数据结构与算法分析：第一次作业

算法分析与排序问题

课程	数据结构与算法分析
姓名	白靖
专业班级	软件 52
学号	2151601024
邮箱	516267116@qq.com
提交日期	2016 年 10 月 7 日

目录

数据结构与算法分析作业报告：第一次作业

算法分析与排序问题

此报告作为数据结构与算法课程的第一次作业报告，内容包括作业题的题目分析、程序实现、结果展示与算法分析。作业中的程序代码采用 Java 语言实现并在 Windows 环境下测试，运行环境参数如下：

处理器及内存	CPU：Intel(R) Core(TM) i7-4710HQ @2.50GHz RAM：16.00GB(DDR3L 1600MHz)
运行操作系统	MS Windows 10 Home x64 (Version: 10.0.14393)
Java 运行环境	Java(TM) SE Runtime Environment (build 1.8.0_102-b14)

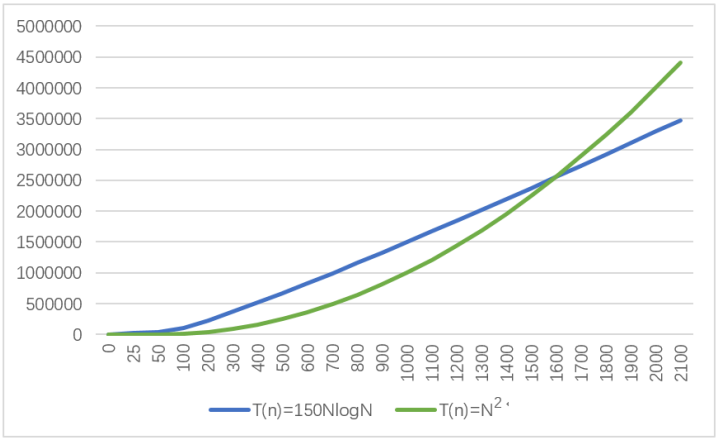
1 时间复杂度判断问题

1.1 作业题目

> *程序 A 和程序 B 经过分析发现其最坏情形运行时间分别不大于 150NlogN 和 N^2。如果可能，请回答下列问题：*
>
> *A、对于 N 的大值（N>10000）,哪一个程序的运行时间有更好的保障？*
>
> *B、对于 N 的小值（N<100），哪一个程序的运行时间有更好的保障？*
>
> *C、对于 N=1000，哪一个程序平均运行得更快？*
>
> *D、对于所有可能的输入，程序 B 是否总够比程序 A 运行得更快？*

1.2 问题分析

对于增长率分别为 150NlogN 和 N^2 的两种算法，下图分别对应了其输入规模（横轴）与时间开销（纵轴）的关系：



则由图可知，对于输入规模 n 较小的情形，增长率为 N^2 的算法的时间开销相对较小；而对于输入规模 n 较大的情形，则增长率为 150NlogN 的算法时间开销较小。两种算法的

具体时间开销的比较与输入规模有很大的关系，并且还与输入数据的情况（最佳、最差、平均）和数据的分布有关。

$$\mathrm{N}=10000 \text{ 时，} 150\mathrm{NlogN}\approx 2\times 10^{7}\text{，而}N^{2}=10^{8}$$

$$\mathrm{N}=100 \text{ 时，} 150\mathrm{NlogN}\approx 10^{5}\text{，而}N^{2}=10^{4}$$

A、 对于 N 的大值（N>10000），程序 B 的运行时间有更好保障。

B、对于 N 的小值（N<100），程序 A 的运行时间有更好保障。

C、对于 N=1000，两程序的运行速度无法比较，因为题目只给出了最坏情况的运行时间，无法得知平均情况下数据的分布情况，故无法做出判断。

D、 不能判断。两程序的运行时间不仅与数据的输入规模有关，还与输入数据的情况（最佳、最差、平均）和数据的分布有关。且题目只给出了最坏情况的运行时间，所以无法得出所有的输入情况的比较。

2 递归方程的大 O 表示与证明

2.1 作业题目

> *考虑以下递归方程，定义函数 T(n):*
>
> *A*、$\mathrm{T(n)}=\begin{cases}1 & \text{如果 } n=1\\ T(n-1)+n & \text{其他情况}\end{cases}$
>
> *B*、$\mathrm{T(n)}=\begin{cases}1 & \text{如果 } n=0\\ 2T(n-1) & \text{其他情况}\end{cases}$
>
> *请给出 A 和 B 两种递归式的大 O 表示，并证明。*

2.2 问题解答

A: 递归式 A 的大 O 表示为$O(n^2)$

以下用数学归纳法证明：

由迭代可得$T(n)=\frac{1}{2}n^2+\frac{1}{2}n$

对于基准情形 n=1，对$T(n)=\frac{1}{2}n^2+\frac{1}{2}n$,

$$\text{满足}T(1)=\frac{1}{2}+\frac{1}{2}=1$$

设$T(k)=\frac{1}{2}k^2+\frac{1}{2}k$，下证对 k+1 也成立：

$$\text{由}\mathrm{T}(k+1)=T(k)+k+1$$

$$T(k+1)=\frac{(k+1)^2+(k+1)}{2}$$

故递归式 A 的大 O 表示为$O(n^2)$。

B：递归式 B 的大 O 表示为$O(2^n)$

以下用数学归纳法证明：

由迭代可得$T(n)=2^n$

对于基准情形 n=0，对$T(n)=2^n$，

满足$T(0)=2^0=1$，

设$T(k)=2^k$，下证对 k+1 也成立：

由$T(k+1)=2T(k)$

$T(k+1)=2^{k+1}$

故递归式 B 的大 O 表示为$O(2^n)$.

3 几种排序算法的测试

3.1 作业题目

> *实现直接插入排序、简单选择排序、希尔排序、快速排序和归并排序，以能够对给定数组的正序排序，并按照满足下列情形进行测试：*
>
> *A、测试数组的大小为[100,200,300,…,10000]100 种大小*
>
> *B、测试数组中的元素分别为正序、逆序和随机序列*
>
> *对测试的结果需要用图形的方式进行展示：*
>
> *a）、展示每个排序算法在满足条件 A 和条件 B 情形下的运行时间趋势变化图，如图 1 所示*

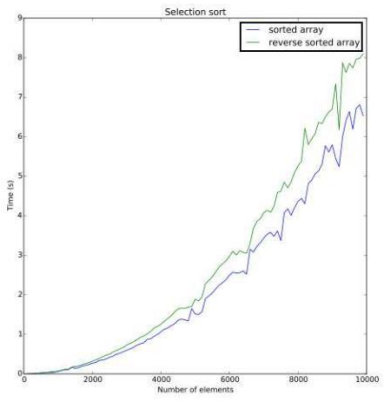


图 1

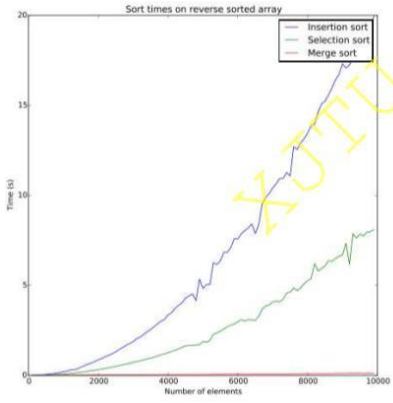


图 2

> *b）、将所有排序算法在正序下、逆序下和随机序列下的运行时间的对比图，如图 2 所示*

3.2 程序实现

只给出算法代码，完整代码详与完整测试数据详见百度云链接 [1]

1 http://pan.baidu.com/s/1kUZdg8n 密码:iqtq

直接插入排序

```
//直接插入排序
public class InsertionSort{
    void sort(int [] array){
        for(int curr = 1; curr < array.length; curr++){
            int temp = array[curr];
            //  较大元素右移
            for(int next = curr; next > 0 && array[next - 1] >
array[next]; next--){
            //  交换位置
                array[next] = array[next - 1];
                array[next - 1] = temp;
            }
        }
    }
}
```

简单选择排序

```
//简单选择排序
public class SelectionSort{
        void sort(int [] array){
        for(int i = 0; i < array.length - 1; i++){
            // 记录最小值索引号及其值
            int low_index = i;
            int low_value = array[i];
            // 遍历数组，寻找最小元素
            for(int j = i + 1; j < array.length; j++){
                if(array[j] < low_value){
                    low_value = array[j];
                    low_index = j;
                }
            }
            // 交换最小值与当前元素
            int temp = array[low_index];
            array[low_index] = array[i];
```

```
            array[i] = temp;
        }
    }
}
```

希尔排序

```
//希尔排序
public class ShellSort{
    void sort(int [] array){
        // 选用已知最好的Marcin Ciura步长
        int [] interval = {1750, 701, 301, 132, 57, 23, 10, 4, 1};
        // 从大至小选用间隔
        for(int i = 0; i < interval.length; i++){
        // 对每个间隔进行直接插入排序
            for (int j = interval[i]; j < array.length; j++ ) {
                int temp = array[j];
                for(int k = j; k >= interval[i] && array[k] > temp; k -=
interval[i]){
                    array[k] = array[k - interval[i]];
                }
                array[j  - interval[i]] = temp;
            }
        }
    }
}
```

快速排序

```
//快速排序（递归实现）
import java.util.Random;

public class QuickSort{
    void sort(int [] array){
        // 调用递归
        subSort(array, 0, array.length - 1);
    }
    void subSort(int [] array, int start, int end){
```

```
        // 随机选取轴值索引pivot
        Random r = new Random();
        // 利用Random类的nextInt()函数生成超出数组下标的随机正整数
        int pivot = r.nextInt(end);
        // 交换轴值与最后一个元素
        int temp = array[end];
        array[end] = array[start + pivot];
        array[start + pivot] = temp;
        // 将大于轴值的元素放在数组的右半边
        int curr = end;
        for (int i = end - 1; i > start; i--) {
    if(array[i] > array[end]){
      curr--;
      temp = array[curr];
      array[curr] = array[i];
      array[i] = temp;
    }
  }
  // 将轴值居中
  temp = array[end];
  array[end] = array[curr];
  array[curr] = temp;
  // 递归调用对左右两部分排序
  subSort(array, start, curr);
  subSort(array, curr + 1, end);
    }
}
```

归并排序

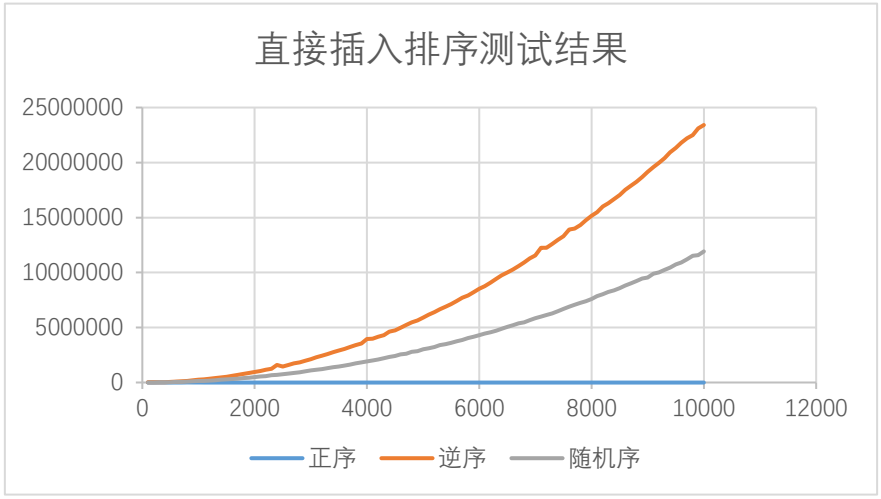
```
//归并排序
public class MergeSort {

    void sort(int[] array) {
        subSort(array, 0, array.length);
    }
```

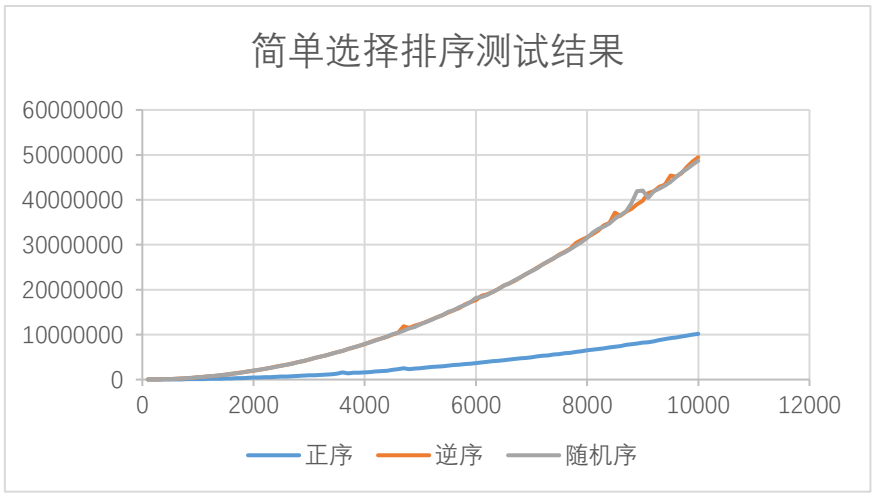
```
void subSort(int[] array, int l, int r) {
    int mid = (l + r) / 2;
    if(l == r) return;//只有一个元素时，返回
    //左边
    subSort(array, l, mid);
    //右边
    subSort(array, mid + 1, r);
    //归并两个有序数组
    int[] temp = new int[r - l + 1];
    int i = l;// 左指针
    int j = mid + 1;// 右指针
    int k = 0;

    // 把较小的数先移到新数组中
    while (i <= mid && j <= r) {
        if (array[i] < array[j]) {
            temp[k++] = array[i++];
        } else {
            temp[k++] = array[j++];
        }
    }

    // 把左边剩余的数移入数组
    while (i <= mid) {
        temp[k++] = array[i++];
    }

    // 把右边边剩余的数移入数组
    while (j <= r) {
        temp[k++] = array[j++];
    }

    // 把新数组中的数覆盖array数组
    for (int k2 = 0; k2 < temp.length; k2++) {
```

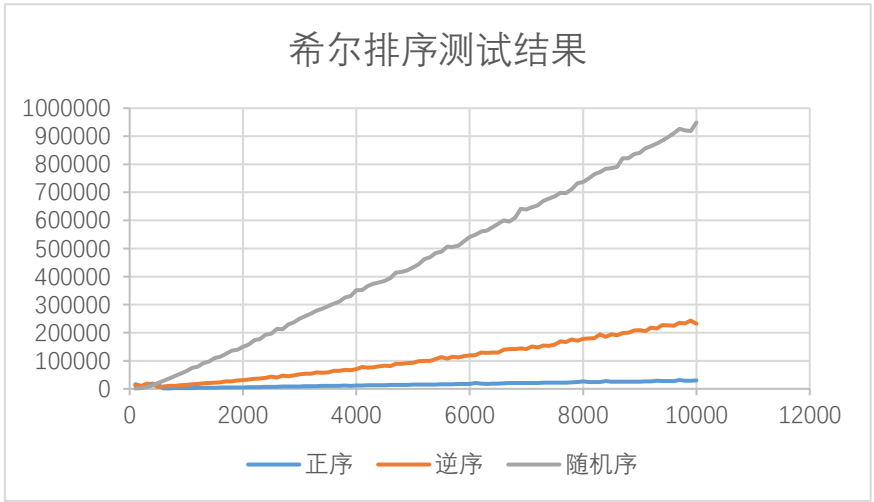
```
            array[k2 + 1] = temp[k2];
        }
    }
```

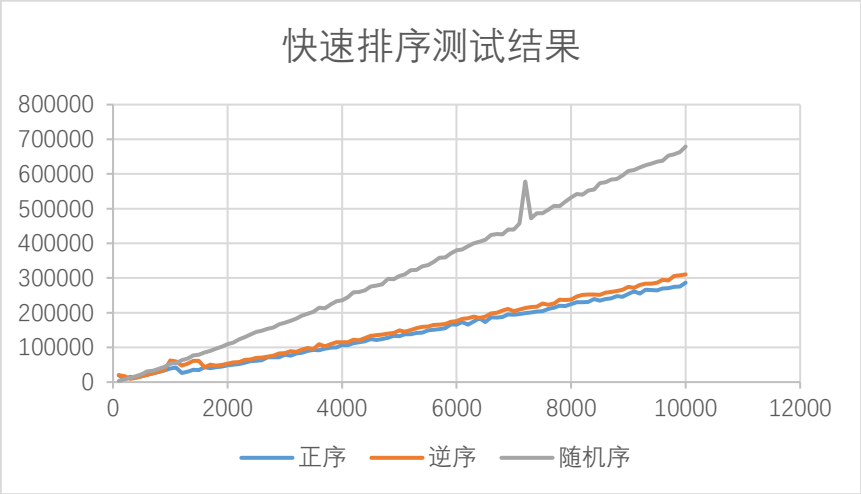
3.3 测试结果

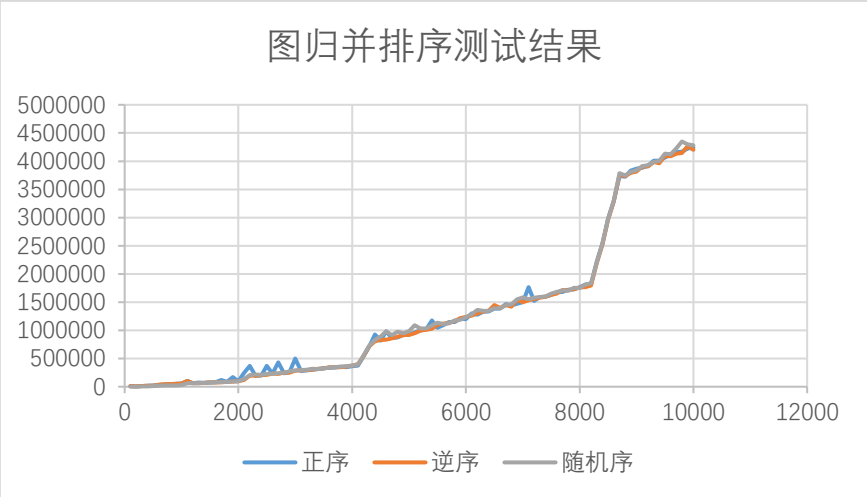
a、 每个排序算法在满足条件 A 和条件 B 情形下的运行时间趋势变化图



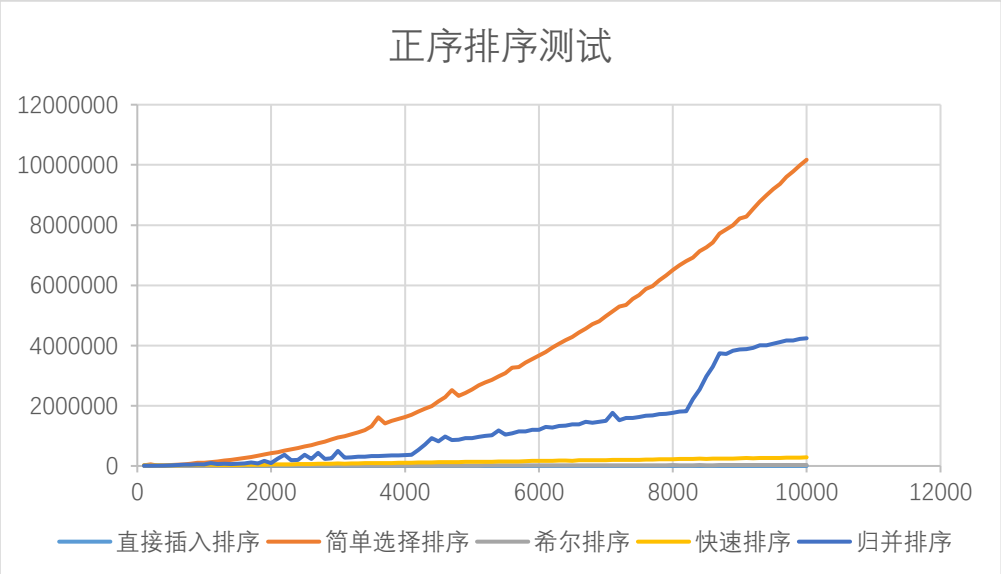


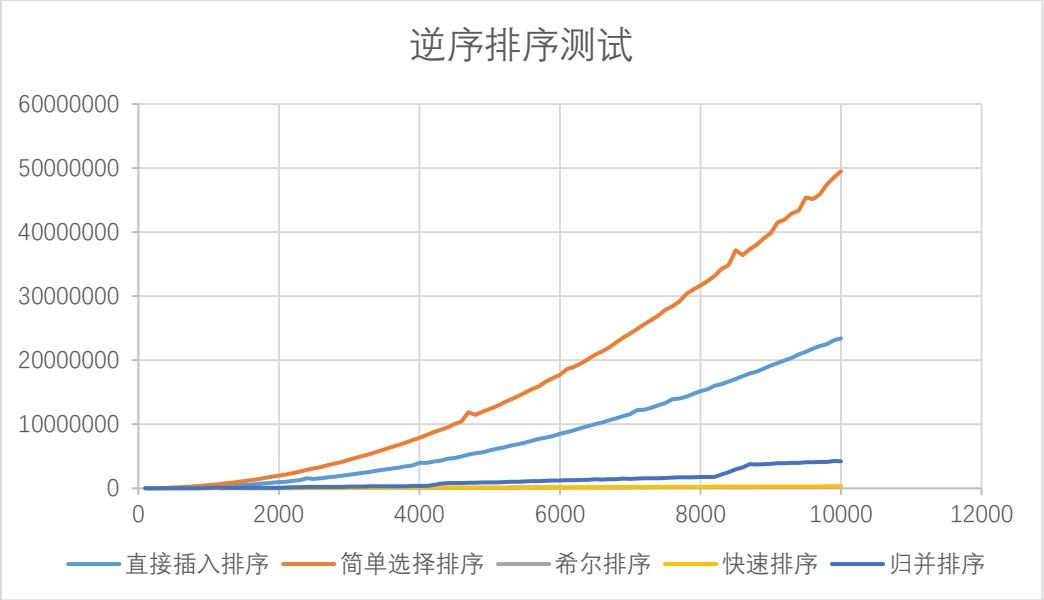


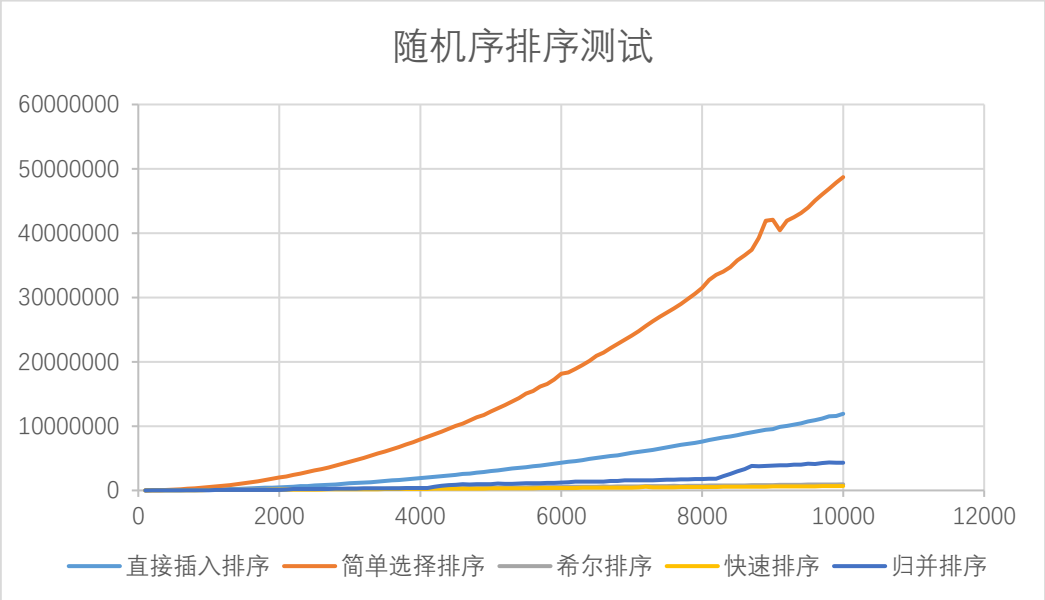




b、 将所有排序算法在正序下、逆序下和随机序列下的运行时间的对比图







3.4 算法分析

面对不同规模与顺序的数据时，不同的排序算法都有各自的优势。处理大量数据时，希尔排序、快速排序和归并排序的性能明显优于直接插入排序和简单选择排序。

4 寻找主元素

4.1 作业题目

> *大小为 N 的数组 A，其主元素是一个出现超过 N/2 次的元素（从而这样的元素最多有一个）。*
> *例如，数组 3，3，4，2，4，4，2，4，4 有一个主元素 4*
> *数组 3，3，4，，4，4，2，4 没有主元素。*
> *使用两种方法实现该问题的求解，并编写程序进行实现。*
> *同时给出两种求解的算法分析。*

4.2 程序实现

```
import java.util.Arrays;

//计数法

public class Solution1 {
    public int majorityElement(int[] arr) {

    int majorElem = arr[0]; // 用于记录主元素，假设第一个是主元素

    int count = 1; // 计数器


    for (int i = 1; i < arr.length; i++) { // 从第二个元素开始到最后一个元

素


    if (majorElem == arr[i]) { // 如果两个数相同就不能抵消

        count++;
        } else {
            if (count > 0) {
                count--;
            } else {
                majorElem = arr[i];
            }
        }
    }

    // 对于数组中可能没有主元素的情况

    count = 0;

    for (int a : arr) {//遍历数组中的每一个数并放在临时变量a中
```

```
            if (a == majorElem) {
                count++;
            }
        }
        if (count >= arr.length / 2) {
            return majorElem;
        } else {
            //若无主元素，抛出“无主元素”
            throw new RuntimeException("没有主元素！");
        }
    }
}

//排序法
public class Solution2 {
    public int majorityElement(int[] arr){
        //排序数组
        Arrays.sort(arr);
        int count = 1;
        int majorElem = arr[0];
        for(int i = 0; i<arr.length; i++){
            if(count > arr.length / 2) return majorElem;
            if(arr[i] == majorElem) count++;
            else{
                //重置计数器与主元素
                count = 1;
                majorElem = arr[i];
            }
        }
        if(count > arr.length / 2) return majorElem;
        else throw new RuntimeException("No majority element");
    }
}

//测试代码
public class MainElem {

    public static void main(String[] args) {
        int [] a = {11, 33, 44, 44, 44, 44, 44, 6, 3};
        Solution1 s1 = new Solution1();
```

```
        Solution2 s2 = new Solution2();
        System.out.println(s1.majorityElement(a));
        System.out.println(s2.majorityElement(a));
    }
}
```

4.3 算法分析

算法 1（Solution1）：利用计数器，假设数组第一个元素为主元素，然后对数组进行遍历，若遇到相同元素，计数器加一；若遇到不同元素，计数器减一，减至负值时，更换主元素。最后再统计最终的主元素出现次数是否大于（数组长度）/ 2 来判断是否存在主元素。该算法的时间复杂度为 O(n);

算法 2（Solution2）：利用 Arrays.sort()对数组进行遍历，统计各个元素出现的次数，以次确定主元素。该算法的时间复杂度为 O(nlongn)。

5 寻找第 k 大元素

5.1 作业题目

> *选择问题：在一组数据中选择第 k 大数据的问题。请给出你的解决方法，并给出该解决方案的时间性能分析。*

5.2 程序实现

```
import java.util.Arrays;
public class Solution {

    //剔除已排序数组的重复元素

    void removeDuplicates(int [] arr, int n) {
        int j = 0;
        for (int i = 1; i < n; ++i) {
            if (arr[i] != arr[j]) {
                arr[++j] = arr[i];
            }
        }
    }
    int FindK(int [] arr, int k){
        if(arr.length == 1) return arr[1];

        //对数组进行排序

        Arrays.sort(arr);
        removeDuplicates(arr, arr.length);
```

```
        int count = arr.length;
        if(k > count) {throw new Error ("k值越界！");}
        else return arr[k-1];
        }
}
//测试代码
public class FindK {
    public static void main(String[] args) {
        int [] arr = {1,2,3,3,2,4,4};
        Solution s= new Solution();
        System.out.println(s.FindK(arr, 2));
}
}
```

5.3 算法分析

经测试，该算法能够得到正确结果。且算法的时间性能主要又 Array.sort()排序过程的时间开销决定：

遍历数组以及剔除已排序数组的重复元素的时间开销都在 O(n)中，而排序的时间开销在 O(nlogn)中，故该算法的平均时间开销$T(n) = \begin{cases} O(1) & n = 1 \\ O(nlogn) & 其他情况 \end{cases}$