# 1.几种常见的排序算法

几个常见排序算法排序过程的演示：

https://www.yijiyong.com/algorithm/visual/01-commondy.html

https://visualgo.net/en

排序(Sort)，就是重新排列表中的元素，使表中的元素满足按关键字有序的过程。在数据结构课里，一般会将查找章节放在排序前面，因此大部分人都会感觉查找比排序容易。但是我们研究过算法之后就会发现查找远远难于排序，因为常见的排序方法是相对固定的。而查找除了最基本的二分查找外，还包含非常广的内容，二叉树，各种树，Hash，大数据下的查找，而动态规划和回溯等本身也是在寻找某个特定的目标，所以查找贯穿整个算法学习，而排序则基本是固定的几种。
常见的排序方法有：1.冒泡排序、2.选择排序、3 插入排序、4.快速排序、5.归并排序、6.归并排序、7.堆或者优先级队列排序、8.桶排序、9.基数排序、10.希尔排序、11.位图排序。

在这些方法中，快速排序和归并排序最重要，而且很多算法题都会用到其思想，因此我们单独成章来分析。快速排序和归并排序网上的介绍材料非常多，实现也很多，但是很多看一看都感觉累，面试的时候怎么可能写出来，我们这里总结了一种特别精简的实现，让你真正能写出来。对于快排我们还以此为基础解决”寻找第K大“的问题，让你彻底不再害怕这个问题。

堆和优先级队列排序，我们前面已经分析过，这里不再谈。而桶排序、基数排序、希尔排序作为拓展内容，我们只要理解其排序原理就行了。位图排序，我们到高级算法部分再说。

我们接下来会不断分析一些常见的算法的实现原理和拓展等问题，但是还有一个重要的问题我们需要注意，那就是各个算法的时间和空间复杂度怎么样，以及如何评价各个算法的好坏。时间和空间复杂度，我们会逐个分析，而综合评价则在最后一章统一分析。在面试的时候要对各些算法的特征非常熟悉才可以的。

再补充一个问题，在排序中，少不了要进行交换，那么交换有几种方式呢？常见有三种；

// 方法一: 利用临时数tmp

```
private void swap(int[] arr, int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
```

// 方法二: 利用加减运算

```
private void swapCal(int[] arr, int i, int j) {
    if(i == j) return; // 若无法保证swapCal被调用时满足 i != j，则需有此句，否则i == j时此数将变为0
    arr[i] = arr[i] + arr[j]; // a = a + b
    arr[j] = arr[i] - arr[j]; // b = a - b
    arr[i] = arr[i] - arr[j]; // a = a - b
}
```

// 方法三: 利用异或运算

```
private void swapXOR(int[] arr, int i, int j) {
    if(i == j) return; // 若无法保证swapXOR被调用时满足 i != j，则需有此句，否则i == j时此数将变为0
    arr[i] = arr[i] ^ arr[j]; // a = a ^ b，也可写成 arr[i] ^= arr[j];
    arr[j] = arr[i] ^ arr[j]; // b = (a ^ b) ^ b = a ^ (b ^ b) = a ^ 0 = a， 也可写成 arr[j] ^= arr[i];
    arr[i] = arr[i] ^ arr[j]; // a = (a ^ b) ^ a = (a ^ a) ^ b = 0 ^ b = b， 也可写成 arr[i] ^= arr[j];
}
```

上面三种方式，第一种是最常用的，后面两种有一定的技巧，如果不是特别研究，不易想到。

## 1.1.调用库函数Arrays.sort

什么？调用库函数也是一种方法？是的，如果算法本身比较复杂，你是可以直接使用库函数接口的，我们在一维数组部分介绍过，关键时刻能救你一命。如果就是为了考察排序，那自然不能直接调用。

```
import java.util.Arrays;
public class Solution {
    public int[] MySort (int[] arr) {
        //调用库函数sort;
        Arrays.sort(arr);
        return arr;
    }
}
```

我们后面介绍的桶排序算法，稍微复杂一些，里面还需要用到排序算法，此时就使用了库方法。

## 1.2 冒泡排序BubbleSort

大话数据结构 第九章

最基本的排序算法，这个题目就是热身或者在某些场景下使用其变型方法，比如前面链表里奇偶调整的例子中就用到了。

我们以关键字序列{26,53,48,11,13,48,32,15}看一下排序过程:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 初始关键字 | 26 | 53 | 48 | 11 | 13 | 48 | 32 | 15 |
| 第 1 趟 | 26 | 48 | 11 | 13 | 48 | 32 | 15 | **53** |
| 第 2 趟 | 26 | 11 | 13 | 48 | 32 | 15 | **48** | **53** |
| 第 3 趟 | 11 | 13 | 26 | 32 | 15 | **48** | **48** | **53** |
| 第 4 趟 | 11 | 13 | 26 | 15 | **32** | **48** | **48** | **53** |
| 第 5 趟 | 11 | 13 | 15 | **26** | **32** | **48** | **48** | **53** |
| 第 6 趟 | 11 | 13 | **15** | **26** | **32** | **48** | **48** | **53** |
| 第 7 趟 | 11 | **13** | **15** | **26** | **32** | **48** | **48** | **53** |

```
public int[] bubbleSort (int[] arr) {
      if(arr.length<2){
          return arr;
      }
      for(int i=0;i<arr.length-1;i++){
          for(int j=0;j<arr.length-i-1;j++){
              if(arr[j]>arr[j+1]){
                  swap(arr,j,j+1);
              }
          }
      }
      return arr;
  }

  public void swap(int[]arr,int i, int j){
      int tmp;
      tmp=arr[i];
      arr[i]=arr[j];
      arr[j]=tmp;
  }
```

**空间复杂度**仅仅使用一个辅助单元 ，因此空间复杂度为O(1)。

**时间复杂度** 假设待排序的元素个数为n，则总共需要进行n-1趟排序，对j个元素的子序列进行一趟排序需要进行j-1次关键字比较，因此总的比较次数为n(n-1)/2，因此时间复杂度为O(n^2)。

**稳定性**，冒泡排序的特点是稳定性好，因为排序过程中始终只交换相邻元素，比较对象大小相等时不交换，相对位置不变，故稳定。

**优化**，冒泡排序可以通过一些策略来提前结束，从而减少遍历的次数。常见的有两种方式：

- 当某一轮比较均未发生交换，说明排序已完成，可设置一个布尔值记录一轮排序是否有发生交换，若无则提前

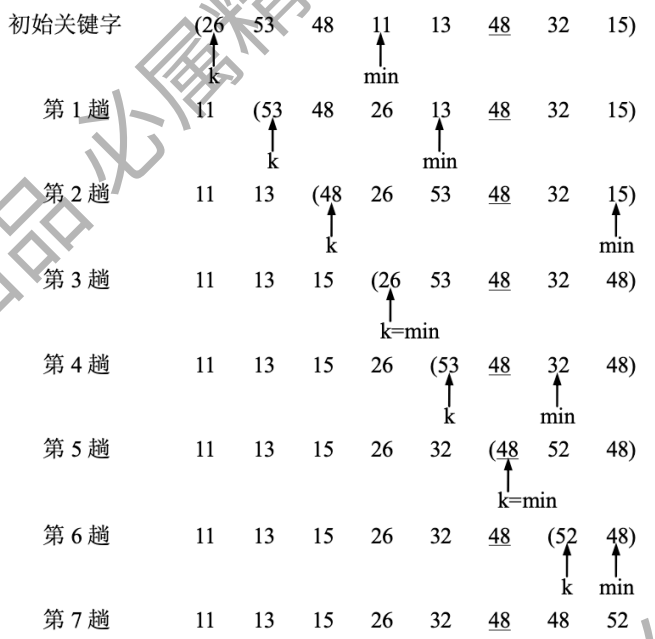
退出循环结束程序。

- 冒泡界优化，记录前一轮交换的最终位置，说明该位置之后的元素为已排序状态，下一轮的交换只需执行到该处。

# 1.3 选择排序

大话数据结构第九章

选择排序是默认前面都是已经排序好的，然后从后面 选择最小的放在前面排序好的的后面，首先第一轮循环的时候默认的排序好的为空，然后从后面选择最小的放到数组的第一个位置，第二轮循环的时候默认第一个元素是已经 排序好的，然后从剩下的找出最小的放到数组的第二个位置，第三轮循环的时候默认前 两个都是已经排序好的，然后再从剩下的选择一个最小的放到数组的第三个位置，以此 类推。还是上面的序列，我们看一下选择排序是怎么做的：




下面看一下代码：

```
public static void selectSort(int[] array) {
       for (int i = 0; i < array.length; i++) {
           int index = i;
           for (int j = i + 1; j < array.length; j++) {
               if (array[index] > array[j]) {
                   index = j;
               }
           }
           if (i != index) {
               swap(array, i, index);
           }
       }
   }

public static void swap(int[] A, int i, int j) {
      int tmp=A[i];
```

```
        A[i] = A[j];
        A[j] = tmp;
    }
```

我们看到每轮循环的时候并没有直接交换，而是从他后面的序列中找到最小的记录一下 他的index索引，最后再交换。

这个是大部分人都会的方法，我们这里再提供两种交换的骚操作，一个是基于位操作的方式，执行效率会更高，如果能理解并在面试时写出来，也是个不错的加分项。另一个是不声明变量的方法，这种方式曾经是很多大佬炫耀的骚写法：

```
     public static void swap2(int[] A, int i, int j) {
        A[i] ^= A[j];
        A[j] ^= A[i];
        A[i] ^= A[j];
    }
    public static void swap3(int[] A, int i, int j) {
        A[i] = A[i] + A[j];
        A[j] = A[i] - A[j];
        A[i] = A[i] - A[j];
    }
```

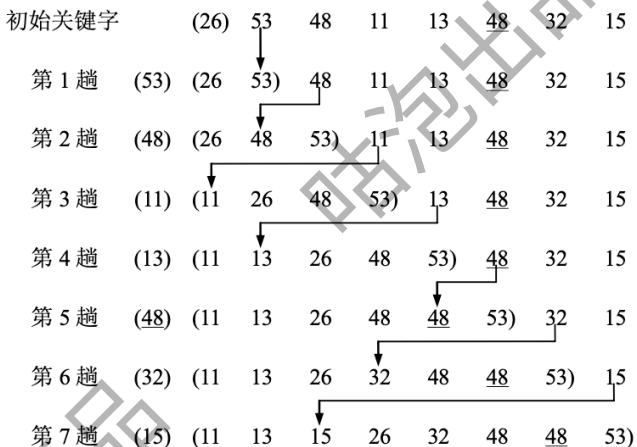
空间复杂度：仅仅使用一个辅助单元 ，因此空间复杂度为O(1)。

时间复杂度：在待排序序列已经有序的情况下，简单选择排序不用移动元素。最坏情况下，也就是序列正好是逆序的，则要进行n(n-1)/2次比较，因此最坏时间复杂度为O(n^2).

# 1.4 插入排序

插入排序的原理是默认前面的元素都是已经排序好的，然后从后面逐个读取插入到前面排序好的合适的位置，就相当于打扑克的时候每获取一张牌的时候就插入到合适的位置一样。

我们仍然以关键字序列{26,53,48,11,13,48,32,15}为例，插入排序的过程如下所示：




插入排序可以分为两种，一种是直接插入还一种是二分法插入，直接插入的原理比较简单，就是往前逐个查找直到找到合适的位置然后插入，二分法插入是先折半查找，找到合适的位置然后再插入。我们先看一下简单的直接插入排序代码：

```
public int[] insertSort (int[] arr) {
        if(arr==null || arr.length<2){
            return null;
        }
        for(int i=1;i<arr.length;i++){
            int j=i;
            int temp=arr[i];
            for(;j>0;j--){
                if(arr[j-1]>temp){
                    arr[j]=arr[j-1];
                }else{
                    break;
                }
            }
            arr[j]=temp;
        }
        return arr;
 }
```

这里还可以在查找的时候使用二分查找，数据量大的时候，效率肯定更高，但是编写的难度也高了。插入排序要进行大量的元素移动，所以即使增加了二分查找，效率本身不见得高多少，这里只做了解：

```
public int[] insertSort (int[] arr) {
        if(arr==null || arr.length<2){
            return null;
        }
        for(int i=1;i<arr.length;i++){
          if(arr[i-1]>arr[i]){
              int key=arr[i];
              int low=0;
              int high=i-1;
              while(low<=high){
                  int mid=(low+high)>>1;
                  if(arr[mid]>key){
                      high=mid-1;
                  }else{
                      low=mid+1;
                  }
              }
              for(int j=i;j>low;j--){
                  arr[j]=arr[j-1];
              }
              arr[low]=key;
          }
        }
        return arr;
}
```

空间复杂度：仅仅使用一个辅助单元 ， 因此空间复杂度为O(1)。

时间复杂度:假设待排序的元素个数为n，则向有序表中逐个插入记录的操作进行了n-1次，每趟操作分为比较关键码和移动记录，而比较的次数和移动的记录次数取决于待排序列按关键码的初始化排列。

（1）在最好的情况下，也即待排序序列已按照关键字有序，每趟操作只需1次比较和0次移动，此时有：

总比较次数=n-1次

总移动次数：0次。

（2）最坏情况下，也就是原始序列正好逆序的，这时在第j趟操作中，为插入元素需要同前面的j个元素进行j次关键字比较，移动元素的次数为j+1次，此时有：

$$\text{总比较次数} = \sum_{j=1}^{n-1} j = n(n-1)/2 \text{ 次}$$

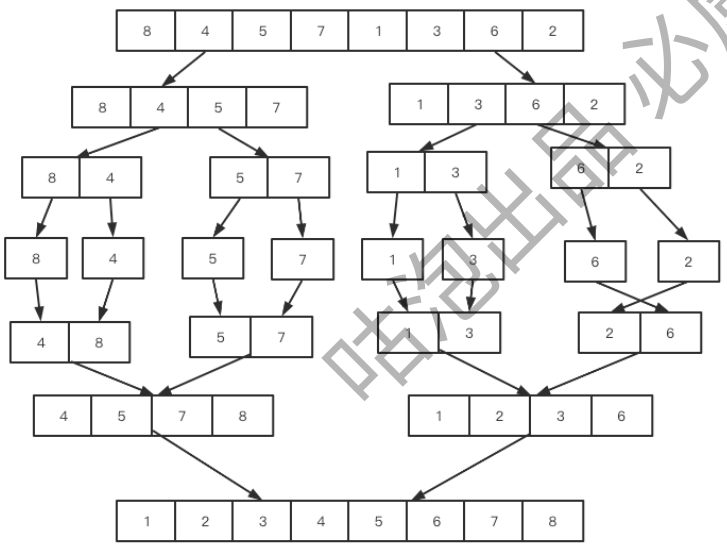
$$\text{总移动次数} = \sum_{j=1}^{n-1} (j+1) = (n+2)(n-1)/2 \text{ 次}$$

对于折半插入的情况，从时间上看，折半仅仅减少了元素的比较次数，没有减少元素的移动次数，因此时间复杂度仍然为O(n^2)。

# 2.归并排序

大话数据结构第九章

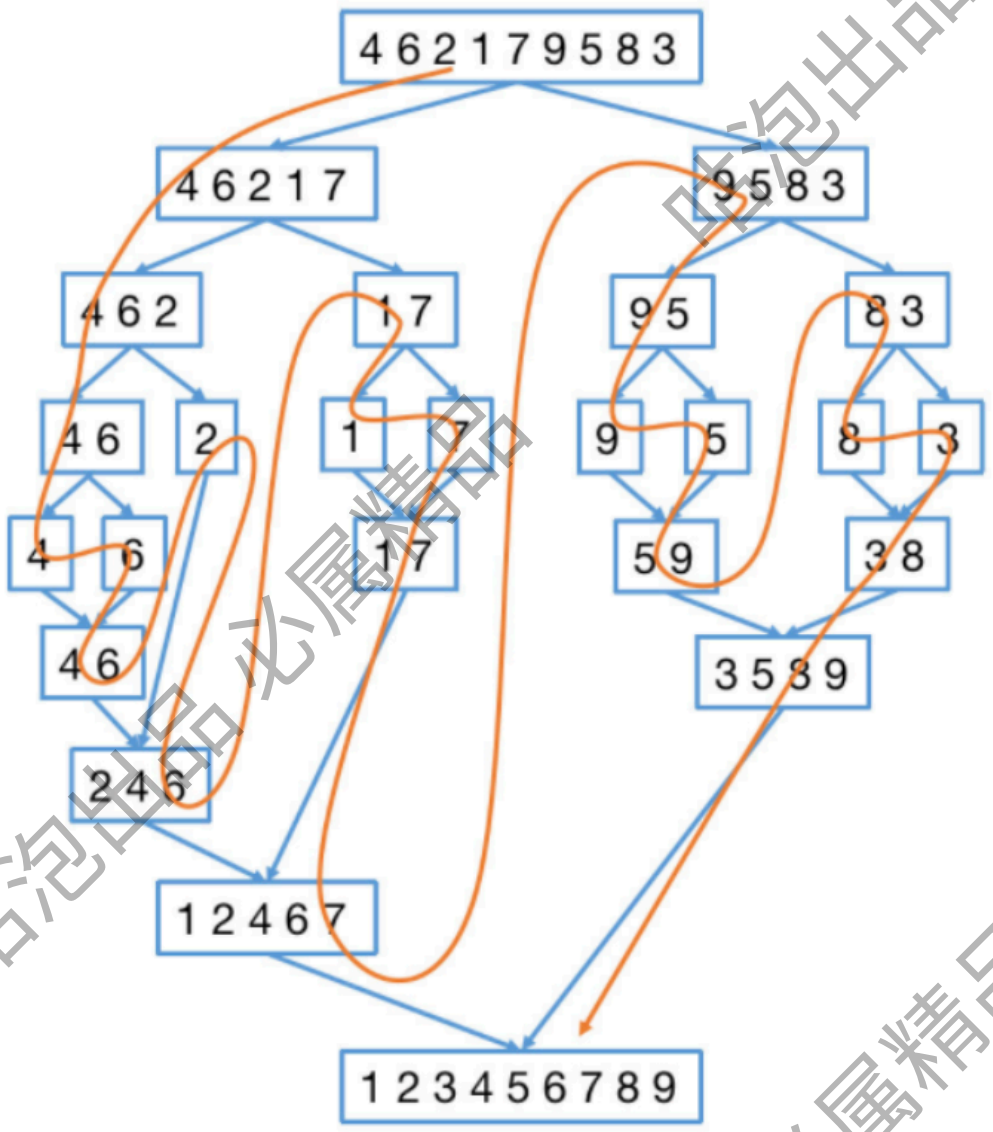
归并排序（MERGE-SORT）简单来说就是将大的序列先视为若干个比较小的数组，分成几个比较小的结构，然后是利用**归并**的思想实现的排序方法，该算法采用经典的分治策略（分就是将问题**分**(divide)成一些小的问题分别求解，而**治(conquer)**则将分的阶段得到的各答案"合"在一起)。




可以看到这种结构很像两棵套在一起的满二叉树。**分**阶段可以理解为就是递归拆分子序列的过程，递归深度为logn。就是图中上面侧的满二叉树。

再来看看**治**阶段，我们需要将两个已经有序的子序列合并成一个有序序列，就是下侧的满二叉树。比如上图中的最后一次合并，要将[4,5,7,8]和[1,2,3,6]两个已经有序的子序列，合并为最终序列[1,2,3,4,5,6,7,8]，这个操作与合并两个有序数组的完全一样，不同的是这里是将数组的两个部分合并。

在看一下遍历时处理元素的过程：




实现代码：

```
void mergeSort(int[] array, int start, int end, int temp[]) {
     if (start >= end) {
         return;
     }
     //先递归访问到每个元素(也在结点)，得到元素之后再分别处理(两个合并)
     //与二叉树的后序遍历非常像
     mergeSort(array, start, (start + end) >>1, temp);
     mergeSort(array, (start + end) >>1 + 1, end, temp);
     merge(array, start, end, temp);
 }

  void merge(int[] array, int start, int end, int[] temp) {
     int middle = (start + end) / 2;
     int left = start;
     int right = middle + 1;
     int index = left;
```

```
        while (left <= middle && right <= end) {
            if (array[left] < array[right]) {
                temp[index++] = array[left++];
            } else {
                temp[index++] = array[right++];
            }
        }
        while (left <= middle) {
            temp[index++] = array[left++];
        }
        while (right <= end) {
            temp[index++] = array[right++];
        }
        for (int i = start; i <= end; i++) {
            array[i] = temp[i];
        }

    }
```

再写一个测试类：

```
    public static void main(String[] args) {
        int[] array = {6, 3, 2, 1, 4, 5, 8, 7};
        int[] temp = new int[array.length];
        mergeSort(array, 0, array.length - 1, temp);
        System.out.println(Arrays.toString(array));
    }
```

归并排序是将一个数组里面的元素进行排序的有效手段。

空间复杂度：归并时需要一个大小最多为n的数组，因此空间复杂度为O(n)

时间复杂度：归并是一个典型的分治算法，为O(nlogn)。

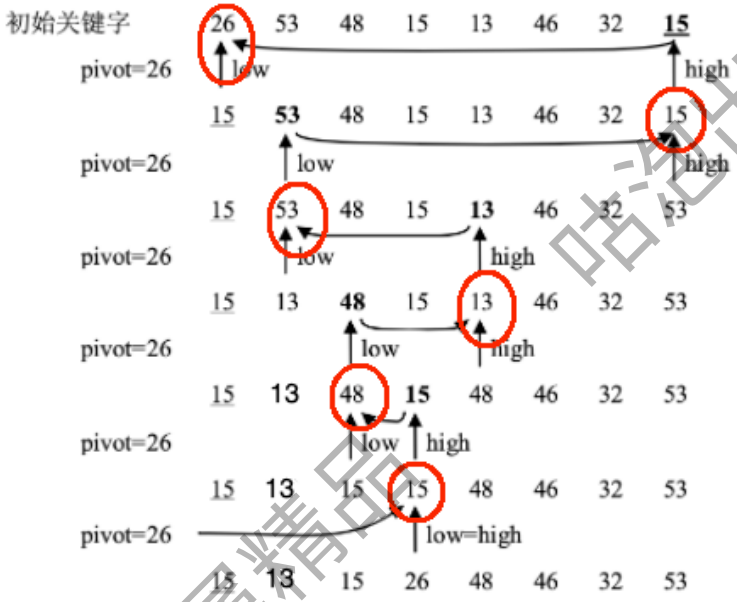
# 3. 快速排序

快速排序也是我们在算法书里面认识的老朋友了。我们在《一维数组》一章提到过"双指针思路"：在处理奇偶等情况时会使用两个游标，一个从前向后，一个是从后向前来比较，根据结果来决定继续移动还是停止等待。快速排序的每一轮进行的时候都是类似的双指针策略，而递归的过程本质上就是二叉树的前序递归调用。

## 3.1 快速排序的基本过程

快速排序是将分治法运用到排序问题的典型例子，基本思想是：通过一个标记pivot元素将n个元素的序列划分为左右两个子序列left和right，其中left中的元素都比pivot小，right的都比pivot的大，然后再次堆left和right各自再执行快速排序，在将左右子序列排好序之后，整个序列就有序了。这里排序进行左右划分的时候是一直划分到子序列只包含一个元素的情况，然后再递归返回。

我们以关键字序列{26,53,48,15,13,48,32,15}看一下一次划分的过程:




上面红框位置表示当前已经被赋值给了pivot或者其他位置，可以空出来放移动来的新元素了。我们可以看到26最终被放到了属于自己的位置上，不会再变化。而左侧的都比15小，左侧都比15大，因此26的左右两侧可以分别再进行排序。

这一轮过程是什么呢？就是数组增删的时候经常用的双指针策略，我们在数组部分讲过，不再赘述。而这里的每一轮都是一个相向的双指针而已，没有任何神秘的。

至于实现，网上有很多，类型千差万别，有的还定义了多个方法，一看就复杂，基本不太可能掌握，我认为下面这种方式最简单、最直接了，代码如下：

```
void quickSort(int[] array, int start, int end) {
        if (start >= end) {
            return;
        }
        //这里就是一个相向的双指针操作
        int left = start, right = end;
        int pivot = array[(start + end) / 2];

        while (left <= right) {
            while (left <= right && array[left] < pivot) {
                left++;
            }
            while (left <= right && array[right] > pivot) {
                right--;
            }
            if (left <= right) {
                int temp = array[left];
                array[left] = array[right];
                array[right] = temp;
                left++;
                right--;
            }
        }
```

```
        //先处理元素再分别递归处理两侧分支，与二叉树的前序遍历非常像
        quickSort(array, start, right);
        quickSort(array, left, end);
    }
```

测试方法：

```
    public static void main(String[] args) {
        int[] array = {6, 3, 2, 1, 4, 5, 8, 7};
        quickSort(array, 0, array.length - 1);
        System.out.println(Arrays.toString(array));
    }
```

**复杂度分析**

快速排序的时间复杂度计算比较麻烦一些。从原理来看，如果我们选择的pivot每次都正好在中间，效率是最高的，但是这是无法保证的，因此我们需要从最好、最坏和中间情况来分析

- 最坏情况就是如果每次选择的恰好都是low结点作为pivot，如果元素恰好都是逆序的，此时时间复杂度为O(n^2)
- 如果元素恰好都是有序的，则时间复杂度为O(n^2)
- 折中的情况是每次选择的都是中间结点，此时序列每次都是长度相等的序列，此时的时间复杂度为(O(nlogn))

# 3.3 数组中第K大的数字

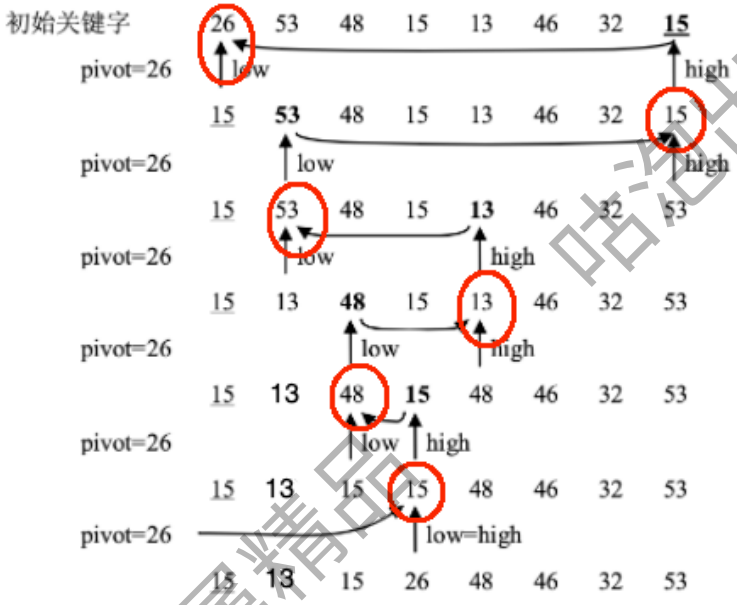
LeetCode215 数组中的第K个最大元素。给定整数数组 `nums` 和整数 `k`，请返回数组中第 `**k**` 个最大的元素。请注意，你需要找的是数组排序后的第 `k` 个最大的元素，而不是第 `k` 个不同的元素。

```
示例1:
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5

示例2:
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4
```

我们在堆部分分析过这个问题，这里看看如何基于快速排序来做，这个题目出现的频率非常高，甚至在很多时候，面试官直接要求基于快速排序来解决这个问题。而且我们要直接改造一下上面的快排来解决，而不是另起炉灶，只有这样平时的练习才有效果。

为什么能用快速排序来解决呢？我们还是看上面排序的序列：{26,53,48,15,13,48,32,15}

我们第一次选择了26为哨兵，进行一轮的排序过程为：




上面红框位置表示当前已经被赋值给了pivot或者其他位置，可以空出来放移动来的新元素了。我们可以看到26最终被放到了属于自己的位置上，不会再变化，而26的左右两侧可以分别再进行排序。

这里还有一个关键信息， 我们可以知道26的索引为3，所以递增排序之后26一定是第4大的元素 。这就是解决本问题的关键，既然知道26是第4大，那如果我要找第2大，一定是要到右边找。如果要找第6大，一定要到左边找（当然，如果降序排序就反过来了），而不需要的那部分就不用管了。这就是为什么能用快速排序解决这个问题。

我们仍然采用升序排列，这样就可以直接改造上面的快速排序的代码来实现。

```
public int findKthLargest(int[] nums, int k) {
       quicksort(nums,0,nums.length-1);
       return nums[k-1];         //因为是从零开始所以k-1;

    }

public  void quicksort(int[] nums, int left, int right) {
        int start = left;
        int end = right;
        int pivot = nums[(end + start) / 2]; //取中位值作左右参考对比

        //每次循环都将大的放到左边，小的放到右边
        while (start < end) {
            while (nums[start] > pivot) {
                start++;
            }
            while (nums[end] < pivot) {
                end--;
            }
            //如果l>=r说明mid左边的值全部大于等于mid的值，右边全是小的，退出
            if (start >= end) {
                break;
            }
            //交换
```

```
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            //解决值相同的情况，如果交换后发现左边的值等于mid目标值，则使右边指针向左移
            if (nums[start] == pivot) {
                end--;
            }
            //右边的值等于mid目标值，则使左边指针向左移
            if (nums[end] == pivot) {
                start++;
            }
        }

        if (start == end) { //退出循环防止栈溢出
            start++;
            end--;
        }
        if (left < end) {
            quicksort(nums, left, end);   //向左递归
        }
        if (right > start) {
            quicksort(nums, start, right); //向右递归
        }
    }
```

给一个测试方法：

```
    public static void main(String[] args) {
        int[] array = {6, 3, 2, 4, 5, 8, 7};
        int k = 2;//找第二大元素
        int n = array.length - k + 1;
        System.out.println(quickSort(array, 0, array.length - 1, n));
    }
```

我们尽量能套用已经练习过的代码，否则核心逻辑都要考虑相反情况，会让我们面试时现场写变得非常困难。

# 4.几种高级排序算法

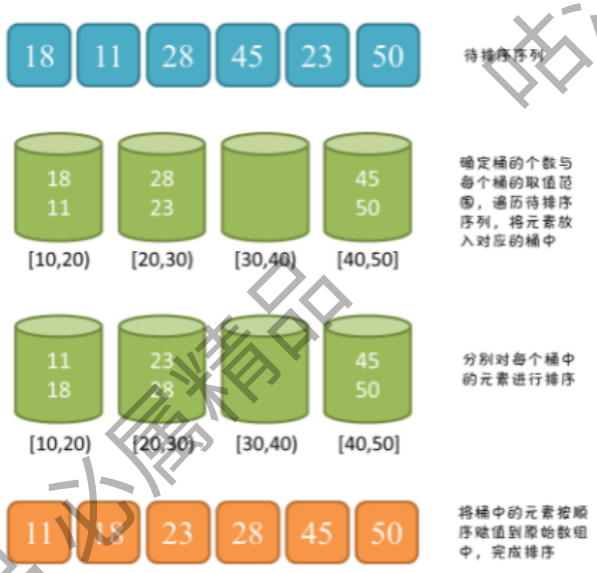
前面我们深入分析了归并排序和快速排序，这两个是面试中最重要的排序算法。除此之外还有几种排序方法，出现的可能性比较低，我们继续看一下。这几种排序算法是：桶排序、基数排序、希尔排序和位图排序，位图排序我们到高级算法课的《位图与异或》会对其进行完整深入的分析。

## 4.1 桶排序

桶排序是将数组分散到有限的桶中，然后每个桶再分别排序，而每个桶的排序又可以使用其他排序方式进行排序，可以是桶排序也可以是其他排序。一句话就是：划分多个范围相同的区间，每个子区间自排序，最后合并。

桶排序是计数排序的扩展，计数排序可以看成每个桶只存储相同元素，而桶排序每个桶存储一定范围的元素，通过映射函数，将待排序数组中的元素映射到各个对应的桶中，对每个桶中的元素进行排序，最后将非空桶中的元素逐个放入原序列中。

桶排序需要尽量保证元素分散均匀，否则当所有数据集中在同一个桶中时，桶排序失效。如下图所示：




桶的大小可以随便定，如果桶的数量足够多就会变成我们后面介绍的计数排序，其实我们完全可以把桶固定在一 个数量，根据数组的大小来确定，也可以自己定，比如3个或者5个7个等，桶的大小确 定之后，下一步就需要把数组中的值一一存放到桶里，小的值就会放到前面的桶里，大 的值就会放到后面的桶里，中间的值就会放到中间的桶里，然后再分别对每个桶进行单 独排序，最后再把所有桶的数据都合并到一起就会得到排序好的数组。
参考实现：

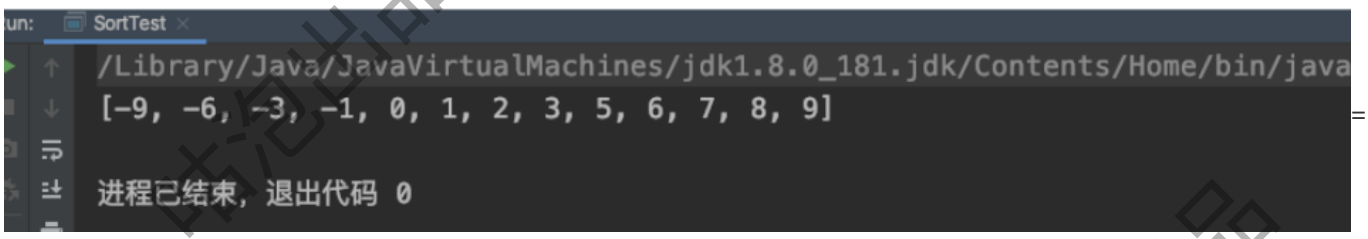
```
public static void bucketSort(int[] array, int bucketSize) {
        int arrayLenth = array.length;
        int max = array[0];
        int min = array[0];

        for (int i = 0; i < arrayLenth; i++) {
            if (array[i] > max)
                max = array[i];
            else if (array[i] < min)
                min = array[i];
        }

        int bucketCount = (max - min) / bucketSize + 1;
        List<List<Integer>> buckets = new ArrayList<List<Integer>>(bucketCount);
        for (int i = 0; i < bucketCount; i++) {
            buckets.add(new ArrayList<Integer>());
        }

        for (int i = 0; i < arrayLenth; i++) {
            buckets.get((array[i] - min) / bucketSize).add(array[i]);
        }

        int currentIndex = 0;
```

```
        for (int i = 0; i < buckets.size(); i++) {
            Integer[] bucketArray = new Integer[buckets.get(i).size()];
            bucketArray = buckets.get(i).toArray(bucketArray);
            Arrays.sort(bucketArray);//我们要实现一个比较复杂的算法，这里理所应当调用库啦
            for (int j = 0; j < bucketArray.length; j++) {
                array[currentIndex++] = bucketArray[j];
            }
        }
    }
```

然后我们写个main方法测试一下：

```
public static void main(String[] args) {
        int array[]={2,6,9,3,5,1,-9,7,-3,-1,-6,8,0};
        bucketSort(array,3);
        System.out.println(Arrays.toString(array));
    }
```

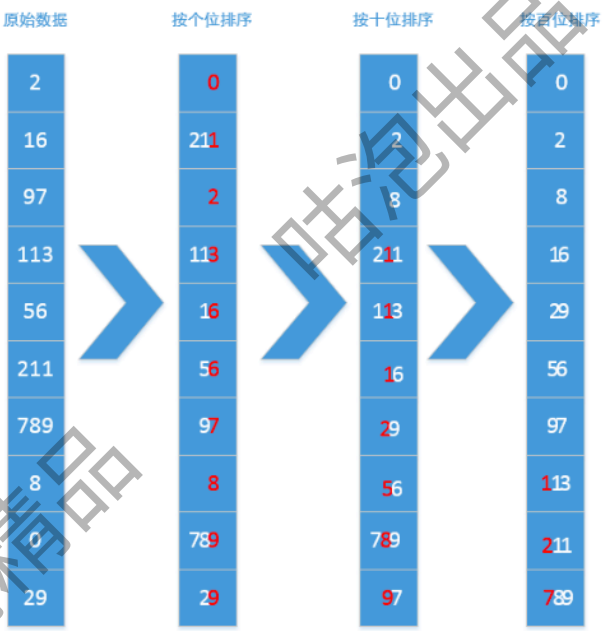
执行结果;




## 4.2 基数排序

基数排序的原理其实非常简单，看一个例子：假如要排序的数据是：{0,2,8,16,29,56,97,113,211,789}。

我们该怎么排呢：



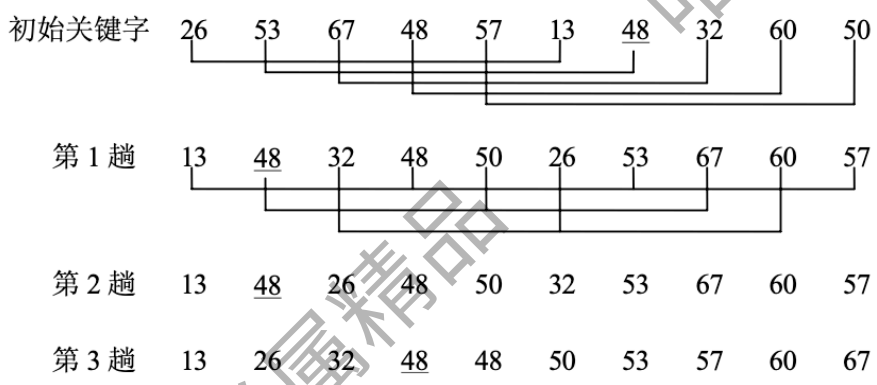
我们首先按照各位对所有的数据排序，然后按照十位排序，然后再按照百位排。等每个位上的都排序完了之后整个数组 也就排序完成了。但是上面代码还不是很完美，因为当出现负数的时候上面代码就没法 排序了。

```
public static void radixSort(int[] array) {
        int digitCount = 10;
        int maxCount = getBitCount(getMaxNum(array));
        int radix = 1;
        int[][] tempArray = new int[digitCount][array.length];
        for (int i = 0; i < maxCount; i++) {
            int[] count = new int[digitCount];
            for (int j = 0; j < array.length; j++) {
                int temp = ((array[j] / radix) % 10);
                tempArray[temp][count[temp]++] = array[j];
            }
            int index = 0;
            for (int j = 0; j < digitCount; j++) {
                if (count[j] == 0) {
                    continue;
                }
                for (int k = 0; k < count[j]; k++) {
                    array[index++] = tempArray[j][k];
                }
            }
            radix *= 10;
        }
    }

    public static int getBitCount(int num) {
        int count = 1;
        int temp = num / 10;
        while (temp != 0) {
            count++;
            temp /= 10;
        }
        return count;
    }

    public static int getMaxNum(int array[]) {
        int max = array[0];
        for (int i = 1, lengh = array.length; i < lengh; i++) {
            if (array[i] > max) {
                max = array[i];
            }
        }
        return max;
    }
```

这里可以采取一些措施处理负数的问题，这里我们就不看了 ， 感兴趣的小伙伴自行查一下。

## 4.3 希尔排序

希尔排序也成缩小增量排序，原理是将待排序列划分为若干组，每组都是不连续的，有 间隔step，step可以自己定，但间隔step最后的值一定是1，也就说最后一步是前后两 两比较。间隔为step的默认划分为一组，先在每一组内进行排序，以使整个序列基本有 序，然后再减小间隔step的值，重新分组再排序……不断重复，直到间隔step小于1则停 止。我们以关键字序列{26,53,48,11,13,48,32,15}为例，执行过程：




上面的排序与冒泡很像，只不过冒泡排序是每次都是间隔为1相邻的两个之间进行比较，而希尔是间隔为step。看代码：

```
public static void shellSort1(int[] array) {
        int length = array.length;
        int step = length >> 1;
        while (step >= 1) {
            for (int i = step; i < length; i++) {
                for (int j = i; j >= step; j -= step) {
                    if (array[j] < array[j - step]) {
                        swap(array, j, j - step);
                    } else {
                        break;
                    }
                }
            }
            step >>= 1;
        }
}
```

**复杂度分析**

希尔排序的时间复杂度比较特殊，不像其他算法那么明确。主要是与步长密切相关，太大太小都不行，如何选择最佳步长是一个有待解决的问题，因此时间复杂度也不是明确的。目前的一些研究表明，当步长delta[k]=2^(t-k_1)-1时，希尔排序的时间复杂度为O(n^(3/2))。

# 5.排序综合比较

我们上面重点分析了一些常见的算法的实现原理和复杂度的问题，其实还有一个比较重要的指标就是稳定性。

假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且r[i]在r[j]之前，而在排序后的序列中，r[i]仍在r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。

堆排序、快速排序、希尔排序和直接选择排序是不稳定的算法。而冒泡排序、插入排序、归并排序是稳定的排序算法。我们分别看一下：

首先，排序算法的稳定性大家应该都知道，通俗地讲就是能保证排序前2个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。在简单形式化一下，如果Ai = Aj, Ai原来在位置前，排序后Ai还是要在Aj位置前。

其次，说一下稳定性的好处。排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。基数排序就是这样，先按低位排序，逐次按高位排序，低位相同的元素其顺序再高位也相同时是不会改变的。

回到主题，现在分析一下常见的排序算法的稳定性，每个都给出简单的理由。

(1)冒泡排序

冒泡排序就是把小的元素往前调或者把大的元素往后调。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，我想你是不会再无聊地把他们俩交换一下的；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，所以相同元素的前后顺序并没有改 变，所以冒泡排序是一种稳定排序算法。

(2)选择排序

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，如果当前元素比一个元素小，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。比较拗口，举个例子，序列5 8 5 2 9， 我们知道第一遍选择第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。

(3)插入排序

插入排序是在一个已经有序的小序列的基础上，一次插入一个元素。当然，刚开始这个有序的小序列只有1个元素，就是第一个元素。比较是从有序序列的末尾开 始，也就是想要插入的元素和已经有序的最大者开始比起，如果比它大则直接插入在其后面，否则一直往前找直到找到它该插入的位置。如果碰见一个和插入元素相 等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，所以插入排序是稳 定的。

(4)快速排序

快速排序有两个方向，左边的i下标一直往右走，当a[i] <= a[center_index]，其中center_index是中枢元素的数组下标，一般取为数组第0个元素。而右边的j下标一直往左走，当a[j] > a[center_index]。如果i和j都走不动了，i <= j, 交换a[i]和a[j],重复上面的过程，直到i>j。 交换a[j]和a[center_index]，完成一趟快速排序。在中枢元素和a[j]交换的时候，很有可能把前面的元素的稳定性打乱，比如序列为 5 3 3 4 3 8 9 10 11， 现在中枢元素5和3(第5个元素，下标从1开始计)交换就会把元素3的稳定性打乱，所以快速排序是一个不稳定的排序算法，不稳定发生在中枢元素和a[j] 交换的时刻。

(5)归并排序

归并排序是把序列递归地分成短序列，递归出口是短序列只有1个元素(认为直接有序)或者2个元素(1次比较和交换),然后把各个有序的段序列合并成一个有 序的长序列，不断合并直到原序列全部排好序。可以发现，在1个或2个元素时，1个元素不会交换，2个元素如果大小相等也没有人故意交换，这不会破坏稳定 性。那么，在短的有序序列合并的过程中，稳定是否受到破坏？没有，合并过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结 果序列的前面，这样就保证了稳定性。所以，归并排序也是稳定的排序算法。

(6)基数排序

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。基数排序基于分别排序，分别收集，所以其是稳定的排序算法。

(7)希尔排序(shell)

希尔排序是按照不同步长对元素进行插入排序，当刚开始元素很无序的时候，步长最大，所以插入排序的元素个数很少，速度很快；当元素基本有序了，步长很小，插入排序对于有序的序列效率很高。所以，希尔排序的时间复杂度会比o(n^2)好一些。由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元 素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以shell排序是不稳定的。

(8)堆排序

我们知道堆的结构是节点i的孩子为2*i*和2i+1节点，大顶堆要求父节点大于等于其2个子节点，小顶堆要求父节点小于等于其2个子节点。在一个长为n 的序列，堆排序的过程是从第n/2开始和其子节点共3个值选择最大(大顶堆)或者最小(小顶堆),这3个元素之间的选择当然不会破坏稳定性。但当为n /2-1, n/2-2, ...1这些个父节点选择元素时，就会破坏稳定性。有可能第n/2个父节点交换把后面一个元素交换过去了，而第n/2-1个父节点把后面一个相同的元素没有交换，那么这2个相同的元素之间的稳定性就被破坏了。所以，堆排序不是稳定的排序算法。

综上，得出结论: 选择排序、快速排序、希尔排序、堆排序不是稳定的排序算法，而冒泡排序、插入排序、归并排序和基数排序是稳定的排序算法。

最后看一个汇总的表格：

| 排序算法 | 平均时间复杂度 | 最好情况 | 最坏情况 | 空间复杂度 | 排序方式 | 稳定性 |
| --- | --- | --- | --- | --- | --- | --- |
| 冒泡排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | In-place | 稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | In-place | 不稳定 |
| 插入排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | In-place | 稳定 |
| 希尔排序 | $O(n \log n)$ | $O(n \log^2 n)$ | $O(n \log^2 n)$ | $O(1)$ | In-place | 不稳定 |
| 归并排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | Out-place | 稳定 |
| 快速排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(\log n)$ | In-place | 不稳定 |
| 堆排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | In-place | 不稳定 |
| 计数排序 | $O(n + k)$ | $O(n + k)$ | $O(n + k)$ | $O(k)$ | Out-place | 稳定 |
| 桶排序 | $O(n + k)$ | $O(n + k)$ | $O(n^2)$ | $O(n + k)$ | Out-place | 稳定 |
| 基数排序 | $O(n \times k)$ | $O(n \times k)$ | $O(n \times k)$ | $O(n + k)$ | Out-place | 稳定 |

# 6 大厂实战

如果要深入研究，排序还有很多问题可以讨论，这个给一个不错的博客：

https://leetcode.cn/circle/discuss/eBo9UB/#%E7%AE%97%E6%B3%95%E6%8F%8F%E8%BF%B0-2

基于排序来解决的问题也不少，很多大厂出的比较难的题目都与之类似，这里给一些常见的问题，感兴趣的同学可以继续研究：

912. 排序数组

56. 合并区间

57. 插入区间

164. 最大间距

169. 多数元素

179. 最大数

215. 数组中的第K个最大元素

210. 课程表 II

217. 存在重复元素

242. 有效的字母异位词

252. 会议室

253. 会议室 II

268. 丢失的数字

279. 完全平方数

274. H 指数

275. H 指数 II

280. 摆动排序

324. 摆动排序 II

378. 有序矩阵中第 K 小的元素

406. 根据身高重建队列

407. 接雨水 II

853. 车队

912. 排序数组

937. 重新排列日志文件

945. 使数组唯一的最小增量

962. 最大宽度坡

969. 煎饼排序

973. 最接近原点的 K 个点

977. 有序数组的平方