《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 计算机类 | 姓名 |  | 学号 |  |
| 实验布置日期 | | 2019.11.14 | | 提交  日期 | 2019.11.22 | | 成绩 |  |

课程实践实验6：排序算法的实现及性能测试及比较

## 一、问题描述及要求

在书中，各种内部排序算法的时间复杂度分析结果只给出了算法执行时间的阶，或大概执行时间。试通过具体数据比较各种算法的关键字比较次数和记录移动次数，以取得直观感受。

要求：

（1）编写程序创建一些整数文件用于排序。创建的这些文件可以根据需要生成不同的长度，如长度分别为20，200和2000，以正序、逆序、随机顺序的方式创建这些文件，通过把所有这些测试数据保存在文件中（而不是每次在测试程序时用随机数生成），可以使用同样的数据去测试不同的方法，因此会更易于比较这些方法的性能。

（2）数据表采用顺序存储结构，实现插入排序，选择排序，希尔排序，归并排序，快速排序，堆排序等排序，并对这些算法的实现效率进行比较和分析。

（3）排序的指标包含：运行时间，比较次数，移动次数。

## 二、概要设计

**（1）对实验内容的理解和二次概括。**

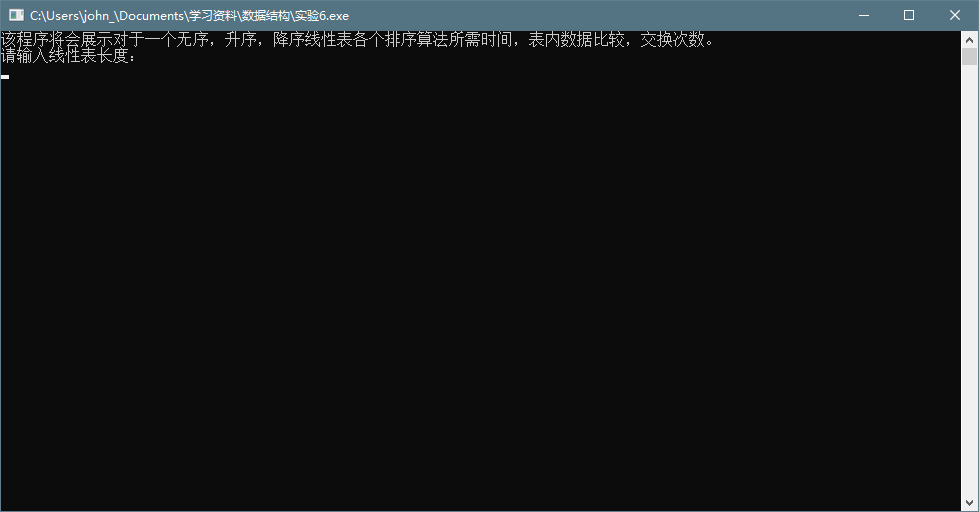
**该实验需要使用ctime库，构建随机数线性表，写出各个排序算法的函数，在各个排序算法中计算比较次数和移动次数，并且计算各个算法的时间。**

**（2）给出系统的功能列表**

**输入线性表长度n**

* **可生成一个长度为n的随机数表**
* **可输出各个排序算法的运行时间，比较次数，移动次数**

**（3）程序运行的界面设计**



**（4）程序结构设计，包括：对已有程序的使用，自己将设计哪些程序文件，各部分关系描述。**

将设计插入排序，归并排序，希尔排序，快速排序，堆排序，冒泡排序，选择排序算法，以及他们的性能检测程序。

## 三、详细设计

开始

生成随机数构成的顺序表

开始计时

输入顺序表元素个数

开始排序，并且计算排序时的比较次数和交换次数

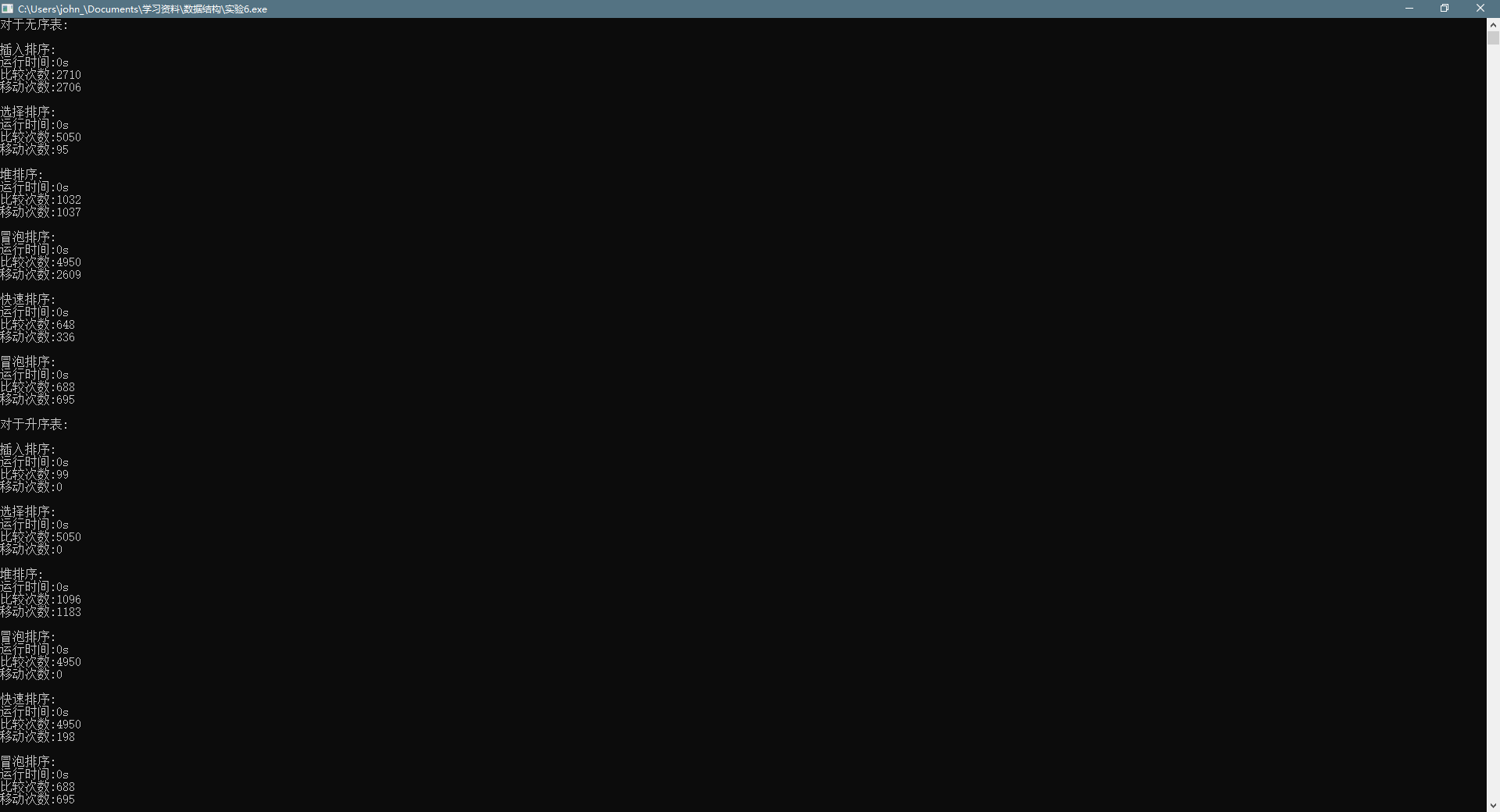
结束计时

输出结果

终止

## 四、实验结果

紧扣题目要求设计提供相应的测试方法和结果。可以给出具体的测试用例，每个测试用例一般可列出：

* 测试输入：100
* 测试目的：检查程序是否正常运作
* 正确输出：
* 实际输出：
* 
* 测试结论：通过

## 五、实验分析与探讨

测试结果的分析：结果较为正常，可以看出各个算法之间的性能差距还是比较大的，且在时间和比较次数交换次数上各有优劣。

**实验设计、实现过程中遇到的问题：当测试表数据过少时，会导致运行时间过小，显示为0.因此测试时应当至少采用1000.**

## 六、小结

本次实验完成了对各个排序算法的时间和操作性能的比较分析，内容较为简易，通过本实验可以看出各个排序算法性能之间的差异。唯一较为困难的地方是希尔排序算法由于本身算法的特性，无法在较大的数据量下操作，而数据量较小时无法体现时间上的差异。

## 附录：源代码

#include <iostream>

#include <random>

#include <fstream>

#include <sstream>

#include <algorithm>

#include <ctime>

#include <iomanip>

using namespace std;

bool cmp(int a, int b)

{

return a > b;

}

void getList(int n)

{

int list[10000];

for (int i = 0; i < n; i++)

{

list[i] = rand() % 10000;

}

ofstream fout1("unordered\_list.txt");

for (int i = 0; i < n; i++)

{

stringstream ss;

string temp;

ss << list[i];

ss >> temp;

fout1 << temp << endl;

}

sort(list, list + n);

ofstream fout2("ascending\_list.txt");

for (int i = 0; i < n; i++)

{

stringstream ss;

string temp;

ss << list[i];

ss >> temp;

fout2 << temp << endl;

}

sort(list, list + n, cmp);

ofstream fout3("descending\_list.txt");

for (int i = 0; i < n; i++)

{

stringstream ss;

string temp;

ss << list[i];

ss >> temp;

fout3 << temp << endl;

}

}

void print(int a[],int n)

{

for (int i = 0; i < n; i++)

{

cout << a[i] << " ";

}

cout << endl;

}

void InsertSort(string fname, int n)

{

ifstream fin(fname);

int a[10000];

while (fin.eof() == false)

{

for (int i = 0; i < n; i++)

{

fin >> a[i];

}

}

int count\_cmp = 0, count\_move = 0;

clock\_t start\_t, end\_t;

double total\_t;

start\_t = clock();

for (int i = 1; i<n; i++)

{

if (a[i] < a[i - 1])

{ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入

count\_cmp++;

int j = i - 1;

int x = a[i]; //复制为哨兵，即存储待排序元素

a[i] = a[i - 1]; //先后移一个元素

count\_move++;

while (x < a[j])

{ //查找在有序表的插入位置

count\_cmp++;

a[j + 1] = a[j];

count\_move++;

j--; //元素后移

}

a[j + 1] = x; //插入到正确位置

}

else count\_cmp++;

}

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

cout << "插入排序:" << endl;

cout << "运行时间:" << total\_t << "s" << endl;

cout << "比较次数:" << count\_cmp << endl << "移动次数:" << count\_move << endl;

cout << endl;

}

void ShellInsertSort(int a[], int n, int dk, int &count\_cmp, int &count\_move)

{

for (int i = dk; i<n; ++i)

{

count\_cmp++;

if (a[i] < a[i - dk])

{ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入

int j = i - dk;

int x = a[i]; //复制为哨兵，即存储待排序元素

a[i] = a[i - dk]; //首先后移一个元素

count\_move++;

while (x < a[j])

{ //查找在有序表的插入位置

count\_cmp++;

a[j + dk] = a[j];

count\_move++;

j -= dk; //元素后移

}

a[j + dk] = x; //插入到正确位置

}

}

}

//先按增量d（n/2,n为要排序数的个数进行希尔排序

void ShellSort(string fname, int n)

{

ifstream fin(fname);

int a[10000];

while (fin.eof() == false)

{

for (int i = 0; i < n; i++)

{

fin >> a[i];

}

}

int count\_cmp = 0, count\_move = 0;

clock\_t start\_t, end\_t;

double total\_t;

start\_t = clock();

int dk = n / 2;

while (dk >= 1)

{

ShellInsertSort(a, n, dk, count\_cmp, count\_move);

dk = dk / 2;

}

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

cout << "希尔排序:" << endl;

cout << "运行时间:" << total\_t << "s" << endl;

cout << "比较次数:" << count\_cmp << endl << "移动次数:" << count\_move << endl;

cout << endl;

}

int SelectMinKey(int a[], int n, int i, int &count\_cmp, int &count\_move)

{

int k = i;

for (int j = i + 1; j< n; ++j)

{

if (a[k] > a[j]) k = j;

count\_cmp++;

}

return k;

}

void SelectSort(string fname, int n)

{

ifstream fin(fname);

int a[10000];

while (fin.eof() == false)

{

for (int i = 0; i < n; i++)

{

fin >> a[i];

}

}

int count\_cmp = 0, count\_move = 0;

clock\_t start\_t, end\_t;

double total\_t;

start\_t = clock();

int key, tmp;

for (int i = 0; i< n; ++i)

{

key = SelectMinKey(a, n, i, count\_cmp, count\_move); //选择最小的元素

count\_cmp++;

if (key != i)

{

tmp = a[i]; a[i] = a[key]; a[key] = tmp;

count\_move++;//最小元素与第i位置元素互换

}

}

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

cout << "选择排序:" << endl;

cout << "运行时间:" << total\_t << "s" << endl;

cout << "比较次数:" << count\_cmp << endl << "移动次数:" << count\_move << endl;

cout << endl;

}

void HeapAdjust(int H[], int s, int length, int &count\_cmp, int &count\_move)

{

int tmp = H[s];

int child = 2 \* s + 1; //左孩子结点的位置。(i+1 为当前调整结点的右孩子结点的位置)

while (child < length)

{

count\_cmp++;

if (child + 1 <length && H[child]<H[child + 1])

{

// 如果右孩子大于左孩子(找到比当前待调整结点大的孩子结点)

++child;

}

count\_cmp++;

if (H[s]<H[child])

{

// 如果较大的子结点大于父结点

count\_move++;

H[s] = H[child]; // 那么把较大的子结点往上移动，替换它的父结点

s = child; // 重新设置s ,即待调整的下一个结点的位置

child = 2 \* s + 1;

}

else

{ // 如果当前待调整结点大于它的左右孩子，则不需要调整，直接退出

break;

}

H[s] = tmp; // 当前待调整的结点放到比其大的孩子结点位置上

count\_move++;

}

}

void BuildingHeap(int H[], int length, int &count\_cmp, int &count\_move)

{

//最后一个有孩子的节点的位置 i= (length -1) / 2

for (int i = (length - 1) / 2; i >= 0; --i)

HeapAdjust(H, i, length, count\_cmp, count\_move);

}

void HeapSort(string fname, int length)

{

ifstream fin(fname);

int H[10000];

while (fin.eof() == false)

{

for (int i = 0; i < length; i++)

{

fin >> H[i];

}

}

int count\_cmp = 0, count\_move = 0;

clock\_t start\_t, end\_t;

double total\_t;

start\_t = clock();

//初始堆

BuildingHeap(H, length, count\_cmp, count\_move);//从最后一个元素开始对序列进行调整

for (int i = length - 1; i > 0; --i)

{

//交换堆顶元素H[0]和堆中最后一个元素

int temp = H[i]; H[i] = H[0]; H[0] = temp;

count\_move++;//每次交换堆顶元素和堆中最后一个元素之后，都要对堆进行调整

HeapAdjust(H, 0, i, count\_cmp, count\_move);

}

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

cout << "堆排序:" << endl;

cout << "运行时间:" << total\_t << "s" << endl;

cout << "比较次数:" << count\_cmp << endl << "移动次数:" << count\_move << endl;

cout << endl;

}

void BubbleSort(string fname, int n)

{

ifstream fin(fname);

int a[10000];

while (fin.eof() == false)

{

for (int i = 0; i < n; i++)

{

fin >> a[i];

}

}

int count\_cmp = 0, count\_move = 0;

clock\_t start\_t, end\_t;

double total\_t;

start\_t = clock();

for (int i = 0; i< n - 1; ++i)

{

for (int j = 0; j < n - i - 1; ++j)

{

count\_cmp++;

if (a[j] > a[j + 1])

{

int tmp = a[j];

a[j] = a[j + 1];

a[j + 1] = tmp;

count\_move++;

}

}

}

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

cout << "冒泡排序:" << endl;

cout << "运行时间:" << total\_t << "s" << endl;

cout << "比较次数:" << count\_cmp << endl << "移动次数:" << count\_move << endl;

cout << endl;

}

void swap(int \*a, int \*b)

{

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

int partition(int a[], int low, int high, int &cmp, int &move)

{

int privotKey = a[low]; //基准元素

while (low < high)

{ //从表的两端交替地向中间扫描

while (low < high && a[high] >= privotKey)

{

--high; //从high 所指位置向前搜索，至多到low+1 位置。将比基准元素小的交换到低端

cmp++;

}

swap(&a[low], &a[high]);

move++;

while (low < high && a[low] <= privotKey)

{

++low;

cmp++;

}

swap(&a[low], &a[high]);

move++;

}

return low;

}

void QuickSort(int a[], int low, int high, int &cmp, int &move)

{

if (low < high)

{

int privotLoc = partition(a, low, high, cmp, move); //将表一分为二

QuickSort(a, low, privotLoc - 1, cmp, move); //递归对低子表递归排序

QuickSort(a, privotLoc + 1, high, cmp, move); //递归对高子表递归排序

}

}

void QuickSortPackage(string fname, int n)

{

ifstream fin(fname);

int a[10000];

while (fin.eof() == false)

{

for (int i = 0; i < n; i++)

{

fin >> a[i];

}

}

int count\_cmp = 0, count\_move = 0;

clock\_t start\_t, end\_t;

double total\_t;

start\_t = clock();

QuickSort(a, 0, n - 1, count\_cmp, count\_move);

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

cout << "快速排序:" << endl;

cout << "运行时间:" << total\_t << "s" << endl;

cout << "比较次数:" << count\_cmp << endl << "移动次数:" << count\_move << endl;

cout << endl;

}

//将r[i…m]和r[m +1 …n]归并到辅助数组rf[i…n]

void Merge(int \*r, int \*rf, int i, int m, int n, int &count\_cmp, int &count\_move)

{

int j, k;

for (j = m + 1, k = i; i <= m && j <= n; ++k)

{

count\_cmp++; count\_move++;

if (r[j] < r[i]) rf[k] = r[j++];

else rf[k] = r[i++];

}

while (i <= m)

{

rf[k++] = r[i++];

count\_cmp++; count\_move++;

}

while (j <= n)

{

rf[k++] = r[j++];

count\_cmp++; count\_move++;

}

}

void MergeSort(int \*r, int \*rf, int lenght)

{

int count\_cmp = 0, count\_move = 0;

clock\_t start\_t, end\_t;

double total\_t;

start\_t = clock();

int len = 1;

int \*q = r;

int \*tmp;

while

(len < lenght)

{

int s = len;

len = 2 \* s;

int i = 0;

while (i + len <lenght)

{

Merge(q, rf, i, i + s - 1, i + len - 1, count\_cmp, count\_move); //对等长的两个子表合并

i = i + len;

}

if (i + s < lenght)

{

Merge(q, rf, i, i + s - 1, lenght - 1, count\_cmp, count\_move); //对不等长的两个子表合并

}

tmp = q; q = rf; rf = tmp;

count\_move++;//交换q,rf，以保证下一趟归并时，仍从q 归并到rf

}

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

cout << "冒泡排序:" << endl;

cout << "运行时间:" << total\_t << "s" << endl;

cout << "比较次数:" << count\_cmp << endl << "移动次数:" << count\_move << endl;

cout << endl;

}

int main()

{

int n;

cout<<"该程序将会展示对于一个无序，升序，降序线性表各个排序算法所需时间，表内数据比较，交换次数。\n请输入线性表长度：(不多于10000)\n";

cin >> n;

getList(n);

cout << "对于无序表:" << endl;

cout << endl;

InsertSort("unordered\_list.txt", n);

//ShellSort("unordered\_list.txt",n);

SelectSort("unordered\_list.txt", n);

HeapSort("unordered\_list.txt", n);

BubbleSort("unordered\_list.txt", n);

QuickSortPackage("unordered\_list.txt", n);

ifstream fin1("unordered\_list.txt");

int a[10000], b[10000];

while (fin1.eof() == false)

{

for (int i = 0; i < n; i++)

{

fin1 >> a[i];

}

}

MergeSort(a, b, n);

cout << "对于升序表:" << endl;

cout << endl;

InsertSort("ascending\_list.txt", n);

//ShellSort("ascending\_list.txt",n);

SelectSort("ascending\_list.txt", n);

HeapSort("ascending\_list.txt", n);

BubbleSort("ascending\_list.txt", n);

QuickSortPackage("ascending\_list.txt", n);

ifstream fin2("ascending\_list.txt");

int c[10000], d[10000];

while (fin2.eof() == false)

{

for (int i = 0; i < n; i++)

{

fin2 >> c[i];

}

}

MergeSort(c, d, n);

cout << "对于降序表:" << endl;

cout << endl;

InsertSort("descending\_list.txt", n);

//ShellSort("descending\_list.txt",n);

SelectSort("descending\_list.txt", n);

HeapSort("descending\_list.txt", n);

BubbleSort("descending\_list.txt", n);

QuickSortPackage("descending\_list.txt", n);

ifstream fin3("descending\_list.txt");

int e[10000], f[10000];

while (fin3.eof() == false)

{

for (int i = 0; i < n; i++)

{

fin3 >> e[i];

}

}

MergeSort(e, f, n);

system("pause");

}