### 第6题：排序算法比较

#### 数据结构：

快速排序中使用的栈

#### 算法设计思想：

由于需要使用八种排序对同一个样本进行比较，故用一个类将输出函数，八种排序以及相关函数进行封装。

生成样本函数分为三种，一种使用rand函数随机生成样本，一种生成升序样本，一种生成降序样本

在快速排序函数中，由于样本数较大，使用递归方法使会使系统给予的堆栈溢出，故使用非递归方法，相比于递归方法，使用栈来记录每次快速排序的起始点，从而使用循环模拟快速排序的递归方法，在循环中需要判断保证基准的左右分区存在

以下为八种排序的思想：

1. 冒泡排序（Bubble Sort）：是一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢"浮"到数列的顶端。
2. 选择排序（Selection Sort）: 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置。第二步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。重复第二步，直到所有元素均排序完毕。
3. 插入排序：插入排序是一种最简单直观的排序算法，它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入
4. 希尔排序: 先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. 归并算法：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。将两个有序表合并成一个有序表。
6. 快速排序：快速排序是在冒泡排序基础上的递归分治法。
7. 堆排序：堆排序是一种利用堆的概念来排序的选择排序。堆近似完全二叉树，它满足子结点的键值或索引总是小于（或者大于）它的父节点。
8. 基数排序：将整数按位数切割成不同的数字，然后按每个位数分别比较

#### 源程序：

##### Sort.h

#ifndef SORT\_H

#define SORT\_H

#include <iostream>

#include <time.h>

using namespace std;

class Sort{

public:

int startTime;

int endTime;

int num;

int \*a;

int \*origin;

int len;

public:

Sort (int \*array,int numArr): origin(array),num(numArr),len(numArr){

}

void ShowSort(int info = -1); //info表示为第几个样本

void copyArr(); //每种排序前的初始化

void StraightInsertionSort();

void ShellSort();

void SimpleSelectionSort();

void HeapSort();

void HeapMax(int start,int end);//大根堆，维护堆的函数

void BubbleSort();

int NumberOfThree(int low,int high);//快排辅助函数

void QuickSort(int left,int right);

int QuickSortNonRecursive(int low,int high); //快排的非递归辅助函数

void MergeSort(); //2路归并非递归

void Merge(int low,int mid,int high);//归并过程

void MSort(int T\_len); //2路归并的非递归调用

void RadixSort();

int maxbit (); //基数排序的辅助函数，用于求数据的最大位数

};

#endif

##### Sort.cpp

#include<iostream>

#include"sort.h"

#include<time.h>

#include<stack>

using namespace std;

//冒泡

void Sort::BubbleSort(){

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

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

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

int temp = a[j+1];

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

a[j+1] = temp;

}

}

}

}

//选择

void Sort::SimpleSelectionSort(){

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

int min = i;

for (int j = i+1;j < len;j++){

if (a[j] < a[min])

min = j;

}

int temp = a[i];

a[i] = a[min];

a[min] = temp;

}

}

//插入

void Sort::StraightInsertionSort(){

for (int i = 1;i< len;i++){

int key = a[i];

int j = i-1;

while((j>=0) && (key<a[j])){

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

j--;

}

a[j+1] = key;

}

}

//希尔

void Sort::ShellSort(){

int h = 1;

while (h < len / 3) {

h = 3 \* h + 1;

}

while (h >= 1) {

for (int i = h; i < len; i++) {

for (int j = i; j >= h && a[j] < a[j - h]; j -= h) {

int temp = a[j];

a[j] = a[j-h];

a[j-h] = temp;

}

}

h = h / 3;

}

}

//归并

void Sort::Merge(int low,int mid,int high){

int \*T = new int[high - low + 1];

int i = low,j = mid + 1,k = 0;

while (i <= mid && j <= high){

if ( a[i] <= a[j] ){

T[k] = a[i]; i++; k++;

}

else {

T[k] = a[j]; j++; k++;

}

}

while ( i <= mid ){

T[k] = a[j]; i++; k++;

}

while ( j <= high ){

T[k] = a[j]; j++; k++;

}

for (k = 0,i = low; i<=high; k++,i++)

a[i] = T[k];

}

void Sort::MSort(int T\_len){

int i = 1;

while (i+2\*T\_len <= len){

//归并两个子序列

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

i = i+2\*T\_len;

}

if (i + T\_len <= len){

Merge(i,i+T\_len-1,len); //归并最后两个子序列

}

}

void Sort::MergeSort(){

for (int T\_len = 1; T\_len <= len; T\_len = 2\* T\_len){

MSort(T\_len);

}

}

//快排

int Sort::NumberOfThree(int low,int high){

int mid = low + ((high - low) >> 1);//右移相当于除以2

if (a[mid] > a[high])

{

std::swap(a[mid],a[high]);

}

if (a[low] > a[high])

{

std::swap(a[low],a[high]);

}

if (a[mid] > a[low])

{

std::swap(a[mid],a[low]);

}

//此时，arr[mid] <= arr[low] <= arr[high]

return a[low];

}

int Sort::QuickSortNonRecursive(int low,int high){

int pivot = NumberOfThree(low,high];

while (low < high) {

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

--high;

}

a[low] = a[high];

while (low < high && a[low] <= pivot) {

++low;

}

a[high] = a[low];

}

a[low] = pivot;

return low;

}

void Sort::QuickSort(int left,int right){

//手动利用栈来存储每次快排的起始点

//栈非空时循环获取中轴入栈

stack<int> s;

if( left<right )

{

int boundary = QuickSortNonRecursive(left,right);

if( boundary-1>left ) //确保左分区存在

{

//将左分区端点入栈

s.push(left);

s.push(boundary-1);

}

if( boundary+1<right ) //确保右分区存在

{

s.push(boundary+1);

s.push(right);

}

while( !s.empty() )

{

//得到某分区的左右边界

int r = s.top();

s.pop();

int l = s.top();

s.pop();

boundary = QuickSortNonRecursive(l,r);

if( boundary-1>l ) //确保左分区存在

{

//将左分区端点入栈

s.push(l);

s.push(boundary-1);

}

if( boundary+1<r ) //确保右分区存在

{

s.push(boundary+1);

s.push(r);

}

}

}

}

//堆排序

void Sort::HeapMax(int start,int end){

int parent = start;

int child = parent \* 2 + 1;

//建立堆

while (child <= end){

if (child + 1 <= end && a[child] < a[child + 1])

child++;

if (a[parent] > a[child])

return; //维护完成

else{

std::swap(a[parent],a[child]);

parent = child;

child = parent \* 2 + 1;

}

}

}

void Sort::HeapSort(){

int i;

//初始化

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

Sort::HeapMax(i,len-1);

for (i = len -1;i > 0;i--){

std::swap(a[0],a[i]);

Sort::HeapMax(0,i-1);

}

}

//基数排序

int Sort::maxbit(){

int maxData = a[0];

for (int i = 1;i < len; ++i){

if (maxData < a[i])

maxData = a[i];

}

int d = 1;

int p = 10;

while (maxData >= p){

maxData /= 10;

++d;

}

return d;

}

void Sort::RadixSort(){

int d = maxbit();

int \*tmp = new int[len];

int \*count = new int[10];//末位计数器

int i,j,k;//循环变量

int radix = 1;

for (i = 1;i <= d;i++){

for(j = 0; j < 10; j++)

count[j] = 0; //每次分配前清空计数器

for(j = 0; j < len; j++){

k = (a[j] / radix) % 10; //统计每个桶中的记录数

count[k]++;

}

for(j = 1; j < 10; j++)

count[j] = count[j - 1] + count[j]; //将tmp中的位置依次分配给每个桶

for(j = len - 1; j >= 0; j--){ //将所有桶中记录依次收集到tmp中

k = (a[j] / radix) % 10;

tmp[count[k] - 1] = a[j];

count[k]--;

}

for(j = 0; j < len; j++) //将临时数组的内容复制到data中

a[j] = tmp[j];

radix = radix \* 10;

}

delete []tmp;

delete []count;

}

void Sort::copyArr(){

a = (int\*)malloc(sizeof(int)\*num);

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

a[i] = origin[i];

}

}

void Sort::ShowSort(int info){

cout<<"第"<<info<<"个样本的排序用时如下:"<<endl;

copyArr();

cout<<"排序前数组初始化完成"<<endl;

startTime = clock();

BubbleSort();

endTime = clock();

free(a);

cout<<"该样本冒泡排序用时："<<endTime - startTime<<" ms"<<endl;

copyArr();

startTime = clock();

SimpleSelectionSort();

endTime = clock();

cout<<"该样本选择排序用时："<<endTime - startTime<<" ms"<<endl;

copyArr();

startTime = clock();

StraightInsertionSort();

endTime = clock();

cout<<"该样本插入排序用时："<<endTime - startTime<<" ms"<<endl;

copyArr();

startTime = clock();

ShellSort();

endTime = clock();

cout<<"该样本希尔排序用时："<<endTime - startTime<<" ms"<<endl;

copyArr();

startTime = clock();

MergeSort();

endTime = clock();

cout<<"该样本归并排序用时："<<endTime - startTime<<" ms"<<endl;

copyArr();

startTime = clock();

HeapSort();

endTime = clock();

cout<<"该样本堆排序用时："<<endTime - startTime<<" ms"<<endl;

copyArr();

startTime = clock();

QuickSort(0,49999);

endTime = clock();

cout<<"该样本快速排序用时："<<endTime - startTime<<" ms"<<endl;

copyArr();

startTime = clock();

RadixSort();

endTime = clock();

cout<<"该样本基数排序用时："<<endTime - startTime<<" ms"<<endl;

}

##### Main.cpp

#include <iostream>

#include"sort.h"

#include <time.h>

#include<fstream>

using namespace std;

#define arrNumber 50000

/\* run this program using the console pauser or add your own getch, system("pause") or input loop \*/

//随机创建样本

void CreateRandomTxt()

{

srand((int)time(0));

int i = 1;

ofstream out;

out.open("input.txt");

if (!out)

{

cout << "打开文件失败!";

exit(1);

}

for (i; i < 50001; i++)

{

out << rand() % 50000 + 1<<' ';

if (i%20==0)

{

out << endl;

}

}

out.close();

}

//创建升序样本

void CreateOrderTxtp()

{

int i = 1;

ofstream out;

out.open("input.txt");

if (!out)

{

cout << "打开文件失败!";

exit(1);

}

for (i; i < 50001; i++)

{

out << i<<' ';

if (i % 20 == 0)

{

out << endl;

}

}

cout<<"创建升序样本完成"<<endl;

out.close();

}

//创建降序样本

void CreateOrderTxtn()

{

int i = 50000;

ofstream out;

out.open("input.txt");

if (!out)

{

cout << "打开文件失败!";

exit(1);

}

for (i; i >0; i--)

{

out << i<<' ';

if (i % 20 == 0)

{

out << endl;

}

}

cout<<"创建降序样本完成"<<endl;

out.close();

}

//将样本输入数组

void Inarr(int a[])

{

int i = 1;

ifstream in;

in.open("input.txt");

if (!in)

{

cout << "打开样本文件失败!";

exit(0);

}

for (i; i <= 50000; i++)

{

in >> a[i];

}

in.close();

}

int main( ) {

int a[arrNumber];

cout <<"第一个样本（升序）排序如下"<<endl;

CreateOrderTxtp();

Inarr(a);

Sort temp(a,arrNumber);

temp.ShowSort(1);

cout <<"第二个样本（降序）排序如下"<<endl;

CreateOrderTxtn();

Inarr(a);

Sort temp1(a,arrNumber);

temp1.ShowSort(2);

for (int i = 3;i <= 10;i++){

CreateRandomTxt();

Inarr(a);

Sort temp3(a,arrNumber);

temp3.ShowSort(i);

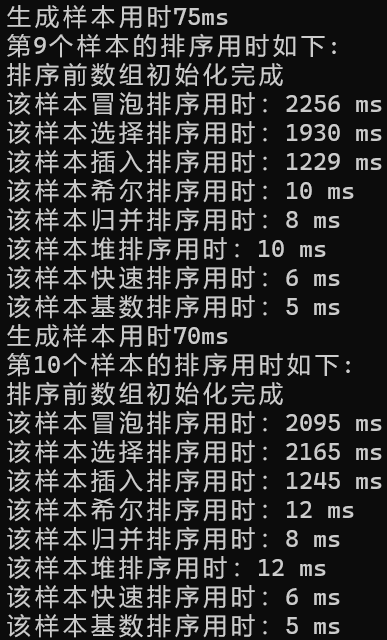
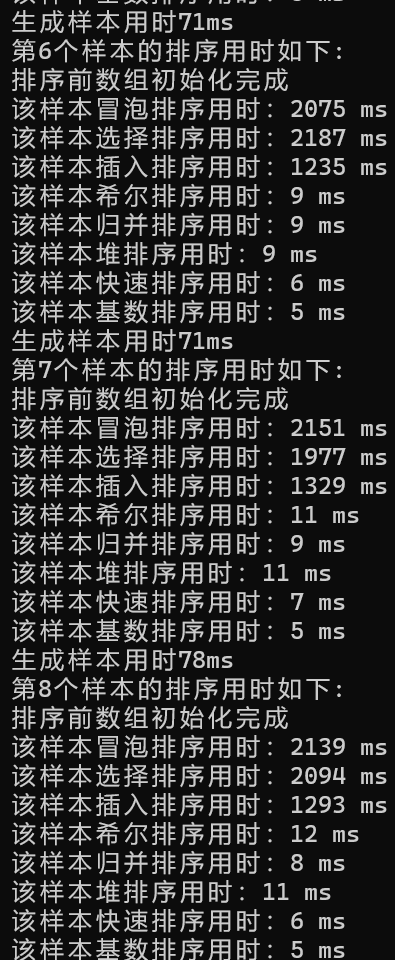
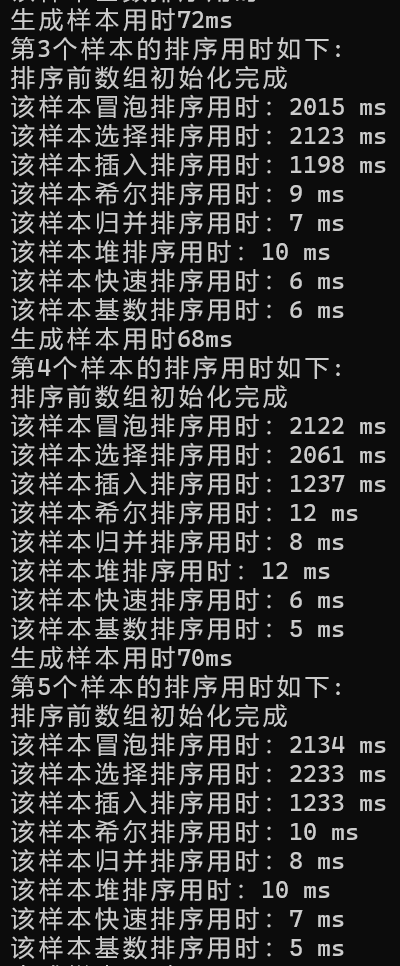
}

return 0;

}

#### 测试数据和结果：

（改进前）



**(改进后的部分结果)**

**快速排序运行时间在升序样本中由1318ms变为3ms;降序样本由1463ms变为167ms**

#### 时间复杂度：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 排序法 | 平均时间 | 最差情形 | 稳定度 | 额外空间 | 备注 |
| 冒泡 | O(n2) | O(n2) | 稳定 | O(1) | n小时较好 |
| 选择 | O(n2) | O(n2) | 不稳定 | O(1) | n小时较好 |
| 插入 | O(n2) | O(n2) | 稳定 | O(1) | 大部分已排序时较好 |
| 基数 | O(logRB) | O(logRB) | 稳定 | O(n) | B是真数(0-9)，  R是基数(个十百) |
| Shell | O(nlogn) | O(ns) 1<s<2 | 不稳定 | O(1) | s是所选分组 |
| 快速 | O(nlogn) | O(n2) | 不稳定 | O(nlogn) | n大时较好 |
| 归并 | O(nlogn) | O(nlogn) | 稳定 | O(1) | n大时较好 |
| 堆 | O(nlogn) | O(nlogn) | 不稳定 | O(1) | n大时较好 |

#### 反思：

由于每次运行都需要随机生成样本到文件中，可增加一个函数来控制是否需要新生成新的样本

可在类中有一个枚举记录排序名称，再将输出八种排序算法时间的函数分解为一个根据参数输出其中一种排序函数时间的函数，再在类中构建另外一个函数来循环输出所有排序算法时间的函数，提高健壮性，同时也便于控制输出排序算法时间的种类个数。

#### 改进

结果显示，在面临升降序样本和随机样本时，快速排序运行时间差距较大，这是由于在待排数组基本有序的情况下，选择使用固定基准效率会大大降低，若数组元素是随机的，使用固定基准常常优于随机基准。可改为三数取中的办法选择基准。在程序中改进后快速排序运行时间在升序样本中由1318ms变为3ms;降序样本由1463ms变为167ms

#### 该题代码行：

430行