j - l + 1 < k)
return select(a, j+1, r, k-j+l-1);
else return select(a, l, j-1, k);
}
```

程序 14-7 在最坏情况下的复杂性是 (n_2)，此时 *left* 总是为空，而且第 k 个元素总是位于 *right*.

如果假定 n 是 2 的幂，则可以取消公式（2-10）中的向下取整操作符。通过使用迭代方法，可以得到 $t(n) = (n)$。若仔细地选择支点元素，则最坏情况下的时间开销也可以变成 (n)。一种选择支点元素的方法是使用“中间的中间（median-of-median）”规则，该规则首先将数组 a 中的 n 个元素分成 n/r 组，r 为某一整常数，除了最后一组外，每组都有 r 个元素。然后通过在每组中对 r 个元素进行排序来寻找每组中位于中间位置的元素。最后根据所得到的 n/r 个

中间元素，递归使用选择算法，求得所需要的支点元素。

例 2-6 [中间的中间] 考察如下情形：r=5, n=27, 并且 a=[2，6，8，1，4，10，20，6，22，11，9，8，4，3，7，8，16，11，10，8，2，14，15，1，12，5，4]。这27个元素可以被分为6组[2,6,8,1,4]，[10,20,6,22,11]，[9,8,4,3,7]，[8,16,11,10,8]，[2,14,15,1,12]和[5,4]，每组的中间元素分别为4,11,7,10,12和4。[4,11,7,10,12,4]的中间元素为7。这个中间元素7被取为支点元素。由此可以得到 $left$=[2,6,1,4,6,4,3,2,1,5,4], $middle$=[7], $right$=[8,10,20,22,11,9,8,8,16,11,10,8,14,15,12]。

如果要寻找第 k 个元素且 k<12，则仅仅需要在 $left$ 中寻找；如果 k=12，则要找的元素就是支点元素；如果 k>12，则需要检查 $right$ 中的15个元素。在最后一种情况下，需在 $right$ 中寻找第(k-12)个元素。

定理 2-2 当按“中间的中间”规则选取支点元素时，以下结论为真：

1) 若 r=9, 那么当 $n \geqslant 90$ 时，有 $\max\{|left|, |right|\} \leqslant 7n/8$。

2) 若 r=5，且 a 中所有元素都不同，那么当 $n \geqslant 24$ 时，有 $\max\{|left|, |right|\} \leqslant 3n/4$。

证明这个定理的证明留作练习23。

根据定理2-2和程序14-7可知，如果采用“中间的中间”规则并取 r=9，则用于寻找第 k 个元素的时间 $t(n)$可按如下递归公式来计算：

在上述递归公式中，假设当 n<90 时使用复杂性为 $n\log n$ 的求解算法，当 $n \geqslant 90$ 时，采用“中间的中间”规则进行分而治之求解。利用归纳法可以证明，当 $n \geqslant 1$ 时有 $t(n) \leqslant 72cn$ (练习24)。

当元素互不相同时，可以使用 r=5 来得到线性时间性能。

2.2.5 距离最近的点对

给定 n 个点（x_i，y_i）（$1 \leqslant i \leqslant n$），要求找出其中距离最近的两个点。

例 14-7 假设在一片金属上钻 n 个大小一样的洞，如果洞太近，金属可能会断。若知道任意两个洞的最小距离，可估计金属断裂的概率。这种最小距离问题实际上也就是距离最近的点对问题。

通过检查所有的 $n(n-1)/2$ 对点，并计算每一对点的距离，可以找出距离最近的一对点。这种方法所需要的时间为(n^2)。我们称这种方法为直接方法。图14-13中给出了分而治之求解算法的伪代码。该算法对于小的问题采用直接方法求解，而对于大的问题则首先把它划分为两个较小的问题，其中一个问题（称为 A）的大小为 $\lceil n/2 \rceil$，另一个问题（称为 B）的大小为 $\lceil n/2 \rceil$。初始时，最近的点对可能属于如下三种情形之一：1) 两点都在 A 中（即最近的点对落在 A 中）；2) 两点都在 B 中；3) 一点在 A，一点在 B。假定根据这三种情况来确定最近点对，则最近点对是所有三种情况中距离最小的一对点。在第一种情况下可对 A 进行递归求解，而在第二种情况下可对 B 进行递归求解。

```
if (n 较小) {用直接法寻找最近点对
Return;}
//n 较大
将点集分成大致相等的两个部分 A 和 B
确定 A 和 B 中的最近点对
确定一点在 A 中、另一点在 B 中的最近点对
从上面得到的三对点中，找出距离最小的一对点
```

图 14-13 寻找最近的点对

为了确定第三种情况下的最近点对，需要采用一种不同的方法。这种方法取决于点集是如何

被划分成 A、B 的。一个合理的划分方法是从 x_i（中间值）处划一条垂线，线左边的点属于 A，线右边的点属于 B。位于垂线上的点可在 A 和 B 之间分配，以便满足 A、B 的大小。

例 2-8 考察图 14-14a 中从 a 到 n 的 14 个点。这些点标绘在图 14-14b 中。中点 $x_i=1$，垂线 $x=1$ 如图 14-14b 中的虚线所示。虚线左边的点(如 b, c, h, n, i)属于 A，右边的点(如 a, e, f, j, k, l) 属于 B。d, g, m 落在垂线上，可将其中两个加入 A, 另一个加入 B，以便 A、B 中包含相同的点数。假设 d ,m 加入 A，g 加入 B。

设是 i 的最近点对和 B 的最近点对中距离较小的一对点。若第三种情况下的最近点对比小。则每一个点距垂线的距离必小于，这样，就可以淘汰那些距垂线距离 ⩾ 的点。图 14-15 中的虚线是分割线。阴影部分以分割线为中线，宽为 2 。边界线及其以外的点均被淘汰掉，只有阴影中的点被保留下来，以便确定是否存在第三类点对（对应于第三种情况）其距离小于。用 R_A、R_B 分别表示 A 和 B 中剩下的点。如果存在点对(p,q)，$p\in A, q\in B$ 且 p, q 的距离小于，则 $p\in R_A$，$q\in R_B$。可以通过每次检查 R_A 中一个点来寻找这样的点对。假设考察 R_A 中的 p 点，p 的 y 坐标为 $p.y$，那么只需检查 R_B 中满足 $p.y- < q.y < p.y+$ 的 q 点，看是否存在与 p 间距小于的点。在图 14-16a 中给出了包含这种 q 点的 R_B 的范围。因此，只需将 R_B 中位于 ×2 阴影内的点逐个与 p 配对，以判断 p 是否是距离小于的第三类点。这个 ×2 区域被称为是 p 的比较区（comparing region）。

例 2-9 考察例 2-8 中的 14 个点。A 中的最近点对为(b,h)，其距离约为 0.316。B 中最近点对为(f, j)，其距离为 0.3，因此=0.3。当考察是否存在第三类点时，除 d, g, i, l, m 以外的点均被淘汰，因为它们距分割线 $x=1$ 的距离 ⩾ 。$R_A=\{d, i, m\}$，$R_B=\{g, l\}$，由于 d 和 m 的比较区中没有点，只需考察 i 即可。i 的比较区中仅含点 l。计算 i 和 l 的距离，发现它小于，因此(i, l)是最近的点对。

为了确定一个距离更小的第三类点，R_A 中的每个点最多只需和 R_B 中的 6 个点比较，如图 14-16 所示。

1. 选择数据结构

为了实现图 14-13 的分而治之算法，需要确定什么是“小问题”以及如何表示点。由于集合中少于两点时不存在最近点对，因此必须保证分解过程不会产生少于两点的点集。如果将少于四点的点集做为“小问题”，就可以避免产生少于两点的点集。

每个点可有三个参数：标号， x 坐标，y 坐标。假设标号为整数，每个点可用 Point1 类（见程序 14-8）来表示。为了便于按 x 坐标对各个点排序，可重载操作符 <=。归并排序程序如 14-3 所示。

程序 14-8 点类

```
class Point1 {
friend float dist(const Point1&, const Point1&);
friend void close(Point1 *, Point2 *, Point2 *, int, int, Point1&, Point1&, float&);
friend bool closest(Point1 *, int, Point1&, Point1&,float&);
friend void main();
public:
int operator<=(Point1 a) const
{return (x <= a.x);}
private:
int ID; // 点的编号
float x, y; // 点坐标
};
class Point2 {
```

```
friend float dist(const Point2&, const Point2&);
friend void close(Point1 *, Point2 *, Point2 *, int, int, Point1&, Point1&, float&);
friend bool closest(Point1 *, int, Point1&, Point1&, float&);
friend void main();
public:
int operator<=(Point2 a) const
{return (y <= a.y);}
private:
int p; // 数组 X 中相同点的索引
float x, y; // 点坐标
};
```

所输入的 n 个点可以用数组 X 来表示。假设 X 中的点已按照 x 坐标排序，在分割过程中如果当前考察的点是 X [l :r]，那么首先计算 m= (l+r) / 2，X[l:m]中的点属于 A，剩下的点属于 B。计算出 A 和 B 中的最近点对之后，还需要计算 R_A 和 R_B，然后确定是否存在更近的点对，其中一点属于 R_A，另一点属于 R_B。如果点已按 y 坐标排序，那么可以用一种很简单的方式来测试图 14-16。按 y 坐标排序的点保存在另一个使用类 Point2 (见程序 14-8) 的数组中。注意到在 Point2 类中，为了便于 y 坐标排序，已重载了操作符 < =。成员 p 用于指向 X 中的对应点。

确定了必要的数据结构之后，再来看看所要产生的代码。首先定义一个模板函数 dist(见程序 14-9)来计算点 a, b 之间的距离。T 可能是 Point1 或 Point2，因此 dist 必须是 Point1 和 Point2 类的友元。

程序 14-9 计算两点距离

```
template<class T>
inline float dist(const T& u, const T& v)
{ //计算点 u 和 v 之间的距离
float dx = u.x-v. x ;
float dy = u.y-v. y ;
return sqrt(dx * dx + dy * dy);
}
```

如果点的数目少于两个，则函数 closest(见程序 14-10)返回 false，如果成功时函数返回 true。当函数成功时，在参数 a 和 b 中返回距离最近的两个点，在参数 d 中返回距离。代码首先验证至少存在两点，然后使用 MergeSort 函数(见程序 14-3) 按 x 坐标对 X 中的点排序。接下来把这些点复制到数组 Y 中并按 y 坐标进行排序。排序完成时，对任一个 i，有 Y [i] . y≤Y [i+ 1] . y，并且 Y [i] .p 给出了点 i 在 X 中的位置。上述准备工作做完以后，调用函数 close (见程序 14-11)，该函数实际求解最近点对。

程序 14-10 预处理及调用 close

```
bool closest(Point1 X[], int n, Point1& a, Point1& b, float& d)
{// 在 n >= 2 个点中寻找最近点对
// 如果少于 2 个点，则返回 false
// 否则，在 a 和 b 中返回距离最近的两个点
if (n < 2) return false;
// 按 x 坐标排序
MergeSort(X, n);
// 创建一个按 y 坐标排序的点数组
Point2 *Y = new Point2 [n];
```

```
for (int i = 0; i < n; i++) {
// 将点 i 从 X 复制到 Y
Y[i].p = i;
Y[i].x = X[i].x;
Y[i].y = X[i].y;
}
M e r g e S o r t ( Y,n); // 按 y 坐标排序
// 创建临时数组
Point2 *Z = new Point2 [n];
// 寻找最近点对
c l o s e ( X , Y, Z , 0 , n - 1 , a , b , d ) ;
// 删除数组并返回
delete [] Y;
delete [] Z;
return true;
}
```

程序 14 - 11 计算最近点对

```
void close(Point1 X[], Point2 Y[], Point2 Z[], int l, int r, Point1& a, Point1& b, float& d)
{//X[l:r] 按 x 坐标排序
//Y[l:r] 按 y 坐标排序
if (r-l == 1) {// 两个点
a = X[l];
b = X[r];
d = dist(X[l], X[r]);
r e t u r n ; }
if (r-l == 2) {// 三个点
// 计算所有点对之间的距离
float d1 = dist(X[l], X[l+1]);
float d2 = dist(X[l+1], X[r]);
float d3 = dist(X[l], X[r]);
// 寻找最近点对
if (d1 <= d2 && d1 <= d3) {
a = X[l];
b = X[l+1];
d = d1;
r e t u r n ; }
if (d2 <= d3) {a = X[l+1];
b = X[r];
d = d2;}
else {a = X[l];
b = X[r];
d = d3;}
r e t u r n ; }
/ /多于三个点，划分为两部分
```

```
int m = (l+r)/2; // X[l:m] 在 A 中，余下的在 B 中
// 在 Z[l:m] 和 Z [ m + 1 : r ]中创建按 y 排序的表
int f = l, // Z[l:m]的游标
g = m+1; // Z[m+1:r]的游标
for (int i = l; i <= r; i++)
if (Y[i].p > m) Z[g++] = Y[i];
else Z[f++] = Y[i];
// 对以上两个部分进行求解
c l o s e ( X , Z , Y, l , m , a , b , d ) ;
float dr;
Point1 ar, br;
c l o s e ( X , Z , Y, m + 1 , r, a r, b r, d r ) ;
// (a,b) 是两者中较近的点对
if (dr < d) {a = ar;
b = br;
d = dr;}
M e r g e ( Z , Y,l,m,r);// 重构 Y
/ /距离小于 d 的点放入 Z
int k = l; // Z 的游标
for (i = l; i <= r; i++)
if (fabs(Y[m].x - Y[i].x) < d) Z[k++] = Y[i];
// 通过检查 Z [ l : k - 1 ]中的所有点对，寻找较近的点对
for (i = l; i < k; i++){
for (int j = i+1; j < k && Z[j].y - Z[i].y < d;
j + + ) {
float dp = dist(Z[i], Z[j]);
if (dp < d) {// 较近的点对
d = dp;
a = X[Z[i].p];
b = X[Z[j].p];}
}
}
}
```

函数 c l o s e（见程序 1 4 - 11）用来确定 X[1:r] 中的最近点对。假定这些点按 x 坐标排序。在 Y [1 : r]中对这些点按 y 坐标排序。Z[1 : r]用来存放中间结果。找到最近点对以后，将在 a, b 中返回最近点对，在 d 中返回距离，数组 Y 被恢复为输入状态。函数并未修改数组 X。

首先考察“小问题”，即少于四个点的点集。因为分割过程不会产生少于两点的数组，因此只需要处理两点和三点的情形。对于这两种情形，可以尝试所有的可能性。当点数超过三个时，通过计算 m = (l + r) / 2 把点集分为两组 A 和 B，X [1 : m]属于 A，X [m + 1 : r]属于 B。通过从左至右扫描 Y 中的点以及确定哪些点属于 A，哪些点属于 B，可以创建分别与 A 组和 B 组对应的，按 y 坐标排序的 Z [1 : m]和 Z [m + 1 : r]。此时 Y 和 Z 的角色互相交换，依次执行两个递归调用来获取 A 和 B 中的最近点对。在两次递归调用返回后，必须保证 Z 不发生改变，但对 Y 则无此要求。不过，仅 Y [l : r]可能会发生改变。通过合并操作（见程序 1 4 - 5）可以以 Z [1 : r]重构 Y [1 : r]。

为实现图 14-16 的策略，首先扫描 Y[1:r]，并收集距分割线小于的点，将这些点存放在 Z[1:k-1]中。可按如下两种方式来把 R_A 中点 p 与 p 的比较区内的所有点进行配对：1) 与 R_B 中 y 坐标≥p.y 的点配对；2) 与 y 坐标≤p.y 的点配对。这可以通过将每个点 Z[i]（1≤i<k，不管该点是在 R_A
还是在 R_B 中）与 Z[j] 配对来实现，其中 i<j 且 Z[j].y-Z[i].y< 。对每一个 Z[i]，在 2 × 区域内所检查的点如图 14-17 所示。由于在每个 2 × 子区域内的点至少相距。因此每一个子区域中的点数不会超过四个，所以与 Z[i]配对的点 Z[j]最多有七个。

2. 复杂性分析

令 $t(n)$ 代表处理 n 个点时，函数 close 所需要的时间。当 $n<4$ 时，$t(n)$ 等于某个常数 d。当 $n\geqslant 4$ 时，需花费(n) 时间来完成以下工作：将点集划分为两个部分，两次递归调用后重构 Y，淘汰距分割线很远的点，寻找更好的第三类点对。两次递归调用需分别耗时 $t(\lceil n/2\rceil)$和 $t(\lfloor n/2\rfloor)$.

这个递归式与归并排序的递归式完全一样，其结果为 $t(n)=(n\log n)$。另外，函数 closest 还需耗时$(n\log n)$来完成如下额外工作：对 X 进行排序，创建 Y 和 Z，对 Y 进行排序。因此分而治之最近点对求解算法的时间复杂性为$(n\log n)$。

练习

8. 编写一个完整的残缺棋盘问题的求解程序，提供以下模块：欢迎用户使用本程序、输入棋盘大小和残缺方格的位置、输出覆盖后的棋盘。输出棋盘时要着色，共享同一边界的覆盖应着不同的颜色。由于棋盘是平面图，所以最多只需用四种颜色便可为整个棋盘着色。在本练习中，应尽量使用较少的颜色。

9. 用迭代方法求解公式（14-7）的递归式。

10. 编写一个归并排序程序，要求用链表来存储元素。输出结果为排序后的链表。把函数做为 Chain 类（见程序 3-8）的一个成员函数。

11. 编写函数 NaturalMergeSort 来实现自然归并排序。其中输入和输出的规定与程序 14-3 相同。

12. 编写一个自然归并排序函数，用来对链表元素进行排序。把函数设计为 Chain 类的一个成员(见程序 3-8)。

13. 用一个 while 循环来替换程序 14-6 中的最后一个递归调用 quickSort。比较修改后的函数与程序 14-6 的平均运行时间。

14. 重写程序 14-6，使用堆栈来模拟递归。堆栈中只需保存 $left$ 和 $right$ 中较小者的边界。

1) 证明所需要的栈空间大小为 O(log$n$)。

2) 比较程序 14-6 和新代码的平均运行时间。

15. 证明在最坏情况下 QuickSort 的运行时间为(n^2)。

16. 假定在划分 $left$、$middle$ 和 $right$ 时按照如下方式来进行：若 n 为奇数，则 $left$ 与 $right$ 的大小相同；若 n 为偶数，则 $left$ 比 $right$ 多一个元素。证明在这种假设条件下，程序 14-6 的时间复杂性为$(n\log n)$。

17. 证明，并利用该结果证明。

18. 试比较使用“中间的中间”规则与不使用该规则时，程序 14-6 的最坏复杂性和平均复杂性。取 n=10, 20, ., 100, 200, 300, 400, 500, 1000 及适当的测试数据来进行比较。

19. 采用随机产生的数作为支点元素完成练习 18。

20. 在 14.2.3 节快速排序结束时，我们曾建议将快速排序与插入排序进行结合，结合后的算法实质上仍是快速排序，只是当排序部分小于等于 ChangeOver=nBreak 时执行插入排序。能否通过改变 ChangeOver 而得到更快的算法?为什么?试试不同的 ChangeOver。确定能提供最佳平均性能的 ChangeOver。

21. 设计一个在最差情况下性能最好的排序算法。

1) 比较插入排序、冒泡排序、选择排序、堆排序、归并排序和快速排序在最坏情况下的运行时间。导致插入排序、冒泡排序、选择排序和快速排序出现最坏复杂性的输入数据很容易产生。试编写一个程序，用来产生导致归并排序出现最坏复杂性的输入数据。这个程序本质上是将 n 个排好序的元素“反归并”。对于堆排序，用随机产生的输入序列来估算最坏情况下的时间复杂性。
2) 利用 1) 中的结果设计一个混合排序函数，使其在最坏情况下具有最佳性能。比如混合函数可以只包含归并排序和插入排序。
3) 测试混合排序函数在最坏情况下的运行时间，并与原排序函数进行比较。
4) 用一个简单的图表来列出七种排序函数在最差情况下的运行时间。
22. 当 n 是 2 幂时，用迭代方法求解公式 14-8。
*23. 证明定理 14-2。
*24. 用归纳法证明，对于公式（14-11），当 $n \geqslant 1$ 时有 $t(n) \leqslant 72cn$。
25. 程序 14-7 所需要的递归栈空间为 O(n)。当用一个 while 或 for 循环来代替递归调用时，可以完全消除这种递归栈空间。根据这种思想重写程序 14-7。比较这两种选择排序函数的运行时间。
26. 重写程序 14-7，用随机数产生器来选择支点元素。试比较这两种代码的平均性能。
27. 重写程序 14-7，使用“中间的中间”规则，其中 $r=9$。
28. 为了加快程序 14-11 的执行速度，可以不执行距离计算公式中的开方运算，而直接用距离的平方来代替距离，所得结果是一样的。为此，程序 14-11 必须做哪些改变？试通过实验来比较这两种版本的性能。
29. 重写程序 14-11，把 Point1 作为模板类，其中 ID 域的类型由用户来决定。
30. 当所有点都在一条直线上时，编写一个更快的算法来寻找最近的点对。例如，假设所有点都在一条水平线上。如果这些点根据 x 坐标排序，则最近点对中的两个点必相邻。虽然使用 MergeSort（见程序 14-3）来实现这种策略时，算法的复杂性仍然是 $O(n\log n)$，但这种算法的额外开销要比程序 14-10 小得多，因此会运行得更快。
31. 考察最近点对问题。假设初始时不是根据 x 坐标来排序，而是使用 Selcet (见程序 14-7)来寻找中点 x_i，以便将点集划分为 A 组和 B 组。
1) 给出按这种思想实现的最近点对问题求解算法的伪代码。
2) 算法的复杂性是多少？
3) 比较新算法与程序 14-11 的运行速度。

2.3 解递归方程

许多分而治之算法的复杂性都是由一个递归方程给出.

练习

32. 用迭代原理证明公式（14-13）是递归式（14-12）的解。
33. 根据图 14-18 中的表求解以下递归式。假定在每种情况下都有 t(1)=1。
1) $t(n)=10t(n/3)+11n$, $n \geqslant 3$ 且为 3 的幂。
2) $t(n)=10t(n/3)+11n^5$, $n \geqslant 3$ 且为 3 的幂。
3) $t(n)=27t(n/3)+11n^3$, $n \geqslant 3$ 且为 3 的幂。
4) $t(n)=64t(n/4)+10n^3\log_2 n$, $n \geqslant 4$ 且为 4 的幂。
5) $t(n)=9t(n/2)+n^2 2^n$, $n \geqslant 2$ 且为 2 的幂。
6) $t(n)=3t(n/8)+n^2 2^n\log n$, $n \geqslant 8$ 且为 8 的幂。
7) $t(n)=128t(n/2)+6n$, $n \geqslant 2$ 且为 2 的幂。
8) $t(n)=128t(n/2)+6n^8$, $n \geqslant 2$ 且为 2 的幂。

9) $t(n)=128t(n/2)+2^n/n, n\geqslant 2$ 且为 2 的幂。

10) $t(n)=128t(n/2)+\log_3 n, n\geqslant 2$ 且为 2 的幂。

2.4 复杂性的下限

当且仅当某个问题至少有一个复杂性为 $O(f(n))$ 的求解算法时，存在一个复杂性的上限（upper bound）$f(n)$。证明一个问题复杂性上限为 $f(n)$ 的一种方法是设计一个复杂性为 $O(f(n))$ 的算法。对于本书中的每一个算法，都给出了所解决问题的复杂性上限。如，在发现 Strassen 矩阵乘法(例 14.3)之前，矩阵乘法的复杂性上限为 n^3（因为程序 2-24 的复杂性为(n^3)）。Strassen 算法的发现使复杂性的上限降为 $n^{2.81}$。当且仅当一个问题所有的求解算法的复杂性均为$\Omega(f(n))$ 时，存在一个复杂性的下限（lower bound）$f(n)$。为了确定一个问题的复杂性下限 $g(n)$，必须证明该问题的每一个求解算法的复杂性均为$\Omega(g(n))$。要得到这样一个结论相当困难，因为要考察所有可能的求解算法。对于大多数问题，可以建立一个基于输入和/或输出数目的简单下限。例如，对 n 个元素进行排序的算法的复杂性为$\Omega(n)$，因为所有的算法对每一个元素都必须检查至少一遍，否则未检查的元素可能会排列在错误的位置上。类似地，每一个计算两个 $n\times n$ 矩阵乘法的算法都有复杂性$\Omega(n^2)$。因为结果矩阵中有 n^2 个元素并且产生每个元素所需要的时间为$\Omega(1)$。只有极少数问题的下限能够找到一个精确的值。

本节将建立本章所介绍的两个分而治之算法的确切下限——寻找 n 个元素中的最大值和最小值问题以及排序。问题对于这两个问题，仅限于考察比较算法（comparison algorithm）。所谓比较算法是指算法的操作主要限于元素比较和元素移动。第 2 章所介绍的最小最大算法以及本章中所介绍的算法都属于比较算法。除箱子排序和基数排序外，本书中所有介绍的其他所有排序算法也都是比较算法。

2.4.1 最小最大问题的下限

程序 14-1 给出了一个寻找 n 个元素中最大值与最小值的分而治之函数，该函数执行了$[3n/2]-2$ 次元素比较。我们将要证明对于该问题的每一个比较算法，都至少需要比较$[3/n]-2$ 次。为了证明该结论，假设 n 个元素互不相同。这种假设不会影响证明的普遍性，因为不同元素的输入是输入空间的一个子集。另外，每个算法对于有重复元素和没有重复元素的输入都能正确工作。证明过程中需要使用状态空间方法（state space method）。这个方法要求首先定义算法的三种状态：起始状态、中间状态和完成状态，并需描述如何从一个状态转换到另一个状态，然后确定从起始状态到完成状态所需的最少转换数。一个算法的起始状态、中间状态和完成状态是一个抽象的概念，不必寻根问底。对于最小最大问题，算法状态可用元组(a, b, c, d)来描述，a 表示算法需考察的候选的最大和最小元素的个数，b 表示不再做为最小候选但仍作为最大候选的元素个数，c 是不再做为最大候选但仍做为最小候选的元素个数，d 是被确定为即非最大也非最小的元素个数。A，B，C，D 代表上述各种元素的集合。

在最小最大算法启动时，所有 n 个元素都是最大与最小元素的候选，状态为$(n,0,0,0)$，当算法结束时，A 为空，B 和 C 中各有 1 个元素，D 中有 $n-2$ 个元素，因此完成状态为$(0,1,1,n-2)$。在比较元素的过程中算法状态发生变化。当 A 中的两个元素比较完时，较小的元素放入 C，较大的元素放入 B(根据假设所有元素都不相同，因此不会出现相等的情形)。下面是一种可能的状态转换：

$(a, b, c, d)\rightarrow(a-2, b+1, c+1, d)$

其他可能的状态转换如下：

- B 中元素比较之后，可能的转换为：

$(a, b, c, d)\rightarrow(a, b-1, c, d+1)$

- C 中元素比较之后，可能的转换为：

$(a, b, c, d) \rightarrow (a, b, c-1, d+1)$

• A 中元素与 B 中元素进行比较，可能的转换为:

$(a, b, c, d) \rightarrow (a-1, b, c, d+1)$ (A 中元素大于 B 中元素)

$(a, b, c, d) \rightarrow (a-1, b, c+1, d)$ (A 中元素小于 B 中元素)

• A 中元素与 C 中元素进行比较，可能的转换为:

$(a, b, c, d) \rightarrow (a-1, b, c, d+1)$ (A 中元素小于 C 中元素)

$(a, b, c, d) \rightarrow (a-1, b+1, c, d)$ (A 中元素大于 C 中元素)

虽然也可能进行其他比较，但它们不能确保能使状态发生变化。考查上述可能的状态转换，可以发现，当 n 为偶数时，欲从起始状态$(n, 0, 0, 0)$到达完成状态$(0, 1, 1, n-2)$，最快的方式是在 A 中执行 $n/2$ 次比较，在 B 中执行 $n/2-1$ 次比较，在 C 中执行 $n/2-1$ 次比较，总共需比较 $3n/2-2$ 次；当 n 为奇数时，最快的方式是在 A 中执行$\lfloor n/2 \rfloor$次比较，在 B 中执行$\lfloor n/2 \rfloor-1$ 次比较，在 C 中执行$\lfloor n/2 \rfloor-1$ 次比较，另有至多两次 A 中剩余元素的比较，总的比较次数为$\lceil 3n/2 \rceil-2$。因为没有哪个算法从起始状态到完成状态的比较次数少于$\lceil 3n/2 \rceil-2$，因此这个数是所有比较算法所需比较次数的下限。所以程序 14-1 是解决最大最小问题的理想算法。

2.4.2 排序算法的下限

用状态空间定理可以证明对 n 个元素进行排序时，在最坏情况下比较算法的复杂性下限为 $n\log n$。对于排序算法，我们把算法的状态定义为仍可能成为输出候选的 n 个元素的排列个数。算法启动时，对应于 n 个元素的所有 $n!$ 种排列都是候选。当算法结束时，只有一种排列保留下来。（假设 n 个元素互不相同。）当 a_i 与 a_j 比较时，当前候选的排列集合被分为两组：一组满足 $a_i < a_j$；另一组满足 $a_i > a_j$。因为已假设元素互不相同，所以 $a_i = a_j$ 不存在。例如，假设 $n=3$，则首先比较 a_1 和 a_3。在比较前，所有六种可能的排列都被作为候选输出。若 $a_1 < a_3$，则删除(a_3, a_1, a_2)，(a_3, a_2, a_1) 和(a_2, a_3, a_1)，余下的三种排列继续做为候选输出。

如果当前候选有 m 个，一次比较之后分成两组，其中一组至少包含$[m/2]$ 种排列。最坏情况下算法的初始候选有 $n!$个，然后降为至少 $n!/2$，再降为至少 $n!/4$，如此等等，直到只有一个候选为止。这种候选下降的次数最少有$[\log n!]$。因为 $n! \geqslant [n/2]^{\lfloor n/2 \rfloor-1}$, $\log n! \geqslant (n/2-1)\log(n/2) = \Omega(n\log n)$。所以每种排序算法（同时也是比较算法）在最坏情况下，要进行$\Omega(n\log n)$次比较。

也可用决策树（decision-tree）来证明下限。在这种证明过程中用树来模拟算法的执行过程。对于树的每个内部节点，算法执行一次比较并根据比较结果移向它的某一孩子。算法在叶节点处终止。图 14-19 给出了对三个元素 a[0:2]使用 InsertionSort（见程序 2-15）排序时的决策树。每个内部节点有一个 i:j 的标志，表示 a[i]与 a[j]进行比较。如果 a[i]<a[j]，算法移向左孩子；如果 a[i]>a[j]，移向右孩子。因为元素互不相同，所以 a[i]=a[j]不会发生。叶节点标出了所产生的排序。图 14-19 中最左路径代表：a[1]<a[0]，a[2]<a[0]，a[2]<a[1]，因此最左叶节点为(a[2], a[1], a[0])。

注意到决策树中每个叶节点代表一种唯一的输出排列。由于一个正确的排序算法对于 n 个输入必须能产生 $n!$ 个可能的排列，因此决策树中至少要有 $n!$ 个叶节点。因为一个高度为 h 的树至多有 2^h 个叶节点，因此决策树的高度至少为$[\log_2 n!] = \Omega(n\log n)$，因而，每一个比较排序算法在最坏情况下至少要进行$\Omega(n\log n)$次比较。另外，由于每个有 $n!$ 个叶节点的二叉树的平均高度为$\Omega(n\log n)$，因此每个比较排序算法的平均复杂性也是$\Omega(n\log n)$。由前面的证明可以看出，堆排序、归并排序在最坏情况下有较好的性能（针对渐进复杂性而言），堆排序、归并排序、快速排序在平均情况下性能较优。

练习

34. 用状态空间方法证明，要找出 n 个元素的最大值，每一种比较算法都至少要比较 $n-1$ 次。

35. 证明 $n! \geqslant \lceil n/2 \rceil^{\lceil n/2 \rceil+1}$。

36. 画出 $n=4$ 时插入排序的决策树。
37. 画出 $n=4$ 时归并排序（见程序 14-3）的决策树。
38. 令 $a_1, ., a_n$ 为 n 个元素的序列。当且仅当 $a_i > a_j$（$i<j$）时，a_i 和 a_j 是颠倒的（inverted）。元素序列中满足颠倒关系的元素对(a_i, a_j)的个数被称为该元素序列的颠倒数（ inversionnumber）。
1) 序列 6, 2, 3, 1 的颠倒数是多少？
2) n 个元素的序列中最大的颠倒数是多少？
3) 假设有一种排序算法只比较相邻的元素，并可能将其交换(实质上冒泡排序、选择排序和插入排序就是这样做的)。证明这种排序算法必须执行$\Omega(n_2)$次比较。

（ 说明：本资料是根据《数据结构、算法与应用》(美，Sartaj Sahni 著）一书第 13-17 章编辑、改写的。考虑到因特网传输速度等因素，大部分插图和公式不得不被删除。对于内容不连贯之处，请网友或读者参阅该书，敬请原谅。 ）

第 3 章 动态规划

动态规划是本书介绍的五种算法设计方法中难度最大的一种，它建立在最优原则的基础上。采用动态规划方法，可以优雅而高效地解决许多用贪婪算法或分而治之算法无法解决的问题。在介绍动态规划的原理之后，本章将分别考察动态规划方法在解决背包问题、图象压缩、矩阵乘法链、最短路径、无交叉子集和元件折叠等方面的应用。

3.1 算法思想

和贪婪算法一样，在动态规划中，可将一个问题的解决方案视为一系列决策的结果。不同的是，在贪婪算法中，每采用一次贪婪准则便做出一个不可撤回的决策，而在动态规划中，还要考察每个最优决策序列中是否包含一个最优子序列。

例 3-1 [最短路经] 考察图 12-2 中的有向图。假设要寻找一条从源节点 $s=1$ 到目的节点 $d=5$ 的最短路径，即选择此路径所经过的各个节点。第一步可选择节点 2，3 或 4。假设选择了节点 3，则此时所要求解的问题变成：选择一条从 3 到 5 的最短路径。如果 3 到 5 的路径不是最短的，则从 1 开始经过 3 和 5 的路径也不会是最短的。例如，若选择的子路径（非最短路径）是 3，2，5 (耗费为 9)，则 1 到 5 的路径为 1，3，2，5 (耗费为 11)，这比选择最短子路径 3，4，5 而得到的 1 到 5 的路径 1，3，4，5 (耗费为 9) 耗费更大。

所以在最短路径问题中，假如在的第一次决策时到达了某个节点 v，那么不管 v 是怎样确定的，此后选择从 v 到 d 的路径时，都必须采用最优策略。

例 3-2 [0/1 背包问题] 考察 13.4 节的 0/1 背包问题。如前所述，在该问题中需要决定 $x_1 . . x_n$ 的值。假设按 $i=1$，2，.，n 的次序来确定 x_i 的值。如果置 $x_1=0$，则问题转变为相对于其余物品（即物品 2，3，.，n），背包容量仍为 c 的背包问题。若置 $x_1=1$，问题就变为关于最大背包容量为 $c-w_1$ 的问题。现设 $r \in \{c, c-w_1\}$ 为剩余的背包容量。

在第一次决策之后，剩下的问题便是考虑背包容量为 r 时的决策。不管 x_1 是 0 或是 1，$[x_2, ., x_n]$ 必须是第一次决策之后的一个最优方案，如果不是，则会有一个更好的方案$[y_2, ., y_n]$，因而$[x_1, y_2, ., y_n]$是一个更好的方案。

假设 n=3, w=[100,14,10], p=[20,18,15], c=116。若设 $x_1=1$，则在本次决策之后，可用的背包

容量为 r= 116-100=16 。[x_2，x_3]=[0,1] 符合容量限制的条件，所得值为 15，但因为[x_2，x_3]= [1，0] 同样符合容量条件且所得值为 18，因此[x_2，x_3] = [0，1] 并非最优策略。即 x= [1，0，1] 可改进为 x= [1，1，0]。若设 x_1= 0，则对于剩下的两种物品而言，容量限制条件为 116。总之，如果子问题的结果[x_2，x_3]不是剩余情况下的一个最优解，则[x_1，x_2，x_3]也不会是总体的最优解。

例 3-3 [航费] 某航线价格表为：从亚特兰大到纽约或芝加哥，或从洛杉矶到亚特兰大的费用为$100; 从芝加哥到纽约票价$20; 而对于路经亚特兰大的旅客，从亚特兰大到芝加哥的费用仅为$20。从洛杉矶到纽约的航线涉及到对中转机场的选择。如果问题状态的形式为（起点，终点），那么在选择从洛杉矶到亚特兰大后，问题的状态变为（亚特兰大，纽约）。从亚特兰大到纽约的最便宜航线是从亚特兰大直飞纽约，票价$100。而使用直飞方式时，从洛杉矶到纽约的花费为$200。不过，从洛杉矶到纽约的最便宜航线为洛杉矶-亚特兰大-芝加哥-纽约，其总花费为$140（在处理局部最优路径亚特兰大到纽约过程中选择了最低花费的路径：亚特兰大-芝加哥-纽约）。

如果用三维数组（tag，起点，终点）表示问题状态，其中 tag 为 0 表示转飞， tag 为 1 表示其他情形，那么在到达亚特兰大后，状态的三维数组将变为（ 0，亚特兰大，纽约），它对应的最优路径是经由芝加哥的那条路径。

当最优决策序列中包含最优决策子序列时，可建立动态规划递归方程（ dynamic -programming recurrence equation），它可以帮助我们高效地解决问题。

例 3-4 [0/1 背包] 在例 3-2 的 0/1 背包问题中，最优决策序列由最优决策子序列组成。假设 $f(i,y)$ 表示例 15-2 中剩余容量为 y，剩余物品为 i，i+1，.，n 时的最优解的值，即：和利用最优序列由最优子序列构成的结论，可得到 f 的递归式。f(1 ,c) 是初始时背包问题的最优解。可使用（ 15-2）式通过递归或迭代来求解 f(1 ,c)。从 f(n, *)开始迭式， f(n, *)由（15-1）式得出，然后由（ 15-2）式递归计算 f(i,*) (i=n- 1，n- 2，.， 2)，最后由（ 15-2）式得出 f(1 ,c)。

对于例 15-2，若 0⩽y<10，则 f(3 ,y) = 0; 若 y⩾10, f(3 ,y) = 15。利用递归式（15-2），可得 f(2, y) = 0 (0⩽y<10); f（2，y）= 15（10⩽y<14）; f（2，y）= 18（14⩽y<24）和 f（2，y）= 33（y⩾24）。因此最优解 f(1 , 116) = max {f（2，116），f（2，116 - w_1）+ p_1} = max {f（2，116），f（2，16）+20 } = max { 33，38 } = 38。

现在计算 x_i 值，步骤如下：若 f(1 ,c) =f(2 ,c)，则 x_1= 0，否则 x_1= 1。接下来需从剩余容量 c-w_1 中寻求最优解，用 f(2, c-w_1) 表示最优解。依此类推，可得到所有的 x_i(i= 1 .n) 值。

在该例中，可得出 f(2 , 116) = 33 ≠ f(1 , 116)，所以 x_1= 1。接着利用返回值 38 -p_1=18 计算 x_2 及 x_3，此时 r = 116 -w_1 = 16，又由 f(2 , 16) = 18，得 f(3 , 16) = 14 ≠ f(2 , 16)，因此 x_2= 1，此时 r= 16 -w_2 = 2，所以 f(3,2) =0，即得 x_3= 0。

动态规划方法采用最优原则（ principle of optimality）来建立用于计算最优解的递归式。所谓最优原则即不管前面的策略如何，此后的决策必须是基于当前状态（由上一次决策产生）的最优决策。由于对于有些问题的某些递归式来说并不一定能保证最优原则，因此在求解问题时有必要对它进行验证。若不能保持最优原则，则不可应用动态规划方法。在得到最优解的递归式之后，需要执行回溯（traceback）以构造最优解。

编写一个简单的递归程序来求解动态规划递归方程是一件很诱人的事。然而，正如我们将在下文看到的，如果不努力地去避免重复计算，递归程序的复杂性将非常可观。如果在递归程序设计中解决了重复计算问题时，复杂性将急剧下降。动态规划递归方程也可用迭代方式来求解，这时很自然地避免了重复计算。尽管迭代程序与避免重复计算的递归程序有相同的复杂性，但迭代程序不需要附加的递归栈空间，因此将比避免重复计算的递归程序更快。

3.2 应用

3.2.1 0/1 背包问题

1. 递归策略

在例 3 - 4 中已建立了背包问题的动态规划递归方程，求解递归式（ 15 - 2）的一个很自然的方法便是使用程序 15 - 1 中的递归算法。该模块假设 p、w 和 n 为输入，且 p 为整型，F(1,c) 返回 $f(1,c)$ 值。

程序 15-1 背包问题的递归函数

```
int F(int i, int y)
{// 返回 f ( i , y ) .
if (i == n) return (y < w[n]) ? 0 : p[n];
if (y < w[i]) return F(i+1,y);
return max(F(i+1,y), F(i+1,y-w[i]) + p[i]);
}
```

程序 15 - 1 的时间复杂性 $t(n)$满足：$t(1)=a$；$t(n)\leqslant 2t(n-1)+b\ (n>1)$，其中 a、b 为常数。通过求解可得 $t(n)=O(2^n)$。

例 3-5 设 $n=5$，$p=[6,3,5,4,6]$，$w=[2,2,6,5,4]$ 且 $c=10$，求 $f(1,10)$。为了确定 $f(1,10)$，调用函数 F (1 , 1 0)。递归调用的关系如图 15 - 1 的树型结构所示。每个节点用 y 值来标记。对于第 j 层的节点有 $i=j$，因此根节点表示 F (1 , 1 0)，而它有左孩子和右孩子，分别对应 F (2 , 1 0) 和 F (2 , 8)。总共执行了 28 次递归调用。但我们注意到，其中可能含有重复前面工作的节点，如 $f(3,8)$计算过两次，相同情况的还有 $f(4,8)$、$f(4,6)$、$f(4,2)$、$f(5,8)$、$f(5,6)$、$f(5,3)$、$f(5,2)$ 和 $f(5,1)$。如果保留以前的计算结果，则可将节点数减至 19，因为可以丢弃图中的阴影节点。

正如在例 3 - 5 中所看到的，程序 15 - 1 做了一些不必要的工作。为了避免 $f(i,y)$的重复计算，必须定义一个用于保留已被计算出的 $f(i,y)$值的表格 L，该表格的元素是三元组$(i,y,f(i,y))$。在计算每一个 $f(i,y)$之前，应检查表 L 中是否已包含一个三元组$(i,y,*)$，其中*表示任意值。如果已包含，则从该表中取出 $f(i,y)$的值，否则，对 $f(i,y)$进行计算并将计算所得的三元组$(i,y,f(i,y))$加入表 L。L 既可以用散列（见 7 . 4 节）的形式存储，也可用二叉搜索树(见 11 章)的形式存储。

2. 权为整数的迭代方法

当权为整数时，可设计一个相当简单的算法（见程序 15 - 2）来求解 $f(1,c)$。该算法基于例 3 - 4 所给出的策略，因此每个 $f(i,y)$ 只计算一次。程序 15 - 2 用二维数组 f [ ][ ]来保存各 f 的值。而回溯函数 Traceback 用于确定由程序 15 - 2 所产生的 x_i 值。函数 Knapsack 的复杂性为 (nc)，而 Traceback 的复杂性为(n)。

程序 15-2 f 和 x 的迭代计算

```
template<class T>
void Knapsack(T p[], int w[], int c, int n, T** f)
{// 对于所有 i 和 y 计算 f [ i ] [ y ]
// 初始化 f [ n ] [ ]
for (int y = 0; y <= yMax; y++)
f[n][y] = 0;
for (int y = w[n]; y <= c; y++)
f[n][y] = p[n];
// 计算剩下的 f
for (int i = n - 1; i > 1; i--) {
for (int y = 0; y <= yMax; y++)
```

```
f[i][y] = f[i+1][y];
for (int y = w[i]; y <= c; y++)
f[i][y] = max(f[i+1][y], f[i+1][y-w[i]] + p[i]);
}
f[1][c] = f[2][c];
if (c >= w[1])
f[1][c] = max(f[1][c], f[2][c-w[1]] + p[1]);
}
template<class T>
void Traceback(T **f, int w[], int c, int n, int x[])
{// 计算 x
for (int i = 1; i < n; i++)
if (f[i][c] == f[i+1][c]) x[i] = 0;
else {x[i] = 1;
c -= w[i];}
x[n] = (f[n][c]) ? 1 : 0;
}
```

3. 元组方法（ 选读）

程序 15-2 有两个缺点：1) 要求权为整数；2) 当背包容量 c 很大时，程序 15-2 的速度慢于程序 15-1。一般情况下，若 $c>2^n$，程序 15-2 的复杂性为 $\Omega(n2^n)$。可利用元组的方法来克服上述两个缺点。在元组方法中，对于每个 i，$f(i,y)$ 都以数对 $(y,f(i,y))$ 的形式按 y 的递增次序存储于表 $P(i)$中。同时，由于 $f(i,y)$ 是 y 的非递减函数，因此 $P(i)$ 中各数对 $(y,f(i,y))$ 也是按 $f(i,y)$ 的递增次序排列的。

例 3-6 条件同例 3-5。对 f 的计算如图 15-2 所示。当 $i=5$ 时，f 由数对集合 $P(5)=[(0,0),(4,6)]$表示。而 $P(4)$、$P(3)$和 $P(2)$分别为$[(0,0),(4,6),(9,10)]$、$[(0,0)(4,6),(9,10),(10,11)]$ 和$[(0,0)(2,3)(4,6)(6,9)(9,10)(10,11)]$。

为求 $f(1,10)$，利用式（15-2）得 $f(1,10)=\max\{f(2,10),f(2,8)+p1\}$。由 $P(2)$得 $f(2,10)=11$、$f(2,8)=9$ ($f(2,8)=9$ 来自数对(6，9))，因此 $f(1,10)=\max\{11,15\}=15$。现在来求 x_i 的值，因为 $f(1,10)=f(2,6)+p_1$，所以 $x_1=1$；由 $f(2,6)=f(3,6-w2)+p_2=f(3,4)+p_2$，得 $x_2=1$；由 $f(3,4)=f(4,4)=f(5,4)$得 $x_3=x_4=0$；最后，因 $f(5,4)\neq 0$ 得 $x_5=1$。

检查每个 $P(i)$ 中的数对，可以发现每对 $(y,f(i,y))$ 对应于变量 $x_i,.,x_n$ 的 0/1 赋值的不同组合。设（a,b）和（c,d）是对应于两组不同 $x_i,.,x_n$ 的 0/1 赋值，若 $a\geqslant c$ 且 $b<d$，则 (a,b) 受 (b,c) 支配。被支配者不必加入 $P(i)$中。若在相同的数对中有两个或更多的赋值，则只有一个放入 $P(i)$。假设 $w_n\leqslant C$，$P(n)=[(0,0),(w_n,p_n)]$，$P(n)$ 中对应于 x_n 的两个数对分别等于 0 和 1。对于每个 i，$P(i)$可由 $P(i+1)$得出。首先，要计算数对的有序集合 Q，使得当且仅当 $w_i\leqslant s\leqslant c$ 且$(s-w_i,t-p_i)$为 $P(i+1)$ 中的一个数对时，（s,t）为 Q 中的一个数对。现在 Q 中包含 $x_i=1$ 时的数对集，而 $P(i+1)$ 对应于 $x_i=0$ 的数对集。接下来，合并 Q 和 $P(i+1)$并删除受支配者和重复值即可得到 $P(i)$。

例 3-7 各数据同例 15-6。$P(5)=[(0,0),(4,6)]$, 因此 $Q=[(5,4),(9,10)]$。现在要将 $P(5)$ 和 Q 合并得到 $P(4)$。因$(5,4)$受$(4,6)$支配，可删除$(5,4)$，所以 $P(4)=[(0,0),(4,6),(9,10)]$。接着计算 $P(3)$，首先由 $P(4)$得 $Q=[(6,5),(10,11)]$，然后又由合并方法得 $P(3)=[(0,0),(4,6),(9,10),(10,11)]$。最后计算 $P(2)$: 由 $P(3)$得 $Q=[(2,3),(6,9)]$，$P(3)$与 Q 合并得 $P(2)=[(0,0),(2,3),(4,6),(6,9),(9,10).(10,11)]$。因为每个 $P(i)$ 中的数对对应于 $x_i,.,x_n$ 的不同 0/1 赋值，因此 $P(i)$ 中的数对不会超过 2^{n-i+1} 个。计算 $P(i)$ 时，计算 Q 需消耗 $(|P(i+1)|)$ 的时间，合并 $P(i+1)$ 和 Q 同

样需要(|*P*(*i*+ 1) |)的时间。计算所有 *P*(*i*) 时所需要的总时间为： $(\sum_{i=2}^{n}|P(i+1)|=O(2^n)$。当权为整数时，$|P(i)|\leqslant c+1$, 此时复杂性为 O (m i n {*n c*, 2^n})。

如 6 . 4 . 3 节定义的，数字化图像是 $m\times m$ 的像素阵列。假定每个像素有一个 0～2 5 5 的灰度值。因此存储一个像素至多需 8 位。若每个像素存储都用最大位 8 位，则总的存储空间为 $8m^2$ 位。为了减少存储空间，我们将采用变长模式（ variable bit scheme），即不同像素用不同位数来存储。像素值为 0 和 1 时只需 1 位存储空间；值 2、3 各需 2 位；值 4，5，6 和 7 各需 3 位；以此类推，使用变长模式的步骤如下:

1) 图像线性化根据图 15-3a 中的折线将 $m\times m$ 维图像转换为 $1\times m^2$ 维矩阵。

2) 分段将像素组分成若干个段，分段原则是：每段中的像素位数相同。每个段是相邻像素的集合且每段最多含 2 5 6 个像素，因此，若相同位数的像素超过 2 5 6 个的话，则用两个以上的段表示。

3) 创建文件创建三个文件：*S e g m e n t L e n g t h*, *BitsPerPixel* 和 *P i x e l s*。第一个文件包含在 2)中所建的段的长度(减 1)，文件中各项均为 8 位长。文件 *BitsPerPixel* 给出了各段中每个像素的存储位数 (减 1)，文件中各项均为 3 位。文件 *Pixels* 则是以变长格式存储的像素的二进制串。

4) 压缩文件压缩在 3) 中所建立的文件，以减少空间需求。

上述压缩方法的效率（用所得压缩率表示）很大程度上取决于长段的出现频率。

例 3-8 考察图 15-3b 的 4 × 4 图像。按照蛇形的行主次序，灰度值依次为 1 0，9，1 2，4 0，5 0，3 5，1 5，1 2，8，1 0，9，1 5，11，1 3 0，1 6 0 和 2 4 0。各像素所需的位数分别为 4，4，4，6，6，6，4，4，4，4，4，4，8，8 和 8，按等长的条件将像素分段，可以得到 4 个段[1 0, 9, 1 2]、[4 0, 5 0, 3 5]、[15, 12, 8, 10, 9, 15, 11] 和[130, 160, 240]。因此，文件 *SegmentLength* 为 2，2，6，2; 文件 *BitsPerSegment* 的内容为 3，5，3，7; 文件 *P i x e l s* 包含了按蛇形行主次序排列的 1 6 个灰度值，其中头三个各用 4 位存储，接下来三个各用 6 位，再接下来的七个各用 4 位，最后三个各用 8 位存储。因此存储单元中前 3 0 位存储了前六个像素:

1010 1001 1100 111000 110010 100011

这三个文件需要的存储空间分别为：文件 *SegmentLength* 需 3 2 位; *BitsPerSegment* 需 1 2 位; *Pixels* 需 8 2 位，共需 1 2 6 位。而如果每个像素都用 8 位存储，则存储空间需 8 × 1 6 = 1 2 8 位，因而在本例图像中，节省了 2 位的空间。

假设在 2) 之后，产生了 *n* 个段。段标题（segment header）用于存储段的长度以及该段中每个像素所占用的位数。每个段标题需 11 位。现假设 l_i 和 b_i 分别表示第 *i* 段的段长和该段每个像素的长度，则存储第 *i* 段像素所需要的空间为 l_i*b_i。在 2) 中所得的三个文件的总存储空间为 $11n+\sum_{i=1}^{n}l_ib_i$。可通过将某些相邻段合并的方式来减少空间消耗。如当段 *i* 和 *i*+ 1 被合并时，合并后的段长应为 l_i+l_{i+1}。此时每个像素的存储位数为 $\max\{b_i, b_{i+1}\}$ 位。尽管这种技术增加了文件 P i x e l s 的空间消耗，但同时也减少了一个段标题的空间。

例 3-9 如果将例 1 5 - 8 中的第 1 段和第 2 段合并，合并后，文件 *S e g m e n t L e n g t h* 变为 5，6，2，*BitsPerSegment* 变为 5，3，7。而文件 Pixels 的前 3 6 位存储的是合并后的第一段: 001010 001001 001100 111000 110010 100011 其余的像素（例 1 5 - 8 第 3 段）没有改变。因为减少了 1 个段标题，文件 *S e g m e n t L e n g t h* 和 *BitsPerPixel* 的空间消耗共减少了 11 位，而文件 *Pixels* 的空间增加 6 位，因此总共节约的空间为 5 位，空间总消耗为 1 2 1 位。

我们希望能设计一种算法，使得在产生 *n* 个段之后，能对相邻段进行合并，以便产生一个具有最小空间需求的新的段集合。在合并相邻段之后，可利用诸如 L Z W 法（见 7 . 5 节）和霍夫曼编码（见 9 . 5 . 3 节）等其他技术来进一步压缩这三个文件。

令 s_q 为前 *q* 个段的最优合并所需要的空间。定义 $s_0=0$。考虑第 *i* 段($i>0$)，假如在最优合并 *C* 中，第 *i* 段与第 *i*- 1，*i*- 2，.，*i*-*r*+1 段相合并，而不包括第 *i*-*r* 段。合并 *C* 所需要的空间消耗等于: 第 1 段到第 *i*-r 段所需空间+ l s u m (*i*-*r*+ 1 ,*i*) * b m a x (*i*-*r*+ 1 ,*i*) + 11

其中 $lsum(a, b)=\sum_{j=a}^{b}$
l_j，$bmax(a, b)=\max\{b_a, ..., b_b\}$。假如在 C 中第 1 段到第 $i\text{-}r$ 段的合并不是最优合并，那么需要对合并进行修改，以使其具有更小的空间需求。因此还必须对段 1 到段 $i\text{-}r$ 进行最优合并，也即保证最优原则得以维持。故 C 的空间消耗为:
$s_i = s_{i-r} + lsum(i\text{-}r+1, i) * bmax(i\text{-}r+1, i) + 11$
r 的值介于 1 到 i 之间，其中要求 $lsum$ 不超过 256 (因为段长限制在 256 之内)。尽管我们不知道如何选择 r，但我们知道，由于 C 具有最小的空间需求，因此在所有选择中， r 必须产生最小的空间需求。

假定 kay_i 表示取得最小值时 k 的值，s_n 为 n 段的最优合并所需要的空间，因而一个最优合并可用 kay 的值构造出来。

例 3-10 假定在 2) 中得到五个段，它们的长度为[6，3，10，2，3]，像素位数为[1，2，3，2，1]，要用公式（15-3）计算 s_n，必须先求出 s_{n-1}，.，s_0 的值。s_0 为 0，现计算 s_1: $s_1=s_0+l_1*b_1+11=17$ $kay_1=1$ s_2 由下式得出:
$s_2=\min\{s_1+l_2b_2, s_0+(l_1+l_2)*\max\{b_1, b_2\}\}+11=\min\{17+6, 0+9*2\}+11=29$
$kay_2=2$
以此类推，可得 $s_1.s_5=$[17，29，67，73，82] ，$kay_1.kay_5=$[1，2，2，3，4]。因为 $s_5=82$，所以最优空间合并需 82 位的空间。可由 kay_5 导出本合并的方式，过程如下: 因为 $kay_5=4$，所以 s_5 是由公式（15-3）在 k=4 时取得的，因而最优合并包括: 段 1 到段(5-4)=1 的最优合并以及段 2，3，4 和 5 的合并。最后仅剩下两个段: 段 1 以及段 2 到段 5 的合并段。

1. 递归方法

用递归式（15-3）可以递归地算出 s_i 和 kay_i。程序 15-3 为递归式的计算代码。l，b，和 kay是一维的全局整型数组，L 是段长限制（ 256），header 为段标题所需的空间(11)。调用 S(n)返回 s_n 的值且同时得出 kay 值。调用 Traceback(kay, n)可得到最优合并。

现讨论程序 15-3 的复杂性。$t(0)=c$（c 为一个常数）： （n>0），因此利用递归的方法可得 $t(n)=O(2^n)$。Traceback 的复杂性为(n)。

程序 15-3 递归计算 s , kay 及最优合并

```
int S(int i)
{ //返回 S(i)并计算 kay[i]
if (i == 0 ) return 0;
//k = 1 时, 根据公式（ 15-3）计算最小值
int lsum = l[i],bmax = b[i];
int s = S(i-1) + lsum * bmax;
kay[i] = 1;
//对其余的 k 计算最小值并求取最小值
for (int k = 2; k <= i && lsum+l[i-k+1] <= L; k++) {
lsum += l[i-k+1];
if (bmax < b[i-k+1]) bmax = b[i-k+1];
int t = S(i-k);
if (s > t + lsum * bmax) {
s = t + lsum * bmax;
kay[i] = k;}
}
return s + header;
}
```

```
void Traceback(int kay[], int n)
{// 合并段
if (n == 0) return;
Tr a c e b a c k ( k a y, n-kay[n]);
cout << "New segment begins at " << (n - kay[n] + 1) << endl;
}
```

2. 无重复计算的递归方法

通过避免重复计算 s_i，可将函数 S 的复杂性减少到(n)。注意这里只有 n 个不同的 s_i。

例 3 - 11 再考察例 15 - 10 中五个段的例子。当计算 s_5 时，先通过递归调用来计算 s_4，.，s_0。计算 s_4 时，通过递归调用计算 s_3，.，s_0，因此 s_4 只计算了一次，而 s_3 计算了两次，每一次计算 s_3 要计算一次 s_2，因此 s_2 共计算了四次，而 s_1 重复计算了 16 次！可利用一个数组 s 来保存先前计算过的 s_i 以避免重复计算。改进后的代码见程序 15 - 4，其中 s 为初值为 0 的全局整型数组。

程序 15-4 避免重复计算的递归算法

```
int S(int i)
{ / /计算 S ( i )和 k a y [ i ]
/ /避免重复计算
if (i == 0) return 0;
if (s[i] > 0) return s[i]; //已计算完
/ /计算 s [ i ]
/ /首先根据公式（15 - 3）计算 k = 1 时最小值
int lsum = l[i], bmax = b[i];
s[i] =S(i-1) + lsum * bmax;
kay[i] = 1;
/ /对其余的 k 计算最小值并更新
for (int k = 2; k <= i && lsum+l[i-k+1] <= L; k++) {
lsum += l[i-k+1];
if (bmax < b[i-k+1]) bmax = b[i-k+1];
int t = S(i-k);
if (s[i] > t + lsum * bmax) {
s[i] = t + lsum * bmax;
kay[i] = k;}
}
s[i] += header;
return s[i];
}
```

为了确定程序 15 - 4 的时间复杂性，我们将使用分期计算模式（ amortization scheme）。在该模式中，总时间被分解为若干个不同项，通过计算各项的时间然后求和来获得总时间。当计算 s_i 时，若 s_j 还未算出，则把调用 S(j) 的消耗计入 s_j；若 s_j 已算出，则把 $S(j)$ 的消耗计入 s_i(这里 s_j依次把计算新 s_q 的消耗转移至每个 s_q)。程序 15 - 4 的其他消耗也被计入 s_i。因为 L 是 256 之内的常数且每个 l_i 至少为 1，所以程序 15 - 4 的其他消耗为(1)，即计入每个 s_i 的量是一个常数，且 s_i 数目为 n，因而总工作量为(n)。

3. 迭代方法

倘若用式（15 - 3）依序计算 s_1, ., s_n，便可得到一个复杂性为(n)的迭代方法。在该方法中，

在 s_i 计算之前， s_j 必须已计算好。该方法的代码见程序 15-5，其中仍利用函数 Traceback（见程序 15-3）来获得最优合并。

程序 15-5 迭代计算 s 和 kay

```
void Vbits (int l[], int b[], int n, int s[], int kay[])
{ //计算 s[i]和 kay[i]
int L = 256, header = 11 ;
s[0] = 0;
//根据式（15-3）计算 s[i]
for (int i = 1; i <= n; i++) {
// k = 1 时,计算最小值
int lsum = l{i},
bmax = b[i];
s[i] = s[i-1] + lsum * bmax;
kay[i] = 1;
//对其余的 k 计算最小值并更新
for (int k=2; k<= i && lsum+l[i-k+1]<= L; k++) {
lsum += l[i-k+1];
if (bmax < b[i-k+1]) bmax = b[i-k+1];
if (s[i] > s[i-k] + lsum * bmax){
s[i] = s[i-k] + lsum * bmax;
kay[i] = k; }
}
s[i] += header;
}
}
```

3.2.3 矩阵乘法链

$m \times n$ 矩阵 A 与 $n \times p$ 矩阵 B 相乘需耗费$(m\,n\,p)$的时间（见第 2 章练习 16）。我们把 $m\,n\,p$ 作为两个矩阵相乘所需时间的测量值。现假定要计算三个矩阵 A、B 和 C 的乘积，有两种方式计算此乘积。在第一种方式中，先用 A 乘以 B 得到矩阵 D，然后 D 乘以 C 得到最终结果，这种乘法的顺序可写为$(A*B)*C$。第二种方式写为 $A*(B*C)$,道理同上。尽管这两种不同的计算顺序所得的结果相同，但时间消耗会有很大的差距。

例 3-12 假定 A 为 100×1 矩阵，B 为 1×100 矩阵，C 为 100×1 矩阵，则 $A*B$ 的时间耗费为 10 000，得到的结果 D 为 100×100 矩阵，再与 C 相乘所需的时间耗费为 1 000 000，因此计算$(A*B)*C$ 的总时间为 1 010 000。$B*C$ 的时间耗费为 10 000，得到的中间矩阵为 1×1 矩阵，再与 A 相乘的时间消耗为 100，因而计算 $A*(B*C)$的时间耗费竟只有 10 100！而且，计算（ $A*B$ ）$*C$ 时，还需 10 000 个单元来存储 $A*B$，而 $A*(B*C)$ 计算过程中，只需用 1 个单元来存储 $B*C$。

下面举一个得益于选择合适秩序计算 $A*B*C$ 矩阵的实例：考虑两个 3 维图像的匹配。图像匹配问题的要求是，确定一个图像需旋转、平移和缩放多少次才能逼近另一个图像。实现匹配的方法之一便是执行约 100 次迭代计算，每次迭代需计算 12×1 个向量 T:

$$T=\sum A(x, y, z) * B(x, y, z) * C(x, y, z)$$

其中 A，B 和 C 分别为 12×3，3×3 和 3×1 矩阵。(x, y, z) 为矩阵中向量的坐标。设 t 表示计算 $A(x, y, z) * B(x, y, z) * C(x, y, z)$的计算量。假定此图像含 $256 \times 256 \times 256$ 个向量，在此条件中，这 100 个迭代所需的总计算量近似为 $100 * 256^3 * t \approx 1.7 * 10^9 t$。若三个矩阵是按由左向

右的顺序相乘的，则 $t=12*3*3+12*3*1=144$; 但如果从右向左相乘， $t=3*3*1+12*3*1=45$。由左至右计算约需 $2.4*10^{11}$ 个操作，而由右至左计算大概只需 $7.5*10^{10}$ 个操作。假如使用一个每秒可执行 1 亿次操作的计算机，由左至右需 40 分钟，而由右至左只需 12.5 分钟。

在计算矩阵运算 *A*B*C* 时，仅有两种乘法顺序（由左至右或由右至左），所以可以很容易算出每种顺序所需要的操作数，并选择操作数比较少的那种乘法顺序。但对于更多矩阵相乘来说，情况要复杂得多。如计算矩阵乘积 $M_1\times M_2\times\cdots\times M_q$，其中 M_i 是一个 $r_i\times r_{i+1}$ 矩阵($1\leqslant i\leqslant q$)。不妨考虑 $q=4$ 的情况，此时矩阵运算 *A*B*C*D* 可按以下方式（顺序）计算:

*A** ((*B*C*) **D*) *A** (*B** (*C*D*)) (*A*B*) * (*C*D*) (*A** (*B*C*)) **D*

不难看出计算的方法数会随 q 以指数级增加。因此，对于很大的 q 来说，考虑每一种计算顺序并选择最优者已是不切实际的。

现在要介绍一种采用动态规划方法获得矩阵乘法次序的最优策略。这种方法可将算法的时间消耗降为(q^3)。用 M_{ij} 表示链 $M_i\times\cdots\times M_j$（$i\leqslant j$）的乘积。设 $c(i,j)$ 为用最优法计算 M_{ij} 的消耗，$kay(i,j)$ 为用最优法计算 M_{ij} 的最后一步 $M_{ik}\times M_{k+1,j}$ 的消耗。因此 M_{ij} 的最优算法包括如何用最优算法计算 M_{ik} 和 M_{kj} 以及计算 $M_{ik}\times M_{kj}$。根据最优原理，可得到如下的动态规划递归式: $kay(i,i+s)=$ 获得上述最小值的 k. 以上求 c 的递归式可用递归或迭代的方法来求解。$c(1,q)$ 为用最优法计算矩阵链的消耗，$kay(1,q)$ 为最后一步的消耗。其余的乘积可由 kay 值来确定。

1. 递归方法

与求解 0/1 背包及图像压缩问题一样，本递归方法也须避免重复计算 $c(i,j)$ 和 $kay(i,j)$，否则算法的复杂性将会非常高。

例 3-13 设 $q=5$ 和 $r=(10,5,1,10,2,10)$，式中待求的 c 中有四个 c 的 $s=0$ 或 1，因此用动态规划方法可立即求得它们的值: $c(1,1)=c(5,5)=0$;$c(1,2)=50$; $c(4,5)=200$。现计算 $C(2,5)$: $c(2,5)=\min\{c(2,2)+c(3,5)+50, c(2,3)+c(4,5)+500, c(2,4)+c(5,5)+100\}$（15-5）其中 $c(2,2)=c(5,5)=0$; $c(2,3)=50$; $c(4,5)=200$ 。再用递归式计算 $c(3,5)$及 $c(2,4)$:$c(3,5)=\min\{c(3,3)+c(4,5)+100, c(3,4)+c(5,5)+20\}=\min\{0+200+100, 20+0+20\}=40$ $c(2,4)=\min\{c(2,2)+c(3,4)+10, c(2,3)+c(4,4)+100\}=\min\{0+20+10, 50+10+20\}=30$ 由以上计算还可得 $kay(3,5)=4$，$kay(2,4)=2$。现在，计算 $c(2,5)$ 所需的所有中间值都已求得，将它们代入式（15-5）得:

$c(2,5)=\min\{0+40+50, 50+200+500, 30+0+100\}=90$ 且 $kay(2,5)=2$

再用式（15-4）计算 $c(1,5)$，在此之前必须算出 $c(3,5)$、$c(1,3)$ 和 $c(1,4)$。同上述过程，亦可计算出它们的值分别为 40、150 和 90，相应的 kay 值分别为 4、2 和 2。代入式（15-4）得:

$c(1,5)=\min\{0+90+500, 50+40+100, 150+200+1000, 90+0+200\}=190$ 且 $kay(1,5)=2$

此最优乘法算法的消耗为 190，由 $kay(1,5)$ 值可推出该算法的最后一步， $kay(1,5)$ 等于 2，因此最后一步为 $M_{12}\times M_{35}$，而 M_{12} 和 M_{35} 都是用最优法计算而来。由 $kay(1,2)=1$ 知 M_{12} 等于 $M_{11}\times M_{22}$，同理由 $kay(3,5)=4$ 得知 M_{35} 由 $M_{34}\times M_{55}$ 算出。依此类推，M_{34} 由 $M_{33}\times M_{44}$ 得出。因而此最优乘法算法的步骤为:

$M_{11}\times M_{22}=M_{12}$

$M_{33}\times M_{44}=M_{34}$

$M_{34}\times M_{55}=M_{35}$

$M_{12}\times M_{35}=M_{15}$

计算 $c(i,j)$ 和 kay(i,j) 的递归代码见程序 15-6。在函数 *C* 中，r 为全局一维数组变量，kay是全局二维数组变量，函数 *C* 返回 $c(ij)$ 之值且置 kay$[a][b]=kay(a,b)$ (对于任何 a, b)，其中 $c(a,b)$在计算 $c(i,j)$ 时皆已算出。函数 Traceback 利用函数 C 中已算出的 kay 值来推导出

最优乘法算法的步骤。

设 $t(q)$为函数 C 的复杂性，其中 $q=j-i+1$（即 M_{ij} 是 q 个矩阵运算的结果）。当 q 为 1 或 2 时，$t(q)=d$，其中 d 为一常数；而 $q>2$ 时，$t(q)=2_{q-1}\sum_{k=1}t(k)+e\,q$，其中 e 是一个常量。因此当 $q>2$ 时，$t(q)>2t(q-1)+e$，所以 $t(q)=\Omega(2_q)$。函数 Traceback 的复杂性为(q)。

程序 15-6 递归计算 *c* (*i*, *j*) 和 *kay* (*i*, *j*)

```
int C(int i, int j)
{ / /返回 c(i,j) 且计算 k(i,j) = kay[i][j]
if (i==j) return 0; //一个矩阵的情形
if (i == j-1) { //两个矩阵的情形
kay[i][i+1] = i;
return r[i]*r[i+1]*r[r+2];}
/ /多于两个矩阵的情形
/ /设 u 为 k = i 时的最小值
int u = C(i,i) + C(i+1,j) + r[i]*r[i+1]*r[j+1];
kay[i][j] = i;
/ /计算其余的最小值并更新 u
for (int k = i+1; k < j; k++) {
int t = C(i,k) + C(k+1,j) + r[i]*r[k+1]*r[j+1];
if (r < u) {//小于最小值的情形
u = t;
kay[i][j] = k;
}
return u;
}
void Traceback (int i, int j ,int **kay)
{ / /输出计算 Mij 的最优方法
if ( i == j) return;
Traceback(i, kay[i][j], kay);
Traceback(kay[i][j]+1, j, kay);
cout << "Multiply M" << i << ", "<< kay[i][j];
cout << " and M " << (kay[i][j]+1) << ", " << j << end1;
}
```

2. 无重复计算的递归方法

若避免再次计算前面已经计算过的 c（及相应的 $k\,a\,y$），可将复杂性降低到（q_3）。而为了避免重复计算，需用一个全局数组 c[ ][ ]存储 $c(i,j)$ 值，该数组初始值为 0。函数 C 的新代码见程序 1 5 - 7:

程序 15-7 无重复计算的 *c* (*i*, *j*) 计算方法

```
int C(int i,int j)
{ / /返回 c(i,j) 并计算 k a y ( i , j ) = k a y [ I ] [ j ]
/ /避免重复计算
/ /检查是否已计算过
if （ c[i][j] > ） return c[i][j];
/ /若未计算,则进行计算
if(i==j) return 0; //一个矩阵的情形
```

```
if ( i = = j - 1 ) { / /两个矩阵的情形
kay[i][i+1]=i;
c [ i ] [ j ] = r [ i ] * r [ i + 1 ] * r [ i + 2 ] ;
return c[i][j];}
/ /多于两个矩阵的情形
/ /设 u 为 k = i 时的最小值
int u=C(i,i)+C(i+1,j)+r[i]*r[i+1]*r[j+1];
k a y [ i ] [ j ] = i ;
/ /计算其余的最小值并更新 u
for (int k==i+1; k<j;k++){
int t=C(i,k)+C(k+1,j)+r[i]*r[k+1]*r[j+1];
if (t<u) {// 比最小值还小
u = t ;
k a y [ i ] [ j ] = k ; }
}
c [ i ] [ j ] = u ;
return u;
}
```

为分析改进后函数 C 的复杂性，再次使用分期计算方法。注意到调用 C(1, q) 时每个 $c(i,j)$（$1 \leqslant i \leqslant j \leqslant q$）仅被计算一次。要计算尚未计算过的 $c(a,b)$，需附加的工作量 $s=j-i>1$。将 s 计入第一次计算 $c(a, b)$ 时的工作量中。在依次计算 $c(a, b)$ 时，这个 s 会转计到每个 $c(a, b)$ 的第一次计算时间 c 中，因此每个 $c(i, i)$ 均被计入 s。对于每个 s，有 $q-s+1$ 个 $c(i, j)$ 需要计算，因此总的工作消耗为 $\sum_{s=1}^{q-1}(q-s+1)=(q^3)$。

3. 迭代方法

c 的动态规划递归式可用迭代的方法来求解。若按 $s=2, 3, ., q-1$ 的顺序计算 $c(i, i+s)$，每个 c 和 kay 仅需计算一次。

例 3-14 考察例 3 - 1 3 中五个矩阵的情况。先初始化 $c(i, i)$ $(0 \leqslant i \leqslant 5)$ 为 0，然后对于 i=1, ., 4 分别计算 $c(i, i+1)$。$c(1, 2)= r_1 r_2 r_3 = 50$，$c(2, 3)= 50$，$c(3,4)=20$ 和 $c(4, 5) = 2000$。相应的 kay 值分别为 1，2，3 和 4。

当 $s=2$ 时，可得:

$c(1,3) = \min\{c(1,1)+c(2,3)+r_1 r_2 r_4, c(1,2)+c(3,3)+r_1 r_3 r_4\}=\min\{0+50+500,50+0+100\}$
$=150$

且 $kay(1,3)=2$。用相同方法可求得 $c(2,4)$和 $c(3,5)$分别为 30 和 40，相应 kay 值分别为 2 和 3。

当 $s=3$ 时，需计算 $c(1,4)$ 和 $c(2,5)$。计算 $c(2,5)$ 所需要的所有中间值均已知(见(15 - 5)式)，代入计算公式后可得 $c(2,5)=90$，$kay(2,5)=2$。$c(1,4)$可用同样的公式计算。最后，当 $s=4$ 时，可直接用（15 - 4）式来计算 $c(1,5)$，因为该式右边所有项都已知。

计算 c 和 kay 的迭代程序见函数 MatrixChain(见程序 15 - 8),该函数的复杂性为(q^3)。计算出 kay 后同样可用程序 15 - 6 中的 Traceback 函数推算出相应的最优乘法计算过程。

程序 15-8 c 和 kay 的迭代计算

```
void MatrixChain(int r[], int q, int **c, int **kay)
{// 为所有的 Mij 计算耗费和 k a y
// 初始化 c[i][i], c[i][i+1]和 k a y [ i ] [ i + 1 ]
for (int i = 1; i < q; i++) {
```

```
c[i][i] = 0;
c[i][i+1] = r[i]*r[i+1]*r[i+2];
kay[i][i+1] = i;
}
c[q][q] = 0;
/ /计算余下的 c 和 k a y
for (int s = 2; s < q; s++)
for (int i = 1; i <= q - s; i++) {
// k = i 时的最小项
c[i][i+s] = c[i][i] + c[i+1][i+s] + r[i]*r[i+1]*r[i+s+1];
kay[i][i+s] = i;
// 余下的最小项
for (int k = i+1; k < i + s; k++) {
int t = c[i][k] + c[k+1][i+s] + r[i]*r[k+1]*r[i+s+1];
if (t < c[i][i+s]) {// 更小的最小项
c[i][i+s] = t;
kay[i][i+s] = k;}
}
}
}
```

3.2.4 最短路径

假设 G 为有向图，其中每条边都有一个长度（或耗费），图中每条有向路径的长度等于该路径中各边的长度之和。对于每对顶点(i, j)，在顶点 i 与 j 之间可能有多条路径，各路径的长度可能各不相同。我们定义从 i 到 j 的所有路径中，具有最小长度的路径为从 i 到 j 的最短路径。

例 3-15 如图 1 5 - 4 所示。从顶点 1 到顶点 3 的路径有

1) 1,2,5,3

2) 1,4,3

3) 1,2,5,8,6,3

4) 1,4,6,3

由该图可知,各路径相应的长度为 1 0、2 8、9、2 7，因而路径 3) 是该图中顶点 1 到顶点 3 的最短路径。

在所有点对最短路径问题（ a l l - p a i r sshorest-paths problem）中，要寻找有向图 G 中每对顶点之间的最短路径。也就是说，对于每对顶点(i, j)，需要寻找从 i 到 j 的最短路径及从 j 到 i 的最短路径。因此对于一个 n 个顶点的图来说，需寻找 $p=n(n\text{-}1)$ 条最短路径。假定图 G 中不含有长度为负数的环路，只有在这种假设下才可保证 G 中每对顶点(i, j) 之间总有一条不含环路的最短路径。当有向图中存在长度小于 0 的环路时，可能得到长度为 $-\infty$ 的更短路径，因为包含该环路的最短路径往往可无限多次地加上此负长度的环路。

设图 G 中 n 个顶点的编号为 1 到 n。令 c (i, j, k)表示从 i 到 j 的最短路径的长度，其中 k 表示该路径中的最大顶点。因此，如果 G 中包含边$<i, j>$，则 $c(i, j, 0)$ =边$<i, j>$ 的长度；若 $i=j$ ，则 $c(i,j, 0)=0$；如果 G 中不包含边$<i, j>$，则 c $(i, j, 0)=+\infty$。$c(i, j, n)$ 则是从 i 到 j 的最短路径的长度。

例 3-16 考察图 1 5 - 4。若 k=0, 1, 2, 3，则 c (1, 3, k)= ∞；c (1, 3, 4)= 2 8；若 k = 5, 6, 7，则 c (1, 3,k) = 1 0；若 k=8, 9, 10，则 c (1, 3, k) = 9。因此 1 到 3 的最短路径长度为 9。对于任意 $k>0$,

如何确定 $c(i,j,k)$ 呢？中间顶点不超过 k 的 i 到 j 的最短路径有两种可能：该路径含或不含中间顶点 k。若不含，则该路径长度应为 $c(i,j,k-1)$，否则长度为 $c(i,k,k-1)+c(k,j,k-1)$。$c(i,j,k)$ 可取两者中的最小值。因此可得到如下递归式：

$c(i,j,k)=\min\{c(i,j,k-1), c(i,k,k-1)+c(k,j,k-1)\}$，$k>0$

以上的递归公式将一个 k 级运算转化为多个 k-1 级运算，而多个 k-1 级运算应比一个 k 级运算简单。如果用递归方法求解上式，则计算最终结果的复杂性将无法估量。令 $t(k)$ 为递归求解 $c(i,j,k)$ 的时间。根据递归式可以看出 $t(k)=2t(k-1)+c$。利用替代方法可得 $t(n)=(2^n)$。因此得到所有 $c(i,j,n)$ 的时间为$(n^2 2^n)$。

当注意到某些 $c(i,j,k-1)$ 值可能被使用多次时，可以更高效地求解 $c(i,j,n)$。利用避免重复计算 $c(i,j,k)$ 的方法，可将计算 c 值的时间减少到(n^3)。这可通过递归方式（见程序 15-7 矩阵链问题）或迭代方式来实现。出迭代算法的伪代码如图 15-5 所示。

```
//寻找最短路径的长度
//初始化 c（i，j，1）
for （int i=1;  i<=n;i++）
  for (int j=1;j<=n;j++)
    c(i,j,0)=a(i,j); // a 是长度邻接矩阵
    //计算 c(i,j,k) (0<k<=n)
for(int k=1;k<=n;k++)
  for (int i=1;i<=n;i++)
    for (int j=1;j<=n;j++)
      if (c(i,k,k-1)+c(k,j,k-1)<c(i,j,k-1))
        c(i,j,k)=c(i,k,k-1)+c(k,j,k-1);
      else c(i,j,k)=c(i,j,k-1);
```

图 15-5 最短路径算法的伪代码

注意到对于任意 i，$c(i,k,k)=c(i,k,k-1)$且 $c(k,i,k)=c(k,i,k-1)$，因而，若用 $c(i,j)$代替图 15-5 的 $c(i,j,k)$，最后所得的 $c(i,j)$ 之值将等于 $c(i,j,n)$ 值。此时图 15-5 可改写成程序 15-9 的 C++代码。程序 15-9 中还利用了程序 12-1 中定义的 AdjacencyWDigraph 类。函数 AllPairs 在 c 中返回最短路径的长度。若 i 到 j 无通路，则 c[i][j]被赋值为 NoEdge。函数 AllPairs 同时计算了 kay[i][j]，其中 kay[i][j] 表示从 i 到 j 的最短路径中最大的 k 值。在后面将看到如何根据 kay 值来推断出从一个顶点到另一顶点的最短路径（见程序 15-10 中的函数 OutputPath）。程序 15-9 的时间复杂性为(n^3)，其中输出一条最短路径的实际时间为 O(n)。

程序 15-9 c 和 kay 的计算

```
template<class T>
void AdjacencyWDigraph<T>::Allpairs(T **c, int **kay)
{//所有点对的最短路径
//对于所有 i 和 j，计算 c[i][j]和 kay[i][j]
//初始化 c[i][j]=c（i，j，0）
for (int i = 1; i <= n; i++)
for (int j = 1; j <= n; j++) {
c[i][j] = a[i][j];
kay[i][j] = 0;
}
```

```
for (i = 1; i <= n; i++)
c[i][i] = 0;
// 计算 c[i][j] = c(i,j,k)
for (int k = 1; k <= n; k++)
for (int i = 1; i <= n; i++)
for (int j = 1; j <= n; j++) {
T t1 = c[i][k];
T t2 = c[k][j];
T t3 = c[i][j];
if (t1 != NoEdge && t2 != NoEdge && (t3 == NoEdge || t1 + t2 < t3)) {
c[i][j] = t1 + t2;
kay[i][j] = k;}
}
}
```

程序 15-10 输出最短路径

```
void outputPath(int **kay, int i, int j)
{// 输出 i 到 j 的路径的实际代码
if (i == j) return;
if (kay[i][j] == 0) cout << j << ' ';
else {outputPath(kay, i, kay[i][j]);
outputPath(kay, kay[i][j], j);}
}
template<class T>
void OutputPath(T **c, int **kay, T NoEdge, int i, int j)
{// 输出从 i 到 j 的最短路径
if (c[i][j] == NoEdge) {
cout << "There is no path from " << i << " to " << j << endl;
return;}
cout << "The path is" << endl;
cout << i << ' ';
outputPath(kay, i, j);
cout << endl;
}
```

例 3-17 图 15-6a 给出某图的长度矩阵 a，15-6b 给出由程序 15-9 所计算出的 c 矩阵，15-6c 为对应的 kay 值。根据 15-6c 中的 kay 值，可知从 1 到 5 的最短路径是从 1 到 kay[1][5]=4 的最短路径再加上从 4 到 5 的最短路径，因为 kay[4][5]=0，所以从 4 到 5 的最短路径无中间顶点。从 1 到 4 的最短路径经过 kay[1][4]=3。重复以上过程，最后可得 1 到 5 的最短路径为：1，2，3，4，5。

3.2.5 网络的无交叉子集

在 11.5.3 节的交叉分布问题中，给定一个每边带 n 个针脚的布线通道和一个排列 C。顶部的针脚 i 与底部的针脚 C_i 相连，其中 $1 \leqslant i \leqslant n$，数对$(i, C_i)$ 称为网组。总共有 n 个网组需连接或连通。假设有两个或更多的布线层，其中有一个为优先层，在优先层中可以使用更细的连线，其电阻也可能比其他层要小得多。布线时应尽可能在优先层中布设更多的网组。而剩下的其他网组将布设在其他层。当且仅当两个网组之间不交叉时，它们可布设在同一层。我们的任务是寻找

一个最大无交叉子集（Maximum Noncrossing Su b s e t，M N S）。在该集中，任意两个网组都不交叉。因(i, C_i) 完全由 i 决定，因此可用 i 来指定(i, C_i)。

例 3-18 考察图 1 5 - 7（对应于图 1 0 - 1 7）。(1 , 8)和(2 , 7)（也即 1 号网组和 2 号网组）交叉，因而不能布设在同一层中。而(1 , 8)，(7,9) 和(9,10) 未交叉，因此可布设在同一层。但这 3 个网组并不能构成一个 M N S，因为还有更大的不交叉子集。图 1 0 - 1 7 中给出的例子中，集合 { (4 , 2),(5 , 5) , (7 , 9) , (9 , 1 0)} 是一个含 4 个网组的 M N S。

设 $MNS(i, j)$ 代表一个 M N S，其中所有的(u, C_u) 满足 $u \leqslant i$，$C_u \leqslant j$。令 $size(i,j)$ 表示 $MNS(i,j)$的大小(即网组的数目)。显然 $MNS(n,n)$是对应于给定输入的 M N S，而 $size(n,n)$是它的大小。

例 3-19 对于图 1 0 - 1 7 中的例子，$MNS(1 0 , 1 0)$是我们要找的最终结果。如例 3 - 1 8 中所指出的，$size(1 0 , 1 0) = 4$，因为(1 , 8)，(2 , 7)，(7 , 9)，(8 , 3)，(9 , 1 0)和(1 0 , 6)中要么顶部针脚编号比 7 大，要么底部针脚编号比 6 大，因此它们都不属于 $MNS(7 , 6)$。因此只需考察剩下的 4 个网组是否属于 $MNS(7 , 6)$，如图 1 5 - 8 所示。子集 {(3 , 4) , (5 , 5)} 是大小为 2 的无交叉子集。没有大小为 3 的无交叉子集，因此 $size(7 , 6) = 2$。

当 $i= 1$ 时，$(1 ,C_1)$ 是 $MNS(1 ,j)$ 的唯一候选。仅当 $j \geqslant C_1$ 时，这个网组才会是 $MNS(1 ,j)$ 的一个成员 .

下一步，考虑 $i > 1$ 时的情况。若 $j < C_i$，则(i,C_i) 不可能是 $MNS(i,j)$ 的成员，所有属于 $MNS(i,j)$ 的(u, C_u) 都需满足 $u < i$ 且 $C_u < j$，因此：$size(i,j) =size(i- 1 ,j), j<C_i$（1 5 - 7）

若 $j \geqslant C_i$，则(i,C_i) 可能在也可能不在 $MNS(i,j)$ 内。若(i,C_1) 在 $MNS(i,j)$ 内，则在 $MNS(i,j)$ 中不会有这样的(u,C_u)：$u < i$ 且 $C_u > C_i$，因为这个网组必与(i, C_i) 相交。因此 $MNS(i,j)$ 中的其他所有成员都必须满足条件 $u < i$ 且 $C_u < C_i$。在 $MNS(i,j)$ 中这样的网组共有 M_{i-1}, C_{i-1} 个。若(i,C_i)不在 $MNS(i,j)$中，则 $MNS(i,j)$ 中的所有(u, C_u) 必须满足 $u < i$；因此 $size(i,j)=size(i- 1 ,j)$。虽然不能确定(i, C_i)是否在 $MNS(i,j)$ 中，但我们可以根据获取更大 M N S 的原则来作出选择。因此：$size(i,j) = max \{size(i-1 ,j), size(i- 1 ,C_i-1)+1\}, j \geqslant C_i$（1 5 - 8）

虽然从（1 5 - 6）式到（ 1 5 - 8）式可用递归法求解，但从前面的例子可以看出，即使避免了重复计算，动态规划递归算法的效率也不够高，因此只考虑迭代方法。在迭代过程中先用式（1 5 - 6）计算出 $size (1 ,j)$，然后再用式（1 5 - 7）和（1 5 - 8）按 i=2, 3, ., n 的顺序计算 $size (i,j)$，最后再用 Traceback 来得到 $MNS(n, n)$ 中的所有网组。

例 3-20 图 1 5 - 9 给出了图 1 5 - 7 对应的 $size(i,j)$ 值。因 $size(1 0 , 1 0) = 4$，可知 M N S 含 4 个网组。为求得这 4 个网组，先从 $size (1 0 , 1 0)$入手。可用（1 5 - 8）式算出 $size(1 0 , 1 0)$。根据式（1 5 - 8）时的产生原因可知 $size (1 0 , 1 0)=size(9 , 1 0)$，因此现在要求 $MNS (9 , 1 0)$。由于 $MNS (1 0 , 1 0) \neq size(8 , 1 0)$，因此 $MNS (9,10)$ 中必包含 9 号网组。$MNS(9,10)$ 中剩下的网组组成 $MNS (8 , C_9- 1)=MNS(8 , 9)$。由 $MNS(8 , 9) =MNS (7,9)$ 知，8 号网组可以被排除。接下来要求 $MNS(7 , 9)$，因为 $size(7 , 9) \neq size(6 , 9)$，所以 M N S 中必含 7 号网组。$MNS (7,9)$ 中余下的网组组成 $MNS (6 , C_7- 1) =MNS(6 , 8)$。根据 $size(6 , 8) =size(5 , 8)$ 可排除 6 号网组。按同样的方法， 5 号网组，3 号网组加入 M N S 中，而 4 号网组等其他网组被排除。因此回溯过程所得到的大小为 4 的 M N S 为{ 3 , 5 , 7 , 9 }。

注意到在回溯过程中未用到 $size(1 0 ,j) (j \neq 1 0)$，因此不必计算这些值。

程序 1 5 - 11 给出了计算 $size (i , j)$ 的迭代代码和输出 M N S 的代码。函数 M N S 用来计算 $size (i,j)$ 的值，计算结果用一个二维数组 M N 来存储。size[i][j] 表示 $size(i,j)$，其中 i=j= n 或 $1 \leqslant i < n$，$0 \leqslant j \leqslant n$，计算过程的时间复杂性为$(n_2)$。函数 Traceback 在 N et［0 : m - 1］中输出所得到的 MNS，其时间复杂性为(n)。因此求解 M M S 问题的动态规划算法的总的时间复杂性为(n_2)。

程序 1 5 - 11 寻找最大无交叉子集

```
void MNS(int C[], int n, int **size)
{ / /对于所有的 i 和 j，计算 s i z e [ i ] [ j ]
/ /初始化 s i z e [ 1 ] [ * ]
for (int j = 0; j < C[1]; j++)
size[1][j] = 0;
for (j = C[1]; j <= n; j++)
size[1][j] = 1;
// 计算 size[i][*], 1 < i < n
for (int i = 2; i < n; i++) {
for (int j = 0; j < C[i]; j++)
size[i][j] = size[i-1][j];
for (j = C[i]; j <= n; j++)
size[i][j] = max(size[i-1][j], size[i-1][C[i]-1]+1);
}
size[n][n] = max(size[n-1][n], size[n-1][C[n]-1]+1);
}
void Traceback(int C[], int **size, int n, int Net[], int& m)
{// 在 N e t [ 0 : m - 1 ]中返回 M M S
int j = n; // 所允许的底部最大针脚编号
m = 0; // 网组的游标
for (int i = n; i > 1; i--)
// i 号 n e t 在 M N S 中?
if (size[i][j] != size[i-1][j]){// 在 M N S 中
Net[m++] = i;
j = C[i] - 1;}
// 1 号网组在 M N S 中?
if (j >= C[1])
Net[m++] = 1; // 在 M N S 中
}
```

3.2.6 元件折叠

在设计电路的过程中，工程师们会采取多种不同的设计风格。其中的两种为位－片设计（bit-slice design）和标准单元设计（standard-cell design）。在前一种方法中，电路首先被设计为一个元件栈（如图 15-10a 所示）。每个元件 C_i 宽为 w_i，高为 h_i，而元件宽度用片数来表示。图 15-10a 给出了一个四片的设计。线路是按片来连接各元件的，即连线可能连接元件 C_i 的第 j 片到元件 C_{i+1} 的第 j 片。如果某些元件的宽度不足 j 片，则这些元件之间不存在片 j 的连线。当图 1 5 -10a 的位－片设计作为某一大系统的一部分时，则在 V L SI (Very Large Scale Integrated) 芯片上为它分配一定数量的空间单元。分配是按空间宽度或高度的限制来完成的。现在的问题便是如何将元件栈折叠到分配空间中去，以便尽量减小未受限制的尺度（如，若高度限制为 H 时，必须折叠栈以尽量减小宽度 W）。由于其他尺度不变，因此缩小一个尺度（如 W）等价于缩小面积。可用折线方式来折叠元件栈，在每一折叠点，元件旋转 1 8 0° 。在图 15-10b 的例子中，一个 1 2 元件的栈折叠成四个垂直栈，折叠点为 C_6, C_9 和 C_{10}。折叠栈的宽度是宽度最大的元件所需的片数。在图 15-10b 中，栈宽各为 4，3，2 和 4。折叠栈的高度等于各栈所有元件高度之和的最大值。在图 15-10b 中栈 1 的元件高度之和最大，该栈的高度决定了包围所有栈的矩形高度。

实际上，在元件折叠问题中，还需考虑连接两个栈的线路所需的附加空间。如，在图 15-10b 中 C_5 和 C_6 间的线路因 C_6 为折叠点而弯曲。这些线路要求在 C_5 和 C_6 之下留有垂直空间，以便能从栈 1 连到栈 2。令 r_i 为 C_i 是折叠点时所需的高度。栈 1 所需的高度为 $\sum_{i=1}^{5} h_i + r_6$，栈 2 所需高度为 $\sum_{i=6}^{8} h_i + r_6 + r_9$。

在标准单元设计中，电路首先被设计成为具有相同高度的符合线性顺序的元件排列。假设此线性顺序中的元件为 C_1，.，C_n，下一步元件被折叠成如图 15-11 所示的相同宽度的行。在此图中， 12 个标准单元折叠成四个等宽行。折叠点是 C_4，C_6 和 C_{11}。在相邻标准单元行之间，使用布线通道来连接不同的行。折叠点决定了所需布线通道的高度。设 l_i 表示当 C_i 为折叠点时所需的通道高度。在图 15-11 的例子中，布线通道 1 的高度为 l_4，通道 2 的高度为 l_6，通道 3 的高度为 l_{11}。位－片栈折叠和标准单元折叠都会引出一系列的问题，这些问题可用动态规划方法来解决。

1. 等宽位－片元件折叠

定义 $r_1 = r_{n+1} = 0$。由元件 C_i 至 C_j 构成的栈的高度要求为 $\sum_{k=i}^{j} l_k + r_i + r_{j+1}$。设一个位－片设计中所有元件有相同宽度 W。首先考察在折叠矩形的高度 H 给定的情况下，如何缩小其宽度。设 W_i 为将元件 C_i 到 C_n 折叠到高为 H 的矩形时的最小宽度。若折叠不能实现（如当 $r_i + h_i > H$ 时），取 $W_i = \infty$。注意到 W_1 可能是所有 n 个元件的最佳折叠宽度。

当折叠 C_i 到 C_n 时，需要确定折叠点。现假定折叠点是按栈左到栈右的顺序来取定的。若第一点定为 C_{k+1}，则 C_i 到 C_k 在第一个栈中。为了得到最小宽度，从 C_{k+1} 到 C_n 的折叠必须用最优化方法，因此又将用到最优原理，可用动态规划方法来解决此问题。当第一个折叠点 $k+1$ 已知时，可得到以下公式:

$$W_i = w + W_{k+1} \quad (15\text{-}9)$$

由于不知道第一个折叠点，因此需要尝试所有可行的折叠点，并选择满足（ 15-9）式的折叠点。令 $hsum(i,k) = \sum_{j=i}^{k} h_j$。因 $k+1$ 是一个可行的折叠点，因此 $hsum(i,k) + r_i + r_{k+1}$ 一定不会超过 H。

根据上述分析，可得到以下动态规划递归式:

这里 $W_{n+1} = 0$，且在无最优折叠点 $k+1$ 时 W_i 为 ∞。利用递归式（15-10），可通过递归计算 $W_n, W_{n-1}., W_2, W_1$ 来计算 W_i。W_i 的计算需要至多检查 $n-i+1$ 个 W_{k+1}，耗时为 O($n-k$)。因此计算所有 W_i 的时间为 O(n^2)。通过保留式（15-10）每次所得的 k 值，可回溯地计算出各个最优的折叠点，其时间耗费为 O(n)。

现在来考察另外一个有关等宽元件的折叠问题：折叠后矩形的宽度 W 已知，需要尽量减小其高度。因每个折叠矩形宽为 w，因此折叠后栈的最大数量为 $s = W/w$。令 $H_{i,j}$ 为 C_i, ., C_n 折叠成一宽度为 jw 的矩形后的最小高度， $H_{1,s}$ 则是所有元件折叠后的最小高度。当 $j=1$ 时，不允许任何折叠，因此: $H_{i,1} = hsum(i,n) + r_i,\ 1 \leqslant i \leqslant n$

另外，当 $i=n$ 时，仅有一个元件，也不可能折叠，因此: $H_{n,j} = h_n + r_n,\ 1 \leqslant j \leqslant s$

在其他情况下，都可以进行元件折叠。如果第一个折叠点为 $k+1$，则第一个栈的高度为 $hsum(i,k) + r_i + r_{k+1}$。其他元件必须以至多 $(j-1)*w$ 的宽度折叠。为保证该折叠的最优性，其他元件也需以最小高度进行折叠.

因为第一个折叠点未知，因此必须尝试所有可能的折叠点，然后从中找出一个使式（15-11）的右侧取最小值的点，该点成为第一个折叠点。

可用迭代法来求解 $H_{i,j}$（$1 \leqslant i \leqslant n, 1 \leqslant j \leqslant s$），求解的顺序为: 先计算 $j=2$ 时的 $H_{i,j}$，再算 $j=3$，.，以此类推。对应每个 j 的 $H_{i,j}$ 的计算时间为 O(n^2)，所以计算所有 $H_{i,j}$ 的时间为 O($s\,n^2$)。通过保存由（ 15-12）式计算出的每个 k 值，可以采用复杂性为 O(n) 的回溯过程来确定各个最优的折叠点。

2. 变宽位－片元件的折叠

首先考察折叠矩形的高度 H 已定，欲求最小的折叠宽度的情况。令 W_i 如式（15-10）所示，按照与（15-10）式相同的推导过程，可得：

$W_i=\min\{width(i,k)+W_{k+1} \mid high(i,k)+r_i+r_{k+1}\leqslant H, i\leqslant k\leqslant n\}$ （15-13）

其中 $W_{n+1}=0$ 且 $width(i,k)=\min_{i\leqslant j\leqslant k}\{w_j\}$。可用与（15-10）式一样的方法求解（15-13）式，所需时间为 $O(n^2)$。

当折叠宽度 W 给定时，最小高度折叠可用折半搜索方法对超过 $O(n^2)$ 个可能值进行搜索来实现，可能的高度值为 $h(i,j)+r_i+r_{j+1}$。在检测每个高度时，也可用（15-13）式来确定该折叠的宽度是否小于等于 W。这种情况下总的时间消耗为 $O(n^2\log n)$。

3. 标准单元折叠

用 w_i 定义单元 C_i 的宽度。每个单元的高度为 h。当标准单元行的宽度 W 固定不变时，通过减少折叠高度，可以相应地减少折叠面积。考察 C_i 到 C_n 的最小高度折叠。设第一个折叠点是 C_{s+1}。从元件 C_{s+1} 到 C_n 的折叠必须使用最小高度，否则，可使用更小的高度来折叠 C_{s+1} 到 C_n，从而得到更小的折叠高度。所以这里仍可使用最优原理和动态规划方法。

令 $H_{i,s}$ 为 C_i 到 C_n 折叠成宽为 W 的矩形时的最小高度，其中第一个折叠点为 C_{s+1}。令 $wsum(i,s)=\sum_{j=i}^{s}w_j$。可假定没有宽度超过 W 的元件，否则不可能进行折叠。对于 $H_{n,n}$ 因为只有一个元件，不存在连线问题，因此 $H_{n,n}=h$。对于 $H_{i,s}$（$1\leqslant i<s\leqslant n$）注意到如果 $wsum(i,s)>W$，不可能实现折叠。若 $wsum(i,s)\leqslant W$，元件 C_i 和 C_{j+1} 在相同的标准单元行中，该行下方布线通道的高度为 l_{s+1}（定义 $l_{n+1}=0$）。因而：

$$H_{i,s}=H_{i+1,k} \quad (15\text{-}14)$$

当 $i=s<n$ 时，第一个标准单元行只包含 C_i。该行的高度为 h 且该行下方布线通道的高度为 l_{i+1}。因 C_{i+1} 到 C_n 单元的折叠是最优的.

为了寻找最小高度折叠，首先使用式（15-14）和（15-15）来确定 $H_{i,s}$（$1\leqslant i\leqslant s\leqslant n$）。最小高度折叠的高度为 $\min\{H_{1,s}\}$。可以使用回溯过程来确定最小高度折叠中的折叠点。

练习

1. 修改程序 15-1，使它同时计算出能导致最优装载的 x_i 值。
2. 修改程序 15-1，使用一个表格来确定 $f(i,y)$ 是否已被计算过。在求 $f(i,y)$ 时,若表中已经存在该值，则直接取用；若不存在该值，则采用一个递归调用来计算该值。
3. 定义 0/1/2 背包问题为：$\max\{\sum_{i=1}^{n}p_ix_i\}$。限制条件为：$\sum_{i=1}^{n}w_ix_i\leqslant c$ 且 $x_i\in\{0,1,2\}$，$1\leqslant i\leqslant n$。设 f 的定义同 0/1 背包问题中的定义。

1) 从 0/1/2 背包问题中推出类似于（15-1）和（15-2）的公式。

2) 假设 ws 为整数。编写一个类似于 15-2 的程序来计算二维数组 f，然后确定最优分配的 x 值。

3) 程序的复杂性是多少?

4. 二维 0/1 背包问题定义为：$\max\{\sum_{i=1}^{n}p_ix_i\}$。限制条件为：$\sum_{i=1}^{n}v_ix_i\leqslant c$，$\sum_{i=1}^{n}w_ix_i\leqslant d$ 且 $x_i\in\{0,1\}$，$1\leqslant i\leqslant n$。设 $f(i,y,z)$ 为二维背包问题最优解的值，其中物品为 i 到 n，$c=y$，$d=z$。

1) 推出类似于（15-1）和（15-2）式的关于 $f(n,y,z)$ 和 $f(i,y,z)$ 的公式。

2) 假设 vs 和 ws 为整数。编写一个类似于 15-2 的程序计算三维数组 f，然后确定最优分配的 x 值。

3) 程序的复杂性是多少?

*5. 编写一个实现元组方法的 C++代码，要求提供一个确定最优装载的 x_i 值的回溯函数。

6. 当取消段长限制时(即在程序 15-3 中 $L=\infty$)，程序 15-3 的时间复杂性按如下方式递归定义：$t(0)=c$（c 为常数）；当 $n>0$ 时 $t(n)=\sum_{j=0}^{n-1}t(j)+n$。

1) 根据 $t(n-1)=\sum_{j=0}^{n-2}t(j)+n-1$ 证明：当 $n>0$ 时，$t(n)=2t(n-1)+1$。

2) 证明 $t(n)=(2^n)$

7. 编写函数 Traceback (见程序 15-3) 的迭代版本。试说明两个版本各自的优缺点。
8. 编写变长图像压缩过程中 1) 和 2) 的实现代码。

9. 证明：$\sum_{s=0}^{q-1} s(q-s+1) = \Theta(q^3)$。

10. 在求解矩阵乘法递归式时仅用到数组 c 和 kay 的上三角。重写程序 15-6，定义 c 和 kay 为 UpperMatrix 类（见 4.3.4 节）的成员。

11. 改写程序 15-9，把它作为 LinkedWDigraph 的类成员，其渐进复杂性应与程序 15-9 相同。

12. 设 G 为有 n 个顶点的有向无环图，G 中各顶点的编号为 1 到 n，且当〈i,j〉为 G 中的一条边时有 $i<j$。设 $l(i,j)$ 为边〈i,j〉的长度：

1) 用动态规划方法计算图 G 中最长路径的长度，算法的时间耗费应为 $O(h+e)$，其中 e 为 G 中的边数。

2) 编写一个函数，利用 1) 中所得到的结果来构造最长路径，其复杂性应为 $O(p)$，其中 p 为该路径的顶点数。

13. 改写程序 15-9，首先从一个有向图的邻接矩阵开始，然后计算其反身传递闭包矩阵 RTC。若从顶点 i 到顶点 j 无通路，则 RTC[i][j]=1，否则 RTC[i][j]=0。要求代码的复杂性为 $\Theta(n^3)$，其中 n 为图中的顶点数。

14. 利用（15-10）式，编写一个复杂性为 $O(n^2)$ 的 C++迭代程序，寻找等宽元件栈的最优折叠点。

15. 用递归式（15-12）代替式（15-10）完成练习 14，时间复杂性要求为 $O(s\,n^2)$。

16. 用式（15-13）得出一个变宽元件栈的最小宽度折叠法，时间复杂性要求为 $O(n^2)$。

17. 利用 15.2.6 节的设计，给出一个寻找折叠矩形宽度为 W 的最小高度折叠算法，其复杂性应为 $O(n^2 \log n)$。位－片元件宽度不等。

18. 利用式（15-14）和（15-15）来确定一个含 n 个标准单元的最小高度折叠。算法的时间复杂性应为 $O(n^2)$。能否使用这两个公式得到一个时间复杂性为 (n) 的算法?

*19. 在 13.3.3 节可知，一个工程可分解为多个任务且这些任务可按拓扑顺序来完成。设任务从 1 到 n 编号，首先完成任务 1，然后完成任务 2，以此类推...。假设我们有两种方法来完成每个任务。$C_{i,1}$ 为使用第一种方法完成任务 i 时的代价，$C_{i,2}$ 为使用第 2 种方法完成任务 i 时的代价。令 $T_{i,1}$ 为第一种方法中任务 i 的时间耗费，$T_{i,2}$ 为第二种方法中任务 i 的时间耗费。并设各个 T 为整数。设计一个动态规划算法，以得到在时间 t 内完成所有任务的最小代价的方法。假定工程的代价为各任务的代价之和，工程所需的时间是各任务时间耗费之和。（提示：可设 $cost(i,j)$ 为在 j 时间内完成任务 i 到 n 的最小代价）。算法的复杂性是多少?

*20. 某一机器中有 n 个零件。每个零件有三个供应商，来自供应商 j 的零件 i 的重量为 $W_{i,j}$，其价格为 $C_{i,j}$（$1 \leqslant j \leqslant 3$）。机器的价格等于所有零件价格之和，其重量也为各零件重量之和。设计一个动态规划算法，以决定在总价格不超过 C 的条件下，从哪些供应商购买零件能组成最轻的机器。（提示：可设 $w(i,j)$ 为价格低于 j 时由零件 i 到 n 组成的最轻机器）。算法的复杂性是多少?

*21. 定义 $w(i,j)$ 为价格低于 j 时由零件 1 到 i 组成的最轻机器，完成练习 20。

*22. 串 s 为串 a 中去掉某些字符而得到的子串。如串“onion”为串“recognition”的子串。当且仅当串 s 既是 a 的子串又是 b 的子串时，串 s 是串 a 和串 b 的公共子串。串 s 的长度指其所含的字符数。试用动态规划算法得到串 a 和串 b 的最长公共子串。（提示：设 $a=a_1a_2...a_n$，$b=b_1b_2...b_m$。定义 $l(i,j)$ 为串 $a_i...a_n$ 和 $b_j...b_m$ 最长公共子串的长度）。算法的复杂性是多少?

*23. 若 $l(i,j)$ 定义为串 $a_1a_2...a_i$ 和 $b_1b_2...b_j$ 的最长公共子串的长度，重做练习 22。

*24. 在串编辑问题中，给出两个串 $a=a_1a_2...a_n$ 和 $b=b_1b_2...b_m$ 及三个耗费函数 C，D 和 I。其中 $C(i,j)$ 为将 a_i 改为 b_j 的耗费，$D(i)$ 为从 a 中删除 a_i 的耗费，$I(i)$ 为将 b_i 插入 a 中的耗费。通过修改、删除和插入操作可把串 a 改为串 b。如，可删除所有 a_i，然后插入所有 b_i；或者当 $n \geqslant m$ 时，可

先把 a_i 变成 b_i（$1 \leqslant i \leqslant n$），然后删除其余的 a_i。整个操作序列的耗费为各个操作的耗费之和。设计一个动态规划算法来确定一个具有最少耗费的编辑操作序列。(提示:定义 $c(i,j)$ 为将 $a_1a_2.a_i$ 转变为 $b_1b_2.b_j$ 的最少耗费）。算法的复杂性是多少?

（ 说明：本资料是根据《数据结构、算法与应用》(美，Sartaj Sahni 著）一书第 13-17 章编辑、改写的。考虑到因特网传输速度等因素，大部分插图和公式不得不被删除。对于内容不连贯之处，请网友或读者参阅该书，敬请原谅。 ）

第 4 章 回溯

寻找问题的解的一种可靠的方法是首先列出所有候选解，然后依次检查每一个，在检查完所有或部分候选解后，即可找到所需要的解。理论上，当候选解数量有限并且通过检查所有或部分候选解能够得到所需解时，上述方法是可行的。不过，在实际应用中，很少使用这种方法，因为候选解的数量通常都非常大(比如指数级，甚至是大数阶乘)，即便采用最快的计算机也只能解决规模很小的问题。对候选解进行系统检查的方法有多种，其中回溯和分枝定界法是比较常用的两种方法。按照这两种方法对候选解进行系统检查通常会使问题的求解时间大大减少(无论对于最坏情形还是对于一般情形)。事实上，这些方法可以使我们避免对很大的候选解集合进行检查，同时能够保证算法运行结束时可以找到所需要的解。因此，这些方法通常能够用来求解规模很大的问题。

本章集中阐述回溯方法，这种方法被用来设计货箱装船、背包、最大完备子图、旅行商和电路板排列问题的求解算法。

4.1 算法思想

回溯(backtracking)是一种系统地搜索问题解答的方法。为了实现回溯，首先需要为问题定义一个解空间(solution space)，这个空间必须至少包含问题的一个解(可能是最优的)。在迷宫老鼠问题中，我们可以定义一个包含从入口到出口的所有路径的解空间;在具有 n 个对象的 0/1 背包问题中(见 1.4 节和 2.2 节)，解空间的一个合理选择是 2^n 个长度为 n 的 0/1 向量的集合，这个集合表示了将 0 或 1 分配给 x 的所有可能方法。当 $n=3$ 时，解空间为{(0,0,0)，(0,1,0)，(0,0,1)，(1,0,0)，(0,1,1)，(1,0,1)，(1,1,0)，(1,1,1)}。

下一步是组织解空间以便它能被容易地搜索。典型的组织方法是图或树。图 16-1 用图的形式给出了一个 3×3 迷宫的解空间。从(1,1)点到(3,3)点的每一条路径都定义了 3×3 迷宫解空间中的一个元素，但由于障碍的设置，有些路径是不可行的。

图 16-2 用树形结构给出了含三个对象的 0/1 背包问题的解空间。从 i 层节点到 $i+1$ 层节点的一条边上的数字给出了向量 x 中第 i 个分量的值 x_i，从根节点到叶节点的每一条路径定义了解空间中的一个元素。从根节点 A 到叶节点 H 的路径定义了解 $x=[1,1,1]$。根据 w 和 c 的值，从根到叶的路径中的一些解或全部解可能是不可行的。

一旦定义了解空间的组织方法，这个空间即可按深度优先的方法从开始节点进行搜索。在迷宫老鼠问题中，开始节点为入口节点(1,1);在 0/1 背包问题中，开始节点为根节点 A。开始节点既是一个活节点又是一个 E-节点(expansion node)。从 E-节点可移动到一个新节点。如果能从当前的 E-节点移动到一个新节点，那么这个新节点将变成一个活节点和新的 E-节点，旧的 E-节点仍是一个活节点。如果不能移到一个新节点，当前的 E-节点就“死”了(即不再是一个

活节点），那么便只能返回到最近被考察的活节点（回溯），这个活节点变成了新的E-节点。当我们已经找到了答案或者回溯尽了所有的活节点时，搜索过程结束。

例 4-1 [迷宫老鼠] 考察图 16-3a 的矩阵中给出的 3×3 的“迷宫老鼠”问题。我们将利用图 16-1 给出的解空间图来搜索迷宫。

从迷宫的入口到出口的每一条路径都与图 16-1 中从(1,1)到(3,3)的一条路径相对应。然而，图 16-1 中有些从(1,1)到(3,3)的路径却不是迷宫中从入口到出口的路径。搜索从点(1,1)开始，该点是目前唯一的活节点，它也是一个E-节点。为避免再次走过这个位置，置 $maze(1,1)$ 为 1。从这个位置，能移动到(1,2)或(2,1)两个位置。对于本例，两种移动都是可行的，因为在每一个位置都有一个值 0。假定选择移动到(1,2)，$maze(1,2)$ 被置为 1 以避免再次经过该点。迷宫当前状态如图 16-3b 所示。这时有两个活节点(1,1) (1,2)。(1,2)成为E-节点。

在图 16-1 中从当前E-节点开始有 3 个可能的移动，其中两个是不可行的，因为迷宫在这些位置上的值为 1。唯一可行的移动是(1,3)。移动到这个位置，并置 $maze(1,3)$ 为 1 以避免再次经过该点，此时迷宫状态为 16-3c。图 16-1 中，从(1,3)出发有两个可能的移动，但没有一个是可行的。所以E-节点(1,3)死亡，回溯到最近被检查的活节点(1,2)。在这个位置也没有可行的移动，故这个节点也死亡了。唯一留下的活节点是(1,1)。这个节点再次变为E-节点，它可移动到(2,1)。现在活节点为(1,1)，(2,1)。继续下去，能到达点(3,3)。此时，活节点表为(1,1)，(2,1)，(3,1)，(3,2)，(3,3)，这即是到达出口的路径。

程序 5-13 是一个在迷宫中寻找路径的回溯算法。

例 4-2 [0/1 背包问题] 考察如下背包问题：$n=3$，$w=[20,15,15]$，$p=[40,25,25]$且 $c=30$。从根节点开始搜索图 16-2 中的树。根节点是当前唯一的活节点，也是E-节点，从这里能够移动到B或C点。假设移动到B，则活节点为A和B。B是当前E-节点。在节点B，剩下的容量 r 为 10，而收益 cp 为 40。从B点，能移动到D或E。移到D是不可行的，因为移到D所需的容量 w_2 为 15。到E的移动是可行的，因为在这个移动中没有占用任何容量。E变成新的E-节点。这时活节点为A,B,E。在节点E，$r=10$，$cp=40$。从E，有两种可能移动（到J和K），到J的移动是不可行的，而到K的移动是可行的。节点K变成了新的E-节点。因为K是一个叶子，所以得到一个可行的解。这个解的收益为 $cp=40$。x 的值由从根到K的路径来决定。这个路径（A,B,E,K）也是此时的活节点序列。既然不能进一步扩充K，K节点死亡，回溯到E，而E也不能进一步扩充，它也死亡了。接着，回溯到B，它也死亡了，A再次变为E-节点。它可被进一步扩充，到达节点C。此时 $r=30$，$cp=0$。从C点能够移动到F或G。假定移动到F。F变为新的E-节点并且活节点为A, C,F。在F，$r=15$，$cp=25$。从F点，能移动到L或M。假定移动到L。此时 $r=0$，$cp=50$。既然L是一个叶节点，它表示了一个比目前找到的最优解（即节点K）更好的可行解，我们把这个解作为最优解。节点L死亡，回溯到节点F。继续下去，搜索整棵树。在搜索期间发现的最优解即为最后的解。

例 4-3 [旅行商问题] 在这个问题中，给出一个 n 顶点网络（有向或无向），要求找出一个包含所有 n 个顶点的具有最小耗费的环路。任何一个包含网络中所有 n 个顶点的环路被称作一个旅行（tour）。在旅行商问题中，要设法找到一条最小耗费的旅行。

图 16-4 给出了一个四顶点网络。在这个网络中，一些旅行如下：1,2,4,3,1; 1,3,2,4,1和1,4,3,2,1。旅行2,4,3,1,2; 4,3,1,2,4和3,1,2,4,3和旅行1,2,4,3,1一样。而旅行1,3,4,2,1是旅行1,2,4,3,1的“逆”。旅行1,2,4,3,1的耗费为66; 而1,3,2,4,1的耗费为25; 1,4,3,2,1为59。故1,3,2,4,1是该网络中最小耗费的旅行。

顾名思义，旅行商问题可被用来模拟现实生活中旅行商所要旅行的地区问题。顶点表示旅行商所要旅行的城市（包括起点）。边的耗费给出了在两个城市旅行所需的时间（或花费）。旅行表示当旅行商游览了所有城市再回到出发点时所走的路线。

旅行商问题还可用来模拟其他问题。假定要在一个金属薄片或印刷电路板上钻许多孔。孔的

位置已知。这些孔由一个机器钻头来钻，它从起始位置开始，移动到每一个钻孔位置钻孔，然后回到起始位置。总共花的时间是钻所有孔的时间与钻头移动的时间。钻所有孔所需的时间独立于钻孔顺序。然而，钻头移动时间是钻头移动距离的函数。因此，希望找到最短的移动路径。

另有一个例子，考察一个批量生产的环境，其中有一个特殊的机器可用来生产 n 个不同的产品。利用一个生产循环不断地生产这些产品。在一个循环中，所有 n 个产品被顺序生产出来，然后再开始下一个循环。在下一个循环中，又采用了同样的生产顺序。例如，如果这台机器被用来顺序为小汽车喷红、白、蓝漆，那么在为蓝色小汽车喷漆之后，我们又开始了新一轮循环，为红色小汽车喷漆，然后是白色小汽车、蓝色小汽车、红色小汽车，..，如此下去。一个循环的花费包括生产一个循环中的产品所需的花费以及循环中从一个产品转变到另一个产品的花费。虽然生产产品的花费独立于产品生产顺序，但循环中从生产一个产品转变到生产另一个产品的花费却与顺序有关。为了使耗费最小化，可以定义一个有向图，图中的顶点表示产品，边 $<(i, j)>$ 上的耗费值为生产过程中从产品 i 转变到产品 j 所需的耗费。一个最小耗费的旅行定义了一个最小耗费的生产循环。

既然旅行是包含所有顶点的一个循环，故可以把任意一个点作为起点（因此也是终点）。针对图 16-4，任意选取点 1 作为起点和终点，则每一个旅行可用顶点序列 $1, v_2, ., v_n, 1$ 来描述，$v_2, ., v_n$ 是 $(2, 3, ., n)$ 的一个排列。可能的旅行可用一个树来描述，其中每一个从根到叶的路径定义了一个旅行。图 16-5 给出了一棵表示四顶点网络的树。从根到叶的路径中各边的标号定义了一个旅行（还要附加 1 作为终点）。例如，到节点 L 的路径表示了旅行 1, 2, 3, 4, 1，而到节点 O 的路径表示了旅行 1, 3, 4, 2, 1。网络中的每一个旅行都由树中的一条从根到叶的确定路径来表示。因此，树中叶的数目为 $(n-1)!$ 。

回溯算法将用深度优先方式从根节点开始，通过搜索解空间树发现一个最小耗费的旅行。对图 16-4 的网络，利用图 16-5 的解空间树，一个可能的搜索为 A B C F L。在 L 点，旅行 1, 2, 3, 4, 1 作为当前最好的旅行被记录下来。它的耗费是 59。从 L 点回溯到活节点 F。由于 F 没有未被检查的孩子，所以它成为死节点，回溯到 C 点。C 变为 E-节点，向前移动到 G，然后是 M。这样构造出了旅行 1, 2, 4, 3, 1，它的耗费是 66。既然它不比当前的最佳旅行好，抛弃它并回溯到 G，然后是 C, B。从 B 点，搜索向前移动到 D，然后是 H, N。这个旅行 1, 3, 2, 4, 1 的耗费是 25，比当前的最佳旅行好，把它作为当前的最好旅行。从 N 点，搜索回溯到 H，然后是 D。在 D 点，再次向前移动，到达 O 点。如此继续下去，可搜索完整个树，得出 1, 3, 2, 4, 1 是最少耗费的旅行。

当要求解的问题需要根据 n 个元素的一个子集来优化某些函数时，解空间树被称作子集树（subset tree）。所以对有 n 个对象的 0/1 背包问题来说，它的解空间树就是一个子集树。这样一棵树有 2^n 个叶节点，全部节点有 $2^{n+1}-1$ 个。因此，每一个对树中所有节点进行遍历的算法都必须耗时 $\Omega(2^n)$。当要求解的问题需要根据一个 n 元素的排列来优化某些函数时，解空间树被称作排列树（permutation tree）。这样的树有 $n!$ 个叶节点，所以每一个遍历树中所有节点的算法都必须耗时 $\Omega(n!)$。图 16-5 中的树是顶点 {2, 3, 4} 的最佳排列的解空间树，顶点 1 是旅行的起点和终点。

通过确定一个新近到达的节点能否导致一个比当前最优解还要好的解，可加速对最优解的搜索。如果不能，则移动到该节点的任何一个子树都是无意义的，这个节点可被立即杀死，用来杀死活节点的策略称为限界函数（ bounding function）。在例 16-2 中，可使用如下限界函数：杀死代表不可行解决方案的节点；对于旅行商问题，可使用如下限界函数：如果目前建立的部分旅行的耗费不少于当前最佳路径的耗费，则杀死当前节点。如果在图 16-4 的例子中使用该限界函数，那么当到达节点 I 时，已经找到了具有耗费 25 的 1, 3, 2, 4, 1 的旅行。在节点 I，部分旅行 1, 3, 4 的耗费为 26，若旅行通过该节点，那么不能找到一个耗费小于 25 的旅行，故搜索以 I 为根节点的子树毫无意义。

小结

回溯方法的步骤如下:

1) 定义一个解空间，它包含问题的解。

2) 用适于搜索的方式组织该空间。

3) 用深度优先法搜索该空间，利用限界函数避免移动到不可能产生解的子空间。

回溯算法的一个有趣的特性是在搜索执行的同时产生解空间。在搜索期间的任何时刻，仅保留从开始节点到当前 E-节点的路径。因此，回溯算法的空间需求为 O（从开始节点起最长路径的长度）。这个特性非常重要，因为解空间的大小通常是最长路径长度的指数或阶乘。所以如果要存储全部解空间的话，再多的空间也不够用。

练习

1. 考察如下 0/1 背包问题: $n=4$，$w=[20,25,15,35]$，$p=[40,49,25,60]$，$c=62$。

1) 画出该 0/1 背包问题的解空间树。

2) 对该树运用回溯算法（利用给出的 ps, ws, c 值），依回溯算法遍历节点的顺序标记节点。确定回溯算法未遍历的节点。

2. 1) 当 $n=5$ 时，画出旅行商问题的解空间树。

2) 在该树上，运用回溯算法（使用图 16-6 的例子）。依回溯算法遍历节点的顺序标记节点。确定未被遍历的节点。

3. 每周六， Mary 和 Joe 都在一起打乒乓球。她们每人都有一个装有 120 个球的篮子。这样一直打下去，直到两个篮子为空。然后她们需要从球桌周围拾起 240 个球，放入各自的篮子。Mary 只拾她这边的球，而 Joe 拾剩下的球。描述如何用旅行商问题帮助 Mary 和 Joe 决定她们拾球的顺序以便她们能走最少的路径。

4.2 应用

4.2.1 货箱装船

1. 问题

在 1.3 节中，考察了用最大数量的货箱装船的问题。现在对该问题做一些改动。在新问题中，有两艘船， n 个货箱。第一艘船的载重量是 c_1，第二艘船的载重量是 c_2，w_i 是货箱 i 的重量且 $\sum_{i=1}^{n} w_i \leqslant c_1+c_2$。我们希望确定是否有一种可将所有 n 个货箱全部装船的方法。若有的话，找出该方法。

例 4-4 当 $n=3$，$c_1=c_2=50$，$w=[10,40,40]$ 时，可将货箱 1，2 装到第一艘船上，货箱 3 装到第二艘船上。如果 $w=[20,40,40]$，则无法将货箱全部装船。当 $\sum_{i=1}^{n} w_i = c_1+c_2$ 时，两艘船的装载问题等价于子集之和（ sum-of-subset）问题，即有 n 个数字，要求找到一个子集（如果存在的话）使它的和为 c_1。当 $c_1=c_2$ 且 $\sum_{i=1}^{n} w_i = 2c_1$ 时，两艘船的装载问题等价于分割问题（ partition problem），即有 n 个数字 $a_i, (1 \leqslant i \leqslant n)$，要求找到一个子集（若存在的话），使得子集之和为 $\left(\sum_{i=1}^{n} a_i \right) / 2$。分割问题和子集之和问题都是 NP-复杂问题。而且即使问题被限制为整型数字，它们仍是 NP-复杂问题。所以不能期望在多项式时间内解决两艘船的装载问题。当存在一种方法能够装载所有 n 个货箱时，可以验证以下的装船策略可以获得成功: 1) 尽可能地将第一艘船装至它的重量极限; 2) 将剩余货箱装到第二艘船。为了尽可能地将第一艘船装满，需要选择一个货箱的子集，它们的总重量尽可能接近 c_1。这个选择可通过求解 0/1 背包问题来实现，即寻找 $\max(\sum_{i=1}^{n} w_i x_i)$，其中 $\sum_{i=1}^{n} w_i x_i \leqslant c_1$，$x_i \in \{0,1\}$，$1 \leqslant i \leqslant n$。当重量是整数时，可用 15.2 节的动态规划方法确定第一艘船的最佳装载。用元组方法所需时间为 $O(\min\{c_1, 2^n\})$。可以使用回溯方法设计一个复杂性为 $O(2^n)$ 的算法，在有些实例中，该方法比动态规划

算法要好。

2. 第一种回溯算法

既然想要找到一个重量的子集，使子集之和尽量接近 c_1，那么可以使用一个子集空间，并将其组织成如图 16-2 那样的二叉树。可用深度优先的方法搜索该解空间以求得最优解。使用限界函数去阻止不可能获得解答的节点的扩张。如果 Z 是树的 $j+1$ 层的一个节点，那么从根到 O 的路径定义了 x_i（$1 \leqslant i \leqslant j$）的值。使用这些值，定义 cw（当前重量）为 $\sum_{i=1}^{n} w_i x_i$。若 $cw > c_1$，则以 O 为根的子树不能产生一个可行的解答。可将这个测试作为限界函数。当且仅当一个节点的 cw 值大于 c_1 时，定义它是不可行的。

例 4-5 假定 $n=4$，$w=[8,6,2,3]$，$c_1=12$。解空间树为图 16-2 的树再加上一层节点。搜索从根 A 开始且 $cw=0$。若移动到左孩子 B 则 $cw=8$，$cw \leqslant c_1=12$。以 B 为根的子树包含一个可行的节点，故移动到节点 B。从节点 B 不能移动到节点 D，因为 $cw+w_2>c_1$。移动到节点 E，这个移动未改变 cw。下一步为节点 J，$cw=10$。J 的左孩子的 cw 值为 13，超出了 c_1，故搜索不能移动到 J 的左孩子。

可移动到 J 的右孩子，它是一个叶节点。至此，已找到了一个子集，它的 $cw=10$。x_i 的值由从 A 到 J 的右孩子的路径获得，其值为[1,0,1,0]。

回溯算法接着回溯到 J，然后是 E。从 E，再次沿着树向下移动到节点 K，此时 $cw=8$。移动到它的左子树，有 $cw=11$。既然已到达了一个叶节点，就看是否 cw 的值大于当前的最优 cw 值。结果确实大于最优值，所以这个叶节点表示了一个比[1,0,1,0]更好的解决方案。到该节点的路径决定了 x 的值[1,0,0,1]。从该叶节点，回溯到节点 K，现在移动到 K 的右孩子，一个具有 $cw=8$ 的叶节点。这个叶节点中没有比当前最优 cw 值还好的 cw 值，所以回溯到 K,E,B 直到 A。从根节点开始，沿树继续向下移动。算法将移动到 C 并搜索它的子树。

当使用前述的限界函数时，便产生了程序 16-1 所示的回溯算法。函数 MaxLoading 返回 ⩽c 的最大子集之和，但它不能找到产生该和的子集。后面将改进代码以便找到这个子集。MaxLoading 调用了一个递归函数 maxLoading，它是类 Loading 的一个成员，定义 Loading 类是为了减少 MaxLoading 中的参数个数。maxLoading(1) 实际执行解空间的搜索。MaxLoading(i) 搜索以 i 层节点（该节点已被隐式确定）为根的子树。从根到该节点的路径定义的子解答有一个重量值 cw，目前最优解答的重量为 bestw，这些变量以及与类 Loading 的一个成员相关联的其他变量，均由 MaxLoading 初始化。

程序 16-1 第一种回溯算法

```
template<class T>
class Loading {
friend MaxLoading(T [], T, int);
private:
void maxLoading(int i);
int n; // 货箱数目
T *w, // 货箱重量数组
c, // 第一艘船的容量
cw, // 当前装载的重量
bestw; // 目前最优装载的重量
};
template<class T>
void Loading<T>::maxLoading(int i)
{// 从第 i 层节点搜索
if (i > n) {// 位于叶节点
```

```
if (cw > bestw) bestw = cw;
return ; }
// 检查子树
if (cw + w[i] <= c) {// 尝试 x[i] = 1
cw += w[i];
maxLoading ( i + 1 ) ;
cw -= w[i];}
maxLoading(i+1);// 尝试 x[i] = 0
}
template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载的重量
Loading<T> X;
// 初始化 X
X.w = w;
X.c = c;
X.n = n;
X.bestw = 0;
X.cw = 0;
// 计算最优装载的重量
X . maxLoading ( 1 ) ;
return X.bestw;
}
```

如果 i＞n，则到达了叶节点。被叶节点定义的解答有重量 cw，它一定≤c，因为搜索不会移动到不可行的节点。若 cw＞bestw，则目前最优解答的值被更新。当 i≤n 时，我们处在有两个孩子的节点 Z 上。左孩子表示 x[i]＝1 的情况，只有 cw＋w[i]≤c 时，才能移到这里。当移动到左孩子时， cw 被置为 cw＋w[i]，且到达一个 i＋1 层的节点。以该节点为根的子树被递归搜索。当搜索完成时，回到节点 Z。为了得到 Z 的 cw 值，需用当前的 cw 值减去 w[i]，Z 的右子树还未搜索。既然这个子树表示 x[i]＝0 的情况，所以无需进行可行性检查就可移动到该子树，因为一个可行节点的右孩子总是可行的。

注意：解空间树未被 maxLoading 显示构造。函数 maxLoading 在它到达的每一个节点上花费(1)时间。到达的节点数量为 O(2^n)，所以复杂性为 O(2^n)。这个函数使用的递归栈空间为(n)。

3. 第二种回溯方法

通过不移动到不可能包含比当前最优解还要好的解的右子树，能提高函数 maxLoading 的性能。令 bestw 为目前最优解的重量， Z 为解空间树的第 i 层的一个节点， cw 的定义如前。以 Z 为根的子树中没有叶节点的重量会超过 cw＋r，其中 $r=\sum_{j=i+1}^{n} w[j]$ 为剩余货箱的重量。因此，当 cw＋r≤bestw 时，没有必要去搜索 Z 的右子树。

例 4-6 令 n, w, c 的值与例 4-5 中相同。用新的限界函数，搜索将像原来那样向前进行直至到达第一个叶节点 J（它是 J 的右孩子）。bestw 被置为 10。回溯到 E，然后向下移动到 K 的左孩子，此时 bestw 被更新为 11。我们没有移动到 K 的右孩子，因为在右孩子节点 cw＝8，r＝0，cw＋r≤bestw。回溯到节点 A。同样，不必移动到右孩子 C，因为在 C 点 cw＝0，r＝11 且 cw＋r≤bestw。加强了条件的限界函数避免了对 A 的右子树及 K 的右子树的搜索。

当使用加强了条件的限界函数时，可得到程序 16-2 的代码。这个代码将类型为 T 的私有

变量r 加到了类Loading的定义中。新的代码不必检查是否一个到达的叶节点有比当前最优解还优的重量值。这样的检查是不必要的，因为加强的限界函数不允许移动到不能产生较好解的节点。因此，每到达一个新的叶节点就意味着找到了比当前最优解还优的解。虽然新代码的复杂性仍是O(2ⁿ)，但它可比程序16-1少搜索一些节点。

程序16-2 程序16-1的优化

```
template<class T>
void Loading<T>::maxLoading(int i)
{// // 从第i 层节点搜索
if (i > n) {// 在叶节点上
bestw = cw;
return ; }
// 检查子树
r -= w[i];
if (cw + w[i] <= c) {//尝试 x[i] = 1
cw += w[i];
maxLoading(i + 1);
cw -= w[i];}
if (cw + r > bestw) //尝试 x[i] = 0
maxLoading(i + 1);
r += w[i];
}
template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载的重量
Loading<T> X;
// 初始化X
X.w = w;
X.c = c;
X.n = n;
X.bestw = 0;
X.cw = 0;
// r 的初始值为所有重量之和
X.r = 0;
for (int i = 1; i <= n; i++)
X.r += w[i];
// 计算最优装载的重量
X.maxLoading(1);
return X.bestw;
}
```

4. 寻找最优子集

为了确定具有最接近c 的重量的货箱子集，有必要增加一些代码来记录当前找到的最优子集。为了记录这个子集，将参数bestx 添加到Maxloading 中。bestx 是一个整数数组，其中元素可为0或1，当且仅当bestx[i]=1时，货箱i 在最优子集中。新的代码见程序16-3。

程序16-3 给出最优装载的代码

```
template<class T>
void Loading<T>::maxLoading(int i)
{ / /从第 i 层节点搜索
if (i > n) {// 在叶节点上
for (int j = 1; j <= n; j++)
bestx[j] = x[j];
bestw = cw; return;}
// 检查子树
r -= w[i];
if (cw + w[i] <= c) {//尝试 x[i] = 1
x[i] = 1;
cw += w[i];
m a x L o a d i n g ( i + 1 ) ;
cw -= w[i];}
if (cw + r > bestw) {//尝试 x[i] = 0
x[i] = 0;
m a x L o a d i n g ( i + 1 ) ; }
r += w[i];
}
template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{// 返回最优装载及其值
Loading<T> X;
// 初始化 X
X.x = new int [n+1];
X.w = w;
X.c = c;
X.n = n;
X.bestx = bestx;
X.bestw = 0;
X.cw = 0;
// r 的初始值为所有重量之和
X.r = 0;
for (int i = 1; i <= n; i++)
X.r += w[i];
X . m a x L o a d i n g ( 1 ) ;
delete [] X.x;
return X.bestw;
}
```

这段代码在 L o a d i n g 中增加了两个私有数据成员： x 和 b e s t x。这两个私有数据成员都是整型的一维数组。数组 x 用来记录从搜索树的根到当前节点的路径（即它保留了路径上的 x_i 值），b e s t x 记录当前最优解。无论何时到达了一个具有较优解的叶节点， bestx 被更新以表示从根到叶的路径。为 1 的 x_i 值确定了要被装载的货箱。数据 x 的空间由 MaxLoading 分配。

因为 bestx 可以被更新 O (2_n)次，故 maxLoading 的复杂性为 O ($n2_n$)。使用下列方法之一，

复杂性可降为 O (2$_n$):

1) 首先运行程序 1 6 - 2 的代码，以决定最优装载重量，令其为 W。然后运行程序 1 6 - 3 的一个修改版本。该版本以 b e s t w = W 开始运行，当 c w + r ≥ b e s t w 时搜索右子树，第一次到达一个叶节点时便终止（即 i > n）。

2) 修改程序 1 6 - 3 的代码以不断保留从根到当前最优叶的路径。尤其当位于 i 层节点时，则到最优叶的路径由 x [j] (1 ≤ *j*<*i*) 和 b e s t x [j] (j ≤ i ≤ n) 给出。按照这种方法，算法每次回溯一级，并在 b e s t x 中存储一个 x [i]。由于算法回溯所需时间为 O (2$_n$)，因此额外开销为 O (2$_n$)。

5. 一个改进的迭代版本

可改进程序 1 6 - 3 的代码以减少它的空间需求。因为数组 x 中记录可在树中移动的所有路径，故可以消除大小为(*n*)的递归栈空间。如例 4 - 5 所示，从解空间树的任何节点，算法不断向左孩子移动，直到不能再移动为止。如果一个叶子已被到达，则最优解被更新。否则，它试图移动到右孩子。当要么到达一个叶节点，要么不值得移动到一个右孩子时，算法回溯到一个节点，条件是从该节点向其右孩子移动有可能找到最优解。这个节点有一个特性，即它是路径中具有 x [i] = 1 的节点中离根节点最近的节点。如果向右孩子的移动是有效的，那么就这么做，然后再完成一系列向左孩子的移动。如果向右孩子的移动是无效的，则回溯到 x [i] = 1 的下一个节点。

该算法遍历树的方式可被编码成与程序 1 6 - 4 一样的迭代（即循环）算法。不像递归代码，这种代码在检查是否该向右孩子移动之前就移动到了右孩子。如果这个移动是不可行的，则回溯。迭代代码的时间复杂性与程序 1 6 - 3 一样。

程序 16-4 迭代代码

```
template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{// 返回最佳装载及其值
// 迭代回溯程序
// 初始化根节点
int i = 1; // 当前节点的层次
// x[1:i-1] 是到达当前节点的路径
int *x = new int [n+1];
T bestw = 0, // 迄今最优装载的重量
cw = 0, // 当前装载的重量
r = 0; // 剩余货箱重量的和
for (int j = 1; j <= n; j++)
r += w[j];
// 在树中搜索
while (true) {
// 下移，尽可能向左
while (i <= n && cw + w[i] <= c) {
// 移向左孩子
r -= w[i];
cw += w[i];
x[i] = 1;
i + + ;
}
if (i > n) {// 到达叶子
for (int j = 1; j <= n; j++)
```

```
bestx[j] = x[j];
bestw = cw;}
else {// 移向右孩子
r -= w[i];
x[i] = 0;
i + + ; }
// 必要时返回
while (cw + r <= bestw) {
// 本子树没有更好的叶子，返回
i - - ;
while (i > 0 && !x[i]) {
// 从一个右孩子返回
r += w[i];
i - - ;
}
if (i == 0) {delete [] x;
return bestw;}
// 移向右子树
x[i] = 0;
cw -= w[i];
i + + ;
}
}
}
```

4.2.2 0/1 背包问题

0/1 背包问题是一个 NP-复杂问题，为了解决该问题，在 1.4 节采用了贪婪算法，在 3.2 节又采用了动态规划算法。在本节，将用回溯算法解决该问题。既然想选择一个对象的子集，将它们装入背包，以便获得的收益最大，则解空间应组织成子集树的形状（如图 16-2 所示）。该回溯算法与 4.2 节的装载问题很类似。首先形成一个递归算法，去找到可获得的最大收益。然后，对该算法加以改进，形成代码。改进后的代码可找到获得最大收益时包含在背包中的对象的集合。

与程序 16-2 一样，左孩子表示一个可行的节点，无论何时都要移动到它；当右子树可能含有比当前最优解还优的解时，移动到它。一种决定是否要移动到右子树的简单方法是令 r 为还未遍历的对象的收益之和，将 r 加到 cp（当前节点所获收益）之上，若（$r+cp$）$\leqslant bestp$（目前最优解的收益），则不需搜索右子树。一种更有效的方法是按收益密度 p_i/w_i 对剩余对象排序，将对象按密度递减的顺序去填充背包的剩余容量，当遇到第一个不能全部放入背包的对象时，就使用它的一部分。

例 4-7 考察一个背包例子：$n=4$，$c=7$，$p=[9,10,7,4]$，$w=[3,5,2,1]$。这些对象的收益密度为[3,2,3.5,4]。当背包以密度递减的顺序被填充时，对象 4 首先被填充，然后是对象 3、对象 1。在这三个对象被装入背包之后，剩余容量为 1。这个容量可容纳对象 2 的 0.2 倍的重量。将 0.2 倍的该对象装入，产生了收益值 2。被构造的解为 $x=[1,0.2,1,1]$，相应的收益值为 22。尽管该解不可行（x_2 是 0.2，而实际上它应为 0 或 1），但它的收益值 22 一定不少于要求的最优解。因此，该 0/1 背包问题没有收益值多于 22 的解。

解空间树为图 16-2 再加上一层节点。当位于解空间树的节点 B 时，$x_1=1$，目前获益为 $cp=$

9。该节点所用容量为 *cw*= 3。要获得最好的附加收益，要以密度递减的顺序填充剩余容量 *cleft*=*ccw*= 4。也就是说，先放对象 4，然后是对象 3，然后是对象 2 的 0.2 倍的重量。因此，子树 *A* 的最优解的收益值至多为 22。当位于节点 *C* 时，*cp*=*cw*= 0，*cleft*= 7。按密度递减顺序填充剩余容量，则对象 4 和 3 被装入。然后是对象 2 的 0.8 倍被装入。这样产生出收益值 19。在子树 *C* 中没有节点可产生出比 19 还大的收益值。

在节点 E，*cp*= 9，*cw*= 3，*cleft*= 4。仅剩对象 3 和 4 要被考虑。当对象按密度递减的顺序被考虑时，对象 4 先被装入，然后是对象 3。所以在子树 *E* 中无节点有多于 *cp*+ 4 + 7 = 20 的收益值。如果已经找到了一个具有收益值 20 或更多的解，则无必要去搜索 *E* 子树。

一种实现限界函数的好方法是首先将对象按密度排序。假定已经做了这样的排序。定义类 Knap（见程序 16-5）来减少限界函数 Bound（见程序 16-6）及递归函数 Knapsack（见程序 16-7）的参数数量，该递归函数用于计算最优解的收益值。参数的减少又可引起递归栈空间的减少以及每一个 Knapsack 的执行时间的减少。注意函数 Knapsack 和函数 maxLoading（见程序 16-2）的相似性。同时注意仅当向右孩子移动时，限界函数才被计算。当向左孩子移动时，左孩子的限界函数的值与其父节点相同。

程序 16-5 Knap 类

```
template<class Tw, class Tp>
class Knap {
friend Tp Knapsack(Tp *, Tw *, Tw, int);
private:
Tp Bound(int i);
void Knapsack(int i);
Tw c; //背包容量
int n; // 对象数目
Tw *w; // 对象重量的数组
Tp *p; // 对象收益的数组
Tw cw; // 当前背包的重量
Tp cp; // 当前背包的收益
Tp bestp; // 迄今最大的收益
};
```

程序 16-6 限界函数

```
template<class Tw, class Tp>
Tp Knap<Tw, Tp>::Bound(int i)
{// 返回子树中最优叶子的上限值 Return upper bound on value of
// best leaf in subtree.
Tw cleft = c - cw; // 剩余容量
Tp b = cp; // 收益的界限
// 按照收益密度的次序装填剩余容量
while (i <= n && w[i] <= cleft) {
cleft -= w[i];
b += p[i];
i++;
}
// 取下一个对象的一部分
if (i <= n) b += p[i]/w[i] * cleft;
```

```
return b;
}
```

程序 16-7 0/1 背包问题的迭代函数

```
template<class Tw, class Tp>
void Knap<Tw, Tp>::Knapsack(int i)
{// 从第 i 层节点搜索
if (i > n) {// 在叶节点上
bestp = cp;
return ; }
// 检查子树
if (cw + w[i] <= c) {//尝试 x[i] = 1
cw += w[i];
cp += p[i];
Knapsack(i + 1);
cw -= w[i];
cp -= p[i];}
if (Bound(i+1) > bestp) // 尝试 x[i] = 0
Knapsack(i + 1);
}
```

在执行程序 16-7 的函数 Knapsack 之前，需要按密度对对象排序，也要确保对象的重量总和超出背包的容量。为了完成排序，定义了类 Object（见程序 16-8）。注意定义操作符<=是为了使归并排序程序（见程序 14-3）能按密度递减的顺序排序。

程序 16-8 Object 类

```
class Object {
friend int Knapsack(int *, int *, int, int);
public:
int operator<=(Object a) const
{return (d >= a.d);}
private:
int ID; // 对象号
float d; // 收益密度
};
```

程序 16-9 首先验证重量之和超出背包容量，然后排序对象，在执行 Knap::Knapsack 之前完成一些必要的初始化。Knap::Knapsack 的复杂性是 $O(n2^n)$，因为限界函数的复杂性为 $O(n)$，且该函数在 $O(2^n)$个右孩子处被计算。

程序 16-9 程序 16-7 的预处理代码

```
template<class Tw, class Tp>
Tp Knapsack(Tp p[], Tw w[], Tw c, int n)
{// 返回最优装包的值
// 初始化
Tw W = 0; // 记录重量之和
Tp P = 0; // 记录收益之和
// 定义一个按收益密度排序的对象数组
Object *Q = new Object [n];
```

```
for (int i = 1; i <= n; i++) {
// 收益密度的数组
Q[i-1].ID = i;
Q[i-1].d = 1.0*p[i]/w[i];
P += p[i];
W += w[i];
}
if (W <= c) return P; // 可容纳所有对象
MergeSort(Q,n); // 按密度排序
// 创建 K n a p 的成员
K n a p < Tw, Tp> K;
K.p = new Tp [n+1];
K.w = new Tw [n+1];
for (i = 1; i <= n; i++) {
K.p[i] = p[Q[i-1].ID];
K.w[i] = w[Q[i-1].ID];
}
K.cp = 0;
K.cw = 0;
K.c = c;
K.n = n;
K.bestp = 0;
// 寻找最优收益
K . K n a p s a c k ( 1 ) ;
delete [] Q;
delete [] K.w;
delete [] K.p;
return K.bestp;
}
```

4.2.3 最大完备子图

令 U 为无向图 G 的顶点的子集，当且仅当对于 U 中的任意点 u 和 v，(u, v)是图 G 的一条边时，U 定义了一个完全子图（complete subgraph）。子图的尺寸为图中顶点的数量。当且仅当一个完全子图不被包含在 G 的一个更大的完全子图中时，它是图 G 的一个完备子图。最大的完备子图是具有最大尺寸的完备子图。

例 4-8 在图 16-7a 中，子集{1, 2}定义了一个尺寸为 2 的完全子图。这个子图不是一个完备子图，因为它被包含在一个更大的完全子图{1, 2, 5}中。{1, 2, 5}定义了该图的一个最大的完备子图。点集{1, 4, 5}和{2, 3, 5}定义了其他的最大的完备子图。

当且仅当对于 U 中任意点 u 和 v，(u, v) 不是 G 的一条边时，U 定义了一个空子图。当且仅当一个子集不被包含在一个更大的点集中时，该点集是图 G 的一个独立集（independent set），同时它也定义了图 G 的空子图。最大独立集是具有最大尺寸的独立集。对于任意图 G，它的补图（complement）是有同样点集的图，且当且仅当（u,v）不是 G 的一条边时，它是的一条边。

例 4-9 图 16-7b 是图 16-7a 的补图，反之亦然。{2, 4}定义了 16-7a 的一个空子图，它也是该图的一个最大独立集。虽然{1, 2}定义了图 16-7b 的一个空子图，它不是一个独立集，因

为它被包含在空子图{1,2,5}中。{1,2,5}是图16-7b 中的一个最大独立集。

如果*U*定义了*G*的一个完全子图，则它也定义了的一个空子图，反之亦然。所以在*G*的完备子图与的独立集之间有对应关系。特别的， *G*的一个最大完备子图定义了的一个最大独立集。

最大完备子图问题是指寻找图*G*的一个最大完备子图。类似地，最大独立集问题是指寻找图*G*的一个最大独立集。这两个问题都是NP-复杂问题。当用算法解决其中一个问题时，也就解决了另一个问题。例如，如果有一个求解最大完备子图问题的算法，则也能解决最大独立集问题，方法是首先计算所给图的补图，然后寻找补图的最大完备子图。

例 4-10 假定有一个 n 个动物构成的集合。可以定义一个有 n 个顶点的相容图（compatibilitygraph）*G*。当且仅当动物 u 和 v 相容时，（*u*,*v*）是 *G* 的一条边。*G* 的一个最大完备子图定义了相互间相容的动物构成的最大子集。

3.2节考察了如何找到一个具有最大尺寸的互不交叉的网组的集合问题。可以把这个问题看作是一个最大独立集问题。定义一个图，图中每个顶点表示一个网组。当且仅当两个顶点对应的网组交叉时，它们之间有一条边。所以该图的一个最大独立集对应于非交叉网组的一个最大尺寸的子集。当网组有一个端点在路径顶端，而另一个在底端时，非交叉网组的最大尺寸的子集能在多项式时间（实际上是(n_2)）内用动态规划算法得到。当一个网组的端点可能在平面中的任意地方时，不可能有在多项式时间内找到非交叉网组的最大尺寸子集的算法。最大完备子图问题和最大独立集问题可由回溯算法在O($n2_n$)时间内解决。两个问题都可使用子集解空间树（如图16-2所示）。考察最大完备子图问题，该递归回溯算法与程序16-3非常类似。当试图移动到空间树的 i 层节点 Z 的左孩子时，需要证明从顶点 i 到每一个其他的顶点 j（$x_j=1$ 且 j 在从根到 Z 的路径上）有一条边。当试图移动到 Z 的右孩子时，需要证明还有足够多的顶点未被搜索，以便在右子树有可能找到一个较大的完备子图。

回溯算法可作为类AdjacencyGraph（见程序12-4）的一个成员来实现，为此首先要在该类

中加入私有静态成员x（整型数组，用于存储到当前节点的路径），bestx（整型数组，保存目前的最优解），bestn（bestx中点的数量），cn（x 中点的数量）。所以类AdjacecyGraph的所有实

例都能共享这些变量。

函数maxClique(见程序16-10)是类AdjacencyGraph的一个私有成员，而MaxClique是一个

共享成员。函数maxClique对解空间树进行搜索，而MaxCique初始化必要的变量。Maxclique(v)

的执行返回最大完备子图的尺寸，同时它也设置整型数组 v，当且仅当顶点i 不是所找到的最大完备子图的一个成员时，v[i]=0。

程序 16-10 最大完备子图

```
void AdjacencyGraph::maxClique(int i)
{// 计算最大完备子图的回溯代码
if (i > n) {// 在叶子上
// 找到一个更大的完备子图，更新
for (int j = 1; j <= n; j++)
bestx[j] = x[j];
bestn = cn;
return;}
// 在当前完备子图中检查顶点 i 是否与其它顶点相连
int OK = 1;
```

```
for (int j = 1; j < i; j++)
if (x[j] && a[i][j] == NoEdge) {
// i 不与 j 相连
OK = 0;
b r e a k ; }
if (OK) {// 尝试 x[i] = 1
x[i] = 1; // 把 i 加入完备子图
c n + + ;
m a x C l i q u e ( i + 1 ) ;
x[i] = 0;
c n - - ; }
if (cn + n - i > bestn) {// 尝试 x[i] = 0
x[i] = 0;
m a x C l i q u e ( i + 1 ) ; }
}
int AdjacencyGraph::MaxClique(int v[])
{// 返回最大完备子图的大小
// 完备子图的顶点放入 v [ 1 : n ]
// 初始化
x = new int [n+1];
cn = 0;
bestn = 0;
bestx = v;
// 寻找最大完备子图
m a x C l i q u e ( 1 ) ;
delete [] x;
return bestn;
}
```

4.2.4 旅行商问题

旅行商问题（例 4.3）的解空间是一个排列树。这样的树可用函数 Perm(见程序 1-10)搜索，并可生成元素表的所有排列。如果以 $x=[1, 2, ., n]$ 开始，那么通过产生从 x_2 到 x_n 的所有排列，可生成 n 顶点旅行商问题的解空间。由于 Perm 产生具有相同前缀的所有排列，因此可以容易地改造 Perm，使其不能产生具有不可行前缀（即该前缀没有定义路径）或不可能比当前最优旅行还优的前缀的排列。注意在一个排列空间树中，由任意子树中的叶节点定义的排列有相同的前缀（如图 16-5 所示）。因此，考察时删除特定的前缀等价于搜索期间不进入相应的子树。旅行商问题的回溯算法可作为类 AdjacencyWDigraph(见程序 12-1)的一个成员。在其他例子中，有两个成员函数： tSP 和 TSP。前者是一个保护或私有成员，后者是一个共享成员。函数 G.TSP(v)返回最少耗费旅行的花费，旅行自身由整型数组 v 返回。若网络中无旅行，则返回 NoEdge。tSP 在排列空间树中进行递归回溯搜索， TSP 是其一个必要的预处理过程。TSP 假定 x（用来保存到当前节点的路径的整型数组），bestx（保存目前发现的最优旅行的整型数组），

cc（类型为 T 的变量，保存当前节点的局部旅行的耗费），bestc（类型为 T 的变量，保存目前最优解的耗费）已被定义为 AdjacencyWDigraph 中的静态数据成员。TSP 见程序 16-11。tSP(2)搜索一棵包含 x[2:n]的所有排列的树。

程序 16 - 11 旅行商回溯算法的预处理程序

```
template<class T>
T AdjacencyWDigraph<T>::TSP(int v[])
{// 用回溯算法解决旅行商问题
// 返回最优旅游路径的耗费，最优路径存入 v [ 1 : n ]
//初始化
x = new int [n+1];
// x 是排列
for (int i = 1; i <= n; i++)
x[i] = i;
bestc = NoEdge;
bestx = v; // 使用数组 v 来存储最优路径
cc = 0;
// 搜索 x [ 2 : n ]的各种排列
t S P ( 2 ) ;
delete [] x;
return bestc;
}
```

函数 t S P 见程序 1 6 - 1 2。它的结构与函数 P e r m 相同。当 i=n 时，处在排列树的叶节点的父节点上，并且需要验证从 x[n-1] 到 x[n] 有一条边，从 x[n] 到起点 x[1] 也有一条边。若两条边都存在，则发现了一个新旅行。在本例中，需要验证是否该旅行是目前发现的最优旅行。若是，则将旅行和它的耗费分别存入 b e s t x 与 b e s t c 中。

当 i＜n 时，检查当前 i-1 层节点的孩子节点，并且仅当以下情况出现时，移动到孩子节点之一：1) 有从 x[i-1] 到 x[i] 的一条边（如果是这样的话， x [1 : i]定义了网络中的一条路径）；2)路径 x[1:i] 的耗费小于当前最优解的耗费。变量 cc 保存目前所构造的路径的耗费。

每次找到一个更好的旅行时，除了更新 bestx 的耗费外， tS P 需耗时 O ((*n*- 1) !)。因为需发生 O ((*n*-1)!) 次更新且每一次更新的耗费为(*n*) 时间，因此更新所需时间为 O (*n** (*n*- 1) !)。通过使用加强的条件（练习 1 6），能减少由 tS P 搜索的树节点的数量。

程序 16-12 旅行商问题的迭代回溯算法

```
void AdjacencyWDigraph<T>::tSP(int i)
{// 旅行商问题的回溯算法
if (i == n) {// 位于一个叶子的父节点
// 通过增加两条边来完成旅行
if (a[x[n-1]][x[n]] != NoEdge &&
a[x[n]][1] != NoEdge &&
(cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc ||
bestc == NoEdge)) {// 找到更优的旅行路径
for (int j = 1; j <= n; j++)
bestx[j] = x[j];
bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];}
}
else {// 尝试子树
for (int j = i; j <= n; j++)
//能移动到子树 x [ j ]吗?
```

```
if (a[x[i-1]][x[j]] != NoEdge &&
(cc + a[x[i-1]][x[i]] < bestc ||
bestc == NoEdge)) {//能
// 搜索该子树
Swap(x[i], x[j]);
cc += a[x[i-1]][x[i]];
t S P ( i + 1 ) ;
cc -= a[x[i-1]][x[i]];
Swap(x[i], x[j]);}
}
}
```

4.2.5 电路板排列

在大规模电子系统的设计中存在着电路板排列问题。这个问题的经典形式为将 n 个电路板放置到一个机箱的许多插槽中，（如图 16-8 所示）。n 个电路板的每一种排列定义了一种放置方法。令 $B=\{b_1, ., b_n\}$表示这 n 个电路板。m 个网组集合 $L=\{N_1, ., N_m\}$由电路板定义，N_i 是 B 的子集，子集中的元素需要连接起来。实际中用电线将插有这些电路板的插槽连接起来。

例 4-11 令 n=8, m= 5。集合 B 和 L 如下:

$B=\{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8\}$

$L=\{N_1, N_2, N_3, N_4, N_5\}$

$N_1=\{b_4, b_5, b_6\}$

$N_2=\{b_2, b_3\}$

$N_3=\{b_1, b_3\}$

$N_4=\{b_3, b_6\}$

$N_5=\{b_7, b_8\}$

令 x 为电路板的一个排列。电路板 x_i 被放置到机箱的插槽 i 中。$density(x)$ 为机箱中任意一对相邻插槽间所连电线数目中的最大值。对于图 16-9 中的排列，$density$ 为 2。有两根电线连接了插槽 2 和 3，插槽 4 和 5，插槽 5 和 6。插槽 6 和 7 之间无电线，余下的相邻插槽都只有一根电线。板式机箱被设计成具有相同的相邻插槽间距，因此这个间距决定了机箱的大小。该间距必须保证足够大以便容纳相邻插槽间的连线。因此这个距离（继而机箱的大小）由 $density(x)$决定。

电路板排列问题的目标是找到一种电路板的排列方式，使其有最小的 $density$。既然该问题是一个 NP-复杂问题，故它不可能由一个多项式时间的算法来解决，而象回溯这样的搜索方法则是解决该问题的一种较好方法。回溯算法为了找到最优的电路板排列方式，将搜索一个排列空间。

用一个 $n \times m$ 的整型数组 B 表示输入，当且仅当 N_j 中包含电路板 b_i 时，B[i][j]=1。令 total[j]为 N_j 中电路板的数量。对于任意部分的电路板排列 x[1:i]，令 now[j]为既在 x[1:i]中又被包含在 N_j 中的电路板的数量。当且仅当 now[j]>0 且 now[j]≠total[j]时，N_j 在插槽 i 和 i+1 之间有连线。插槽 i 和 i+1 间的线密度可利用该测试方法计算出来。在插槽 k 和 k+1 (1≤k≤i) 间的线密度的最大值给出了局部排列的密度。

为了实现电路板排列问题的回溯算法，使用了类 Board（见程序 16-13）。程序 16-14 给出了私有方法 BestOrder，程序 16-15 给出了函数 ArrangeBoards。ArrangeBoards 返回最优的电路板排列密度，最优的排列由数组 bestx 返回。ArrangeBoards 创建类 Board 的一个成员 x 并初始化与之相关的变量。尤其是 total 被初始化以使 total[j] 等于 N_j 中电路板的数量。

now[1:n] 被置为 0，与一个空的局部排列相对应。调用 x .BestOrder(1,0) 搜索 x[1:n] 的排列树，以从密度为 0 的空排列中找到一个最优的排列。通常，x.BestOrder(i,cd) 寻找最优的局部排列 x [1 : i - 1]，该局部排列密度为 c d。函数 B e s t O r d e r（见程序 1 6 - 1 4）和程序 1 6 - 1 2 有同样的结构，它也搜索一个排列空间。当 i=n 时，表示所有的电路板已被放置且 cd 为排列的密度。既然这个算法只寻找那些比当前最优排列还优的排列，所以不必验证 cd 是否比 beste 要小。当 i<n 时，排列还未被完成。x[1:i-1] 定义了当前节点的局部排列，而 cd 是它的密度。这个节点的每一个孩子通过在当前排列的末端增加一个电路板而扩充了这个局部排列。对于每一个这样的扩充，新的密度 l d 被计算，且只有 ld<bestd 的节点被搜索，其他的节点和它们的子树不被搜索。在排列树的每一个节点处，函数 B e s t O r d e r 花费(m)的时间计算每一个孩子节点的密度。所以计算密度的总时间为 O (m n!)。此外，产生排列的时间为 O (n!) 且更新最优排列的时间为 O (m n)。

注意每一个更新至少将 b e s t d 的值减少 1，且最终 b e s t d ≥ 0。所以更新的次数是 O (m)。B e s t O r d e r 的整体复杂性为 O (m n!)。

程序 16-13 Board 的类定义

```
class Board {
friend ArrangeBoards(int**, int, int, int []);
p r i v a t e :
void BestOrder(int i, int cd);
int *x, // 到达当前节点的路径
* b e s t x , // 目前的最优排列
* t o t a l , // total[j] = 带插槽 j 的板的数目
* n o w, // now[j] = 在含插槽 j 的部分排列中的板的数目
b e s t d , // bestx 的密度
n , / /板的数目
m , // 插槽的数目
* * B ; // 板的二维数组
} ;
```

程序 16-14 搜索排列树

```
void Board::BestOrder(int i, int cd)
{// 按回溯算法搜索排列树
if (i == n) {
for (int j = 1; j <= n; j++)
bestx[j] = x[j];
bestd = cd;}
else // 尝试子树
for (int j = i; j <= n; j++) {
// 用 x[j] 作为下一块电路板对孩子进行尝试
// 在最后一个插槽更新并计算密度
int ld = 0;
for (int k = 1; k <= m; k++) {
now[k] += B[x[j]][k];
if (now[k] > 0 && total[k] != now[k])
l d + + ;
}
```

```
// 更新 ld 为局部排列的总密度
if (cd > ld) ld = cd;
// 仅当子树中包含一个更优的排列时，搜索该子树
if (ld < bestd) {// 移动到孩子
Swap(x[i], x[j]);
BestOrder(i+1, ld);
Swap(x[i], x[j]);}
// 重置
for (k = 1; k <= m; k++)
now[k] -= B[x[j]][k];
}
}
```

程序 16-15 BestOrder(程序 16-14)的预处理代码

```
int ArrangeBoards(int **B, int n, int m, int bestx[ ])
{// 返回最优密度
// 在 bestx 中返回最优排列
Board X;
// 初始化 X
X.x = new int [n+1];
X.total = new int [m+1];
X.now = new int [m+1];
X.B = B;
X.n = n;
X.m = m;
X.bestx = bestx;
X.bestd = m + 1;
// 初始化 total 和 now
for (int i = 1; i <= m; i++) {
X.total[i] = 0;
X.now[i] = 0;
}
// 初始化 x 并计算 total
for (i = 1; i <= n; i++) {
X.x[i] = i;
for (int j = 1; j <= m; j++)
X.total[j] += B[i][j];
}
// 寻找最优排列
X.BestOrder(1,0);
delete [] X.x;
delete [] X.total;
delete [] X.now;
return X.bestd;
```

练习

4. 证明在两船装载问题中，只要存在一种方法能装载所有货箱，则通过尽可能装满第一艘船就可找到一种可行的装载方法。
5. 运行程序 16-3 和 16-4 的代码，测试它们的相对运行时间。
6. 运用 16.2.1 节中第 4 小节的方法 1) 来更新程序 16-3，使其能得到时间复杂性 $O(2^n)$。
7. 运用 16.2.1 节中第 4 小节的方法 2) 来修改程序 16-3，使其运行时间减少至 $O(2^n)$。
8. 为子集之和问题写一个递归回溯算法。注意，只要找到一个子集，其和为 c_1，则可终止程序运行。没有必要记住目前的最优解。代码不应使用像程序 16-3 中的数组 x。在找到和为 c_1 的子集之后展开递归，可以重构出最优解。
9. 优化程序 16-7 和 16-9 以便能产生出与背包问题最优解相对应的一个 0/1 数组 x。
10. 用迭代回溯算法求解 0/1 背包问题。该算法与程序 16-4 类似。可以修改 Knap::Bound，使其返回被装入背包的最后一个对象 i，这样可避免根据 Bound 重新向左移动而可直接移动到最左节点（原先由 Bound 确定）。
11. 编写程序 16-10（与程序 16-4 相对应）的迭代版本并比较这两个版本。
12. 改写程序 16-10，使其首先按度的递减次序来排列各顶点。你认为该版本比程序 16-10 好吗？
13. 编写一个求解最大独立集问题的回溯算法。
14. 重写最大完备子图代码（见程序 16-10），把它作为类 UNetwork 的成员。对于类 ADjacencyGraph, AdjacencyWGraph, LinkedGraph 和 LinkedWGraph（见 12.7 节）的成员，该代码同样有效。
15. 令 G 为一个 n 顶点的有向图，Max_i 为从顶点 i 出发的具有最大耗费的边的耗费
1) 证明旅行商的每一个旅行有一个小于 $\sum_{i=1}^{n} Max_i + 1$ 的耗费。
2) 使用上述界限作为 bestc 的初始值。重写 TSP 和 tSP，尽可能简化它们。
16. 令 G 为具有 n 个顶点的有向图，$MinOut_i$ 为从顶点 i 出发的具有最小耗费的边的耗费
1) 证明具有前缀 x_1 到 x_i 的旅行商的所有旅行耗费至少为 $\sum_{j=2}^{i} A(x_{j-1}, x_j) + \sum_{y=i}^{n} MinOut_{xy}$ 其中 $A(u,v)$是边(u,v)的耗费。
2) 在程序 16-12 中，使用

```
if (a [x [i-1] ] [x [j] ] ! = NoEdge &&(cc + a [x [i-1] ] [x [i] ] < bestc | |bestc == NoEdge) )
```

来决定何时移动到一个孩子节点。要求使用 1) 的结果得到一个更强的条件。第一个和可根据 cc 计算出来，通过用一个新变量 r 保留不在当前路径中的顶点的 MinOut[i]的和，可以很容易地计算出第二个和。
3) 测试 tSP 的新版本。与程序 16-12 比较，它访问了排列树的多少节点？
17. 考察电路板排列问题。N_i 的长度为 N_i 中第一块和最后一块电路板间的距离。N_4 中第一个电路板在插槽 3 中，最后一个电路板在插槽 6 中，则 N_4 的长度为 3。N_2 的长度为 2。N_i 最大值为 3。编写一个回溯算法以找到具有最小的最大长度的板排列。试测试代码的正确性。
18. [顶点覆盖] 令 G 为一个无向图。当且仅当对于 G 中的每一条边(u, v)，u 或 v 或 u, v 在 U 中时，G 的顶点子集 U 是一个顶点覆盖（vertex cover）。U 中顶点的数量是覆盖的大小。在图 16-7a 中，{1, 2, 5}是大小为 3 的一个顶点覆盖。编写一个回溯算法寻找具有最小尺寸的顶点覆盖。算法的复杂性是多少？
19. [简易最大切割] 令 G 是一个无向图。U 是 G 中顶点的任意子集。V 是 G 余下的点的集合。一个端点在 U 中，另一个端点在 V 中的边的数量是 U 所定义的切割（cut）的大小。编写一个回溯算法，寻找最大切割的大小和相应的 U。算法的复杂性是多少？
20. [机器设计] 某机器由 n 个部件组成，每一个部件可从 3 个投资者那里获得。令 w_{ij} 是从投资者 j 那里得到的零件 i 的重量，c_{ij} 则为该零件的耗费。编写一个回溯算法，找出耗费不超过 c 的机器构成方案，使其重量最少。算法的复杂性是多少？

21. [网络设计] 一个汽油传送网络可由加权的有向无环图 G 表示。G 中有一个称为原点的顶点 S。从 S 出发，汽油被输送到图中的其他顶点。S 的入度为 0，每一条边上的权给出了它所连接的两点间的距离。通过网络输送汽油时，压力的损失是所走距离的函数。为了保证网络的正常运转，在网络传输中必须保证最小压力 P_{min}。为了维持这个最小压力，可将压力放大器放在网络中的一些或全部顶点。压力放大器可将压力恢复至最大可允许的量级 P_{max}。令 d 为汽油在压力由 P_{max} 降为 P_{min} 时所走的距离。在设置信号放大器问题中，需要放置最少数量的放大器，以便在遇到一个放大器之前汽油所走的距离不超过 d。编写一个回溯算法来求解该问题。算法的复杂性是多少？

22. [n 皇后问题] 在 n 皇后问题中，我们希望在 $n \times n$ 的棋盘上找到一个 n 皇后的放置方法以便任意两个皇后之间不冲突。当且仅当两个皇后在相同的排、列、对角线或反对角线上时，她们之间将发生冲突。假定在任何可行的解决方案中，皇后 i 被放置在棋盘的第 i 排。所以只对决定每一个皇后所在的列感兴趣。令 c_i 为皇后 i 所处的列。如果任意两个皇后不冲突，则$[c_1, .,c_n]$是$[1, 2, ., n]$的一个排列。n 皇后问题的解空间因此被限制到$[1, 2, ., n]$的所有排列中。

1) 将 n 皇后的解空间组织成一棵树。

2) 编写一个回溯算法，搜索 n 皇后问题的可行排列。

*23. 编写一个函数，使用回溯算法来搜索一个子集空间树，该树为一个二叉树。函数中的参数应包含如下函数：确定一个节点是否可行的函数，计算该节点的界限值的函数，决定界限是否优于另一个值的函数等。用 0/1 背包问题来测试程序。

*24. 使用排列空间树来完成练习 2 3。

*25. 编写一个函数，用回溯法搜索一个解空间。函数中的参数应包括下列函数：产生节点的下一个孩子的函数，决定下一个孩子是否是可行的函数，计算该节点界限的函数，决定该界限值是否优于另一个值的函数等。用 0/1 背包问题来测试程序。

（ 说明：本资料是根据《数据结构、算法与应用》（美，Sartaj Sahni 著）一书第 13-17 章编辑、改写的。考虑到因特网传输速度等因素，大部分插图和公式不得不被删除。对于内容不连贯之处，请网友或读者参阅该书，敬请原谅。 ）

第 5 章 分枝定界

任何美好的事情都有结束的时候。现在我们学习的是本书的最后一章。幸运的是，本章用到的大部分概念在前面各章中已作了介绍。类似于回溯法，分枝定界法在搜索解空间时，也经常使用树形结构来组织解空间（常用的树结构是第 16 章所介绍的子集树和排列树）。然而与回溯法不同的是，回溯算法使用深度优先方法搜索树结构，而分枝定界一般用宽度优先或最小耗费方法来搜索这些树。本章与第 16 章所考察的应用完全相同，因此，可以很容易比较回溯法与分枝定界法的异同。相对而言，分枝定界算法的解空间比回溯法大得多，因此当内存容量有限时，回溯法成功的可能性更大。

5.1 算法思想

分枝定界（branch and bound）是另一种系统地搜索解空间的方法，它与回溯法的主要区别在于对 E-节点的扩充方式。每个活节点有且仅有一次机会变成 E-节点。当一个节点变为 E-节点时，则生成从该节点移动一步即可到达的所有新节点。在生成的节点中，抛弃那些不可能导出（最优）可行解的节点，其余节点加入活节点表，然后从表中选择一个节点作为下一个 E-节点。

从活节点表中取出所选择的节点并进行扩充，直到找到解或活动表为空，扩充过程才结束。

有两种常用的方法可用来选择下一个E-节点（虽然也可能存在其他的方法）：

1) 先进先出（FIFO） 即从活节点表中取出节点的顺序与加入节点的顺序相同，因此活节点表的性质与队列相同。

2) 最小耗费或最大收益法在这种模式中，每个节点都有一个对应的耗费或收益。如果查找一个具有最小耗费的解，则活节点表可用最小堆来建立，下一个E-节点就是具有最小耗费的活节点；如果希望搜索一个具有最大收益的解，则可用最大堆来构造活节点表，下一个E-节点是具有最大收益的活节点。

例 5-1 [迷宫老鼠] 考察图 16-3a 给出的迷宫老鼠例子和图 16-1 的解空间结构。使用 FIFO 分枝定界，初始时取（1，1）作为 E-节点且活动队列为空。迷宫的位置（ 1,1）被置为 1，以免再次返回到这个位置。（1，1）被扩充，它的相邻节点（ 1，2）和（2，1）加入到队列中（即活节点表）。为避免再次回到这两个位置，将位置（ 1，2）和（2，1）置为 1。此时迷宫如图 17-1a 所示，E-节点（1，1）被删除。

节点（1，2）从队列中移出并被扩充。检查它的三个相邻节点（见图 16-1 的解空间），只有（1，3）是可行的移动（剩余的两个节点是障碍节点），将其加入队列，并把相应的迷宫位置置为 1，所得到的迷宫状态如图 17-1b 所示。节点（1，2）被删除，而下一个 E-节点（2，1）将会被取出，当此节点被展开时，节点（3，1）被加入队列中，节点（3，1）被置为 1，节点（2，1）被删除，所得到的迷宫如图 17-1c 所示。此时队列中包含（1，3）和（3，1）两个节点。随后节点（1，3）变成下一个 E-节点，由于此节点不能到达任何新的节点，所以此节点即被删除，节点（3，1）成为新的 E-节点，将队列清空。节点（ 3，1）展开，（3，2）被加入队列中，而（3，1）被删除。（3，2）变为新的 E-节点，展开此节点后，到达节点（3，3），即迷宫的出口。

使用 FIFO 搜索，总能找出从迷宫入口到出口的最短路径。需要注意的是：利用回溯法找到的路径却不一定是最短路径。有趣的是，程序 6-11 已经给出了利用 FIFO 分枝定界搜索从迷宫的（1，1）位置到（n，n） 位置的最短路径的代码。

例 5-2 [0/1 背包问题] 下面比较分别利用 FIFO 分枝定界和最大收益分枝定界方法来解决如下背包问题：n=3, w=[20,15,15], p=[40,25,25], c= 30。FIFO 分枝定界利用一个队列来记录活节点，节点将按照 FIFO 顺序从队列中取出；而最大收益分枝定界使用一个最大堆，其中的 E-节点按照每个活节点收益值的降序，或是按照活节点任意子树的叶节点所能获得的收益估计值的降序从队列中取出。本例所使用的背包问题与例 16.2 相同，并且有相同的解空间树。

使用 FIFO 分枝定界法搜索，初始时以根节点 A 作为 E-节点，此时活节点队列为空。当节点 A 展开时，生成了节点 B 和 C，由于这两个节点都是可行的，因此都被加入活节点队列中，节点 A 被删除。下一个 E-节点是 B，展开它并产生了节点 D 和 E，D 是不可行的，被删除，而 E 被加入队列中。下一步节点 C 成为 E-节点，它展开后生成节点 F 和 G，两者都是可行节点，加入队列中。下一个 E-节点 E 生成节点 J 和 K，J 不可行而被删除，K 是一个可行的叶节点，并产生一个到目前为止可行的解，它的收益值为 40。

下一个 E-节点是 F，它产生两个孩子 L、M，L 代表一个可行的解且其收益值为 50，M 代表另一个收益值为 15 的可行解。G 是最后一个 E-节点，它的孩子 N 和 O 都是可行的。由于活节点队列变为空，因此搜索过程终止，最佳解的收益值为 50。

可以看到，工作在解空间树上的 FIFO 分枝定界方法非常象从根节点出发的宽度优先搜索。它们的主要区别是在 FIFO 分枝定界中不可行的节点不会被搜索。最大收益分枝定界算法以解空间树中的节点 A 作为初始节点。展开初始节点得到节点 B 和 C，两者都是可行的并被插入堆中，节点 B 获得的收益值是 40（设 x_1=1），而节点 C 得到的收益值为 0。A 被删除，B 成为下一个 E-节点，因为它的收益值比 C 的大。当展开 B 时得到了节点 D 和 E，D 是不可行的而被删

除，E 加入堆中。由于 E 具有收益值 40，而 C 为 0，因为 E 成为下一个 E-节点。

展开 E 时生成节点 J 和 K，J 不可行而被删除，K 是一个可行的解，因此 K 为作为目前能找到的最优解而记录下来，然后 K 被删除。由于只剩下一个活节点 C 在堆中，因此 C 作为 E-节点被展开，生成 F、G 两个节点插入堆中。F 的收益值为 25，因此成为下一个 E-节点，展开后得到节点 L 和 M，但 L、M 都被删除，因为它们是叶节点，同时 L 所对应的解被作为当前最优解记录下来。最终，G 成为 E-节点，生成的节点为 N 和 O，两者都是叶节点而被删除，两者所对应的解都不比当前的最优解更好，因此最优解保持不变。此时堆变为空，没有下一个 E-节点产生，搜索过程终止。终止于 J 的搜索即为最优解。

犹如在回溯方法中一样，可利用一个定界函数来加速最优解的搜索过程。定界函数为最大收益设置了一个上限，通过展开一个特殊的节点可能获得这个最大收益。如果一个节点的定界函数值不大于目前最优解的收益值，则此节点会被删除而不作展开，更进一步，在最大收益分枝定界方法中，可以使节点按照它们收益的定界函数值的非升序从堆中取出，而不是按照节点的实际收益值来取出。这种策略从可能到达一个好的叶节点的活节点出发，而不是从目前具有较大收益值的节点出发。

例 5-3 [旅行商问题] 对于图 16-4 的四城市旅行商问题，其对应的解空间为图 16-5 所示的排列树。FIFO 分枝定界使用节点 B 作为初始的 E-节点，活节点队列初始为空。当 B 展开时，生成节点 C、D 和 E。由于从顶点 1 到顶点 2，3，4 都有边相连，所以 C、D、E 三个节点都是可行的并加入队列中。当前的 E-节点 B 被删除，新的 E-节点是队列中的第一个节点，即节点 C。因为在图 16-4 中存在从顶点 2 到顶点 3 和 4 的边，因此展开 C，生成节点 F 和 G，两者都被加入队列。下一步，D 成为 E-节点，接着又是 E，到目前为止活节点队列中包含节点 F 到 K。下一个 E-节点是 F，展开它得到了叶节点 L。至此找到了一个旅行路径，它的开销是 59。展开下一个 E-节点 G，得到叶节点 M，它对应于一个开销为 66 的旅行路径。接着 H 成为 E-节点，从而找到叶节点 N，对应开销为 25 的旅行路径。下一个 E-节点是 I，它对应的部分旅行 1-3-4 的开销已经为 26，超过了目前最优的旅行路径，因此， I 不会被展开。最后，节点 J，K 成为 E-节点并被展开。经过这些展开过程，队列变为空，算法结束。找到的最优方案是节点 N 所对应的旅行路径。

如果不使用 FIFO 方法，还可以使用最小耗费方法来搜索解空间树，即用一个最小堆来存储活节点。这种方法同样从节点 B 开始搜索，并使用一个空的活节点列表。当节点 B 展开时，生成节点 C、D 和 E 并将它们加入最小堆中。在最小堆的节点中， E 具有最小耗费（因为 1-4 的局部旅行的耗费是 4），因此成为 E-节点。展开 E 生成节点 J 和 K 并将它们加入最小堆，这两个节点的耗费分别为 14 和 24。此时，在所有最小堆的节点中，D 具有最小耗费，因而成为 E-节点，并生成节点 H 和 I。至此，最小堆中包含节点 C、H、I、J 和 K，H 具有最小耗费，因此 H 成为下一个 E-节点。展开节点 E，得到一个完整的旅行路径 1-3-2-4-1，它的开销是 25。节点 J 是下一个 E-节点，展开它得到节点 P，它对应于一个耗费为 25 的旅行路径。节点 K 和 I 是下两个 E-节点。由于 I 的开销超过了当前最优的旅行路径，因此搜索结束，而剩下的所有活节点都不能使我们找到更优的解。

对于例 5-2 的背包问题，可以使用一个定界函数来减少生成和展开的节点数量。这种函数将确定旅行的最小耗费的下限，这个下限可通过展开某个特定的节点而得到。如果一个节点的定界函数值不能比当前的最优旅行更小，则它将被删除而不被展开。另外，对于最小耗费分枝定界，节点按照它在最小堆中的非降序取出。

在以上几个例子中，可以利用定界函数来降低所产生的树型解空间的节点数目。当设计定界函数时，必须记住主要目的是利用最少的时间，在内存允许的范围内去解决问题。而通过产生具有最少节点的树来解决问题并不是根本的目标。因此，我们需要的是一个能够有效地减少计算时间并因此而使产生的节点数目也减少的定界函数。

回溯法比分枝定界在占用内存方面具有优势。回溯法占用的内存是O(解空间的最大路径长度)，而分枝定界所占用的内存为O(解空间大小)。对于一个子集空间，回溯法需要(n)的内存空间，而分枝定界则需要O(2_n)的空间。对于排列空间，回溯需要(n)的内存空间，分枝定界需要O($n!$)的空间。虽然最大收益(或最小耗费)分枝定界在直觉上要好于回溯法，并且在许多情况下可能会比回溯法检查更少的节点，但在实际应用中，它可能会在回溯法超出允许的时间限制之前就超出了内存的限制。

练习

1. 假定在一个LIFO分枝定界搜索中，活节点列表的行为与堆栈相同，请使用这种方法来解决例5-2的背包问题。LIFO分枝定界与回溯有何区别?

2. 对于如下0/1背包问题: n=4, p=[4,3,2,1], w=[1,2,3,4], c=6。

1) 画出有四个对象的背包问题的解空间树。

2) 像例17-2那样，描述用FIFO分枝定界法解决上述问题的过程。

3) 使用程序16-6的Bound函数来计算子树上任一叶节点可能获得的最大收益值，并根据每一步所能得到的最优解对应的定界函数值来判断是否将节点加入活节点列表中。解空间中哪些节点是使用以上机制的FIFO分枝定界方法产生的?

4) 像例17-2那样，描述用最大收益分枝定界法解决上述问题的过程。

5) 在最大收益分枝定界中，若使用3)中的定界函数，将产生解空间树中的哪些节点?

5.2 应用

5.2.1 货箱装船

1. FIFO分枝定界

4.2.1节的货箱装船问题主要是寻找第一条船的最大装载方案。这个问题是一个子集选择问题，它的解空间被组织成一个子集树。对程序16-1进行改造，即得到程序17-1中的FIFO分枝定界代码。程序17-1只是寻找最大装载的重量。

程序17-1 货箱装船问题的FIFO分枝定界算法

```
template<class T>
void AddLiveNode(LinkedQueue<T> &Q, T wt,
T& bestw, int i, int n)
{// 如果不是叶节点，则将节点权值wt加入队列Q
if (i == n) {// 叶子
if (wt > bestw) bestw = wt;}
else Q.Add(wt); // 不是叶子
}
template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载值
// 使用FIFO分枝定界算法
// 为层次1初始化
LinkedQueue<T> Q; // 活节点队列
Q.Add(-1); //标记本层的尾部
int i = 1; // E-节点的层
T Ew = 0, // E-节点的权值
bestw = 0; // 目前的最优值
```

```
// 搜索子集空间树
while (true) {
// 检查 E-节点的左孩子
if (Ew + w[i] <= c) // x[i] = 1
AddLiveNode(Q, Ew + w[i], bestw, i, n);
// 右孩子总是可行的
AddLiveNode(Q, Ew, bestw, i, n); // x[i] = 0
Q.Delete(Ew); // 取下一个 E-节点
if (Ew == -1) { // 到达层的尾部
if (Q.IsEmpty()) return bestw;
Q.Add(-1); //添加尾部标记
Q.Delete(Ew); // 取下一个 E-节点
i++;} // Ew 的层
}
}
```

其中函数 MaxLoading 在解空间树中进行分枝定界搜索。链表队列 Q 用于保存活节点，其中记录着各活节点对应的权值。队列还记录了权值-1，以标识每一层的活节点的结尾。函数 AddLiveNode 用于增加节点（即把节点对应的权值加入活节点队列），该函数首先检验 i（当前 E-节点在解空间树中的层）是否等于 n，如果相等，则已到达了叶节点。叶节点不被加入队列中，因为它们不被展开。搜索中所到达的每个叶节点都对应着一个可行的解，而每个解都会与目前的最优解来比较，以确定最优解。如果 $i<n$，则节点 i 就会被加入队列中。

MaxLoading 函数首先初始化 i=1（因为当前 E-节点是根节点），bestw=0（目前最优解的对应值），此时，活节点队列为空。下一步， A-1 被加入队列以说明正处在第一层的末尾。当前 E-节点对应的权值为 E_w。在 while 循环中，首先检查节点的左孩子是否可行。如果可行，则调用 AddLiveNode，然后将右孩子加入队列（此节点必定是可行的），注意到 AddliveNode 可能会失败，因为可能没有足够的内存来给队列增加节点。AddLiveNode 并没有去捕获 Q.Add 中的 NoMem 异常，这项工作留给用户完成。

如果 E-节点的两个孩子都已经被生成，则删除该 E-节点。从队列中取出下一个 E-节点，此时队列必不为空，因为队列中至少含有本层末尾的标识- 1。如果到达了某一层的结尾，则从下一层寻找活节点，当且仅当队列不为空时这些节点存在。当下一层存在活节点时，向队列中加入下一层的结尾标志并开始处理下一层的活节点。

MaxLoading 函数的时间和空间复杂性都是 $O(2^n)$。

2. 改进

我们可以尝试使用程序 16-2 的优化方法改进上述问题的求解过程。在程序 16-2 中，只有当右孩子对应的重量加上剩余货箱的重量超出 bestw 时，才选择右孩子。而在程序 17-1 中，在 i 变为 n 之前，bestw 的值一直保持不变，因此在 i 等于 n 之前对右孩子的测试总能成功，因为 bestw=0 且 r>0。当 i 等于 n 时，不会再有节点加入队列中，因此这时对右孩子的测试不再有效。

如想要使右孩子的测试仍然有效，应当提早改变 bestw 的值。我们知道，最优装载的重量是子集树中可行节点的重量的最大值。由于仅在向左子树移动时这些重量才会增大，因此可以在每次进行这种移动时改变 bestw 的值。根据以上思想，我们设计了程序 17-2。当活节点加入队列时，wt 不会超过 bestw，故 bestw 不用更新。因此用一条直接插入 MaxLoading 的简单语句取代了函数 AddLiveNode。

程序 17-2 对程序 17-1 改进之后

```
template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载值
// 使用 F I F O 分枝定界算法
// 为层 1 初始化
LinkedQueue<T> Q; // 活节点队列
Q . A d d ( - 1 ) ; / /标记本层的尾部
int i = 1; // E-节点的层
T Ew = 0, // E-节点的重量
bestw = 0; // 目前的最优值
r = 0; // E-节点中余下的重量
for (int j = 2; j <= n; j++)
r += w[i];
// 搜索子集空间树
while (true) {
// 检查 E-节点的左孩子
T wt = Ew + w[i]; // 左孩子的权值
if (wt <= c) { // 可行的左孩子
if (wt > bestw) bestw = wt;
// 若不是叶子，则添加到队列中
if (i < n) Q.Add(wt);}
// 检查右孩子
if (Ew + r > bestw && i < n)
Q.Add(Ew); // 可以有一个更好的叶子
Q.Delete(Ew); // 取下一个 E-节点
if (Ew == -1) { // 到达层的尾部
if (Q.IsEmpty()) return bestw;
Q.Add(-1); //添加尾部标记
Q.Delete(Ew); // 取下一个 E-节点
i++; // E-节点的层
r -= w[i];} // E-节点中余下的重量
}
}
```

3. 寻找最优子集

为了找到最优子集，需要记录从每个活节点到达根的路径，因此在找到最优装载所对应的叶节点之后，就可以利用所记录的路径返回到根节点来设置 x 的值。活节点队列中元素的类型是 Q N o d e (见程序 1 7 - 3)。这里，当且仅当节点是它的父节点的左孩子时， L C h i l d 为 t r u e。

程序 17-3 类 Q N o d e

```
template<class T>
class QNode {
p r i v a t e :
QNode *parent; // 父节点指针
bool LChild; // 当且仅当是父节点的左孩子时，取值为 t r u e
T weight; // 由到达本节点的路径所定义的部分解的值
```

};

程序 17-4 是新的分枝定界方法的代码。为了避免使用大量的参数来调用 AddLiveNode，可以把该函数定义为一个内部函数。使用内部函数会使空间需求稍有增加。此外，还可以把 AddLiveNode 和 MaxLoading 定义成类成员函数，这样，它们就可以共享诸如 Q, i, n, bestw, E, bestE 和 bestw 等类成员。

程序 17-4 并未删除类型为 QNode 的节点。为了删除这些节点，可以保存由 AddLiveNode 创建的所有节点的指针，以便在程序结束时删除这些节点。

程序 17-4 计算最优子集的分枝定界算法

```
template<class T>
void AddLiveNode(LinkedQueue<QNode<T>*> &Q, T wt, int i, int n, T bestw, QNode<T> *E,
QNode<T> *&bestE, int bestx[], bool ch)
{// 如果不是叶节点，则向队列 Q 中添加一个 i 层、重量为 wt 的活节点
// 新节点是 E 的一个孩子。当且仅当新节点是左孩子时， ch 为 true。
// 若是叶子，则 ch 取值为 bestx[n]
if (i == n) {// 叶子
if (wt == bestw) {
// 目前的最优解
bestE = E;
bestx[n] = ch;}
return ; }
// 不是叶子, 添加到队列中
QNode<T> *b;
b = new QNode<T>;
b->weight = wt;
b->parent = E;
b->LChild = ch;
Q.Add(b);
}
template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{// 返回最优装载值，并在 bestx 中返回最优装载
// 使用 FIFO 分枝定界算法
// 初始化层 1
LinkedQueue<QNode<T>*> Q; // 活节点队列
Q.Add(0); // 0 代表本层的尾部
int i = 1; // E-节点的层
T Ew = 0, // E-节点的重量
bestw = 0; // 迄今得到的最优值
r = 0; // E-节点中余下的重量
for (int j = 2; j <= n; j++)
r += w[i];
QNode<T> *E = 0, // 当前的 E-节点
* bestE; // 目前最优的 E-节点
// 搜索子集空间树
```

```
while (true) {
// 检查 E-节点的左孩子
T wt = Ew + w[i];
if (wt <= c) {// 可行的左孩子
if (wt > bestw) bestw = wt;
AddLiveNode(Q, wt, i, n, bestw, E, bestE, bestx, true);}
// 检查右孩子
if (Ew + r > bestw) AddLiveNode(Q, Ew, i, n, bestw, E, bestE, bestx, false);
Q.Delete(E); // 下一个 E-节点
if (!E) { // 层的尾部
if (Q.IsEmpty()) break;
Q.Add(0); // 层尾指针
Q.Delete(E); // 下一个 E-节点
i++; // E-节点的层次
r -= w[i];} // E-节点中余下的重量
Ew = E->weight; // 新的 E-节点的重量
}
// 沿着从 bestE 到根的路径构造 x[]， x[n]由 AddLiveNode 来设置
for (j = n - 1; j > 0; j--) {
bestx[j] = bestE->LChild; // 从 bool 转换为 int
bestE = bestE->parent;
}
return bestw;
}
```

4. 最大收益分枝定界

在对子集树进行最大收益分枝定界搜索时，活节点列表是一个最大优先级队列，其中每个活节点 x 都有一个相应的重量上限（最大收益）。这个重量上限是节点 x 相应的重量加上剩余货箱的总重量，所有的活节点按其重量上限的递减顺序变为 E-节点。需要注意的是，如果节点 x 的重量上限是 x.uweight，则在子树中不可能存在重量超过 x.uweight 的节点。另外，当叶节点对应的重量等于它的重量上限时，可以得出结论：在最大收益分枝定界算法中，当某个叶节点成为 E-节点并且其他任何活节点都不会帮助我们找到具有更大重量的叶节点时，最优装载的搜索终止。

上述策略可以用两种方法来实现。在第一种方法中，最大优先级队列中的活节点都是互相独立的，因此每个活节点内部必须记录从子集树的根到此节点的路径。一旦找到了最优装载所对应的叶节点，就利用这些路径信息来计算 x 值。在第二种方法中，除了把节点加入最大优先队列之外，节点还必须放在另一个独立的树结构中，这个树结构用来表示所生成的子集树的一部分。当找到最大装载之后，就可以沿着路径从叶节点一步一步返回到根，从而计算出 *x* 值。

最大优先队列可用 HeapNode 类型的最大堆来表示（见程序 17-5）。uweight 是活节点的重量上限，level 是活节点所在子集树的层， ptr 是指向活节点在子集树中位置的指针。子集树中节点的类型是 bbnode（见程序 17-5）。节点按 uweight 值从最大堆中取出。

程序 17-5 bbnode 和 HeapNode 类

```
class bbnode {
private:
bbnode *parent; // 父节点指针
```

```
bool LChild; // 当且仅当是父节点的左孩子时，取值为 true
};
template<class T>
class HeapNode {
public:
operator T () const {return uweight;}
private:
bbnode *ptr; // 活节点指针
T uweight; // 活节点的重量上限
int level; // 活节点所在层
};
```

程序 17-6 中的函数 AddLiveNode 用于把 bbnode 类型的活节点加到子树中，并把 HeapNode 类型的活节点插入最大堆。AddLiveNode 必须被定义为 bbnode 和 HeapNode 的友元。

程序 17-6

```
template<class T>
void AddLiveNode(MaxHeap<HeapNode<T> > &H, bbnode *E, T wt, bool ch, int lev)
{// 向最大堆 H 中增添一个层为 lev 上限重量为 wt 的活节点
// 新节点是 E 的一个孩子
// 当且仅当新节点是左孩子 ch 为 true
bbnode *b = new bbnode;
b->parent = E;
b->LChild = ch;
HeapNode<T> N;
N.uweight = wt;
N.level = lev;
N.ptr = b;
H.Insert(N);
}
template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{// 返回最优装载值，最优装载方案保存于 bestx
// 使用最大收益分枝定界算法
// 定义一个最多有 1000 个活节点的最大堆
MaxHeap<HeapNode<T> > H(1000);
// 第一剩余重量的数组
// r[j] 为 w[j + 1 : n]的重量之和
T *r = new T [n+1];
r[n] = 0;
for (int j = n-1; j > 0; j--)
r[j] = r[j+1] + w[j+1];
// 初始化层 1
int i = 1; // E-节点的层
bbnode *E = 0; // 当前 E-节点
```

```
T Ew = 0; // E-节点的重量
// 搜索子集空间树
while (i != n+1) {// 不在叶子上
// 检查 E-节点的孩子
if (Ew + w[i] <= c) {// 可行的左孩子
AddLiveNode(H, E, Ew+w[i]+r[i], true, i+1);}
// 右孩子
AddLiveNode(H, E, Ew+r[i], false, i+1);
// 取下一个 E-节点
HeapNode<T> N;
H.DeleteMax(N); // 不能为空
i = N.level;
E = N.ptr;
Ew = N.uweight - r[i-1];
}
// 沿着从 E-节点 E 到根的路径构造 b e s t x [ ]
for (int j = n; j > 0; j--) {
bestx[j] = E->LChild; // 从 b o o l 转换为 i n t
E = E-> p a r e n t ;
}
return Ew;
}
```

函数 M a x L o a d i n g（见程序 1 7 - 6）首先定义了一个容量为 1 0 0 0 的最大堆，因此，可以用它来解决优先队列中活节点数在任何时候都不超过 1 0 0 0 的装箱问题。对于更大型的问题，需要一个容量更大的最大堆。接着，函数 M a x L o a d i n g 初始化剩余重量数组 r。第 i + 1 层的节点（即 x [1 : i]的值都已确定）对应的剩余容器总重量可以用如下公式求出:

$$r[i]=\sum_{j=i+1}^{n} w[j]。$$

变量 E 指向子集树中的当前 E-节点，Ew 是该节点对应的重量， i 是它所在的层。初始时，根节点是 E-节点，因此取 i=1, Ew=0。由于没有明确地存储根节点，因此 E 的初始值取为 0。while 循环用于产生当前 E-节点的左、右孩子。如果左孩子是可行的（即它的重量没有超出容量），则将它加入到子集树中并作为一个第 i + 1 层节点加入最大堆中。一个可行的节点的右孩子也被认为是可行的，它总被加入子树及最大堆中。在完成添加操作后，接着从最大堆中取出下一个 E-节点。如果没有下一个 E-节点，则不存在可行的解。如果下一个 E-节点是叶节点（即是一个层为 n + 1 的节点），则它代表着一个最优的装载，可以沿着从叶到根的路径来确定装载方案。

5. 说明

1) 使用最大堆来表示活节点的最大优先队列时，需要预测这个队列的最大长度（程序 1 7 - 6 中是 1 0 0 0）。为了避免这种预测，可以使用一个基于指针的最大优先队列来取代基于数组的队列，这种表示方法见 9 . 4 节的左高树。

2) bestw 表示当前所有可行节点的重量的最大值，而优先队列中可能有许多其 u w e i g h t 不超过 b e s t w 的活节点，因此这些节点不可能帮助我们找到最优的叶节点，这些节点浪费了珍贵的队列空间，并且它们的插入/删除动作也浪费了时间，所以可以将这些节点删除。有一种策略可以减少这种浪费，即在插入某个节点之前检查是否有 u w e i g h t < b e s t w。然而，由于 b e s t w 在算法执行过程中是不断增大的，所以目前插入的节点在以后并不能保证 u w e i g h t < b e s t w。另一种更好的方法是在每次 b e s t w 增大时，删除队列中所有 u w e i g h t < b e s e w 的节点。这

种策略要求删除具有最小uweight的节点。因此，队列必须支持如下的操作：插入、删除最大节点、删除最小节点。这种优先队列也被称作双端优先队列（ double-ended priority queue）。这种队列的数据结构描述见第9章的参考文献。

5.2.2 0/1背包问题

0/1背包问题的最大收益分枝定界算法可以由程序16-6发展而来。可以使用程序16-6的Bound函数来计算活节点N的收益上限up,使得以N为根的子树中的任一节点的收益值都不可能超过uprofit。活节点的最大堆使用uprofit作为关键值域，最大堆的每个入口都以HeapNode作为其类型，HeapNode有如下私有成员：uprofit, profit, weight，level，ptr，其中level和ptr的定义与装箱问题（见程序17-5）中的含义相同。对任一节点N，N.profit是N的收益值，N uprofit是它的收益上限， N. weight 是它对应的重量。bbnode类型如程序17-5中的定义，各节点按其uprofit值从最大堆中取出。

程序17-7使用了类Knap, 它类似于回溯法中的类Knap（见程序16-5）。两个Knap版本中数据成员之间的区别见程序17-7: 1) bestp 不再是一个成员； 2) bestx 是一个指向 int 的新成员。新增成员的作用是：当且仅当物品j包含在最优解中时， bestx[j]=1。函数AddLiveNode用于将新的bbnode类型的活节点插入子集树中，同时将HeapNode类型的活节点插入到最大堆中。这个函数与装箱问题（见程序17-6）中的对应函数非常类似，因此相应的代码被省略。

程序 17-7 0/1背包问题的最大收益分枝定界算法

```
template<class Tw, class Tp>
Tp Knap<Tw, Tp>::MaxProfitKnapsack()
{// 返回背包最优装载的收益
// bestx[i] = 1 当且仅当物品 i 属于最优装载
// 使用最大收益分枝定界算法
// 定义一个最多可容纳 1000 个活节点的最大堆
H = new MaxHeap<HeapNode<Tp, Tw> > (1000);
// 为 bestx 分配空间
bestx = new int [n+1];
// 初始化层 1
int i = 1;
E = 0;
cw = cp = 0;
Tp bestp = 0; // 目前的最优收益
Tp up = Bound(1); // 在根为 E 的子树中最大可能的收益
// 搜索子集空间树
while (i != n+1) { // 不是叶子
// 检查左孩子
Tw wt = cw + w[i];
if (wt <= c) {// 可行的左孩子
if (cp+p[i] > bestp) bestp = cp+p[i];
AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);}
up = Bound(i+1);
// 检查右孩子
if (up >= bestp) // 右孩子有希望
AddLiveNode(up, cp, cw, false, i+1);
```

```
// 取下一个 E-节点
HeapNode<Tp, Tw> N;
H->DeleteMax(N); // 不能为空
E = N.ptr;
cw = N.weight;
cp = N.profit;
up = N.uprofit;
i = N.level;
}
// 沿着从 E-节点 E 到根的路径构造 bestx[]
for (int j = n; j > 0; j--) {
bestx[j] = E-> L C h i l d ;
E = E-> p a r e n t ;
}
return cp;
}
```

函数 MaxProfitKnapsack 在子集树中执行最大收益分枝定界搜索。函数假定所有的物品都是按收益密度值的顺序排列，可以使用类似于程序 16-9 中回溯算法所使用的预处理代码来完成这种排序。函数 MaxProfitKnapsack 首先初始化活节点的最大堆，并使用一个数组 bestx 来记录最优解。由于需要不断地利用收益密度来排序，物品的索引值会随之变化，因此必须将 MaxProfitKnapsack 所生成的结果映射回初始时的物品索引。可以用 Q 的 ID 域来实现上述映射（见程序 16-9）。

在函数 MaxProfitKnapSack 中，E 是当前 E-节点，cw 是节点对应的重量， cp 是收益值，up 是以 E 为根的子树中任一节点的收益值上限。while 循环一直执行到一个叶节点成为 E-节点为止。由于最大堆中的任何剩余节点都不可能具有超过当前叶节点的收益值，因此当前叶即对应了一个最优解。可以从叶返回到根来确定这个最优解。

MaxProfitKnapsack 中 while 循环的结构很类似于程序 17-6 的 while 循环。首先，检验 E-节点左孩子的可行性，如它是可行的，则将它加入子集树及活节点队列（即最大堆）；仅当节点右孩子的 Bound 值指明有可能找到一个最优解时才将右孩子加入子集树和队列中。

5.2.3 最大完备子图

4.2.3 节完备子图问题的解空间树也是一个子集树，故可以使用与装箱问题、背包问题相同的最大收益分枝定界方法来求解这种问题。解空间树中的节点类型为 bbnode，而最大优先队列中元素的类型则是 CliqueNode。CliqueNode 有如下域：cn（该节点对应的完备子图中的顶点数目），un（该节点的子树中任意叶节点所对应的完备子图的最大尺寸），level（节点在解空间树中的层），cn（当且仅当该节点是其父节点的左孩子时， cn 为 1），ptr（指向节点在解空间树中的位置）。un 的值等于 cn+n-leve+1。因为根据 un 和 cn（或 level）可以求出 level（或 cn），所以可以去掉 cn 或 level 域。当从最大优先队列中选取元素时，选取的是具有最大 un 值的元素。在程序 17-8 中，CliqueNode 包含了所有的三个域：cn，un 和 level，这样便于尝试为 un 赋予不同的含义。函数 AddCliqueNode 用于向生成的子树和最大堆中加入节点，由于其代码非常类似于装箱和背包问题中的对应函数，故将它略去。

函数 BBMaxClique 在解空间树中执行最大收益分枝定界搜索，树的根作为初始的 E-节点，该节点并没有在所构造的树中明确存储。对于这个节点来说，其 cn 值（E-节点对应的完备子图的大小）为 0，因为还没有任何顶点被加入完备子图中。E-节点的层由变量 i 指示，它的初值为 1，对应于树的根节点。当前所找到的最大完备子图的大小保存在 bestn 中。

在 while 循环中，不断展开 E-节点直到一个叶节点变成 E-节点。对于叶节点，u n = c n。由于所有其他节点的 un 值都小于等于当前叶节点对应的 un 值，所以它们不可能产生更大的完备子图，因此最大完备子图已经找到。沿着生成的树中从叶节点到根的路径，即可构造出这个最大完备子图。

为了展开一个非叶 E-节点，应首先检查它的左孩子，如果左孩子对应的顶点 i 与当前 E-节点所包含的所有顶点之间都有一条边，则 i 被加入当前的完备子图之中。为了检查左孩子的可行性，可以沿着从 E-节点到根的路径，判断哪些顶点包含在 E-节点之中，同时检查这些顶点中每个顶点是否都存在一条到 i 的边。如果左孩子是可行的，则把它加入到最大优先队列和正在构造的树中。下一步，如果右孩子的子树中包含最大完备子图对应的叶节点，则把右孩子也加入。

由于每个图都有一个最大完备子图，因此从堆中删除节点时，不需要检验堆是否为空。仅当到达一个可行的叶节点时，w h i l e 循环终止。

程序 17-8 最大完备子图问题的分枝定界算法

```
int AdjacencyGraph::BBMaxClique(int bestx[])
{// 寻找一个最大完备子图的最大收益分枝定界程序
// 定义一个最多可容纳 1 0 0 0 个活节点的最大堆
MaxHeap<CliqueNode> H(1000);
// 初始化层 1
bbnode *E = 0; // 当前的 E-节点为根
int i = 1, // E-节点的层
cn = 0, // 完备子图的大小
bestn = 0; // 目前最大完备子图的大小
// 搜索子集空间树
while (i != n+1) {// 不是叶子
// 在当前完备子图中检查顶点 i 是否与其它顶点相连
bool OK = true;
bbnode *B = E;
for (int j = i - 1; j > 0; B = B->parent, j--)
if (B->LChild && a[i][j] == NoEdge) {
OK = false;
b r e a k ; }
if (OK) {// 左孩子可行
if (cn + 1 > bestn) bestn = cn + 1;
AddCliqueNode(H, cn+1, cn+n-i+1, i+1, E, true);}
if (cn + n - i >= bestn)
// 右孩子有希望
AddCliqueNode(H, cn, cn+n-i, i+1, E, false);
// 取下一个 E-节点
CliqueNode N;
H.DeleteMax(N); // 不能为空
E = N.ptr;
cn = N.cn;
i = N.level;
}
// 沿着从 E 到根的路径构造 bestx[]
```

```
for (int j = n; j > 0; j--) {
bestx[j] = E-> L C h i l d ;
E = E-> p a r e n t ;
}
return bestn;
}
```

5.2.4 旅行商问题

旅行商问题的介绍见 4.2.4 节，它的解空间是一个排列树。与在子集树中进行最大收益和最小耗费分枝定界搜索类似，该问题有两种实现的方法。第一种是只使用一个优先队列，队列中的每个元素中都包含到达根的路径。另一种是保留一个部分解空间树和一个优先队列，优先队列中的元素并不包含到达根的路径。本节只实现前一种方法。

由于我们要寻找的是最小耗费的旅行路径，因此可以使用最小耗费分枝定界法。在实现过程中，使用一个最小优先队列来记录活节点，队列中每个节点的类型为 MinHeapNode。每个节点包括如下区域： x（从 1 到 n 的整数排列，其中 x[0]=1），s（一个整数，使得从排列树的根节点到当前节点的路径定义了旅行路径的前缀 x[0:s], 而剩余待访问的节点是 x[s+1:n-1]），cc（旅行路径前缀，即解空间树中从根节点到当前节点的耗费），lcost（该节点子树中任意叶节点中的最小耗费）， rcost（从顶点 x[s:n-1]出发的所有边的最小耗费之和）。当类型为 MinHeapNode(T)的数据被转换成为类型 T 时，其结果即为 lcost 的值。分枝定界算法的代码见程序 17-9。

程序 17-9 首先生成一个容量为 1000 的最小堆，用来表示活节点的最小优先队列。活节点按其 lcost 值从最小堆中取出。接下来，计算有向图中从每个顶点出发的边中耗费最小的边所具有的耗费 MinOut。如果某些顶点没有出边，则有向图中没有旅行路径，搜索终止。如果所有的顶点都有出边，则可以启动最小耗费分枝定界搜索。根的孩子（图 16-5 的节点 B）作为第一个 E-节点，在此节点上，所生成的旅行路径前缀只有一个顶点 1，因此 s=0, x[0]=1, x[1:n-1]是剩余的顶点（即顶点 2， 3，.， n）。旅行路径前缀 1 的开销为 0，即 cc=0，并且，$rcost=\sum_{i=1}^{n} MinOut[i]$。在程序中，bestc 给出了当前能找到的最少的耗费值。初始时，由于没有找到任何旅行路径，因此 bestc 的值被设为 NoEdge。

程序 17-9 旅行商问题的最小耗费分枝定界算法

```
template<class T>
T AdjacencyWDigraph<T>::BBTSP(int v[])
{// 旅行商问题的最小耗费分枝定界算法
// 定义一个最多可容纳 1000 个活节点的最小堆
MinHeap<MinHeapNode<T> > H(1000);
T *MinOut = new T [n+1];
// 计算 MinOut[i] = 离开顶点 i 的最小耗费边的耗费
T MinSum = 0; // 离开顶点 i 的最小耗费边的数目
for (int i = 1; i <= n; i++) {
T Min = NoEdge;
for (int j = 1; j <= n; j++)
if (a[i][j] != NoEdge &&
(a[i][j] < Min || Min == NoEdge))
Min = a[i][j];
if (Min == NoEdge) return NoEdge; // 此路不通
MinOut[i] = Min;
```

```
MinSum += Min;
}
// 把 E-节点初始化为树根
MinHeapNode<T> E;
E.x = new int [n];
for (i = 0; i < n; i++)
E.x[i] = i + 1;
E.s = 0; // 局部旅行路径为 x [ 1 : 0 ]
E.cc = 0; // 其耗费为 0
E.rcost = MinSum;
T bestc = NoEdge; // 目前没有找到旅行路径
// 搜索排列树
while (E.s < n - 1) {// 不是叶子
if (E.s == n - 2) {// 叶子的父节点
// 通过添加两条边来完成旅行
// 检查新的旅行路径是不是更好
if (a[E.x[n-2]][E.x[n-1]] != NoEdge && a[E.x[n-1]][1] != NoEdge && (E.cc +
a[E.x[n-2]][E.x[n-1]] + a[E.x[n-1]][1] < bestc || bestc == NoEdge)) {
// 找到更优的旅行路径
bestc = E.cc + a[E.x[n-2]][E.x[n-1]] + a[E.x[n-1]][1];
E.cc = bestc;
E.lcost = bestc;
E . s + + ;
H . I n s e r t ( E ) ; }
else delete [] E.x;}
else {// 产生孩子
for (int i = E.s + 1; i < n; i++)
if (a[E.x[E.s]][E.x[i]] != NoEdge) {
// 可行的孩子，限定了路径的耗费
T cc = E.cc + a[E.x[E.s]][E.x[i]];
T rcost = E.rcost - MinOut[E.x[E.s]];
T b = cc + rcost; //下限
if (b < bestc || bestc == NoEdge) {
// 子树可能有更好的叶子
// 把根保存到最大堆中
MinHeapNode<T> N;
N.x = new int [n];
for (int j = 0; j < n; j++)
N.x[j] = E.x[j];
N.x[E.s+1] = E.x[i];
N.x[i] = E.x[E.s+1];
N.cc = cc;
N.s = E.s + 1;
N.lcost = b;
```

```
N.rcost = rcost;
H.Insert(N);}
} // 结束可行的孩子
delete [] E.x;} // 对本节点的处理结束
try {H.DeleteMin(E);} // 取下一个 E-节点
catch (OutOfBounds) {break;} // 没有未处理的节点
}
if (bestc == NoEdge) return NoEdge; // 没有旅行路径
// 将最优路径复制到 v[1:n] 中
for (i = 0; i < n; i++)
v[i+1] = E.x[i];
while (true) {// 释放最小堆中的所有节点
delete [] E.x;
try {H.DeleteMin(E);}
catch (OutOfBounds) {break;}
}
return bestc;
}
```

while 循环不断地展开 E-节点，直到找到一个叶节点。当 s＝n - 1 时即可说明找到了一个叶节点。旅行路径前缀是 x [0 : n - 1]，这个前缀中包含了有向图中所有的 n 个顶点。因此 s＝n - 1 的活节点即为一个叶节点。由于算法本身的性质，在叶节点上 lcost 和 cc 恰好等于叶节点对应的旅行路径的耗费。由于所有剩余的活节点的 lcost 值都大于等于从最小堆中取出的第一个叶节点的 lcost 值，所以它们并不能帮助我们找到更好的叶节点，因此，当某个叶节点成为 E-节点后，搜索过程即终止。

while 循环体被分别按两种情况处理，一种是处理 s＝n - 2 的 E-节点，这时，E-节点是某个单独叶节点的父节点。如果这个叶节点对应的是一个可行的旅行路径，并且此旅行路径的耗费小于当前所能找到的最小耗费，则此叶节点被插入最小堆中，否则叶节点被删除，并开始处理下一个 E-节点。

其余的 E-节点都放在 while 循环的第二种情况中处理。首先，为每个 E-节点生成它的两个子节点，由于每个 E-节点代表着一条可行的路径 x [0 : s]，因此当且仅当＜x[s],x[i]＞ 是有向图的边且 x [i]是路径 x [s＋1 : n - 1]上的顶点时，它的子节点可行。对于每个可行的孩子节点，将边<x[s],x[i]＞ 的耗费加上 E.cc 即可得到此孩子节点的路径前缀(x [0 : s]，x[i]) 的耗费 c c。由于每个包含此前缀的旅行路径都必须包含离开每个剩余顶点的出边，因此任何叶节点对应的耗费都不可能小于 cc 加上离开各剩余顶点的出边耗费的最小值之和，因而可以把这个下限值作为 E-节点所生成孩子的 lcost 值。如果新生成孩子的 lcost 值小于目前找到的最优旅行路径的耗费 b e s t c，则把新生成的孩子加入活节点队列（即最小堆）中。

如果有向图没有旅行路径，程序 1 7 - 9 返回 N o E d g e; 否则，返回最优旅行路径的耗费，而最优旅行路径的顶点序列存储在数组 v 中。

5.2.5 电路板排列

电路板排列问题（ 1 6 . 2 . 5 节）的解空间是一棵排列树，可以在此树中进行最小耗费分枝定界搜索来找到一个最小密度的电路板排列。我们使用一个最小优先队列，其中元素的类型为 B o a r d N o d e，代表活节点。B o a r d N o d e 类型的对象包含如下域： x（电路板的排列），s（电路板 x[1:s]) 依次放置在位置 1 到 s 上），c d（电路板排列 x [1 : s]的密度，其中包括了到达 x[s] 右边的连线），n o w(now[j] 是排列 x[1:s] 中包含 j 的电路板的数目）。当一个 BoardNode

类型的对象转换为整型时，其结果即为对象的 cd 值。代码见程序 17-10。

程序 17-10 电路板排列问题的最小耗费分枝定界算法

```
int BBArrangeBoards(int **B, int n, int m, int* &bestx)
{// 最小耗费分枝定界算法, m 个插槽, n 块板
MinHeap<BoardNode> H(1000); // 容纳活节点
// 初始化第一个 E 节点、total 和 bestd
BoardNode E;
E.x = new int [n+1];
E.s = 0; // 局部排列为 E.x[1:s]
E.cd = 0; // E.x[1:s]的密度
E.now = new int [m+1];
int *total = new int [m+1];
// now[i] = x[1:s]中含插槽 i 的板的数目
// total[i] = 含插槽 i 的板的总数目
for (int i = 1; i <= m; i++) {
total[i] = 0;
E.now[i] = 0;
}
for (i = 1; i <= n; i++) {
E.x[i] = i; // 排列为 1 2 3 4 5 . . . n
for (int j = 1; j <= m; j++)
total[j] += B[i][j]; // 含插槽 j 的板
}
int bestd = m + 1; / /目前的最优密度
bestx = 0; // 空指针
do {// 扩展 E 节点
if (E.s == n - 1) {// 仅有一个孩子
int ld = 0; // 最后一块板的局部密度
for (int j = 1; j <= m; j++)
ld += B[E.x[n]][j];
if (ld < bestd) {// 更优的排列
delete [] bestx;
bestx = E.x;
bestd = max(ld, E.cd);
}
else delete [] E.x;
delete [] E.now;}
else {// 生成 E-节点的孩子
for (int i = E.s + 1; i <= n; i++) {
BoardNode N;
N.now = new int [m+1];
for (int j = 1; j <= m; j++)
// 在新板中对插槽计数
N.now[j] = E.now[j] + B[E.x[i]][j];
```

```
int ld = 0; // 新板的局部密度
for (j = 1; j <= m; j++)
if (N.now[j] > 0 && total[j] != N.now[j]) ld++;
N.cd = max(ld, E.cd);
if (N.cd < bestd) {// 可能会引向更好的叶子
N.x = new int [n+1];
N.s = E.s + 1;
for (int j = 1; j <= n; j++)
N.x[j] = E.x[j];
N.x[N.s] = E.x[i];
N.x[i] = E.x[N.s];
H.Insert(N);}
else delete [] N.now;}
delete [] E.x;} // 处理完当前 E-节点
try {H.DeleteMin(E);} // 下一个 E-节点
catch (OutOfBounds) {return bestd;} //没有 E-节点
} while (E.cd < bestd);
// 释放最小堆中的所有节点
do {delete [] E.x;
delete [] E.now;
try {H.DeleteMin(E);}
catch (...) {break;}
} while (true);
return bestd;
}
```

程序 17-10 首先初始化 E-节点为排列树的根，此节点中没有任何电路板，因此有 s=0, cd=0,now[i]=0（1≤i≤n），x 是整数 1 到 n 的任意排列。接着，程序生成一个整型数组 total，其中 total[i] 的值为包含 i 的电路板的数目。目前能找到的最优的电路板排列记录在数组 bestx 中，对应的密度存储在 bestd 中。程序中使用一个 do-while 循环来检查每一个 E-节点，在每次循环的尾部，将从最小堆中选出具有最小 cd 值的节点作为下一个 E-节点。如果某个 E-节点的 cd 值大于等于 bestd，则任何剩余的活节点都不能使我们找到密度小于 bestd 的电路板排列，因此算法终止。

do-while 循环分两种情况处理 E-节点，第一种是处理 s=n-1 时的情况，此种情况下，有 n-1 个电路板被放置好， E-节点即解空间树中的某个叶节点的父节点。节点对应的密度会被计算出来，如果需要，bested 和 bestx 将被更新。在第二种情况中，E-节点有两个或更多的孩子。每当一个孩子节点 N 生成时，它对应的部分排列(x[1:s+1])的密度 N.cd 就会被计算出来，如果 N.cd<bestd,则 N 被存放在最小优先队列中；如果 N.cd≥bestd，则它的子树中的所有叶节点对应的密度都满足 density≥bestd，这就意味着不会有优于 bestx 的排列。

练习

3. 在程序 17-4 中增加代码，将指向由函数 AddLiveNode 生成的节点的指针存储在一个链表队列中。MaxLoading 必须利用这些指针信息在程序终止之前删除所有生成的节点。

4. 本节所使用的 AddLiveNode 函数直到程序终止前才删除所生成的节点。实际上，没有活动孩子且不产生叶节点的那些节点都可以被立即删除。类似地，在第 n 层节点中，若节点没有重量为 bestw 的孩子，则可以立即删除该节点。讨论怎样尽快删除不需要的节点。描述实现这

种方法时所涉及的时间/空间变化。你推荐使用上述方法吗？

5. 在程序 17-6 中，定义一个 bestw 来记录目前生成的可行节点所对应的重量的最大值。修改程序 17-6，使得如果活节点的重量大于等于 bestw，则将它加入子集树及最大堆中。此外，还必须增加初始化和更新 bestw 的代码。

6. 只使用一个最大优先队列，来实现用最大收益分枝定界方法求解货箱装船问题，即不要使用程序 17-6 中所用到的部分解空间树，而在每个优先队列的节点中都加入通向根节点的路径信息。

7. 修改程序 17-6，把删除 bbnode 类型和 HeapNode 类型节点的任务放在程序结尾处。

8. 只使用一个最大优先队列，利用最大收益分枝定界法求解 0/1 背包问题，即不必保存一个部分解空间树，所有优先队列中的节点都记录着通往根节点的路径。

9. 修改程序 17-7，使得删除 bbnode 和 HeapNode 类型的节点的任务放在程序的结尾处执行。

10. 1) 程序 17-8 中，若右孩子的 un 值大于等于 bestn，则将它加入最大堆中，如果将条件设为 un>bestn，程序能否正确执行呢？为什么？

2) 程序是否将 un≥bestn 的左孩子加入最大堆中？

3) 修改程序，使得只将 un>bestn 的节点加入到最大堆和生成的解空间树中。

11. 考察最大完备子图问题的解空间树。对于任意层(第 i 层)的子树中的节点 x，令 MinDegree(x) 为 x 所包含的顶点的度的最小值。

1) 证明任何以 x 为根的子树的叶节点都不可能表示一个尺寸超过 X.un = min{X.cn+n-i+1,MinDegree(X)+1}的完备子图。

2) 使用以上 X.un 的定义重写 BBMaxClique。

3) 比较两种 BBMaxClique 版本在运行时间及产生解空间树节点的数目上的不同。

12. 只使用最大优先队列，实现最大完备子图问题的最大收益分枝定界算法。即：不必保存一个部分解空间树，而在每一个最大优先队的节点内包含通向根的路径。

13. 修改程序 17-8，使得删除 bbnode 和 CliqueNode 类型的节点的工作放在程序结尾处执行。

14. 修改程序 17-9，使得 s=n-2 的节点不进入优先队列，并且，将当前最优排列放在数组 bestp 中。当下一个 E-节点的 lcostbestc 时，算法终止。

15. 使用指向父节点的指针来实现部分解空间树，并使用包含 lcost，cc，rcost 和 ptr（指向解空间树中对应节点的指针）域的优先队列来实现程序 17-9。

16. 写出用 FIFO 分枝定界方法求解电路板排列问题的代码。代码必须输出最优电路板排列的排列次序及对应的密度。使用合适的数据来测试代码的正确性。

17. 用 FIFO 分枝定界方法来搜索一种电路板的排列，使得最长的网组的长度最小（参见 4 章练习 17）。

18. 使用最小耗费分枝定界法来完成练习 17。

19. 用最小耗费分枝定界算法求解 4 章练习 18 的顶点覆盖问题。

20. 用最大收益分枝定界算法求解 4 章练习 19 的简易最大切割问题。

21. 用最小耗费分枝定界算法求解 4 章练习 20 的机器设计问题。

22. 用最小耗费分枝定界算法求解 4 章练习 21 的网络设计问题。

23. 用 FIFO 分枝定界算法求解 4 章练习 22 的 n-皇后放置问题。

* 24. 用 FIFO 分枝定界完成 4 章练习 23。

* 25. 用 FIFO 分枝定界完成 4 章练习 24。

* 26. 用 FIFO 分枝定界完成 4 章练习 25。

* 27. 用最小耗费分枝定界完成 4 章练习 23。

* 28. 用最小耗费分枝定界完成 4 章练习 2 4。
* 29. 用最小耗费分枝定界完成 4 章练习 2 5。
* 30. 用任意的分枝定界方法完成 4 章的练习 2 5。在本练习中，必须把增加活节点的函数以及选择下一个 E-节点的函数作为函数的参数。

（ 说明：本资料是根据《数据结构、算法与应用》（美，Sartaj Sahni 著）一书第 13-17 章编辑、改写的。考虑到因特网传输速度等因素，大部分插图和公式不得不被删除。对于内容不连贯之处，请网友或读者参阅该书，敬请原谅。 ）