```
int a[101];
int main()
{
  int n,i;
  cin>>n;
  for (i=1;i<=n;i++)
    cin>>a[i];
  qsort(1,n);
  for (i=1;i<=n;i++)
    cout<<a[i]<<" ";
  cout<<endl;
}
void qsort(int l,int r)
  {
    int i,j,mid,p;
    i=l;j=r;
    mid=a[(l+r)/2];          //将当前序列在中间位置的数定义为分隔数
    do
      {
        while (a[i]<mid) i++;//在左半部分寻找比中间数大的数
        while (a[j]>mid) j--;//在右半部分寻找比中间数小的数
        if (i<=j)
          {                  //若找到一组与排序目标不一致的数对,则交换它们
            p=a[i];a[i]=a[j];a[j]=p;
            i++;j--;         //继续找
          }
      }while(i<=j);          //注意这里不能少了等号
    if (l<j) qsort(l,j);     //若未到两个数的边界,则递归搜索左右区间
    if (i<r) qsort(i,r);
  }
```

快速排序的时间的复杂性是 $O(n\log_2 n)$,速度快,但它是不稳定的排序方法。就平均时间而言,快速排序是目前被认为是最好的一种内部排序方法。

由以上讨论可知,从时间上看,快速排序的平均性能优于前面讨论过的各种排序方法,但快速排序需一个栈空间来实现递归。若每一趟排序都将记录序列均匀地分割成长度相接近的两个子序列,则栈的最大深度为 $\log(n+1)$。

6. 归并排序

归并排序是建立在归并操作上的一种有效的排序算法,该算法是采用分治法(Divide and Conquer)的一个非常典型的应用。将已有序的子序列合并,得到完全有序的序列;即先使每个子序列有序,再使子序列段间有序。若将两个有序表合并成一个有序表,称为二路归并。

例如有 8 个数据需要排序:10 4 6 3 8 2 5 7

归并排序主要分两大步:分解、合并。

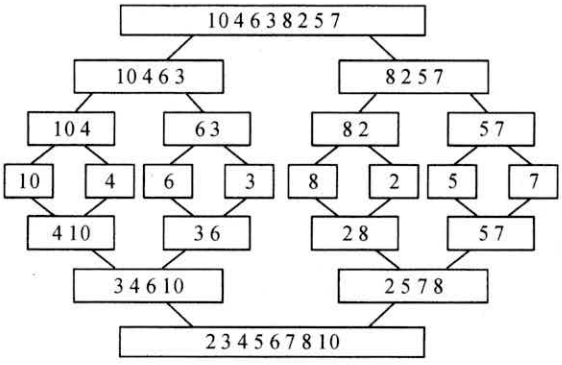

图 2-1

合并过程为:比较 a[i]和 a[j]的大小,若 a[i]≤a[j],则将第一个有序表中的元素 a[i]复制到 r[k]中,并令 i 和 k 分别加上 1;否则将第二个有序表中的元素 a[j]复制到 r[k]中,并令 j 和 k 分别加上 1,如此循环下去,直到其中一个有序表取完,然后再将另一个有序表中剩余的元素复制到 r 中从下标 k 到下标 t 的单元。归并排序的算法我们通常用递归实现,先把待排序区间[s,t]以中点二分,接着把左边子区间排序,再把右边子区间排序,最后把左区间和右区间用一次归并操作合并成有序的区间[s,t]。

【程序实现】

```
void msort(int s,int t)
{
    if(s==t) return;                //如果只有一个数字则返回,无须排序
    int mid=(s+t)/2;
    msort(s,mid);                   //分解左序列
    msort(mid+1,t);                 //分解右序列
    int i=s, j=mid+1, k=s;          //接下来合并

    while(i<=mid && j<=t)
    {
        if(a[i]<=a[j])
        {
        r[k]=a[i]; k++; i++;
        }else{
        r[k]=a[j]; k++; j++;
        }
    }
    while(i<=mid)                   //复制左边子序列剩余
    {
        r[k]=a[i]; k++; i++;
    }
```

```
    while(j<=t)                    //复制右边子序列剩余
    {
        r[k]=a[j]; k++; j++;
    }
    for(int i=s; i<=t; i++) a[i]=r[i];
}
```

归并排序的时间复杂度是O(nlogn)，速度快。同时，归并排序是稳定的排序。即相等的元素的顺序不会改变。如输入记录1(1) 3(2) 2(3) 2(4) 5(5)（括号中是记录的关键字）时输出的1(1) 2(3) 2(4) 3(2) 5(5)中的2和2是按输入的顺序。这对要排序数据包含多个信息而要按其中的某一个信息排序，要求其它信息尽量按输入的顺序排列时很重要，这也是它比快速排序优势的地方。

7. 逆序对

上述提到归并排序是稳定的排序，相等的元素的顺序不会改变，进而用其可以解决逆序对的问题。首先我们了解一下什么是逆序对。

逆序对：设A为一个有n个数字的有序集（n>1），其中所有数字各不相同。如果存在正整数i，j使得$1 \leqslant i < j \leqslant n$而且A[i] > A[j]，则<A[i]，A[j]>这个有序对称为A的一个逆序对，也称作逆序数。

例如，数组(3,1,4,5,2)的逆序对有(3,1)，(3,2)，(4,2)，(5,2)，共4个。

所谓逆序对的问题，即对给定的数组序列，求其逆序对的数量。

从逆序对定义上分析，逆序对就是数列中任意两个数满足大的在前，小的在后的组合。如果将这些逆序对都调整成顺序（小的在前，大的在后），那么整个数列就变得有序，即排序。因而，容易想到冒泡排序的机制正好是利用消除逆序来实现排序的，也就是说，交换相邻两个逆序数，最终实现整个序列有序，那么交换的次数即为逆序对的数量。

冒泡排序可以解决逆序对问题，但是由于冒泡排序本身效率不高，时间复杂度为O(n^2)，对于n比较大的情况就没用武之地了。我们可以这样认为，冒泡排序求逆序对效率之所以低，是因为其在统计逆序对数量的时候是一对一对统计的，而对于范围为n的序列，逆序对数量最大可以是(n+1) * n/2，因此其效率太低。那怎样可以一下子统计多个，而不是一个一个累加呢？这个时候，归并排序就可以帮我们来解决这个问题。

在合并操作中，我们假设左右两个区间元素为：

左边：{3 4 7 9}　　右边：{1 5 8 10}

那么合并操作的第一步就是比较3和1，然后将1取出来，放到辅助数组中，这个时候我们发现，右边的区间如果是当前比较的较小值，那么其会与左边剩余的数字产生逆序关系，也就是说1和3、4、7、9都产生了逆序关系，我们可以一下子统计出有4对逆序对。接下来3,4取下来放到辅助数组后，5与左边剩下的7、9产生了逆序关系，我们可以统计出2对。依此类推，8与9产生1对，那么总共有4+2+1对。这样统计的效率就会大大提高，便可较好地解决逆序对问题。

而在算法的实现中，我们只需略微修改原有归并排序，当右边序列的元素为较小值时，就统计其产生的逆序对数量，即可完成逆序对的统计。

【程序实现】

```
void msort(int s,int t)
{
    if(s==t) return;                    //如果只有一个数字则返回,无须排序
  int mid=(s+t)/2;
  msort(s,mid);                         //分解左序列
  msort(mid+1,t);                       //分解右序列
  int i=s, j=mid+1, k=s;                //接下来合并
  while(i<=mid && j<=t)
  {
      if(a[i]<=a[j])
      {
          r[k]=a[i]; k++; i++;
      }else{
          r[k]=a[j]; k++; j++;
          ans+=mid-i+1;                 //统计产生逆序对的数量
      }
  }
  while(i<=mid)                         //复制左边子序列剩余
  {
      r[k]=a[i]; k++; i++;
  }
  while(j<=t)                           //复制右边子序列剩余
  {
      r[k]=a[j]; k++; j++;
  }
  for(int i=s; i<=t; i++) a[i]=r[i];
}
```

其中 ans+=mid-i+1 这句代码统计新增逆序对的数量,ans 作为全局变量,用于统计逆序对的数量,此时 ans 要增加左边区间剩余元素的个数。当归并排序结束后,逆序对问题也得到解决,ans 即为逆序对的数量。

8. 各种排序算法的比较

(1)稳定性比较

插入排序、冒泡排序、二叉树排序、二路归并排序及其他线性排序是稳定的。

选择排序、希尔排序、快速排序、堆排序是不稳定的。即有跨度的交换都会导致不稳定。

(2)时间复杂性比较

插入排序、冒泡排序、选择排序的时间复杂性为 $O(n^2)$;快速排序、堆排序、归并排序的时间复杂性为 $O(n\log_2 n)$;桶排序的时间复杂性为 $O(n)$;