第 9 章 排序

启示

排序：

假设含有n个记录的序列为$\{r_1, r_2, \cdots\cdots, r_n\}$，其相应的关键字分别为$\{k_1, k_2, \cdots\cdots, k_n\}$，需确定 1，2，……，n的一种排列$p_1, p_2, \cdots\cdots, p_n$，使其相应的关键字满足$k_{p1} \leqslant kp_2 \leqslant \cdots\cdots \leqslant k_{pn}$（非递减或非递增）关系，即使得序列成为一个按关键字有序的序列$\{r_{p1}, r_{p2}, \cdots\cdots, r_{pn}\}$，这样的操作就称为排序。

9.1 开 场 白

大家好！你们有没有在网上买过东西啊？

嗯？居然还有人说没有。呵呵，在座的都是大学生，应该很多同学都有过网购的经历。哪怕真的没有，也看到或听到过一些，现在网上购物已经相对成熟，对用户来说带来了很大的方便。

假如我想买一台 iPhone4 的手机，于是上了某电子商务网站去搜索。可搜索后发现（如图 9-1-1 所示），有 8863 个相关的物品，如此之多，这叫我如何选择。我其实是想买便宜一点的，但是又怕遇到骗子，想找信誉好的商家，如何做？

图 9-1-1

下面的有些购物达人给我出主意了，排序呀。对呀，排序就行了（如图 9-1-2 所示）。我完全可以根据自己的需要对搜索到的商品进行排序，比如按信用从高到低、再按价格从低到高，将最符合我预期的商品列在前面，最终找到我愿意购买的商家，非常的方便。

图 9-1-2

网站是如何做到快速地将商品按某种规则有序的呢？这就是我们今天要讲解的重

要课题——排序。

9.2 排序的基本概念与分类

排序是我们生活中经常会面对的问题。同学们做操时会按照从矮到高排列；老师查看上课出勤情况时，会按学生学号顺序点名；高考录取时，会按成绩总分降序依次录取等。那排序的严格定义是什么呢？

假设含有n个记录的序列为{r_1,r_2,……,r_n}，其相应的关键字分别为{k_1,k_2,……,k_n}，需确定 1，2，……，n的一种排列p_1,p_2,……,p_n，使其相应的关键字满足$k_{p1}\leqslant k_{p2}\leqslant\cdots\cdots\leqslant k_{pn}$（非递减或非递增）关系，即使得序列成为一个按关键字有序的序列{r_{p1},r_{p2},……,r_{pn}}，这样的操作就称为排序。

注意我们在排序问题中，通常将数据元素称为记录。显然我们输入的是一个记录集合，输出的也是一个记录集合，所以说，可以将排序看成是线性表的一种操作。

排序的依据是关键字之间的大小关系，那么，对同一个记录集合，针对不同的关键字进行排序，可以得到不同序列。

这里关键字k_i可以是记录r的主关键字，也可以是次关键字，甚至是若干数据项的组合。比如我们某些大学为了选拔在主科上更优秀的学生，要求对所有学生的所有科目总分降序排名，并且在同样总分的情况下将语数外总分做降序排名。这就是对总分和语数外总分两个次关键字的组合排序。如图 9-2-1 所示，对于组合排序的问题，当然可以先排序总分，若总分相等的情况下，再排序语数外总分，但这是比较土的办法。我们还可以应用一个技巧来实现一次排序即完成组合排序问题，例如，把总分与语数外都当成字符串首尾连接在一起（注意语数外总分如果位数不够三位，需要在前面补零），很容易可以得到令狐冲的“753229”要小于张无忌的“753236”，于是张无忌就排在了令狐冲的前面。

编号	姓名	语	数	外	物	化	历	政	生	地	总分	语数外
1	令狐冲	85	60	84	86	89	94	87	83	85	753	229
2	郭靖	66	64	56	45	76	56	56	78	76	573	186
3	杨过	85	78	64	68	84	78	73	88	64	682	227
4	张无忌	84	85	67	90	87	83	94	79	84	753	236

排序前

编号	姓名	语	数	外	物	化	历	政	生	地	总分	语数外
4	张无忌	84	85	67	90	87	83	94	79	84	753	236
1	令狐冲	85	60	84	86	89	94	87	83	85	753	229
3	杨过	85	78	64	68	84	78	73	88	64	682	227
2	郭靖	66	64	56	45	76	56	56	78	76	573	186

总分排名再语数外排名

图 9-2-1

从这个例子也可看出，多个关键字的排序最终都可以转化为单个关键字的排序，因此，我们这里主要讨论的是单个关键字的排序。

9.2.1 排序的稳定性

也正是由于排序不仅是针对主关键字，那么对于次关键字，因为待排序的记录序列中可能存在两个或两个以上的关键字相等的记录，排序结果可能会存在不唯一的情况，我们给出了稳定与不稳定排序的定义。

假设$k_i=k_j$（$1\leqslant i\leqslant n,1\leqslant j\leqslant n,i\neq j$），且在排序前的序列中$r_i$领先于$r_j$（即i<j）。如果排序后$r_i$仍领先于$r_j$，则称所用的排序方法是稳定的；反之，若可能使得排序后的序列中r_j领先r_i，则称所用的排序方法是不稳定的。如图 9-2-2 所示，经过对总分的降序排序后，总分高的排在前列。此时对于令狐冲和张无忌而言，未排序时是令狐冲在前，那么它们总分排序后，分数相等的令狐冲依然应该在前，这样才算是稳定的排序，如果他们二者颠倒了，则此排序是不稳定的了。只要有一组关键字实例发生类似情况，就可认为此排序方法是不稳定的。排序算法是否稳定的，要通过分析后才能得出。

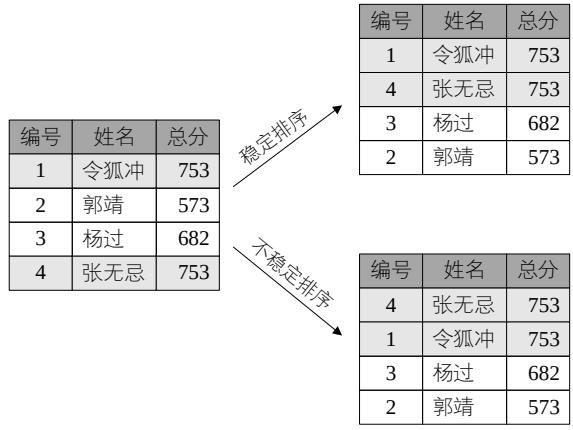


图 9-2-2

9.2.2 内排序与外排序

根据在排序过程中待排序的记录是否全部被放置在内存中，排序分为：内排序和外排序。

内排序是在排序整个过程中，待排序的所有记录全部被放置在内存中。外排序是由于排序的记录个数太多，不能同时放置在内存，整个排序过程需要在内外存之间多次交换数据才能进行。我们这里主要就介绍内排序的多种方法。

对于内排序来说，排序算法的性能主要是受 3 个方面影响：

1. 时间性能

排序是数据处理中经常执行的一种操作，往往属于系统的核心部分，因此排序算法的时间开销是衡量其好坏的最重要的标志。在内排序中，主要进行两种操作：比较和移动。比较指关键字之间的比较，这是要做排序最起码的操作。移动指记录从一个位置移动到另一个位置，事实上，移动可以通过改为记录的存储方式来予以避免（这个我们在讲解具体的算法时再谈）。总之，高效率的内排序算法应该是具有尽可能少的关键字比较次数和尽可能少的记录移动次数。

2. 辅助空间

评价排序算法的另一个主要标准是执行算法所需要的辅助存储空间。辅助存储空间是除了存放待排序所占用的存储空间之外，执行算法所需要的其他存储空间。

3. 算法的复杂性

注意这里指的是算法本身的复杂度，而不是指算法的时间复杂度。显然算法过于复杂也会影响排序的性能。

根据排序过程中借助的主要操作，我们把**内排序分为：插入排序、交换排序、选择排序和归并排序。**可以说，这些都是比较成熟的排序技术，已经被广泛地应用于许许多多的程序语言或数据库当中，甚至它们都已经封装了关于排序算法的实现代码。因此，我们学习这些排序算法的目的更多并不是为了去在现实中编程排序算法，而是通过学习来提高我们编写算法的能力，以便于去解决更多复杂和灵活的应用性问题。

本章一共要讲解七种排序的算法，按照算法的复杂度分为两大类，冒泡排序、简单选择排序和直接插入排序属于简单算法，而希尔排序、堆排序、归并排序、快速排序属于改进算法。后面我们将依次讲解。

9.2.3 排序用到的结构与函数

为了讲清楚排序算法的代码，我先提供一个用于排序用的顺序表结构，此结构也将用于之后我们要讲的所有排序算法。

```
#define MAXSIZE 10       /* 用于要排序数组个数最大值，可根据需要修改 */
typedef struct
{
    int r[MAXSIZE+1];    /* 用于存储要排序数组，r[0]用作哨兵或临时变量 */
    int length;          /* 用于记录顺序表的长度 */
}SqList;
```

另外，由于排序最最常用到的操作是数组两元素的交换，我们将它写成函数，在之后的讲解中会大量的用到。

```
/* 交换 L 中数组 r 的下标为 i 和 j 的值 */
void swap（SqList *L,int i,int j）
{
    int temp=L->r[i];
    L->r[i]=L->r[j];
    L->r[j]=temp;
}
```

好了，说了这么多，我们来看第一个排序算法。

9.3 冒泡排序

无论你学习哪种编程语言，在学到循环和数组时，通常都会介绍一种排序算法来作为例子，而这个算法一般就是冒泡排序。并不是它的名称很好听，而是说这个算法的思路最简单，最容易理解。因此，哪怕大家可能都已经学过冒泡排序了，我们还是从这个算法开始我们的排序之旅。

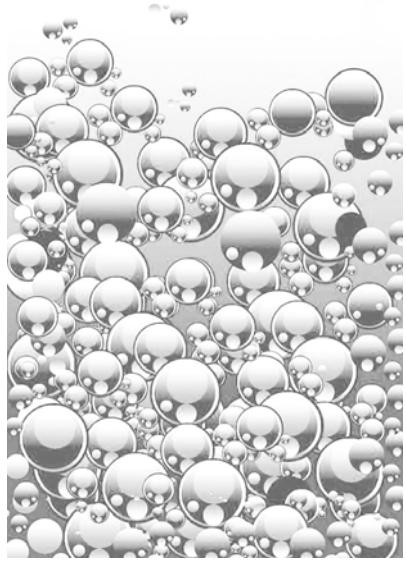


图 9-3-1

9.3.1 最简单排序实现

冒泡排序（Bubble Sort）一种交换排序，它的基本思想是：两两比较相邻记录的关键字，如果反序则交换，直到没有反序的记录为止。冒泡的实现在细节上可以很多种变化，我们将分别就 3 种不同的冒泡实现代码，来讲解冒泡排序的思想。这里，我们就先来看看比较容易理解的一段。

```
/* 对顺序表 L 作交换排序（冒泡排序初级版） */
void BubbleSort0（SqList *L）
{
    int i,j;
    for（i=1;i<L->length;i++）
    {
        for（j=i+1;j<=L->length;j++）
        {
            if（L->r[i]>L->r[j]）
            {
                swap（L,i,j）; /* 交换 L->r[i]与 L->r[j]的值 */
            }
        }
    }
}
```

这段代码严格意义上说，不算是标准的冒泡排序算法，因为它不满足“两两比较相邻记录”的冒泡排序思想，它更应该是最最简单的交换排序而已。它的思路就是让每一个关键字，都和它后面的每一个关键字比较，如果大则交换，这样第一位置的关键字在一次循环后一定变成最小值。如图 9-3-2 所示，假设我们待排序的关键字序列是{9,1,5,8,3,7,4,6,2}，当 i=1 时，9 与 1 交换后，在第一位置的 1 与后面的关键字比较都小，因此它就是最小值。当 i=2 时，第二位置先后由 9 换成 5，换成 3，换成 2，完成了第二小的数字交换。后面的数字变换类似，不再介绍。

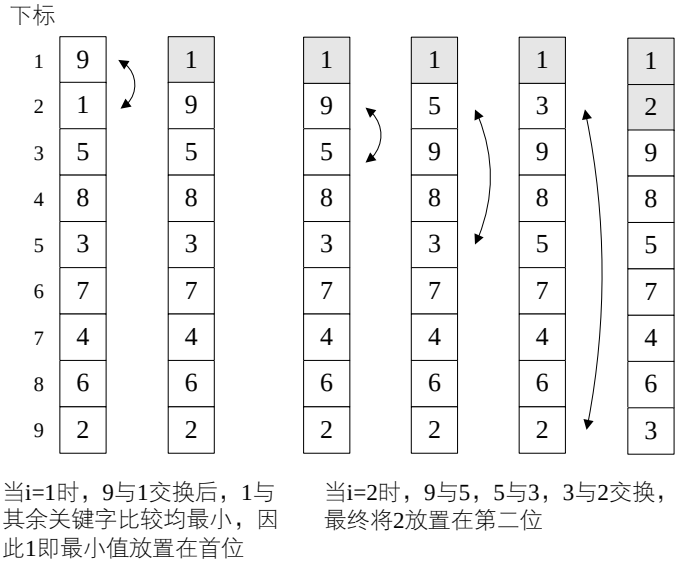


图 9-3-2

它应该算是最最容易写出的排序代码了，不过这个简单易懂的代码，却是有缺陷的。观察后发现，在排序好 1 和 2 的位置后，对其余关键字的排序没有什么帮助（数字 3 反而还被换到了最后一位）。也就是说，这个算法的效率是非常低的。

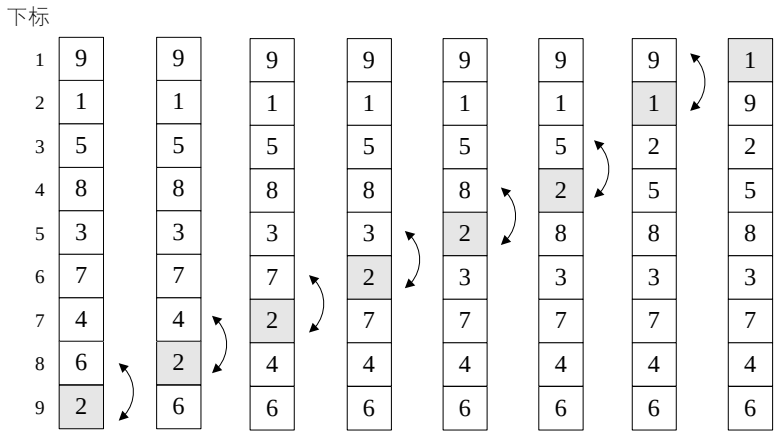
9.3.2 冒泡排序算法

我们来看看正宗的冒泡算法，有没有什么改进的地方。

```
/* 对顺序表 L 作冒泡排序 */
void BubbleSort（SqList *L）
{
    int i,j;
    for（i=1;i<L->length;i++）
    {
```

```
        for（j=L->length-1;j>=i;j--） /* 注意 j 是从后往前循环 */
        {
            if（L->r[j]>L->r[j+1]） /* 若前者大于后者（注意这里与上一算法差异）*/
            {
                swap（L,j,j+1）; /* 交换 L->r[j]与 L->r[j+1]的值 */
            }
        }
    }
}
```

依然假设我们待排序的关键字序列是{9,1,5,8,3,7,4,6,2}，当 i=1 时，变量 j 由 8 反向循环到 1，逐个比较，将较小值交换到前面，直到最后找到最小值放置在了第 1 的位置。如图 9-5-3 所示，当 i=1、j=8 时，我们发现 6>2，因此交换了它们的位置，j=7 时，4>2，所以交换……直到 j=2 时，因为 1<2，所在不交换。j=1 时，9>1，交换，最终得到最小值 1 放置第一的位置。事实上，在不断循环的过程中，除了将关键字 1 放到第一的位置，我们还将关键字 2 从第九位置提到到了第三的位置，显然这一算法比前面的要有进步，在上十万条数据的排序过程中，这种差异会体现出来。图中较小的数字如同气泡般慢慢浮到上面，因此就将此算法命名为冒泡算法。



当i=1时将最小值1冒泡到顶端

图 9-3-3

当 i=2 时，变量 j 由 8 反向循环到 2，逐个比较，在将关键字 2 交换到第二位置的同时，也将关键字 4 和 3 有所提升。

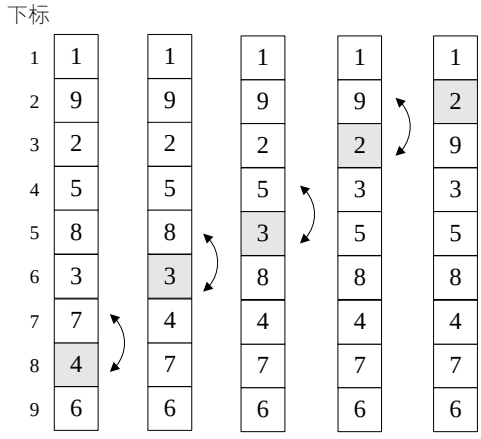


图 9-3-4

后面的数字变换很简单，这里就不在赘述了。

9.3.3 冒泡排序优化

这样的冒泡程序是否还可以优化呢？答案是肯定的。试想一下，如果我们待排序的序列是{2,1,3,4,5,6,7,8,9}，也就是说，除了第一和第二的关键字需要交换外，别的都已经是正常的顺序。当 i=1 时，交换了 2 和 1，此时序列已经有序，但是算法仍然不依不饶地将 i=2 到 9 以及每个循环中的 j 循环都执行了一遍，尽管并没有交换数据，但是之后的大量比较还是大大地多余了，如图 9-3-5 所示。

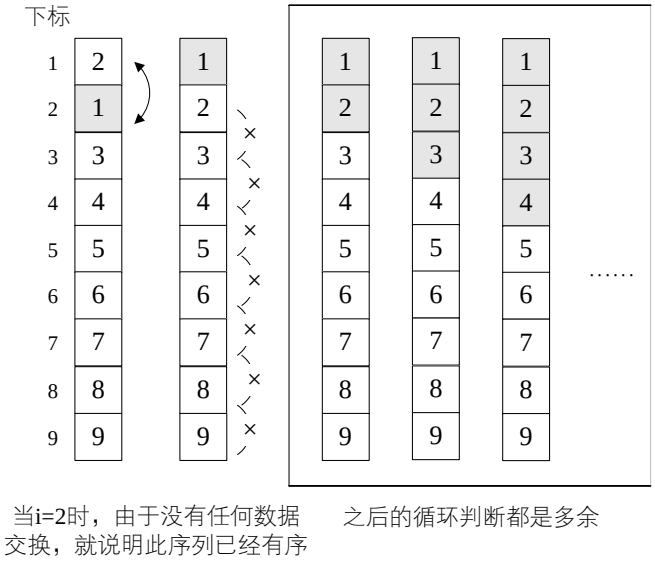


图 9-3-5

当 i=2 时，我们已经对 9 与 8，8 与 7，……，3 与 2 作了比较，没有任何数据交换，这就说明此序列已经有序，不需要再继续后面的循环判断工作了。为了实现这个想法，我们需要改进一下代码，增加一个标记变量 flag 来实现这一算法的改进。

```
/* 对顺序表 L 作改进冒泡算法 */
void BubbleSort2（SqList *L）
{
    int i,j;
    Status flag=TRUE;        /* flag 用来作为标记 */
    for（i=1;i<L->length && flag;i++） /*若 flag 为 true 则退出循环 */
    {
        flag=FALSE;              /* 初始为 false */
        for（j=L->length-1;j>=i;j--）
        {
            if（L->r[j]>L->r[j+1]）
            {
                swap（L,j,j+1）; /* 交换 L->r[j]与 L->r[j+1]的值 */
                flag=TRUE;       /* 如果有数据交换，则 flag 为 true */
            }
        }
    }
}
```

代码改动的关键就是在 i 变量的 for 循环中，增加了对 flag 是否为 true 的判断。经过这样的改进，冒泡排序在性能上就有了一些提升，可以避免因已经有序的情况下的无意义循环判断。

9.3.4 冒泡排序复杂度分析

分析一下它的时间复杂度。当最好的情况，也就是要排序的表本身就是有序的，那么我们比较次数，根据最后改进的代码，可以推断出就是n-1 次的比较，没有数据交换，时间复杂度为$O(n)$。当最坏的情况，即待排序表是逆序的情况，此时需要比较$\sum_{i=2}^{n}(i-1)=1+2+3+\cdots+(n-1)=\frac{n(n-1)}{2}$次，并作等数量级的记录移动。因此，总的时间复杂度为$O(n^2)$。

9.4 简单选择排序

爱炒股票短线的人，总是喜欢不断的买进卖出，想通过价差来实现盈利。但通常这种频繁操作的人，即使失误不多，也会因为操作的手续费和印花税过高而获利很少。还有一种做股票的人，他们很少出手，只是在不断的观察和判断，等到时机一到，果断买进或卖出。他们因为冷静和沉着，以及交易的次数少，而最终收益颇丰。

冒泡排序的思想就是不断地在交换，通过交换完成最终的排序，这和做股票短线频繁操作的人是类似的。我们可不可以像只有在时机非常明确到来时才出手的股票高手一样，也就是在排序时找到合适的关键字再做交换，并且只移动一次就完成相应关键字的排序定位工作呢？这就是选择排序法的初步思想。

选择排序的基本思想是每一趟在 n－i＋1(i=1,2,…,n－1)个记录中选取关键字最小的记录作为有序序列的第 i 个记录。我们这里先介绍的是简单选择排序法。

9.4.1 简单选择排序算法

简单选择排序法（Simple Selection Sort）就是通过 n－i 次关键字间的比较，从 n－i＋1 个记录中选出关键字最小的记录，并和第 i（1≤i≤n）个记录交换之。

我们来看代码。

```
/* 对顺序表 L 作简单选择排序 */
void SelectSort（SqList *L）
{
    int i,j,min;
    for（i=1;i<L->length;i++）
    {
        min = i;                          /* 将当前下标定义为最小值下标 */
        for （j = i+1;j<=L->length;j++）/* 循环之后的数据 */
        {
            if （L->r[min]>L->r[j]）      /* 如果有小于当前最小值的关键字 */
                  min = j;                /* 将此关键字的下标赋值给 min */
        }
        if（i!=min）                    /* 若 min 不等于 i，说明找到最小值，交换 */
            swap（L,i,min）;            /* 交换 L->r[i]与 L->r[min]的值 */
    }
```

```
    }
```

代码应该说不难理解，针对待排序的关键字序列是{9,1,5,8,3,7,4,6,2}，对 i 从 1 循环到 8。当 i=1 时，L.r[i]=9，min 开始是 1，然后与 j=2 到 9 比较 L.r[min]与 L.r[j]的大小，因为 j=2 时最小，所以 min=2。最终交换了 L.r[2]与 L.r[1]的值。如图 9-4-1 所示，注意，这里比较了 8 次，却只交换数据操作一次。

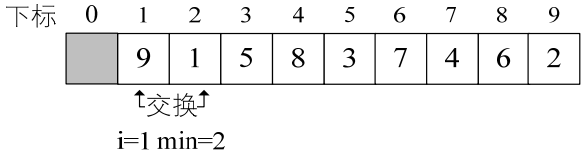


图 9-4-1

当 i=2 时，L.r[i]=9，min 开始是 2，经过比较后，min=9，交换 L.r[min]与 L.r[i]的值。如图 9-4-2 所示，这样就找到了第二位置的关键字。

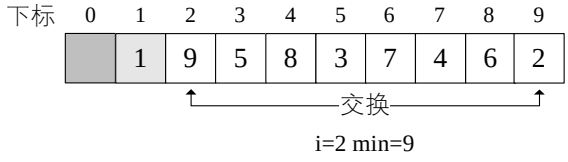


图 9-4-2

当 i=3 时，L.r[i]=5，min 开始是 3，经过比较后，min=5，交换 L.r[min]与 L.r[i]的值。如图 9-4-3 所示。

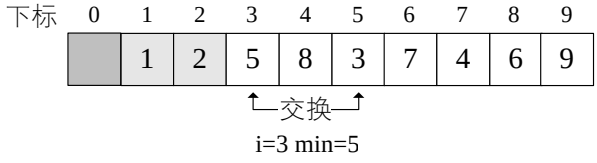


图 9-4-3

之后的数据比较和交换完全雷同，最多经过 8 次交换，就可完成排序工作。

9.4.2 简单选择排序复杂度分析

从简单选择排序的过程来看，它最大的特点就是交换移动数据次数相当少，这样也就节约了相应的时间。分析它的时间复杂度发现，无论最好最差的情况，其比较次数都是一样的多，第i趟排序需要进行$n-i$次关键字的比较,此时需要比较

$\sum_{i=1}^{n-1}(n-i)=n-1+n-2+\cdots+1=\frac{n(n-1)}{2}$ 次。而对于交换次数而言，当最好的时候，交换为 0 次，最差的时候，也就初始降序时，交换次数为$n-1$ 次，基于最终的排序时间是比较

与交换的次数总和，因此，总的时间复杂度依然为$O(n^2)$。

应该说，尽管与冒泡排序同为$O(n^2)$，但简单选择排序的性能上还是要略优于冒泡排序。

9.5 直接插入排序

扑克牌是我们几乎每个人都可能玩过的游戏。最基本的扑克玩法都是一边摸牌，一边理牌。假如我们拿到了这样一手牌，如图 9-5-1 所示。啊，似乎是同花顺呀，别急，我们得理一理顺序才知道是否是真的同花顺。请问，如果是你，应该如何理牌呢？

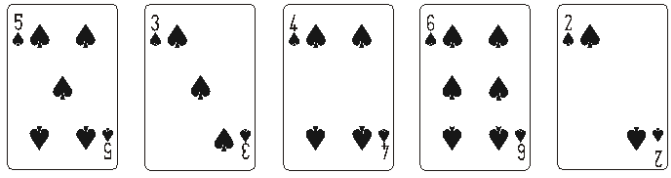


图 9-5-1

应该说，哪怕你是第一次玩扑克牌，只要认识这些数字，理牌的方法都是不用教的。将 3 和 4 移动到 5 的左侧，再将 2 移动到最左侧，顺序就算是理好了。这里，我们的理牌方法，就是直接插入排序法。

9.5.1 直接插入排序算法

直接插入排序（Straight Insertion Sort）的基本操作是将一个记录插入到已经排好序的有序表中，从而得到一个新的、记录数增 1 的有序表。

顾名思义，从名称上也可以知道它是一种插入排序的方法。我们来看直接插入排序法的代码。

```
/* 对顺序表 L 作直接插入排序 */
1   void InsertSort（SqList *L）
2   {
3       int i,j;
4       for（i=2;i<=L->length;i++）
5       {
6           if （L->r[i]<L->r[i-1]）     /* 需将 L->r[i]插入有序子表 */
```

```
7          {
8              L->r[0]=L->r[i];          /* 设置哨兵 */
9              for（j=i-1;L->r[j]>L->r[0];j--）
10                 L->r[j+1]=L->r[j];    /* 记录后移 */
11             L->r[j+1]=L->r[0];        /* 插入到正确位置 */
12         }
13     }
14 }
```

1．程序开始运行，此时我们传入的 SqList 参数的值为 length=6,r[6]={0,5,3,4,6,2}，其中 r[0]=0 将用于后面起到哨兵的作用。

2．第 4～13 行就是排序的主循环。i 从 2 开始的意思是我们假设 r[1]=5 已经放好位置，后面的牌其实就是插入到它的左侧还是右侧的问题。

3．第 6 行，此时 i=2，L.r[i]=3 比 L.r[i-1]=5 要小，因此执行第 8～11 行的操作。第 8 行，我们将 L.r[0]赋值为 L.r[i]=3 的目的是为了起到第 9～10 行的循环终止的判断依据。如图 9-5-2 所示。图中下方的虚线箭头，就是第 10 行，L.r[j-1]=L.r[j]的过程，将 5 右移一位。

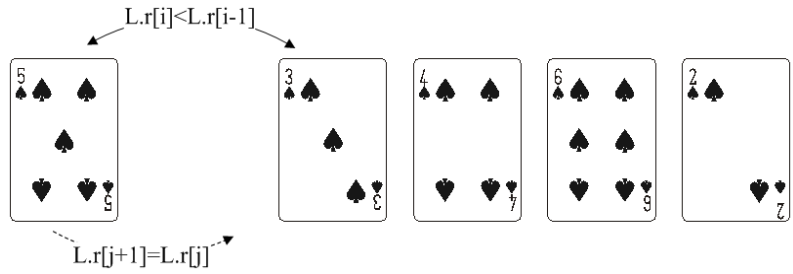


图 9-5-2

4．此时，第 10 行就是在移动完成后，空出了空位，然后第 11 行 L.r[j+1]=L.r[0]，将哨兵的 3 赋值给 j=0 时的 L.r[j+1]，也就是说，将扑克牌 3 放置到 L.r[1]的位置。如图 9-5-3 所示。

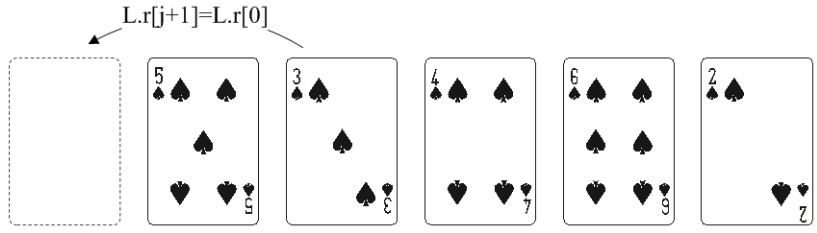


图 9-5-3

5．继续循环，第 6 行，因为此时 i=3，L.r[i]=4 比 L.r[i-1]=5 要小，因此执行第 8～11 行的操作，将 5 再右移一位，将 4 放置到当前 5 所在位置，如图 9-5-4 所示。

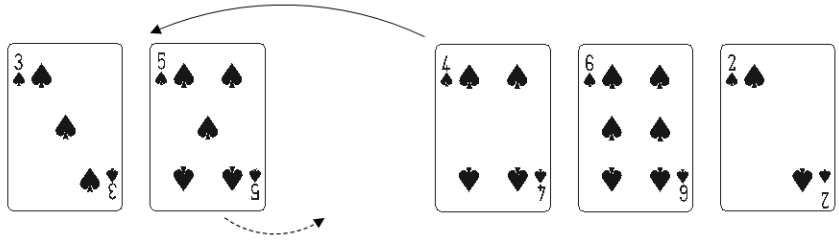


图 9-5-4

6．再次循环，此时 i=4。因为 L.r[i]=6 比 L.r[i-1]=5 要大，于是第 8～11 行代码不执行，此时前三张牌的位置没有变化。如图 9-5-5 所示

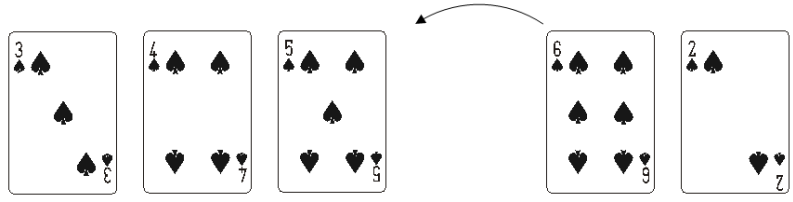


图 9-5-5

7．再次循环，此时 i=5，因为 L.r[i]=2 比 L.r[i-1]=6 要小，因此执行第 8～11 行的操作。由于 6、5、4、3 都比 2 小，它们都将右移一位，将 2 放置到当前 3 所在位置。如图 9-5-6 所示。此时我们的排序也就完成了。

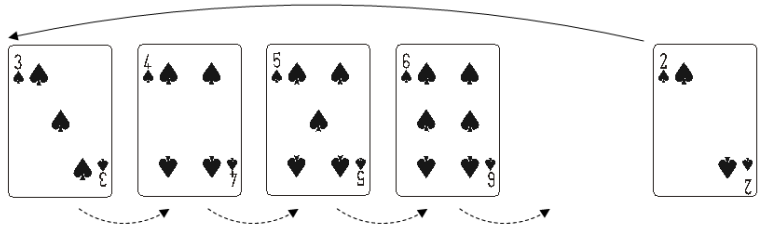


图 9-5-6

9.5.2 直接插入排序复杂度分析

我们来分析一下这个算法，从空间上来看，它只需要一个记录的辅助空间，因此关键是看它的时间复杂度。

当最好的情况，也就是要排序的表本身就是有序的，比如纸牌拿到后就是{2,3,4,5,6}，那么我们比较次数，其实就是代码第 6 行每个 L.r[i]与 L.r[i-1]的比较，共比较了 $n-1\left(\sum_{i=2}^{n}1\right)$次，由于每次都是 L.r[i]>L.r[i-1]，因此没有移动的记录，时间复杂度为 $O(n)$。

当最坏的情况，即待排序表是逆序的情况，比如{6,5,4,3,2}，此时需要比较 $\sum_{i=2}^{n}i=2+3+\cdots+n=\frac{(n+2)(n-1)}{2}$ 次，而记录的移动次数也达到最大值 $\sum_{i=2}^{n}(i+1)=\frac{(n+4)(n-1)}{2}$ 次。

如果排序记录是随机的，那么根据概率相同的原则，平均比较和移动次约为 $\frac{n^2}{4}$ 次。因此，我们得出直接插入排序法的时间复杂度为$O(n^2)$。从这里也看出，同样的$O(n^2)$时间复杂度，直接插入排序法比冒泡和简单选择排序的性能要好一些。

9.6 希尔排序

给大家出一道智力题。请问“VII”是什么?

嗯，很好，它是罗马数字的 7。现在我们要给它加上一笔，让它变成 8（VIII），应该是非常简单，只需要在右侧加一竖线即可。

现在我请大家试着对罗马数字 9，也就是“IX”增加一笔，把它变成 6，应该怎么做?

（几分钟后）

我已经听不少声音说，“这怎么可能!” 可为什么一定要用常规方法呢?

我这里有 3 种另类的方法可以实现它。

方法一：观察发现“X”其实可以看作是一个正放一个倒置两个“V”。因此我们，给“IX”中间加一条水平线，上下颠倒，然后遮住下面部分，也就是说，我们所谓的加上一笔就是遮住一部分，于是就得到“VI”，如图 9-6-1 所示。

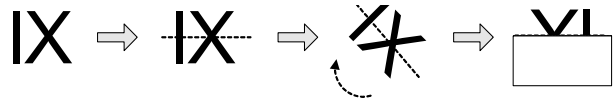


图 9-6-1

方法二：在“IX”前面加一个“S”，此时构成一个英文单词“SIX”，这就等于得到一个 6 了。哈哈，我听到下面一片哗然，我刚有没有说一定要是“VI”呀，我只说把它变成 6 而已，至于是罗马数字还是英文单词，我可没有限制。显然，你们的思维受到了我前面举例的“VII”转变为“VIII”的影响，如图 9-6-2 所示。

图 9-6-2

方法三：在“IX”后面加一个“6”，得到“1X6”，其结果当然是数字 6 了。大家笑了，因为这个想法实在是过分，把字母“I”当成了数字 1，字母“X”看成了乘号。可谁又规定说这是不可以的呢？只要没违反规则，得到 6 即可，如图 9-6-3 所示。

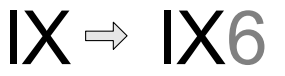


图 9-6-3

智力题的答案介绍完了[1]。大家会发现，看似解决不了的问题，还真不一定就没有办法，也许只是暂时没想到罢了。

我们都能理解，优秀排序算法的首要条件就是**速度**[2]。于是人们想了许许多多的办法，目的就是为了提高排序的速度。而在很长的时间里，众人发现尽管各种排序算法花样繁多（比如前面我们提到的三种不同的排序算法），但时间复杂度都是$O(n^2)$，似乎没法超越了[3]。此时，计算机学术界充斥着“排序算法不可能突破$O(n^2)$”的声音。就像刚才大家做智力题的感觉一样，“不可能”成了主流。

终于有一天，当一位科学家发布超越了$O(n^2)$新排序算法后，紧接着就出现了好几种可以超越$O(n^2)$的排序算法，并把内排序算法的时间复杂度提升到了$O(n\log n)$。“不可能超越$O(n^2)$”彻底成为了历史。

从这里也告诉我们，做任何事，你解决不了时，想一想“Nothing is impossible!”，虽然有点唯心，但这样的思维方式会让你更加深入地思考解决方案，而不是匆忙的放弃。

注：[1]本智力题摘自《在脑袋一侧猛敲一下》。

注：[2]还有其他要求，速度是第一位。

注意：[3]这里排序是指内排序。

9.6.1 希尔排序原理

现在，我要讲解的算法叫**希尔排序（Shell Sort）**。希尔排序是D.L.Shell于 1959 年提出来的一种排序算法，在这之前排序算法的时间复杂度基本都是$O(n^2)$的，希尔排序算法是突破这个时间复杂度的第一批算法之一。

我们前一节讲的直接插入排序，应该说，它的效率在某些时候是很高的，比如，我们的记录本身就是基本有序的，我们只需要少量的插入操作，就可以完成整个记录集的排序工作，此时直接插入很高效。还有就是记录数比较少时，直接插入的优势也比较明显。可问题在于，两个条件本身就过于苛刻，现实中记录少或者基本有序都属于特殊情况。

不过别急，有条件当然是好，条件不存在，我们创造条件也是可以去做的。于是科学家希尔研究出了一种排序方法，对直接插入排序改进后可以增加效率。

如何让待排序的记录个数较少呢？很容易想到的就是将原本有大量记录数的记录进行分组。分割成若干个子序列，此时每个子序列待排序的记录个数就比较少了，然后在这些子序列内分别进行直接插入排序，当整个序列都基本有序时，注意只是基本有序时，再对全体记录进行一次直接插入排序。

此时一定有同学开始疑惑了。这不对呀，比如我们现在有序列是{9,1,5,8,3,7,4,6,2}，现在将它分成三组，{9,1,5}，{8,3,7}，{4,6,2}，哪怕将它们各自排序排好了，变成{1,5,9}，{3,7,8}，{2,4,6}，再合并它们成{1,5,9,3,7,8,2,4,6}，此时，这个序列还是杂乱无序，谈不上基本有序，要排序还是重来一遍直接插入有序，这样做有用吗？需要强调一下，**所谓的基本有序，就是小的关键字基本在前面，大的基本在后面，不大不小的基本在中间，**像{2,1,3,6,4,7,5,8,9}这样可以称为基本有序了。但像{1,5,9,3,7,8,2,4,6}这样的 9 在第三位，2 在倒数第三位就谈不上基本有序。

问题其实也就在这里，我们分割待排序记录的目的是减少待排序记录的个数，并使整个序列向基本有序发展。而如上面这样分完组后就各自排序的方法达不到我们的要求。因此，我们需要采取跳跃分割的策略：**将相距某个“增量”的记录组成一个子序列，这样才能保证在子序列内分别进行直接插入排序后得到的结果是基本有序而不是局部有序。**

9.6.2 希尔排序算法

好了，为了能够真正弄明白希尔排序的算法，我们还是老办法——模拟计算机在

执行算法时的步骤，还研究算法到底是如何进行排序的。

希尔排序算法代码如下。

```
/* 对顺序表 L 作希尔排序 */
1   void ShellSort（SqList *L）
2   {
3       int i,j;
4       int increment=L->length;
5       do
6       {
7           increment=increment/3+1;     /* 增量序列 */
8           for（i=increment+1;i<=L->length;i++）
9           {
10              if （L->r[i]<L->r[i-increment]）
11              {/* 需将 L->r[i]插入有序增量子表 */
12                L->r[0]=L->r[i];    /* 暂存在 L->r[0] */
13                for（j=i-increment;j>0&&L->r[0]<L->r[j];j-=increment）
14                    L->r[j+increment]=L->r[j];/*记录后移，查找插入位置 */
15                L->r[j+increment]=L->r[0]; /* 插入 */
16              }
17          }
18      }
19      while（increment>1）;
20  }
```

1．程序开始运行，此时我们传入的 SqList 参数的值为 length=9,r[10]={0,9,1,5,8,3,7,4,6,2}。这就是我们需要等待排序的序列，如图 9-6-4 所示。

下标	0	1	2	3	4	5	6	7	8	9
		9	1	5	8	3	7	4	6	2

图 9-6-4

2．第 4 行，变量 increment 就是那个“增量”，我们初始值让它等于待排序的记录数。

3．第 5～19 行是一个 do 循环，它提终止条件是 increment 不大于 1 时，其实也就是增量为 1 时就停止循环了。

4．第 7 行，这一句很关键，但也是难以理解的地方，我们后面还要谈到它，先放

一放。这里执行完成后，increment=9/3+1=4。

5．第 8～17 行是一个 for 循环，i 从 4+1=5 开始到 9 结束。

6．第 10 行，判断 L.r[i]与 L.r[i-increment]大小，L.r[5]=3 小于 L.r[i-increment]=L.r[1]=9，满足条件，第 12 行，将 L.r[5]=3 暂存入 L.r[0]。第 13～14 行的循环只是为了将 L.r[1]=9 的值赋给 L.r[5]，由于循环的增量是 j-=increment，其实它就循环了一次，此时 j=-3。第 15 行，再将 L.r[0]=3 赋值给 L.r[j+increment]=L.r[-3+4]=L.r[1]=3。如图 9-6-5 所示，事实上，这一段代码就干了一件事，就是将第 5 位的 3 和第 1 位的 9 交换了位置。

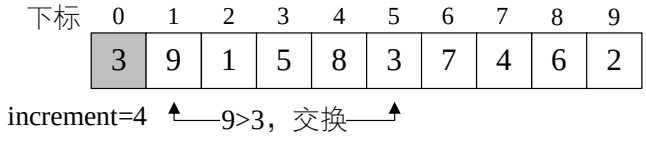


图 9-6-5

7．循环继续，i=6，L.r[6]=7>L.r[i-increment]=L.r[2]=1，因此不交换两者数据。如图 9-6-6 所示。

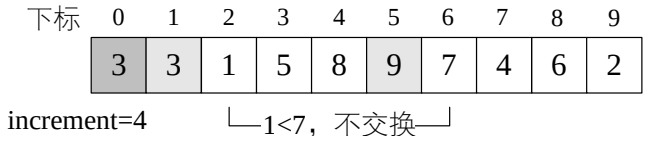


图 9-6-6

8．循环继续，i=7，L.r[7]=4<L.r[i-increment]=L.r[3]=5，交换两者数据。如图 9-6-7 所示。

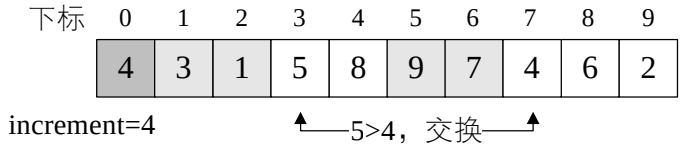


图 9-6-7

9．循环继续，i=8，L.r[8]=6<L.r[i-increment]=L.r[4]=8，交换两者数据。如图 9-6-8 所示。

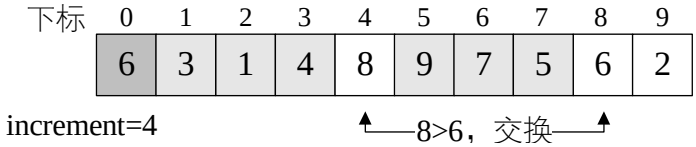


图 9-6-8

10．循环继续，i=9，L.r[9]=2<L.r[i-increment]=L.r[5]=9，交换两者数据。注意，第 13～14 行是循环，此时还要继续比较 L.r[5]与 L.r[1]的大小，因为 2<3，所以还要交换 L.r[5]与 L.r[1]的数据，如图 9-6-9 所示。

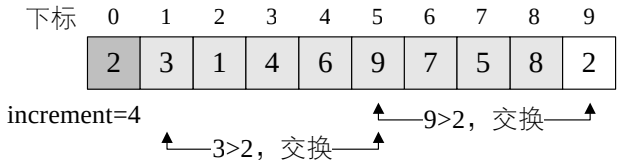


图 9-6-9

最终第一轮循环后，数组的排序结果为图 9-6-10 所示。细心的同学会发现，我们的数字 1、2 等小数字已经在前两位，而 8、9 等大数字已经在后两位，也就是说，通过这样的排序，我们已经让整个序列基本有序了。这其实就是希尔排序的精华所在，它将关键字较小的记录，不是一步一步地往前挪动，而是跳跃式地往前移，从而使得每次完成一轮循环后，整个序列就朝着有序坚实地迈进一步。

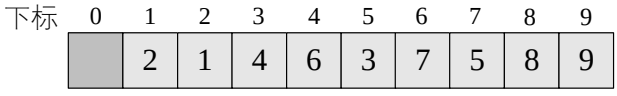


图 9-6-10

11．我们继续，在完成一轮 do 循环后，此时由于 increment=4>1 因此我们需要继续 do 循环。第 7 行得到 increment=4/3+1=2。第 8～17 行 for 循环，i 从 2+1=3 开始到 9 结束。当 i=3、4 时，不用交换，当 i=5 时，需要交换数据，如图 9-6-11 所示。

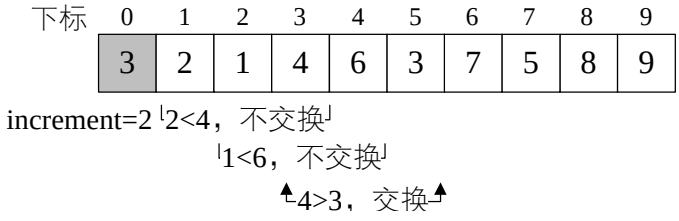


图 9-6-11

12．此后，i=6、7、8、9 均不用交换，如图 9-6-12 所示。

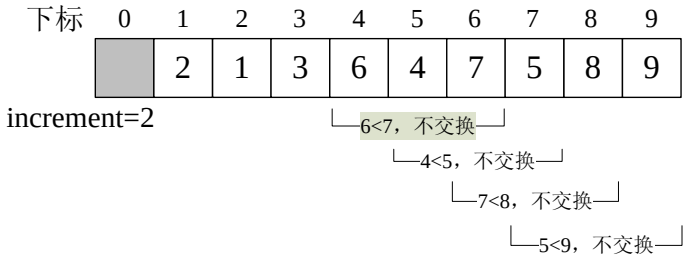


图 9-6-12

13．再次完成一轮 do 循环，increment=2>1，再次 do 循环，第 7 行得到 increment=2/3+1=1，此时这就是最后一轮 do 循环了。尽管第 8～17 行 for 循环，i 从 1+1=2 开始到 9 结束，但由于当前序列已经基本有序，可交换数据的情况大为减少，效率其实很高。如图 9-6-13 所示，图中箭头连线为需要交换的关键字。

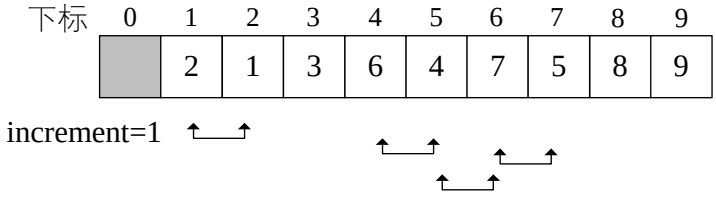


图 9-6-13

最终完成排序过程，如图 9-6-14 所示。

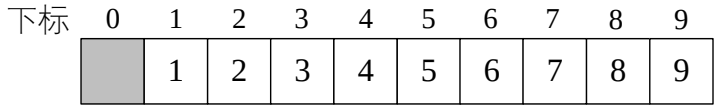


图 9-6-14

9.6.3 希尔排序复杂度分析

通过这段代码的剖析，相信大家有些明白，希尔排序的关键并不是随便分组后各自排序，而是将相隔某个“增量”的记录组成一个子序列，实现跳跃式的移动，使得排序的效率提高。

这里“增量”的选取就非常关键了。我们在代码中第 7 行，是用increment=increment/3+1;的方式选取增量的，可究竟应该选取什么样的增量才是最好，目前还是一个数学难题，迄今为止还没有人找到一种最好的增量序列。不过大量的研究表明，当增量序列为$dlta[k]=2^{t-k+1}-1$（$0\leqslant k\leqslant t\leqslant \lfloor \log_2(n+1) \rfloor$）时，可以获得不错的效

率，其时间复杂度为$O(n^{3/2})$，要好于直接排序的$O(n^2)$。需要注意的是，**增量序列的最后一个增量值必须等于 1 才行。**另外由于记录是跳跃式的移动，希尔排序并不是一种稳定的排序算法。

不管怎么说，希尔排序算法的发明，使得我们终于突破了慢速排序的时代（超越了时间复杂度为$O(n^2)$），之后，相应的更为高效的排序算法也就相继出现了。

9.7 堆 排 序

我们前面讲到简单选择排序，它在待排序的 n 个记录中选择一个最小的记录需要比较 n－1 次。本来这也可以理解，查找第一个数据需要比较这么多次是正常的，否则如何知道它是最小的记录。

可惜的是，这样的操作并没有把每一趟的比较结果保存下来，在后一趟的比较中，有许多比较在前一趟已经做过了，但由于前一趟排序时未保存这些比较结果，所以后一趟排序时又重复执行了这些比较操作，因而记录的比较次数较多。

如果可以做到每次在选择到最小记录的同时，并根据比较结果对其他记录做出相应的调整，那样排序的总体效率就会非常高了。而堆排序（Heap Sort），就是对简单选择排序进行的一种改进，这种改进的效果是非常明显的。堆排序算法是 Floyd 和 Williams 在 1964 年共同发明的，同时，他们发明了“堆”这样的数据结构。

回忆一下我们小时候，特别是男同学，基本都玩过叠罗汉的恶作剧。通常都是先把某个要整的人按倒在地，然后大家就一拥而上扑了上去……后果？后果当然就是一笑了之，一个恶作剧而已。不过在西班牙的加泰罗尼亚地区，他们将叠罗汉视为了正儿八经的民族体育活动，如图 9-7-1 所示，可以想象当时场面的壮观。

图 9-7-1

叠罗汉运动是把人堆在一起，而我们这里要介绍的“堆”结构相当于把数字符号堆成一个塔型的结构。当然，这绝不是简单的堆砌。大家看图 9-7-2 所示，能够找到什么规律吗？

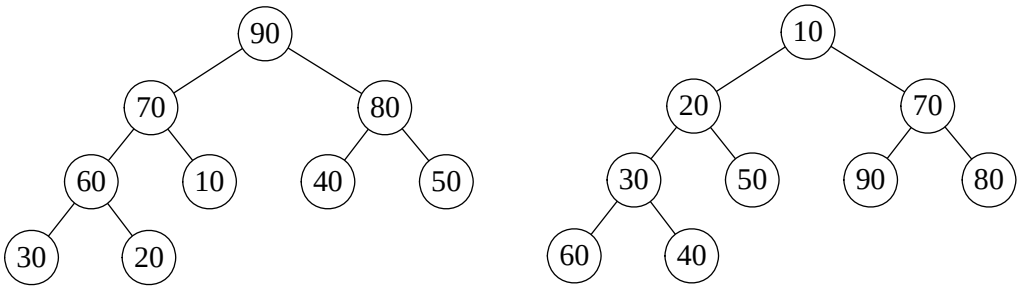


图 9-7-2

很明显，我们可以发现它们都是二叉树，如果观察仔细些，还能看出它们都是完全二叉树。左图中根结点是所有元素中最大的，右图的根结点是所有元素中最小的。再细看看，发现左图每个结点都比它的左右孩子要大，右图每个结点都比它的左右孩子要小。这就是我们要讲的堆结构。

堆是具有下列性质的完全二叉树：**每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆**（例如图 9-7-2 左图所示）；**或者每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆**（例如图 9-7-2 右图所示）。

这里需要注意从堆的定义可知，根结点一定是堆中所有结点最大（小）者。较大（小）的结点靠近根结点（但也不绝对，比如右图小顶堆中 60、40 均小于 70，但它们并没有 70 靠近根结点）。

如果按照层序遍历的方式给结点从 1 开始编号，则结点之间满足如下关系：

$$\begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \text{或} \begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad 1 \leqslant i \leqslant \left\lfloor \frac{n}{2} \right\rfloor$$

这里为什么i要小于等于⌊n/2⌋呢？相信大家可能都忘记了二叉树的性质五[4]，其实忘记也不奇怪，这个性质在我们讲完之后，就再也没有提到过它。可以说，这个性质仿佛就是在为堆准备的。性质 5 的第一条就说一棵完全二叉树，如果i=1，则结点i是二叉树的根，无双亲；如果i>1，则其双亲是结点⌊i/2⌋。那么对于有n个结点的二叉树而言，它的i值自然就是小于等于⌊n/2⌋了。性质 5 的第二、三条，也是在说明下标i与2i和 2i+1 的双亲子女关系。如果完全忘记的同学不妨去复习一下。

如果将图 9-7-2 的大顶堆和小顶堆用层序遍历存入数组，则一定满足上面的关系表达，如图 9-7-3 所示。

注：[4]详见本书 6.6 节。

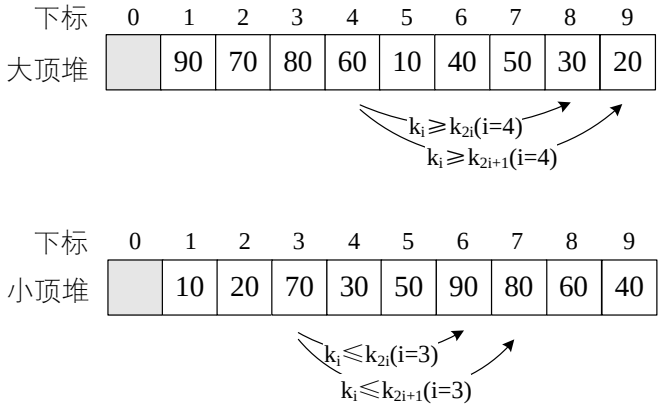


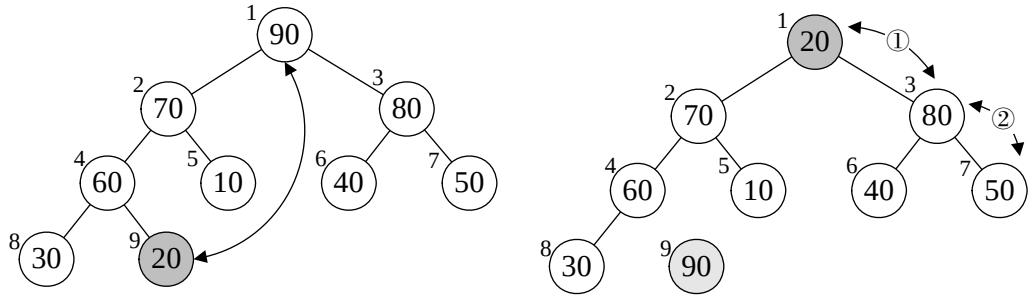
图 9-7-3

我们现在讲这个堆结构，其目的就是为了堆排序用的。

9.7.1 堆排序算法

堆排序（Heap Sort）就是利用堆（假设利用大顶堆）进行排序的方法。它的基本思想是，**将待排序的序列构造成一个大顶堆。此时，整个序列的最大值就是堆顶的根结点。将它移走（其实就是将其与堆数组的末尾元素交换，此时末尾元素就是最大值），然后将剩余的n–1 个序列重新构造成一个堆，这样就会得到n个元素中的次小值。如此反复执行，便能得到一个有序序列了。**

例如图 9-7-4 所示，左图是一个大顶堆，90 为最大值，将 90 与 20（末尾元素）互换，如中图所示，此时 90 就成了整个堆序列的最后一个元素，将 20 经过调整，使得除 90 以外的结点继续满足大顶堆定义（所有结点都大于等于其子孩子），见右图，然后再考虑将 30 与 80 互换……



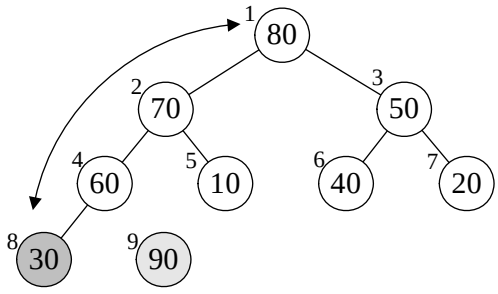


图 9-7-4

相信大家有些明白堆排序的基本思想了，不过要实现它还需要解决两个问题：

1．如何由一个无序序列构建成一个堆?

2．如果在输出堆顶元素后，调整剩余元素成为一个新的堆?

要解释清楚它们，让我们来看代码。

```
/* 对顺序表 L 进行堆排序 */
1   void HeapSort（SqList *L）
2   {
3       int i;
4       for（i=L->length/2;i>0;i--）  /* 把 L 中的 r 构建成一个大顶堆 */
5           HeapAdjust（L,i,L->length）;

6       for（i=L->length;i>1;i--）
7       {
8           swap（L,1,i）; /*将堆顶记录和当前未经排序子序列的最后一个记录交换*/
9           HeapAdjust（L,1,i-1）;  /* 将 L->r[1..i-1]重新调整为大顶堆 */
10      }
11  }
```

从代码中也可以看出，整个排序过程分为两个 for 循环。第一个循环要完成的就是将现在的待排序序列构建成一个大顶堆。第二个循环要完成的就是逐步将每个最大值的根结点与末尾元素交换，并且再调整其成为大顶堆。

假设我们要排序的序列是{50,10,90,30,70,40,80,60,20}[5]，那么L.length=9，第一个for循环，代码第 4 行，i是从⌊9/2⌋=4 开始，4→3→2→1 的变量变化。为什么不是从 1 到 9 或者从 9 到 1，而是从 4 到 1 呢？其实我们看了图 9-7-5 就明白了，它们都有

注：[5] 这里把每个数字乘以 10，是为了与下标的个位数字进行区分，因为我们在讲解中，会大量的提到数组下标的数字。

什么规律？它们都是有孩子的结点。注意灰色结点的下标编号就是 1、2、3、4。

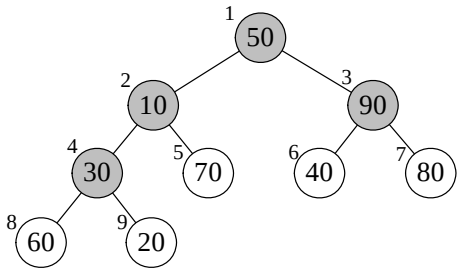


图 9-7-5

我们所谓的将待排序的序列构建成为一个大顶堆，其实就是从下往上、从右到左，将每个非终端结点（非叶结点）当作根结点，将其和其子树调整成大顶堆。i 的 4→3→2→1 的变量变化，其实也就是 30，90，10、50 的结点调整过程。

既然已经弄清楚 i 的变化是在调整哪些元素了，现在我们来看关键的 HeapAdjust（堆调整）函数是如何实现的。

```
/* 已知 L->r[s..m]中记录的关键字除 L->r[s]之外均满足堆的定义 */
/* 本函数调整 L->r[s]的关键字,使 L->r[s..m]成为一个大顶堆 */
1  void HeapAdjust（SqList *L,int s,int m）
2  {
3      int temp,j;
4      temp=L->r[s];
5      for（j=2*s;j<=m;j*=2）    /* 沿关键字较大的孩子结点向下筛选 */
6      {
7          if（j<m && L->r[j]<L->r[j+1]）
8              ++j;              /* j 为关键字中较大的记录的下标 */
9          if（temp>=L->r[j]）
10             break;            /* rc 应插入在位置 s 上 */
11         L->r[s]=L->r[j];
12         s=j;
13     }
14     L->r[s]=temp;             /* 插入 */
15 }
```

1．函数被第一次调用时，s=4，m=9，传入的 SqList 参数的值为 length=9,r[10]={0,50,10,90, 30,70,40,80,60,20}。

2．第 4 行，将 L.r[s]=L.r[4]=30 赋值给 temp，如图 9-7-6 所示。

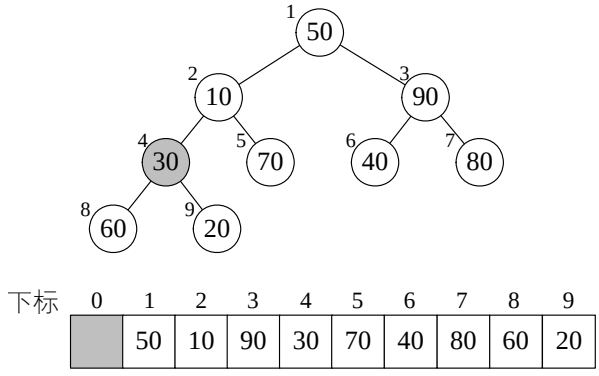


图 9-7-6

3．第 5～13 行，循环遍历其结点的孩子。这里 j 变量为什么是从 2*s 开始呢？又为什么是 j*=2 递增呢？原因还是二叉树的性质 5，因为我们这棵是完全二叉树，当前结点序号是 s，其左孩子的序号一定是 2s，右孩子的序号一定是 2s+1，它们的孩子当然也是以 2 的位数序号增加，因此 j 变量才是这样循环。

4．第 7～8 行，此时 j=2*4=8，j<m 说明它不是最后一个结点，如果 L.r[j]<L.r[j+1]，则说明左孩子小于右孩子。我们的目的是要找到较大值，当然需要让 j+1 以便变成指向右孩子的下标。当前 30 的左右孩子是 60 和 20，并不满足此条件，因此 j 还是 8。

5．第 9～10 行，temp=30，L.r[j]=60，并不满足条件。

6．第 11～12 行，将 60 赋值给 L.r[4]，并令 s=j=8。也就是说，当前算出，以 30 为根结点的子二叉树，当前最大值是 60，在第 8 的位置。注意此时 L.r[4]和 L.r[8]的值均为 60。

7．再循环因为 j=2*j=16，m=9，j>m，因此跳出循环。

8．第 14 行，将 temp=30 赋值给 L.r[s]=L.r[8]，完成 30 与 60 的交换工作。如图 9-7-7 所示。本次函数调用完成。

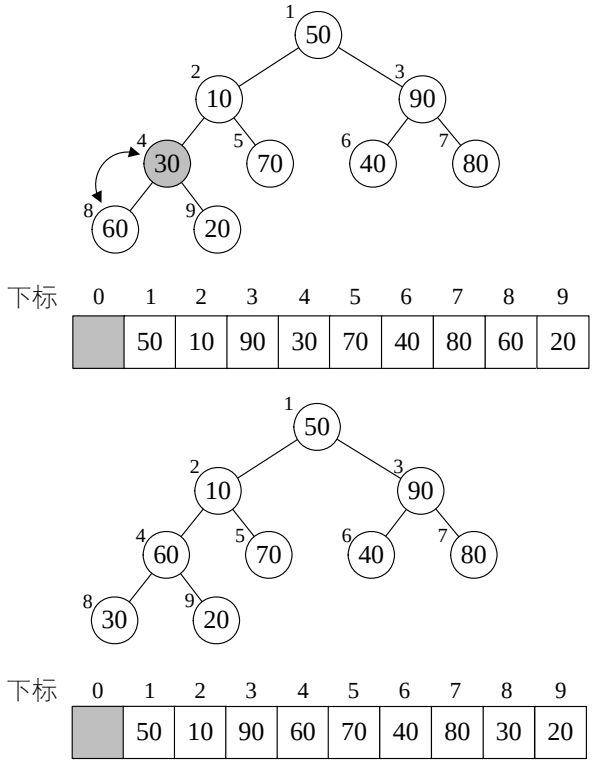
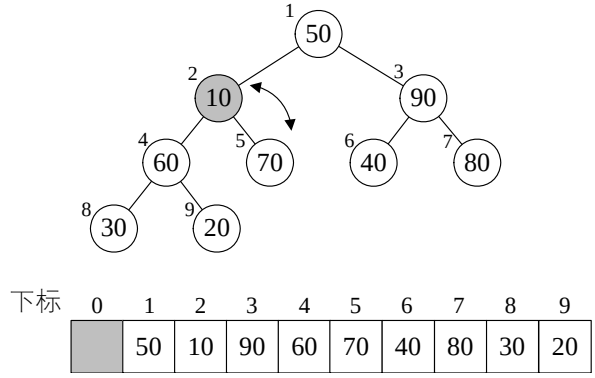


图 9-7-7

9．再次调用 HeapAdjust，此时 s=3，m=9。第 4 行，temp=L.r[3]=90，第 7～8 行，由于 40<80 得到 j+1=2*s+1=7。9～10 行，由于 90>80，因此退出循环，最终本次调用，整个序列未发什么改变。

10．再次调用 HeapAdjust，此时 s=2，m=9。第 4 行，temp=L.r[2]=10，第 7～8 行，60<70，使得 j=5。最终本次调用使得 10 与 70 进行了互换。



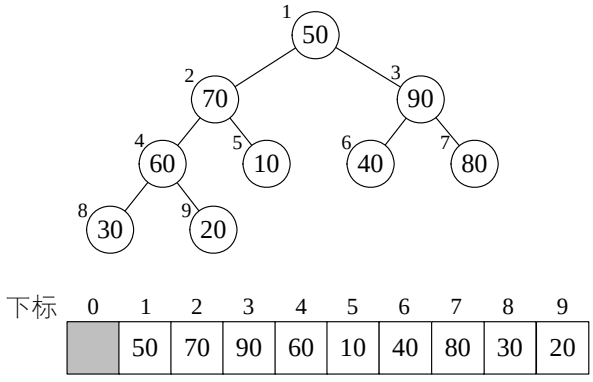


图 9-7-8

11．再次调用 HeapAdjust，此时 s=1，m=9。第 4 行，temp=L.r[1]=50，第 7～8 行，70<90，使得 j=3。第 11～12 行，L.r[1]被赋值了 90，并且 s=3，再循环，由于 2j=6 并未大于 m，因此再次执行循环体，使得 L.r[3]被赋值了 80，完成循环后，L.[7]被赋值为 50，最终本次调用使得 50、90、80 进行了轮换。

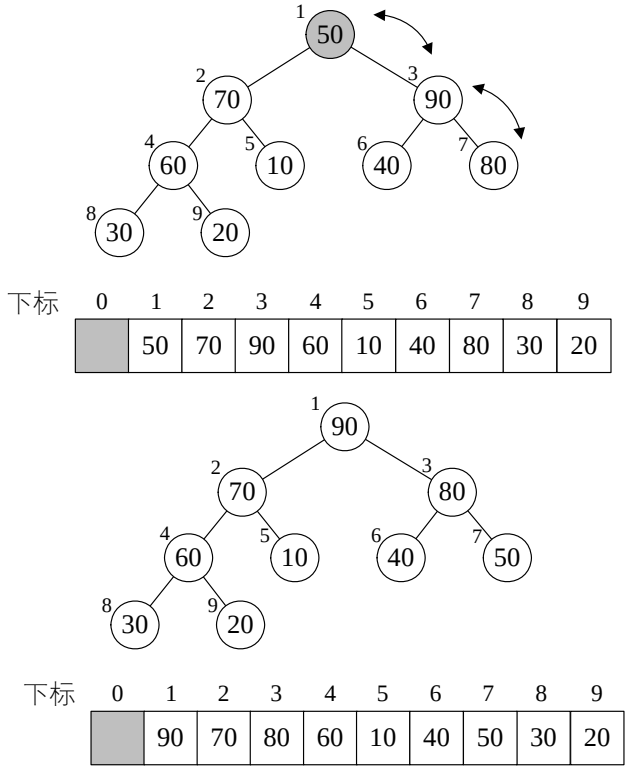


图 9-7-9

到此为止，我们构建大顶堆的过程算是完成了，也就是 HeapSort 函数的第 4～5 行循环执行完毕。或许是有点复杂，如果不明白，多试着模拟计算机执行的方式走几遍，应该就可以理解其原理。

接下来 HeapSort 函数的第 6～11 行就是正式的排序过程，由于有了前面的充分准备，其实这个排序就比较轻松了。下面是这部分代码。

```
6   for（i=L->length;i>1;i--）
7   {
8       swap（L,1,i）; /* 将堆顶记录和当前未经排序子序列的最后一个记录交换 */
9       HeapAdjust（L,1,i-1）; /* 将 L->r[1..i-1]重新调整为大顶堆 */
10  }
```

1．当 i=9 时，第 8 行，交换 20 与 90，第 9 行，将当前的根结点 20 进行大顶堆的调整，调整过程和刚才流程一样，找到它左右子结点的较大值，互换，再找到其子结点的较大值互换。此时序列变为{80,70,50,60,10,40,20,30,90}，如图 9-7-10 所示。

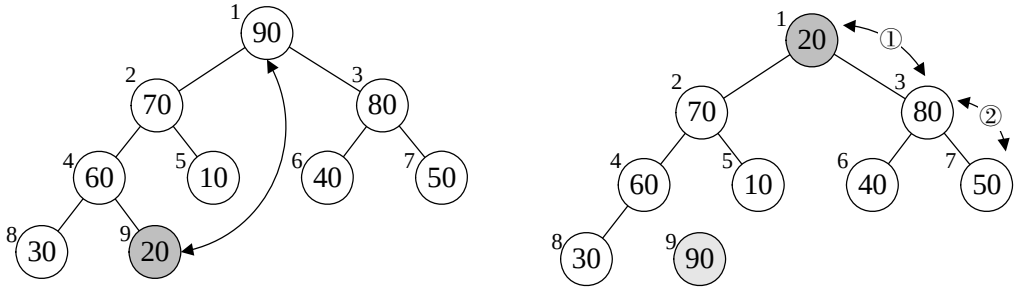


图 9-7-10

2．当 i=8 时，交换 30 与 80，并将 30 与 70 交换，再与 60 交换，此时序列变为{70,60,50,30,10,40,20,80,90}，如图 9-7-11 所示。

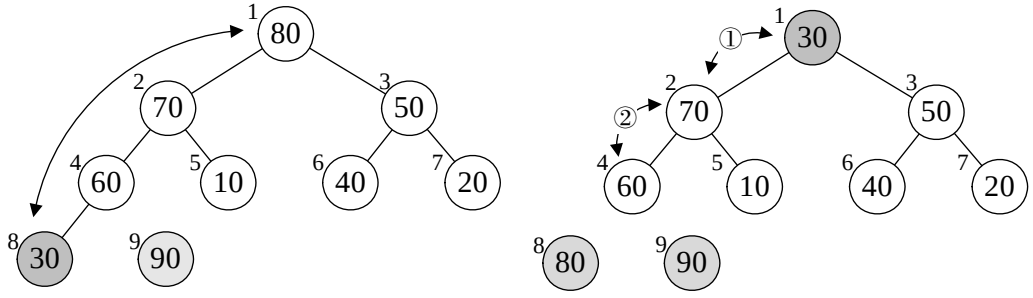


图 9-7-11

3．后面的变化完全类似，不解释，只看图。

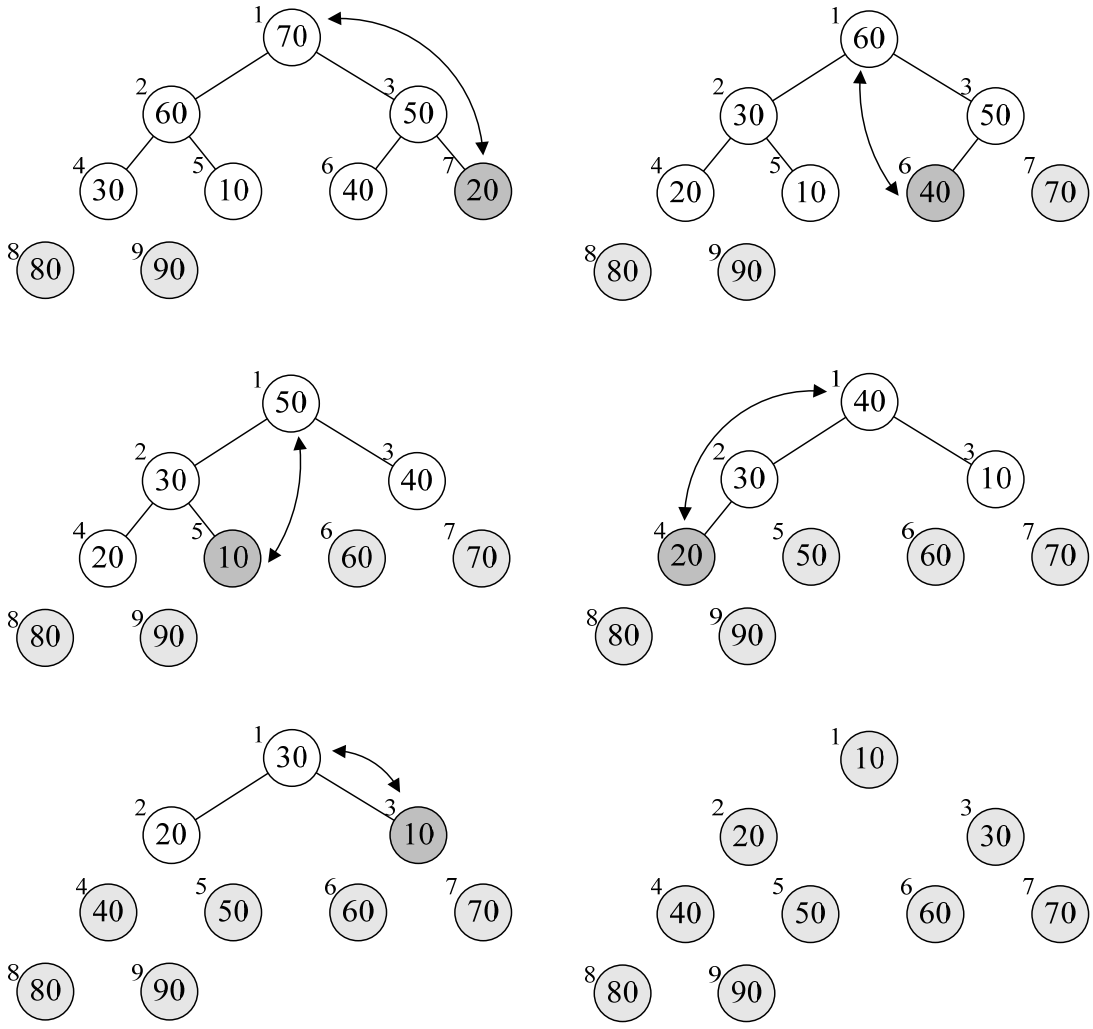


图 9-7-12

最终就得到一个完全有序的序列了。

9.7.2 堆排序复杂度分析

堆排序的效率到底有多高呢？我们来分析一下。

它的运行时间主要是消耗在初始构建堆和在重建堆时的反复筛选上。

在构建堆的过程中，因为我们是完全二叉树从最下层最右边的非终端结点开始构建，将它与其孩子进行比较和若有必要的互换，对于每个非终端结点来说，其实最多进行两次比较和互换操作，因此整个构建堆的时间复杂度为 $O(n)$。

在正式排序时，第i次取堆顶记录重建堆需要用$O(\log i)$的时间（完全二叉树的某个结点到根结点的距离为$\lfloor \log_2 i \rfloor + 1$），并且需要取$n-1$ 次堆顶记录，因此，重建堆的时间复杂度为$O(n\log n)$。

所以总体来说，堆排序的时间复杂度为$O(n\log n)$。由于堆排序对原始记录的排序状态并不敏感，因此它无论是最好、最坏和平均时间复杂度均为$O(n\log n)$。这在性能上显然要远远好过于冒泡、简单选择、直接插入的$O(n^2)$的时间复杂度了。

空间复杂度上，它只有一个用来交换的暂存单元，也非常的不错。不过由于记录的比较与交换是跳跃式进行，因此堆排序也是一种不稳定的排序方法。

另外，由于初始构建堆所需的比较次数较多，因此，它并不适合待排序序列个数较少的情况。[6]

9.8 归并排序

前面我们讲了堆排序，因为它用到了完全二叉树，充分利用了完全二叉树的深度是$\lfloor \log_2 n \rfloor + 1$ 的特性，所以效率比较高。不过堆结构的设计本身是比较复杂的，老实说，能想出这样的结构就挺不容易，有没有更直接简单的办法利用完全二叉树来排序呢？当然有。

先来举一个例子。你们知道高考一本、二本、专科分数线是如何划分出来的吗?

简单地说，如果各高校本科专业在某省高三理科学生中计划招收 1 万名，那么将全省参加高考的理科学生分数倒排序，第 1 万名的总分数就是当年本科生的分数线（现实可能会比这复杂，这里简化之）。也就是说，即使你是你们班级第一、甚至年级第一名，如果你没有上分数线，则说明你的成绩排不到全省前 1 万名，你也就基本失去了当年上本科的机会了。

换句话说，所谓的全省排名，其实也就是每个市、每个县、每个学校、每个班级的排名合并后再排名得到的。注意我这里用到了合并一词。

注：[6] 关于堆排序算法更详细讲解，请参考《算法导论》第二部分第六章“堆排序”的内容。

我们要比较两个学生的成绩高低是很容易的，比如甲比乙分数低，丙比丁分数低。那么我们也就可以很容易得到甲乙丙丁合并后的成绩排名，同样的，戊己庚辛的排名也容易得到，由于他们两组分别有序了，把他们八个学生成绩合并有序也是很容易做到的了，继续下去……最终完成全省学生的成绩排名，此时高考状元也就诞生了。

为了更清晰地说清楚这里的思想，大家来看图 9-8-1 所示，我们将本是无序的数组序列{16,7,13,10,9,15,3,2,5,8,12,1,11,4,6,14}，通过两两合并排序后再合并，最终获得了一个有序的数组。注意仔细观察它的形状，你会发现，它像极了一棵倒置的完全二叉树，通常涉及到完全二叉树结构的排序算法，效率一般都不低的——这就是我们要讲的归并排序法。

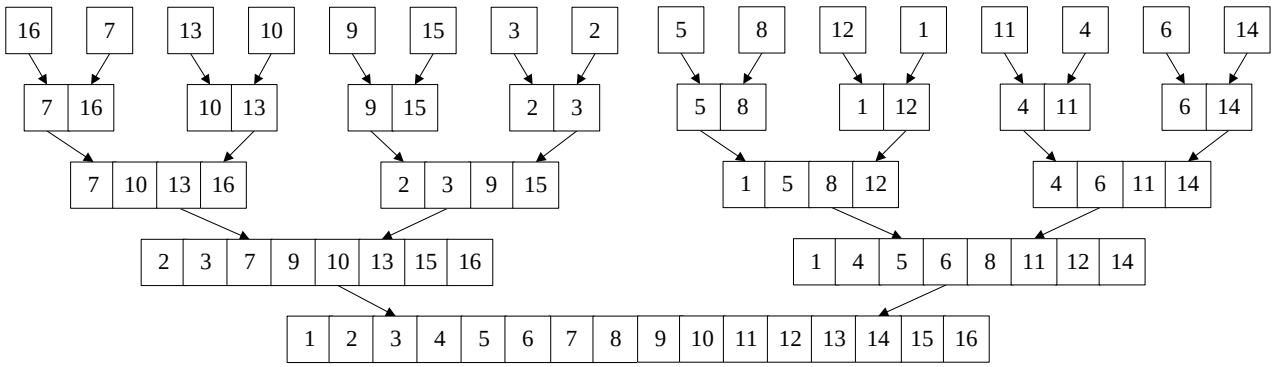


图 9-8-1

9.8.1 归并排序算法

归并”一词的中文含义就是合并、并入的意思，而在数据结构中的定义是将两个或两个以上的有序表组合成一个新的有序表。

归并排序（Merging Sort）就是利用归并的思想实现的排序方法。它的原理是**假设初始序列含有n个记录，则可以看成是n个有序的子序列，每个子序列的长度为 1，然后两两归并，得到⌈n/2⌉（⌈x⌉表示不小于x的最小整数）个长度为 2 或 1 的有序子序列；再两两归并，……，如此重复，直至得到一个长度为n的有序序列为止，这种排序方法称为 2 路归并排序。**[7]

好了，有了对归并排序的初步认识后，我们来看代码。

注：[7]本书只介绍 2 路归并排序。

```
/* 对顺序表 L 作归并排序 */
void MergeSort（SqList *L）
{
    MSort（L->r,L->r,1,L->length）;
}
```

一句代码，别奇怪，它只是调用了另一个函数而已。为了与前面的排序算法统一，我们用了同样的参数定义SqList *L，由于我们要讲解的归并排序实现需要用到递归调用[8]，因此我们外封装了一个函数。假设现在要对数组{50,10,90,30,70,40,80,60,20}进行排序，L.length=9，我现来看看MSort的实现。

```
/* 将 SR[s..t]归并排序为 TR1[s..t] */
1   void MSort（int SR[],int TR1[],int s, int t）
2   {
3       int m;
4       int TR2[MAXSIZE+1];
5       if（s==t）
6          TR1[s]=SR[s];
7       else
8       {
9          m=（s+t）/2; /* 将 SR[s..t]平分为 SR[s..m]和 SR[m+1..t] */
10         MSort(SR,TR2,s,m);/*递归将 SR[s..m]归并为有序的 TR2[s..m]*/
11         MSort(SR,TR2,m+1,t);/*递归将 SR[m+1..t]归并为有序 TR2[m+1..t]*/
12         Merge(TR2,TR1,s,m,t);/*将 TR2[s..m]和 TR2[m+1..t]*/
                                /*归并到 TR1[s..t]*/
13      }
14  }
```

1．MSort 被调用时，SR 与 TR1 都是{50,10,90,30,70,40,80,60,20}，s=1，t=9，最终我们的目的就是要将 TR1 中的数组排好顺序。

2．第 5 行，显然 s 不等于 t，执行第 8～13 行语句块。

3．第 9 行，m=（1+9）/2=5。m 就是序列的正中间下标。

4．此时第 10 行，调用“MSort（SR,TR2,1,5）;”的目标就是将数组 SR 中的第 1～5 的关键字归并到有序的 TR2（调用前 TR2 为空数组），第 11 行，调用“MSort（SR,TR2,6,9）;”的目标就是将数组 SR 中的第 6～9 的关键字归并到

注：[8]也可以不用递归实现，后面有提及。

有序的 TR2。也就是说，在调用这两句代码之前，代码已经准备将数组分成了两组了，如图 9-8-2 所示。

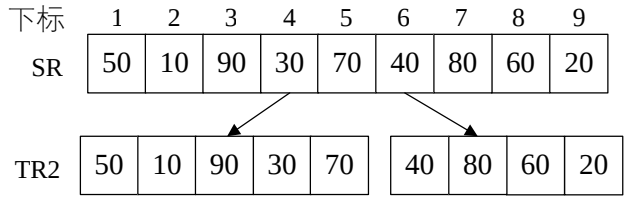


图 9-8-2

5．第 12 行，函数 Merge 的代码细节一会再讲，调用“Merge（TR2,TR1,1,5,9）;”的目标其实就是将第 10 和 11 行代码获得的数组 TR2（注意它是下标为 1～5 和 6～9 的关键字分别有序）归并为 TR1，此时相当于整个排序就已经完成了，如图 9-8-3 所示。

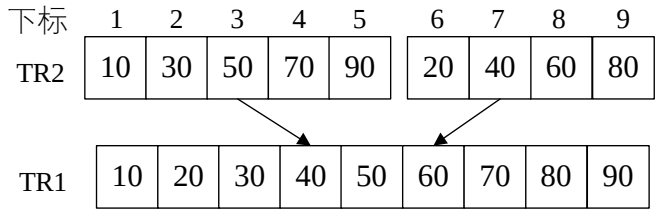


图 9-8-3

6．再来看第 10 行递归调用进去后，s=1，t=5，m=（1+5）/2=3。此时相当于将 5 个记录拆分为三个和两个。继续递归进去，直到细分为一个记录填入 TR2，此时 s 与 t 相等，递归返回，如图 9-8-4 的左图所示。每次递归返回后都会执行当前递归函数的第 12 行，将 TR2 归并到 TR1 中，如图 9-8-4 的右图所示，最终使得当前序列有序。

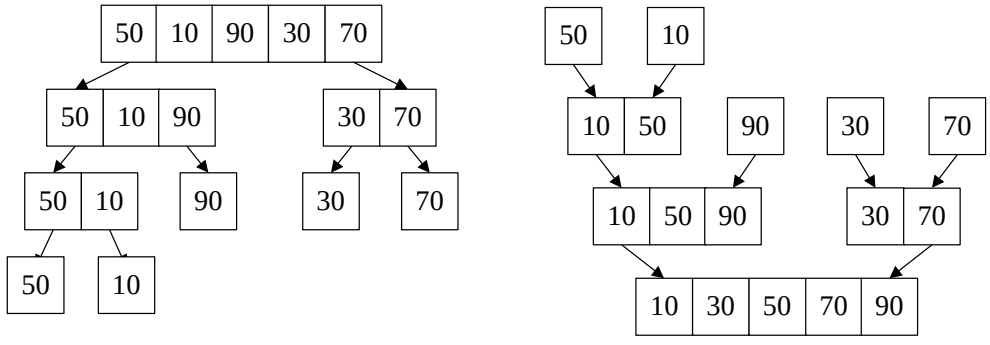


图 9-8-4

7．同样的第 11 行也是类似方式，如图 9-8-5 所示。

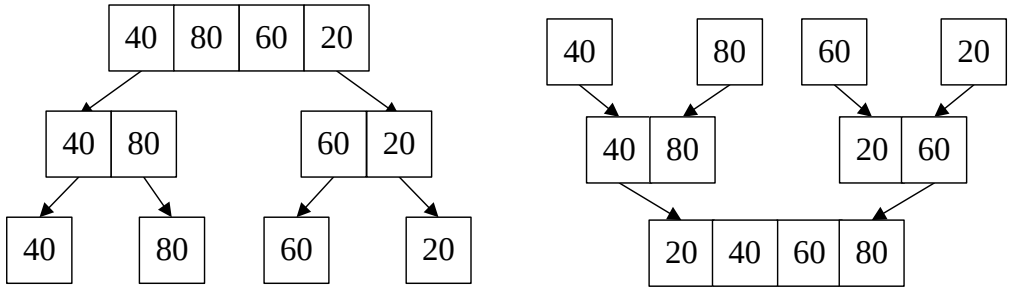


图 9-8-5

8．此时也就是刚才所讲的最后一次执行第 12 行代码，将{10,30,50,70,90}与{20,40,60,80}归并为最终有序的序列。

可以说，如果对递归函数的运行方式理解比较透的话，MSort 函数还是很好理解的。我们来看看整个数据变换示意图，如图 9-8-6 所示。

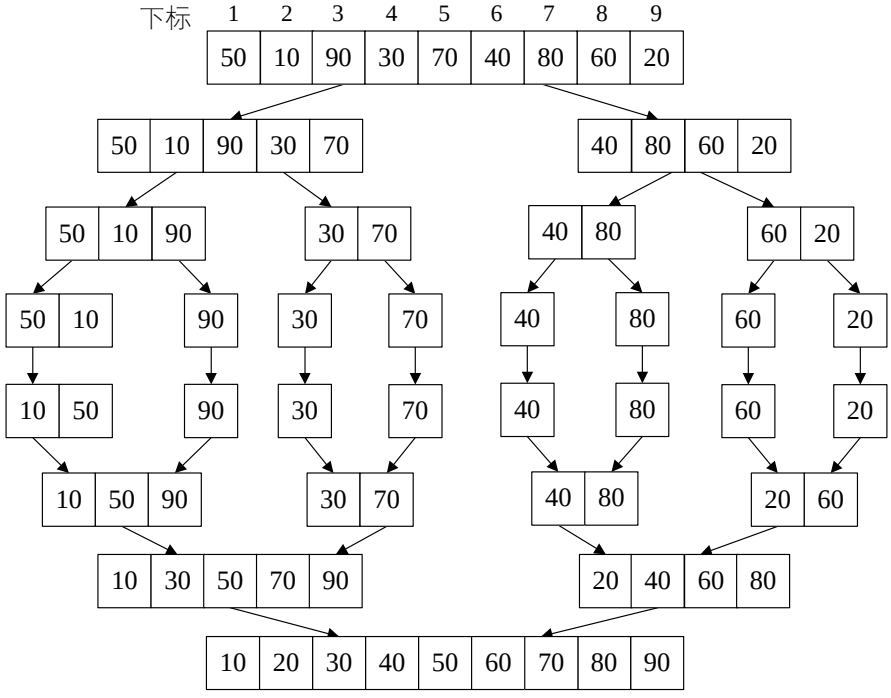


图 9-8-6

现在我们来看看 Merge 函数的代码是如何实现的。

```
/* 将有序的 SR[i..m]和 SR[m+1..n]归并为有序的 TR[i..n] */
1   void Merge（int SR[],int TR[],int i,int m,int n）
2   {
3       int j,k,l;
```

```
4       for（j=m+1,k=i;i<=m && j<=n;k++）/* 将 SR 中记录由小到大归并入 TR */
5       {
6           if （SR[i]<SR[j]）
7               TR[k]=SR[i++];
8           else
9               TR[k]=SR[j++];
10      }
11      if（i<=m）
12      {
13          for（l=0;l<=m-i;l++）
14              TR[k+l]=SR[i+l];          /* 将剩余的 SR[i..m]复制到 TR */
15      }
16      if（j<=n）
17      {
18          for（l=0;l<=n-j;l++）
19              TR[k+l]=SR[j+l];          /* 将剩余的 SR[j..n]复制到 TR */
20      }
21  }
```

1．假设我们此时调用的 Merge 就是将{10,30,50,70,90}与{20,40,60,80}归并为最终有序的序列，因此数组 SR 为{10,30,50,70,90,20,40,60,80}，i=1，m=5，n=9。

2．第 4 行，for 循环，j 由 m+1=6 开始到 9，i 由 1 开始到 5，k 由 1 开始每次加 1，k 值用于目标数组 TR 的下标。

3．第 6 行，SR[i]=SR[1]=10，SR[j]= SR[6]=20，SR[i]<SR[j]，执行第 7 行，TR[k]=TR[1]=10，并且 i++。如图 9-8-7 所示。

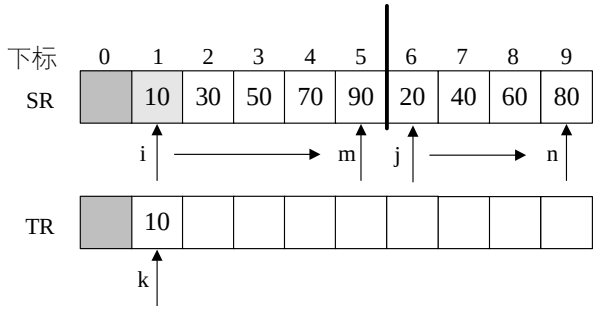


图 9-8-7

4．再次循环，k++得到 k=2，SR[i]=SR[2]=30，SR[j]= SR[6]=20，SR[i]>SR[j]，执

行第 9 行，TR[k]=TR[2]=20，并且 j++，如图 9-8-8 所示。

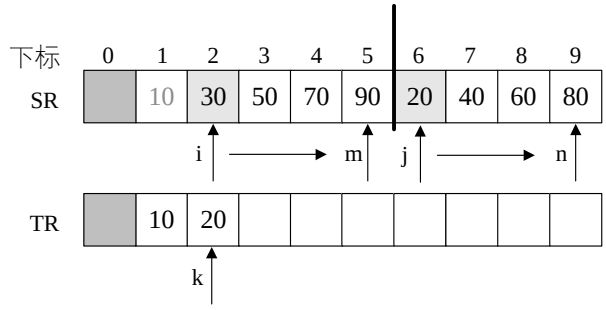


图 9-8-8

5．再次循环，k++得到 k=3，SR[i]=SR[2]=30，SR[j]= SR[7]=40，SR[i]<SR[j]，执行第 7 行，TR[k]=TR[3]=30，并且 i++，如图 9-8-9 所示。

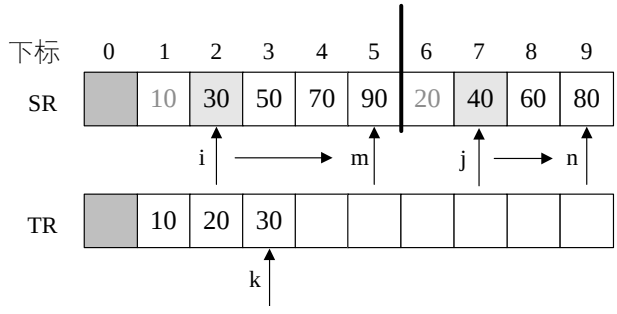


图 9-8-9

6．接下来完全相同的操作，一直到 j++后，j=10，大于 9 退出循环，如图 9-8-10 所示。

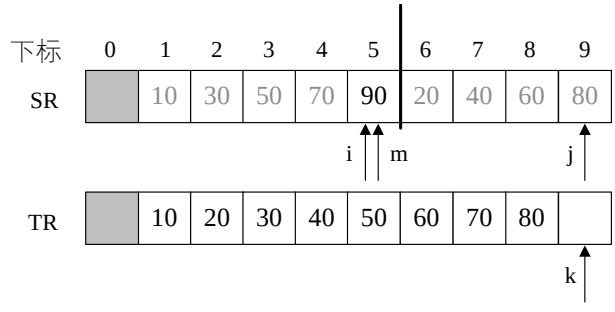


图 9-8-10

7．第 11～20 行的代码，其实就将归并剩下的数组数据，移动到 TR 的后面。当前 k=9，i=m=5，执行第 13～20 行代码，for 循环 l=0，TR[k+l]=SR[i+l]=90，大功告成。

就这样，我们的归并排序就算是完成了一次排序工作，怎么样，和堆排序比，是不是要简单一些呢?

9.8.2 归并排序复杂度分析

我们来分析一下归并排序的时间复杂度，一趟归并需要将SR[1]～SR[n]中相邻的长度为h的有序序列进行两两归并。并将结果放到TR1[1]～TR1[n]中，这需要将待排序序列中的所有记录扫描一遍，因此耗费O(n)时间，而由完全二叉树的深度可知，整个归并排序需要进行⌈$\log_2 n$⌉次，因此，总的时间复杂度为O($n\log n$)，而且这是归并排序算法中最好、最坏、平均的时间性能。

由于归并排序在归并过程中需要与原始记录序列同样数量的存储空间存放归并结果以及递归时深度为$\log_2 n$的栈空间，因此空间复杂度为O($n+\log n$)。

另外，对代码进行仔细研究，发现 Merge 函数中有 if (SR[i]<SR[j])语句，这就说明它需要两两比较，不存在跳跃，因此归并排序是一种稳定的排序算法。

也就是说，归并排序是一种比较占用内存，但却效率高且稳定的算法。

9.8.3 非递归实现归并排序

我们常说，“没有最好，只有更好。”归并排序大量引用了递归，尽管在代码上比较清晰，容易理解，但这会造成时间和空间上的性能损耗。我们排序追求的就是效率，有没有可能将递归转化成迭代呢？结论当然是可以的，而且改动之后，性能上进一步提高了，来看代码。

```
/* 对顺序表 L 作归并非递归排序 */
1   void MergeSort2(SqList *L)
2   {
3       int* TR=(int*)malloc(L->length*sizeof(int));/*申请额外空间*/
4       int k=1;
5       while(k<L->length)
6       {
7           MergePass(L->r,TR,k,L->length);
8           k=2*k;                          /* 子序列长度加倍 */
9           MergePass(TR,L->r,k,L->length);
10          k=2*k;                          /* 子序列长度加倍 */
11      }
```

```
12 }
```

1．程序开始执行，数组 L 为{50,10,90,30,70,40,80,60,20}，L.length=9。

2．第 3 行，我们事先申请了额外的数组内存空间，用来存放归并结果。

3．第 5～11 行，是一个 while 循环，目的是不断地归并有序序列。注意 k 值的变化，第 8 行与第 10 行，在不断循环中，它将由 1→2→4→8→16，跳出循环。

4．第 7 行，此时 k=1，MergePass 函数将原来的无序数组两两归并入 TR（此函数代码稍后再讲），如图 9-8-11 所示。

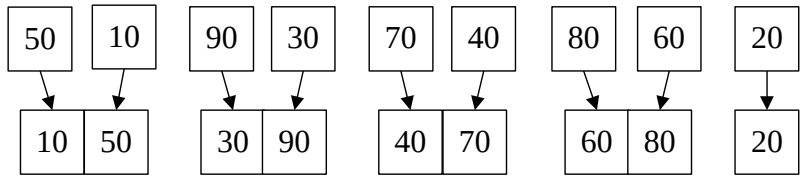


图 9-8-11

5．第 8 行，k=2。

6．第 9 行，MergePass 函数将 TR 中已经两两归并的有序序列再次归并回数组 L.r 中，如图 9-8-12 所示。

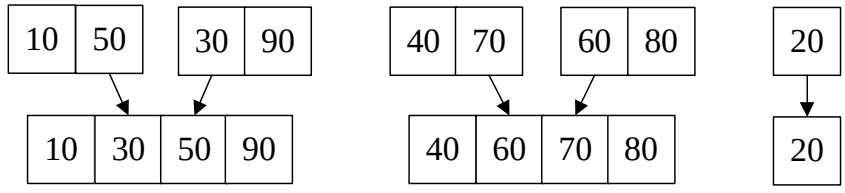


图 9-8-12

7．第 10 行，k=4，因为 k<9，所以继续循环，再次归并，最终执行完第 7～10 行，k=16，结束循环，完成排序工作，如图 9-8-13 所示。

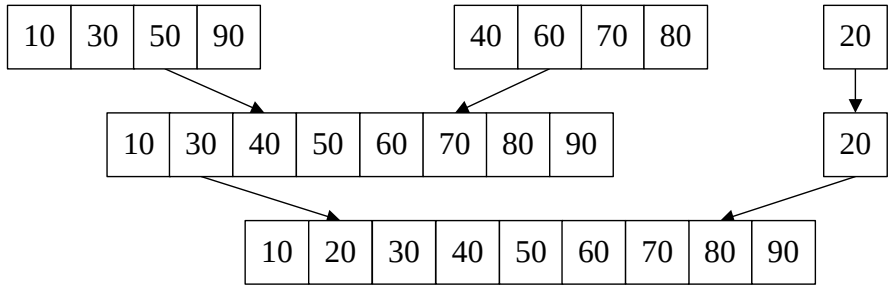


图 9-8-13

从代码中，我们能够感受到，非递归的迭代做法更加直截了当，从最小的序列开始归并直至完成。不需要像归并的递归算法一样，需要先拆分递归，再归并退出递归。

现在我们来看 MergePass 代码是如何实现的。

```
/* 将SR[]中相邻长度为 s 的子序列两两归并到 TR[] */
1   void MergePass（int SR[],int TR[],int s,int n）
2   {
3       int i=1;
4       int j;
5       while（i <= n-2*s+1）
6       {
7           Merge（SR,TR,i,i+s-1,i+2*s-1）; /* 两两归并 */
8           i=i+2*s;
9       }
10      if（i<n-s+1）  /* 归并最后两个序列 */
11          Merge（SR,TR,i,i+s-1,n）;
12      else          /* 若最后只剩下单个子序列 */
13          for（j =i;j <= n;j++）
14              TR[j] = SR[j];
15  }
```

1．程序执行。我们第一次调用“MergePass（L.r,TR,k,L.length）;”，此时 L.r 是初始无序状态，TR 为新申请的空数组，k=1，L.length=9。

2．第 5～9 行，循环的目的就两两归并，因 s=1，n-2×s+1=8，为什么循环 i 从 1 到 8，而不是 9 呢？就是因为两两归并，最终 9 条记录定会剩下来，无法归并。

3．第 7 行，Merge 函数我们前面已经详细讲过，此时 i=1，i+s-1=1，i+2×s-1=2。也就是说，我们将 SR（即 L.r）中的第一个和第二个记录归并到 TR 中，然后第 8 行，i=i+2×s=3，再循环，我们就是将第三个和第四个记录归并到 TR 中，一直到第七和第八个记录完成归并，如图 9-8-14 所示。

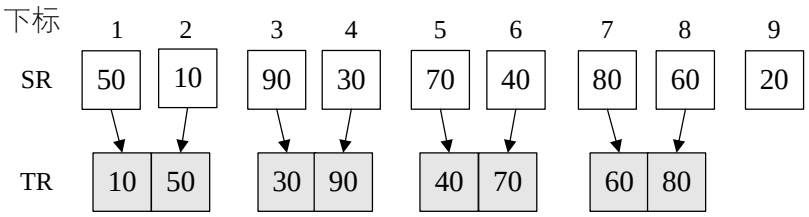


图 9-8-14

4．第 10～14 行，主要是处理最后的尾数，第 11 行是说将最后剩下的多个记录归并到 TR 中。不过由于 i=9，n-s+1=9，因此执行第 13～14 行，将 20 放入到 TR 数组的最后。

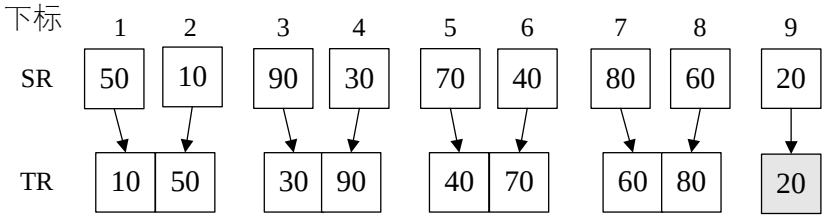


图 9-8-15

5．再次调用 MergePass 时，s=2，第 5～9 行的循环，由第 8 行的 i=i+2×s 可知，此时 i 就是以 4 为增量进行循环了，也就是说，是将两个有两个记录的有序序列进行归并为四个记录的有序序列。最终再将最后剩下的第九条记录“20”插入 TR。

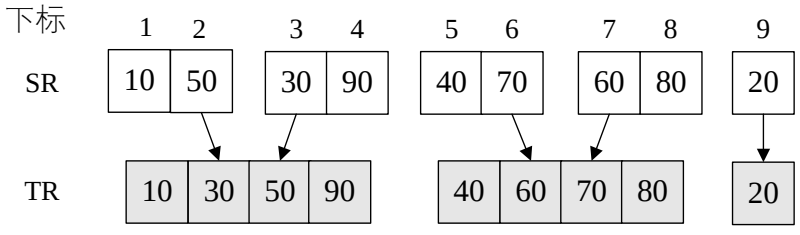


图 9-8-16

6．后面的类似，略。

非递归的迭代方法，避免了递归时深度为$\log_2 n$的栈空间，空间只是用到申请归并临时用的TR数组，因此空间复杂度为$O(n)$，并且避免递归也在时间性能上有一定的提升，**应该说，使用归并排序时，尽量考虑用非递归方法。**[9]

注：[9]关于归并排序算法更详细讲解，请参考《算法导论》第一部分第 2 章“算法入门”的 2.3.1 节“分治法”的内容。

9.9 快速排序

终于我们的高手要登场了，如果将来你工作后，你的老板要让你写个排序算法，而你会的算法中竟然没有快速排序，我想你还是不要声张，偷偷去把快速排序算法找来敲进电脑，这样至少你不至于被大伙儿取笑。

事实上，不论是 C++ STL、Java SDK 或者.NET FrameWork SDK 等开发工具包中的源代码中都能找到它的某种实现版本。

快速排序算法最早由图灵奖获得者 Tony Hoare 设计出来的，他在形式化方法理论以及 ALGOL60 编程语言的发明中都有卓越的贡献，是上世纪最伟大的计算机科学家之一。而这快速排序算法只是他众多贡献中的一个小发明而已。

更牛的是，我们现在要学习的这个快速排序算法，被列为 20 世纪 10 大算法之一。我们这些玩编程的人还有什么理由不去学习它呢?

希尔排序相当于直接插入排序的升级，它们同属于插入排序类，堆排序相当于简单选择排序的升级，它们同属于选择排序类。而快速排序其实就是我们前面认为最慢的冒泡排序的升级，它们都属于交换排序类。即它也是通过不断比较和移动交换来实现排序的，只不过它的实现，增大了记录的比较和移动的距离，将关键字较大的记录从前面直接移动到后面，关键字较小的记录从后面直接移动到前面，从而减少了总的比较次数和移动交换次数。

9.9.1 快速排序算法

快速排序（Quick Sort）的基本思想是：通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序的目的。

从字面上感觉不出它的好处来。假设现在要对数组{50,10,90,30,70,40,80,60,20}进行排序。我们通过代码的讲解来学习快速排序的精妙。

我们来看代码。

```
/* 对顺序表 L 作快速排序 */
void QuickSort（SqList *L）
{
    QSort（L,1,L->length）;
```

```
}
```

又是一句代码，和归并排序一样，由于需要递归调用，因此我们外封装了一个函数。现在我们来看 QSort 的实现。

```
/* 对顺序表 L 中的子序列 L->r[low..high]作快速排序 */
void QSort（SqList *L,int low,int high）
{
    int pivot;
    if（low<high）
    {
        pivot=Partition（L,low,high）; /*将 L->r[low..high]一分为二，*/
                                       /*算出枢轴值 pivot */
        QSort（L,low,pivot-1）;        /*对低子表递归排序 */
        QSort（L,pivot+1,high）;       /*对高子表递归排序 */
    }
}
```

从这里，你应该能理解前面代码“QSort(L,1,L->length);”中 1 和 L->length 代码的意思了，它就是当前待排序的序列最小下标值 low 和最大下标值 high。

这一段代码的核心是“pivot=Partition(L,low,high);”在执行它之前，L.r 的数组值为{50,10,90,30,70,40,80,60,20}。**Partition 函数要做的，就是先选取当中的一个关键字，**比如选择第一个关键字 50，然后**想尽办法将它放到一个位置，使得它左边的值都比它小，右边的值比它大，我们将这样的关键字称为枢轴（pivot）。**

在经过Partition(L,1,9)的执行之后，数组变成{20,10,40,30,**50**,70,80,60,90}，并返回值 5 给pivot，数字 5 表明 50 放置在数组下标为 5 的位置。此时，计算机把原来的数组变成了两个位于 50 左和右小数组{20,10,40,30}和{70,80,60,90}，而后的递归调用“QSort(L,1,5-1);”和“QSort(L,5+1,9);”语句，其实就是在对{20,10,40,30}和{70,80,60,90}分别进行同样的Partition操作，直到顺序全部正确为止。

到了这里，应该说理解起来还不算困难。下面我们就来看看快速排序最关键的 Partition 函数实现。

```
/* 交换顺序表 L 中子表的记录，使枢轴记录到位，并返回其所在位置 */
/* 此时在它之前（后）的记录均不大（小）于它。 */
1   int Partition（SqList *L,int low,int high）
2   {
3       int pivotkey;
```

```
4       pivotkey=L->r[low];       /* 用子表的第一个记录作枢轴记录 */
5       while（low<high）         /* 从表的两端交替向中间扫描 */
6       {
7            while（low<high&&L->r[high]>=pivotkey）
8              high--;
9            swap（L,low,high）;  /* 将比枢轴记录小的记录交换到低端 */
10           while（low<high&&L->r[low]<=pivotkey）
11             low++;
12           swap（L,low,high）;  /* 将比枢轴记录大的记录交换到高端 */
13      }
14      return low;               /* 返回枢轴所在位置 */
15 }
```

1．程序开始执行，此时 low=1，high=L.length=9。第 4 行，我们将 L.r[low]=L.r[1]=50 赋值给枢轴变量 pivotkey，如图 9-9-1 所示。

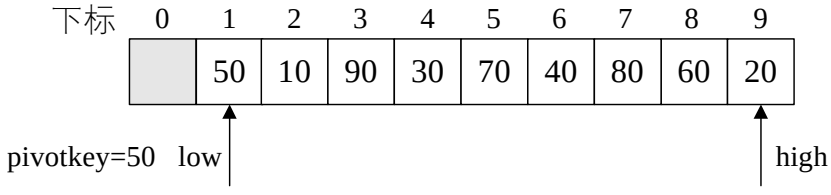


图 9-9-1

2．第 5～13 行为 while 循环，目前 low=1<high=9，执行内部语句。

3．第 7 行，L.r[high]= L.r[9]=20＞pivotkey=50，因此不执行第 8 行。

4．第 9 行，交换 L.r[low]与 L.r[high]的值，使得 L.r[1]=20，L.r[9]=50。为什么要交换，就是因为通过第 7 行的比较知道，L.r[high]是一个比 pivotkey=50（即 L.r[low]）还要小的值，因此它应该交换到 50 的左侧，如图 9-9-2 所示。

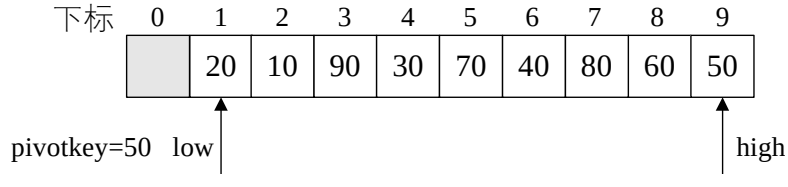


图 9-9-2

5．第 10 行，当 L.r[low]= L.r[1]=20，pivotkey=50，L.r[low]<pivotkey，因此第 11 行，low++，此时 low=2。继续循环，L.r[2]=10<50，low++，此时 low=3。L.r[3]=90>50，退出循环。

6．第 12 行，交换 L.r[low]=L.r[3]与 L.r[high]=L.r[9]的值，使得 L.r[3]=50，L.r[9]=90。此时相当于将一个比 50 大的值 90 交换到了 50 的右边。注意此时 low 已经指向了 3，如图 9-9-3 所示。

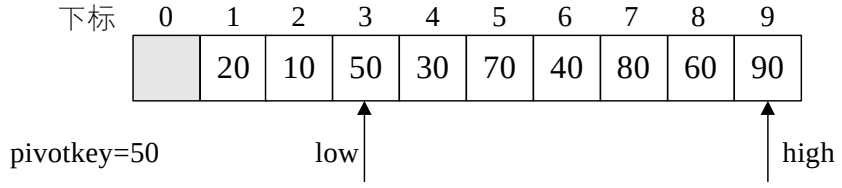


图 9-9-3

7．继续第 5 行，因为 low=3<high=9，执行循环体。

8．第 7 行，当 L.r[high]= L.r[9]=90，pivotkey=50，L.r[high]>pivotkey，因此第 8 行，high--，此时 high=8。继续循环，L.r[8]=60>50，high--，此时 high=7。L.r[7]=80>50，high--，此时 high=6。L.r[6]=40<50，退出循环。

9．第 9 行，交换 L.r[low]=L.r[3]=50 与 L.r[high]=L.r[6]=40 的值，使得 L.r[3]=40，L.r[6]=50，如图 9-9-4 所示。

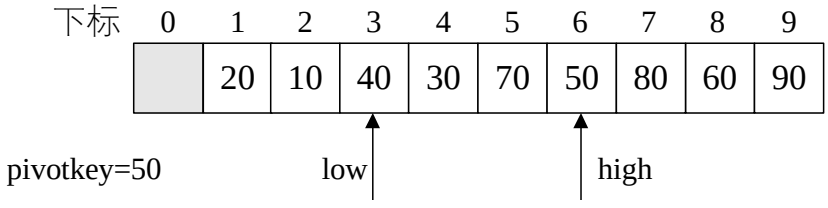


图 9-9-4

10．第 10 行，当 L.r[low]= L.r[3]=40，pivotkey=50，L.r[low]<pivotkey，因此第 11 行，low++，此时 low=4。继续循环 L.r[4]=30<50，low++，此时 low=5。L.r[5]=70>50，退出循环。

11．第 12 行，交换 L.r[low]=L.r[5]=70 与 L.r[high]=L.r[6]=50 的值，使得 L.r[5]=50，L.r[6]=70，如图 9-9-5 所示。

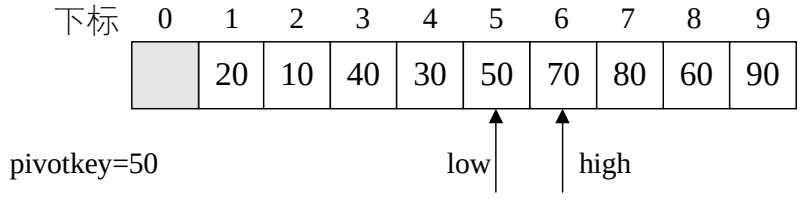


图 9-9-5

12．再次循环。因 low=5<high=6，执行循环体后，low=high=5，退出循环，如图 9-9-6 所示。

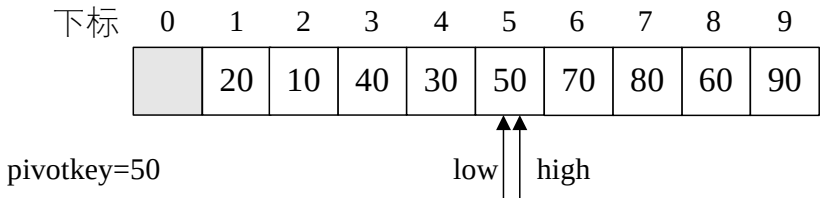


图 9-9-6

13．最后第 14 行，返回 low 的值 5。函数执行完成。接下来就是递归调用“QSort（L,1,5-1）;”和“QSort（L,5+1,9）;”语句，对{20,10,40,30}和{70,80,60,90}分别进行同样的 Partition 操作，直到顺序全部正确为止。我们就不再演示了。

通过这段代码的模拟，大家应该能够明白，Partition 函数，其实就是将选取的 pivotkey 不断交换，将比它小的换到它的左边，比它大的换到它的右边，它也在交换中不断更改自己的位置，直到完全满足这个要求为止。

9.9.2 快速排序复杂度分析

我们来分析一下快速排序法的性能。快速排序的时间性能取决于快速排序递归的深度，可以用递归树来描述递归算法的执行情况。如图 9-9-7 所示，它是{50,10,90,30,70,40,80,60,20}在快速排序过程中的递归过程。由于我们的第一个关键字是 50，正好是待排序的序列的中间值，因此递归树是平衡的，此时性能也比较好。

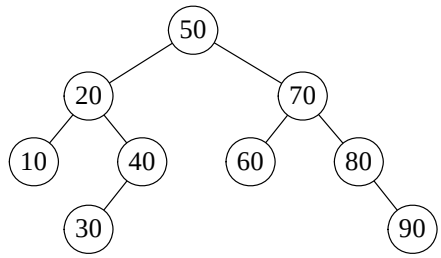


图 9-9-7

在最优情况下，Partition每次都划分得很均匀，如果排序n个关键字，其递归树的深度就为$\lfloor \log_2 n \rfloor+1$（$\lfloor x \rfloor$表示不大于x的最大整数），即仅需递归$\log_2 n$次，需要时间为T（$n$）的话，第一次Partiation应该是需要对整个数组扫描一遍，做n次比较。然后，获得的枢轴将数组一分为二，那么各自还需要T（$n/2$）的时间（注意是最好情况，所以

平分两半)。于是不断地划分下去，我们就有了下面的不等式推断。

```
T(n)≤2T(n/2) +n, T(1)=0
T(n)≤2(2T(n/4)+n/2) +n=4T(n/4)+2n
T(n)≤4(2T(n/8)+n/4) +2n=8T(n/8)+3n
……
T(n)≤nT(1)+(log2n)×n= O(nlogn)
```

也就是说，在最优的情况下，快速排序算法的时间复杂度为 O(*n*log*n*)。

在最坏的情况下，待排序的序列为正序或者逆序，每次划分只得到一个比上一次划分少一个记录的子序列，注意另一个为空。如果递归树画出来，它就是一棵斜树。此时需要执行n-1 次递归调用，且第i次划分需要经过n-i次关键字的比较才能找到第i个记录，也就是枢轴的位置，因此比较次数为 $\sum_{i=1}^{n-1}(n-i)=n-1+n-2+\cdots+1=\frac{n(n-1)}{2}$，最终其时间复杂度为O($n^2$)。

平均的情况，设枢轴的关键字应该在第 k 的位置（1≤k≤n），那么：

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

由数学归纳法可证明，其数量级为 O(*n*log*n*)。

就空间复杂度来说，主要是递归造成的栈空间的使用，最好情况，递归树的深度为$\log_2 n$，其空间复杂度也就为O(log*n*)，最坏情况，需要进行n-1 递归调用，其空间复杂度为O(*n*)，平均情况，空间复杂度也为O(log*n*)。

可惜的是，由于关键字的比较和交换是跳跃进行的，因此，快速排序是一种不稳定的排序方法。

9.9.3 快速排序优化

刚才讲的快速排序还是有不少可以改进的地方，我们来看一些优化的方案。

1. 优化选取枢轴

如果我们选取的 pivotkey 是处于整个序列的中间位置，那么我们可以将整个序列分成小数集合和大数集合了。但注意，我刚才说的是“如果……是中间”，那么假如我们选取的 pivotkey 不是中间数又如何呢？比如我们前面讲冒泡和简单选择排序一直用到的数组{9,1,5,8,3,7,4,6,2}，由代码第 4 行“pivotkey=L->r[low];”知道，我们应该选

取 9 作为第一个枢轴 pivotkey。此时，经过一轮“pivot=Partition（L,1,9）;”转换后，它只是更换了 9 与 2 的位置，并且返回 9 给 pivot，整个系列并没有实质性的变化，如图 9-9-8 所示。

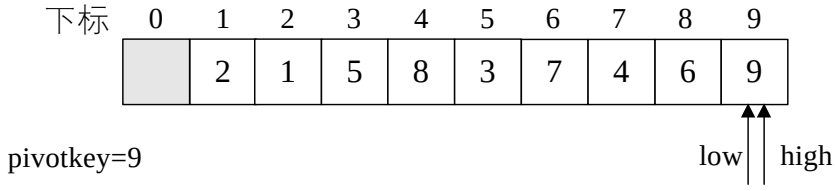


图 9-9-8

就是说，代码第 4 行“pivotkey=L->r[low];”变成了一个潜在的性能瓶颈。排序速度的快慢取决于 L.r[1]的关键字处在整个序列的位置，L.r[1]太小或者太大，都会影响性能(比如第一例子中的 50 就是一个中间数，而第二例子的 9 就是一个相对整个序列过大的数)。因为在现实中，待排序的系列极有可能是基本有序的，此时，总是**固定选取**第一个关键字（其实无论是固定选取哪一个位置的关键字）作为首个枢轴就变成了极为不合理的作法。

改进办法，有人提出，应该随机获得一个 low 与 high 之间的数 rnd，让它的关键字 L.r[rnd]与 L.r[low]交换，此时就不容易出现这样的情况，这被称为**随机选取**枢轴法。应该说，这在某种程度上，解决了对于基本有序的序列快速排序时的性能瓶颈。不过，随机就有些撞大运的感觉，万一没撞成功，随机到了依然是很小或很大的关键字怎么办呢?

再改进，于是就有了**三数取中（median-of-three）**法。即**取三个关键字先进行排序，将中间数作为枢轴，一般是取左端、右端和中间三个数，**也可以随机选取。这样至少这个中间数一定不会是最小或者最大的数，从概率来说，取三个数均为最小或最大数的可能性是微乎其微的，因此中间数位于较为中间的值的可能性就大大提高了。由于整个序列是无序状态，随机选取三个数和从左中右端取三个数其实是一回事，而且随机数生成器本身还会带来时间上的开销，因此随机生成不予考虑。

我们来看看取左端、右端和中间三个数的实现代码，在 Partition 函数代码的第 3 行与第 4 行之间增加这样一段代码。

```
3   int pivotkey;

    int m = low + （high - low） / 2; /* 计算数组中间的元素的下标 */
    if （L->r[low]>L->r[high]）
```

```
        swap(L,low,high);              /* 交换左端与右端数据，保证左端较小 */
    if (L->r[m]>L->r[high])
        swap(L,high,m);                /* 交换中间与右端数据，保证中间较小 */
    if (L->r[m]>L->r[low])
        swap(L,m,low);                 /* 交换中间与左端数据，保证左端较小 */
    /* 此时L.r[low]已经为整个序列左中右三个关键字的中间值。*/

4   pivotkey=L->r[low];                /* 用子表的第一个记录作枢轴记录 */
```

试想一下，我们对数组{9,1,5,8,3,7,4,6,2}，取左 9、中 3、右 2 来比较，使得 L.r[low]=3，一定要比 9 和 2 来得更为合理。

三数取中对小数组来说有很大的概率选择到一个比较好的 pivotkey，但是对于非常大的待排序的序列来说还是不足以保证能够选择出一个好的 pivotkey，因此还有个办法是所谓**九数取中（median-of-nine）**，它先从数组中分三次取样，每次取三个数，三个样品各取出中数，然后从这三个中数当中再取出一个中数作为枢轴。显然这就更加保证了取到的 pivotkey 是比较接近中间值的关键字。有兴趣的同学可以自己去实现一下代码，这里不再详述了。

2. 优化不必要的交换

观察图 9-9-1～图 9-9-6，我们发现，50 这个关键字，其位置变化是 1→9→3→6→5，可其实它的最终目标就是 5，当中的交换其实是不需要的。因此我们对 Partition 函数的代码再进行优化。

```
/* 快速排序优化算法 */
int Partition1(SqList *L,int low,int high)
{
    int pivotkey;
    //这里省略三数取中代码
    pivotkey=L->r[low];          /* 用子表的第一个记录作枢轴记录 */
    L->r[0]=pivotkey;            /* 将枢轴关键字备份到L->r[0] */
    while(low<high)              /* 从表的两端交替向中间扫描 */
    {
        while(low<high&&L->r[high]>=pivotkey)
            high--;
        L->r[low]=L->r[high];    /* 采用替换而不是交换的方式进行操作 */
        while(low<high&&L->r[low]<=pivotkey)
            low++;
```

```
            L->r[high]=L->r[low];   /* 采用替换而不是交换的方式进行操作 */
        }
        L->r[low]=L->r[0];          /* 将枢轴数值替换回 L.r[low] */
        return low;                 /* 返回枢轴所在位置 */
    }
```

注意代码中**加粗**部分的改变。我们事实将 pivotkey 备份到 L.r[0]中，然后在之前是 swap 时，只作替换的工作，最终当 low 与 high 会合，即找到了枢轴的位置时，再将 L.r[0]的数值赋值回 L.r[low]。因为这当中少了多次交换数据的操作，在性能上又得到了部分的提高。如图 9-9-9 所示。

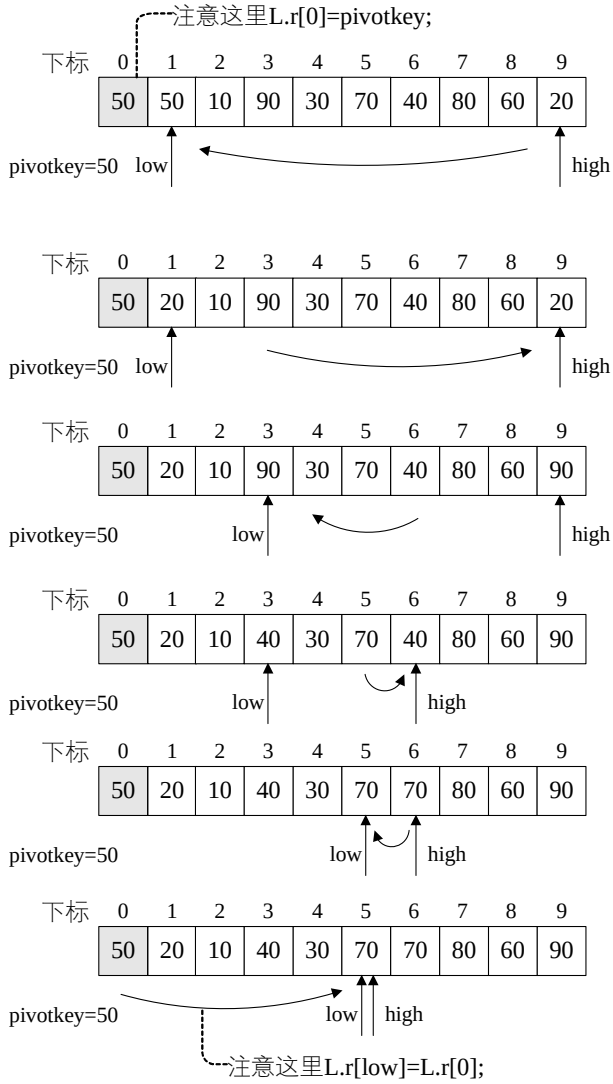


图 9-9-9

3. 优化小数组时的排序方案

对于一个数学科学家、博士生导师，他可以攻克世界性的难题，可以培养最优秀的数学博士，但让他去教小学生“1+1=2”的算术课程，那还真未必会比常年在小学里耕耘的数学老师教得好。换句话说，大材小用有时会变得反而不好用。刚才我谈到了对于非常大的数组的解决办法。那么相反的情况，如果数组非常小，其实快速排序反而不如直接插入排序来得更好（直接插入是简单排序中性能最好的）。其原因在于快速排序用到了递归操作，在大量数据排序时，这点性能影响相对于它的整体算法优势而言是可以忽略的，但如果数组只有几个记录需要排序时，这就成了一个大炮打蚊子的大问题。因此我们需要改进一下 QSort 函数。

```
#define MAX_LENGTH_INSERT_SORT 7    /* 数组长度阈值 */
/* 对顺序表 L 中的子序列 L.r[low..high]作快速排序 */
void QSort（SqList &L,int low,int high）
{
    int pivot;
    if（（high-low）>MAX_LENGTH_INSERT_SORT）
    {/*当 high-low 大于常数时用快速排序*/
        pivot=Partition（L,low,high）; /* 将 L.r[low..high]一分为二，*/
                                       /* 并算出枢轴值 pivot */
        QSort（L,low,pivot-1）;        /* 对低子表递归排序 */
        QSort（L,pivot+1,high）;       /* 对高子表递归排序 */
    }
    else   /* 当 high-low 小于等于常数时用直接插入排序 */
        InsertSort（L）;
}
```

我们增加了一个判断，当 high-low 不大于某个常数时（有资料认为 7 比较合适，也有认为 50 更合理，实际应用可适当调整），就用直接插入排序，这样就能保证最大化地利用两种排序的优势来完成排序工作。

4. 优化递归操作

大家知道，递归对性能是有一定影响的，QSort函数在其尾部有两次递归操作。如果待排序的序列划分极端不平衡，递归深度将趋近于n，而不是平衡时的$\log_2 n$，这就不仅仅是速度快慢的问题了。栈的大小是很有限的，每次递归调用都会耗费一定的栈空间，函数的参数越多，每次递归耗费的空间也越多。因此如果能减少递归，将会大大提高性能。

于是我们对 QSort 实施尾递归优化。来看代码。

```
/* 对顺序表 L 中的子序列 L.r[low..high]作快速排序 */
void QSort1（SqList *L,int low,int high）
{
    int pivot;
    if（（high-low）>MAX_LENGTH_INSERT_SORT）
    {
        while（low<high）
        {
            pivot=Partition1（L,low,high）; /*L.r[low..high]一分为二,*/
                                            /*算出枢轴值 pivot */
            QSort1（L,low,pivot-1）;         /* 对低子表递归排序 */
            low=pivot+1;                    /* 尾递归 */
        }
    }
    else
        InsertSort（L）;
}
```

当我们将 if 改成 while 后（见**加粗**代码部分），因为第一次递归以后，变量 low 就没有用处了，所以可以将 pivot+1 赋值给 low，再循环后，来一次 Partition（L,low,high），其效果等同于“QSort（L,pivot+1,high）;”。结果相同，但因采用迭代而不是递归的方法可以缩减堆栈深度，从而提高了整体性能。

在现实的应用中，比如C++、java、PHP、C#、VB、JavaScript等都有对快速排序算法的实现[10]，实现方式上略有不同，但基本上都是在我们讲解的快速排序法基础上的精神体现。

我们现在学过的排序算法，有按照实现方法分类命名的，如简单选择排序、直接插入排序、归并排序，有按照其排序的方式类比现实世界命名的，比如冒泡排序、堆排序，还有用人名命名的，比如希尔排序。但是刚才我们讲的排序，却用“快速”来命名，这也就意味着只要再有人找到更好的排序法，此“快速”就会名不符实，不过，至少今天，TonyHoare发明的快速排序法经过多次的优化后，在整体性能上，依

注：[10]有兴趣可以想办法到网上下载阅读它们的源代码。

然是排序算法王者，我们应该要好好研究并掌握它。[11]

9.10 总结回顾

本章内容只是在讲排序，我们需要对已经提到的各个排序算法进行对比来总结回顾。

首先我们讲了排序的定义，并提到了排序的稳定性，排序稳定对于某些特殊需求来说是至关重要的，因此在排序算法中，我们需要关注此算法的稳定性如何。

我们根据将排序记录是否全部被放置在内存中，将排序分为内排序与外排序两种，外排序需要在内外存之间多次交换数据才能进行。我们本章主要讲的是内排序的算法。

根据排序过程中借助的主要操作，我们将内排序分为：插入排序、交换排序、选择排序和归并排序四类。之后介绍的 7 种排序法，就分别是各种分类的代表算法。

图 9-10-1

事实上，目前还没有十全十美的排序算法，有优点就会有缺点，即使是快速排序法，也只是在整体性能上优越，它也存在排序不稳定、需要大量辅助空间、对少量数据排序无优势等不足。因此我们就来从多个角度来剖析一下提到的各种排序的长与短。

我们将 7 种算法的各种指标进行对比，如表 9-10-1 所示。

表 9-10-1

注：[11]关于快速排序算法更详细讲解，请参考《算法导论》第二部分第七章“快速排序”的内容。

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	O(1)	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	O(1)	稳定
希尔排序	$O(n\log n) \sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	O(1)	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	O(1)	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n) \sim O(n)$	不稳定

从算法的简单性来看，我们将 7 种算法分为两类：

- 简单算法：冒泡、简单选择、直接插入。
- 改进算法：希尔、堆、归并、快速。

从平均情况来看，显然最后 3 种改进算法要胜过希尔排序，并远远胜过前 3 种简单算法。

从最好情况看，反而冒泡和直接插入排序要更胜一筹，也就是说，如果你的待排序序列总是基本有序，反而不应该考虑 4 种复杂的改进算法。

从最坏情况看，堆排序与归并排序又强过快速排序以及其他简单排序。

从这三组时间复杂度的数据对比中，我们可以得出这样一个认识。堆排序和归并排序就像两个参加奥数考试的优等生，心理素质强，发挥稳定。而快速排序像是很情绪化的天才，心情好时表现极佳，碰到较糟糕环境会变得差强人意。但是他们如果都来比赛计算个位数的加减法，它们反而算不过成绩极普通的冒泡和直接插入。

从空间复杂度来说，归并排序强调要马跑得快，就得给马吃个饱。快速排序也有相应的空间要求，反而堆排序等却都是少量索取，大量付出，对空间要求是 O(1)。如果执行算法的软件所处的环境非常在乎内存使用量的多少时，选择归并排序和快速排序就不是一个较好的决策了。

从稳定性来看，归并排序独占鳌头，我们前面也说过，对于非常在乎排序稳定性的应用中，归并排序是个好算法。

从待排序记录的个数上来说，待排序的个数 n 越小，采用简单排序方法越合适。反之，n 越大，采用改进排序方法越合适。这也就是我们为什么对快速排序优化时，增加了一个阀值，低于阀值时换作直接插入排序的原因。

从表 9-10-1 的数据中，似乎简单选择排序在 3 种简单排序中性能最差，其实也不完全是，比如，如果记录的关键字本身信息量比较大（例如，关键字都是数十位的数

字)，此时表明其占用存储空间很大，这样移动记录所花费的时间也就越多，我们给出3种简单排序算法的移动次数比较，如表9-10-2所示。

表9-10-2

排序方法	平均情况	最好情况	最坏情况
冒泡排序	$O(n^2)$	0	$O(n^2)$
简单选择排序	$O(n)$	0	$O(n)$
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$

你会发现，此时简单选择排序就变得非常有优势，原因也就在于，它是通过大量比较后选择明确记录进行移动，有的放矢。因此对于数据量不是很大而记录的关键字信息量较大的排序要求，简单排序算法是占优的。另外，记录的关键字信息量大小对那四个改进算法影响不大。

总之，从综合各项指标来说，经过优化的快速排序是性能最好的排序算法，但是不同的场合我们也应该考虑使用不同的算法来应对它。

9.11 结 尾 语

学完排序，你能够感受到，我们的算法研究者们都是在“似乎不可能”的情况下，逐步提高排序算法的性能的。在剩下的几分钟时间里，我们再来做一道智力题，感受一下把不可能变为可能。

请问如何把图9-11-1中用四段直线一笔将这九个点连起来?

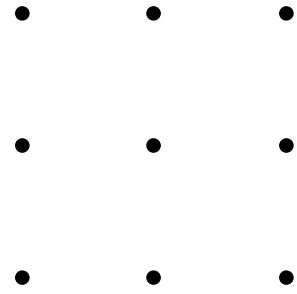


图9-11-1

大家举手很快，因为绝大多数同学应该都看过这道题目。没有做过题目的同学通常十有八九会落入一个小小的陷阱，在九个点围成的框中打转转，然后发现至少要五段以上的直线才能连成。结果是，要找到答案，必须在思维上突破这九个点所围成的

框框的限制，如图 9-11-2 所示。

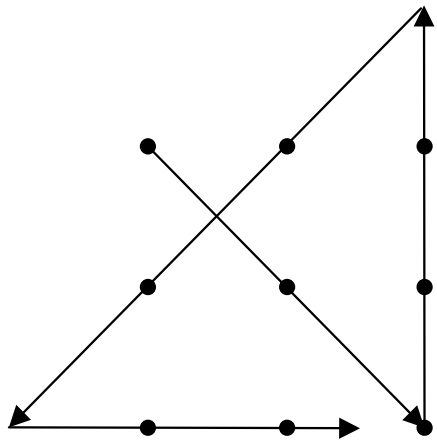


图 9-11-2

如果智力题这就结束了，那就不考大家了。现在我的问题是如何做到三段直线一笔将这九个点连起来?

此时，大家都在交头接耳，心里一定想着，“这怎么可能？”我来公布答案，那就是用一条“Z”字线即可一笔连成。也许，最快找出这个答案的是那些没有学过数学的孩子。作为成人，我们已被另一些“框框”所框住大脑。那就是数学上有一条基本公理：两条平行线永不相交。另外数学上有另一个基本假设：点没有大小。可在现实中任何一点都会有大小。突破这一限制，只要无限延长“Z”字三段线，九点必可一笔连。来看图 9-11-3。

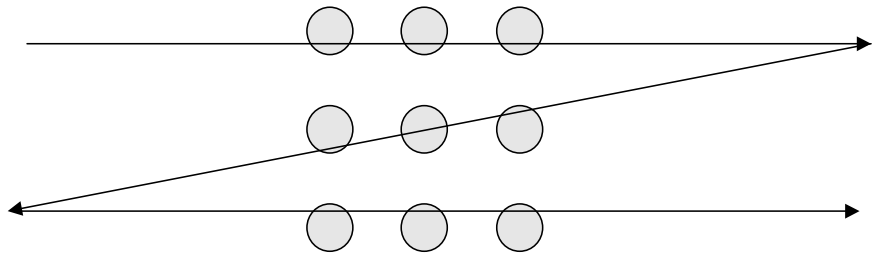


图 9-11-3

有同学说，我图中的点比刚才的要大，这不符合题意。我想有这样想法的同学，可能还是没有理解我想表达的意思，事实上，刚才的小黑点再小，它也是有大小的，你可以想像三根直线足够长，它们就可以将这九个点相连了。

别急，题目没完，我现在要求只用一条直线将这九点一笔连，如何做?

显然，大家的思维已经被打开。我们可轻易找到答案，因为只要再次突破几何学

中“线没粗细”的框框，用一条很粗的线，比如蘸了墨水的大刷子，画一条粗粗的直线将九点全部包含其中即可。

不是不可能用四段、三段、一段直线一笔连九点，只是暂时还没有找到方法而已。现实生活中所有的发明创造都是建立在打破前人所认定的“框框”的思维定势基础上的。这道智力题当然不是要挑战数学的权威，它只是在给我们启示：“所有的事情都是可能的，只是我们暂时还没有找到方法而已。”

本章的结束，其实也就是数据结构这门课的结束了。数据结构和算法，还有很多内容我们并没有涉及。要想真正掌握数据结构，并把它应用到工作中，你们的路还很长。

我们生命中，矛盾和困惑往往一直伴随。很多同学来学习数据结构，其实并不是真的明白它的重要性，通常只是因为学校开了这门课，而不得不来这里弄个 PASS，过后，真到需要用时，却发现力不从心而追悔莫及。比如图 9-11-4 所示，悲剧通常就是这样产生的。因此尽管现在是课程的最后，对于个别没有重视这门课的同学来说有些晚了，我还是想再亡羊补牢：**数据结构和算法对于程序员的职业人生来说，那就是两个圆圈的交集部分，用心去掌握它，你的编程之路将会是坦途。**

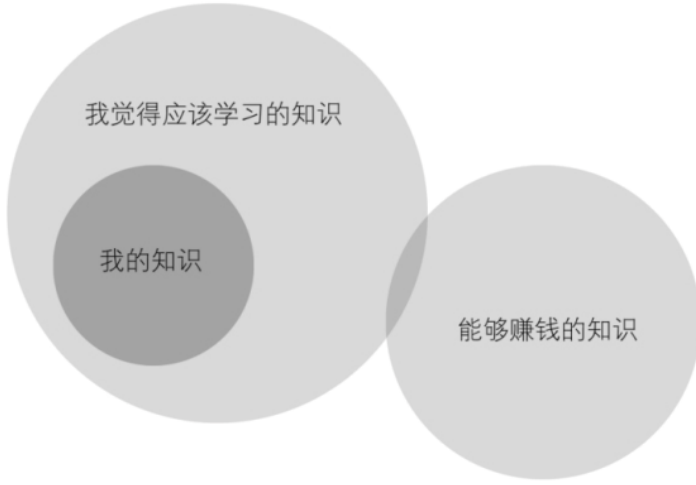


图 9-11-4

最后送大家电影《当幸福来敲门》中的一句话：

You got a dream,you gotta protect it. People can’t do something themselves, they wanna tell you you can’t do it. If you want something, go get it. Period.（如果你有梦想

的话，就要去捍卫它。当别人做不到的时候，他们就想要告诉你，你也不能。如果你想要些什么，就得去努力争取。就这样!）

同学们，再见!