项目说明文档

数据结构课程设计

——排序算法比较

作 者 姓 名： 杨滕超

学 号： 2151298

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 项目概述](#_Toc495668153) 5

[1.1 项目背景](#_Toc495668154) 5

[1.2 项目功能](#_Toc495668155) 6

[1.3 项目分析](#_Toc495668154) 6

[2 项目设计](#_Toc495668156) 7

[2.1 总体设计](#_Toc495668158) 7

[3 项目实现](#_Toc495668161) 7

[3.1 冒泡排序](#_Toc495668162) 7

[3.1.1 流程图](#_Toc495668167) 7

[3.1.2 算法说明](#_Toc495668167) 7

[3.1.3 代码实现](#_Toc495668167) 8

[3.1.4 算法分析](#_Toc495668167) 8

[3.2 选择排序](#_Toc495668162) 9

[3.2.1 流程图](#_Toc495668167) 9

[3.2.2 算法说明](#_Toc495668167) 9

[3.2.3 代码实现](#_Toc495668167) 9

[3.2.4 算法分析](#_Toc495668167) 10

[3.3 直接插入排序](#_Toc495668162) 11

[3.3.1 流程图](#_Toc495668167) 11

[3.3.2 算法说明](#_Toc495668167) 11

[3.3.3 代码实现](#_Toc495668167) 11

[3.3.4 算法分析](#_Toc495668167) 12

[3.4 希尔排序](#_Toc495668162) 13

[3.4.1 流程图](#_Toc495668167) 13

[3.4.2 算法说明](#_Toc495668167) 13

[3.4.3 代码实现](#_Toc495668167) 13

[3.4.4 算法分析](#_Toc495668167) 14

[3.5 快速排序递归方法](#_Toc495668162) 14

[3.5.1 流程图](#_Toc495668167) 14

[3.5.2 算法说明](#_Toc495668167) 15

[3.5.3 代码实现](#_Toc495668167) 15

[3.5.4 算法分析](#_Toc495668167) 16

[3.6 快速排序非递归方法](#_Toc495668162) 17

[3.6.1 流程图](#_Toc495668167) 17

[3.6.2 算法说明](#_Toc495668167) 18

[3.6.3 代码实现](#_Toc495668167) 18

[3.6.4 算法分析](#_Toc495668167) 19

[3.7 堆排序非递归方法](#_Toc495668162) 20

[3.7.1 流程图](#_Toc495668167) 20

[3.7.2 算法说明](#_Toc495668167) 20

[3.7.3 代码实现](#_Toc495668167) 20

[3.7.4 算法分析](#_Toc495668167) 21

[3.8 堆排序递归方法](#_Toc495668162) 22

[3.8.1 流程图](#_Toc495668167) 22

[3.8.2 算法说明](#_Toc495668167) 22

[3.8.3 代码实现](#_Toc495668167) 22

[3.8.4 算法分析](#_Toc495668167) 23

[3.9 归并排序](#_Toc495668162) 24

[3.9.1 流程图](#_Toc495668167) 24

[3.9.2 算法说明](#_Toc495668167) 24

[3.9.3 代码实现](#_Toc495668167) 24

[3.9.4 算法分析](#_Toc495668167) 25

[3.10 计数排序](#_Toc495668162) 26

[3.10.1 流程图](#_Toc495668167) 26

[3.10.2 算法说明](#_Toc495668167) 26

[3.10.3 代码实现](#_Toc495668167) 26

[3.10.4 算法分析](#_Toc495668167) 27

[3.11 基数排序LSD](#_Toc495668162) 28

[3.11.1 流程图](#_Toc495668167) 28

[3.11.2 算法说明](#_Toc495668167) 28

[3.11.3 代码实现](#_Toc495668167) 28

[3.11.4 算法分析](#_Toc495668167) 29

[3.12 基数排序LSD](#_Toc495668162) 30

[3.12.1 流程图](#_Toc495668167) 30

[3.12.2 算法说明](#_Toc495668167) 31

[3.12.3 代码实现](#_Toc495668167) 31

[3.12.4 算法分析](#_Toc495668167) 32

[4 项目测试](#_Toc495668161) 33

[4.1 10个数字测试](#_Toc495668174) 33

[4.2 100个数字测试](#_Toc495668174) 34

[4.3 1000个数字测试](#_Toc495668174) 35

[4.4 10000个数字测试](#_Toc495668174) 36

[4.5 100000个数字测试](#_Toc495668174) 37

[4.6 1000000个数字测试](#_Toc495668174) 38

[4.7 10000000个数字测试](#_Toc495668174) 39

[4.8 99999999个数字测试](#_Toc495668174) 40

[4.9 linux下10个数字测试](#_Toc495668174) 42

[4.10 linux下100个数字测试](#_Toc495668174) 43

[4.11 linux下1000个数字测试](#_Toc495668174) 45

[4.12 linux下10000个数字测试](#_Toc495668174) 47

[4.13 linux下100000个数字测试](#_Toc495668174) 49

[4.14 linux下1000000个数字测试](#_Toc495668174) 50

[4.15 linux下10000000个数字测试](#_Toc495668174) 51

[4.16 linux下99999999个数字测试](#_Toc495668174) 52

[4.17 错误输入判断](#_Toc495668174) 53

[5 测试分析比较](#_Toc495668161) 54

1 项目概述

* 1. 项目背景

排序是对数据元素的逻辑顺序或物理顺序的一种重新排列，其中排成非递减顺序叫做“正序”，反之叫做“逆序”。排序在计算机数据处理中经常遇到，占据了很大的比重。在日常的数据处理中，一般认为有25%的时间花在了排序上，而对于安装程序，多达一半的时间花在了对表的排序上。

所谓数据表，就是待排序数据元素的有限集合。

排序的依据是排序码，即元素或记录中的用于作为排序依据的项，通常数据元素有多个属性域，即多个数据成员组成，其中有一个属性域可以用来区分元素，作为排序依据。如果在数据表中各个元素的排序码互不相同，这种排序码就是主排序码，利用某一种排序方式按照主排序码进行排序，其结果是唯一的。但也有可能数据表中有些元素的排序码相同，这种排序码成为次排序码，其结果可能不唯一。

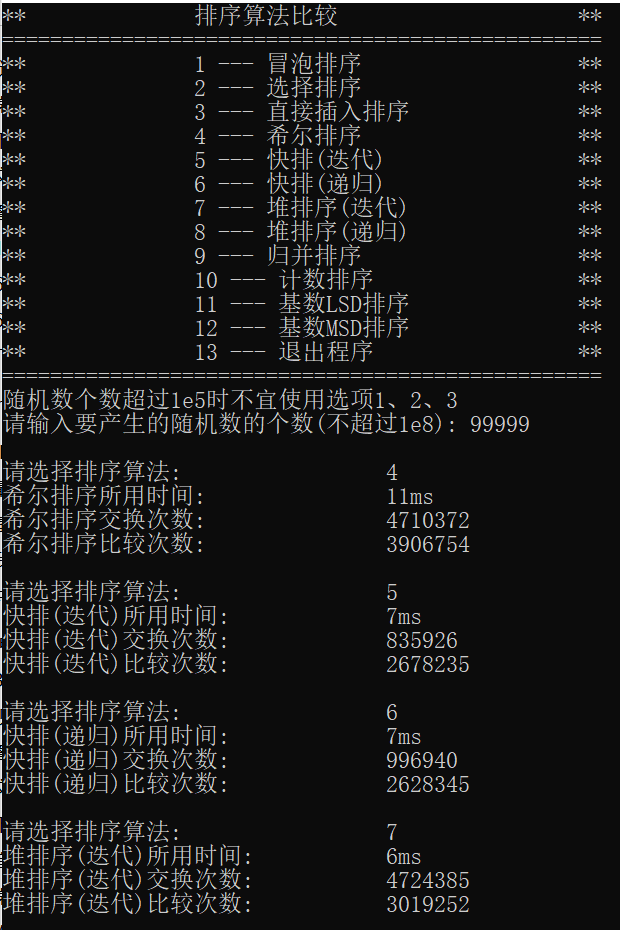
排序在大量领域上得到非常多的重视，并且在处理大数据方面，一个好的算法可以节省大量的时间与成本。通过不断地迭代更新，一个算法才能不断优化。

本项目讨论在课程中所学的排序算法，并比较它们的性能，为日后的算法研打下基础。

* 1. 项目功能

根据用户输入的随机数个数，生成该个数数量的随机数，利用冒泡排序、选择排序、直接插入排序等排序进行进行各个算法性能的分析与比较，排序过程中，我们记录下所用的时间、元素交换的次数与元素比较的次数，并将结果输出。

项目示例如下：



* 1. 项目分析

1. 用户输入不合法的数据时候，程序输出相关输入错误的提示，而非程序崩溃。具有一定的健壮性。
2. 通过丰富的提示，显示出程序运行的信息，用户可以直观方便地了解程序运行的情况，具有用户友好性
3. 通过输入不同的数据个数与多种排序算法的选择，用户可以清晰地对比出各个排序算法的优劣。

2 项目设计

2.1 总体设计

利用根据用户的输入选择，调用相应的排序函数。此前需要根据用户输入的待排序元素个数生成排序数组。这里需要注意的是，我们需要根据生成树随机数组深拷贝出另一个数组进行排序，否则当上一个排序函数完成排序之后，下一个排序函数的排序对象应该是复制出来的随机数组，而非上一个排序函数已经排好序的数组。

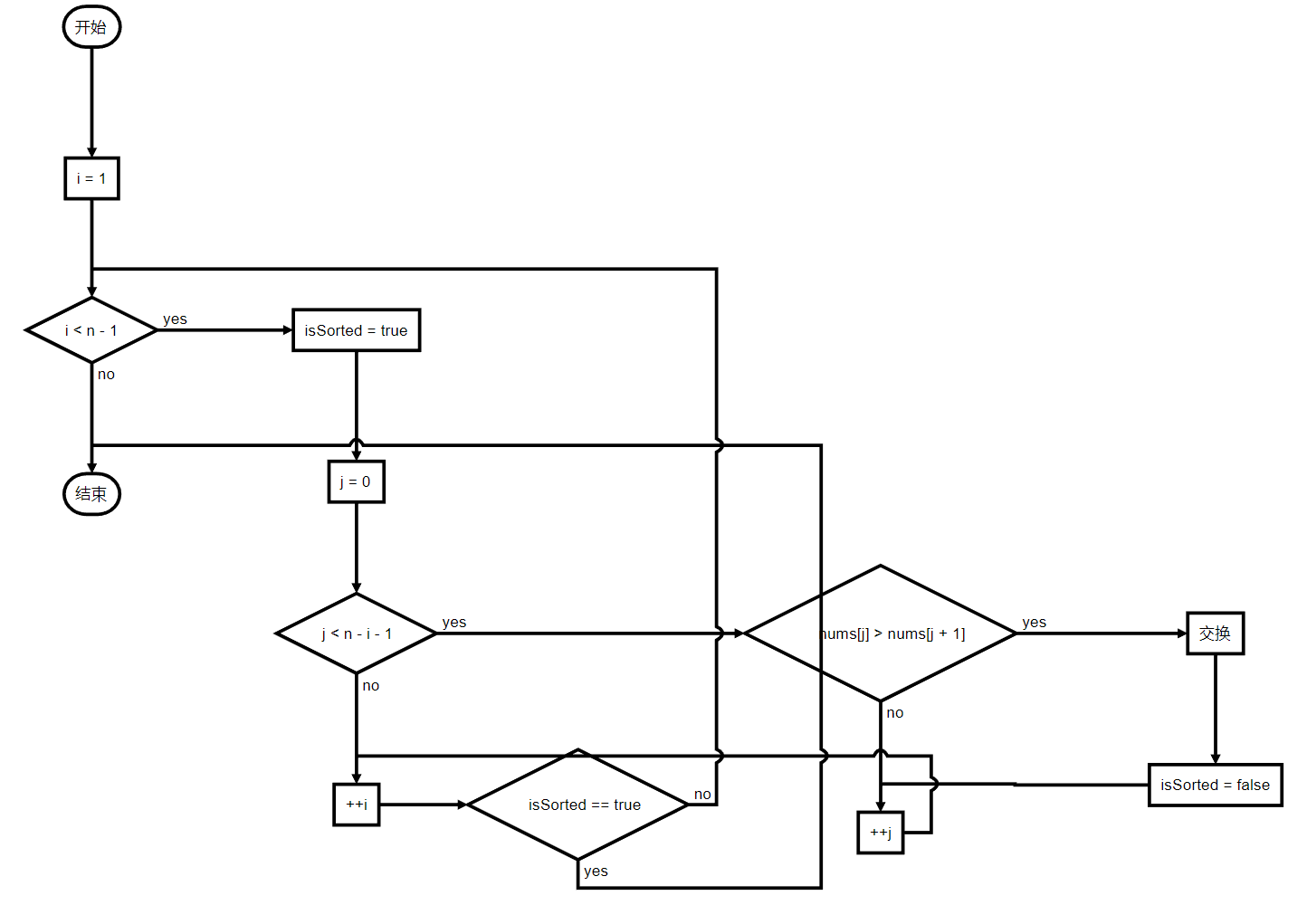
3 项目实现

本项目实现了12种排序算法，其中包括：冒泡排序、选择排序、直接插入排序、希尔排序、快速排序非递归方法、快速排序递归方法、堆排序非递归方法、堆排序递归方法、归并排序、计数排序、基数排序LSD、基数排序MSD。

以下将分别介绍每种排序算法的说明与实现，性能与分析。

3.1 冒泡排序

3.1.1 流程图



3.1.2 算法说明

冒泡排序是一种把每一轮把最小的元素往前移动或把最大的元素往后移动的排序。每一轮移动的的最终结果，即为未排序序列种找到一个最大（或最小）数据的最终位置。在排序的过程中，通过比较相邻两个数据的大小，不满足则进行交换，扫描整个未排序的序列，依次完成排序。由于该过程将最大元素移动至最后，接着将次大元素移动至次最后，以此类推，该过程类似气泡的上浮，由此得名冒泡排序。

同时我们需要注意到的是，当相邻两个元素大小相同时，并不进行数据的移动，因此对于相同大小的元素，排序前后的相对位置不发生改变，由此得出冒泡排序是一种稳定排序。

3.1.3代码实现

void bubbleSort(int\* nums, int n, accord& data){

//每一轮将原来的nums[i]换到能去的最后的位置

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

//有序标记，若检查一遍仍为true说明可以结束

bool isSorted = true;

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

if (++data.cmp, nums[j] > nums[j + 1]){

//有元素交换，后面的不是有序

isSorted = false;

//交换

int t = nums[j];

nums[j] = nums[j + 1];

nums[j + 1] = t;

//移动步数增加

data.move += 3;

}

}

//后面都有序提前结束

if (isSorted)

break;

}

}

3.2.4 算法分析

对于n个元素，需要进行n - 1趟位置的确定，每趟排序最坏情况，即逆序，需要进行n - i次的关键字比较，总共进行n(n - 1) / 2 次关键码的比较和 3n(n - 1) / 2 次元素的交换；对于最好的情况，数组本身已经有序，冒泡排序只需要进行一轮检查，进行 n - 1次元素比较和0次交换，此时时间复杂度为O(n)。但总体而言，冒泡排序的时间复杂度为O(n^2)。

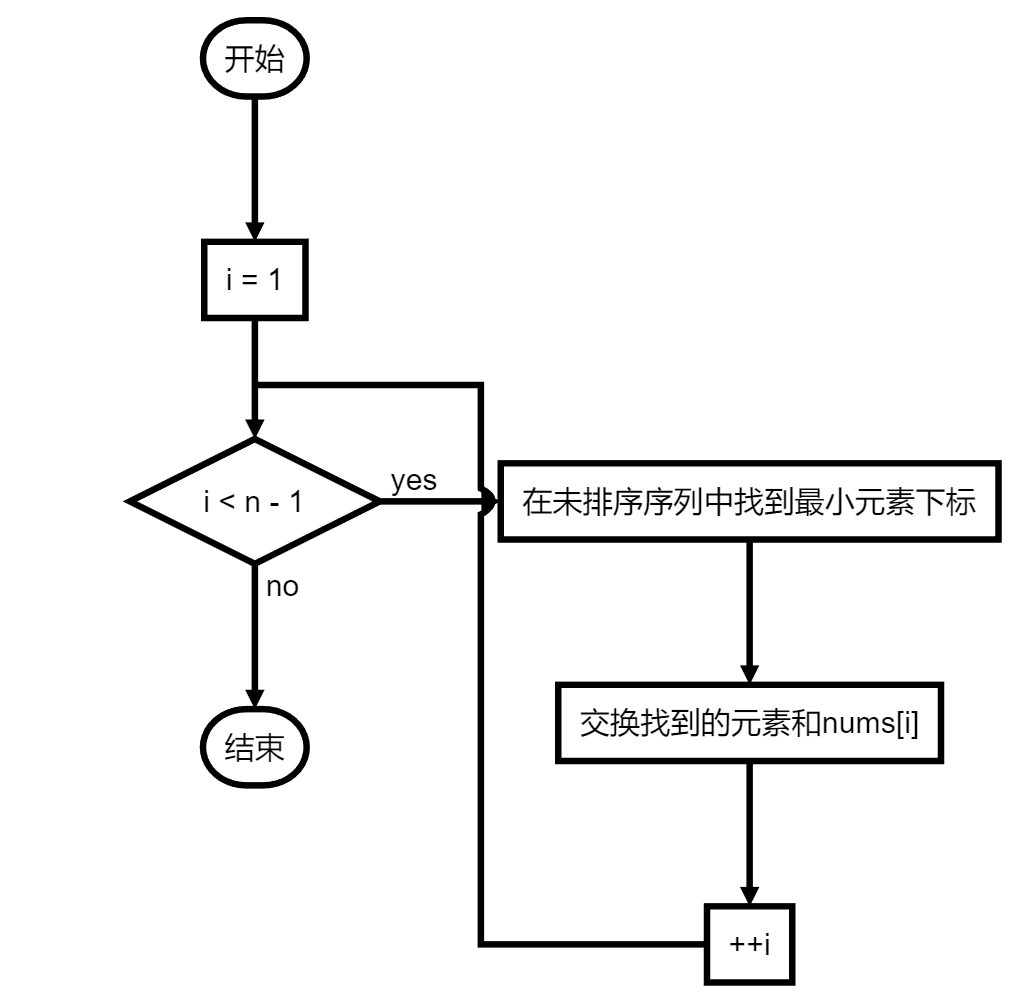
对于冒泡排序，其过程中不需要另外开辟新的数组空间，因此著需要常数个用于交换或标记的变量，所以冒泡排序是一种原地排序算法，空间复杂度为O(1)。

当相邻两个元素大小相同时，并不进行数据的移动，因此对于相同大小的元素，排序前后的相对位置不发生改变，由此得出冒泡排序是一种稳定排序。

关于算法优化，若一趟检查下来发现没有元素发生交换移动，说明此时未排序序列中已经达到了有序状态，可以提前结束排序，由此进行优化算法的速度，比较次数和交换次数。

3.2 选择排序

3.2.1 流程图



3.2.2 流程图

选择排序的思想是从待排序序列中选择最小的元素放到整个序列的第一个位置，然后选择第二小的元素放到整个序列中的第二个位置，依次类推完成排序，即每趟遍历未排序的序列，在 n - i + 1个元素中选择最小的元素，并将其作为有序序列中的第i个元素。我们之许哟啊完成n - 1趟的选择，最后剩下的就是最大的元素。

3.2.3 代码实现

void selectSort(int\* nums, int n, accord& data){

//在没排序元素中找到最小放到最前面

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

//找到最小元素的下标，换到最前面

int minIndex = i;

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

if (++data.cmp, nums[minIndex] > nums[j]){

minIndex = j;

}

//交换

int t = nums[minIndex];

nums[minIndex] = nums[i];

nums[i] = t;

//移动步数

data.move += 3;

}

}

3.2.4 算法分析

选择排序的比较次数与初始排列无关，均为 n(n - 1) / 2，但元素的移动次数与初始排列有关，若最好情况，即正序，移动次数为0；最坏情况，即逆序，为3(n - 1)。要进行n - 1趟选择，每趟选择要进行n - i + 1个元素的比较，因此时间复杂度为O(n^2)。

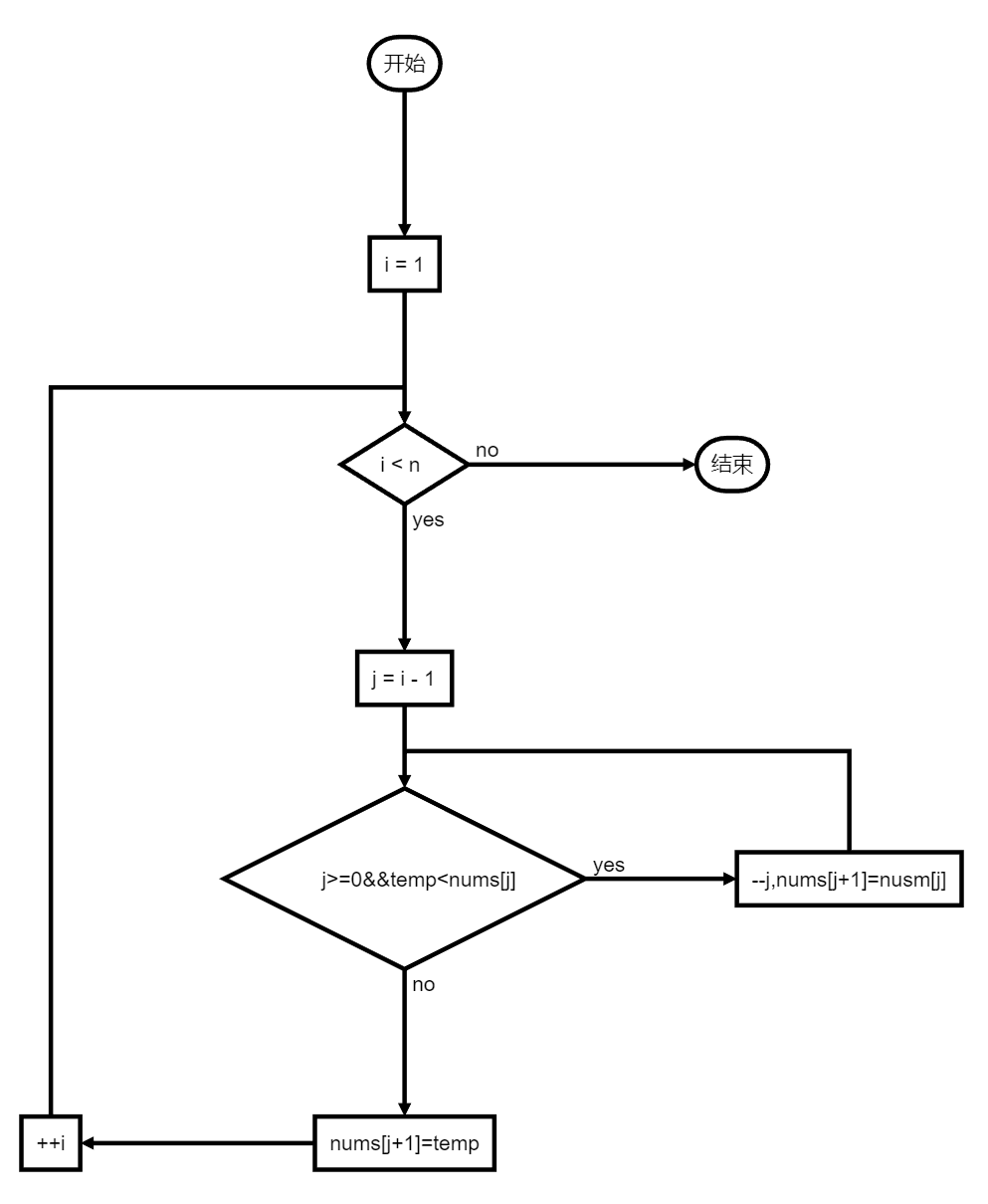
选择排序过程中不需要另外开辟新的数组空间，因此著需要常数个用于交换或标记的变量，所以选择排序是一种原地排序算法，空间复杂度为O(1)。

选择排序是给每个位置选择当前元素最小的，比如给第一个位置选择最小的，在剩余元素里面给第二个元素选择第二小的，依次类推，直到第n-1个元素，第n个元素不用选择了，因为只剩下它一个最大的元素了。那么，在一趟选择，如果一个元素比当前元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那么交换后稳定性就被破坏了。因此，选择排序是一个不稳定的排序算法。

这里利用minIndex记录最小元素的下标，找到之后在进行元素的交换，减少元素的交换次数。

3.3 直接插入排序

3.3.1 流程图



3.3.2 算法说明

直接插入排序的思想为：将未排序的序列中的一个元素按其排序码的大小插入到已经排好序的有序序列中的适当位置，使得有序序列扩大一个元素。以此类推，直到将整个待排序序列中的所有元素插入到有序序列当中。这里采用从后往前寻找插入位置，寻找过程的同时将有序序列后移，由此腾出插入位置。因此我们需要将当前需要插入的元素暂时存入一个临时变量，以减少交换次数。

3.3.3 代码实现

void insertSort(int\* nums, int n, accord& data){

//从当前位置往前扫描，插入正确位置

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

int temp = nums[i];

++data.move;

int j = i - 1;

//在判断语句中记录次数

while (j >= 0 && ( ++data.cmp, temp < nums[j])){

nums[j + 1] = nums[j];//元素往后移，给temp腾出位置

++data.move;

--j;

}

nums[j + 1] = temp;

//移动步数

++data.move;

}

}

3.3.4 算法分析

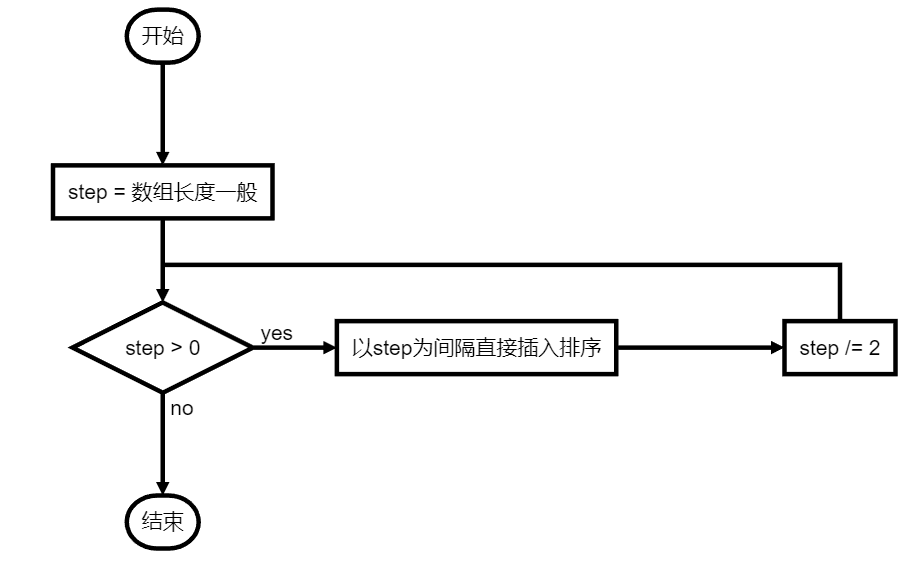
直接插入排序严重依赖于待排序的初始排列，最好情况下，即待排序元素序列中所有元素已经从小到大按正序排列好，每趟插入元素只需要与有序序列的最后一个元素比较一次，因此，总比较次数为n - 1，移动次数为2(n - 1)，时间复杂度为O(1)；反之，最坏情况下，即待排序元素序列中所有元素按逆序排列，需要将它们全部逆转过来，第i趟排序时，将第i个元素与前面i - 1个元素进行比较，总比较次数为n(n - 1) / 2，移动次数为(n + 4)(n - 1) / 2，时间复杂度为O(n^2)。总体看来时间复杂为O(n^2)

该算法只需要一个暂时存放要插入元素的变量，使用常数个临时变量，因此属于原地排序，空间复杂度为O(1)

算法每次比较，发生逆序就把大者往后移，如果排序码相等，不执行这种移动，所以不会发生原来的在后面的反而移动到相等元素前面的现象，因此直接插入排序是一种稳定的排序。

3.4 希尔排序

3.4.1 流程图



3.4.2 流程图

希尔排序又称为缩小增量排序，其基本思想为，每趟按照一个增量gap作为间隔，将全部元素序列分为gap个子序列，所有距离为gap的元素位于同一个子序列当中，在每一个子序列中分别执行直接插入排序。然后缩小增量gap，重复上面子序列的划分和排序工作，最后取gap = 1，将所有元素视为同一个序列，进行排序。由于开始时候gap的取值较大，因此子序列中的元素个数较少，排序的速度比较快，等到排序的后期，随着gap的取值逐渐变小，子序列中元素的个数变多，但由于前面工作的积累，序列逐渐变得有序，因此排序速度同样较快。

3.4.3 代码实现

void shellSort(int\* nums, int n, accord& data){

//有间隔的插入排序，间隔step不断缩小，大致有序->有序

int i, temp;

for (int step = n >> 2; step > 0; step >>= 2)

for (int pos = step; pos < n; ++pos){

temp = nums[pos];//记录要变位置的值

++data.move;

//插入排序

for (i = pos - step; i >= 0 && (++data.cmp, nums[i] > temp); i -= step){

nums[i + step] = nums[i];//前面的元素往后移，给temp腾出位置

//记录步数

++data.move;

}

nums[i + step] = temp;//找到插入位置

//记录步数

++data.move;

}

}

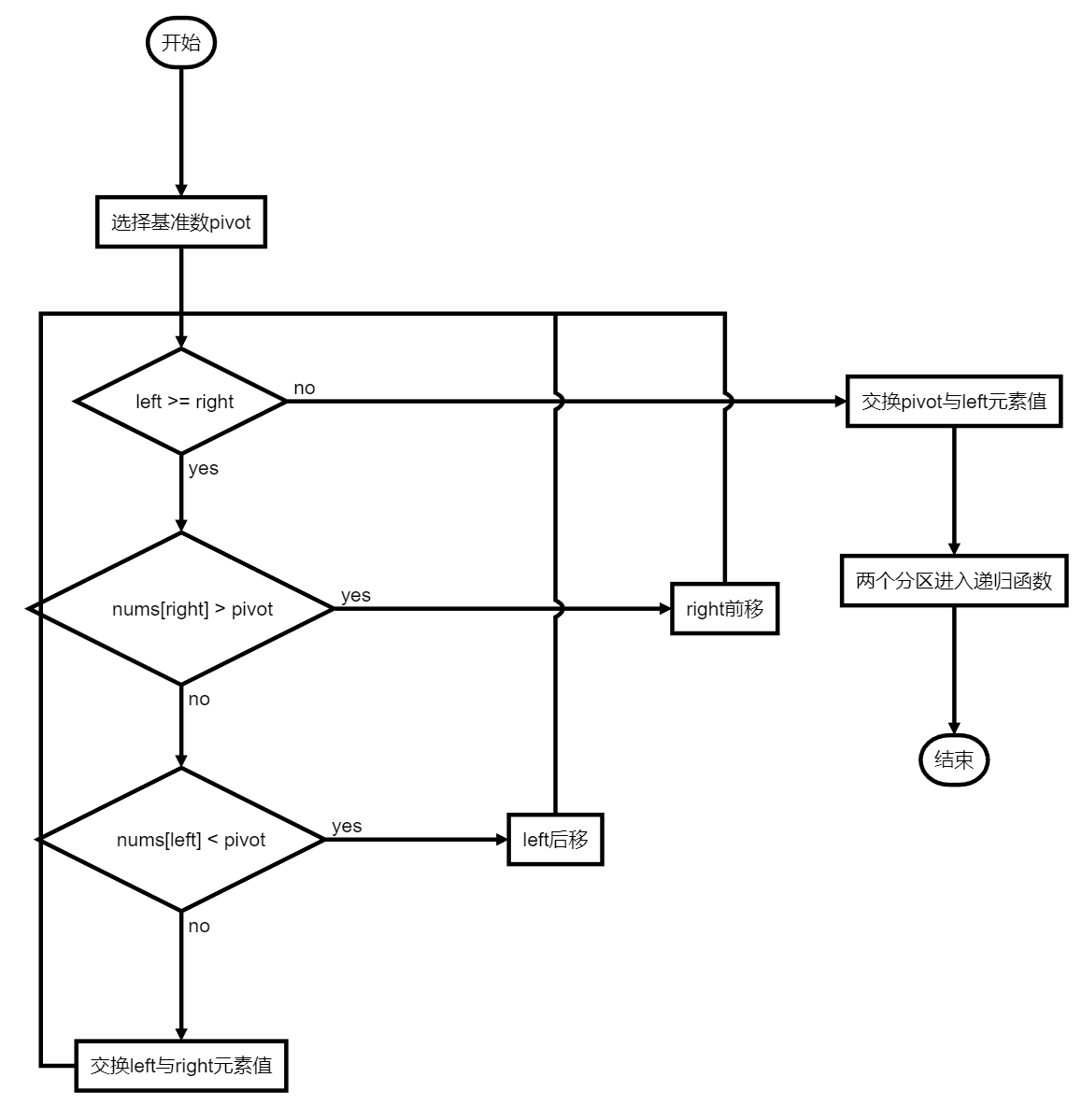
3.4.4 算法分析

希尔排序根据不同的步长进行直接插入排序，因为时跳跃性的比较，因此希尔排序是一种不稳定的排序算法。同时，由于步长小时，排序数量小，速度快；步长大时，基本有序，同样速度快，因此希尔排序具有较高的效率，因为科学家门尚未找出步长如何划分时间效率最佳，因此通过实践估计，希尔排序的时间复杂度在n^1.25 - 1.6n^1.25之间。

该算法只需要一个暂时存放要插入元素的变量，使用常数个临时变量，因此属于原地排序，空间复杂度为O(1)

3.5 快速排序递归方法

3.5.1 流程图



3.5.2 算法说明

快速排序的基本思想为：通过选择一个基准数，再通过一趟排序将待排序的序列分成两个部分，其中一部分的所有元素的排序码都比基准值小，另一部分的所有元素的排序码都比基准值大。再利用分治的思想，使得两部分序列进入递归函数，重复上述步骤，当待排序蓄力中只剩下一个元素的时候递归结束，即完成从上至下的排序。其中关于分区的方法，本算法采用的是设置两个指针，left指向数组除了pivot的第一个数据，right指向数组的最后一个数据。首先向前移动right指针，若遇到比pivot还小的数据则停止，接着移动left指针，遇到比pivot还大的元素停止。待两个指针都停止之后，交换两指针所指向的元素的值。接着重复上述步骤，直到左右指针相遇，标志着找到了pivot最终的位置，交换，结束本趟分区操作。

3.5.3 代码实现

//快排需要的插入函数，[beg, end]

void insertSort(int\* nums, int beg, int end, accord& data){

//计算数组大小

int n = end - beg + 1;

//从当前位置往前扫描，插入正确位置

for (int i = beg + 1; i <= end; ++i){

int temp = nums[i];

++data.move;

int j = i - 1;

//在判断语句中记录次数

while (j >= beg && (++data.cmp, temp < nums[j])){

nums[j + 1] = nums[j];//元素往后移，给temp腾出位置

++data.move;

--j;

}

nums[j + 1] = temp;

//移动步数

++data.move;

}

}

//beg为第一个下标，end为最后一个的下标，[beg, end]

void \_quickSort1(int\* nums, int beg, int end, int n, accord& data){

if (beg >= end)//递归结束条件

return;

//数据量较小时采用插入排序

if (end - beg + 1 <= 20){

insertSort(nums, beg, end, data);

return;

}

//取左边第一个为基准

int pivot = nums[beg];

//两个指针左右检查比pivot小的放左边，否则右边

int left = beg, right = end;

while (left < right){

//不超范围，不比pivot小的一直移动右边指针

while (left < right &&(++data.cmp, nums[right] >= pivot))

--right;

//与左边指针交换位置值

nums[left] = nums[right];

//移动步数

++data.move;

//不超范围，不必pivot大的一直移动左边指针

while (left < right && (++data.cmp, nums[left] <= pivot))

++left;

//交换位置

nums[right] = nums[left];

//移动步数

++data.move;

}

//出循环left == right，找到最开始pivot的位置

nums[left] = pivot;

//移动步数

++data.move;

//分治为左右两边

\_quickSort1(nums, beg, left - 1, n, data);

\_quickSort1(nums, left + 1, end, n, data);

}

3.5.4 算法分析

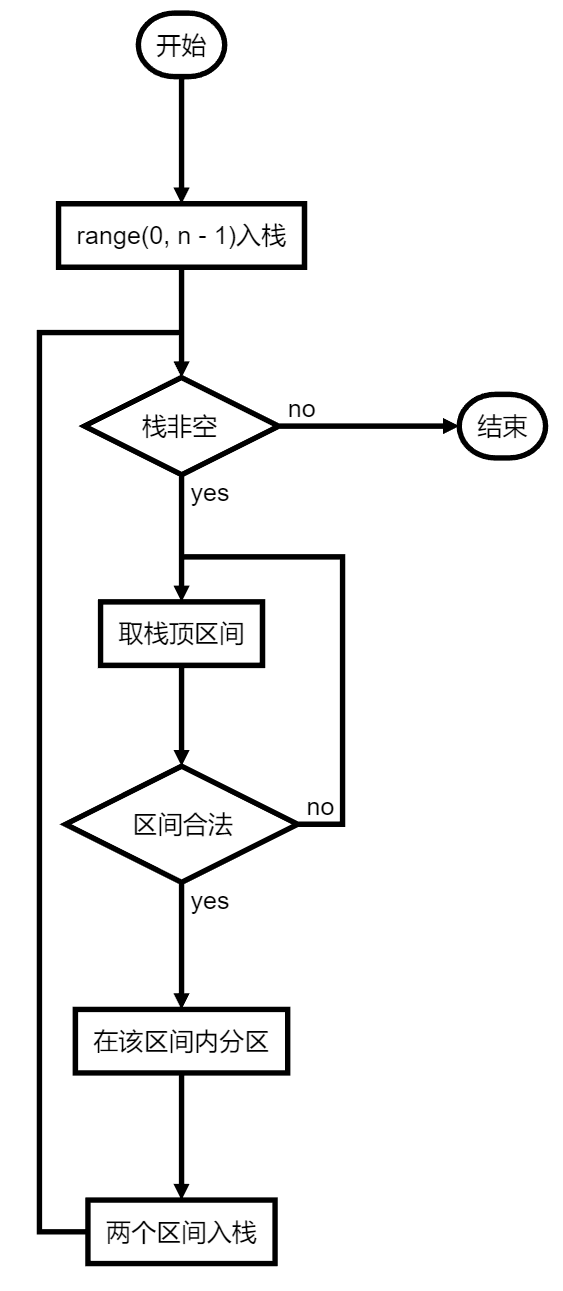
快速排序的平均时间复杂度为O(nlogn)，因此，该排序算法被认为是目前最好的一种内部排序算法。但是在最坏情况下，即在待排序序列最初有序的情况下，快速排序会退化为O(n^2)。

因为是跳跃性的比较元素排序码，所以快速排序是一种不稳定的排序算法。

快速排序需要栈空间进行递归操作，每趟都把序列分成两部分，平均下来空间复杂度为O(logn)；但在最坏情况下，空间复杂度可能达到O(n)。

3.6 快速排序非递归方法

3.6.1 流程图



3.6.2 算法说明

快速排序递归算法中递归操作会占用较多资源，在实际应用中，我们会尽量不去使用递归算法。因此，我们需要将递归展开，实现非递归算法，本算法利用栈的特点，保存需要进行分区的区间边界信息，由此模拟递归的执行顺序。

3.6.3 代码实现

//二元结构体存储数组范围信息

struct RANGE {

int beg, end;

//构造函数

RANGE(const int beg = 0, const int end = 0)

:beg(beg), end(end) {}

};

//快速排序非递归算法

void quickSort2(int\* nums, int n, accord& data){

//判断数组长度是否合法，申请空间时候需要

if (n <= 0)

return;

RANGE\* stk = new(nothrow) RANGE[n];

//保证申请到空间了才能继续运行

assert(stk != NULL);

//栈顶指针

int top = 0;

//最大范围入栈

stk[top++] = RANGE(0, n - 1);

//辅助变量

int pivot = -1, left = -1, right = -1;

while (top){

RANGE range = stk[--top];

//无需排序

if (range.beg >= range.end)

continue;

//取左边第一个为基准

pivot = nums[range.beg];

//两个指针左右检查比pivot小的放左边，否则右边

left = range.beg, right = range.end;

while (left < right){

//不超范围，不比pivot小的一直移动右边指针

while (left < right && (++data.cmp, nums[right] >= pivot))

--right;

//与左边指针交换位置值

nums[left] = nums[right];

//移动步数

++data.move;

//不超范围，不必pivot大的一直移动左边指针

while (left < right && (++data.cmp, nums[left] <= pivot))

++left;

//交换位置

nums[right] = nums[left];

//移动步数

++data.move;

}

//出循环left == right，找到最开始pivot的位置

nums[left] = pivot;

//移动步数

++data.move;

//入栈

stk[top++] = RANGE(range.beg, left - 1);

stk[top++] = RANGE(left + 1, range.end);

}

//释放空间

delete[] stk;

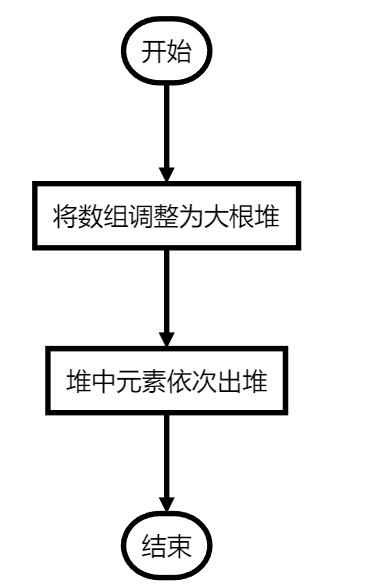
}

3.6.4 算法分析

本算法利用栈这种数据结构，将递归展开。栈中存储需要排序的数据序列的范围的上下界。一定程度上避免了递归的调用，优化了算法的时间和空间效率。

3.7 堆排序非递归方法

3.7.1 流程图



3.7.2 算法说明

堆排序是利用堆这种数据结构进行的排序。为了得到正序序列，本算法使用的是大根堆。所谓大根堆就是一个完全二叉树，并且每个结点的值都大于等于其左右孩子的值。将整个有序序列调整为一个大根堆后，执行出堆操作，依次将堆中排序码最大的元素调整到数组最后，以此类推，从而得到从小到大的正序序列。

3.7.3 代码实现

void heapify1(int\* nums, int start, int end, accord& data)//[beg, end]{

int dad = start, son = 2 \* dad + 1;

while (son <= end){

//取大的那个儿子

if ((son + 1) <= end && (++data.cmp, nums[son] < nums[son + 1]))

++son;

if (++data.cmp, nums[dad] > nums[son])//dad比son大，直接结束，因为要的是top最大

return;

//小的nums[dad]下沉到较大的子结点的位置

int t = nums[dad];

nums[dad] = nums[son];

nums[son] = t;

//记录步数

data.move += 3;

dad = son;//迭代到更大的子结点

son = dad \* 2 + 1;

}

}

void heapSort1(int\* nums, int n, accord& data){

//初始化堆，从最后一个非叶子结点往前

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

heapify1(nums, i, n - 1, data);//下标从[i, n - 1]

//堆排序，将已经排好的堆 top 放到最后（相当于出堆）

for (int i = n - 1; i > 0; --i)//从后往前放top

{

int t = nums[0];

nums[0] = nums[i];

nums[i] = t;

data.move += 3;

heapify1(nums, 0, i - 1, data);//放到最后了，重新堆化

}

}

3.7.4 算法分析

堆排序作为一种选择排序，整体主要有初始化堆和出堆操作组成。其中初始化堆的时间复杂度为O(n)，进行n - 1次出堆的操作，每次出堆的时间复杂度为O(logn)，因此该算法整体的时间复杂度为O(nlogn)。

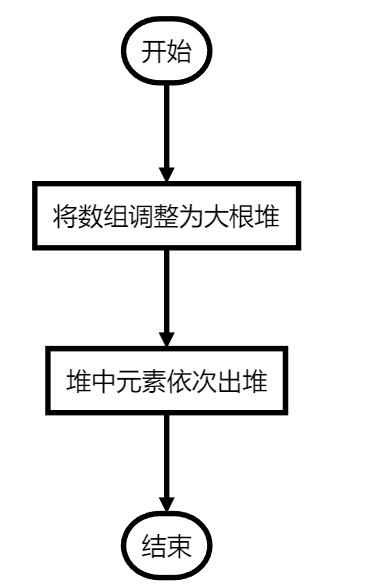
由于是跳跃性地比较元素的排序码，因此，堆排序算法是一种不稳定排序。

该算法只需要一个暂时存放要插入元素的变量，使用常数个临时变量，因此属于原地排序，空间复杂度为O(1)

非递归方法利用迭代循环将递归展开，一定程度少优化了时间和空间的效率。

3.8 堆排序递归方法

3.8.1 流程图



3.8.2 算法说明

递归方法与非递归方法的不同在于对调整为堆的处理，直接将本轮函数的父节点作为下一次调整的子结点，由此实现向上调整。此过程中我们需要注意的是，计算父节点的同时，应该考虑到数组越界问题。

3.8.3 代码实现

void heapify2(int\* nums, int start, int end, accord& data){

int dad = start, son = 2 \* dad + 1;

if (son > end)//递归结束条件

return;

//取较大儿子

if (son + 1 <= end && (++data.cmp, nums[son] < nums[son + 1]))

++son;

//dad已经比son大了就不用下沉了

if (++data.cmp, nums[dad] > nums[son])

return;

int t = nums[dad];

nums[dad] = nums[son];

nums[son] = t;

//记录步数

data.move += 3;

//进入递归函数，以son作为下一个的start

heapify2(nums, son, end, data);

}

void heapSort2(int\* nums, int n, accord& data){

//初始化堆，从中间往前

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

heapify2(nums, i, n - 1, data);//下标从[i, n - 1]

//堆排序，将已经排好的堆 top 放到最后（相当于出堆）

for (int i = n - 1; i > 0; --i)//从后往前放top

{

int t = nums[0];

nums[0] = nums[i];

nums[i] = t;

//记录步数

data.move += 3;

heapify2(nums, 0, i - 1, data);//放到最后了，重新堆化

}

}

3.8.4 算法分析

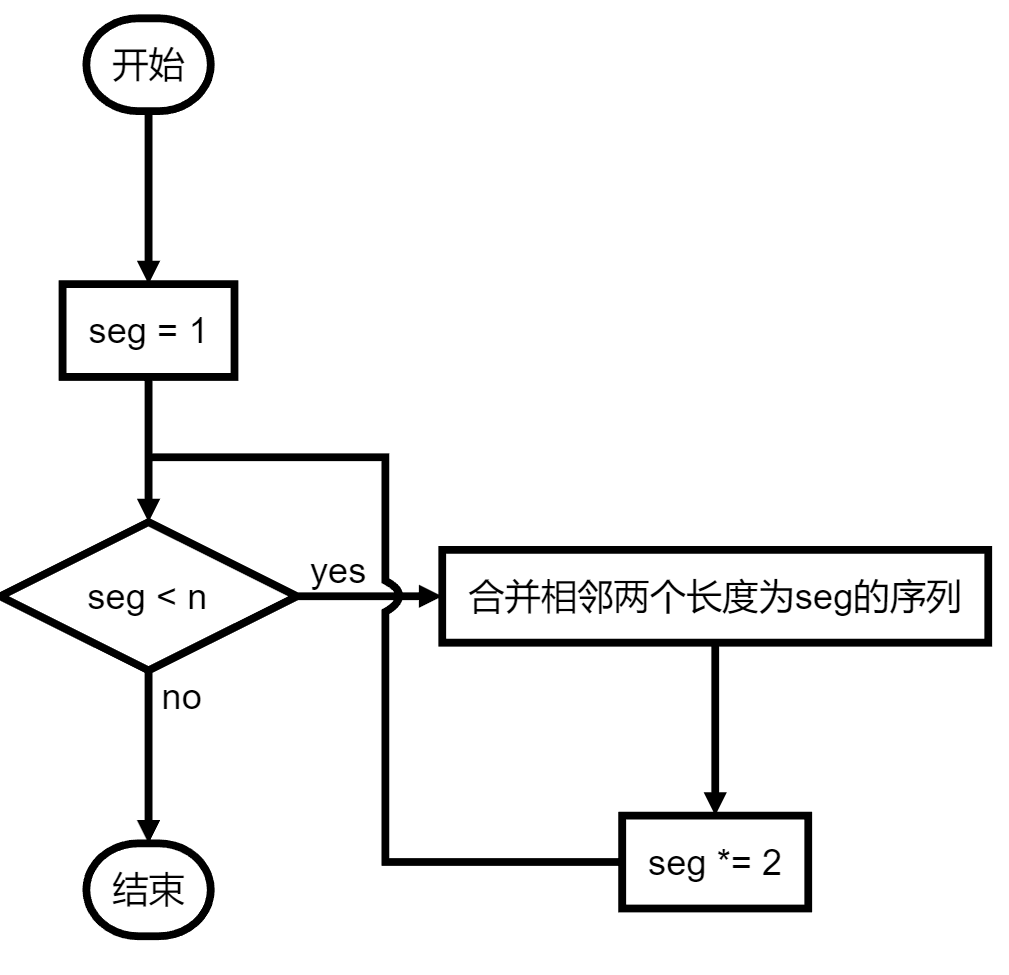
堆排序作为一种选择排序，整体主要有初始化堆和出堆操作组成。其中初始化堆的时间复杂度为O(n)，进行n - 1次出堆的操作，每次出堆的时间复杂度为O(logn)，因此该算法整体的时间复杂度为O(nlogn)。

由于是跳跃性地比较元素的排序码，因此，堆排序算法是一种不稳定排序。

该算法只需要一个暂时存放要插入元素的变量，使用常数个临时变量，因此属于原地排序，空间复杂度为O(1)。

3.9 归并排序

3.9.1 流程图



3.9.2 算法说明

归并排序的主要思想为：将两段有序的序列依次合并为一段有序的序列，该过程的时间复杂度为线性级别。而对于如何获得两段有序的序列，这时候可以考虑分治的思想，与快速排序分治思想不同，此处是严格的平均的二分，因此不会出现快速排序退化的问题。利用递归或者递归展开的方法，由下至上地实现每一小段的归并排序。本算法采用的是非递归方法，设置seg = 1，作为整个序列中区分的小段的长度，进行合并，然后将seg \*= 2，逐渐扩大，最终每次执行的都是合并两段有序序列。

3.9.3 代码实现

void mergeSort(int\* nums, int n, accord& data)//自下而上迭代方式实现

{

//辅助数组，其中a是更新前的数组，b是更新后的数组

int\* a = nums;

int\* b = new int[n];

//申请空间不成功

assert(b != NULL);

//从下至上，区间不断扩大

for (int seg = 1; seg < n; seg <<= 1){

for (int start = 0; start < n; start += seg + seg)//一次比较两个seg

{

int start1 = start, start2 = min(start + seg, n);

int end1 = start2, end2 = min(start + seg + seg, n);

int index = start1;

while (start1 < end1 && start2 < end2)//选到谁谁就指针往后移

{

b[index++] = a[start1] < a[start2] ? a[start1++] : a[start2++];

++data.cmp;

++data.move;

}

//比完了剩下的直接放到后面

while (start1 < end1)

b[index++] = a[start1++], ++data.move;

while (start2 < end2)

b[index++] = a[start2++], ++data.move;

}

//保持a是更新前的数组，b是更新后的数组

int\* t = a;

a = b;

b = t;

}

//如果没有换回来（首地址），此时a是申请空间的那个，b是nums

if (a != nums)

{

for (int i = 0; i < n; ++i)//最终的答案数组赋值给nums

b[i] = a[i];

b = a;//申请空间的给b

}

//如果换回来了就不用赋值了

//释放申请的空间

delete[] b;

}

3.9.4 算法分析

归并排序通过分治思想，将待排序序列不断划分，直到只剩下一个或零个元素。接着进行两段有序序列的合并，其中的时间复杂度为O(n)，将序列划分的时间复杂的一般通过递归实现，因此该算法的总体时间复杂度为O(nlogn)。

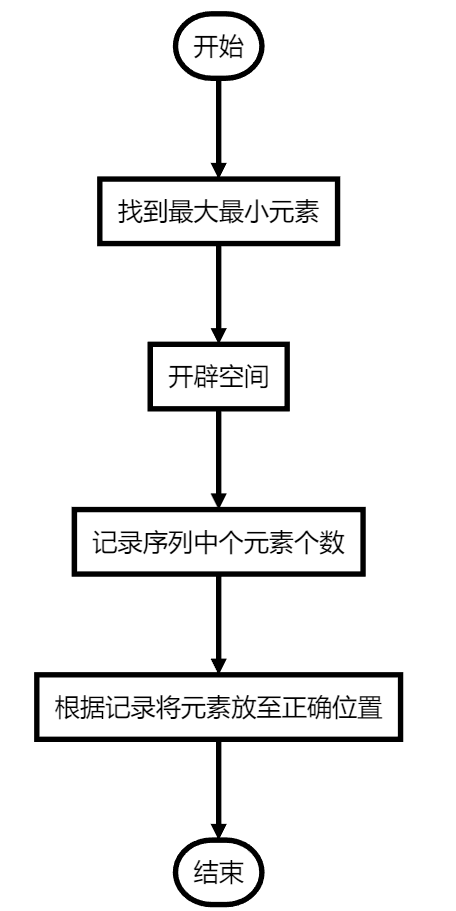
同时，归并排序是一种稳定排序算法。

但是归并排序同样有其不足，它需要辅助数组暂时存储合并的结果，因此需要的空间复杂度为O(n)。

本算法将递归展开，利用seg表示划分的区间，实现由下到上的合并操作。一定程度上优化了时间和空间上的效率。

3.10 计数排序

3.10.1 流程图



3.10.2 算法说明

所谓计数排序，与上述排序的思想不同，这并不是基于比较的方法，而是开辟一段足够大的数组空间，记录未排序序列中各个数据的排序码的个数，最后遍历整个记录数组，依次将数据放回原序列的正确位置上。这是一种牺牲空间换取时间的算法，因此在数据比较大，而基本没有重复排序码的数据时，反而会降低速度。

3.10.3 代码实现

void countingSort(int\* nums, int n, accord& data){

//额外开辟数组空间记录值为i的元素出现的次数

//先找到最值两个边界

int Max = INT\_MIN, Min = INT\_MAX;

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

if (Max < nums[i])

Max = nums[i];

if (Min > nums[i])

Min = nums[i];

}

int\* temp = new int[Max - Min + 1];

assert(temp != NULL);

//申请空间失败

if (temp == NULL)

return;

//初始化

for (int i = 0; i < Max - Min + 1; ++i)

temp[i] = 0;

//记录出现次数

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

++temp[nums[i] - Min];

for (int i = 0, j = 0; i < Max - Min + 1; ){

if (temp[i])//里面还有元素

{

nums[j++] = i + Min;//元素的值

++data.move;

--temp[i];

}

else//没有元素了

++i;

}

delete[] temp;

}

3.10.4 算法分析

计数排序并不是一种基于比较的排序算法，它的优势在于对一定范围内的整数排序，由于对记录数组的遍历，其时间复杂度为O(n + k)，其中k表示为整数的范围。这快于上述任何一种比较算法，但牺牲了一定的空间。

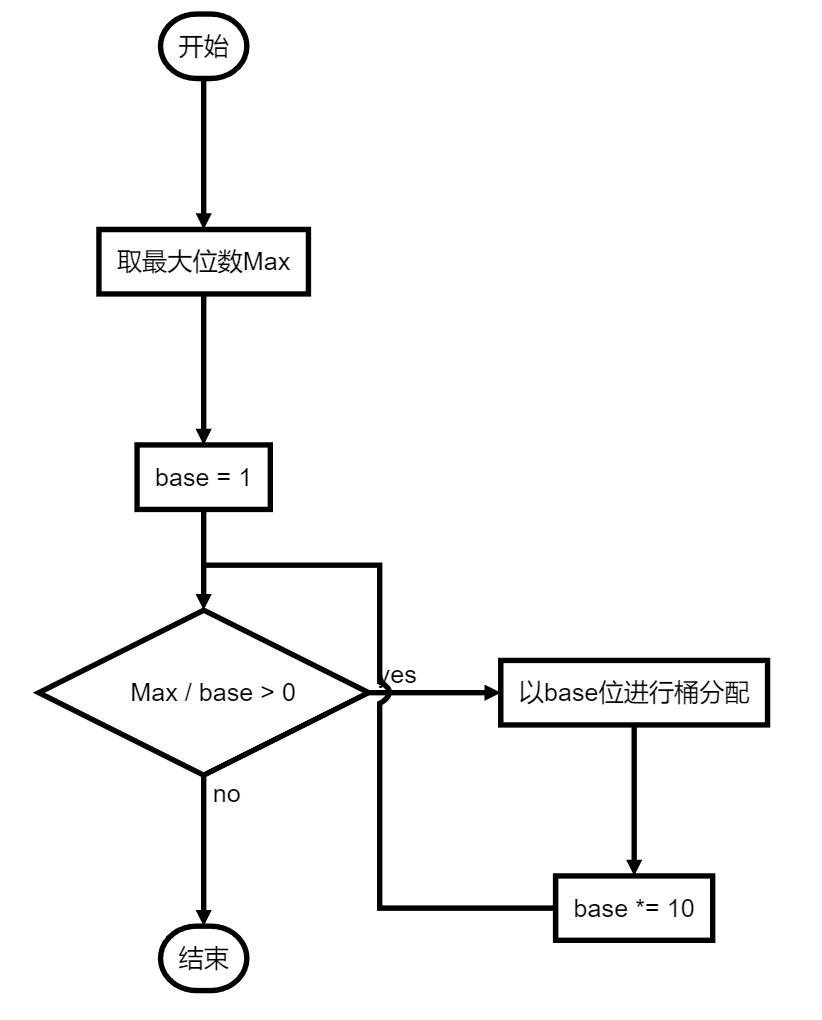
本算法中对于整数范围的处理为，找出最小值和最大值，开辟这一段大小的数组空间记录元素排序码出现的次数，形成排序码到出现次数的映射。利用对最小值的偏移量形成在记录数组的下标。通过这种方法可以应程度上减少空间的浪费。

基数排序是一种稳定的排序算法：由于统计数组可以知道该索引在原数组中排第几位，相同的元素其在原数组中排列在后面，其从原数组的后面遍历，其在最终数组中的索引也在后面，所以相同的元素其相对位置不会改变。

但是对于负数、小数等情况，计数排序可能并不能发挥出它的优势。

3.11 基数排序LSD

3.11.1 流程图



3.11.2 算法说明

基数排序是一种基于多关键码的排序，并且一个数据中的关键码具有优先级。对于LSD算法而言，从优先级小的关键码开始，通过关键码将元素分配到不同的“桶”中，然后再将桶中的元素放回原序列中，完成一趟分配，接着按照第二小的关键码分配，重复上述步骤，直到按照最大的关键码分配，整个数组呈现有序。本算法适用于一个元素关键码较少的序列，在此情况下可以发挥较大的优势，拥有较好的时间效率；若一个元素关键码较多，则下面的MSD算法可能会有更好的时间效率。对于数字来说，我们见个十百千万等等位数上的值做欸不同的关键码进行比较。

3.11.3 代码实现

//LSD辅助函数

void bucket(int\* nums, int n, int base, accord& data){

const int BUCKETNUM = 10;

int\* bucket = new int[BUCKETNUM];

assert(bucket != NULL);

int\* temp = new int[n];

assert(temp != NULL);

//桶的初始化

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

bucket[i] = 0;

//记录每位数字对应的桶

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

++bucket[(nums[i] / base) % 10];

//bucket[i]表示i前面元素个数

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

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

//从后往前将元素放入temp，得到的就是按某位数字排好的

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

int index = (nums[i] / base) % 10;

temp[bucket[index] - 1] = nums[i];//因为桶里面记录的个数，-1转换为下标

--bucket[index];//用掉一个元素减去一个

++data.move;

}

//将temp中按某位数字排好的元素放回nums

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

nums[i] = temp[i];

data.move += n;

//释放申请的动态空间

delete[] temp;

delete[] bucket;

}

//基数排序LSD，适用于位数较少的数列

void radixSortLSD(int\* nums, int n, accord& data){

int Max = INT\_MIN;

//获得最大值

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

if (++data.cmp && nums[i] > Max)

Max = nums[i];

//base从个位增大到元素中的最大位数

for (int base = 1; Max / base > 0; base \*= 10)

bucket(nums, n, base, data);

}

3.11.4 算法分析

LSD基数排序算法，对于整数，从低位向高位比较。需要进行最大位数次数的收集，每次收集需要遍历整个数组，因此该算法整体的时间复杂度为O(dn)，其中d表示为整形的为高位数。

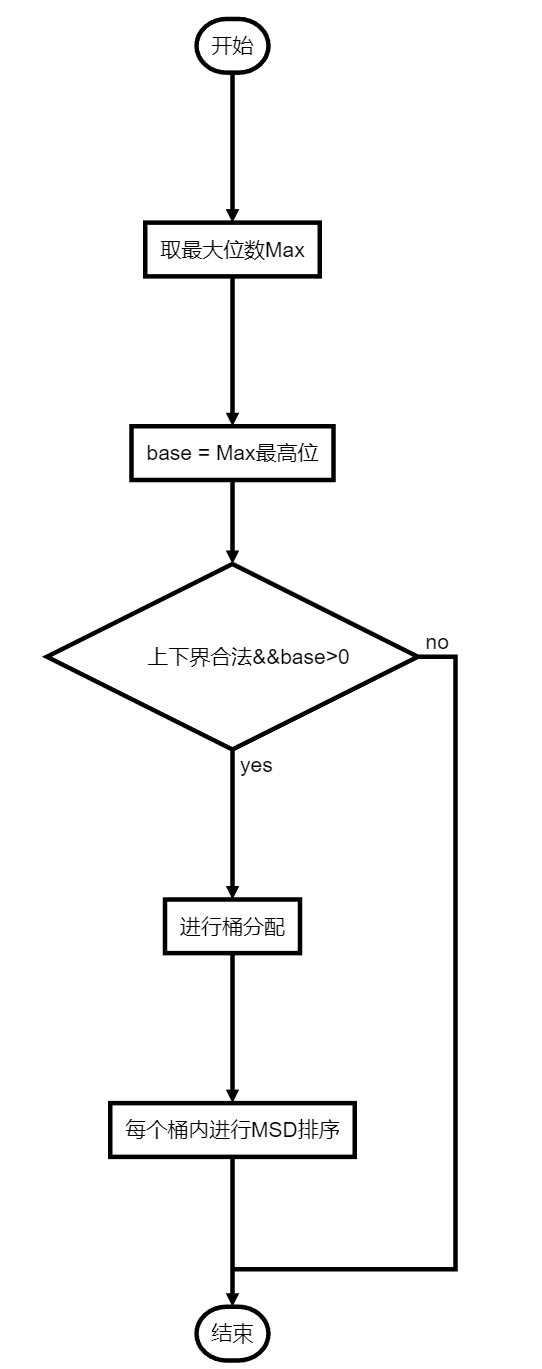
对于空间的消耗，需要一个桶记录每个元素经过一趟排序后的位置，需要辅助数组暂时保存排序后数组的记录，因此空间复杂度为O(n)。这里体现出对该算法的优化，原因在于，原本需要n\*n大小的二维数组作为桶收集各个元素。但本算法使用了bucket数组，首先记录每个排序码对应的元素的个数，然后求前缀和，由此得到每段元素对应排序码的正确位置。遍历原数组，遇到一个元素将其放置于正确位置后，bucket[i]--，以便获得下一个有相同关键码的元素的目标位置。当然我们也可通过类似邻接表的形式实现元素的收集与分配。

本质上LSD算法属于一种稳定的排序算法，在收集和分配的过程中，数组符合先入先出的关系。

该算法适用于位数小的排序，当位数较大时，采用MSD算法可以获得更高的效率。

3.12 基数排序MSD

3.12.1 算法说明



3.12.2 流程图

与LSD算法的不同在于，MSD算法按照元素关键码从大到小进行“桶”的分配，并且分配完成后并不是对整个序列进行重新分配，而是在桶中建立新的“子桶”，由此从上至下地进行分配，最后同样可以达到排序的效果。对于数字来说，我们见个十百千万等等位数上的值做欸不同的关键码进行比较。

3.12.3 代码实现

//基数排序MSD，适用于位数较多的数列

void MSDhelper(int\* nums, int beg, int end, int base, accord& data){

//桶中元素不大于1个

if (beg + 1 >= end)

return;

//基数不合法

if (base <= 0)

return;

const int BUCKETNUM = 10;

int Size = end - beg + 1;

//申请空间

int\* bucket = new int[BUCKETNUM];

int\* temp = new int[Size];

assert(bucket != NULL && temp != NULL);

//bucket数组初始化

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

bucket[i] = 0;

//记录base位的个数

for (int i = beg; i <= end; ++i)

++bucket[(nums[i] / base) % 10];

//求bucket前缀和以得到目标下标

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

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

//放入temp中的目标位置

for (int i = beg; i <= end; ++i){

int index = (nums[i] / base) % 10;

temp[bucket[index] - 1] = nums[i];

--bucket[index];

++data.move;

}

//temp数组中排好的序列放回nums数组

for (int i = beg; i <= end; ++i)

nums[i] = temp[i - beg];

data.move += Size;

//此时bucket[i]存放base位为i的，在temp数组中的第一个位置

//进入递归函数

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

MSDhelper(nums, beg + bucket[i], beg + bucket[i + 1] - 1, base / 10, data);

//最后一部分的范围上界为end

MSDhelper(nums, beg + bucket[BUCKETNUM - 1], end, base / 10, data);

//释放空间

delete[] temp;

delete[] bucket;

}

void radixSortMSD(int\* nums, int n, accord& data){

int Max = INT\_MIN;

//获得最大值

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

if (++data.cmp, nums[i] > Max)

Max = nums[i];

//获得最大位数

int base = 1;

while (Max / base > 10)

base \*= 10;

MSDhelper(nums, 0, n - 1, base, data);

}

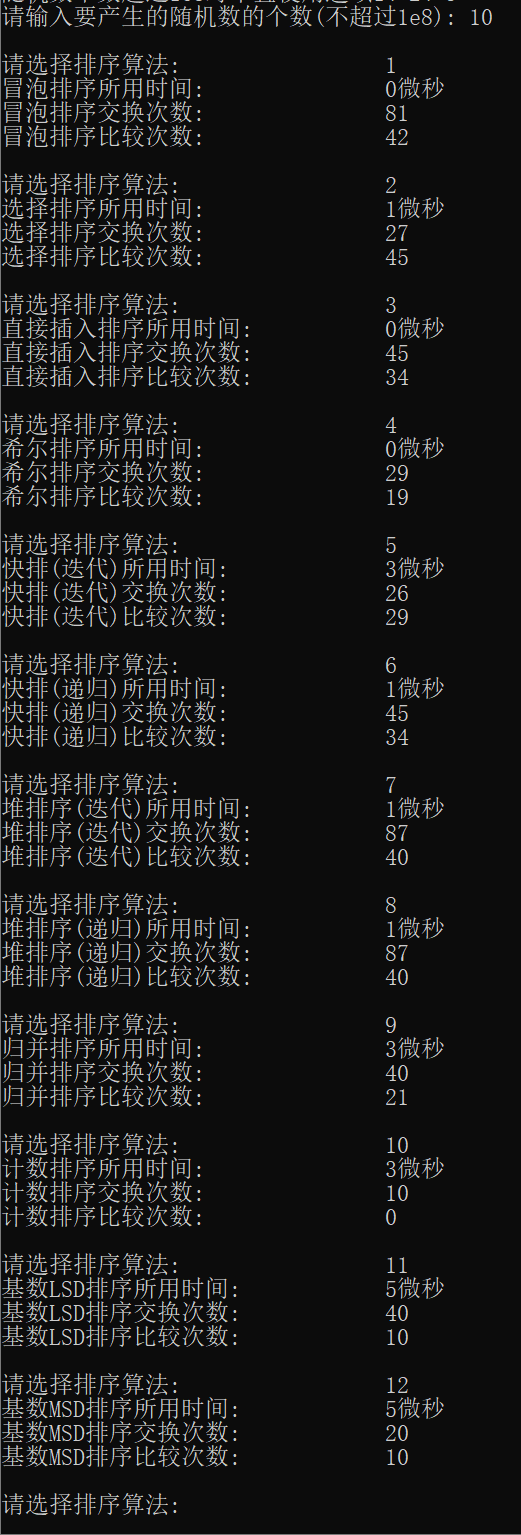
3.12.4 算法分析

高位优先排序的MSD排序算法于LSD相反，按照优先级高的排序码进行收集和分配，因此我们只需要在桶中收集的元素递归进行MSD排序。

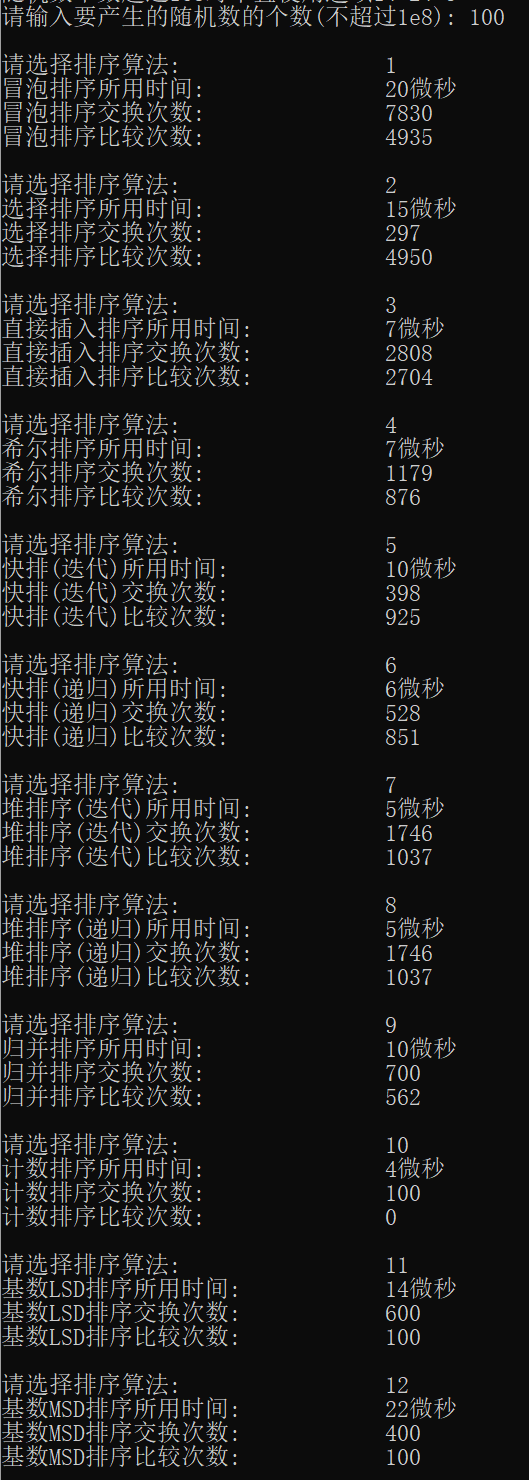
其余形式与LSD大致相同，不再过多阐述。

4 项目测试

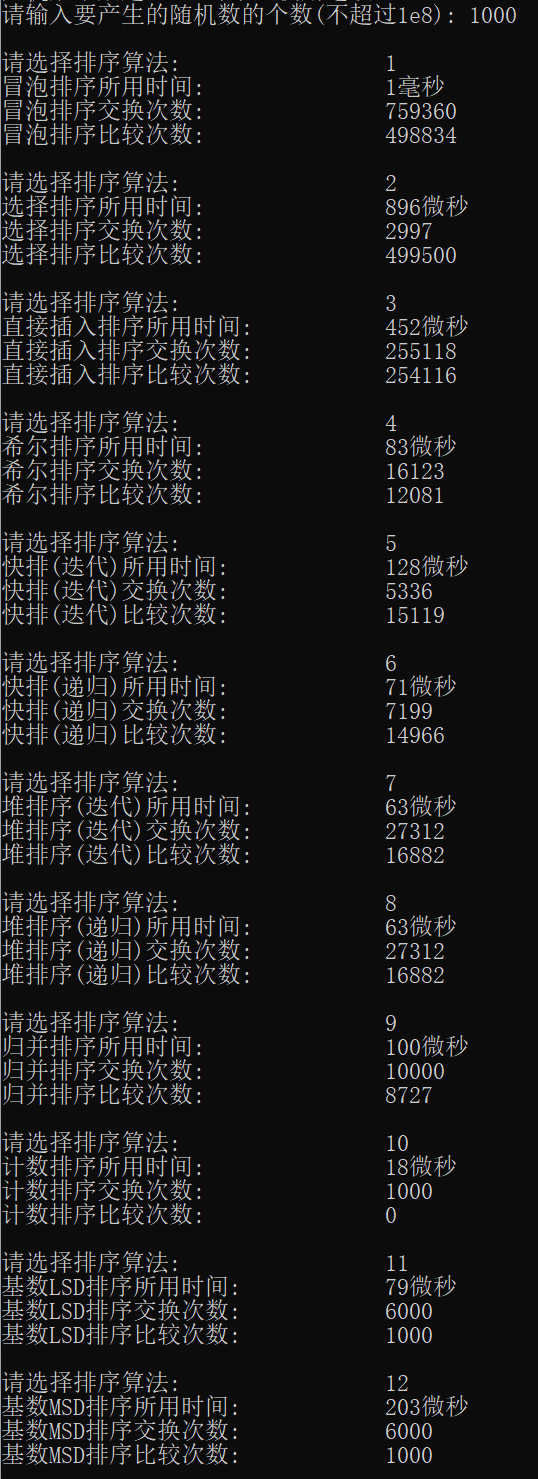
4.1 10个数字测试



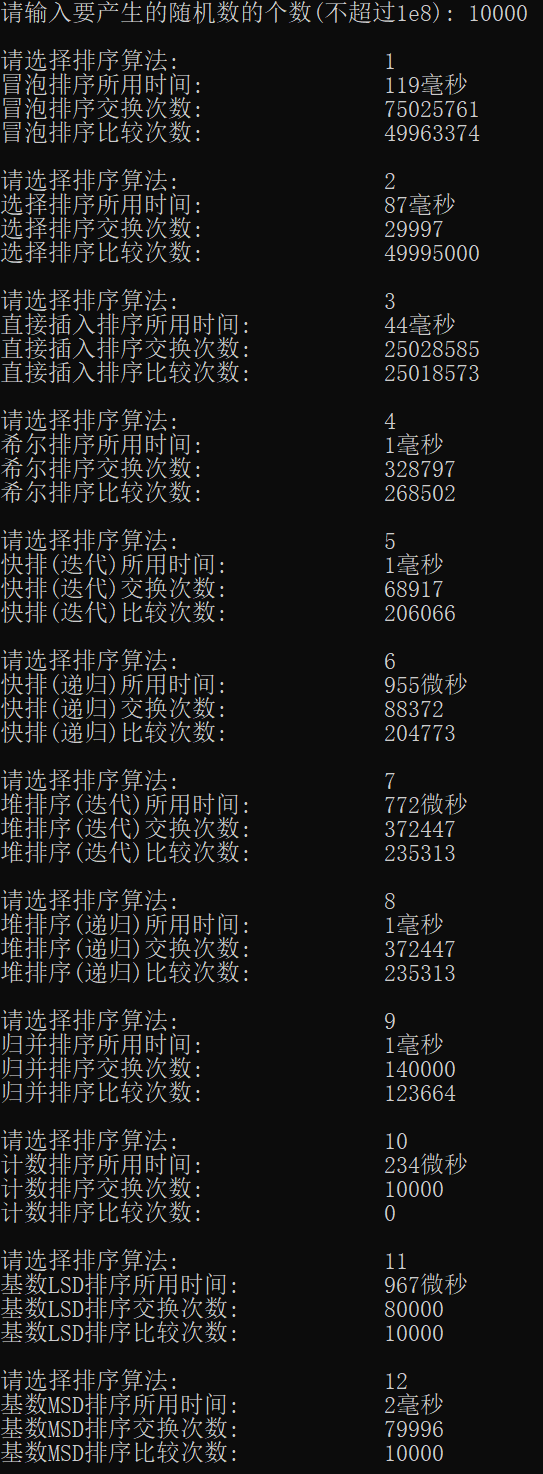
4.2 100个数字测试



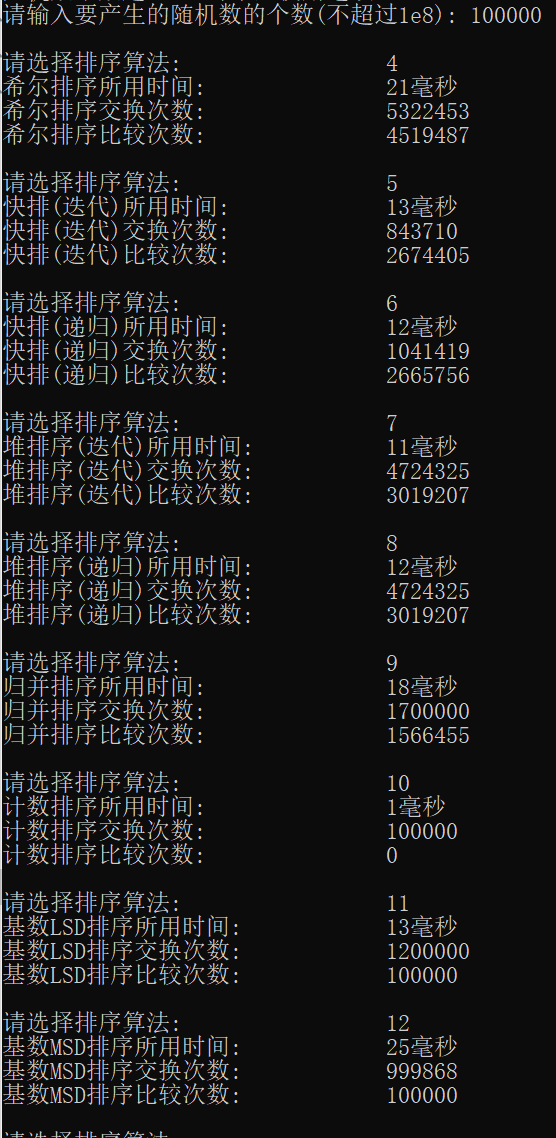
4.3 1000个数字测试



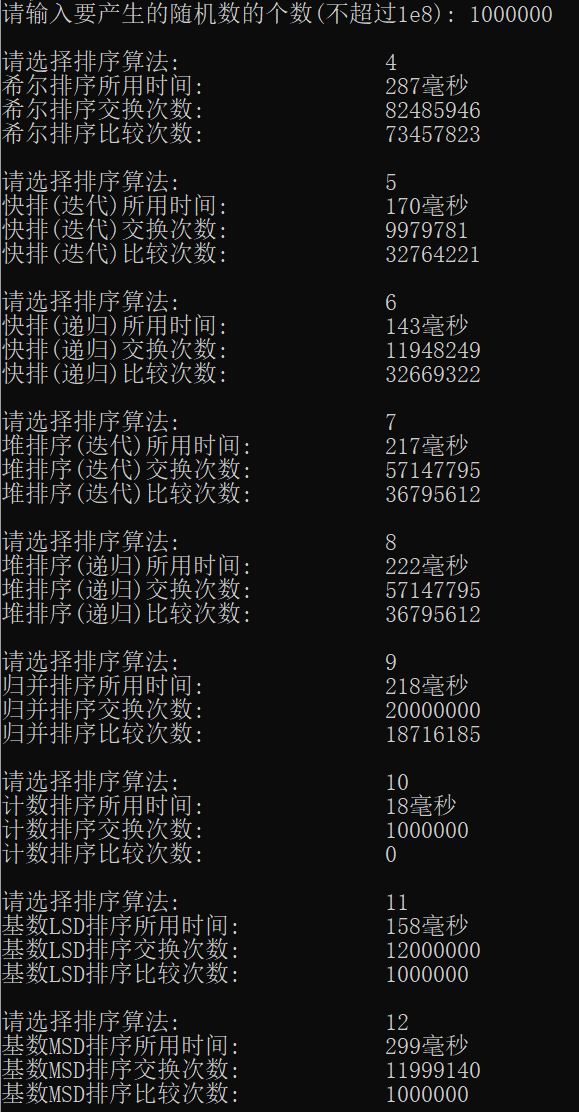
4.4 10000个数字测试



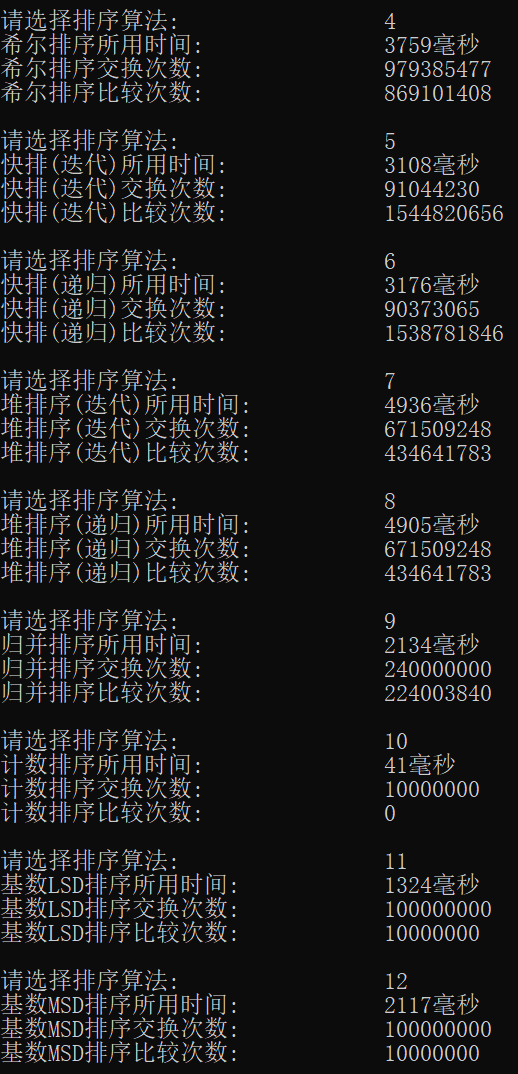
4.5 100000个数字测试



4.6 1000000个数字测试

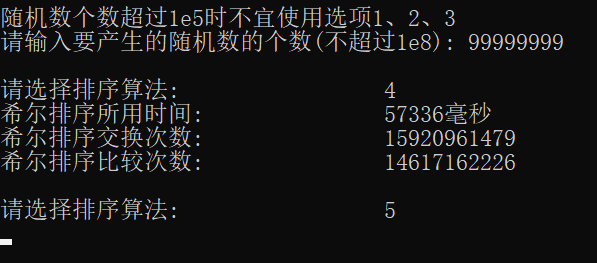


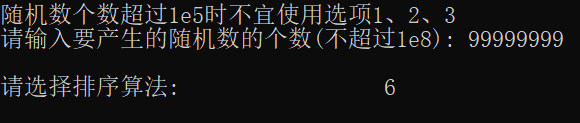
4.7 10000000个数字测试

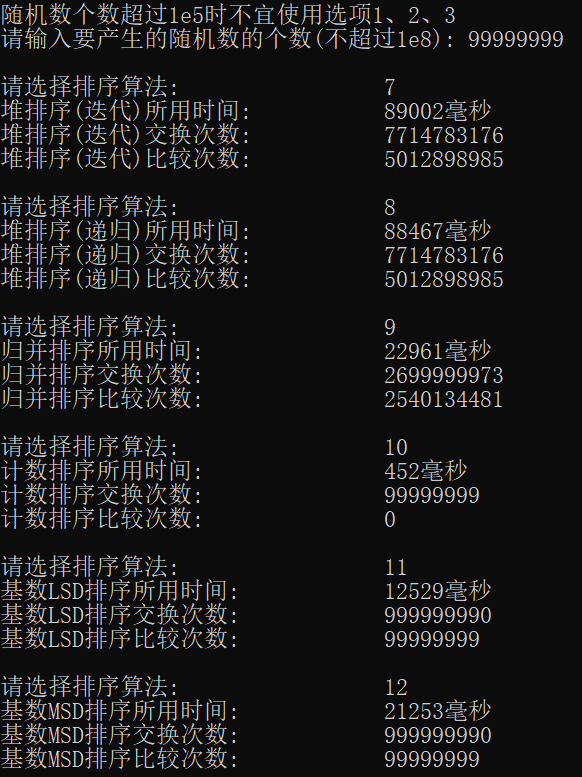


4.8 99999999个数字测试

4.8.1 利用rand()生成随机数

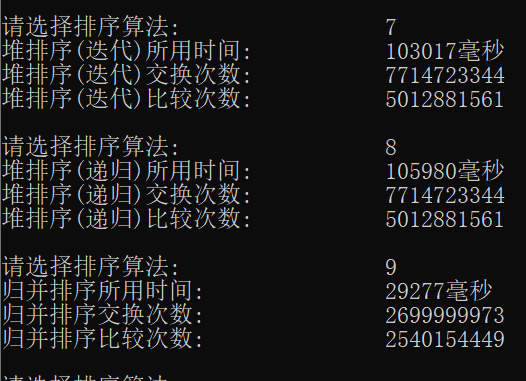




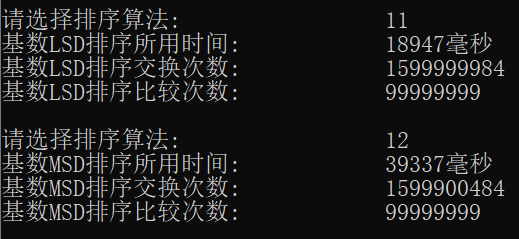


4.8.2 利用<random>中的函数生成随机数

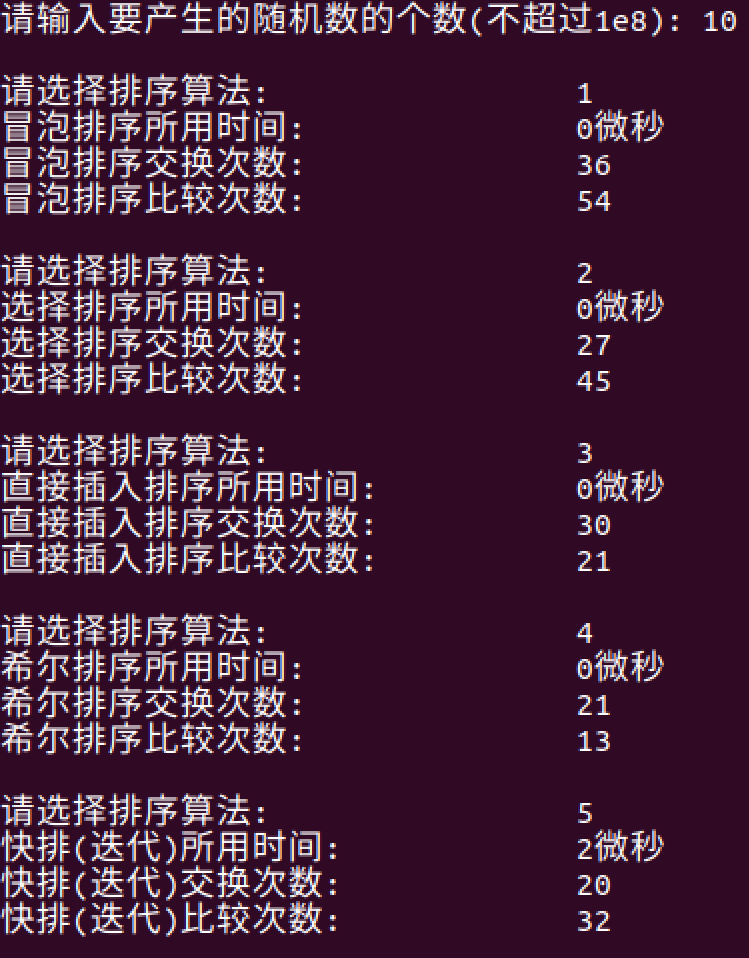


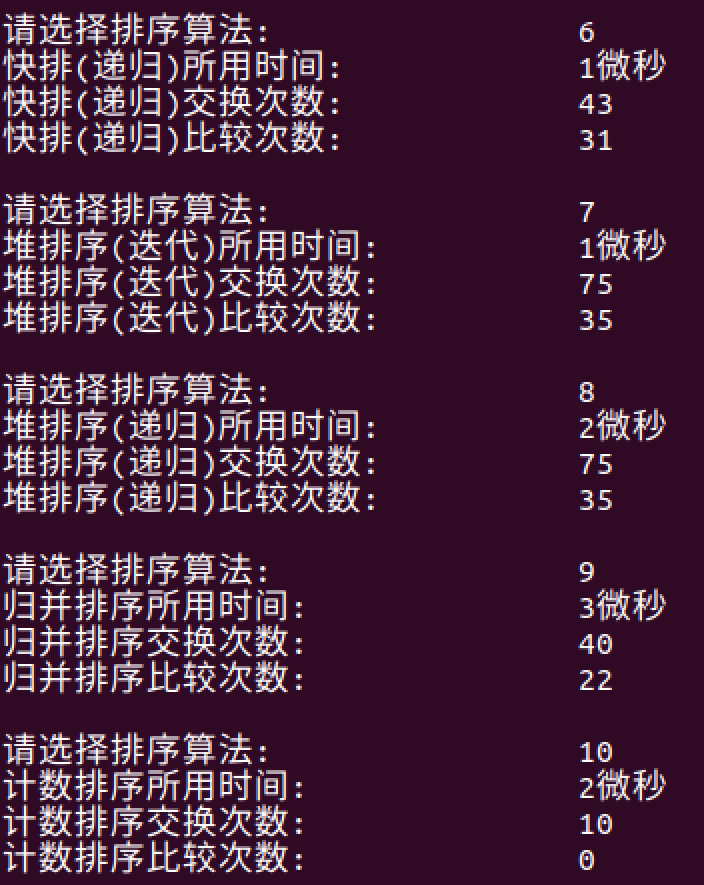


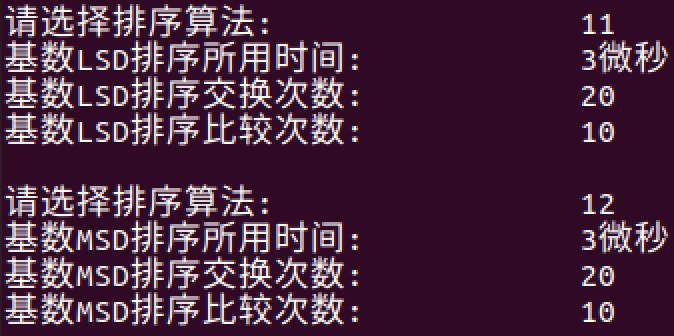




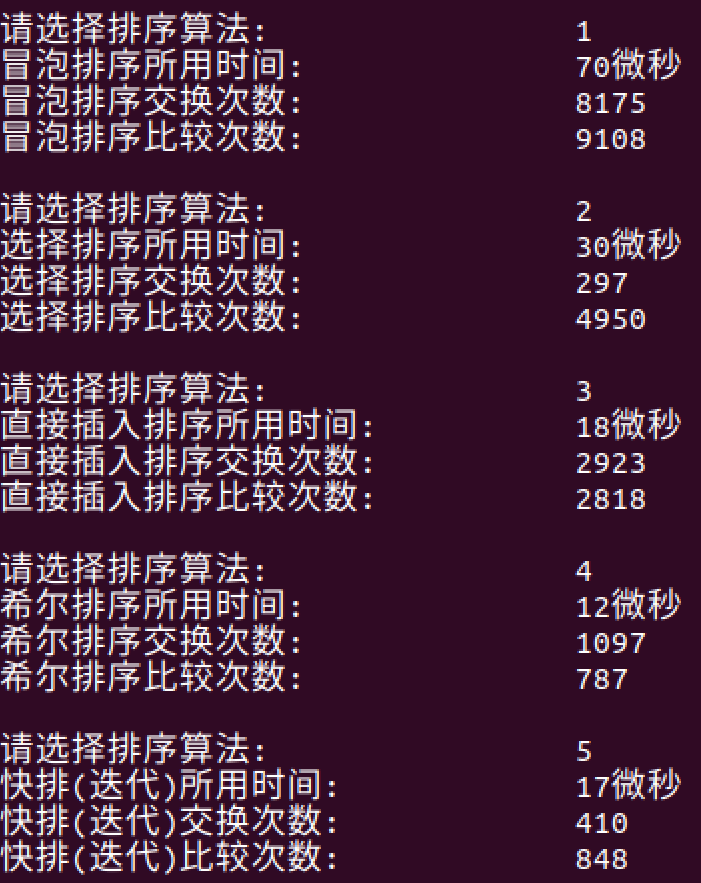
4.9 linux下10个数字测试

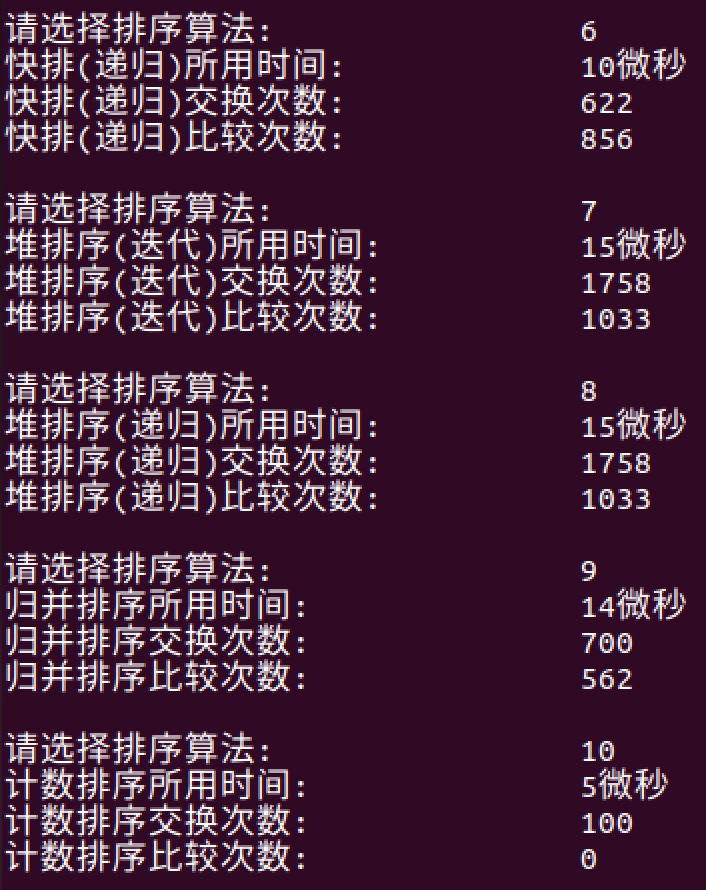


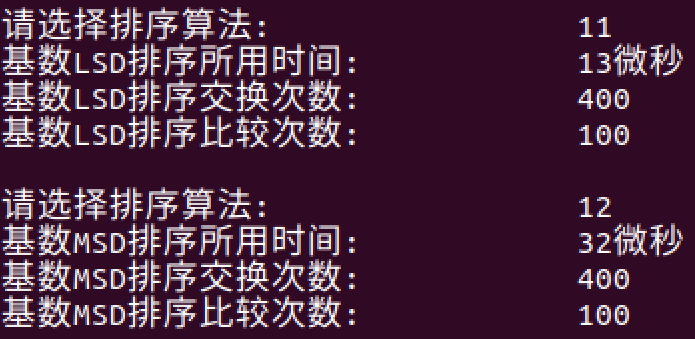




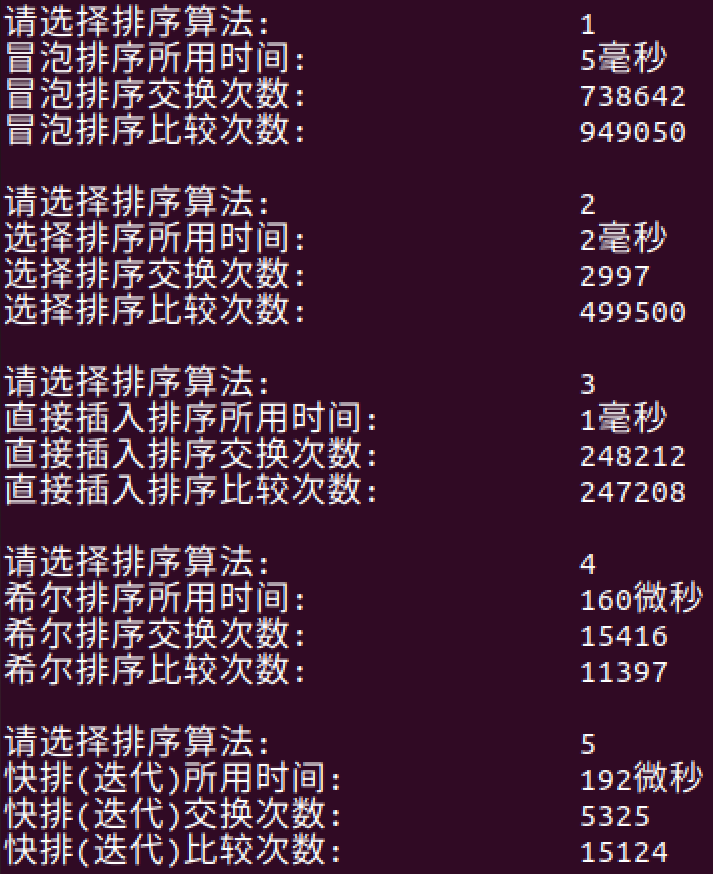
4.10 linux下100个数字测试

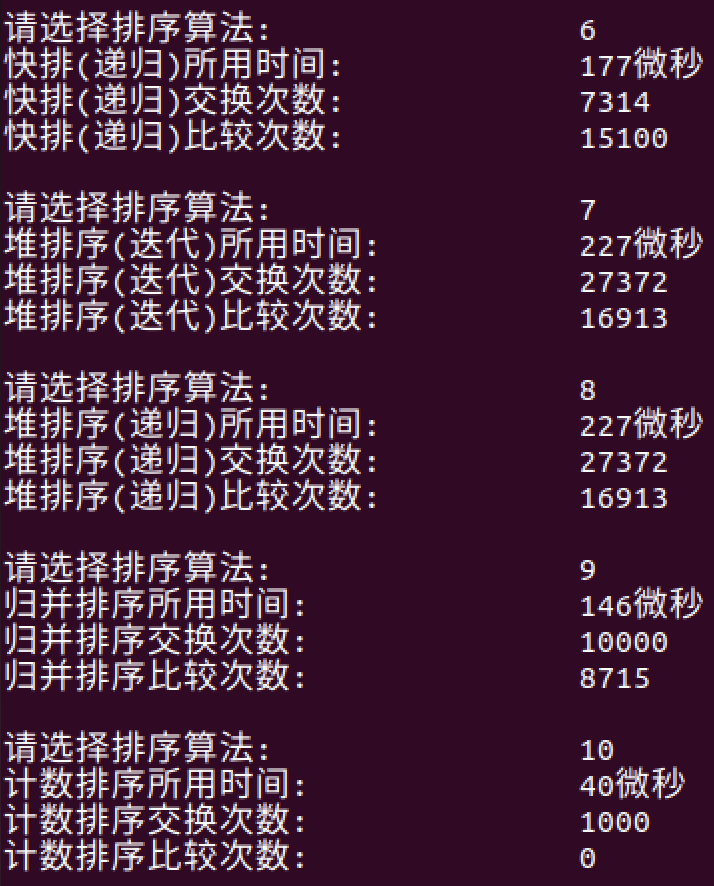


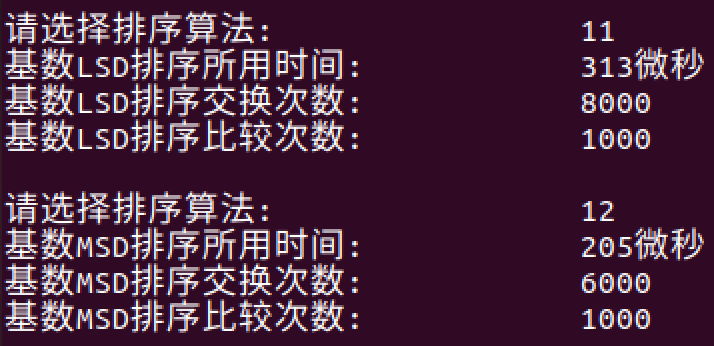




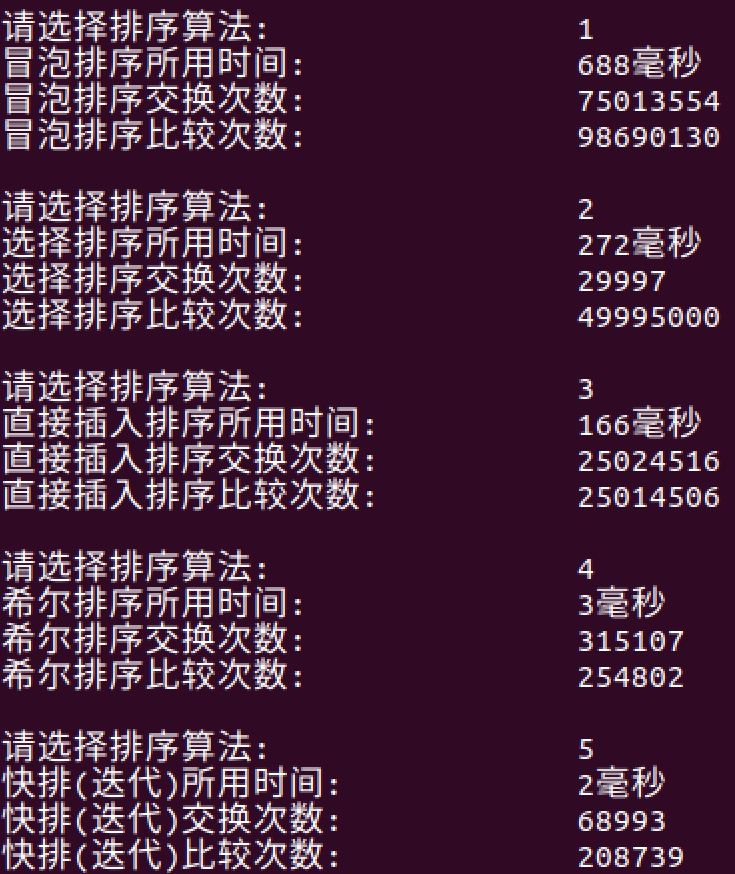
4.11 linux下1000个数字测试



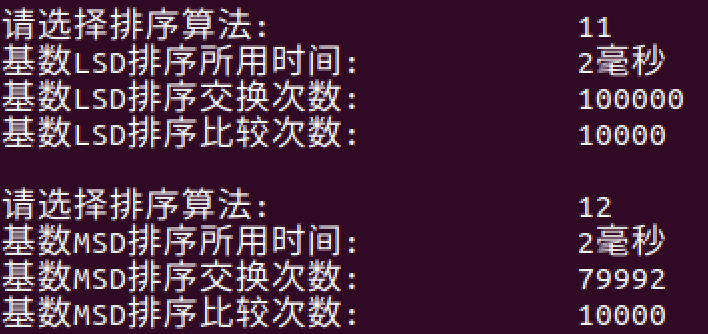




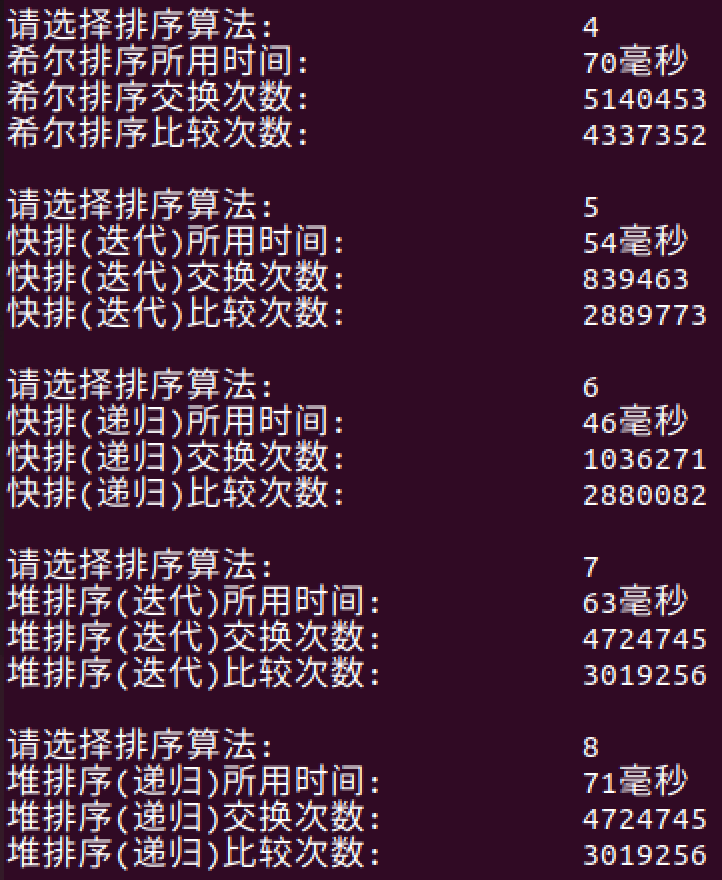
4.12 linux下10000个数字测试

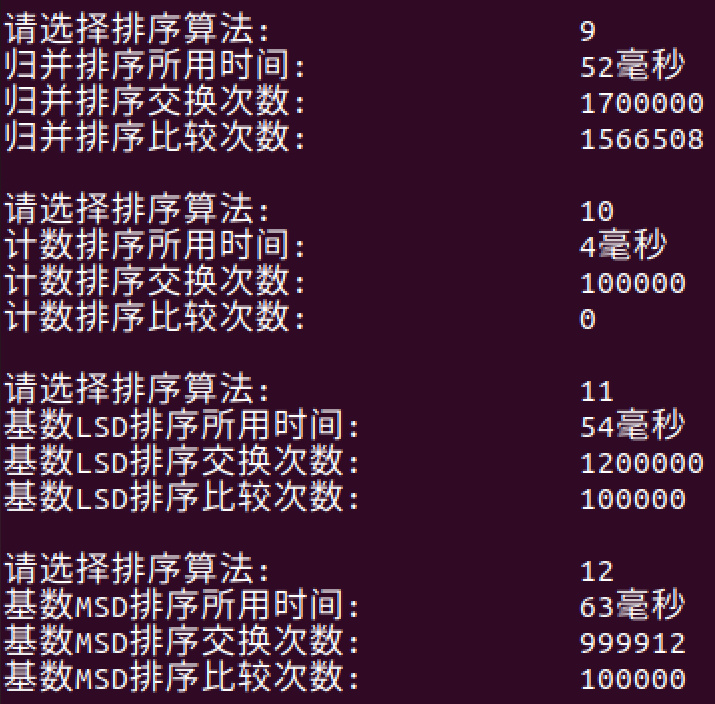




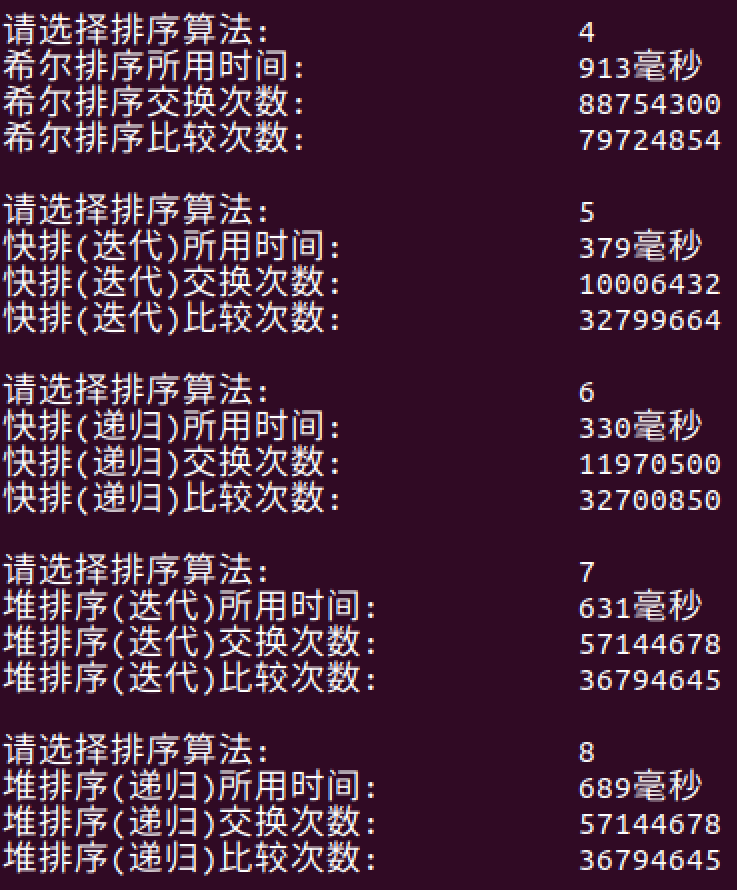


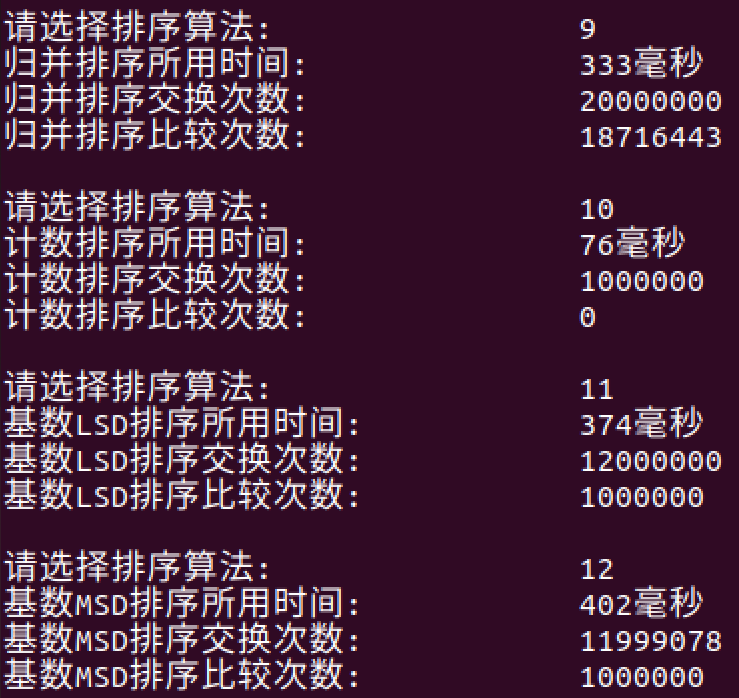
4.13 linux下100000个数字测试





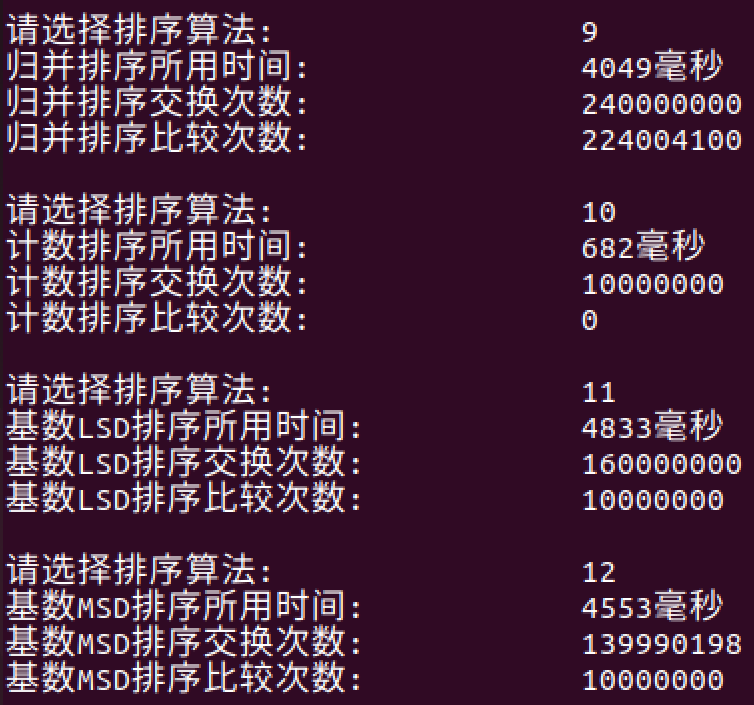
4.14 linux下1000000个数字测试



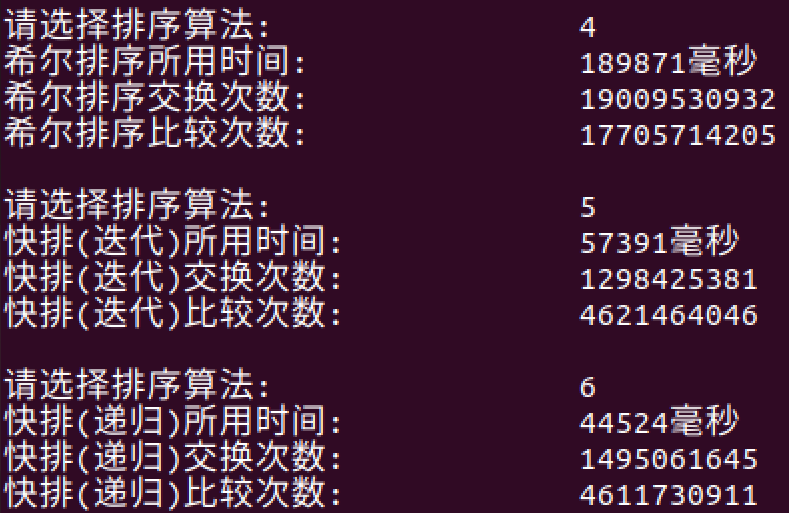


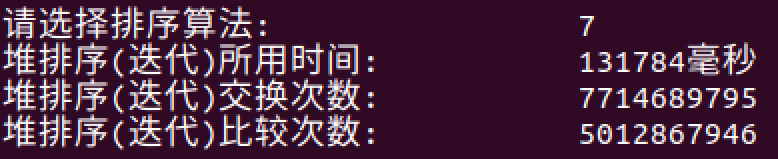
4.15 linux下10000000个数字测试

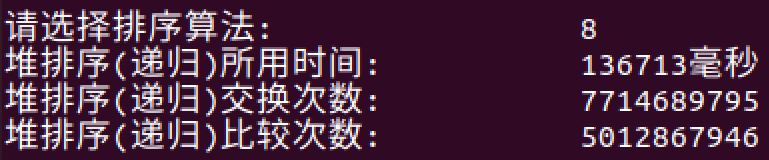


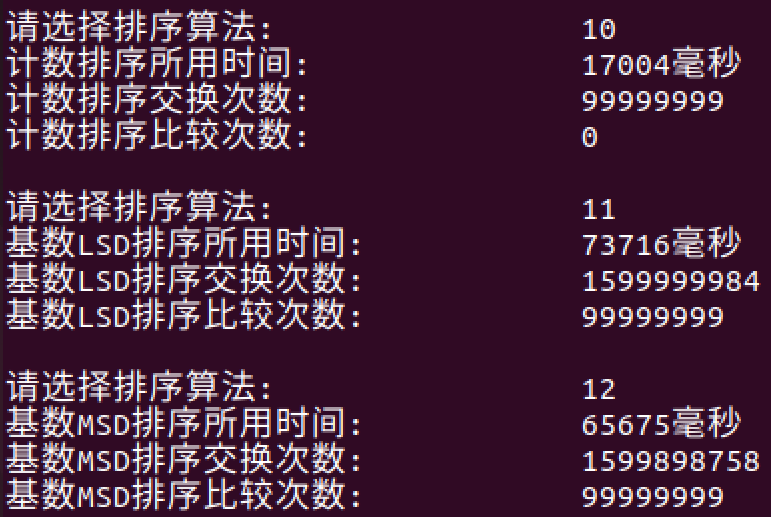


4.16 linux下99999999个数字测试

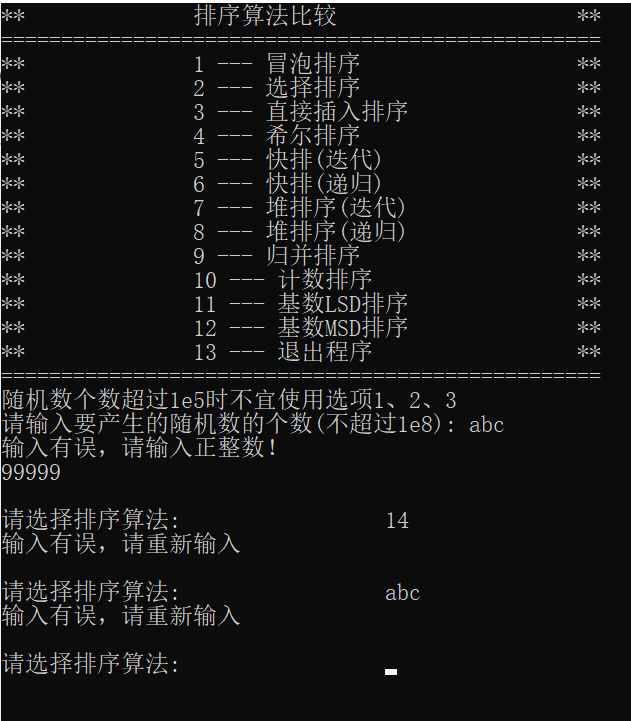




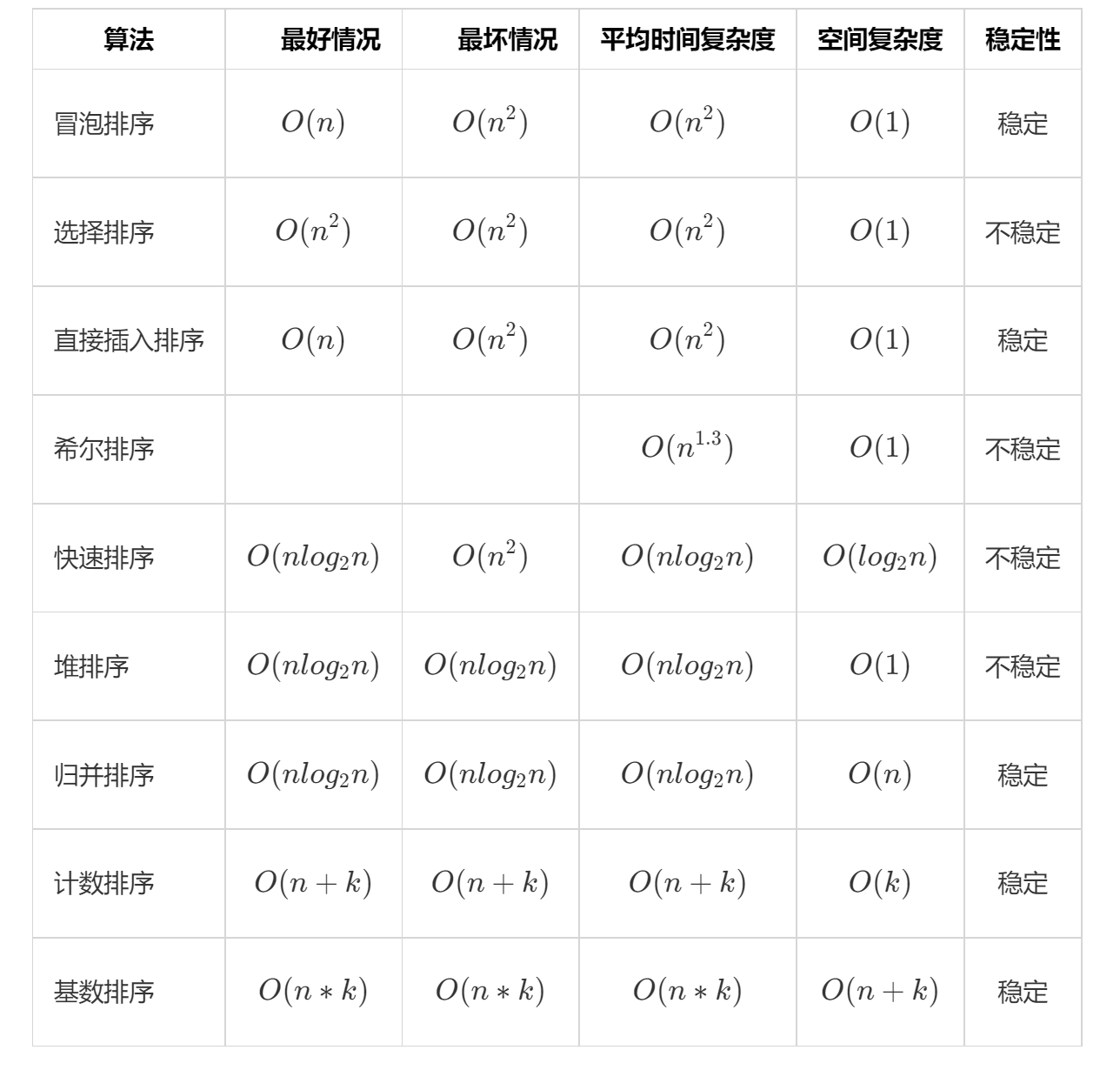




4.17 输入错误判断



5 测试分析比较



当排序元素个数较少，少于一百的时候，使用简单排序性能较好。在测试过程中，反而快速排序、堆排序、归并排序、计数排序、基数排序这些复杂排序算法需要更长的时间。

对于较多的位数，MSD算法确实优于LSD算法；相反，较少位数的时候LSD算法要快于MSD算法。但windows系统下并未能观察到此现象，LSD算法时间效率一直比MSD算法高，而在linux系统下，两个算法时间效率的分界大概在10000000左右。

对于计数排序，通过多次比较，在数据量较大而内存空间足够的情况下，发挥着较大的优势，时间效率上优于其他算法。

对于快速排序，考虑到windows系统下不设置的情况下，随机数最大值未32768，因此在这种情况下，当元素个数未100000及以上的时候，会出现大量相同的数字。因此对于快速排序算法的影响非常大。在此前本人并没有注意到这一点，因此产生windows系统下快速排序居然跑不出来的情况。同时，在linux系统下运行正常。其原因在于linux系统下的rand函数最大值要远远大于windows系统下的最大值，因此生成的随机数较为平均。为了在两个系统下得到更为客观的比较，本项目最后采用<random>库函数里面的随机数生成器，生成均匀分布。由此来解决在windows下出现大量相同值的随机数的问题。

本项目还考虑了使用递归与非递归写法的时间效率的差异，体现在堆排序与快速排序上面。对于堆排序的递归与非递归写法，在windows系统下感受不到明显的差异，但在linux系统运行的时候，当数据量大于1000时，差异显现非常明显。

但对于快速排序的递归与非递归算法的比较，本项目的测试结果均为递归写法优于非递归写法。这是什么原因呢？有待探究。