数据结构与算法
实验报告

第 1 次

姓名	丁昌灏
班级	软件 72 班
学号	2174213743
电话	15327962577
Email	dch0031@stu.xjtu.edu.cn
日期	2020-10-23

目录

实验 1 2
1、题目 2
2、数据结构设计 2
3、算法设计 2
4、主干代码说明 2
5、运行结果展示 6
6、总结和收获 6
实验 2 6
1、题目 6
2、算法设计 6
3、主干代码说明 6
4、运行结果展示 7
实验 3 15
1、题目 15
2、算法设计 15
3、主干代码说明 15
4、运行结果展示 15
实验 4 17
1、题目 17
2、算法设计 17
3、主干代码说明 17
4、运行结果展示 18
5、总结和收获 19

实验 1

1、题目

完成直接插入排序、简单选择排序、冒泡排序、快速排序和归并排序的实现；

2、数据结构设计

为了便于统计交换和比较次数，声明三个属性

```
/**
 * 记录比较次数
 */
public static int compareTimes = 0;

/**
 * 记录交换次数
 */
public static int swapTimes = 0;

/**
 * 记录归并排序移动次数
 */
public static int moveTimes = 0;
```

3、算法设计

算法参考教科书、老师所讲以及我对两者的理解，并未进行特殊设计。

4、主干代码说明

由于这些排序算法会频繁使用交换，于是交换单独写成一个函数便于复用

```
/**
 * 交换数组中两个元素位置
 *
 * @param array 输入的数组
 * @param pos1 原位置
 * @param pos2 新位置
 */
public static void swap(double[] array, int pos1, int pos2) {
    double temp = array[pos1];
    array[pos1] = array[pos2];
    array[pos2] = temp;
}
```

直接插入排序算法：

```java
/**
 * 直接插入排序:逐个处理待排序的记录
 *
 * @param array 待排序的数组
 */
public static void inssort(double[] array) {
    if (null == array) {
        return;
    }
    for (int i = 1; i < array.length; i++) {
        for (int j = i; j > 0; j--) {
            compareTimes++;
            if (array[j] < array[j - 1]) {
                swap(array, j, j - 1);
                swapTimes++;
            } else {
                break;
            }
        }
    }
}
```

简单选择排序：

```java
/**
 * 简单选择排序:选择未排序中最小的记录
 *
 * @param array 待排序的数组
 */
public static void selsort(double[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        int lowIndex = i;
        // 找未排序中最小的记录的数组下标
        for (int j = array.length - 1; j > i; j--) {
            compareTimes++;
            if (array[j] < array[lowIndex]) {
                lowIndex = j;
            }
        }
        if (lowIndex != i) {
            swap(array, lowIndex, i);
            swapTimes++;
        }
    }
}
```

冒泡排序：

```java
/**
 * 冒泡排序
 *
 * @param array 待排数组
 */
public static void bubsort(double[] array) {
    for (int i = 0; i < array.length; i++) {
        for (int j = array.length - 1; j > i; j--) {
            compareTimes++;
            if (array[j] < array[j - 1]) {
                swap(array, j, j - 1);
                swapTimes++;
            }
        }
    }
}
```

快速排序：在选择轴值时选择三值取中法，同时也计算这个交换和比较次数

```java
/**
 * 三值选中法找轴值
 *
 * @param array 待排数组
 * @param l     数组最小索引
 * @param r     数组最大索引
 * @return 轴值
 */
public static double findPivot(double[] array, int l, int r) {
    int center = (l + r) / 2;
    compareTimes++;
    if (array[l] > array[center]) {
        swap(array, l, center);
        swapTimes++;
    }
    compareTimes++;
    if (array[l] > array[r]) {
        swap(array, l, r);
        swapTimes++;
    }
    compareTimes++;
    if (array[center] > array[r]) {
        swap(array, center, r);
        swapTimes++;
    }
    swap(array, center, r);
    swapTimes++;
    return array[r];
}
```

快速排序中，采用分治递归思想，当数组个数小于 2 时直接比较，大于 2 进行分治递归

```java
public static void qsort(double[] array, int l, int r) {
    if ((r + 1 - l) > 2) {
        // 数组元素大于二进行快速排序
        double pivot = findPivot(array, l, r);
        int start = l;
        int end = r;
        while (start < end) {
            while (start != r && array[++start] < pivot) {
                compareTimes++; // 满足条件进行比较，比较次数增加
            }
            compareTimes++; // 不满足条件也进行了一次比较，次数加一
            while (end != l && array[--end] >= pivot) {
                compareTimes++;
            }
            compareTimes++;
            swap(array, start, end);
            swapTimes++;
        }
        swap(array, start, end);
        swapTimes++;
        swap(array, r, start);
        swapTimes++;
        qsort(array, l, start - 1);
        qsort(array, start + 1, r);
    } else {
        // 数组元素小于等于2,直接比较大小
        if (array[l] > array[r]) {
            swap(array, l, r);
            swapTimes++;
        }
    }
}
```

归并排序：

```
    /**
     * 归并排序
     *
     * @param array 待排数组
     * @param temp  辅助数组
     * @param l     数组最小索引
     * @param r     数组最大索引
     */
    public static void mergeSort(double[] array, double[] temp, int l, int r) {
        int mid = (l + r) / 2;
        if (l == r)
            return; // 只有一个元素
        mergeSort(array, temp, l, mid);
        mergeSort(array, temp, mid + 1, r);
        for (int i = l; i <= r; i++) {
            temp[i] = array[i];
        }
        int i1 = l;
        int i2 = mid + 1;
        for (int curr = l; curr <= r; curr++) {
            // 其中一个数组进行全部排完
            if (i1 > mid) {
                for (; curr <= r; curr++) {
                    array[curr] = temp[i2++];
                    if (curr != (i2 - 1)) {
                        moveTimes++;
                    }
                }
                return;
            }

        if (i2 > r) {
            for (; curr <= r; curr++) {
                array[curr] = temp[i1++];
                if (curr != (i1 - 1)) {
                    moveTimes++;
                }
            }
            return;
        }

        // 进行比较，比较次数+1
        compareTimes++;
        if (temp[i1] == temp[i2]) {
            array[curr] = temp[i1++];
            // 原位置与当先位置不同，记位一次移动
            if (curr != (i1 - 1)) {
                moveTimes++;
            }
        } else if (temp[i1] > temp[i2]) {
            array[curr] = temp[i2++];
            if (curr != (i2 - 1)) {
                moveTimes++;
            }
        } else {
            array[curr] = temp[i1++];
            if (curr != (i1 - 1)) {
                moveTimes++;
            }
        }
    }
}
```

5、运行结果展示

测试代码

```
double[] array = new double[10];
for (int i = 0; i < 10; i++) {
    array[i] = (int) (Math.random() * 500);
}
System.out.println("待排数组为:" + Arrays.toString(array));
inssort(array);
// selsort(a);
// bubsort(a);
// qsort(a, 0, a.length - 1);
// mergeSort(a, temp, 0, a.length - 1);
System.out.println("排序后数组为:" + Arrays.toString(array));
```

依次测试插入排序、简单选择排序、冒泡排序、快速排序和归并排序

```
待排数组为:[230.0, 335.0, 487.0, 256.0, 478.0, 195.0, 380.0, 286.0, 413.0, 323.0]
排序后数组为:[195.0, 230.0, 256.0, 286.0, 323.0, 335.0, 380.0, 413.0, 478.0, 487.0]
```

```
待排数组为:[264.0, 472.0, 154.0, 250.0, 135.0, 415.0, 338.0, 155.0, 435.0, 29.0]
排序后数组为:[29.0, 135.0, 154.0, 155.0, 250.0, 264.0, 338.0, 415.0, 435.0, 472.0]
```

```
待排数组为:[249.0, 58.0, 62.0, 24.0, 126.0, 155.0, 349.0, 435.0, 62.0, 274.0]
排序后数组为:[24.0, 58.0, 62.0, 62.0, 126.0, 155.0, 249.0, 274.0, 349.0, 435.0]
```

```
待排数组为:[230.0, 298.0, 150.0, 156.0, 111.0, 188.0, 55.0, 110.0, 147.0, 206.0]
排序后数组为:[55.0, 110.0, 111.0, 147.0, 150.0, 156.0, 188.0, 206.0, 230.0, 298.0]
```

```
待排数组为:[381.0, 426.0, 305.0, 85.0, 256.0, 387.0, 99.0, 468.0, 90.0, 113.0]
排序后数组为:[85.0, 90.0, 99.0, 113.0, 256.0, 305.0, 381.0, 387.0, 426.0, 468.0]
```

均通过测试

6、总结和收获

通过编程实现算法刚开始还是遇到了许多问题，反应了实践的重要性，在编程之前没有想到有这么多的细节需要处理。

实验 2

1、题目

完成对每一个排序算法在输入规模为：100、200、300、……、10000 的排序时间、比较次数和交换次数的统计。

2、算法设计

依次生成规模为 100、200、……、10000 的顺序和逆序数组，每种算法都计算三次取时间平均值，然后把时间、比较次数、交换次数写入.txt 文件中，最后借助 matlab 画图。

3、主干代码说明

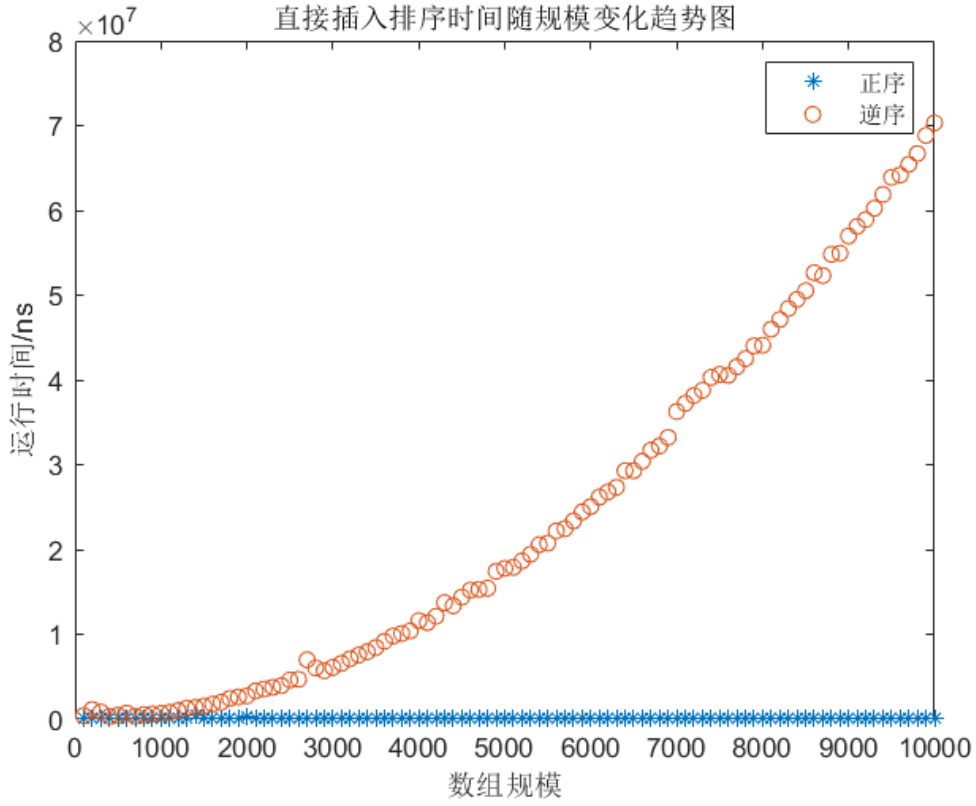
计算代码：

```
public static void batchTest(int size, File f) {
    long sum = 0;
    // 每个算法的每个规模计算3次,取平均值
    for (int i = 0; i < 3; i++) {
        double[] array = new double[size];
        double[] temp = new double[size];
        for (int j = 0; j < size; j++) {
            // array[j] = j + 1;
            array[j] = size - j;
        }
        compareTimes = 0;
        swapTimes = 0;
        moveTimes = 0;
        long start = System.nanoTime();
        // inssort(array);
        // selsort(array);
        // bubsort(array);
        // qsort(array, 0, array.length - 1);
        mergeSort(array, temp, 0, array.length - 1);
        long end = System.nanoTime();
        long time = end - start;
        sum += time;
    }
    long avg = sum / 3;
```

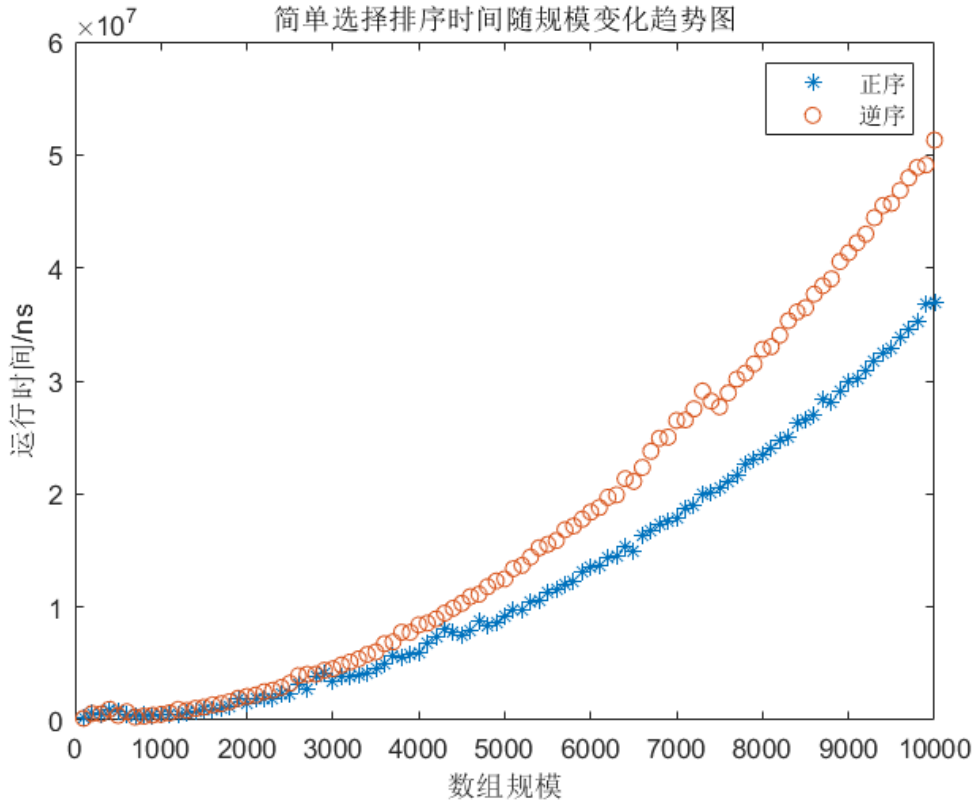
写入文件代码：

```
FileOutputStream fos = null;
try {
    fos = new FileOutputStream(f, true);
    fos.write(String.valueOf(avg).getBytes());
    fos.write("\t".getBytes());
    fos.write(String.valueOf(compareTimes).getBytes());
    fos.write("\t".getBytes());
    fos.write(String.valueOf(moveTimes / 3).getBytes());
    fos.write("\n".getBytes());
    fos.flush();
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        fos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

4、运行结果展示

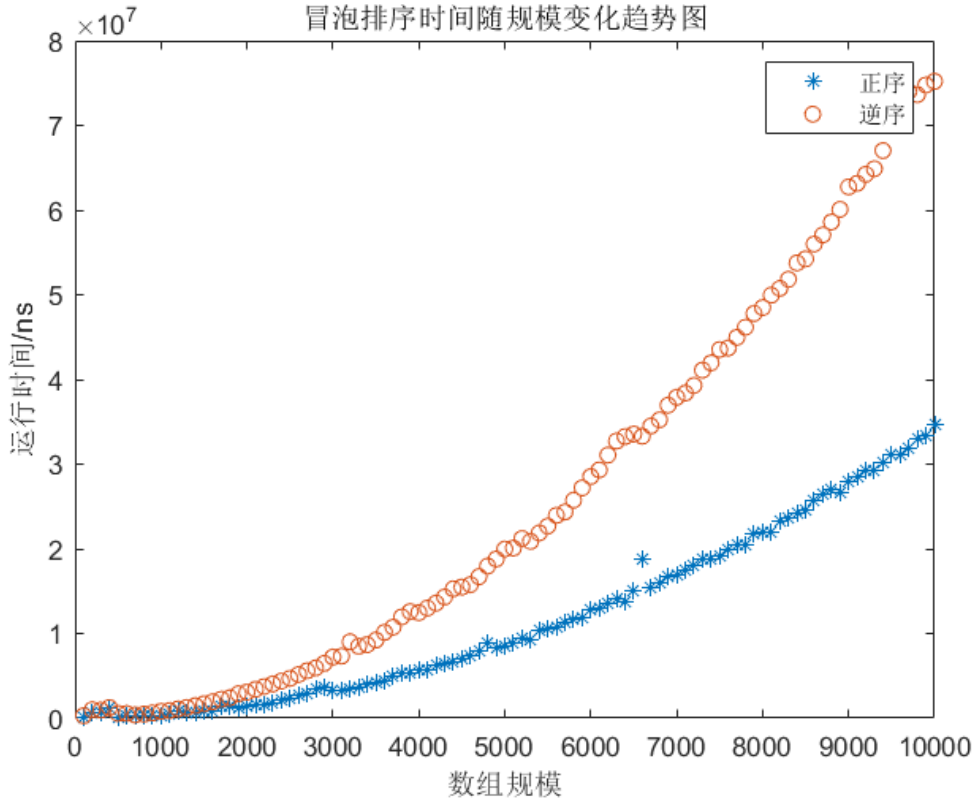


逆序需要从尾到头都比较并交换，复杂度为 $O(n^2)$而顺序只需要比较一次，复杂度为 $O(n)$ 因此逆序增加比正序快很多

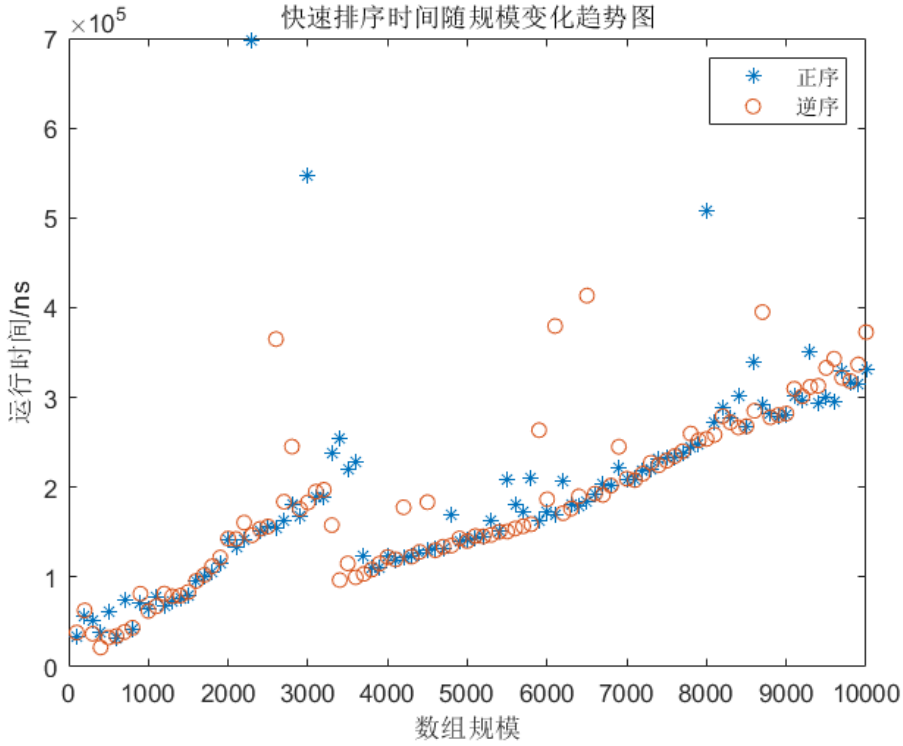


简单选择排序，无论顺序还是逆序，都需要进行相同次数比较，正序不需要交换，因此

两者增长趋势相似，但顺序时间小于逆序

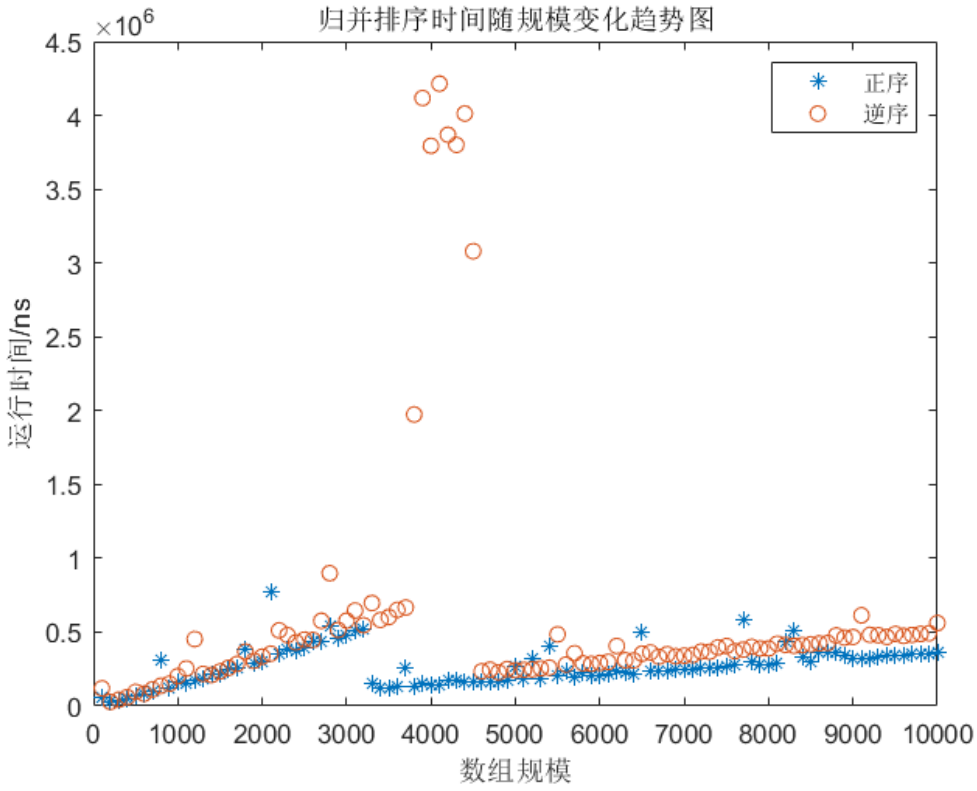


冒泡排序与简单选择排序情况相似，无论正序还是逆序都需要比较相同次数，但是正序不需要交换，因此时间增长慢与逆序

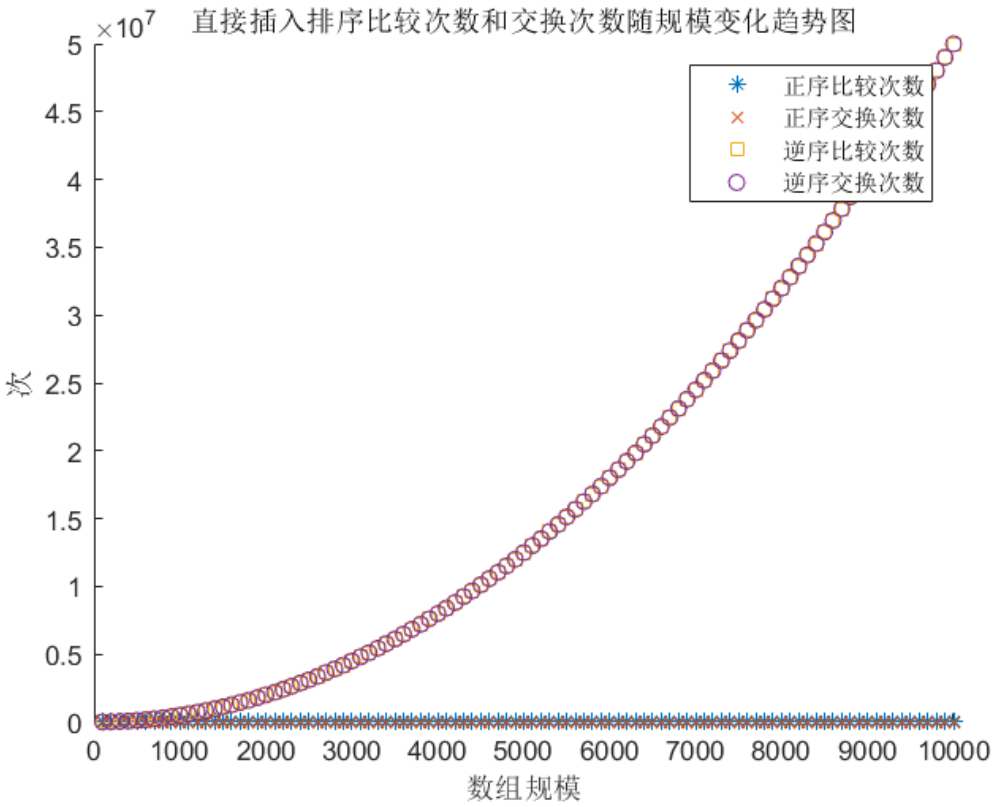


在实验中多次出现规模大的运行时间要低于规模小，这里无法解决，但是整体趋势两者基本相同，因为正序和逆序比较次数相似，两者划分数组规模相同，而逆序交换次数多，所

以逆序略大于正序但是不明显

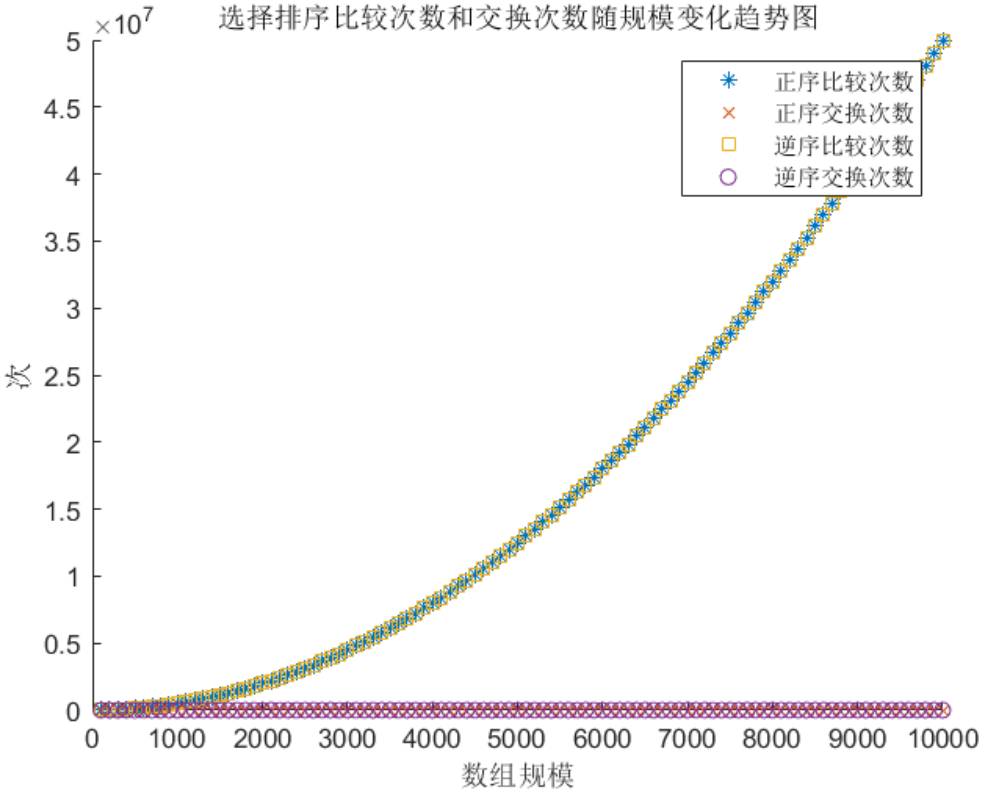


归并排序也出现时间不正常，但是总体规律符合，与快排类似，归并每次分隔数组规模相同，因此时间增长相似，但是正序交换少，所以时间略少于逆序。

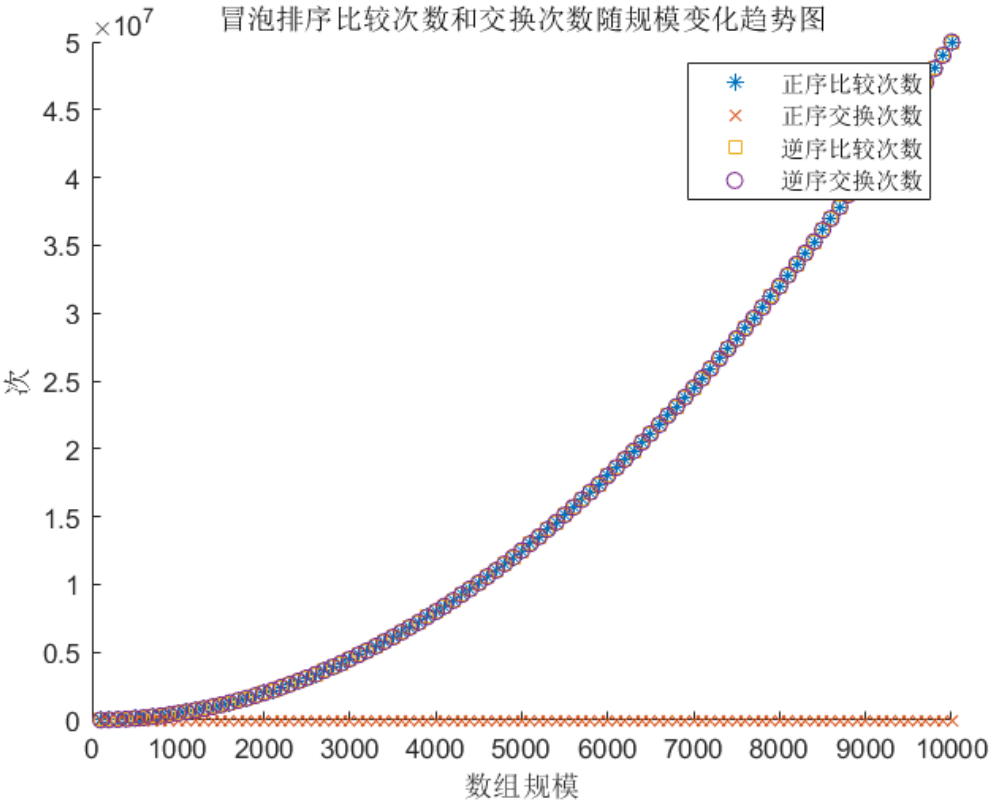


直接插入排序中逆序交换次数和比较次数相同为 *n(n-1)/2*，正序交换次数为 *0*，正序比

较次数为*(n-1)*

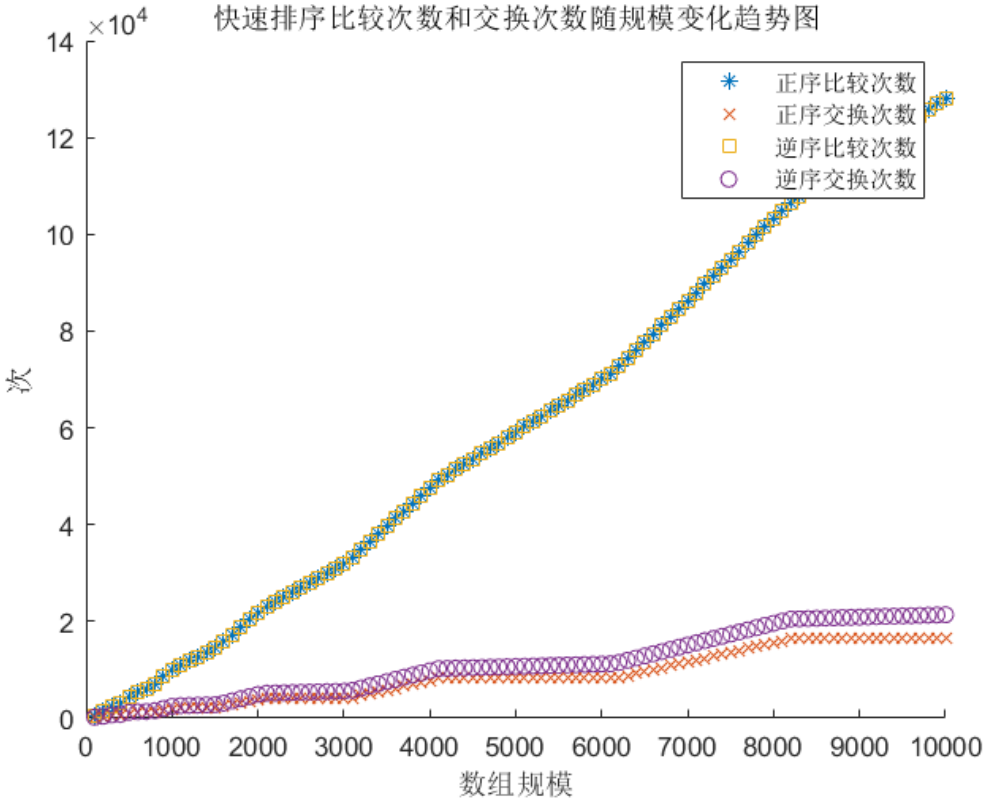


简单选择排序正序和逆序比较次数相同为 *n(n-1)/2*，正序交换次数为 *0*，逆序交换次数为 *n/2*

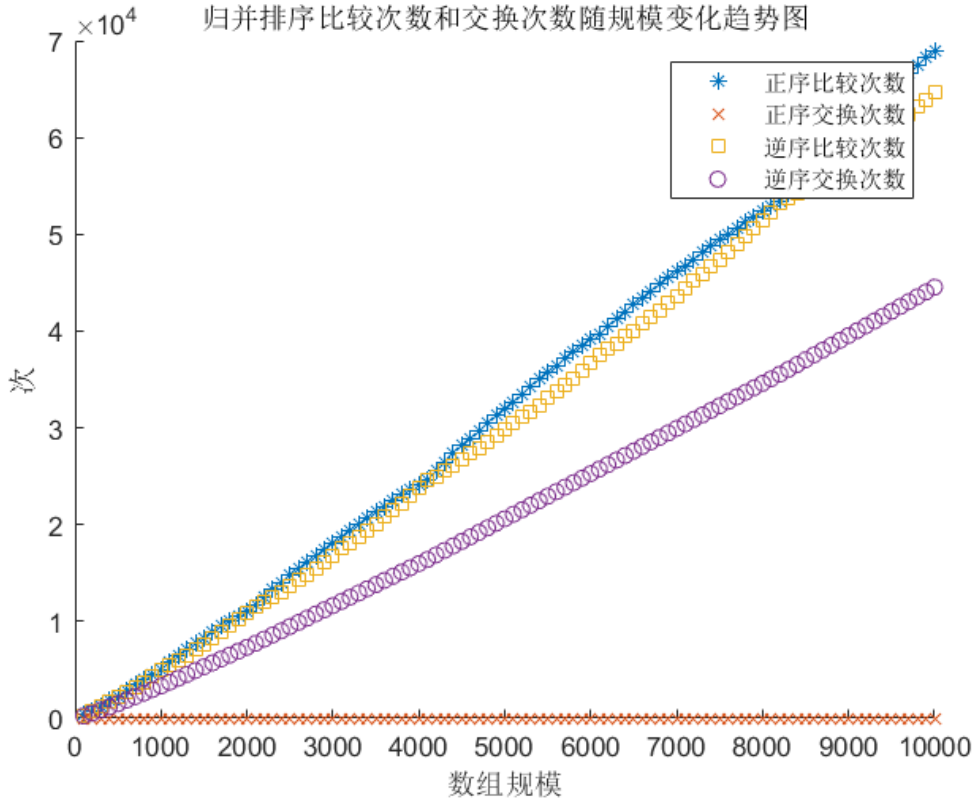


冒泡排序正序比较次数、逆序比较次数和逆序交换次数相同为 *n(n-1)/2*，正序交换次数

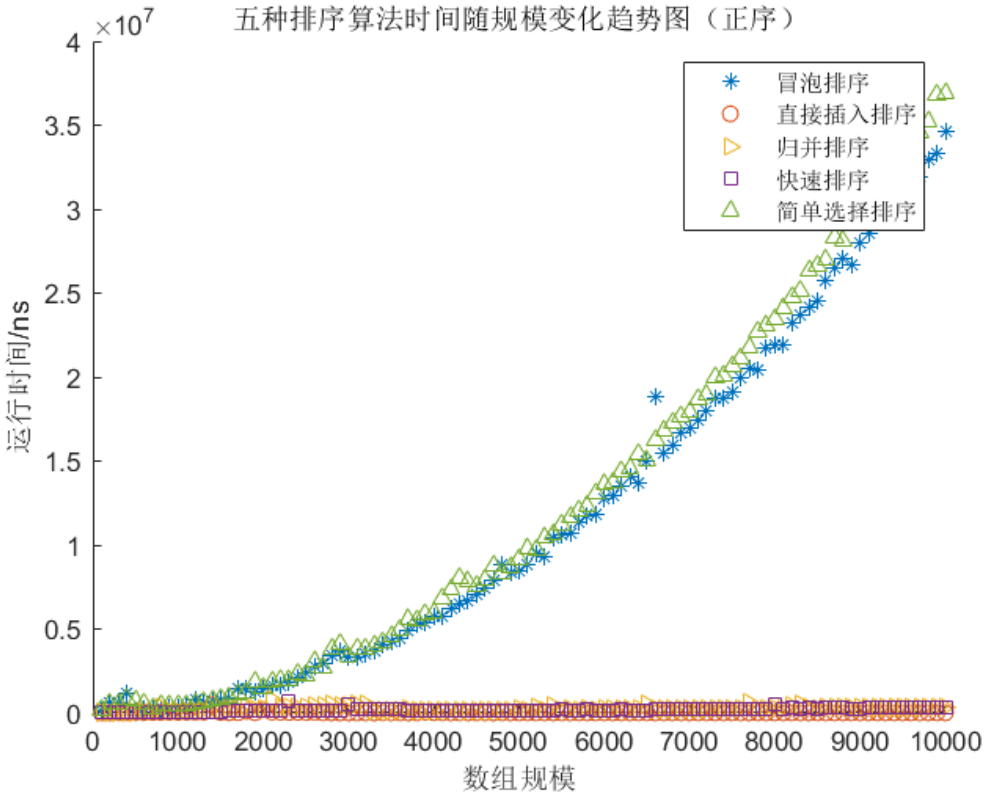
为 *0*



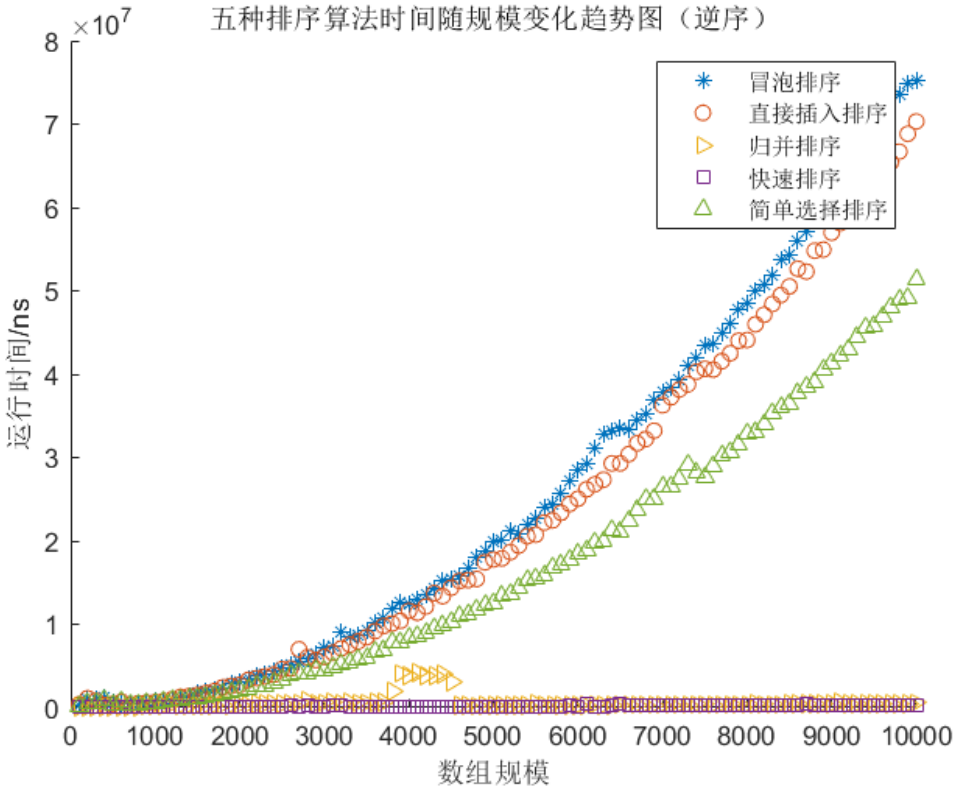
快速排序（统计了轴值时交换比较次数），正序和逆序比较次数相同，正序交换次数小于逆序交换次数



归并排序正序和逆序比较次数相近，正序交换次数为 0。

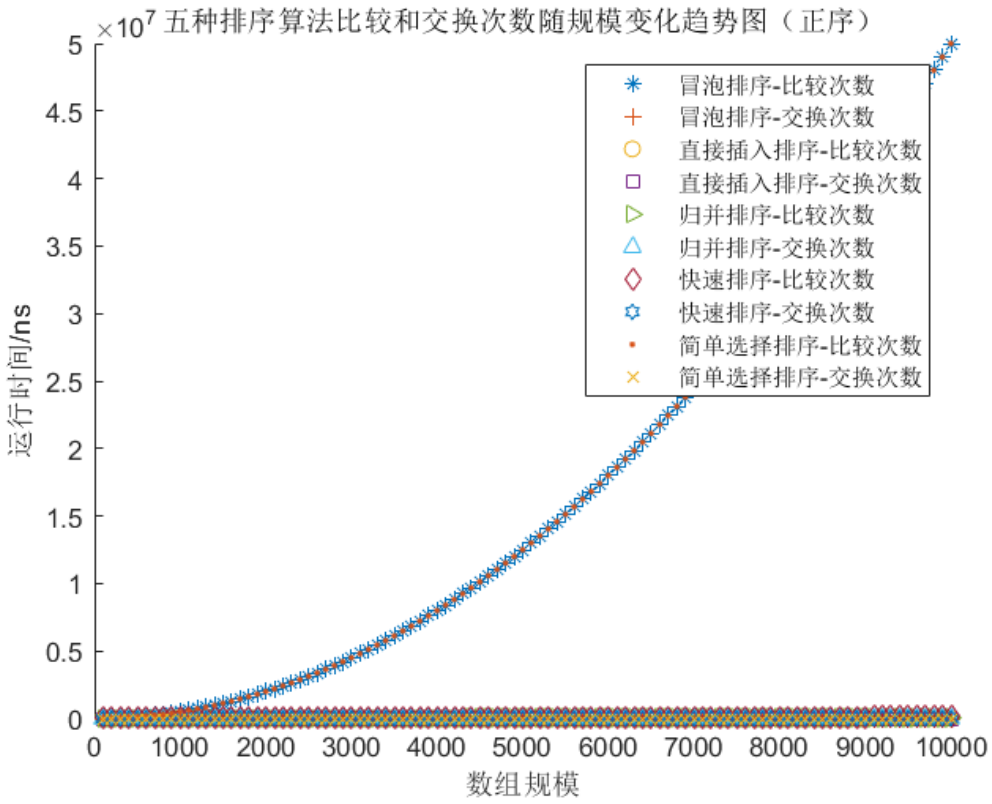


正序情况下，直接插入排序复杂度 $O(n)$速度最快，归并排序和快速排序时间较短为 $O(nlogn)$，冒泡排序与简单选择排序差不多都为 $O(n^2)$

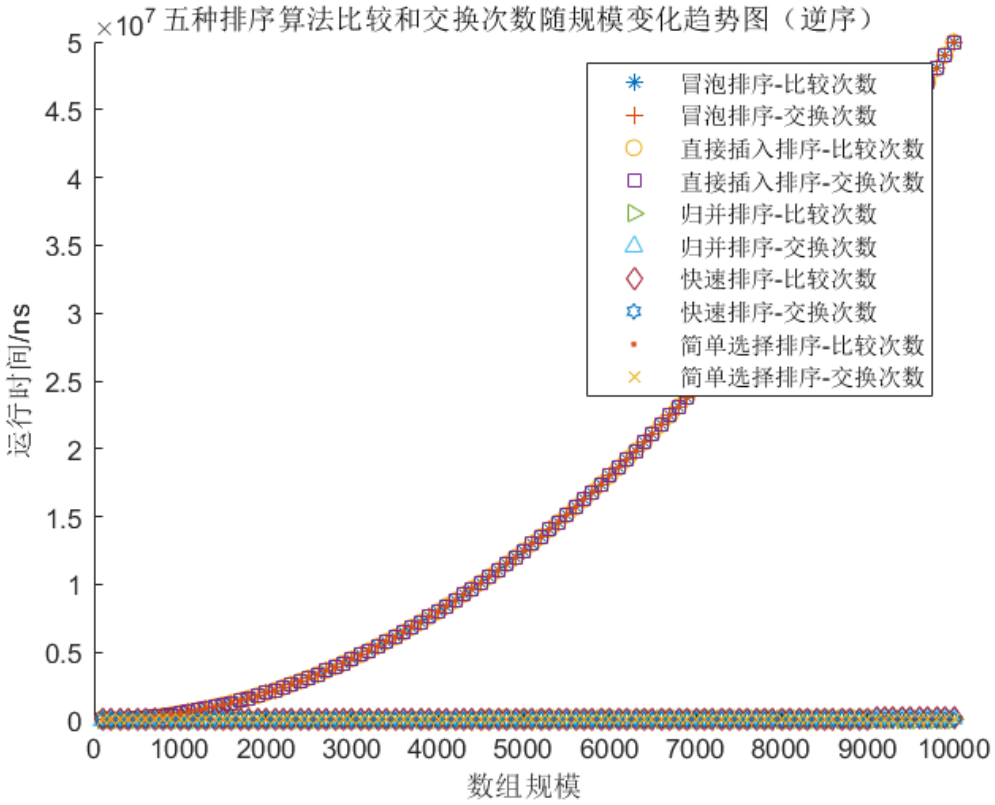


逆序情况下，归并排序和快速排序相似，复杂度为 $O(nlogn)$，此时冒泡排序、直接插入排序、简单选择排序都为 $O(n^2)$，简单选择排序交换次数较少，所以时间略短于冒泡排序和

直接插入排序。



正序中冒泡排序和简单选择排序都需要 *n(n-1)/2* 次的比较，直接插入排序需要 *n-1* 次比较，快速排序需要交换次数，而别的排序都不需要。



逆序中冒泡排序和直接插入排序都需要 *n(n-1)/2* 次的比较和交换次数，简单选择排序需

要 *n(n-1)/2* 次比较，只需要 *n/2* 次交换。

实验 3

1、题目

完成对每一个排序算法在输入规模为：100、200、300、……、10000 的随机数据序列的排序时间统计。

2、算法设计

每种规模不同算法的随机序列需要相同，其余与实验 2 中一致

3、主干代码说明

生成一个数组，要对其进行深拷贝，否则第一次排序好，后面成为顺序数组

生成随机数组：

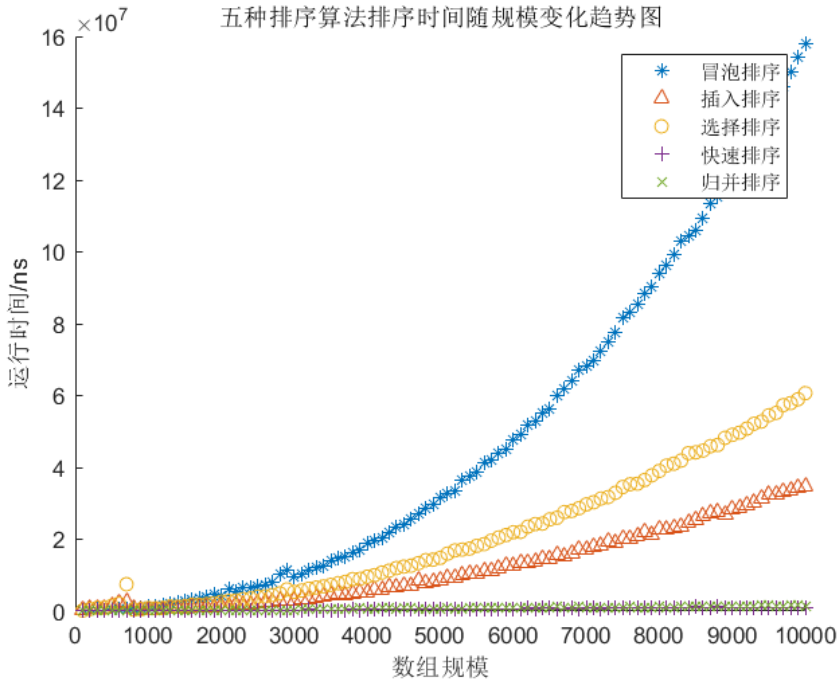
```
// 随机生成规模为size的数组
double[] array = new double[size];
double[] temp = new double[size];
for (int i = 0; i < size; i++) {
    array[i] = (int) (Math.random() * 10000);
}
```

每个算法使用之前，先进行复制：

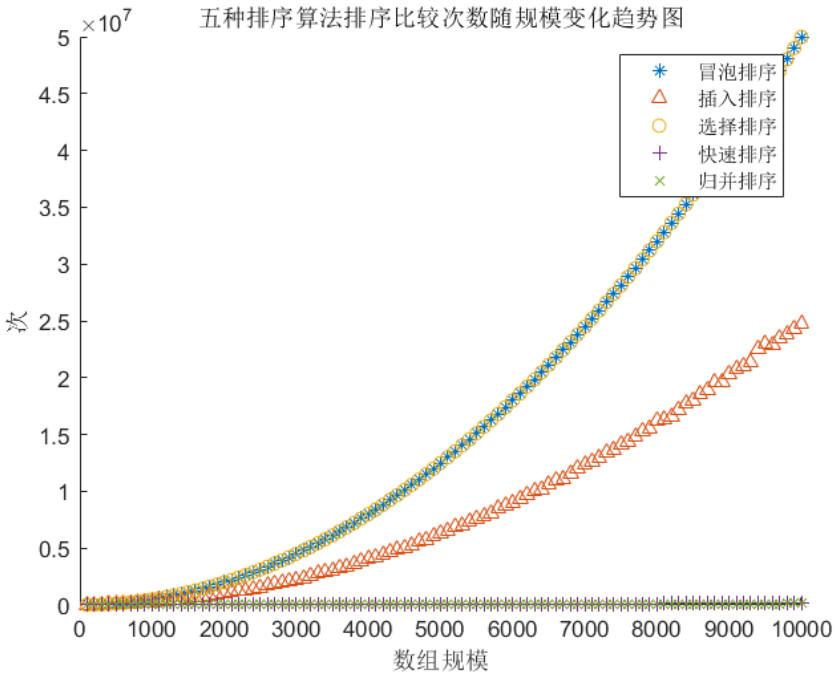
```
// 初始化数组
double[] temp1 = new double[size];
for (int j = 0; j < size; j++) {
    temp1[j] = array[j];
}
```

其余步骤和实验 2 相同，在这里不再赘述

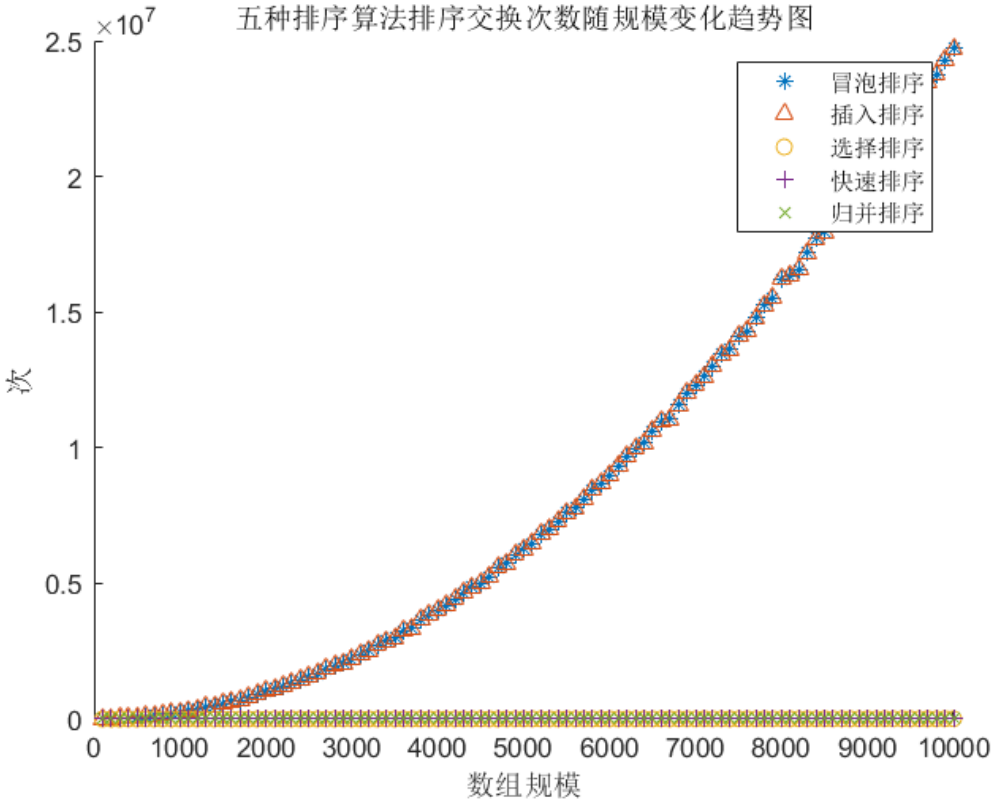
4、运行结果展示



快速排序与归并排序效率相似，归并排序略慢与快速排序，冒泡排序、插入排序、简单选择排序虽然时间复杂度都是 $O(n^2)$但是随着问题规模扩大，冒泡排序增加更快



冒泡排序与简单选择排序比较次数相同，快排比较次数略多余归并排序



冒泡排序与插入排序交换次数相同，简单选择排序交换次数最少，大约为快速排序的 1/5，归并排序的 1/4

实验 4

1、题目

实现获得数据集合的中位数的有效算法。

2、算法设计

一般寻找中位数可以先进行排序，然后找到中位数，如果采用快速排序，其时间复杂度为 *O(nlogn)*。快速排序将数组分为比轴值大、小的两部分，并对两部分分别进行递归，实际上找中位数时并不需要对两边都进行处理。

在快速排序的基础上，在轴值将集合划分为两部分时，如果轴值下标正好为中间下标，则轴值即为中位数，如果轴值下标大于中位数下标，则中位数一定在比轴值小的那部分，同理如果轴值下标小于中位数下标，则中位数一定在比轴值大的那部分。这样每次我们能舍弃一部分数组，并且不需要排序，降低了时间复杂度。

复杂度分析如下：

$$\mathrm{T}(n)=\begin{cases}O(1) & n=1\\ T(n/k)+O(n) & n>1\end{cases}$$

采用三值选中法，如果能排除则至少排除两个（轴值和比轴值大或小的）最坏时间复杂度为 $O(n^2)$。一般情况下能都减少 *n/k* 则平均时间复杂度为 *O(n)*。

3、主干代码说明

```
     * @param array 待寻找的数组
     * @param l     数组最小下标
     * @param r     数组最大下标
     * @return 中位数的值
     */
    public static double findMid(double[] array, int l, int r) {
        int index = (0 + array.length - 1) / 2;
        if ((r + 1 - l) > 2) {
            double pivot = findPivot(array, l, r);
            int start = l;
            int end = r;
            while (start < end) {
                while (start != r && array[++start] < pivot)
                    ;
                while (end != l && array[--end] >= pivot)
                    ;
                swap(array, start, end);
            }
            swap(array, start, end);
            swap(array, r, start);
            if (start == index) {
                return array[start];
            }
            if (start > index) {
                return findMid(array, l, start - 1);
            } else {
                return findMid(array, start + 1, r);
            }
        } else {
            // 数组元素小于等于2,直接比较大小
            if (array[l] > array[r]) {
                swap(array, l, r);
            }
            return array[index];
        }
    }
```

4、运行结果展示

生成一组随机数，采用上述算法找出中位数，然后进行排序后通过下标找出中位数，判断是否一致且正确

测试代码：

```
for (int j = 10; j <= 100; j = j + 10) {
    double[] array = new double[j];
    for (int i = 0; i < j; i++) {
        array[i] = (int) (Math.random() * 500);
    }
    System.out.println(j + "个数测试");
    System.out.println("无序数组:" + Arrays.toString(array));
    System.out.println("中位数:" + findMid(array, 0, array.length - 1));
    // 排序
    qsort(array, 0, array.length - 1);
    System.out.println("排序后:" + Arrays.toString(array));
    System.out.println("排序后找中位数:" + array[(0 + array.length - 1) / 2]);
    System.out.println("**********************");
}
```

测试结果：

```
10个数测试
无序数组:[296.0, 201.0, 159.0, 434.0, 490.0, 38.0, 177.0, 285.0, 28.0, 380.0]
中位数:201.0
排序后:[28.0, 38.0, 159.0, 177.0, 201.0, 285.0, 296.0, 380.0, 434.0, 490.0]
排序后找中位数:201.0
**********************
20个数测试
无序数组:[320.0, 486.0, 406.0, 103.0, 361.0, 466.0, 87.0, 379.0, 253.0, 281.0, 57.0, 469.0, 346.0, 136.0, 133
中位数:270.0
排序后:[21.0, 57.0, 61.0, 87.0, 103.0, 111.0, 133.0, 136.0, 253.0, 270.0, 281.0, 320.0, 346.0, 361.0, 379.0,
排序后找中位数:270.0
**********************
30个数测试
无序数组:[175.0, 85.0, 270.0, 236.0, 455.0, 174.0, 74.0, 314.0, 314.0, 405.0, 150.0, 369.0, 418.0, 318.0, 212
中位数:270.0
排序后:[13.0, 49.0, 50.0, 74.0, 85.0, 104.0, 112.0, 150.0, 172.0, 174.0, 174.0, 175.0, 212.0, 236.0, 270.0, 2
排序后找中位数:270.0
**********************
40个数测试
无序数组:[125.0, 218.0, 382.0, 353.0, 45.0, 190.0, 155.0, 212.0, 289.0, 187.0, 15.0, 370.0, 431.0, 18.0, 236.
中位数:254.0
排序后:[10.0, 15.0, 18.0, 42.0, 45.0, 48.0, 60.0, 96.0, 122.0, 125.0, 155.0, 187.0, 190.0, 212.0, 218.0, 236.
排序后找中位数:254.0
**********************
50个数测试
无序数组:[150.0, 44.0, 323.0, 438.0, 450.0, 228.0, 445.0, 54.0, 127.0, 240.0, 128.0, 103.0, 357.0, 378.0, 383
中位数:233.0
排序后:[5.0, 28.0, 30.0, 31.0, 42.0, 44.0, 54.0, 76.0, 80.0, 103.0, 108.0, 116.0, 127.0, 128.0, 144.0, 150.0,
排序后找中位数:233.0
**********************
```

```
60个数测试
无序数组:[383.0, 455.0, 67.0, 60.0, 55.0, 164.0, 388.0, 422.0, 364.0, 416.0, 444.0, 379.0, 22.0, 410.0, 232.(
中位数:269.0
排序后:[8.0, 16.0, 22.0, 36.0, 40.0, 41.0, 55.0, 58.0, 60.0, 62.0, 67.0, 76.0, 81.0, 82.0, 94.0, 95.0, 108.0,
排序后找中位数:269.0
**********
70个数测试
无序数组:[346.0, 91.0, 179.0, 23.0, 394.0, 145.0, 113.0, 360.0, 108.0, 478.0, 452.0, 409.0, 461.0, 152.0, 7.(
中位数:268.0
排序后:[4.0, 7.0, 20.0, 21.0, 23.0, 26.0, 40.0, 47.0, 56.0, 67.0, 84.0, 84.0, 91.0, 108.0, 112.0, 113.0, 114.
排序后找中位数:268.0
**********
80个数测试
无序数组:[366.0, 37.0, 74.0, 213.0, 284.0, 445.0, 494.0, 410.0, 181.0, 308.0, 38.0, 456.0, 151.0, 477.0, 404.
中位数:288.0
排序后:[6.0, 14.0, 26.0, 32.0, 37.0, 38.0, 53.0, 74.0, 79.0, 86.0, 105.0, 106.0, 107.0, 127.0, 129.0, 149.0,
排序后找中位数:288.0
**********
90个数测试
无序数组:[255.0, 239.0, 220.0, 472.0, 158.0, 471.0, 235.0, 467.0, 156.0, 43.0, 3.0, 97.0, 172.0, 114.0, 394.(
中位数:221.0
排序后:[1.0, 3.0, 8.0, 9.0, 16.0, 21.0, 25.0, 34.0, 37.0, 43.0, 43.0, 56.0, 56.0, 59.0, 60.0, 65.0, 68.0, 74.
排序后找中位数:221.0
**********
100个数测试
无序数组:[262.0, 285.0, 442.0, 146.0, 10.0, 397.0, 174.0, 12.0, 189.0, 180.0, 171.0, 244.0, 25.0, 192.0, 81.(
中位数:242.0
排序后:[1.0, 10.0, 12.0, 15.0, 19.0, 22.0, 25.0, 25.0, 26.0, 35.0, 37.0, 39.0, 46.0, 47.0, 54.0, 58.0, 70.0,
排序后找中位数:242.0
**********
```

测试均正确

5、总结和收获

寻找中位数算法让我加深了对于快速排序的理解，同时懂得灵活掌握算法精髓，这样一个算法才是“活”的。