第3章 排 序

读者从第 2 章的讨论中已经看到，在很多场合，用有序表相对无序表可以节省算法的时间，提高解决问题的效率。那么，如何得到有序的顺序表？当然可以在构造顺序表时依值的有序性进行插入求得，对无序的顺序表进行“排序”也是将它转化为有序的顺序表的一种途径。

从 2.4 节有序表的定义可见，可以转化为有序表的线性表中的数据元素必须是相互之间可以进行比较，在此称这种线性表为“可排序的表”。更一般化的情况，设数据元素由多个数据项构成，其中有一个被称做“关键字”的数据项，数据元素之间可按其关键字的“大小”进行比较，并且这个“大小”的含义是广义的，它可以理解为关键字之间存在某种“领先”的关系。本章中将称上述定义的数据元素为“记录”，可用 C 语言描述如下：

```
typedef int KeyType;          // 为简单起见,定义关键字类型为整型
typedef struct {
  KeyType key;                // 关键字项
  InfoType otherinfo;         // 其他数据项
} RcdType;                    // 记录类型
```

本章讨论的排序算法将对上述定义的记录进行排序，但在解释排序过程的图例中仅标出了记录的关键字。本章将首先提出有关排序的基本概念，然后介绍几种常用的内部排序方法，并分析它们的时间复杂度，最后对各种方法进行综合比较。

3.1 排序的基本概念

排序(sorting)是按关键字[①]的非递减或非递增顺序对一组记录重新进行整队(或排列)的操作。确切描述如下：

假设含有 n 个记录的序列为

$$\{r_1, r_2, \cdots, r_n\} \tag{3-1}$$

它们的关键字相应为

$$\{k_1, k_2, \cdots, k_n\}$$

对式(3-1)的记录序列进行排序就是要确定序号 1,2,…,n 的一种排列

$$p_1, p_2, \cdots, p_n$$

使其相应的关键字满足如下的非递减(或非递增[②])的关系：

---

① 从排序的本意而言，排序可以对单个关键字进行，也可以对多个关键字的组合进行，可统称排序时所依赖的准绳为“排序码”。为讨论方便起见，本章约定排序只对单关键字进行排序。

② 若将式(3-2)中的“≤”改为“≥”，则满足非递增关系。

$$k_{p_1} \leqslant k_{p_{21}} \leqslant \cdots \leqslant k_{p_n} \tag{3-2}$$

也就是使式(3-1)的记录序列重新排列成一个按关键字有序的序列

$$\{r_{p_1} \leqslant r_{p_2} \leqslant \cdots \leqslant r_{p_n}\}^{①} \tag{3-3}$$

当待排序记录中的关键字 $k_i(i=1,2,\cdots,n)$都不相同时，则任何一个记录的无序序列经排序后得到的结果是唯一的；反之，若待排序的序列中存在两个或两个以上关键字相等的记录时，则排序所得到的记录序列的结果不唯一。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 $i<j$)。若在排序后的序列中 r_i 仍领先于 r_j，则称所用的排序方法是**稳定的**；反之，若可能使排序后的序列中 r_j 领先于 r_i，则称所用的排序方法是**不稳定的**②。在某些有特殊要求的应用中需要考虑稳定性的问题。

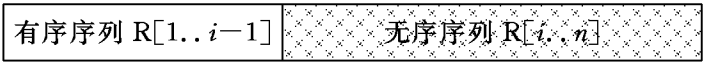
根据在排序过程中涉及的存储器不同，可将排序方法分为两大类：(1) **内部排序**：在排序进行的过程中不使用计算机外部存储器的排序过程。(2) **外部排序**：在排序进行的过程中需要对外存进行访问的排序过程。本章仅讨论各种内部排序的方法。

待排序的记录序列可以用顺序表表示，也可以用链表表示。本章讨论的排序算法一律以下列说明的顺序表为操作对象。

```
const MAXSIZE=20;              // 一个用作示例的小顺序表的最大长度
typedef struct {
  RcdType  r[MAXSIZE+1];       // r[0]闲置或作为判别标志的"哨兵"单元
  int      length;             // 顺序表排序的记录空间为 r[1..length]
} SqList;                      // 顺序表类型
```

内部排序的过程是一个逐步扩大记录的有序序列长度的过程。通常在排序的过程中，参与排序的记录序列中可划分为两个区域：有序序列区和无序序列区，其中有序序列区中的记录已按关键字非递减有序排列。使有序序列区中记录的数目增加一个或几个的操作称为一趟排序。下面以**选择排序**(selection sort)为例剖析内部排序的过程。

在选择排序的过程中，待排记录序列的状态为



并且有序序列中所有记录的关键字均不大于无序序列中记录的关键字，则第 i 趟选择排序的操作是，从无序序列 R[i..n]的 $n-i+1$ 个记录中选出关键字最小的记录 R[j]和 R[i] 交换，从而使有序序列区从 R[1..$i-1$]扩大至 R[1..i]，如图 3.1 所示。

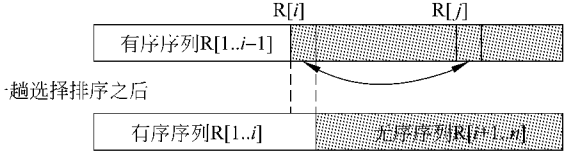


图 3.1　一趟选择排序操作示意图

① "≤"是偏序关系符号，读作"小于等于"。

② 对不稳定的排序方法，只要列举一个关键字实例，说明它不稳定即可。

一趟选择排序的算法如下：

算法 3.1

```
void SelectPass(SqList &L, int i)
{
  // 已知 L.r[1..i-1]中记录按关键字非递减有序,本算法实现第 i 趟选择排序,
  // 即在 L.r[i..n]的记录中选出关键字最小的记录 L.r[j]和 L.r[i]交换
  RcdType W;
  j=i;                  // j 指示关键字最小记录的位置,初值设为 i
  for(k=i+1; k<=L.length; k++)
    if(L.r[k].key<L.r[j].key) j=k;          // 暂不进行记录交换,只记录位置
  if(i !=j)
    { W=L.r[j];L.r[j]=L.r[i];L.r[i]=W;}   // 最后互换记录 R[j] 和 R[i]
} // SelectPass
```

整个选择排序的过程是一趟选择排序过程的多次重复，融合 SelectPass，其算法如下：

算法 3.2

```
void SelectSort(SqList &L)
{
  // 对顺序表 L 作简单选择排序
  RcdType W;
  for(i=1; i<L.length; ++i) {           // 选择第 i 个小的记录,并交换到位
    j=i;
    for(k=i+1; k<=L.length; k++)        // 在 L.r[i..L.length]中选择 key 最小的记录
      if(L.r[k].key<L.r[j].key) j=k;
    if(i!=j)
      { W=L.r[j];L.r[j]=L.r[i];L.r[i]=W;}     // 与第 i 个记录交换
  }//for
} // SelectSort
```

例如，对下列一组关键字：

$$(49_1, 38, 65, 49_2, 76, 13, 27, 52)$$

进行选择排序过程中，每一趟排序之后的状况如图 3.2 所示。其中 49_1 和 49_2 表示两个关键字同为 49 的不同记录。

初始关键字：	49_1	38	65	49_2	76	13	27	52
i=1	**(13)**	38	65	49_2	76	**49_1**	27	52
i=2	**(13**	**27)**	65	49_2	76	49_1	**38**	52
i=3	**(13**	**27**	**38)**	49_2	76	49_1	**65**	52
i=4	**(13**	**27**	**38**	**49_2)**	76	49_1	65	52
i=5	**(13**	**27**	**38**	**49_2**	**49_1)**	**76**	65	52
i=6	**(13**	**27**	**38**	**49_2**	**49_1**	**52)**	65	**76**
i=7	**(13**	**27**	**38**	**49_2**	**49_1**	**52**	**65)**	**94**

图 3.2　选择排序示例

从上述选择排序的过程可见，在内部排序的过程中主要进行下列两种基本操作：(1)比较两个关键字的大小；(2)将元素从一个位置移动至另一个位置。因此对内部排序的时间复杂度的分析就是以这两种操作的执行次数为依据。从算法 3.1 可见，在第 i 趟选择排序过程中，需进行 $n-i$ 次关键字间的“比较”和交换记录时所需的至多 3 次“移动”记录操作。

整个选择排序过程中，需进行 $\frac{n(n-1)}{2}$ 次关键字间的比较和至多 $3(n-1)$ 次移动记录，因此它的时间复杂度为 $O(n^2)$。选择排序是在原记录数据空间上通过记录的交换进行的，只在交换记录时需要用一个辅助工作变量，因此它的空间复杂度为 $O(1)$。

就选择排序方法本身讲，它是一种稳定的排序方法，但图 3.2 所表现出来的现象是不稳定的，这是由于上述实现选择排序的算法采用的“交换记录”的策略所造成的，若改变这个策略，可以写出不产生“不稳定现象”的选择排序算法。

内部排序的方法很多，就排序算法的时间复杂度来区分，则可分为三类：(1)简单的排序方法，其时间复杂度为 $O(n^2)$；(2)先进的排序方法，其时间复杂度为 $O(n\log n)$；(3)基数排序，其时间复杂度为 $O(d\times n)$。本章仅就每一类介绍几种常用的排序方法。

3.2 简单排序方法

简单排序算法中，除上节讨论的选择排序之外，常用的还有插入排序和起泡排序。

3.2.1 插入排序

插入排序(insertion sort)的基本操作是将当前无序序列区 R[$i..n$]中的记录 R[i]“插入”到有序序列区 R[$1..i-1$]中，使有序序列区的长度增 1，如图 3.3 所示。

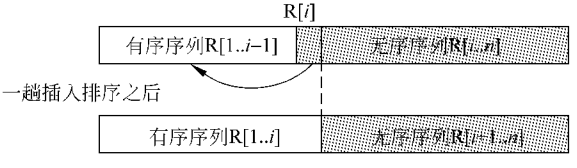


图 3.3 一趟插入排序操作示意图

例如，对下列一组记录的关键字：

$$(49,38,65,76,27,13,91,52) \tag{3-4}$$

进行插入排序的过程中，前 4 个记录已按关键字非递减的顺序有序排列，构成一个含 4 个记录的有序序列

$$(38,49,65,76) \tag{3-5}$$

现要将式(3-4)中第 5 个(关键字为 27 的)记录插入到式(3-5)的序列中去，以得到一个新的含 5 个记录的有序序列

$$(27,38,49,65,76) \tag{3-6}$$

称这个过程为“一趟插入排序”。

这个插入操作显然应该利用第 2 章讨论的算法 2.25 来完成。回顾算法 2.25，为了防

止循环变量出界，在循环结束的条件中加上了(i>=0)的判别，若将这个算法用在插入排序上，将会因为这个条件的判别增加排序的时间。当排序问题的规模较大或经常需要进行排序的应用场合，这一操作的时间开销就该有所计较。为此改写在有序表中进行插入的算法如下：利用 L.r[0]分量“复制”待插入的记录，则在向前查找插入位置时，循环变量就不可能发生出界的情况，称 L.r[0]为“哨兵”。由此可以改写这个插入算法如下：

算法 3.3

```
void InsertPass(SqList &L, int i)
{
  // 已知 L.r[1..i-1]中的记录已按关键字非递减的顺序有序排列,本算法实现
  // 将 L.r[i]插入其中,并保持 L.r[1..i]中记录按关键字非递减顺序有序
  L.r[0]=L.r[i];                          // 复制为哨兵
  for(j=i-1; L.r[0].key<L.r[j].key; --j)
    L.r[j+1]=L.r[j];                      // 记录后移
  L.r[j+1]=L.r[0];                        // 插入到正确位置
} // InsertPass
```

整个插入排序需进行 $n-1$ 趟“插入”。只含一个记录的序列必定是个有序序列，因此插入应从 $i=2$ 起进行。此外，若第 i 个记录的关键字不小于第 $i-1$ 个记录的关键字，“插入”也就不需要进行了。插入排序的算法如下：

算法 3.4

```
void InsertSort(SqList &L)
{
  // 对顺序表 L 作插入排序
  for(i=2; i<=L.length; ++i)
    if(L.r[i].key<L.r[i-1].key) {         // 当“<”时,才需将 L.r[i]插入有序子表
      L.r[0]=L.r[i];                      // 复制为哨兵
      for(j=i-1; L.r[0].key<L.r[j].key; --j)
        L.r[j+1]=L.r[j];                  // 记录后移
      L.r[j+1]=L.r[0];                    // 插入到正确位置
    } // if
} // InsertSort
```

例如，对下列一组关键字进行插入排序过程中，每一趟排序之后的状况如图 3.4 所示。

插入排序算法的分析如下：由于在一趟插入排序中，L.r[0].key 至多和 i 个关键字进行比较，再加上“之前”的一次比较，则对于每个“i”至多进行($i+1$)次关键字间的比较，而整个插入排序中，i 从 2 变化到 n，因此插入排序的时间复杂度为 $O(n^2)$。类似选择排序，整个排序过程中也仅需一个哨兵的辅助空间，所以它的空间复杂度为 $O(1)$。

显然，如果原始记录已按关键字“非递减顺序”有序排列，则将使插入排序呈现最好状态，在每一趟中仅作一次比较，总的比较次数达到最小值 $C_{\min}=n-1$，且记录不作移动；反之，如果原始记录是按关键字“非递增顺序”有序(又称“逆序”)排列，则插入排序呈现最坏状态。此时总的比较次数取最大值 $C_{\max}=(n+4)(n-1)/2$，并作同样次数的记录移动。

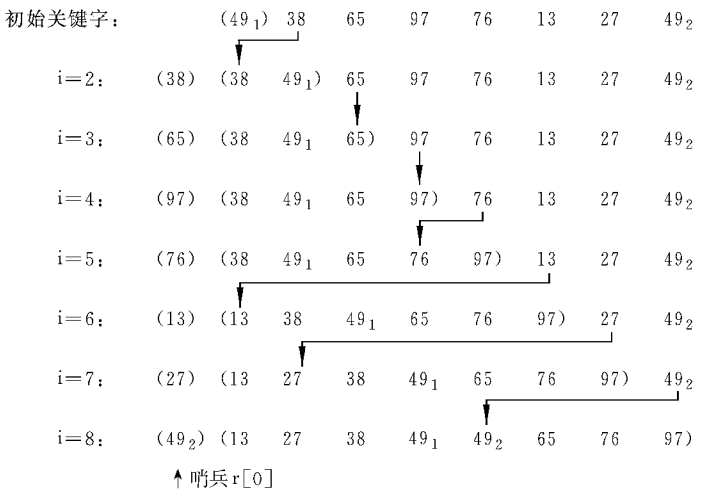


图 3.4　插入排序示例

从以上分析可知，当关键字分布情况不同时，算法在执行过程中的时间消耗也颇有差异。在随机情况下，实际的比较次数估计比最坏情况的要少，而且对插入排序而言，关键字分布的有序性越强，比较次数也越少。但对选择排序而言，关键字的分布对比较次数没有影响。

插入排序是稳定的排序方法。

3.2.2　起泡排序

起泡排序(bubble sort)的基本思想是通过对无序序列区中的记录进行相邻记录关键字间的“比较”和记录位置的“交换”实现关键字较小的记录向“一头”飘浮，而关键字较大的记录向“另一头”下沉，从而达到记录按关键字非递减顺序有序排列的目标。

假设在排序过程中，记录序列 R[1..n]分为无序序列 R[1..i]和有序序列 R[i+1..n]两个区域，则本趟起泡排序的基本操作是从第 1 个记录起，比较第 1 个记录和第 2 个记录的关键字，若呈“逆序”关系，则将两个记录交换，然后比较第 2 个记录和第 3 个记录的关键字，若呈“逆序”，则交换之。依此类推，直至比较了 R[i−1]和 R[i]之后，该无序区中关键字最大的记录将定位在 R[i]的位置上，如图 3.5 所示。

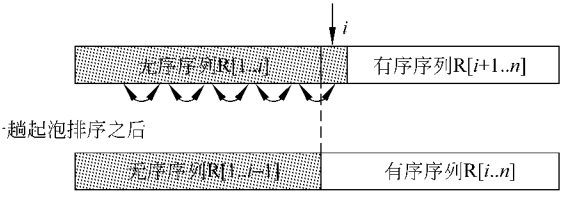


图 3.5　一趟起泡排序操作示意图

一般情况下，整个起泡排序只需进行 $k(1\leqslant k<n)$趟起泡操作，起泡排序的结束条件是“在某一趟排序过程中没有进行记录交换的操作”。图 3.6 展示了起泡排序的一个例子。从

图示中可见，在起泡排序的过程中，关键字较小的记录如“起泡”般逐趟往上“飘浮”，而关键字较大的记录如石头般“下沉”，每一趟有一块“最大”的石头沉落水底。

初始关键字	第一趟排序后	第二趟排序后	第三趟排序后	第四趟排序后	第五趟排序后	第六趟排序后
49_1	38	38	38	38	13	**13**
38	49_1	49_1	49_1	13	26	**27**
65	65	65	13	27	38	**38**
97	76	13	27	49_1	**49_1**	
76	13	27	49_2	**49_2**		
13	27	49_2	**65**			
27	49_2	**76**				
49_2	**97**					

图 3.6　起泡排序示例

起泡排序的算法描述如下：

算法 3.5

```
void BubbleSort(SqList &L)
{
  // 对顺序表 L 作起泡排序
  RcdType W;
  i=L.length;
  while(i >1) {  // i>1 表明上一趟曾进行过记录的交换
      lastExchangeIndex=1;
      for(j=1; j<i; j++){
        if(L.r[j+1].key<L.r[j].key) {
          W=L.r[j]; L.r[j]=L.r[j+1]; L.r[j+1]=W;     // 互换记录
          lastExchangeIndex=j;
        }//if
      }//for
      i=lastExchangeIndex;       // 一趟排序中无序序列中最后一个记录的位置
  }// while
}// BubbleSort
```

分析起泡排序的时间和空间效率，从以上讨论中可知，起泡排序和插入排序一样，对不同组的记录所需进行的关键字间的比较次数和记录的移动次数不同，最好的情况是，原始记录按关键字顺序有序排列，此时只需进行一趟起泡排序，则只进行 n－1 次关键字间的比较，且没有移动记录。反之，最坏的情况是，记录按关键字逆序有序排列，此时需进行 n－1 趟起泡，整个排序过程中进行的关键字间的比较次数为

$$\sum_{i=n}^{2}(i-1)=\frac{n(n-1)}{2}$$

记录的移动次数为

$$3\sum_{i=n}^{2}(i-1)=\frac{3n(n-1)}{2}$$

因此，起泡排序的时间复杂度为 $O(n^2)$。和选择排序类似，起泡排序的过程中也只需要一个辅助空间，故空间复杂度为 $O(1)$。

起泡排序也是稳定的排序方法。

3.3 先进排序方法

3.3.1 快速排序

快速排序(quick sort)是从起泡排序改进而得的一种“交换”排序方法。它的基本思想是通过一趟排序将待排记录分割成相邻的两个区域，其中一个区域中记录的关键字均比另一区域中记录的关键字小(区域内不见得有序)，则可分别对这两个区域的记录进行再排序，以达到整个序列有序。

假设待排序的原始记录序列为

$$(R_s,R_{s+1},\cdots,R_{t-1},R_t)$$

则一趟快速排序的基本操作是：任选一个记录(通常选记录 R_s)，以它的关键字作为“枢轴”，凡序列中关键字小于枢轴的记录均移动至该记录之前；反之，凡序列中关键字大于枢轴的记录均移动至该记录之后。致使一趟排序之后，记录的无序序列 R[$s..t$]将分割成两部分：R[$s..i-1$]和 R[$i+1..t$]，且使

```
R[j].key ≤ R[i].key ≤ R[j].key
(s≤j≤i-1) 枢轴 (i+1≤j≤t)
```

具体操作过程描述如下：假设枢轴记录的关键字为 pivotkey，附设两个指针 low 和 high，它们的初值分别为 s 和 t。首先将枢轴记录移至临时变量，之后检测指针 high 所指记录，若 R[high]. key≥pivotkey，则减小 high，否则将 R[high]移至指针 low 所指位置，之后检测指针 low 所指记录，若 R[low]. key≤pivotkey，则增加 low，否则将 R[low]移至指针 high 所指位置，重复进行上述两个方向的检测，直至 high 和 low 两个指针指向同一位置重合为止，如算法 3.6 所述。

算法 3.6

```
int Partition(RcdType R[], int low, int high)
{
  // 对记录子序列 R[low..high]进行一趟快速排序,并返回枢轴记录所在位置,
  // 使得在它之前的记录的关键字均不大于它的关键字,在它之后的记录的关键
  // 字均不小于它的关键字
  R[0]=R[low];                  // 将枢轴记录移至数组的闲置分量
  pivotkey=R[low].key;          // 枢轴记录关键字
  while(low<high) {             // 从表的两端交替地向中间扫描
    while(low<high && R[high].key>=pivotkey)
```

```
    --high;
  if(low<high)
    R[low++]=R[high];      // 将比枢轴记录小的记录移到低端
  while(low< high && R[low].key<=pivotkey)
    ++low;
  if(low<high)
    R[high--]=R[low];      // 将比枢轴记录大的记录移到高端
 } //while
 R[low]=R[0];              // 枢轴记录移到正确位置
 return low;               // 返回枢轴位置
} // Partition
```

例如：将关键字序列(49_1,38,65,97,76,13,27,49_2)调整为(27,38,13,(49_1),76,97,65,49_2)(其中(49_1)为枢轴记录的关键字)的过程如图 3.7 所示。

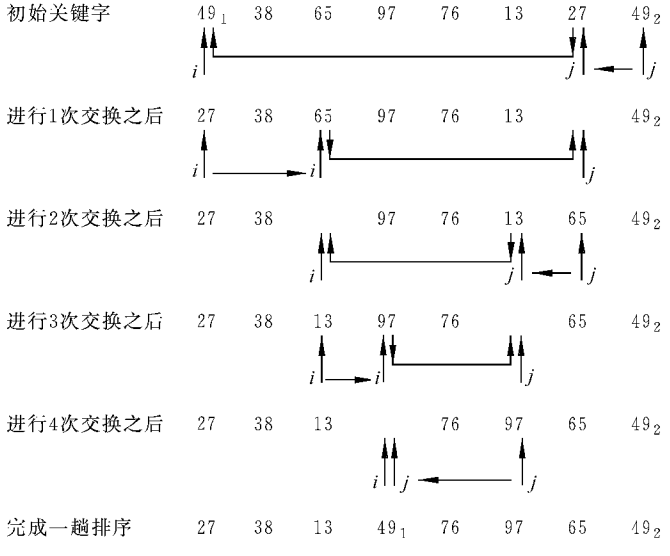


图 3.7　一趟快速排序过程示例

一趟快速排序的过程又称“一次划分”。对枢轴两侧的左右区域继续如法炮制，即整个快速排序的过程可递归进行。若待排的原始记录序列中只有一个记录，则显然已有序，不再需要进行排序；否则首先对该记录序列进行“**一次划分**”，之后**分别**对分割所得两个子序列“递归”进行快速排序，如图 3.8 所示。快速排序的算法如算法 3.7 所示。

初始状态	{49_1	38	65	97	76	13	27	49_2}
一次划分之后	{27	38	13}	49_1	{76	97	65	49_2}
分别进行快排序	{13}	27	{38}					
	结束		结束		{49_2	65}	76	{97}
					49_2	{65}		结束
						结束		
有序序列	(13	27	38	49_1	49_2	65	76	97)

图 3.8　快速排序过程示例

算法 3.7

```
void QSort(RcdType R[], int s, int t)
{
  // 对记录序列 R[s..t]进行快速排序
  if(s<t) {                          // 长度大于 1
    pivotloc=Partition(R, s, t);     // 对 R[s..t] 进行一次划分,并返回枢轴位置
    QSort(R, s, pivotloc-1);         // 对低端子序列递归排序
    QSort(R, pivotloc+1, t);         // 对高端子序列递归排序
  } // if
} // Qsort
```

算法 3.7 中使用了一对参数 s 和 t 作为待排序区域的上下界。在算法的递归调用过程执行中,这两个参数随着“区域的划分”而不断变化。在对顺序表 L 进行快速排序调用算法 3.7 时,s 和 t 的初值应分别置为 1 和 L.length,如算法 3.8 所示。

算法 3.8

```
void QuickSort(SqList &L)
{
  // 对顺序表 L 进行快速排序
   QSort(L.r, 1, L.length);
} // QuickSort
```

快速排序在一般情况下是效率很高的排序方法。可推导证得,快速排序的平均时间复杂度为 $O(n\log n)$。快速排序目前被认为是同数量级($O(n\log n)$)中最快的内部排序方法,这是由于对区域不断“一分为二”所带来的效益,但这仅就平均性能而言。如果待排序的原始记录序列已按关键字有序或“基本有序”排列时,快速排序的时间复杂度将蜕化为 $O(n^2)$,因为在这种情况下经常会发生这样的情况,即长度为 n 的记录序列经一次划分后得到的两个子序列的长度分别为 0 和 $n-1$,也就是说未能进行“一分为二”的划分,从而失去了快速排序的优势。为避免出现“蜕化”情形,通常依“三者取中”的规则选取枢轴记录,即对 R[s].key、R[t].key 和 R[(s+t)/2].key 三者进行比较,以它们中取“中值”的记录为枢轴记录。只要将它和 R[s]互换,之后仍然可按算法 3.6 进行一次划分。经验证,采用三者取中规则可以大大改善快速排序在最坏情况下的性能。然而,即使如此,也不能使快速排序在待排记录序列已经有序的情况下达到和起泡排序相同的时间复杂度为 $O(n)$ 的结果。快速排序的空间复杂度在一般情况下为 $O(\log n)$,最坏情况下为 $O(n^2)$。

快速排序是不稳定的排序方法。

3.3.2 归并排序

归并排序(merge sort)是利用“归并”操作的一种排序方法。从 2.4 节有序表的讨论中得知,将两个有序表“归并”为一个有序表,无论是顺序表还是链表,归并操作都可以在线性时间复杂度内实现。归并排序的基本操作是将两个位置相邻的有序记录子序列