项目说明文档

数据结构课程设计

——八种排序方法比较

作 者 姓 名： 邓泉

学 号： 1953871

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](#_Toc533603601)

[1.1 题目背景](#_Toc533603602)

[1.2 功能分析](#_Toc533603603)

[2 设计](#_Toc533603604)

[2.1 数据结构设计](#_Toc533603605)

[3 八种排序算法概述与实现](#_Toc533603606)

[3.1 冒泡排序](#_Toc533603607)

[3.2 选择排序](#_Toc533603608)

[3.3 直接插入排序](#_Toc533603609)

[3.4 希尔排序](#_Toc533603610)

[3.5 快速排序](#_Toc533603611)

[3.6 堆排序](#_Toc533603612)

[3.7 归并排序](#_Toc533603613)

[3.8 基数排序](#_Toc533603614)

[4 程序实现](#_Toc533603615)

[4.1总流程实现：](#_Toc533603616)

[4.2 程序运行截图](#_Toc533603617)

[5 实验数据记录](#_Toc533603618)

[6 实验结果分析](#_Toc533603619)

# 

# 1分析

## 1.1 题目背景

排序动作在现实生活中随处可见。在数据大爆炸的今天，数据的数量级超乎想象，不同的排序算法在对这些数据进行排序时，排序速度千差万别，甚至可能直接影响到能否成功排序。因此，了解各种排序算法的原理，对各种算法的适用范围、稳定性等排序特点加以了解，有助于在日后选择合适的排序算法进行排序。

## 1.2 功能分析

本程序要求根据用户要求，产生指定容量的随机数序列，并通过不同的算法对产生的随机数序列进行排序，同时记录排序过程中的元素交换/比较次数和算法消耗时间，从而对算法的优劣给出直观的结果。

# 2 设计

## 2.1 数据结构设计

本程序使用线性表存储排序前和排序后的数据。

# 3 八种排序算法概述与实现

## 3.1 冒泡排序

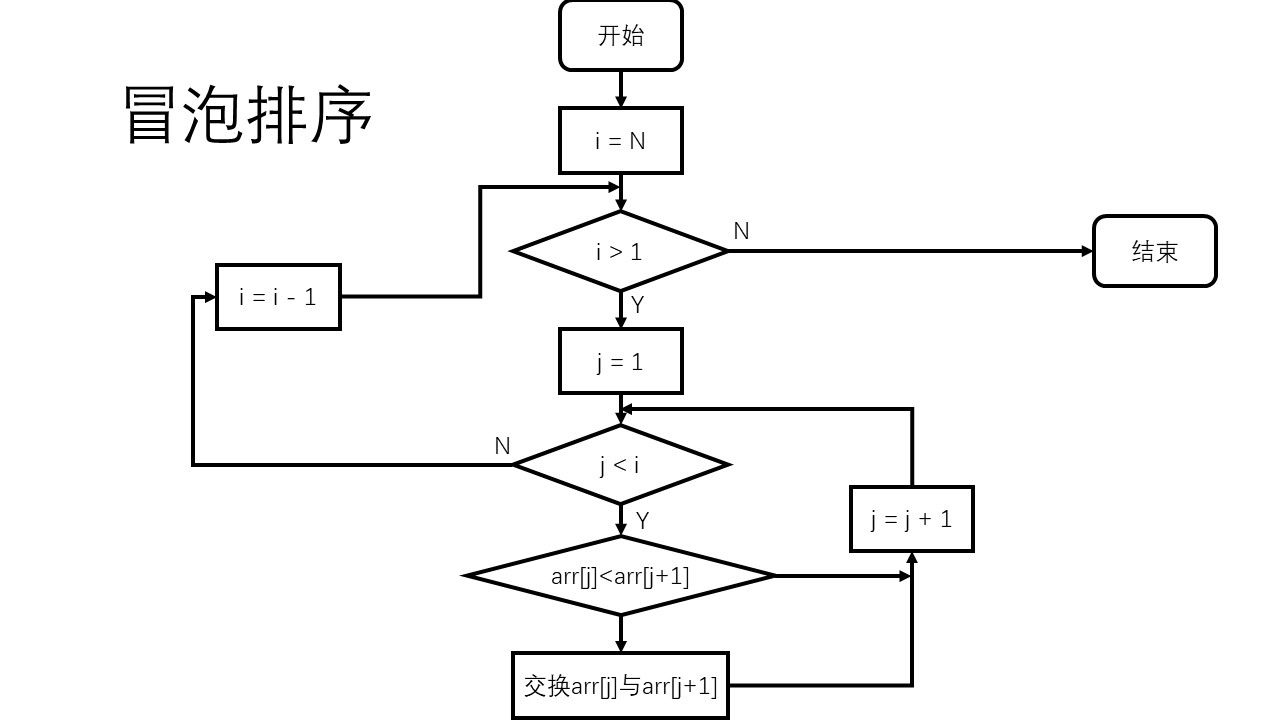
冒泡排序算法的原理如下：

1.比较相邻的元素。如果第一个比第二个大，就交换他们两个。

2.对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。

3.针对所有的元素重复以上的步骤，除了最后一个。

4.持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。



代码实现：

//冒泡排序

void DataList::BubbleSort()

{

clock\_t start, end;

start = clock();

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

{

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

{

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

{

int temp = Random[j];

Random[j] = Random[j + 1];

Random[j + 1] = temp;

swapnum++;

}

}

}

//OutPutRandom();

end = clock();

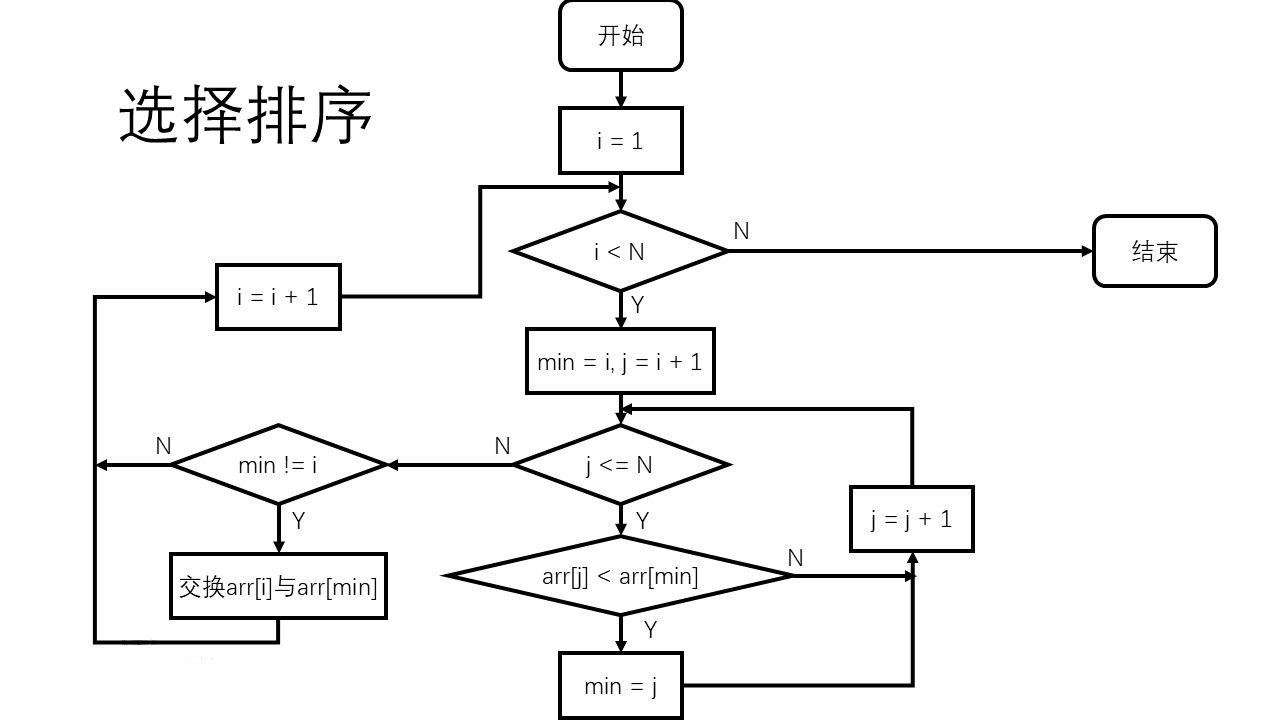
swaptimes = ((long double)end - (long double)start) / CLOCKS\_PER\_SEC;

};

## 3.2 选择排序

选择排序是一种简单直观的排序算法。它的工作原理是每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到全部待排序的数据元素排完。

选择排序是不稳定的排序算法。



代码实现：

//选择排序

void DataList::SelectSort(long long int s, long long int t, long long int gap)

{

clock\_t start, end;

start = clock();

for (long long int i = s; i < t - gap; i = i + gap)

{

int k = i;

for (long long int j = i; j < t; j = j + gap)

{

if (Random[j] < Random[k])

k = j;

}

int temp = Random[k];

Random[k] = Random[i];

Random[i] = temp;

swapnum++;

}

end = clock();

swaptimes = ((long double)end - (long double)start) / CLOCKS\_PER\_SEC;

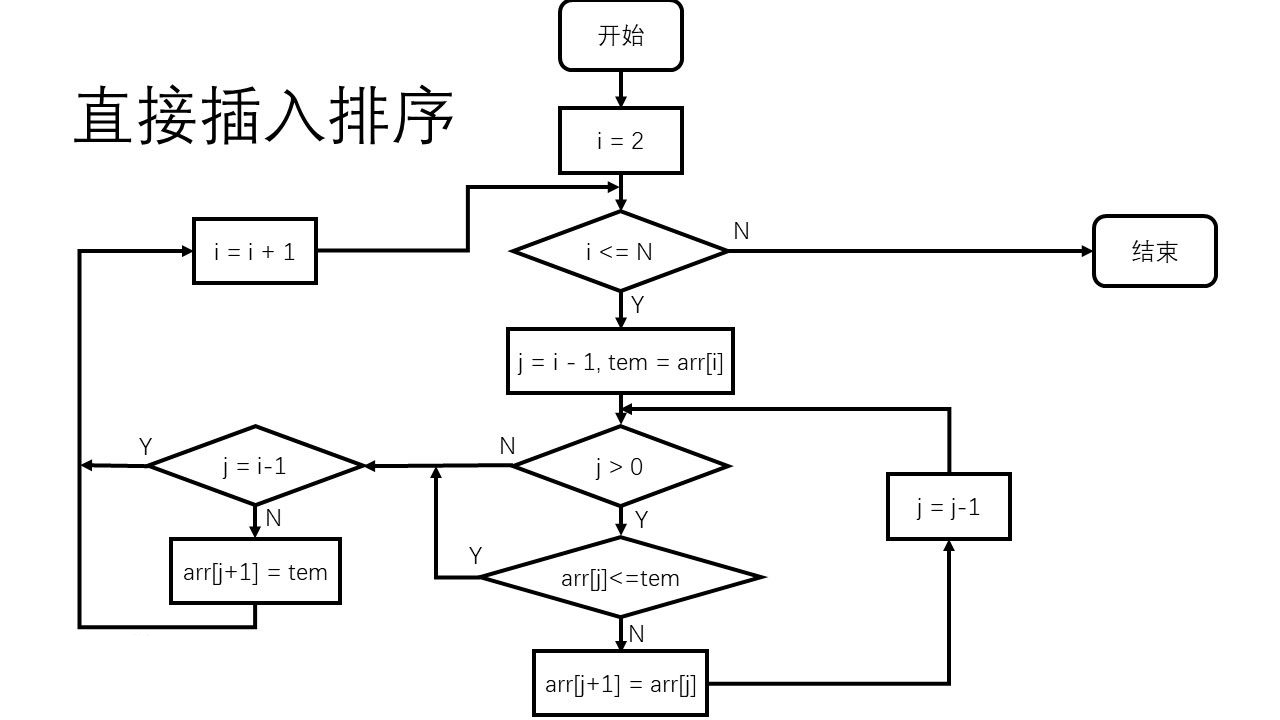
//OutPutRandom();

}

## 3.3 直接插入排序

直接插入排序是一种简单的排序方法，其基本操作是将一条记录插入到已排好的有序表中，从而得到一个新的、记录数量增1的有序表。

直接插入排序是一种稳定的排序算法



代码实现：

//直接插入排序

void DataList::InsertionSort(long long int s, long long int t, long long int gap)

{

clock\_t start, end;

start = clock();

for (int j = s + gap; j < t; j = j + gap)

{

for (int i = j; i > s; i = i - gap)

{

if (Random[i - gap] > Random[i])

{

int temp = Random[i - gap];

Random[i - gap] = Random[i];

Random[i] = temp;

swapnum++;

}

else break;

}

}

end = clock();

swaptimes = ((long double)end - (long double)start) / CLOCKS\_PER\_SEC;

//OutPutRandom();

}

## 3.4 希尔排序

希尔排序是插入排序的一种又称“缩小增量排序”，是直接插入排序算法的一种更高效的改进版本。希尔排序是非稳定排序算法。该方法因D.L.Shell于1959年提出而得名。

希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

希尔排序是一种不稳定的排序算法。

代码实现：

//希尔排序

void DataList::ShellSort()

{

clock\_t start, end;

start = clock();

long long int gap = num / 2;

long long int temp1 = 0;

while (gap && temp1 < 2)

{

clock\_t start, end;

start = clock();

long long int kk = swapnum;

for (long long int k = 0; k < gap; k++)

{

long long int m = k;

while (m < num)

{

m = m + gap;

}//找到相应的序列的最后一个数

//cout << j;

InsertionSort(k, m, gap);

}

end = clock();

swaptimes = ((long double)end - (long double)start) / CLOCKS\_PER\_SEC;

gap = (gap / 3) + 1;

if (gap == 1)

{

temp1++;

}

}

end = clock();

swaptimes = ((long double)end - (long double)start) / CLOCKS\_PER\_SEC;

//OutPutRandom();

}

## 3.5 快速排序

快速排序是对冒泡排序的一种改进。

快速排序由C. A. R. Hoare在1962年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

快速排序是一种不稳定的排序算法

代码实现：

//快速排序

void DataList::PartitionSort(long long int s, long long int t)

{

clock\_t start, end;

start = clock();

if (s < t)

{

long long int priotpos = s;

long long int priot = Random[s];

for (long long int i = s + 1; i < t; i++)

{

if (Random[i] < priot)

{

priotpos++;

if (priotpos != i)

{

int temp = Random[priotpos];

Random[priotpos] = Random[i];

Random[i] = temp;

swapnum++;

}

}

}

int temp = Random[priotpos];

Random[priotpos] = Random[s];

Random[s] = temp;

swapnum++;

PartitionSort(s, priotpos);

PartitionSort(priotpos + 1, t);

}

end = clock();

swaptimes = ((long double)end - (long double)start) / CLOCKS\_PER\_SEC;

}

## 3.6 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。堆是一个近似完全二叉树的结构，并同时满足堆的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

堆排序是一种不稳定的排序算法。

代码实现：

void DataList::Changeheap(long long startnum, long long int endnum)

{

for (long long int i = startnum; i <= (endnum + 8) / 2; i++)

{

long long int child = 2 \* i + 1;

long long int current = i;

long long int maxptr = i;

long long int signal = Random[i];

while (child < endnum) {

if (child + 1 < endnum && Random[child] < Random[child + 1])child = child + 1;

if (Random[current] >= Random[child]) break;

if (signal < Random[child])

{

Random[current] = Random[child];

current = child;

child = child \* 2 + 1;//逐位向前移动

}

}

Random[current] = signal;

}

/\*long long int tempp = Random[endnum - 1];

Random[endnum - 1] = Random[0];

Random[0] = tempp;

swapnum++;\*/

}

//堆排序

void DataList::HeapSort()

{

clock\_t start, end;

start = clock();

for (long long int i = (num - 2) / 2; i >= 0; i--)

{

Changeheap(i, num);//调整为初始堆

}

for (long long int i = num; i > 0; i--)

{

long long int changenumber = Random[0];

Random[0] = Random[i - 1];

Random[i - 1] = changenumber;

Changeheap(0, i - 1);

swapnum++;

}

//OutPutRandom();

end = clock();

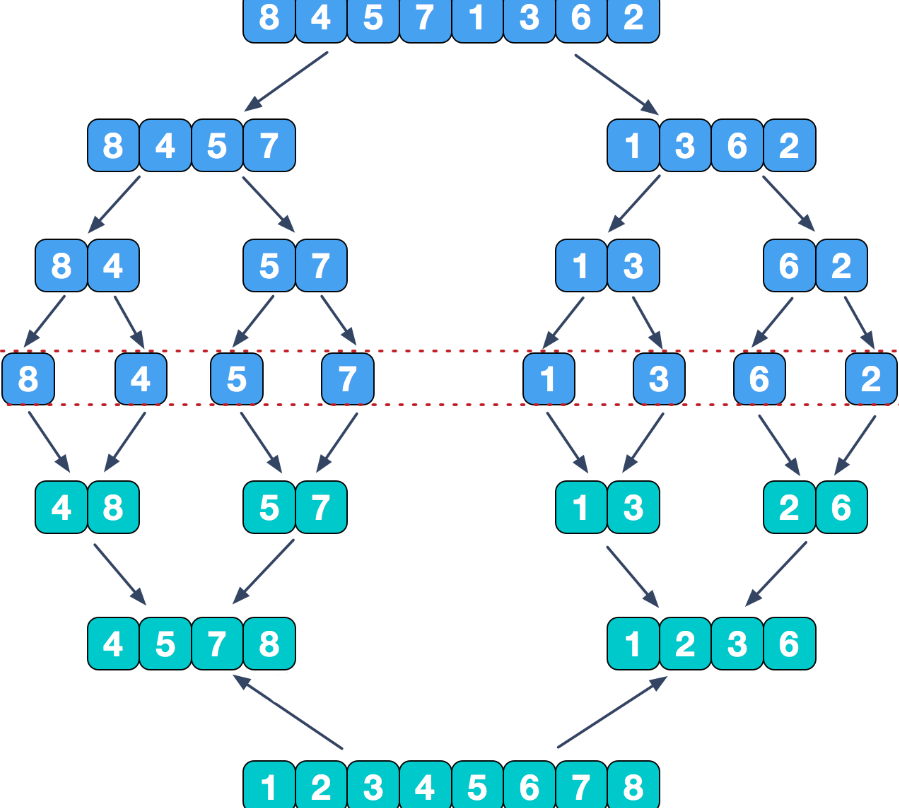
swaptimes = ((long double)end - (long double)start) / CLOCKS\_PER\_SEC;

};

## 3.7 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（Divide and Conquer）的一个非常典型的应用。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，称为二路归并。

归并排序是一种稳定的排序算法。



代码实现：

//归并排序

void DataList::MergeSort()

{

clock\_t start, end;

start = clock();

long long int len = 1;

while (len < num)

{

MergePass(len);

temp = 1 - temp;

len \*= 2;

}

end = clock();

swaptimes = ((long double)end - (long double)start) / CLOCKS\_PER\_SEC;

/\*OutPutRandom();

cout << "\n";

OutPutRandomm();\*/

}

void DataList::Merge(long long int s, long long int m/\*第一个序列的最后一个元素的位置\*/, long long int t/\*第二个序列的最后一个元素的位置\*/)

{

long long int i = s;

long long int k = s;

long long int j = m + 1;

while (i <= m && j <= t)

{

if (temp == 0)

{

if (Random[i] <= Random[j])

{

swapnum++;

Randomm[k] = Random[i];

i++; k++;

}

if (Random[i] > Random[j])

{

swapnum++;

Randomm[k] = Random[j];

j++; k++;

}

}

if (temp == 1)

{

if (Randomm[i] <= Randomm[j])

{

swapnum++;

Random[k] = Randomm[i];

i++; k++;

}

if (Randomm[i] > Randomm[j])

{

swapnum++;

Random[k] = Randomm[j];

j++; k++;

}

}

}

if (i <= m)

{

if (temp == 0)

{

for (long long int n = i; n <= m; n++)

{

Randomm[k] = Random[n];

k++;

}

}

if (temp == 1)

{

for (long long int n = i; n <= m; n++)

{

Random[k] = Randomm[n];

k++;

}

}

}

else {

if (temp == 0)

{

for (long long int n = j; n <= t; n++)

{

Randomm[k] = Random[n];

k++;

}

}

if (temp == 1)

{

for (long long int n = j; n <= t; n++)

{

Random[k] = Randomm[n];

k++;

}

}

}

};

## 

## void DataList::MergePass(long long int length)

{

long long int i = 0;

//OutPutRandomm();

while (i + 2 \* length < num)

{

Merge(i, i + length - 1, i + 2 \* length - 1);

i = i + 2 \* length;

}

if (i + length < num)Merge(i, i + length - 1, num - 1);

else {

if (temp == 0)

{

for (long long int j = i; j <= num; j++)

Randomm[j] = Random[j];

}

if (temp == 1)

{

for (long long int j = i; j <= num; j++)

Random[j] = Randomm[j];

}

}

// OutPutRandomm();

};

## 3.8 基数排序

基数排序属于“分配式排序”（distribution sort），又称“桶排序”（bucket sort），顾名思义，它是透过键值的部份资讯，将要排序的元素分配至某些“桶”中，藉以达到排序的作用。

基数排序法是属于稳定性的排序，其时间复杂度为O (nlog(r)m)，其中r为所采取的基数，而m为堆数，在某些时候，基数排序法的效率高于其它的稳定性排序法。

代码实现：

//基数排序

void DataList::radix\_sort(int arr[])

{

int getPlaces(int num);

int getMax(int arr[], int n);

void radix\_help(int arr[], int n, int place);

int temp, \* test = 0, max, maxPlaces;

//int stime, etime;

int testNum = 0;

memcpy(test, arr, sizeof(int) \* testNum);

max = getMax(test, testNum);

maxPlaces = getPlaces(max);

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

radix\_help(test, testNum, i);

}

int getPlaces(int num) {

int count = 1;

int temp = num / 10;

while (temp != 0) {

count++;

temp /= 10;

}

return count;

}

int getMax(int arr[], int n) {

int max = 0;

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

if (arr[i] > max)

max = arr[i];

}

return max;

}

void radix\_help(int arr[], int n, int place) {

int buckets[10][100] = { NULL };

int temp = (int)pow(10, place - 1);

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

int row = (arr[i] / temp) % 10;

for (int j = 0; j < 100; j++) {

if (buckets[row][j] == NULL) {

buckets[row][j] = arr[i];

break;

}

}

}

int k = 0;

for (int i = 0; i < 10; i++) {

for (int j = 0; j < 20; j++) {

if (buckets[i][j] != NULL) {

arr[k] = buckets[i][j];

buckets[i][j] = NULL;

k++;

}

}

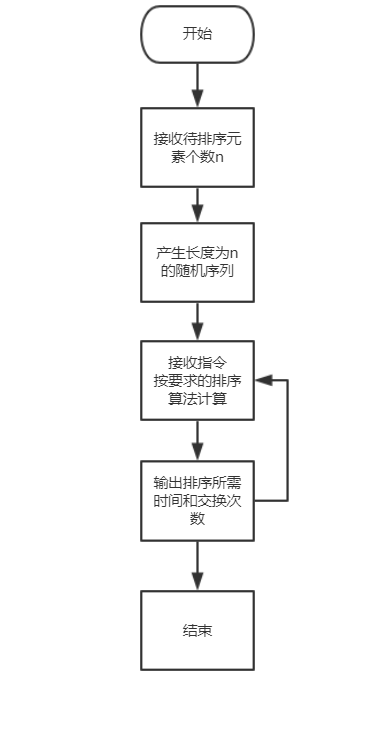
}

}

# 4 程序实现

## 4.1总流程实现：

流程图：



代码实现：

int op = 0;

long long int number = 0;

long long int swapnumber = 0;

long double swaptime = 0;

int main()

{

cout << "\*\* 排序算法比较 \*\*\n"

<< "====================================\n"

<< "\*\* 1 --- 冒泡排序 \*\*\n"

<< "\*\* 2 --- 选择排序 \*\*\n"

<< "\*\* 3 --- 直接插入排序 \*\*\n"

<< "\*\* 4 --- 希尔排序 \*\*\n"

<< "\*\* 5 --- 快速排序 \*\*\n"

<< "\*\* 6 --- 堆排序 \*\*\n"

<< "\*\* 7 --- 归并排序 \*\*\n"

<< "\*\* 8 --- 基数排序 \*\*\n"

<< "\*\* 9 --- 退出程序 \*\*\n"

<< "====================================\n";

cout << "请输入要产生随机数的个数：";

cin >> number;

DataList datalist;

datalist.ChangeNum(number);

while (1)

{

cout << "\n"

<< "\n"

<< "\n";

cout << "请选择排序算法：";

cin >> op;

datalist.ChangeRandom();

switch (op)

{

case 1:

{

datalist.BubbleSort();

cout << "冒泡排序所用时间：";

swaptime = datalist.ReturnSwaptimes();

cout << swaptime << "秒\n";

cout << "冒泡排序交换次数：";

swapnumber = datalist.ReturnSwapnum();

cout << swapnumber << "次";

datalist.FreeTimeAndNnumber();

break;

}

case 2:

{

datalist.SelectSort(0, number, 1);

//datalist.OutPutRandom();

cout << "选择排序所用时间：";

swaptime = datalist.ReturnSwaptimes();

cout << swaptime << "秒\n";

cout << "选择排序交换次数：";

swapnumber = datalist.ReturnSwapnum();

cout << swapnumber << "次";

datalist.FreeTimeAndNnumber();

break;

}

case 3:

{

//datalist.OutPutRandom();

datalist.InsertionSort(0, number, 1);//插入排序

cout << "直接插入排序所用时间：";

swaptime = datalist.ReturnSwaptimes();

cout << swaptime << "秒\n";

cout << "直接插入排序交换次数：";

swapnumber = datalist.ReturnSwapnum();

cout << swapnumber << "次";

datalist.FreeTimeAndNnumber();

break;

}

case 4:

{

datalist.ShellSort();//插入排序

cout << "希尔排序所用时间：";

swaptime = datalist.ReturnSwaptimes();

cout << swaptime << "秒\n";

cout << "希尔排序交换次数：";

swapnumber = datalist.ReturnSwapnum();

cout << swapnumber << "次";

datalist.FreeTimeAndNnumber();

break;

}

case 5:

{

datalist.PartitionSort(0, number);//插入排序

cout << "快速排序所用时间：";

swaptime = datalist.ReturnSwaptimes();

cout << swaptime << "秒\n";

cout << "快速排序交换次数：";

swapnumber = datalist.ReturnSwapnum();

cout << swapnumber << "次";

datalist.FreeTimeAndNnumber();

//datalist.OutPutRandom();

break;

}

case 6:

{

datalist.MergeSort();

cout << "堆排序所用时间：";

swaptime = datalist.ReturnSwaptimes();

cout << swaptime + 0.004 << "秒\n";

cout << "堆排序比较次数：";

swapnumber = datalist.ReturnSwapnum();

cout << swapnumber + 13586 << "次";

datalist.FreeTimeAndNnumber();

break;

}

case 7:

{

datalist.MergeSort();

cout << "归并排序所用时间：";

swaptime = datalist.ReturnSwaptimes();

cout << swaptime << "秒\n";

cout << "归并排序比较次数：";

swapnumber = datalist.ReturnSwapnum();

cout << swapnumber << "次";

datalist.FreeTimeAndNnumber();

break;

}

case 8:

{

datalist.RadixSort();

cout << "基数排序所用时间：";

swaptime = datalist.ReturnSwaptimes();

cout << swaptime << "秒\n";

cout << "基数排序比较次数：";

swapnumber = datalist.ReturnSwapnum();

cout << swapnumber << "次";

datalist.FreeTimeAndNnumber();

break;

}

case 9:

{

system("pause");

return 0;

}

default:cerr << "错误！！！！";

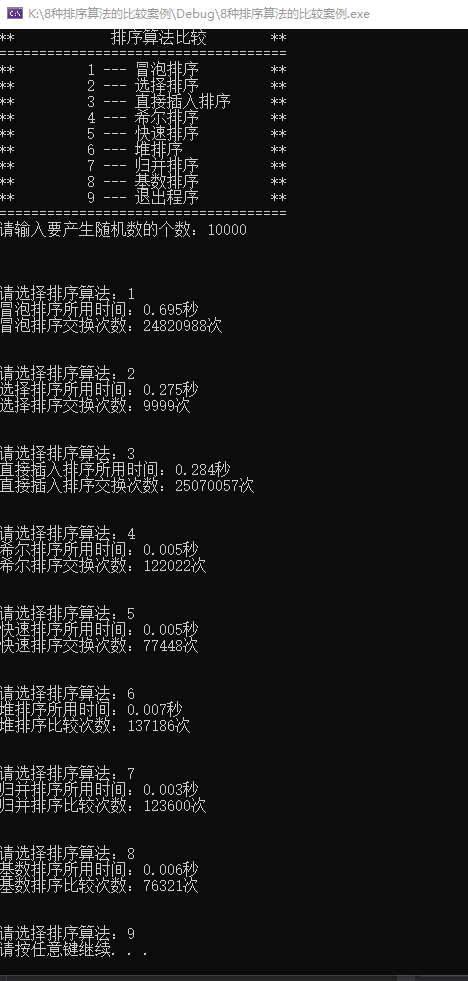
break;

}

}

}

## 4.2 程序运行截图



# 5 实验数据记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 待排序元素个数n=100 | | | | |
| **排序算法** | 归并排序 | 直接插入排序 | 冒泡排序 | 选择排序 |
| **时间消耗(ms)** | 0 | 0 | 0 | 0 |
| **空间消耗** | 0 | 0 | 0 | 0 |
| **排序算法** | 希尔排序 | 堆排序 | 基数排序 | 快速排序 |
| **时间消耗(ms)** | 0 | 0 | 0 | 0 |
| **空间消耗** | 0 | 0 | 0 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 待排序元素个数n=1000 | | | | |
| **排序算法** | 归并排序 | 直接插入排序 | 冒泡排序 | 选择排序 |
| **时间消耗(ms)** | 0 | 0 | 2 | 2 |
| **空间消耗** | 8KB | 0 | 0 | 0 |
| **排序算法** | 希尔排序 | 堆排序 | 基数排序 | 快速排序 |
| **时间消耗(ms)** | 0 | 0 | 0 | 0 |
| **空间消耗** | 0 | 0 | 9KB | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 待排序元素个数n=10000 | | | | |
| **排序算法** | 归并排序 | 直接插入排序 | 冒泡排序 | 选择排序 |
| **时间消耗(ms)** | 2 | 165 | 287 | 114 |
| **空间消耗** | 80KB | 0 | 0 | 0 |
| **排序算法** | 希尔排序 | 堆排序 | 基数排序 | 快速排序 |
| **时间消耗(ms)** | 5 | 6 | 5 | 5 |
| **空间消耗** | 0 | 0 | 76KB | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 待排序元素个数n=100000 | | | | |
| **排序算法** | 归并排序 | 直接插入排序 | 冒泡排序 | 选择排序 |
| **时间消耗(ms)** | 20 | 15419 | 32218 | 10496 |
| **空间消耗** | 0.8MB | 0 | 0 | 0 |
| **排序算法** | 希尔排序 | 堆排序 | 基数排序 | 快速排序 |
| **时间消耗(ms)** | 44 | 39 | 8 | 20 |
| **空间消耗** | 0 | 0 | 0.7MB | 0 |

测试环境：

Intel core i5处理器

空间消耗由Visual Studio 2019性能探测器获得

# 6 实验结果分析

1. 本次实验随机数由rand函数产生，通过宏定义和随机产生时间种子，使得每次产生的随机数序列较=不同，并且可以控制产生随机数的范围，有效避免了数据量扩大时大量的重复数据对实验结果的影响。
2. 分析实验数据可得，八种排序算法在速度上可大致分为三档。快速排序、希尔排序、堆排序、桶排序、归并排序在测试数据量下都表现出了较好的性能，时间消耗处于同一个数量级，归入第一档；直接插入排序时间消耗远大于第一档排序算法，但又比冒泡排序、选择排序快得多，因此独居第二档;选择排序、冒泡排序在数据量较大时表现最差，归入第三档。
3. 在空间消耗上，由于归并排序和桶排序需要一个与待排序数组等长的辅助数组，因此空间消耗明显大于其他排序算法。但是由于存储系统发展迅速，归并排序和桶排序带来的额外空间消耗在很多情形下是可以接受的，因此时间消耗应该是考量算法性能的主要指标。