项目说明文档

数据结构课程设计

——八种排序算法的比较

作 者 姓 名： 李佳诺

学 号： 1751188

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 4

1.1 背景分析 4

1.2 功能分析 4

2 设计 5

2．1 算法设计 5

2.1.1 冒泡排序 5

2.1.2 选择排序 5

2.1.3 插入排序 5

2.1.4 快速排序 5

2.1.5 堆排序 6

2.1.6 基数排序 6

2.1.7 归并排序 6

2.1.8 希尔排序 6

2.2 类与成员设计 7

3 实现 8

3.1 冒泡排序 8

3.1.1 冒泡排序流程图 8

3.2 选择排序 10

3.2.1 选择排序流程图 10

3.3 插入排序 11

3.3.1 插入排序流程图 12

3.4 快速排序 14

3.4.1 快速排序流程图 14

3.5 堆排序 19

3.5.1 堆排序流程图 19

3.6 基数排序 21

3.6.1 基数排序流程图 21

3.7 归并排序 24

3.7.1 归并排序流程图 24

3.8 希尔排序 27

3.8.1 希尔排序流程图 27

4 测试 28

4.1 冒泡排序 28

4.2 选择排序 29

4.3 插入排序 29

4.4 快速排序 30

4.5 堆排序 30

4.6 基数排序 30

4.7 归并排序 31

4.8 希尔排序 31

4.9 出错测试 32

5 总结比较 32

# 1 分析

## 背景分析

在日常生活中经常需要对所收集的各种数据进行处理，这些数据处理中经常遇到的核心运算就是排序。例如，图书管理员将书籍按照馆藏编号排序放置在书架上，方便读者查找；打开计算机的资源管理器，可以选择按名称、类型、创建日期、等来排序图标。排序已被广泛应用在几乎所有的领域。

## 功能分析

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。

# 设计

## 2．1 算法设计

### 2.1.1 冒泡排序

冒泡排序的思想就是重复地遍历过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。遍历数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。

### 2.1.2 选择排序

选择排序的思想就是每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。

### 2.1.3 插入排序

插入排序的基本方法就是第一趟比较前两个数，然后把第二个数按大小插入到有序表中； 第二趟把第三个数据与前两个数从后向前扫描，把第三个数按大小插入到有序表中；依次进行下去，进行了(n-1)趟扫描以后就完成了整个排序过程。

### 2.1.4 快速排序

快速排序的基本方法就是通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

### 2.1.5 堆排序

堆排序的基本思想就是利用堆积树（堆）这种数据结构所设计的一种排序算法，它是选择排序的一种。可以利用数组的特点快速定位指定索引的元素。堆分为大根堆和小根堆，是**完全二叉树**。大根堆的要求是每个节点的值都不大于其父节点的值，即**A[PARENT[i]] >= A[i]。**在数组的非降序排序中，需要使用的就是大根堆，因为根据大根堆的要求可知，最大的值一定在堆顶。

### 2.1.6 基数排序

基数排序的基本方法就是透过键值的部份资讯，将要排序的元素分配至某些“桶”中，藉以达到排序的作用。

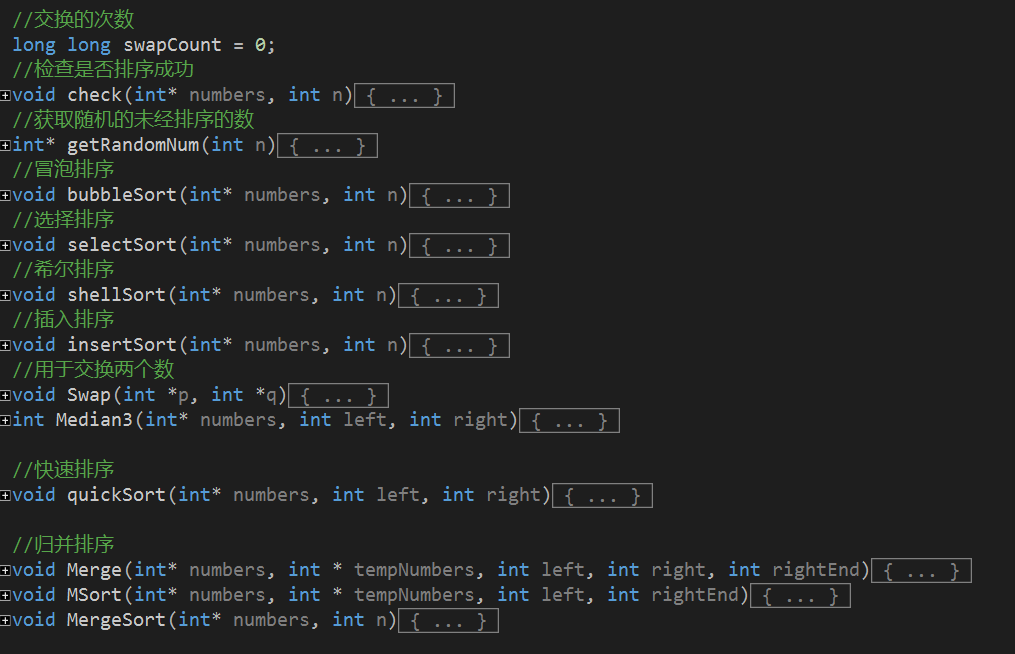
### 2.1.7 归并排序

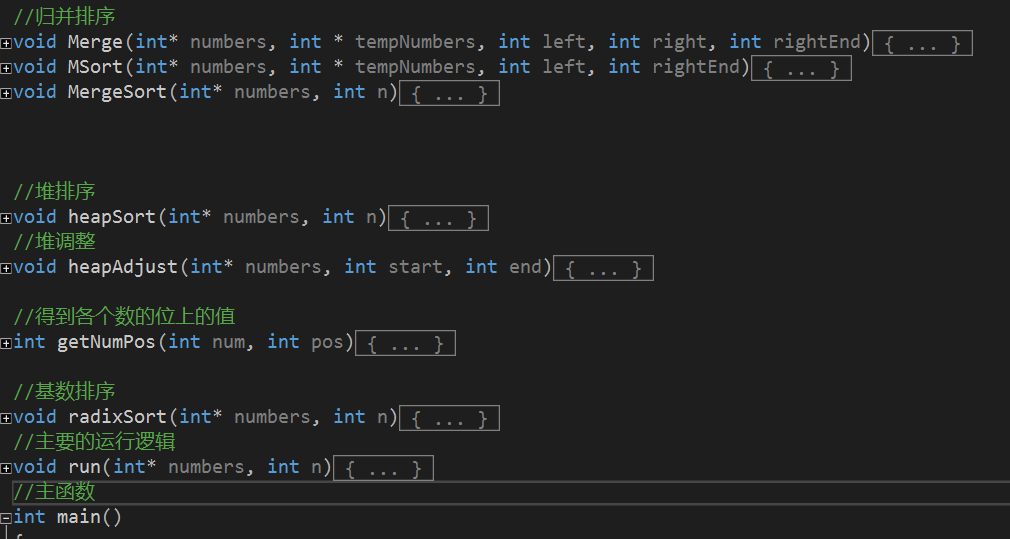
归并排序的基本方法就是比较a[i]和a[j]的大小，若a[i]≤a[j]，则将第一个有序表中的元素a[i]复制到r[k]中，并令i和k分别加上1；否则将第二个有序表中 的元素a[j]复制到r[k]中，并令j和k分别加上1，如此循环下去，直到其中一个有序表取完，然后再将另一个有序表中剩余的元素复制到r中从下标k到 下标t的单元。归并排序的算法我们通常用递归实现，先把待排序区间[s,t]以中点二分，接着把左边子区间排序，再把右边子区间排序，最后把左区间和右区 间用一次归并操作合并成有序的区间[s,t]。

### 2.1.8 希尔排序

希尔排序的基本方法就是：把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

## 2.2 类与成员设计





# 实现

## 冒泡排序

### 冒泡排序流程图

开 始

结 束

遍 历

是否排好序

current>next

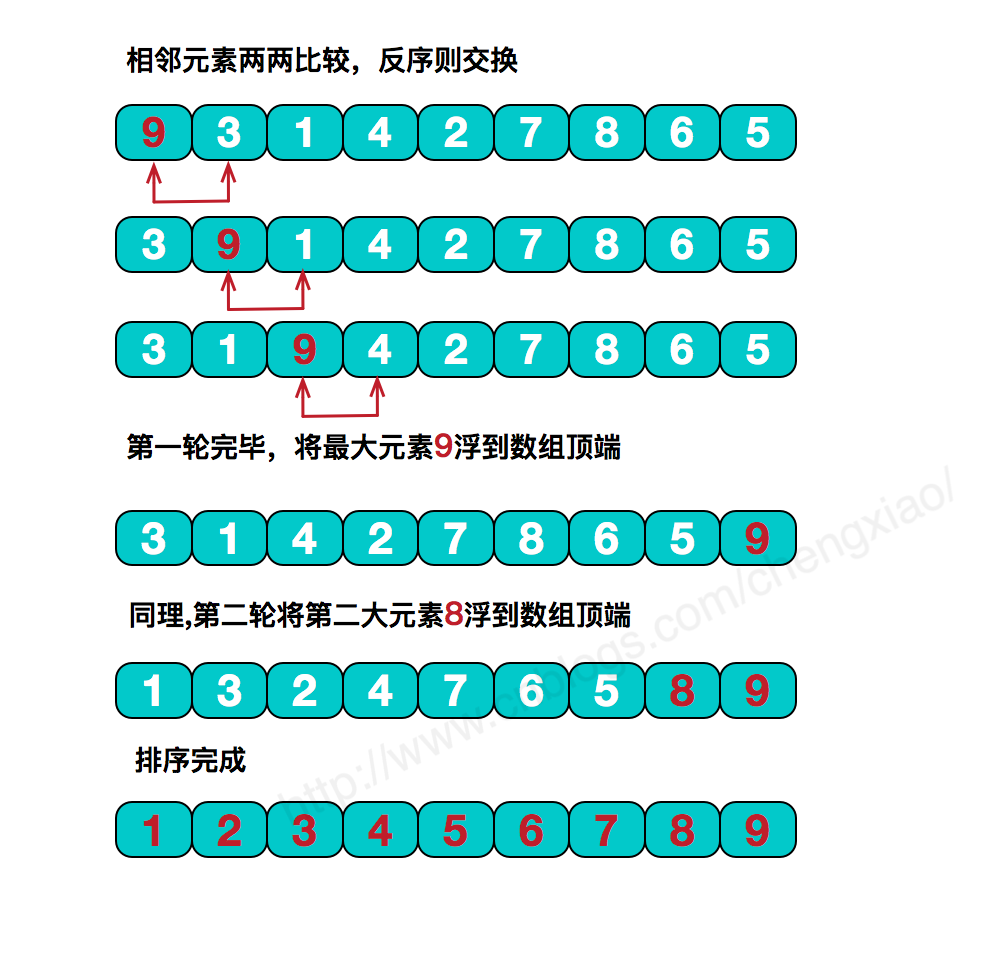
否

是

否

是

交换 始



//冒泡排序

void bubbleSort(int\* numbers, int n)

{

int temp;

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

for (int j = 0; j < n – i - 1; j++)

{

if (numbers[j] > numbers[j + 1])

{

swapCount++;

temp = numbers[j];

numbers[j] = numbers[j + 1];

numbers[j + 1] = temp;

}

}

}

## 选择排序

### 3.2.1 选择排序流程图

开始

结束

将未排好序部分的最小的数放到最后

是否排好序

是

否

//选择排序

void selectSort(int\* numbers, int n)

{

int temp;

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

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

{

if (numbers[i] > numbers[j])

{

swapCount++;

temp = numbers[i];

numbers[i] = numbers[j];

numbers[j] = temp;

}

}

}

## 插入排序

### 3.3.1 插入排序流程图

开 始

将当前数插入有序表中

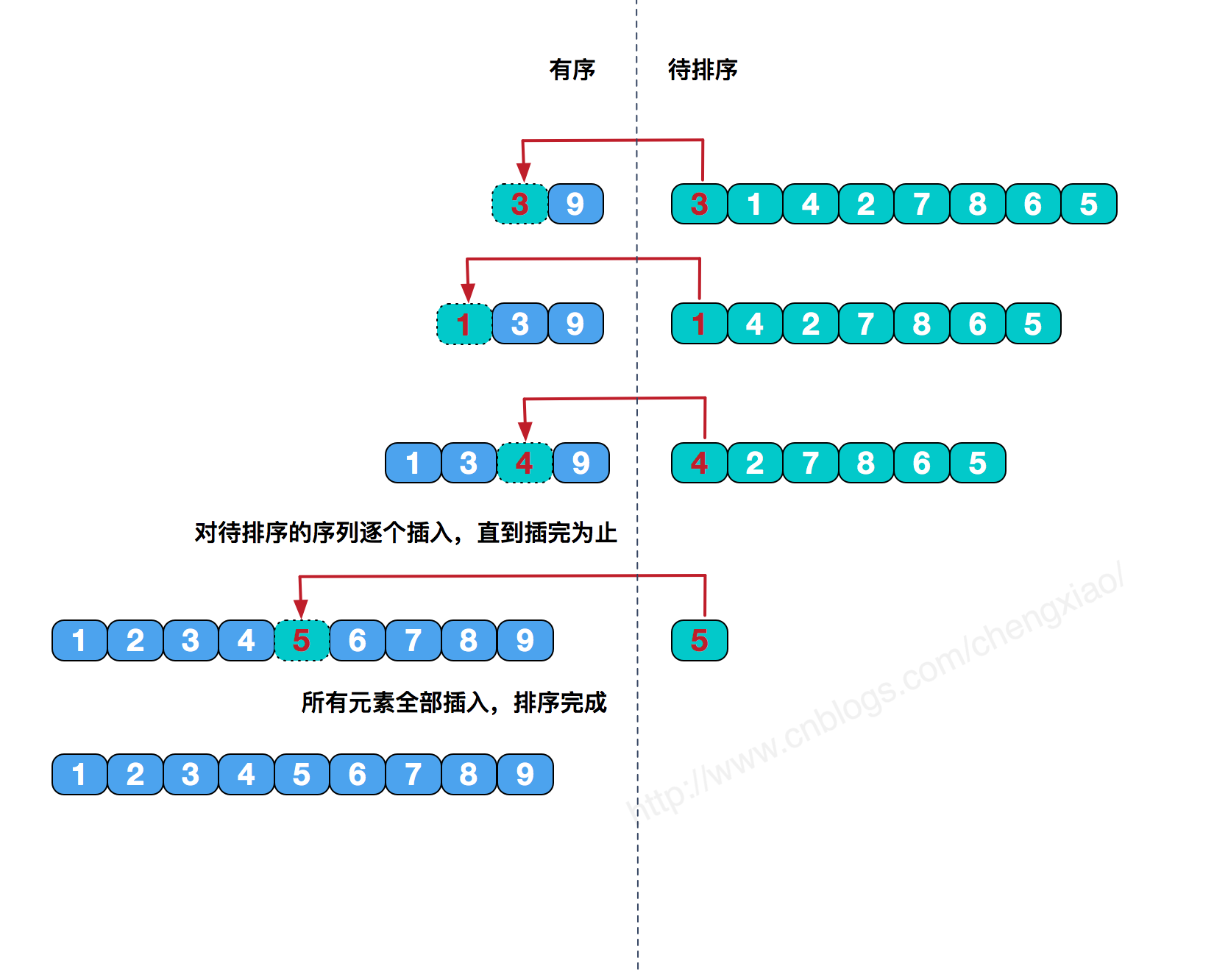
返回有序表

结 束

是否已经遍历完

是

否



//插入排序

void insertSort(int\* numbers, int n)

{

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

{

auto temp = numbers[i];

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

{

int j = 0;

for (j = i - 1; j >= 0 && temp < numbers[j]; j--)

{

numbers[j + 1] = numbers[j];

}

numbers[j + 1] = temp;

}

}

}

## 快速排序

### 3.4.1 快速排序流程图

开始

将List划分为两个子序列LeftList和RightList

对LeftList进行快速排序

RightList进行快速排序

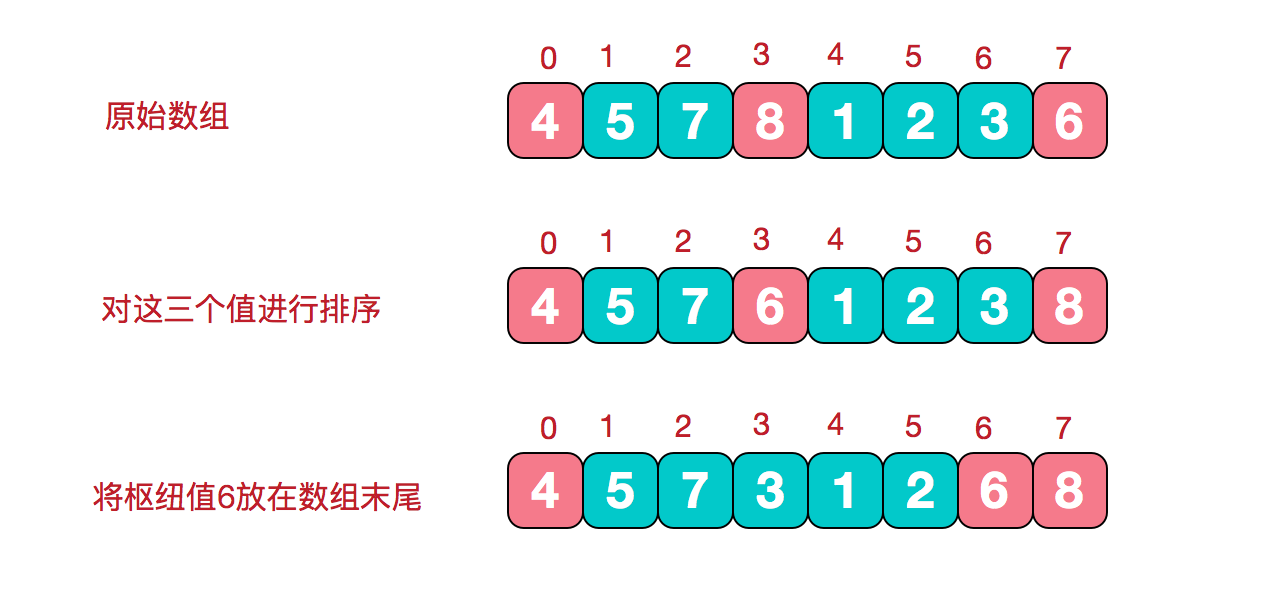
List的长度>1?

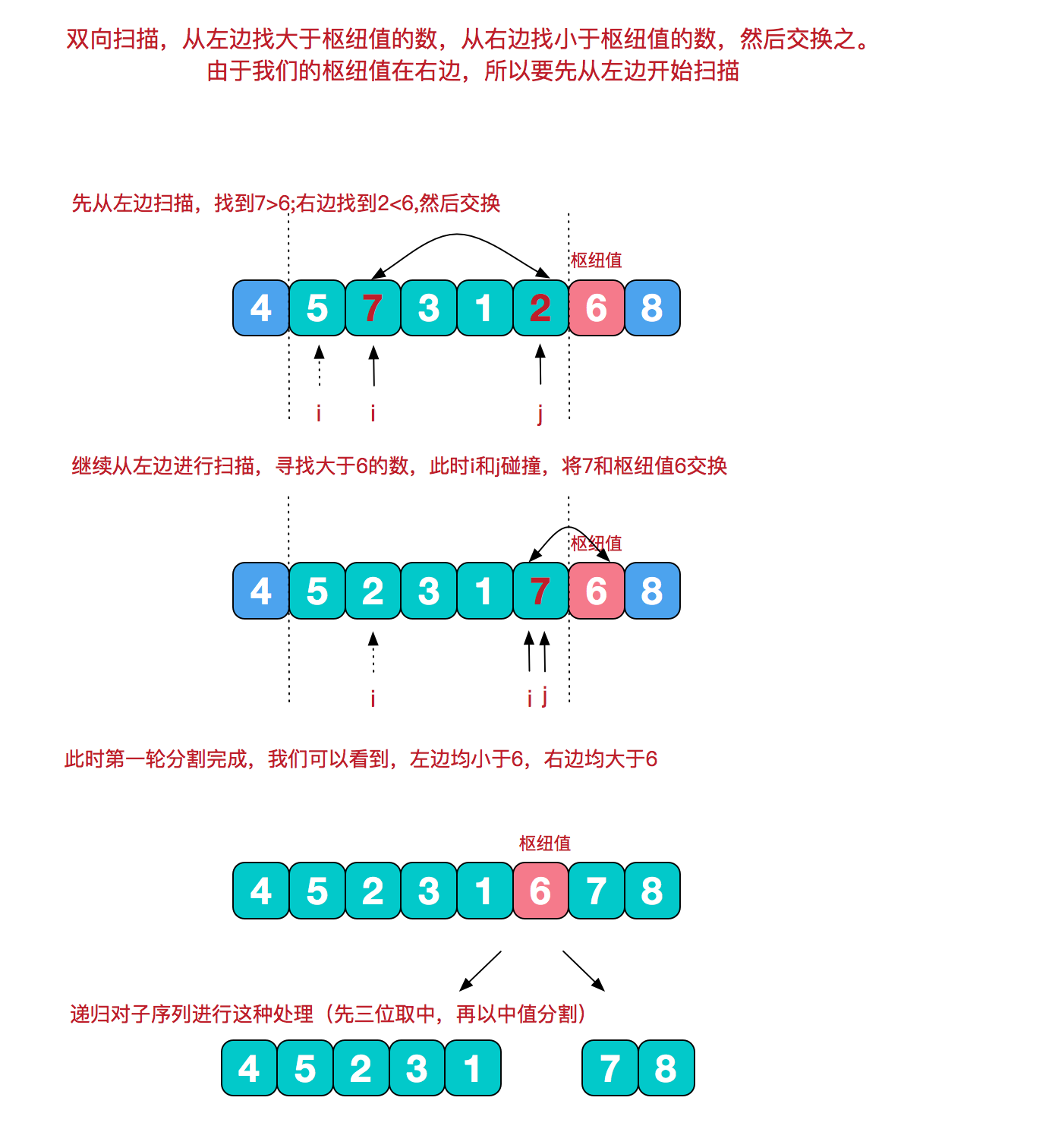
将LeftList和RightList合并为新的List

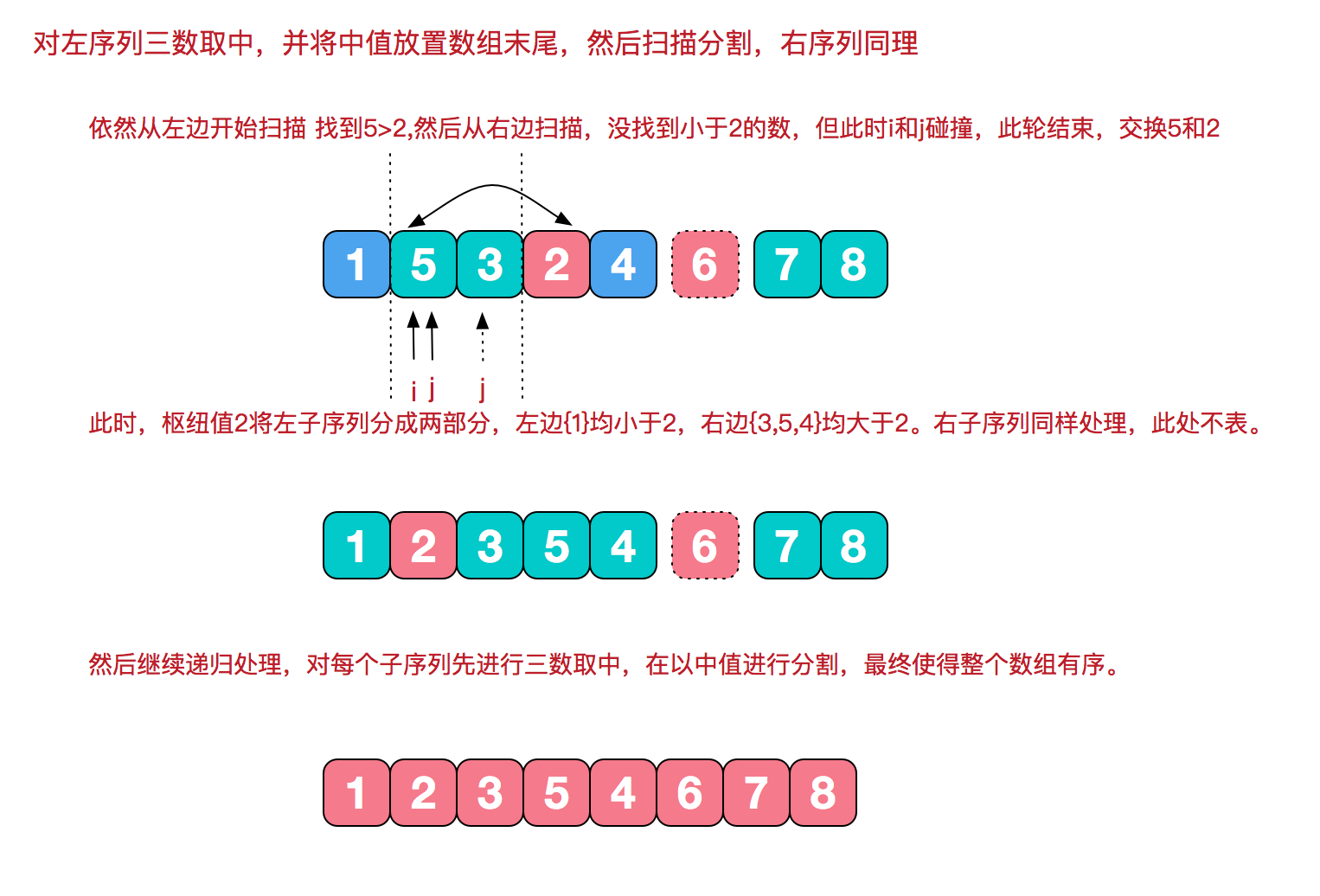
结束

是

否







//快速排序

void quickSort(int\* numbers, int left, int right)

{

int pivot, i, j;

if (right - left >= 50)

{

if (left >= right)

return;

pivot = Median3(numbers, left, right);

i = left;

j = right - 1;

while (1)

{

while (numbers[++i] < pivot);

while (numbers[--j] > pivot);

if (i < j)

{

Swap(&numbers[i], &numbers[j]);

swapCount++;

}

else

break;

}

Swap(&numbers[i], &numbers[right - 1]);

swapCount++;

quickSort(numbers, left, i - 1);

quickSort(numbers, i + 1, right);

}

else

{

insertSort(numbers + left, right - left + 1);

}

}

## 堆排序

### 3.5.1 堆排序流程图

开 始

建立空堆

将数列中的数依次加入堆并调整

将堆中元素依次返回

结 束

//堆排序

void heapSort(int\* numbers, int n)

{

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

heapAdjust(numbers, i, n - 1);

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

{

Swap(&numbers[0], &numbers[i]);

heapAdjust(numbers, 0, i - 1);

}

bubbleSort(numbers, 10);

}

//堆调整

void heapAdjust(int\* numbers, int start, int end)

{

int child = 2 \* start + 1;

int temp = numbers[start];

while (child <= end)

{

if (child + 1 < end && numbers[child] <= numbers[child + 1])

child++;

if (numbers[child] <= temp)

break;

numbers[start] = numbers[child];

swapCount++;

start = child;

child = start \* 2 + 1;

}

numbers[start] = temp;

}

## 基数排序

### 3.6.1 基数排序流程图

开 始

根据最大数的位数生相应的桶数

每一位依次将数以先进后出的方式放入拿出

结 束

//得到各个数的位上的值

int getNumPos(int num, int pos)

{

int division = 1;

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

division \*= 10;

return (num / division) % 10;

}

//基数排序

void radixSort(int\* numbers, int n)

{

auto radixArrays = new int\*[10];

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

{

radixArrays[i] = new int[n + 1];

radixArrays[i][0] = 0;

}

int count = 0;

int maxNum = n;

while (maxNum /= 10)

count++;

count++;

int num = 0, position = 0;

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

{

//分配在各个桶里面

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

{

num = getNumPos(numbers[j], i);

position = ++radixArrays[num][0];

radixArrays[num][position] = numbers[j];

}

int index = 0;

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

{

for (int k = 1; k < radixArrays[j][0] + 1; k++, index++)

{

numbers[index] = radixArrays[j][k];

swapCount++;

}

radixArrays[j][0] = 0;

}

}

//释放空间

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

delete(radixArrays[i]);

delete(radixArrays);

}

## 归并排序

### 3.7.1 归并排序流程图

开 始

left>=right?

Mid = (left+right)/2

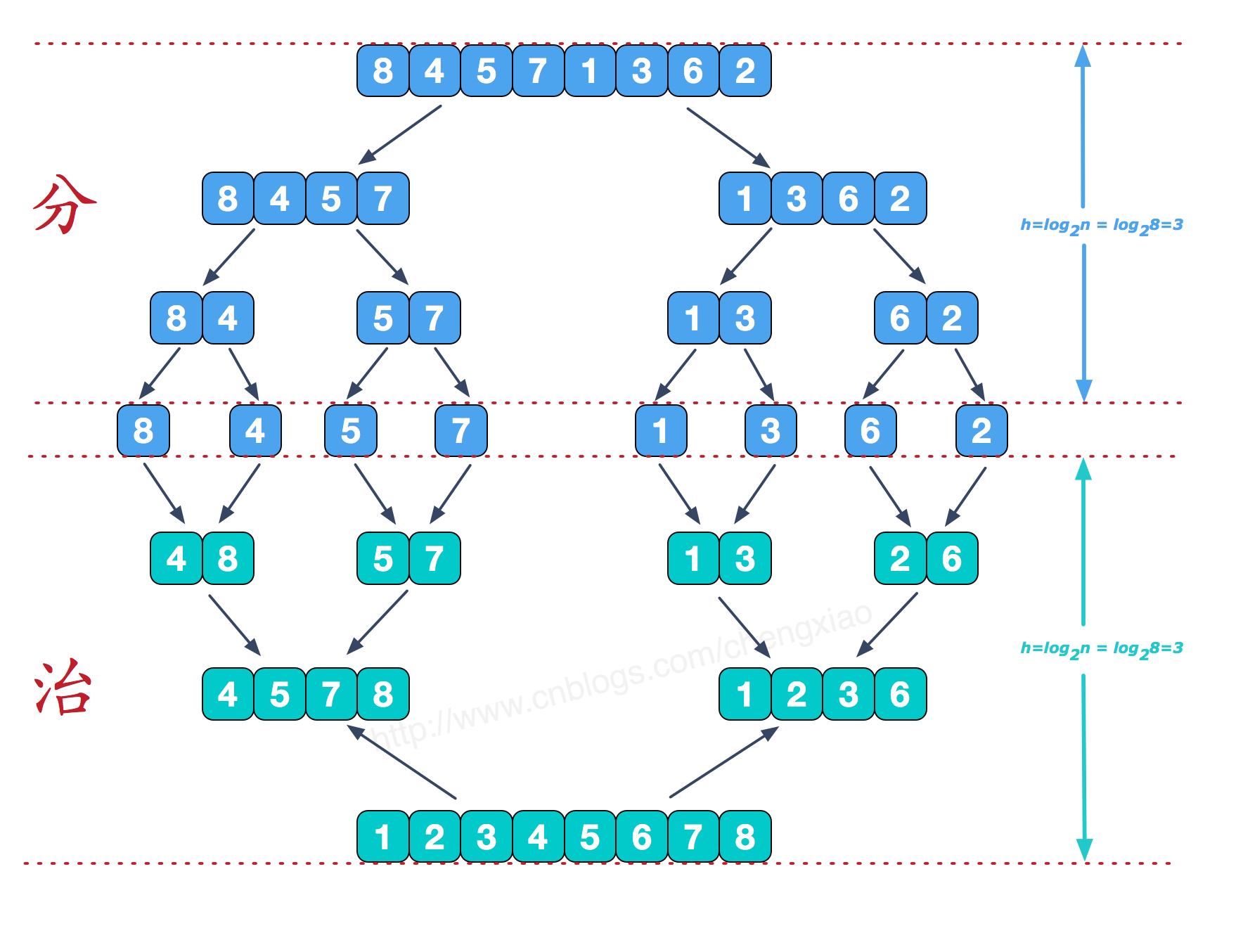
分别对左右子序列递归排序

合 并

结 束

否

是



//归并排序

template<class Iterator>

void mergeSort(Iterator begin, Iterator end)

{

if (begin == end)

{

return;

}

int len = end - begin;

if (len == 1)

{

return;

}

Iterator mid = begin + len / 2;

mergeSort(begin, mid);

mergeSort(mid, end);

auto type = \*begin;

decltype(type)\* temp = new decltype(type)[len];

Iterator i = begin;

Iterator j = mid;

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

{

if ((i != mid) && (j == end || (\*i) < (\*j)))

{

temp[c] = \*i;

++i;

swapCount++;

}

else

{

temp[c] = \*j;

++j;

swapCount++;

}

}

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

{

\*(begin + c) = temp[c];

}

delete[] temp;

}

## 希尔排序

### 3.8.1 希尔排序流程图

开 始

gap = n

gap>0?

gap /= 2

每组分别用插入排序

结 束

否

是

//希尔排序

void shellSort(int\* numbers, int n)

{

int j, gap;

for (gap = n / 2; gap > 0; gap /= 2)

for (j = gap; j < n; j++)//从数组第gap个元素开始

if (numbers[j] < numbers[j - gap])//每个元素与自己组内的数据进行直接插入排序

{

int temp = numbers[j];

int k = j - gap;

while (k >= 0 && numbers[k] > temp)

{

numbers[k + gap] = numbers[k];

k -= gap;

swapCount++;

}

numbers[k + gap] = temp;

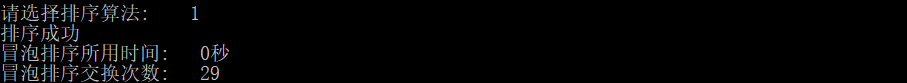
}

}

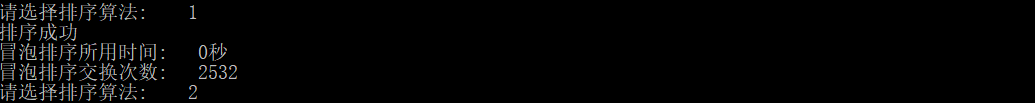
# 测试

## 冒泡排序

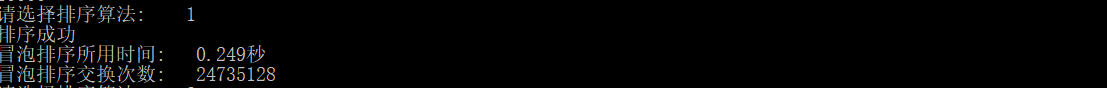
N=10



N=100

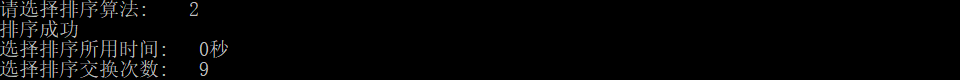


N=10000

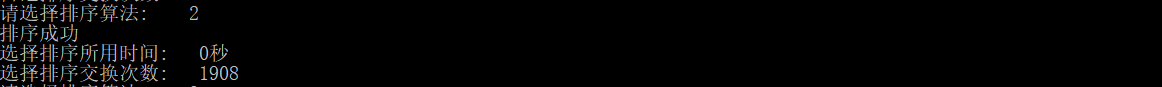


## 选择排序

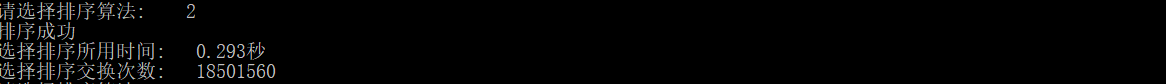
N=10



N=100

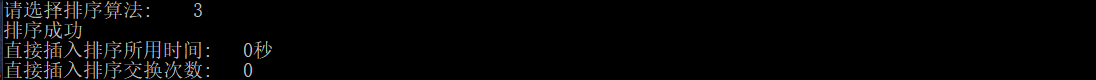


N=10000

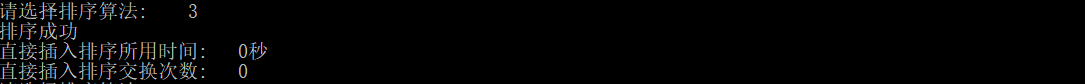


## 插入排序

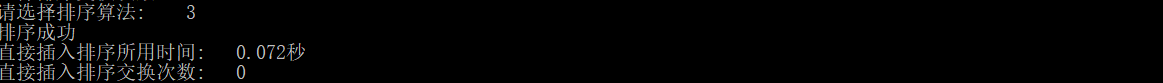
N=10



N=100

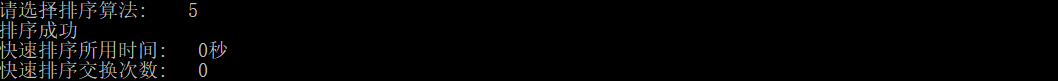


N=10000

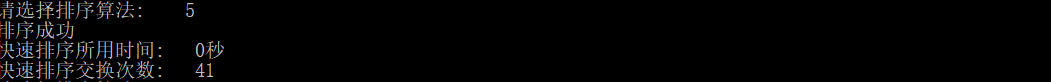


## 快速排序

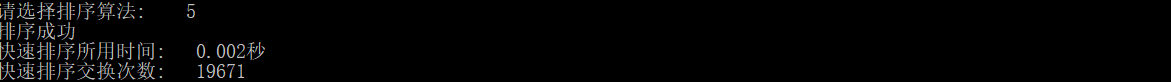
N=10



N=100

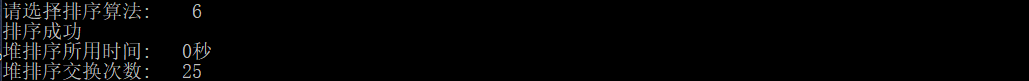


N=10000

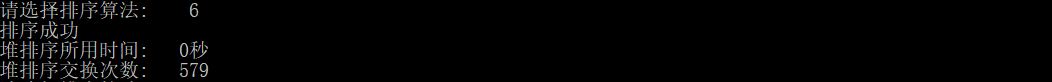


## 堆排序

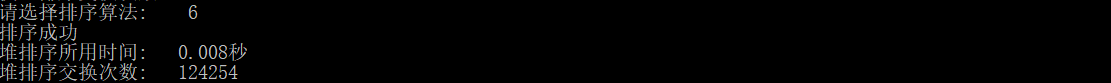
N=10



N=100

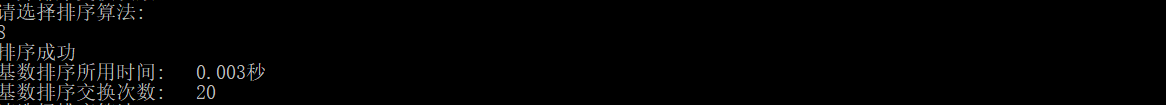


N=10000

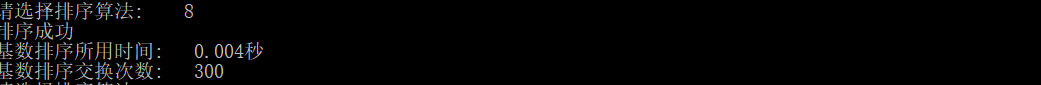


## 基数排序

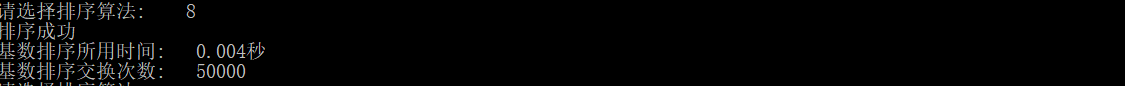
N=10



N=100

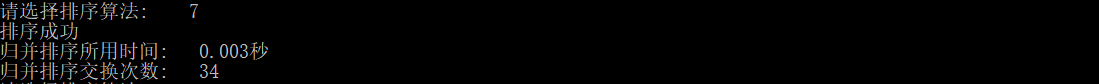


N=10000

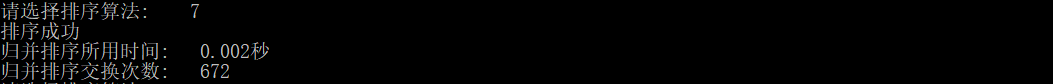


## 归并排序

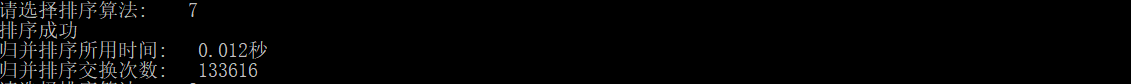
N=10



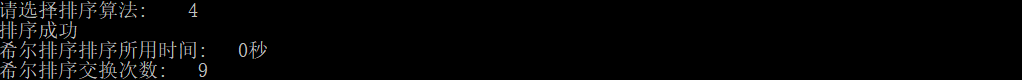
N=100



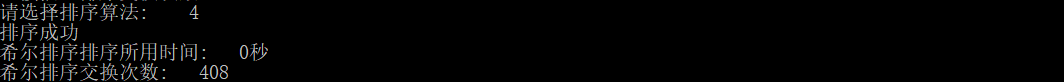
N=10000



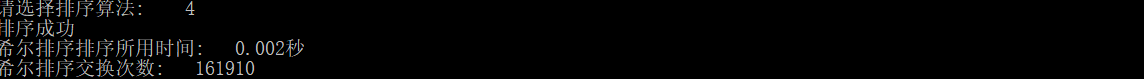
## 希尔排序

N=10

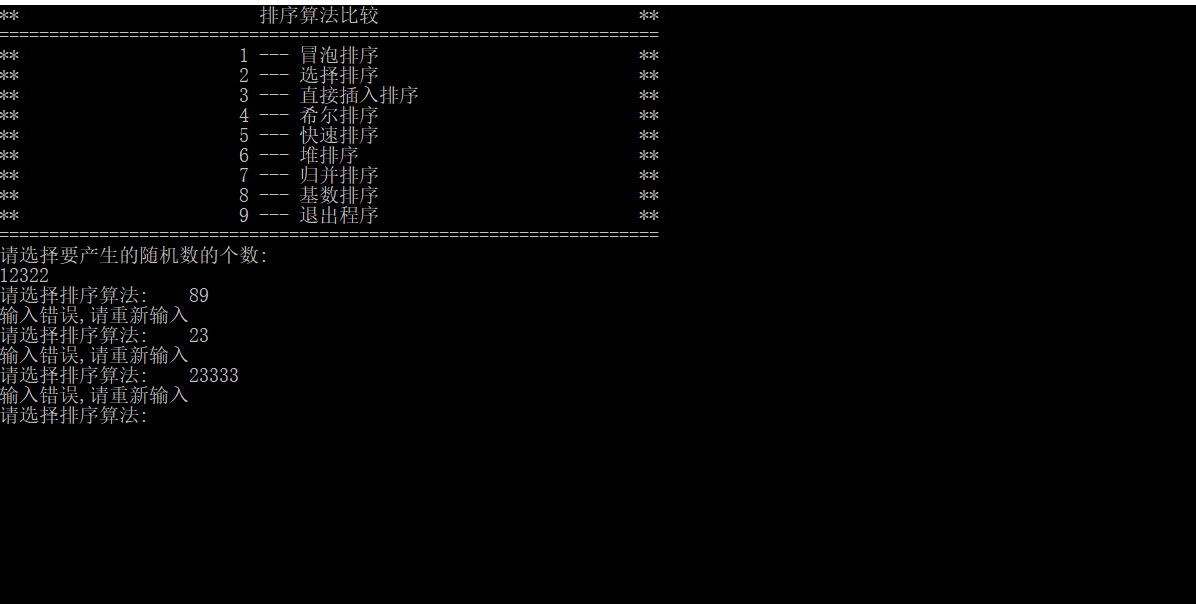
N=100



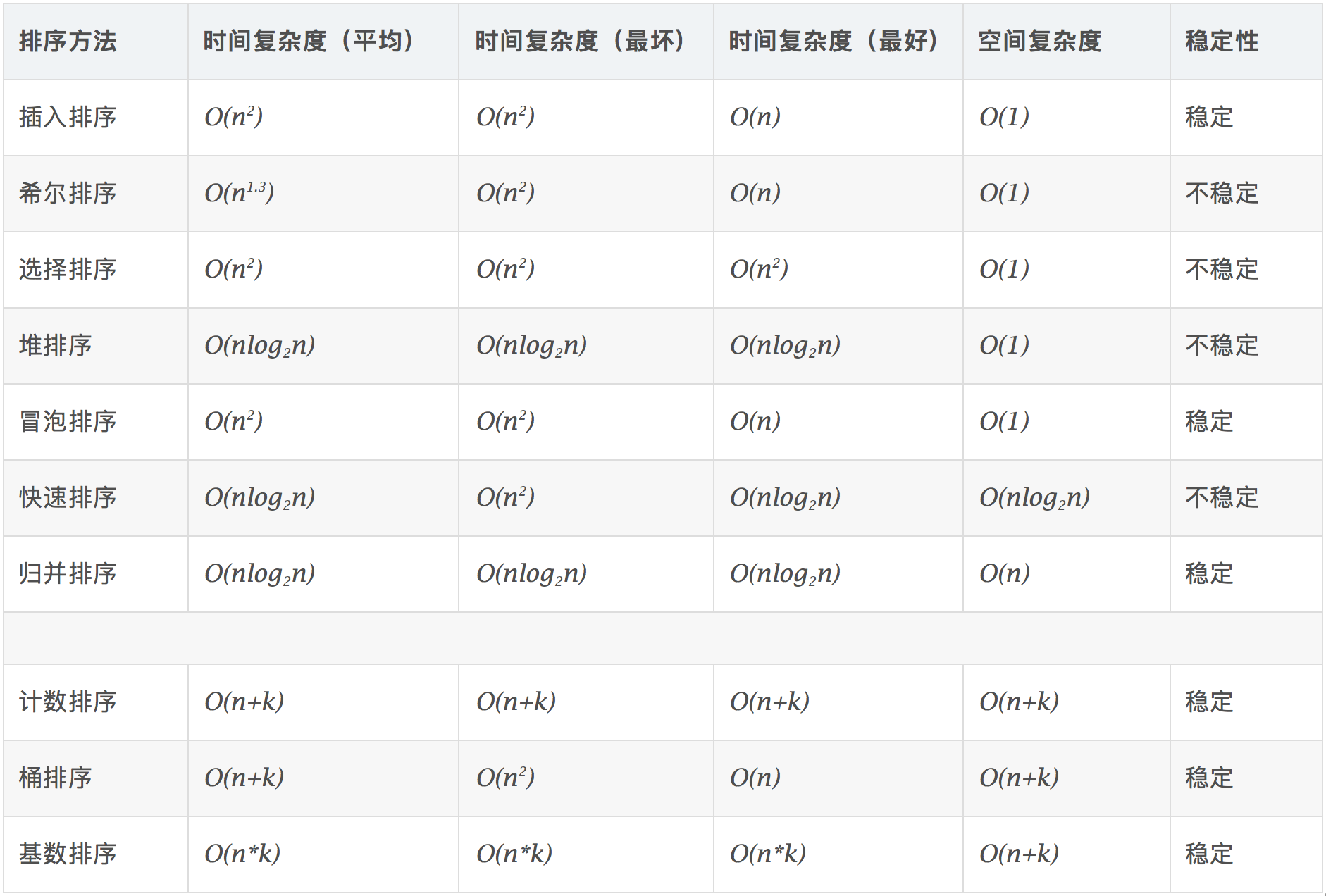
N=10000



## 出错测试



# 总结比较



1. 当数据规模极小时，8大排序算法性能上并没有出现明显的差异。2. 当数据规模增大时时间复杂度为o(n^2)的三种算法开销明显高过其他5种算法。3. 在数字位数较小的情况下，基数排序性能最强。4. 对于数据量开始增大的情况下冒泡排序、直接插入排序、选择排序性能出现了一定的差异。其中冒泡时间最长、直接插入次之、选择排序最好。导致这一结果的原因可能是由于交换操作产生的，冒泡排序和直接插入排序交换次数较多，选择排序最少。而冒泡排序和直接插入排序的性能差异应该是直接插入排序在遇到比自己小的元素便停止这一轮的遍历，进行了剪枝操作，而冒泡排序没有。这一点从测试种的比较次数也可以看出。5. 对于堆排序，快速排序和归并排序来说，虽然三者的时间复杂度都是o(nlogn)，但测试结果显示，快速排序最快、归并排序次之、堆排序略最慢。据分析，原因应该是：快速排序除进行比较外，几乎没有任何额外操作，时间复杂度常数接近于1，归并排序有合并操作，堆排序有建堆操作。故快速排序最快。6. 而堆排序性能略逊于归并排序的可能的原因有几个：一是归并排序同样在合并的过程中同样有剪枝的操作减少了比较次数，这一点从测试的比较次数中也可以看出，二是堆排序的堆调整使用递归写法，开销较大，三是在编写函数时的写法问题导致常数略有不同，四是可能是由于堆排序使用迭代器造成的开销大于归并排序。但具体究竟为何导致堆排序慢于归并排序，由于个人水平有限，未能给出很好的答案。7. 对于大量重复元素的情况，由于快速排序对此进行了优化，使得快速排序的递归次数远少于归并和堆排序，故在这个情况下快速排序性能表现极佳。8. 在一般情况下，基数排序的性能要比其他几种排序都要好，但基数排序适用范围较窄。