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: 沈星宇

学　　号: 1951576

指 导 教 师: 张颖

学院、 专业: 软件学院 软件工程

同济大学

Tongji University

一、 分析

（1） 应用背景

快速排序，直接插入排序，冒泡排序，选择排序等排序方法各有优缺点，没有绝对的最优排序算法，但在不同的情况下，各个算法有不同的表现。

现要通过统计每种排序在不同规模的随机数下所花费的排序时间和交换次数来比较每种方法的优缺点和适用场景。

（2） 项目功能要求

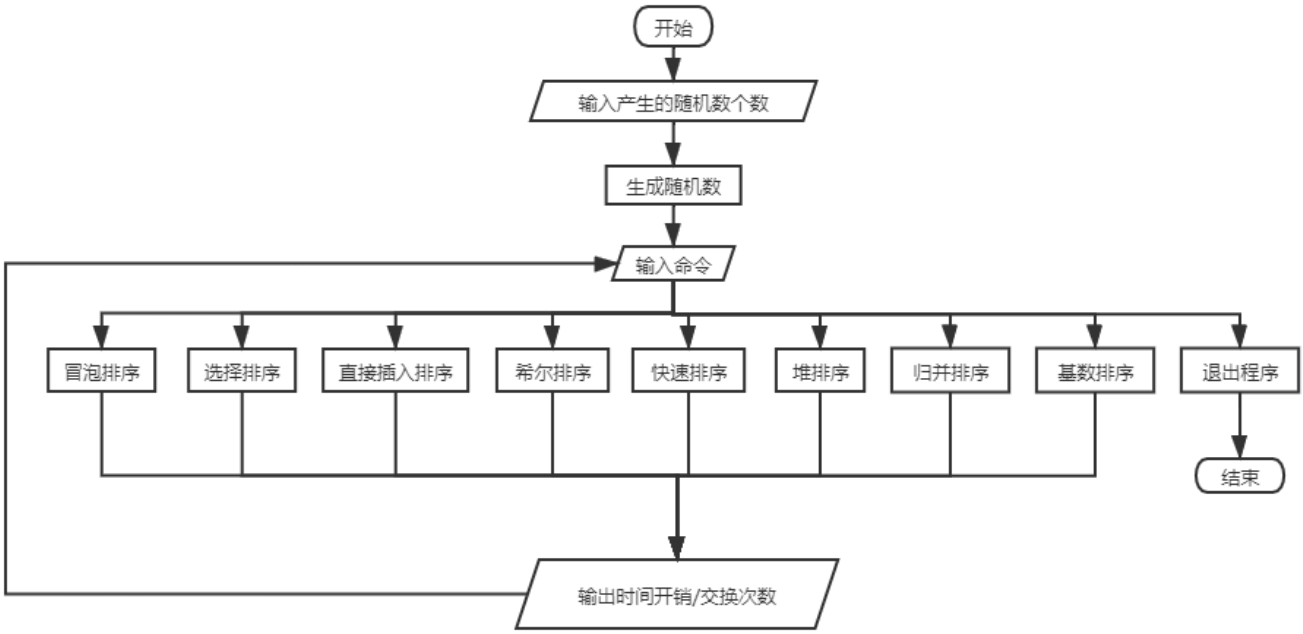
1、随机函数产生一百，一千，一万和十万个随机数，用快速排序，直接插入排序，冒泡排序，选择排序的排序方法排序，并统计每种排序所花费的排序时间和交换次数。

2、随机数的个数由用户定义，系统产生随机数，并显示他们的比较次数。

3、文档中记录上述数据量下，各种排序的计算时间和存储开销，并且根据实验结果说明这些方法的优缺点。

二、 设计

（1） 程序流程设计



1、输入产生随机数的个数，调用 rand()生成随机数，并将其存储在数组中

2、循环请求用户输入命令，接收到命令后调用相应的函数

3、输出每个排序算法的时间开销和交换次数

三、 实现

1、冒泡排序：bubbleSort()的实现

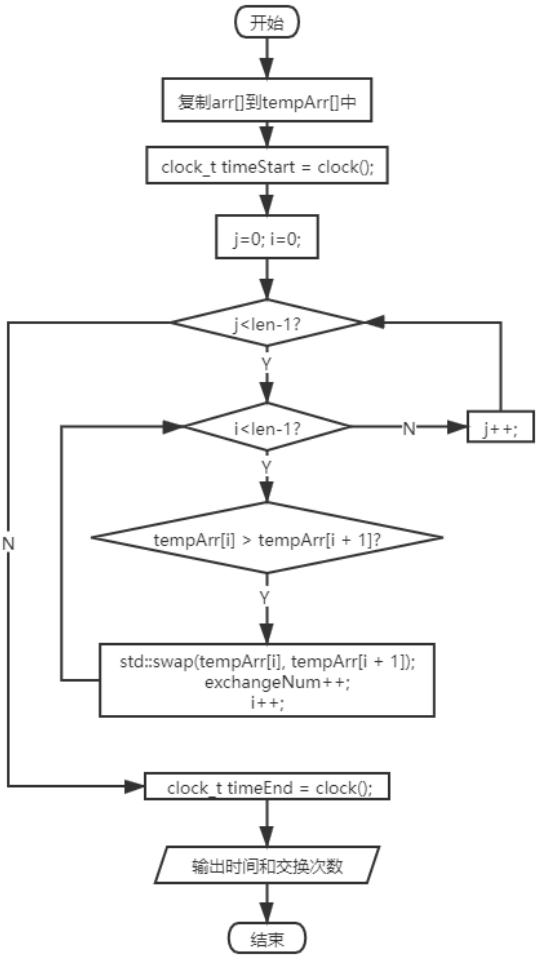
（1） 首先为了防止排序改变原有的随机数序列影响后面的排序，故首先用一个数组 tempArr 将原有数组复制过去，则下列所有的操作都是对 tempArr

进行的，在出函数之前对其进行释放。

（2） 从前往后循环依次比较相邻两个数的大小关系，大的数往后冒

（3） 输出排序所用的时间和交换次数

（4） 时间复杂度 O(n^2) 空间复杂度 O(1)




```cpp
void bubbleSort(int arr[], int len){
    long long int exchangeNum = 0;
    int* tempArr = new int[len];
    for (int i = 0; i < len; i++) {
        tempArr[i] = arr[i];
    }
    //-----------------------------------------
    clock_t timeStart = clock();
    for (int j = 0; j < len - 1; j++){
        for (int i = 0; i < len - 1 - j; i++) {
            if (tempArr[i] > tempArr[i + 1]){
                std::swap(tempArr[i], tempArr[i + 1]);
                exchangeNum++;
            }
        }
    }
    //-------------------------------------------------------------------------------------------
    clock_t timeEnd = clock();
    std::cout << "冒泡排序所用时间: " << 1000 * (timeEnd - timeStart) / (long)CLOCKS_PER_SEC << "ms\n";
    std::cout << "冒泡排序交换次数: " << exchangeNum << std::endl;
    std::cout << std::endl;
    //for (int i = 0; i < len; i++) {
    //  std::cout << tempArr[i] << ' ';
    //}
    //std::cout << std::endl;
    delete[]tempArr;
}
```

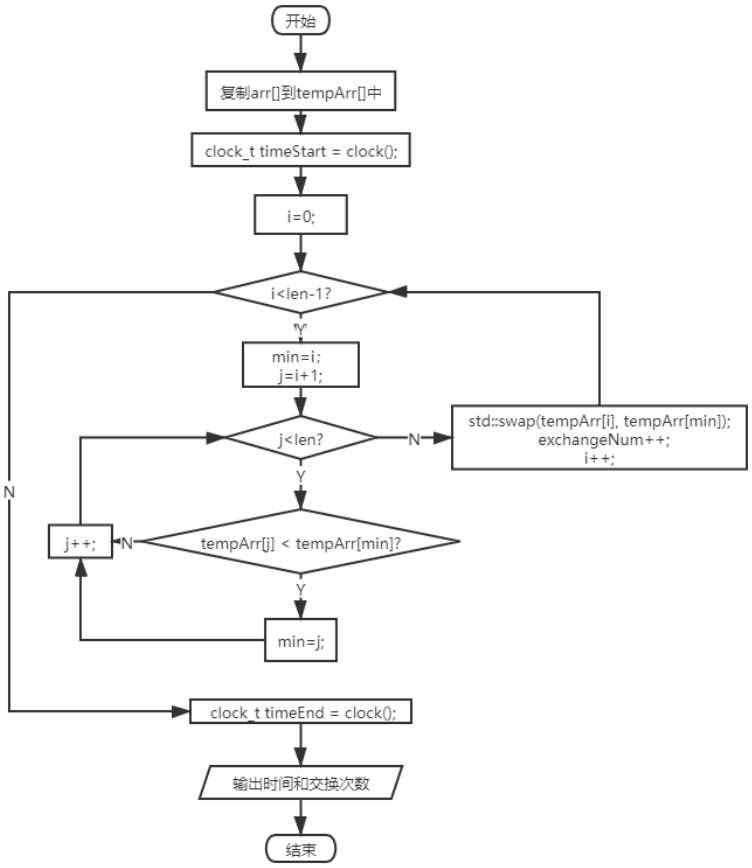
2、选择排序：selectionSort()的实现

(1) 首先为了防止排序改变原有的随机数序列影响后面的排序，故首先用一个数组 tempArr 将原有数组复制过去，则下列所有的操作都是对 tempArr 进行的，在出函数之前对其进行释放。

(2) 从前往后遍历，找到最小的数字，放到排列过的有序子序列后的第一个数字的位置，每次都把最小的数字放到前面。

(3) 输出排序所用的时间和交换次数。

(4) 时间复杂度 O(n^2) 空间复杂度 O(1)。




```
void slectionSort(int arr[],int len) {
    long long int exchangeNum = 0;
    int* tempArr = new int[len];
    for (int i = 0; i < len; i++) {
        tempArr[i] = arr[i];
    }
    //-----------------------------------------
    clock_t timeStart = clock();
    for (int i = 0; i < len - 1; i++) {
        int min = i;
        for (int j = i + 1; j < len; j++) {
            if (tempArr[j] < tempArr[min]) {
                min = j;
            }
        }
        std::swap(tempArr[i], tempArr[min]);
        exchangeNum++;
    }
    clock_t timeEnd = clock();
    std::cout << "选择排序所用时间: " << 1000 * (timeEnd - timeStart) / (long)CLOCKS_PER_SEC << "ms\n";
    std::cout << "选择排序交换次数: " << exchangeNum << std::endl;
    std::cout << std::endl;
    //for (int i = 0; i < len; i++) {
    //  std::cout << tempArr[i] << ' ';
    //}
    //std::cout << std::endl;
    delete[]tempArr;
}
```

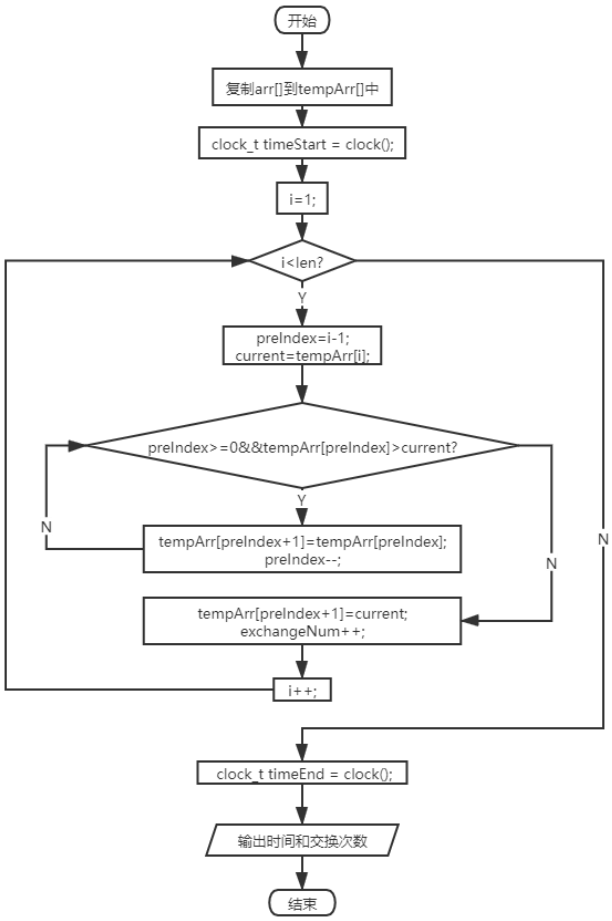
3、 插入排序：insertionSort()的实现

(1) 首先为了防止排序改变原有的随机数序列影响后面的排序，故首先用一个数组 tempArr 将原有数组复制过去，则下列所有的操作都是对 tempArr 进行的，在出函数之前对其进行释放。

(2) 从前往后扫描，第 i 次扫描会将第 i+1 个数字插入到前 i 个子序列当中，并且使它们符合从小到大的顺序，如此循环直到读取到最后一个数字。

(3) 输出排序所用的时间和交换次数。

(4) 时间复杂度 O(n^2) 空间复杂度 O(1)。




```
void insertionSort(int arr[],int len) {
    long long int exchangeNum = 0;
    int* tempArr = new int[len];
    for (int i = 0; i < len; i++) {
        tempArr[i] = arr[i];
    }
    //-----------------------------------------
    clock_t timeStart = clock();
    int preIndex, current;
    for (int i = 1; i < len; i++) {
        preIndex = i - 1;
        current = tempArr[i];
        while (preIndex >= 0 && tempArr[preIndex] > current) {
            tempArr[preIndex + 1] = tempArr[preIndex];
            preIndex--;
        }
        tempArr[preIndex + 1] = current;
        exchangeNum++;
    }
    clock_t timeEnd = clock();
    std::cout << "插入排序所用时间: " << 1000 * (timeEnd - timeStart) / (long)CLOCKS_PER_SEC << "ms\n";
    std::cout << "插入排序交换次数: " << exchangeNum << std::endl;
    std::cout << std::endl;
    //for (int i = 0; i < len; i++) {
    //  std::cout << tempArr[i] << ' ';
    //}
    //std::cout << std::endl;
    delete[]tempArr;
}
```

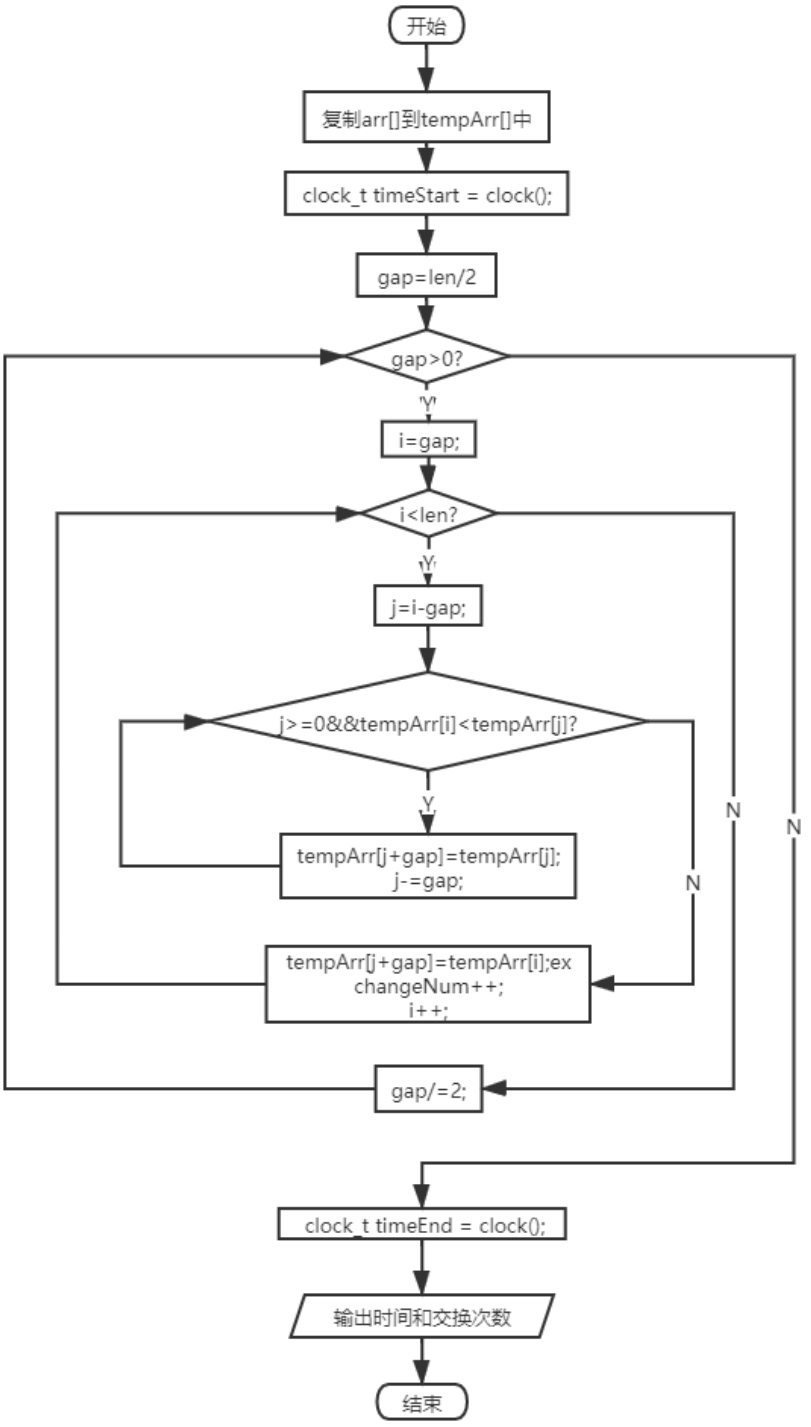
4、 希尔排序：shellSort 的实现

(1) 首先为了防止排序改变原有的随机数序列影响后面的排序，故首先用一个数组 tempArr 将原有数组复制过去，则下列所有的操作都是对 tempArr 进行的，在出函数之前对其进行释放。

(2) 希尔排序是在插入排序的基础上进行改进，比较的是第 i 位和第 i+gap 位上面的两个数字，使其有序，通过不断缩小 gap 的值来确保最后整个序列有序

(3) 输出排序所用的时间和交换次数。

(4) 时间复杂度 O(n^1.3)　空间复杂度 O(1)



```cpp
void shellSort(int arr[], int len) {
    long long int exchangeNum = 0;
    int* tempArr = new int[len];
    for (int i = 0; i < len; i++) {
        tempArr[i] = arr[i];
    }
    //-----------------------------------------
    clock_t timeStart = clock();
    for (int gap = len / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < len; i++) {
            int j;
            int inserted = tempArr[i];
            for (j = i - gap; j >= 0 && inserted < tempArr[j]; j -= gap) {
                tempArr[j + gap] = tempArr[j];
            }
            tempArr[j + gap] = inserted;
            exchangeNum++;
        }
    }
    clock_t timeEnd = clock();
    //-------------------------------------------------------------------------------------------
    std::cout << "希尔排序所用时间: " << 1000 * (timeEnd - timeStart) / (long)CLOCKS_PER_SEC << "ms\n";
    std::cout << "希尔排序交换次数: " << exchangeNum << std::endl;
    std::cout << std::endl;
    //for (int i = 0; i < len; i++) {
    //  std::cout << tempArr[i] << ' ';
    //}
    //std::cout << std::endl;
    delete[]tempArr;
}
```

5、 快速排序：quickSort 的实现

（1） 首先为了防止排序改变原有的随机数序列影响后面的排序，故首先用一个数组 tempArr 将原有数组复制过去，则下列所有的操作都是对 tempArr 进行的，在出函数之前对其进行释放。

（2） 快速排序通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

（3） 输出排序所用的时间和交换次数

（4） 时间复杂度 O(nlogn)　空间复杂度 O(nlogn)

```cpp
void quickSort(int tempArr[], int len, int left, int right,long long int& exchangeNum){
    if (left < right) {
        int i = left, j = right, key = tempArr[left];
        while (i < j) {
            while (i < j && tempArr[j] >= key) {
                j--;
            }
            if (i < j) {
                tempArr[i++] = tempArr[j];
            }
            while (i < j && tempArr[i] < key) {
                i++;
            }
            if (i < j) {
                tempArr[j--] = tempArr[i];
            }
        }
        tempArr[i] = key;
        exchangeNum++;
        quickSort(tempArr, len, left, i - 1,exchangeNum);
        quickSort(tempArr, len, i + 1, right,exchangeNum);
    }
}
```

6、 堆排序：heapify()的实现

（1） 首先为了防止排序改变原有的随机数序列影响后面的排序，故首先用一个数组 tempArr 将原有数组复制过去，则下列所有的操作都是对 tempArr 进行的，在出函数之前对其进行释放。

（2） 利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

（3） 输出排序所用的时间和交换次数

（4） 时间复杂度 O(nlogn)　空间复杂度 O(1)

```
void heapify(int tempArr[], int i, int len, long long int& exchangeNum) {
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int maxIndex = i;
    if (left<len && tempArr[left]>tempArr[maxIndex]) {
        maxIndex = left;
    }
    if (right<len && tempArr[right]>tempArr[maxIndex]) {
        maxIndex = right;
    }
    if (maxIndex != i) {
        std::swap(tempArr[i], tempArr[maxIndex]);
        exchangeNum++;
        heapify(tempArr, maxIndex, len,exchangeNum);
    }
}
```

7、 归并排序：mergeSort()的实现

（1） 首先为了防止排序改变原有的随机数序列影响后面的排序，故首先用一个数组 tempArr 将原有数组复制过去，则下列所有的操作都是对 tempArr 进行的，在出函数之前对其进行释放。

（2） 归并排序将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为 2-路归并。

（3） 输出排序所用的时间和交换次数

（4） 时间复杂度 O(nlogn)　空间复杂度 O(n)

```
void mergeSort(int arr[], int left, int right,long long int& exchangeNum) {
    if (left == right) {
        return;
    }
    int mid = (left + right)/2;
    mergeSort(arr, left, mid,exchangeNum);
    mergeSort(arr, mid + 1, right,exchangeNum);
    //-----------------------------------------------------------
    int* temp = new int[right - left + 1];
    int i = 0, p1 = left, p2 = mid + 1, length = right - left + 1;
    while (p1 <= mid && p2 <= right) {
        if (arr[p1] < arr[p2]) {
            temp[i++] = arr[p1++];
        }
        else {
            temp[i++] = arr[p2++];
            exchangeNum++;
        }
    }
    while (p1 <= mid) {
        temp[i++] = arr[p1++];
    }
    while (p2 <= right) {
        temp[i++] = arr[p2++];
    }
    for (i = 0; i < length; i++) {
        arr[left + i] = temp[i];
    }
```

8、 基数排序：radixSort()的实现

（1） 首先为了防止排序改变原有的随机数序列影响后面的排序，故首先用一个数组 tempArr 将原有数组复制过去，则下列所有的操作都是对 tempArr 进行的，在出函数之前对其进行释放。

（2） 基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。

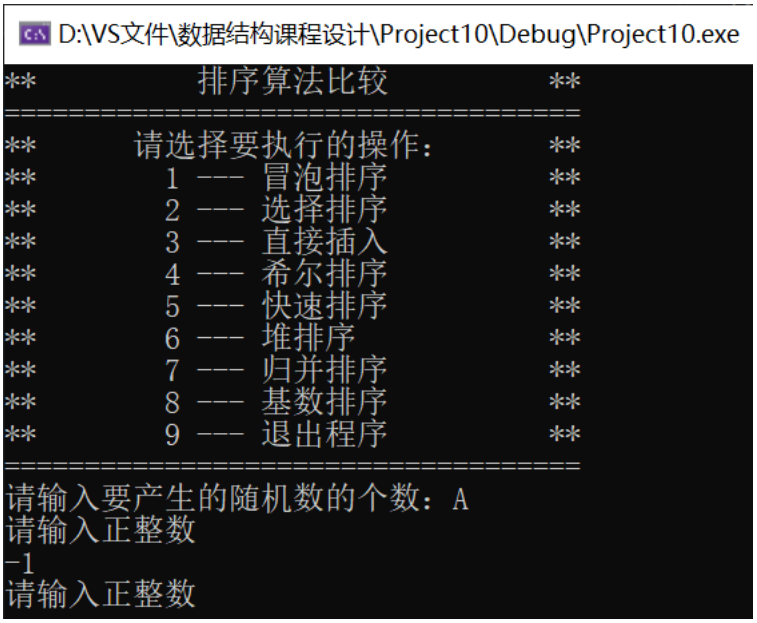
（3） 输出排序所用的时间和交换次数

（4） 时间复杂度 O(n*k)　空间复杂度 O(n+k)

```
int maxData = tempArr[0];
for (int i = 1; i < len; ++i) {
    if (maxData < tempArr[i]) {
        maxData = tempArr[i];
    }
}
int d = 1, p = 10;
while (maxData >= p) {
    maxData /= 10;
    ++d;
}
//-----------------------------
int* tmp = new int[len];
int* count = new int[10];
int k;
int radix = 1;
for (int i = 1; i <= d; i++) {
    for (int j = 0; j < 10; j++) {
        count[j] = 0;
    }
    for (int j = 0; j < len; j++) {
        k = (tempArr[j] / radix) % 10;
        count[k]++;
    }
    for (int j = 1; j < 10; j++) {
        count[j] = count[j - 1] + count[j];
    }
    for (int j = len - 1; j >= 0; j--) {
        k = (tempArr[j] / radix) % 10;
        tmp[count[k] - 1] = tempArr[j];
        exchangeNum++;
        count[k]--;
    }
    for (int j = 0; j < len; j++) {
        tempArr[j] = tmp[j];
    }
    radix = radix * 10;
}
delete[]tmp;
delete[]count;
```

四、 测试

1、 合法性检测



2、各排序算法性能分析

（1）冒泡排序：

对于规模较小的元素序列比较有效，然而在元素数量提升之后，其时间开销较之其他排序算法非常大，不适用于大规模数量的数据。优点是内存开销不大，排序非常稳定。

（2）选择排序：

对于规模较小的元素序列有效，不适用于大规模数量的数据。优点是内存开销不大，排序非常稳定。

（3）插入排序：

对于规模很小的元素序列非常有效，其时间复杂度与待排序元素序列的初始排列有关，排序非常稳定。

（4）希尔排序：

时间复杂度结于基本排序算法和高效算法之间，基本不需要什么额外内存，空间复杂度低，对于中等规模的元素序列是一种很好的选择

（5）快速排序：

非常高效的排序算法，适合于元素个数很大的情况。

（6）堆排序：

非常高效的排序算法，适合于元素个数很大的情况，没有什么最坏情况会导致堆排序的运行明显变慢，基本不需要额外的空间，排序不稳定。

（7）归并排序：

是一种稳定的高效排序算法，适合于元素个数很大的情况，其性能与输入元素序列无关，主要缺点是直接执行时会需要 O（n）的附加内存空间。

（8）基数排序：

其线性时间开销其实不必快速排序的时间开销小很多，并不适用于规模很小的元素序列。

3、一般情况

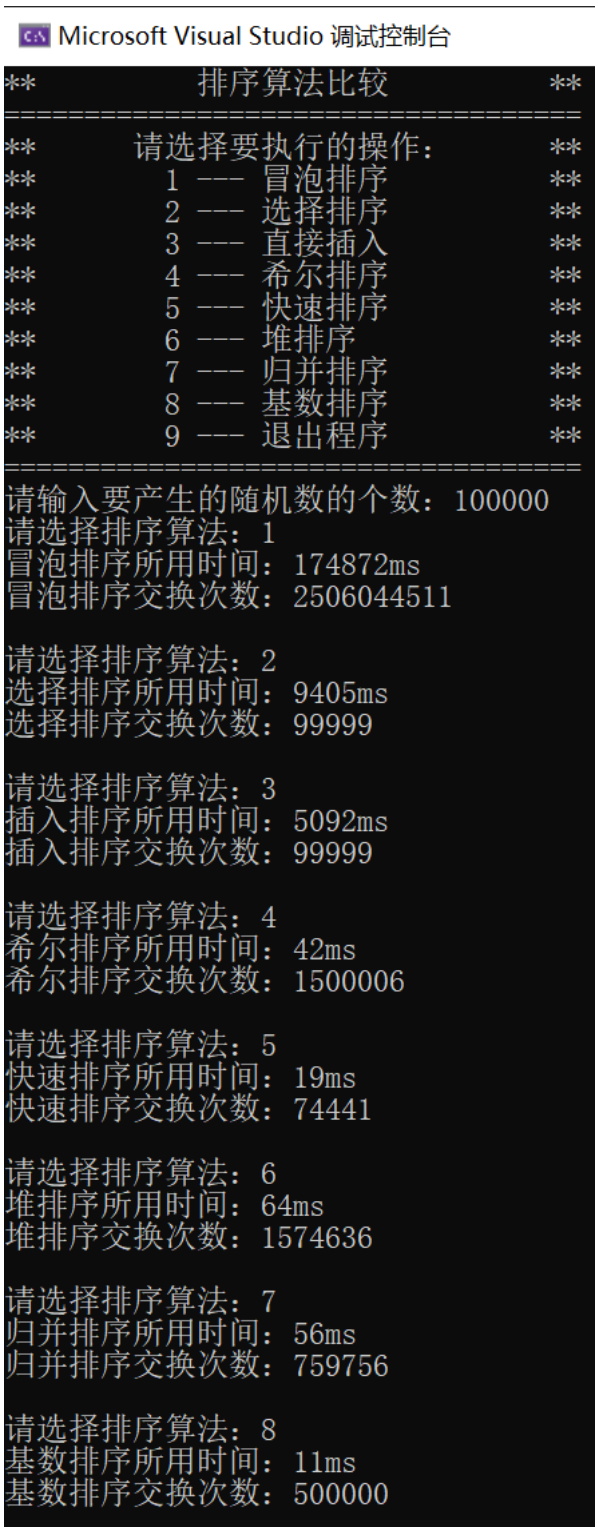
Microsoft Visual Studio 调试控制台

```
**                排序算法比较                **
====================================
**            请选择要执行的操作:             **
**              1 --- 冒泡排序                **
**              2 --- 选择排序                **
**              3 --- 直接插入                **
**              4 --- 希尔排序                **
**              5 --- 快速排序                **
**              6 --- 堆排序                  **
**              7 --- 归并排序                **
**              8 --- 基数排序                **
**              9 --- 退出程序                **
====================================
请输入要产生的随机数的个数：100
请选择排序算法：1
冒泡排序所用时间：0ms
冒泡排序交换次数：2609

请选择排序算法：2
选择排序所用时间：0ms
选择排序交换次数：99

请选择排序算法：3
插入排序所用时间：0ms
插入排序交换次数：99

请选择排序算法：4
希尔排序所用时间：0ms
希尔排序交换次数：503

请选择排序算法：5
快速排序所用时间：0ms
快速排序交换次数：65

请选择排序算法：6
堆排序所用时间：1ms
堆排序交换次数：581

请选择排序算法：7
归并排序所用时间：0ms
归并排序交换次数：258

请选择排序算法：8
基数排序所用时间：0ms
基数排序交换次数：200
```

Microsoft Visual Studio 调试控制台

```
**                排序算法比较                **
====================================
**            请选择要执行的操作:             **
**              1 --- 冒泡排序                **
**              2 --- 选择排序                **
**              3 --- 直接插入                **
**              4 --- 希尔排序                **
**              5 --- 快速排序                **
**              6 --- 堆排序                  **
**              7 --- 归并排序                **
**              8 --- 基数排序                **
**              9 --- 退出程序                **
====================================
请输入要产生的随机数的个数：1000
请选择排序算法：1
冒泡排序所用时间：34ms
冒泡排序交换次数：253384

请选择排序算法：2
选择排序所用时间：3ms
选择排序交换次数：999

请选择排序算法：3
插入排序所用时间：1ms
插入排序交换次数：999

请选择排序算法：4
希尔排序所用时间：0ms
希尔排序交换次数：8006

请选择排序算法：5
快速排序所用时间：0ms
快速排序交换次数：679

请选择排序算法：6
堆排序所用时间：1ms
堆排序交换次数：9087

请选择排序算法：7
归并排序所用时间：1ms
归并排序交换次数：4306

请选择排序算法：8
基数排序所用时间：0ms
基数排序交换次数：3000
```


Microsoft Visual Studio 调试控制台
** 排序算法比较 **
==========
** 请选择要执行的操作： **
** 1 --- 冒泡排序 **
** 2 --- 选择排序 **
** 3 --- 直接插入 **
** 4 --- 希尔排序 **
** 5 --- 快速排序 **
** 6 --- 堆排序 **
** 7 --- 归并排序 **
** 8 --- 基数排序 **
** 9 --- 退出程序 **
==========
请输入要产生的随机数的个数：10000
请选择排序算法：1
冒泡排序所用时间：1662ms
冒泡排序交换次数：25341218
请选择排序算法：2
选择排序所用时间：108ms
选择排序交换次数：9999
请选择排序算法：3
插入排序所用时间：59ms
插入排序交换次数：9999
请选择排序算法：4
希尔排序所用时间：2ms
希尔排序交换次数：120005
请选择排序算法：5
快速排序所用时间：3ms
快速排序交换次数：6808
请选择排序算法：6
堆排序所用时间：7ms
堆排序交换次数：124191
请选择排序算法：7
归并排序所用时间：9ms
归并排序交换次数：59128
请选择排序算法：8
基数排序所用时间：2ms
基数排序交换次数：40000


Microsoft Visual Studio 调试控制台
** 排序算法比较 **
==========
** 请选择要执行的操作： **
** 1 --- 冒泡排序 **
** 2 --- 选择排序 **
** 3 --- 直接插入 **
** 4 --- 希尔排序 **
** 5 --- 快速排序 **
** 6 --- 堆排序 **
** 7 --- 归并排序 **
** 8 --- 基数排序 **
** 9 --- 退出程序 **
==========
请输入要产生的随机数的个数：100000
请选择排序算法：1
冒泡排序所用时间：174872ms
冒泡排序交换次数：2506044511
请选择排序算法：2
选择排序所用时间：9405ms
选择排序交换次数：99999
请选择排序算法：3
插入排序所用时间：5092ms
插入排序交换次数：99999
请选择排序算法：4
希尔排序所用时间：42ms
希尔排序交换次数：1500006
请选择排序算法：5
快速排序所用时间：19ms
快速排序交换次数：74441
请选择排序算法：6
堆排序所用时间：64ms
堆排序交换次数：1574636
请选择排序算法：7
归并排序所用时间：56ms
归并排序交换次数：759756
请选择排序算法：8
基数排序所用时间：11ms
基数排序交换次数：500000