第 3 章

分 治 算 法

学习目标

☑ 掌握分治算法的基本思想和求解步骤；

☑ 理解分治算法的精髓，即如何分？如何治？才能使得算法效率更高；

☑ 通过实例学习，能够运用分治策略设计解决实际问题的算法并编程实现。

凡治众如治寡，分数是也。

——《孙子兵法》

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关：问题的规模越小，越容易直接求解，所需的计算时间也就越少。例如，对于 n 个元素的排序问题，当 $n=1$ 时，不需任何计算；$n=2$ 时，只要做一次比较即可排好序；$n=3$ 时只要做 3 次比较即可，当 n 较大时，问题就不那么容易处理了。可见，要想直接解决一个规模较大的问题，有时是很困难的。为了更好地解决这些规模较大的问题，分治算法应运而生。

在计算机科学中，分治算法是一种很重要的算法。它采取各个击破的技巧解决一个规模较大的问题，该技巧是很多高效算法的基础，如排序算法（快速排序、归并排序）、傅里叶变换（快速傅里叶变换）等。

微课视频

3.1 分治算法概述

3.1.1 分治算法的基本思想

分治算法字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同子问题，再把子问题分成更小的子问题，直到最后各个子问题可以简单地直接求解，对各个子问题的解进行合并即得原问题的解。

可见，分治算法的基本思想是将一个难以直接解决的大问题，分解成一些规模较小的相同问题，以便各个击破，分而治之。

那么，何时能、何时应该采用分治算法解决问题呢？即分治算法所能解决的问题应该具备哪些特征？从许多可以用分治算法求解的问题中，总结出这些问题一般具有以下几个

特征：

(1) 问题的规模缩小到一定程度就可以容易地解决。

(2) 问题可以分解为若干个规模较小的相同子问题。

(3) 问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

(4) 问题分解出的子问题的解可以合并为原问题的解。

上述的第(1)条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般随着问题规模的增大而增加；第(2)条特征是应用分治算法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用；第(3)条特征涉及分治算法的效率，如果各个子问题是不独立的，则分治算法要做许多不必要的工作——重复求解公共的子问题；第(4)条特征是关键，能否利用分治算法完全取决于问题是否具有第(4)条特征。

3.1.2　分治算法的解题步骤

通常，分治算法的求解过程都要遵循两大步骤：分解和治理。

步骤 1：分解。

既然是分治算法，当然要对待求解问题进行分解，即将问题分解为若干个规模较小、相互独立、与原问题形式相同的子问题。

那么，究竟该如何合理地对问题进行分解呢？应把原问题分解为多少个子问题才较适宜？每个子问题是否规模相同才为适当？这些问题很难给予肯定的回答。人们从大量的实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同，即将一个问题分为大小相等的 k 个子问题(通常 $k=2$)，这种处理方法行之有效。这种使子问题规模大致相等的做法是出自一种平衡子问题的思想，它几乎总是比子问题规模不等的做法要好。也有 $k=1$ 的划分，这仍然是把问题划分为两部分，取其中的一部分，而丢弃另一部分，例如二分查找问题在采用分治算法求解时就是这样划分的。

步骤 2：治理。

步骤 2-1：求解各个子问题。若子问题规模较小而容易被解决则直接求解，否则再继续分解为更小的子问题，直到容易解决为止。

那么，如何对各个子问题进行求解呢？由于采用分治法求解的问题被分解为若干个规模较小的相同子问题，各个子问题的解法与原问题的解法是相同的。因此，很自然想到采取递归技术来对各个子问题进行求解。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而规模不断缩小，最终使子问题缩小到很容易求解的规模，这导致递归过程的产生。分治与递归就像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。有时候，递归处理也可以采用循环来实现。

步骤 2-2：合并。它是将已求得的各个子问题的解合并为原问题的解。

合并这一步对分治算法的算法性能至关重要，算法的有效性在很大程度上依赖于合并步的实现，因情况的不同合并的代价也有所不同。

下面通过几个实例具体讲述分治算法求解问题的具体过程。

3.2 二分查找

1. 问题描述

二分查找又称为折半查找,它要求待查找的数据元素必须是按关键字大小有序排列的。问题描述:给定已排好序的 n 个元素 $s_1,s_2,\cdots,s_n$,现要在这 n 个元素中找出一特定元素 x。

首先较容易想到使用顺序查找方法,逐个比较 $s_1,s_2,\cdots,s_n$,直至找出元素 x 或搜索整个序列后确定 x 不在其中。显然,该方法没有很好地利用 n 个元素已排好序这个条件。因此,在最坏情况下,顺序查找方法需要 n 次比较。

2. 算法思想及设计

该算法的思想是:假定元素序列已经由小到大排好序,将有序序列分成规模大致相等的两部分,然后取中间元素与特定查找元素 x 进行比较,如果 x 等于中间元素,则算法终止;如果 x 小于中间元素,则在序列的左半部继续查找,即在序列的左半部重复分解和治理操作;否则,在序列的右半部继续查找,即在序列的右半部重复分解和治理操作。可见,二分查找算法重复利用了元素间的次序关系。

算法的求解步骤设计如下:

步骤 1:确定合适的数据结构。设置数组 $s[n]$来存放 n 个已排好序的元素;变量 low 和 high 分别表示查找范围在数组中的下界和上界;middle 表示查找范围的中间位置;x 为特定元素。

步骤 2:初始化。令 low=0,即指示 s 中第一个元素;high=n-1,即指示 s 中最后一个元素。

步骤 3:middle=(low+high)/2,即指示查找范围的中间元素。

步骤 4:判定 low≤high 是否成立,如果成立,转步骤 5;否则,算法结束。

步骤 5:判断 x 与 s[middle]的关系。如果 x 等于 s[middle],算法结束;如果 $x>s$[middle],则令 low=middle+1;否则令 high=middle-1,转到步骤 3。

3. 二分查找算法的构造实例

【例 3-1】 用二分查找算法在有序序列[6 12 15 18 22 25 28 35 46 58 60]中查找元素 12。假定该有序序列存放在一维数组 s[11]中。

步骤 1:令 low=0,high=10。计算 middle=(0+10)/2=5,即利用中间位置 middle 将序列一分为二,如图 3-1 所示。

步骤 2:将 x 与 s[middle]进行比较。此时 $x<s$[middle],说明 x 可能位于序列的左半部,即查找范围变为 s[0: middle-1]。令 high=middle-1=4。

步骤 3:计算 middle=(0+4)/2=2,利用此时的中间位置 middle=2 将子序列[6 12 15 18 22]一分为二,如图 3-2 所示。

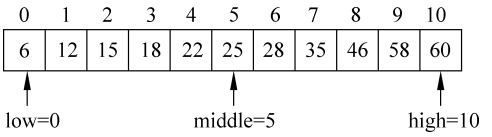

图 3-1　第一次划分示意图

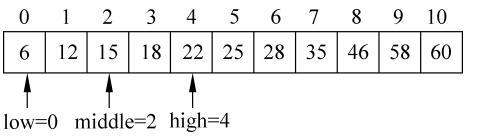

图 3-2　第二次划分示意图

步骤 4：将 x 与 s[middle]进行比较。此时 $x<s$[middle]，说明 x 可能位于子序列 s[0：middle－1]中。令 high＝middle－1＝1。

步骤 5：重新计算 middle＝(0＋1)/2＝0。利用此时的中间位置 middle＝0 将子序列[6　12]一分为二，如图 3-3 所示。

步骤 6：将 x 与 s[middle]进行比较。此时 $x>s$[middle]，说明 x 可能位于 s[middle ＋1：high]中。令 low＝middle＋1＝1。

步骤 7：计算 middle＝(1＋1)/2＝1，如图 3-4 所示。

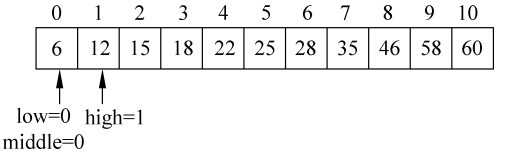

图 3-3　第三次划分示意图

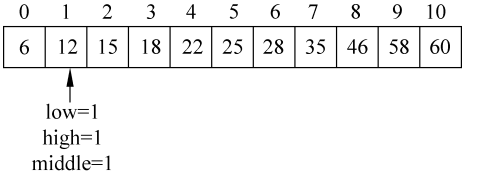

图 3-4　第四次划分示意图

此时 $x=s$[middle]＝12，查找成功。

4. 算法描述

二分查找算法的思想易于理解，但要写一个正确的二分查找算法并不是一件简单的事情。Knuth 在他的著作 *The Art of Computer Programming*：*Sorting and Searching* 中提到，第一个二分查找算法早在 1946 年就出现了，但直到 1962 年第一个完全正确的二分查找算法才出现。

本书给出了该算法的两种描述形式。其中，数组 $s[n]$、变量 low、high、middle 的含义与求解步骤中的含义相同。

(1) 二分查找算法实现的非递归形式。

```
int NBinarySearch(int n,int s[n],int x)
{int low = 0,high = n - 1;
 while(low < = high)
  {int middle = (low + high)/2;
  if(x = = s[middle])   return middle;
  else if(x > s[middle]) low = middle + 1;
      else high = middle - 1;
  }
 return  - 1;
}
```

(2) 二分查找算法实现的递归形式。

```
int BinarySearch(int s[n],int x,int low,int high)
{
    if (low > high) return -1;
    int middle = (low + high)/2;
    if(x == s[middle])  return middle;
    else if(x > s[middle])
         return BinarySearch(s, x, middle + 1, high);
       else
         return BinarySearch(s, x, low, middle - 1);
}
```

5. 算法分析

从算法描述可知,每执行一次 while 循环或递归调用一次 BinarySearch 算法,待搜索范围的大小就减少一半。在查找的过程中,每次均将下标为 middle 的元素与给定值 x 进行比较,如果 $x<s[\text{middle}]$,则修改 high 为 middle−1,如果 $x>s[\text{middle}]$,则修改 low 为 middle+1,并重新计算 middle 的值;以此类推,直到 x 等于 $s[\text{middle}]$,则查找成功;或者 low>high,则查找失败。

设给定的有序序列中具有 n 个元素。

显然,当 $n=1$ 时,查找一个元素需要常量时间,因而 $T(n)=O(1)$。

当 $n>1$ 时,计算序列的中间位置及进行元素的比较,需要常量时间 $O(1)$。递归地求解规模为 $n/2$ 的子问题,所需时间为 $T(n/2)$。

因此,二分查找算法所需的运行时间 $T(n)$的递归形式如下:

$$T(n)=\begin{cases}O(1), & n=1\\ T(n/2)+O(1), & n>1\end{cases}$$

当 $n>1$ 时,有

$$\begin{aligned}T(n)&=T(n/2)+O(1)\\&=T(n/4)+2O(1)\\&=T(n/8)+3O(1)\\&=\cdots\\&=T(n/2^x)+xO(1)\end{aligned}$$

简单起见,令 $n=2^x$,则 $x=\log n$。

由此,$T(n)=T(1)+\log n=O(1)+O(\log n)$。因此,二分查找算法的时间复杂性为 $O(\log n)$。

采用非递归形式实现的二分查找算法,其空间复杂性为常量级 $O(1)$;采用递归形式实现的二分查找算法,其空间复杂性为 $O(\log n)$。

6. C++实战

相关代码如下。

```
#include<iostream>
#include<malloc.h>
using namespace std;
int NBinarySearch(int n,int s[],int x){
    int low=0,high=n-1;
    while(low<=high){
        int middle=(low+high)/2;
        if(x==s[middle])
            return middle;
        else if(x>s[middle])
            low=middle+1;
        else
            high=middle-1;
    }
    return -1;
}
int BinarySearch(int s[],int x,int low,int high)
{
    if (low>high) return -1;
    int middle=(low+high)/2;
    if(x==s[middle]) return middle;
    else if(x>s[middle])
        return BinarySearch(s, x, middle+1, high);
    else
        return BinarySearch(s, x, low, middle-1);
}
int main(){
    int n = 0;
    cout<<"请输入元素个数 n: ";
    cin>>n;
    int * s = (int *)malloc(n*sizeof(int));
    cout<<"请输入 n 个有序元素:";
    for(int i=0;i<n;i++){
        cin>>s[i];
    }
    int k;
    cout<<"请输入要查找的元素 k: ";
    cin>>k;
    int position = NBinarySearch(n,s,k);
    int position_recursion = BinarySearch(s,k,0,n-1);
    cout<<"非递归算法——k 在 s 中的位置是: "<<position<<endl;
    cout<<"递归算法——k 在 s 中的位置是: "<<position_recursion<<endl;
}
```


微课视频

3.3 循环赛日程表

1. 问题描述

设有 $n=2^k$ 个运动员要进行羽毛球循环赛，现要设计一个满足以下要求的比赛日程表：

(1) 每个选手必须与其他 $n-1$ 个选手各赛一次。

(2) 每个选手一天只能比赛一次。

(3) 循环赛一共需要进行 $n-1$ 天。

由于 $n=2^k$，显然 n 为偶数。

2. 分治算法求解的算法思路

(1) 如何分，即如何合理地进行问题的分解？

根据分治算法的思想，将选手一分为二，n 个选手的比赛日程表可以通过对 $n/2=2^{k-1}$ 个选手设计的比赛日程表来实现，而 2^{k-1} 个选手的比赛日程表可通过对 $2^{k-1}/2=2^{k-2}$ 个选手设计的比赛日程表来实现，以此类推，2^2 个选手的比赛日程表可通过对两个选手设计的比赛日程表来实现。此时，问题的求解将变得异常简单。

(2) 如何治，即如何进行问题的求解？

根据问题的分解思想可知，在对问题的求解过程中，递归地执行一分为二的分解策略，直至只剩下两个选手时，比赛日程表的制定就变得很简单：只要让这两个选手直接比赛就可以了。反过来，如果两个选手的比赛日程表已经制定出来，那么 $2\times2=2^2$ 个选手的比赛日程表可以合并求出，以此类推，直到 2^{k-1} 个选手的比赛日程表制定出来时，那么 2^k 个选手的比赛日程表的制定就迎刃而解了。可见，2^k 个选手的比赛日程安排问题是通过依次求解 $2^1,2^2,\cdots,2^k$ 个选手的比赛日程问题得出的。

(3) 问题的关键——发现循环赛日程表制定过程中存在的规律性。

下面从简单的实例入手，期望发现循环赛日程表制定过程中存在的规律。

假设 n 位选手的编号为 $1,2,3,\cdots,n$。按照问题描述和分治算法求解思路，可将比赛日程表设计成一个 n 行 $n-1$ 列的二维表，其中，行表示选手编号，列表示选手比赛的天数，第 i 行第 j 列表示第 i 个选手在第 j 天所遇到的选手。

【例 3-2】 $n=2^1$ 个选手的比赛日程表的制定。

根据上述分析，两个选手的比赛日程表非常容易制定，直接进行比赛即可，且需要 $n-1=2^1-1=1$ 天完成。2^1 个选手的比赛日程表如表 3-1 所示。

表 3-1 的意思是：第一天和 1 号选手进行比赛的是 2 号选手，当然，第一天和 2 号选手进行比赛的是 1 号选手。仔细研究发现，该表有一定的规律性：左上角的 1 和右下角的 1 对称，左下角的 2 和右上角的 2 对称。

【例 3-3】 $n=2^2$ 个选手的比赛日程表的制定。

首先，依据分治算法的思想，将问题一分为二，即分别求出 1 号和 2 号选手的比赛日程

表、3 号和 4 号选手的比赛日程表，如表 3-2 所示。

其次，依据例 3-2 中发现的规律性，将表 3-2 中求得的两个子问题的解进行合并，就可求出 2^2 个选手的比赛日程表，具体结果如表 3-3 所示。

表 3-1　2^1 个选手的比赛日程表

编号	天数
	1
1	2
2	1

表 3-2　子问题的比赛日程表

编号	天数
	1
1	2
2	1
3	4
4	3

表 3-3　2^2 个选手的比赛日程表

编号	天数		
	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

观察表 3-3，其制定完全符合要求，4 个选手共比赛 3 天，每个选手一天只比赛一次，且在 3 天内与其他选手均进行了比赛。由此可见，例 3-2 中的规律性是可取的。

【例 3-4】　2^3 个选手的比赛日程表的制定。

首先，根据分治算法的思想将问题一分为二，得到两个子问题：即问题 1(1,2,3,4)和问题 2(5,6,7,8)，两个子问题继续一分为二，最终得到可直接求解的 4 个子问题：问题 1-1(1,2)、问题 1-2(3,4)、问题 2-1(5,6)、问题 2-2(7,8)。

分别求出这 4 个子问题的解，具体求解结果如表 3-4 所示。

其次，根据表制定的规律性，将问题 1-1(1,2)和问题 1-2(3,4)的解进行合并，可得到问题 1(1,2,3,4)的解；同样，将问题 2-1(5,6)和问题 2-2(7,8)的解进行合并，可得到问题 2(5,6,7,8)的解，求解结果如表 3-5 所示。

表 3-4　4 个子问题的比赛日程表

编号	天数
	1
1	2
2	1
3	4
4	3
5	6
6	5
7	8
8	7

表 3-5　子问题解的合并

编号	天数		
	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1
5	6	7	8
6	5	8	7
7	8	5	6
8	7	6	5

最后，将问题 1(1,2,3,4)和问题 2(5,6,7,8)的解进行合并，可得到 2^3 个选手的比赛日程表，求解结果如表 3-6 所示。

表 3-6　2^3 个选手的比赛日程表

编号	天数						
	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

至此，2^3 个选手的比赛日程表制定完毕。

3. 算法描述

算法描述如下：

```
void Round_Robin_Calendar(int n,int k, int **a)
{  //整数 k、空的二维数组 a 用来表示比赛日程安排表
   for(i=1;i<=n;i++)  a[1][i]=i;   //用一个 for 循环输出日程表的第一行
   int m=1;                        //m 用来控制每一次填充数组时 i(i 表示行)和 j(j 表示
                                   //列)的起始填充位置
   for(int s=1;s<=k;s++)           //将问题划分为 k 部分,依次处理
     {
        n/=2;
        for(int t=1;t<=n;t++)      //对每一部分的问题进行划分,然后根据分治法的思想,
                                   //进行每一个单元格的填充,填充原则是:对角线填充
           for(int i=m+1;i<=2*m;i++)          //i 控制行
               for(int j=m+1;j<=2*m;j++)      //j 控制列
                   {
                    a[i][j+(t-1)*m*2]=a[i-m][j+(t-1)*m*2-m];  //右下角的值等
                                                              //于左上角的值
                    a[i][j+(t-1)*m*2-m]=a[i-m][j+(t-1)*m*2]; }  //左下角的值等
                                                              //于右上角的值
         m*=2;
      }
}
```

4. C++实战

相关代码如下。

```
#include<iostream>
using namespace std;
void Round_Robin_Calendar(int n,int k, int **a)
```

```
{  //整数 k、空的二维数组 a 用来表示比赛日程安排表
   int i,j,s;
   for(i = 1;i < = n;i++) a[1][i] = i;     //用一个 for 循环输出日程表的第一行
   int m = 1;                              //m 用来控制每一次填充数组时 i(i 表示行)和 j(j 表示
                                           //列)的起始填充位置
    for(s = 1;s < = k;s++)                 //将问题划分为 k 部分,依次处理
    {
        n/ = 2;
        for(int t = 1;t < = n;t++)         //对每一部分的问题进行划分,然后根据分治法的思想,
                                           //进行每一个单元格的填充,填充原则是: 对角线填充
          for(i = m + 1;i < = 2 * m;i++)   //i 控制行
              for(j = m + 1;j < = 2 * m;j++) //j 控制列
                {
                    a[i][j + (t - 1) * m * 2] = a[i - m][j + (t - 1) * m * 2 - m];
                                                          //右下角的值等于左上角的值
                    a[i][j + (t - 1) * m * 2 - m] = a[i - m][j + (t - 1) * m * 2];
                                                          //左下角的值等于右上角的值
                }
        m * = 2;
    }
}
int main(){
    cout <<"请输入运动员数 2 的 k 次方: " ;
    int k;
    cin >> k;
    int n = 1;
    //计算 2 的 k 次方,即计算运动员数 n
   for(int i = 1;i < = k;i++) n * = 2;
    // 动态分配二维数组
   int **a = new int *[n + 1];
   for(int i = 0;i < = n;i++){
        a[i] = new int[n + 1];
    }
    //比赛日程安排
   Round_Robin_Calendar(n,k,a);
   //输出循环日程安排
   for(int i = 1;i < = n;i++)
   {
        for(int j = 1;j < = n;j++){
            cout << a[i][j]<<" ";
        }
        cout << endl;
   }
}
  for(int i = 0;i < = n;i++)
            delete [] a[i];
   delete [] a;
}
```


微课视频

3.4 合并排序

1. 算法思想

合并排序是采用分治策略实现对 n 个元素进行排序的算法,是分治算法的一个典型应用和完美体现。它是一种平衡、简单的二分分治策略,其计算过程分为三大步:

(1) 分解:将待排序元素分成大小大致相同的两个子序列。

(2) 求解子问题:用合并排序法分别对两个子序列递归地进行排序。

(3) 合并:将排好序的有序子序列进行合并,得到符合要求的有序序列。

那么,该如何更好地理解合并排序算法的思想呢?由于排序问题给定的是一个无序的序列,而合并是合并两个已经排好序的序列。为此,可以把待排序元素分解成两个规模大致相等的子序列,如果不易解决,再将得到的子序列继续分解,直到子序列中包含的元素个数为 1。众所周知,单个元素的序列本身是有序的,此时可进行合并,这就是分治策略的巧妙运用。

2. 算法设计及描述

(1) 合并过程。

从算法的思想很容易看出:合并排序的关键步骤在于如何合并两个已排好序的有序子序列。为了进行合并,引入一个辅助过程 Merge(A,low,middle,high),该过程将排好序的两个子序列 A[low:middle]和 A[middle+1:high]进行合并。其中,low、high 表示待排序范围在数组中的下界和上界,middle 表示两个序列的分开位置,满足 low≤middle<high;由于在合并过程中可能会破坏原来的有序序列,因此,合并最好不要就地进行,本算法采用了辅助数组 B[low:high]来存放合并后的有序序列。

合并方法:设置 3 个工作指针 i,j,k。其中,i 和 j 指示两个待排序序列中当前需比较的元素,k 指向辅助数组 B 中待放置元素的位置。比较 $A[i]$和 $A[j]$的大小关系,如果 $A[i]$小于或等于 $A[j]$,则 $B[k]=A[i]$,同时将指针 i 和 k 分别推进一步;反之,$B[k]=A[j]$,同时将指针 j 和 k 分别推进一步。如此反复,直到其中一个序列为空。最后,将非空序列中的剩余元素按原次序全部放到辅助数组 B 的尾部。

合并两个有序子序列的算法描述如下:

```
void Merge(int A[],int low,int middle,int high)
 {
  int i,j,k;                          //参数 i, j 分别表示两个待合并的有序子序列的当前位置; k
                                      //表示合并后的有序序列的当前位置
  int *B = new int[high - low + 1];
  i = low; j = middle + 1; k = 0;
  while(i <= middle&&j <= high)       //两个子序列非空
      if(A[i]<= A[j])  B[k++] = A[i++];
      else  B[k++] = A[j++];
  while (i <= middle)                 //如果子序列 A[low:middle]非空,则进行收尾处理
```

```
            B[k++] = A[i++];
    while (j <= high)                    //如果子序列 A[middle + 1:high]非空,则进行收尾处理
            B[k++] = A[j++];
    for(i = low;i <= high;i++)           //将合并后的序列复制回数组 A
            j = 0
            A[i] = B[j++];
}
```

最坏的情况是两个子序列 A[low:middle]和 A[middle+1:high]的数据是一个交替的序列,例如 (1,3,5,7)和(2,4,6,8),则需要比较的次数为 $n-1=7$ 次。

(2) 递归形式的合并排序算法。

递归形式的合并排序算法就是把序列分为两个子序列,然后对子序列进行递归排序,再把两个已排好序的子序列合并成一个有序的序列。

可以将 Merge 过程作为合并排序算法的一个子程序使用。设置 MergeSort(A,low,high)对子序列 A[low:high]进行排序,如果 low≥high,则该子序列中至多只有一个元素,当然是已排序好的。否则,依据分治算法的思想对问题进行分解,即计算出一个下标 middle,将 A[low:high]分解成 A[low:middle]和 A[middle+1:high],分解原则是使二者的大小大致相等。

合并排序算法描述如下:

```
void MergeSort(int A[],int low,int high)
  {     int middle;
         if (low < high)
             {
                middle = (low + high)/2;            //取中点
                MergeSort(A,low,middle);            //对 A[low:middle]中的元素进行排序
                MergeSort(A,middle + 1,high);       //对 A[middle + 1:high]中的元素进行排序
                Merge(A,low,middle,high);           //合并
             }
    }
```

3. 合并排序算法的构造实例

【例 3-5】 设待排序序列 $A=<8,3,2,9,7,1,5,4>$,采用 MergeSort 算法对序列 A 进行排序。具体排序过程如图 3-5 所示。

其实,通过综合算法的设计思想和上述求解过程展示,很容易看出合并排序算法的求解过程实质上是:经过迭代分解,待排序序列 A 最终被分解成 8 个只含一个元素的序列,然后两两合并,最终合并成一个有序序列。

4. 算法分析

假设待排序序列中元素个数为 n。

显然,当 $n=1$ 时,合并排序一个元素需要常数时间,因而 $T(n)=O(1)$。

当 $n>1$ 时,将时间 T 如下分解:

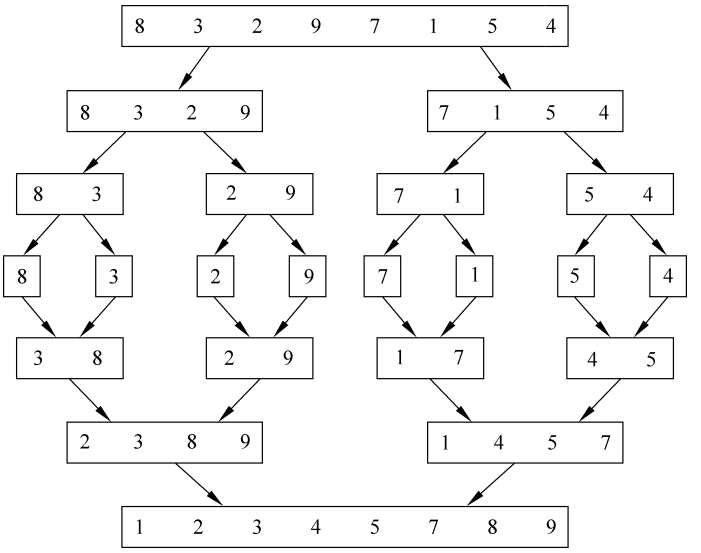

图 3-5 合并排序过程示意图

分解：这一步仅仅是计算出子序列的中间位置，需要常数时间 $O(1)$。

解决子问题：递归求解两个规模为 $n/2$ 的子问题，所需时间为 $2T(n/2)$。

合并：对于一个含有 n 个元素的序列，Merge 算法可在 $O(n)$时间内完成。

将以上阶段所需的时间进行相加，即得到合并排序算法对 n 个元素进行排序，在最坏情况下所需的运行时间 $T(n)$的递归形式为

$$T(n)=\begin{cases}O(1), & n=1\\ 2T(n/2)+O(n), & n>1\end{cases}$$

当 $n>1$ 时，有

$$\begin{aligned}T(n)&=2T(n/2)+O(n)\\&=2(2T(n/4)+O(n/2))+O(n)=4T(n/4)+2O(n)\\&=4(2T(n/8)+O(n/4))+2O(n)=8T(n/8)+3O(n)\\&=\cdots\\&=2^xT(n/2^x)+xO(n)\end{aligned}$$

令 $n=2^x$，则 $x=\log n$。

由此可得，$T(n)=nT(1)+\log n\ O(n)=O(n)+O(n\log n)$，即合并排序算法的时间复杂性为$O(n\log n)$。

合并排序算法所使用的工作空间取决于 Merge 算法，每调用一次 Merge 算法，便分配一个适当大小的缓冲区，退出 Merge 便释放它。在最后一次调用 Merge 算法时，所分配的缓冲区最大，此时，它把两个子序列合并成一个长度为 n 的序列，需要 $O(n)$个工作单元。所以，合并排序算法的空间复杂性为 $O(n)$。

5. C++实战

相关代码如下。

```
#include<iostream>
#include<iterator>
#include<algorithm>
using namespace std;
void Merge(int A[],int low,int middle,int high)
{
    int i,j,k;                          //参数 i, j 分别表示两个待合并的有序子序列的当前位
                                        //置; k 表示合并后的有序序列的当前位置
     int *B=new int[high-low+1];
    i=low;
    j=middle+1;
    k=0;
    while(i<=middle&&j<=high)           //两个子序列非空
    {
        if(A[i]<=A[j])
            B[k++]=A[i++];
        else
            B[k++]=A[j++];
    }
    while (i<=middle)                   //如果子序列 A[low:middle]非空,则进行收尾处理
        B[k++]=A[i++];
    while (j<=high)                     //如果子序列 A[middle+1:high]非空,则进行收尾处理
        B[k++]=A[j++];
    j = 0;
    for(i=low;i<=high;i++)              //将合并后的序列复制回数组 A
        A[i]=B[j++];
    delete []B;
}

void MergeSort (int A[],int low,int high)
{
    int middle;
    if (low<high)
        {
            middle=(low+high)/2;                //取中点
            MergeSort(A,low,middle);            //对 A[low:middle]中的元素进行排序
            MergeSort(A,middle+1,high);         //对 A[middle+1:high]中的元素进行排序
            Merge(A,low,middle,high);           //合并
        }
}
int main(){
    cout<<"请输入待排序元素个数: ";
    int n;
    cin>>n;
    int *a=new int[n];
    cout<<"请输入待排序元素: ";
    for(int i=0 ;i<n;i++)
        cin>>a[i];
```

```
    MergeSort(a,0,n-1);
    copy(a,a+n,ostream_iterator<int>(cout," ")); //输出 a 中的元素
    delete []a;
}
```


微课视频

3.5 快速排序

1. 算法思想

快速排序是 C. A. R. Hoare 于 1962 年提出的划分交换排序方法,其基本思想是:通过一趟扫描将待排序的元素分割成独立的三个序列,第一个序列中所有元素均不大于基准元素,第二个序列是基准元素,第三个序列中所有元素均不小于基准元素。由于第二个序列已经处于正确位置,因此需要再按此方法对第一个序列和第三个序列分别进行排序,整个排序过程可以递归进行,最终可使整个序列变成有序序列。

(1) 快速排序算法的分治策略体现。

快速排序的基本思想是基于分治策略的,利用分治法可将快速排序的基本思想描述如下:设当前待排序的序列为 $R[low:high]$,其中 low≤high,如果序列的规模足够小则直接进行排序,否则分三步处理。

① 分解。

在 $R[low:high]$中选定一个元素作为基准元素(pivot),该基准元素的位置(pivotpos)在划分的过程中确定。以此基准元素为标准将待排序序列划分为两个子序列 $R[low:pivotpos-1]$和 $R[pivotpos+1:high]$,并使序列 $R[low:pivotpos-1]$中所有元素的值均小于或等于 $R[pivotpos]$,序列 $R[pivotpos+1:high]$中所有元素的值均大于或等于 $R[pivotpos]$。此时基准元素已位于正确的位置上,它无须参加后面的排序。

注意:划分序列的关键是要计算出所选定的基准元素所在的位置 pivotpos。其中 low≤pivotpos≤high。

② 求解子问题。

对两个子序列 $R[low:pivotpos-1]$和 $R[pivotpos+1:high]$,分别通过递归调用快速排序算法来进行排序。

③ 合并。

由于对 $R[low:pivotpos-1]$和 $R[pivotpos+1:high]$的排序是就地进行的,所以在 $R[low:pivotpos-1]$和 $R[pivotpos+1:high]$都已排好序后,合并步骤无须做什么,序列 $R[low:high]$就已排好序了。

(2) 基准元素的选取。

从待排序序列中选取指导划分的基准元素是决定算法性能的关键。基准元素的选取应该遵循平衡子问题的原则,即使得划分后的两个子序列的长度尽量相同。基准元素的选择方法有很多种,常用的有:①取第一个元素。即以待排序序列的首元素作为基准元素。

②取最后一个元素。即以待排序序列的尾元素作为基准元素。③取位于中间位置的元素。即以待排序序列的中间位置的元素作为基准元素。④“三者取中的规则”。即在待排序序列中，将该序列的第一个元素、最后一个元素和中间位置的元素进行比较，取三者之中值作为基准元素。⑤取位于 low 和 high 之间的随机数 k(low≤k≤high)，用 $R[k]$作为基准元素。即采用随机函数产生一个位于 low 和 high 之间的随机数 k(low≤k≤high)，用 $R[k]$作为基准，这相当于强迫 R[low:high]中的元素是随机分布的。

无论如何，在快速排序算法中选取的基准元素一定要保证算法正常结束。

2. 划分方法

从算法思想容易看出：实现快速排序的关键在于依据所选的基准元素对序列进行划分，那么，究竟如何来实现划分呢？

(1) 划分方法的过程设计。

假设待排序序列为 R[low:high]，该划分过程以第一个元素作为基准元素。

步骤 1：设置两个参数 i 和 j，它们的初值分别为待排序序列的下界和上界，即 i = low，j = high。

步骤 2：选取待排序序列的第一个元素 R[low]作为基准元素，并将该值赋给变量 pivot。

步骤 3：令 j 自 j 位置开始向左扫描，如果 j 位置所对应的元素的值大于或等于 pivot，则 j 前移一个位置(即 j--)。重复该过程，直至找到第 1 个小于 pivot 的元素 $R[j]$，将 $R[j]$与 $R[i]$进行交换，i++。其实，交换后 $R[j]$所对应的元素就是 pivot。

步骤 4：令 i 自 i 位置开始向右扫描，如果 i 位置所对应的元素的值小于或等于 pivot，则 i 后移一个位置(即 i++)。重复该过程，直至找到第 1 个大于 pivot 的元素 $R[i]$，将$R[j]$与 $R[i]$进行交换，j--。其实，交换后 $R[i]$所对应的元素就是 pivot。

步骤 5：重复步骤 3、步骤 4，交替改变扫描方向，从两端各自往中间靠拢直至 i = j。此时 i 和 j 指向同一个位置，即基准元素 pivot 的最终位置。

(2) 划分方法的算法描述。

```
int Partition(int R[],int low,int high)
  {
      int i = low,j = high,pivot = R[low];      //用序列的第一个元素作为基准元素
      while(i < j)                              //从序列的两端交替向中间扫描,直至 i 等于 j 为止
       {
          while(i < j&&R[j]> = pivot)           //pivot 相当于在位置 i 上
             j -- ;                             //从右向左扫描,查找第 1 个小于 pivot 的元素
          if(i < j)                             //表示找到了小于 pivot 的元素
             swap(R[i++],R[j]);                 //交换 R[i]和 R[j],交换后 i 执行加 1 操作
          while(i < j&&R[i]< = pivot)
             i++;                               //从左向右扫描,查找第 1 个大于 pivot 的元素
           if(i < j)                            //表示找到了大于 pivot 的元素
             swap(R[i],R[j -- ]);               //交换 R[i]和 R[j],交换后 j 执行减 1 操作
        }
         return j;
  }
```

(3) 划分方法的构造实例。

【例 3-6】 划分的例子(黑体表示基准元素),设定第一个元素 49 作为基准元素。

① 初始序列,如图 3-6 所示。

② 向左扫描,由于 $i<j$ 且 $27<49$,因此,$R[i]$与 $R[j]$交换且 i 后移一位。进行 1 次交换后的状态如图 3-7 所示。

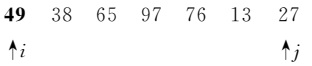

图 3-6 划分的初始状态

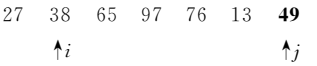

图 3-7 1 次交换后的状态

③ 向右扫描,由于 $i<j$ 且 $38<49$,i 后移 1 位。i 和 j 的位置关系如图 3-8 所示。

④ 向右扫描,由于 $i<j$ 且 $65>49$,因此,$R[i]$与 $R[j]$交换且 j 前移一位。进行 2 次交换后的状态如图 3-9 所示。

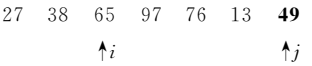

图 3-8 i 后移 1 位后的状态

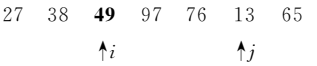

图 3-9 2 次交换后的状态

⑤ 向左扫描,由于 $i<j$ 且 $13<49$,因此,$R[i]$与 $R[j]$交换且 i 后移一位。进行 3 次交换后状态如图 3-10 所示。

⑥ 向右扫描,由于 $i<j$ 且 $97>49$,因此,$R[i]$与 $R[j]$交换且 j 前移一位。进行 4 次交换后的状态如图 3-11 所示。

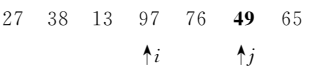

图 3-10 3 次交换后的状态

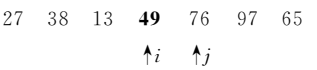

图 3-11 4 次交换后的状态

⑦ 向左扫描,由于 $i<j$ 且 $76>49$,j 前移一位,i 和 j 的位置关系如图 3-12 所示。

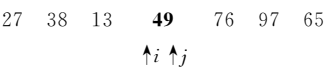

图 3-12 j 前移一位后的状态

⑧ 此时 $i=j$,循环结束,返回 j,即基准元素所处的最终位置。至此,划分过程结束。

3. 快速排序算法描述

以基准元素为标准将待排序序列划分为两个子序列后,对每一个子序列分别采用递归技术来进行排序,最终可获得一个有序的序列。至此,快速排序算法描述如下:

```
void QuickSort(int R[],int low,int high)      //对 R[low..high]快速排序
 {
   int pivotpos;                              //划分后的基准元素所对应的位置
   if(low < high)                             //仅当区间长度大于 1 时才须排序
     {
```

```
            pivotpos = Partition(R,low,high);      //对 R[low..high]做划分
            QuickSort(R,low,pivotpos - 1);         //对左区间递归排序
            QuickSort(R,pivotpos + 1,high);        //对右区间递归排序
        }
    }
```

【例 3-7】 序列[**49**　38　65　97　76　13　27]，经过一次划分后，得到的序列为[27　38　13　**49**　76　97　65]，对两个子序列分别进行递归排序。

接上述划分过程：

① 以 27 为基准元素。

向左扫描，1 次交换之后为[13　38　**27**]，向右扫描，2 次交换之后为[13　**27**　38]，得到序列 [13　27　38]。

② 以 76 为基准元素。

向左扫描，1 次交换之后为[65　97　**76**]，向右扫描，2 次交换之后为[65　**76**　97]，得到序列[65　**76**　97]。

③ 最终得到的有序序列为[13　27　38　49　65　76　97]。

4. 算法分析

快速排序算法的时间主要耗费在划分操作上，并与划分是否平衡密切相关。对于长度为 n 的待排序序列，一次划分算法 Partition 需要对整个待排序序列扫描一遍，其所需的计算时间显然为 $O(n)$。

下面从三种情况来讨论快速排序算法 QuickSort 的时间复杂性。

(1) 最坏时间复杂性。

最坏情况是每次划分选取的基准元素都是在当前待排序序列中的最小(或最大)元素，划分的结果是基准元素左边的子序列为空(或右边的子序列为空)，而划分所得的另一个非空的子序列中元素个数，仅仅比划分前的排序序列中元素个数少一个。

在这样的情况下，快速排序算法 QuickSort 必须做 $n-1$ 次划分，那么算法的运行时间 $T(n)$的递归形式为

$$T(n)=\begin{cases}O(1), & n=1\\ T(n-1)+O(n), & n>1\end{cases}$$

当 $n>1$ 时，有

$$\begin{aligned}T(n)&=T(n-1)+O(n)\\&=T(n-2)+O(n-1)+O(n)\\&=T(n-3)+O(n-2)+O(n-1)+O(n)\\&=\cdots\\&=T(1)+O(2)+\cdots+O(n-1)+O(n)\\&=O(1+2+\cdots+(n-1)+n)\\&=O(n(n+1)/2)\end{aligned}$$

因此,快速排序算法 QuickSort 的最坏时间复杂性为 $O(n^2)$。

如果按上面给出的 Partition 划分算法,每次取当前排序序列的第 1 个元素为基准,那么当序列中的元素已按递增序(或递减序)排列时,每次划分所取的基准元素就是当前序列中值最小(或最大)的元素,则完成快速排序所需的运行时间反而最多。

(2) 最好时间复杂性。

在最好情况下,每次划分所取的基准元素都是当前待排序序列的"中值"元素,划分的结果是基准元素的左、右两个子序列的长度大致相等,此时,算法的运行时间 $T(n)$的递归形式为

$$T(n)=\begin{cases}O(1), & n=1\\ 2T(n/2)+O(n), & n>1\end{cases}$$

解此递归式,可得快速排序算法的最好时间复杂性为 $O(n\log n)$。

注意:用递归树来分析最好情况下的比较次数更简单。因为每次划分后左、右两个子序列的长度大致相等,故递归树的高度为 $O(\log n)$,而递归树每一层上各个结点所对应的划分过程中所需要的元素的比较次数总和不超过 n,故整个排序过程所需要的元素间的比较总次数为 $O(n\log n)$,即时间复杂性为 $O(n\log n)$。

(3) 平均时间复杂性。

在平均情况下,设基准元素的值为第 $k(1\leqslant k\leqslant n)$个,则有

$$T(n)=\frac{1}{n}\sum_{k=1}^{n}(T(n-k)+T(k-1))+n$$

$$=\frac{2}{n}\sum_{k=1}^{n}T(k)+n$$

采用归纳法,最终求得 $T(n)$的数量级也为 $O(n\log n)$。

尽管快速排序的最坏时间为 $O(n^2)$,但就平均性能而言,它是基于元素比较的内部排序算法中速度最快者,快速排序也因此而得名。

(4) 空间复杂性。

由于快速排序算法是递归执行的,需要一个栈来存放每一层递归调用的必要信息,其最大容量应与递归调用的深度一致。最好情况下,若每次划分较为均匀,则递归树的高度为 $O(\log n)$,故递归所需栈空间为 $O(\log n)$。最坏情况下,递归树的高度为 $O(n)$,所需的栈空间为 $O(n)$。平均情况下,所需栈空间为 $O(\log n)$。

5. C++实战

相关代码如下。

```
#include<iostream>
#include<iterator>
#include<algorithm>
using namespace std;
int Partition(int R[],int low,int high)
```

```
{
    int i = low,j = high,pivot = R[low];          //用序列的第一个元素作为基准元素
    while(i < j)                                  //从序列的两端交替向中间扫描,直至 i 等于 j
    {
        while(i < j&&R[j]> = pivot)               //pivot 相当于在位置 i 上
            j -- ;                                //从右向左扫描,查找第 1 个小于 pivot 的元素
        if(i < j)                                 //表示找到了小于 pivot 的元素
            swap(R[i++],R[j]);                    //交换 R[i]和 R[j],交换后 i 执行加 1 操作
        while(i < j&&R[i]< = pivot)
            i++;                                  //从左向右扫描,查找第 1 个大于 pivot 的元素
        if(i < j)                                 //表示找到了大于 pivot 的元素
            swap(R[i],R[j -- ]);                  //交换 R[i]和 R[j],交换后 j 执行减 1 操作
    }
    return j;
}
void QuickSort(int R[ ],int low,int high)         //对 R[low..high]快速排序
{
    int pivotpos;                                 //划分后的基准元素所对应的位置
    if(low < high)                                //仅当区间长度大于 1 时才须排序
    {
        pivotpos = Partition(R,low,high);         //对 R[low..high]做划分
        QuickSort(R,low,pivotpos - 1);            //对左区间递归排序
        QuickSort(R,pivotpos + 1,high);           //对右区间递归排序
    }
}
int main(){
    cout <<"请输入待排序元素个数: ";
    int n;
    cin >> n;
    int * a = new int[n];
    cout <<"请输入待排序元素: ";
    for(int i = 0 ;i < n;i++)
        cin >> a[i];
    QuickSort(a,0,n - 1);
    copy(a,a + n,ostream_iterator < int >(cout," "));  //输出 a 中元素
    delete [ ] a;
}
```

3.6　最接近点对问题


微课视频

在计算机应用中,常用诸如点、圆等简单的几何对象描述现实世界的实体,在涉及这些几何对象的问题时,常需要了解其邻域中其他几何对象的信息。例如,一个控制空中或海上交通的系统就需要了解两个最近的交通工具,以预测可能产生的相撞事故。实质上,这就是要找出空间最接近的一对点问题。因此,研究该问题有很大的实用价值。

1. 问题描述

最接近点对问题要求在给定平面上 n 个点组成的集合 S 中,找出其中 n 个点组成的点

对中距离最近的一对点。

将所给平面上的 n 个点的集合 S 分成规模大致相等的两个子集 S_1 和 S_2。递归求解 S_1 和 S_2 中的最接近点对。在这里，采用分治算法求解的关键在于合并步骤，即由 S_1 和 S_2 中的最接近点对，如何求得 S 中的最接近点对？很明显，集合 S 中的最接近点对或者是子问题 S_1 的解，或者是子问题 S_2 的解，或者是一个点在 S_1 中、一个点在 S_2 中的情况组成的最接近点对。S_1 和 S_2 中的最接近点对可以用递归求得。但是一个点在 S_1 中、另一个点在 S_2 中的情况组成的最接近点对怎么求得呢？

为了讲明白这个问题，先来考虑一维情形。

2. 一维情形

(1) 算法思想。

令平面上的点的纵坐标全部等于 0，S 中的 n 个点退化为 X 轴上的 n 个实数 $x_1, x_2, \cdots, x_n$。最接近点对即为这 n 个实数中相差最小的两个实数，每个实数对应数轴上的一个点。用 x_m 将 $x_1, x_2, \cdots, x_n$ 分成两部分 S_1 和 S_2，这种情况下如何找出 S_1 中的一个点 p 和 S_2 中的一个点 q 组成的点对(p,q)呢？令 S_1 中最接近点对的距离为 d_1，S_2 中的最接近点对的距离为 d_2，取 $d=\min\{d_1,d_2\}$。如果 S 的最接近点对是(p,q)，则$|x_p-x_q|$小于或等于 d。由于 $p\in S_1, q\in S_2$，所以二者与 x_m 的距离不超过 d，即 $m-d<x_p\leqslant m, m<x_q\leqslant m+d$。

在 S_1 中任何两个点的距离均小于或等于 d_1，同样也小于或等于 d。因此，在区域$(m-d,m]$中至多包含 S_1 中的一个点且是 S_1 中最右边的点，即 S_1 中 X 坐标的最大点。同理，在区域$(m,m+d]$中至多包含 S_2 中的一个点且是 S_2 中最左边的点，即 S_2 中 X 坐标的最小点。可见，线性时间就可以找出 S_1 中的 p 和 S_2 中的 q，因此线性时间就可以完成子问题的解合并成原问题解的过程。

(2) 算法设计。

步骤 1：基于平衡子问题的思想，通过 S 中各点的 X 坐标的中位数 x_m 将 S 划分为两个子集 $S_1=\{i\mid x_i\leqslant x_m\}$和 $S_2=\{i\mid x_i>x_m\}$，如图 3-13 所示。

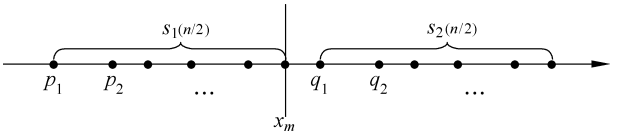

图 3-13　点集 S 划分成 S_1 和 S_2

步骤 2：递归地在 S_1 中找出其最接近点对(p_i,p_j)，距离为 d_1。

步骤 3：递归地在 S_2 中找出其最接近点对(q_i,q_j)，距离为 d_2。

步骤 4：取 $d=\min\{d_1,d_2\}$。

步骤 5：求 S_1 中的最大值 x_p；求 S_2 中的最小值 x_q。

步骤 6：计算 $d_3=x_q-x_p$。

步骤 7：取 $\min\{d,d_3\}$且记录最接近点对的坐标，即求得 S 中的最接近点对。

(3) 算法描述。

```
bool CPAIR1(POINT S[], double &d,double &p,double &q)
{
double p1,p2,q1,q2,p3,q3;            //用来记录 3 种情况的最接近点对
n = |S|;
if (n<2)
      {d = ∞;return false;}
else if(n == 2)
       {
         d = |x[2] - x[1]|;          // x[1..n]存放的是 S 中 n 个点的坐标
         p = x[2];q = x[1];
         return true;
       }
m = S 中各点的横坐标值的中位数;
构造 S1 和 S2;
CPAIR1(S1,d1,p1,p2);
CPAIR1(S2,d2,q1,q2);
p3 = max(S1);
q3 = min(S2);
double d3 = q3 - p3;
if(d1 <= d2 && d1 <= d3)
   {d = d1;p = p1;q = p2;}
if(d1 < d2)
   {d = d2;p = q1;q = q2;}
else
   {d = d3;p = p3;q = q3;}
  return true;
}
```

(4) 算法分析。

假设用 $T(n)$表示从 n 个点中找出最接近点对所耗的时间，则每个子问题所耗的时间为 $T(n/2)$；找出中位数可以在线性时间 $O(n)$实现；合并子问题的解所耗时间为线性时间 $O(n)$，故一维情形下时间的递归表达式为

$$T(n)=\begin{cases}O(1), & n<3\\ 2T(n/2)+O(n), & n\geqslant 3\end{cases}$$

解此递归方程可得 $T(n)=O(n\log n)$。

在一维情形下，更容易想到的解决方法是先排序，再线性扫描就可以了。但是这种方法不容易推广到二维情形。上述分治思想的解决方法很容易推广到二维情形。

(5) C++实战。

相关代码如下。

```
#include<iostream>
#include<cmath>
#include<cfloat>
```

```
#include<algorithm>
#define INF DBL_MAX
using namespace std;
bool CPAIR1(double S[], int left,int right,double &d,double &p,double &q)
{
    double p1,p2,q1,q2,p3,q3;       //用来记录3种情况的最接近点对
    int n=right-left+1;
    if (n<2)
    {
        d=INF;
        return false;
    }
    else if(n==2)
    {
        d=abs(S[right]-S[left]); // x[1..n]存放的是S中n个点的坐标
        p=S[left];
        q=S[right];
        return true;
    }
    int m=(left+right)/2;
    double d1,d2;
    CPAIR1(S,left,m,d1,p1,q1);
    CPAIR1(S,m+1,right,d2,p2,q2);
    p3=S[m];
    q3=S[m+1];
    double d3=q3-p3;
    if(d1<=d2 and d1<=d3)
    {
        d=d1;
        p=p1;
        q=q1;
    }
    else if(d2<d3){
        d=d2;
        p=p2;
        q=q2;
    }
    else{
        d=d3;
        p=p3;
        q=q3;
    }
    return true;
}
int main(){
    int n = 13;
    double points[n]={1,3,-1,7,5,-2,-6,4,15,10,0.6,0,-0.5};
    sort(points,points+n);
    double d = INF;
    double p,q;
```

```
    CPAIR1(points,0,n-1,d,p,q);
    cout <<"最接近点对为: "<< endl;
    cout << p <<","<< q << endl;
    cout <<"最接近点对距离 d = "<< d << endl;
}
```

3. 二维情形

(1) 算法思想。

选取一垂直线 l，$x=x_m$ 作为分割线，其中 x_m 为 S 中各点 X 坐标的中位数。由此将 S 分割为 S_1 和 S_2。递归地在 S_1 和 S_2 上找出其最小距离 d_1 和 d_2，并设 $d=\min\{d_1,d_2\}$，S 中的最接近点对或者是 d 所对应的点对，或者是 S_1 中的某个点 p 和 S_2 中的某个点 q 组成的点对(p,q)。如何找出点对(p,q)呢？如果$|p-q|$小于 d，则 p 点分布在 P_1 带形区域内(左虚线和分割线 l 所夹的区域)，q 点分布在 P_2 带形区域内(右虚线和分割线 l 所夹的区域)，如图 3-14 所示。

另外，对于 P_1 中任意一点 p，与它距离小于 d 的点分布在以 p 点为圆心、以 d 为半径的圆内。因此，与点 p 构成最接近点对的 P_2 中的点一定落在一个 $d\times 2d$ 的矩形 R 中，如图 3-15 所示。

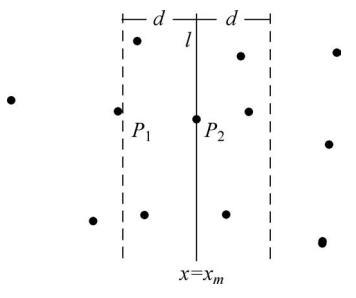

图 3-14　距离直线 l 的距离小于 d 的所有点

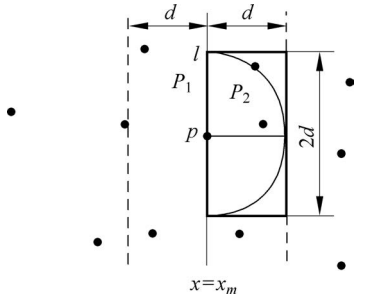

图 3-15　包含点 q 的 $d\times 2d$ 矩形区 R

在矩形 R 中，有多少个点可能与 P_1 中的点 p 构成最接近点对呢？由 d 的意义可知，矩形 R 中任何两个 S 中的点的距离都大于或等于 d。由此可知，至少可以将 $d\times 2d$ 的矩形 R 分割成如图 3-16 所示的 6 部分，其中任何一部分包含 P_2 中的点最多有一个，如果包含 P_2 中的两个点，则这两个点的距离最大为 $\sqrt{(d/2)^2+(2d/3)^2}=5d/6<d$。这与 P_2 中任何两个 S 中的点的距离都大于或等于 d 矛盾。因此，在矩形 R 中最多只有 6 个 P_2 中的点与 p 构成最接近点对。故在分治法的合并步骤中最多只需要检查 $6\times n/2=3n$ 个候选者。

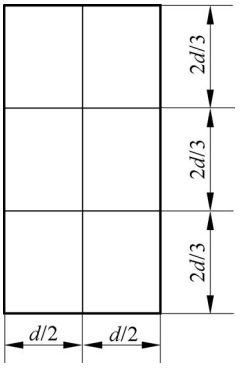

图 3-16　矩形 R 中点的稀疏性

针对 P_1 中的任意一点 p，检查 P_2 中的哪 6 个点，从而可以找出最接近点对呢？为了确切地知道要检查哪 6 个点，可以将 p

和P_2中所有点投影到垂直线l上。由于能与p点一起构成最接近点对候选者的P_2中的点一定在矩形R中，所以它们在直线l上的投影点与p在l上的投影点的距离小于d。由上面的分析可知，这种投影点最多只有6个。因此，若将P_1和P_2中的点按其Y坐标排好序，则对P_1中所有点，对排好序的点做一次扫描，就可以找出所有最接近点对的候选者。对P_1中每一点最多只要检查P_2中排好序的相继6个点。

(2) 算法描述。

首先定义POINT结构体，除X坐标和Y坐标外，重载了操作符<，定义排序操作按照X坐标升序排序，X坐标相同时，按照Y坐标升序排序。POINT定义如下：

```
typedef struct POINT
{
    double x;                                   //点的横坐标
    double y;                                   //点的纵坐标
    bool operator <(const POINT &a)const{
        if(a.x == x)
            return y < a.y;
        else
            return x < a.x;
    }
};
```

在最接近点对问题的求解中，需要用到两个点之间的欧几里得距离。求解两点间距离的算法描述如下：

```
double Distance(POINT u, const POINT v)         //计算平面上任意两点 u 和 v 之间的距离
{
    double dx = u.x - v.x, dy = u.y - v.y;
    return sqrt(dx * dx + dy * dy);
}
```

求解最接近点对问题的算法描述如下：

```
bool CPAIR1(POINT S[], int left,int right,double &d,POINT &p,POINT &q)
{
    POINT p1,p2,q1,q2,p3,q3;                    //用来记录 3 种情况的最接近点对
    int n = right - left + 1;
    if (n < 2)
    {
        d = INF;
        return false;
    }
    else if(n == 2)
    {
        d = Distance(S[0],S[1]);
```

```
            p = S[left];
            q = S[right];
            return true;
        }
        int m = (left + right)/2;
        double d1,d2;
        CPAIR1(S,left,m,d1,p1,q1);
        CPAIR1(S,m + 1,right,d2,p2,q2);
        d = min(d1,d2);
        double d3 = INF;
        for(int i = left;i <= m;i++)
            if((S[m].x - S[i].x) < d)
                for(int j = m + 1;j <= right;j++)
                    if(((S[j].x - S[i].x)< d)&&((S[j].y - S[i].y)< d))
                    {
                        double dd = Distance(S[i],S[j]);
                        if (dd < d3){
                            d3 = dd;
                            p3 = S[i];
                            q3 = S[j];
                        }
                    }
        if(d1 <= d2 and d1 <= d3)
        {
            d = d1;
            p = p1;
            q = q1;
        }
        else if(d2 < d3){
            d = d2;
            p = p2;
            q = q2;
        }
        else{
            d = d3;
            p = p3;
            q = q3;
        }
        return true;
    }
```

(3) 算法分析。

假设用 $T(n)$表示从 n 个点中找出最接近点对所耗的时间,则每个子问题所耗的时间为 $T(n/2)$。找出中位数可以用线性时间选择算法来实现,耗时 $O(n)$。合并子问题的解所耗时间为线性时间 $O(n)$。故二维情形下的时间的递归表达式为:

$$T(n)=\begin{cases}O(1), & n<3\\ 2T(n/2)+O(n), & n\geqslant 3\end{cases}$$

由此可得 $T(n)=O(n\log n)$。

(4) C++实战。

相关代码如下。

```
#include<iostream>
#include<cmath>
#include<cfloat>
#include<algorithm>
#define INF DBL_MAX
using namespace std;
typedef struct POINT
{
    double x;                          //点的横坐标
    double y;                          //点的纵坐标
    bool operator <(const POINT &a)const{
        if(a.x==x)
            return y<a.y;
        else
            return x<a.x;
    }
} POINT;

double Distance(POINT u,POINT v){      //计算平面上任意两点 u 和 v 之间的距离,u,v 为元组(x,y)
    double dx = u.x - v.x;
    double dy = u.y - v.y;
    return sqrt(dx*dx+dy*dy);
}
bool CPAIR1(POINT S[], int left,int right,double &d,POINT &p,POINT &q)
{
    POINT p1,p2,q1,q2,p3,q3;           //用来记录 3 种情况的最接近点对
    int n=right-left+1;
    if (n<2)
    {
        d=INF;
        return false;
    }
    else if(n==2)
    {
        d=Distance(S[0],S[1]);
        p=S[left];
        q=S[right];
        return true;
    }
    int m = (left+right)/2;
    double d1,d2;
    CPAIR1(S,left,m,d1,p1,q1);
    CPAIR1(S,m+1,right,d2,p2,q2);
    d = min(d1,d2);
    double d3 = INF;
```

```
    for(int i = left;i < = m;i++)
        if((S[m].x - S[i].x) < d)
            for(int j = m + 1;j < = right;j++)
                if(((S[j].x - S[i].x)< d)&&((S[j].y - S[i].y)< d))
                {
                    double dd = Distance(S[i],S[j]);
                    if (dd < d3){
                        d3 = dd;
                        p3 = S[i];
                        q3 = S[j];
                    }
                }
    if(d1 < = d2 and d1 < = d3)
    {
        d = d1;
        p = p1;
        q = q1;
    }
    else if(d2 < d3){
        d = d2;
        p = p2;
        q = q2;
    }
    else{
        d = d3;
        p = p3;
        q = q3;
    }
    return true;
}
int main(){
    int n = 12;
    POINT points[12] = {{0,1},{3,2},{4,3},{5,1},{1,2},{2,1},{6,2},{7,2},{8,3},{4,5},{9,
0},{6,4}};
    sort(points,points + n);
    double d = INF;
    POINT p,q;
    CPAIR1(points,0,n - 1,d,p,q);
    cout <<"最接近点对为: "<< endl;
    cout <<"("<< p.x <<","<< p.y <<")"<<" -- -- "<<"("<< q.x <<","<< q.y <<")"<< endl;
    cout <<"最接近点对距离 d = "<< d << endl;
}
```

拓展知识：禁忌搜索算法

禁忌搜索(Tabu Search 或 Taboo Search,TS)的思想最早由 Fred Glover(美国工程院院士,科罗拉多大学教授)于 1986 年提出,它是对局部领域搜索的一种扩展,是一种全局逐

步寻优算法,是对人类智力过程的一种模拟。TS 算法通过引入一个灵活的存储结构和相应的禁忌准则来避免迂回搜索,并通过藐视准则来赦免一些被禁忌的优良状态,进而保证多样化的有效探索以最终实现全局优化。

禁忌 Tabu 一词来源于汤加语,是波利尼西亚的一种语言,其含义是保护措施或者危险禁止,这正符合禁忌搜索算法的主题:一方面,当沿着产生相反结果的道路走下去时,也不会陷入一个圈套而导致无处可逃;另一方面,在必要情况下,保护措施允许被淘汰,也就是说,某种措施被强制运用时,禁忌条件就宣布无效。

与模拟退火和遗传算法相比,TS 是又一种搜索特点不同的 meta-heuristic 算法。迄今为止,TS 算法在组合优化、生产调度、机器学习、电路设计和神经网络等领域取得了很大的成功,近年来又在函数全局优化方面得到较多的研究,并大有发展的趋势。

1. 禁忌搜索算法的基本思想

给定一个当前解(初始解)和它的邻域,然后在当前解的邻域中确定若干个候选解;若最佳候选解所对应的目标值优于"Best so far"状态,则忽视其禁忌特性,用其替代当前解和"Best so far"状态,并将相应的对象加入禁忌表,同时修改禁忌表;若不存在上述候选解,则在候选解中选择非禁忌的最佳状态为新的当前解,而无视它与当前解的优劣,同时将相应的对象加入禁忌表,并修改禁忌表;如此重复上述迭代搜索过程,直至满足停止准则。

2. 禁忌搜索算法的构成

从禁忌搜索算法的思想中,容易看出该算法涉及的概念包括邻域、禁忌表、禁忌长度、候选解、藐视准则等。它们构成了算法,对整个禁忌搜索起着关键的作用。

(1) 邻域移动。

禁忌搜索算法采用了邻域选优的搜索策略,即通过对当前解的"移动"产生其邻域解,选择优秀的邻域解作为当前解,再进行邻域搜索。可见,邻域移动是保证产生好的解和算法搜索速度的最重要因素之一。因此,如何设计有效合理的"移动"方法来产生邻域解是算法的核心。

通常,邻域移动定义的方法很多,对于不同的问题应采用不同的定义方法。通过移动,目标函数值将产生变化,移动前后的目标函数值之差,称为移动值。如果移动值是非负的,则称此移动为改进移动;否则称作非改进移动。最好的移动不一定是改进移动,也可能是非改进移动,这一点就保证搜索陷入局部最优时,禁忌搜索算法能自动跳出局部最优。

(2) 禁忌表。

禁忌表是防止在搜索过程中出现循环,避免陷入局部最优,它通常记录最近接受的若干次移动,在一定次数内禁止再次被访问,超过了一定次数之后,这些移动从禁忌表中退出,又可以重新被访问。禁忌表是禁忌搜索算法的核心,它的功能和人类的短期记忆功能十分相似。

禁忌表包含两个很重要的概念。一是禁忌对象,就是放入禁忌表中的那些元素,而禁忌的目的就是避免迂回搜索,尽量搜索一些有效的途径。禁忌对象的选择十分灵活,可以是最近访问过的点、状态、状态的变化以及目标函数值等。二是禁忌长度(Tabu size),就是禁忌

表的大小。一个禁忌对象进入禁忌表后，只有经过一个确定的迭代次数或者满足一定的条件，才能从禁忌表中退出。也就是说，在当前迭代之后的确定次迭代中或者未达到一定的条件时，这个发生不久的相同操作是被禁止的。容易知道，禁忌表长度越小，计算时间和存储空间越少，这是任何一个算法都希望的。但是，如果禁忌表过小，会造成搜索的循环，这又是要避免的。禁忌长度不但影响了搜索的时间，还直接关系着搜索的两个关键策略：局部搜索策略和广域搜索策略。如果禁忌表比较长，便于在更广阔的区域搜索，广域搜索能力比较好；而禁忌表比较短，则使得搜索在小的范围进行，局部搜索性能比较好。禁忌长度的设定要依据问题的规模、邻域的大小来确定，从而达到平衡这两种搜索策略的目的。

(3) 候选解选择策略。

候选解选择策略即择优规则，是对当前的邻域移动选择一个移动而采用的准则。择优规则可以采用多种策略，不同的策略对算法的性能影响不同。一个好的选择策略应该是既保证解的质量又保证计算速度。当前采用最广泛的两类策略是最好改进解优先策略和第一个改进解优先策略。最好改进解优先策略就是对当前邻域中选择移动值最好的移动产生的解，作为下一次迭代的开始，而第一个改进解优先策略是搜索邻域移动时选择第一改进当前解的邻域移动产生的解作为下一次迭代的开始。最好改进解优先策略相当于寻找最陡的下降，这种择优规则效果比较好，但是它需要更多的计算时间；而最快的下降对应寻找第一个改进解的移动，由于它无须搜索整个一次邻域移动，所以它所花费的计算时间较少，对于比较大的邻域，往往比较适合。

(4) 藐视准则。

在某些特定的条件下，不管某个移动是否在禁忌表中，都接受这个移动，并更新当前解和历史最优解。这个移动满足的这个特定条件，称为渴望水平(aspiration level)，或称为破禁水平、特赦准则、藐视准则等。

藐视准则的设定有多种形式，通常选取当前迭代之前所获得的最好解的目标值或此移动禁忌时的目标值作为藐视准则函数。

(5) 算法停止准则。

算法的停止准则一般有以下三类：一是给定外循环的最大次数，达到该次数时，算法终止；二是如果当前最优解连续 K 次相同，则算法终止，K 是一个给定的正整数，表示算法已经收敛，无须再继续；三是目标值控制规则，给定优化问题(目标最小化)的一个下界和一个误差值，当算法得到的目标值同下界之差小于给定的误差值时，算法终止。

另外，短期记忆能够用来避免最近所做的一些移动被重复，但是在很多的情况下短期记忆并不足以把算法搜索带到能够改进解的区域。因此在实际应用中常常短期记忆与长期记忆相结合使用，以保持局部的强化和全局多样化之间的平衡，即在加强搜索到较优解的同时还能把搜索带到未搜索过的区域。

3. 禁忌搜索算法的求解过程

步骤 1：给定算法参数，选定一个初始解 x，置禁忌表 H 为空。

步骤 2：若满足停止规则，停止计算，输出优化结果；否则继续以下步骤。

步骤 3：利用当前解 x 的邻域函数产生其所有邻域解，并从中确定若干候选解。

步骤 4：判断候选解是否满足藐视准则？若是，则用满足藐视准则的最佳状态 y 替代 x 成为新的当前解，即 $x=y$，并用与 y 对应的禁忌对象替换最早进入禁忌表的禁忌对象，同时用 y 替换 Best so far 状态，然后转到步骤 2；否则，继续以下步骤。

步骤 5：判断候选解对应的各对象的禁忌属性，选择候选解集中非禁忌对象对应的最佳状态为新的当前解，同时用与之对应的禁忌对象替换最早进入禁忌表的禁忌对象元素。转到步骤 2。

在步骤 3 中，x 的邻域中满足禁忌要求的元素包含两类：一类是那些没有被禁忌的元素；另一类是可以被解除禁忌的元素。此时应用藐视准则使某些状态解禁，就是对优良状态的奖励，对禁忌策略的放松，以实现更高效的优化性能。

4. 禁忌搜索算法性能的简单分析

禁忌搜索算法通过对当前解进行“移动”操作来产生当前解邻域中可行的邻域解，取其最好的邻域解作为新的当前解来完成解空间的局部搜索。但该算法的局部搜索能力与其每次“移动”操作所产生的邻域解的数目密切相关。邻域解数目太少，对当前解邻域空间的搜索也就太少，这将影响该算法的局部优化性能；邻域解数目太多，则将增加算法的执行时间。

用于求解具体的组合优化问题时，禁忌搜索算法中当前解的邻域解是通过在当前解中随机选择两个交换的位置产生的，这样做容易导致局部搜索具有分散性。

该局部搜索的分散性如图 3-17 所示。

局部搜索的分散性似乎有利于对局部空间进行全方位搜索，但当邻域解数目一定及邻域解中的优秀解较为集中时，这种分散性反而影响了局部优化的性能。通常采用的解决方案是：邻域解的产生不是随机的，而是以某种机制来确定其产生，在“移动”过程中以某种概率为依据来找出进行“移动”操作的点，由此产生的禁忌搜索算法的当前解的邻域解大部分集中在邻域空间的特定区域内，而小部分分散在邻域空间的其他区域，这种现象称为禁忌搜索算法的集中性。

该局部搜索的集中性如图 3-18 所示。

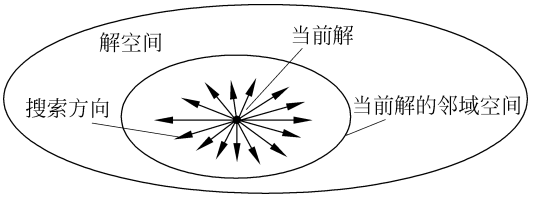

图 3-17　局部搜索的分散性

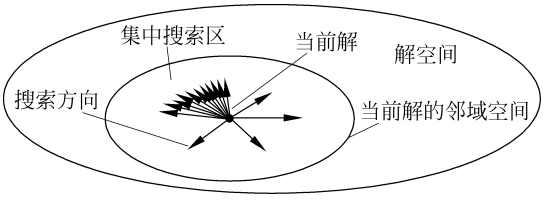

图 3-18　局部搜索的集中性

局部搜索的集中性显著增加了发现局部最优解的可能性，提高了局部搜索的性能。

禁忌搜索算法在搜索过程中允许接受劣解，使得该算法具有很强的“爬山”能力，可以跳出局部最优解。此外，为了避免搜索路径的往返重复，该算法使用“Tab 表”来记录搜索过程的历史信息，这可在一定程度上避开局部极值点，开辟新的搜索区域。因此，虽然禁忌搜索

算法并不能从理论上保证一定能找到全局最优解，但却能在较短的时间内找到非常优秀的近似最优解。

该算法的一大亮点为：允许在搜索过程中接受劣解，这样可使算法跳出局部最优解，扩大搜索空间。

本章习题

3-1　简述分治算法的基本思想和求解步骤。

3-2　在一个划分成网格的操场上，n 名士兵散乱地站在网格点上。网格点由整数坐标(x,y)表示。士兵们可以沿网格边上、下、左、右移动一步，但在同一时刻任一网格点上只能有一名士兵。按照军官的命令，士兵们要整齐地列成一个纵队，即排列成$(x,y),(x+1,y),\cdots,(x+n-1,y)$。如何选择 x 和 y 的值才能使士兵们以最少的总移动步数排成一列。

3-3　某石油公司计划建造一条由东向西的主输油管道。该管道要穿过一个有 n 口油井的油田。从每口油井都要有一条输油管道沿最短路径(或南或北)与主管道相连。如果给定 n 口油井的位置，即它们的 x 坐标(东西向)和 y 坐标(南北向)，应如何确定主管道的最优位置，从而使得各油井到主管道的输油管道长度总和最小？证明可在线性时间内确定主管道的最优位置。

3-4　若二分查找变为三分查找，即从 $a_1<a_2<\cdots<a_n$ 序列中寻找元素 x。方法如下：先与 $a_{\frac{n}{3}}$ 比较，若 $x>a_{\frac{n}{3}}$，则与 $a_{\frac{2n}{3}}$ 比较，总之，使余下的序列约有 $n/3$ 个元素。试讨论其复杂性。

3-5　设 $a[0:n]$是一个已排好序的数组。请改写二分查找算法，使得当搜索元素 x 不在数组中时，返回小于 x 的最大元素位置 j 和大于 x 的最小元素位置 i。当搜索元素 x 在数组中时，i 和 j 相同，均为 x 在数组中的位置。

3-6　设 n 个不同的排好序的整数存于数组 $T[0:n-1]$中。若存在一个下标 i，$0\leqslant i<n$，使得 $T[i]$等于 i，设计一个有效算法找到这个下标 i；要求算法在最坏情况下的计算时间为 $O(\log n)$。

3-7　设 n 个元素存于数组 $T[0:n-1]$。对任一元素 x，设 $s(x)=\{i\mid T[i]=x\}$；当 $|s(x)|>n/2$ 时，称 x 为 T 的主元素，设计一个线性时间算法，确定 T 中是否存在主元素。

3-8　用快速排序算法对如下数据进行排序：45,23,65,57,18,2,90,84,12,76。说明划分方法的具体过程。

3-9　应用分治算法完成下面的整数乘法计算：2348×3825。

3-10　在一个由元素组成的表中，出现次数最多的元素称为众数。试写一个寻找众数的算法，并分析其计算复杂性。

3-11　设有 n 个运动员要进行网球循环赛。设计一个满足以下要求的比赛日程表。

(1) 每个选手必须与其他 $n-1$ 个选手各赛一次。

(2) 每个选手一天只能赛一次。

(3) 当 n 是偶数时，循环赛进行 $n-1$ 天。当 n 是奇数时，循环赛进行 n 天。