第14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）。

14.1　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14-1　[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 $B1a$，另一组为$B1b$。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14-2 [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$2n-3$。程序2-26和2-27是另外两种方法，前者需要进行$2n-2$次比较，后者最多需要进行$2n-2$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， $n\leq 2$，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$A1$和$A2$。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， $c(n)$=2$c(n/2)$+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14-3 [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

$$C(i,j)=\sum_{k=1}^{n}A(i,k)*B(k,j),\quad 1\leqslant i\leqslant n,\quad 1\leqslant j\leqslant n \tag{14-1}$$

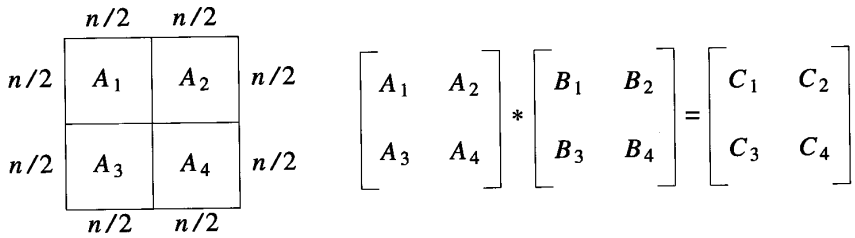
假如每一个$C(i, j)$ 都用此公式计算，则计算C所需要的操作次数为$n^3m+n^2(n-1)a$，其中m表示一次乘法，a 表示一次加法或减法。

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，如

图14-1a 所示。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，$1 \le i \le 4$。矩阵乘积的结果见图14-1b。



a) b)

图14-1 把一个矩阵划分成几个小矩阵

a) 把A分为4个小矩阵 b) A*B＝C

可以利用公式（14-1）来证明以下公式：

$$C_1 = A_1B_1 + A_2B_3 \tag{14-2}$$

$$C_2 = A_1B_2 + A_2B_4 \tag{14-3}$$

$$C_3 = A_3B_1 + A_4B_3 \tag{14-4}$$

$$C_4 = A_3B_2 + A_4B_4 \tag{14-5}$$

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$\Theta(n^3)$，此复杂性与程序2-24直接使用公式（14-1）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$D, E, \cdots, J$，它们分别定义为：

$$D = A_1(B_2 - B_4)$$

$$E = A_4(B_3 - B_1)$$

$$F = (A_3 + A_4)B_1$$

$$G = (A_1 + A_2)B_4$$

$$H = (A_3 - A_1)(B_1 + B_2)$$

$$I = (A_2 - A_4)(B_3 + B_4)$$

$$J = (A_1 + A_4)(B_1 + B_4)$$

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，方法如下：

$$C_1 = E + I + J - G$$

$$C_2 = D + G$$

$$C_3 = E + F$$

$$C_4 = D + H + J - F$$

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：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

因为n>1，所以将A、B两矩阵分别划分为4个小矩阵，如图14-1a 所示。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～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：

$$C_1 = 8 - 30 + 65 - 24 = 19$$
$$C_2 = -2 + 24 = 22$$
$$C_3 = 8 + 35 = 43$$
$$C_4 = -2 + 22 + 65 - 35 = 50$$

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（14-1），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$n \geq 8$的矩阵乘法，而对于$n<8$的矩阵乘法则直接利用公式（14-1）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$n \geq 16$的情况，对于n<16的矩阵相乘，直接利用公式（ 14-1）。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），t 的递归表达式如下：

$$t(n) = \begin{cases} d & n \leqslant k \\ 7t(n/2) + cn^2 & n > k \end{cases} \qquad (14\text{-}6)$$

这里cn^2 表示完成18次$n/2 \times n/2$阶矩阵加/减法以及把大小为n 的矩阵分割成小矩阵所需要

的时间。用迭代方法计算，可得$t(n)=\Theta(n\log_2^7)$。因为$\log_2^7 \approx 2.81$，所以与直接计算方法的复杂性$\Theta(n^3)$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14-2）和归并排序方法（14.3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14-4 [金块问题] 用例14-2的算法寻找8个金块中最轻和最重金块的工作可以用图14-2中的二叉树来表示。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14-2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。

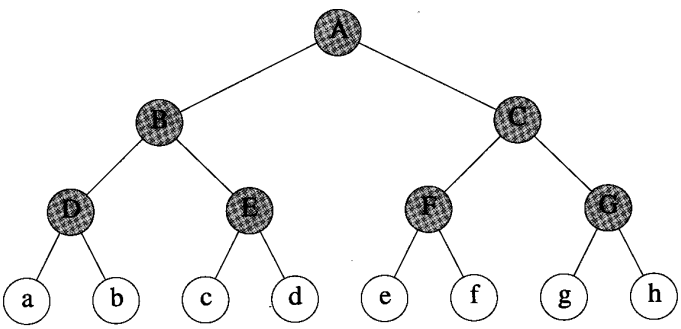


图14-2 找出8个金块中最轻和最重的金块

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/2ù-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false
  // 特殊情形：n <= 1
  if (n < 1) return false;
  if (n == 1) {Min = Max = 0;
               return true;}

  //对Min 和Max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[Max]进行比较
    // 将较小者与 w[Min]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14-1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14-1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$\Theta(n^3)$。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$\Theta(n^{\log_2^7})$。

*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$\Theta(m^{2.81})$。给出以n 为变量的运行时间表达式。

14.2 应用

14.2.1 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14-3给出$k \leq 2$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。

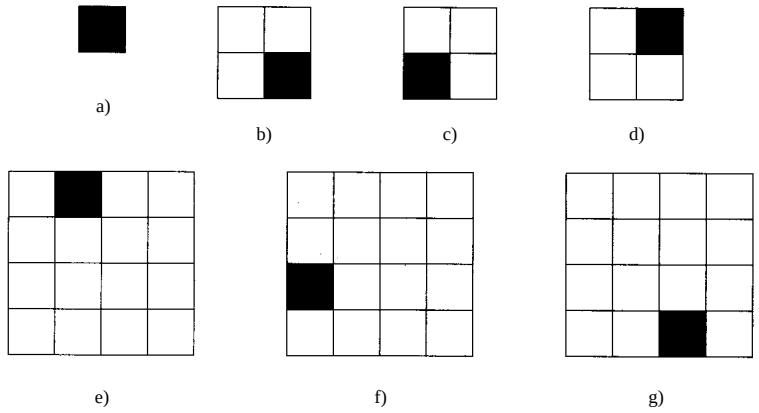


图14-3 残缺棋盘

a) k=0 b) k=1 c) k=1 d) k=1 e) k=2 f) k=2 g) k=2

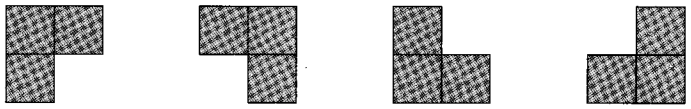


图14-4 不同方向的三格板

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

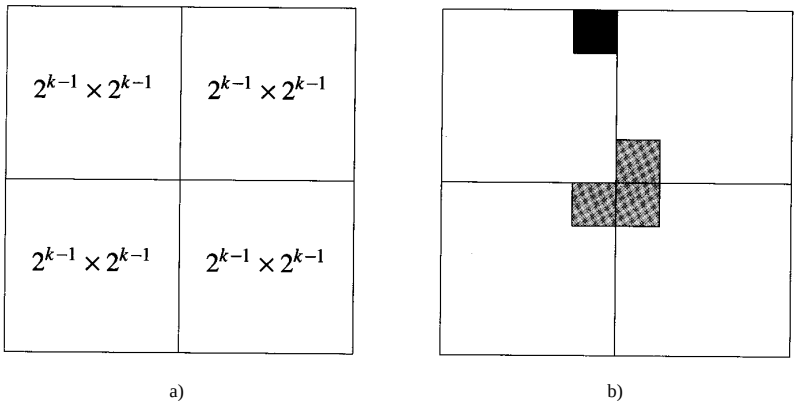


图14-5 分割$2^k \times 2^k$ 棋盘

a) 划分方法 b) 放置三格板

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：

$$t(k)=\begin{cases} d & k=0 \\ 4t(k-1)+c & k>0 \end{cases} \qquad (14\text{-}7)$$

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$t(k)=\Theta(4^k)=\Theta$（所需的三格板的数目）。由于必须花费至少$\Theta(1)$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。

14.2.2 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 $\Theta(n^2)$。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14-5 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[10,4]和[6,3,8,2,5,7]将被排序。排序结果分别为[4,10]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14-6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
       sort(A,i);
       sort(B,j);
       merge(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14-6所示）在最坏情况下所需花费的时间，则有以下递推公式：

$$t(n)=\begin{cases} d & n<k \\ t(n/k)+t(n-n/k)+cn & n\geqslant k \end{cases}$$

其中c 和d 为常数。当$n/k \approx n-n/k$ 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。

在$t(n)$ 的递推公式中，取k=2,可得到如下递推公式：

$$t(n)=\begin{cases} d & n\leqslant 1 \\ t(\lfloor n/2\rfloor)+t(\lceil n/2\rceil)+cn & n>1 \end{cases}$$

由于上面递推公式中存在两个取整运算，因此计算该递推公式比较困难。如果仅考虑 n 为2的幂，递推公式可简化为：

$$t(n)=\begin{cases} d & n\leqslant 1 \\ 2t(n/2)+cn & n>1 \end{cases}$$

可以用迭代方法来计算这一递推方式，结果为 $t(n)=\Theta(n\log n)$。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。$t(n)=\Theta(n\log n)$ 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 $t(n)=\Theta(n\log n)$。

1. 将图14-6改写为C++代码

图14-6中 k=2的排序方法被称为归并排序（ merge sort），或更精确地说是二路归并排序（two-way merge sort）。下面根据图 14-6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 chain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 14-6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14-6的改进版见图14-7。

```
template<class T>
MergeSort( 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];
}
```

2. 自然归并排序

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$\lceil \log_2 n \rceil$趟归并。因此自然归并排序将在$\Theta(n)$的时间内完成排序。而程序14-3将花费$\Theta(n \log n)$的时间。

14.2.3 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 pivot
  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 $\Theta(n^2)$。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 $\Theta(n\log n)$。令人吃惊的是，快速排序的平均复杂性也是$\Theta(n\log n)$。

定理14-1 快速排序的平均复杂性为$\Theta(n\log n)$。

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$n\leq 1$时，$t(n)\leq d$，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 $t(s)$, $t(n\text{-}s\text{-}1)$。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，故可以得到下面的递归表达式：

$$t(n)\leqslant cn+\frac{1}{n}\sum_{s=0}^{n-1}[t(s)+t(n-s-1)]$$

可将上式化简为：

$$t(n)=cn+\frac{2}{n}\sum_{s=0}^{n-1}t(s)\leqslant cn+\frac{4d}{n}+\frac{2}{n}\sum_{s=2}^{n-1}t(s) \tag{14-8}$$

如对（14-8）式中的n 使用归纳法，可得到$t(n)\leq kn\log_e n$，其中$n>1$且$k=2(c+d)$，$e\sim 2.718$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$t(2)\leq 2c+2d\leq kn\log_e 2$。在归纳假设部分，假定$t(n)\leq kn\log_e n$（当$2\leq n<m$ 时，m 是任意一个比2大的整数），然后需证明$t(m)\leq km\log_e m$。根据公式（14-8）和归纳假设，可以得到：

$$t(m)\leqslant cm+\frac{4d}{m}+\frac{2}{m}\sum_{s=2}^{m-1}t(s)\leqslant cm+\frac{4d}{m}+\frac{2k}{m}\sum_{s=2}^{m-1}s\log_e s \tag{14-9}$$

为了进一步证明的需要，提供以下事实：

- $s\log_e s$ 是s 的一个递增函数
- $\displaystyle\int_2^m s\log_e s\,ds<\frac{m^2\log_e m}{2}-\frac{m^2}{4}$

利用以上事实和公式（14-9），可以得到：

$$t(m)<cm+\frac{4d}{m}+\frac{2k}{m}\int_2^m s\log_e s\,ds<cm+\frac{4d}{m}+\frac{2k}{m}\left[\frac{m^2\log_e m}{2}-\frac{m^2}{4}\right]$$

$$=cm+\frac{4d}{m}+km\log_e m-\frac{km}{2}<km\log_e m$$

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。

n	插入排序	堆 排 序	归并排序	快速排序
10	0.000068	0.000126	0.000108	0.000081
20	0.000151	0.000234	0.000222	0.000148
30	0.000267	0.000360	0.000323	0.000224
40	0.000416	0.000502	0.000434	0.000304
50	0.000594	0.000646	0.000541	0.000385
60	0.000809	0.000791	0.000646	0.000468
70	0.001057	0.000946	0.000855	0.000552
80	0.001340	0.001103	0.000971	0.000638
90	0.001626	0.001260	0.001090	0.000726
100	0.001984	0.001423	0.001216	0.000814
200	0.006593	0.003140	0.002498	0.001733
300	0.015934	0.004950	0.004194	0.002693
400	0.026923	0.006593	0.005495	0.003688
500	0.041758	0.008791	0.007143	0.004696
600	0.060440	0.010989	0.008242	0.005495
700	0.080220	0.012637	0.009890	0.006593
800	0.106044	0.015385	0.012088	0.008242
900	0.132418	0.017033	0.012637	0.008791
1000	0.164835	0.019231	0.014835	0.009890

时间(s)

图14-11 各种排序算法的平均时间

图14-11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14-11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14-11中的数据对于比较各种排序算法是很有用的。当 $n \leq 100$时，用图14-12中的曲线来显示这些数据。

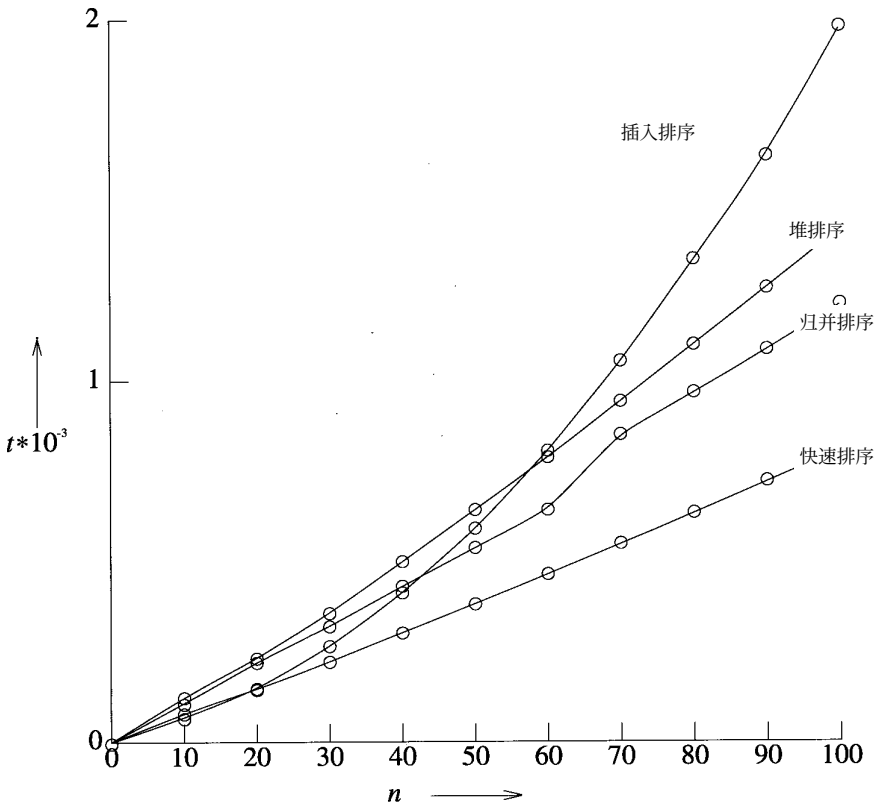


图14-12 各排序算法平均时间的曲线图

如图14-12所示，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）。

14.2.4 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 $k=\lceil n/2 \rceil$。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pivot
    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$\Theta(n^2)$，此时*left* 总是为空，而且第k个元素总是位于*right*，如果*left*和*right*总是同样大小或者相差不超过一个元素，那么可以得到以下递归表达式：

$$t(n) \leqslant \begin{cases} d & n \leqslant 1 \\ t(\lfloor n/2 \rfloor) + cn & n > 1 \end{cases} \tag{14-10}$$

如果假定n 是2的幂，则可以取消公式（14-10）中的向下取整操作符。通过使用迭代方法，可以得到$t(n)=\Theta(n)$。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 $\Theta(n)$。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14-6 [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14-2 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$n \geq 90$时，有$\max\{|lefe|, |right|\} \leq 7n/8$。

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

证明 这个定理的证明留作练习23。

根据定理14-2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：

$$t(n)=\begin{cases} cn\log n & n<90 \\ t(\lceil n/9\rceil)+t(\lfloor 7n/8\rfloor)+cn & n\geq 90 \end{cases} \qquad (14\text{-}11)$$

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≥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≥1时有$t(n)$≤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。

14.2.5 距离最近的点对

给定n个点(x_i, y_i) (1≤i≤n)，要求找出其中距离最近的两个点。两点间的距离公式如下：

$$\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$$

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$n(n\text{-}1)/2$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$\Theta(n^2)$。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$\lfloor n/2\rfloor$。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
```

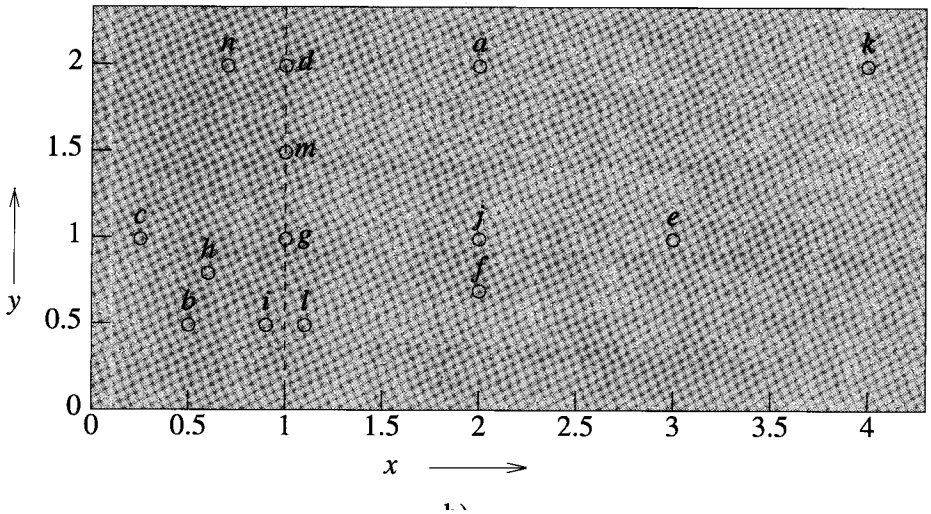
图14-13 寻找最近的点对

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14-8 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*。

点	*a*	*b*	*c*	*d*	*e*	*f*	*g*
x_i	2	0.5	0.25	1	3	2	1
y_i	2	0.5	1	2	1	0.7	1
点	*h*	*i*	*j*	*k*	*l*	*m*	*n*
x_i	0.6	0.9	2	4	1.1	1	0.7
y_i	0.8	0.5	1	2	0.5	1.5	2

a)



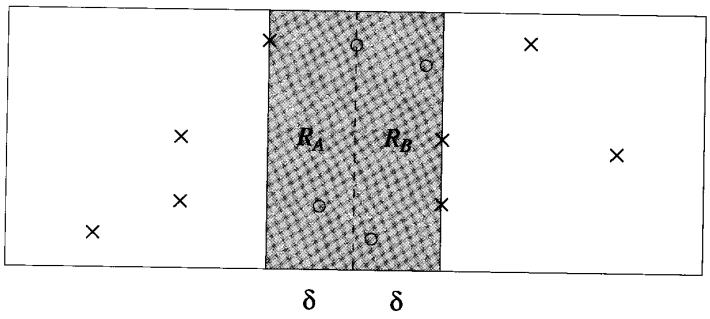
b)

图14-14 14个点

a) 14个点 b) 点的分布

设δ 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 δ 小。则每一个点距垂线的距离必小于 δ，这样，就可以淘汰那些距垂线距离≥ δ 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 2δ。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δ。

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* *A*, *q* *B*且*p*, *q* 的距离小于δ，则*p* R_A，*q* 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* - δ＜*q.y*＜*p.y*+δ 的*q* 点，看是否存在与*p* 间距小于δ 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δ×2δ 阴影内的点逐个与*p* 配对，以判断*p* 是否是距离小于δ 的第三类点。这个δ×2δ 区域被称为是*p* 的比较区（comparing region）。



×表示被淘汰的点

图14-15 淘汰距分割线很远的点

例14-9 考察例14-8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δ=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≥δ。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δ，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。

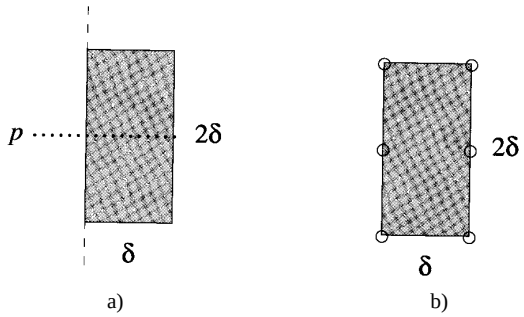


图14-16 p的比较区

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;
     }
   MergeSort(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
   close(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]);
       return;}

   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;
        return;}
    if (d2 <= d3) {a = X[l+1];
                   b = X[r];
                   d = d2;}
    else {a = X[l];
          b = X[r];
          d = d3;}
    return;}

  //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
  close(X,Z,Y,l,m,a,b,d);
  float dr;
  Point1 ar, br;
  close(X,Z,Y,m+1,r,ar,br,dr);

  //  (a,b) 是两者中较近的点对
  if (dr < d) {a = ar;
               b = br;
               d = dr;}

  Merge(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close（见程序14-11）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=(1+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14-5）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 δ 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<δ。对每一个Z[i]，在2δ×δ 区域内所检查的点如图14-17所示。由于在每个2δ×δ 子区域内的点至少相距 δ。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。

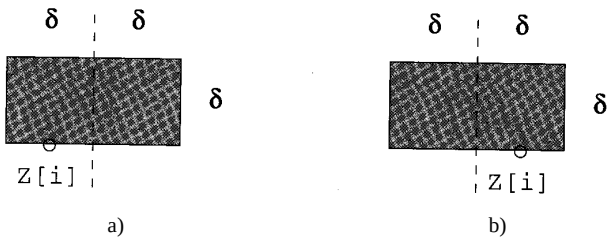


图14-17 与Z[i] 配对的点的区域

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≥4时，需花费$\Theta(n)$ 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$t(\lfloor n/2 \rfloor)$，因而可得到如下递归式：

$$t(n)=\begin{cases} d & n<4 \\ t(\lfloor n/2 \rfloor)+t(\lceil n/2 \rceil)+cn & n\geqslant 4 \end{cases}$$

这个递归式与归并排序的递归式完全一样，其结果为 $t(n)=\Theta(n\log n)$。另外，函数closest还需耗时$\Theta(n\log n)$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$\Theta(n\log n)$。

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$\Theta(n^2)$。

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$\Theta(n\log n)$。

17. 证明 $\int s\log_e s\,ds = \frac{s^2\log_e s}{2} - \frac{s^2}{4}$，并利用该结果证明 $\int_2^m s\log_e s\,ds < \frac{m^2\log_e m}{2} - \frac{m^2}{4}$ 。

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$n \geq 1$时有$t(n) \leq 72cn$。

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。

14.3 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，形式如下：

$$t(n)=\begin{cases} t(1) & n=1 \\ a*t(n/b)+g(n) & n>1 \end{cases} \qquad (14\text{-}12)$$

其中a, b为已知常数。需假设$t(1)$已知，且n为b的幂(即$n=b^k$)。利用迭代原理，可以证明：

$$t(n)=n^{\log_b a}[t(1)+f(n)] \qquad (14\text{-}13)$$

其中 $f(n)=\sum_{j=1}^{k}h(b^j)$　　$h(n)=g(n)/n^{\log_b a}$。

图14-18列出了不同$h(n)$时的$f(n)$值。根据这张表，在分析分而治之算法时可以很容易得到$t(n)$的值。

考察一些例子。当n是2的幂时，折半搜索的递归式为：

$$t(n)=\begin{cases} t(1) & n=1 \\ t(n/2)+c & n>1 \end{cases}$$

将这个递归式与公式(14-11)比较，可看出a=1，b=2，$g(n)$=c，因而$\log_b a$=0，$h(n)=g(n)/n^{\log_b^a}=c=c(\log n)^0=\Theta((\log n)^0)$。根据图14-18可知，$f(n)=\Theta(\log n)$，因而$t(n)=n^{\log_b^a}=(c+\Theta(\log n))=\Theta(\log n)$。

对于归并排序，有a=2，b=2，$g(n)$=cn。因此，$\log_b a$=1，$h(n)=g(n)/n=c=\Theta((\log n)^0)$。所以$f(n)=\Theta(\log n)$且$t(n)=n(t(1)+\Theta(\log n))=\Theta(n\log n)$。

考察另外一个例子，其递归表达式如下：

$$t(n)=7t(n/2)+18n^2,\ n\geq 2\text{且为2的幂}$$

该表达式为k=1，c=18时Strassen矩阵乘法的递归表达式（公式(14-6)）。因a=7，b=2，$g(n)=18n^2$，所以$\log_b a=\log_2 7\approx 2.81$，$h(n)=18n^2/n^{\log_2 7}=18n^{2-\log_2 7}=O(n^r)$，其中$r=2-\log_2 7<0$，因而$f(n)=O(1)$，

$t(n)=n^{\log_2 7}(t(1)+O(1))=\Theta(n^{\log_2 7})$，$t(1)$假设为常数。

最后一个例子，考察下面的递归式：

$$t(n)=9t(n/3)+4n^6，\quad n\geq 3且为3的幂$$

将这个递归式与式(14-11)比较，可得到$a=9$, $b=3$, $g(n)=4n^6$，因而$\log_b a=2$，$h(n)=4n^6/n^2=4n^4=\Omega(n^4)$。根据图14-12可知$f(n)=\Theta(h(n))=\Theta(n^4)$，因而$t(n)=n^2(t(1)+\Theta(n^4))=\Theta(n^6)$，其中$t(1)$假设为常数。

$h(n)$	$f(n)$
$O(n^r), r<0$	$O(1)$
$\Theta((\log n)^i), i\geq 0$	$\Theta(((\log n)^{i+1})/(i+1))$
$\Omega(n^r), r>0$	$\Theta(h(n))$

图14-18 $f(n)$与$h(n)$的对应关系

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) $t(n)=10t(n/3)+11n$, $n\geq 3$且为3的幂。
2) $t(n)=10t(n/3)+11n^5$, $n\geq 3$且为3的幂。
3) $t(n)=27t(n/3)+11n^3$, $n\geq 3$且为3的幂。
4) $t(n)=64t(n/4)+10n^3\log^2 n$, $n\geq 4$且为4的幂。
5) $t(n)=9t(n/2)+n^2 2^n$, $n\geq 2$且为2的幂。
6) $t(n)=3t(n/8)+n^2 2^n \log n$, $n\geq 8$且为8的幂。
7) $t(n)=128t(n/2)+6n$, $n\geq 2$且为2的幂。
8) $t(n)=128t(n/2)+6n^8$, $n\geq 2$且为2的幂。
9) $t(n)=128t(n/2)+2^n/n$, $n\geq 2$且为2的幂。
10) $t(n)=128t(n/2)+\log^3 n$, $n\geq 2$且为2的幂。

14.4 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$\Theta(n^3)$）。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。

14.4.1 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：

$$(a, b, c, d) \rightarrow (a-1, b, c, d+1) \quad (A\text{中元素大于}B\text{中元素})$$
$$(a, b, c, d) \rightarrow (a-1, b, c+1, d) \quad (A\text{中元素小于}B\text{中元素})$$

- A中元素与C中元素进行比较，可能的转换为：

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

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

14.4.2 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$\lceil m/2 \rceil$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$\lceil \log n! \rceil$。

因为$n! \geq \lceil n/2 \rceil^{\lceil n/2 \rceil - 1}$，$\log n! \geq (n/2-1)\log(n/2) = \Omega(n\log n)$。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])。

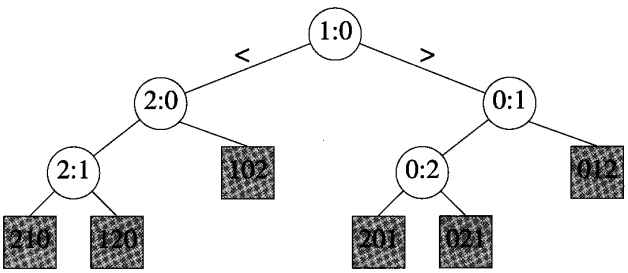


图14-19 n=3 时InsertionSort 的决策树

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$n!\geq\lceil n/2\rceil^{\lceil n/2\rceil+1}$。

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（inversion number）。

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。