目　录

第1章

一大波数正在靠近

排序

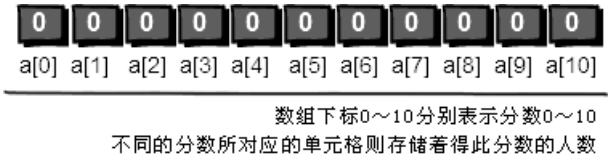
第 1 节　最快最简单的排序——桶排序

在我们生活的这个世界中到处都是被排序过的东东。站队的时候会按照身高排序，考试的名次需要按照分数排序，网上购物的时候会按照价格排序，电子邮箱中的邮件按照时间排序……总之很多东东都需要排序，可以说排序是无处不在。现在我们举个具体的例子来介绍一下排序算法。

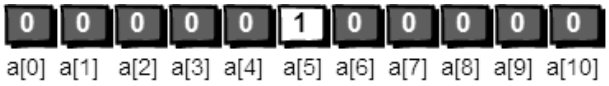
首先出场的是我们的主人公小哼，上面这个可爱的娃就是啦。期末考试完了老师要将同学们的分数按照从高到低排序。小哼的班上只有 5 个同学，这 5 个同学分别考了 5 分、3 分、5 分、2 分和 8 分，哎考得真是惨不忍睹（满分是 10 分）。接下来将分数进行从大到小排序，排序后是 8 5 5 3 2。你有没有什么好方法编写一段程序，让计算机随机读入 5 个数然后将这 5 个数从大到小输出？请先想一想，至少想 15 分钟再往下看吧(*^__^*) 。

我们这里只需借助一个一维数组就可以解决这个问题。请确定你真的仔细想过再往下看哦。

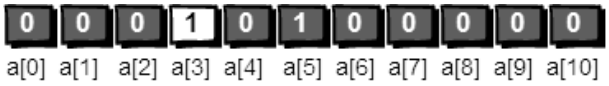
首先我们需要申请一个大小为 11 的数组 int a[11]。OK，现在你已经有了 11 个变量，编号从 a[0]~a[10]。刚开始的时候，我们将 a[0]~a[10]都初始化为 0，表示这些分数还都没有人得过。例如 a[0]等于 0 就表示目前还没有人得过 0 分，同理 a[1]等于 0 就表示目前还没有人得过 1 分……a[10]等于 0 就表示目前还没有人得过 10 分。



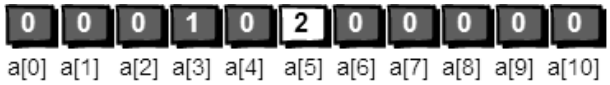
下面开始处理每一个人的分数，第一个人的分数是 5 分，我们就将相对应的 a[5]的值在原来的基础增加 1，即将 a[5]的值从 0 改为 1，表示 5 分出现过了一次。



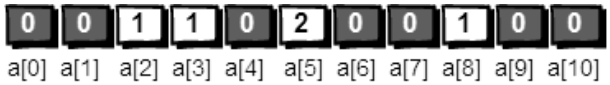
第二个人的分数是 3 分，我们就把相对应的 a[3]的值在原来的基础上增加 1，即将 a[3]的值从 0 改为 1，表示 3 分出现过了一次。



注意啦！第三个人的分数也是 5 分，所以 a[5]的值需要在此基础上再增加 1，即将 a[5]的值从 1 改为 2，表示 5 分出现过了两次。



按照刚才的方法处理第四个和第五个人的分数。最终结果就是下面这个图啦。



你发现没有，a[0]~a[10]中的数值其实就是 0 分到 10 分每个分数出现的次数。接下来，我们只需要将出现过的分数打印出来就可以了，出现几次就打印几次，具体如下。

a[0]为 0，表示“0”没有出现过，不打印。

a[1]为 0，表示“1”没有出现过，不打印。

a[2]为 1，表示“2”出现过 1 次，打印 2。

a[3]为 1，表示“3”出现过 1 次，打印 3。

a[4]为 0，表示“4”没有出现过，不打印。

a[5]为 2，表示“5”出现过 2 次，打印 5 5。

a[6]为 0，表示“6”没有出现过，不打印。

a[7]为 0，表示“7”没有出现过，不打印。

a[8]为 1，表示“8”出现过 1 次，打印 8。

a[9]为 0，表示“9”没有出现过，不打印。

a[10]为 0，表示“10”没有出现过，不打印。

最终屏幕输出“2 3 5 5 8”，完整的代码如下。

```
#include <stdio.h>
int main()
{
    int a[11],i,j,t;
    for(i=0;i<=10;i++)
        a[i]=0;  //初始化为0

    for(i=1;i<=5;i++)  //循环读入5个数
    {
```

```
        scanf("%d",&t);   //把每一个数读到变量t中
        a[t]++;   //进行计数
    }

    for(i=0;i<=10;i++)   //依次判断a[0]~a[10]
        for(j=1;j<=a[i];j++)   //出现了几次就打印几次
            printf("%d ",i);

    getchar();getchar();
    //这里的getchar();用来暂停程序，以便查看程序输出的内容
    //也可以用system("pause");等来代替
    return 0;
}
```

输入数据为：

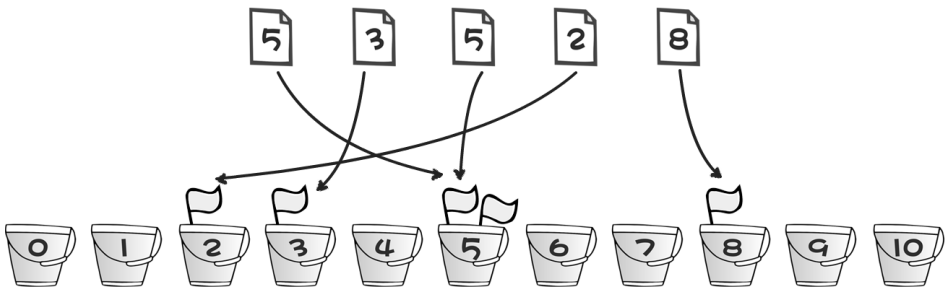
```
5 3 5 2 8
```

仔细观察的同学会发现，刚才实现的是从小到大排序。但是我们要求是从大到小排序，这该怎么办呢？还是先自己想一想再往下看哦。

其实很简单。只需要将 for(i=0;i<=10;i++)改为 for(i=10;i>=0;i--)就 OK 啦，快去试一试吧。

这种排序方法我们暂且叫它“桶排序”。因为其实真正的桶排序要比这个复杂一些，以后再详细讨论，目前此算法已经能够满足我们的需求了。

这个算法就好比有 11 个桶，编号从 0~10。每出现一个数，就在对应编号的桶中放一个小旗子，最后只要数数每个桶中有几个小旗子就 OK 了。例如 2 号桶中有 1 个小旗子，表示 2 出现了一次；3 号桶中有 1 个小旗子，表示 3 出现了一次；5 号桶中有 2 个小旗子，表示 5 出现了两次；8 号桶中有 1 个小旗子，表示 8 出现了一次。



现在你可以尝试一下输入 n 个 0~1000 之间的整数，将它们从大到小排序。提醒一下，

如果需要对数据范围在 0~1000 之间的整数进行排序，我们需要 1001 个桶，来表示 0~1000 之间每一个数出现的次数，这一点一定要注意。另外，此处的每一个桶的作用其实就是“标记”每个数出现的次数，因此我喜欢将之前的数组 a 换个更贴切的名字 book（book 这个单词有记录、标记的意思），代码实现如下。

```
#include <stdio.h>

int main()
{
    int book[1001],i,j,t,n;
    for(i=0;i<=1000;i++)
        book[i]=0;
    scanf("%d",&n);//输入一个数n，表示接下来有n个数
    for(i=1;i<=n;i++)//循环读入n个数，并进行桶排序
    {
        scanf("%d",&t);  //把每一个数读到变量t中
        book[t]++;  //进行计数，对编号为t的桶放一个小旗子
    }
    for(i=1000;i>=0;i--)  //依次判断编号1000~0的桶
        for(j=1;j<=book[i];j++)  //出现了几次就将桶的编号打印几次
             printf("%d ",i);

    getchar();getchar();
    return 0;
}
```

可以输入以下数据进行验证。

```
10
8 100 50 22 15 6 1 1000 999 0
```

运行结果是：

```
1000 999 100 50 22 15 8 6 1 0
```

最后来说下时间复杂度的问题。代码中第 6 行的循环一共循环了 *m* 次（*m* 为桶的个数），第 9 行的代码循环了 *n* 次（*n* 为待排序数的个数），第 14 行和第 15 行一共循环了 *m*+*n* 次。所以整个排序算法一共执行了 *m*+*n*+*m*+*n* 次。我们用大写字母 O 来表示时间复杂度，因此该

算法的时间复杂度是 $O(m+n+m+n)$即 $O(2*(m+n))$。我们在说时间复杂度的时候可以忽略较小的常数，最终桶排序的时间复杂度为 $O(m+n)$。还有一点，在表示时间复杂度的时候，n 和 m 通常用大写字母即 $O(M+N)$。

这是一个非常快的排序算法。桶排序从 1956 年就开始被使用，该算法的基本思想是由 E.J.Issac 和 R.C.Singleton 提出来的。之前我说过，其实这并不是真正的桶排序算法，真正的桶排序算法要比这个更加复杂。但是考虑到此处是算法讲解的第一篇，我想还是越简单易懂越好，真正的桶排序留在以后再聊吧。需要说明一点的是：我们目前学习的简化版桶排序算法，其本质上还不能算是一个真正意义上的排序算法。为什么呢？例如遇到下面这个例子就没辙了。

现在分别有 5 个人的名字和分数：huhu 5 分、haha 3 分、xixi 5 分、hengheng 2 分和 gaoshou 8 分。请按照分数从高到低，输出他们的名字。即应该输出 gaoshou、huhu、xixi、haha、hengheng。发现问题了没有？如果使用我们刚才简化版的桶排序算法仅仅是把分数进行了排序。最终输出的也仅仅是分数，但没有对人本身进行排序。也就是说，我们现在并不知道排序后的分数原本对应着哪一个人！这该怎么办呢？不要着急，请看下节——冒泡排序。

第 2 节　邻居好说话——冒泡排序

简化版的桶排序不仅仅有上一节所遗留的问题，更要命的是：它非常浪费空间！例如需要排序数的范围是 0~2100000000 之间，那你则需要申请 2100000001 个变量，也就是说要写成 int a[2100000001]。因为我们需要用 2100000001 个“桶”来存储 0~2100000000 之间每一个数出现的次数。即便只给你 5 个数进行排序（例如这 5 个数是 1、1912345678、2100000000、18000000 和 912345678），你也仍然需要 2100000001 个“桶”，这真是太浪费空间了！还有，如果现在需要排序的不再是整数而是一些小数，比如将 5.56789、2.12、1.1、3.123、4.1234 这五个数进行从小到大排序又该怎么办呢？现在我们来学习另一种新的排序算法：冒泡排序。它可以很好地解决这两个问题。

冒泡排序的基本思想是：每次比较两个相邻的元素，如果它们的顺序错误就把它们交换过来。

例如我们需要将 12 35 99 18 76 这 5 个数进行从大到小的排序。既然是从大到小排序，也就是说**越小的越靠后**，你是不是觉得我在说废话，但是这句话很关键(∩_∩)。

首先比较第 1 位和第 2 位的大小，现在第 1 位是 12，第 2 位是 35。发现 12 比 35 要小，

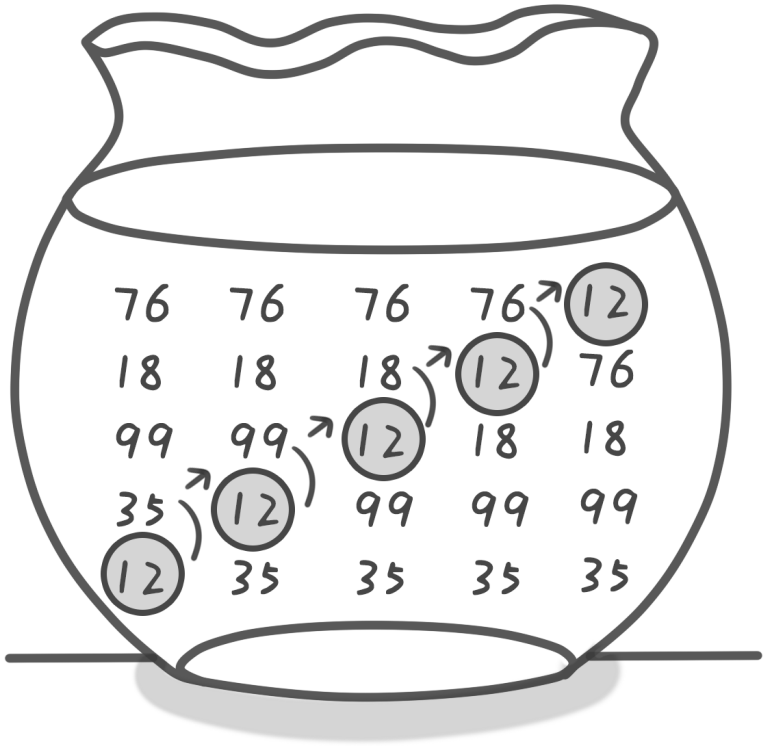
因为我们希望越小越靠后嘛，因此需要交换这两个数的位置。交换之后这 5 个数的顺序是 35 12 99 18 76。

按照刚才的方法，继续比较第 2 位和第 3 位的大小，第 2 位是 12，第 3 位是 99。12 比 99 要小，因此需要交换这两个数的位置。交换之后这 5 个数的顺序是 35 99 12 18 76。

根据刚才的规则，继续比较第 3 位和第 4 位的大小，如果第 3 位比第 4 位小，则交换位置。交换之后这 5 个数的顺序是 35 99 18 12 76。

最后，比较第 4 位和第 5 位。4 次比较之后 5 个数的顺序是 35 99 18 76 12。

经过 4 次比较后我们发现最小的一个数已经就位（已经在最后一位，请注意 12 这个数的移动过程），是不是很神奇。现在再来回忆一下刚才比较的过程。每次都是比较**相邻**的两个数，如果后面的数比前面的数大，则交换这两个数的位置。一直比较下去直到最后两个数比较完毕后，最小的数就在最后一个了。就如同是一个气泡，一步一步往后“翻滚”，直到最后一位。所以这个排序的方法有一个很好听的名字“冒泡排序”。



说到这里其实我们的排序只将 5 个数中最小的一个归位了。每将一个数归位我们将其称

为“一趟”。下面我们将继续重复刚才的过程，将剩下的 4 个数一一归位。

好，现在开始“第二趟”，目标是将第 2 小的数归位。首先还是先比较第 1 位和第 2 位，如果第 1 位比第 2 位小，则交换位置。交换之后这 5 个数的顺序是 99 35 18 76 12。接下来你应该都会了，依次比较第 2 位和第 3 位，第 3 位和第 4 位。注意此时已经不需要再比较第 4 位和第 5 位。因为在第一趟结束后已经可以确定第 5 位上放的是最小的了。第二趟结束之后这 5 个数的顺序是 99 35 76 18 12。

“第三趟”也是一样的。第三趟之后这 5 个数的顺序是 99 76 35 18 12。

现在到了最后一趟“第四趟”。有的同学又要问了，这不是已经排好了吗？还要继续？当然，这里纯属巧合，你若用别的数试一试可能就不是了。你能找出这样的数据样例来吗？请试一试。

“冒泡排序”的原理是：每一趟只能确定将一个数归位。即第一趟只能确定将末位上的数（即第 5 位）归位，第二趟只能将倒数第 2 位上的数（即第 4 位）归位，第三趟只能将倒数第 3 位上的数（即第 3 位）归位，而现在前面还有两个位置上的数没有归位，因此我们仍然需要进行“第四趟”。

“第四趟”只需要比较第 1 位和第 2 位的大小。因为后面三个位置上的数归位了，现在第 1 位是 99，第 2 位是 76，无需交换。这 5 个数的顺序不变仍然是 99 76 35 18 12。到此排序完美结束了，5 个数已经有 4 个数归位，那最后一个数也只能放在第 1 位了。

最后我们总结一下：如果有 n 个数进行排序，只需将 n–1 个数归位，也就是说要进行 n-1 趟操作。而“每一趟”都需要从第 1 位开始进行相邻两个数的比较，将较小的一个数放在后面，比较完毕后向后挪一位继续比较下面两个相邻数的大小，重复此步骤，直到最后一个**尚未归位的数**，已经归位的数则无需再进行比较（已经归位的数你还比较个啥，浪费表情）。

这个算法是不是很强悍？记得我每次拍集体照的时候就总是被别人换来换去的，当时特别烦。不知道发明此算法的人当时的灵感是否来源于此。啰里吧嗦地说了这么多，下面是代码。建议先自己尝试去实现一下看看，再来看我是如何实现的。

```
#include <stdio.h>
int main()
{
    int a[100],i,j,t,n;
    scanf("%d",&n);   //输入一个数n，表示接下来有n个数
    for(i=1;i<=n;i++)   //循环读入n个数到数组a中
        scanf("%d",&a[i]);
```

```
        //冒泡排序的核心部分
        for(i=1;i<=n-1;i++) //n个数排序，只用进行n-1趟
        {
            for(j=1;j<=n-i;j++) //从第1位开始比较直到最后一个尚未归位的数，想一想为什么到n-i就可以了。
            {
                if(a[j]<a[j+1]) //比较大小并交换
                {  t=a[j]; a[j]=a[j+1]; a[j+1]=t;  }
            }
        }
        for(i=1;i<=n;i++)  //输出结果
            printf("%d ",a[i]);

        getchar();getchar();
        return 0;
    }
```

可以输入以下数据进行验证。

```
10
8 100 50 22 15 6 1 1000 999 0
```

运行结果是：

```
0 1 6 8 15 22 50 100 999 1000
```

将上面代码稍加修改，就可以解决第 1 节遗留的问题，如下。

```
#include <stdio.h>
struct student
{
    char name[21];
    char score;
};//这里创建了一个结构体用来存储姓名和分数
int main()
{
    struct student a[100],t;
    int i,j,n;
    scanf("%d",&n); //输入一个数n
    for(i=1;i<=n;i++) //循环读入n个人名和分数
        scanf("%s %d",a[i].name,&a[i].score);
```

```
    //按分数从高到低进行排序
    for(i=1;i<=n-1;i++)
    {
        for(j=1;j<=n-i;j++)
        {
            if(a[j].score<a[j+1].score)//对分数进行比较
            {  t=a[j]; a[j]=a[j+1]; a[j+1]=t;  }
        }
    }
    for(i=1;i<=n;i++)//输出人名
        printf("%s\n",a[i].name);

    getchar();getchar();
    return 0;
}
```

可以输入以下数据进行验证。

```
5
huhu 5
haha 3
xixi 5
hengheng 2
gaoshou 8
```

运行结果是：

```
gaoshou
huhu
xixi
haha
hengheng
```

冒泡排序的核心部分是双重嵌套循环。不难看出冒泡排序的时间复杂度是 $O(N^2)$。这是一个非常高的时间复杂度。冒泡排序早在 1956 年就有人开始研究，之后有很多人都尝试过对冒泡排序进行改进，但结果却令人失望。如 Donald E. Knuth（中文名为高德纳，1974 年图灵奖获得者）所说：“冒泡排序除了它迷人的名字和导致了某些有趣的理论问题这一事实之外，似乎没有什么值得推荐的。”你可能要问：那还有没有更好的排序算法呢？不要走开，请看下节——快速排序。

第 3 节　最常用的排序——快速排序

上一节的冒泡排序可以说是我们学习的第一个真正的排序算法，并且解决了桶排序浪费空间的问题，但在算法的执行效率上却牺牲了很多，它的时间复杂度达到了 $O(N^2)$。假如我们的计算机每秒钟可以运行 10 亿次，那么对 1 亿个数进行排序，桶排序只需要 0.1 秒，而冒泡排序则需要 1 千万秒，达到 115 天之久，是不是很吓人？那有没有既不浪费空间又可以快一点的排序算法呢？那就是“快速排序”啦！光听这个名字是不是就觉得很高端呢？

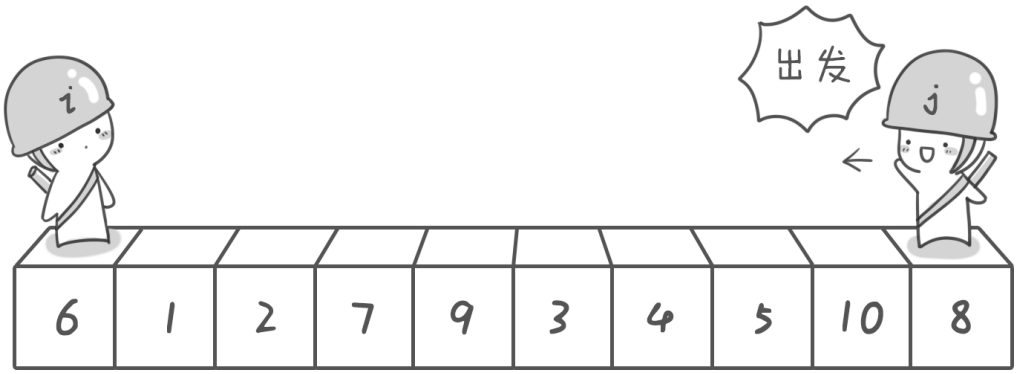
假设我们现在对“6 1 2 7 9 3 4 5 10 8”这 10 个数进行排序。首先在这个序列中随便找一个数作为**基准数**（不要被这个名词吓到了，这就是一个用来参照的数，待会儿你就知道它用来做啥了）。为了方便，就让第一个数 6 作为基准数吧。接下来，需要将这个序列中所有比基准数大的数放在 6 的右边，比基准数小的数放在 6 的左边，类似下面这种排列。

```
3  1  2  5  4  6  9  7  10  8
```

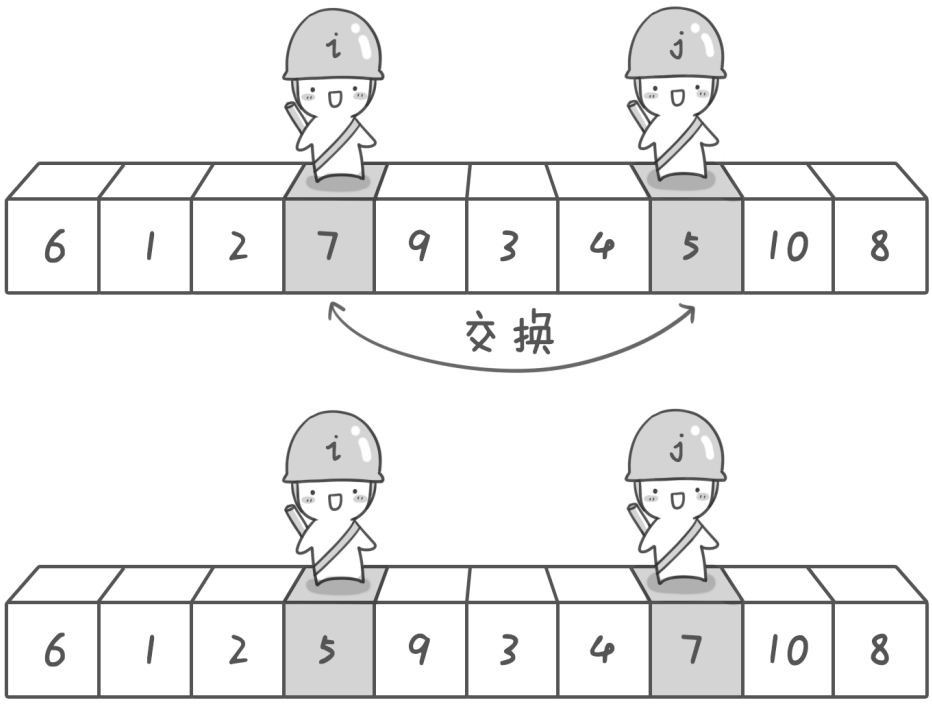
在初始状态下，数字 6 在序列的第 1 位。我们的目标是将 6 挪到序列中间的某个位置，假设这个位置是 k。现在就需要寻找这个 k，并且以第 k 位为分界点，左边的数都小于等于 6，右边的数都大于等于 6。想一想，你有办法可以做到这点吗？

给你一个提示吧。请回忆一下冒泡排序是如何通过“交换”一步步让每个数归位的。此时你也可以通过“交换”的方法来达到目的。具体是如何一步步交换呢？怎样交换才既方便又节省时间呢？先别急着往下看，拿出笔来，在纸上画画看。我高中时第一次学习冒泡排序算法的时候，就觉得冒泡排序很浪费时间，每次都只能对相邻的两个数进行比较，这显然太不合理了。于是我就想了一个办法，后来才知道原来这就是“快速排序”，请允许我小小地自恋一下(^o^)。

方法其实很简单：分别从初始序列“6 1 2 7 9 3 4 5 10 8”两端开始“探测”。先从**右**往**左**找一个小于 6 的数，再从**左**往**右**找一个大于 6 的数，然后交换它们。这里可以用两个变量 i 和 j，分别指向序列最左边和最右边。我们为这两个变量起个好听的名字“哨兵 i”和“哨兵 j”。刚开始的时候让哨兵 i 指向序列的最左边（即 i=1），指向数字 6。让哨兵 j 指向序列的最右边（即 j=10），指向数字 8。



首先哨兵 *j* 开始出动。因为此处设置的基准数是最左边的数，所以需要让哨兵 *j* 先出动，这一点非常重要（请自己想一想为什么）。哨兵 *j* 一步一步地向左挪动（即 *j*––），直到找到一个小于 6 的数停下来。接下来哨兵 *i* 再一步一步向右挪动（即 *i*++），直到找到一个大于 6 的数停下来。最后哨兵 *j* 停在了数字 5 面前，哨兵 *i* 停在了数字 7 面前。

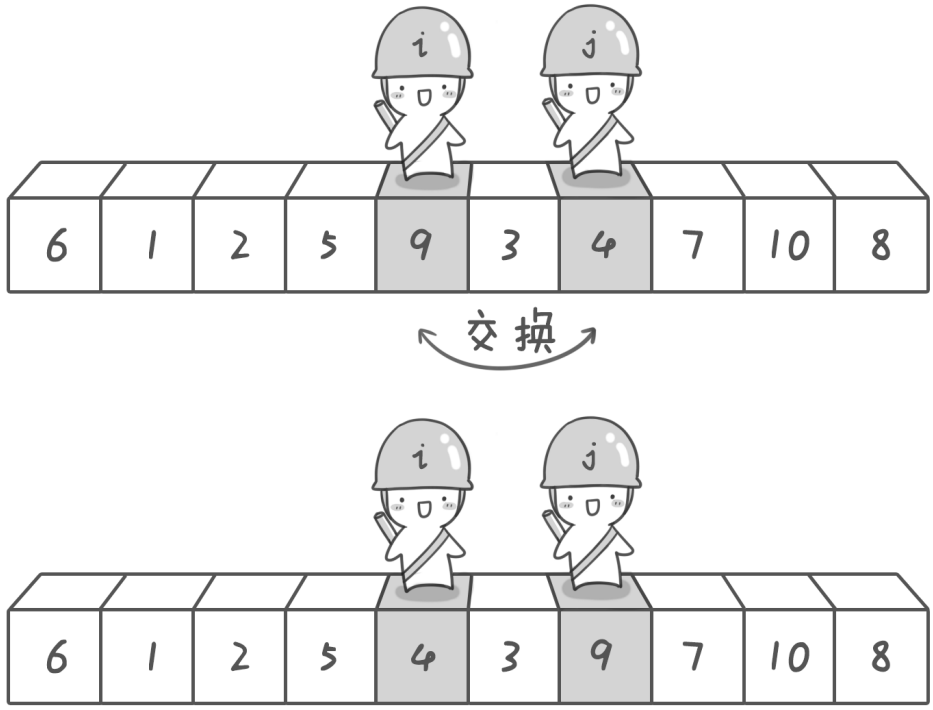


现在交换哨兵 *i* 和哨兵 *j* 所指向的元素的值。交换之后的序列如下。

```
6  1  2  5  9  3  4  7  10  8
```

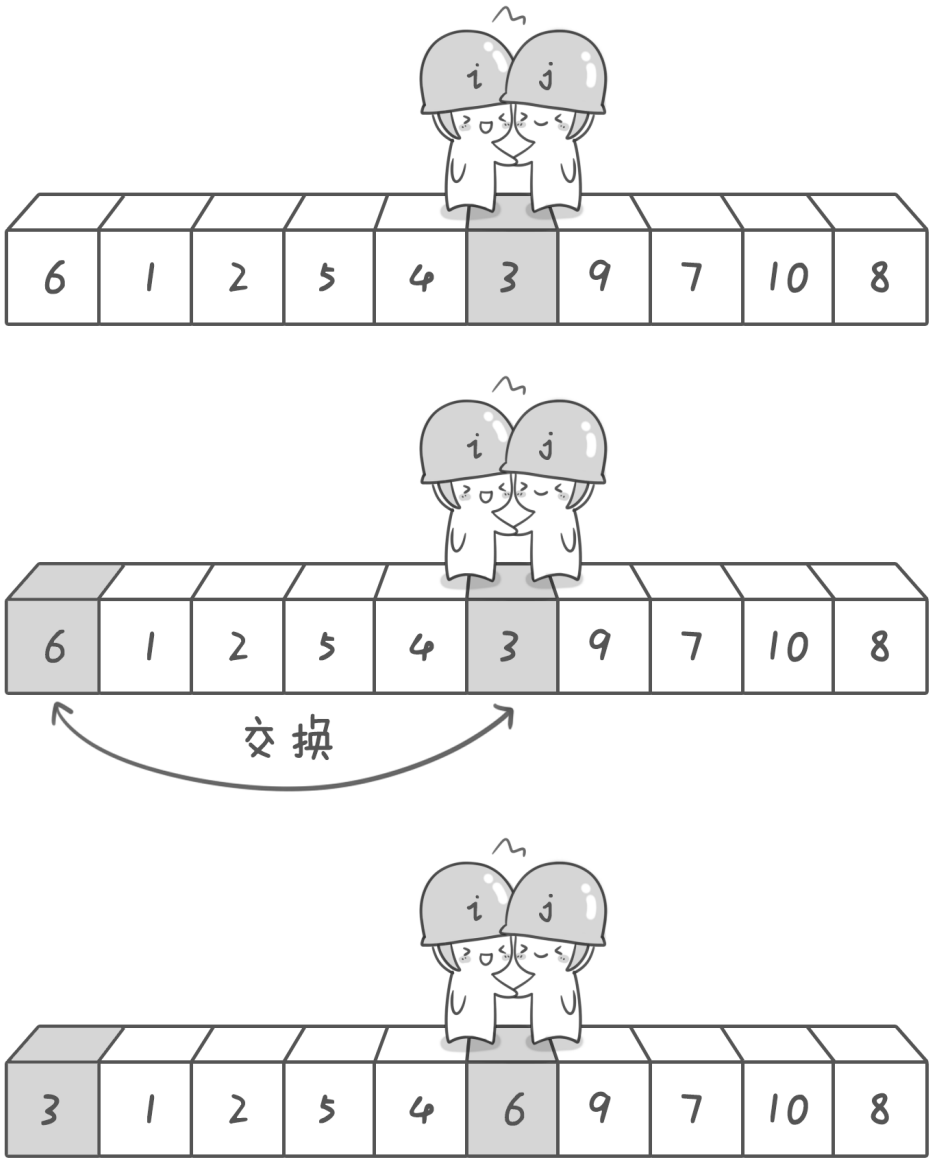
到此，第一次交换结束。接下来哨兵 *j* 继续向左挪动（再次友情提醒，每次必须是哨兵 *j* 先出发）。他发现了 4（比基准数 6 要小，满足要求）之后停了下来。哨兵 *i* 也继续向右挪动，他发现了 9（比基准数 6 要大，满足要求）之后停了下来。此时再次进行交换，交换之后的序列如下。

6 1 2 5 **4** 3 **9** 7 10 8



第二次交换结束，“探测”继续。哨兵 *j* 继续向左挪动，他发现了 3（比基准数 6 要小，满足要求）之后又停了下来。哨兵 *i* 继续向右移动，糟啦！此时哨兵 *i* 和哨兵 *j* 相遇了，哨兵 *i* 和哨兵 *j* 都走到 3 面前。说明此时“探测”结束。我们将基准数 6 和 3 进行交换。交换之后的序列如下。

3 1 2 5 4 **6** 9 7 10 8



到此第一轮“探测”真正结束。此时以基准数 6 为分界点，6 左边的数都小于等于 6，6 右边的数都大于等于 6。回顾一下刚才的过程，其实哨兵 *j* 的使命就是要找小于基准数的数，而哨兵 *i* 的使命就是要找大于基准数的数，直到 *i* 和 *j* 碰头为止。

OK，解释完毕。现在基准数 6 已经归位，它正好处在序列的第 6 位。此时我们已经将原来的序列，以 6 为分界点拆分成了两个序列，左边的序列是“3 1 2 5 4”，右边的序列是

“9 7 10 8”。接下来还需要分别处理这两个序列，因为 6 左边和右边的序列目前都还是很混乱的。不过不要紧，我们已经掌握了方法，接下来只要模拟刚才的方法分别处理 6 左边和右边的序列即可。现在先来处理 6 左边的序列吧。

左边的序列是“3 1 2 5 4”。请将这个序列以 3 为基准数进行调整，使得 3 左边的数都小于等于 3，3 右边的数都大于等于 3。好了开始动笔吧。

不用找纸了，别以为我不知道你的小伎俩，你肯定又没有动手尝试！就准备继续往下看了吧。这里我留了一个空白区域，赶快自己动手模拟一下吧！

如果你模拟得没有错，调整完毕之后的序列的顺序应该是：

```
2  1  3  5  4
```

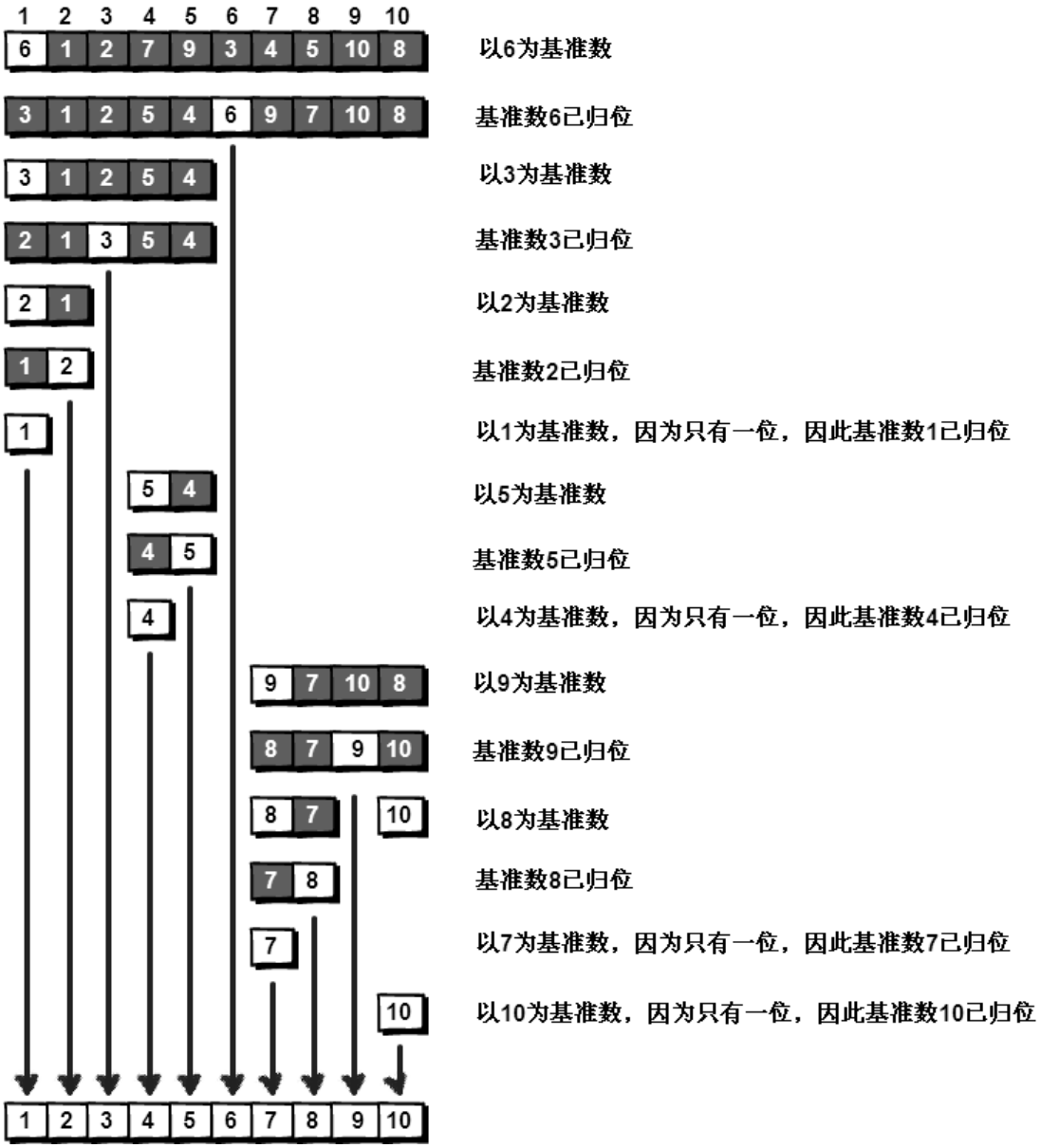
OK，现在 3 已经归位。接下来需要处理 3 左边的序列“2 1”和右边的序列“5 4”。对序列“2 1”以 2 为基准数进行调整，处理完毕之后的序列为“1 2”，到此 2 已经归位。序列“1”只有一个数，也不需要进行任何处理。至此我们对序列“2 1”已全部处理完毕，得到的序列是“1 2”。序列“5 4”的处理也仿照此方法，最后得到的序列如下。

```
1  2  3  4  5  6  9  7  10  8
```

对于序列“9 7 10 8”也模拟刚才的过程，直到不可拆分出新的子序列为止。最终将会得到这样的序列：

```
1  2  3  4  5  6  7  8  9  10
```

到此，排序完全结束。细心的同学可能已经发现，快速排序的每一轮处理其实就是将这一轮的基准数归位，直到所有的数都归位为止，排序就结束了。下面上个霸气的图来描述下整个算法的处理过程。



快速排序之所以比较快，是因为相比冒泡排序，每次交换是跳跃式的。每次排序的时候设置一个基准点，将小于等于基准点的数全部放到基准点的左边，将大于等于基准点的数全部放到基准点的右边。这样在每次交换的时候就不会像冒泡排序一样只能在相邻的数之间进行交换，交换的距离就大得多了。因此总的比较和交换次数就少了，速度自然就提高了。当然在最坏的情况下，仍可能是相邻的两个数进行了交换。因此快速排序的最差时间复杂度和冒泡排序是一样的，都是 $O(N^2)$，它的平均时间复杂度为 $O(N\log N)$。其实快速排序是基于一种叫做“二分”的思想。我们后面还会遇到“二分”思想，到时候再聊。先上代码，如下。

```
#include <stdio.h>
int a[101],n;//定义全局变量，这两个变量需要在子函数中使用

void quicksort(int left,int right)
{
    int i,j,t,temp;
    if(left>right)
       return;

    temp=a[left]; //temp中存的就是基准数
    i=left;
    j=right;
    while(i!=j)
    {
        //顺序很重要，要先从右往左找
        while(a[j]>=temp && i<j)
            j--;
        //再从左往右找
        while(a[i]<=temp && i<j)
            i++;

        //交换两个数在数组中的位置
        if(i<j)//当哨兵i和哨兵j没有相遇时
        {
            t=a[i];
            a[i]=a[j];
            a[j]=t;
        }
    }
    //最终将基准数归位
```

```
    a[left]=a[i];
    a[i]=temp;

    quicksort(left,i-1);//继续处理左边的，这里是一个递归的过程
    quicksort(i+1,right);//继续处理右边的，这里是一个递归的过程
}

int main()
{
    int i,j,t;
    //读入数据
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);

    quicksort(1,n); //快速排序调用

    //输出排序后的结果
    for(i=1;i<=n;i++)
        printf("%d ",a[i]);

    getchar();getchar();
    return 0;
}
```

可以输入以下数据进行验证。

```
10
6  1  2  7  9  3  4  5  10  8
```

运行结果是：

```
1 2 3 4 5 6 7 8 9 10
```

下面是程序执行过程中数组 a 的变化过程，带下划线的数表示的是已归位的基准数。

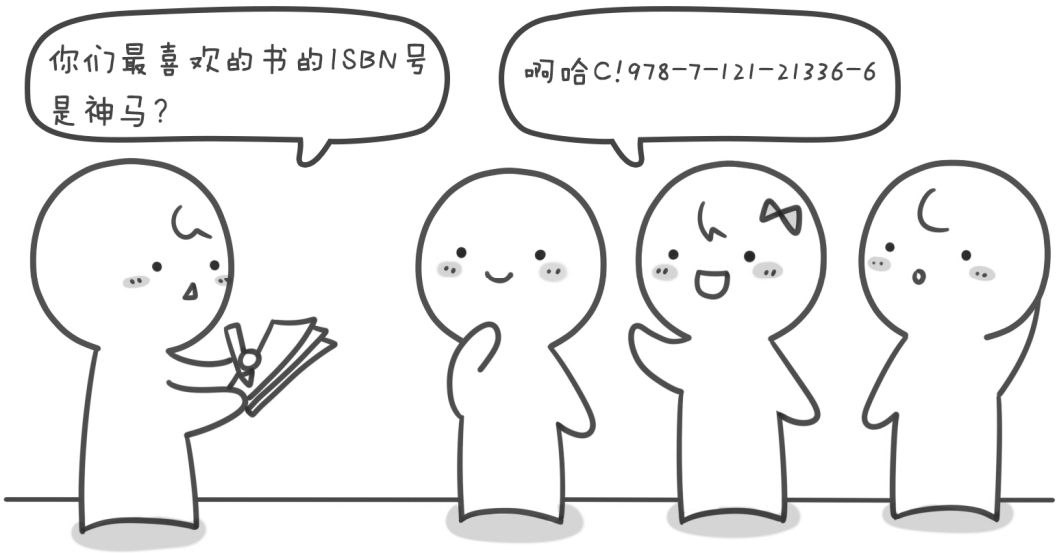
```
6 1 2 7 9 3 4 5 10 8
3 1 2 5 4 6 9 7 10 8
2 1 3 5 4 6 9 7 10 8
1 2 3 5 4 6 9 7 10 8
1 2 3 5 4 6 9 7 10 8
```

```
1 2 3 4 5 6 9 7 10 8
1 2 3 4 5 6 9 7 10 8
1 2 3 4 5 6 8 7 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

快速排序由 C. A. R. Hoare（东尼·霍尔，Charles Antony Richard Hoare）在 1960 年提出，之后又有许多人做了进一步的优化。如果你对快速排序感兴趣，可以去看看东尼·霍尔 1962 年在 *Computer Journal* 发表的论文“Quicksort”以及《算法导论》的第七章。快速排序算法仅仅是东尼·霍尔在计算机领域才能的第一次显露，后来他受到了老板的赏识和重用，公司希望他为新机器设计一种新的高级语言。你要知道当时还没有 PASCAL 或者 C 语言这些高级的东东。后来东尼·霍尔参加了由 Edsger Wybe Dijkstra（1972 年图灵奖得主，这个大神我们后面还会遇到的，到时候再细聊）举办的 ALGOL 60 培训班，他觉得自己与其没有把握地去设计一种新的语言，还不如对现有的 ALGOL 60 进行改进，使之能在公司的新机器上使用。于是他便设计了 ALGOL 60 的一个子集版本。这个版本在执行效率和可靠性上都在当时 ALGOL 60 的各种版本中首屈一指，因此东尼·霍尔受到了国际学术界的重视。后来他在 ALGOL X 的设计中还发明了大家熟知的 case 语句，也被各种高级语言广泛采用，比如 PASCAL、C、Java 语言等等。当然，东尼·霍尔在计算机领域的贡献还有很多很多，他在 1980 年获得了图灵奖。

第 4 节　小哼买书

排序算法还有很多，例如我在《啊哈 C！思考快你一步》一书中讲过的选择排序，另外还有计数排序、基数排序、插入排序、归并排序和堆排序等等。堆排序是基于二叉树的排序，我会在后面的章节讲到。现在来看一个具体的例子“小哼买书”（根据全国青少年信息学奥林匹克联赛 NOIP2006 普及组第一题改编），来实践一下本章所学的三种排序算法。



小哼的学校要建立一个图书角，老师派小哼去找一些同学做调查，看看同学们都喜欢读哪些书。小哼让每个同学写出一个自己最想读的书的 ISBN 号（你知道吗？每本书都有唯一的 ISBN 号，不信的话你去找本书翻到背面看看）。当然有一些好书会有很多同学都喜欢，这样就会收集到很多重复的 ISBN 号。小哼需要去掉其中重复的 ISBN 号，即每个 ISBN 号只保留一个，也就说同样的书只买一本（学校真是够抠门的）。然后再把这些 ISBN 号从小到大排序，小哼将按照排序好的 ISBN 号去书店买书。请你协助小哼完成“去重”与“排序”的工作。

输入有 2 行，第 1 行为一个正整数，表示有 n 个同学参与调查（$n \leqslant 100$）。第 2 行有 n 个用空格隔开的正整数，为每本图书的 ISBN 号（假设图书的 ISBN 号在 1~1000 之间）。

输出也是 2 行，第 1 行为一个正整数 k，表示需要买多少本书。第 2 行为 k 个用空格隔开的正整数，为从小到大已排好序的需要购买的图书的 ISBN 号。

例如输入：

```
10
20 40 32 67 40 20 89 300 400 15
```

则输出：

```
8
15 20 32 40 67 89 300 400
```

最后，程序运行的时间限制为 1 秒。

解决这个问题的方法大致有两种。第一种方法：先将这 *n* 个图书的 ISBN 号去重，再进行从小到大排序并输出；第二种方法：先从小到大排序，输出的时候再去重。这两种方法都可以。

先来看第一种方法。通过第一节的学习我们发现，桶排序稍加改动正好可以起到去重的效果，因此我们可以使用桶排序的方法来解决此问题。

```
#include <stdio.h>
int main()
{
    int a[1001],n,i,t;
    for(i=1;i<=1000;i++)
        a[i]=0; //初始化

    scanf("%d",&n); //读入n
    for(i=1;i<=n;i++) //循环读入n个图书的ISBN号
    {
        scanf("%d",&t); //把每一个ISBN号读到变量t中
        a[t]=1; //标记出现过的ISBN号
    }

    for(i=1;i<=1000;i++) //依次判断1~1000这个1000个桶
    {
        if(a[i]==1)//如果这个ISBN号出现过则打印出来
             printf("%d ",i);
    }

    getchar();getchar();
    return 0;
}
```

这种方法的时间复杂度就是桶排序的时间复杂度，为 $O(N+M)$。

第二种方法我们需要先排序再去重。排序我们可以用冒泡排序或者快速排序。

```
20 40 32 67 40 20 89 300 400 15
```

将这 10 个数从小到大排序之后为 15 20 20 32 40 40 67 89 300 400。

接下来，要在输出的时候去掉重复的。因为我们已经排好序，所以相同的数都会紧挨在

一起。只要在输出的时候，预先判断一下当前这个数 $a[i]$与前面一个数 $a[i-1]$是否相同。如果相同则表示这个数之前已经输出过了，不用再次输出；不同则表示这个数是第一次出现，需要输出这个数。

```
#include <stdio.h>
int main()
{
    int a[101],n,i,j,t;

    scanf("%d",&n);   //读入n
    for(i=1;i<=n;i++) //循环读入n个图书ISBN号
    {
        scanf("%d",&a[i]);
    }

   //开始冒泡排序
    for(i=1;i<=n-1;i++)
    {
     for(j=1;j<=n-i;j++)
        {
            if(a[j]>a[j+1])
            {  t=a[j]; a[j]=a[j+1]; a[j+1]=t;  }
        }
    }
    printf("%d ",a[1]); //输出第1个数
    for(i=2;i<=n;i++) //从2循环到n
    {
        if( a[i] != a[i-1] ) //如果当前这个数是第一次出现则输出
             printf("%d ",a[i]);
    }

    getchar();getchar();
    return 0;
}
```

这种方法的时间复杂度由两部分组成，一部分是冒泡排序的时间复杂度，是 $N(N^2)$，另一部分是读入和输出，都是 $O(N)$，因此整个算法的时间复杂度是 $O(2*N+N^2)$。相对于 N^2 来说，$2*N$ 可以忽略（我们通常忽略低阶），最终该方法的时间复杂度是 $O(N^2)$。

接下来我们还需要看下数据范围。每个图书 ISBN 号都是 1~1000 之间的整数，并且参加调查的同学人数不超过 100，即 $n \leqslant 100$。之前已经说过，在粗略计算时间复杂度的时候，我们通常认为计算机每秒钟大约运行 10 亿次（当然实际情况要更快）。因此以上两种方法都可以在 1 秒钟内计算出解。如果题目中图书的 ISBN 号范围不是在 1~1000 之间，而是 –2147483648~2147483647 之间的话，那么第一种方法就不可行了，因为你无法申请出这么大的数组来标记每一个 ISBN 号是否出现过。另外如果 n 的范围不是小于等于 100，而是小于等于 10 万，那么第二种方法的排序部分也不能使用冒泡排序。因为题目要求的时间限制是 1 秒，使用冒泡排序对 10 万个数进行排序，计算机要运行 100 亿次，需要 10 秒钟，因此要替换为快速排序，快速排序只需要 $100000 \times \log_2 100000 \approx 100000 \times 17 \approx 170$ 万次，这还不到 0.0017 秒。是不是很神奇？同样的问题使用不同的算法竟然有如此之大的时间差距，这就是算法的魅力！

我们来回顾一下本章三种排序算法的时间复杂度。桶排序是最快的，它的时间复杂度是 $O(N+M)$；冒泡排序是 $O(N^2)$；快速排序是 $O(N\log N)$。

最后，你可以到“添柴–编程学习网”提交本题的代码，来验证一下你的解答是否完全正确。《小哼买书》题目的地址如下：

```
www.tianchai.org/problem-12001.html
```

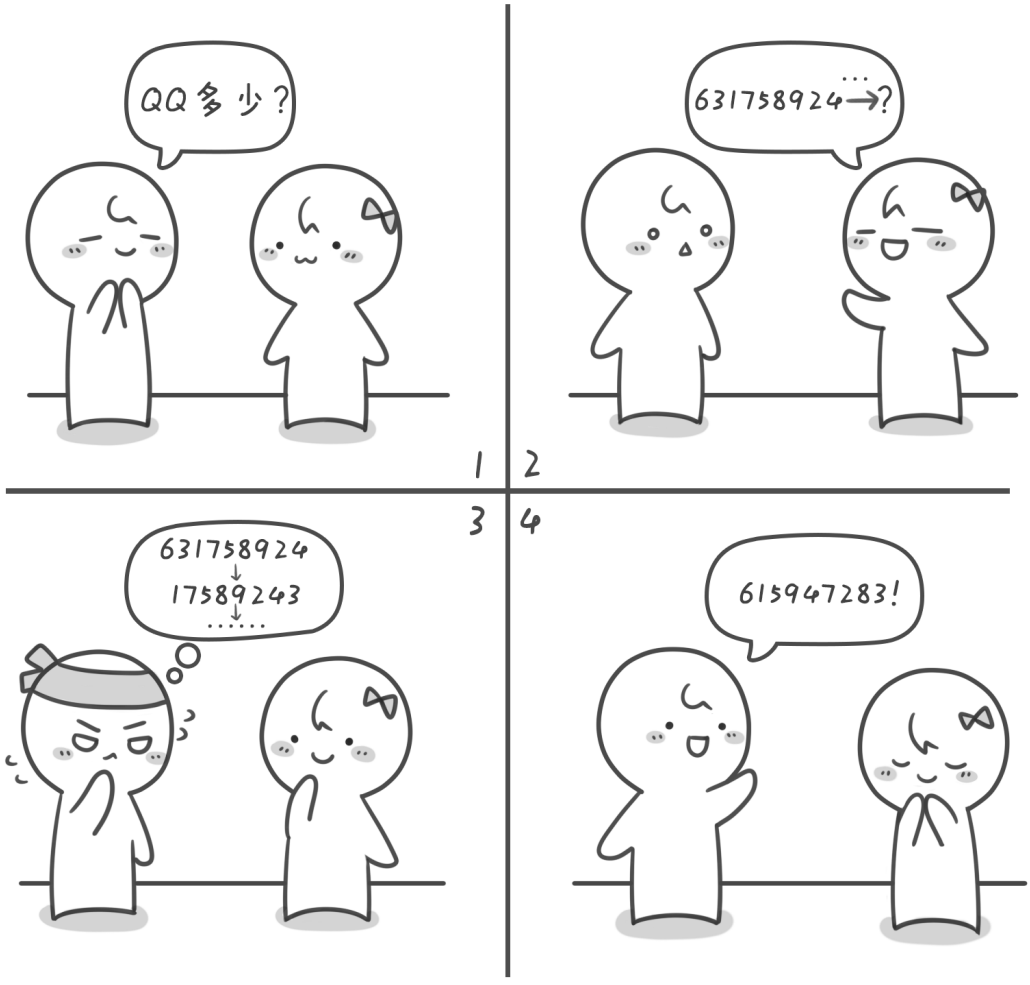
接下来，本书中的所有算法都可以去“添柴–编程学习网”一一验证。如果你从来没有使用过类似“添柴–编程学习网”这样的在线自动评测系统（online judge），那么我推荐你可以先尝试提交下这道题：A+B=? 地址如下：

```
www.tianchai.org/problem-10000.html
```

第2章 栈、队列、链表

第 1 节　解密 QQ 号——队列

新学期开始了，小哈是小哼的新同桌（小哈是个小美女哦~），小哼向小哈询问 QQ 号，小哈当然不会直接告诉小哼啦，原因嘛你懂的。所以小哈给了小哼一串加密过的数字，同时小哈也告诉了小哼解密规则。规则是这样的：首先将第 1 个数删除，紧接着将第 2 个数放到这串数的末尾，再将第 3 个数删除并将第 4 个数放到这串数的末尾，再将第 5 个数删除……直到剩下最后一个数，将最后一个数也删除。按照刚才删除的顺序，把这些删除的数连在一起就是小哈的 QQ 啦。现在你来帮帮小哼吧。小哈给小哼加密过的一串数是“6 3 1 7 5 8 9 2 4”。



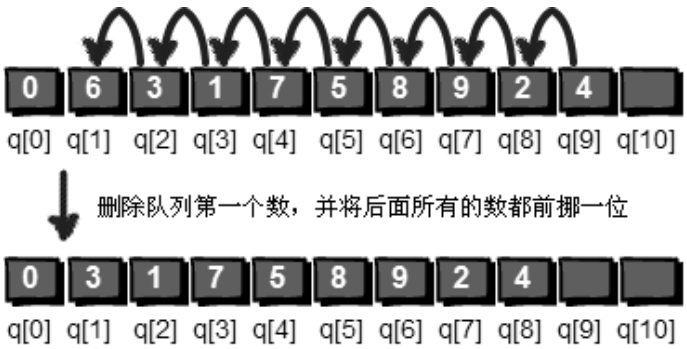
OK，现在轮到你动手的时候了。快去找出 9 张便签或小纸片，将“6 3 1 7 5 8 9 2 4”这 9 个数分别写在 9 张便签上，模拟一下解密过程。如果你没有理解错解密规则的话，解密后小哈的 QQ 号应该是“6 1 5 9 4 7 2 8 3”。

其实解密的过程就像是将这些数“排队”。每次从最前面拿两个，第 1 个扔掉，第 2 个放到尾部。具体过程是这样的：刚开始这串数是“6 3 1 7 5 8 9 2 4”，首先删除 6 并将 3 放到这串数的末尾，这串数更新为“1 7 5 8 9 2 4 3”。接下来删除 1 并将 7 放到末尾，即更新为“5 8 9 2 4 3 7”。再删除 5 并将 8 放到末尾即“9 2 4 3 7 8”，删除 9 并将 2 放到末尾即“4 3 7 8 2”，删除 4 并将 3 放到末尾即“7 8 2 3”，删除 7 并将 8 放到末尾即“2 3 8”，删除 2 并将 3 放到末尾即“8 3”，删除 8 并将 3 放到末尾即“3”，最后删除 3。因此被删除的顺序是“6 1 5 9 4 7 2 8 3”，这就是小哈的 QQ 号码了，你可以加她试试看^_^。

既然现在已经搞清楚了解密法则，不妨自己尝试一下去编程，我相信你一定可以写出来的。

首先需要一个数组来存储这一串数即 int q[101]，并初始化这个数组即 int q[101]={0,6,3,1,7,5,8,9,2,4};（此处初始化是我多写了一个 0，用来填充 q[0]，因为我比较喜欢从 q[1]开始用，对数组初始化不是很理解的同学可以去看一下我的上本书《啊哈 C！思考快你一步》）。接下来就是模拟解密的过程了。

解密的第一步是将第一个数删除，你可以想一下如何在数组中删除一个数呢。最简单的方法是将所有后面的数都往前面挪动一位，将前面的数覆盖。就好比我们在排队买票，最前面的人买好离开了，后面所有的人就需要全部向前面走一步，补上之前的空位，但是这样的做法很耗费时间。



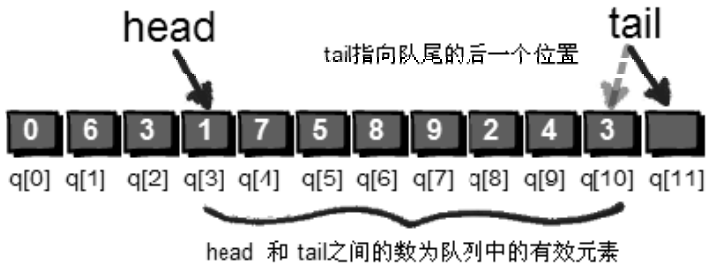
在这里，我将引入两个整型变量 head 和 tail。head 用来记录队列的队首（即第一位），tail 用来记录队列的队尾（即最后一位）的下一个位置。你可能会问：为什么 tail 不直接记录队尾，却要记录队尾的下一个位置呢？这是因为当队列中只剩下一个元素时，队首和队尾

重合会带来一些麻烦。我们这里规定队首和队尾重合时，队列为空。

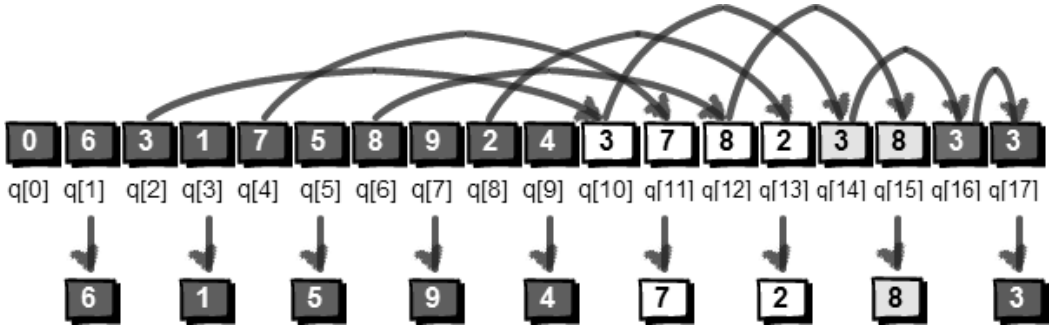
现在有 9 个数，9 个数全部放入队列之后 head=1;tail=10;此时 head 和 tail 之间的数就是目前队列中“有效”的数。如果要删除一个数的话，就将 head++就 OK 了，这样仍然可以保持 head 和 tail 之间的数为目前队列中“有效”的数。这样做虽然浪费了一个空间，却节省了大量的时间，这是非常划算的。新增加一个数也很简单，把需要增加的数放到队尾即 q[tail]之后再 tail++就 OK 啦。

我们来小结一下，在队首删除一个数的操作是 head++;。

在队尾增加一个数（假设这个数是 x）的操作是 q[tail]=x;tail++;。



整个解密过程，请看下面这个霸气外漏的图。



最后的输出就是 6 1 5 9 4 7 2 8 3，代码实现如下。

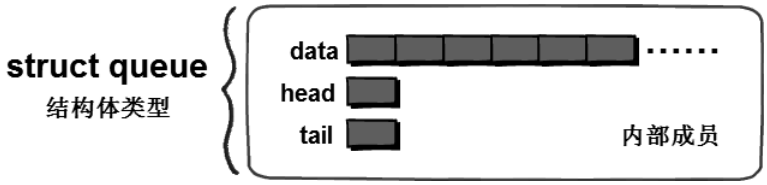
```
#include <stdio.h>
int main()
{
    int q[102]={0,6,3,1,7,5,8,9,2,4},head,tail;
    int i;
    //初始化队列
    head=1;
    tail=10; //队列中已经有9个元素了，tail指向队尾的后一个位置
    while(head<tail) //当队列不为空的时候执行循环
    {
        //打印队首并将队首出队
        printf("%d ",q[head]);
        head++;

        //先将新队首的数添加到队尾
        q[tail]=q[head];
        tail++;
        //再将队首出队
        head++;
    }

    getchar();getchar();
    return 0;
}
```

怎么样，上面的代码运行成功没有？现在我们再来总结一下队列的概念。队列是一种特殊的线性结构，它只允许在队列的首部（head）进行删除操作，这称为“出队”，而在队列的尾部（tail）进行插入操作，这称为“入队”。当队列中没有元素时（即 head==tail），称为空队列。在我们的日常生活中有很多情况都符合队列的特性。比如我们之前提到过的买票，每个排队买票的窗口就是一个队列。在这个队列当中，新来的人总是站在队列的最后面，来得越早的人越靠前，也就越早能买到票，就是先来的人先服务，我们称为“先进先出”（First In First Out，FIFO）原则。

队列将是我们今后学习广度优先搜索以及队列优化的 Bellman-Ford 最短路算法的核心数据结构。所以现在将队列的三个基本元素（一个数组，两个变量）封装为一个结构体类型，如下。

```
struct queue
{
    int data[100];//队列的主体，用来存储内容
    int head;//队首
    int tail;//队尾
};
```

上面定义了一个结构体类型，我们通常将其放在 main 函数的外面，请注意结构体的定义末尾有个;号。struct 是结构体的关键字，queue 是我们为这个结构体起的名字。这个结构体有三个成员分别是：整型数组 data、整型 head 和整型 tail。这样我们就可以把这三个部分放在一起作为一个整体来对待。你可以这么理解：我们定义了一个新的数据类型，这个新类型非常强大，用这个新类型定义出的每一个变量可以同时存储一个整型数组和两个整数。



有了新的结构体类型，如何定义结构体变量呢？很简单，这与我们之前定义变量的方式是一样的，具体做法如下。

```
struct queue q;
```

请注意 struct queue 需要整体使用，不能直接写 queue q;。这样我们就定义了一个结构体变量 q。这个结构体变量就可以满足队列的所有操作了。那又该如何访问结构体变量的内部成员呢？可以使用.号，它叫做成员运算符或者点号运算符，如下：

```
q.head=1;
q.tail=1;
scanf("%d",&q.data[q.tail]);
```

好了，下面这段代码就是使用结构体来实现的队列操作。

```
#include <stdio.h>
struct queue
{
    int data[100];//队列的主体，用来存储内容
    int head;//队首
```

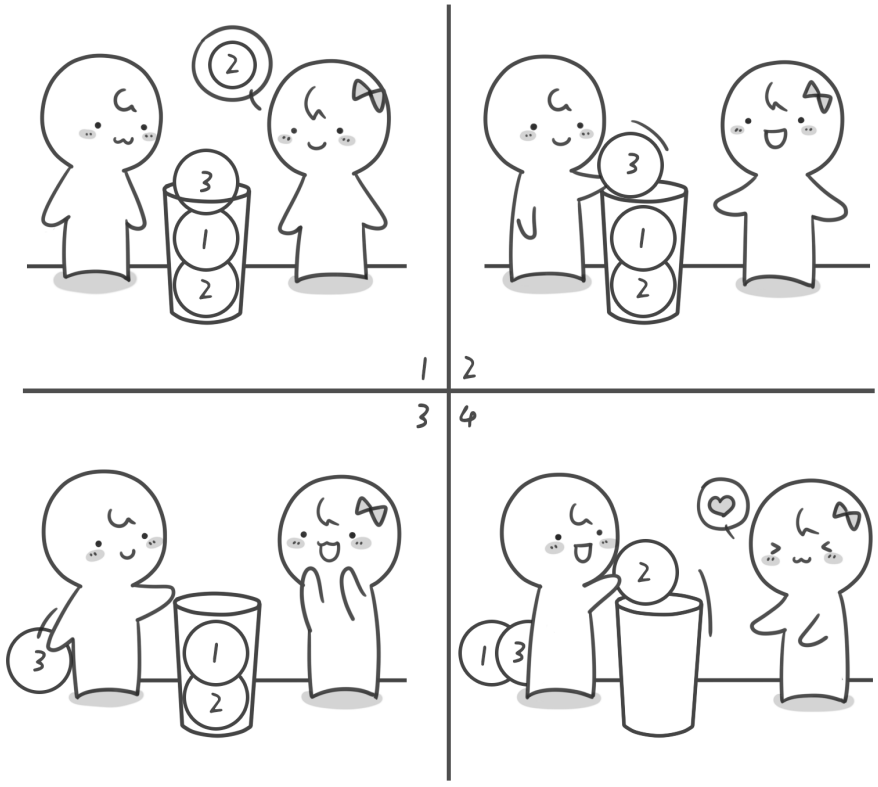
```
    int tail;//队尾
};

int main()
{
    struct queue q;
    int i;
    //初始化队列
    q.head=1;
    q.tail=1;
    for(i=1;i<=9;i++)
    {
        //依次向队列插入9个数
        scanf("%d",&q.data[q.tail]);
        q.tail++;
    }

    while(q.head<q.tail) //当队列不为空的时候执行循环
    {
        //打印队首并将队首出队
        printf("%d ",q.data[q.head]);
        q.head++;

        //先将新队首的数添加到队尾
        q.data[q.tail]=q.data[q.head];
        q.tail++;
        //再将队首出队
        q.head++;
    }

    getchar();getchar();
    return 0;
}
```

上面的这种写法看起来虽然冗余了一些，但是可以加强你对队列这个算法的理解。C++ 的 STL 库已经有队列的实现，有兴趣的同学可以参看相关材料。队列的起源已经无法追溯。在还没有数字计算机之前，数学应用中就已经有对队列的记载了。我们生活中队列的例子也比比皆是，比如排队买票，又或者吃饭时候用来排队等候的叫号机，又或者拨打银行客服选择人工服务时，每次都会被提示“客服人员正忙，请耐心等待”，因为客服人员太少了，拨

打电话的客户需要按照打进的时间顺序进行等候等等。这里表扬一下肯德基的宅急送，没有做广告的嫌疑啊，每次一打就通，基本不需要等待。但是我每次打银行的客服（具体是哪家银行就不点名了）基本都要等待很长时间，总是告诉我“正在转接，请稍候”，嘟嘟嘟三声后就变成“客服正忙，请耐心等待！”然后就给我放很难听的曲子。看来钱在谁那里谁就是老大啊……

第 2 节　解密回文——栈

上一节中我们学习了队列，它是一种先进先出的数据结构。还有一种是后进先出的数据结构，它叫做栈。栈限定为只能在一端进行插入和删除操作。比如说有一个小桶，小桶的直径只能放一个小球，我们现在小桶内依次放入 2、1、3 号小球。假如你现在需要拿出 2 号小球，那就必须先将 3 号小球拿出，再拿出 1 号小球，最后才能将 2 号小球拿出来。在刚才取小球的过程中，我们最先放进去的小球最后才能拿出来，最后放进去的小球却可以最先拿出来。



生活中也有很多这样的例子，比如我们在吃桶装薯片的时候，要想吃掉最后一片，就必须把前面的全部吃完（貌似现在的桶装薯片为了减少分量，在桶里面增加了一个透明的抽屉）；再比如浏览网页的时候需要退回到之前的某个网页，我们需要一步步地点击后退键。还有手枪的弹夹，在装子弹的时候，最后装入的那发子弹，是被第一个打出去的。栈的实现也很简单，只需要一个一维数组和一个指向栈顶的变量 top 就可以了。我们通过 top 来对栈进行插入和删除操作。

栈究竟有哪些作用呢？我们来看一个例子。“xyzyx”是一个回文字符串，所谓回文字符串就是指正读反读均相同的字符序列，如“席主席”、“记书记”、“aha”和“ahaha”均是回文，但“ahah”不是回文。通过栈这个数据结构我们将很容易判断一个字符串是否为回文。

首先我们需要读取这行字符串，并求出这个字符串的长度。

```
char a[101];
int len;
gets(a);
len=strlen(a);
```

如果一个字符串是回文的话，那么它必须是中间对称的，我们需要求中点，即：

```
mid=len/2-1;
```

接下来就轮到栈出场了。

我们先将 mid 之前的字符全部入栈。因为这里的栈是用来存储字符的，所以这里用来实现栈的数组类型是字符数组即 char s[101];，初始化栈很简单，top=0;就可以了。入栈的操作是 top++; s[top]=x;（假设需要入栈的字符暂存在字符变量 x 中），其实可以简写为 s[++top]=x;。

现在我们就来将 mid 之前的字符依次全部入栈。

```
for(i=0;i<=mid;i++)
{
    s[++top]=a[i];
}
```

接下来进入判断回文的关键步骤。将当前栈中的字符依次出栈，看看是否能与 mid 之后的字符一一匹配，如果都能匹配则说明这个字符串是回文字符串，否则这个字符串就不是回文字符串。

```
for(i=mid+1;i<=len-1;i++)
{
    if (a[i]!=s[top])
    {
          break;
    }
    top--;
}
if(top==0)
    printf("YES");
else
    printf("NO");
```

最后如果 top 的值为 0，就说明栈内所有的字符都被一一匹配了，那么这个字符串就是回文字符串。完整的代码如下。

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[101],s[101];
    int i,len,mid,next,top;

    gets(a); //读入一行字符串
    len=strlen(a); //求字符串的长度
    mid=len/2-1; //求字符串的中点

    top=0;//栈的初始化
    //将mid前的字符依次入栈
    for(i=0;i<=mid;i++)
        s[++top]=a[i];

    //判断字符串的长度是奇数还是偶数，并找出需要进行字符匹配的起始下标
    if(len%2==0)
        next=mid+1;
    else
        next=mid+2;

    //开始匹配
    for(i=next;i<=len-1;i++)
```

```
        {
            if(a[i]!=s[top])
                break;
            top--;
        }

        //如果top的值为0，则说明栈内所有的字符都被一一匹配了
        if(top==0)
            printf("YES");
        else
            printf("NO");

        getchar();getchar();
        return 0;
    }
```

可以输入以下数据进行验证。

```
ahaha
```

运行结果是：

```
YES
```

栈还可以用来进行验证括号的匹配。比如输入一行只包含“()[]{}”的字符串，请判断形如“([{}()])”或者“{()[]{}}”的是否可以正确匹配。显然上面两个例子都是可以正确匹配的。“([)]”是不能匹配的。有兴趣的同学可以自己动手来试一试。

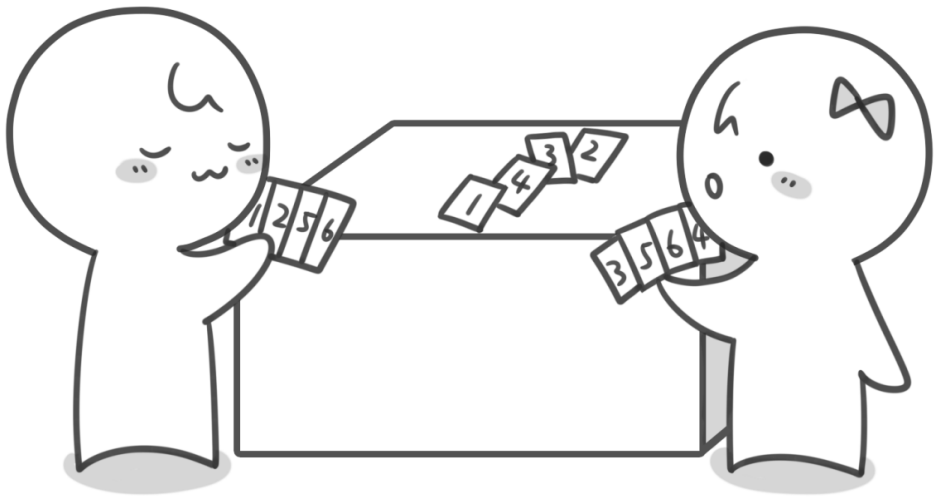
堆栈最早由 Alan M. Turing（艾伦 • 图灵）于 1946 年提出，当时是为了解决子程序的调用和返回。艾伦 • 图灵这个大帅哥可是个大牛人，图灵奖就是以他的名字命名的。如果你对他感兴趣不妨去读一读《艾伦 • 图灵传：如谜的解谜者》和《图灵的秘密》。

第 3 节　纸牌游戏——小猫钓鱼

星期天小哼和小哈约在一起玩桌游，他们正在玩一个非常古怪的扑克游戏——“小猫钓鱼”。游戏的规则是这样的：将一副扑克牌平均分成两份，每人拿一份。小哼先拿出手中的第一张扑克牌放在桌上，然后小哈也拿出手中的第一张扑克牌，并放在小哼刚打出的扑克牌的上面，就像这样两人交替出牌。出牌时，如果某人打出的牌与桌上某张牌的牌面相同，即

可将两张相同的牌及其中间所夹的牌全部取走，并依次放到自己手中牌的末尾。当任意一人手中的牌全部出完时，游戏结束，对手获胜。

假如游戏开始时，小哼手中有 6 张牌，顺序为 2 4 1 2 5 6，小哈手中也有 6 张牌，顺序为 3 1 3 5 6 4，最终谁会获胜呢？现在你可以拿出纸牌来试一试。接下来请你写一个程序来自动判断谁将获胜。这里我们做一个约定，小哼和小哈手中牌的牌面只有 1~9。



我们先来分析一下这个游戏有哪几种操作。小哼有两种操作，分别是出牌和赢牌。这恰好对应队列的两个操作，出牌就是出队，赢牌就是入队。小哈的操作和小哼是一样的。而桌子就是一个栈，每打出一张牌放到桌上就相当于入栈。当有人赢牌的时候，依次将牌从桌上拿走，这就相当于出栈。那如何解决赢牌的问题呢？赢牌的规则是：如果某人打出的牌与桌上的某张牌相同，即可将两张牌以及中间所夹的牌全部取走。那如何知道桌上已经有哪些牌了呢？最简单的方法就是枚举桌上的每一张牌，当然也有更好的办法我们待会再说。OK，小结一下，我们需要两个队列、一个栈来模拟整个游戏。

首先我们先来创建一个结构体用来实现队列，如下。

```
struct queue
{
    int data[1000];
    int head;
    int tail;
}
```

上面代码中 head 用来存储队头，tail 用来存储队尾。数组 data 用来存储队列中的元素，数组 data 的大小我预设为 1000，其实应该设置得更大一些，以防数组越界。当然对于本题的数据来说 1000 已经足够了。

再创建一个结构体用来实现栈，如下。

```
struct stack
{
    int data[10];
    int top;
};
```

其中 top 用来存储栈顶，数组 data 用来存储栈中的元素，大小设置为 10。因为只有 9 种不同的牌面，所以桌上最多可能有 9 张牌，因此数组大小设置为 10 就够了。提示一下：为什么不设置为 9 呢？因为 C 语言数组下标是从 0 开始的。

接下来我们需要定义两个队列变量 q1 和 q2。q1 用来模拟小哼手中的牌，q2 用来模拟小哈手中的牌。定义一个栈变量 s 用来模拟桌上的牌。

```
struct queue  q1,q2;
struct stack  s;
```

接下来来初始化一下队列和栈。

```
//初始化队列q1和q2为空，此时两人手中都还没有牌
q1.head=1; q1.tail=1;
q2.head=1; q2.tail=1;
//初始化栈s为空，最开始的时候桌上也没有牌
s.top=0;
```

接下来需要读入小哼和小哈最初时手中的牌，分两次读入，每次读入 6 个数，分别插入 q1 和 q2 中。

```
//先读入6张牌，放到小哼手上
for(i=1;i<=6;i++)
{
        scanf("%d",&q1.data[q1.tail]); //读入一个数到队尾
        q1.tail++;//队尾往后挪一位
    }
//再读入6张牌，放到小哈手上
```

```
    for(i=1;i<=6;i++)
    {
        scanf("%d",&q2.data[q2.tail]); //读入一个数到队尾
            q2.tail++;//队尾往后挪一位
    }
```

现在准备工作已经基本上做好了，游戏正式开始，小哼先出牌。

```
    t=q1.data[q1.head]; //小哼先亮出一张牌
```

小哼打出第一张牌，也就是 q1 的队首，我们将这张牌存放在临时变量 t 中。接下来我们要判断小哼当前打出的牌是否能赢得桌上的牌。也就是判断桌上的牌与 t 有没有相同的，如何实现呢？我们需要枚举桌上的每一张牌与 t 进行比对，具体如下：

```
    flag=0;
    for(i=1;i<=top;i++)
    {
            if(t==s[i]) { flag=1; break; }
    }
```

如果 flag 的值为 0 就表明小哼没能赢得桌上的牌，将打出的牌留在桌上。

```
    if(flag==0)
    {
            //小哼此轮没有赢牌
        q1.head++; //小哼已经打出一张牌，所以要把打出的牌出队
        s.top++;
        s.data[s.top]=t; //再把打出的牌放到桌上，即入栈
    }
```

如果 flag 的值为 1 就表明小哼可以赢得桌上的牌，需要将赢得的牌依次放入小哼的手中。

```
    f(flag==1)
    {
            //小哼此轮可以赢牌
        q1.head++;//小哼已经打出一张牌，所以要把打出的牌出队
        q1.data[q1.tail]=t; //因为此轮可以赢牌，所以紧接着把刚才打出的牌又放到手中牌的
末尾
            q1.tail++;
        while(s.data[s.top]!=t) //把桌上可以赢得的牌（从当前桌面最顶部一张牌开始取，直
至取到与打出的牌相同为止）依次放到手中牌的末尾
```

```
        {
                q1.data[q1.tail]=s.data[s.top]; //依次放入队尾
                q1.tail++;
                s.top--; //栈中少了一张牌，所以栈顶要减1
        }
    }
```

小哼出牌的所有阶段就模拟完了，小哈出牌和小哼出牌是一样的。接下来我们要判断游戏如何结束。即只要两人中有一个人的牌用完了游戏就结束了。因此需要在模拟两人出牌代码的外面加一个 while 循环来判断，如下。

```
    while(q1.head<q1.tail && q2.head<q2.tail ) //当队列q1和q2都不为空的时候执行循环
```

最后一步，输出谁最终赢得了游戏，以及游戏结束后获胜者手中的牌和桌上的牌。如果小哼获胜了那么小哈的手中一定没有牌了（队列 q2 为空），即 q2.head==q2.tail，具体输出如下。

```
    if(q2.head==q2.tail)
    {
        printf("小哼win\n");
        printf("小哼当前手中的牌是");
        for(i=q1.head;i<=q1.tail-1;i++)
            printf(" %d",q1.data[i]);
        if(s.top>0) //如果桌上有牌则依次输出桌上的牌
        {
            printf("\n桌上的牌是");
            for(i=1;i<=s.top;i++)
                printf(" %d",s.data[i]);
        }
        else
            printf("\n桌上已经没有牌了");
        }
    }
```

反之，小哈获胜，代码的实现也是差不多的，就不再赘述了。到此，所有的代码实现就都讲完了。

在上面我们讲解的所有实现中，每个人打出一张牌后，判断能否赢牌这一点可以优化。之前我们是通过枚举桌上的每一张牌来实现的，即用了一个 for 循环来依次判断桌上的每一张牌是否与打出的牌相等。其实有更好的办法来解决这个问题，就是用一个数组来记录桌上

有哪些牌。因为牌面只有 1~9，因此只需开一个大小为 10 的数组来记录当前桌上已经有哪些牌面就可以了。

```
int book[10];
```

这里我再一次使用了 book 这个单词，因为这个单词有记录、登记的意思，而且单词拼写简洁。另外很多国外的算法书籍在处理需要标记问题的时候也都使用 book 这个单词，因此我这里就沿用了。当然你也可以使用 mark 等你自己觉得好理解的单词啦。下面需要将数组 book[1]~book[9]初始化为 0，因为刚开始桌面上一张牌也没有。

```
for(i=1;i<=9;i++)
    book[i]=0;
```

接下来，如果桌面上增加了一张牌面为 2 的牌，那就需要将 book[2]设置为 1，表示牌面为 2 的牌桌上已经有了。当然如果这张牌面为 2 的牌被拿走后，需要及时将 book[2]重新设置为 0，表示桌面上已经没有牌面为 2 的牌了。这样一来，寻找桌上是否有与打出的牌牌面相同的牌，就不需要再循环枚举桌面上的每一张牌了，而只需用一个 if 判断即可。这一点是不是有点像第 1 章第 1 节的桶排序的方法呢？具体如下。

```
t=q1.data[q1.head]; //小哼先亮出一张牌
if(book[t]==0) // 表明桌上没有牌面为t的牌
{
        //小哼此轮没有赢牌
    q1.head++; //小哼已经打出一张牌，所以要把打出的牌出队
    s.top++;
    s.data[s.top]=t; //再把打出的牌放到桌上，即入栈
    book[t]=1; //标记桌上现在已经有牌面为t的牌
}
```

OK，算法的实现讲完了，下面给出完整的代码，如下：

```
#include <stdio.h>
struct queue
{
    int data[1000];
    int head;
    int tail;
};
```

```
struct stack
{
    int data[10];
    int top;
};

int main()
{
    struct queue q1,q2;
    struct stack s;
    int book[10];
    int i,t;

    //初始化队列
    q1.head=1; q1.tail=1;
    q2.head=1; q2.tail=1;
    //初始化栈
    s.top=0;
    //初始化用来标记的数组，用来标记哪些牌已经在桌上
    for(i=1;i<=9;i++)
        book[i]=0;

    //依次向队列插入6个数
    //小哼手上的6张牌
    for(i=1;i<=6;i++)
    {
        scanf("%d",&q1.data[q1.tail]);
        q1.tail++;
    }
    //小哈手上的6张牌
    for(i=1;i<=6;i++)
    {
        scanf("%d",&q2.data[q2.tail]);
        q2.tail++;
    }
    while(q1.head<q1.tail && q2.head<q2.tail ) //当队列不为空的时候执行循环
    {
        t=q1.data[q1.head];//小哼出一张牌
        //判断小哼当前打出的牌是否能赢牌
        if(book[t]==0) //表明桌上没有牌面为t的牌
```

```
{
    //小哼此轮没有赢牌
    q1.head++; //小哼已经打出一张牌，所以要把打出的牌出队
    s.top++;
    s.data[s.top]=t; //再把打出的牌放到桌上，即入栈
    book[t]=1; //标记桌上现在已经有牌面为t的牌
}
else
{
    //小哼此轮可以赢牌
    q1.head++;//小哼已经打出一张牌，所以要把打出的牌出队
    q1.data[q1.tail]=t;//紧接着把打出的牌放到手中牌的末尾
    q1.tail++;
    while(s.data[s.top]!=t) //把桌上可以赢得的牌依次放到手中牌的末尾
    {
      book[s.data[s.top]]=0;//取消标记
        q1.data[q1.tail]=s.data[s.top];//依次放入队尾
        q1.tail++;
        s.top--; //栈中少了一张牌，所以栈顶要减1
    }
}

t=q2.data[q2.head]; //小哈出一张牌
//判断小哈当前打出的牌是否能赢牌
if(book[t]==0) //表明桌上没有牌面为t的牌
{
    //小哈此轮没有赢牌
    q2.head++; //小哈已经打出一张牌，所以要把打出的牌出队
    s.top++;
    s.data[s.top]=t; //再把打出的牌放到桌上，即入栈
    book[t]=1; //标记桌上现在已经有牌面为t的牌
}
else
{
    //小哈此轮可以赢牌
    q2.head++;//小哈已经打出一张牌，所以要把打出的牌出队
    q2.data[q2.tail]=t;//紧接着把打出的牌放到手中牌的末尾
    q2.tail++;
    while(s.data[s.top]!=t) //把桌上可以赢得的牌依次放到手中牌的末尾
    {
        book[s.data[s.top]]=0;//取消标记
```

```
                q2.data[q2.tail]=s.data[s.top];//依次放入队尾
                q2.tail++;
                s.top--;
            }
        }
    }

    if(q2.head==q2.tail)
    {
        printf("小哼win\n");
        printf("小哼当前手中的牌是");
        for(i=q1.head;i<=q1.tail-1;i++)
            printf(" %d",q1.data[i]);
        if(s.top>0) //如果桌上有牌则依次输出桌上的牌
        {
            printf("\n桌上的牌是");
            for(i=1;i<=s.top;i++)
                printf(" %d",s.data[i]);
        }
        else
            printf("\n桌上已经没有牌了");
    }
    else
    {
        printf("小哈win\n");
        printf("小哈当前手中的牌是");
        for(i=q2.head;i<=q2.tail-1;i++)
            printf(" %d",q2.data[i]);
        if(s.top>0) //如果桌上有牌则依次输出桌上的牌
        {
            printf("\n桌上的牌是");
            for(i=1;i<=s.top;i++)
                printf(" %d",s.data[i]);
        }
        else
            printf("\n桌上已经没有牌了");
    }

    getchar();getchar();
    return 0;
}
```

可以输入以下数据进行验证。

```
2 4 1 2 5 6
3 1 3 5 6 4
```

运行结果是：

```
小哼win
小哼当前手中的牌是 5 6 2 3 1 4 6 5
桌上的牌是 2 1 3 4
```

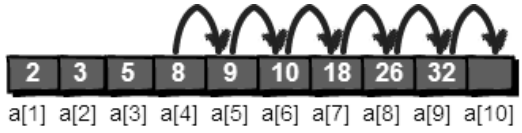
接下来你需要自己设计一些测试数据来检验你的程序。在设计测试数据的时候你要考虑各种情况，包括各种极端情况。通过设计测试数据来检测我们的程序是否“健壮”是非常重要的，如果你的程序可以通过任何一组测试数据，才表示你的程序是完全正确的。

如果你设计一些测试数据来验证的话，会发现我们刚才的代码其实还是有问题的。比如游戏可能无法结束。就是小哼和小哈可以永远玩下去，谁都无法赢得对方所有的牌。请你自己想一想如何解决游戏无法结束的问题。

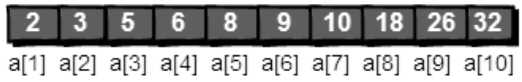
第 4 节　链表

在存储一大波数的时候，我们通常使用的是数组，但有时候数组显得不够灵活，比如下面这个例子。

有一串已经从小到大排好序的数 2 3 5 8 9 10 18 26 32。现需要往这串数中插入 6 使其得到的新序列仍符合从小到大排列。如我们使用数组来实现这一操作，则需要将 8 和 8 后面的数都依次往后挪一位，如下：



将8和8后面的数依次往后挪一位后得到的结果，如下



这样操作显然很耽误时间，如果使用链表则会快很多。那什么是链表呢？请看下图。

此时如果需要在 8 前面插入一个 6，就只需像下图这样更改一下就可以了，而无需再将 8 及后面的数都依次往后挪一位。是不是很节省时间呢？

那么如何实现链表呢？在 C 语言中可以使用指针和动态分配内存函数 malloc 来实现。指针？天啊！如果你在学习 C 语言的时候没有搞懂指针，或者还不知道指针是啥，不要紧，我们现在就回顾一下指针。指针其实超级简单。如果你已经对指针和 malloc 了如指掌则可以跳过下面这一小段，继续往后看。

先看下面两行语句。

```
int a;
int *p;
```

第一行我们很熟悉了，就是定义一个整型变量 a。第二行你会发现在 p 前面多了一个*号，这就表示定义了一个整型指针变量 p。即定义一个指针，只需在变量前面加一个*号就 OK 啦。

接下来，指针有什么作用呢？答案是：存储一个地址。确切地说是存储一个内存空间的地址，比如说整型变量 a 的地址。严格地说这里的指针 p 也只能存储“一个存放整数的内存空间”的地址，因为在定义的时候我们已经限制了这一点（即定义的时候*p 的前面是 int）。当然你也可以定义一个只能用来存储“一个存放浮点数的内存空间”的地址，例如：

```
double *p;
```

简单地说，指针就是用来存储地址的。你可能要问：不就是存储地址嘛，地址不都一样吗，为什么还要分不同类型的指针呢？不要着急，待会后面再解释。接下来需要解决的一个问题：整型指针 p 如何才能存储整型变量 a 的地址呢？很简单，如下：

```
p=&a;
```

&这个符号很熟悉吧，就是经常在 scanf 函数中用到的&。&叫取地址符。这样整型指针 p 就获得了（存储了）整型变量 a 的地址，我们可以形象地理解整型指针 p 指向了整型变量

a。p 指向了 a 之后，有什么用呢？用处就是我们可以用指针 p 来操作变量 a 了。比如我们可以通过操作指针 p 来输出变量 a 的值，如下：

```
#include <stdio.h>
int main()
{
    int a=10;
    int *p; //定义一个指针p
    p=&a; //指针p获取变量a的地址
    printf("%d",*p); //输出指针p所指向的内存中的值

    getchar();getchar();
    return 0;
}
```

运行结果是：

```
10
```

这里 printf 语句里面*p 中的*号叫做间接运算符，作用是取得指针 p 所指向的内存中的值。在 C 语言中*号有三个用途，分别是：

1. 乘号，用做乘法运算，例如 5*6。
2. 申明一个指针，在定义指针变量时使用，例如 int *p;。
3. 间接运算符，取得指针所指向的内存中的值，例如 printf("%d",*p);。

到目前为止，你可能还是觉得指针没啥子实际作用，好好的变量 a 想输出是的话直接 printf("%d",a); 不完了，没事搞个什么指针啊，多此一举。嗯，到目前为止貌似是这样的 O(∩_∩)O 哈哈~~不要着急，真枪实弹地来了。

回想一下，我们想在程序中存储一个整数 10，除了使用 int a;这种方式在内存中申请一块区域来存储，还有另外一种动态存储方法。

```
malloc(4);
```

malloc 函数的作用就是从内存中申请分配指定字节大小的内存空间。上面这行代码就申请了 4 个字节。如果你不知道 int 类型是 4 个字节的，还可以使用 sizeof(int)获取 int 类型所占用的字节数，如下：

```
malloc(sizeof(int));
```

现在你已经成功地从内存中申请了 4 个字节的空间来准备存放一个整数，可是如何来对这个空间进行操作呢？这里我们就需要用一个指针来指向这个空间，即存储这个空间的首地址。

```
int *p;
p=(int *)malloc(sizeof(int));
```

需要注意，malloc 函数的返回类型是 void * 类型。void * 表示未确定类型的指针。在 C 和 C++中，void * 类型可以强制转换为任何其他类型的指针。上面代码中我们将其强制转化为整型指针，以便告诉计算机这里的 4 个字节作为一个整体用来存放整数。还记得我们之前遗留了一个问题：指针就是用来存储内存地址的，为什么要分不同类型的指针呢？因为指针变量存储的是一个内存空间的首地址（第一个字节的地址），但是这个空间占用了多少个字节，用来存储什么类型的数，则是由指针的类型来标明的。这样系统才知道应该取多少个连续内存作为一个数据。

OK，现在我们可以通过指针 p 对刚才申请的 4 个字节的空间进行操作了，例如我们向这个空间中存入整数 10，如下：

```
*p=10;
```

完整代码如下，注意当在程序中使用 malloc 函数时需要用到 stdlib.h 头文件。

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *p; //定义一个指针p
    p=(int *)malloc(sizeof(int)); //指针p获取动态分配的内存空间地址
    *p=10; //向指针p所指向的内存空间中存入10
    printf("%d",*p); //输出指针p所指向的内存中的值

    getchar();getchar();
    return 0;
}
```

运行结果是：

```
10
```

到这里你可能要问：为什么要用这么复杂的办法来存储数据呢？因为之前的方法，我们必须预先准确地知道所需变量的个数，也就是说我们必须定义出所有的变量。比如我们定义了 100 个整型变量，那么程序就只能存储 100 个整数，如果现在的实际情况是需要存储 101 个，那必须修改程序才可以。如果有一天你写的软件已经发布或者交付使用，却发现要存储 1000 个数才行，那就不得不再次修改程序，重新编译程序，发布一个新版本来代替原来的。而有了 malloc 函数我们便可以在程序运行的过程中根据实际情况来申请空间。

啰嗦了半天，总算介绍完了什么是指针以及如何动态申请空间。注意，本节接下来的代码对于还没有理解指针的朋友来说可能不太容易，不要紧，如果你痛恨指针，大可直接跳过下面的内容直接进入下一节。下一节中我将介绍链表的另外一种实现方式——数组模拟链表。

首先我们来看一下，链表中的每一个结点应该如何存储。

每一个结点都由两个部分组成。左边的部分用来存放具体的数值，那么用一个整型变量就可以；右边的部分需要存储下一个结点的地址，可以用指针来实现（也称为后继指针）。这里我们定义一个结构体类型来存储这个结点，如下。

```
struct node
{
    int data;
    struct node *next;
};
```

上面代码中，我们定义了一个叫做 node 的结构体类型，这个结构体类型有两个成员。第一个成员是整型 data，用来存储具体的数值；第二个成员是一个指针，用来存储下一个结点的地址。因为下一个结点的类型也是 struct node，所以这个指针的类型也必须是 struct node * 类型的指针。

如何建立链表呢？首先我们需要一个头指针 head 指向链表的最开始。当链表还没有建立的时候头指针 head 为空（也可以理解为指向空结点）。

```
struct node *head;
head = NULL;//头指针初始为空
```

现在我们来创建第一个结点，并用临时指针 p 指向这个结点。

```
struct node *p;
//动态申请一个空间，用来存放一个结点，并用临时指针p指向这个结点
p=(struct node *)malloc(sizeof(struct node));
```

接下来分别设置新创建的这个结点的左半部分和右半部分。

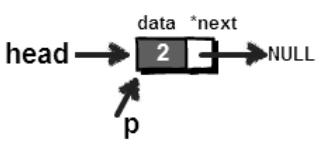
```
scanf("%d",&a);
p->data=a;//将数据存储到当前结点的data域中
p->next=NULL;//设置当前结点的后继指针指向空，也就是当前结点的下一个结点为空
```

上面的代码中我们发现了一个很奇怪的符号“->”。->叫做结构体指针运算符，也是用来访问结构体内部成员的。因为此处 p 是一个指针，所以不能使用.号访问内部成员，而要使用->。

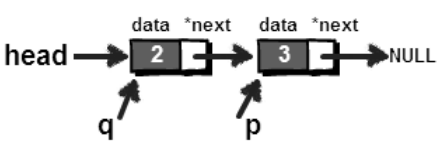
下面来设置头指针并设置新创建结点的*next 指向空。头指针的作用是方便以后从头遍历整个链表。

```
if(head==NULL)
    head=p;//如果这是第一个创建的结点，则将头指针指向这个结点
else
    q->next=p;//如果不是第一个创建的结点，则将上一个结点的后继指针指向当前结点
```

如果这是第一个创建的结点，则将头指针指向这个结点。



如果不是第一个创建的结点，则将上一个结点的后继指针指向当前结点。



最后要将指针 q 也指向当前结点，因为待会儿临时指针 p 将会指向新创建的结点。

```
q=p;//指针q也指向当前结点
```

完整代码如下。

```
#include <stdio.h>
#include <stdlib.h>
//这里创建一个结构体用来表示链表的结点类型
struct node
{
    int data;
    struct node *next;
};

int main()
{
    struct node *head,*p,*q,*t;
    int i,n,a;
    scanf("%d",&n);
    head = NULL;//头指针初始为空
    for(i=1;i<=n;i++)//循环读入n个数
    {
        scanf("%d",&a);
        //动态申请一个空间，用来存放一个结点，并用临时指针p指向这个结点
        p=(struct node *)malloc(sizeof(struct node));
        p->data=a;//将数据存储到当前结点的data域中
        p->next=NULL;//设置当前结点的后继指针指向空，也就是当前结点的下一个结点为空
        if(head==NULL)
            head=p;//如果这是第一个创建的结点，则将头指针指向这个结点
        else
            q->next=p;//如果不是第一个创建的结点,则将上一个结点的后继指针指向当前结点

        q=p;//指针q也指向当前结点
    }

    //输出链表中的所有数
    t=head;
    while(t!=NULL)
    {
```

```
            printf("%d ",t->data);
            t=t->next;//继续下一个结点
        }

        getchar();getchar();
        return 0;
    }
```

需要说明的一点是：上面这段代码没有释放动态申请的空间，虽然没有错误，但是这样会很不安全，有兴趣的朋友可以去了解一下 free 命令。

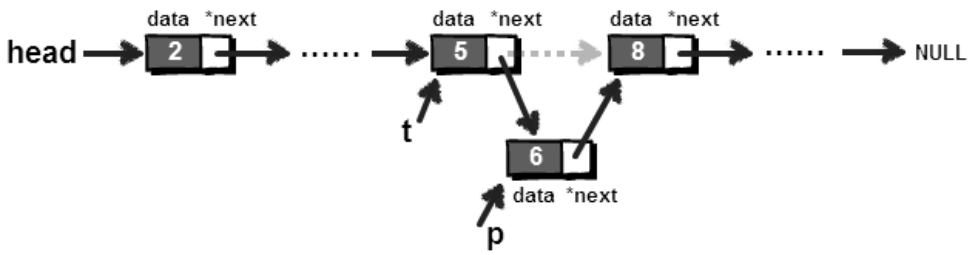
可以输入以下数据进行验证。

```
9
2 3 5 8 9 10 18 26 32
```

运行结果是：

```
2 3 5 8 9 10 18 26 32
```

接下来需要往链表中插入 6，操作如下。



首先用一个临时指针 t 从链表的头部开始遍历。

```
t=head;//从链表头部开始遍历
```

等到指针 t 的下一个结点的值比 6 大的时候，将 6 插入到中间。即 t->next->data 大于 6 时进行插入，代码如下。

```
scanf("%d",&a);//读入待插入的数
while(t!=NULL)//当没有到达链表尾部的时候循环
{
    if(t->next->data > a)//如果当前结点下一个结点的值大于待插入数，将数插入到中间
```

```
        {
            p=(struct node *)malloc(sizeof(struct node));//动态申请一个空间，用来
存放新增结点
            p->data=a;
            p->next=t->next;//新增结点的后继指针指向当前结点的后继指针所指向的结点
            t->next=p;//当前结点的后继指针指向新增结点
            break;//插入完毕退出循环
        }
        t=t->next;//继续下一个结点
    }
```

完整代码如下。

```
#include <stdio.h>
#include <stdlib.h>

//这里创建一个结构体用来表示链表的结点类型
struct node
{
    int data;
    struct node *next;
};

int main()
{
    struct node *head,*p,*q,*t;
    int i,n,a;
    scanf("%d",&n);
    head = NULL;//头指针初始为空
    for(i=1;i<=n;i++)//循环读入n个数
    {
        scanf("%d",&a);
        //动态申请一个空间，用来存放一个结点，并用临时指针p指向这个结点
        p=(struct node *)malloc(sizeof(struct node));
        p->data=a;//将数据存储到当前结点的data域中
        p->next=NULL;//设置当前结点的后继指针指向空，也就是当前结点的下一个结点为空
        if(head==NULL)
            head=p;//如果这是第一个创建的结点，则将头指针指向这个结点
        else
            q->next=p;//如果不是第一个创建的结点，则将上一个结点的后继指针指向当前结点
```

```
            q=p;//指针q也指向当前结点
        }

        scanf("%d",&a);//读入待插入的数
        t=head;//从链表头部开始遍历
        while(t!=NULL)//当没有到达链表尾部的时候循环
        {
            if(t->next->data > a)//如果当前结点下一个结点的值大于待插入数，将数插入到中间
            {
                p=(struct node *)malloc(sizeof(struct node));//动态申请一个空间，
用来存放新增结点
                p->data=a;
                p->next=t->next;//新增结点的后继指针指向当前结点的后继指针所指向的结点
                t->next=p;//当前结点的后继指针指向新增结点
                break;//插入完毕退出循环
            }
            t=t->next;//继续下一个结点
        }

        //输出链表中的所有数
        t=head;
        while(t!=NULL)
        {
            printf("%d ",t->data);
            t=t->next;//继续下一个结点
        }

        getchar();getchar();
        return 0;
    }
```

可以输入以下数据进行验证。

```
9
2 3 5 8 9 10 18 26 32
6
```

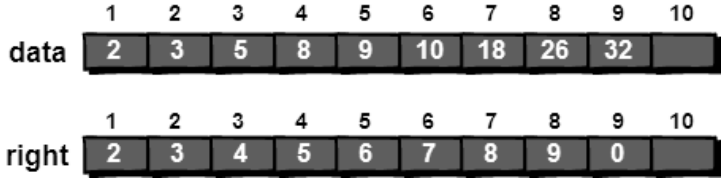
运行结果是：

```
2 3 5 6 8 9 10 18 26 32
```

第 5 节　模拟链表

如果你觉得上一节的代码简直是天书，或者你压根就很讨厌指针这些东西，没关系！链表还有另外一种使用数组来实现的方式，叫做模拟链表，我们一起来看看。

链表中的每一个结点只有两个部分。我们可以用一个数组 data 来存储每序列中的每一个数。那每一个数右边的数是谁，这一点该怎么解决呢？上一节中是使用指针来解决的，这里我们只需再用一个数组 right 来存放序列中每一个数右边的数是谁就可以了，具体怎么做呢？



上图的两个数组中，第一个整型数组 data 是用来存放序列中具体数字的，另外一个整型数组 right 是用来存放当前序列中每一个元素右边的元素在数组 data 中位置的。例如 right[1] 的值为 2，就表示当前序列中 1 号元素右边的元素存放在 data[2]中；如果是 0，例如 right[9] 的值为 0，就表示当前序列中 9 号元素的右边没有元素。

现在需要在 8 前面插入一个 6，只需将 6 直接存放在数组 data 的末尾即 data[10]=6。接下来只需要将 right[3]改为 10，表示新序列中 3 号元素右边的元素存放在 data[10]中。再将 right[10] 改为 4，表示新序列中 10 号元素右边的元素存放在 data[4]中。这样我们通过 right 数组就可以从头到尾遍历整个序列了（序列的每个元素的值存放在对应的数组 data 中），如下。

完整的代码实现如下。

```
#include <stdio.h>
int main()
{
    int data[101],right[101];
```

```
        int i,n,t,len;
        //读入已有的数
        scanf("%d",&n);
        for(i=1;i<=n;i++)
            scanf("%d",&data[i]);
        len=n;
        //初始化数组right
        for(i=1;i<=n;i++)
        {
            if(i!=n)
                right[i]=i+1;
            else
                right[i]=0;
        }
        //直接在数组data的末尾增加一个数
        len++;
        scanf("%d",&data[len]);

        //从链表的头部开始遍历
        t=1;
        while(t!=0)
        {
            if(data[right[t]]>data[len])//如果当前结点下一个结点的值大于待插入数，将
数插入到中间
            {
                right[len]=right[t];//新插入数的下一个结点标号等于当前结点的下一个结
点编号
                right[t]=len;//当前结点的下一个结点编号就是新插入数的编号
                break;//插入完成跳出循环
            }
            t=right[t];
        }
        //输出链表中所有的数
        t=1;
        while(t!=0)
        {
            printf("%d ",data[t]);
            t=right[t];
        }
```

```
    getchar();
    getchar();
    return 0;
}
```

可以输入以下数据进行验证。

```
9
2 3 5 8 9 10 18 26 32
6
```

运行结果是：

```
2 3 5 6 8 9 10 18 26 32
```

使用模拟链表也可以实现双向链表和循环链表，大家可以自己来试一试。