项目说明文档

数据结构课程设计

——排序算法比较

作 者 姓 名： 罗佳瑞

学 号： 1952528

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 1](#_Toc17775)

[1.1 背景分析 1](#_Toc32289)

[1.2 功能分析 1](#_Toc30923)

[2 设计 2](#_Toc18172)

[2.1 数据结构设计 2](#_Toc23693)

[2.2 类结构设计 2](#_Toc11105)

[2.3 成员与操作设计 2](#_Toc8321)

[2.4 系统设计 3](#_Toc11192)

[3 实现 4](#_Toc23546)

[3.0 随机数生成的实现 4](#_Toc24692)

[3.0.1 随机数生成核心代码 4](#_Toc12923)

[3.1 冒泡排序的实现 4](#_Toc116)

[3.1.1 冒泡排序流程图 4](#_Toc25357)

[3.1.2 冒泡排序核心代码 5](#_Toc19366)

[3.1.3 冒泡排序算法分析 5](#_Toc16101)

[3.2 选择排序的实现 6](#_Toc888)

[3.2.1 选择排序流程图 6](#_Toc10879)

[3.2.2 选择排序核心代码 6](#_Toc10267)

[3.2.3 选择排序算法分析 7](#_Toc9313)

[3.3 直接插入排序的实现 8](#_Toc10413)

[3.3.1 直接插入排序流程图 8](#_Toc26399)

[3.3.2 直接插入排序核心代码 8](#_Toc26038)

[3.3.3 直接插入排序算法分析 9](#_Toc10330)

[3.4 希尔排序的实现 10](#_Toc8359)

[3.4.1 希尔排序流程图 10](#_Toc17608)

[3.4.2 希尔排序核心代码 10](#_Toc2106)

[3.4.3 希尔排序算法分析 11](#_Toc27028)

[3.5 快速排序的实现 12](#_Toc20706)

[3.5.1 快速排序流程图 12](#_Toc3472)

[3.5.2 快速排序核心代码 12](#_Toc431)

[3.5.3 快速排序算法分析 13](#_Toc24840)

[3.6 堆排序的实现 14](#_Toc8524)

[3.6.1 堆排序流程图 14](#_Toc20803)

[3.6.2 堆排序核心代码 14](#_Toc3080)

[3.6.3 堆排序算法分析 15](#_Toc25840)

[3.7 归并排序的实现 16](#_Toc17156)

[3.7.1 归并排序流程图 16](#_Toc8464)

[3.7.2 归并排序核心代码 16](#_Toc17922)

[3.7.3 归并排序算法分析 17](#_Toc25118)

[3.8 基数排序的实现 18](#_Toc21602)

[3.8.1 基数排序流程图 18](#_Toc13185)

[3.8.2 基数排序核心代码 18](#_Toc5967)

[3.8.3 基数排序算法分析 19](#_Toc16273)

[3.9 总体系统的实现 20](#_Toc1591)

[3.9.1 总体系统流程图 20](#_Toc1201)

[3.9.2 总体系统核心代码 20](#_Toc8180)

[3.9.3 总体系统截屏示例 23](#_Toc8266)

[4 测试 24](#_Toc9814)

[4.1 功能测试 24](#_Toc18105)

[4.2 出错测试 27](#_Toc14064)

[5 小结 28](#_Toc30556)

# 1 分析

## 1.1 背景分析

排序是数据处理中一种很重要也很常用的运算，一般情况下，排序操作在数据处理过程中要花费许多时间，为了提高计算机的运行效率，我们提出并不断改进各种各样的排序算法，这些算法也从不同角度展示了算法设计的重要原则和技巧，因此，研究排序算法对于理解数据结构与提升算法基础有着很重要的意义。

## 1.2 功能分析

根据各种排序的计算时间和存储开销，我们可以定量判断这些方法的优缺点。编写排序算法系统，通过随机函数产生一百，一千，一万和十万个随机数，根据用户选择的排序方法排序，并统计每种排序所花费的排序时间和比较、交换次数并显示。

# 2 设计

## 2.1 数据结构设计

如上功能分析所述，该系统要求产生随机数并对其进行排序操作，选用数组存储序列，将序列、计数辅助变量、排序函数封装到一个类中进行安全调用，控制无关变量便于比较，同时使得程序更为简洁。

## 2.2 类结构设计

排序类（Sort）封装进了待排序序列（）、各类排序函数，并且设置了两个辅助变量——交换次数（）和比较次数（），用于评判算法的优缺点。

## 2.3 成员与操作设计

**排序类（Sort）**

**私有成员：**

int amount;//随机数个数

long long compare\_degree,exchange\_degree;//比较次数，交换次数

int\* randomArray;//随机数序列

**公有操作：**

void Sort::bubbleSort();//冒泡排序

void Sort::selectionSort();//选择排序

void Sort::straightInsertion(int left, int right);//直接插入排序

void Sort::shellSort(int left, int right);//希尔排序

void Sort::quickSort(int left, int right);//快速排序

int Sort::partition(const int left, const int right);//划分函数

void Sort::heapSort(int left, int right);//堆排序

void Sort::siftDown(int start, int m);//对从start到m的结点调整

void Sort::mergeSort();//归并排序

void Sort::mergesort(int\* L, int left, int right);//递归进行划分

void Sort::merge(int\* L, const int left, const int mid, const int right);//合并

void Sort::radixSort(int left, int right);//基数排序

## 2.4 系统设计

系统首先提示用户输入随机数个数，完成对随机数序列T的创建和随机生成数据工作，然后根据用户所输入的操作码（op）执行T对应的成员排序函数。

# 3 实现

## 3.0 随机数生成的实现

### 3.0.1 随机数生成核心代码

void Sort::create()//随机生成一定数量的随机数

{

srand((unsigned)time(NULL));

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

{

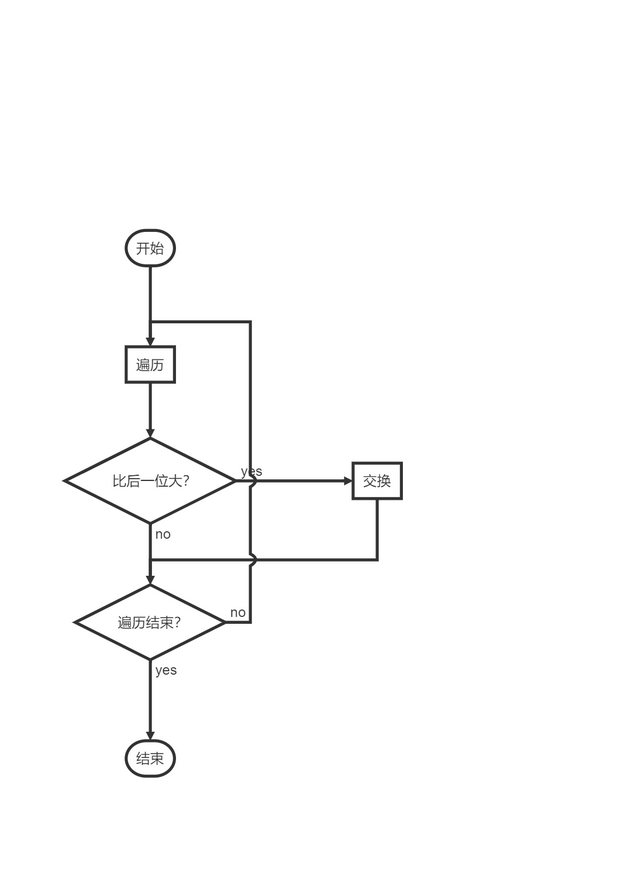
randomArray[i] = rand();

}

return;

## 3.1 冒泡排序的实现

### 3.1.1 冒泡排序流程图



### 3.1.2 冒泡排序核心代码

void Sort::bubbleSort()//冒泡算法

{

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

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

{

if (randomArray[j - 1] > randomArray[j])

{

swap(randomArray[j], randomArray[j - 1]);//不断交换，让此后的最小值到位

exchange\_degree++;

}

compare\_degree++;

}

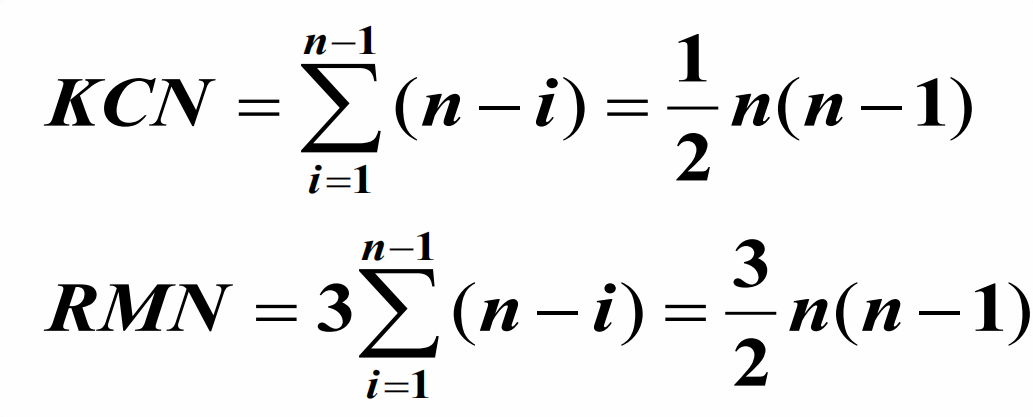
return;

}

### 3.1.3 冒泡排序算法分析

起泡排序需要一个附加对象以实现对象值的对换。

在对象的初始排列已经按关键字从小到大排好序时，此算法只执行一趟起泡，做n-1次关键字比较，不移动对象，这是最好的情形；最坏的情形是算法执行了n-1趟起泡，第i趟(1<=i<n)做了n-i次关键字比较，执行了n-i次对象交换。这样在最坏情形下总的关键字比较次数KCN和对象移动次数RMN为：



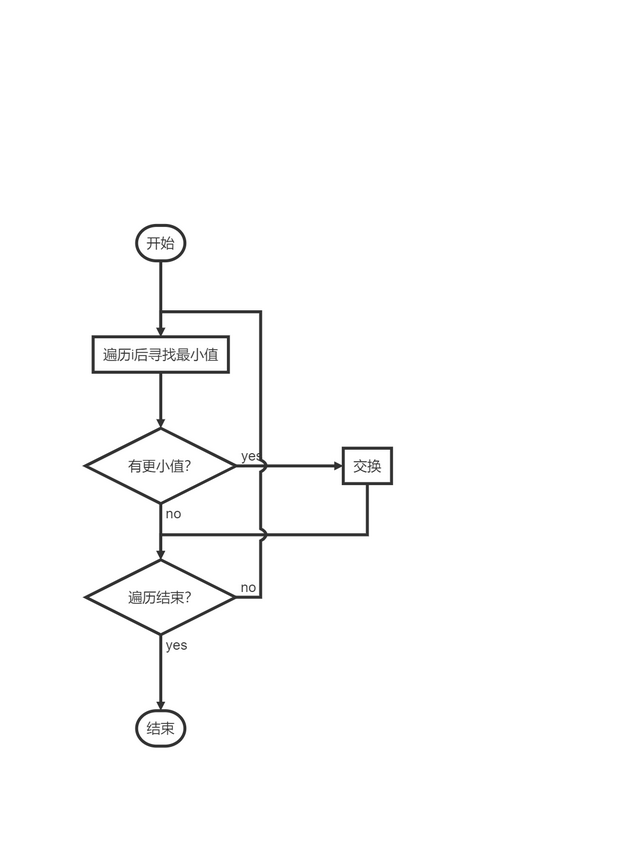
特点：

1、一趟排序后就有一个元素的位置确定，可实现部分排序。

2、起泡排序是稳定的排序方法。

## 3.2 选择排序的实现

### 3.2.1 选择排序流程图



### 3.2.2 选择排序核心代码

void Sort::selectionSort()//选择排序

{

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

{

int min = i;

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

{

if (randomArray[min] > randomArray[j])//找出此后的最小值并记录

{

min = j;

}

compare\_degree++;

}

if (min != i)

{

swap(randomArray[i], randomArray[min]);

exchange\_degree++;

}

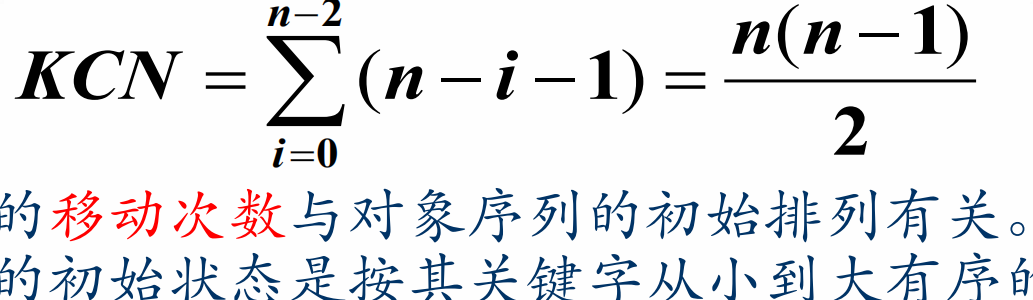
}

return;

}

### 3.2.3 选择排序算法分析

直接选择排序的关键字比较次数KCN与对象的初始排列无关。第i趟选择具有最小关键字对象所需的比较次数总是n-i-1次，此处假定整个待排序对象序列有n个对象。因此，总的关键字比较次数为

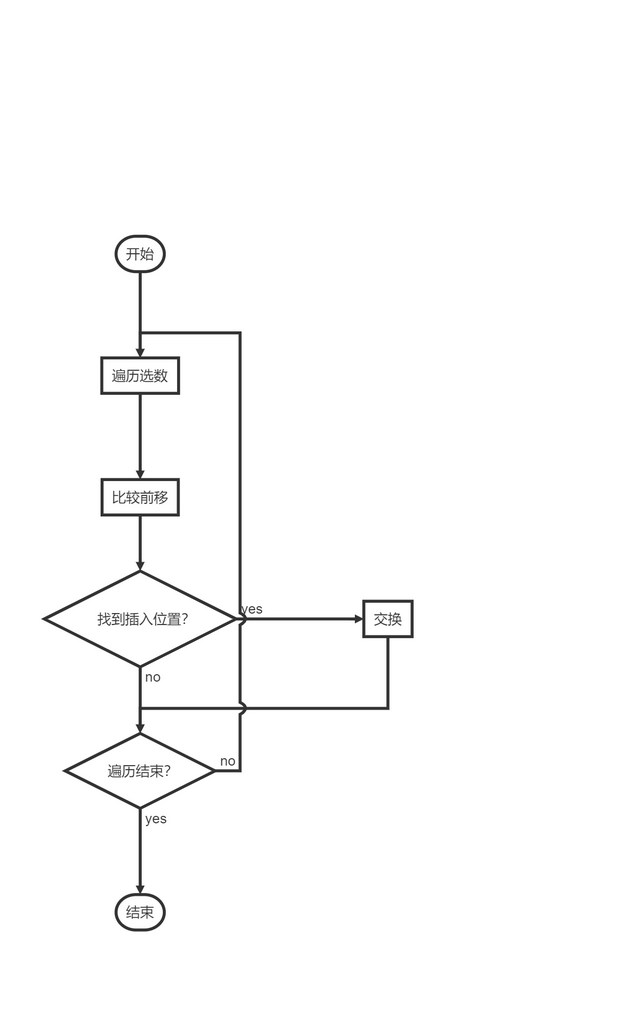


对象的移动次数与对象序列的初始排列有关。当这组对象的初始状态是按其关键字从小到大有序的时候，对象的移动次数RMN=0，达到最少。最坏情况是每一趟都要进行交换，总的对象移动次数为RMN=3(n-1)。

特点： 直接选择排序是一种不稳定的排序方法。

## 3.3 直接插入排序的实现

### 3.3.1 直接插入排序流程图



### 3.3.2 直接插入排序核心代码

void Sort::straightInsertion(int left, int right)//直接插入排序

{

int i, j;

for (i = left + 1;i < right;i++)

{

if (randomArray[i] < randomArray[i - 1]) {//假设前面都已经排好

int temp = randomArray[i];

for (j = i - 1;j >= left && temp <= randomArray[j];j--)//从要确定的数向前依次后移

{

randomArray[j + 1] = randomArray[j];

exchange\_degree++;

}

compare\_degree++;

randomArray[j + 1] = temp;//直到找到它应该在的位置将其插入

}

}

return;}

### 3.3.3 直接插入排序算法分析

直接插入排序算法需要执行n-1趟。

其空间代价为O(1)——需要一个记录的辅助存储空间监视哨。

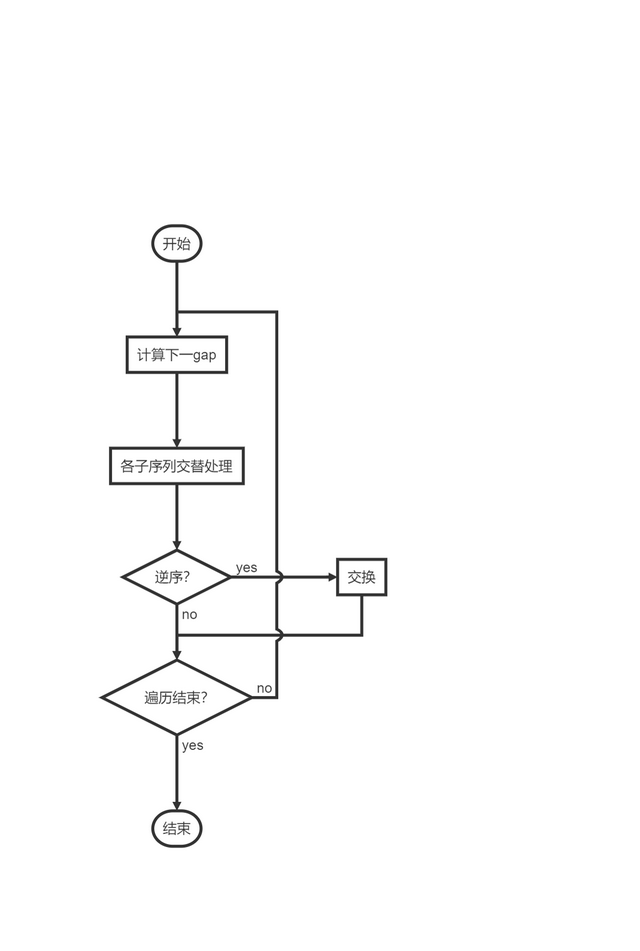
其时间代价与关键字的初始排列有关：最好情况下，排序前对象已经按关键字有序，每趟只需与前面的有序对象序列的最后一个对象的关键字比较1次，总的关键字比较次数为n-1，不需要移动元素；最坏情况下，排序前对象完全逆序，每趟比较i次，移动i次，总比较次数和移动次数2。

若待排序对象序列中出现各种可能排列的概率相同，则可取上述最好情况和最坏情况的平均情况。在平均情况下的关键字比较次数和对象移动次数约为n^2/4。因此，直接插入排序的时间复杂度为O(n^2)。

特点：直接插入排序是一种稳定的排序方法。

## 3.4 希尔排序的实现

### 3.4.1 希尔排序流程图



### 3.4.2 希尔排序核心代码

void Sort::shellSort(int left, int right)//希尔排序

{

int i, j, gap = right - left + 1, temp;

while (gap > 1) {

gap = gap / 3 + 1;//下一增量值

for (i = left + gap;i < right;i++)//各子序列交替处理

if (randomArray[i] < randomArray[i - gap]) {//当当前位置与前逆序

compare\_degree++;

temp = randomArray[i];

j = i - gap;

do {

randomArray[j + gap] = randomArray[j];//从逆序位置开始后移排序

j -= gap;//将每个子列相同位置的排好顺序，直到每个子列只剩一个值

exchange\_degree++;

} while (j >= left && temp < randomArray[j]);

randomArray[j + gap] = temp;

}

}

return;}

### 3.4.3 希尔排序算法分析

希尔排序是直接插入排序的一种改进，最好情况下(序列已有序)，时间代价为O(n),对于短序列，直接插入排序比较有效。

对特定的待排序对象序列，可以准确地估算关键字的比较次数和对象移动次数。但想要弄清关键字比较次数和对象移动次数与增量选择之间的依赖关系，并给出完整的数学分析，还没有人能够做到。

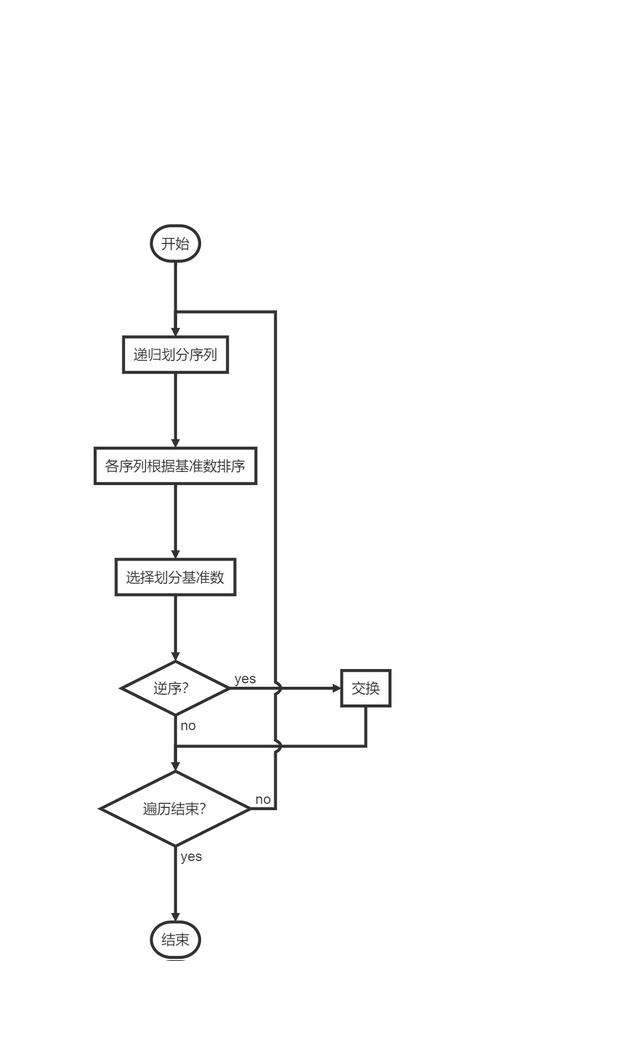
特点：

1、希尔排序是一种不稳定的排序方法。

2、gap的取法有多种。

## 3.5 快速排序的实现

### 3.5.1 快速排序流程图



### 3.5.2 快速排序核心代码

//快速排序

void Sort::quickSort(int left, int right)

{

if (right - left <= M) { straightInsertion(left, right);return; }

int pivotpos = partition(left, right - 1);

quickSort(left, pivotpos);

quickSort(pivotpos + 1, right);

return;}

//划分函数

int Sort::partition(const int left, const int right)

{

int mid = (left + right) / 2, k = left;

if (randomArray[mid] < randomArray[k])

k = mid;

if (randomArray[right] < randomArray[k])

k = right;

if (k != left) { swap(randomArray[k], randomArray[left]);exchange\_degree++; }

if (mid != right && randomArray[mid] < randomArray[right]) { swap(randomArray[mid], randomArray[right]);exchange\_degree++; }

int pivot = randomArray[right];//将最小值交换到左侧，中间值到最右侧作为划分基准

int i = left, j = right - 1;

while (i < j) {

while (i < j && randomArray[i] < pivot)i++;

while (i < j && pivot < randomArray[j])j--;

if (i < j) { swap(randomArray[i], randomArray[j]);i++;j--; }

compare\_degree++;

} //比基准小放左，大的放右，直到两头相遇

if (randomArray[i] > pivot) { randomArray[right] = randomArray[i];randomArray[i] = pivot; }//pivot移到它排序后应该在的位置

return i;

}

### 3.5.3 快速排序算法分析

快速排序的趟数取决于递归树的深度。如果每次划分对一个对象定位后，该对象的左侧子序列与右侧子序列的长度相同，则下一步将是对两个长度减半的子序列进行排序，这是最理想的情况。

在n个元素的序列中，对一个对象定位所需时间为O(n)。若设t(n)是对n个元素的序列进行排序所需的时间，而且每次对一个对象正确定位后，正好把序列划分为长度相等的两个子序列，此时，平均计算时间是o(nlog2n)。

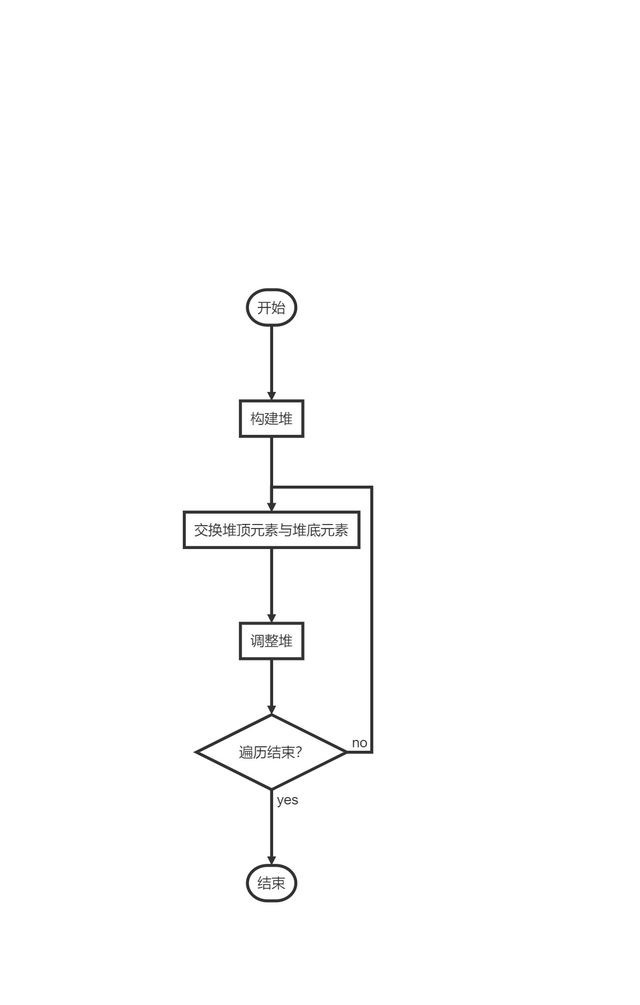
特点：

1、快速排序是一种不稳定的排序方法。

2、就平均计算时间而言，快速排序是所有内排序方法中最好的一个。

## 3.6 堆排序的实现

### 3.6.1 堆排序流程图



### 3.6.2 堆排序核心代码

//堆排序

void Sort::heapSort(int left, int right)

{

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

siftDown(i, right - 1);

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

{

swap(randomArray[0], randomArray[i]);

exchange\_degree++;

siftDown(0, i - 1);

}

}

//对从start到m的结点调整

void Sort::siftDown(int start, int m){

int i = start;int j = 2 \* i + 1;

int temp = randomArray[i];

while (j <= m) {

if (j < m && randomArray[j] < randomArray[j + 1])j++;//选择两子女中大的

if (temp >= randomArray[j])break;//当前结点大不作调整

else {

randomArray[i] = randomArray[j];//当前结点小则让大的子女上移

i = j;j = 2 \* j + 1;//i下移

}

compare\_degree++;

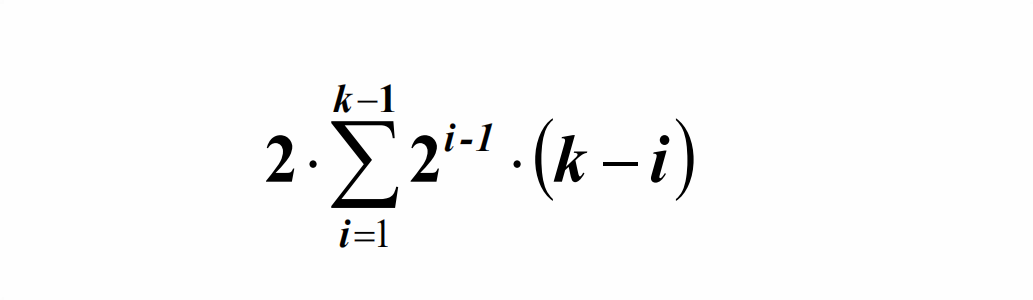
}

randomArray[i] = temp;

}

### 3.6.3 堆排序算法分析

若设堆中有n个结点，且2^(k-1)<=n<2^k，则对应的完全二叉树有k层。在第i层上的结点数<=2^(i-1)(i=1,…,k)。在第一个形成初始堆的for循环中对每一个非叶结点调用了一次堆调整算法siftDown，因此该循环所用的计算时间为：



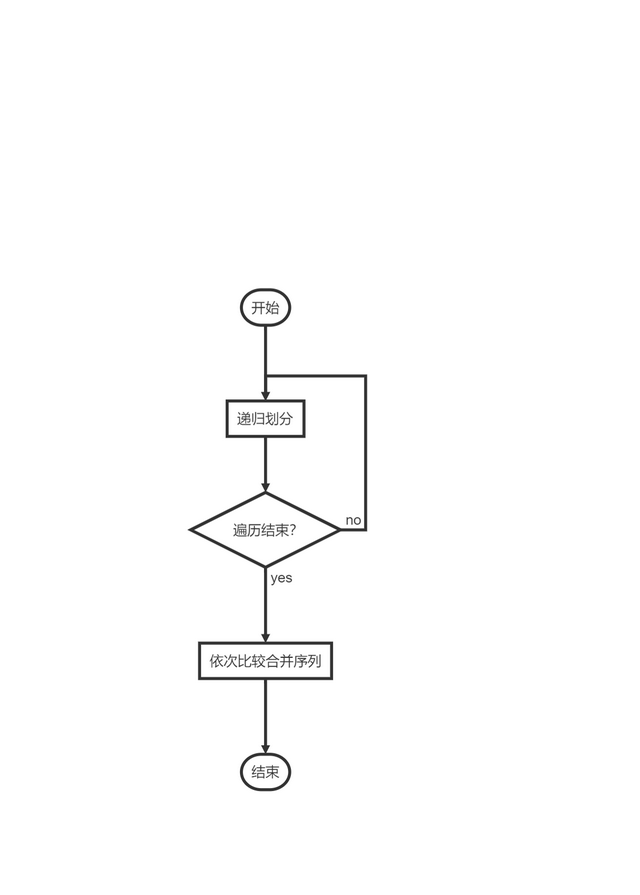
在第二个for循环中，调用了n-1次siftDown算法，该循环的计算时间为O(nlog2n)。因此，堆排序的时间复杂性为O(nlog2n)。

该算法的附加存储主要是在第二个for循环中用来执行对象交换时所用的一个临时对象。因此，该算法的空间复杂性为O(1)。

特点：堆排序是一种不稳定的排序方法。

## 3.7 归并排序的实现

### 3.7.1 归并排序流程图



### 3.7.2 归并排序核心代码

//归并排序

void Sort::mergeSort()

{

int\* temp = new int[amount];

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

temp[i] = randomArray[i];

mergesort(temp, 0, amount - 1);

delete[] temp;

temp = NULL;

}

//递归进行划分

void Sort::mergesort(int\* L, int left, int right){

if (left >= right)return;

int mid = (left + right) / 2;

mergesort(L, left, mid);

mergesort(L, mid + 1, right);

merge(L, left, mid, right);

return;

}

//合并

void Sort::merge(int\* L, const int left, const int mid, const int right)

{

int s1 = left, s2 = mid + 1, t = left;//s1、s2检测指针，t存放指针

while (s1 <= mid && s2 <= right)

{

if (randomArray[s1] <= randomArray[s2])//两划分序列比较合并

L[t++] = randomArray[s1++];

else

L[t++] = randomArray[s2++];

compare\_degree++;

}

for (;s1 <= mid;t++, s1++)L[t] = randomArray[s1];

for (;s2 <= right;t++, s2++)L[t] = randomArray[s2];

for (int i = left; i <= right; i++)//将合并调整好的字段写入结果序列

{

randomArray[i] = L[i];

exchange\_degree++;

}

}

### 3.7.3 归并排序算法分析

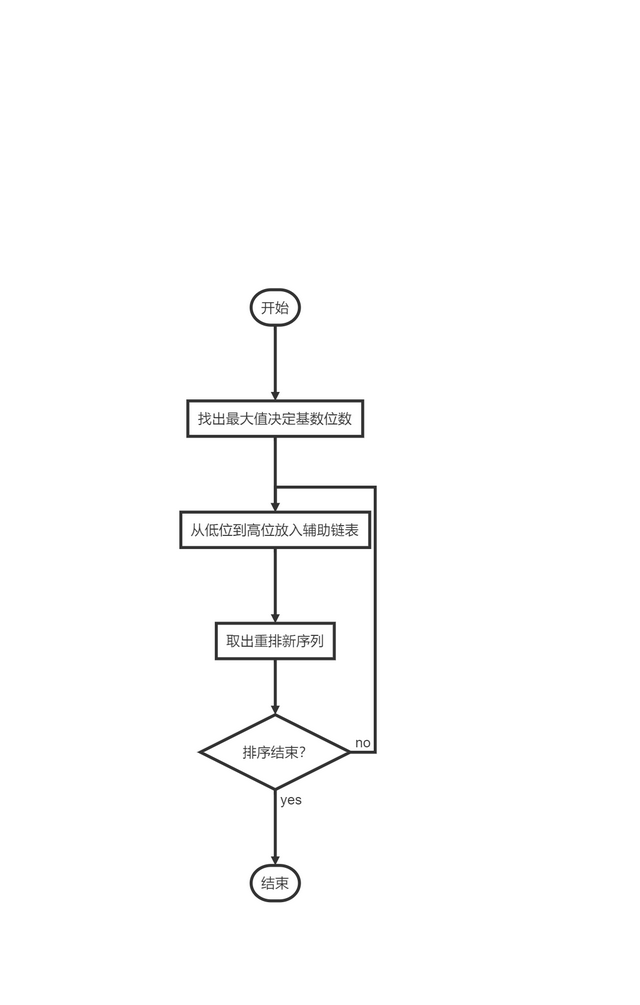
在归并排序算法中，递归深度为O(log2n)，对象关键字的比较次数为O(nlog2n)。算法总的时间复杂度为O(nlog2n)。

归并排序占用附加存储较多，需要另外一个与原待排序对象数组同样大小的辅助数组。这是这个算法的缺点。

特点：归并排序是一个稳定的排序方法。

## 3.8 基数排序的实现

### 3.8.1 基数排序流程图



### 3.8.2 基数排序核心代码

void Sort::radixSort(int left, int right) // 基数排序

{

int i, maxNum = left;

for (i = left + 1;i < right;i++)//找出最大值，用最大值位数决定基数

if (randomArray[i] > randomArray[maxNum])

maxNum = i;

int max = randomArray[maxNum];

queue<int> bucket[10];

for (int power = 1; max / power > 0; power \*= 10) {//从低位到高位处理

for (i = left; i < right; i++)//按每位0~9链入

bucket[(randomArray[i] / power) % 10].push(randomArray[i]);

int t = left;

for (i = 0;i < 10;i++) //从每个链依次取出组成新排序

while (bucket[i].empty() == false)

{

randomArray[t++] = bucket[i].front();

bucket[i].pop();

compare\_degree++;

exchange\_degree++;

}

}

return;

}

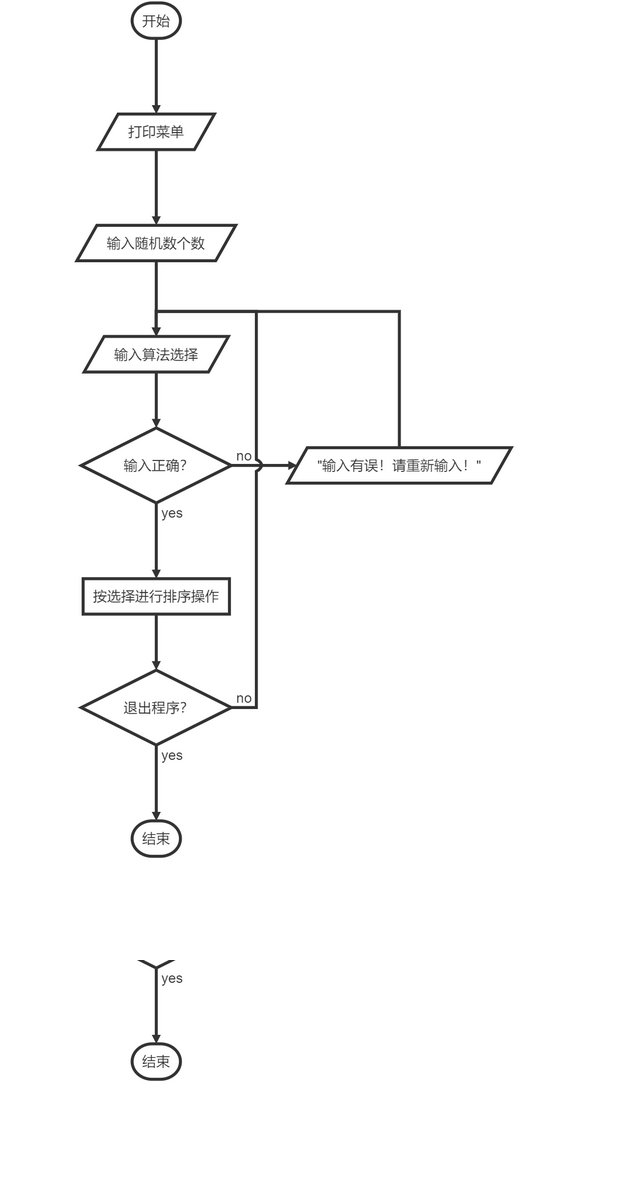
### 3.8.3 基数排序算法分析

基数排序的时间复杂度为O(2\*k\*n)，其中n是排序元素个数，k是数字位数；空间复杂度采用数组为O(n\*d)，采用链表是O(d+n)，d是进制（关键字的取值范围，根据最大值的位数确定）。

特点：基数排序是一个稳定的排序算法

## 3.9 总体系统的实现

### 3.9.1 总体系统流程图



### 3.9.2 总体系统核心代码

//打印选择菜单

void menu()

{

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'

<< "================================================" << endl;

cout << "请输入要产生的随机数个数：";

return;

}

//清零函数

void Sort::clear() { compare\_degree = 0;exchange\_degree = 0; }

//打印计数

oid Sort::print() {

cout << "该排序算法比较次数：" << compare\_degree << endl;

cout << "该排序算法交换次数：" << exchange\_degree << endl;

}

int main()

{

menu();

int type=0;

clock\_t startTime, endTime;

int amount;

while (1) {

cin >> amount;

if (cin.good())break;

else cout << "输入有误！请重新输入！";

cin.ignore(INT32\_MAX, '\n');

}

Sort T(amount);

while (type != 9){

cout << "\n请选择排序算法:";

cin >> type;

if (0 < type && type < 9)

{

T.create();

startTime = GetTickCount64();

}

switch (type)

{

case 1:T.bubbleSort();break;

case 2:T.selectionSort();break;

case 3:T.straightInsertion(0, amount);break;

case 4:T.shellSort(0, amount);break;

case 5:T.quickSort(0, amount);break;

case 6:T.heapSort(0, amount);break;

case 7:T.mergeSort();break;

case 8:T.radixSort(0, amount);break;

case 9:cout << "正在退出……" << endl;break;

default:cout << "操作输入错误！请重新输入！" << endl;

}

if (0 < type && type < 9)

{

endTime = GetTickCount64();

cout << "该排序算法所用时间：" << endTime - startTime << "ms" << endl;

T.print();

T.clear();

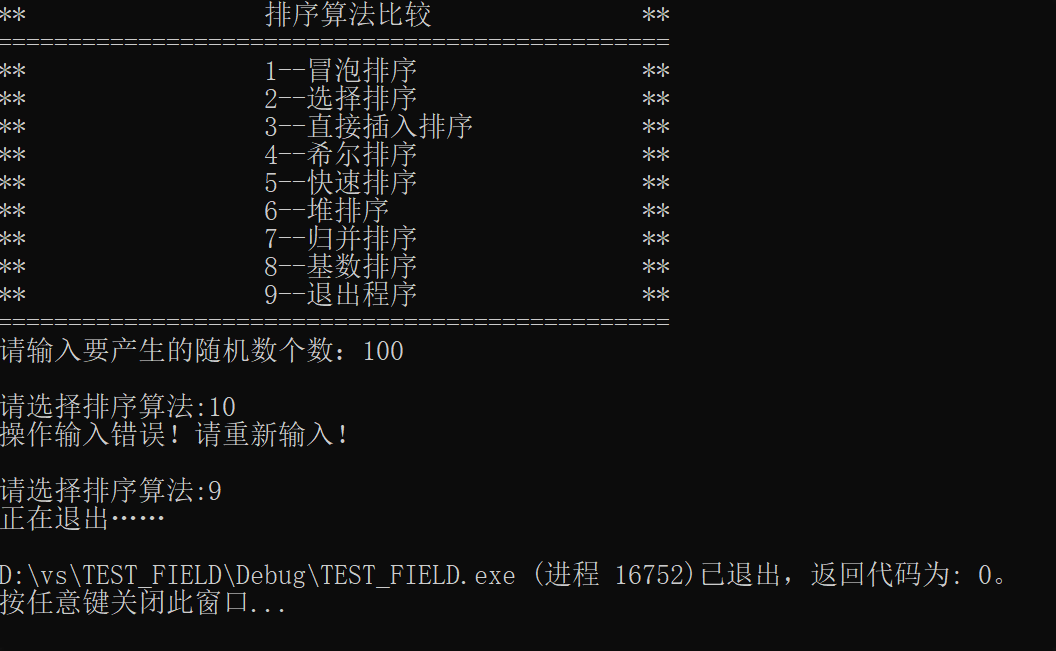
}

}

return 0;

}

### 3.9.3 总体系统截屏示例

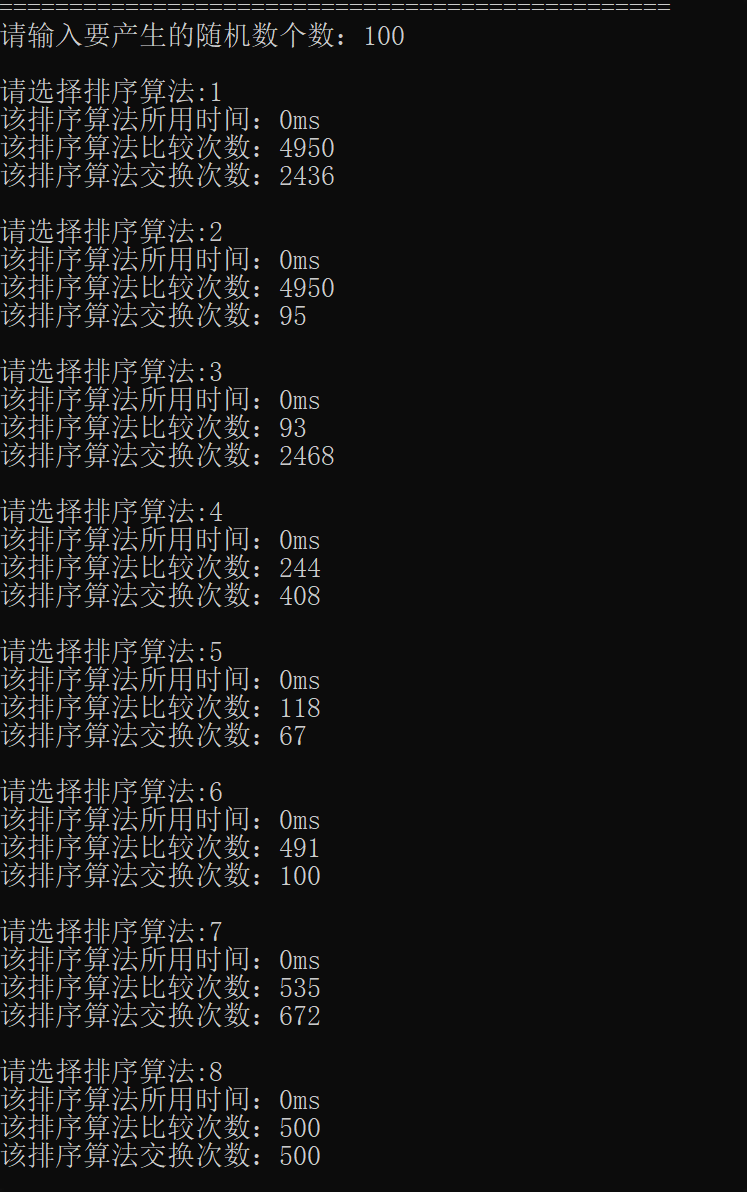


# 4 测试

## 4.1 功能测试

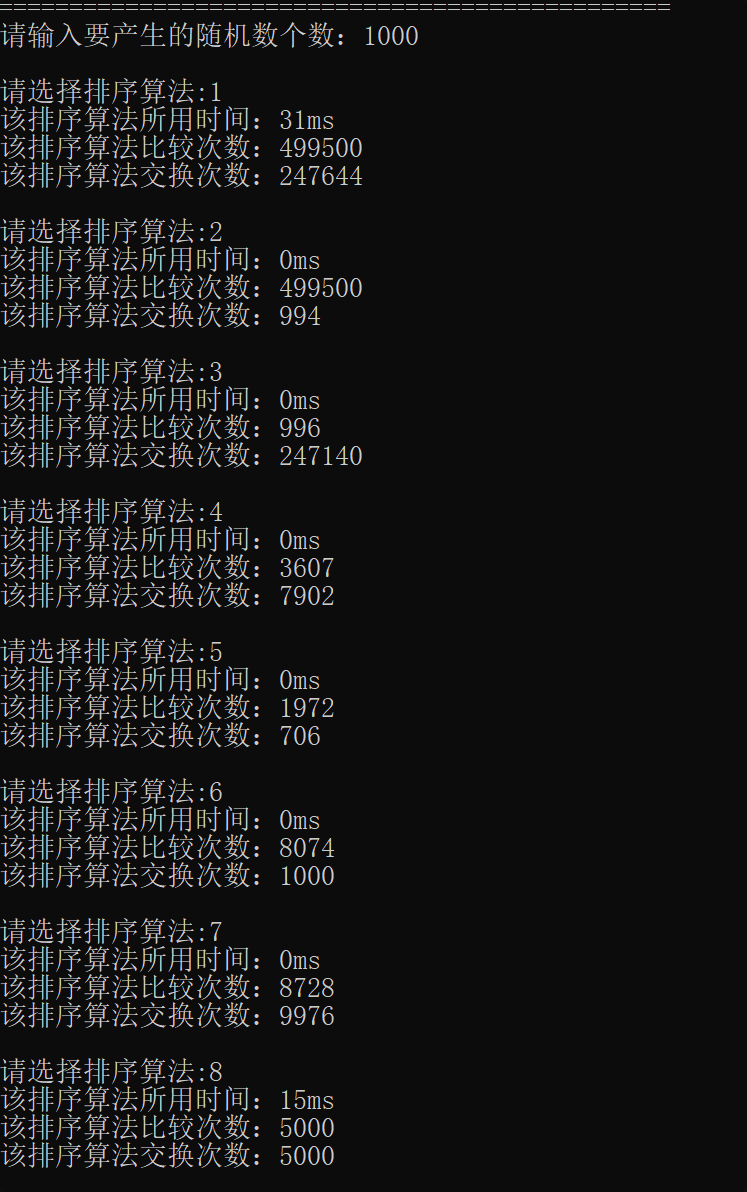
**测试用例：**100

**实验结果：**根据不同排序算法排序，输出所用时间、比较次数、交换次数



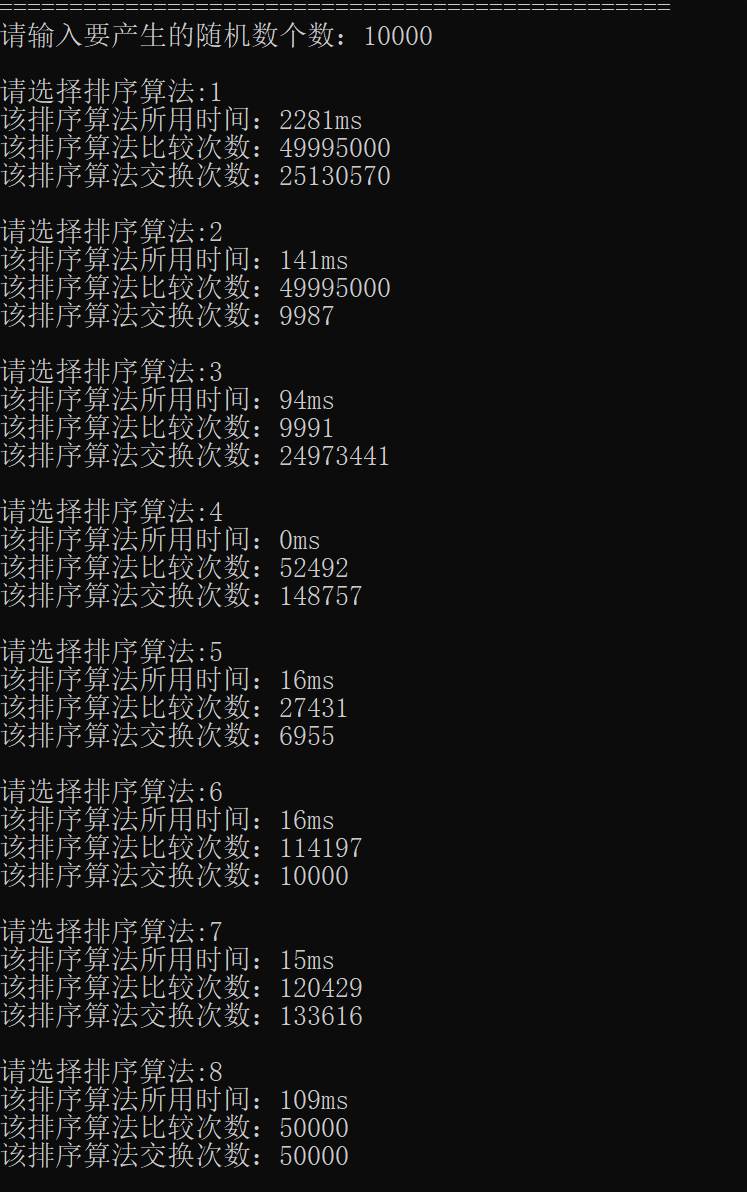
**测试用例：**1000

**实验结果：**根据不同排序算法排序，输出所用时间、比较次数、交换次数



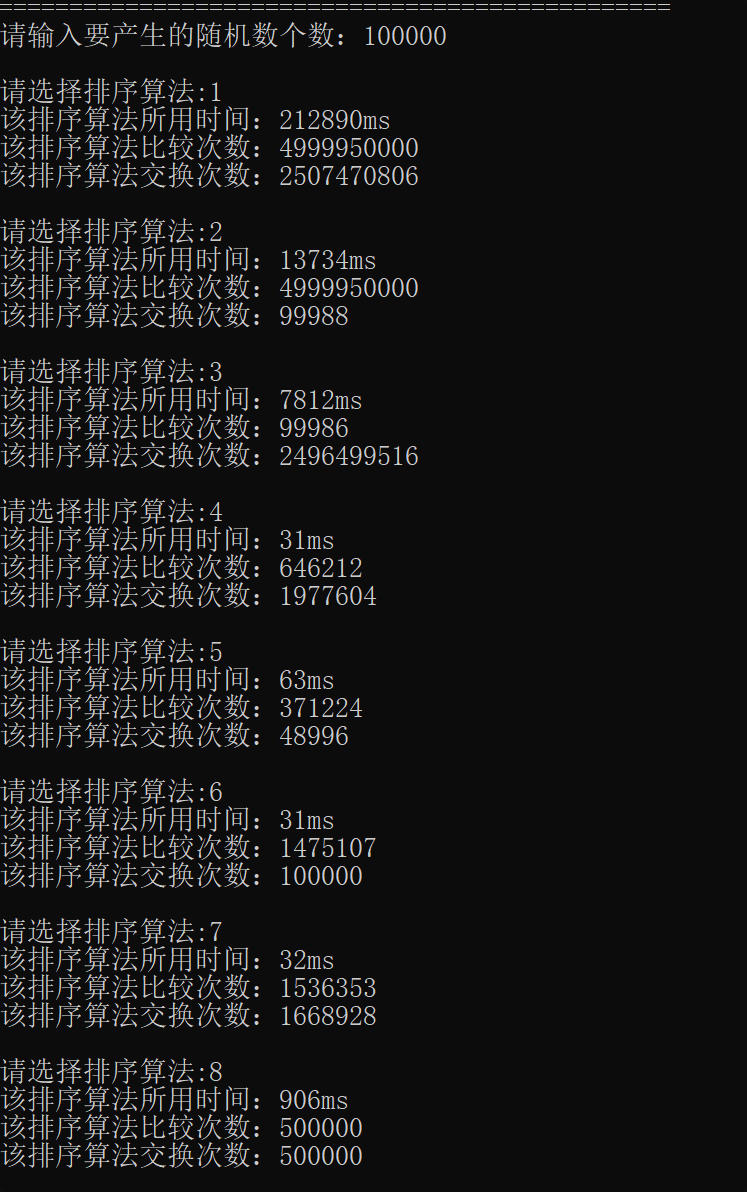
**测试用例：**10000

**实验结果：**根据不同排序算法排序，输出所用时间、比较次数、交换次数



**测试用例：**100000

**实验结果：**根据不同排序算法排序，输出所用时间、比较次数、交换次数

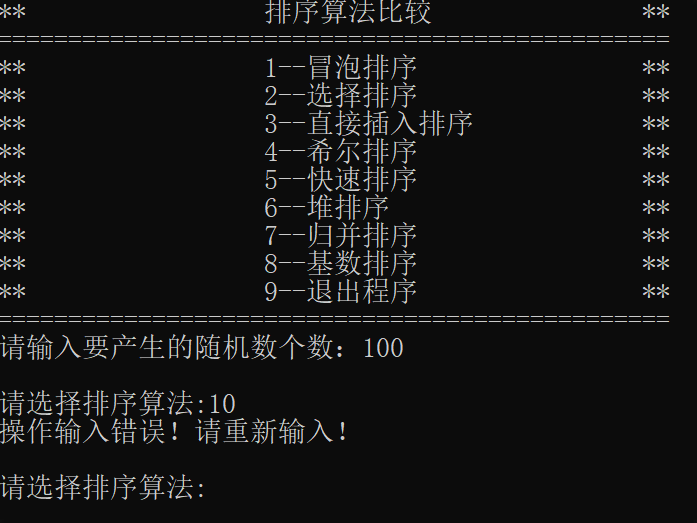


## 4.2 出错测试

**测试用例：**错误操作码

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

****

# 5 小结

