

第 9 章 排序算法

教学要点

有序的数据便于处理，例如字典和电话簿一般都会按某种顺序排列其数据以便于使用。排序是将某种数据结构按照其数据元素的关键字值的大小以递增或递减的次序排列的过程，它在计算机数据处理中有着广泛的应用。

本章介绍排序的基本概念，讨论多种经典排序算法，包括插入、交换、选择和归并等排序算法，并比较各种排序算法的运行效率。

本章在 Visual Studio 中用名为 sort 的类库型项目实现有关数据结构与算法的定义，用名为 sorttest 的应用程序型项目实现测试和演示程序。

建议本章授课 6 学时，实验 4 学时。

9.1 数据序列及其排序

9.1.1 排序的基本概念

1. 数据序列、关键字和排序

数据序列（data series）是特定数据结构中的一系列数据元素，它是待加工处理的数据元素的有限集合。以学生信息系统为例，每个学生的信息是待处理的数据元素，若干相关学生的信息则组成一个待加工处理的数据序列，例如，某个院系同年级的学生成绩表。

数据序列的排序（sort）建立在数据元素间的比较操作基础之上。一个数据元素可由多个数据项组成，以数据元素的某个数据项作为比较和排序的依据，则该数据项称为排序关键字（sort key）。例如，学生信息由学号、专业、姓名、成绩等多个数据项组成。如果按学号进行排序，则学号就是排序关键字；如果按成绩排序，则成绩成为排序关键字。在数据序列中，如果某一关键字能唯一地标识一个数据元素，则称这样的关键字为主关键字（primary key）。不同数据元素，其主关键字的值也不同，因此用主关键字进行排序会得到唯一确定的结果。例如，在一所学校，学生的学号是唯一的，可以作为学生信息排序的主关键字，而学生的姓名则可能有重复，姓名不是主关键字，依据姓名排序的结果可能不是唯一的。

排序是将某种数据结构或数据序列按其数据元素的关键字的值以递增或递减的次序排列的过程，它在计算机数据处理中有着广泛的应用。

2. 内排序与外排序

根据被处理的数据规模大小，排序过程中涉及的存储器类型可能不同，由此，排序问题一般可分为内排序和外排序两大类：

- 内排序：如果待排序的数据序列中的数据元素个数较少，在整个排序过程中，所有的数据元素及中间数据可以同时保留在内存中。
- 外排序：待排序的数据元素非常多，它们必须存储在磁盘等外部存储介质上，在整个排序过程中，需要多次访问外存逐步完成数据的排序。

显然，内排序是基础，外排序建立在内排序的基础之上，但增加了一些复杂性。

3. 排序算法的性能评价

像评价其他算法一样，对排序算法的性能也是从算法的时间复杂度和空间复杂度两个方面进行评价的。

- 时间复杂度：排序算法包含的基本操作的重复执行次数与待排序的数据序列长度之间的关系。数据排序的基本操作是数据元素的比较与移动操作，分析某个排序算法的时间复杂度，就是要确定该算法在执行过程中，数据元素比较次数和数据元素移动次数与待排序的数据序列长度之间的关系。
- 空间复杂度：排序算法所需内存空间与待排序的数据序列长度之间的关系。数据的排序过程需要一定的内存空间才能完成，这包括待排序数据序列本身所占用的内存空间，以及其他附加的内存空间。分析某个排序算法的空间复杂度，就是要确定该算法在执行过程中，所需附加的内存空间与待排序数据序列的长度之间的关系。

一个好的排序算法应该具有相对低的时间复杂度和空间复杂度，即算法应该尽可能减少运行时间，占用较少的额外空间。

4. 排序算法的稳定性

用主关键字进行排序会得到唯一的结果，而用可能有重复的非主关键字进行排序，结果不是唯一的。假设，在数据序列中有两个数据元素 r_i 和 r_j ，它们的关键字 k_i 等于 k_j ，且在未排序时， r_i 位于 r_j 之前。如果排序后，元素 r_i 仍在 r_j 之前，则称这样的排序算法是稳定的排序（stable sort）。如果排序后，元素 r_i 和 r_j 的顺序可能保持不变也可能会发生改变，这样的排序称为不稳定的排序（unstable sort）。

本章主要讨论几种经典的内排序算法。为了将注意力集中在算法的本质，如不作特别说明，本章假设待排序的数据序列保存在一个数组中，并假定每个数据元素只含关键字，排序一般都是按关键字值非递减的次序对数据进行排列。关键字为某种可比较的类型，如 C# 编程语言中的 `int`、`double`、`string` 等类型，以及实现了 `IComparable` 接口的各种自定义类型。

设计一个称为 `Sort` 的类，在其中以静态方法的形式实现各种排序算法。它们具有如下形式：

```
public class Sort<T> where T: IComparable {  
    public static void InsertSort(T[] items);  
    public static void ShellSort(T[] items);  
    public static void BubbleSort(T[] items);  
    public static void QuickSort(T[] items, int nLower, int nUpper);  
    public static void SelectSort(T[] items);  
    public static void HeapSort(T[] items);  
    public static void MergeSort(T[] items);  
}
```

```
}
```

9.1.2 C#数组的排序操作

.NET Framework 的类库中定义的 `System.Array` 类是 C# 程序中各种数组的基类。`Array` 类提供了许多用于排序、查找和复制数组的方法，其中排序功能以具有多种重载形式的 `Sort` 方法提供。它们使用 `QuickSort` 算法进行排序，该排序算法属于不稳定排序，亦即，如果两元素相等，则其原顺序在排序后可能会发生改变。

`Array` 类的各个 `Sort` 方法都是静态方法，它们分别具有下列形式：

1) `public static void Sort (Array ar);`

参数 `ar` 为待排序的一维数组。这个数组的元素类型要实现接口 `IComparable`，该实现定义了元素间的比较协议，据此对整个一维数组中的元素进行排序。

2) `public static void Sort (Array ar, int index, int length);`

参数 `ar` 为待排序的一维数组，参数 `index` 为排序范围的起始索引，参数 `length` 为排序范围内的元素数。

3) `public static void Sort (Array ar, IComparer comparer);`

参数 `ar` 为待排序的一维数组，参数 `comparer` 为比较元素时要使用的“比较器”对象，它实现 `IComparer` 接口（interface），在该接口规定的 `Compare` 方法中决定数据元素之间的比较规则。

下面的例子演示 `Array` 类的 `Sort` 方法和有关的类（如 `Array`）与接口（如 `IComparable`, `IComparer`）的使用方法。

【例9.1】 学生信息类型的定义与学生信息表的排序演示。

学生信息类型 `StudentInfo` 定义如下所示，`StudentInfo` 类型的对象之间是可以比较的，缺省的比较规则是通过实现由 `IComparable` 接口定义的 `CompareTo()` 方法来确定的，在这里学生信息间的比较等价于比较学生的学号。

```
public class StudentInfo: IComparable{
    private int studentID;
    private string name;
    private double score;

    public StudentInfo(int id, string name, double score ) {
        this.studentID = id;
        this.name = name;
        this.score = score;
    }

    public int StudentID {
        get { return studentID; }
        set { studentID = value; }
    }

    public string Name {
        get { return name; }
        set { name = value; }
    }
}
```

```

    }

    public double Score{
        get{ return score;}
        set{ score = value;}
    }

    public int CompareTo(object obj) {
        if(obj is StudentInfo) {
            StudentInfo di = (StudentInfo)obj;
            return this.studentID.CompareTo(di.StudentID);
        }
        throw new ArgumentException(String.Format("object is not a StudentInfo"));
    }
}

```

在程序中可以直接调用 `Array.Sort` 方法，以按照学生信息类 `StudentInfo` 定义的缺省比较方式完成学生信息表的排序，`StudentInfo` 类型的对象的缺省比较方式定义为按学生的学号来比较。如果要按其他方式比较并排序 `StudentInfo` 类型的对象，则调用需要两个参数的 `Array.Sort` 方法来完成，其中第二个参数指定一个实现了 `IComparer` 接口的“比较器”对象，它定义所需的比较规则。

下面定义的 `StudentComparer` 类，它在 `IComparer` 接口所规定的 `Compare` 方法中根据私有数据成员 `key` 的值决定 `StudentInfo` 对象之间的比较规则，而 `key` 的值由该“比较器”类的构造方法设置。

```

public enum CompareKey { ID, Name, Score, IDD, NameD, ScoreD }

class StudentComparer: IComparer{
    private CompareKey key;
    public CompareKey Key {
        get{return key;}
        set{key = value;}
    }
    public StudentComparer(CompareKey k) {
        key = k;
    }
    int IComparer.Compare( Object x, Object y ) {
        StudentInfo X = (StudentInfo)x;
        StudentInfo Y = (StudentInfo)y;
        switch (key) {
            case CompareKey.Name:
                return (X.Name.CompareTo(Y.Name));
            case CompareKey.NameD:
                return (Y.Name.CompareTo(X.Name));
            case CompareKey.Score:

```

```

        return (X.Score.CompareTo(Y.Score));
    case CompareKey.ScoreD:
        return (Y.Score.CompareTo(X.Score));
    case CompareKey.IDD:
        return (Y.StudentID.CompareTo(X.StudentID));
    default:
        return X.StudentID.CompareTo(Y.StudentID);
    }
}
}

```

下面的程序分别按学号、成绩和姓名对学生信息表进行排序。

```

using System;
public class ArraySortTest {
    static void Main(string[] args) {
        StudentInfo[] items = new StudentInfo[5];
        SetData(items);
        Show(items);
        Console.WriteLine("按学号排序");
        Array.Sort(items);
        Show(items);
        Console.WriteLine("按成绩排序（从高到低）");
        Array.Sort(items, new StudentComparer(CompareKey.ScoreD));
        Show(items);
        Console.WriteLine("按姓名排序");
        Array.Sort(items, new StudentComparer(CompareKey.Name));
        Show(items);
    }

    static void Show(StudentInfo[] items) {
        Console.WriteLine("学号\t姓名\t成绩");
        for(int j=0; j<items.Length; j++) {
            Console.WriteLine(items[j].StudentID+"\t"+items[j].Name+"\t"+items[j].Score);
        }
    }

    static void SetData(StudentInfo[] items) {
        items[0] = new StudentInfo(3016, "张超", 89);
        items[1] = new StudentInfo(3053, "马飞", 80);
        items[2] = new StudentInfo(3041, "刘羽", 96);
        items[3] = new StudentInfo(3025, "赵备", 79);
        items[4] = new StudentInfo(3039, "关云", 85);
    }
}

```

运行这个程序，将显示学生信息表的各种排序结果。

9.2 插入排序

插入排序（insertion sort）基于一个简单的基本思想：将待排序的数据序列依次有序地插入成一个有序的数据序列。该算法将整个数据序列视为由两个子序列组成：处于前面的已排序的子序列和处于后面的待排序的子序列；在排序过程中，分趟将待排序的数据元素，按其关键字值的大小插入到前面已排序的子序列中，从而得到一个新的、元素个数增 1 的有序序列，重复该过程直到全部元素插入完毕。下面介绍直接插入排序算法和希尔排序算法。

9.2.1 直接插入排序

1. 算法的基本思想

直接插入排序（straight insertion sort）分趟将待排序的数据依次有序地插入成一个有序的数据序列，其核心思想是：在第 m 趟插入第 m 个数据元素 k 时，前 $m-1$ 个数据元素已组成有序数据序列 S_{m-1} ，将 k 与 S_{m-1} 中各数据元素依次进行比较并插入到适当位置，得到新的序列 S_m 仍是有序的。

设有一个待排序的数据序列为 $items=\{36, 91, 31, 26, 61\}$ ，直接插入排序算法的执行过程如下：

- 1) 初始化：以 $items[0]=36$ 建立有序子序列 $S_0=\{36\}$ ， $m=1$ 。
- 2) 在第 m 趟，欲插入元素值 $k=items[m]$ ，在 S_{m-1} 中进行顺序查找，找到 k 值应插入的位置 i ；从序列 S_{m-1} 末尾开始到 i 位置的元素依次向后移动一位，空出位置 i ；将 k 置入 $items[i]$ ，得到有序子序列 S_m ， $m++$ 。例如，当 $m=1$ 时， $k=91$ ， $i=1$ ， $S_1=\{36, 91\}$ 。当 $m=2$ 时， $k=31$ ， $i=0$ ， $S_2=\{31, 36, 91\}$
- 3) 重复步骤 2，依次将其他数据元素插入到已排序的子序列中。

图 9.1 显示了对上述数据序列的直接插入排序过程。

2. 数组的直接插入排序算法实现

```
public static void InsertSort(T[] items) {
    T k;
    int i, j, m;
    int n = items.Length;
    for (m=1; m<n; m++) {
        k = items[m];
        for (i=0; i<m; i++) {
            if (k.CompareTo(items[i])<0) {
                for (j=m; j>i; j--)
                    items[j] = items[j-1];
```

```

        items[i] = k;
        break;
    }
}
Show(m, items);
}
}

```


index	0	1	2	3	4
items	36				

(a) $m=0$, 插入36, 得到 $S_0=\{36\}$

index	0	1	2	3	4
items	36	91			


(b) $m=1$, 在 S_0 中查找 $k=\text{items}[m]=91$ 的插入位置 $i=1$, 插入 k 得到 $S_1=\{36,91\}$

index	0	1	2	3	4
items	31	36	91		




(c) $m=2$, 在 S_1 中查找 $k=\text{items}[m]=31$ 的插入位置 $i=0$, 插入 k 得到 $S_2=\{31,36,91\}$

index	0	1	2	3	4
items	26	31	36	91	



(d) $m=3$, 在 S_2 中查找 $k=\text{items}[m]=26$ 的插入位置 $i=0$, 插入 k 得到 $S_3=\{26,31,36,91\}$

index	0	1	2	3	4
items	26	31	36	61	91



(e) $m=4$, 在 S_3 中查找 $k=\text{items}[m]=61$ 的插入位置 $i=3$, 插入 k 得到 $S_4=\{26,31,36,61,91\}$

图 9.1 序列的直接插入排序过程描述

Sort 类中还定义了名为 Show 的帮助方法, 用以显示当前排序趟数及数据序列的值。

```

public static void Show(int i, T[] items){
    if(i==0)
        Console.WriteLine("数据序列: ");
    else
        Console.WriteLine("第" + i + "趟排序后: ");
    for(int j=0; j<items.Length; j++)
        Console.Write(items[j] + " ");
}

```

```

        Console.WriteLine();
    }

```

【例9.2】 整型数组的直接插入排序算法测试。

```

static void Main(string[] args) {
    int[] items = { 36, 91, 31, 26, 61, 37};
    Sort<int>.Show(0, items);
    Sort<int>.InsertSort(items);
    Console.Write("排序后");
    Sort<int>.Show(0, items);
}

```

程序运行结果如下：

```

数据序列:   36 91 31 26 61 37
第1趟排序后: 36 91 31 26 61 37
第2趟排序后: 31 36 91 26 61 37
第3趟排序后: 26 31 36 91 61 37
第4趟排序后: 26 31 36 61 91 37
第5趟排序后: 26 31 36 37 61 91
排序后数据序列: 26 31 36 37 61 91

```

3. 算法分析

数据排序的基本操作是数据元素的比较与移动，下面来分析在直接插入排序算法中，数据元素的比较次数和数据元素的移动次数与待排序数据序列的长度之间的关系。由第8章查找算法可知，在具有 m 个数据元素的有序线性表中顺序查找一个数据元素的平均比较次数为 $(m+1)/2$ 。所以，直接插入排序过程中的平均比较次数为：

$$C = \sum_{m=1}^{n-1} \frac{m+1}{2} = \frac{1}{4}n^2 + \frac{1}{4}n - \frac{1}{2} \approx \frac{n^2}{4}$$

由第2章线性表可知，在长度为 m 的数据序列中，在等概率条件下，插入一个数据元素的平均移动次数是 $m/2$ ，即需要移动序列全部数据元素的一半。所以，直接插入排序过程中的平均移动次数为：

$$M = \sum_{m=1}^{n-1} \frac{m}{2} = \frac{n(n-1)}{4} \approx \frac{n^2}{4}$$

由以上两个方面的分析可知，直接插入排序算法的时间复杂度为 $O(n^2)$ 。

直接插入排序过程只需几个辅助变量，其存储空间的大小与序列的长度无关，所以算法的空间复杂度为 $O(1)$ 。

很明显，对于关键字相同的元素，直接插入排序不会改变它们原有的次序。所以，直接插入排

序算法是稳定的排序。

直接插入排序算法在每趟的插入过程中，要首先用查找算法在有序子表中确定待排序元素应插入的位置。可以用二分查找算法代替顺序查找算法完成在有序表中查找的工作，这样可以降低平均比较次数。但是，用二分查找算法代替顺序查找算法并不能减少移动数据元素操作的次数，故算法的总体时间复杂度仍为 $O(n^2)$ 。

改进后的算法实现如下所示：

```
public static void InsertSortBS(T[] items) {
    T k;
    int i, j, m, n = items.Length;
    for (m = 1; m < n; m++) {
        k = items[m];
        i = Array.BinarySearch<T>(items, 0, m, k);
        if (i < 0)
            i = ~i;
        else {
            while (k.Equals(items[i])) i++;
        }
        for (j = m; j > i; j--)
            items[j] = items[j - 1];
        items[i] = k;
        Show(m, items);
    }
}
```

9.2.2 希尔排序

1. 算法的基本思想

希尔排序（Shell sort）又称缩小增量排序（diminishing increment sort），它也属于插入排序类的方法。它的基本思想是：先将整个序列分割成若干子序列分别进行排序，待整个序列基本有序时，再进行全序列直接插入排序，这样可使排序过程加快。

直接插入排序每次比较的是相邻的数据元素，一趟排序后数据元素最多移动一个位置。如果待排序序列的数据元素个数为 n ，假定序列中第 1 个数据元素的关键字值最大，排序后的最终位置应该是序列的最后一个单元，则将它从序列头部移动到序列尾部需要运行 $n-1$ 步。如果有某种办法将该元素一次移动到尾部或尽可能靠近尾部，那么排序的速度就可能快得多。希尔排序算法在排序之初，将位置相隔较远的若干数据元素归为一个子序列，因而进行相互比较的是位置相隔较远的数据元素，这就使得数据元素移动时能够跨越多个位置；然后逐渐减少被比较数据元素间的距离（缩小增量），直至距离为 1 时，各数据元素都已按序排好。

2. 数组的希尔排序算法实现

```
public static void ShellSort(T[] items) {  
    T t;  
    int n = items.Length, jump = n / 2;  
    int i, j, m = 1;  
    while (jump > 0) {  
        for (i = jump; i < n; i++) {  
            j = i - jump;  
            while (j >= 0) {  
                if (items[j].CompareTo(items[j + jump]) > 0) {  
                    t = items[j];  
                    items[j] = items[j + jump];  
                    items[j + jump] = t;  
                    j -= jump;  
                }  
                else  
                    j = -1;  
            }  
        }  
        Console.WriteLine("jump=" + jump + " ");  
        Show(m, items); m++;  
        jump /= 2;  
    }  
}
```

在希尔排序算法的代码中有三重循环：

- 最外层循环（while 语句）：控制增量 **jump**，其初值为数组长度 n 的一半，以后逐次减半缩小，直至增量为 1。整个序列分割成 **jump** 个子序列，分别进行直接插入排序。
- 中层循环（for 语句）：相隔 **jump** 的元素进行比较、交换，完成一轮子序列的直接插入排序。
- 最内层循环（while 语句）：将元素 **items[j]** 与相隔 **jump** 的元素 **items[j+jump]** 进行比较，如果两者是反序的，则执行交换。重复往前（**j -= jump**）与相隔 **jump** 的元素再比较、交换；当 **items[j] ≤ items[j+jump]** 时，表示元素 **items[j]** 已在这趟排序后的位置，不需交换，则退出最内层循环。

例如，对于一个待排序的数据序列 **items={36, 91, 31, 26, 61, 37, 97, 1, 93, 71}**，数据序列长度 **n=10**，初始增量 **jump=n/2**。希尔排序的执行过程如下所叙：

- 1) **jump=5**，**j** 从第 0 个位置元素开始，将相隔 **jump** 的元素 **items[j]** 与元素 **items[j+jump]** 进行比较。如果反序，则交换，依次重复进行完一趟排序，得到序列 {36, 91, 1, 26, 61, 37, 97, 31, 93, 71}。

- 2) $\text{jump}=2$ ，相隔 jump 的元素组成子序列 {36, 1, 61, 97, 93} 和子序列 {91, 26, 37, 31, 71}。在子序列内比较元素 $\text{items}[j]$ 与元素 $\text{items}[j+\text{jump}]$ ，如果反序，则交换，依次重复。得到序列 {1, 26, 36, 31, 61, 37, 93, 71, 97, 91}。
- 3) $\text{jump}=1$ ，在全序列内比较元素 $\text{items}[j]$ 与元素 $\text{items}[j+\text{jump}]$ ，如果反序，则交换；得到序列 {1, 26, 31, 36, 37, 61, 71, 91, 93, 97}。

【例9.3】 整型数组的希尔排序算法测试。

```
int[] items = {36, 91, 31, 26, 61, 37, 97, 1, 93, 71};  
Sort<int>.ShellSort(items);  
Console.WriteLine("排序后");  
Sort<int>.Show(0, items);
```

程序运行结果如下：

```
数据序列: 36 91 31 26 61 37 97 1 93 71  
jump=5 第1趟排序后: 36 91 1 26 61 37 97 31 93 71  
jump=2 第2趟排序后: 1 26 36 31 61 37 93 71 97 91  
jump=1 第3趟排序后: 1 26 31 36 37 61 71 91 93 97  
排序后数据序列: 1 26 31 36 37 61 71 91 93 97
```

3. 算法分析

希尔排序算法的时间复杂度分析比较复杂，实际所需的时间取决于每次排序时增量的取值。研究证明，若增量的取值比较合理，希尔排序算法的时间复杂度为约 $O(n(\log_2 n)^2)$ 。希尔排序算法的空间复杂度为 $O(1)$ 。希尔排序算法是一种不稳定的排序算法，对于关键字相同的元素，排序可能会改变它们原有的次序。

9.3 交换排序

在基于交换的排序算法中有两个算法非常经典：冒泡排序（bubble sort）和快速排序（quick sort）。冒泡排序是一种直接的交换排序算法；快速排序是目前平均性能较好的一种排序算法，.NET Framework 的 System.Array 类的 Sort 方法使用 quick sort 算法进行排序。

9.3.1 冒泡排序

1. 冒泡排序基本思想

冒泡排序算法的基本思想简单直接，它依次比较相邻的两个数据元素的关键字值，如果反序，则交换它们的位置。对于一个待排序的数据序列，经过一趟交换排序后，具有最大值的数据元素将移到序列的最后位置，值较小的数据元素向最终位置移动一位，这一趟交换过程又称为一趟起泡。如果在一趟排序中，没有发生一次数据交换（起泡），则说明序列已排好序。

对于有 n 个数据元素的数据序列，最多需 $n-1$ 趟排序，第 m 趟对从位置 0 到位置 $n-m-1$ 的数据元素与其后一位的元素进行比较、交换，如果该趟没有发生一次数据的交换，则整个序列的排序过程结束。因此冒泡排序算法可用二重循环实现。

2. 数组的冒泡排序算法实现

```
public static void BubbleSort(T[] items) {
    T t; int n = items.Length;
    bool exchanged;
    for (int m = 1; m < n; m++) {
        exchanged = false;
        for (int j = 0; j < n - m; j++) {
            if (items[j].CompareTo(items[j + 1]) > 0) {
                t = items[j];
                items[j] = items[j + 1];
                items[j + 1] = t;
                exchanged = true;
            }
        }
        Show(m, items);
        if (!exchanged)
            break;
    }
}
```

假设有一个待排序的数据序列为 $\text{items}=\{36, 91, 31, 26, 61, 37\}$ ，在该序列上进行冒泡排序的过程如图 9.2 所示。

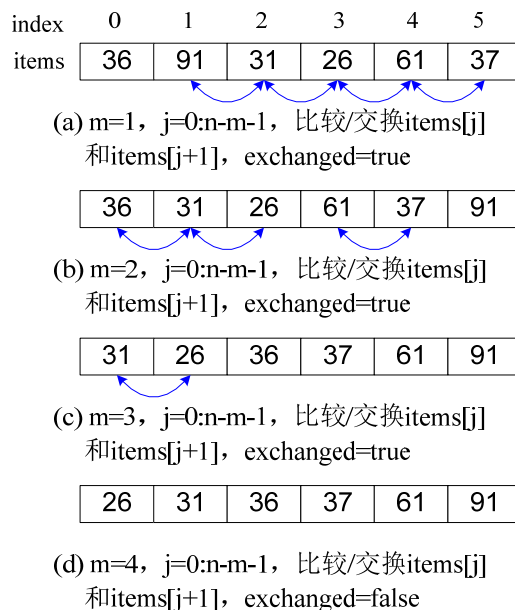


图 9.2 序列的冒泡排序过程描述

算法测试的程序运行结果如下：

```
第1趟排序后: 36 31 26 61 37 91
第2趟排序后: 31 26 36 37 61 91
第3趟排序后: 26 31 36 37 61 91
第4趟排序后: 26 31 36 37 61 91
```

3. 算法分析

BubbleSort 方法用两重循环分趟实现交换排序算法。外循环控制排序趟数，在最好的情况下，序列已排序，只需一趟排序即可，进行比较操作的次数为 $n-1$ ，进行移动操作的次数为 0，算法的时间复杂度为 $O(n)$ ；最坏的情况是序列已按反序排列，这时需要 $n-1$ 趟排序，每趟过程中进行比较和移动操作的次数均为 $n-1$ ，算法的时间复杂度为 $O(n^2)$ 。

就平均情况而言，冒泡排序算法的时间复杂度为 $O(n^2)$ 。

在冒泡排序过程中，需要一个辅助存储空间来交换两个数据元素，这与序列的长度无关，故冒泡排序算法的空间复杂度为 $O(1)$ 。

从交换的过程易看出，对于关键字相同的元素，排序不会改变它们原有的次序，故冒泡排序是稳定的。

9.3.2 快速排序

1. 快速排序算法

快速排序的基本思想是：将长序列以其中的某值为基准（这个值称作枢纽 **pivot**）分成两个独立的子序列，第一个子序列中的所有元素的关键字值均比 **pivot** 小，第二个子序列所有元素的关键字值则均比 **pivot** 大；再以相同的方法分别对两个子序列继续进行排序，直到整个序列有序。具体做法是，在待排序的数据序列中任意选择一个元素（如第一个元素）作为基准值 **pivot**，由序列的两端交替地向中间进行比较、交换，使得所有比 **pivot** 小的元素都交换到序列的左端，所有比 **pivot** 大的元素都交换到序列的右端，这样序列就被划分成三部分：左子序列，**pivot** 和右子序列。再对两个子序列分别进行同样的操作，直到子序列的长度为1。每趟排序过程中，将找到基准值 **pivot** 在序列中的最终排序位置，并据此将原序列分成两个小序列。

2. 数组的快速排序算法实现

```
public static void QuickSort(T[] items, int nLower, int nUpper) {
    if (nLower < nUpper) {
        int nSplit = Partition(items, nLower, nUpper);
        Console.WriteLine("left=" + nLower + " right=" + nUpper + " Pivot=" + nSplit + "\t");
        Show(0, items);
        QuickSort(items, nLower, nSplit - 1);
        QuickSort(items, nSplit + 1, nUpper);
    }
}
```

```

private static int Partition(T[] items, int nLower, int nUpper) {
    T t, pivot = items[nLower];
    int nLeft = nLower + 1;
    int nRight = nUpper;
    while (nLeft <= nRight) {
        while (nLeft <= nRight && (items[nLeft]).CompareTo(pivot) <= 0)
            nLeft = nLeft + 1;
        while (nLeft <= nRight && (items[nRight]).CompareTo(pivot) > 0)
            nRight = nRight - 1;
        if (nLeft < nRight) {
            t = items[nLeft];
            items[nLeft] = items[nRight];
            items[nRight] = t;
            nLeft = nLeft + 1;
            nRight = nRight - 1;
        }
    }
    t = items[nLower];
    items[nLower] = items[nRight];
    items[nRight] = t;
    return nRight;
}

```

QuickSort 方法以递归方式实现快速排序算法。设 `nLower` 和 `nUpper` 分别表示待排序的子序列的左右边界，`Partition` 方法选取子序列的第一个元素为基准值 `pivot` 进行一趟排序，将作为基准值的元素交换到它在最终完全排好序的序列中的应有位置，并将该位置值作为方法的返回值返回到调用它的 `QuickSort` 方法中，并记录在变量 `nSplit` 中。这样经一趟排序后，原序列分为两个子序列，分别为 `[nLower, nSplit - 1]` 和 `[nSplit + 1, nUpper]`。对两个子序列再分别调用 `QuickSort` 方法进行递归排序。

在 `Partition` 方法中，选取子序列的第一个元素作为基准值 `pivot`，开始时 `nLeft` 和 `nRight` 分别表示子序列除基准值外的第一个元素和最后一个元素的位置，`while (nLeft <= nRight)` 循环进行一轮比较，`nLeft`，`nRight` 分别从序列的最左、右端开始向中间扫描。在左端发现大于 `pivot` 或右端发现小于 `pivot` 的元素，则交换到另一端，并收缩两端的范围，最终确定基准值 `pivot` 应有的最终排序位置。最后将 `pivot` 交换到该位置，并将该位置值作为方法的结果返回。

假设有一个待排序的数据序列为 `items={36, 91, 31, 26, 61, 37}`，在这个序列上进行快速排序的过程描述如图 9.3 所示。算法测试的程序运行结果如下：

```

left=0 right=5 Pivot=2 数据序列: 31 26 36 91 61 37
left=0 right=1 Pivot=1 数据序列: 26 31 36 91 61 37
left=3 right=5 Pivot=5 数据序列: 26 31 36 37 61 91
left=3 right=4 Pivot=3 数据序列: 26 31 36 37 61 91
排序后数据序列: 26 31 36 37 61 91

```

3. 算法分析

快速排序的执行时间与序列的初始排列及基准值的选取有关。最坏情况是，当序列已排序时，例如，对于序列 {1, 2, 3, 4, 5, 6, 7, 8}，如果选取序列的第一个元素作为基准值，那么分成的两个子序列将分别是 {1} 和 {2, 3, 4, 5, 6, 7, 8}，而且它们仍然是已排序的。这样必须经过 $n-1=7$ 趟才能完成最终的排序。在这种情况下，其时间复杂度为 $O(n^2)$ ，排序速度已退化，比冒泡排序法还慢。一般而言，对于接近已排序的数据序列，快速排序算法的时间效率并不理想。

快速排序的最好情况是，每趟排序将序列分成两个长度相同的子序列。

研究证明，当 n 较大时，对平均情况而言，快速排序名符其实，其时间复杂度为 $O(n\log_2n)$ 。但当 n 很小时，或基准值选取不适当时，快速排序的时间复杂度可能退化为 $O(n^2)$ 。在算法实现中，常常以随机方法在待排序的数据序列中选择一个元素作为初始基准值，而不是固定选第一个元素。

快速排序是递归过程，需要在系统栈中传递递归函数的参数及返回地址，算法的空间复杂度为 $O(\log_2n)$ 。

快速排序算法是不稳定排序算法。对于关键字相同的元素，排序可能会改变它们原有的次序。

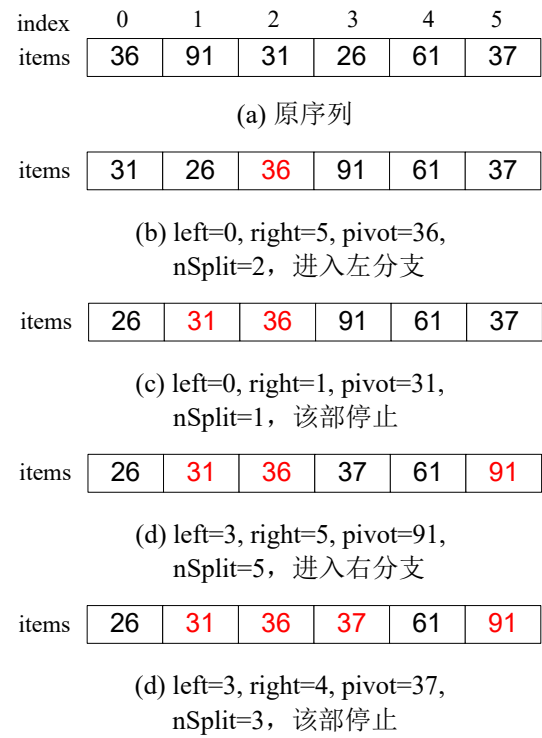


图 9.3 序列的快速排序过程描述

9.4 选择排序

选择排序算法常用的有两种：直接选择排序（straight selection sort）和堆排序（heap sort）。

9.4.1 直接选择排序

1. 直接选择排序算法

直接选择排序算法的基本思想是依次选择出待排序数据中的最小者使其排列有序，具体过程是：对于有 n 个元素的待排序数据序列 `items`，第 1 趟排序，比较 n 个元素，找到关键字最小的元素 `items[min]`，将其交换到序列的首位置 `items[0]`；第 2 趟排序，在余下的 $n-1$ 个元素中选取最小的元素，交换到序列的 `items[1]` 位置；这样经过 $n-1$ 趟排序，完成 n 个元素的排序。

2. 数组的直接选择排序算法实现

```
public static void SelectSort(T[] items) {
    T t;
    int min, n = items.Length;
    for (int m = 1; m < n; m++) {
        min = m - 1;
        for (int j = m; j < n; j++) {
            if (items[j].CompareTo(items[min]) < 0)
                min = j;
        }
        if (min != m - 1) {
            t = items[m - 1];
            items[m - 1] = items[min];
            items[min] = t;
        }
        Console.WriteLine("min=" + min + " ");
        Show(m, items);
    }
}
```

`SelectSort` 方法用一个二重循环实现直接选择排序。外层 `for` 循环控制 $m=1:n-1$ 趟排序，每趟排序找到一个最小值置于 `items[m-1]`；内层 `for` 循环控制 $j=m:n-1$ 在序列剩余的数据元素中进行每趟的比较，找到关键字最小的元素 `items[min]`，然后与 `items[m-1]` 交换。

假设有一个待排序的数据序列为 `items={36, 91, 31, 26, 61}`，在其上进行直接选择排序的过程描述如图 9.4 所示。算法测试的程序运行结果如下：

```
min=3 第趟排序后: 26 91 31 36 61
min=2 第趟排序后: 26 31 91 36 61
min=3 第趟排序后: 26 31 36 91 61
min=4 第趟排序后: 26 31 36 61 91
排序后数据序列:  26 31 36 61 91
```

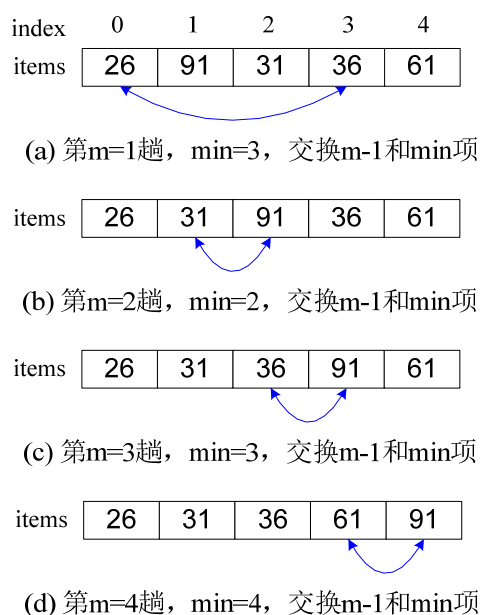



图 9.4 序列的选择排序过程描述

3. 算法分析

在直接选择排序算法中, 比较操作的次数与数据序列的初始排列无关。对于有 n 个数据元素的待排序数据序列, 在第 m 趟排序中, 查找最小值所需的比较次数为 $n - m$ 次。所以, 直接选择排序算法总的比较次数为

$$C = \sum_{m=1}^{n-1} (n - m) = \frac{1}{2} n(n - 1) \approx \frac{n^2}{2}$$

序列的初始排列对算法执行中数据元素的移动次数是有影响的。最好的情况是, 数据序列的初始状态已是按数据元素的关键字值递增排列的, 那么算法执行中无需移动元素, 数据移动操作的次数 $M = 0$ 。最坏情况是, 每一趟排序过程都要交换数据元素的位置, 此时总的的数据元素移动次数为 $M = 3 \times (n - 1)$ 。所以, 直接选择排序算法的时间复杂度为 $O(n^2)$ 。

在直接选择过程中, 需要一个辅助存储空间来交换两个数据元素, 这与序列的长度无关, 故算法的空间复杂度为 $O(1)$ 。

直接选择排序算法是稳定的。对于关键字相同的元素, 排序不会改变它们原有的次序。

9.4.2 堆排序

1. 堆排序算法

直接选择排序算法每趟选择最小值, 都没有利用上前一趟进行比较所得到的结果, 重复的操作比较多。堆排序算法的基本思想是: 在每次选择最小或最大值时, 利用以前的比较结果以提高排序的速度。

n 个元素的序列 $\{k_0, k_1, \dots, k_{n-1}\}$ 当且仅当满足下列关系时, 称之为堆。

$$\begin{cases} k_i \geq k_{2i+1} \\ k_i \geq k_{2i+2} \end{cases} \quad i = 0, 1, \dots, \frac{n-1}{2}$$

如果将此序列看成是一个完全二叉树的数组表示，则对应的完全二叉树中所有非终端结点的值均大于等于其左右孩子结点的值，树的根结点（堆顶）的值为序列的最大值。

先将序列建成一个堆，若在输出堆顶的最大值后，调整剩余的序列重新建成一个堆，则可以取得次大值，如此反复，便得到一个有序序列，该过程称为堆排序。

2. 数组的堆排序算法实现

```
public static void HeapSort(T[] items) {
    T t;
    int i, n = items.Length;
    for (i = (n-1)/2; i >= 0; i--) {           //从最后一个非终端结点建大顶堆
        HeapAdjust(items, i, n);
    }
    for (i = n-1; i > 1; i--) {
        t = items[0];
        items[0] = items[i];
        items[i] = t;                          //根(最大)值交换到后面
        HeapAdjust(items, 0, i);              //调整成堆
    }
}

private static void HeapAdjust(T[] items, int s, int m) {
    int i = s; int j = 2 * i + 1;              //第j个元素是第i个元素的左孩子
    T t = items[i];                            //获得第i个元素的值
    while (j < m-1) {
        if (items[j].CompareTo(items[j+1]) < 0) //如果右孩子值较大时, j表示右孩子
            j++;
        if (t.CompareTo(items[j]) < 0) {        //根小, 子树调整成堆
            items[i] = items[j];               //设置第i个元素为j的值
            i = j;                             //i, j向下滑动一层
            j = 2 * i + 1;
        }
        else break;
    }
    items[i] = t;
}
```

HeapSort 方法实现堆排序算法，它从最后一个非终端结点开始调用 HeapAdjust 方法 $(n+1)/2$ 次，将待排序序列建成堆。再调用 HeapAdjust 方法 $n-2$ 次，每次将根（最大值）交换到依次缩小的序列尾部。

当序列长度 n 较小时，不提倡使用堆排序算法，当 n 较大时，堆排序算法还是很有效的，它的时间复杂度为 $O(n \log_2 n)$ ，即使在最坏的情况下亦能保持这样的时间复杂度，相对于快速排序，这是

堆排序的一大优点。

堆排序算法在运行过程中需要一个辅助存储空间来交换两个数据元素，这与序列的长度无关，故算法的空间复杂度为 $O(1)$ 。

堆排序算法是不稳定的，对于关键字相同的元素，排序会改变它们原有的次序。

9.5 归并排序

有序的数据便于处理，如果待排序序列内已存在某种有序性，排序算法利用上这种内在的有序性，那么将加快排序操作的运行。

1. 算法的基本思想

将两个有序子数据序列合并，形成一个大的有序序列的过程称为归并（merge），又称两路归并。对于有 n 个元素的待排序数据序列，两路归并排序算法的过程如下：

- 1) 将待排序序列看成是 n 个长度为 1 的已排序子序列。
- 2) 依次将两个相邻的子序列合并成一个大的有序序列。
- 3) 重复第 2 步，合并更大的有序子序列，直到完成整个序列的排序。

2. 数组的归并排序算法实现

```
public static void MergeSort(T[] items) {
    int len = 1;                                //已排序的序列长度, 初始值为1
    T[] temp = new T[items.Length];             //temp所需空间与items一样
    do {
        MergePass(items, temp, len);             //将items中元素归并到temp中
        Show(0, temp);
        len *= 2;
        MergePass(temp, items, len);             //将temp中元素归并到items中
        Show(0, items);
        len *= 2;
    } while (len < items.Length);
}

private static void MergePass(T[] src, T[] dst, int len) {
    int i = 0, j;
    Console.WriteLine("len=" + len + " ");
    while (i < src.Length - 2 * len) {           //src至少包含两块子序列
        Merge(src, dst, i, i + len, len);
        i += 2 * len;
    }
    if (i + len < src.Length)
        Merge(src, dst, i, i + len, len);       //src余下不足两块子序列, 再一次归并
    else
        for (j = i; j < src.Length; j++)        //src余下不足一块子序列, 直接复制到dst
```

```
        dst[j] = src[j];
    }

    private static void Merge(T[] src, T[] dst, int r1, int r2, int n) {
        int i = r1, j = r2, k = r1;
        while (i < r1 + n && j < r2 + n && j < src.Length) {
            if (src[i].CompareTo(src[j]) < 0) { //较小的值送到dst中
                dst[k] = src[i]; k++; i++;
            }
            else {
                dst[k] = src[j]; k++; j++;
            }
        }
        while (i < r1 + n) { //将一子序列余下的值复制到dst中
            dst[k] = src[i]; k++; i++;
        }
        while (j < r2 + n && j < src.Length) { //将另一子序列余下的值复制到dst中
            dst[k] = src[j]; k++; j++;
        }
    }
}
```

MergeSort 方法实现两路归并排序算法。待排序的数据序列存放在数组 **items** 中，**temp** 是排序中使用的一个辅助数组，它具有与 **items** 相同的长度。变量 **len** 是归并过程中当前已排序的子序列的长度，初始值为 1，每次归并后，**len** 的值扩大一倍。一轮外循环（**do...while** 循环）中通过两次调用 **MergePass** 方法完成两趟归并排序，分别从 **items** 归并到 **temp**，再从 **temp** 归并到 **items**，使得排序后的数据序列仍保存在数组 **items** 中。

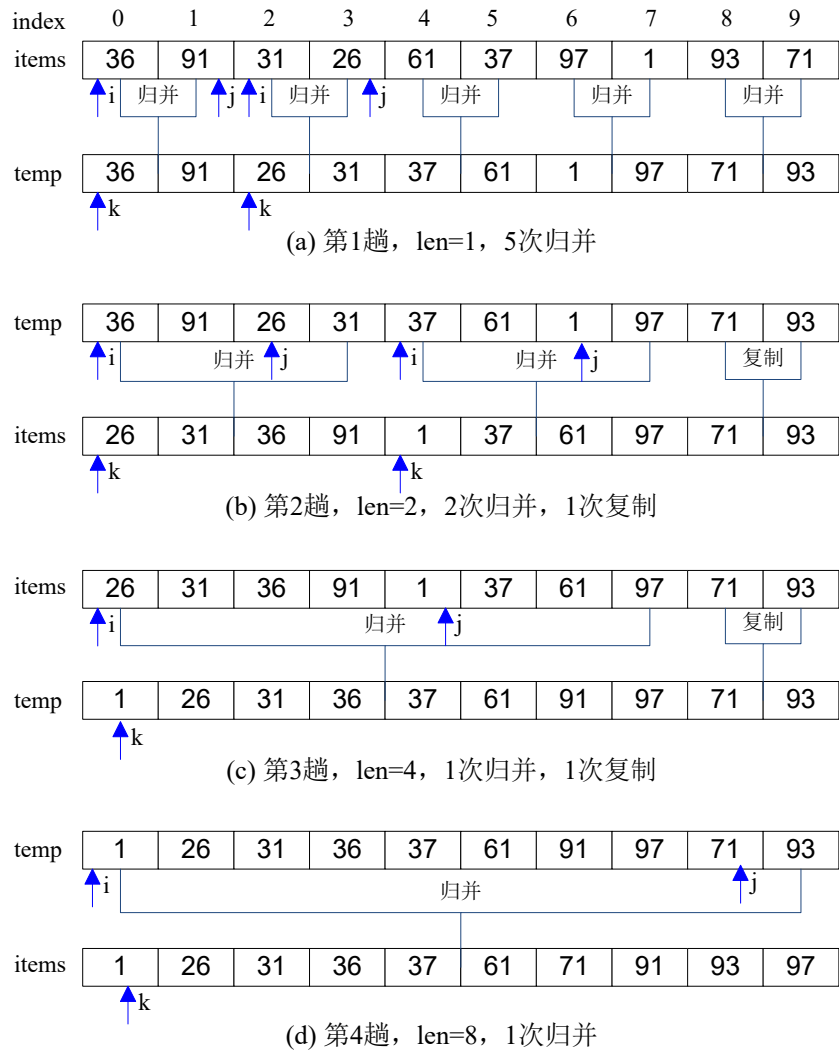


图 9.5 序列的两路归并排序过程描述

MergePass 方法完成一趟归并排序。它调用 Merge 方法, 依次将数组 src (例如 items) 中相邻两个有序子序列归并到数组 dst (例如 temp) 中, 子序列的长度为 len。如果相邻的子序列已归并完, 数组 src 中仍有数据, 则将其复制到 dst 中。

Merge 方法完成两个有序子序列的归并, 将数组 src 中相邻的两个子序列 (起始位置分别为 r_1 和 r_2 , 长度为 n, n 为形参, 其值由调用者设定为当前子序列的长度)

$\text{src}[r_1], \dots, \text{src}[r_1+n-1]$ 和 $\text{src}[r_2], \dots, \text{src}[r_2+n-1]$

归并到 dst 中

$\text{dst}[r_1], \dots, \text{dst}[r_1+n+n-1]$

假设有一个待排序的数据序列为 $\text{items}=\{36, 91, 31, 26, 61, 37, 97, 1, 93, 71\}$, 在其上进行两路归并排序的过程描述如错误! 未找到引用源。所示。算法测试的程序运行结果如下:

len=1 数据序列: 36 91 26 31 37 61 1 97 71 93
 len=2 数据序列: 26 31 36 91 1 37 61 97 71 93
 len=4 数据序列: 1 26 31 36 37 61 91 97 71 93
 len=8 数据序列: 1 26 31 36 37 61 71 91 93 97
 排序后数据序列: 1 26 31 36 37 61 71 91 93 97

3. 算法分析:

Merge 方法完成两个有序子序列的归并, 需要进行 $O(\text{len})$ 次比较。MergePass 方法完成一趟归并排序, 需要调用 Merge 方法 $O(n/\text{len})$ 次。MergeSort 方法实现归并排序算法, 需要调用 MergePass 方法 $O(\log_2 n)$ 次。所以, 归并算法的时间复杂度为 $O(n \log_2 n)$ 。

归并排序算法在运行过程中需要与存储原数据序列的空间相同大小的辅助空间, 所以它的空间复杂度为 $O(n)$ 。

归并排序算法是稳定的, 对于关键字相同的元素, 排序不会改变它们原有的次序。

习题 9

9.1 设要将序列 (12, 61, 8, 70, 97, 75, 53, 26, 54, 61) 按非递减顺序重新排列, 则:

冒泡排序一趟的结果是_____;

插入排序一趟的结果是_____;

二路归并排序一趟的结是_____;

快速排序一趟的结果 (以原首元素为枢轴) 是_____;

上述算法中稳定的排序算法有_____;

9.2 设有一个待排序的数据序列, 其关键字序列如下: {3, 17, 12, 61, 8, 70, 97, 75, 53, 26, 54, 61}, 试写出下列排序算法对这个数据序列进行排序的中间及最终结果:

1) 直接插入排序。

2) 希尔排序。

3) 冒泡排序。

4) 快速排序。

5) 选择排序。

6) 归并排序。

9.3 说明本章介绍的各个排序算法的特点, 并比较它们的的时间复杂度与空间复杂度。

9.4 排序算法的稳定性的含义是什么? 说明本章介绍的各个排序算法的稳定性。

9.5 排序的关键字不同, 排序的结果也不一样。说明 C# 程序中指定排序关键字的一些方法。

9.6 分析用冒泡排序对数据序列 $\text{items}=\{70, 30, 12, 61, 80, 20, 97, 46\}$ 进行升序排序所需的比较操作的总次数。

9.7 分析快速排序在最好情况和最坏情况下的时间复杂度。