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： 王星洲

学 号： 1652977

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](#_Toc534302009)

[1.1 背景分析 1](#_Toc534302010)

[1.2 功能分析 1](#_Toc534302011)

[2 设计 1](#_Toc534302012)

[2.1 数据结构设计 1](#_Toc534302013)

[2.2 类结构设计 2](#_Toc534302014)

[2.3 成员与操作设计 2](#_Toc534302015)

[2.4 系统设计 3](#_Toc534302016)

[3 实现 4](#_Toc534302017)

[3.1 改进后的起泡排序功能的实现 4](#_Toc534302018)

[3.1.1 起泡排序核心代码 4](#_Toc534302019)

[3.1.2 起泡排序说明 4](#_Toc534302020)

[3.1.3 起泡排序截屏示例 5](#_Toc534302021)

[3.2 选择排序功能的实现 6](#_Toc534302022)

[3.2.1 选择排序功能核心代码 6](#_Toc534302023)

[3.2.2 选择排序功能说明 7](#_Toc534302024)

[3.2.3 选择排序功能截屏示例 7](#_Toc534302025)

[3.3 直接插入排序功能的实现 8](#_Toc534302026)

[3.3.1 直接插入排序功能核心代码 8](#_Toc534302027)

[3.3.2 直接插入排序功能说明 9](#_Toc534302028)

[3.3.3 直接插入排序功能截屏示例 9](#_Toc534302029)

[3.4 希尔排序功能的实现 10](#_Toc534302030)

[3.4.1 希尔排序核心代码 10](#_Toc534302031)

[3.4.2 希尔排序功能说明 11](#_Toc534302032)

[3.4.3 希尔排序功能截图示例 12](#_Toc534302033)

[3.5 快速排序功能的实现 13](#_Toc534302034)

[3.5.1 快速排序功能核心代码 13](#_Toc534302035)

[3.5.2 快速排序功能说明 14](#_Toc534302036)

[3.5.3 快速排序功能截屏示例 14](#_Toc534302037)

[3.6 堆排序功能的实现 15](#_Toc534302038)

[3.6.1 堆排序功能核心代码 15](#_Toc534302039)

[3.6.2 堆排序功能说明 16](#_Toc534302040)

[3.6.3 堆排序功能截屏示例 16](#_Toc534302041)

[3.7 归并排序功能的实现 17](#_Toc534302042)

[3.7.1 归并排序核心代码 17](#_Toc534302043)

[3.7.2 归并排序功能说明 18](#_Toc534302044)

[3.7.3 归并排序功能截屏示例 19](#_Toc534302045)

[3.8 基数排序功能的实现 20](#_Toc534302046)

[3.8.1 基数排序核心代码 20](#_Toc534302047)

[3.8.2 基数排序说明 21](#_Toc534302048)

[3.8.3 基数排序截屏示例 22](#_Toc534302049)

[3.8 随机数生成方法 23](#_Toc534302050)

[3.8.1 随机数生成代码 23](#_Toc534302051)

[3.8.2 随机数生成说明 24](#_Toc534302052)

[3.8.3 随机数生成截屏示例 25](#_Toc534302053)

[3.9 总体系统实现 25](#_Toc534302054)

[3.9.1 总体系统代码 25](#_Toc534302055)

[3.9.2 总体系统说明 27](#_Toc534302056)

[4 测试 28](#_Toc534302057)

[4.1 功能测试 28](#_Toc534302058)

[4.1.1 起泡排序功能测试 28](#_Toc534302059)

[4.1.2 选择排序功能测试 28](#_Toc534302060)

[4.1.3 直接插入排序功能测试 29](#_Toc534302061)

[4.1.4 希尔排序功能测试 30](#_Toc534302062)

[4.1.5 快速排序功能测试 30](#_Toc534302063)

[4.1.6 堆排序功能测试 31](#_Toc534302064)

[4.1.7 归并排序功能测试 32](#_Toc534302065)

[4.1.8 基数排序功能测试 33](#_Toc534302066)

[4.1.9 退出系统测试 33](#_Toc534302067)

[4.2 边界测试 34](#_Toc534302068)

[4.2.1 只有一个随机数进行排序 34](#_Toc534302069)

[4.2.2 100000个随机数 35](#_Toc534302070)

[4.3 出错测试 36](#_Toc534302071)

[4.3.1 随机数个数错误 36](#_Toc534302072)

[4.3.2 随机数越界错误 37](#_Toc534302073)

[4.3.3 指令错误 37](#_Toc534302074)

[5 亮点 38](#_Toc534302075)

[5.1 人机交互 38](#_Toc534302076)

[5.2 最大堆和静态链表的使用 38](#_Toc534302077)

[5.3 计时精确性 38](#_Toc534302078)

[6. 不同情况下几种排序方法的效率及比较 39](#_Toc534302079)

[6.1 100个数据 39](#_Toc534302080)

[6.2 1000个数据 41](#_Toc534302081)

[6.3 10000个数据 43](#_Toc534302082)

[6.4 100000个数据 45](#_Toc534302083)

[6.5 综述 45](#_Toc534302084)

# 1 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

内部排序方式有很多，主要有起泡，插入，选择， 归并，快速排序，分配排序等大类。每类中又有若干小类，本题中主要使用改进后的起泡算法，直接插入排序，希尔排序，直接选择排序，堆排序，快速排序，归并排序，LSD基数排序8种排序方式。比较它们的利弊。总体思路为接收用户输入随机数个数，系统产生随机数，之后接收指令进行排序，反馈排序中比较次数和消耗时间。

# 2 设计

## 2.1 数据结构设计

如上功能分析所述，该系统要求一个接收用户输入的数字作为接下来生成随机数的数量，我是用一个全局变量int number表示，之后需要一个接收用户指令的order，用char类型即可。之后是生成随机数，需要一个存储随机数的数组vector<int> original，生成的随机数不进行范围化处理，默认为0~32768之间的整数，所以int足够使用。接下来起泡排序，直接插入排序，直接选择排序，希尔排序，归并排序，快速排序不需要额外创建数据类型，而堆排序需要使用最大堆，因此创建一个最大堆MaxHeap，基数排序需要用到静态链表，因此创建一个staticList。统计时间时，为了保证精确度，使用了float类型统计到毫秒，统计交换次数时，100000个数据有10亿级次数，因此使用long long类型存储，这样就可以解决以上所有问题。

## 2.2 类结构设计

本题中主要创建的类为MaxHeap和StaticList，这里对这两个类进行介绍。最大堆不需要辅助类，只要自己本身即可解决问题，而StaticList类需要一个SLinkNode来表示它的结点，其中包括data和link两个部分，这里使用Struct来表示结点，并作为StaticList的私有成员。

## 2.3 成员与操作设计

class MaxHeap { // K为关键码的数据类型，int为记录的结构类型

private:

int \*heap; //存放最大堆中元素的数组

int currentSize; //最大堆中当前元素个数

int maxHeapSize; //最大堆最多允许元素个数

public:

MaxHeap(int sz = DefaultSize);//构造函数：建立空堆

MaxHeap(vector<int> arr, int n); //构造函数：通过一个数组建堆

~MaxHeap();

bool Insert(const int &x);

bool RemoveMax(int &x);

bool IsEmpty()const;

bool IsFull()const;

void MakeEmpty();

void Swap(int i, int j);

void output();

void siftDown(int start, int m);//从start到m下滑调整成为最大堆

void siftDown(int start, int m, long long& swapTimes);//从start到m下滑调整成为最大堆

void siftUp(int start); //从start到0上滑调整成为最大堆

};

struct SLinkNode

{

int data;

int link; //结点链接指针

};

class StaticList {

private:

SLinkNode elem[maxSize];

int m\_length;

int avil; //当前可分配空间首地址

public:

void InitList();

//计算静态链表的长度

int length();

//在静态链表中查找具有给定值的结点

int Search(int x);

//在静态链表中查找第i个结点

int Locate(int i);

//在静态链表的表尾追加一个新结点

bool Append(int x);

//在静态链表的第i个节点后插入新结点

bool Insert(int i, int x);

//在静态链表中释放第i个结点

bool Remove(int i);

//判空

bool IsEmpty();

//更改linkNode

void setLink(const int location, const int newLink);

//返回结点data

int getNode(const int location);

//返回结点link

int getLink(const int location);

//输出

void print();

};

## 2.4 系统设计

系统首先接收用户的输入number，创建一个初始随机数数组，再通过循环和选择语句接收用户的指令order，进行对应的排序，返回时间和交换次数，接收到退出指令时退出系统。

# 3 实现

## 3.1 改进后的起泡排序功能的实现

### 3.1.1 起泡排序核心代码

//起泡排序

void BubbleSort(vector<int>& original)

{

bool exchange;

int i, j;

//记录交换次数

long long stepNumber = 0;

//记录时间

clock\_t start, end;

start = clock();

for (i = 1; i < original.size(); i++)

{

exchange = false;

for (j = original.size() - 1; j >= i; j--)

{

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

{

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

stepNumber++;

exchange = true;

}

}

if (exchange == false)

break;

}

end = clock();

cout << "冒泡排序所用时间:" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "冒泡排序交换次数:" << '\t' << stepNumber << endl;

}

### 3.1.2 起泡排序说明

设待排序元素序列中的元素个数为n，首先比较第n-2个元素和n-1个元素，如果发生逆序，则将这两个元素交换，然后对前面的元素依次处理，重复此过程直到处理完第0和1个元素，这样就完成了一趟起泡，结果将最小的元素交换到待排序列的第一个位置，其他元素也完成了向排序最终序列的移动，这样最多进行n-1趟起泡就可以将所有元素排好序，本题中我使用的改进式起泡排序在起泡算法的基础上使用一个标志exchange，标识本趟起泡结果是否发生了逆序和交换，如果没有发生交换，则exchange=false，这种情况下可以判断所有元素已经排好了序，这样就不用执行之后的几趟起泡了。

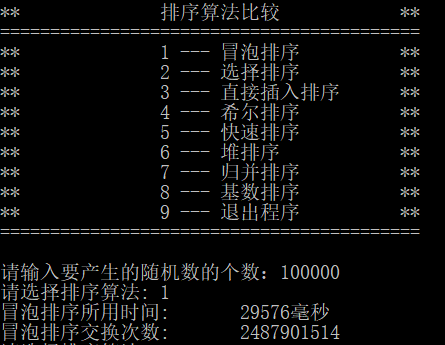
计算时间和计数的代码都在函数中完成计算和输出。

我在代码中加入了已经注释掉的输出结果语句，以下图片中有一项测试示例为输出结果以检验排序的准确性。

起泡排序是一种稳定的排序方式。

### 3.1.3 起泡排序截屏示例





## 3.2 选择排序功能的实现

### 3.2.1 选择排序功能核心代码

//选择排序

void SelectSort(vector<int>& original)

{

long long stepNumber = 0;

clock\_t start, end;

start = clock();

for (int i = 0; i < original.size() - 1; i++)

{

int k = i;

for (int j = i + 1; j < original.size() - 1; j++)

{

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

{

k = j;

}

}

if (k != i)

{

swap(original[i], original[k]);

stepNumber++;

}

}

end = clock();

cout << "选择排序所用时间:" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "选择排序交换次数:" << '\t' << stepNumber << endl;

}

### 3.2.2 选择排序功能说明

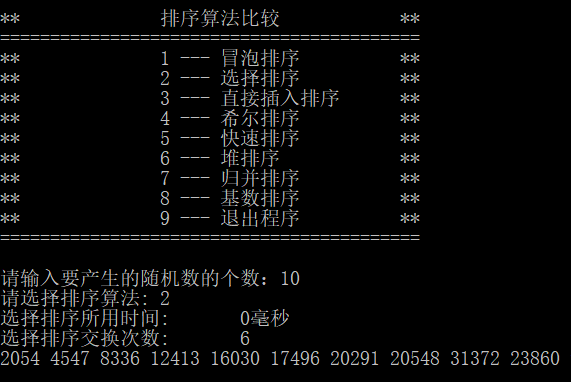
选择排序的基本思想是：从待排序数组中取排序码最小的元素，并添加到已排序列末尾，等到第n-2趟做完，就不用再选了。其中直接选择排序基本步骤是：在一组元素中选择最小排序码的元素，若他不是第一个元素，则将这两个元素对调，在剩下的元素里重复执行以上步骤，直到剩余元素只有一个为止。

计算时间和计数的代码都在函数中完成计算和输出。

我在代码中加入了已经注释掉的输出结果语句，以下图片中有一项测试示例为输出结果以检验排序的准确性。

选择排序是一种稳定的排序方式。

### 3.2.3 选择排序功能截屏示例





## 3.3 直接插入排序功能的实现

### 3.3.1 直接插入排序功能核心代码

//直接插入排序

void InsertSort(vector<int>& original)

{

long long stepNumber = 0;

clock\_t start, end;

int i, j, temp;

start = clock();

for (i = 1; i <= original.size() - 1; i++)

{

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

{

temp = original[i];

j = i - 1;

do

{

original[j + 1] = original[j];

j--;

stepNumber++;

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

original[j + 1] = temp;

stepNumber++;

}

}

end = clock();

cout << "直接插入排序所用时间:" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "直接插入排序交换次数:" << '\t' << stepNumber << endl;

}

### 3.3.2 直接插入排序功能说明

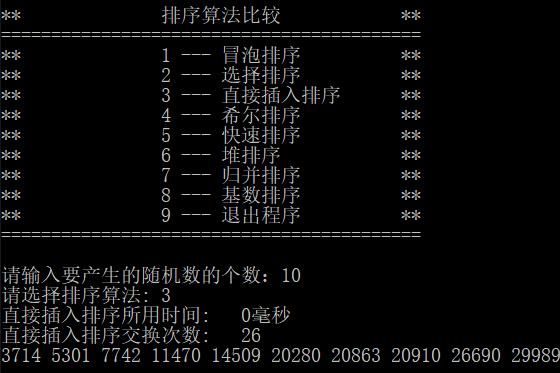
插入排序的基本方法是，每步将一个待排序的元素，按其排序码的大小，插入到前面已经排好序的一组元素的适当位置上去，直到所有元素全部插入为止。其中，直接插入排序当插入第i个元素时，前面的i-1个元素已经排好了序，这时，将这个排序码与前面的数依次比较，直到找到自己的插入位置，将其插入，它之后原来位置上的元素依次后移。

计算时间和计数的代码都在函数中完成计算和输出。

我在代码中加入了已经注释掉的输出结果语句，以下图片中有一项测试示例为输出结果以检验排序的准确性。

直接插入排序是一种稳定的排序方式。

### 3.3.3 直接插入排序功能截屏示例





## 3.4 希尔排序功能的实现

### 3.4.1 希尔排序核心代码

//希尔排序

void ShellSort(vector<int>& original)

{

int i, j, gap = original.size();

int temp;

long long stepNumber = 0;

clock\_t start, end;

start = clock();

do {

gap = gap / 3 + 1;

for (i = gap; i < original.size(); i++)

{

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

{

temp = original[i];

j = i - gap;

do {

original[j + gap] = original[j];

j = j - gap;

stepNumber++;

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

original[j + gap] = temp;

}

}

} while (gap > 1);

end = clock();

cout << "希尔排序所用时间:" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "希尔排序交换次数:" << '\t' << stepNumber << endl;

}

### 3.4.2 希尔排序功能说明

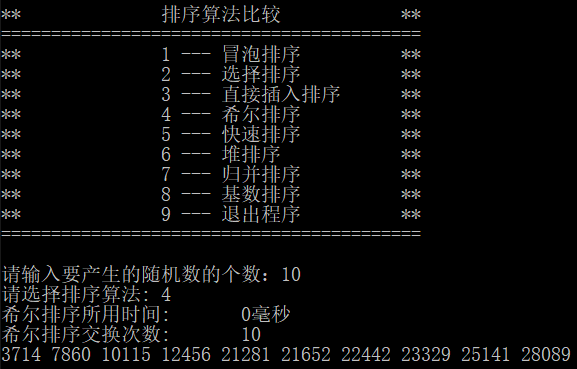
希尔排序又称缩小增量排序，该方法的基本思想是：设待排元素序列有n个元素，首先取一个整数gap<n为间隔，将所有元素分为gap个子序列，所有距离为gap的元素放在同一个子序列中，在每一个子序列中实施直接插入排序。然后缩小间隔gap，本题中取gap=gap/3+1，这样使得整个元素序列达到基本有序，排序速度很快，整体次数少于直接插入排序。

计算时间和计数的代码都在函数中完成计算和输出。

我在代码中加入了已经注释掉的输出结果语句，以下图片中有一项测试示例为输出结果以检验排序的准确性。

希尔排序是一种不稳定的排序方式。

### 3.4.3 希尔排序功能截图示例





## 3.5 快速排序功能的实现

### 3.5.1 快速排序功能核心代码

//快排划分函数

int Partition(vector<int>& original, const int low, const int high, long long & stepNumber)

{

int pivotpos = low;

int pivot = original[low];

for (int i = low + 1; i <= high; i++)

{

if (original[i] < pivot)

{

pivotpos++;

if (pivotpos != i)

{

swap(original[pivotpos], original[i]);

stepNumber++;

}

}

}

original[low] = original[pivotpos];

original[pivotpos] = pivot;

stepNumber++;

return pivotpos;

}

//快速排序

void QuickSort(vector<int>& original, const int left, const int right, long long &stepNumber)

{

if (left < right)

{

int pivotpos = Partition(original, left, right, stepNumber);

QuickSort(original, left, pivotpos - 1, stepNumber);

QuickSort(original, pivotpos + 1, right, stepNumber);

}

}

//主函数中使用

case '5':

{

long long stepNumber = 0;

clock\_t start, end;

start = clock();

QuickSort(original1, 0, original1.size() - 1, stepNumber);

end = clock();

cout << "快速排序所用时间:" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "快速排序交换次数:" << '\t' << stepNumber << endl;

break;

}

### 3.5.2 快速排序功能说明

快速排序是一种划分交换的方法，它采用分治法进行排序，其基本思想是任取待排序元素序列中的某个元素作为基准（本代码中默认第一个），按照该基准将元素序列分为左右两个子序列：左侧序列中所有元素小于该基准的排序码，右侧子序列中所有元素大于等于该排序码，基准元素排在这两个子序列中间，然后分别对这两个子序列重复实行上述方法，直到所有元素排在相应位置上为止。

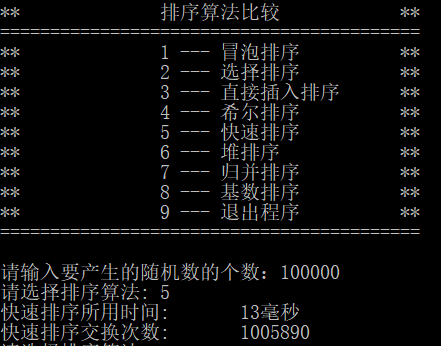
由于函数是递归调用的，所以不能直接输出消耗的时间和交换次数，因此对应的代码在主函数的对应case中，而函数需要添加int& stepNumber参数进行引用传递获得对应的交换次数。

我在代码中加入了已经注释掉的输出结果语句，以下图片中有一项测试示例为输出结果以检验排序的准确性。

快速排序是一种不稳定的排序方式。

### 3.5.3 快速排序功能截屏示例





## 3.6 堆排序功能的实现

### 3.6.1 堆排序功能核心代码

//堆排序

void HeapSort(MaxHeap& a, int currentSize)

{

long long stepNumber = 0;

clock\_t start, end;

start = clock();

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

{

a.siftDown(i, currentSize - 1, stepNumber);

}

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

{

a.Swap(0, i);

a.siftDown(0, i - 1, stepNumber);

}

end = clock();

cout << "堆排序所用时间：" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "堆排序交换次数：" << '\t' << stepNumber << endl;

}

//主函数中使用

case '6':

{

MaxHeap a = MaxHeap(original, number);

HeapSort(a, number);

break;

}

### 3.6.2 堆排序功能说明

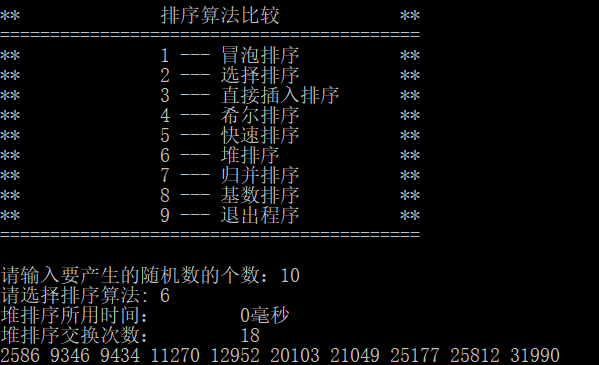
堆排序是一种特殊的选择排序，首先根据输入的初始数据生成一个初始最大堆，接下来将最大堆的堆顶与堆中的最后一个元素对调，然后将除最后一个元素以外的元素重新调整为最大堆，再交换堆顶堆尾的元素，直至堆中只剩最后一个元素为止。这样整个堆中的数据就满足了从小到大有序排列。

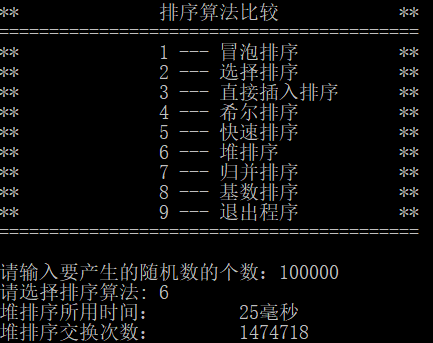
计算时间和计数的代码都在函数中完成计算和输出。

我在代码中加入了已经注释掉的输出结果语句，以下图片中有一项测试示例为输出结果以检验排序的准确性。

堆排序是一种不稳定的排序方式。

### 3.6.3 堆排序功能截屏示例





## 3.7 归并排序功能的实现

### 3.7.1 归并排序核心代码

//二路归并算法

void merge(vector<int> &LeftSubArray, vector<int> &RightSubArray,const int left,const int mid,const int right, long long& stepNumber)

{

for (int k = left; k <= right; k++)

{

RightSubArray[k] = LeftSubArray[k];

}

int s1 = left, s2 = mid + 1, t = left;

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

{

if (RightSubArray[s1] <= RightSubArray[s2])

{

LeftSubArray[t++] = RightSubArray[s1++];

stepNumber++;

}

else

{

LeftSubArray[t++] = RightSubArray[s2++];

stepNumber++;

}

}

while (s1 <= mid)

{

LeftSubArray[t++] = RightSubArray[s1++];

stepNumber++;

}

while (s2 <= right)

{

LeftSubArray[t++] = RightSubArray[s2++];

stepNumber++;

}

}

//归并排序

void MergeSort(vector<int> &leftSubArray, vector<int> &rightSubArray, int front, int end, long long& stepNumber) {

if (front >= end)

return;

int mid = (front + end) / 2;

MergeSort(leftSubArray, rightSubArray, front, mid, stepNumber);

MergeSort(leftSubArray, rightSubArray, mid + 1, end, stepNumber);

merge(leftSubArray, rightSubArray, front, mid, end, stepNumber);

}

//归并排序主函数中使用

case '7':

{

vector<int> temp(number);

long long stepNumber = 0;

clock\_t start, end;

start = clock();

MergeSort(original1, temp, 0, original1.size()-1, stepNumber);

end = clock();

/\*for (int i = 0; i < original.size(); i++)

{

cout << original1[i] << " ";

}\*/

cout << "归并排序所用时间:" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "归并排序交换次数:" << '\t' << stepNumber << endl;

break;

}

### 3.7.2 归并排序功能说明

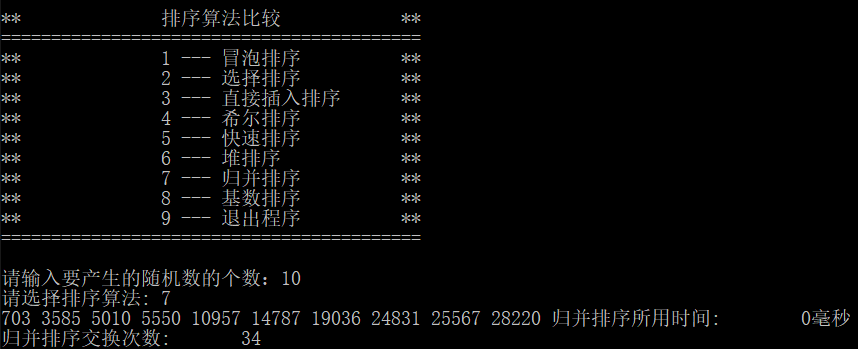
归并排序也是基于分治法的，先将所有待排元素分为两个长度相等的子序列，为每一个子序列排序，然后使用两路归并将它们合并为一个序列。归并排序的算法先是调用一个划分过程直到子序列为空或者只有一个元素，然后再对每两个序列进行归并成为长度为2的子序列，再归并为长度为4的子序列，以此类推，直到整个序列有序为止。

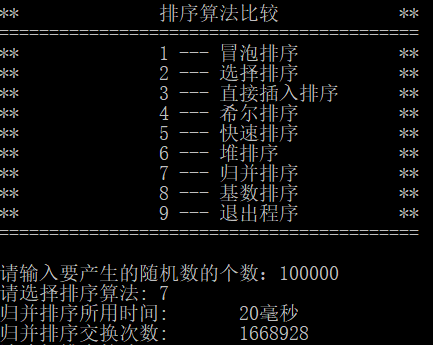
由于函数是递归调用的，所以不能直接输出消耗的时间和交换次数，因此对应的代码在主函数的对应case中，而函数需要添加int& stepNumber参数进行引用传递获得对应的交换次数。

我在代码中加入了已经注释掉的输出结果语句，以下图片中有一项测试示例为输出结果以检验排序的准确性。

归并排序是一种稳定的排序方式。

### 3.7.3 归并排序功能截屏示例





## 3.8 基数排序功能的实现

### 3.8.1 基数排序核心代码

//获取对应位数上的数

int getDigit(int number, int digit)

{

int result;

for (int i = 0; i < (5 - digit); i++)

{

number /= radix;

}

result = number % radix;

return result;

}

//基数排序

void RadixSort(StaticList& L, int d)

{

long long stepNumber = 0;

clock\_t start, end;

start = clock();

int rear[radix], front[radix];

int i, j, k, last, current, n = L.length();

L.setLink(n, 0);

current = 1;

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

{

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

{

front[j] = 0;

}

while (current != 0)

{

k = getDigit(L.getNode(current), i);

if (front[k] == 0)

{

front[k] = current;

}

else

{

L.setLink(rear[k], current);

}

rear[k] = current;

current = L.getLink(current);

}

j = 0;

while (front[j] == 0)

{

j++;

}

L.setLink(0, front[j]);

current = front[j];

last = rear[j];

for (k = j + 1; k < radix; k++)

{

if (front[k] != 0) {

L.setLink(last, front[k]);

last = rear[k];

}

}

L.setLink(last, 0);

}

end = clock();

cout << "基数排序所用时间:" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "基数排序交换次数:" << '\t' << stepNumber << endl;

}

//主函数中使用

case '8':

{

StaticList a;

a.InitList();

for (int i = 1; i <= original1.size(); i++)

{

a.Append(original1[i - 1]);

}

RadixSort(a, 5);

/\*a.print();\*/

break;

}

### 3.8.2 基数排序说明

基数排序主要用来应对多排序码排序，而对于整数来说，它的每一位数字就可以作为排序码，本题我选择使用低位优先的基数排序LSD基数排序法，它是种自底向上的基数排序方法，它将每一位数字，作为单排序码k，利用分配与收集两种运算对单排序码进行排序。而桶有radix（本题为10）个，所有元素被分配到了对应桶中，然后按桶的编号依次把桶中元素搜集起来，这样所有元素就排序完成了。

在排序过程中为了有效存储和重拍待排序元素，我们以静态链表作为存储结构，在元素重排时就不必移动元素，只要更改link指针即可。

计算时间和计数的代码都在函数中完成计算和输出。

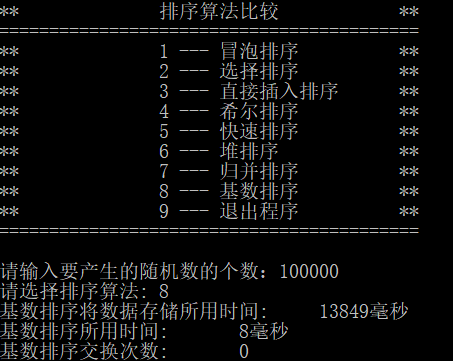
值得提出的是，当数据量变大时，静态链表添加数据以形成初始静态链表的效率并不理想，100000个数据达到了10秒以上的耗时，于是我在代码中添加了对于创建静态链表耗时的检测，可见示例。

我在代码中加入了已经注释掉的输出结果语句，以下图片中有一项测试示例为输出结果以检验排序的准确性。

基数排序是一种稳定的排序方式。

### 3.8.3 基数排序截屏示例





## 3.8 随机数生成方法

### 3.8.1 随机数生成代码

void setOrigin(vector<int>& original)

{

cout << "请输入要产生的随机数的个数：";

while (1)

{

if (!(cin >> number))

{

cout << "输入非法，请重新输入" << endl;

cin.clear();

while (cin.get() != '\n')

{

continue;

}

}

else

{

if (number > 100000)

{

cout << "请不要输入超过100000的数字" << endl;

}

else if (number <= 0)

{

cout << "请输入一个正整数" << endl;

}

else

{

break;

}

}

}

srand(time(NULL));

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

{

original.push\_back(rand());

}

/\*for (int i = 0; i < original.size(); i++)

{

cout << original[i] << " ";

}\*/

}

### 3.8.2 随机数生成说明

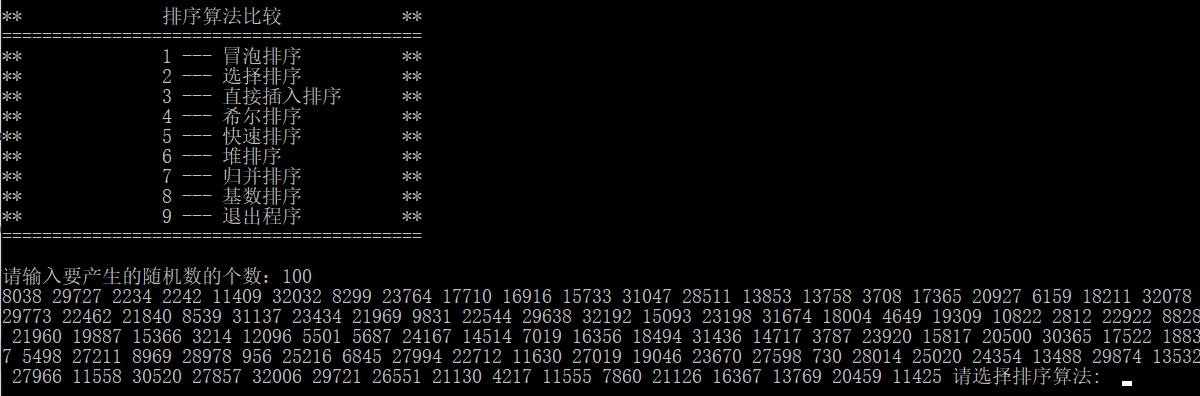
随机数生成的环节需要接收用户输入，因此我加入了大量人机交互用代码，保证程序的健壮性。

在生成随机数时，需要使用rand（）函数，这个函数可以生成0~32768的随机数，为了使每一次生成的随机数不同，需要用srand（time（NULL））来清空随机数种子，得到随机数后，让它加入original的数组即可。

我在代码中加入了已经注释掉的输出结果语句，以下图片中有一项测试示例为输出结果以检验生成随机数的准确性。

### 3.8.3 随机数生成截屏示例





## 3.9 总体系统实现

### 3.9.1 总体系统代码

int main()

{

char order;

showMenu();

vector<int> original;

setOrigin(original);

vector<int> original1 = original;

while (1)

{

original1 = original;

cout << "请选择排序算法:" << '\t';

cin >> order;

switch (order)

{

case '1':

BubbleSort(original1);

break;

case '2':

SelectSort(original1);

break;

case '3':

InsertSort(original1);

break;

case '4':

ShellSort(original1);

break;

case '5':

{

long long stepNumber = 0;

clock\_t start, end;

start = clock();

QuickSort(original1, 0, original1.size() - 1, stepNumber);

end = clock();

cout << "快速排序所用时间:" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "快速排序交换次数:" << '\t' << stepNumber << endl;

/\*for (int i = 0; i < original.size(); i++)

{

cout << original1[i] << " ";

}\*/

break;

}

case '6':

{

MaxHeap a = MaxHeap(original, number);

HeapSort(a, number);

break;

}

case '7':

{

vector<int> temp(number);

long long stepNumber = 0;

clock\_t start, end;

start = clock();

MergeSort(original1, temp, 0, original1.size()-1, stepNumber);

end = clock();

/\*for (int i = 0; i < original.size(); i++)

{

cout << original1[i] << " ";

}\*/

cout << "归并排序所用时间:" << '\t' << (float)(end - start) \* 1000 / CLOCKS\_PER\_SEC << "毫秒" << endl;

cout << "归并排序交换次数:" << '\t' << stepNumber << endl;

break;

}

case '8':

{

StaticList a;

a.InitList();

for (int i = 1; i <= original1.size(); i++)

{

a.Append(original1[i - 1]);

}

RadixSort(a, 5);

break;

}

case '9':

cout << "退出系统成功。" << endl;

system("pause");

return 0;

default:

cout << "输入的指令有误，请重新输入。" << endl;

break;

}

}

system("pause");

return 0;

}

### 3.9.2 总体系统说明

总体系统结构简单明了便于理解，由于需要接收用户输入，所以我使用swich语句来对用户的输入进行判断处理。其中大部分排序的处理都可以直接解决，比较值得提出的是快速排序算法，归并排序算法的函数是递归调用的，所以需要在主函数中得到时间和交换次数，而堆排序，基数排序需要将数组元素转存到对应的堆/静态链表中，所以主函数中包含一步数据处理。其他代码基本实现了模块化。

# 4 测试

## 4.1 功能测试

### 4.1.1 起泡排序功能测试

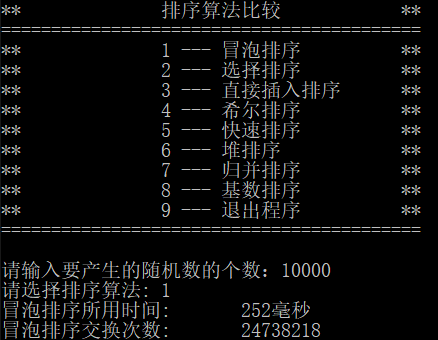
**测试用例**：

10000个数值

**预期结果**：

获得测试时间和交换次数

**实验结果**



### 4.1.2 选择排序功能测试

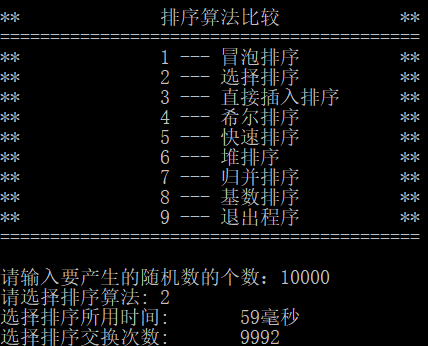
**测试用例：**

10000个数值

**预期结果：**

获得测试时间和交换次数

**实验结果：**



### 4.1.3 直接插入排序功能测试

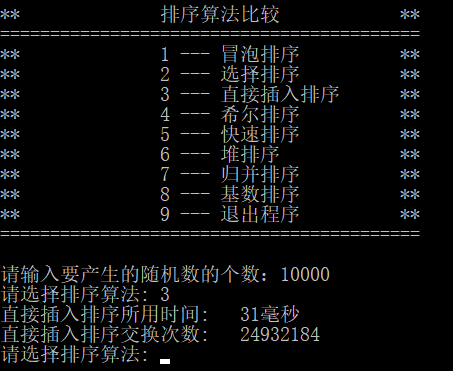
**测试用例：**

10000个数值

**预期结果：**

获得测试时间和交换次数

**实验结果：**



### 4.1.4 希尔排序功能测试

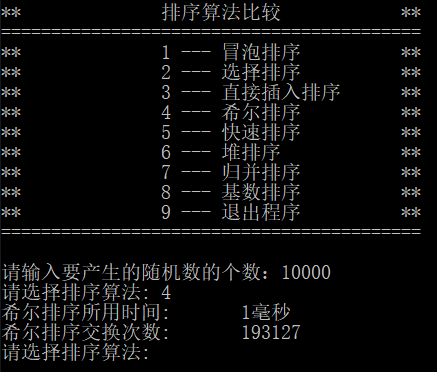
**测试用例：**

10000个数值

**预期结果：**

获得测试时间和交换次数

**实验结果：**



### 4.1.5 快速排序功能测试

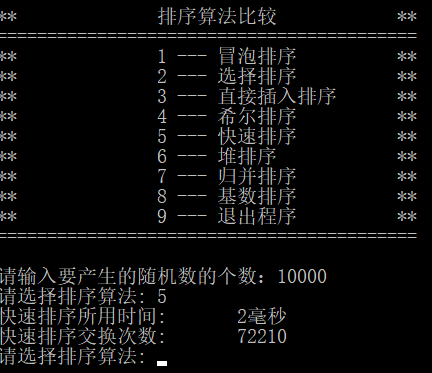
**测试用例：**

10000个数值

**预期结果：**

获得测试时间和交换次数

**实验结果：**



### 4.1.6 堆排序功能测试

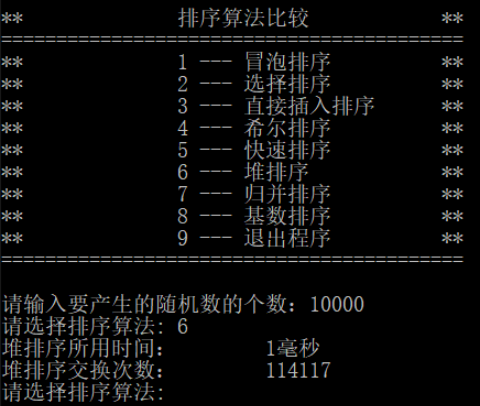
**测试用例：**

10000个数值

**预期结果：**

获得测试时间和交换次数

**实验结果：**



### 4.1.7 归并排序功能测试

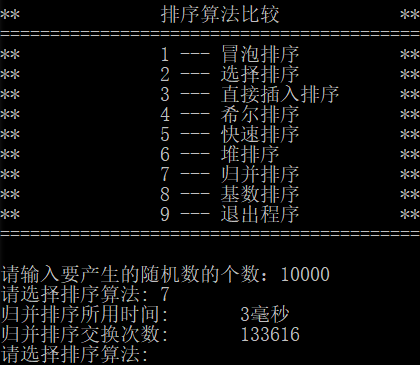
**测试用例：**

10000个数值

**预期结果：**

获得测试时间和交换次数

**实验结果：**



### 4.1.8 基数排序功能测试

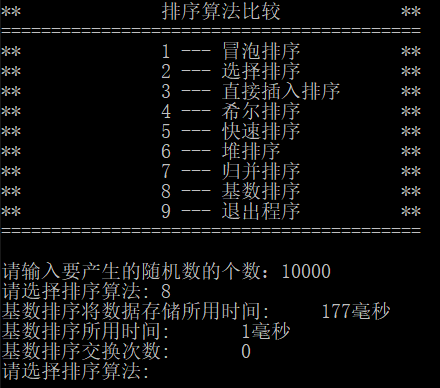
**测试用例：**

10000个数值

**预期结果：**

获得测试时间和交换次数

**实验结果：**



### 4.1.9 退出系统测试

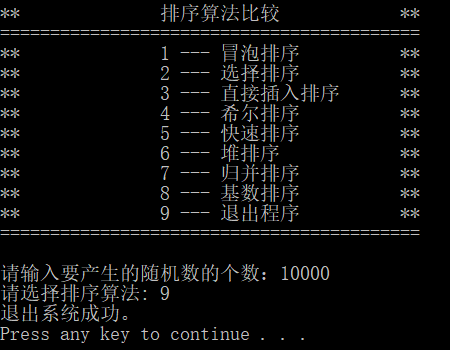
**测试用例：**

退出系统

**预期结果：**

得到提示，成功退出

**实验结果：**



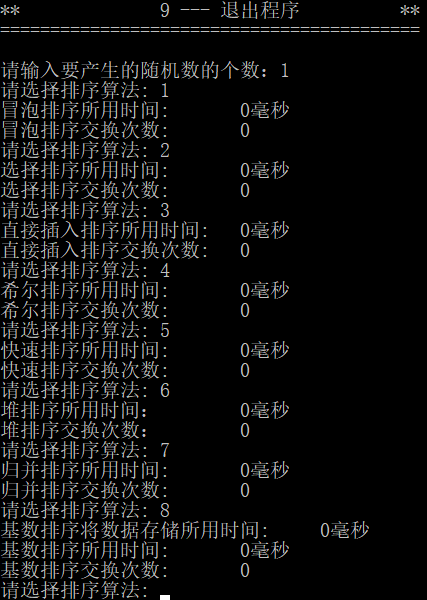
## 4.2 边界测试

### 4.2.1 只有一个随机数进行排序

**测试用例：**只有一个随机数

**预期结果：**程序正常运行

**实验结果：**

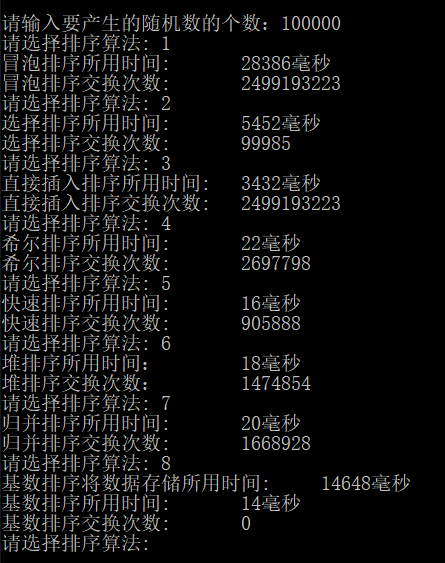


### 4.2.2 100000个随机数

**测试用例：**生成上限100000个随机数

**预期结果：**程序正常运行，不崩溃。

**实验结果：**



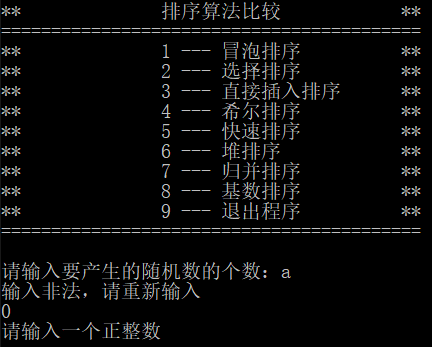
## 4.3 出错测试

### 4.3.1 随机数个数错误

**测试用例：**输入随机数个数为负数或字母

**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：**

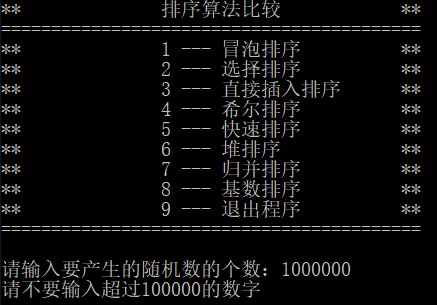


### 4.3.2 随机数越界错误

**测试用例：**输入超过100000随机数

**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：**

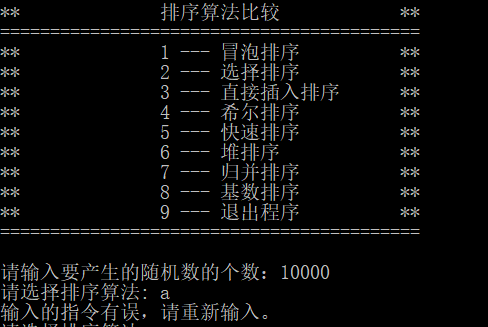


### 4.3.3 指令错误

**测试用例：**输入不存在的指令

**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 亮点

## 5.1 人机交互

本题中我添加了代码满足人机交互需求，防止意外退出或者崩溃。详情见3.8，3.9，4.3

## 5.2 最大堆和静态链表的使用

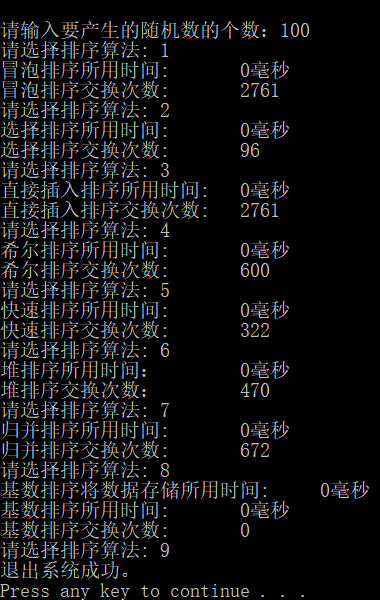
为了使用堆排序和LSD基数排序，我使用了堆和静态链表类解决问题，具体见2.1，2.2，2.3，3.6，3.8

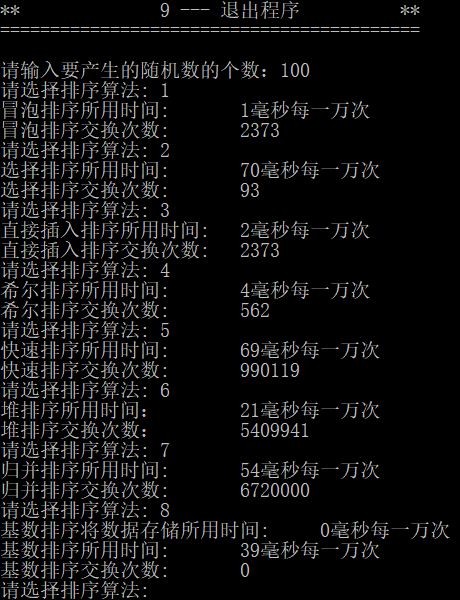
## 5.3 计时精确性

照比示例，我将计时的结果精确到了毫秒，更易说明问题。详见示例截屏

# 6. 不同情况下几种排序方法的效率及比较

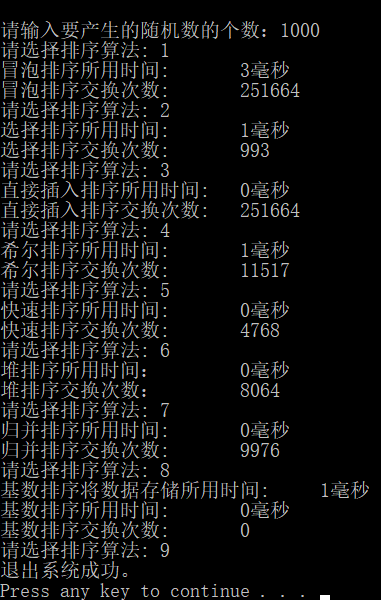
## 6.1 100个数据

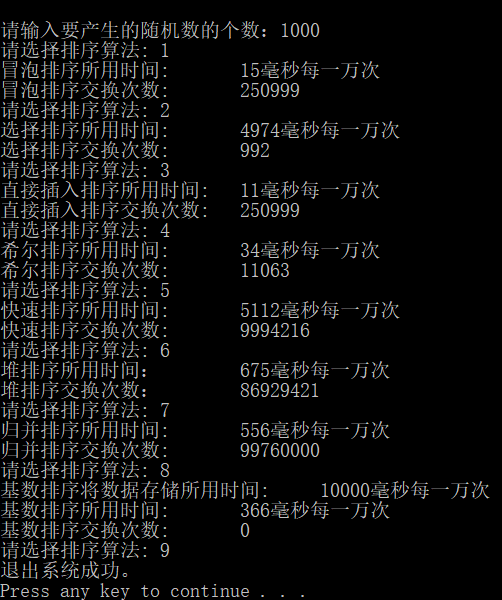




点评：100个数据时，每一种排序方式基本都完全不消耗时间，而如果对同一个数组进行10000次排序后，就有了时间差异，因为经过一次排序后数列已经有序，接下来如改进的冒泡算法，插入算法等就可以极为节省时间，而如选择排序，快速排序，归并排序等表现得就极为痛苦。

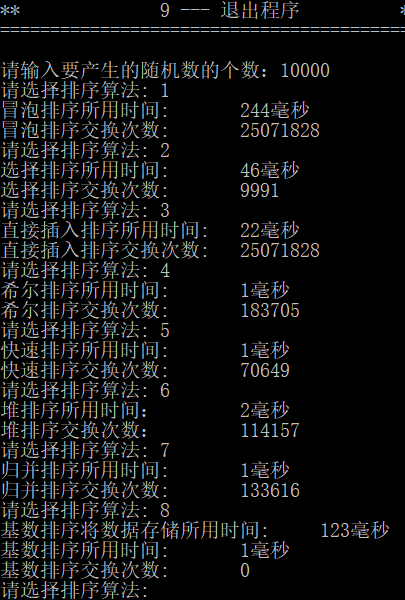
## 6.2 1000个数据

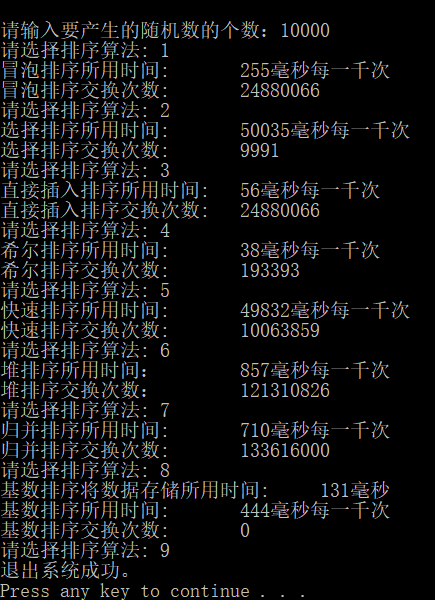




点评：1000个数据时，每一种排序方式仍无明显差异，冒泡排序，选择排序，希尔排序相对慢一些，但仍可以接受。而如果对同一个数组进行10000次排序后，可见选择排序，快速排序劣势已经明显，基数排序也已经渐渐表现出自己存储到静态链表的费时。

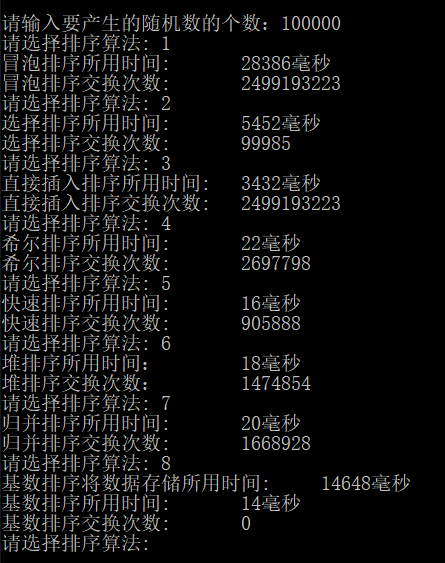
## 6.3 10000个数据





点评:当数据达到10000级别时，各种排序之间已经有了速度差异，冒泡算法明显很慢，冒泡，选择，直插的排序速度已经落后了。而基数排序存储速度的劣势也已经足够明显，但其进行排序的速度还是很快的。重复进行1000次的话，选择排序的时间已经让人不能接受了，快排次之，堆排序，归并也很挣扎。

## 6.4 100000个数据



点评：由于数量级已经很大了，就不进行多次已排序数组排序了，不过可以预想选择排序会非常慢，快排归并等表现也并不会好。而只进行一次排序时，快排，堆排，归并，希尔都是极好的选择，基数排序速度非常快，但它的存储耗时已经让人不能接受。冒泡排序在冒泡、直插、选择三大慢中慢到令人发指，难以忍受。

## 6.5 综述

当数据量小时，排序选择哪一种的差距并不大，因此考虑空间和特殊情况（如数组已经有序，数组大量相等数字）还是不建议使用归并排序、堆排序、基数排序这种占空间的方式，快速排序应对特殊情况的能力也并不优秀，建议使用三大慢速排序：改进的冒泡、选择、插入或者是希尔排序来解决问题。

当数据量达到一定规模（不算大量）时，冒泡选择直插就有了速度劣势，不过如果数据过于奇葩，有特殊情况存在，那么归并和快排的劣势就体现了出来，因此，如果数据很随机，不特殊，建议快排、希尔省空间又快速，而不确定数据是否奇葩时，可以用希尔或者堆排基数排序等用空间换高效性。当数据必然奇葩时，建议直接改进的冒泡或者直插解决。

当数据量足够大，且无序时，快排是一种省空间又省时的好方式，但如果数据重复量多或者一定有序还是需要斟酌快排的效率。希尔排序表现一直中规中矩，可以使用，归并堆排都很快，不过需要开辟大量存储空间，没有特殊需求时可以表现很棒，基数排序的话，如果数据直接录入静态链表，不需要重新拷贝的话是一种完美的选择，如果需要重新拷贝的话，那还是不要用了，排序2分钟，存储两小时。至于冒泡，选择。直插，如果不想等到天荒地老还是不要用了。