第三章　分 治 算 法

§1. 算法基本思想

先来分析折半搜索算法

程序 3-1-1 折半搜索

```
BiFind(a,n)
   //在数组 a[1..n]中搜索 x，数组中的元素满足 a[1] ≤ a[2] ≤ … ≤ a[n]。
   //如果找到 x，则返回所在位置（数组元素的下标），否则返回 －1
   global a[1..n], n;
   integer left,right,middle;
   left:=1; right:=n;
   while left ≤ right do
     middle:=(left+right)/2;
     if x=a[middle] then return(middle); end{if}
     if x>a[middle] then left:=middle+1;
     else right:=middle-1;
     end{if}
   end{while}
   return(－1); //未找到 x
end{BiFind}
```

while 的每次循环（最后一次除外）都将以减半的比例缩小搜索范围，所以，该循环在最坏的情况下需要执行 $\Theta(\log n)$ 次。由于每次循环需耗时 $\Theta(1)$，因此在最坏情况下，总的时间复杂度为 $\Theta(\log n)$。

折半搜索算法贯彻一个思想，即分治法。当人们要解决一个输入规模，比如 n，很大的问题时，往往会想到将该问题分解。比如将这 n 个输入分成 k 个不同的子集。如果能得到 k 个不同的可独立求解的子问题，而且在求出这些子问题的解之后，还可以找到适当的方法把它们的解合并成整个问题的解，那么复杂的难以解决的问题就可以得到解决。这种将整个问题分解成若干个小问题来处理的方法称为分治法。一般来说，被分解出来的子问题应与原问题具有相同的类型，这样便于算法实现（多数情况下采用递归算法）。如果得到的子问题相对来说还较大，则再用分治法，直到产生出不用再分解就可求解的子问题为止。人们考虑

和使用较多的是 k=2 的情形，即将整个问题二分。以下用 A[1..n]来表示 n 个输入，用 DiCo(p,q)表示用分治法处理输入为 A[p..q]的问题。

分治法控制流程

```
DiCo(p,q)
    global n, A[1..n];
    integer m,p,q;  // 1≤p≤q≤n
    if Small(p,q) then return(Sol(p,q));
    else m:=Divide(p,q); // p≤m<q
       return(Combine(DiCo(p,m),DiCo(m+1,q)));
    end{if}
  end{DiCo}
```

这里，Small(p,q)是一个布尔值函数，用以判断输入规模为 q-p+1 的问题是否小到无需进一步细分即可容易求解。若是，则调用能直接计算此规模下子问题解的函数 Sol(p,q). 而 Divide(p,q)是分割函数，决定分割点；Combine(x,y)是解的合成函数。如果假定所分成的两个问题的输入规模大致相等，则 DiCo 总的计算时间可用下面的递归关系来估计：

$$T(n)=\begin{cases} g(n) \quad , & if\ n\ is\ small \\ 2T(n/2)+f(n) \quad , & f(n)\ is\ the\ time\ of\ Combine \end{cases} \qquad (3.1.1)$$

例 3.1.1 求 n 元数组中的最大和最小元素

最容易想到的算法是直接比较算法：将数组的第 1 个元素分别赋给两个临时变量：fmax:=A[1]; fmin:=A[1]; 然后从数组的第 2 个元素 A[2]开始直到第 n 个元素逐个与 fmax 和 fmin 比较，在每次比较中，如果 A[i] > fmax，则用 A[i]的值替换 fmax 的值；如果 A[i] < fmin，则用 A[i]的值替换 fmin 的值；否则保持 fmax（fmin）的值不变。这样在程序结束时的 fmax、fmin 的值就分别是数组的最大值和最小值。这个算法在最好、最坏情况下，元素的比较次数都是 2(n-1)，而平均比较次数也为 2(n-1)。 如果将上面的比较过程修改为：

从数组的第 2 个元素 A[2]开始直到第 n 个元素，每个 A[i]都是首先与 fmax 比较，如果 A[i]>fmax,则用 A[i]的值替换 fmax 的值；否则才将 A[i]与 fmin 比较，如果 A[i] < fmin，则用 A[i]的值替换 fmin 的值。

这样的算法在最好、最坏情况下使用的比较次数分别是 n-1 和 2(n-1),而平均比较次数是 3(n-1)/2,因为在比较过程中，将有一半的几率出现 A[i]>fmax 情况。

如果采用分治的思想，可以构造算法，其时间复杂度在最坏情况下和平均用时均为 3n/2-2:

程序 3-1-2 **递归求最大最小值算法伪代码**

```
MaxMin（i,j,fmax,fmin）//A[1:n]是 n 个元素的数组，参数 i，j
  //是整数，1≤i≤j≤n，使用该过程将数组 A[i..j]中的最大最小元
  //分别赋给 fmax 和 fmin。
  global n, A[1..n];
  integer i, j;
  if i=j then
    fmax:=A[i]; fmin:=A[i]; //子数组 A[i..j]中只有一个元素
  elif i=j-1 then //子数组 A[i..j]中只有两个元素
    if A[i]<A[j] then
      fmin:=A[i]; fmax:=A[j];
    else fmin:=A[j]; fmax:=A[i];
    end{if}
  else
    mid:=⌊(i+j)/2⌋; //子数组 A[i..j]中的元素多于两个
    MaxMin(i, mid, lmax, lmin);
    MaxMin(mid+1, j, rmax, rmin);
    fmax:=max(lmax, rmax);
    fmin:=main(lmin, rmin);
  end{if}
end{Maxmin}
```

如果用 T(n)来表示 MaxMin 所用的元素比较次数，则上述递归算法导出一个递归关系式：

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+2 & n>2\end{cases} \qquad (3.1.2)$$

当 n 是 2 的方幂时，设 $n=2^k$，有

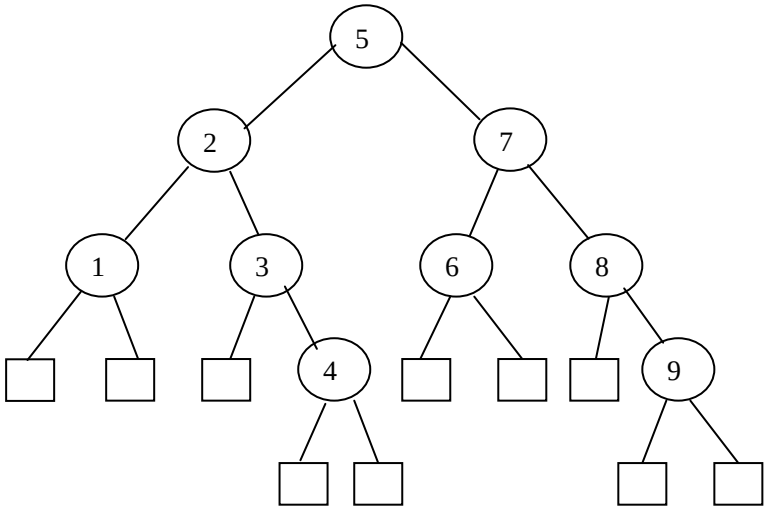
$$\begin{aligned}T(n)&=2T(n/2)+2\\&=2(2T(n/4)+2)+2\\&=4T(n/4)+4+2\end{aligned}$$

$$
\begin{aligned}
&\cdots \quad \cdots \quad \cdots \\
&= 2^{k-1}T(2) + \sum_{1 \le i \le k-1} 2^i \\
&= 2^{k-1} + 2^k - 2 \\
&= 3n/2 - 2
\end{aligned}
$$

无论是最好、最坏、还是平均情况，MaxMin 所用的比较次数都是 3n/2-2，比前面提到的算法(在最坏情况下)节省了 25%的时间。实际上，任何一种以元素比较为基础的找最大最小元素的算法，其元素比较次数的下界是$\lceil 3n/2 \rceil - 2$。从这种意义上来说，MaxMin 算法是最优的。然而，由于需要$\lfloor \log n \rfloor + 1$级的递归，每次递归调用需要将 i，j，fmax，fmin 和返回地址的值保留到栈中，需要多占用内存空间。而且由于这些值出入栈时也会带来时间开销，特别当 A 中元素的比较次数和整数 i 与 j 的比较次数相差无几时，递归求最大最小值算法未必比直接求最大最小值算法效率高。

例 3.1.2 搜索算法的时间下界

分析上节提到的折半搜索算法，我们已经知道其时间复杂度是$O(\log n)$。事实上，我们可以用一个二元比较树来分析折半搜索算法的时间复杂性。以下是 n=9 的二元比较树：



N=9 情况下，折半搜索的二元比较树

由图可见，当 x 在数组 A 中时，算法在圆形结点结束；不在 A 中时，算法在方形结点结束。因为$2^3 \le 9 < 2^4$，所以比较树的叶结点的深度都是 3 或 4。因

而元素比较的最多次数为 4。一般地有：

当$n \in \left[2^{k-1},2^{k}\right)$时，成功的折半搜索至多做$k$次比较，而不成功的折半搜索或者做 k-1 次比较，或者做 k 次比较。

现在假设数组 A[1..n]满足：A[1]< A[2]< …< A[n]。要搜索元素 x 是否在 A 中。*如果只允许进行元素间的比较而不允许对它们进行其它的运算，则所设计的算法称为以比较为基础的算法。*

任何以比较为基础的搜索算法的执行过程都可以用一棵二元比较树来描述。每次可能的比较用一个内结点表示，对应于不成功的结果有一个外结点（叶结点）与之对应。线性搜索和折半搜索的二元比较树如下：

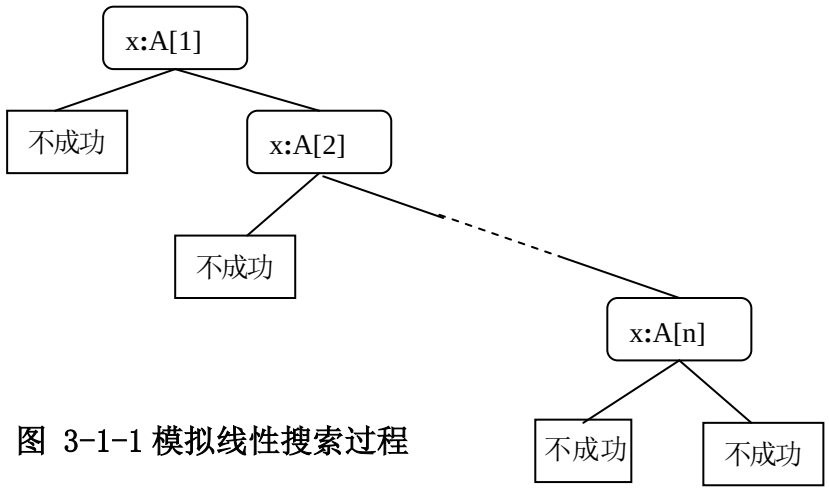


图 3-1-1 模拟线性搜索过程

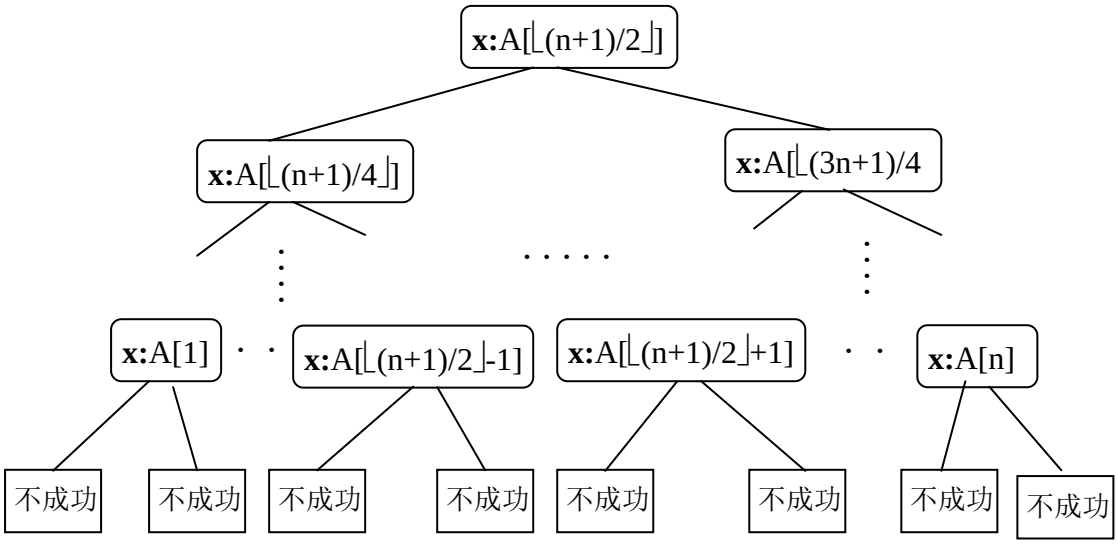


图 3-1-2 模拟折半搜索过程

定理 3.1.1 设数组 A[1..n]的元素满足 A[1]< A[2]< …< A[n]。则以比较为基础，判断$x \in A[1..n]$的任何算法，在最坏情况下所需的最少比较次数 F(n)不

小于⌈log(n+1)⌉。

证明 通过考察模拟求解搜索问题的各种可能算法的比较树可知，F(n)不大于树中由根到叶子的最长路径的距离，即树的高度。也就是说，在最坏情况下每种搜索算法的比较次数都是比较树的高度 k。对于每个二叉比较树，必有 n 个内结点与 x 在 A 中的 n 种可能的情况相对应。而每个内结点的深度都不会超过该树的高度减 1，即 k-1。因而，内结点的个数不超过2^k-1，即$n \le 2^k-1$。由此得 F(n)= k ≥⌈log(n+1)⌉。 证毕

定理 3.1.1 说明，任何一种以比较为基础的搜索算法，其最坏情况下所用时间不可能低于$\Theta(\log n)$。不可能存在其最坏情况下时间比折半搜索数量级（阶）还低的算法。事实上，折半搜索所产生的比较树的所有叶结点都在相邻的两个层次上，而且这样的二叉树使得比较树的高度最低。因此，折半搜索是解决搜索问题在最坏情况下的最优算法。

§2. 排序算法

问题：已知 n 个元素的数组 A[1..n]，将 A 中元素按不降顺序排列。

● 归并排序算法

先来分析插入排序算法

程序 3-2-1 向有序数组插入元素

```
Insert(a, n, x)
  //向数组 a[1..n]中插入元素 x
  //假定 a 的大小超过 n
  int i;
  for i from n by -1 to 1 do
    if x<a[i] then
      a[i+1]:=a[i];
  end{if}
  end{for}
  a[i+1]:=x;
end{Insert}
```

程序 3-2-2 插入排序

```
InSort(a, n)
    //对 a[1..n]进行排序
    for i from 2 to n do
        t:=a[i];
        Insert(a, i-1, t);
    end{for}
  end{InSort}
```

将上述两个算法合并在一起，得到下述插入排序算法

程序 3-2-3 插入排序算法

```
InSort(a, n)
  for i from 2 to n do
    t:=a[i];
    integer j;
    for j from i-1 by -1 to 1 do
      if t<a[j] then a[j+1]:=a[j]; end{if}
    end{for}
    a[j+1]:=t;
  end{for}
 end{InSort}
```

内层的 for 循环语句可能执行 i 次（$i=1,2,\cdots,n-1$），因此最坏情况下的时间是

$$\sum_{1\le i\le n-1} i = n(n-1)/2 = \Theta(n^2)$$

在这个算法中，大部分的时间都用在挪动元素上，随着已经排好顺序的数组的增长，被挪动的元素的个数也在增加，而且在整个过程中，很多元素不止一次被挪动。以下程序从某种程度上减少了这种浪费。这一算法的基本思想是采用分治的方法，将要排序的数组分成两部分，先对每部分进行排序，然后再将两部分已经排好序的子数组的元素按照从小到大的顺序逐一摆放在一个新的数组中。这一过程也许需要多次分解和组合，是一个递归过程。

程序 3-2-4 归并排序主程序伪代码

```
MergeSort(low, high) // A[low .. high]是一个全程数组，含有
   // high-low+1 个待排序的元素。
   integer low, high;
   if low < high  then
     mid:=⌊(low+high)/2⌋ //求当前数组的分割点
     MergeSort(low, mid)  //将第一子数组排序
     MergeSort(mid+1, high)  //将第二子数组排序
     Merge(low, mid, high)  //归并两个已经排序的子数组
   end{if}
 end{MergeSort}
```

这里我们使用了辅助程序 Merge:

程序 3-2-5 合并过程伪代码

```
Merge(low, mid, high) //已知全程数组 A[low .. high], 其由
   //两部分已经排好序的子数组构成：A[low .. mid]和 A[mid+1 .. high]。
   //本程序的任务是将这两部分子数组合并成一个整体排好序的数组，
   //再存于数组 A[low .. high].
   integer h, i, j, k, low, mid, high;
   global A[low .. high];
   local B[low .. high]; //借用临时数组 B
   h:=low, i:=low, j:=mid+1;
   // h, j 是拣取游标, i 是向 B 存放元素的游标
   while h≤mid and j≤high do  //当两个集合都没有取尽时
      if A[h]≤A[j] then B[i]:=A[h], h:=h+1;
      else B[i]:=A[j], j:=j+1;
      end{if}
      i:＝i+1;
   end{while}
   if h>mid then
   //当第一子组元素被取尽，而第二组元素未被取尽时
     for k from j to high do
        B[i]:=A[k]; i:=i+1;
     end{for}
   else
   //当第二子组元素被取尽，而第一组元素未被取尽时
     for k from h to mid do
       B[i]:=A[k]; i:=i+1;
     end{for}
   end{if}
   //将临时数组 B 中元素再赋给数组 A
   for k from low to high do
     A[k]:=B[k];
   end{for}
 end{Merge}
```

可见，归并排序由分解与合并两部分组成，整个过程可用两棵树表示出来（参见本章附页“归并排序树”）。如果用 T(n)表示归并排序所用的时间，并假定合并

过程所用时间与 n 成正比：cn，其中 c 是一个正数，则有

$$T(n)=\begin{cases}a & n=1\\ 2T(n/2)+cn & n>1\end{cases} \qquad (3.2.1)$$

其中，a 是一个常数。若 n 是 2 的方幂：$n=2^k$，直接推导可得

$$\begin{aligned}T(n)&=2(2T(n/4)+cn/2)+cn\\&=4T(n/4)+2cn\\&\cdots\cdots\\&=2^kT(1)+kcn\\&=an+cn\log n\end{aligned}$$

对于一般的整数 n，我们可以假定 $2^k<n\le 2^{k+1}$，于是，由 $T(2^k)\le T(n)\le T(2^{k+1})$，得 $T(n)=O(n\log n)$。

- **以比较为基础的排序时间的下界**

类似于估计以比较为基础的搜索算法的时间下界，也可以用树来模拟以比较为基础的排序算法，在此我们考虑最坏情况下的时间下限，并假定数组中的元素互不相同。在树的内部结点上，算法执行一次比较，并根据比较的结果移向它的某一个子结点。由于每两个元素 A[i]和 A[j]的比较只有两种可能：A[i]<A[j]或 A[j]<A[i]，所以这颗树是二叉树。当 A[i]<A[j]时进入左分支，当 A[j]<A[i]进入右分支。各个叶结点表示算法终止。从根到叶结点的每一条路径与一种唯一的排列相对应。由于 n 个不同元素的不同排列共有 n！个，因此比较树有 n！个外部结点（参看本章附页“排序比较树”）。直接观察可知，由根到外结点路径即描述了该外结点所代表的排列生成过程，路径的长度即是经历的比较次数。因此，比较树中最长路径的长度（其是比较树的高）即是算法在最坏情况下所做的比较次数。要求出所有以比较为基础的排序算法在最坏情况下的时间下界，只需求出这些算法所对应的比较树的最小高度。如果比较树的高是 k，则该二叉树的外结点至多是 2^k 个。于是，$n!\le 2^k$。注意到

$$n!\ge n(n-1)\cdots(\lceil n/2\rceil)\ge(n/2)^{n/2-1} \qquad (3.2.2)$$

因而

$$k\ge(n/2-1)\log(n/2)=\Theta(n\log n) \qquad (3.2.3)$$

$T(n)\ge\Theta(n\log n)$，即 $\Theta(n\log n)$ 是以比较为基础的排序算法在最坏情况下的时间下界。

从上式看出，归并排序是时间复杂度最低的排序算法（以比较为基础）。然而，仔细观察可以发现，归并排序有两个地方值得商榷：一是分解直到只有一个元素。事实上，当元素比较少时，直接进行排序，比如用插入排序算法，比起进一步分拆、合并手续要快得多。因为，在这种情况下，大量的时间都花在调用分解、合并函数上。所以，在归并排序算法中，对于*归并起点的规模* 应该有适当的限制，即加 Small(p,q)判断。二是辅助数组 B 的借用，虽然不可避免，但应该采用另一种方式，以避免数组 A 中的元素的频繁换位。为此，我们可以采用链表（该表中存储的是数组元素的下标），将数组 A 中的元素位置变动转化成链表值的变化。例如

LINK:

↓k=0 ↓k =2

位置	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
数据		50	10	25	30	15	70	35	55
指针	2	0	3	4	1	7	0	8	6

假定前 4 个已经排好，后 4 个也已经排好，链表如上，头指针分别是 2,5。下面是在此基础上将两个排好的子链表连接起来的过程。

q=2; r=5

```
A[2]<A[5]:  LINK[0]:=2; k:=i(=2);  i:=LINK[i](=3);
A[3]>A[5]:  LINK[k]:=5; k:=j(=5);  j:=LINK[j](=7);
A[3]<A[7]:  LINK[k]:=3; k:=i(=3);  i:=LINK[i](=4);
A[4]<A[7]:  LINK[k]:=4; k:=i(=4);  i:=LINK[i](=1);
A[1]>A[7]:  LINK[k]:=7; k:=j(=7);  j:=LINK[7](=8);
A[1]<A[8]:  LINK[k]:=8; k:=i(=1);  i:=LINK[i](=0);
                        k:=j(=8);  j:=LINK[j](=6)
                        k:=j(=6);  j:=LINK[j](=0)
```

Q = (10, 25, 30, 50)
↑i=2 ↑i=4

R = (15, 35, 55, 70)
↑j=5

k: →2→5→3→4→7→1→8→6

指针移动过程

```
if A[i]≤A[j] then
  LINK[k]:=i; k:=i; i:=LINK[i];
else
  LINK[k]:=j; k:=j; j:=LINK[j];
end{if}
```

一般规则

3-2-6 使用链接的归并排序算法

```
MergeSortL(low, high, p) // Link 是全程数组 A[low..high]
     //的下标表，p 指示这个表的开始处。利用 Link 将 A 按非降顺序排列。
    global A[low..high]; Link[low..high];
    if high-low+1<16 then //设定子问题的最小规模 Small
      InSort(A,Link, low,high,p);
    else mid:=⌊(low+high)/2⌋;
      MergeSortL(low,mid,q); //返回 q 表
      MergeSortL(mid+1,high,r); //返回 r 表
      MergeL(q,r,p); 将表 q 和 r 合并成表 p
    end{if}
  end{MergeSortL}
```

其中，合并程序 MergeL 是合并函数 Merge 的改进：

程序 3-2-7 使用连接表的合并程序

```
MergeL(q,r,p) // 由链接表 q 和 r 构造新的连接表。p、q、r 是
    //全程数组 Link[0..n]中两个表指针，这两个链表指出被划分的
    //两个子组的地址排序，而 p 指针指出两组归并后的地址排序。
    global n,A[1..n], Link[0..n];
    local integer i, j, k;
    i:=q; j:=r; k:=0; // 初始化，新表在 Link[0]处开始
    while i≠0 and j≠0 do //当两个表皆非空时
      if A[i]≤A[j] then
        Link[k]:=i; k:=i; i:=Link[i]; //加一个新元素到此表
      else Link[k]:=j; k:=j; j:=Link[j];
      end{if}
    end{while}
    if i=0 then
      Link[k]:=j;
    else Link[k]:=i;
    end{if}
    p:=Link[0];
  end{MergeL}
```

例 3.2.1 考虑将数组 A=[50,10,25,30,15,70,35,55]按非降次序排列问题，采用改进的归并算法。这里主要说明链接表在合并函数 MergeL 被调用时的变化过程（参看本章附页“归并链接表”）。

● 快速排序算法

另一个利用分治法排序的例子是*快速排序*，是由计算机科学家 C.A.R.Hoare 提出的。基本策略是：将数组 A[1..n]分解成两个子数组 B[1..p]和 B[p+1..n]，使得 B[1..p]中的元素均不大于 B[p+1..n]中的元素，然后分别对这里的两个数组中的元素进行排序（非降的），最后再把两个排好序的数组接起来即可。一般的分解是从 A 中选定一个元素，然后将 A 中的所有元素同这个元素比较，小于或等于这个元素的放在一个子组里，大于这个元素的放在另一个子组里。这个过程叫做划分。

程序 3-2-8 划分程序伪代码

```
Partition(m,p) // 被划分的数组是 A[m,p-1]，
    //选定做划分元素的是 v:=A[m]。
    integer m, p, i;
    global A[m ..p-1];
    v:=A[m]; i:=m;
    loop
      loop i:=i+1; until A[i]>v; end{loop} //自左向右查
      loop p:=p-1; until A[p]≤v; end{loop} //自右向左查
      if i<p then
        Swap(A[i],A[p]); //交换 A[i]和 A[p]的位置
      else go to *;
      end{if}
    end{loop}
  *: A[m]:=A[p]; A[p]:= v; // 划分元素在位置 p
  end{Partition}
```

例 3.2.2 划分程序的执行情况： m=1, p=10 的情形

原数组：65 70 75 80 85 60 55 50 45 (＋∞)被划分成:(60, 45, 50, 55) , 65, (85, 80, 75, 70)（参看本章附页划分程序执行过程）。

程序 3-2-9 快速排序算法伪代码

```
QuickSort(p,q) //将数组 A[1..n]中的元素 A[p], A[p+1], ... , A[q]
   //按不降次序排列，并假定 A[n+1]是一个确定数，且大于 A[1..n]中所
   // 有的数。
   integer p,q;
   global n, A[1..n];
   if p<q then
      j:=q+1; Partition(p,j); // 划分后 j 成为划分元素的位置
      QuickSort(p,j-1);
      QuickSort(j+1,q);
   end{if}
end{QuickSort}
```

由前面关于以比较为基础的排序算法在最坏情况下的时间下界可知，快速排序算法在最坏情况下的时间复杂性应不低于$\Omega(n\log n)$。事实上，在快速算法中元素比较的次数，在最坏情况下是$O(n^2)$。这是因为，在 Partition(m,p)的每一次调用中，元素的比较次数至多是 p-m.把 QuickSort 过程按照划分来分层，则第一层只调用 Partition 一次，即 Partition(1,n+1)，涉及的元素为 n 个； 在第二层调用 Partition 两次，所涉及到的元素是 n-1 个，因为在第一层被选定的划分元素不在其中。若用 N_k表示第 k 层调用 Partition 时所涉及的元素总个数，则 $N_k \le n-k+1$。这样在第 k 层调用 Partition 时发生的元素比较次数应不大于 n-k.注意到 $1\le k\le n$，因此 QuickSort 算法在最坏情况下总的元素比较次数不超过 n(n-1)/2，即 QuickSort 最坏情况下的时间复杂度为$O(n^2)$。

为了得到时间复杂性的平均值，我们不妨假设

1. 参加排序的 n 个元素互不相同；
2. Partition 中的划分元素 v 是随机选取的。

以 $C_A(n)$记时间复杂性的平均值。在上述假设条件下，调用 Partition(m,p)时，所取划分元素 v 是 A[m, p-1]中第 $i(1\le i\le p-m)$小元素具有相等的概率，因而留下待排序的两个子组为 A[m..j-1]和 A[j+1..p-1]的概率是 1/(p-m)，$m\le j\le p-1$。由此得递归关系式

$$C_A(n)=n-1+\frac{1}{n}\sum_{1\le k\le n}(C_A(k-1)+C_A(n-k)) \qquad (3.2.4)$$

其中，$n-1$是 Partition 第一次被调用时所需要的元素比较次数，

$C_A(0)=C_A(1)=0$。将(3.2.4)式两端乘以 n 得

$$nC_A(n)=n(n-1)+2(C_A(0)+C_A(1)+\cdots+C_A(n-1)) \qquad (3.2.5)$$

用 n-1 替换(3.2.5)中的 n 得

$$(n-1)C_A(n-1)=(n-1)(n-2)+2(C_A(0)+C_A(1)+\cdots+C_A(n-2)) \qquad (3.2.6)$$

再用(3.2.5)减去(3.2.6)式得

$$\begin{aligned} C_A(n)/(n+1) &= C_A(n-1)/n+\frac{2(n-1)}{n(n+1)} \\ &\le C_A(n-1)/n+2/n \end{aligned} \qquad (3.2.7)$$

由递推关系式(3.2.7)，并注意到$C_A(0)=C_A(1)=0$，得

$$C_A(n)/(n+1)\le C_A(1)/2+2\sum_{2\le i\le n}1/i \qquad (3.2.8)$$

利用积分不等式

$$\sum_{2\le i\le n}1/i<\int_1^n\frac{dx}{x}=\ln n$$

得 $C_A(n)<2(n+1)\ln n=O(n\log n)$

看来快速排序与归并排序具有相同的平均时间复杂性。但是在实际表现中却是有所不同的。实验表明，快速排序一般要比归并排序效率更高些（参看《数据结构、算法与应用》，Sartaj Sahni 著，汪诗林 孙晓东 译 p.457）。

关于排序的算法我们已经接触了 5 种：冒泡排序、插入排序、选择排序、归并排序和快速排序，它们的时间复杂性列出如下：

算 法	最坏复杂性	平均复杂性
冒泡排序	n^2	n^2
插入排序	n^2	n^2
选择排序	n^2	n^2
快速排序	n^2	n log n
归并排序	n log n	n log n

§3.选 择 问 题

问题：已知 n 元数组 A[1..n]，试确定其中第 k 小的元素。

最容易想到的算法是采用一种排序算法先将数组按不降的次序排好，然后从排好序的数组中捡出第 k 个元素。这样的算法在最坏情况下时间复杂度是 $O(n\log n)$。实际上，我们可以设计出在最坏情况下的时间复杂度为 $O(n)$ 的算法。为此，考察上节提到的算法 Partition。假设在一次划分中，划分元素 v 处于第 j 个位置。如果 k<j，则要找的第 k 小元素在新数组 A[1..j-1]中，而且是 A[1..j-1]的第 k 小元素；如果 k＝j，则划分元素 v 即是要找的第 k 小元素；如果 k>j，则要找的第 k 小元素在新数组 A[j＋1..n]中，而且是 A[j＋1..n]的第 k-j 小元素。

程序 3-3-1 采用划分的选择算法

```
PartSelect(A, n, k) //在数组 A[1..n]中找第 k 小元素 t，并将其存
    //放于位置 k，即 A[k]=t。而剩下的元素按着以 t 为划分元素的划分
    //规则存放。再令 A[n+1]=+∞.
    integer n, k, m, r, j;
    m:=1; r:=n+1; A[n+1]:= +∞;
    loop
      j:=r;
      Partition(m,j);
      case
        k=j : return // 返回 j,当前数组的元素 A[j]是第 j 小元素
        k<j : r:=j; // j 是新的下标上界
        else : m:=j+1; //j+1 是新的下标下界
      end{case}
    end{loop}
end{PartSelect}
```

（参看本章附页 PartSelect 程序的执行过程）

这个算法在最坏情况下的时间复杂度是 $O(n^2)$。事实上，假定数组 A[1..n]中的元素互不相同，而且假定**划分元素是随机选取**的。注意，在每次调用 Partition(m, j)都要耗去 $O(j-m)$ 的时间。而下一次被划分的数组的元素个数至少比上一次减少 1。因而，从最初的划分中 m=1, j=n+1 开始，至多需要做 n-1 次划分。第一次划分，耗去时间至多为 n，第二次耗去时间至多是 n-1,… .所以，

PartSelect 在最坏情况下的时间复杂度为$O(n^2)$。但是，可以推出，PartSelect 的平均时间复杂度为$O(n)$。事实上，我们可以改进 PartSelect 算法，通过精心挑选划分元素 v，得到在最坏情况下的时间复杂度为$O(n)$的算法。

程序 3-3-2 改进的选择算法伪代码

```
Select(A, m, p, k) // 返回一个 i 值，使得 A[i]是 A[m..p]中第
   // k 小元素。r 是一个大于 1 的整数。
   global r;
   integer n, i, j;
   if p-m+1≤r then
     InSort(A, m, p);
     return(m+k-1);
   end{if}
   loop
      n:=p-m+1;
      for i to ⌊n/r⌋ do  //计算中间值
        InSort(A, m+(i-1)*r, m+i*r-1);
        //将中间值收集到 A[m..p]的前部:
        Swap(A[m+i-1], A[m+(i-1)*r+⌊r/2⌋-1]);
      end{for}
     j:=Select(A, m, m+⌊n/r⌋-1, ⌈⌊n/r⌋/2⌉);
     Swap(A[m], A[j]); //产生划分元素
     j:=p+1;
     Partition(m, j);
     case:
        j-m+1=k : return(j);
        j-m+1>k : p:=j-1;
        else   m:=j+1;
     end{case}
   end{loop}
end{Select}
```

这里，程序 Select 只在划分元素的选取上做了改进，其余部分沿用 PartSelect 的步骤。划分元素的选取方案是：取定正整数 r(>1),将原始数组按 r 个元素一段的原则分成⌊n/r⌋段（可能剩余 n-r*⌊n/r⌋个元素）。对每一段求取中

间元素，并把这⌊n/r⌋个中间元素搜集在数组 A[m..p]的前部（免去另开空间收存的操作）。现在调用程序

Select(A, m, m+⌊n/r⌋-1, ⌈⌊n/r⌋/2⌉)，

就产生了所要的划分元素。因为⌊n/r⌋一定小于 n，这样的递归过程是可行的。为了直接应用程序 PartSelect，将刚刚找到的划分元素放在数组 A[m..p]的首位。

我们以 r=5 来分析 Select 算法的时间复杂性。假设数组 A 中的元素都是互不相同的。由于每个具有 5 个元素的数组的中间值 u 是该数组的第 3 小元素，此数组至少有 3 个元素不大于 u;⌊n/5⌋个中间值中至少有⌈⌊n/5⌋/2⌉个不大于这些中间值的中间值 v。因而，在数组 A 中至少有

3*⌈⌊n/5⌋/2⌉ ≥ 1.5*⌊n/5⌋

个元素不大于 v。换句话说，A 中至多有

n-1.5*⌊n/5⌋= n-1.5*(n/5-e/5) ≤ 0.7n+1.2

个元素大于 v。同理，至多有 0.7n+1.2 个元素小于 v。这样，以 v 为划分元素所产生的新的数组至多有 0.7n+1.2 个元素。当 n≥24 时，0.7n+1.2≤0.75n=3n/4。

注意到程序 Select 中，从一层到下一层递归时，实际上相当于两次调用了 Select：一次体现在语句

j:=Select(A, m, m+⌊n/r⌋-1, ⌈⌊n/r⌋/2⌉);

另一次体现在 Partition(m, j)及后面的 case 语句组,其关键操作为Θ(n)次。主程序接着就要调用自身，执行规模不超过 3n/4 的选择问题。这两步涉及的数组规模分别是 n/5 和 ≤3n/4。程序中其它执行步的时间复杂度都至多是 n 的倍数。如果用 T(n)表示算法在数组长度为 n 的时间复杂度，则当 n≥24 时，有递归关系

$$T(n) \le T(n/5) + T(3n/4) + cn \qquad (3.3.1)$$

其中 c 是常数。从递推关系式(3.3.1)出发，用数学归纳法可以证明

$$T(n) \le 20cn \qquad (3.3.2)$$

所以，在最坏情况下，Select 算法的时间复杂度是 $O(n)$。

§4. 关于矩阵乘法

假定 A, B 都是 n×n 矩阵，它们的 i 行 j 列元素分别记为 A(i,j)和 B(i,j)。如果用 S 和 C 分别记 A+B 和 A*B, 则有

$$\begin{aligned} S(i,j) &= A(i,j) + B(i,j) \qquad 1 \le i,j \le n \\ C(i,j) &= \sum_{k=1}^{n} A(i,k) * B(k,j) \qquad 1 \le i,j \le n \end{aligned} \qquad (3.4.1)$$

可见，矩阵加法运算的时间复杂度是$\Theta(n^2)$，而矩阵乘法的时间复杂度是$\Theta(n^3)$。后者是因为求每个元素$C(i,j)$都需要 n 次乘法和 n-1 次加法运算，而 C 共有n^2个元素。

如果用分治法解决矩阵的乘法问题，可以设想将矩阵分块，然后用分块矩阵乘法完成原矩阵的乘法运算。不妨假定 n 是 2 的方幂，$n=2^k$，将矩阵 A 和 B 等分成四块，于是

$$A*B=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix}=\begin{bmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix}$$

其中

$$\begin{aligned}C_{11}&=A_{11}B_{11}+A_{12}B_{21} \qquad C_{12}=A_{11}B_{12}+A_{12}B_{22}\\ C_{21}&=A_{21}B_{11}+A_{22}B_{21} \qquad C_{22}=A_{21}B_{12}+A_{22}B_{22}\end{aligned} \tag{3.4.2}$$

如果用 T(n)记两个 n 阶矩阵相乘所用的时间，则有如下递归关系式：

$$T(n)=\begin{cases}b & n\le 2\\ 8T(n/2)+dn^2 & n>2\end{cases} \tag{3.4.3}$$

这是因为在计算 C 的块 C_{ij}时，需要计算 8 次 n/2 阶矩阵的乘法计算和 4 次 n/2 阶矩阵的加法计算，后者需要dn^2加法运算，这里 d 是一个常数。直接递推关系式(3.4.3)得

$$T(n)=bn^3/8+4d(n^2-16)/3$$

因为$b\neq 0$，所以$T(n)=\Theta(n^3)$。

虽然没有降低时间复杂度，但给我们一个启示。1969 年，斯特拉森(V.Strassen)发现了降低矩阵乘法时间复杂度的可能性。注意到，计算矩阵的加法比计算乘法的时间复杂度具有较低的阶$(n^2:n^3)$，而在用分块矩阵乘法时，既有矩阵加法又有矩阵乘法。如果能通过增加加法次数来减少乘法次数，则可能达到降低矩阵乘法的时间复杂度的目的。为此他设计了一个算法用以计算(3.4.2)式中的 C_{ij}，共用了 7 次乘法和 18 次加(减)法。令

$$\begin{aligned}P&=(A_{11}+A_{22})(B_{11}+B_{22}), \qquad T=(A_{11}+A_{12})B_{22}\\ Q&=(A_{21}+A_{22})B_{11}, \qquad U=(A_{21}-A_{11})(B_{11}+B_{12})\\ R&=A_{11}(B_{12}-B_{22}), \qquad V=(A_{12}-A_{22})(B_{21}+B_{22})\\ S&=A_{22}(B_{21}-B_{11})\end{aligned} \tag{3.4.4}$$

则
$$\begin{aligned} C_{11} &= P+S-T+V \\ C_{12} &= R+T \\ C_{21} &= Q+S \\ C_{22} &= P+R-Q+U \end{aligned} \tag{3.4.5}$$

由此得到的递推关系式为

$$T(n)=\begin{cases} b & n\le 2 \\ 7T(n/2)+an^2 & n>2 \end{cases} \tag{3.4.6}$$

直接推导可得

$$\begin{aligned} T(n) &= an^2(1+7/4+(7/4)^2+\cdots+(7/4)^{k-2})+7^{k-1}T(2) \\ &= an^2\left(\frac{16}{21}\left(\frac{7}{4}\right)^{\log n}-\frac{4}{3}\right)+\frac{b}{7}(7)^{\log n} \\ &= an^2\left(\frac{16}{21}(n)^{\log\frac{7}{4}}-\frac{4}{3}\right)+\frac{b}{7}(n)^{\log 7} \\ &= \left(\frac{16a}{21}+\frac{b}{7}\right)n^{\log 7}-\frac{4a}{3}n^2 \\ &= \Theta\left(n^{2.81}\right) \end{aligned} \tag{3.4.7}$$

从所得的结果看出，Strassen 算法的时间复杂度依赖于 2×2 矩阵的乘法运算所使用的乘法数。然而 Hoperoft 和 Kerr 在 1971 年已经证明：计算 2×2 矩阵的乘积，7 次乘法是必要的。因而，降低矩阵乘法运算的时间复杂度应改用其它分块的阶数，如采用 3×3 或 5×5 的块等。目前已知的最低的时间复杂度是$O(n^{2.36})$。而目前所知道的矩阵乘法的最好下界仍是它的平凡下界$\Omega(n^2)$。因此，到目前为止还无法确切知道矩阵乘法的时间复杂性。

Strassen 矩阵乘法采用的技巧也可以用于计算大整数的乘积。参看《计算机算法设计与分析》－王晓东编著，电子工业出版社，2001。

§5　快速 Fourier 变换

连续函数$a(t)$的 Fourier 变换

$$A(f)=\int_{-\infty}^{\infty}a(t)e^{2\pi ift}dt$$

$A(f)$ 的逆变换为

$$a(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty}A(f)e^{-2\pi ift}dt$$

N 个数据 $a=(a_0,a_1,\cdots,a_{N-1})$ 的离散 Fourier 变换

$$A_j=\sum_{0\le k\le N-1}a_k e^{2\pi ijk/N},\quad 0\le j<N$$

其逆变换为

$$a_k=\frac{1}{N}\sum_{0\le j\le N-1}A_j e^{-2\pi ijk/N},\quad 0\le k<N$$

这里，i 是虚数单位，N 是正整数。如果令 $\omega=e^{2\pi i/N}$，则 ω 是复数域上的 N 次本原单位根，此时，计算 A_j 相当于求多项式

$$a(x)=\sum_{0\le k\le N-1}a_k x^k$$

在 ω^j 处的值。根据 Horner 法则，需要做 $2(N-1)$ 次加法和乘法运算，因而，N 个数据的离散 Fourier 变换需要做 $2N(N-1)$ 次复数的加法和乘法运算。但是，我们可以设计一个分治算法，其时间复杂度为 $O(N\log N)$。

首先，若 ω 是 $N=2n$ 次本原单位根，则 ω^2 是 n 次本原单位根，而且，

$$\omega^{j+n}=-\omega^j,\ j=0,1,\cdots,n-1。$$

将多项式 $a(x)$ 的奇次项和偶次项分开，则

$$\begin{aligned}a(x)&=\left(a_1+a_3x^2+\cdots+a_{2n-1}x^{2(n-1)}\right)x+\left(a_0+a_2x^2+\cdots+a_{2(n-1)}x^{2(n-1)}\right)\\&=b(x^2)x+c(x^2)\end{aligned}$$

于是，

$$\begin{aligned}a(\omega^j)&=b(\omega^{2j})\omega^j+c(\omega^{2j})\\a(\omega^{j+n})&=-b(\omega^{2j})\omega^j+c(\omega^{2j})\end{aligned},\quad j=0,1,\cdots,n-1$$

其中，

$$\begin{aligned}b(y)&=a_1+a_3y^2+\cdots+a_{2n-1}y^{n-1}\\c(y)&=a_0+a_2y+\cdots+a_{2(n-1)}y^{n-1}\end{aligned}$$

这样，求 $N=2n$ 个数据的 Fourier 变换就归结为求两次具有 n 个数据的 Fourier 变换。

程序 3-5-1　快速 Fourier 变换

```
FFT(N, a,w,A)
   # N=2^m,w 是 n 次单位根，a 是已知的 N 元数组，代表多项式 a(x)的系数，
   # A 是计算出来的 N 元数组，A[j]=a(w^j), j=0,1,…,N-1.
  real b[ ], c[ ];   int j;
  complex B[ ], C[ ], wp[ ];
  if N=1 then A[0]:=a[0];
  else
      n:=N/2;
      for j from 0 to n-1 do
          b[j]:=a[2*j+1];   c[j]:=a[2*j];
      end{for}
   end{if}
   FFT(n,b,w*w,B);
   FFT(n,c,w*w,C);
   wp[0]:=1;
   for j from 0 to n-1 do
      wp[j+1]:=w*wp[j];
      A[j]:= C[j]+B[j]*wp[j];   A[j+n]:= C[j]-B[j]*wp[j];
   end{for}
end{FFT}
```

以 $T(N)$ 记算法 FFT 的时间复杂度，则

$$T(N)=\begin{cases} a, \text{if } N=1 \\ 2T(N/2)+cN, \text{if } N>1 \end{cases}$$

所以，算法的复杂度为 $T(N)=O(N\log N)$。

利用快速 Fourier 变换可以计算出两个多项式 $f(x), g(x)$ 的乘积，其时间复杂度为 $T(N)=O(N\log N)$，这里 N 是两个多项式的次数之和加 1。这可以分三步完成：第一步，采用 Fourier 变换计算出 $f(x), g(x)$ 在 $\omega^j=e^{2\pi ij/N}$ 处的值 $f(\omega^j), g(\omega^j)$，$j=0,1,\cdots,N-1$，这需要 $O(N\log N)$ 的计算量；第二步，构造 N 个数据 $A=\left[f(\omega^0)g(\omega^0), f(\omega^1)g(\omega^{1a}),\cdots, f(\omega^{N-1})g(\omega^{N-1})\right]$，它们实际上是乘积多项式 $a(x)=f(x)g(x)$ 在 $\omega^j=e^{2\pi ij/N}$，$j=0,1,\cdots,N-1$，处的值，这步需要 $O(N)$ 的计算量；第三步，利用 Fourier 变换的逆变换求出乘积多项式的系数 $a_k, k=0,1,\cdots,N-1$，这需要执行一次快速 Fourier 变换 FFT 即可，只不过那里的

ω取为$e^{-2\pi i/N}$，因而，这一步需要$O(N\log N)$的计算量。

需要指出的是，快速 Fourier 变换需要本原单位根，被变换的数据的个数是$N=2^m$，在复数域中任何N次本原单位根都是存在的，复数$\omega=e^{2\pi i/N}$就是一个。但是计算机对复数的计算是近似的，因此，快速 Fourier 变换得到近似结果。象多项式乘积这样的问题（比如有理系数多项式）需要精确计算，这时我们需要能够进行精确计算的本原单位根，这需要我们考虑一般交换环上的本原单位根。

定义 3.5.1 设R是一个有单位元的交换环，n是正整数，$\omega\in R$，

1）ω称为一个n次单位根，如果$\omega^n=1$；

2）ω称为一个n次本原单位根，如果ω是R的可逆元，ω是一个n次单位根，但对于n的任何素因子p，$\omega^{n/p}-1$都不是环R的零因子。

n次本原单位根具有如下性质：

I. 对于任何正整数$0<l<n$，ω^l-1都不是零因子；

II. $\sum_{0\le j<n}\omega^{lj}=0$，$l=0,1,\cdots,n-1$；

III. 如果ω是n次本原单位根，则ω^{-1}也是；

IV. 如果ω是$n=2m$次本原单位根，则ω^2是m次本原单位根，而且

$$\omega^{j+m}=-\omega^j,\ j=0,1,\cdots,m-1。$$

根据上述 4 条性质，我们有一般形式的 Fourier 变换和逆变换，快速 Fourier 变换，及应用 Fourier 变换设计求解其它问题的快速算法。

§6 最接近点对问题

已知空间中的 n 个点，如何找到最近的两个点即为最近点对问题。在直线上，最近点对问题可以通过一次排序和 1 维扫描来解决，这样做最好的时间复杂度为$O(n\log n)$。实际上，可以设计一个分治算法来解决直线上的最近点对问题。

程序 3-6-1 求一维最近点对距离分治算法

```
ClosPair1(S, d)
  //S 是实轴上点的集合，参数 d 表示 S 中最近点对的距离
  global S,d;
  integer n;
```

```
    float m,p,q;
    n:=|S|;
    if n<2 then d:=∞; return(false); end{if}
    m:=S 中各点坐标的中位数;
    //划分集合 S
    S1:={x∈S| x≤m}; S2:={x∈S| x>m};
    ClosPair1(S1,d1);
    ClosPair1(S2,d2);
    p:=max(S1); q:=min(S2);
    d:=min(d1,d2,q-p);
    return(true);
end{ClosPair1}
```

以 T(n)记算法的时间复杂度，则有如下的递归关系式

$$T(n)=\begin{cases}O(1), & n<4;\\ 2T(n/2)+O(n), & n\geq 4\end{cases}$$

因为求中位数的时间复杂度可为$O(n\log n)$。

上述解决一维最近点对问题的分治算法可以推广解决二维最近点对问题。这里有两个地方需要加工：

一是集合的划分，因为两个坐标，我们只能选取一个坐标作为划分的参考，比如选择集合 S 中各点的 x 坐标的中位数作为划分的标准

$$S1:=\{p\in S\mid x(p)\leq m_x\},\quad S2:=\{p\in S\mid x(p)>m_x\}$$

二是假设$S1,S2$中最近点对的距离分别是d_1,d_2，令$d=\min\{d_1,d_2\}$，而且d不是 S 中最近点对的距离，则 S 中最近的点对(p,q)的两个点应该分布在$S1,S2$两个集合中，譬如$p\in S_1,q\in S_2$。与一维的情形不同，为找到这样的点对(p,q)，需要比照的点对有许多，最坏情况下会有$n^2/4$对。为此，需要通过分析缩小检查的范围。取定S_1中一点p，则S_2中使得点对(p,q)的距离不超过d的点q应该在下面的集合中

$$S_p=\{q\in S\mid m_x<x(q)\leq m_x+d\ and\ y(p)-d\leq y(q)\leq y(p)+d\}$$

S_p处在一个$d\times 2d$的矩形区域中。因为S_p中任意两点的距离都不小于d，所以，

S_p中至多有 6 个点。事实上，我们可以将这个矩形区域均分成 6 个子区域：纵向三等分，水平方向二等分，则每个区域中两点间的距离不超过：$\sqrt{(d/2)^2+(2d/3)^2}=5d/6$，因而，每个区域中至多有$S_p$的一个点。这样，为找到最近的点对$(p,q)$，其中$p\in S_1,q\in S_2$，只需检查至多$6\times\lceil n/2\rceil=3n+6$个点对。根据以上分析，可以设计一个分治算法，求解平面上最近点对问题。

程序 3-6-2 求一维最近点对距离分治算法

```
ClosPair2(S, d)
  //S 是平面上点的集合，但是已经按照 y－坐标不降的次序排好，假定不同
  //点的 x－坐标是不同的。参数 d 表示 S 中最近点对的距离,dist(p,q)表示
  //点对(p, q)之间的距离。
  global S,d;
  integer n;
  float m,p,q;
  n:=|S|;
  if n<2 then d:=∞; return(false); end{if}
  mx:=S 中各点 x－坐标的中位数;
  //划分集合 S 成 S1 和 S2，它们中的点也都是按 y－坐标不降地排列。
  S1:={p∈S| x(p)≤mx}; S2:={q∈S| x(q)>mx};
  ClosPair2(S1,d1);
  ClosPair2(S2,d2);
  dm:=min{d1,d2}; d:=dm;
  //检查距离直线 x=mx 不远于 dm 的两个条形区域中的点对
  P1:={p∈S1|mx-dm≤x(p)}; P2:={q∈S2| x(q)≤mx+dm};
  flag:=1;
  for i to |P1| do
    k:=flag;
    while y(P2[k])<y(P1[i])-dm do
      k:=k+1;
    end{while}
    flag:=k;
    for j from flag to |P2| do
```

```
        if y(P2[j])>y(P1[i])+dm then break;
        else d:=min{d,dist(p,P2[j])};
        end{if}
      end{for}
    end{for}
    return(true);
end{ClosPair2}
```

分析这个算法的时间复杂度，假定为 T(n)。集合 S 划分成集合 S1 和 S2 只需要对集合 S 进行一次扫描即可，复杂度为Θ(n)次操作；程序本身的两次自调用复杂度为 2T(n/2)；形成集合 P1 和 P2 的复杂度为Θ(n)；在外部的 for 循环中包含两部分，while 循环和一个内部 for 循环。while 循环总的循环次数不会超过 n,因而复杂度为Θ(n)；每个内部 for 循环的循环次数不会超过 6，而外部循环次数不超过$\lceil n/2 \rceil$，故所有的内部 for 循环的循环次数总起来不超过 3n+6。程序的其它操作总起来都是常数，因而，

$$T(n) \le \begin{cases} a, & n < 4 \\ 2T(n/2) + cn, & n \ge 4 \end{cases}$$

其中，c为正实数。据此可解出$T(n) = O(n \log n)$。

习题 三

1. 编写程序实现归并排序算法 MergeSortL 和快速排序算法 QuickSort;

2. 用长分别为 100、200、300、400、500、600、700、800、900、1000 的 10 个数组的排列来统计这两种算法的时间复杂性；

3．讨论归并排序算法 MergeSort 的空间复杂性。

4．说明算法 PartSelect 的平均时间复杂性为$O(n)$。

提示：假定数组中的元素各不相同，且第一次划分时划分元素v是第i小元素的概率为$1/n$。因为 Partition 中的 case 语句所要求的时间都是$O(n)$，所以，存在常数c，使得算法 PartSelect 的平均时间复杂度$C_A^k(n)$可以表示为

$$C_A^k(n) \le cn + \frac{1}{n}\left(\sum_{1 \le i < k} C_A^{k-i}(n-i) + \sum_{k < i \le n} C_A^k(i-1) \right) \qquad (1)$$

令$R(n) = \max_k(C_A^k(n))$，取$c \ge R(1)$，试证明$R(n) \le 4cn$。

附页

归并排序的 C++语言描述

```
#include<iostream.h>
template<class T>void MergeSort(T a[],int left,int right);
template<class T>void Merge(T c[],T d[], int l,int m,int r);
template<class T>void Copy(T a[],T b[],int l,int r);
void main()
{
    int const n(5);
    int a[n];
    cout<<"Input "<<n<<"numbers please:";
    for(int i=0;i<n;i++)
        cin>>a[i];
    //for(int j=0;j<n;j++)
        //b[j]=a[j];
    MergeSort(a,0,n-1);
    cout<<"The sorted array is"<<endl;
    for(i=0;i<n;i++)
        cout<<a[i];
    cout<<endl;
}

template<class T>
void MergeSort(T a[],int left,int right) //
{
    if(left<right)
    {
        int i=(left+right)/2;
         T *b=new T[];
        MergeSort(a,left,i);
        MergeSort(a,i+1,right);
        Merge(a,b,left,i,right);
        Copy(a,b,left,right);
    }
```

```
}

template<class T>
void Merge(T c[],T d[],int l,int m,int r)
{
    int i=l;
    int j=m+1;
    int k=l;
    while((i<=m)&&(j<=r))
    {
        if(c[i]<=c[j])d[k++]=c[i++];
        else d[k++]=c[j++];
    }
    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];
}

template<class T>
void Copy(T a[],T b[], int l,int r)
{
    for(int i=l;i<=r;i++)
        a[i]=b[i];
}
```

快速排序的 C++语言描述

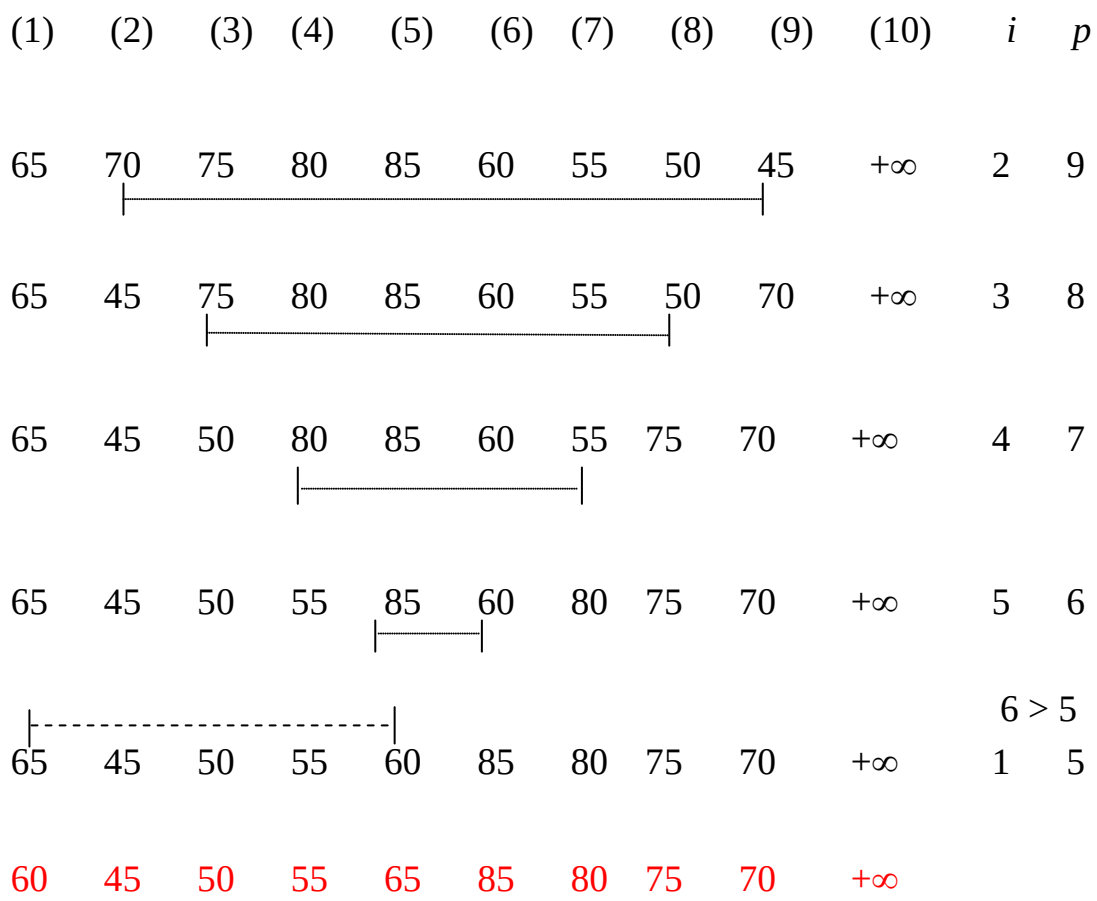
```
#include<iostream.h>
template<class T>void QuickSort(T a[],int p,int r);
```

```
template<class T>int Partition(T a[],int p,int r);
void main()
{
    int const n(5);
    int a[n];
    cout<<"Input "<<n<<"numbers please:";
    for(int i=0;i<n;i++)
        cin>>a[i];
    QuickSort(a,0,n-1);
    cout<<"The sorted array is"<<endl;
    for(i=0;i<n;i++)
        cout<<a[i]<<" ";
    cout<<endl;
}

template<class T>
void QuickSort(T a[],int p,int r)
{
    if(p<r)
    {
        int q=Partition(a,p,r);
        QuickSort(a,p,q-1);
        QuickSort(a,q+1,r);
    }
}

template<class T>
int Partition(T a[],int p,int r)
{
    int i=p,j=r+1;
    T x=a[p];
    while(true)
    {
        while(a[++i]<x);
        while(a[--j]>x);
```

```
        if(i>=j)break;
        Swap(a[i],a[j]);
    }
    a[p]=a[j];
    a[j]=x;
    return j;
}

template<class T>
inline void Swap(T &s,T &t)
{
    T temp=s;
    s=t;
    t=temp;
}
```

附页： PartSelect 程序的执行过程

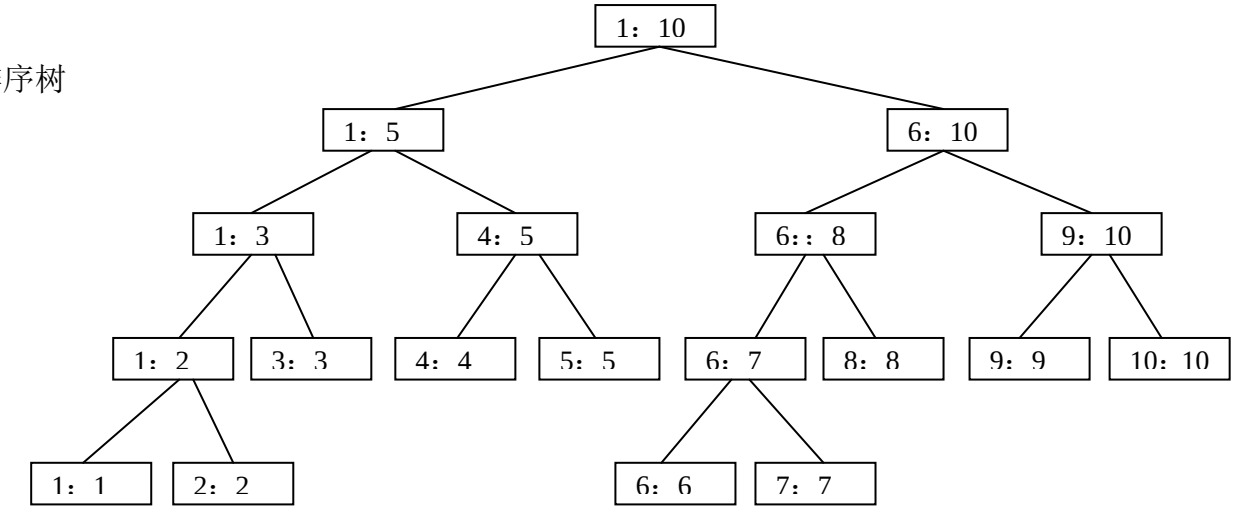
数组 A[1:9]=[65,70,75,80,85,60,55,50,45] 的划分过程



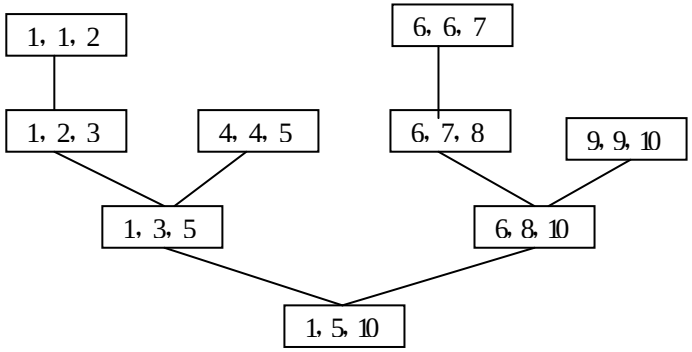
例子，数组 A[1:9]=[65,70,75,80,85,60,55,50,45]，求第 7 小元素

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	*i*	*p*
Partition(1,10) j=5	60	45	50	55	65	85	80	75	70	+∞	10	9
Partition(6,10) j=9	60	45	50	55	65	70	80	75	85	+∞	7	6
Partition(6,9) j=6	60	45	50	55	65	70	80	75	85	+∞	9	8
Partition(7,9) j=8	60	45	50	55	65	70	75	80	85	+∞	8	7
Partition(7,8) j=7												

附页：归并排序树

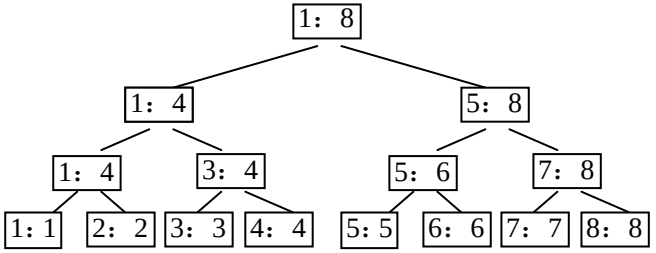


（含有 10 个元素数组的）MergeSort 调用（分拆）过程



Merge 调用（合并）过程

附页：链表归并过程



MergeSort 调用（分拆）过程

链表的归并过程

A = [50 , 10 , 25 , 30 , 15 , 70 , 35 , 55]　　数据表

(0) , (1) , (2) , (3) , (4) , (5) , (6) , (7) , (8)

Link=[0 ,　0 ,　0 ,　0 ,　0 ,　0 ,　0 ,　0 ,　0]　　初始化为零，逐步修改

q　r　p

1>2→ 2 ,　0 ,　1 ,　0 ,　0 ,　0 ,　0 ,　0 ,　0 [10,50]

3<4→ 3 ,　0 ,　1 ,　4 ,　0 ,　0 ,　0 ,　0 ,　0 [10,50], [25,30]

2<3→ 2 ,　0 ,　3 ,　4 ,　1 ,　0 ,　0 ,　0 ,　0 [10,25,30,50]

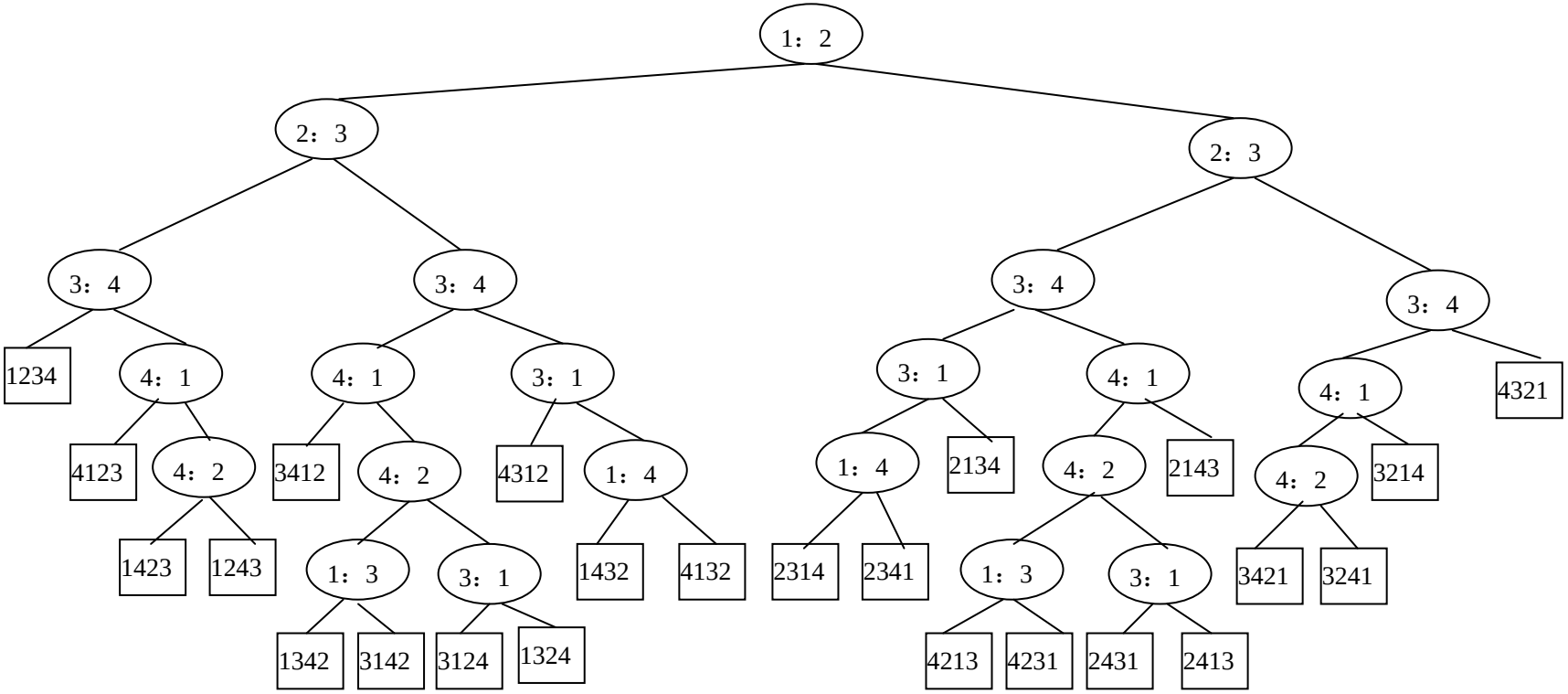
5<6→ 5 ,　0 ,　3 ,　4 ,　1 ,　6 ,　0 ,　0 ,　0 [10,25,30,50],[15,70]

7<8→7,　0,　3 ,　4 ,　1 ,　6 ,　0 ,　8,　0 [10,25,30,50],[15,70],[35,55]

5<7→5,　0,　3 ,　4 ,　1 ,　7 ,　0 ,　8，　6 [10,25,30,50],[15, 35,55,70]

2<5→2,　8 ,　5 ,　4 ,　7 ,　3 ,　0 ,　1，　6 [10, 15,25,30, 35,50, 55,70]

附页：排序比较树



N=4 时的一棵比较树