

**《数据结构课程设计》报告**

**题 目** 综合排序系统

**院 系** 应用技术学院

**年级班级** 2019计科1班

**学生姓名** 成凯

**学 号** 201833050025

**学 期** 2021-2022（一）

**任课教师** 黄 群

**二Ｏ二一年 十二 月十 日**

目录

[1. 引言 3](#_Toc1056728085)

[1.1研究背景 3](#_Toc739163905)

[1.2研究意义 3](#_Toc2082337087)

[2. 需求分析 3](#_Toc298426050)

[3. 概要设计 3](#_Toc1272306605)

[3.1功能模块设计 3](#_Toc1162437056)

[3.2主函数的流程设计： 4](#_Toc1420863433)

[4. 详细设计 5](#_Toc433563791)

[4.1算法分析 5](#_Toc494621066)

[4.1.1 直接插入排序 5](#_Toc187058725)

[4.1.2 希尔排序 7](#_Toc2127415514)

[4.1.3 冒泡排序 9](#_Toc2017304895)

[4.1.4 快速排序 11](#_Toc371551429)

[4.1.5 二分排序 13](#_Toc1929905374)

[5. 测试数据和测试结果 15](#_Toc326616530)

[5.1测试数据 15](#_Toc475817978)

[5.2测试结果 15](#_Toc1991138465)

[5.3结果分析 17](#_Toc826510054)

[5.4总结 17](#_Toc1230248782)

[参考文献 18](#_Toc818725758)

# 引言

## 1.1研究背景

随着科学技术的不断发展，计算机的应用领域越来越广。但是，由于计算机硬件的速度和存储空间的有限，如何提高计算机速度和节省存储空间一直是软件编译器努力的方向。在众多措施中，排序操作成为了程序员考虑的因素之一。排序方式的选择直接影响程序执行的速度和辅助存储空间的大小，进而影响整个软件的性能。

## **1.2研究意义**

排序算法有很多种，它们的复杂度也不同。在数据量较小的情况下，很难区分它们的优劣，但是在分析大量数据时，不同算法所花费的时间和空间是不同的。一般数据处理任务有 25%的时间都用在排序上。通过对排序算法的比较和研究，可以在编写程序时采用适当的排序方法，从而显著提高数据处理的效率。

# 需求分析

使用六种排序算法分别对随机生成的六组数据进行排序，给每种算法添加相应的标志来直观感受每种算法的优劣情况，该系统需要实现以下功能：

(1) 编写排序算法：直接插入排序、希尔排序、冒泡排序、快速排序、二分排序；

(2) 编写随机数生成函数，以及数组遍历函数。

(3) 编写主函数，给用户一个选择菜单，并调用各种算法。

# 概要设计

## **3.1功能模块设计**

(1) 随机数生成模块

(2) 五个排序算法模块

(3) 数组遍历模块

(4) 排序算法比较次数和关键字移动次数和耗时计算模块

(5) 主函数模块

## **3.2主函数的流程设计：**

综合排序系统的执行流程如图1功能模块图所示：

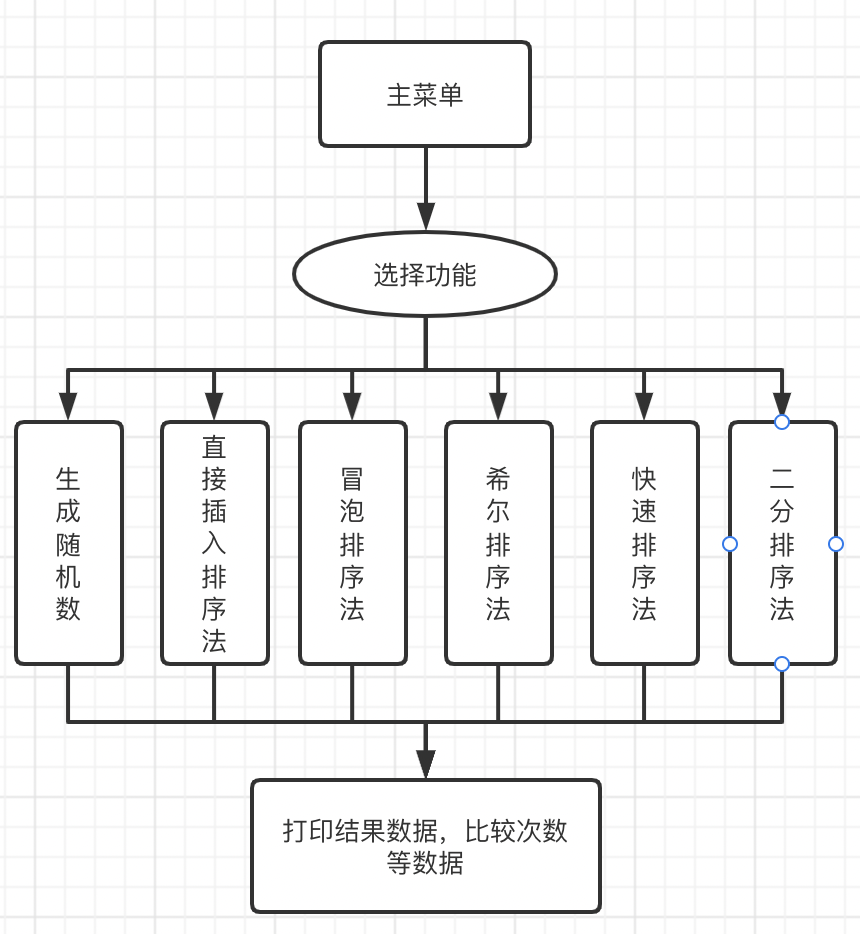


图1 功能模块图

程序运行后会自动生成一批随机数，同时用户可以重新生成新的随机数。功能按键的有效键值为1~6之间的任意数值，分别代表对应的功能，其他任何按键均代表退出程序，每个功能按键对应执行的功能为：

“1”：直接插入排序，调用insertionSort()算法对数组进行排序，再调用printCost()打印排序比较次数以及时间耗费和printArray()方法打印数组;

“2”： 希尔排序，先调用shellSort()算法进行排序，再调用printCost()打印排序比较次数以及时间耗费和printArray()方法打印数组;

“3”： 冒泡排序，先调用bubbleSort()算法进行排序，再调用printCost()打印排序比较次数以及时间耗费和printArray()方法打印数组;

“4”： 快速排序，先调用quickSort()方法，quickSort()方法中调用split()方法来划分并获取比较元素的位置，split()方法再调用swap()方法，将比较元素交换到正确的位置上。最后再调用printCost()打印排序比较次数以及时间耗费和printArray()方法打印数组;

“5”：二分排序，先调用binaryInsertSort()方法进行排序，再调用printCost()打印排序比较次数以及时间耗费和printArray()方法打印数组;

“其他任意按键”：退出程序；

# 详细设计

4.1算法分析

4.1.1 直接插入排序

函数原型：insertionSort();

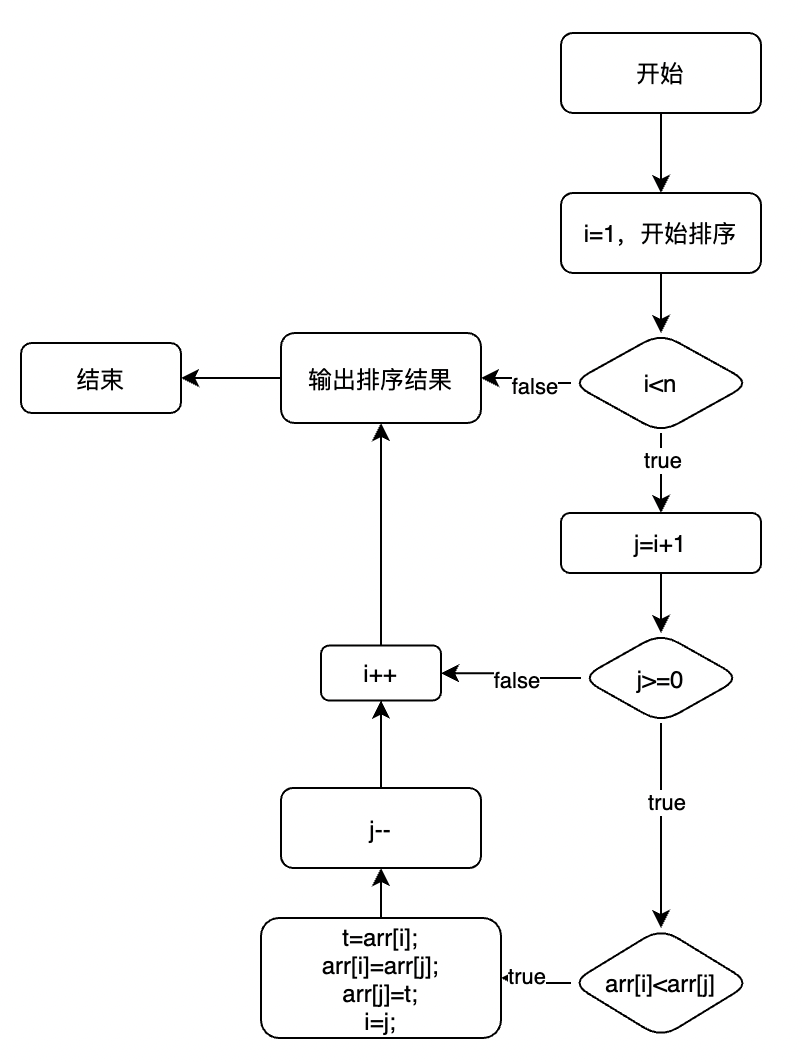


图2 直接插入排序流程图

基本思想：插入排序基本思想是每一步将一个待排序的记录，插入到前面已经排好序的有序序列中去，直到插完所有元素为止。

算法描述：

1. 从第一个元素开始，可以认为该元素已经排序；
2. 取出下一个元素，按照元素排序后的顺序从后往前扫描；
3. 如果元素（已排序）大于新元素，则将元素移动到下一个位置；
4. 重复步骤3，直到找到排序元素小于等于新元素的位置；
5. 在这个位置插入新元素后；重复步骤 2~5。

实现代码：

public static void insertionSort(int[] arr) {

int count = 0;

int transferCount = 0;

long start = System.currentTimeMillis();

// 从下标为1的元素开始选择合适的位置插入，因为下标为0的只有一个元素，默认是有序的

for (int i = 1; i < arr.length; i++) {

// 记录要插入的数据

int temp = arr[i];

// 从已经排序的序列最右边的开始比较，找到比其小的数

int index = i - 1;

while (index >= 0 && temp < arr[index]) {

arr[index + 1] = arr[index];

// 存在比其小的数，插入

arr[index] = temp;

index--;

transferCount++;

count++;

}

count++;

}

printCost(count, transferCount, start);

printArray(arr);

}

4.1.2 希尔排序

函数原型：shellSort();

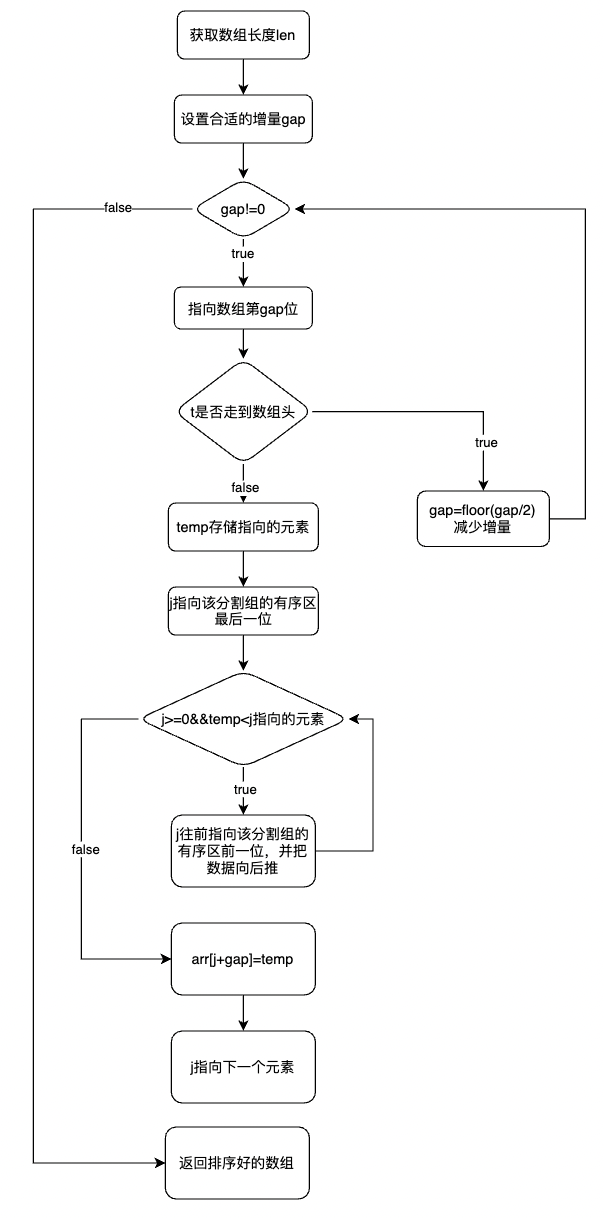


图4 希尔排序流程图

基本思想：希尔排序是将记录按目标的一定增量进行分组，使用直接插入排序算法对每组进行排序； 随着增量的减少，每组包含越来越多的关键字。 当增量减小到 1 时，整个文件被分成一组，算法终止。

希尔排序在数组中采用跳跃式分组的策略，通过某个增量将数组元素划分为若干组，然后分组进行插入排序，随后逐步缩小增量，继续按组进行插入排序操作，直至增量为1。选择增量gap=length/2，缩小增量继续以gap = gap/2的方式，这种增量选择可以用序列表示为：{n/2,(n/2)/2...1}

实现代码：

public static void shellSort(int[] arr) {

int count = 0;

int transferCount = 0;

long start = System.currentTimeMillis();

// 每次都将增量除以2

for (int gap = arr.length; gap > 0; gap /= 2) { // gap/=2 表示将增量再缩小一倍

for (int i = gap; i < arr.length; i++) {

count++;

int temp= arr[i];

int index = i - gap;

while (index >= 0 && temp< arr[index]) {

// 数组arr在index+gap位置的等于在index位置的元素

arr[index + gap] = arr[index];

// 将要插入的值替换进数组在index位置的元素

arr[index] = temp;

// 计算增量

index -= gap;

transferCount++;

}

}

}

printCost(count, transferCount, start);

printArray(arr);

}

4.1.3 冒泡排序

函数原型：bubbleSort();

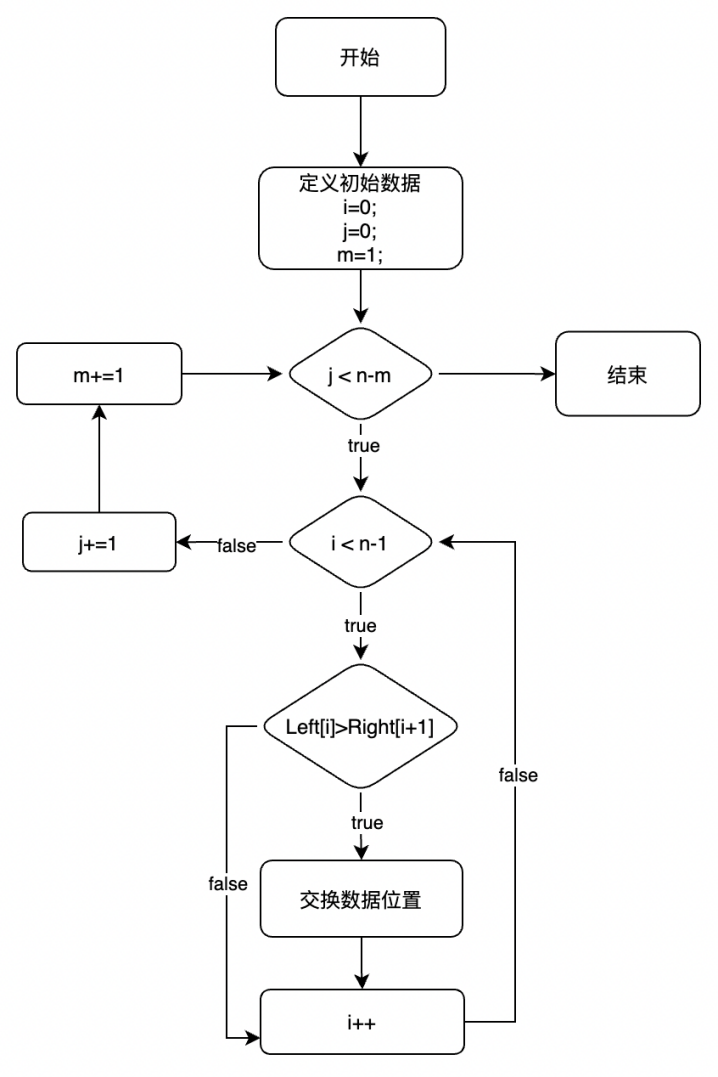


图3 冒泡排序流程图

基本思想：冒泡排序的基本思想是将相邻元素成对比较，并以相反的顺序交换。 这样每次都会把最小或最大的元素“浮动”到最上面，最终实现完全排序。

算法描述：

1. 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；

3. 针对所有的元素重复以上的步骤，除了最后一个；重复步骤1~3，直到排序完成。

实现代码：

private static void bubbleSort(int[] arr) {

int count = 0;

int transferCount = 0;

long start = System.currentTimeMillis();

int len = arr.length;

if (len == 0 || len == 1) {

return;

}

// 开始遍历数组

for (int i = 0; i < len; i++) {

// 初始位置

for (int j = 0, subLen = len - 1 - i; j < subLen; j++) {

// 开始两两比较

count++;

if (arr[j + 1] < arr[j]) {

// 比较完成，交换大的数字到后面

int tmp = arr[j + 1];

arr[j + 1] = arr[j];

arr[j] = tmp;

transferCount++;

}

}

}

printCost(count, transferCount, start);

printArray(arr);

}

4.1.4 快速排序

函数原型：quickSort();

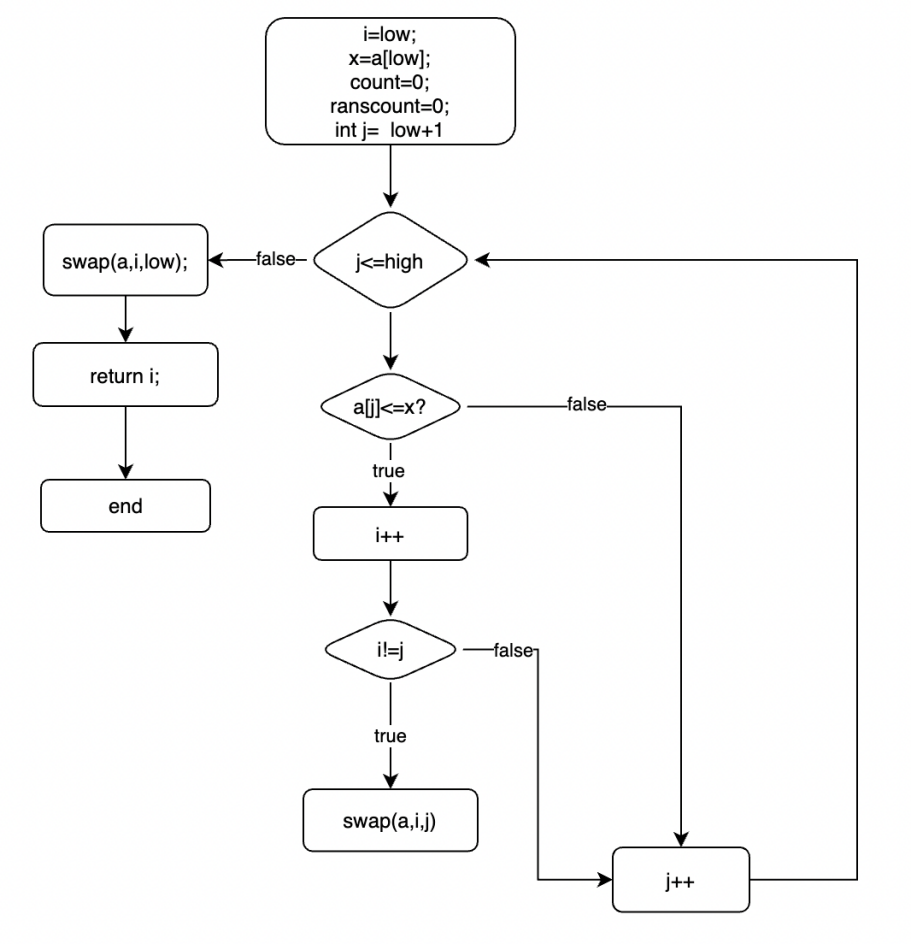


图5 快速排序流程图

基本思想：快速排序采用分治的思想，通过一次排序将待排序的序列分成两部分，其中一部分记录的关键字小于另一部分记录的关键字。然后继续分别对两部分记录进行排序，实现整个序列的排序。

首先从数列的右边开始往左找，设下标为i，并进行减减操作（i--），找到第1个比基准数小的值，让它与基准值交换；接着从左开始往右找，设下标为j，并执行加加操作（j++），找到第1个比基准数大的值，让它与基准值交换；然后继续寻找，直到i与j相遇时结束，最后基准值所在的位置即k的位置，也就是说k左边的值均比k上的值小，而k右边的值都比k上的值大。

实现代码：

public static void swap(int[] a, int i, int j) {

int temp = a[i];

a[i] = a[j];

a[j] = temp;

quickTransferCount++;

}

/\*\*

\* 返回比较元素的位置

\*/

public static int split(int[] a, int low, int high) {

// i指向比较元素的期望位置

int i = low;

// 将该组的第一个元素作为比较元素

int x = a[low];

// 从第二个元素开始，若当前元素大于比较元素，将其跳过

for (int j = low + 1; j <= high; j++) {

quickCount++;

// 若找到了小于比较元素的元素，将其与前面较大的元素进行交换

if (a[j] <= x) {

i++;

if (i != j) {

swap(a, i, j);

}

}

}

// 将比较元素交换到正确的位置上

swap(a, i, low);

return i;

}

/\*\*

\* 快速排序

\*/

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

// 划分并获取比较元素的位置

int i = split(arr, low, high);

// 对比较元素左边的数组进行排序

quickSort(arr, low, i - 1);

// 对比较元素右边的数字进行排序

quickSort(arr, i + 1, high);

}

}

4.1.5 二分排序

函数原型： binaryInsertSort();

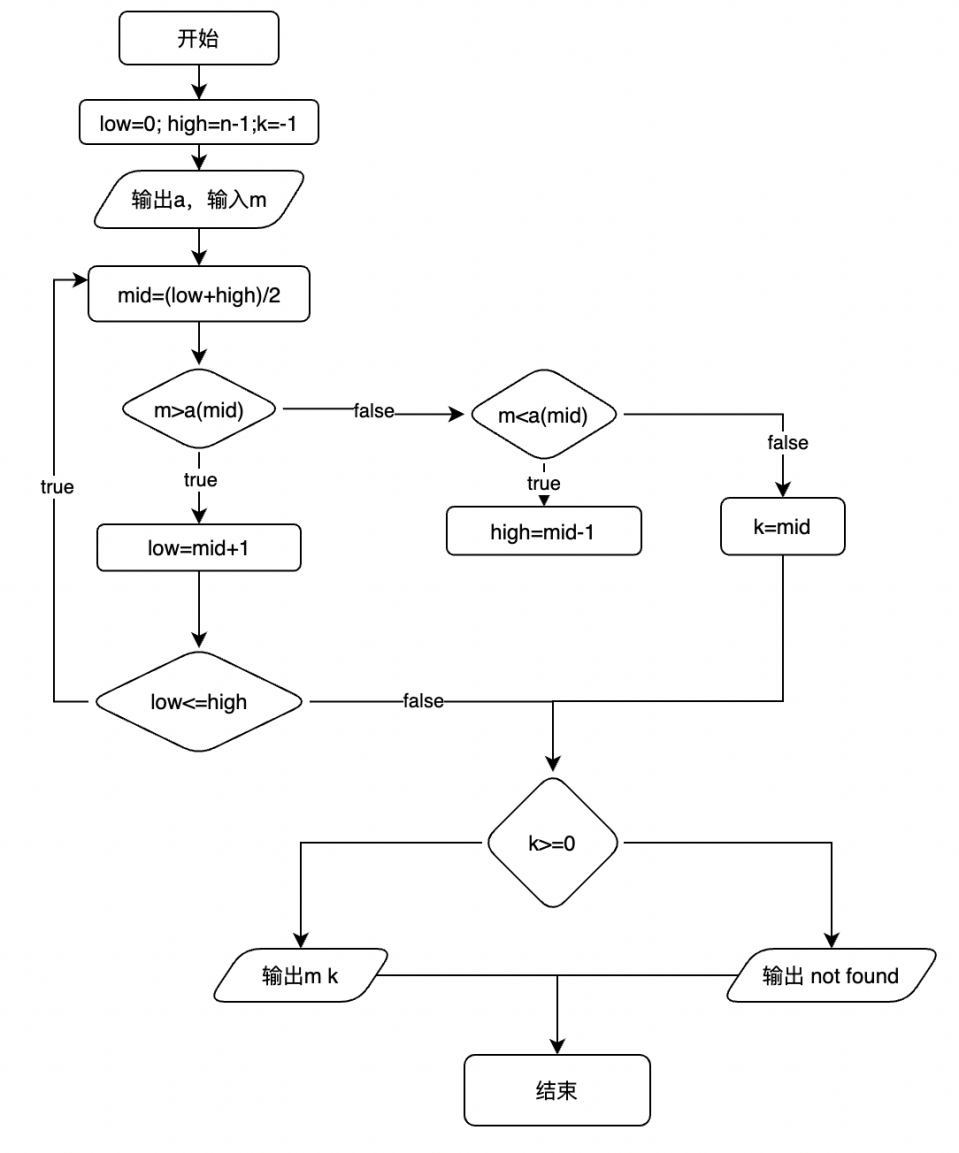


图6 二分排序流程图

基本思想：二分插入排序是在插入第i个元素时，将前0~i-1个元素减半，先比较中间的元素。如果较小，则前半部分减半，否则后半部分减半。直到 left>right ，然后将第 i 个元素的第一个位置和目标位置之间的所有元素向后移动，然后将第 i 个元素放在目标位置。

实现代码：

public static void binaryInsertSort(int[] arr) {

int count = 0;

int transcount = 0;

long start = System.currentTimeMillis();

for (int i = 1; i < arr.length; i++) {

// 每次二分查找确定temp需要插入的合适位置

int temp = arr[i];

// 取i之前的一组数据进行二分查找

int low = 0; // 有序的第一个

int high = i - 1; // 有序的第二个

int mid;

// 寻找位置

while (low <= high) {

// 取一个中间值

mid = low + (high - low) / 2;

if (arr[mid] > temp) {

// 对左半小数值区域再次进行二分查找

high = mid - 1;

} else {

// 对右半大数值区域再次进行二分查找

low = mid + 1;

}

}

for (int j = i - 1; j >= low; j--) {

arr[j + 1] = arr[j];

count++;

}

arr[low] = temp;

transcount++;

}

printCost(count, transcount, start);

printArray(arr);

}

# 测试数据和测试结果

## **5.1测试数据**

随机生成100个随机数字。

## **5.2测试结果**

本程序在IntelliJ IDEA环境下开发，以下是对测试数据的测试结果：

1. 系统主界面：

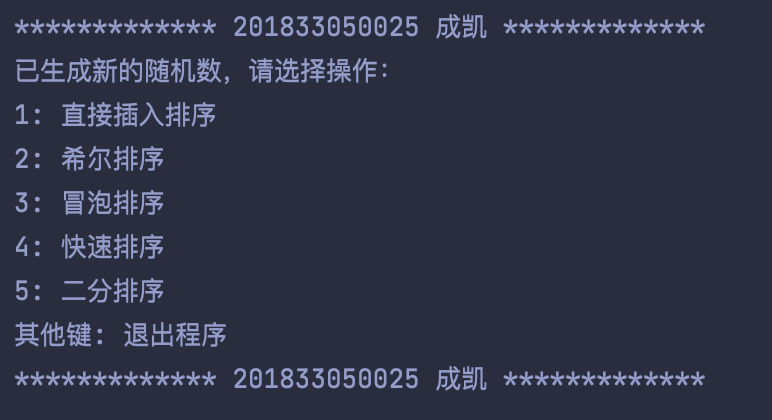
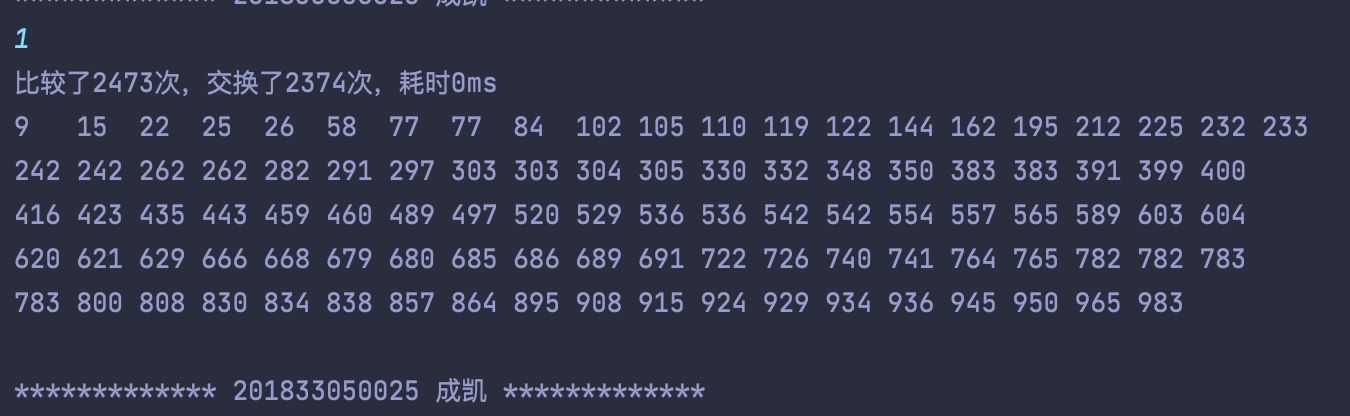
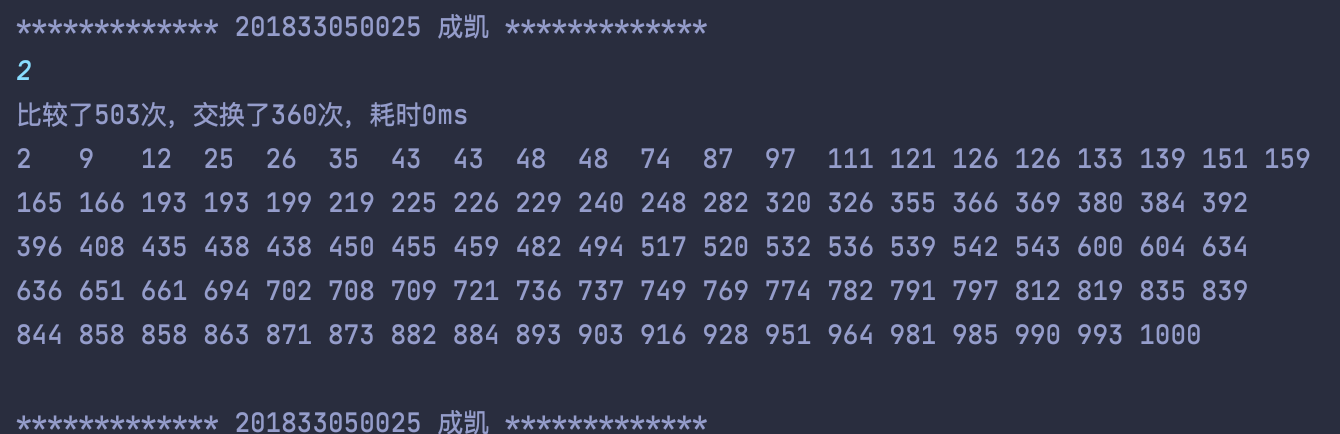


图7 功能截图

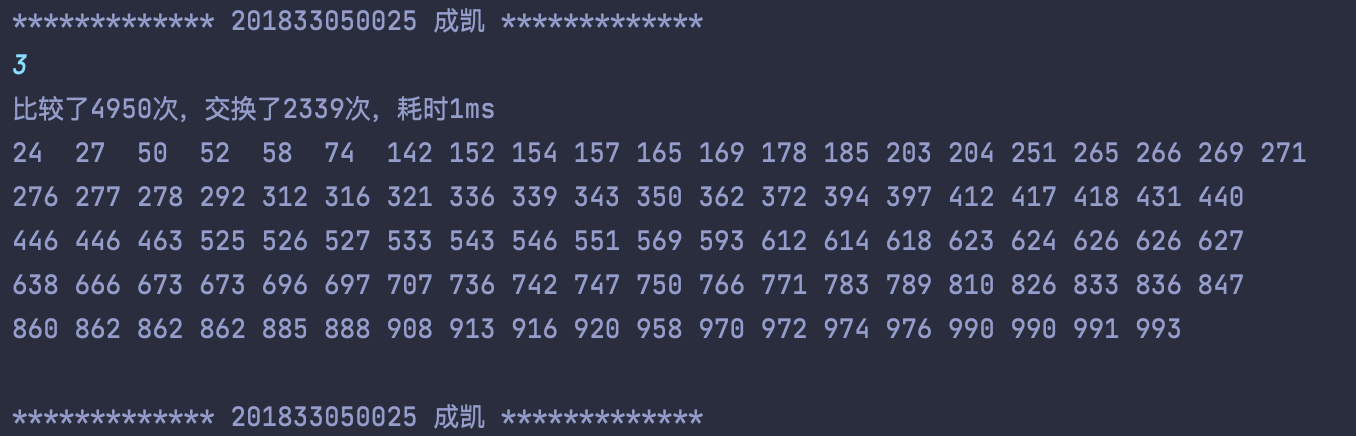
（2）执行排序并查看各项排序方法的结果:



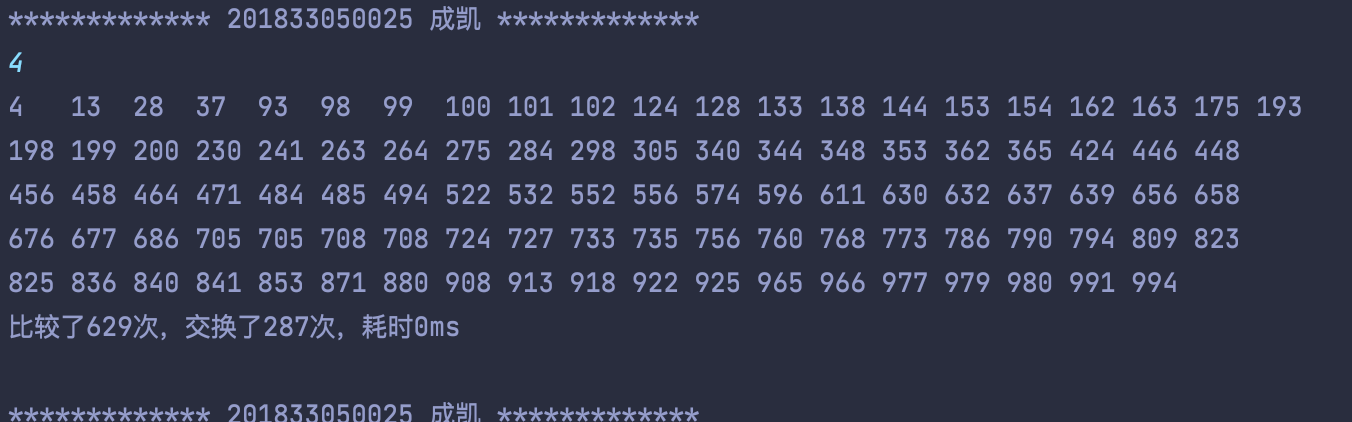
**图8 直接插入排序结果**



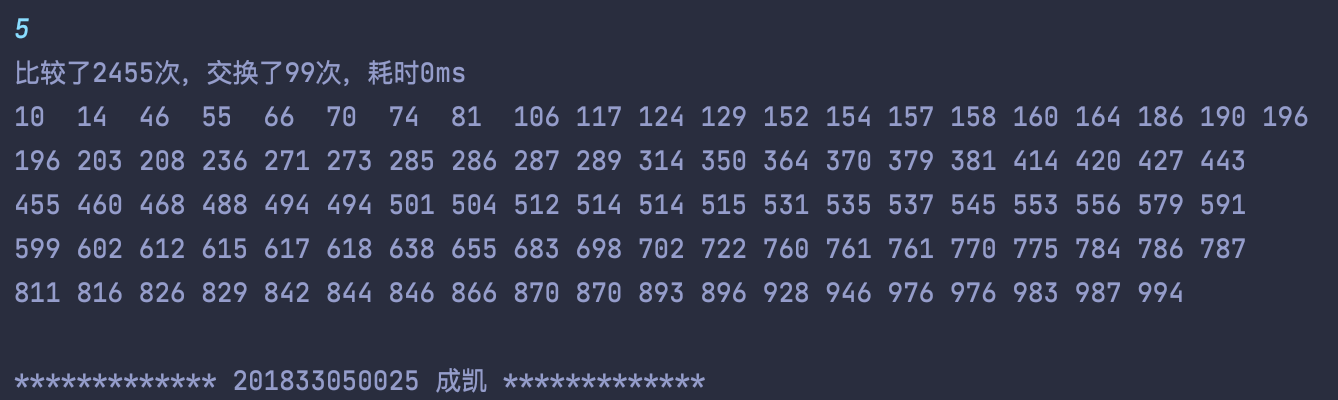
**图9 希尔排序结果**



**图10 冒泡排序结果**



**图11 快速排序结果**



**图12二分排序结果**

## **5.3结果分析**

**各个排序算法的性能总结：**

**表1 算法时间，空间复杂度分析**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **排序方式** | **时间复杂度** | | | **空间复杂度** | **稳定性** |
| **最好情况** | **最坏情况** | **平均情况** |
| **直接插入排序** | **O(n²)** | **O(n²)** | **O(n)** | **O(1)** | **稳定** |
| **希尔排序** | **-** | **-** | **O(n**1.3**)** | **O(1)** | **不稳定** |
| **冒泡排序** | **O(n²)** | **O(n²)** | **O(n)** | **O(1)** | **稳定** |
| **快速排序** | **O(**log2**n)** | **O(n²)** | **O(n**log2**n)** | **O(**log2**n)** | **不稳定** |
| **二分排序** | **O(**log2**n)** | **O(n²)** | **O(n²)** | **O(1)** | **稳定** |

**各个排序算法的性能分析：**

插入和冒泡排序相对较慢，但当参与排序的序列是部分或整体有序时，这种排序可以达到更快的速度，相反就会比较慢了。当数据量较小时，求稳定性时可以采用直接插入或冒泡排序。当数据量较大时，关键元素比较随机且不要求稳定性，可使用快速排序。随着数据量的增大，算法的实际增长速度也跟随变大。直接插入排序和冒泡排序的结果相差较小。

## **5.4总结**

在本次实验中，通过数据我清楚地意识到了不同时间复杂度的算法在处理大规模数据时的区别。 同时，我也意识到时间复杂度是相对的。 虽然时间复杂度的水平是一样的，运行时花费的时间也会有所不同。 这个实验揭示了以后在设计算法解决问题的时候，不仅要考虑时间复杂度的高低，还要注意实际语句的执行次数，进行优化，做到最好。

# **参考文献**

［1］《数据结构》（C语言版），严蔚敏，清华大学出版社

［2］《C程序设计（第三版）》，谭浩强，清华大学出版社