数据结构课程设计报告

设计题目：内部排序算法的实现与比较

班　　级：计科221

学　　号：19222126

姓　　名：董自经

南京农业大学计算机系

数据结构课程设计报告内容

1.课程设计题目

内部排序算法的实现与比较

【问题描述】

从冒泡排序、希尔排序、折半插入排序、二路归并排序、堆排序、选择排序、快速排序、基数排序等排序方法中选用 5 种，实现数据的排序。

【基本要求】

对随机生成的 3 组 500 个待排整数进行排序，并从关键字的比较次数和关键字的移动次数来对选用的算法进行比较分析。

【选作内容】

随机产生 3 组数据，分别有 1000、10000、100000 个待排整数，存入文件，从文件读入待排数据进行排序，统计每种算法的执行时间，并进行分析比较

2. 算法设计思想

(1) 选取算法

算法中选取了经典算法中的冒泡排序、希尔排序、快速排序、二路归并排序、堆排序作为目标算法对数据进行排序。由于要求是若干个随机数，无法确定数组的具体个数，所以在进行赋值时使用了指针类型替代了数组进行排序，在用户输入具体的数字之后为指针分配相应大小的空间。

(2) 要求

基本要求中的500个待排序的数是每次进行重新赋值，同时为了确保每次排序的数组值不重复（具有普遍性），在头文件中使用了<stdlib.h>和<time.h>对指针进行初始化。选做内容中，先用fopen函数打开存有相应数据的.txt文件并将数据写入1000/10000/100000的指针，再调用fclose函数将文件关闭。之后在用户选择数据时，用fopen函数打开相应的文件并读取其中的内容赋给相应的指针后再进行排序。由于选做内容中有超过10000个待排整数而且选用了冒泡排序作为一种排序方式，比较次数会超出int的范围，故选取了double去存储比较次数和交换次数确保了最后结果的部分正常输出。

(3) 冒泡排序

冒泡排序的思想是从第一个关键字开始，将每一个关键字都与所有关键字进行一次比较，如果符合排序的条件，就将两数字进行交换，不符合则继续向后比较，直至每一个关键字都结束比较后才完成排序。

(4)希尔排序

希尔排序也称递减增量排序算法，是插入排序的一种更高效的改进版本，只是把插入排序中的步长改为了一个指定长度的变化的gap，希尔排序的基本思想是先根据gap将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录"基本有序"时，再对全体记录进行依次直接插入排序。

(5)快速排序

快速排序是对冒泡排序的一种改进算法，该算法的基本思想是先在数组中选一个基准数

（通常为数组第一个），将数组中小于基准数的数据移到基准数左边，大于基准数的移到右边。对于基准数左、右两边的数组，不断重复以上两个过程，直到每个子集只有一个元素，即为全部有序。

(6)二路归并排序

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法的一个非常典型的应用。作为一种典型的分而治之思想的算法应用，归并排序的实现有两种方法：自上而下的递归（所有递归的方法都可以用迭代重写）和自下而上的迭代；归并排序首先申请一块大小为两个已经排序序列之和空间，使用该空间用来存放合并后的序列；之后设定两个指针，最初位置分别为两个已经排序序列的起始位置；再比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置；重复上述步骤直到某一指针达到序列尾，将另一序列剩下的所有元素直接复制到合并序列尾，排序结束。

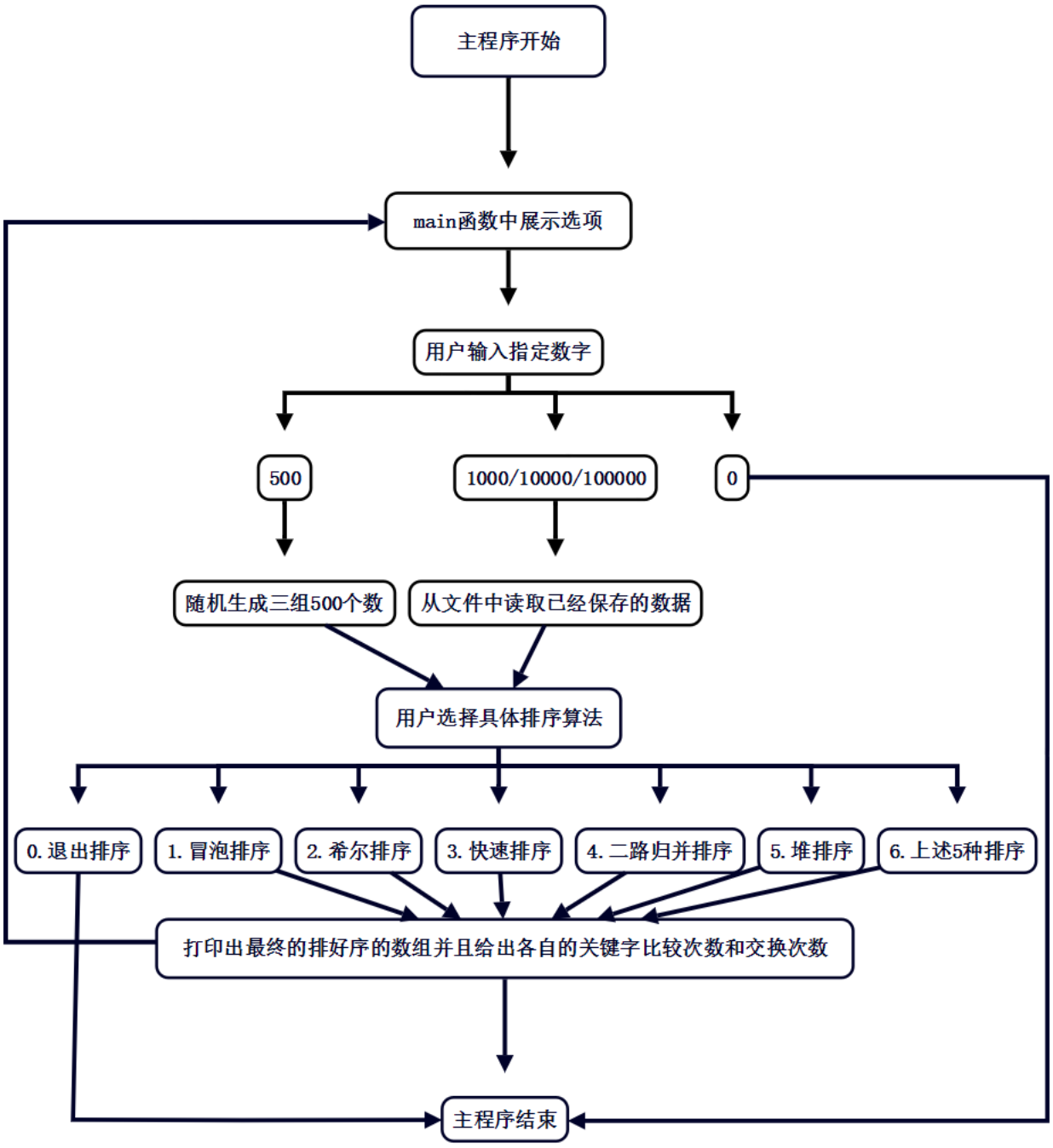
(7)堆排序

由堆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，即子结点的值或索引总是小于（或者大于）它的父节点。堆排序是一种利用堆的概念来排序的特殊的选择排序。分为两种方法：大顶堆：每个节点的值都大于或等于其子节点的值，在堆排序算法中用于升序排列；小顶堆：每个节点的值都小于或等于其子节点的值，在堆排序算法中用于降序排列；堆排序算法的主要思想是首先创建一个堆H[0……N-1]；之后把堆首（最大/小值）和堆尾互换；输出堆顶后再进行排序，并将堆的尺寸缩小1。重复如此，直到堆的尺寸为1，排序结束。

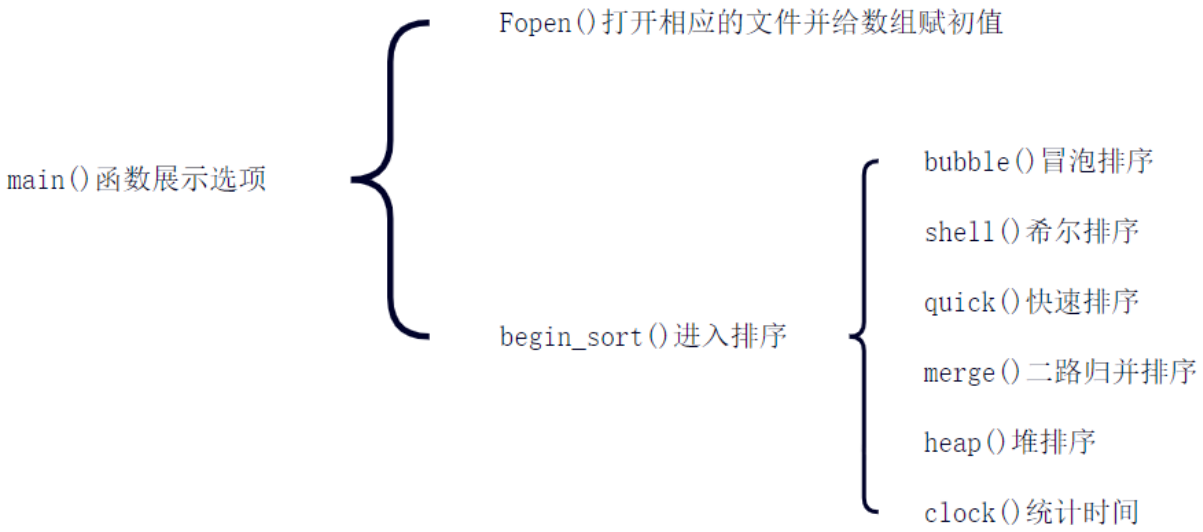
3.程序结构

主程序：展示界面，先选择需要排序的关键字个数->系统根据需求生成或读取相应的数据->选择排序的算法->展示排序的结果以及关键字的交换、比较次数->询问是否需要继续->程序结束。

主程序流程图(使用xmind绘制)：


主程序开始
main函数中展示选项
用户输入指定数字
500
1000/10000/100000
0
随机生成三组500个数
从文件中读取已经保存的数据
用户选择具体排序算法
0. 退出排序
1. 冒泡排序
2. 希尔排序
3. 快速排序
4. 二路归并排序
5. 堆排序
6. 上述5种排序
打印出最终的排好序的数组并且给出各自的关键字比较次数和交换次数
主程序结束

函数的调用关系：



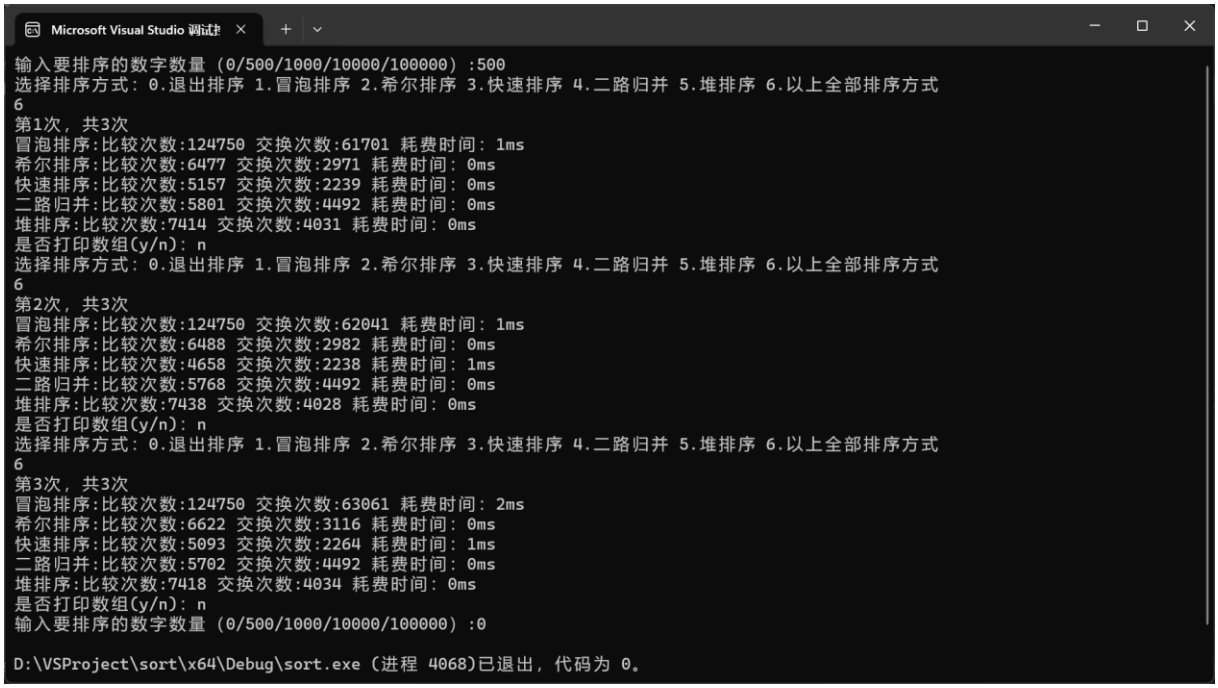
4. 实验结果与分析

4.1用户使用说明

a. 运行程序，可以看到一个选项界面
b. 输入要选择的数字和排序方式，系统会根据需求运行相应的代码
c. 每次排序结束，系统都会再次回到输入选择的界面，供用户继续排序
d. 一次排序结束后，不想继续进行排序，输入0程序自动结束

4.2测试结果

下图为三次对500个数据排序的输出结果：

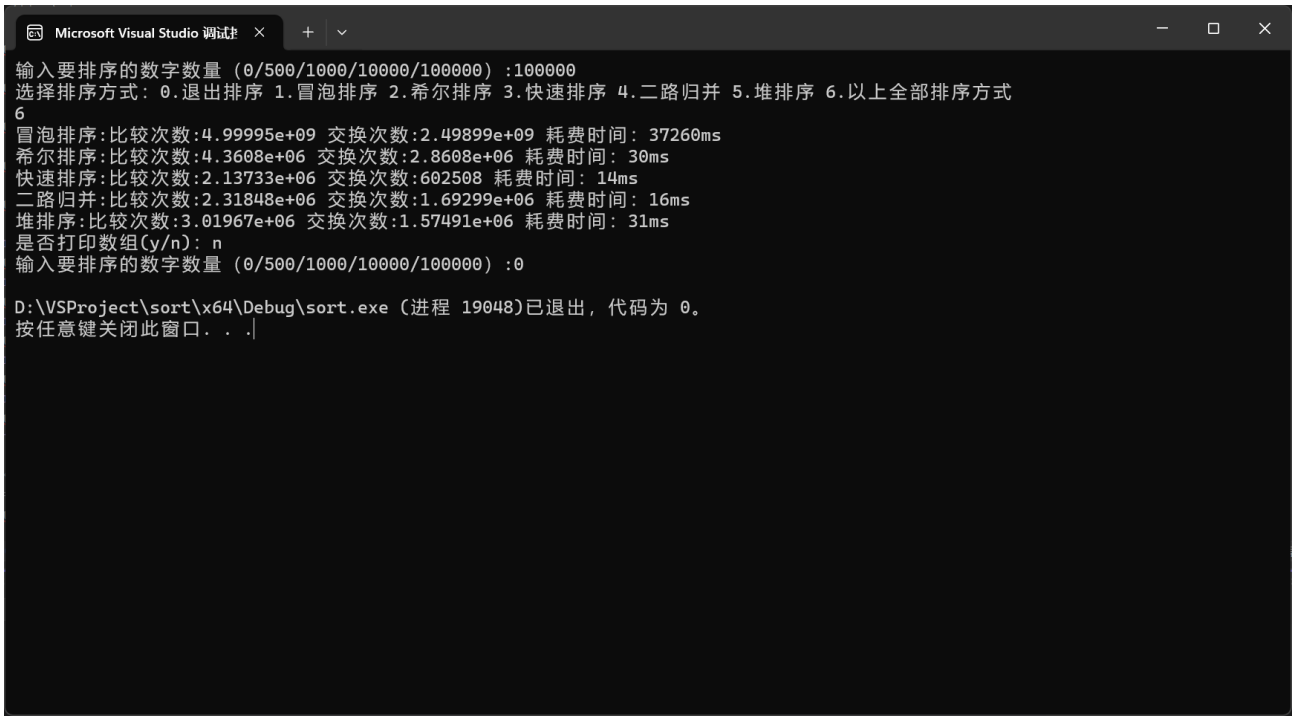


测试1000/10000/100000个数据：文件sort_1000.txt/sort_10000.txt/sort_10000.txt，文件中存储的数据均由rand()函数随机生成并存入。使用clock（）函数统计运行时间。

1000个数据的运行时间以及比较次数

10000个数据的运行时间以及比较次数

100000个数据的运行时间以及比较次数



4.3一些问题

vs2022中对数组变量要求较为严格，数组不允许任意定义大小，且在数组定义的规模过大时会弹出waring警告，不建议使用超大数组进行程序的运行。故采用了较为灵活的指针变量对待排序数据进行存取

在对比较次数进行计数时，如前文所提，冒泡排序中的比较次数随着数据的增长成指数级增长，单纯使用int类型存取在数据量为10000时已经超出范围，无法正常输出，后换用double可以解决问题，但当数据量再大时，需要采用数据的高精度加法才能输出正确的结果。

在收集运行时间时，一开始个人采用vs2022提供性能探查器统计运行时间，但性能探查器对部分机型不适用，无法做到程序的普及性。在网上翻阅相关资料后，了解到clock函数可以统计时间，遂改用clock()函数

4.4时间复杂度和空间复杂度分析

对于500个数据的排序中已经能够初步看出排序算法的速度的差别。冒泡算法对于给定数据的比较次数是恒定的，而且在500个数据时比较次数已经达到了10^6的数量级，交换次数也达到了10^5的数量级，是一种耗时较高的排序算法；其余的几个排序数量级都在10^4以下，都是排序速度比较快的算法。其中，快速排序对于给定的数据比较次数和交换次数相对来说更少。

比较次数和交换次数反映出了算法的时间复杂度，比较次数达到了10^6的数量级的冒泡排序时间复杂度为$O(N^2)$，时间复杂度远远大于其他几种时间复杂度为O(NlogN)的算法，导致其运行时间、比较次数超出其余算法几个数量级。之后的对1000/10000/100000个数据的排序时间更是无限放大了冒泡排序和其他几种排序的差距，在排序100000个数据时，冒泡排序用时高达42.09秒，而快速排序仅仅只用了55.65毫秒，差距显而易见。

4.4.1冒泡排序

冒泡排序在找关键字时要对关键字本身遍历一次，在之后比较时又要对关键字进行一次遍历，所以冒泡排序的时间复杂度为$O(N^2)$,是耗费时间比较长的一种排序算法。但冒泡排序在排序中只占用一个临时交换变量的空间，所以空间复杂只有O(1)。

4.4.2希尔排序

一般的插入排序在对几乎已经排好序的数据操作时的具有较高的效率（可以达到线性排序的效率），但因为插入排序一每次只能将数据移动一位，所以插入排序一般是低效的（时间复杂度达到了$O(N^2)$；希尔排序中，最好的情况下，只需要对关键字遍历一次，在对最大的步长进行一次遍历就可以实现排序，此时的时间复杂度为O(NlogN)，是一种比较快速的排序。但是最坏的情况下，要一直比较直至步长为1，此时的希尔排序就变成了插入排序，时间复杂度为$O(N^2)$,所以，希尔排序的时间复杂度与初始步长是紧密相连的，初始步长的合适选择可以大大降低希尔排序的时间复杂度。希尔排序在排序时只需要占用一个临时的交换变量的空间，时间复杂度也是O(1)。

4.4.3快速排序

快速排序最优的情况就是每一次取到的元素都刚好平分整个数组，此时的时间复杂度公式则为：T[N] = 2T[N/2] + f(N)（T[N/2]为平分后的子数组的时间复杂度，f[N] 为平分这个数组时所花的时间）；在最优的情况下快速排序时间复杂度的计算(用迭代法)：第一次递归->$T[N]=2T[N/2]+N$；第二次递归->令$N=N/2=2\{2T[N/4]+(N/2)\}+N$，$T[N]=2^2T[N/(2^2)]+2N$；第三次递归->令$N=N/(2^2)=2^{2\{2T[N/(2^3)]+N/(2^2)\}}+2N$，$T[N]=2^{3T[N/(2^3)]}+3N$；第m次递归(m次后结束)->令$N=N/(2^{(m-1)})=2^{mT[1]}+mN$；最后不能再平分时，已经完成迭代，得到：$T[N/(2^m)]=T[1]$->$N=2^m$->$m = \log N$；$T[N]=2^{mT[1]}+mN$；其中$m=\log N$;$T[N]=2^{\log_N T[1]}+N\log N=NT[1]+N\log N=N+N\log N$；其中N为元素个数，又因为当N>=2时：NlogN>=N 所以取NlogN；最差的情况就是每一次取到的元素就是数组中最小/最大的，这种情况其实就是冒泡排序，所以快速排序最差的情况下时间复杂度为$O(N^2)$。

因为快速排序是一种原地排序算法，它通过在原始数组上进行交换和划分操作来实现排

序，而不需要额外的空间来存储临时数据。在每一次递归调用中，快速排序只需要使用O(logN)的额外空间来保存递归调用的栈空间。

4.4.4归并排序

在归并排序中，将数列分为若干小数列要进行logN次，每一次比较又都是一个合并有序数列的过程，要进行N次，所以归并排序的时间复杂度为O(NlogN)。由于归并排序在一开始要申请一块大小为两个已经排序序列之和的空间，所以归并排序算法的空间复杂度为O(N)。

4.4.5堆排序

堆排序在排序时元素插入最后一位，再进行向上冒泡，即如果父节点的值小于被插入的元素，父节点下移，被插入的元素上移。这样一来时间复杂度取决于树的高度h。而完全二叉树的树的高度为[logN+1]的上取整，所以堆排序的平均时间复杂度为O(NlogN)。由于堆排序是对现有数据的直接排序，只占用一个临时交换变量的存储空间，所以空间复杂度为O(1)。

名称	算法稳定性	空间复杂度	时间复杂度	简要描述
冒泡排序	稳定	O(1)	O(N2)	从无序区中通过交换找出最大元素放到最后
希尔排序	不稳定	O(1)	O(NlogN)	按照给定的gap进行插入排序直至gap为1
快速排序	不稳定	O(1)	O(NlogN)	选取基准后大于基准的数字和小于基准的数字交换
归并排序	稳定	O(N)	O(NlogN)	数据的那段，从中一次选取小的加入新数组
堆排序	不稳定	O(1)	O(NlogN)	从堆顶把数据拿出去在排序，反复进行

表：5种算法总结

4.5算法稳定性分析

4.5.1冒泡排序

冒泡排序算法是通过比较相邻的元素并交换它们的位置来排序数组的算法。在每次遍历中，将最大(小)的元素冒泡到最后的位置。由于每次比较的是相邻元素，所以对于相同的元素，它们之间的相对顺序不会改变，故冒泡排序是一种稳定的算法

4.5.2希尔排序

单次插入排序是稳定的，不会改变相同元素的相对顺序，但由于多次插入排序，在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以希尔排序是不稳定的。

4.5.3快速排序

因为基准数字的随机选取，会导致小于基准数的数字与第一个大于基准的数字位置交换，每次遍历结束之后，头部的基准数字与最后一个小于基准的字交换位置，都会导致相同的数字出现在不同的位置，故快速排序算法是一种不稳定的算法。

4.5.4归并排序

归并排序的稳定性在于merge() 函数的具体内容。在合并的过程中，如果 a[0…m]和 a[m+1…r]之间有值相同的元素，那在转移元素时，可以像伪代码中那样，先把 a[p…q]中的元素放入 tmp 数组。这样就保证了值相同的元素，在合并前后的先后顺序不变，反之，值相同的元素在排序中的顺序就会被打乱。所以，只要先操作左半部分,再操作右半部分,归并排序就是稳定的。

4.5.5堆排序

堆排序算法是一种不稳定的算法，主要体现在两方面：在建堆阶段，首先将待排序的元素构建成一个堆（大顶堆或小顶堆）。从最后一个非叶子节点开始，逐步向上调整，使得每个节点都满足堆的性质。在调整的过程中，可能需要交换节点的位置来满足堆的性质。这个交换操作会改变相同键值元素的相对顺序，从而导致排序的不稳定性；在排序阶段，每次从堆顶取出最大（或最小）元素，将其放到已排序部分的末尾，然后对剩余未排序的部分进行堆调整，重复这个过程直到所有元素都被放置到正确的位置上。这个过程中同样会涉及交换操作，进一步破坏了相同键值元素的相对顺序，导致排序的不稳定性。

5.总结

学习数据结构是计算机科学和编程中至关重要的一环。数据结构是对数据的组织、管理和存储方式，而算法则是对数据进行操作的一种方法，二者有着紧密的联系。在完成课程设计报告的过程中，更加体会到了这层联系的重要性。在诸多算法当中，排序算法是最基本、最常用的一种。冒泡排序、希尔排序、快速排序、二路归并排序和堆排序是常见的排序算法，通过研究这些排序算法，我不仅对每种排序算法的原理和实现有了更深入的了解，还对数据结构有了更深入的理解。

在初步阶段，最麻烦的就是对合适的排序算法的选取，所选择的算法既要具有一般算法的普适性，又要是在个人理解能力范围之内。按照这个标准，从简单到复杂，我一次选取了冒泡排序，希尔排序，快速排序，归并排序和堆排序作为研究对象。一开始选择了链表作为数据的存取工具，但在实际操作时，链表需要频繁的分配空间，且对数据的操作如建立数据表、交换数据、正序输出数据时耗费时间较多，后续改为用指针作为存取数据的工具，大大缩短了程序的运行时间。尤其需要注意的是在研究冒泡排序时，100000个数据的比较次数已经远超过int的存取范围，改用double问题得到部分解决。

这些算法中，有的不仅仅是算法，还包含了其他的重要的常见方法。快速排序中用到了递归的方法，如果不运用递归的方法，快速排序的代码会变得非常的繁琐，而且对部分数据的排序效果不尽人意。同时，快速排序和归并排序中用到了分而治之的方法，在数据的处理中，分治法的应用是非常普遍的，二分搜索、最近点对等问题都是可以用分治法解决的经典问题。

对几种排序算法的研究使得我对算法的时间复杂度和空间复杂度对算法性能的影响有了更加深刻的认识，也了解了如何通过优化和改进算法来提高排序的效率。这些知识提升了我的编程能力，也让我认识到自身能力的有限，对于一些复杂的排序方法仍然束手无策。意识到了自己对计算机邻域的知识积累的微薄，以后要投入更多精力去学习理论知识。

6.源程序

文件名：sort.h、main.cpp、begin_sort.cpp、bubble.cpp、shell.cpp、quick.cpp、heap.cpp、merge.cpp

sort.h

```
#include"iostream"
#include"stdlib.h"
#include"time.h"
using namespace std;
```

```
//cnt1为比较次数,cnt2为交换次数
void begin_sort(int* b, int num, int flag);
void bubble(int* a, int num, double& cnt1, double& cnt2);
void heap_sort(int* array, int num, double& cnt1, double& cnt2);
void merge_sort(int* a, int num, double& cnt1, double& cnt2);
void quick(int* a, int left, int right, double& cnt1, double& cnt2);
void shell(int* a, int num, double& cnt1, double& cnt2);

main.cpp
#include"sort.h"
int main() {
	/*
	//将1000/10000/100000个数据写入文件备用
	for (int i = 0; i < num; i++) a[i] = rand();
	if (num == 1000) {
	fp = fopen("D:\sort_1000.txt", "w+");
	for (int i = 0; i < num; i++) fprintf(fp, "%d ", a[i]);  //将a[i]写入文件
	fclose(fp);
	}
	else if (num == 10000) {
	fp = fopen("D:\sort_10000.txt", "w+");
	for (int i = 0; i < num; i++) fprintf(fp, "%d ", a[i]);  //将a[i]写入文件
	fclose(fp);
	}
	else {
	fp = fopen("D:\sort_100000.txt", "w+");
	for (int i = 0; i < num; i++) fprintf(fp, "%d ", a[i]);  //将a[i]写入文件
	fclose(fp);
	}
	*/
	int num, flag; FILE* fp;
	cout << "输入要排序的数字数量（0/500/1000/10000/100000）:"; cin >> num;
	while (num) {
	cout << "选择排序方式：0.退出排序 1.冒泡排序 2.希尔排序 3.快速排序 4.二路
	归并 5.堆排序 6.以上全部排序方式" << endl; cin >> flag;
	int* b; b = (int*)malloc(sizeof(int) * num);  srand((unsigned
	int)time(NULL));   void Fopen(int, int, FILE * fp, int* b);
	if (flag) {
```

```
        if (num == 500) {                    //基础，三组随机的500个数据
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < num; j++)  b[j] = rand();
                cout << "第" << i + 1 << "次，共3次" << endl;
                if (i)     cout << "选择排序方式：0.退出排序 1.冒泡排序 2.希尔排序
      3.快速排序 4.二路归并 5.堆排序 6.以上全部排序方式" << endl, cin >> flag;
                begin_sort(b, num, flag);
            }
        }
        else if (num == 1000)
            fp = fopen("D:\sort_1000.txt", "r"), Fopen(num, flag, fp, b);
        else if (num == 10000)
            fp = fopen("D:\sort_10000.txt", "r"), Fopen(num, flag, fp, b);
        else if (num == 1000)
            fp = fopen("D:\sort_100000.txt", "r"), Fopen(num, flag, fp, b);
    }
    else exit(0);
    cout << "输入要排序的数字数量（0/500/1000/10000/100000）:"; cin >> num;
    }
    return 0;
}
void Fopen(int num, int flag, FILE* fp, int* b) {
    for (int i = 0; i < num; i++)
    fscanf_s(fp, "%d", &b[i]);        //读取文件内容到b[i]
    fclose(fp);
    begin_sort(b, num, flag);
}

begin_sort.cpp
#include"sort.h"
void begin_sort(int* b, int num, int flag) {
    double cnt1, cnt2; int* a; a = (int*)malloc(sizeof(int) * num); char c;
    double start, end;
    switch (flag) {
    case 1:for (int i = 0; i < num; i++)  a[i] = b[i]; start = clock();
    bubble(a, num, cnt1, cnt2); end = clock();
    cout << "冒泡排序:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗
    费时间：" << end - start << "ms";       break;
```

```
case 2:for (int i = 0; i < num; i++)  a[i] = b[i];start = clock();
shell(a, num, cnt1, cnt2); end = clock();
cout << "希尔排序:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗费时间：" << end - start << "ms";       break;
case 3:for (int i = 0; i < num; i++)  a[i] = b[i]; cnt1 = 0; cnt2 = 0;
start = clock(); quick(a, 0, num - 1, cnt1, cnt2); end = clock();
cout << "快速排序:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗费时间：" << end - start << "ms";       break;
case 4:for (int i = 0; i < num; i++)  a[i] = b[i]; cnt1 = 0; cnt2 = 0;
start = clock();   merge_sort(a, num, cnt1, cnt2); end = clock();
cout << "二路归并:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗费时间：" << end - start << "ms";       break;
case 5:for (int i = 0; i < num; i++)  a[i] = b[i]; cnt1 = 0; cnt2 = 0;
start = clock(); heap_sort(a, num, cnt1, cnt2); end = clock();
cout << "堆排序:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗费时间：" << end - start << "ms";     break;
case 6:
for (int i = 0; i < num; i++)  a[i] = b[i]; start = clock(); bubble(a, num, cnt1, cnt2); end = clock();
cout << "冒泡排序:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗费时间：" << end - start << "ms";
for (int i = 0; i < num; i++)  a[i] = b[i];start = clock(); shell(a, num, cnt1, cnt2); end = clock();
cout << endl << "希尔排序:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗费时间：" << end - start << "ms";
for(int i = 0; i < num; i++)  a[i] = b[i]; cnt1 = 0; cnt2 = 0;  start = clock(); quick(a, 0, num - 1, cnt1, cnt2); end = clock();
cout << endl << "快速排序:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗费时间：" << end - start << "ms";
for (int i = 0; i < num; i++)  a[i] = b[i]; cnt1 = 0; cnt2 = 0; start = clock();merge_sort(a, num, cnt1, cnt2); end = clock();
cout << endl << "二路归并:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗费时间：" << end - start << "ms";
for (int i = 0; i < num; i++)  a[i] = b[i]; cnt1 = 0; cnt2 = 0; start = clock(); heap_sort(a, num, cnt1, cnt2); end = clock();
cout << endl << "堆排序:" << "比较次数:" << cnt1 << " 交换次数:" << cnt2 << " 耗费时间：" << end - start << "ms";       break;
default:cout << "错误的数字"; break;
}
```

```
    cout << endl << "是否打印数组(y/n)："; cin >> c;
    if (c == 'y') {
    cout << endl; for (int i = 0; i < num; i++) cout << a[i] << " "; cout <<
    endl;
    }
}bubble.cpp
//时间复杂度：O(n^2) 最优的情况也就是开始就已经排序好序了，那么就可以不用交换元
    素了，由于外层循环为n,内层所需要循环比较的次数为（n-1）、（n-2）…1由等差
    数列求和得时间花销为：[ n(n-1) ] / 2；所以最优的情况时间复杂度为：
    O( n^2 )。
//空间复杂度：O(1) 辅助变量空间仅仅是一个临时变量，并且不会随着排序规模的扩大而
    进行改变，所以空间复杂度为O(1)
//稳定性：稳定
#include "sort.h"
void bubble(int* a, int num, double& cnt1, double& cnt2) {
    int i, j; cnt1 = 0; cnt2 = 0;
    for (i = 0; i < num - 1; i++) {
      for (j = 0; j < num - 1 - i; j++) {
        if (++cnt1 && a[j] > a[j + 1]) {
          a[j] = a[j + 1] + a[j];
          a[j + 1] = a[j] - a[j + 1];
          a[j] = a[j] - a[j + 1];
          cnt2++;
        }
      }
    }
}

shell.cpp
//时间复杂度：平均时间复杂度：O(Nlog2N) 最佳时间复杂度：O(nlog2n) 最差时间复杂
    度：O(N ^ 2)
//空间复杂度：O(1)
//稳定性：不稳定
//特殊的直接插入排序，只是比较宽度变为gap
#include"sort.h"
void shell(int* a, int num, double& cnt1, double& cnt2){
  int i, j, t, gap, temp; cnt1 = 0; cnt2 = 0;
  for (gap = num / 2; gap > 0; gap /= 2) {        // 步长初始化为数组长度的一
    半，每次遍历后步长减半,
```

```
        for (i = 0; i < gap; ++i) {                    // 变量 i 为每次分组的第一个
    元素下标
            for (j = i + gap; j < num; j += gap) {
                temp = a[j];
                t = j - gap;
                while (cnt1++ && t >= 0 && a[t] > temp) {
                    cnt2++;
                    a[t + gap] = a[t];
                    a[t] = temp;
                    t -= gap;
                }
            }
        }
    }
}
```

quick.cpp

```
//时间复杂度：0(nlog2n) 平均时间复杂度为：0(nlog2n)
//空间复杂度：0(n) 空间复杂度：logn 主要是由于递归造成的栈空间的使用，最好的情况
    下其树的深度为：log2(n) 空间复杂度为 0(logn)而最坏的情况下:需要n - 1次调
    用，每2个数都需要交换，此时退化为冒泡排序 空间复杂度为 0(n)
//稳定性：一般不稳定
#include"sort.h"
void quick(int* a, int left, int right, double& cnt1, double& cnt2){
    int i, j, temp;
    if (left > right) return;
    temp = a[left]; i = left; j = right;
    while (i != j) {
    while (a[j] >= temp && i < j) {
      j--;
      cnt1++;
    }
    while (a[i] <= temp && i < j) {
      i++;
      cnt1++;
    }
    if (i < j) {
      cnt2++;
```

```
            a[j] = a[i] + a[j];
            a[i] = a[j] - a[i];
            a[j] = a[j] - a[i];
        }
        }
        a[left] = a[i];                                    //基准数归位
        a[i] = temp;
        cnt2 += 2;
        cnt2++;
        quick(a, left, i - 1, cnt1, cnt2);
        quick(a, i + 1, right, cnt1, cnt2);
}

merge.cpp
//时间复杂度：0(nlog2n)
//空间复杂度：0(n)
//稳定性：稳定
//此处的cnt2包含了将原数组中的元素送入新数组中
#include"sort.h"
void merge(int* input, int* output, int start, int mid, int end, double& cnt1,
     double& cnt2) {
    int i = start, j = mid + 1, k = start;  // i第一块初始下标,j第二块初始下标,k
     充当合并块的下标
    // 判断遍历两个分块结束
    while ((i <= mid) && (j <= end)) { // 第一块第 i 个值比第二块第 j 值小
        if (++cnt1 && input[i] <= input[j]) {
            output[k++] = input[i++]; // 把小的值存入第二个数组，即第一块第 i 个
     值
            cnt2++;
        }
        else {
            output[k++] = input[j++]; // 小的值存入第二个数组，即第二块第 j 个值
            cnt2++;
        }
    }
    while (i <= mid) {    // 第一块没遍历完，而第二块遍历结束，说明第一块剩余值
     都大于第二块，直接把剩余第一块数据都存入第二个数组
        output[k++] = input[i++];
```

```
        cnt2++;
    }
    while (j <= end) {   // 第二块没遍历完，而第一块遍历结束，说明第二块剩余值都
     大于第一块，直接把剩余第二块数据都存入第二个数组
        output[k++] = input[j++];
        cnt2++;
    }
}
void merge_split(int* input, int* output, int gap, int num, double& cnt1,
     double& cnt2) {
    int i = 0;
    while (i + 2 * gap - 1 < num) {
        merge(input, output, i, i + gap - 1, i + 2 * gap - 1, cnt1, cnt2); // 归
     并分块排序
        i = i + 2 * gap;                                                    // 归
     并间隔，指向下一个分块的起始点
    }
    if ((i + gap - 1) < num - 1)
        merge(input, output, i, i + gap - 1, num - 1, cnt1, cnt2);

    else
        for (int j = i; j < num; j++)
            output[j] = input[j];
}
void merge_sort(int* a, int num, double& cnt1, double& cnt2) {
    int* b = (int*)malloc(sizeof(int) * num);
    int gap = 1;
    while (gap < num) {
        merge_split(a, b, gap, num, cnt1, cnt2); // 归并，结果在 b 中
        gap = 2 * gap;
        merge_split(b, a, gap, num, cnt1, cnt2); // 归并，结果在 a 中
        gap = 2 * gap;
    }
}
```

heap.cpp

```
//时间复杂度：O(n)
//空间复杂度：O(1)
```

```
//稳定性：不稳定
#include"sort.h"
void swap(int* a, int* b, double& cnt2) {
    int temp = *b; *b = *a;  *a = temp;
    cnt2++;
}
void heapify(int* array, int start, int end, double& cnt1, double& cnt2) {
     //建立父节点指标和子节点指标
    int dad = start;    int son = dad * 2 + 1;
    while (son <= end) {
     //若子节点指标在范围内才做比较
        if (++cnt1 && son + 1 <= end && array[son] < array[son + 1])
     //先比较两个子节点大小，选择最大的
            son++;
        if (++cnt1 && array[dad] > array[son])
     //如果父节点大于子节点代表调整完毕，直接跳出函数
            return;
        else {
     //否则交换父子内容再继续子节点和孙节点比较
            swap(&array[dad], &array[son], cnt2);
            dad = son;
            son = dad * 2 + 1;
        }
    }
}
void heap_sort(int* array, int num, double& cnt1, double& cnt2) {
    int i;
    for (i = num / 2 - 1; i >= 0; i--)
     //初始化，i从最后一个父节点开始调整
        heapify(array, i, num - 1, cnt1, cnt2);

    for (i = num - 1; i > 0; i--)
     {                                                  //先将第一个元素和已排好元素
     前一位做交换，再从新调整，直到排序完毕
        swap(&array[0], &array[i], cnt2);
        heapify(array, 0, i - 1, cnt1, cnt2);
    }
}
```