项目说明文档

数据结构课程设计

——8种排序算法的比较案例

作 者 姓 名： 王加炜

学 号： 2150265

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

# 分析

## 1.1背景分析

随机函数产生一百，一千，一万和十万个随机数，用快速排序，直接插入排序，冒泡排序，选择排序的排序方法排序，并统计每种排序所花费的排序时间和交换次数。其中，随机数的个数由用户定义，系统产生随机数。并且显示他们的比较次数。

## 1.2功能分析

请在文档中记录上述数据量下，各种排序的计算时间和存储开销，并且根据实验结果说明这些方法的优缺点。

# 设计

## 2.1数据结构设计

如上功能分析所述，该系统旨在比较各种排序算法，故设计了一个数据表（DataList）类存储生成的随机数以供排序算法调用。DataList中设置了一个数组用于存放初始生成的随机数的数组（elements）。

## 2.2类结构设计

此外，由于堆排序算法涉及到最大队的构造与调用等操作，故额外设置了一个堆的数据结构用于对数据进行堆排序的处理。

## 2.3成员与操作设计

Datalist类：

class Datalist {//数据表

private:

int maxsize;//最大可容纳的长度

int currentsize;//当前的长度

public:

int\* element;//存放数据元素

Datalist()

{

element = new int[defaultsize ];

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

{

element[i] = 0;

}

currentsize = 0;

maxsize = defaultsize;

}

~Datalist()

{

delete[]element;

}

int length()

{

return currentsize;

}

bool inset\_element(int x)//插入元素

{

if (currentsize+1 > maxsize)

{

cout << "空间已满，无法插入！" << endl;

return false;

}

else

{

element[currentsize] = x;

currentsize++;

return true;

}

}

void bulid\_element( int n)//创建数组表

{

srand(unsigned int(time(nullptr)));

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

{

inset\_element(rand() % 10000);

}

}

void output()//输出数组表

{

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

{

cout << element[i] << endl;

}

}

};

Maxheap类：

class MaxHeap

{

private:

int\* heap; //存储数据的数组表

int currentSize; //当前数组的大小

int operationNum; //交换次数

public:

MaxHeap(int num, int elems[])

{

heap = new int[num];

if (heap == NULL)

{

cerr << "Memory request error!" << endl;

exit(1);

}

currentSize = num;

operationNum = 0;

for (int i = 0; i < num; i++) //初始化

{

heap[i] = elems[i];

}

}

bool Remove();//移除数组

bool Empty()//清空数组

{

return currentSize == 0;

}

void SiftDown(const int start, const int m);//最大堆调整算法

void Swap(const int i, const int j);//交换两元素

void HeapSort();//堆排序

int GetoperationNum()//获取交换次数

{

return operationNum;

}

~MaxHeap()//析构函数

{

delete[]heap;

}

};

## 2.4功能函数

void DisplayMenu()//显示菜单

void input(int& n）//输入函数

void BubbleSort(int element[], int len, int& num)//冒泡排序

void SelectSort(int element[],int len,int& num)//选择排序

void InsertSort(int element[], int len, int& num)//插入排序

void ShellSort(int element[], int len, int& num)//希尔排序

int Partition(int element[],int low,int high,int& num)//划分函数

void QuickSort(int element[], int low,int high, int& num)//快速排序

void Merge(int List1[], int List2[], int left, int mid, int right, int& operationNum)//归并函数

void MergeSort(int List[],int List2[], int left,int right, int& operationNum)//归并排序

int getDigit(int x, int k)//获取数位函数

void RadixSort(int element[],int left,int right,int k,int& operationNum)//基排序

void OperationModule(Datalist a)//功能选择函数

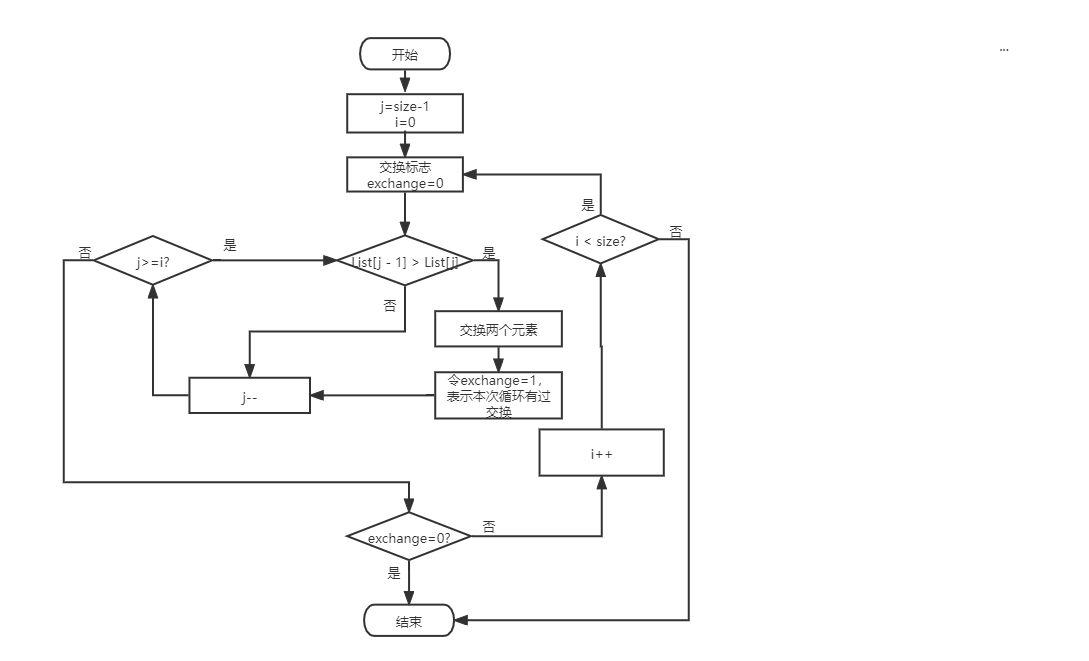
## 2.5系统设计

系统首先输出菜单，根据输入的随机数个数建立数据表，之后调用void OperationModule(Datalist a)操作模块循环输入操作码并由此执行对应的操作函数。每一个操作函数开始与结束时调用time函数获取时间最后输出函数执行时间，同时调用函数时传入变量统计交换次数。若用户输入的操作码是9，则退出系统。

# 实现

## 3.1冒泡排序

### 3.1.1流程图



### 3.1.2代码实现

void BubbleSort(int element[], int len, int& num)//冒泡排序

{

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

{

for (int j = len - 1; j > i; j--)

{

if (element[j] < element[j - 1])

{

swap(element, j, j - 1);

num++;//交换次数

}

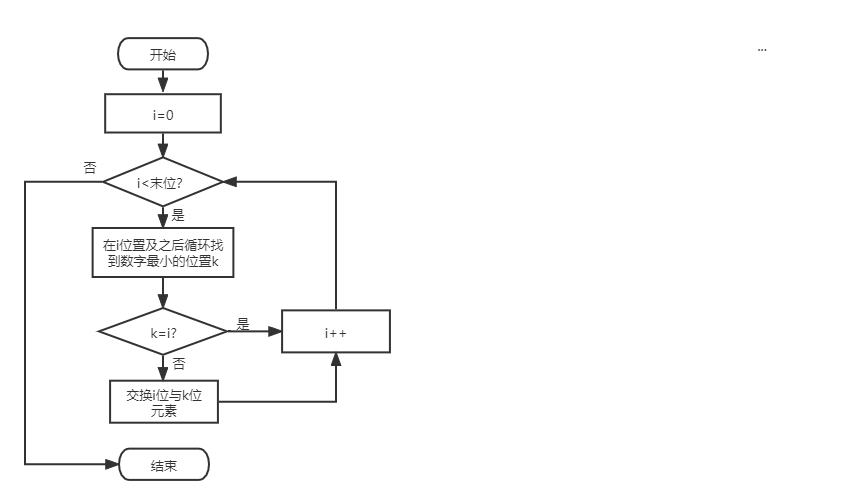
}

}

}

## 3.2选择排序

### 3.2.1流程图



### 3.2.2代码实现

void SelectSort(int element[],int len,int& num)//选择排序

{

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

{

int k = i;

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

{

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

{

k = j;

}

}

if (k != i)

{

swap(element,k, i);

num++;//交换次数

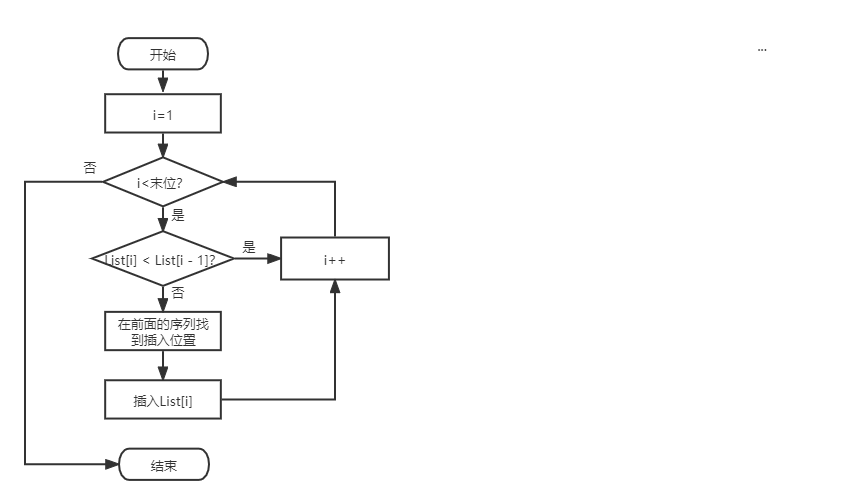
}

}

}

## 3.3直接插入排序

### 3.3.1流程图



### 3.3.2代码实现

void InsertSort(int element[], int len, int& num)//直接插入排序

{

int tmp;

int j;

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

{

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

{

tmp = element[i];

j = i - 1;

do {

element[j + 1] = element[j];

j--;

num++;//比较次数

} while (j >= 0 && tmp < element[j]);

element[j + 1] = tmp;

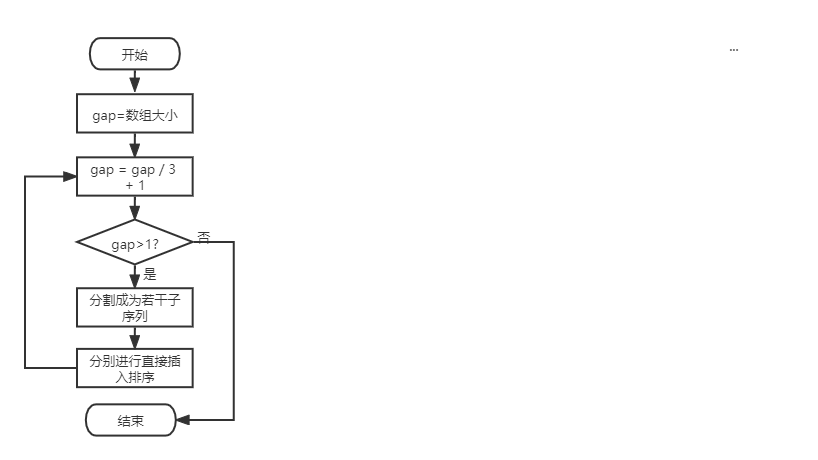
}

}

}

## 3.4希尔排序

### 3.4.1流程图



### 3.4.2代码实现

void ShellSort(int element[], int len, int& num)//希尔排序

{

int i, j, gap = len;

int temp;

do {

gap = gap / 3+1;

for (i = gap; i <= len - 1; i++)

{

if (element[i] < element[i - gap])

{

temp = element[i];

j = i - gap;

do {

element[j + gap] = element[j];

j = j - gap;

num++;//比较次数

} while (j >= 0 && temp < element[j]);

element[j + gap] = temp;

}

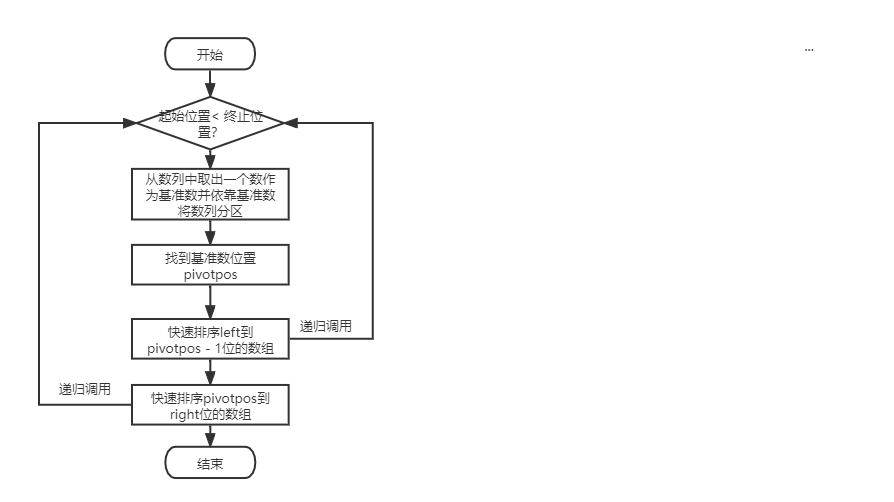
}

} while (gap > 1);

}

## 3.5快速排序

### 3.5.1流程图



### 3.5.2代码实现

int Partition(int element[],int low,int high,int& num)//划分函数

{

int pivotpos =low;

int pivot = element[low];//基准元素

for (int i = low + 1; i <= high; i++)//检索整个序列，进行划分

{

if (element[i] < pivot)//小于基准的交换到左侧

{

pivotpos++;

if (pivotpos != i)

{

swap(element, pivotpos, i);

num++;//交换次数

}

}

}

element[low] = element[pivotpos];

element[pivotpos] = pivot;//将基准元素就位

num++;

return pivotpos;//返回基准元素位置

}

void QuickSort(int element[], int low,int high, int& num)//快速排序

{

if (low < high)//元素长度大于1时

{

int pivotpos = Partition(element, low, high, num);//划分

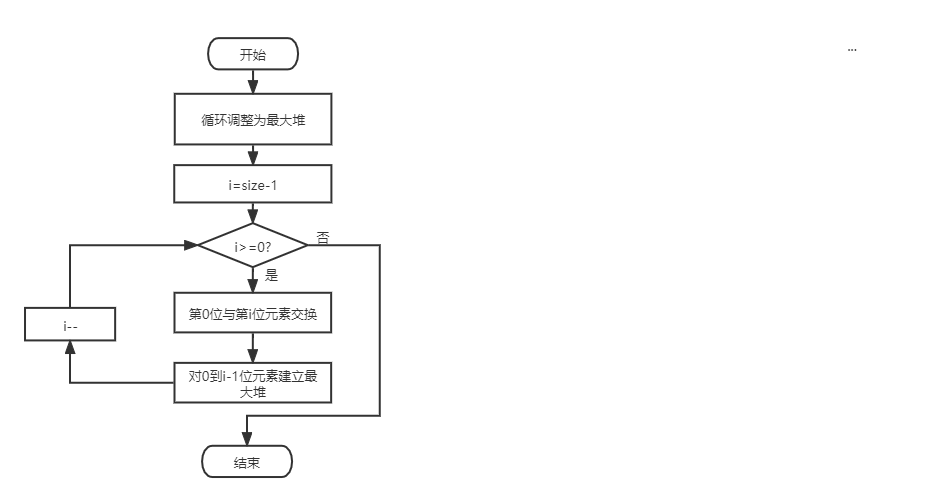
QuickSort(element, low, pivotpos - 1,num);//对左侧子序列进行同样处理

QuickSort(element, pivotpos + 1, high,num);//对右侧子序列进行同样处理

}

## 3.6堆排序

### 3.6.1流程图



### 3.6.2代码实现

void MaxHeap::SiftDown(const int start, const int end)

{

int current = start;

int child = 2 \* start + 1; //左孩子

int temp = heap[start];

while (child <= end)

{

if (child + 1 < end && heap[child] < heap[child + 1])

child = child + 1; //指向较大的孩子

if (temp >= heap[child])

break;

else { //较大的孩子往上移动

heap[current] = heap[child];

current = child;

child = 2 \* child + 1;

}

}

heap[current] = temp; //变量储存到合适位置

operationNum++;

}

void MaxHeap::HeapSort()

{

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

{

SiftDown(i, currentSize - 1); //创建堆

}

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

{

Swap(0, i);

operationNum++;

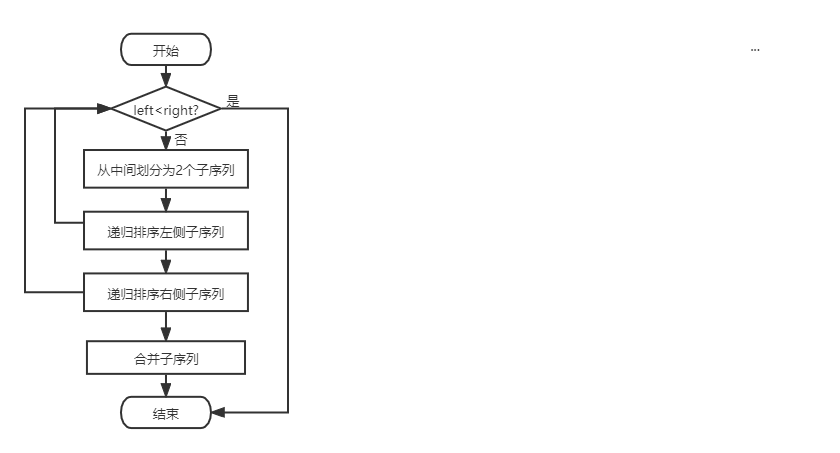
SiftDown(0, i - 1); //交换，重建最大堆

}

}

## 3.7归并排序

### 3.7.1流程图



### 3.7.2代码实现

void Merge(int List1[], int List2[], int left, int mid, int right, int& operationNum)

{

for (int k = left; k <= right; k++)//list2是list1的拷贝

{

List2[k] = List1[k];

}

int s1 = left, s2 = mid + 1,t=left;//t是当前List1的指针

while (s1 <= mid && s2 <= right) //两个表未检测完，两两比较

{

if (List2[s1] < List2[s2])

{

List1[t++] = List2[s1++];

}

else

{

List1[t++] = List2[s2++];

}

operationNum++;//比较次数

}

while (s1<= mid) //第一个表还未检测完

List1[t++] = List2[s1++];

while (s2 <= right) //第二个表还未检测完

List1[t++] = List1[s2++];

}

void MergeSort(int List[],int List2[], int left,int right, int& operationNum)

{

if (left >= right)

return;

int mid = (left + right) / 2;//从中间划分两个子序列

MergeSort(List, List2, left, mid, operationNum);//对左子序列进行递归排序

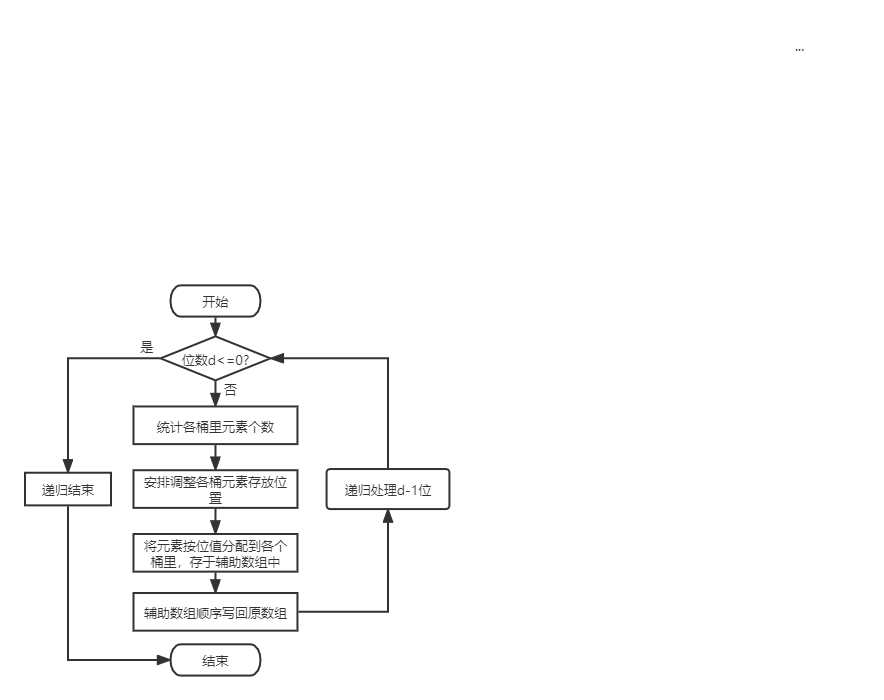
MergeSort(List, List2, mid + 1, right, operationNum);//对右子序列进行递归排序

Merge(List, List2, left, mid, right, operationNum);//合并

}

## 3.8基数排序

### 3.8.1流程图



### 3.8.2代码实现

int getDigit(int x, int k)//获取该位的值，k是从左往右数的第k位

{

if (k < 1 || k>5)

{

return -1;

}

for (int i = 1; i <= 5 - k; i++)

{

x = x / 10;

}

return x % 10;

}

void RadixSort(int element[],int left,int right,int k,int& operationNum)//基数排序

{

if (left >= right || k > 5)

return ;

int i, j, p1, p2, count[10], posit[10];

int\* auxArray = new int[right - left + 1];//暂存分配结果

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

{

count[j] = 0;

}

for (i = left; i <= right; i++)

{

count[getDigit(element[i], k)]++;//统计各桶元素个数

}

posit[0] = 0;

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

{

posit[j] = count[j - 1] + posit[j - 1];//安排各桶元素位置

}

for (i = left; i <= right; i++)//元素按位置分配到各桶

{

j = getDigit(element[i], k);//取元素A【i】的第k位的值

auxArray[posit[j]++] = element[i];//按预先计算位置存放

}

for (i = left, j = 0; i <= right; i++, j++)

{

element[i] = auxArray[j];//从辅助数组写入原数组

}

delete[]auxArray;

p1 = left;

for (j = 0; j < 10; j++)//按桶递归从第k+1位处理

{

p2 = p1 + count[j] - 1;//取子桶的首末位置

RadixSort(element, p1, p2, k + 1,operationNum);//对子桶内元素做桶排序

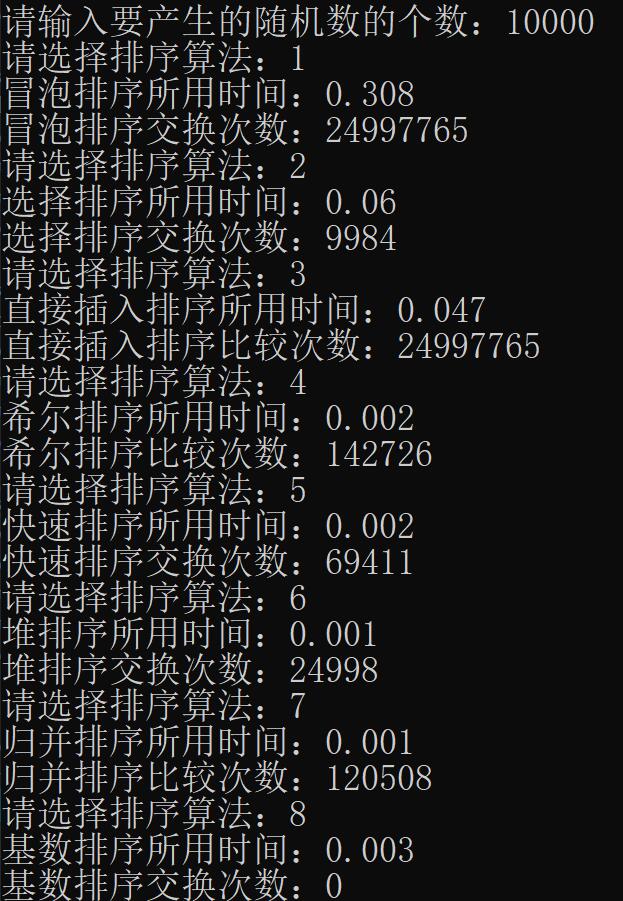
p1 = p2 + 1;

}

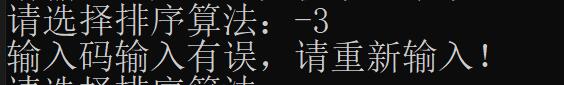
}

# 测试

## 4.1正确执行功能测试



## 4.2错误测试



# 五．总结

根据以上结果，可以得出堆排序和归并排序所用时间最短，效率最高。