

# 数据结构第一次作业

沈俞霖

2016 年 10 月 6 日

课程 数据结构与算法

姓名 沈俞霖

专业班级 软件 51

学号 2151601013

邮箱 sylxjtu@outlook.com

提交日期 2016 年 10 月 6 日

## 目录

<b>1</b>	<b>复杂度与时间估计</b>	<b>3</b>
1.1	作业题目	3
1.2	问题分析	3
<b>2</b>	<b>递归方程求复杂度</b>	<b>3</b>
2.1	作业题目	3
2.2	问题分析	4
<b>3</b>	<b>测试排序算法</b>	<b>5</b>
3.1	作业题目	5
3.2	程序实现	6
3.3	程序运行结果与时间统计	12
3.4	算法分析	16
<b>4</b>	<b>选取主元素</b>	<b>16</b>
4.1	作业题目	16
4.2	程序实现	17
4.3	程序运行结果与时间统计	19
4.4	算法分析	20
<b>5</b>	<b>选择问题</b>	<b>20</b>
5.1	作业题目	20
5.2	程序实现	20
5.3	程序运行结果与时间统计	22
5.4	算法分析	22

## 1 复杂度与时间估计

### 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  $150N \log N$  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 是否总比程序 A 运行得快?

### 1.2 问题分析

- A A 程序运行时间有更好的保障。当  $N = 10000$  时, 在最坏情况下,  $150N \log N \approx 2 \times 10^7$ , 而  $N^2 = 10^8$ , 故 A 程序运行时间有更好的保障。
- B B 程序运行时间有更好的保障。当  $N = 100$  时, 在最坏情况下,  $150N \log N \approx 10^5$ , 而  $N^2 = 10^4$ , 故 B 程序运行时间有更好的保障。
- C 不能判断, 因为题目只给出了最坏情况运行时间, 没有给出平均情况运行时间。
- D 不能判断, 因为题目只给出了最坏情况运行时间, 不能以此来判断所有的输入。

## 2 递归方程求复杂度

### 2.1 作业题目

考虑以下递归方程, 定义函数  $T(n)$  :

**A**

$$T(n) = \begin{cases} 1, & \text{如果 } n = 1 \\ T(n-1) + n, & \text{其他情况} \end{cases} \quad (1)$$

**B**

$$T(n) = \begin{cases} 1, & \text{如果 } n = 0 \\ 2T(n-1), & \text{其他情况} \end{cases} \quad (2)$$

请给出 A 和 B 两种递归式的大 O 表示, 并证明。

## 2.2 问题分析

**A**  $T(n) = O(n^2)$ 

**证明** (1) 令  $n = 1$ ,

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

(2) 若

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

则  $T(k+1) = T(k) + k + 1$ , 即

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

由 (1)(2) 可知,

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

故  $T(n) = O(n^2)$ , 证毕。

**B**  $T(n) = O(2^n)$ 

**证明** (1) 令  $n = 0$ ,

$$T(n) = 1 = 2^n$$

(2) 若

$$T(k) = 2^k$$

则  $T(k+1) = 2T(k)$ , 即

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

由 (1)(2) 可知,

$$T(n) = 2^n$$

故  $T(n) = O(2^n)$ , 证毕。

### 3 测试排序算法

#### 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:

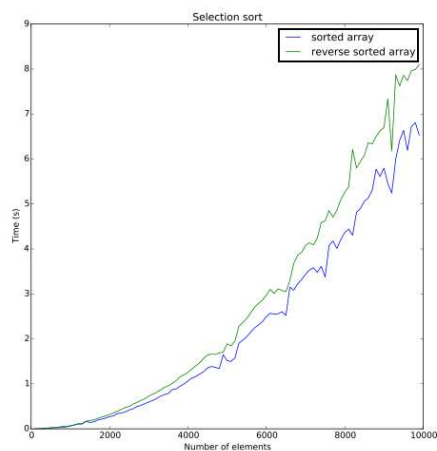
**A** 测试数组的大小为  $[100, 200, 300, \dots, 10000]$  100 种大小

**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:

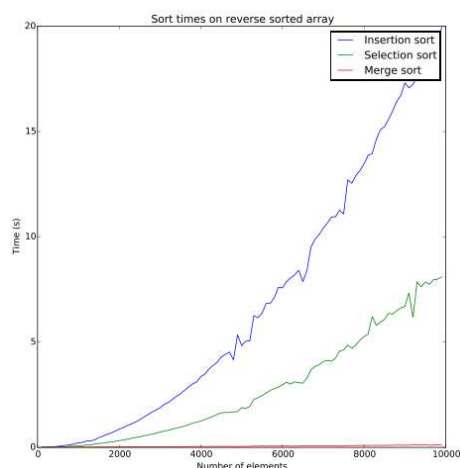
**a** 展示每个排序算法在满足条件 A 和条件 B 情形下的运行时间趋势变化图, 如图 1 所示

图 1: 对选择排序的分析



**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

图 2: 对逆序排序的分析



### 3.2 程序实现

仅列出算法代码，完整代码详见 Github 代码仓库 <sup>1</sup>

#### 插入排序

```
// InsertionSort.java 插入排序
public class InsertionSort extends SortFunction{
    InsertionSort(DataGenerator gen, int size){
        super(gen, size);
    }
    void sort(int[] arr){
        for (int cur = 1; cur < arr.length; cur++) {
            // 在当前值之前的元素都已排序
            int t = arr[cur];
            int nxt = cur;
            // 将所有大于当前值的已排序元素右移
```

<sup>1</sup><https://github.com/sylxjtu/datastructure-homework/tree/master/homework1/src>

```
        for (; nxt >= 1 && arr[nxt - 1] > t; nxt--) {
            arr[nxt] = arr[nxt - 1];
        }
        // 将当前值归位
        arr[nxt] = t;
    }
}
```

### 选择排序

```
// SelectionSort.java 选择排序
public class SelectionSort extends SortFunction{
    SelectionSort(DataGenerator gen, int size){
        super(gen, size);
    }
    void sort(int[] arr){
        for (int i = 0; i < arr.length; i++) {
            // 初始化当前最小元素及其位置
            int curm = arr[i];
            int curmi = i;
            // 遍历数组，寻找最小元素
            for (int j = i + 1; j < arr.length; j++) {
                if(arr[j] < curm){
                    curm = arr[j];
                    curmi = j;
                }
            }
            // 将当前元素与最小元素交换
            int t = arr[i];
            arr[i] = arr[curmi];
            arr[curmi] = t;
        }
    }
}
```

```
}
```

### 希尔排序

// ShellSort.java 希尔排序

```
public class ShellSort extends SortFunction{
    ShellSort(DataGenerator gen, int size){
        super(gen, size);
    }
    void sort(int[] arr){
        // 使用 Marcin 间隔序列
        int[] gaps = new int[]{701, 301, 132, 57, 23, 10, 4, 1};
        for(int i = 0; i < gaps.length; i++){
            // 对每个间隔进行一次插入排序
            for(int j = gaps[i]; j < arr.length; j++){
                int t = arr[j];
                int k = j;
                for( ; k >= gaps[i] && arr[k - gaps[i]] > t; k -= gaps[i]){
                    arr[k] = arr[k - gaps[i]];
                }
                arr[k] = t;
            }
        }
    }
}
```

### 快速排序

// QuickSort.java 快速排序

```
import java.util.Random;

public class QuickSort extends SortFunction{
    QuickSort(DataGenerator gen, int size){
        super(gen, size);
    }
}
```



```
void sort(int[] arr){
    // 调用递归的排序方法
    rsort(arr, 0, arr.length);
}
// 递归快速排序
void rsort(int[] arr, int l, int r){
    // 在元素个数小于等于 16 时调用插入排序
    if(r - l <= 16){
        isort(arr, l, r);
        return;
    }
    // 随机选取轴值并与第一个元素交换
    Random ran = new Random();
    int pivot = ran.nextInt(r - l);
    int t = arr[l + pivot];
    arr[l + pivot] = arr[l];
    arr[l] = t;
    int cur = l;
    // 选取所有小于轴值的元素放到数组左半部分
    for (int i = l + 1; i < r; i++) {
        if(arr[i] < arr[l]){
            cur++;
            t = arr[cur];
            arr[cur] = arr[i];
            arr[i] = t;
        }
    }
    // 将轴值放到数组中间
    t = arr[cur];
    arr[cur] = arr[l];
    arr[l] = t;
    // 递归对左右两部分进行排序
    rsort(arr, l, cur);
}
```

```
        rsort(arr, cur + 1, r);
    }
    // 插入排序
    void isort(int[] arr, int l, int r){
        for (int cur = l + 1; cur < r; cur++) {
            int t = arr[cur];
            int nxt = cur;
            for (; nxt >= l + 1 && arr[nxt - 1] > t; nxt--) {
                arr[nxt] = arr[nxt - 1];
            }
            arr[nxt] = t;
        }
    }
}
```

归并排序

```
// MergeSort.java 归并排序
public class MergeSort extends SortFunction{
    MergeSort(DataGenerator gen, int size){
        super(gen, size);
    }
    void sort(int[] arr){
        // 调用递归的排序方法
        rsort(arr, 0, arr.length);
    }
    // 递归归并排序
    void rsort(int[] arr, int l, int r){
        // 在元素个数小于等于 16 时调用插入排序
        if(r - l <= 16){
            isort(arr, l, r);
            return;
        }
        // 把数组均分，递归进行排序
```

```
rsort(arr, l, (l + r) / 2);
rsort(arr, (l + r) / 2, r);

// 申请临时空间
int[] arr2 = new int[arr.length];
int arr2p = 0;

// 归并两个有序数组
int ls = l, le = (l + r) / 2, es = (l + r) / 2, ee = r;
while(ls < le && es < ee){
    if(arr[ls] < arr[es]){
        arr2[arr2p++] = arr[ls++];
    }
    else{
        arr2[arr2p++] = arr[es++];
    }
}
while(ls < le){
    arr2[arr2p++] = arr[ls++];
}
while(es < ee){
    arr2[arr2p++] = arr[es++];
}

// 将临时结构拷贝回原数组
System.arraycopy(arr2, 0, arr, 0, arr.length);
}

// 插入排序
void isort(int[] arr, int l, int r){
    for (int cur = l + 1; cur < r; cur++) {
        int t = arr[cur];
        int nxt = cur;
        for (; nxt >= l + 1 && arr[nxt - 1] > t; nxt--) {
```

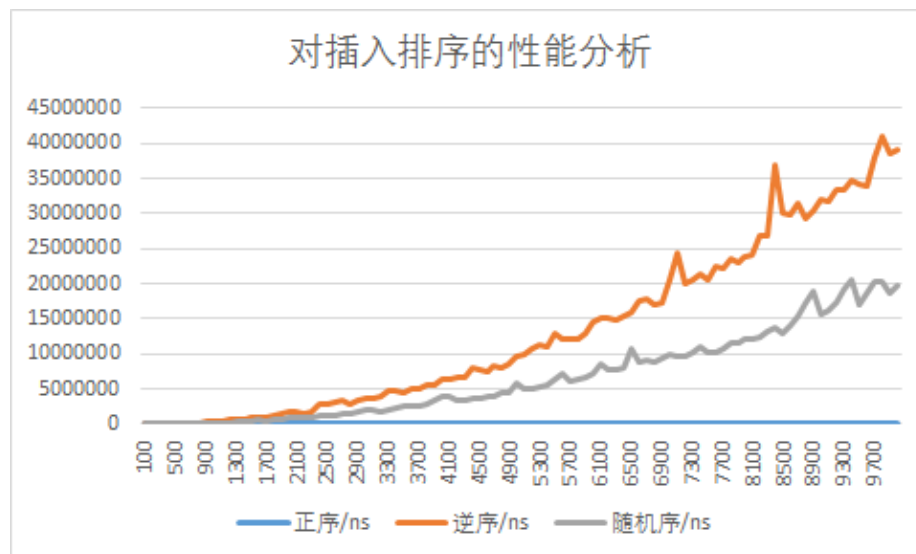
```
        arr[nxt] = arr[nxt - 1];  
    }  
    arr[nxt] = t;  
}  
}
```

### 3.3 程序运行结果与时间统计

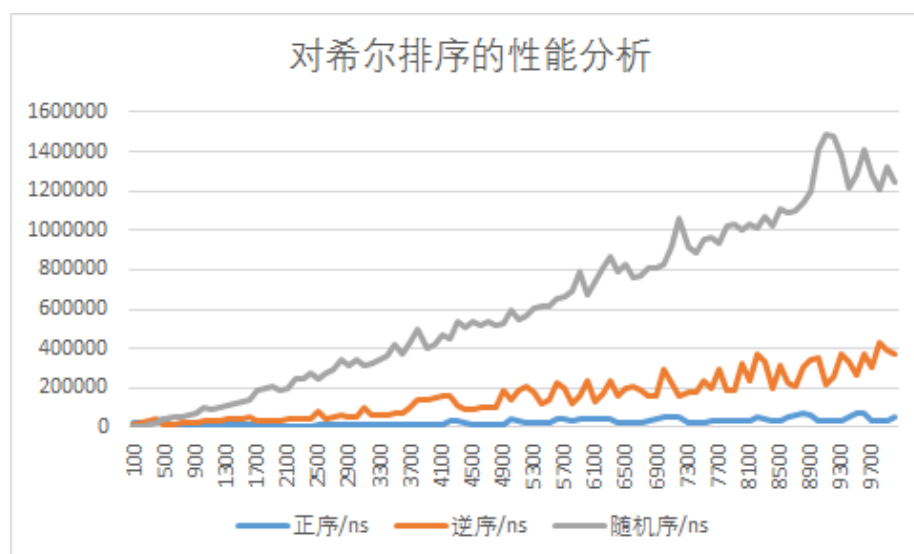
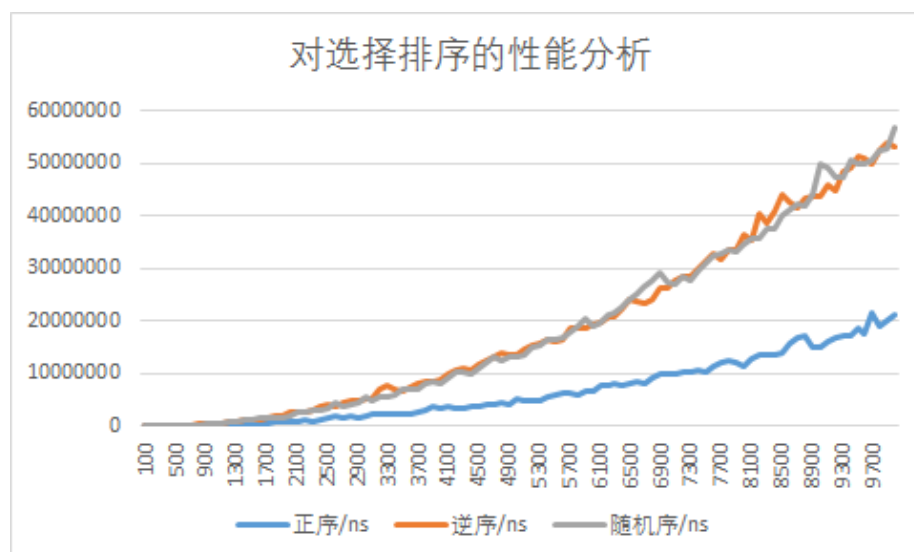
**运行结果** 所有排序程序全部通过正确性测试

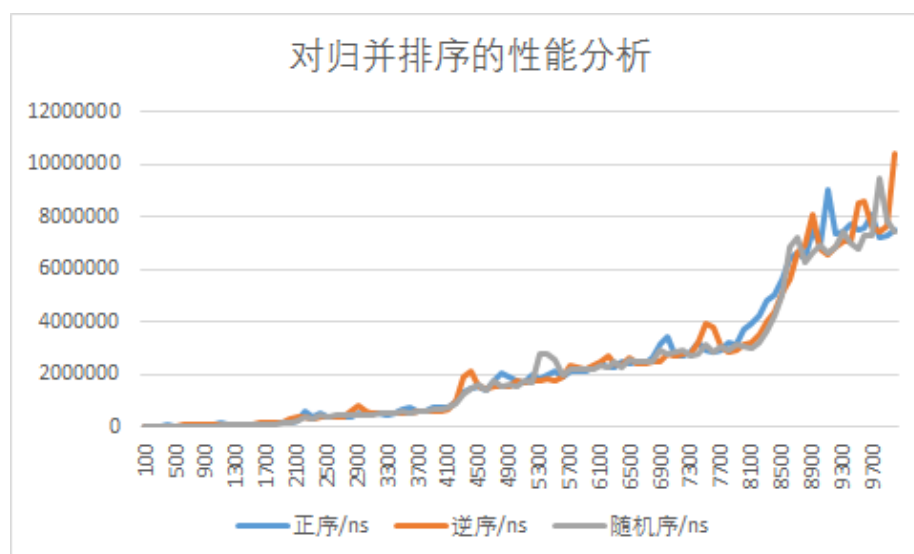
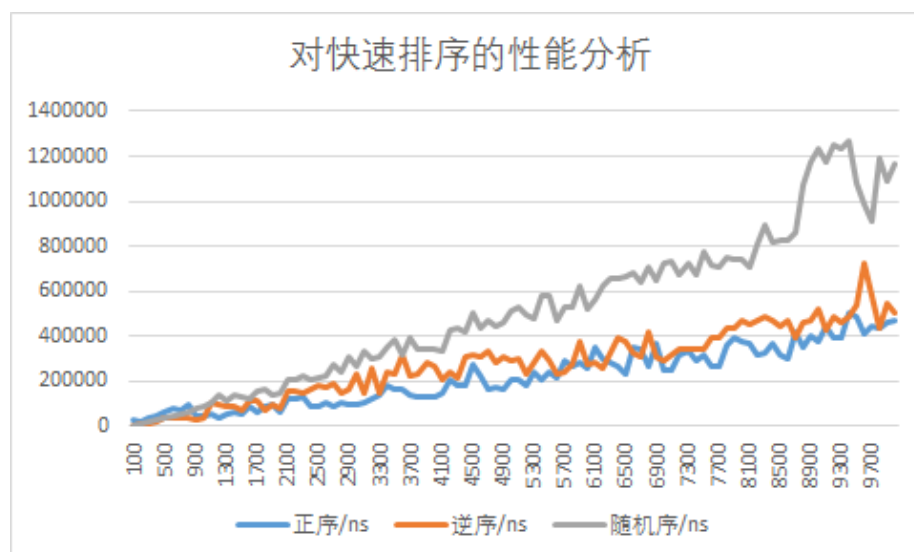
**时间统计** 以图表形式给出，详细数据详见 Github 代码仓库<sup>2</sup>

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

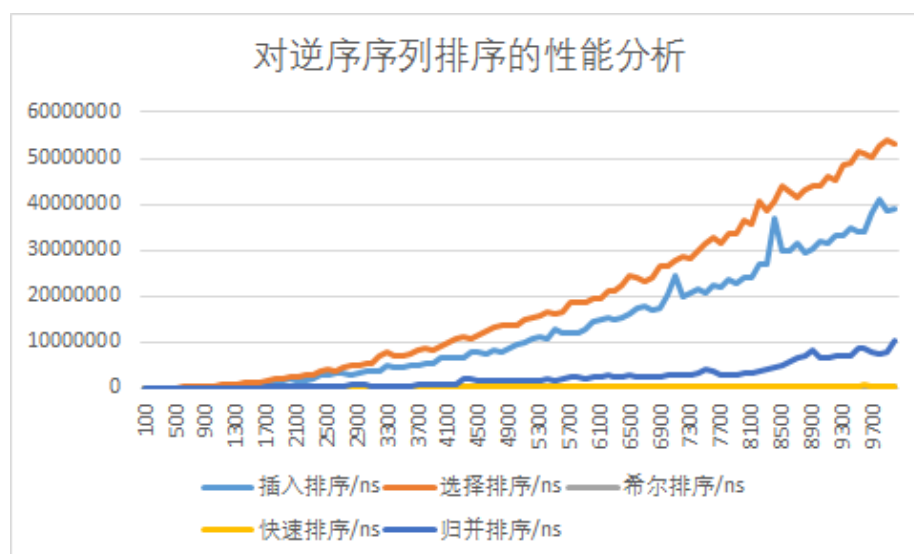
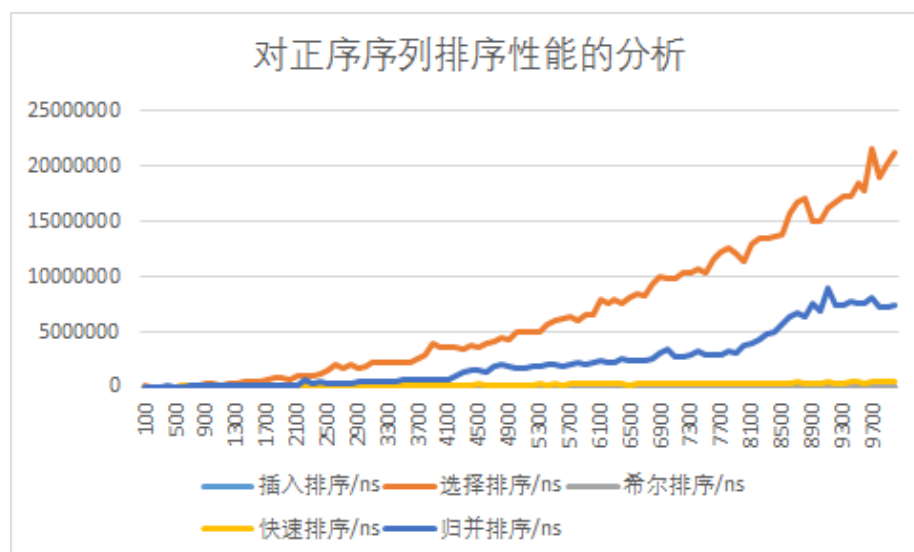


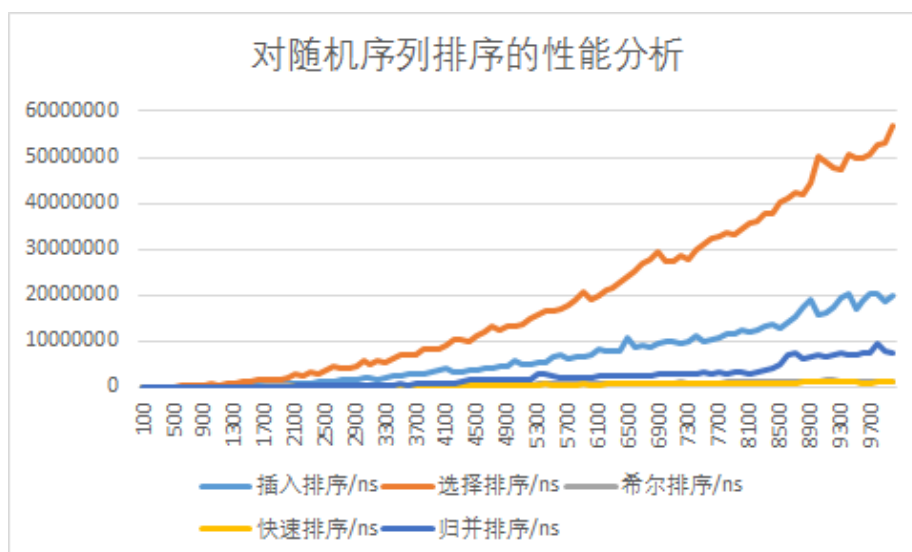
<sup>2</sup><https://github.com/sylxjtu/datastructure-homework/blob/master/homework1/data/out.txt>





b 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

每个算法在不同数据条件下都有一些不同的性能表现，插入排序和希尔排序的性能表现在正序时明显优于其他情况，而归并排序的时间复杂度最为稳定，三种情况下所需时间基本相等。

在处理大量数据时，希尔排序、快速排序和归并排序速度明显优于插入排序和选择排序的速度，表现了复杂度低的算法在面对大量数据时的优越性。

## 4 选取主元素

### 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

使用了两种求解方法，分别是排序后顺序查找的方法和用哈希表统计后再查找的方法。

代码

```
// Problem4.java 第 4 题-选取主元素
import java.util.HashMap;
import java.util.Arrays;

// 第一种算法-排序求解
class SolutionSort{
    static int mainElem(int[] arr){
        // 长度为 0 , 没有主元素
        if(arr.length == 0){
            throw new Error("No main element");
        }
        // 对数组进行排序
        Arrays.sort(arr);
        // 遍历数组找到每个元素出现次数
        int last = arr[0];
        int cnt = 1;
        for(int i = 1; i < arr.length; i++){
            if(cnt > arr.length / 2) return last;
            if(arr[i] == last) cnt++;
            else{
                cnt = 1;
                last = arr[i];
            }
        }
        if(cnt > arr.length / 2) return last;
        // 没有一个元素出现次数大于 N/2 , 没有主元素
        else throw new Error("No main element");
    }
}
```

```
}

// 第二种算法-使用哈希表
class SolutionHash{
    static int mainElem(int[] arr){
        // 长度为 0 , 没有主元素
        if(arr.length == 0){
            throw new Error("No main element");
        }
        // 建立一个 int 到 int 的哈希表
        HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>();
        // 对每个元素计数
        for(int i = 0; i < arr.length; i++){
            hm.put(arr[i], hm.getOrDefault(arr[i], 0) + 1);
        }
        // 对每个元素的计数进行判断
        for(int i = 0; i < arr.length; i++){
            if(hm.get(arr[i]) > arr.length / 2) return arr[i];
        }
        // 没有找到主元素
        throw new Error("No main element");
    }
}

// 测试代码
public class Problem4{
    public static void main(String[] args) {
        System.out.println(SolutionSort.mainElem(
            new int[]{1, 1, 3, 3, 2, 2, 2, 2, 2}
        ));
        System.out.println(SolutionHash.mainElem(
            new int[]{1, 1, 3, 3, 2, 2, 2, 2, 2}
        ));
    }
}
```

```
DataGenerator ran = new RandomGenerator();
int[] arr = ran.generate(1000000);
long n1 = System.nanoTime(), n2;
try{
    System.out.println(SolutionSort.mainElem(arr));
}
catch(Error e){
    System.out.println(" 没有主元素");
}
finally{
    n2 = System.nanoTime();
}
System.out.println(n2 - n1);
long n3 = System.nanoTime(), n4;
try{
    System.out.println(SolutionHash.mainElem(arr));
}
catch(Error e){
    System.out.println(" 没有主元素");
}
finally{
    n4 = System.nanoTime();
}
System.out.println(n4 - n3);
}
```

### 4.3 程序运行结果与时间统计

运行结果 两种解法均通过测试

运行时间 使用  $10^6$  个随机数据进行最坏情况运行时间测试

解法 1 运行时间  $1.86 \times 10^8 \text{ns}$

解法 2 运行时间  $8.50 \times 10^8 \text{ns}$

#### 4.4 算法分析

解法 1 解法 1 理论复杂度为  $O(n \log n)$  (排序) +  $O(n)$  (遍历), 即为  $O(n \log n)$ 。

解法 2 解法 2 理论复杂度为  $3nO(1)$  ( $2n$  次哈希表查找,  $n$  次哈希表修改), 即为  $O(n)$ , 由于哈希表本身的常数复杂度使得该解法求解较慢。

## 5 选择问题

### 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

按照分治的策略，随机选取轴值，把元素划分成小于轴值、大于等于轴值和轴值本身三部分，分类讨论目标元素可能处在的部分，并继续查找。

代码

// Problem5.java 第 5 题-选择问题

```
import java.util.Random;
```

```
class Problem5{
```

```
    // 求解函数
```

```
    static int kthElement(int[] arr, int k){
```

```
        // 验证  $k$  的合法性
```

```
        if(k > arr.length || k < 1) throw new Error("k must in [1, arr.length]");
```

```
        // 调用递归求解方法
```

```
        return kthElement_r(arr, 0, arr.length, k);
```

```
    }
```

```
    static int kthElement_r(int[] arr, int l, int r, int k){
```

```
// 当数组只有一个元素时，答案即此元素
if(r - l == 1){
    return arr[l];
}
// 随机选取轴值并与第一个元素交换
Random ran = new Random();
int pivot = ran.nextInt(r - l);
int t = arr[l + pivot];
arr[l + pivot] = arr[l];
arr[l] = t;
int cur = l;
// 选取所有小于轴值的元素放到数组左半部分
for (int i = l + 1; i < r; i++) {
    if(arr[i] < arr[l]){
        cur++;
        t = arr[cur];
        arr[cur] = arr[i];
        arr[i] = t;
    }
}
// 将轴值放到数组中间
t = arr[cur];
arr[cur] = arr[l];
arr[l] = t;
// 分类讨论所求元素所在位置
if(k - 1 < cur - l){
    return kthElement_r(arr, l, cur, k);
}
else if(k - 1 == cur - l){
    return arr[cur];
}
else{
    return kthElement_r(arr, cur + 1, r, k - (cur - l + 1));
}
```

```

    }
}
// 测试代码
public static void main(String[] args) {
    Random r = new Random();
    DataGenerator dg = new RandomGenerator();
    int[] arr = dg.generate(1000000);
    long s = System.nanoTime();
    for(int i = 1; i <= 100; i++){
        kthElement(arr, r.nextInt(1000000) + 1);
    }
    long e = System.nanoTime();
    System.out.println(e - s);
}
}

```

### 5.3 程序运行结果与时间统计

运行结果 解法通过了测试，有正确性

运行时间  $10^6$  个数据，100 次查询，最终用时  $3.51 \times 10^8$  ns

### 5.4 算法分析

在平均情况下，对大小为  $n$  的数组，程序每次运行，都以  $O(n)$  复杂度遍历了整个数组，并且递归调用了自身，数量级为  $n/2$ ，可以得出递归方程为

$$T(n) = \begin{cases} O(1), & \text{如果 } n = 1 \\ T(n/2), & \text{其他情况} \end{cases} \quad (3)$$

根据等比数列求和公式， $T(n) = 2nO(1) = O(n)$

在最坏情况下，递归调用的数据大小为  $n - 1$ ，可以得出递归方程为

$$T(n) = \begin{cases} O(1), & \text{如果 } n = 1 \\ T(n - 1), & \text{其他情况} \end{cases} \quad (4)$$

根据等差数列求和公式,  $T(n) = \frac{n^2+n}{2}O(1) = O(n^2)$

由于轴值随机选择, 因此最坏情况基本不可能发生, 该算法是有时间保障的。