**南京航空航天大学**

**数据结构课程设计报告**

|  |  |
| --- | --- |
| 学生姓名 | 张洪禹 |
| 学 号 | 161930310 |
| 学 院 | 计算机科学与技术学院/人工智能学院 |
| 专 业 | 软件工程 |
| 班 级 | 1619303 |
| 指导教师 | 高航 |

二〇二一年一月

**目 录**

课程设计报告要求：

1.所有的课程设计报告，均要有封面，包括：数据结构课程设计、班级、学号、姓名、和指导教师；

2.目录；

3.对每一题，给出自己采用的数据结构；

4.给出算法设计思想；

5.给出实现的源程序，并在必要的代码处给出注释；

6.给出测试数据和结果；

7.给出算法的时间复杂度、另外可以提出算法的改进方法；

8.结束语：说明总体完成情况，每一题的程序代码行，总代码行，心得体会；

目录

[一、系统进程统计 4](#_Toc27254)

[二、迷宫问题 15](#_Toc23605)

[三、地铁修建 18](#_Toc9969)

[四、公交线路提示 22](#_Toc17255)

[五、排序算法比较 34](#_Toc5206)

[六、数字排序 47](#_Toc4969)

[七、魔法优惠券 50](#_Toc30792)

**一、系统进程统计（必做）（链表）**

[问题描述]

**设计一个程序，每秒统计一次当前系统的进程状况，并按照内存使用自多到少排序打印输出相关信息。对已经结束的进程，另外给出一个列表，并显示该进程的结束时间和持续时间。**

[基本要求]

（1） 该题目要求使用两个链式线性表。一个链表存储当前活动进程，要求使用双向链表，排序要求是按照内存使用自多到少排序。另外一个链表存储已结束进程，要求使用单向链表，按照结束时间离当前时间的关系排序，最近的最前，最远的最后。

**（2） 每秒在窗口内更新一次当前系统进程情况，输出内容包括：进程名，持续时间，内存使用情况。**

**（3） 每秒在窗口内更新一次已结束进程情况，输出内容包括：进程名，持续时间，结束时间。**

**（4） 注意进程在这两个链表中的切换，一个进程既可被结束，也可以过一段时间后再被运行。**

**数据结构**

struct CurrentProcess { //当前进程节点

string name; //进程名

string duration; //持续时间

long memory; //使用内存

CurrentProcess\* pre = nullptr;

CurrentProcess\* next = nullptr;

CurrentProcess(string name, string duration, long memory) {

this->name = name;

this->duration = duration;

this->memory = memory;

}

};

class TwoLinkList { //双向链表类

public:

CurrentProcess\* head; //头指针

CurrentProcess\* tail; //尾指针

int length; //链表长度

TwoLinkList(); //构造函数

~TwoLinkList(); //析构函数

void clone(const TwoLinkList& l); //拷贝函数

void downSort(); //降序排列

void addNode(string name, string duration, long memory); //向尾部添加节点

bool findNode(string name); //查找结点

void deleteNode(); //删除节点

void showNode(); //输出节点

};

struct FinishedProcess { //已结束进程节点

string name; //进程名

string duration; //持续时间

string endTime; //结束时间

FinishedProcess\* next = nullptr;

FinishedProcess(string name, string duration, string endTime) {

this->name = name;

this->duration = duration;

this->endTime = endTime;

}

};

class OneLinkList { //单向链表类

public:

FinishedProcess\* head; //头指针

int length; //链表长度

OneLinkList(); //构造函数

~OneLinkList(); //析构函数

void addNode(string name, string duration, string endTime); //向头部添加节点

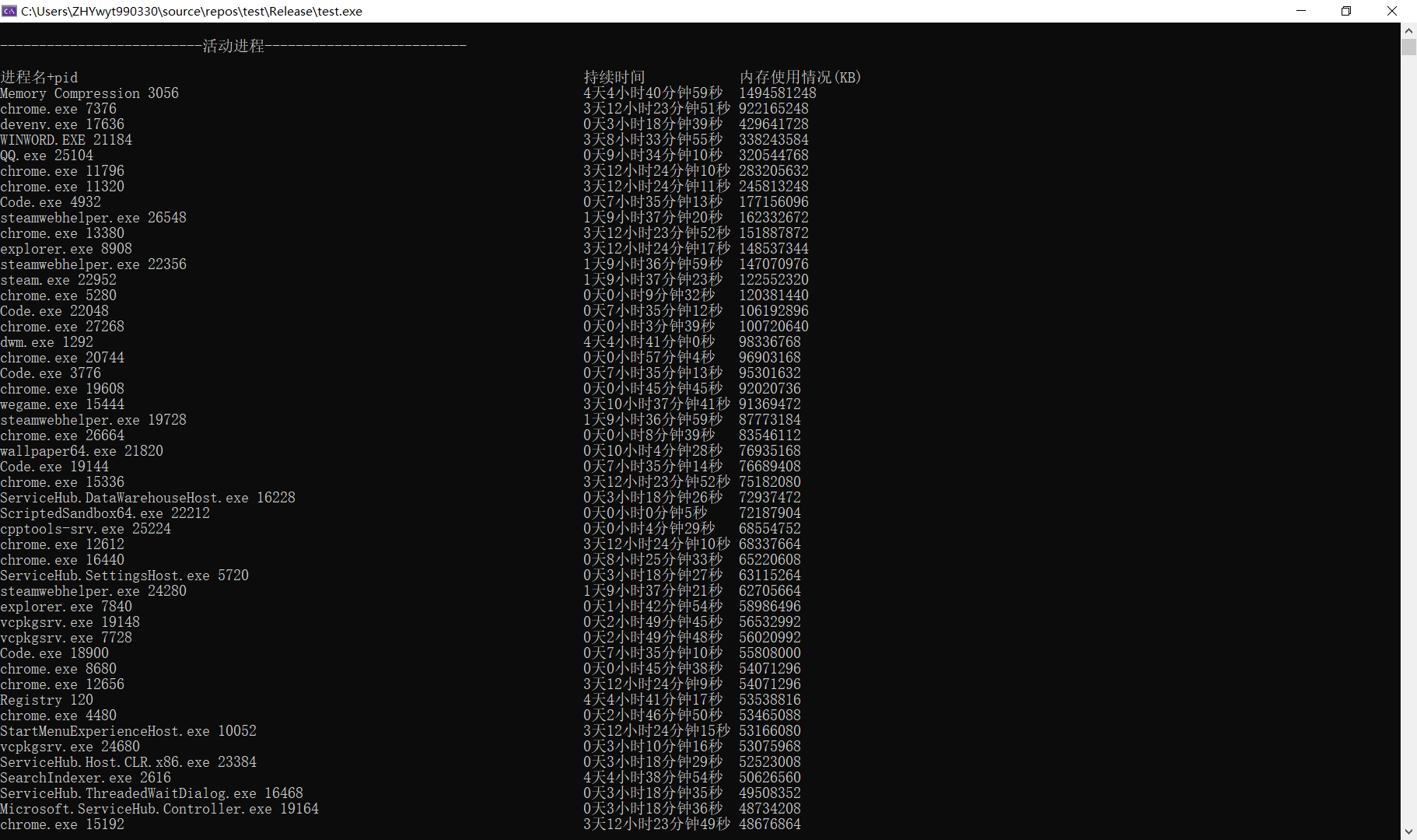
void showNode(); //输出节点

};

**算法设计思想**

通过链式结构将每个进程的信息存进节点中，通过比对每次读入的进程信息判断之前的进程中是否有已死亡的进程，放入存已死亡进程的链表中并输出。

**测试数据和结果**

**时间复杂度**

有序双向表的创建:O(n^2) 单向表的创建:O(n)

读取系统进程:O(n) 单向表的排序:O(n^2) 表的打印:O(n)  
  
**源代码**

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <windows.h>

#include <time.h>

#include <stdio.h>

#include <stdlib.h>

#include <string>

#include <vector>

#include "CurrentProcess.h"

#include "FinishedProcess.h"

using namespace std;

struct CurrentProcess { //当前进程节点

string name; //进程名

string duration; //持续时间

long memory; //使用内存

CurrentProcess\* pre = nullptr;

CurrentProcess\* next = nullptr;

CurrentProcess(string name, string duration, long memory) {

this->name = name;

this->duration = duration;

this->memory = memory;

}

};

class TwoLinkList { //双向链表类

public:

CurrentProcess\* head; //头指针

CurrentProcess\* tail; //尾指针

int length; //链表长度

TwoLinkList(); //构造函数

~TwoLinkList(); //析构函数

void clone(const TwoLinkList& l); //拷贝函数

void downSort(); //降序排列

void addNode(string name, string duration, long memory); //向尾部添加节点

bool findNode(string name); //查找结点

void deleteNode(); //删除节点

void showNode(); //输出节点

};

struct FinishedProcess { //已结束进程节点

string name; //进程名

string duration; //持续时间

string endTime; //结束时间

FinishedProcess\* next = nullptr;

FinishedProcess(string name, string duration, string endTime) {

this->name = name;

this->duration = duration;

this->endTime = endTime;

}

};

class OneLinkList { //单向链表类

public:

FinishedProcess\* head; //头指针

int length; //链表长度

OneLinkList(); //构造函数

~OneLinkList(); //析构函数

void addNode(string name, string duration, string endTime); //向头部添加节点

void showNode(); //输出节点

};

class Process

{

public:

Process();

};

TwoLinkList::TwoLinkList() {

head = nullptr;

tail = nullptr;

length = 0;

}

TwoLinkList::~TwoLinkList() {

CurrentProcess\* temp = head;

while (head != nullptr) {

head = head->next;

delete temp; //释放内存

temp = head;

}

length = 0;

head = tail = nullptr;

}

void TwoLinkList::clone(const TwoLinkList& backup) {

head = tail = nullptr;

length = 0;

if (backup.head != nullptr) {

CurrentProcess\* p = backup.head;

head = new CurrentProcess(p->name, p->duration, p->memory);

length++;

p = p->next;

CurrentProcess\* q = head;

while (p != nullptr)

{

q->next = new CurrentProcess(p->name, p->duration, p->memory);

q = q->next;

length++;

tail = p;

p = p->next;

}

int abc = 0;

}

}

void TwoLinkList::downSort() {

CurrentProcess\* p, \* q;

CurrentProcess temp("", "", 0);

for (p = head; p->next != nullptr; p = p->next) {

for (q = p->next; q != nullptr; q = q->next) {

if (p->memory < q->memory) {

temp.memory = p->memory;

temp.duration = p->duration;

temp.name = p->name;

p->memory = q->memory;

p->duration = q->duration;

p->name = q->name;

q->memory = temp.memory;

q->duration = temp.duration;

q->name = temp.name;

}

}

}

}

void TwoLinkList::addNode(string name, string duration, long memory) {

if (head == nullptr) {

head = new CurrentProcess(name, duration, memory);

length++;

tail = head;

}

else {

tail->next = new CurrentProcess(name, duration, memory);

length++;

tail->next->pre = tail;

tail = tail->next;

}

}

bool TwoLinkList::findNode(string name) {

CurrentProcess\* p;

p = head;

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

if (p->name != name) {

p = p->next;

}

else {

return true;

}

}

return false;

}

void TwoLinkList::deleteNode() {

CurrentProcess\* p = head;

CurrentProcess\* q;

while (p != nullptr) {

q = p;

p = p->next;

delete q; //释放内存

}

length = 0;

head = nullptr;

p = nullptr;

q = nullptr;

}

void TwoLinkList::showNode() {

CurrentProcess\* p;

p = head;

if (p == nullptr) {

cout << endl << "\*\*\*\*当前无活动进程！\*\*\*\*" << endl << endl;

}

else {

cout << endl << "--------------------------活动进程--------------------------" << endl << endl;

cout << setw(75) << left << "进程名+pid" << setw(20) << left << "持续时间" << "内存使用情况(KB)" << endl;

while (p != tail->next) {

cout << setw(75) << left << p->name << setw(20) << left << p->duration << p->memory << endl;

p = p->next;

}

cout << endl << "--------------------------活动进程--------------------------" << endl << endl;

}

}

OneLinkList::OneLinkList() {

head = nullptr;

length = 0;

}

OneLinkList::~OneLinkList() {

FinishedProcess\* temp = head;

if (head != nullptr) {

head = head->next;

delete temp; //释放内存

temp = head;

}

length = 0;

head = nullptr;

}

void OneLinkList::addNode(string name, string duration, string endTime) {

if (head == nullptr) {

head = new FinishedProcess(name, duration, endTime);

length++;

}

else {

FinishedProcess\* p;

p = head;

head = new FinishedProcess(name, duration, endTime);

head->next = p;

length++;

}

}

void OneLinkList::showNode() {

FinishedProcess\* p;

p = head;

if (p == nullptr) {

cout << "\*\*\*\*当前无已结束进程！\*\*\*\*" << endl;

}

else {

cout << endl << "--------------------------结束进程--------------------------" << endl << endl;

cout << setw(75) << left << "进程名+pid" << setw(20) << left << "持续时间" << "结束时间" << endl;

while (p) {

cout << setw(75) << left << p->name << setw(20) << left << p->duration << p->endTime;

p = p->next;

}

cout << endl << "--------------------------结束进程--------------------------" << endl << endl;

}

}

time\_t SwitchTime(string time) { //将yyyymmddhhmmss转为s

struct tm toSecond;

toSecond.tm\_year = stoi(time.substr(0, 4)) - 1900;

toSecond.tm\_mon = stoi(time.substr(4, 2)) - 1;

toSecond.tm\_mday = stoi(time.substr(6, 2));

toSecond.tm\_hour = stoi(time.substr(8, 2));

toSecond.tm\_min = stoi(time.substr(10, 2));

toSecond.tm\_sec = stoi(time.substr(12, 2));

return mktime(&toSecond);

}

string SwitchTime(time\_t second) { //将s转为结束时间

struct tm todate;

char time[32] = { 0 };

localtime\_s(&todate, &second);

asctime\_s(time, &todate);

return time;

}

string FormatTime(long time) { //将s转为持续时间

int days = 0, hours = 0, minutes = 0, seconds = 0;

days = time / (24 \* 60 \* 60);

hours = (time - days \* 24 \* 60 \* 60) / (60 \* 60);

minutes = (time - days \* 24 \* 60 \* 60 - hours \* 60 \* 60) / 60;

seconds = time - days \* 24 \* 60 \* 60 - hours \* 60 \* 60 - minutes \* 60;

return to\_string(days) + "天" + to\_string(hours) + "小时" + to\_string(minutes) + "分钟"

+ to\_string(seconds) + "秒";

}

Process:: Process() {

FILE\* file; //文件流

char\* cmd = (char\*)"wmic process get caption,processid,creationdate,workingsetsize"; //命令行语句

char result[128] = { 0 }; //读取结果数组

TwoLinkList oldCurrentProcess; //旧活动进程

TwoLinkList newCurrentProcess; //新活动进程

OneLinkList finishedProcess; //死亡进程

while (true) {

time\_t begin = time(NULL), end; //程序开始时间和结束时间，用于每秒间隔输出

newCurrentProcess.deleteNode(); //清空新活动进程链表

if ((file = \_popen(cmd, "r")) == NULL) {

cout << "Popen Error!" << endl;

exit(-1);

}

else {

vector<char\*>content;

string name;

string pid;

string beginTime;

string duration;

long memory;

time\_t nowTime;

char\* temp; //临时变量

memset(result, 0, 128); //初始化置0

fgets(result, 128, file); //读取第一行caption、processid、creationdate、workingsetsize不存储

while (fgets(result, 128, file) != NULL) { //读取每行数据

if (result[0] == '\r') { //跳过最后回车符

continue;

}

content.clear(); //清空vector

temp = strtok(result, " ");

do {

content.push\_back(temp);

temp = strtok(NULL, " ");

} while (temp != NULL);

memory = stol(content.at(content.size() - 2)); //读取内存大小

beginTime = content.at(content.size() - 4); //读取开始时间

beginTime = beginTime.substr(0, 14);

beginTime = to\_string(SwitchTime(beginTime));

nowTime = time(NULL); //获取当前时间

duration = FormatTime((long)nowTime - stol(beginTime)); //计算持续时间

name = "";

for (int i = 0; i < content.size() - 4; i++) {

name = name + content.at(i) + " ";

} //拼接进程名（有带空格的进程名）

pid = content.at(content.size() - 3); //获取PID

newCurrentProcess.addNode(name + pid, duration, memory); //读取一行添加一次节点

}

newCurrentProcess.downSort();

newCurrentProcess.showNode(); //输出活动进程

if (oldCurrentProcess.length == 0) { //第一次直接覆盖旧活动进程

oldCurrentProcess.clone(newCurrentProcess);

}

else { //之后先对比并添加结束进程再覆盖旧活动进程

CurrentProcess\* p = oldCurrentProcess.head;

while (p != nullptr) {

if (!newCurrentProcess.findNode(p->name)) { //如果旧进程不存在新进程链表钟，则死亡

nowTime = time(NULL); //获取当前时间

string endTime = SwitchTime(nowTime);

finishedProcess.addNode(p->name, p->duration, endTime);

}

p = p->next;

}

oldCurrentProcess.deleteNode(); //清空旧活动进程

oldCurrentProcess.clone(newCurrentProcess);

}

finishedProcess.showNode(); //输出死亡进程

if ((\_pclose(file)) == -1) {

cout << "Pclose Error!" << endl;

exit (-2);

}

end = time(NULL);

while (end - begin < 1) { //如果运行时间不足1秒，则不结束本次循环

end = time(NULL);

}

}

}

exit(0);

}

**二、迷宫问题（必做）（栈与队列）**

[问题描述]

　利用栈操作实现迷宫问题求解。

[基本要求]

（1）从文件中读取数据，生成模拟迷宫地图，不少于20行20列。

（2）给出任意入口和出口，显示输出迷宫路线。

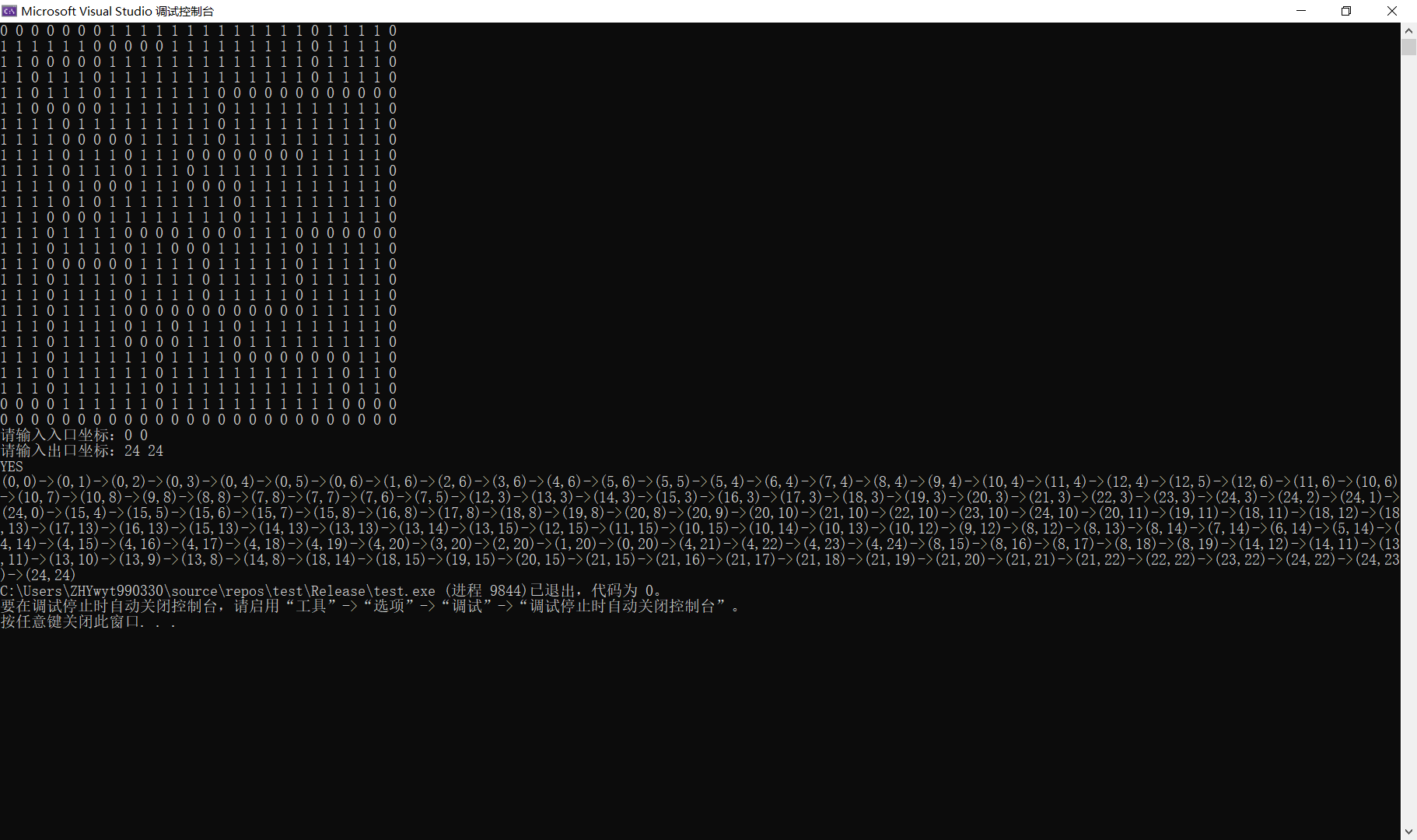
**数据结构**

由于使用BFS的搜索方法，用到的数据结构为栈

**算法设计思想**

通过对迷宫的路径进行深度优先遍历寻找出口。

**测试数据和结果**



**时间复杂度**

BFS时间复杂度:O(n+e)

读取文件形成迷宫：O(n)

输出：O(n)  
**源代码**

#include<vector>

#include<queue>

#include<iostream>

using namespace std;

struct nodesearch {

int x, y;

};

class Search

{

private:

long long length, width;

int map[100][100];

int dx[4] = { 1, -1, 0, 0 }; //定义左右两个方向

int dy[4] = { 0,0,1,-1 };//定义上下两个方向

int t, n, x, y, a, b, nx, ny, m;

char str;//x,y是起点的横纵坐标,a,b是终点的横纵坐标,str是构成迷宫的字符

bool pd, flag[100][100];//后文有介绍

queue<nodesearch> road;

public:

Search();

void DFS(int x, int y);

};

#include<fstream>

Search::Search()

{

ifstream in;

in.open("map.txt", ios::in);

in >> length >> width;

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

{

for (int j = 0; j < width; j++)

{

in >> map[i][j];

}

}

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

{

for (int j = 0; j <= width; j++)

{

cout << map[i][j] << " ";

}

cout << endl;

}

cout << "请输入入口坐标：";

cin >> x >> y;

cout << "请输入出口坐标：";

cin >> a >> b;

road.push(nodesearch{ x,y });

DFS(x, y);

if (!pd) cout << "未找到路径";

}

void Search::DFS(int x, int y)

{

for (int i = 0; i < 4; i++)//判断4个方向哪个可以走

{

nx = x + dx[i];//搜索左右两个方向,判断是否可以走

ny = y + dy[i];//搜索上下两个方向,判断是否可以走

if (nx >= 0 && nx < width && ny >= 0 && ny < length && !map[nx][ny]) //nx和ny判断是否到达边界，fyg判断可不可以走([nx][ny]是不是#)

{

map[nx][ny] = true;//标志着已经走过

nodesearch now;

now.x = nx, now.y = ny;

road.push(now);

if (nx == a && ny == b)//如果到达了终点

{

cout << "YES" << endl;//输出YES

while (road.size()>1)

{

cout << "(" << road.front().x << "," << road.front().y << ")->";

road.pop();

}

cout << "(" << road.front().x << "," << road.front().y << ")";

road.pop();

pd = true;//判断是否到达了终点，并赋值为真,目的是main函数里输出"NO"时进行判断

break;//跳出循环并进行下一组数据

}

else DFS(nx, ny);//如果没到终点，就从该点再次遍历

}

}

}

**三、**地铁修建 (必做)（图）

[问题描述]

　A市有n个交通枢纽，其中1号和n号非常重要，为了加强运输能力，A市决定在1号到n号枢纽间修建一条地铁。

　　地铁由很多段隧道组成，每段隧道连接两个交通枢纽。经过勘探，有m段隧道作为候选，两个交通枢纽之间最多只有一条候选的隧道，没有隧道两端连接着同一个交通枢纽。

　　现在有n家隧道施工的公司，每段候选的隧道只能由一个公司施工，每家公司施工需要的天数一致。而每家公司最多只能修建一条候选隧道。所有公司同时开始施工。

　　作为项目负责人，你获得了候选隧道的信息，现在你可以按自己的想法选择一部分隧道进行施工，请问修建整条地铁最少需要多少天。

输入格式

　　输入的第一行包含两个整数n, m，用一个空格分隔，分别表示交通枢纽的数量和候选隧道的数量。

第2行到第m+1行，每行包含三个整数a, b, c，表示枢纽a和枢纽b之间可以修建一条隧道，需要的时间为c天。

[基本要求]

输出格式

　　输出一个整数，修建整条地铁线路最少需要的天数。

样例输入

6 6

1 2 4

2 3 4

3 6 7

1 4 2

4 5 5

5 6 6

样例输出

6

样例说明

　　可以修建的线路有两种。

　　第一种经过的枢纽依次为1, 2, 3, 6，所需要的时间分别是4, 4, 7，则整条地铁线需要7天修完；

　　第二种经过的枢纽依次为1, 4, 5, 6，所需要的时间分别是2, 5, 6，则整条地铁线需要6天修完。

第二种方案所用的天数更少。

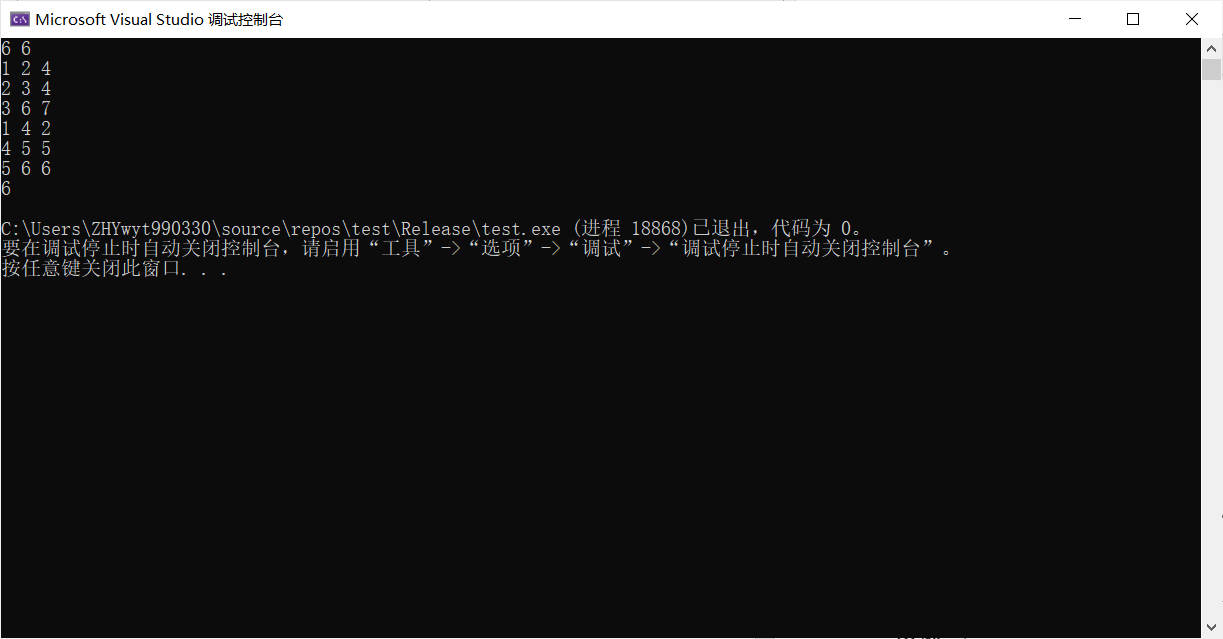
**数据结构**

**图，最小生成树**

**算法设计思想**

**kruskal算法**

**测试数据和结果**



时间复杂度

Kruskal算法：O(ElogE)

建堆(priority\_queue)：O(n)

堆排序(priority\_queue):O(nlogn)  
**源代码**

#pragma once

#include <cstdio>

#include <algorithm>

#include <iostream>

#include <queue>

using namespace std;

const int maxn(1e5 + 10);

struct edge {

int src, dest;

int dis;

edge(int src, int dest, int dis) {

this->src = src;

this->dest = dest;

this->dis = dis;

}

edge() {}

};

inline bool operator < (const edge& first, const edge& second) {

return (first.dis > second.dis);

}

class subway {

private:

int n, m;

int father[maxn];

public:

subway();

void init();

int find(int);

};

subway::subway()

{

cin >> n >> m;

init();

priority\_queue <edge> pq;

int rest = n - 1;

int src, dest, dis;

int ans = 0;

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

{

cin >> src >> dest >> dis;

pq.push(edge(src, dest, dis));

}

while (rest && !pq.empty() && find(1) != find(n)) {

edge cur = pq.top();

pq.pop();

if (find(cur.src) != find(cur.dest)) {

rest--;

father[find(cur.src)] = find(cur.dest);

ans = max(ans, cur.dis);

}

}

printf("%d\n", ans);

}

void subway::init()

{

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

father[i] = i;

}

}

int subway::find(int index)

{

if (father[index] == index) {

return index;

}

return (father[index] = find(father[index]));

}

**四、**公交线路提示 (必做)（图）

[问题描述]

上网下载真实南京公交线路图，建立南京主要公交线路图的存储结构。

[基本要求]

（1）输入任意两站点，给出转车次数最少的乘车路线。

（2）输入任意两站点，给出经过站点最少的乘车路线。

**数据结构**

typedef struct station

{

int bus\_number; //公交车号

int index; //在邻接表中的编号

struct station\* next; //下一站

} ST;

typedef struct BusGraghN

{

int stations\_number; //公交站台的数量

vector<ST> stations; //链表连接的一条公交路线

vector<ST> N; //邻接表存储

vector<ST> c\_N; //邻接表存彼此之间可以直达的路线

vector<string> st\_names; //存储对应的站台名字

vector<vector<int> > bus\_numbers; //存储可以经过的公交线路号码

} BG;

typedef struct node

{

int before;

int index;

} no; //BFS遍历的时候便于输出路径

class BUS {

public:

BUS();

void menu(BG& G, string str1, string str2, int& st\_1, int& st\_2);

void creatGragh(BG& G); //创建公交线路图

int findSt(BG& G, string name); //寻找该车站是否存在于图中

int change(string bus\_name); //将字符串转换为int型

int is\_in(BG& G, int i, int j); //判断两个站点是否之前已经相连

int is\_in\_c\_N(BG& G, int i, int j); //判断两个图站点之间是否相连（直达图）

void short\_change(BG& G, int st\_1, int st\_2); //最短转车的乘车路线

void short\_way(BG& G, int st\_1, int st\_2); //寻找两个站台之间的最短乘车路线（经过站点最少）

private:

BG G;

int st\_1, st\_2;

string str1, str2;

ST\* p;

};

**算法设计思想**

**设线路个数为RN，站点个数为N，两站之间可以直接到达（相邻）的关系数目为M，线与线之间有相同的中转站的关系数目为RM。**

**时间复杂度：**

**求最少的换乘次数的路径时，bfs是在路线为点，中转站为边的关系进行的。所以最坏情况下，所有的路线都经过一次，所有的中转站都访问一遍，O(RN+RM)。最后输出路径时，如果最坏是所有站点都经过的话（对于除给出的南京公交线路外，另外构造的特定数据而言），则输出路径要花费的时间为O(N)。**

**求最少经过站点的路径时，BFS是在站为点，站与站之间的相邻关系为边的情况下进行的。所以最坏情况下（对于除给出的南京公交线路外，另外构造的特定数据而言）,时间复杂度O(N+M)。**

**空间复杂度：**

**用到了多个辅助数组，大小有MAX\_STATIONS和MAX\_ROUTES，所以空间复杂度为O(MAX\_STATIONS+MAX\_ROUTES)。**

**测试数据和结果**

  
**源代码**

#include <iostream>

#include <cstring>

#include <queue>

#include <fstream>

#include <vector>

#include <string>

using namespace std;

typedef struct station

{

int bus\_number; //公交车号

int index; //在邻接表中的编号

struct station\* next; //下一站

} ST;

typedef struct BusGraghN

{

int stations\_number; //公交站台的数量

vector<ST> stations; //链表连接的一条公交路线

vector<ST> N; //邻接表存储

vector<ST> c\_N; //邻接表存彼此之间可以直达的路线

vector<string> st\_names; //存储对应的站台名字

vector<vector<int> > bus\_numbers; //存储可以经过的公交线路号码

} BG;

typedef struct node

{

int before;

int index;

} no; //BFS遍历的时候便于输出路径

class BUS {

public:

BUS();

void menu(BG& G, string str1, string str2, int& st\_1, int& st\_2);

void creatGragh(BG& G); //创建公交线路图

int findSt(BG& G, string name); //寻找该车站是否存在于图中

int change(string bus\_name); //将字符串转换为int型

int is\_in(BG& G, int i, int j); //判断两个站点是否之前已经相连

int is\_in\_c\_N(BG& G, int i, int j); //判断两个图站点之间是否相连（直达图）

void short\_change(BG& G, int st\_1, int st\_2); //最短转车的乘车路线

void short\_way(BG& G, int st\_1, int st\_2); //寻找两个站台之间的最短乘车路线（经过站点最少）

private:

BG G;

int st\_1, st\_2;

string str1, str2;

ST\* p;

};

BUS::BUS()

{

creatGragh(G);

while (1)

{

menu(G, str1, str2, st\_1, st\_2); //两个字符串在此时被赋值

short\_change(G, st\_1, st\_2); //最短转车

short\_way(G, st\_1, st\_2); //最短站台

}

system("pause");

}

void BUS::menu(BG& G, string str1, string str2, int& st\_1, int& st\_2)

{

int choice;

cout << "欢迎使用南京公交线路查询" << endl;

cout << "请输入起点:";

cin >> str1;

st\_1 = findSt(G, str1);

while (st\_1 == -1)

{

cout << "找不到起点，请重新输入:";

cin >> str1;

st\_1 = findSt(G, str1);

}

cout << "请输入终点:";

cin >> str2;

st\_2 = findSt(G, str2);

while (st\_2 == -1)

{

cout << "找不到终点，请重新输入:";

cin >> str2;

st\_2 = findSt(G, str2);

}

}

void BUS::creatGragh(BG& G)

{

G.stations\_number = 0;

string temp\_station\_name; //当前站台

string temp\_station\_name\_pre; //当前公交的前一个站台

string temp\_bus; //记录某一行的公交号信息

ST\* t, \* p; //申请空间

vector<int> temp\_v;

char temp\_sy;

int bus\_n; //记录对应的公交车号

int flag = 0; //记录是否未第一个公交站台

fstream f;

f.open("南京公交线路.txt", ios::in);

if (f.fail())

{

cout << "公交信息文件打开失败" << endl;

exit(0);

}

while (1)

{

flag = 0;

temp\_sy = ' ';

f >> temp\_bus;

if (f.eof())

break;

bus\_n = change(temp\_bus); //转换为公交号

t = new ST;

t->bus\_number = bus\_n; //创建公交线路地链表头

t->next = NULL;

G.stations.push\_back(\*t);

p = &G.stations.back();

while (temp\_sy == ' ') //跳过空格

f.get(temp\_sy);

while (1) //处理后面一长段的公交信息

{

int ch\_pos = 0;

temp\_station\_name.clear();

while (temp\_sy != ',' && temp\_sy != '\n') //到','为止

{

temp\_station\_name += temp\_sy;

f.get(temp\_sy);

}

//cout << temp\_station\_name << endl;

int pos = findSt(G, temp\_station\_name);

if (pos == -1) //该站并未被创建

{

G.stations\_number++;

G.st\_names.push\_back(temp\_station\_name);

t = new ST;

t->next = NULL;

t->index = G.stations\_number - 1;

G.N.push\_back(\*t);

G.c\_N.push\_back(\*t);

G.bus\_numbers.push\_back(temp\_v); //存储一个新的容器

G.bus\_numbers[G.stations\_number - 1].push\_back(bus\_n); //公交号

}

else //该站已经被创建

{

G.bus\_numbers[pos].push\_back(bus\_n);

}

t = new ST; //连接链表

int i = findSt(G, temp\_station\_name);

t->index = i;

t->bus\_number = bus\_n;

t->next = p->next;

p->next = t;

p = p->next;

if (flag) //非第一个站点才需要进行连接

{

int i = findSt(G, temp\_station\_name\_pre);

int j = findSt(G, temp\_station\_name);

if (!is\_in(G, i, j))

{

t = new ST;

t->bus\_number = bus\_n;

t->index = j;

t->next = G.N[i].next;

G.N[i].next = t;

t = new ST;

t->bus\_number = bus\_n;

t->index = i;

t->next = G.N[j].next;

G.N[j].next = t;

}

}

flag = 1; //之后为非第一个节点

temp\_station\_name\_pre = temp\_station\_name; //进行更替

if (temp\_sy == '\n')

break;

else

f.get(temp\_sy);

}

}

f.close();

//以下为根据链表创建直达图，一条路线上面的两个站台之间彼此相连

for (int i = 0; i < G.stations.size(); i++)

{

p = G.stations[i].next; //将p和t连接在一起

while (p->next)

{

t = p->next;

while (t)

{

if (!is\_in\_c\_N(G, p->index, t->index)) //创建直达图

{

ST\* temp;

temp = new ST;

temp->index = t->index;

temp->next = G.c\_N[p->index].next;

G.c\_N[p->index].next = temp;

temp = new ST;

temp->index = p->index;

temp->next = G.c\_N[t->index].next;

G.c\_N[t->index].next = temp;

}

t = t->next;

}

p = p->next;

}

}

}

int BUS::findSt(BG& G, string name)

{

for (int i = 0; i < G.stations\_number; i++)

{

if (G.st\_names[i] == name)

return i;

}

return -1;

}

int BUS::change(string bus\_name)

{

int sum = 0;

for (int i = 0; bus\_name[i] <= '9' && bus\_name[i] >= '0' && i < bus\_name.size(); i++)

{

sum += bus\_name[i] - '0';

sum \*= 10;

}

sum /= 10;

return sum;

}

int BUS::is\_in(BG& G, int i, int j)

{

ST\* p;

p = G.N[i].next;

while (p)

{

if (p->index == j)

return 1;

p = p->next;

}

return 0;

}

int BUS::is\_in\_c\_N(BG& G, int i, int j)

{

ST\* p;

p = G.c\_N[i].next;

while (p)

{

if (p->index == j)

return 1;

p = p->next;

}

return 0;

}

void BUS::short\_change(BG& G, int st\_1, int st\_2)

{

if (st\_1 == st\_2)

{

cout << "跳一下，你就到了!" << endl;

return;

}

vector<no> q;

no t, k;

ST\* p;

int base = 0, flag = 0;

//同样是采用BFS遍历即可，但是在输出的时候必须将所有的中间站点输出，即给定一条路线的起点以及

//终点，必须输出所有的中间站点，这一个功能会单独写出一个函数进行输出2

int visit[10007] = { 0 };

no head;

head.before = -1;

head.index = st\_1;

q.push\_back(head); //将根节点入队

while (base != q.size() && !flag)

{

t = q[base];

base++;

p = G.c\_N[t.index].next;

flag = 0;

while (p && !flag)

{

if (!visit[p->index])

{

k.before = base - 1;

k.index = p->index;

if (p->index == st\_2)

flag = 1;

q.push\_back(k);

}

p = p->next;

}

}

if (base == q.size())

{

cout << "未找到路径" << endl;

return;

}

else //找到路径

{

vector<int> pr; //用于输出的数组，倒序输出

vector<int>::iterator i;

while (k.before != -1)

{

pr.push\_back(k.index);

k = q[k.before];

}

pr.push\_back(k.index);

i = pr.end() - 1;

cout << "起点：" << G.st\_names[pr.back()];

for (int j = 1; j < pr.size(); j++)

{

cout << " --- " << G.st\_names[\*(i - j)];

if ((j - 1) % 8 == 0 && j != 1)

cout << "\n";

}

cout << "\n";

}

}

void BUS::short\_way(BG& G, int st\_1, int st\_2)

{

if (st\_1 == st\_2)

{

cout << "跳一下，你就到了!" << endl;

return;

}

vector<no> q;

no t, k;

ST\* p;

int base = 0, flag = 0;

//采用BFS遍历即可

int visit[10007] = { 0 };

no head;

head.before = -1;

head.index = st\_1;

q.push\_back(head); //将根节点入队

while (base != q.size() && !flag)

{

t = q[base];

base++;

p = G.N[t.index].next;

flag = 0;

while (p && !flag)

{

if (!visit[p->index])

{

k.before = base - 1;

k.index = p->index;

if (p->index == st\_2)

flag = 1;

q.push\_back(k);

}

p = p->next;

}

}

if (base == q.size())

{

cout << "未找到路径" << endl;

return;

}

else //找到路径

{

vector<int> pr; //用于输出的数组，倒序输出

vector<int>::iterator i;

while (k.before != -1)

{

pr.push\_back(k.index);

k = q[k.before];

}

pr.push\_back(k.index);

i = pr.end() - 1;

cout << "起点：" << G.st\_names[pr.back()];

for (int j = 1; j < pr.size(); j++)

{

cout << " --- " << G.st\_names[\*(i - j)];

if ((j - 1) % 8 == 0 && j != 1)

cout << "\n";

}

cout << "\n";

}

}

**五、**排序算法比较 (必做)（排序）

[问题描述]

利用随机函数产生10个样本，每个样本有50000个随机整数（并使第一个样本是正序，第二个样本是逆序），利用直接插入排序、希尔排序，冒泡排序、快速排序、选择排序、堆排序，归并排序、基数排序8种排序方法进行排序（结果为由小到大的顺序），并统计每一种排序算法对不同样本所耗费的时间。

[基本要求]

（1） 原始数据存在文件中，用相同样本对不同算法进行测试；

（2） 屏幕显示每种排序算法对不同样本所花的时间；

**数据结构**

**排序算法比较这个题我把所有的排序函数打包到了一个类里，通过调用构造函数即可完成整个题目**

const int CountNumber = 50000;

class SortSolution

{

protected:

vector<int> data[10];

struct timeb startTime, endTime;

public:

SortSolution();

bool check();//检查排序是否成功

void Insert\_Sort(); //1.插入排序

void insertsort(vector<int>& a);//插入排序辅助函数

void Shell\_Sort(); //2.希尔排序

void ShellSort(vector<int>& a);//希尔排序辅助函数

void Bubble\_Sort(); //3.冒泡排序

void bubblesort(vector<int>& a);//冒泡排序辅助函数

void Quick\_Sort();//4.快排

void quicksort(vector<int>& a, int l, int r);//快排辅助函数

void Select\_Sort();//5.选择排序

void selectSort(vector<int>& v);//选择排序辅助函数

void Heap();//6.堆排序

void adjust(vector<int>& arr, int len, int index);

void heapSort(vector<int>& arr);

void Merge\_Sort();//7.归并排序

void merge(vector<int>& v, int left, int mid, int right);

void merge\_Sort(vector<int>& v, int left, int right);

void mergeSort(vector<int>& v);

void Radix\_Sort();//8.基数排序

int getMax(vector<int>& v);

void countSort(vector<int>& v, int exp);

void radixsort(vector<int>& v);

};

**算法设计思想**

**插入排序：对所有数据挨个遍历，并将它放到已排序好的序列里的正确位置**

**希尔排序：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。**

**冒泡排序：重复地走访过要排序的元素列，依次比较两个相邻的元素，如果顺序（如从大到小、首字母从Z到A）错误就把他们交换过来。**

**快速排序：分治思想：**

**1．先从数列中取出一个数作为基准数。**

**2．分区过程，将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。**

**3．再对左右区间重复第二步，直到各区间只有一个数。**

**选择排序：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。**

**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。**

**堆排序：将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余n-1个元素重新构造成一个堆，这样会得到n个元素的次小值。如此反复执行，便能得到一个有序序列了**

**归并排序：归并排序是用分治思想，分治模式在每一层递归上有三个步骤：**

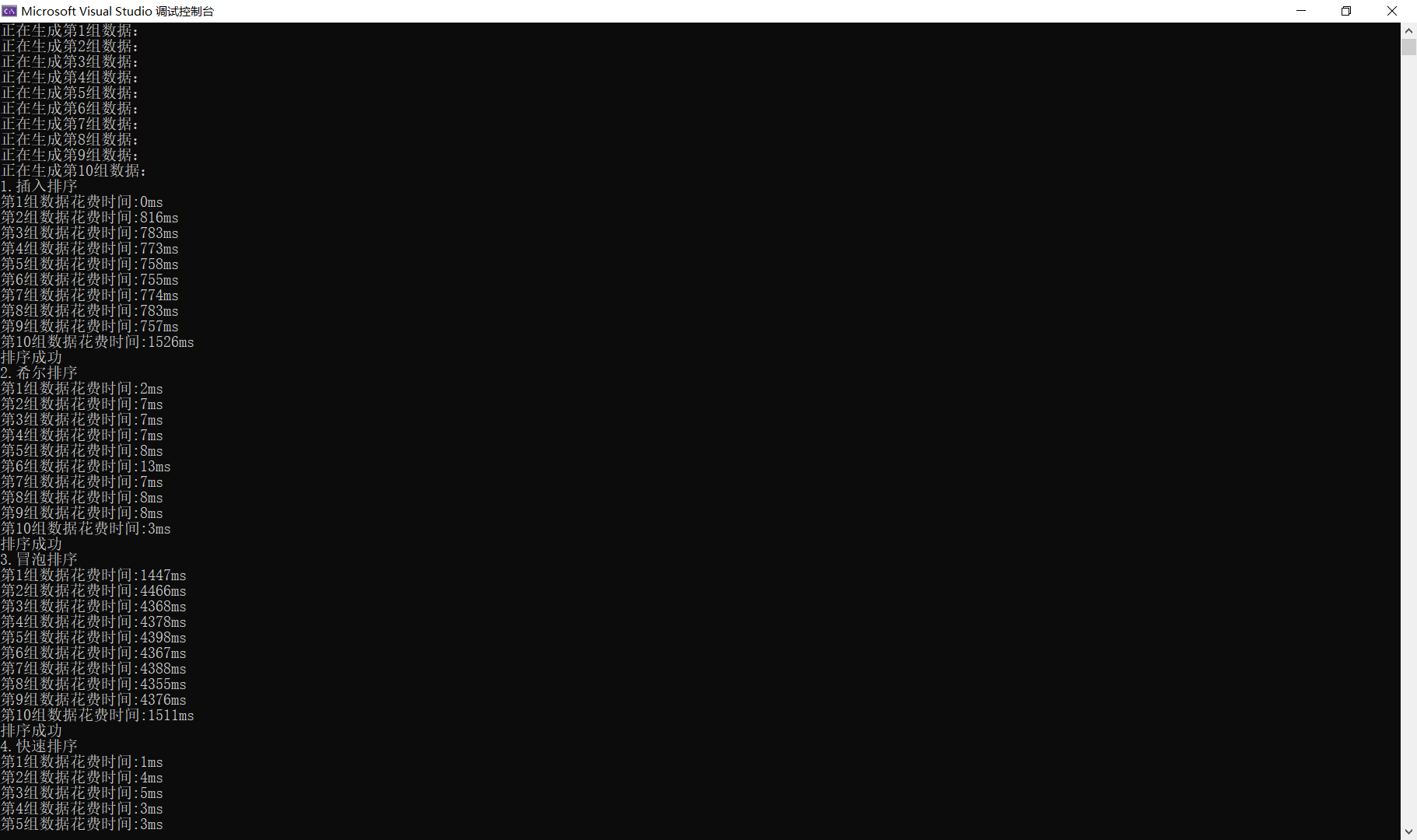
**分解（Divide）：将n个元素分成个含n/2个元素的子序列。**

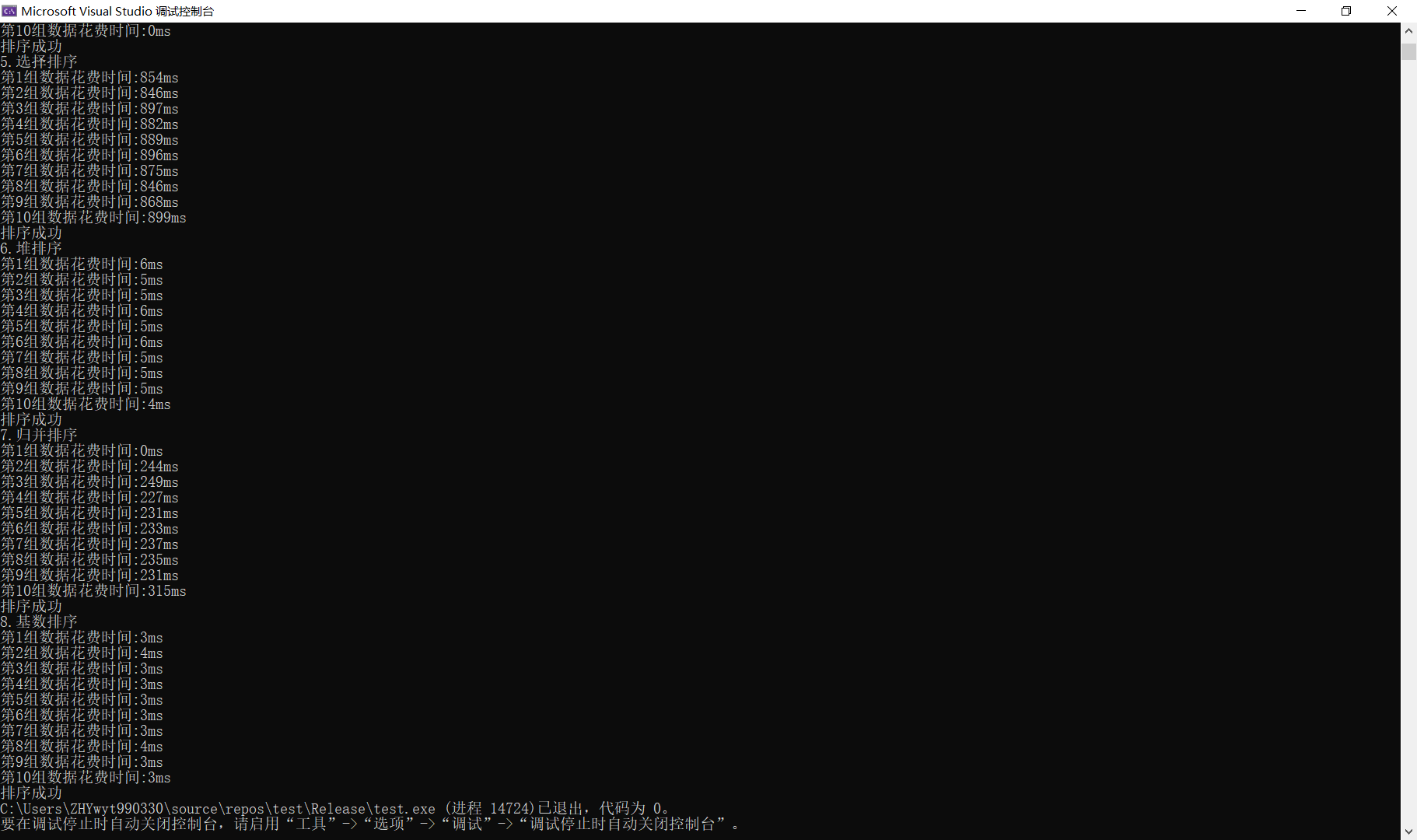
**解决（Conquer）：用合并排序法对两个子序列递归的排序。**

**合并（Combine）：合并两个已排序的子序列已得到排序结果。**

**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。**

**测试数据和结果**





时间复杂度

插入排序：O(n2)

希尔排序：O(n1.3-2)

冒泡排序：O(n2)

快速排序：**O(nlogn)**

选择排序：O(n2)

**堆排序：O(nlogn)**

**归并排序：O(nlogn)**

**基数排序：O(nlog(r)m)**

**源代码:**

#include"sort.h"

SortSolution::SortSolution()

{

srand((int)time(0));

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

{

data[i].clear();

cout << "正在生成第" << i + 1 << "组数据：";

if (i == 0)

{

for (int j = 1; j <= CountNumber; j++)

data[i].push\_back(j);

}//第一组正序

else if (i == 9)

{

for (int j = CountNumber; j >= 1; j--)

data[i].push\_back(j);

}//最后一组逆序

else

{

for (int j = 1; j <= CountNumber; j++)

{

data[i].push\_back(rand());

}

}//剩余的随机

cout << endl;

}

ofstream out("data.dat", ios::out);

ifstream in("data.dat", ios::in);

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

{

for (int j = 0; j < data[i].size(); j++)

{

out << data[i][j] << " ";

}

}

out.close();

vector<int> copy[10];

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

{

copy[i].assign(data[i].begin(), data[i].end());

}

for (int s = 1; s <= 8; s++)

{

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

{

data[i].assign(copy[i].begin(), copy[i].end());

}

switch (s)

{

case 1:

cout << s << ".插入排序" << endl;

Insert\_Sort();

break;

case 2:

cout << endl << s << ".希尔排序" << endl;

Shell\_Sort();

break;

case 3:

cout << endl << s << ".冒泡排序" << endl;

Bubble\_Sort();

break;

case 4:

cout << endl << s << ".快速排序" << endl;

Quick\_Sort();

break;

case 5:

cout << endl << s << ".选择排序" << endl;

Select\_Sort();

break;

case 6:

cout << endl << s << ".堆排序" << endl;

Heap();

break;

case 7:

cout << endl << s << ".归并排序" << endl;

Merge\_Sort();

break;

case 8:

cout << endl << s << ".基数排序" << endl;

Radix\_Sort();

break;

}

if (check())

{

cout << "排序成功";

}

else

{

cout << "排序失败" << endl;

break;

}

}

}

bool SortSolution::check()

{

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

{

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

{

if (data[i][j] > data[i][j + 1])

return false;

}

}

return true;

}

void SortSolution::Insert\_Sort()

{

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

{

ftime(&startTime);

insertsort(data[i]);

ftime(&endTime);

cout << "第" << i + 1 << "组数据花费时间:" << (endTime.time - startTime.time) \* 1000 + (endTime.millitm - startTime.millitm) << "ms" << endl;

}

}

void SortSolution::insertsort(vector<int>& a)

{

int n = a.size();

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

for (int j = i; j > 0; j--) {

if (a[j] < a[j - 1]) {

swap(a[j], a[j - 1]);

}

else {

break;

}

}

}

}

void SortSolution::Shell\_Sort()

{

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

{

ftime(&startTime);

ShellSort(data[i]);

ftime(&endTime);

cout << "第" << i + 1 << "组数据花费时间:" << (endTime.time - startTime.time) \* 1000 + (endTime.millitm - startTime.millitm) << "ms" << endl;

}

}

void SortSolution::ShellSort(vector<int>& a)

{

int increment = a.size() / 2;

while (increment)

{

for (int i = increment; i < a.size(); i += 1)//对各子表做直接插入排序

{

int temp = a[i], j;

for (j = i - increment; j >= 0 && temp < a[j]; j -= increment)

{

a[j + increment] = a[j];

}

a[j + increment] = temp;

}

increment /= 2;

}

}

void SortSolution::Bubble\_Sort()

{

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

{

ftime(&startTime);

bubblesort(data[i]);

ftime(&endTime);

cout << "第" << i + 1 << "组数据花费时间:" << (endTime.time - startTime.time) \* 1000 + (endTime.millitm - startTime.millitm) << "ms" << endl;

}

}

void SortSolution::bubblesort(vector<int>& a)

{

int n = a.size();

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

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

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

swap(a[j], a[j + 1]);

}

}

}

}

void SortSolution::Quick\_Sort()

{

int l = 0, r = CountNumber - 1;

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

{

ftime(&startTime);

quicksort(data[i], l, r);

ftime(&endTime);

cout << "第" << i + 1 << "组数据花费时间:" << (endTime.time - startTime.time) \* 1000 + (endTime.millitm - startTime.millitm) << "ms" << endl;

}

}

void SortSolution::quicksort(vector<int>& a, int l, int r)

{

int mid = a[(l + r) / 2];

int i = l, j = r;

do

{

while (a[j] > mid) j--;

while (a[i] < mid) i++;

if (i <= j)

{

swap(a[i], a[j]);

i++;

j--;

}

} while (i <= j);

if (l < j) quicksort(a, l, j);

if (i < r) quicksort(a, i, r);

}

void SortSolution::Select\_Sort()

{

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

{

ftime(&startTime);

selectSort(data[i]);

ftime(&endTime);

cout << "第" << i + 1 << "组数据花费时间:" << (endTime.time - startTime.time) \* 1000 + (endTime.millitm - startTime.millitm) << "ms" << endl;

}

}

void SortSolution::selectSort(vector<int>& v)

{

int n = v.size();

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

int index = 0;

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

if (v[j] > v[index]) {

index = j;

}

}

swap(v[index], v[n - 1 - i]);

}

}

void SortSolution::Heap()

{

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

{

ftime(&startTime);

heapSort(data[i]);

ftime(&endTime);

cout << "第" << i + 1 << "组数据花费时间:" << (endTime.time - startTime.time) \* 1000 + (endTime.millitm - startTime.millitm) << "ms" << endl;

}

}

void SortSolution::adjust(vector<int>& arr, int len, int index)

{

int left = 2 \* index + 1; // index的左子节点

int right = 2 \* index + 2;// index的右子节点

int maxIdx = index;

if (left<len && arr[left] > arr[maxIdx]) maxIdx = left;

if (right<len && arr[right] > arr[maxIdx]) maxIdx = right;

if (maxIdx != index)

{

swap(arr[maxIdx], arr[index]);

adjust(arr, len, maxIdx);

}

}

void SortSolution::heapSort(vector<int>& arr)

{

int size = arr.size();

// 构建大根堆（从最后一个非叶子节点向上）

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

{

adjust(arr, size, i);

}

// 调整大根堆

for (int i = size - 1; i >= 1; i--)

{

swap(arr[0], arr[i]); // 将当前最大的放置到数组末尾

adjust(arr, i, 0); // 将未完成排序的部分继续进行堆排序

}

}

void SortSolution::Merge\_Sort()

{

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

{

ftime(&startTime);

mergeSort(data[i]);

ftime(&endTime);

cout << "第" << i + 1 << "组数据花费时间:" << (endTime.time - startTime.time) \* 1000 + (endTime.millitm - startTime.millitm) << "ms" << endl;

}

}

void SortSolution::merge(vector<int>& v, int left, int mid, int right)

{

vector<int> temp = v;

int i = left, j = mid + 1;

int index = left;

while (i <= mid || j <= right) {

if (i > mid) {

v[index++] = temp[j];

j++;

}

else if (j > right) {

v[index++] = temp[i];

i++;

}

else if (temp[i] < temp[j]) {

v[index++] = temp[i];

i++;

}

else {

v[index++] = temp[j];

j++;

}

}

}

void SortSolution::merge\_Sort(vector<int>& v, int left, int right)

{

if (left >= right) return;

int mid = (left + right) / 2;

merge\_Sort(v, left, mid);

merge\_Sort(v, mid + 1, right);

if (v[mid] > v[mid + 1]) {

merge(v, left, mid, right);

}

}

void SortSolution::mergeSort(vector<int>& v)

{

int n = v.size();

merge\_Sort(v, 0, n - 1);

}

void SortSolution::Radix\_Sort()

{

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

{

ftime(&startTime);

radixsort(data[i]);

ftime(&endTime);

cout << "第" << i + 1 << "组数据花费时间:" << (endTime.time - startTime.time) \* 1000 + (endTime.millitm - startTime.millitm) << "ms" << endl;

}

}

int SortSolution::getMax(vector<int>& v)

{

int n = v.size();

int mx = v[0];

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

if (v[i] > mx)

mx = v[i];

return mx;

}

void SortSolution::countSort(vector<int>& v, int exp)

{

int n = v.size();

int output[CountNumber];

int i, count[10] = { 0 };

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

count[(v[i] / exp) % 10]++;

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

count[i] += count[i - 1];

for (i = n - 1; i >= 0; i--)

{

output[count[(v[i] / exp) % 10] - 1] = v[i];

count[(v[i] / exp) % 10]--;

}

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

v[i] = output[i];

}

void SortSolution::radixsort(vector<int>& v)

{

int m = getMax(v);

for (int exp = 1; m / exp > 0; exp \*= 10)

countSort(v, exp);

}

**六、**数字排序 (选做)（哈希、排序）

[问题描述]

　　给定n个整数，请统计出每个整数出现的次数，按出现次数从多到少的顺序输出。

输入格式

　　输入的第一行包含一个整数n，表示给定数字的个数。

　　第二行包含n个整数，相邻的整数之间用一个空格分隔，表示所给定的整数。

输出格式

输出多行，每行包含两个整数，分别表示一个给定的整数和它出现的次数。按出现次数递减的顺序输出。如果两个整数出现的次数一样多，则先输出值较小的，然后输出值较大的。

样例输入

12

5 2 3 3 1 3 4 2 5 2 3 5

样例输出

3 4

2 3

5 3

1 1

4 1

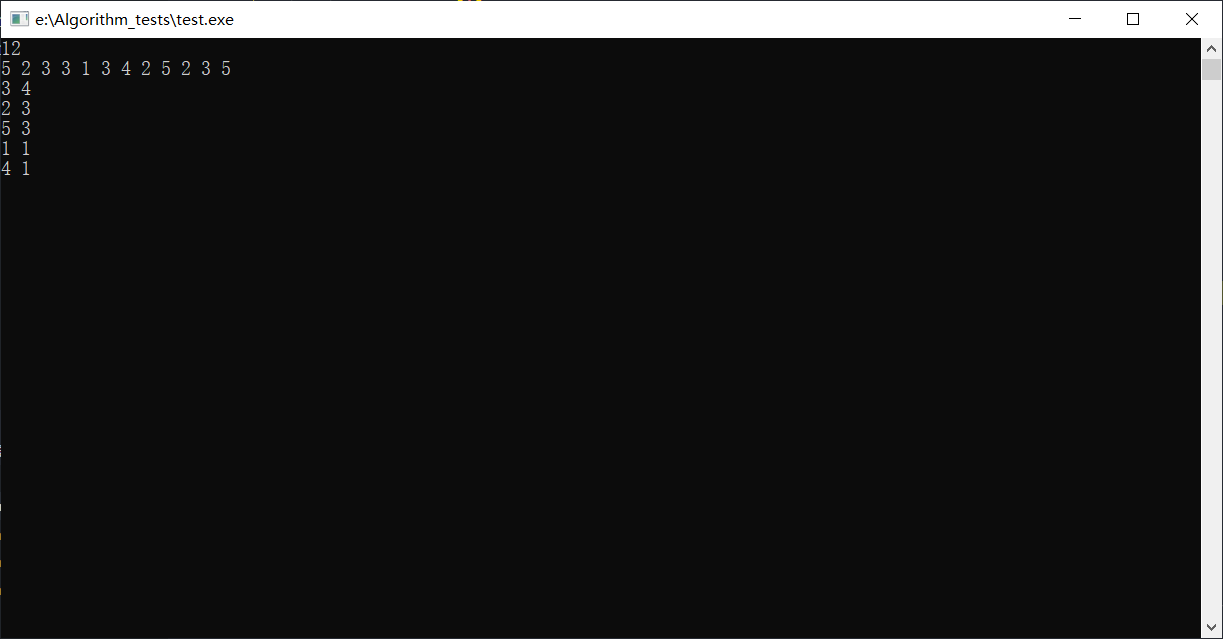
问题分析：简单地模拟样例输出，输出时有两层循环，外循环从1000遍历到1，查找出现次数最多的数，内循环从1遍历到1000，即如果两数出现的次数一样多则先输出较小的。

**数据结构**

**排序**

**算法设计思想**

**测试数据和结果**



时间复杂度

O(n2)

**源代码**

#include <bits/stdc++.h>

using namespace std;

int arr[1001];

int cnt[1001];

int main() {

int i, n;

cin >> n;

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

cin >> arr[i];

}

sort(arr, arr + n);

memset(cnt, -1, sizeof(cnt));

int tmp = arr[0];

cnt[tmp] = 1;

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

if (tmp == arr[i]) {

++cnt[tmp];

}

else {

tmp = arr[i];

cnt[tmp] = 1;

}

}

for (int j = 1000; j >= 0; --j) {

for (int i = 0; i < 1001; ++i) {

if (cnt[i] == j) {

cout << i << " " << j << endl;

}

}

}

return 0;

}

**七、**魔法优惠券(选做)（排序）

**[问题描述]**

**在火星上有个魔法商店，通过魔法优惠券。每个优惠券上印有一个整数面值K，表示若你在购买某商品使用这张优惠券，可以得到K倍该商品价值的回报。该商店还免费赠送一些有价值的商品，但如果你在领取免费赠品的时候使用面值为正的优惠券，则必须倒贴给商品K倍该商品价值的金额……但是不要紧，还有面值为负的优惠券可以用。**

**例如，给定一组优惠券，面值分别为1、2、4、-1；对应一组商品，价值为火星币7、6、-2、-3，其中负的价值表示该商品是免费赠品。我们可以将优惠券3（面值4）用在商品1（价值7）上，得到火星币28的回报。优惠券4（面值-1）用在商品4（价值-3）上，得到火星币3的回报。但是，如果一不小心把优惠券3（面值4）用到商品4（价值-3）上，你必须倒贴给商店火星币12个。同样，把优惠券4（面值-1）用到商品1（价值7）上，你必须倒贴给商店火星币7个。**

**规定每张优惠券和每件商品都只能最多被使用一次，求你可以得到的最大回报。**

**[基本要求]**

**（1）输入说明：输入有两行。第一行首先给出优惠券的个数N，随后给出N个优惠券的整数面值。 第二行首先给出商品的个数M，随后给出M个商品的整数价值。N和M在[1,106]之间，所有的数据大小不超过230，数字间以空格分隔。**

**（2）输出说明：输出可以得到的最大回报。**

**（3）测试用例：**

**输入 4 1 2 4 -1**

**4 7 6 -2 -3**

**输出 43**

**输入：4 3 2 6 1**

**3 2 6 3**

**输出：49**

**输入： 7 3 36 -1 73 2 3 6**

**6 -1 -1 -1 -1 -1 -1**

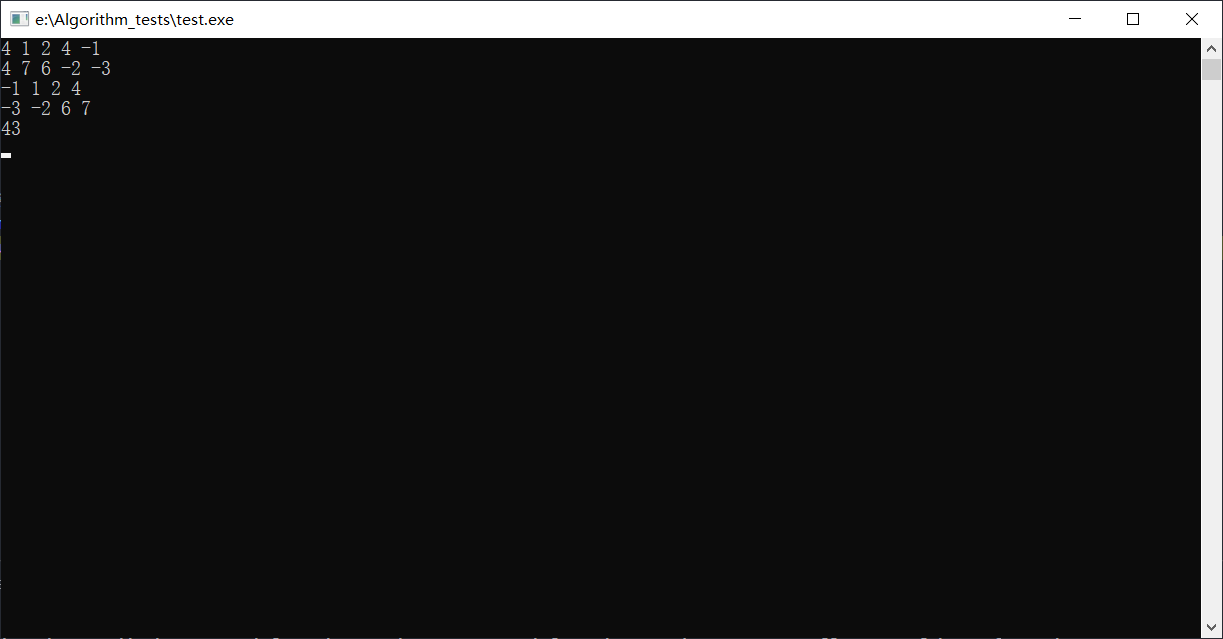
**输出： 1**

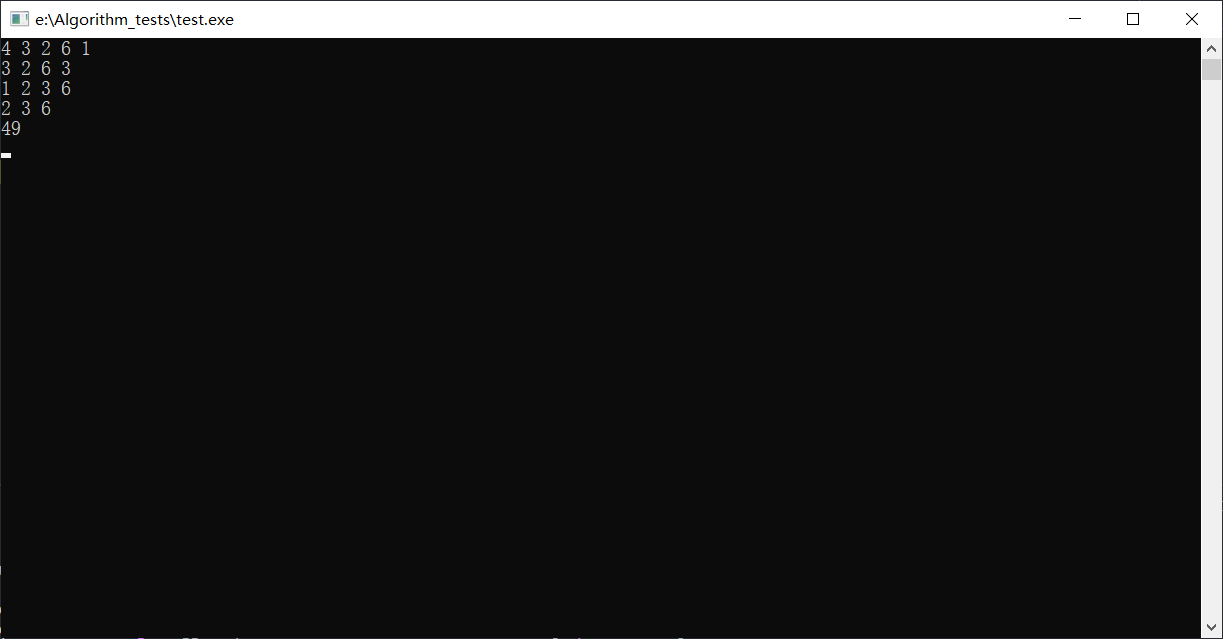
**算法设计思想**

**题目给出两行数，要求选取相同个数的数，使第一行的选出的数与第二行选出的数分别相乘的和最大**

**不难看出可以用贪心策略解决：同号相乘，对于正数，降序排序，从前到后对应下标相乘计入答案。对于负数，升序排序，从前到后对应下标相乘计入答案。直到一方或双方都没有数字。**

**测试数据和结果**





时间复杂度

**主要时间花在排序上，使用了归并排序，时间复杂度O(n\*logn)**  
**源代码**

#include <cstdio>

#define MAXN 1001000

using namespace std;

typedef long long LL;

/\*

n 第一行的数目

m 第二行数的数目

a 第一行的数字

b 第二行的数字

c 用于归并排序

ans 答案

\*/

int n, m;

LL a[MAXN], b[MAXN], c[MAXN], ans;

void Msort(LL\* now, int l, int r)

{ //归并排序 \*now：排序的int数组

if (l == r)

return;

int mid = l + r >> 1, i = l, j = mid + 1, cnt = l, k, p;

Msort(now, l, mid);

Msort(now, mid + 1, r);

// 排序当前区间

while (i <= mid && j <= r)

{

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

c[cnt++] = now[i++];

else

c[cnt++] = now[j++];

}

while (i <= mid)

c[cnt++] = now[i++];

while (j <= r)

c[cnt++] = now[j++];

for (k = l; k <= r; k++)

now[k] = c[k];

return;

}

int main()

{

// 读入数据

scanf("%d", &n);

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

{

scanf("%lld", &a[i]);

}

// 归并排序

Msort(a, 1, n);

scanf("%d", &m);

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

{

scanf("%lld", &b[i]);

}

// 归并排序

Msort(b, 1, m);

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

printf("%lld ", a[i]);

puts("");

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

printf("%lld ", b[i]);

puts("");

// 计算负数的内容 每次取最小的两个负数相乘

for (int i = n, j = m; i && j; i--, j--)

{

if (a[i] <= 0 || b[j] <= 0)

{

break;

}

ans += a[i] \* b[j];

}

// 正数每次取两行中最大的相乘

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

{

if (a[i] >= 0 || b[i] >= 0)

{

break;

}

ans += a[i] \* b[i];

}

// 输出答案

printf("%lld\n", ans);

return 0;

}