项目说明文档

数据结构课程设计

——8 种排序算法的比较案例

作 者 姓 名：杨鑫

学　　号：1950787

指 导 教 师：张颖

学院、 专业：软件学院 软件工程

同济大学

Tongji University

目录

1 分析 - 1 -
1.1 背景分析 - 1 -
1.2 功能分析 - 1 -
2 设计 - 1 -
2.1 生成随机数设计 - 1 -
2.2 类结构设计 - 2 -
2.3 成员与操作设计 - 2 -
2.4 系统设计 - 3 -
3 实现 - 3 -
3.1 主体功能的实现 - 3 -
3.1.1 主体功能的核心思想 - 3 -
3.1.2 主体功能的核心代码 - 3 -
4 测试 - 4 -
4.1 功能测试 - 4 -
4.1.1 100 个随机数 - 4 -
4.1.2 1000 个随机数 - 5 -
4.1.3 10000 个随机数 - 6 -
4.2 边界测试 - 7 -
4.2.1 只有一个随机数测试 - 7 -
4.3 错误测试 - 8 -
4.3.1 输入随机数个数过多（超过题目规定：10000） - 8 -
4.3.2 输入随机数个数过少 - 8 -
5 思考与总结 - 9 -

1 分析

1.1 背景分析

排序问题也是我们在日常生活中经常会遇到的问题。有时候我们对人进行排序，有时候我们对物品进行排序，还有的时候我们仅仅是对数字进行排序。不管对什么进行排序，它总要有一个用来排序的依据，例如根据人的高低进行排序，此时排序的依据就是身高，按人的身高将人依次排列。

在计算机中，也有排序的存在，计算机中的排序是指将杂乱无章的数据元素，通过一定的方法按关键字顺序排列的过程。排序算法就是如何使得记录按照要求排列的方法。计算机中的排序算法有很多种，每一种算法都有它们独特的实现方式与优化方法。在本案例中我们就来讨论其中使用频率比较高的几种排序算法。

1.2 功能分析

在本案例中，我们选择了冒泡排序，选择排序，直接插入排序，希尔排序，快速排序，堆排序，归并排序，基数排序八种排序算法来进行分析。

随机函数产生若干个随机数，用各种排序方法排序，并统计每种排序所花费的排序时间和交换次数，比较次数。其中，随机数的个数由用户定义，系统产生随机数，并且显示他们的比较次数，并且根据实验结果说明这些方法的优缺点。

2 设计

2.1 生成随机数设计

由于题目中要求最多会有 10000 个随机数，但 c++中的 rand()范围很小，容易导致重复元素过多等情况，使得产生的随机数质量不高。为了模拟产生一些数据范围比较大的随机数，本程序使用 rand() % 10 反复产生一些 0~9 的随机数字，然后将他们分别作为一个数字的不同位数，这样就产生了一系列范围比较大的随机数字。

核心代码：

```
1.  void System::getRand(int num) {
2.  this->nums = num;
3.  srand((unsigned)time(NULL));
4.
5.  // 产生 num 个随机数
6.  for (int i = 0; i < num; i++) {
```

```
        array[i] = 0;

        // 对每一个随机数，产生 RANDOM_LENGTH 个 0~9 的数字，分别作为其最高位至最低位的数字
        for (int j = 0; j < RANDOM_LENGTH; j++) {
            array[i] = rand() % 10 + array[i] * 10;
        }
    }
```

2.2 类结构设计

本程序的类比较简单，只有一个系统类（System）。它整合了随机数组和各种排序算法，统一了各种操作，简化并规范了程序。

2.3 成员与操作设计

系统类（System）：

私有成员：

```
    int nums; // 随机数个数
    long long exchangeNums; // 交换次数
    long long compareNums; // 比较次数
    int array[MAXNUMS + 5]; // 数组
    void swap(int i, int j); // 交换
    void quickSort(int left, int right); // 快速排序
    void mergeSort(int left, int right); // 归并排序
    void siftDown(int start, int end); // 堆的下移函数
    void radixSort(int left, int right, int rank); // 基数排序
    int getRank(int num, int rank); // 得到一个数字的某一位
```

公有操作：

```
    System() : nums(0), exchangeNums(0), compareNums(0) {} // 构造函数
    void loop(); // 主循环

    void getRand(int nums); // 产生随机数
    void bubbleSort(); // 冒泡排序
    void selectionSort(); // 选择排序
    void insertSort(); // 插入排序
    void shellSort(); // 希尔排序
    void quickSort(); // 快速排序
    void heapSort(); // 堆排序
    void mergeSort(); // 归并排序
    void radixSort(); // 基数排序
```

2.4 系统设计

系统在使用时，会生成菜单，用户根据菜单中的指示。通过输入操作码来完成对应的排序算法。同时系统也会记录本次算法所使用的时间，数据交换次数等信息并显示在屏幕上。

程序兼容了 windows 和 LINUX 平台，在双平台下均可以正常运行。

3 实现

3.1 主体功能的实现

3.1.1 主体功能的核心思想

根据输入的指令，执行相应的排序算法或者退出程序。如果是进行排序，则每次都会产生一组新的随机数进行排序，并且在产生随机数后，开始排序前记录下起始时间点；在排序完成后记录结束时间点，这样即可统计所花费的时间。同时在排序算法里面需要交换元素时将交换次数+1（每次开始前会把交换次数置为0），这样也就完成了交换次数的统计。

3.1.2 主体功能的核心代码

```
while (true) {
    cout << "请选择排序算法：";
    int op;
    cin >> op;
    clock_t start, end;
    string str;
    getRand(nums); // 产生随机数
    start = clock(); // 记录起始时间点
    switch (op) {...} // 根据操作码执行对应操作（此处省略）
    // 记录结束时间点
    end = clock();
                cout << str << "所用时间：" << ((double)(end - start) / CLOCKS_PER_SEC) * 1000 << "毫秒" << endl;
     cout << str << "所用交换次数：" << exchangeNums << "次" << endl;
    if (op != 8) {
        cout << str << "所用比较次数：" << compareNums << "次" << endl << endl;
    }
    else {
```

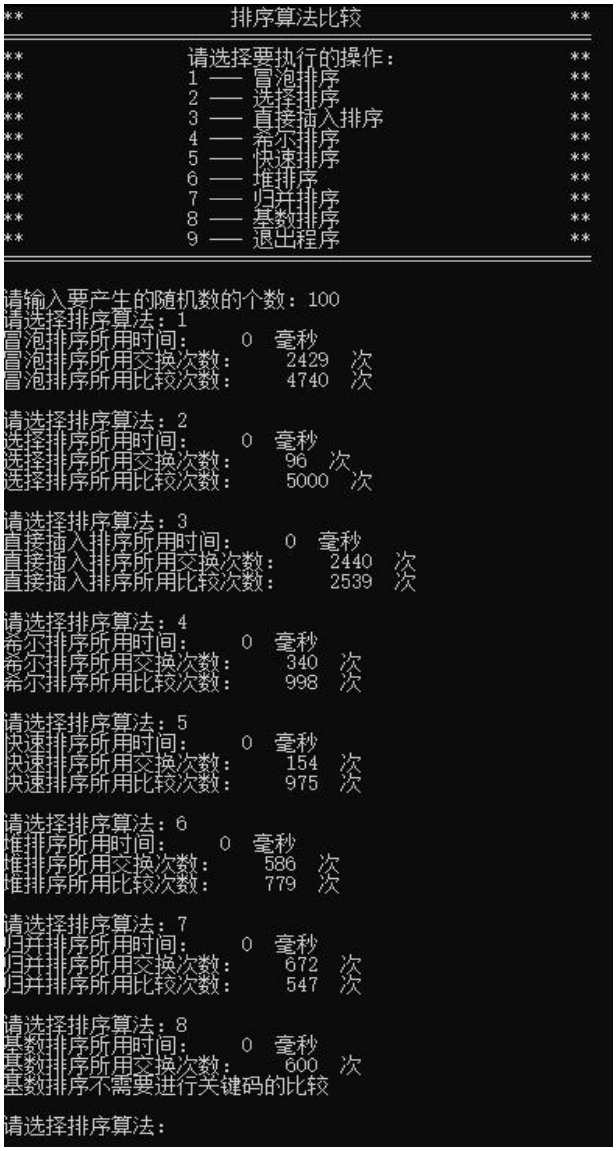
```
18.             cout << str << "不需要进行关键码的比较" << endl << endl;
19.         }
20. }
```

4 测试

4.1 功能测试

4.1.1 100 个随机数

测试用例：
100
预期结果：
程序正常运行，输出相应的结果。
实验结果：



4.1.2 1000 个随机数

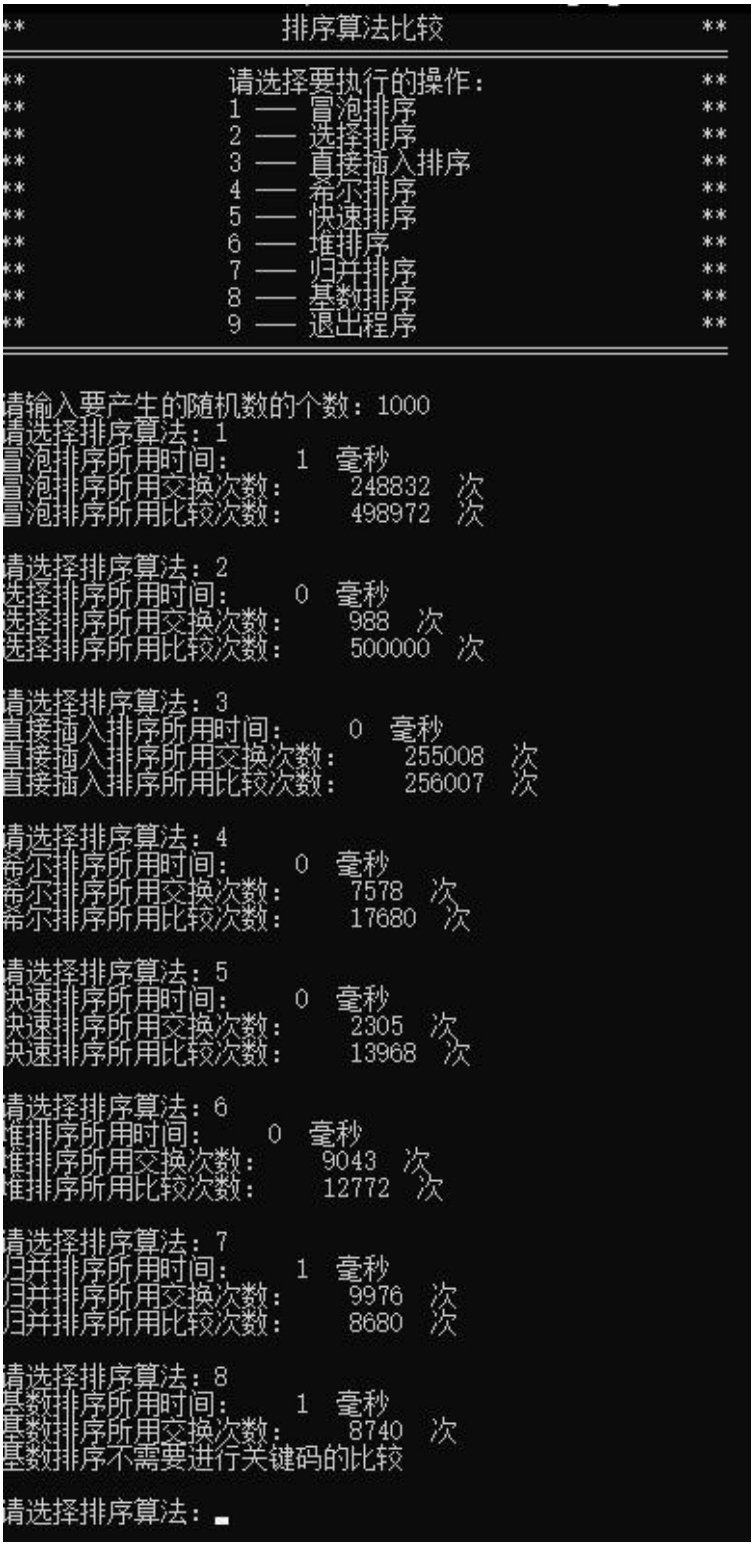
测试用例：

1000

预期结果：

程序正常运行，输出相应的结果。

实验结果：



4.1.3 10000 个随机数

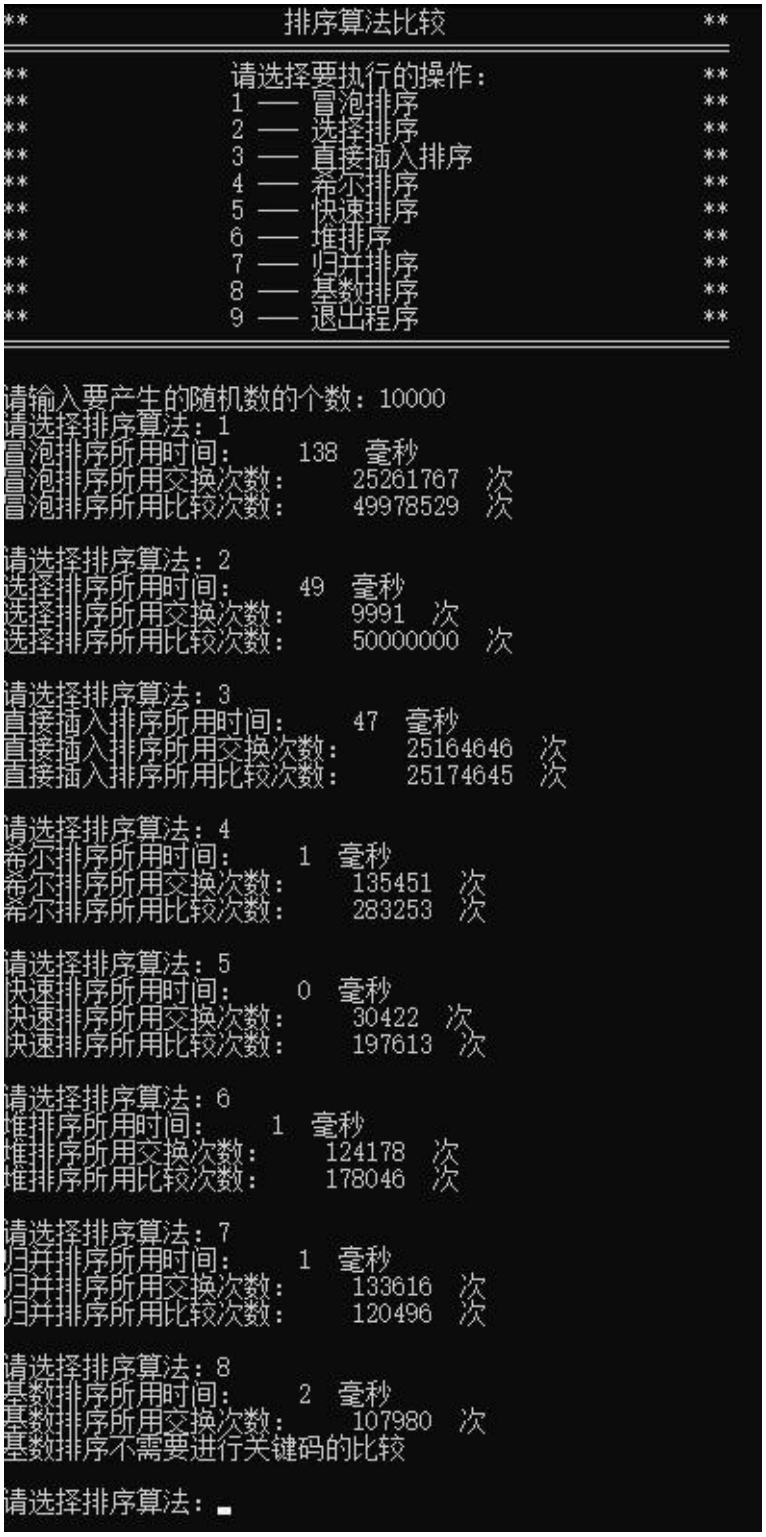
测试用例：

10000

预期结果：

程序正常运行，输出相应的结果。

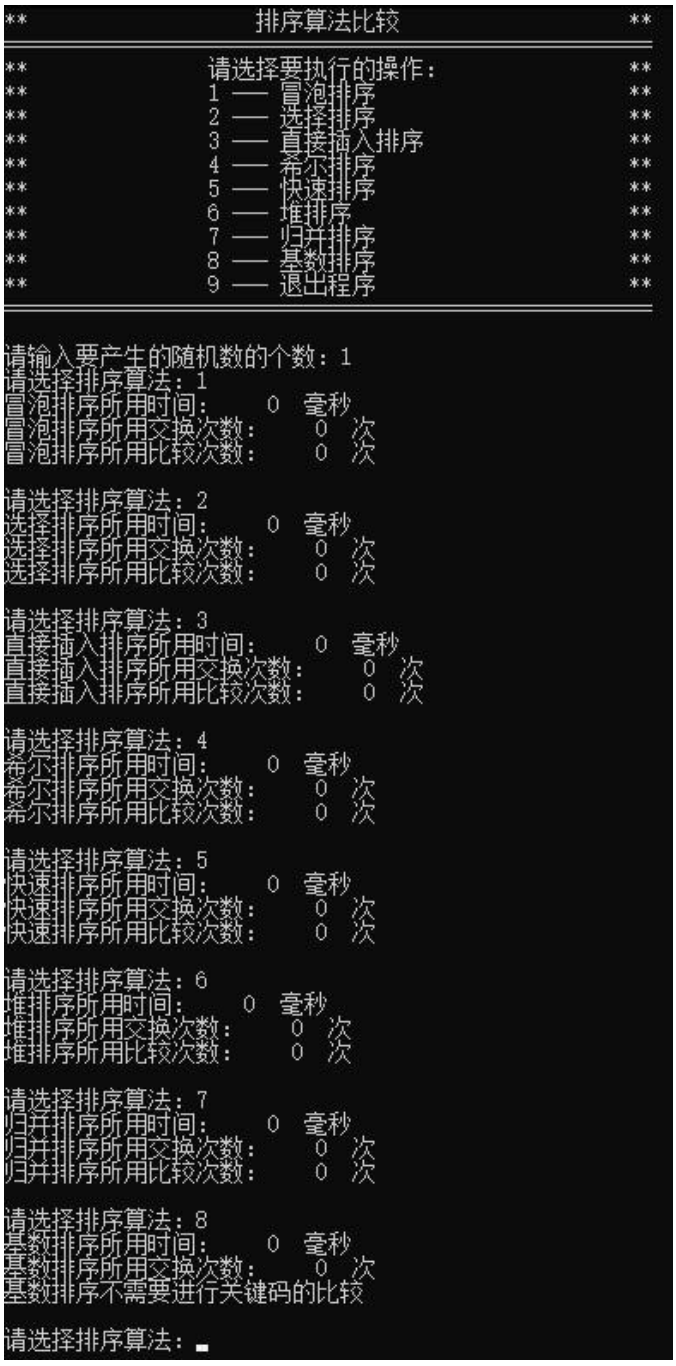
实验结果：



4.2 边界测试

4.2.1 只有一个随机数测试

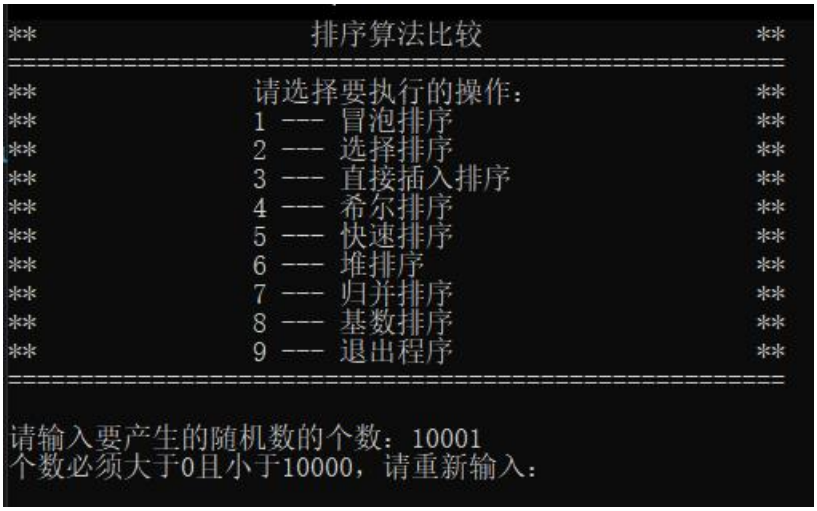
测试用例：
1
预期结果：程序正常运行不崩溃，因为只有一个数字，所以交换次数均为 0，时间也非常小（非常接近 0），不会发生边界错误而导致程序无法正常运行。
实验结果：



4.3 错误测试

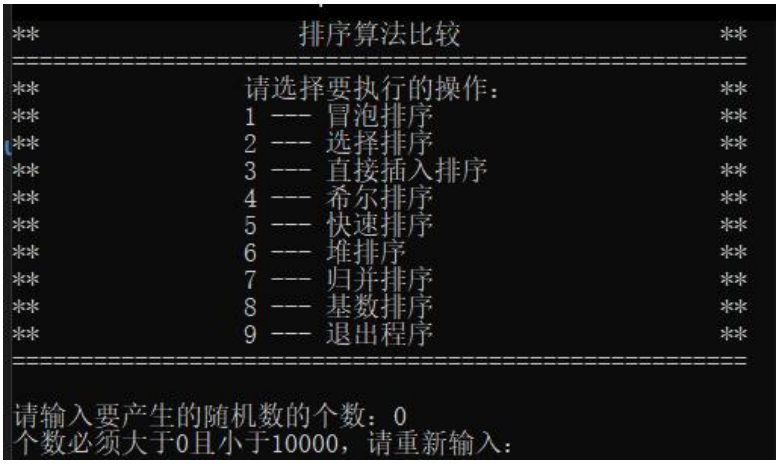
4.3.1 输入随机数个数过多（超过题目规定：10000）

测试用例：
10001
预期结果：程序正常运行不崩溃，提示用户输入不合法并要求重新输入。
实验结果：



4.3.2 输入随机数个数过少

测试用例：
0
预期结果：程序正常运行不崩溃，提示用户输入不合法并要求重新输入。
实验结果：



5 思考与总结

1. 直接插入排序，冒泡排序，选择排序是最基本的排序方法。它们平均情况下的时间复杂度都是 $O(n^2)$，这三种基本的排序方法除了一个辅助元素外，都不需要其他额外内存。从稳定性来看，直接插入排序与冒泡排序都是稳定的，选择排序不是。它们适用于元素个数不是很多的情况。
2. 冒泡排序数据比较次数和输入序列中个排序元素的初始排列无关，数据的移动次数和各排序元素的初始序列有关。当元素某次不在发生变化时就说明排序完成，不需要继续搜索。
3. 直接插入排序的时间复杂度与待排序序列的初始排列有关，在最好情况下，直接插入排序只需要 n-1 次比较操作就可以完成，不需要交换操作。
4. 选择排序的数据比较次数和输入序列中个排序元素的初始排列无关，数据的移动次数和各排序元素的初始序列有关。最优情况下，例如升序和重复序列时，一次也不用移动。
5. 希尔排序的时间复杂度介于基本排序算法与高效算法之间，这主要取决于 gap 序列的选择。希尔排序是一种不稳定的排序算法，数据比较次数和移动次数与输入序列中个排序元素的初始排列有关。一般在元素个数在几千时，希尔排序是很好的选择。
6. 快速排序，堆排序，归并排序都是高效算法，适合于元素个数很大的情况。
7. 快速排序是最通用的排序算法，它的时间复杂度为 $O(n\log_2 n)$，所需额外内存为 $O(\log_2 n)$，它的数据移动次数相比于其他高效算法来说较少，是一种不稳定的排序算法。快速排序的效率在序列越乱的时候，效率越高。在某些情况下，例如在数据有序时，会退化成冒泡排序，时间复杂度增加至 $O(n^2)$。可以通过选择更好的基准元素来优化此排序。
8. 堆排序的时间复杂度是 $O(n\log_2 n)$，是一种不稳定的排序算法。它的效率相对稳定，不会因为某些原因使时间复杂度明显增加，是对数据的有序性不敏感的一种算法。并且堆排序也不需要额外的空间，相比于另外两种高效算法，不需要担心可能会发生堆栈溢出错误。但由于需要时刻进行堆的维护，因此实际应用中不如快速排序广泛。
9. 归并排序的时间复杂度是 $O(n\log_2 n)$，它的优点在于它是一种稳定的高效算法，但它需要 $O(n)$ 的附加空间。对于元素较多，且要求稳定性时，可以使用归并排序。
10. 基数排序是一种相对特殊的排序算法，它是将排序码的不同部分进行处理和比较。基数排序基于的排序码抽取算法受到系统和排序元素的影响，其适应性远不如普通的比较与排序，因此实际工作中使用不多。
11. 不同的排序算法有时可以集成起来，例如在归并排序中对小规模子数组使用插入排序可以使得运行时间缩短。