# 1 需求

## 1.1 背景分析

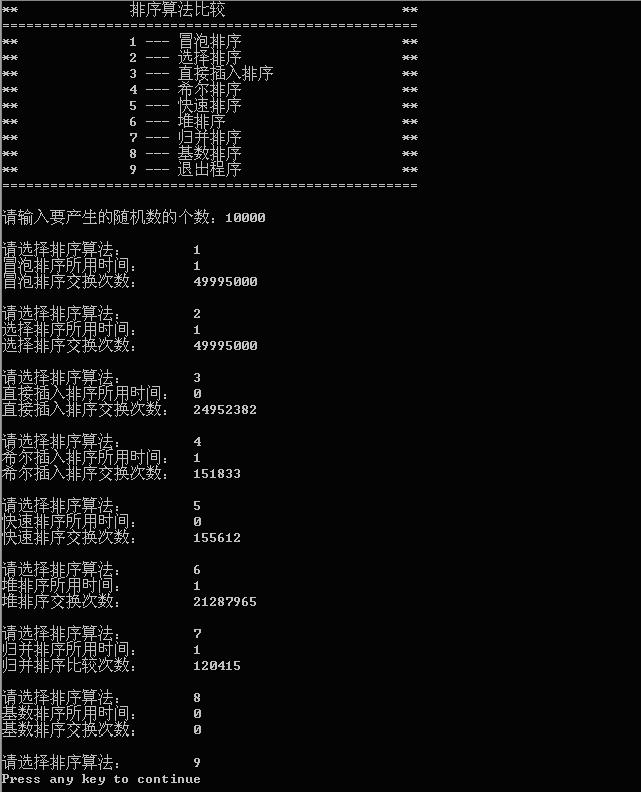
排序是按照其中的某个或某些关键字的大小，将一组杂乱无章的数据按照一定的次序排列起来的操作。排序算法可以分为内部排序和外部排序，内部排序是数据记录在内存中进行排序，而外部排序是因排序的数据很大，一次不能容纳全部的排序记录，在排序过程中需要访问外存。本项目侧重于内部排序的模拟与实现，包括冒泡排序、选择排序、直接插入排序、希尔插入排序、快速排序、堆排序、归并排序、基数排序等排序方法。

在计算机数据处理中，排序是最重要的操作之一。排序算法是使一组数据按照特定要求排列的方法，在很多领域得到相当地重视，尤其是在大量数据的处理方面，一个优秀的算法可以节省大量的资源。故本项目拟比对各种排序的计算时间和存储开销，并且根据实验结果比较这些方法的优缺点。

## 1.2 功能分析

利用随机函数产生随机数序列。对序列采用冒泡排序、选择排序、直接插入排序、希尔插入排序、快速排序、堆排序、归并排序、基数排序等八种排序方法排序，并统计在不同大小的序列中每种排序所花费的排序时间和排序交换次数。

其中，随机数的个数由用户定义，系统产生随机数组成的序列。在上述八种排序方法中选择某种对该序列排序，并且在排序后显示每种排序花费的排序时间和交换次数。在cmd界面上显示出来。



# 2 设计

## 2.1 数据结构设计

### 2.1.1 逻辑结构设计

基于功能分析，本测试系统的逻辑结构中各数据元素间存在“一对一”的简单关系，所以本测试系统只要使用线性结构一般线性表。

### 2.1.2 存储结构设计

基于功能分析和逻辑结构设计，存储结构应当使用顺序存储或链式存储。

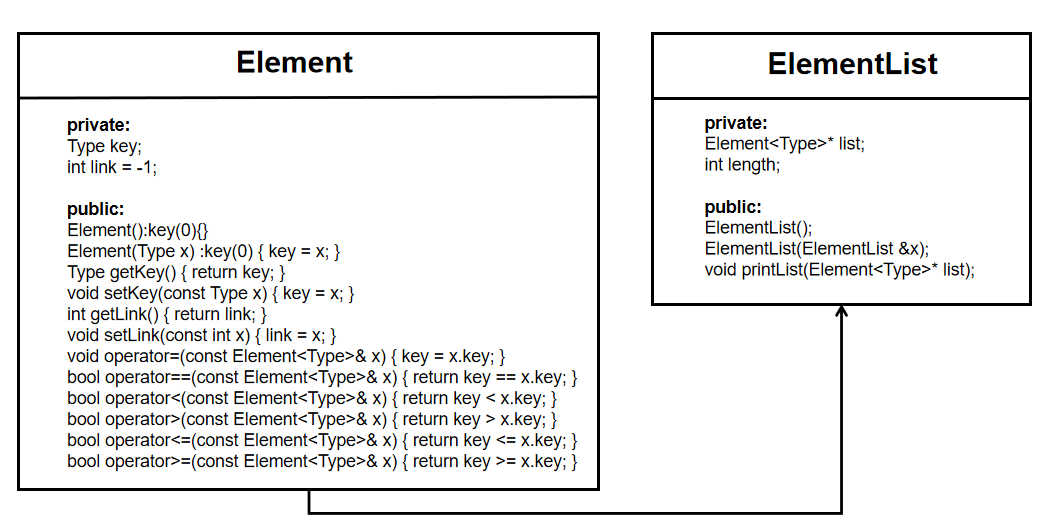
绝大多数排序算法建立在元素交换上，并不需要向特定位置插入删除。顺序存储与链式存储的时间复杂度一致情况下，顺序存储的空间复杂度优于链式存储，故选用顺序存储的顺序表结构。但对于某些特定的排序算法比如归并排序与基数排序，排序过程中需要大量的移动操作，此时可以使用链式存储顺序表，以降低操作的时间复杂度。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 顺序存储 | 链式存储 | 比较结果 |
| 插入功能 | O(n) | O(1) | 链式存储更优 |
| 删除功能 | O(n) | O(1) | 链式存储更优 |
| 查找功能 | O(n) | O(n) | 相同 |
| 修改功能 | O(1) | O(1) | 相同 |
| 统计功能 | O(n) | O(n) | 相同 |

为了统一顺序存储与链式存储的实现，本程序在每个数据的基础上增加了一个link指针指向它的下一个元素，对于使用顺序存储的排序算法，只需将link置于-1即可，表示此时link指针无效。

## 2.2 类设计

本项目共定义了两个类，分别是数据表节点类与数据表类，其中数据表类是多个数据节点类组成的序列。为了提高代码的重用性，本项目在定义数据表节点类与数据表类时均选择了类模板的方式，使其不仅能用于对数字序列进行排序，还能按照自定义的规则对任意类型数据序列进行排序，具有较高的现实适用性。



### 2.2.1 数据表节点的模板类

数据表节点类用于存储序列中的各项数据，以及逻辑上该数据指向的下一个数据。故数据表节点类Element的private属性包括数据域与指针域。对于选择排序、插入排序等不需要链式存储的排序算法，只需要将数据表节点的指针域置为-1表示该指针域失效即可。Element的public属性包含针对数据节点的各项操作。包括构造函数、析构函数、拷贝构造函数、读取数据域、修改数据域、读取指针域、修改指针域等。其中为了提高代码的可读性，程序使用operator进行了运算符重载，以直观表现数据间的大小关系。

//数据表节点类

**template** <**class** Type>

**class** Element {

**private**:

    Type key;

**int** link = -1;

**public**:

    Element():key(0){}

    Element(Type x) :key(0) { key = x; }

    Type getKey() { **return** key; }

**void** setKey(**const** Type x) { key = x; }

**int** getLink() { **return** link; }

**void** setLink(**const** **int** x) { link = x; }

**void** operator=(**const** Element<Type>& x) { key = x.key; }

**bool** operator==(**const** Element<Type>& x) { **return** key == x.key; }

**bool** operator<(**const** Element<Type>& x) { **return** key < x.key; }

**bool** operator>(**const** Element<Type>& x) { **return** key > x.key; }

**bool** operator<=(**const** Element<Type>& x) { **return** key <= x.key; }

**bool** operator>=(**const** Element<Type>& x) { **return** key >= x.key; }

};

### 2.2.2 数据表的模板类

数据表类是在数据表节点类的基础上构建的。其private属性包括指向数据表节点的指针，通过指针的方式构成一段数据序列。同时数据表类的private属性还包括数据表的长度length。数据表的public属性主要用于存放针对数据序列的各项操作，包括构造函数、析构函数、打印链表等。由于之后各项排序算法需要访问数据表中各节点的数值，因此还需在public属性中声明各项友元函数。

//数据表类

**class** ElementList {

**private**:

    Element<Type>\* list;

**int** length;

**public**:

    ElementList();

    ElementList(ElementList &x);

**void** printList(Element<Type>\* list);

**friend** **void** bubbleSort<Type>(ElementList<Type>& list, **long** **long**& KCN);

**friend** **void** selectSort<Type>(ElementList<Type>& list, **long** **long**& KCN);

**friend** **void** quickSort<Type>(ElementList<Type>& list, **long** **long**& KCN);

**friend** **void** quickSortR<Type>(ElementList<Type>& list, **int** left, **int** right, **long** **long**& KCN);

**friend** **int** partition<Type>(ElementList<Type>& list, **int** left, **int** right, **long** **long**& KCN);

**friend** **void** insertSort<Type>(ElementList<Type>& list, **long** **long**& KCN);

**friend** **void** shellSort<Type>(ElementList<Type>& list, **long** **long**& KCN);

**friend** **void** heapSort<Type>(ElementList<Type>& list, **long** **long**& KCN);

**friend** **void** siftDown<Type>(ElementList<Type>& list, **int** start, **int** end, **long** **long**& KCN);

**friend** **void** mergeSort<Type>(ElementList<Type>& list, **long** **long**& KCN);

**friend** **void** mergeSortR<Type>(ElementList<Type>& list, **int** left, **int** right, **long** **long**& KCN);

**friend** **void** merge<Type>(ElementList<Type>& list, ElementList<Type>& listTemp, **int** left, **int** right, **long** **long**& KCN);

**friend** **void** redixSort<Type>(ElementList<Type>& list, **long** **long**& KCN);

};

# 

# 3 实现

## 3.1 序列初始化功能的实现

### 3.1.1 序列初始化功能实现思路

数据表初始化的功能在数据表的构造函数中实现。用户输入希望产生的随机数个数，程序生成相应的序列，其中通过调用random库产生随机数。为了更好地开辟空间以及给序列的每个元素赋值，本程序采用了morden C++中的一种实现方式。对于传统的new关键字命令，计算机是同时执行了分配空间与依次调用每个元素的构造函数两项操作。而本程序通过operator new和static\_cast将上述两项操作分开执行，先给序列动态开辟空间，产生随机数后再对序列元素一个一个调用构造函数，从而提高程序的可读性。

### 3.1.2 序列初始化功能实现代码

//构造函数

**template** <**class** Type>

ElementList<Type>::ElementList() {

**int** sum;

    cout << "请输入要产生的随机数的个数：";

**while** (1) {

        cin >> sum;

**if** (cin.fail()) {

            cout << "输入错误,请重新输入：";

            cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

        }

**else** **if** (sum < 0 || sum>2147483647) {

            cout << "数值应当在0-2147483647范围内,请重新输入：";

            cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

        }

**else**

**break**;

    }

    length = sum;

    list = **static\_cast**<Element<Type>\*>(::operator **new**(**sizeof**(Element<Type>) \* sum));

**for** (**int** i = 0; i < sum; i++) {

**new**(&list[i]) Type(rand());

    }

}

## 3.2 冒泡排序功能的实现

### 3.2.1 冒泡排序功能实现思路

冒泡[排序算法](https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95/5399605?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E5%86%92%E6%B3%A1%E6%8E%92%E5%BA%8F/_blank)的核心是相邻两个元素的比较与交换，通过反复执行使得序列元素像气泡一样逐渐有序。如果前一个数据大于后一个数据，则执行交换操作，并对之后从初始元素到末尾元素的每一对相邻元素做相同的操作，确保每次循环中子序列最小的元素处于首位。持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要发生交换。分析冒泡排序的过程，可知其主要思路就是把小的元素往前调或者把大的元素往后调。由于比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，是不会再交换的。而如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，故相同元素的前后顺序并没有改变，由此可证得冒泡排序是一种[稳定排序](https://baike.baidu.com/item/%E7%A8%B3%E5%AE%9A%E6%8E%92%E5%BA%8F/4975546?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E5%86%92%E6%B3%A1%E6%8E%92%E5%BA%8F/_blank)算法。

### 3.2.2 冒泡排序功能实现代码

//冒泡排序

**template** <**class** Type>

**void** bubbleSort(ElementList<Type>& list, **long** **long**& KCN) {

**int** exchange = 0;

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

**for** (**int** j = list.length - 1; j > i; j--) {

            KCN++;

**if** (list.list[j - 1] > list.list[j]) {

                Element<Type> temp = list.list[j];

                list.list[j] = list.list[j - 1];

                list.list[j - 1] = temp;

                exchange = 1;

            }

**if** (!exchange)

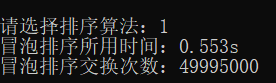
**break**;

        }

    }

}

### 3.2.3 冒泡排序功能实现样例



## 3.3 选择排序功能的实现

### 3.3.1 选择排序功能实现思路

选择排序是一种简单直观的排序算法，对于任意的数据都是 O(n²) 的时间复杂度。若数据规模越小，选择排序的执行效率越高。具体来说，假设长度为n的数组arr，要按照从小到大排序，那么先从n个数字中找到最小值min1，如果最小值min1的位置不在数组的最左端，也就是min1不等于arr[0]，则将最小值min1和arr[0]交换，接着在剩下的n-1个数字中找到最小值min2，如果最小值min2不等于arr[1]，则交换这两个数字，依次类推，直到数组arr有序排列。算法的时间复杂度为O(n²)。

### 3.3.2 选择排序功能实现代码

//选择排序

**template** <**class** Type>

**void** selectSort(ElementList<Type>& list, **long** **long**& KCN) {

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

**for** (**int** j = i + 1; j < list.length; j++) {

            KCN++;

**if** (list.list[i] > list.list[j]) {

                Element<Type> temp = list.list[i];

                list.list[i] = list.list[j];

                list.list[j] = temp;

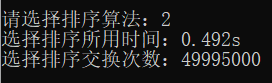
            }

        }

    }

}

### 3.3.3 选择排序功能实现样例



## 3.4 直接插入排序功能的实现

### 3.4.1 直接插入排序功能实现思路

​ 插入排序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。例如要将数组arr=[4,2,8,0,5,1]排序，可以将4看做是一个有序序列，将[2,8,0,5,1]看做一个无序序列。无序序列中2比4小，于是将2插入到4的左边，此时有序序列变成了[2,4]，无序序列变成了[8,0,5,1]。无序序列中8比4大，于是将8插入到4的右边，有序序列变成了[2,4,8],无序序列变成了[0,5,1]。以此类推，最终数组按照从小到大排序。该算法的时间复杂度为O(n²)。其实现步骤主要如下，首先从第一个元素开始，该元素可以认为已经被排序。之后取出下一个元素，在已经排序的元素序列中从后向前扫描，如果该元素（已排序）大于新元素，将该元素移到下一位置。反复移动元素位置，直到找到已排序的元素小于或者等于新元素的位置，此时可将新元素插入到该位置后。由于直接插入排序中不涉及远距离交换，故直接插入排序是稳定的。

### 3.4.2 直接插入排序功能实现代码

//直接插入排序

**template** <**class** Type>

**void** insertSort(ElementList<Type>& list, **long** **long**& KCN) {

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

**int** j = i - 1;

        Element<Type> temp = list.list[i];

**while** (j >= 0 && temp < list.list[j]) {

            list.list[j + 1] = list.list[j];

            j--;

            KCN++;

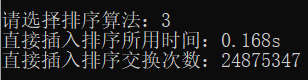
        }

        list.list[j + 1] = temp;

    }

}

### 3.4.3 直接插入排序功能实现样例



## 3.5 希尔排序功能的实现

### 3.5.1 希尔排序功能实现思路

希尔排序在插入排序算法的基础上进行了改进，算法的时间复杂度有较大幅度降低，但希尔排序是非稳定排序算法。其算法的基本思想是先将待排记录序列分割成为若干子序列分别进行插入排序，待整个序列中的记录基本有序时，再对全体记录进行一次直接插入排序。该算法时间复杂度为O(n log n)。

### 3.5.2 希尔排序功能实现代码

//希尔排序

**template** <**class** Type>

**void** shellSort(ElementList<Type>& list, **long** **long**& KCN) {

**int** gap = list.length / 2;

**while** (gap) {

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

**int** j = i - gap;

            Element<Type> temp = list.list[i];

**while** (j >= 0 && temp < list.list[j]) {

                list.list[j + gap] = list.list[j];

                j -= gap;

                KCN++;

            }

            list.list[j + gap] = temp;

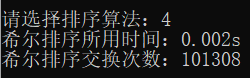
        }

        gap = gap == 2 ? 1 : (**int**)gap / 2.2;

    }

}

### 3.5.3 希尔排序功能实现样例



## 3.6 快速排序功能的实现

### 3.6.1 快速排序功能实现思路

快速排序是当前排序效率最高的算法，因为它的内部循环可以在大部分的架构上很有效率地被实现出来。快速排序的基本思想是通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。一趟快速排序的具体过程可描述为：从待排序列中任意选取一个记录作为基准值，然后将记录中关键字比它小的记录都安置在它的位置之前，将记录中关键字比它大的记录都安置在它的位置之后。这样，以该基准值为分界线，将待排序列分成的两个子序列。快速排序是处理大数据最快的排序算法之一，时间复杂度为O(n log n)。

### 3.6.2 快速排序功能实现代码

//快速排序

**template** <**class** Type>

**void** quickSort(ElementList<Type>& list,**long** **long**& KCN) {

    quickSortR(list, 0, list.length, KCN);

}

//快速排序递归实现

**template** <**class** Type>

**void** quickSortR(ElementList<Type>& list, **int** left, **int** right, **long** **long**& KCN) {

**if** (left < right) {

**int** locate = partition(list, left, right, KCN);

        quickSortR(list, locate + 1, right, KCN);

        quickSortR(list, left, locate - 1, KCN);

    }

}

//快速排序的序列划分

**template** <**class** Type>

**int** partition(ElementList<Type>& list, **int** left, **int** right, **long** **long**& KCN) {

**int** pivotpos = left;

**for** (**int** i = pivotpos + 1; i < right; i++) {

        KCN++;

**if** (list.list[i] < list.list[left]) {

            pivotpos++;

            Element<Type> temp = list.list[i];

            list.list[i] = list.list[pivotpos];

            list.list[pivotpos] = temp;

        }

    }

    Element<Type> temp = list.list[left];

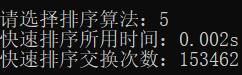
    list.list[left] = list.list[pivotpos];

    list.list[pivotpos] = temp;

**return** pivotpos;

}

### 3.6.3 快速排序功能实现样例



## 3.7 堆排序功能的实现

### 3.7.1 堆排序功能实现思路

堆排序是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于或者大于它的父节点。堆排序可以说是一种利用堆的概念来排序的选择排序。分为两种方法：每个结点的值都大于等于其左右孩子结点的值，称为大顶堆。或者每个结点的值都小于等于其左右孩子结点的值，称为小顶堆。该算法时间复杂度为O(n log n)。堆排序首先将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区，之后将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区,且满足R[1,2…n-1]<=R[n]。由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

### 3.7.2 堆排序功能实现代码

//堆排序

**template** <**class** Type>

**void** heapSort(ElementList<Type>& list, **long** **long**& KCN) {

**for** (**int** i = (list.length - 2) / 2; i >= 0; i--) {

        siftDown(list, i, list.length - 1, KCN);

    }

**for** (**int** i = list.length - 1; i >= 0; i--) {

        Element<Type> temp = list.list[0];

        list.list[0] = list.list[i];

        list.list[i] = temp;

        siftDown(list, 0, i - 1, KCN);

    }

}

//堆的自上而下调整算法

**template** <**class** Type>

**void** siftDown(ElementList<Type>& list, **int** start, **int** end, **long** **long**& KCN) {

**int** i = start, j = 2 \* i + 1;

    Element<Type> temp = list.list[i];

**while** (j <= end) {

**if** (j < end && list.list[j + 1] > list.list[j])

            j++;

**if** (temp > list.list[j])

**break**;

**else** {

            list.list[i] = list.list[j];

            KCN++;

            i = j;

            j = 2 \* i + 1;

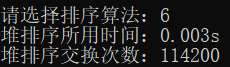
        }

    }

    list.list[i] = temp;

}

### 3.7.3 堆排序功能实现样例



## 3.8 归并排序功能的实现

### 3.8.1 归并排序功能实现思路

归并排序是建立在归并操作上的一种有效的排序算法。该算法是分治法的一个非常典型的应用。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。代价是需要额外的内存空间。若将两个有序表合并成一个有序表，称为二路归并。 该算法时间复杂度为O(nlog n)。该算法首先把长度为n的输入序列分成两个长度为n/2的子序列，之后对这两个子序列分别采用归并排序，最后将两个排序好的子序列合并成一个最终的排序序列。

### 3.8.2 归并排序功能实现代码

//归并排序

**template** <**class** Type>

**void** mergeSort(ElementList<Type>& list, **long** **long**& KCN) {

    mergeSortR(list, 0, list.length - 1, KCN);

}

//归并排序递归实现

**template** <**class** Type>

**void** mergeSortR(ElementList<Type>& list, **int** left, **int** right, **long** **long**& KCN) {

**if** (left < right) {

**int** mid = (left + right) / 2;

        mergeSortR(list, left, mid, KCN);

        mergeSortR(list, mid + 1, right, KCN);

        ElementList<Type> listTemp(list);

        merge(list, listTemp, left, right, KCN);

    }

}

//二路归并

**template** <**class** Type>

**void** merge(ElementList<Type>& list, ElementList<Type>& listTemp, **int** left, **int** right, **long** **long**& KCN) {

**int** mid = (left + right) / 2;

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

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

**if** (listTemp.list[s1] <= listTemp.list[s2])

            list.list[t++] = listTemp.list[s1++];

**else**

            list.list[t++] = listTemp.list[s2++];

        KCN++;

    }

**while** (s1 <= mid) {

        list.list[t++] = listTemp.list[s1++];

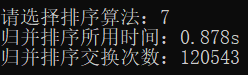
    }

**while** (s2 <= right)

        list.list[t++] = listTemp.list[s2++];

}

### 3.8.3 归并排序功能实现样例



## 3.9 基数排序功能的实现

### 3.9.1 基数排序功能实现思路

​ 基数排序是一种非比较型整数排序算法，其原理是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。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。该算法时间复杂度为O(n+k)。基数排序算法先取得数组中的最大数，并取得位数。之后以arr为原始数组，从最低位开始取每个位组成radix数组，再利用计数排序适用于小范围数的特点对radix进行计数排序。

### 3.9.2 基数排序功能实现代码

//获取数值

**int** getDigit(**int** number, **int** d) {

**return** (number / (**int**)pow(10, d)) % 10;

}

//基数排序

**template** <**class** Type>

**void** redixSort(ElementList<Type>& list, **long** **long**& KCN) {

**int** front[10], rear[10], begin = 0, start = 0;

**for** (**int** i = 0; i < list.length - 1; i++) {

        list.list[i].setLink(i + 1);

    }

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

**int** j = start;

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

            front[i] = rear[i] = -1;

        }

**for** (**int** k = 0; k < list.length; k++) {

**int** digit = getDigit(list.list[j].getKey(), i);

**if** (front[digit] == -1) {

                front[digit] = rear[digit] = j;

            }

**else** {

                list.list[rear[digit]].setLink(j);

                rear[digit] = j;

            }

            j = list.list[j].getLink();

        }

**int** before = 0, after = 0;

**while** (before < 10) {

**if** (front[before] != -1) {

                after = before + 1;

**while** (front[after] == -1) {

                    after++;

                }

                list.list[rear[before]].setLink(front[after]);

                before = after;

            }

**if** (after == 0) {

                before++;

                begin++;

            }

        }

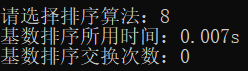
        start = front[begin];

    }

**int** step = start;

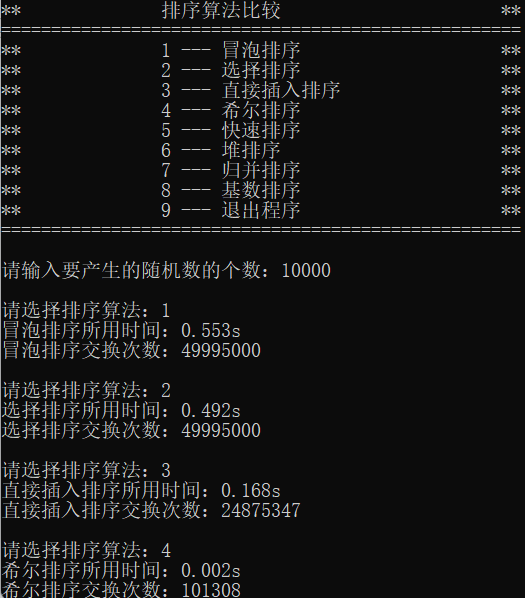
}

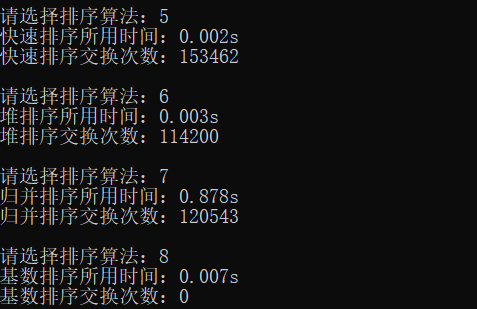
### 3.9.3 基数排序功能实现样例



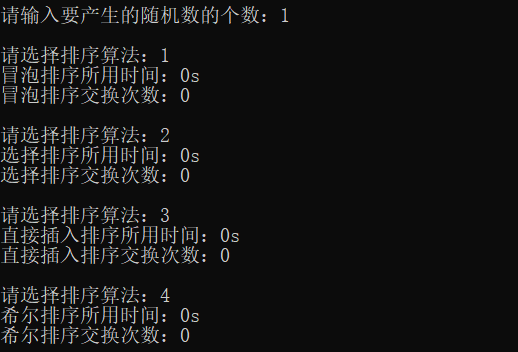
# 4 测试

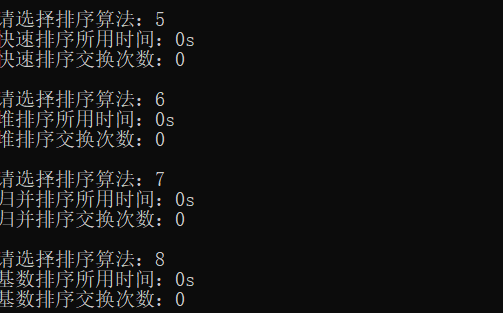
## 4.1 常规测试





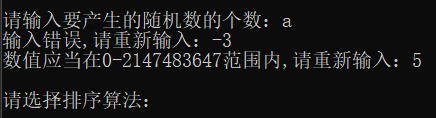
## 4.2 边界测试



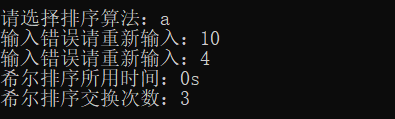


## 4.3 出错测试

### 4.3.1 随机数个数输入错误

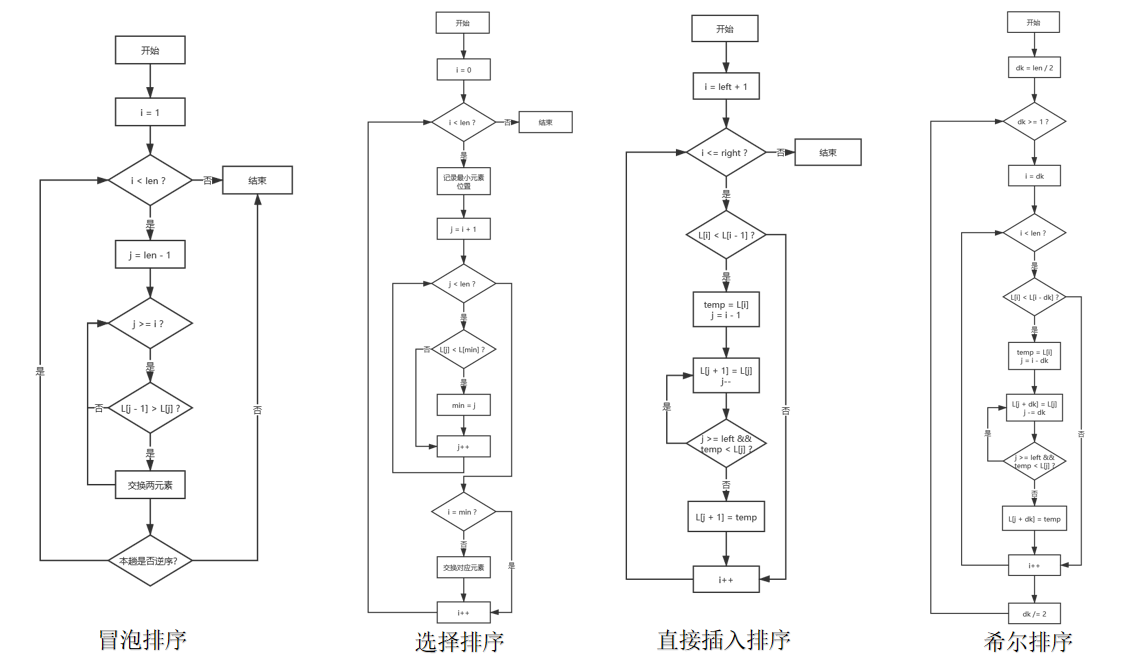


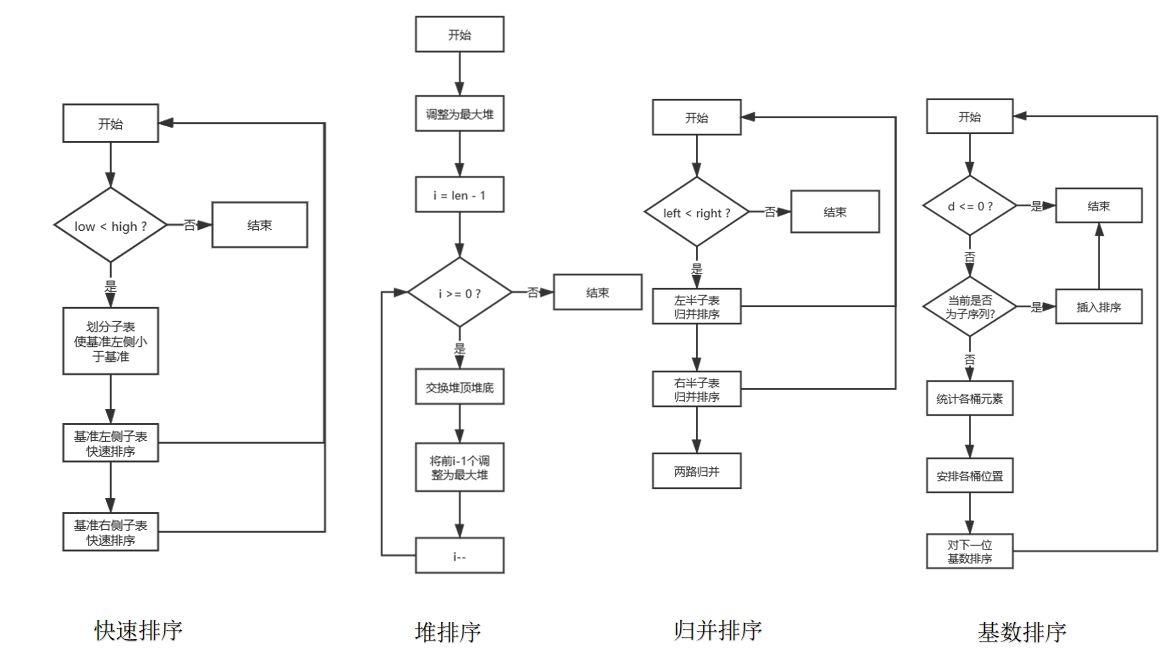
### 4.3.2 操作码输入错误



# 5 结论

## 5.1 八种排序思路比较





## 5.2 八种排序的优缺点比较

### 5.2.1 冒泡排序

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法种类 | 时间复杂度 | | | 空间复杂度 | 是否稳定 |
| 最好情况 | 平均情况 | 最坏情况 |
| 冒泡排序 |  |  |  |  | 稳定 |

优点：冒泡排序最好情况下时间复杂度仅为*O*(*n*)。在序列关键字基本有序时，排序效率较高。

冒泡排序空间复杂度仅为*O*(1)。

冒泡排序是稳定的排序算法。

缺点：冒泡排序平均情况和最坏情况下时间复杂度达到*O*(*n*2)，在未知序列关键字分布情况或序列关键字基本无序时，排序效率太低。

### 5.2.2 选择排序

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法种类 | 时间复杂度 | | | 空间复杂度 | 是否稳定 |
| 最好情况 | 平均情况 | 最坏情况 |
| 选择排序 |  |  |  |  | 不稳定 |

优点：选择排序移动次数较少。当元素记录信息量较大且问题规模n较小时，选择排序较好。

选择排序空间复杂度仅为*O*(1)。

缺点：选择排序所有情况下时间复杂度都达到*O*(*n*2)，在未知序列关键字分布情况时，排序效率太低。

选择排序是不稳定的排序算法。

### 5.2.3 直接插入排序

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法种类 | 时间复杂度 | | | 空间复杂度 | 是否稳定 |
| 最好情况 | 平均情况 | 最坏情况 |
| 直接插入排序 |  |  |  |  | 稳定 |

优点：直接插入排序最好情况下时间复杂度仅为*O*(*n*)。在序列关键字基本有序时，排序效率较高。

直接插入排序空间复杂度仅为*O*(1)。

直接插入排序是稳定的排序算法。

缺点：直接插入排序平均情况和最坏情况下时间复杂度达到*O*(*n*2)，在未知序列关键字分布情况或序列关键字基本无序时，排序效率太低。

### 5.2.4 希尔排序

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法种类 | 时间复杂度 | | | 空间复杂度 | 是否稳定 |
| 最好情况 | 平均情况 | 最坏情况 |
| 希尔排序 |  |  |  |  | 不稳定 |

优点：希尔排序的增量步长有多种取法，在特定取法可以有很高的效率。

希尔排序空间复杂度仅为*O*(1)。

希尔排序是稳定的排序算法。

缺点：希尔排序的增量步长有多种取法，很难分析时间复杂度。在序列关键字基本有序时，排序效率不高。

### 5.2.5 快速排序

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法种类 | 时间复杂度 | | | 空间复杂度 | 是否稳定 |
| 最好情况 | 平均情况 | 最坏情况 |
| 快速排序 |  |  |  |  | 不稳定 |

优点：快速排序平均情况和最好情况下时间复杂度仅为*O*(*n*)。在未知序列关键字分布情况时，排序效率较高。

缺点：快速排序最坏情况下时间复杂度达到*O*(*n*2)，在序列关键字基本无序时，排序效率太低。

快速排序空间复杂度达到*O*()。

快速排序是不稳定的排序算法。

### 5.2.6 堆排序

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法种类 | 时间复杂度 | | | 空间复杂度 | 是否稳定 |
| 最好情况 | 平均情况 | 最坏情况 |
| 堆排序 |  |  |  |  | 不稳定 |

优点：堆排序所有情况下时间复杂度仅为*O*(*n*)。对任何序列排序时，排序效率都较高。

堆排序空间复杂度仅为*O*()。

缺点：堆排序是不稳定的排序算法。

### 5.2.7 归并排序

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法种类 | 时间复杂度 | | | 空间复杂度 | 是否稳定 |
| 最好情况 | 平均情况 | 最坏情况 |
| 归并排序 |  |  |  |  | 稳定 |

优点：归并排序所有情况下时间复杂度仅为*O*(*n*)。对任何序列排序时，排序效率都较高。

归并排序是稳定的排序算法。

缺点：归并排序空间复杂度达到*O*(*n*)。

### 5.2.8 基数排序

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法种类 | 时间复杂度 | | | 空间复杂度 | 是否稳定 |
| 最好情况 | 平均情况 | 最坏情况 |
| 基数排序 |  |  |  |  | 稳定 |

优点：基数排序与问题规模复杂度n的关系为一次函数。若n很大，关键码位数d较少且可以分解时，排序非常高。

基数排序空间复杂度仅为*O*(*r*)。

基数排序时稳定的排序算法。

缺点：基数排序需要关键码位数d较少且可以分解，没有非常广泛的适用性。