《数据结构》上机报告

2020 年 12 月 24 日

姓名： 王上游　　学号：1850767　班级：　19 计科 2 班　　得分：

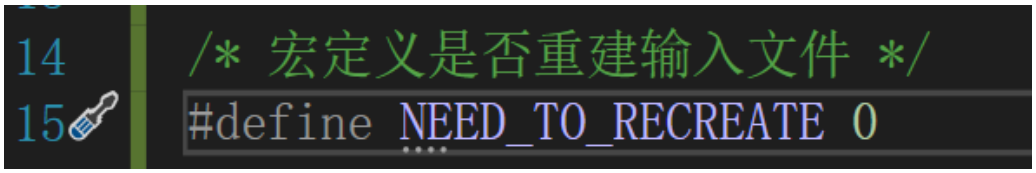
实验题目	排序算法实验报告
实验目的	1、 掌握基于比较的各种排序算法的实现； 2、掌握各种排序算法的特点，时间复杂度、稳定性。
问题描述	1. 随机生成不同规模的数据（10，100，1K，10K，100K，1M，10K 正序，10K 逆序），并保存到文件中，以 input1.txt, input2.txt, ..., input8.txt 命名。 2. 用上面产生的数据，对不同的排序方法（插入排序、选择排序、冒泡排序、希尔排序、堆排序、快速排序、归并排序）进行排序测试，给出实验时间。 3. 分析各种排序算法的特点，给出时间复杂度，以及是否是稳定排序。
基本要求	1. 程序要添加适当的注释，程序的书写要采用 缩进格式 。 2. 程序要具在一定的 健壮性，即当输入数据非法时， 程序也能适当地做出反应，如 插入删除时指定的位置不对 等等。 3. 程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作。 4. 根据实验报告模板详细书写实验报告，在实验报告中给出主要算法的复杂度分析。 5. 实验结果以表格形式列出，给出测试环境（硬件环境和软件环境），并对结果进行分析，说明各种排序算法的优缺点。 6. 将实验报告和输入文件打包成.zip 文件，上传。
选做要求	无
	已完成选做内容（序号）：无

数据结构设计	所有排序均在动态内存申请的 int 数组中完成 部分外部排序算法，辅助空间同样是动态内存申请的 int 数组

功能（函数）说明	/* 插入排序 */
	void InsertionSort(int array[], const int size);
	
	/* 选择排序 */
	void SelectionSort(int array[], const int size);
	
	/* 冒泡排序 */
	void BubbleSort(int array[], const int size);
	
	/* 希尔排序 */
	void ShellSort(int array[],const int size);
	
	/* 堆排序调整堆顶 */
	void HeapAdjust(int array[], int i, int length);
	
	/* 堆排序 */
	void HeapSort(int array[],const int size);
	
	/* 快速排序 */
	void QuickSort(int a[], int l, int r);
	
	/* 归并排序 */
	void MergeSort(int array[], const int size);

界面设计和使用说明

若希望继续使用原有的输入文件测试，将宏定义 NEED_TO_RECREATE 置 0
否则每次运行会生成新的文件 input1.txt, input2.txt, ...,input8.txt



运行后，屏幕上显示当前正在进行的测试项目，可以对速度作直观感受

```
[root@localhost 文档]# c++ -Wall -o 10-3 10-3-linux.cpp
[root@localhost 文档]# ./10-3
已成功建立测试输入文件0x7ffdad047360,大小10,模式为随机
已成功建立测试输入文件0x7ffdad047360,大小100,模式为随机
已成功建立测试输入文件0x7ffdad047360,大小1024,模式为随机
已成功建立测试输入文件0x7ffdad047360,大小10240,模式为随机
已成功建立测试输入文件0x7ffdad047360,大小102400,模式为随机
已成功建立测试输入文件0x7ffdad047360,大小1048576,模式为随机
已成功建立测试输入文件0x7ffdad047360,大小10240,模式为顺序
已成功建立测试输入文件0x7ffdad047360,大小10240,模式为逆序
测试输入文件创建完成
当前正在测试,输入 : input1.txt  大小 :     10 排序方法 : 插入排序
当前正在测试,输入 : input1.txt  大小 :     10 排序方法 : 选择排序
当前正在测试,输入 : input1.txt  大小 :     10 排序方法 : 冒泡排序
当前正在测试,输入 : input1.txt  大小 :     10 排序方法 : 希尔排序
当前正在测试,输入 : input1.txt  大小 :     10 排序方法 :  堆排序
当前正在测试,输入 : input1.txt  大小 :     10 排序方法 : 快速排序
当前正在测试,输入 : input1.txt  大小 :     10 排序方法 : 归并排序
当前正在测试,输入 : input2.txt  大小 :    100 排序方法 : 插入排序
当前正在测试,输入 : input2.txt  大小 :    100 排序方法 : 选择排序
当前正在测试,输入 : input2.txt  大小 :    100 排序方法 : 冒泡排序
当前正在测试,输入 : input2.txt  大小 :    100 排序方法 : 希尔排序
当前正在测试,输入 : input2.txt  大小 :    100 排序方法 :  堆排序
当前正在测试,输入 : input2.txt  大小 :    100 排序方法 : 快速排序
当前正在测试,输入 : input2.txt  大小 :    100 排序方法 : 归并排序
当前正在测试,输入 : input3.txt  大小 :   1024 排序方法 : 插入排序
当前正在测试,输入 : input3.txt  大小 :   1024 排序方法 : 选择排序
当前正在测试,输入 : input3.txt  大小 :   1024 排序方法 : 冒泡排序
当前正在测试,输入 : input3.txt  大小 :   1024 排序方法 : 希尔排序
当前正在测试,输入 : input3.txt  大小 :   1024 排序方法 :  堆排序
当前正在测试,输入 : input3.txt  大小 :   1024 排序方法 : 快速排序
当前正在测试,输入 : input3.txt  大小 :   1024 排序方法 : 归并排序
当前正在测试,输入 : input4.txt  大小 :  10240 排序方法 : 插入排序
当前正在测试,输入 : input4.txt  大小 :  10240 排序方法 : 选择排序
当前正在测试,输入 : input4.txt  大小 :  10240 排序方法 : 冒泡排序
当前正在测试,输入 : input4.txt  大小 :  10240 排序方法 : 希尔排序
当前正在测试,输入 : input4.txt  大小 :  10240 排序方法 :  堆排序
当前正在测试,输入 : input4.txt  大小 :  10240 排序方法 : 快速排序
当前正在测试,输入 : input4.txt  大小 :  10240 排序方法 : 归并排序
当前正在测试,输入 : input5.txt  大小 : 102400 排序方法 : 插入排序
当前正在测试,输入 : input5.txt  大小 : 102400 排序方法 : 选择排序
当前正在测试,输入 : input5.txt  大小 : 102400 排序方法 : 冒泡排序
当前正在测试,输入 : input5.txt  大小 : 102400 排序方法 : 希尔排序
当前正在测试,输入 : input5.txt  大小 : 102400 排序方法 :  堆排序
当前正在测试,输入 : input5.txt  大小 : 102400 排序方法 : 快速排序
当前正在测试,输入 : input5.txt  大小 : 102400 排序方法 : 归并排序
当前正在测试,输入 : input6.txt  大小 : 1048576 排序方法 : 插入排序
当前正在测试,输入 : input6.txt  大小 : 1048576 排序方法 : 选择排序
当前正在测试,输入 : input6.txt  大小 : 1048576 排序方法 : 冒泡排序
```

测试的精确用时结果，写入 test_results.txt 文件，程序全部运行结束后可以查看。

```
以下是input7.txt作为输入,数组大小为10240,使用冒泡排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 2005797
0.200580秒
以下是input7.txt作为输入,数组大小为10240,使用希尔排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 7480
0.000748秒
以下是input7.txt作为输入,数组大小为10240,使用堆排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 23401
0.002340秒
以下是input7.txt作为输入,数组大小为10240,使用快速排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 7468
0.000747秒
以下是input7.txt作为输入,数组大小为10240,使用归并排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 117695
0.011770秒
以下是input8.txt作为输入,数组大小为10240,使用插入排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 2504487
0.250449秒
以下是input8.txt作为输入,数组大小为10240,使用选择排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 2254649
0.225465秒
以下是input8.txt作为输入,数组大小为10240,使用冒泡排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 3194983
0.319498秒
以下是input8.txt作为输入,数组大小为10240,使用希尔排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 8396
0.000840秒
以下是input8.txt作为输入,数组大小为10240,使用堆排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 18111
0.001811秒
以下是input8.txt作为输入,数组大小为10240,使用快速排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 8225
0.000822秒
以下是input8.txt作为输入,数组大小为10240,使用归并排序的测试,测试用时如下:
计数器频率 : 10000000Hz
计数器计数 : 288463
0.028846秒
=======================测试结束=======================

```

调试分析

调试结果发现，100K 和 1M 数据量时，插入、选择、冒泡三种排序速度无法接受，与其他排序算法耗时差距很大。1MB 数据量的 Input6.txt 输入，插入、选择、冒泡三种排序用时非常长，冒泡排序需要 1 小时以上。各排序算法的测试用时与算法的复杂度一致。

下图是 Windows 系统运行结果。

	插入	选择	冒泡	希尔	堆	快速	归并
input1	0.000001	0.000001	0.000001	0.000002	0.000002	0.000001	0.000009
input2	0.000012	0.000018	0.000027	0.000008	0.000011	0.00001	0.00014
input3	0.000735	0.001413	0.00204	0.000154	0.000112	0.00009	0.000694
input4	0.052691	0.104357	0.20762	0.002409	0.001256	0.000972	0.006736
input5	5.411578	9.784334	23.636488	0.028424	0.015675	0.010763	0.068737
input6	640.782873	1192.94285	3761.667948	2.0467	0.304949	0.177799	1.237173
input7	0.000035	0.179378	0.20058	0.000748	0.00234	0.000747	0.01177
input8	0.250449	0.225465	0.319498	0.00084	0.001811	0.000822	0.028846

硬件环境：

制造商:	HUAWEI
型号:	MateBook X Pro
处理器:	Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz
已安装的内存(RAM):	8.00 GB (7.85 GB 可用)
系统类型:	64 位操作系统，基于 x64 的处理器
笔和触控:	为 10 触摸点提供触控支持

软件环境：VS2019 16.7.6

下图是 Linux 虚拟机运行结果：

	插入	选择	冒泡	希尔	堆	快速	归并
input1	0.000001	0.000001	0	0.000001	0.000001	0.000001	0.000002
input2	0.000007	0.000013	0.000021	0.000007	0.000006	0.000006	0.00001
input3	0.000613	0.001094	0.001792	0.000129	0.000089	0.00008	0.000118
input4	0.060491	0.122749	0.223741	0.00237	0.001101	0.000944	0.001448
input5	5.925202	10.599004	27.895223	0.050236	0.026089	0.020878	0.030751
input6	797.003181	1390.948391	4115.337851	3.074135	0.371758	0.236576	0.385799
input7	0.000098	0.183498	0.194046	0.000763	0.001866	0.000443	0.002398
input8	0.202013	0.176851	0.332802	0.000993	0.001897	0.000491	0.002447

硬件环境：硬盘(SISC) 20GB 内存 8GB
软件环境：VMWare CentOS-7 gcc 版本 8.3.1 20191121 (Red Hat 8.3.1-5) (GCC)

详细结果已放入 zip 随报告提交。

心得体会

（对整个实验过程做出总结，对重要的算法做出性能分析。）

（1） 插入排序

插入排序是一种简单直观的排序算法。

它的工作原理为将待排列元素划分为“已排序”和“未排序”两部分，每次从“未排序的”元素中选择一个插入到“已排序的”元素中的正确位置。

插入排序是一种稳定的排序算法。插入排序的最优时间复杂度为 $O(n)$，在数列几乎有序时效率很高。插入排序的最坏时间复杂度和平均时间复杂度都为 $O(n^2)$。

优点是稳定，缺点是数据量庞大时，每次需要移动数据太多。

（2） 选择排序

选择排序是一种简单直观的排序算法。

选择排序的工作原理是每次找出第 i 小的元素（也就是[i,size-1]中最小的元素），然后将这个元素与数组第 i 个位置上的元素交换。

由于交换操作的存在，选择排序不稳定。选择排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 $O(n^2)$。

优点是移动次数固定。缺点是比较次数太多，交换次数不稳定。

（3） 冒泡排序

冒泡排序是一种简单的排序算法。由于在算法的执行过程中，较小的元素像是气泡般慢慢浮到数列的顶端，故叫做冒泡排序。

它的工作原理是每次检查相邻两个元素，如果前面的元素与后面的元素满足给定的排序条件，就将相邻两个元素交换。当没有相邻的元素需要交换时，排序就完成了。冒泡排序是一种稳定的排序算法。经过 i 次扫描后，数列的末尾 i 项必然是最大的 i 项，因此冒泡排序最多需要扫描 n-1 遍数组就能完成排序。

冒泡排序的最优时间复杂度为 $O(n)$，在数列几乎有序时效率很高。冒泡排序的最坏时间复杂度和平均时间复杂度都为 $O(n^2)$。

优点是稳定，趟数和比较次数都固定。缺点是太慢。

（4） 希尔排序

希尔排序，也称为缩小增量排序法，是插入排序的一种改进版本。

工作原理是排序对不相邻的记录进行比较和移动：

1. 将待排序序列分为若干子序列（每个子序列的元素在原始数组中间距相同）；
2. 对这些子序列进行插入排序；
3. 减小每个子序列中元素之间的间距，重复上述过程直至间距减少为 1。

希尔排序是一种不稳定的排序算法。希尔排序的最优时间复杂度为 $O(n)$。希尔排序的平均时间复杂度和最坏时间复杂度与间距序列的选取（就是间距如何减小到 1）有关，比如

间距每次除以 3 的希尔排序的时间复杂度是 $O(n^{3/2})$。已知最好的最坏时间复杂度为 $O(n(logn)^2)$ 。空间复杂度 $O(n)$。

缺点是不稳定，选取合适的间距序列只能凭经验或感觉。

（5） 堆排序

堆排序是指利用堆这种数据结构所设计的一种排序算法。堆排序的适用数据结构为数组。

它的工作原理为对所有待排序元素建堆，然后依次取出堆顶元素，就可以得到排好序的序列。当当前的结点下标为 i 时，父结点、左子结点和右子结点的选择方式如下：

```
//这里 floor 函数将实数映射到最小的前导整数。
iParent(i) = floor((i - 1) / 2);
iLeftChild(i) = 2 * i + 1;
iRightChild(i) = 2 * i + 2;
```

堆排序是一种不稳定的排序算法。堆排序的最优时间复杂度、平均时间复杂度、最坏时间复杂度均为 O(nlogn)。

优点是适合找前 k 大/小元素。缺点是不稳定，而且需要少量辅助空间。

（6） 快速排序

快速排序，又称分区交换排序，简称快排，是一种被广泛运用的排序算法。

快速排序的工作原理是通过分治的方式来将一个数组排序。
快速排序分为三个过程：

1. 将数列划分为两部分（要求保证相对大小关系）；
2. 递归到两个子序列中分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序。

和归并排序不同，第一步并不是直接分成前后两个序列，而是在分的过程中要保证相对大小关系。具体来说，第一步要是要把数列分成两个部分，然后保证前一个子数列中的数都小于后一个子数列中的数。为了保证平均时间复杂度，一般是随机选择一个数 m 来当做两个子数列的分界。之后，维护一前一后两个指针 p 和 q，依次考虑当前的数是否放在了应该放的位置（前还是后）。如果当前的数没放对，比如说如果后面的指针 q 遇到了一个比 m 小的数，那么可以交换 p 和 q 位置上的数，再把 p 向后移一位。当前的数的位置全放对后，再移动指针继续处理，直到两个指针相遇。其实，快速排序没有指定应如何具体实现第一步，不论是选择 m 的过程还是划分的过程，都有不止一种实现方法。第三步中的序列已经分别有序且第一个序列中的数都小于第二个数，所以直接拼接起来就好了。

快速排序是一种不稳定的排序算法。快速排序的最优时间复杂度和平均时间复杂度为 O(nlogn)，最坏时间复杂度为 $O(n^2)$。

	优点是速度快，缺点是递归空间深，长度对递归划分子集的性能有影响，可能出现分治不均。此外，在规模很小时表现不好。 （7） 归并排序 归并排序是一种采用了 分治 思想的排序算法。 归并排序分为三个步骤： 将数列划分为两部分； 递归地分别对两个子序列进行归并排序； 合并两个子序列。 不难发现，归并排序的前两步都很好实现，关键是如何合并两个子序列。注意到两个子序列在第二步中已经保证了都是有序的了，第三步中实际上是想要把两个 有序 的序列合并起来。 归并排序是一种稳定的排序算法。归并排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 O(nlogn)。归并排序的空间复杂度为 O(n)。 优点是速度快而稳定。缺点是需要很大辅助空间，空间复杂度高。

代码实现	**(1) Winsows**
	
	`#define _CRT_SECURE_NO_WARNINGS`
	`#include <iostream>`
	`#include <fstream>`
	`#include <time.h>`
	`#include <windows.h>`
	`#include <cstdio>`
	`#include <cstring>`
	`#include <cstdlib>`
	`#include <algorithm>`
	`#include <iomanip>`
	`#include <string>`
	`using namespace std;`
	
	`/* 宏定义是否重建输入文件 */`
	`#define NEED_TO_RECREATE 0`
	
	`#define MAX_SIZE 1024*1024`
	
	`#define TEST_METHOD_NUM 7`
	`/* 随机数据三种模式 */`

```
enum class INPUT_MODE {
    RAND,ORDERING,REVERSE
};

/* 排序方法 */
enum class SORT_METHOD {
    Insertion,Selection,Bubble,Shell,Heap,Quick,Merge
};

const string SORT_METHOD_NAME[TEST_METHOD_NUM] = {
    "插入排序","选择排序","冒泡排序","希尔排序",
    "  堆排序","快速排序","归并排序"
};

void Create_Input_File(const char* filename,const int size,const INPUT_MODE im)
{
    if (size < 0)
        return;

    ofstream fout;
    fout.open(filename, ios::out | ios::binary);
    if (!fout.is_open())
        cerr << "open " << filename << "error!" << endl;

    int* p = new(nothrow) int[size];
    if (!p)
    {
        fout.close();
        exit(-1);
    }

    for (int i = 0; i < size; i++)
        p[i] = rand();

    if (im == INPUT_MODE::RAND)
    {
        for (int j = 0; j < size; j++)
            fout << p[j] << " ";
    }
    else if (im == INPUT_MODE::ORDERING)
    {
        sort(p, p + size);
        for (int j = 0; j < size; j++)
            fout << p[j] << " ";
```

```
    }
    else if (im == INPUT_MODE::REVERSE)
    {
        sort(p, p + size);
        for (int j = size - 1; j >= 0; j--)
            fout << p[j] << " ";
    }

    fout.close();

    return;
}

void Create_My_Test_Files()
{
    const int size[] = { 10,100,1024,10 * 1024,100 * 1024,1024 * 1024,10 * 1024,10 * 1024 ,0 };
    const INPUT_MODE im[] = { INPUT_MODE::RAND,INPUT_MODE::RAND, INPUT_MODE::RAND,
INPUT_MODE::RAND,
                              INPUT_MODE::RAND, INPUT_MODE::RAND,
INPUT_MODE::ORDERING, INPUT_MODE::REVERSE };
    const string filename[] = {

    "input1.txt","input2.txt","input3.txt","input4.txt","input5.txt","input6.txt","input7.txt","input8.txt",
    };

    for (int i = 0; size[i]; i++)
    {
        Create_Input_File(filename[i].c_str(), size[i], im[i]);

        cout << "已成功建立测试输入文件" << filename[i] << ",大小" << size[i] << ",模式为";
        switch (im[i])
        {
            case INPUT_MODE::RAND:
                cout << "随机";
                break;
            case INPUT_MODE::ORDERING:
                cout << "顺序";
                break;
            case INPUT_MODE::REVERSE:
                cout << "逆序";
                break;
        }
        cout << endl;
    }
```

```
    return;
}

/* 插入排序 */
void InsertionSort(int array[], const int size)
{
    for (int i = 1; i < size; i++)// size - 1 趟插入
    {
        /* 寻找元素 array[i] 合适的插入位置 */
        if (array[i] < array[i - 1])//否则不用往前插
        {
            int t = array[i];
            array[i] = array[i - 1];
            int j = i - 2;
            for (j = i - 2; j >= 0 && array[j] > t; j--)//后移一位
                array[j + 1] = array[j];
            array[j + 1] = t;//在次插入
        }
    }
}

/* 选择排序 */
void SelectionSort(int array[], const int size)
{
    for (int i = 0; i < size; i++)// size - 1 趟选择
    {
        int k = i;
        for (int j = i; j < size; j++)//从[i,size)选出最小者,其下标为k
        {
            if (array[j] < array[k])
                k = j;
        }
        /* 最小者array[k]换到位置i */
        int t = array[i];
        array[i] = array[k];
        array[k] = t;
    }
}

/* 冒泡排序 */
void BubbleSort(int array[], const int size)
{
    for (int i = 0; i < size - 1; i++)// size - 2 趟冒泡
```

```
    {
        for (int j = 0; j < size - i - 1; j++)
        {
            if (array[j] > array[j + 1])
            {
                int t = array[j];
                array[j] = array[j + 1];
                array[j + 1] = t;
            }
        }
    }
}

/* 希尔排序 */
void ShellSort(int array[],const int size)
{
    //增量gap，并逐步缩小增量
    for (int gap = size / 2; gap > 0; gap /= 2)
    {
        //从第gap个元素，逐个对其所在组进行直接插入排序操作
        for (int i = gap; i < size; i++)
        {
            int j = i;
            int temp = array[j];
            if (array[j] < array[j - gap])
            {
                while (j - gap >= 0 && temp < array[j - gap])
                {
                    //移动法
                    array[j] = array[j - gap];
                    j -= gap;
                }
                array[j] = temp;
            }
        }
    }
}

/* 堆排序调整堆顶 */
void HeapAdjust(int array[], int i, int length)
{
    int temp = array[i];//先取出当前元素i
    for (int k = i * 2 + 1; k < length; k = k * 2 + 1)
    {
```

```
            //从i结点的左子结点开始，也就是2i+1处开始
            if (k + 1 < length && array[k] < array[k + 1])//如果左子结点小于右子结点，k指向右子结点
                k++;

            if (array[k] > temp)
            {   //如果子节点大于父节点，将子节点值赋给父节点（不用进行交换）
                array[i] = array[k];
                i = k;
            }
            else
                break;
        }
        array[i] = temp;//将temp值放到最终的位置
}

/* 堆排序 */
void HeapSort(int array[],const int size)
{
    //1.构建大顶堆
    for (int i = size / 2 - 1; i >= 0; i--)
    {
        //从第一个非叶子结点从下至上，从右至左调整结构
        HeapAdjust(array, i, size);
    }
    //2.调整堆结构+交换堆顶元素与末尾元素
    for (int j = size - 1; j > 0; j--)
    {
        //将堆顶元素与末尾元素进行交换
        int t = array[0];
        array[0] = array[j];
        array[j] = t;
        HeapAdjust(array, 0, j);//重新对堆进行调整
    }

}

/* 快速排序 */
void QuickSort(int a[], int l, int r)
{
    int mid = a[(l + r) / 2];
    int i = l;
    int j = r;
    do
    {
```

```
            while (a[i] < mid)//找到mid左侧比mid大的数
                i++;
            while (a[j] > mid)//找到mid右侧比mid小的数
                j--;

            if (i <= j)//要加这句
            {
                int t = a[i];
                a[i] = a[j];
                a[j] = t;
                i++;
                j--;
            }
        } while (i <= j);

        if (i < r)
            QuickSort(a, i, r);//[i,r]区间内不保证按序，仅能确定他们都大于mid，递归排序

        if (j > l)
            QuickSort(a, l, j);//[l,j]区间内不保证按序，仅能确定他们都小于mid，递归排序
}

/* 归并排序 */
void MergeSort(int array[], const int size)
{
    if (size < 2)
        return;

    int mid = size / 2;

    int* l = new(nothrow) int[mid + 1];
    if (!l)
        exit(-1);

    int* r = new(nothrow) int[mid + 1];
    if (!r)
    {
        delete[] l;
        exit(-1);
    }

    int li = 0;
    int ri = 0;
```

```
    for (int i = 0; i < mid; i++)
        l[li++] = array[i];
    for (int i = mid; i < size; i++)
        r[ri++] = array[i];

    int l_size = mid;
    int r_size = size - mid;

    MergeSort(l,l_size);
    MergeSort(r,r_size);

    int i, j, k;
    i = j = k = 0;

    while (i < l_size && j < r_size)
        if (l[i] <= r[j])
            array[k++] = l[i++];
        else
            array[k++] = r[j++];

    while (i < l_size)
        array[k++] = l[i++];
    while (j < r_size)
        array[k++] = r[j++];

    delete[] l;
    delete[] r;
}

void test(const char* testfilename, const int size, const SORT_METHOD sm,ofstream& fout)
{
    ifstream fin;
    fin.open(testfilename, ios::in | ios::binary);
    if (!fin.is_open())
        cerr << "open " << testfilename << "error!" << endl;

    int* array = new(nothrow) int[MAX_SIZE];
    if (!array)
        exit(-1);

    for (int i = 0; i < size; i++)
        fin >> array[i];

    fin.close();
```

```
LARGE_INTEGER tick, begin, end;

QueryPerformanceFrequency(&tick); //获得计数器频率
QueryPerformanceCounter(&begin);  //获得初始硬件计数器计数

switch (sm)
{
    case SORT_METHOD::Insertion:
    {
        InsertionSort(array, size);
        break;
    }
    case SORT_METHOD::Selection:
    {
        SelectionSort(array, size);
        break;
    }
    case SORT_METHOD::Bubble:
    {
        BubbleSort(array, size);
        break;
    }
    case SORT_METHOD::Shell:
    {
        ShellSort(array, size);
        break;
    }
    case SORT_METHOD::Heap:
    {
        HeapSort(array, size);
        break;
    }
    case SORT_METHOD::Quick:
    {
        QuickSort(array, 0, size - 1);
        break;
    }
    case SORT_METHOD::Merge:
    {
        MergeSort(array, size);
        break;
    }
}
```

```
QueryPerformanceCounter(&end);          //获得终止硬件计数器计数

fout << "以下是" << testfilename << "作为输入,数组大小为" << size << ",使用";
switch (sm)
{
    case SORT_METHOD::Insertion:
    {
        fout << "插入";
        break;
    }
    case SORT_METHOD::Selection:
    {
        fout << "选择";
        break;
    }
    case SORT_METHOD::Bubble:
    {
        fout << "冒泡";
        break;
    }
    case SORT_METHOD::Shell:
    {
        fout << "希尔";
        break;
    }
    case SORT_METHOD::Heap:
    {
        fout << "堆";
        break;
    }
    case SORT_METHOD::Quick:
    {
        fout << "快速";
        break;
    }
    case SORT_METHOD::Merge:
    {
        fout << "归并";
        break;
    }
}

fout << "排序的测试,测试用时如下:" << endl;
```

```
    fout << "计数器频率 ：" << tick.QuadPart << "Hz" << endl;
    fout << "计数器计数 ：" << end.QuadPart - begin.QuadPart << endl;
    fout << setiosflags(ios::fixed) << setprecision(6) << double(end.QuadPart - begin.QuadPart) /
tick.QuadPart << "秒" << endl;

    delete[]array;
}

int main()
{
    srand(unsigned int(time(NULL)));

    /* 是否重建输入文件 */
    if (NEED_TO_RECREATE)
    {
        Create_My_Test_Files();
        cout << "测试输入文件创建完成" << endl;
    }

    const int size[] = { 10,100,1024,10 * 1024,100 * 1024,1024 * 1024,10 * 1024,10 * 1024 ,0 };
    const INPUT_MODE im[] = { INPUT_MODE::RAND,INPUT_MODE::RAND, INPUT_MODE::RAND,
INPUT_MODE::RAND,
                              INPUT_MODE::RAND, INPUT_MODE::RAND,
INPUT_MODE::ORDERING, INPUT_MODE::REVERSE };
    const string filename[] = {

    "input1.txt","input2.txt","input3.txt","input4.txt","input5.txt","input6.txt","input7.txt","input8.txt",
    };
    const SORT_METHOD sm[TEST_METHOD_NUM] = {
        SORT_METHOD::Insertion,SORT_METHOD::Selection,SORT_METHOD::Bubble,

    SORT_METHOD::Shell,SORT_METHOD::Heap,SORT_METHOD::Quick,SORT_METHOD::Merge
    };

    ofstream outres;
    outres.open("test_results.txt", ios::out | ios::binary);
    if (!outres.is_open())
    {
        cerr << "open test_results.txt error!" << endl;
        exit(-1);
    }

    outres << "======================测试正式开始======================" << endl;
```

```
    for (int i = 0; size[i]; i++)
    {
        for (int j = 0;j < TEST_METHOD_NUM;j++)
        {
            cout << "当前正在测试,输入 ：" << filename[i] << "   大小 ：" << setw(6) << size[i] << "  排
序方法 ：" << setw(6) << SORT_METHOD_NAME[j] << endl;
            test(filename[i].c_str(), size[i], sm[j], outres);
        }
    }

    outres << "=======================测试结束=======================" << endl;
    outres.close();

    return 0;
}
```

(2) Linux

```
#include <iostream>
#include <fstream>
#include <time.h>
#include <sys/time.h>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <algorithm>
#include <iomanip>
#include <string>
using namespace std;

/* 宏定义是否重建输入文件 */
#define NEED_TO_RECREATE 1

#define MAX_SIZE 1024 * 1024

#define TEST_METHOD_NUM 7
/* 随机数据三种模式 */
enum class INPUT_MODE
{
    RAND,
    ORDERING,
    REVERSE
};

/* 排序方法 */
```

```
enum class SORT_METHOD
{
	Insertion,
	Selection,
	Bubble,
	Shell,
	Heap,
	Quick,
	Merge
};

const string SORT_METHOD_NAME[TEST_METHOD_NUM] = {
	"插入排序", "选择排序", "冒泡排序", "希尔排序",
	"  堆排序", "快速排序", "归并排序"};

void Create_Input_File(const char *filename, const int size, const INPUT_MODE im)
{
	if (size < 0)
		return;

	ofstream fout;
	fout.open(filename, ios::out | ios::binary);
	if (!fout.is_open())
		cerr << "open " << filename << "error!" << endl;

	int *p = new (nothrow) int[size];
	if (!p)
	{
		fout.close();
		exit(-1);
	}

	for (int i = 0; i < size; i++)
		p[i] = rand();

	if (im == INPUT_MODE::RAND)
	{
		for (int j = 0; j < size; j++)
			fout << p[j] << " ";
	}
	else if (im == INPUT_MODE::ORDERING)
	{
		sort(p, p + size);
		for (int j = 0; j < size; j++)
```

```
            fout << p[j] << " ";
    }
    else if (im == INPUT_MODE::REVERSE)
    {
        sort(p, p + size);
        for (int j = size - 1; j >= 0; j--)
            fout << p[j] << " ";
    }

    fout.close();

    return;
}

void Create_My_Test_Files()
{
    const int size[] = {10, 100, 1024, 10 * 1024, 100 * 1024, 1024 * 1024, 10 * 1024, 10 * 1024, 0};
    const INPUT_MODE im[] = {INPUT_MODE::RAND, INPUT_MODE::RAND, INPUT_MODE::RAND,
INPUT_MODE::RAND,
                                    INPUT_MODE::RAND, INPUT_MODE::RAND,
INPUT_MODE::ORDERING, INPUT_MODE::REVERSE};
    const string filename[] = {
        "input1.txt",
        "input2.txt",
        "input3.txt",
        "input4.txt",
        "input5.txt",
        "input6.txt",
        "input7.txt",
        "input8.txt",
    };

    for (int i = 0; size[i]; i++)
    {
        Create_Input_File(filename[i].c_str(), size[i], im[i]);

        cout << "已成功建立测试输入文件" << filename[i] << ",大小" << size[i] << ",模式为";
        switch (im[i])
        {
        case INPUT_MODE::RAND:
            cout << "随机";
            break;
        case INPUT_MODE::ORDERING:
            cout << "顺序";
```

```
                break;
            case INPUT_MODE::REVERSE:
                cout << "逆序";
                break;
            }
            cout << endl;
        }

        return;
}

/* 插入排序 */
void InsertionSort(int array[], const int size)
{
        for (int i = 1; i < size; i++) // size - 1 趟插入
        {
            /* 寻找元素 array[i] 合适的插入位置 */
            if (array[i] < array[i - 1]) //否则不用往前插
            {
                int t = array[i];
                array[i] = array[i - 1];
                int j = i - 2;
                for (j = i - 2; j >= 0 && array[j] > t; j--) //后移一位
                    array[j + 1] = array[j];
                array[j + 1] = t; //在次插入
            }
        }
}

/* 选择排序 */
void SelectionSort(int array[], const int size)
{
        for (int i = 0; i < size; i++) // size - 1 趟选择
        {
            int k = i;
            for (int j = i; j < size; j++) //从[i,size)选出最小者,其下标为k
            {
                if (array[j] < array[k])
                    k = j;
            }
            /* 最小者array[k]换到位置i */
            int t = array[i];
            array[i] = array[k];
            array[k] = t;
```

```
    }
}

/* 冒泡排序 */
void BubbleSort(int array[], const int size)
{
    for (int i = 0; i < size - 1; i++) // size - 2 趟冒泡
    {
        for (int j = 0; j < size - i - 1; j++)
        {
            if (array[j] > array[j + 1])
            {
                int t = array[j];
                array[j] = array[j + 1];
                array[j + 1] = t;
            }
        }
    }
}

/* 希尔排序 */
void ShellSort(int array[], const int size)
{
    //增量gap，并逐步缩小增量
    for (int gap = size / 2; gap > 0; gap /= 2)
    {
        //从第gap个元素，逐个对其所在组进行直接插入排序操作
        for (int i = gap; i < size; i++)
        {
            int j = i;
            int temp = array[j];
            if (array[j] < array[j - gap])
            {
                while (j - gap >= 0 && temp < array[j - gap])
                {
                    //移动法
                    array[j] = array[j - gap];
                    j -= gap;
                }
                array[j] = temp;
            }
        }
    }
}
```

```
/* 堆排序调整堆顶 */
void HeapAdjust(int array[], int i, int length)
{
    int temp = array[i]; //先取出当前元素i
    for (int k = i * 2 + 1; k < length; k = k * 2 + 1)
    {
        //从i结点的左子结点开始，也就是2i+1处开始
        if (k + 1 < length && array[k] < array[k + 1]) //如果左子结点小于右子结点，k指向右子结点
            k++;

        if (array[k] > temp)
        { //如果子节点大于父节点，将子节点值赋给父节点（不用进行交换）
            array[i] = array[k];
            i = k;
        }
        else
            break;
    }
    array[i] = temp; //将temp值放到最终的位置
}

/* 堆排序 */
void HeapSort(int array[], const int size)
{
    //1.构建大顶堆
    for (int i = size / 2 - 1; i >= 0; i--)
    {
        //从第一个非叶子结点从下至上，从右至左调整结构
        HeapAdjust(array, i, size);
    }
    //2.调整堆结构+交换堆顶元素与末尾元素
    for (int j = size - 1; j > 0; j--)
    {
        //将堆顶元素与末尾元素进行交换
        int t = array[0];
        array[0] = array[j];
        array[j] = t;
        HeapAdjust(array, 0, j); //重新对堆进行调整
    }
}

/* 快速排序 */
void QuickSort(int a[], int l, int r)
```

```
{
    int mid = a[(l + r) / 2];
    int i = l;
    int j = r;
    do
    {
        while (a[i] < mid) //找到mid左侧比mid大的数
            i++;
        while (a[j] > mid) //找到mid右侧比mid小的数
            j--;

        if (i <= j) //要加这句
        {
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
            i++;
            j--;
        }
    } while (i <= j);

    if (i < r)
        QuickSort(a, i, r); //[i,r]区间内不保证按序，仅能确定他们都大于mid，递归排序

    if (j > l)
        QuickSort(a, l, j); //[l,j]区间内不保证按序，仅能确定他们都小于mid，递归排序
}

/* 归并排序 */
void MergeSort(int array[], const int size)
{
    if (size < 2)
        return;

    int mid = size / 2;

    int *l = new (nothrow) int[mid + 1];
    if (!l)
        exit(-1);

    int *r = new (nothrow) int[mid + 1];
    if (!r)
    {
        delete[] l;
```

```
            exit(-1);
        }

        int li = 0;
        int ri = 0;

        for (int i = 0; i < mid; i++)
            l[li++] = array[i];
        for (int i = mid; i < size; i++)
            r[ri++] = array[i];

        int l_size = mid;
        int r_size = size - mid;

        MergeSort(l, l_size);
        MergeSort(r, r_size);

        int i, j, k;
        i = j = k = 0;

        while (i < l_size && j < r_size)
            if (l[i] <= r[j])
                array[k++] = l[i++];
            else
                array[k++] = r[j++];

        while (i < l_size)
            array[k++] = l[i++];
        while (j < r_size)
            array[k++] = r[j++];

        delete[] l;
        delete[] r;
}

void test(const char *testfilename, const int size, const SORT_METHOD sm, ofstream &fout)
{
        ifstream fin;
        fin.open(testfilename, ios::in | ios::binary);
        if (!fin.is_open())
            cerr << "open " << testfilename << "error!" << endl;

        int *array = new (nothrow) int[MAX_SIZE];
        if (!array)
```

```
        exit(-1);

    for (int i = 0; i < size; i++)
        fin >> array[i];

    fin.close();

    struct timeval t_start, t_end;
    gettimeofday(&t_start, NULL);

    switch (sm)
    {
    case SORT_METHOD::Insertion:
    {
        InsertionSort(array, size);
        break;
    }
    case SORT_METHOD::Selection:
    {
        SelectionSort(array, size);
        break;
    }
    case SORT_METHOD::Bubble:
    {
        BubbleSort(array, size);
        break;
    }
    case SORT_METHOD::Shell:
    {
        ShellSort(array, size);
        break;
    }
    case SORT_METHOD::Heap:
    {
        HeapSort(array, size);
        break;
    }
    case SORT_METHOD::Quick:
    {
        QuickSort(array, 0, size - 1);
        break;
    }
    case SORT_METHOD::Merge:
    {
```

```
        MergeSort(array, size);
        break;
    }
    }

    fout << "以下是" << testfilename << "作为输入,数组大小为" << size << ",使用";
    switch (sm)
    {
    case SORT_METHOD::Insertion:
    {
        fout << "插入";
        break;
    }
    case SORT_METHOD::Selection:
    {
        fout << "选择";
        break;
    }
    case SORT_METHOD::Bubble:
    {
        fout << "冒泡";
        break;
    }
    case SORT_METHOD::Shell:
    {
        fout << "希尔";
        break;
    }
    case SORT_METHOD::Heap:
    {
        fout << "堆";
        break;
    }
    case SORT_METHOD::Quick:
    {
        fout << "快速";
        break;
    }
    case SORT_METHOD::Merge:
    {
        fout << "归并";
        break;
    }
    }
```

```
    gettimeofday(&t_end, NULL);
    double delta = double(t_end.tv_sec - t_start.tv_sec) * 1e6 + double(t_end.tv_usec -
t_start.tv_usec);
    fout << "排序的测试,测试用时如下:" << endl;
    fout << setiosflags(ios::fixed) << setprecision(6) << delta / 1e6 << "秒" << endl;

    delete[] array;
}

int main()
{
    srand((unsigned int)(time(NULL)));

    /* 是否重建输入文件 */
    if (NEED_TO_RECREATE)
    {
        Create_My_Test_Files();
        cout << "测试输入文件创建完成" << endl;
    }

    const int size[] = {10, 100, 1024, 10 * 1024, 100 * 1024, 1024 * 1024, 10 * 1024, 10 * 1024, 0};
    const string filename[] = {
        "input1.txt",
        "input2.txt",
        "input3.txt",
        "input4.txt",
        "input5.txt",
        "input6.txt",
        "input7.txt",
        "input8.txt",
    };
    const SORT_METHOD sm[TEST_METHOD_NUM] = {
        SORT_METHOD::Insertion, SORT_METHOD::Selection, SORT_METHOD::Bubble,
        SORT_METHOD::Shell, SORT_METHOD::Heap, SORT_METHOD::Quick,
SORT_METHOD::Merge};

    ofstream outres;
    outres.open("test_results.txt", ios::out | ios::binary);
    if (!outres.is_open())
    {
        cerr << "open test_results.txt error!" << endl;
        exit(-1);
    }
```

```
    outres << "======================测试正式开始======================" << endl;

    for (int i = 0; size[i]; i++)
    {
        for (int j = 0; j < TEST_METHOD_NUM; j++)
        {
            cout << "当前正在测试,输入 ：" << filename[i] << "   大小 ：" << setw(6) << size[i] << "  排
序方法 ：" << setw(6) << SORT_METHOD_NAME[j] << endl;
            test(filename[i].c_str(), size[i], sm[j], outres);
        }
    }

    outres << "======================测试结束======================" << endl;
    outres.close();

    return 0;
}
```