数据结构与算法题解（2）：数组题解

数组相关题解java实现。

一、二维数组中的查找（剑3）

在一个二维数组中，每一行都按照从左到右的递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一维数组和一个整数，判断数组中是否含有该整数。

首先选取数组中右上角的数字，如果该数字等于我们要查找的数组，查找过程结束；如果该数字大于要查找的数组，剔除这个数字所在的列；如果该数字小于要查找的数组，剔除这个数字所在的行。也就是说如果要查找的数字不再数组的右上角，则每一次都在数组的查找范围中剔除一行或者一列，这样每一步都可以缩小查找的范围，直到找到要查找的数字，或者查找范围为空。

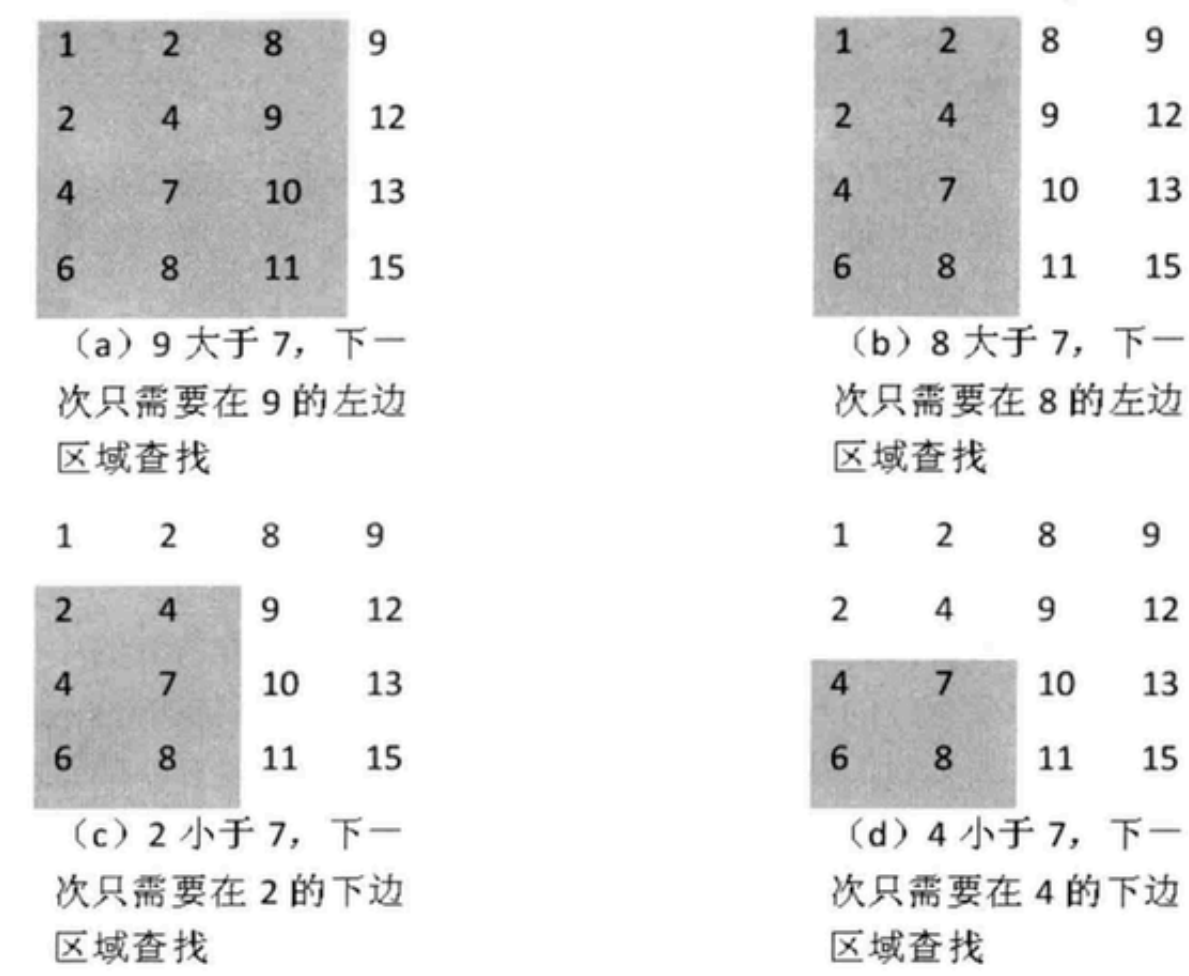


图 2.2　在二维数组中查找 7 的步骤

注：矩阵中加阴影背景的区域是下一步查找的范围。

> java

```java
public class Solution {
    public boolean Find(int target, int [][] array) {
        int row = 0;
        int col = array[0].length - 1;
        while(row<=array.length-1&&col>=0){
            if(target == array[row][col]){
                return true;
            }
            else if(target>array[row][col]){
                row++;
            }
            else{
                col--;
            }
        }
        return false;

    }
}
```

> python

```python
# -*- coding:utf-8 -*-
class Solution:
    # array 二维列表
    def Find(self, target, array):
        # write code here
        row = 0
        col = len(array[0])-1
        while row<=len(array)-1 and col>=0:
            if target==array[row][col]:
                return True
            elif target>array[row][col]:
                row+=1
            else:
                col-=1
        return False
```

也可以把每一行看做是一个递增的序列，利用二分查找。

> java

```java
public class Solution {
    public boolean Find(int target, int [][] array) {
        for(int i=0;i<array.length;i++){
```

```
            int low =0;
            int high = array[i].length-1;
            while(low<=high){
                int mid = (low+high)/2;
                if(array[i][mid]==target)
                    return true;
                else if(array[i][mid]>target)
                    high =mid-1;
                else
                    low=mid+1;
            }

        }
        return false;
    }
}
```

二、用两个栈实现队列（剑7）

栈是一个非常常见的数据结构，它在计算机领域中被广泛应用，比如操作系统会给每个线程创建一个栈来存储函数调用时各个函数的参数、返回地址及临时变量等。栈的特点是后进先出，即最后被压入（push）栈的元素会第一个被弹出（pop）。

队列是另外一种很重要的数据结构。和栈不同的是，队列的特点是先进先出，即第一个进入队列的元素将会第一个出来。

栈和队列虽然是针锋相对的两个数据结构，但有意思的是他们却相互联系。

通过一个具体的例子来分析往队列插入和删除元素的过程。首先插入一个元素a，不妨先把它插入到stack1，此时stack1中的元素有{a}，stack2为空，再向stack1压入b和c，此时stack1中的元素有{a,b,c}，其中c处于栈顶，而stack2仍然是空的。

因为a是最先进的，最先被删除的元素应该是a，但a位于栈低。我们可以把stack1中的元素逐个弹出并压入stack2，元素在stack2的顺序正好和原来在stack1的顺序相反因此经过三次弹出stack1和压入stack2操作之后，stack1为空，而stack2的元素是{c,b,a}，这时就可以弹出stack2的栈顶a了，随后弹出stack2中的b和c，而这个过程中stack1始终为空.

从上面的分析我们可以总结出删除一个元素的步骤：**当stack2中不为空时，在stack2的栈顶元素是最先进入队列的元素，可以弹出。如果stack2为空时，我们把stack1中的元素逐个弹出并压入stack2。由于先进入队列的元素被压到stack1的底端，经过弹出和压入之后就处于stack2的顶端了，又可以直接弹出。**

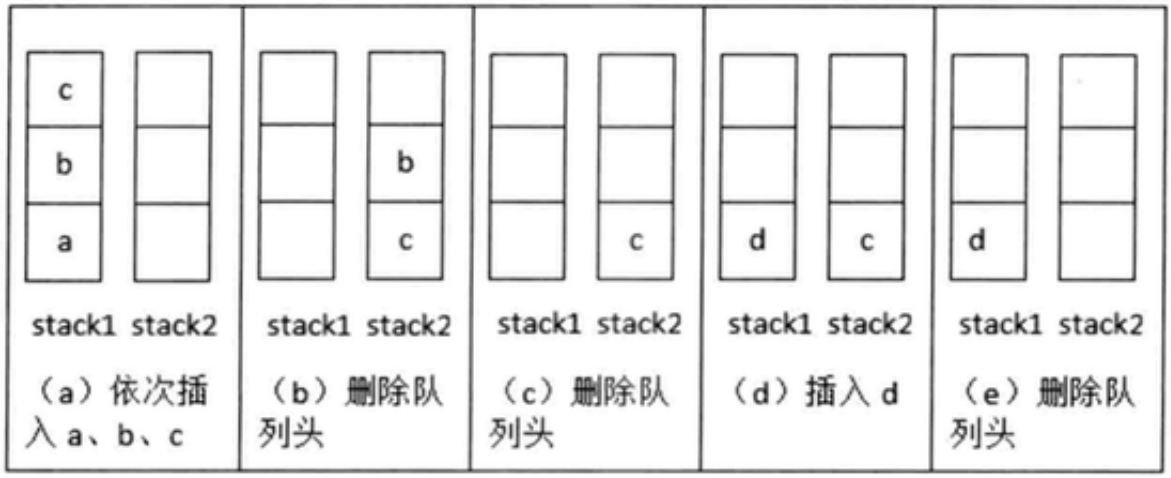


图 2.8　用两个栈模拟一个队列的操作

```
import java.util.Stack;
public class Solution {
    Stack<Integer> stack1 = new Stack<Integer>();
    Stack<Integer> stack2 = new Stack<Integer>();

    public void push(int node) {
        stack1.push(node);
    }

    public int pop() {
        while(!stack2.isEmpty()){
            return stack2.pop();
        }
        while(!stack1.isEmpty()){
            stack2.push(stack1.pop());
        }
        return stack2.pop();

    }
}
```

三、旋转数组的最小数字（剑8）

在准备面试的时候，我们应该重点掌握二分查找、归并排序和快速排序，做到能随时正确、完整地写出它们的代码。

若面试题是要求在排序的数组（或部分排序的数组）中查找一个数字或者统计某个数字出现的次数，我们都可以尝试用二分查找算法。

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。 输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。 例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。

可以采用二分法解答这个问题， mid = low + (high - low)/2 ，需要考虑三种情况：

1. array[mid] > array[high]: 出现这种情况的array类似[3,4,5,6,0,1,2]，此时最小数字一定在mid的右边。 low = mid + 1
2. array[mid] == array[high]: 出现这种情况的array类似 [1,0,1,1,1] 或者[1,1,1,0,1]，此时最小数字不好判断在mid左边，还是右边,这时只好一个一个试，low = low + 1 或者 high = high - 1
3. array[mid] < array[high]: 出现这种情况的array类似[2,2,3,4,5,6,6],此时最小数字一定就是array[mid]或者在mid的左边。因为右边必然都是递增的。 high = mid。注意这里有个坑：如果待查询的范围最后只剩两个数，那么mid一定会指向下标靠前的数字，比如 array = [4,6], array[low] = 4 ;array[mid] = 4 ; array[high] = 6 ; 如果high = mid - 1，就会产生错误， 因此high = mid，但情形(1)中low = mid + 1就不会错误。

代码如下：

> java

```java
import java.util.ArrayList;
public class Solution {
    public int minNumberInRotateArray(int [] array) {
        int low = 0;
        int high = array.length-1;
        while(low<high){
            int mid = low+(high-low)/2;
            if(array[mid]>array[high]){
                low=mid+1;
            }
            else if(array[mid]==array[high]){
                high=high-1;
            }
            else{
                high = mid;
            }

        }
        return array[low];

    }
}
```

四、斐波那契数列（剑9）

4.1 斐波那契数列

大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项。n<=39。这个题可以说是迭代（Iteration） VS 递归（Recursion），f(n) = f(n-1) + f(n-2)，第一眼看就是递归啊，简直完美的递归环境，递归肯定很爽，这样想着关键代码两三行就搞定了，注意这题的n是从0开始的：

```
if(n<=1) return n;
else return Fibonacci(n-1)+Fibonacci(n-2);
```

然而并没有什么用，测试用例里肯定准备着一个超大的n来让Stack Overflow，为什么会溢出？因为重复计算，而且重复的情况还很严重，举个小点的例子，n=4，看看程序怎么跑的：

```
Fibonacci(4) = Fibonacci(3) + Fibonacci(2);
                    = Fibonacci(2) + Fibonacci(1) + Fibonacci(1) + Fibonacci(0);
                    = Fibonacci(1) + Fibonacci(0) + Fibonacci(1) + Fibonacci(1) + F
ibonacci(0);
```

由于我们的代码并没有记录Fibonacci(1)和Fibonacci(0)的结果，对于程序来说它每次递归都是未知的，因此光是n=4时f(1)就重复计算了3次之多。

更简单的办法是从下往上计算，首先根据f(0)和f(1)算出f(2)，再根据f(1)和f(2)算出f(3).....依此类推就可以算出第n项了。很容易理解，这种思路的时间复杂度是O(n)。实现代码如下：

> java

```
public class Solution {
    public int Fibonacci(int n) {
        if(n==0)
            return 0;
        if(n==1)
            return 1;
        int num1 = 0;
        int num2 = 1;
        int fibN=0;
        for(int i=2;i<=n;++i){
            fibN=num1+num2;
            num1=num2;
            num2=fibN;
```

```
        }
        return fibN;
    }
}
```

4.2 跳台阶

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级台阶总共有多少种跳法。

我们把n级台阶的跳法看成是n的函数，记为f(n)。当n>2时，第一次跳的时候就有两种不同的选择：一是第一次只跳1级，此时跳法数目等于后面剩下的n-1级台阶的跳法数目，即为f(n-1)；另外一种选择是第一次跳2级，此时跳法数目等于后面剩下的n-2级台阶的跳法数目，即为f(n-2)，因此n级台阶的不同跳法的总数f(n)=f(n-1)+f(n-2)。分析到这里，我们不难看出这实际上是斐波那契数列了。
代码如下：

> java

```
public class Solution {
    public int JumpFloor(int target) {
        if(target == 0)
            return 0;
        if(target == 1)
            return 1;
        if(target == 2)
            return 2;
        int num1 = 0;
        int num2 = 1;
        int jump = 0;
        for(int i=0;i<target;i++){
            jump = num1+num2;
            num1=num2;
            num2=jump;
        }
        return jump;

    }
}
```

4.3 变态跳台阶

一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的

台阶总共有多少种跳法。

因为n级台阶，第一步有n种跳法：跳1级、跳2级、到跳n级。跳1级，剩下n-1级，则剩下跳法是f(n-1)，跳2级，剩下n-2级，则剩下跳法是f(n-2)。所以f(n)=f(n-1)+f(n-2)+...+f(1)，因为f(n-1)=f(n-2)+f(n-3)+...+f(1)，所以f(n)=2*f(n-1)

> java

```
public class Solution {
    public int JumpFloorII(int target) {
        if(target==0)
            return 0;
        if(target==1)
            return 1;
        else{
            return 2*JumpFloorII(target-1);
        }
    }
}
```

4.4 矩形覆盖

我们可以用$2*1$的小矩形横着或者竖着去覆盖更大的矩形。请问用n个$2*1$的小矩形无重叠地覆盖一个$2*n$的大矩形，总共有多少种方法?

把$2*8$的覆盖方法记为f(8)。用一个$1*2$小矩形去覆盖大矩形的最左边有两个选择。竖着放或者横着放。当竖着放时，右边剩下$2*7$的区域，记为f(7)。横着放时，当$1*2$的小矩阵横着放在左上角的时候，左下角必须横着放一个$1*2$的小矩阵，剩下$2*6$，记为f(6)，因此f(8)=f(7)+f(6)。此时可以看出，仍然是斐波那契数列。

代码如下:

> java

```
public class Solution {
    public int RectCover(int target) {
        if(target==0)
            return 0;
        if(target==1)
            return 1;
        int num1=0;
        int num2=1;
        int cover =0;
```

```
        for(int i=0;i<target;i++){
            cover = num1+num2;
            num1=num2;
            num2=cover;
        }
        return cover;
    }
}
```

五、调整数组顺序使奇数位于偶数前面（剑14）

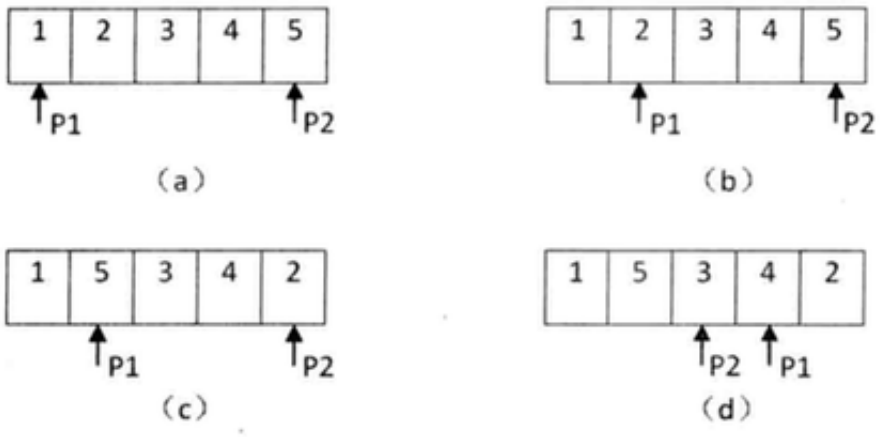


图 3.4　调整数组{1, 2, 3, 4, 5} 使得奇数位于偶数前面的过程

书上的方法类似于快排，但快排是不稳定的，即其相对位置会发生变化。

> java

```
public class Solution {
    public void reOrderArray(int [] array) {
        int length = array.length;
        if(array==null||length==0)
            return;
        int left = 0;
        int right = length-1;
        while(left<right){
            while(left<right&&array[left]%2==1){
                left++;
            }
            while(left<right&&array[right]%2==0){
                right--;
            }
            int temp =array[right];
            array[right]=array[left];
```

```
            array[left]=temp;
        }
    }
}
```

这里要保证奇数和奇数，偶数和偶数之间的相对位置不变。可以使用插入排序的思想

> java

```
public class Solution {
    public void reOrderArray(int [] array) {
        int length = array.length;
        if(array==null||length==0)
            return;
        for(int i=1;i<length;i++){
            if(array[i]%2==1){
                int curr = array[i];
                int j=i-1;
                while(j>=0&&array[j]%2==0){
                    array[j+1]=array[j];
                    j--;
                }
                array[j+1]=curr;
            }
        }
    }
}
```

六、顺时针打印矩阵（剑20）

输入一个矩阵，按照从外向里以顺时针依次打印出每一个数字。例如，输入如下矩阵：

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

则依次打印出数字 1、2、3、4、8、12、16、15、14、13、9、5、6、7、11、10。

> java

```java
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> printMatrix(int [][] matrix) {
        int row = matrix.length;
        int col = matrix[0].length;
        ArrayList<Integer> result = new ArrayList<Integer> ();
        // 输入的二维数组非法，返回空的数组
        if(row==0&&col==0)return result;
        // 定义四个关键变量，表示左上和右下的打印范围
        int left =0,top=0,right=col-1,bottom=row-1;
        while(left<=right&&top<=bottom){
             // left to right
            for(int i=left;i<=right;i++){result.add(matrix[top][i]);}
            // top to bottom
            for(int i=top+1;i<=bottom;i++){result.add(matrix[i][right]);}
            // right to left
            if(top!=bottom){
                for(int i=right-1;i>=left;i--){result.add(matrix[bottom][i]);}}
            // bottom to top
            if(left!=right){
                for(int i=bottom-1;i>=top+1;i--){result.add(matrix[i][left]);}}
            left++;right--;top++;bottom--;
        }
        return result;

    }

}
```

七、包含min函数的栈（剑21）

定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的min函数。在该栈中，调用min、push及pop的时间复杂度都是O(1)。

可以利用一个辅助栈来存放最小值

表 4.1　栈内压入 3、4、2、1 之后接连两次弹出栈顶数字再压入 0 时，数据栈、辅助栈和最小值的状态

步骤	操作	数据栈	辅助栈	最小值
1	压入 3	3	3	3
2	压入 4	3, 4	3, 3	3
3	压入 2	3, 4, 2	3, 3, 2	2
4	压入 1	3, 4, 2, 1	3, 3, 2, 1	1
5	弹出	3, 4, 2	3, 3, 2	2
6	弹出	3, 4	3, 3	3
7	压入 0	3, 4, 0	3, 3, 0	0

每入栈一次，就与辅助栈顶比较大小，如果小就入栈，如果大就入栈当前的辅助栈顶 。
当出栈时，辅助栈也要出栈
这种做法可以保证辅助栈顶一定都是最小元素。

```
import java.util.Stack;

public class Solution {
    Stack<Integer> data = new Stack<Integer>();
    Stack<Integer> min = new Stack<Integer>();

    public void push(int node) {
        data.push(node);
        if(min.empty()){min.push(data.peek());}
        else if(data.peek()<min.peek()){min.push(data.peek());}
        else min.push(min.peek());
        }
    public void pop() {
        data.pop();
        min.pop();
    }

    public int top() {
        return data.peek();

    }
    public int min() {
        return min.peek();
```

```
    }
}
```

八、栈的压入、弹出序列（剑22）

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，序列4，5，3，2，1是该压栈序列对应的一个弹出序列，但4，3，5，1，2就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）

借用一个辅助的栈，遍历压栈顺序，先讲第一个放入栈中，这里是1，然后判断栈顶元素是不是出栈顺序的第一个元素，这里是4，很显然1≠4，所以我们继续压栈，直到相等以后开始出栈，出栈一个元素，则将出栈顺序向后移动一位，直到不相等，这样循环等压栈顺序遍历完成，如果辅助栈还不为空，说明弹出序列不是该栈的弹出顺序。
举例：
入栈1,2,3,4,5
出栈4,5,3,2,1
首先1入辅助栈，此时栈顶1≠4，继续入栈2
此时栈顶2≠4，继续入栈3
此时栈顶3≠4，继续入栈4
此时栈顶4＝4，出栈4，弹出序列向后一位，此时为5，,辅助栈里面是1,2,3
此时栈顶3≠5，继续入栈5
此时栈顶5=5，出栈5,弹出序列向后一位，此时为3，,辅助栈里面是1,2,3
....
依次执行，最后辅助栈为空。如果不为空说明弹出序列不是该栈的弹出顺序。

> java

```
import java.util.ArrayList;
import java.util.Stack;
public class Solution {
    public boolean IsPopOrder(int [] pushA,int [] popA) {
        if(pushA.length==0||popA.length==0)return false;
        Stack<Integer> S=new Stack<Integer>();
        int popIndex = 0;
        for(int i=0;i<pushA.length;i++){
            S.push(pushA[i]);
            while(!S.empty()&&popA[popIndex]==S.peek()){
                S.pop();
                popIndex++;
            }
```

```
        }
        return S.empty();
    }
}
```

九、数组中出现次数超过一半的数字（剑29）

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。

数组中有一个数字出现的次数超过数组长度的一半，也就是说它出现的次数比其他所有数字出现的次数的和还要多。因此我们可以考虑在遍历数组的时候保存两个值：一个是数组的一个数字，一个是次数。当我们遍历到下一个数字的时候，如果下一个数字和我们之前保存的数字相同，则次数加1；如果不同，则次数减1；如果次数为0，则保存下一个数字，并把次数设为1。

还要判断这个数字是否超过数组长度的一半，如果不存在输出0。

```
public class Solution {
    public int MoreThanHalfNum_Solution(int [] array) {
        if(array==null||array.length==0){
            return 0;
        }

        int result=array[0];
        int count=1;
        for(int i=1;i<array.length;i++){
           if(result==array[i]){
               count++; }
           else if(result!=array[i]){
               count--; }
           if(count==0){
               result=array[i];
               count=1;
           }
        }
        int times=0;
        for(int i=0;i<array.length;i++){
            if(array[i]==result){
                times++;
            }
        }
        if(times*2<=array.length){
            System.out.println(times);
```

```
            return 0;
        }
        else
            return result;

    }
}
```

十、最小的K个数（剑30）

输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4,。

第一种方法，借用partition函数

> java

```
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> GetLeastNumbers_Solution(int[] input, int k) {
        ArrayList<Integer> output = new ArrayList<Integer>();
        int length = input.length;
        if (input == null || length <= 0 || length < k || k<= 0) {
            return output;
        }
        int low = 0;
        int high = length - 1;
        while(low<high) {
            int pivotloc = partition(input,low,high);
            if(pivotloc==k-1)break;
            else if(pivotloc < k - 1) {
                low = pivotloc + 1; //在右边
            }
            else {
                high = pivotloc - 1; //在左边
            }
        }
        for (int i = 0;i < k;i++) {
            output.add(input[i]);
        }
        return output;
    }
   //基准左右分区
    private int partition(int[] input,int low,int high) {
        int pivot = input[low];
```

```
        while(low < high) {
            while(input[high] >= pivot && low < high) high--;
            input[low] = input[high];
            while(input[low] <= pivot && low <high) low++;
            input[high] = input[low];
        }
        input[low] = pivot;
        return low;
    }
}
```

第二种方法

用最大堆保存这k个数，每次只和堆顶比，如果比堆顶小，删除堆顶，新数入堆。时间 $O(NlogK)$ 空间 $O(K)$

> java

```
import java.util.ArrayList;
import java.util.PriorityQueue;
import java.util.Comparator;
public class Solution {
   public ArrayList<Integer> GetLeastNumbers_Solution(int[] input, int k) {
       ArrayList<Integer> result = new ArrayList<Integer>();
       int length = input.length;
       if(k > length || k == 0){
           return result;
       }
        PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(k, new Comparat
or<Integer>() {
            @Override//PriorityQueue默认是小顶堆，实现大顶堆，需要反转默认排序器
            public int compare(Integer o1, Integer o2) {
                return o2.compareTo(o1);
            }
        });
        //遍历数组时将数字加入优先队列（堆），一旦堆的大小大于k就将堆顶元素去除，确保堆的大
小为k。遍历完后堆中的数就是最小的K个数。
       for (int i:input) {
            maxHeap.offer(i);
            if(maxHeap.size()>k)
                maxHeap.poll();
        }
       //输出大顶堆中的数
        for (int integer : maxHeap) {
            result.add(integer);
        }
        return result;
```

```
    }
}
```

十一、连续子数组的最大和（剑31）

输入一个整型数组，数组中有正数也有负数。数组中一个或连续的多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为O(n)

第一种方法

> java

```java
public class Solution {
    public int FindGreatestSumOfSubArray(int[] array) {
        if(array.length==0||array==null)
            return 0;
        int cSum = 0;
        int result = array[0];// result存储最大和，不能初始为0，存在负数
        for(int i=0;i<array.length;i++){
            if(cSum<0){
                cSum=array[i];// 当前和<0，抛弃不要
            }else{
                cSum += array[i];//否则累加上去
            }
            if(cSum>result){
                result = cSum;// 存储当前的最大结果
            }
        }
        return result;
    }
}
```

第二种方法：动态规划

```
F（i）：以array[i]为末尾元素的子数组的和的最大值，子数组的元素的相对位置不变
     F（i）=max（F（i-1）+array[i] ，  array[i]）
     res：所有子数组的和的最大值
     res=max（res，F（i））

     如数组[6, -3, -2, 7, -15, 1, 2, 2]
     初始状态：
         F（0）=6
         res=6
```

```
    i=1:
        F(1) =max(F(0) -3, -3) =max(6-3, 3) =3
        res=max(F(1) , res) =max(3, 6) =6
    i=2:
        F(2) =max(F(1) -2, -2) =max(3-2, -2) =1
        res=max(F(2) , res) =max(1, 6) =6
    i=3:
        F(3) =max(F(2) +7, 7) =max(1+7, 7) =8
        res=max(F(2) , res) =max(8, 6) =8
    i=4:
        F(4) =max(F(3) -15, -15) =max(8-15, -15) =-7
        res=max(F(4) , res) =max(-7, 8) =8
    以此类推
    最终res的值为8
```

> java

```java
public class Solution {
    public int FindGreatestSumOfSubArray(int[] array) {
        int res = array[0];
        int max = array[0];
        for(int i=1;i<array.length;i++){
            max=Math.max(max+array[i],array[i]);
            res = Math.max(max,res);
        }
        return res;
    }
}
```

十二、从1到n整数中1出现的次数（剑32）

输入一个整数n，求1到n这n个整数的十进制表示中1出现的次数。例如输入12，从1到12这些整数中包含1的数字有1、10、11、12，1一共出现了5次。

一、1的数目

编程之美上给出的规律：

1. 如果第i位（自右至左，从1开始标号）上的数字为0，则第i位可能出现1的次数由更高位决定（若没有高位，视高位为0），等于更高位数字X当前位数的权重10^{i-1}。
2. 如果第i位上的数字为1，则第i位上可能出现1的次数不仅受更高位影响，还受低位影响（若没有低位，视低位为0），等于更高位数字X当前位数的权重10^{i-1}+（低位数字+1）。
3. 如果第i位上的数字大于1，则第i位上可能出现1的次数仅由更高位决定（若没有高位，视高

位为0），等于（更高位数字+1）X当前位数的权重10^{i-1}。

二、X的数目
这里的 X∈[1,9]，因为 X=0 不符合下列规律，需要单独计算。
首先要知道以下的规律：

- 从 1 至 10，在它们的个位数中，任意的 X 都出现了 1 次。
- 从 1 至 100，在它们的十位数中，任意的 X 都出现了 10 次。
- 从 1 至 1000，在它们的百位数中，任意的 X 都出现了 100 次。

依此类推，从 1 至 10^i，在它们的左数第二位（右数第 i 位）中，任意的 X 都出现了 10^{i-1} 次。

这个规律很容易验证，这里不再多做说明。

接下来以 n=2593,X=5 为例来解释如何得到数学公式。从 1 至 2593 中，数字 5 总计出现了 813 次，其中有 259 次出现在个位，260 次出现在十位，294 次出现在百位，0 次出现在千位。

现在依次分析这些数据，首先是个位。从 1 至 2590 中，包含了 259 个 10，因此任意的 X 都出现了 259 次。最后剩余的三个数 2591, 2592 和 2593，因为它们最大的个位数字 3 < X，因此不会包含任何 5。（也可以这么看，3<X，则个位上可能出现的X的次数仅由更高位决定，等于更高位数字（259）$\times 10^{1-1} = 259$）。

然后是十位。从 1 至 2500 中，包含了 25 个 100，因此任意的 X 都出现了 25×10=250 次。剩下的数字是从 2501 至 2593，它们最大的十位数字9>X，因此会包含全部10个5。最后总计250 + 10 = 260。（也可以这么看，9>X，则十位上可能出现的X的次数仅由更高位决定，等于更高位数字（$25+1$）$\times 10^{2-1} = 260$）。

接下来是百位。从 1 至 2000 中，包含了 2 个 1000，因此任意的 X 都出现了 2×100=200 次。剩下的数字是从 2001 至 2593，它们最大的百位数字 5 == X，这时情况就略微复杂，它们的百位肯定是包含 5 的，但不会包含全部 100 个。如果把百位是 5 的数字列出来，是从 2500 至 2593，数字的个数与百位和十位数字相关，是 93+1 = 94。最后总计 200 + 94 = 294。（也可以这么看，5==X，则百位上可能出现X的次数不仅受更高位影响，还受低位影响，等于更高位数字（2）$\times 10^{3-1} +$（$93+1$）$= 294$）。

最后是千位。现在已经没有更高位，因此直接看最大的千位数字 2 < X，所以不会包含任何 5。（也可以这么看，2<X，则千位上可能出现的X的次数仅由更高位决定，等于更高位数字（0）$\times 10^{4-1} = 0$）。

到此为止，已经计算出全部数字 5 的出现次数。
总结一下以上的算法，可以看到，当计算右数第 i 位包含的 X 的个数时：

- 取第 i 位左边（高位）的数字，乘以10^{i-1}，得到基础值a 。

- 取第 i 位数字，计算修正值：
  - 如果大于 X，则结果为 $a + 10^{i-1}$ 。
  - 如果小于 X，则结果为 a 。
  - 如果等 X，则取第 i 位右边（低位）数字，设为 b ，最后结果为 a+b+1 。

相应的代码非常简单，效率也非常高，时间复杂度只有 $O(log_{10}n)$。

代码如下：

```
public class Solution {
    public int NumberOf1Between1AndN_Solution(int n) {
        if (n < 1)
            return 0;
        int len = getLenOfNum(n);
        if (len == 1)
            return 1;
        int tmp = (int) Math.pow(10, len - 1);
        int first = n / tmp;
        int firstOneNum = first == 1 ? n % tmp + 1 : tmp;
        int otherOneNUm = first * (len - 1) * (tmp / 10);
        return firstOneNum + otherOneNUm + NumberOf1Between1AndN_Solution(n % tmp);
    }

    private int getLenOfNum(int n) {
        int len = 0;
        while (n != 0) {
            len++;
            n /= 10;
        }
        return len;
    }
}
```

十三、把数组排成最小的数（剑33）

输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。例如输入数组{3，32，321}，则打印出这三个数字能排成的最小数字为321323。

> java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
```

```
public class Solution {
    public String PrintMinNumber(int [] numbers) {
        int n;
        String s="";
        ArrayList<Integer> list=new ArrayList<Integer>();
        n=numbers.length;
        for(int i=0;i<n;i++){
            list.add(numbers[i]);//将数组放入arrayList中
        }
        //实现了Comparator接口的compare方法，将集合元素按照compare方法的规则进行排序
        Collections.sort(list,new Comparator<Integer>(){
            @Override
            public int compare(Integer str1, Integer str2) {
                // TODO Auto-generated method stub
                    String s1=str1+""+str2;
                    String s2=str2+""+str1;

                    return s1.compareTo(s2);
                }
            });

        for(int j:list){
            s+=j;
        }
        return s;
    }
}
```

十四、丑数（剑34）

把只包含因子2、3和5的数称作丑数（Ugly Number）。例如6、8都是丑数，但14不是，因为它包含因子7。 习惯上我们把1当做是第一个丑数。求按从小到大的顺序的第N个丑数。

> java

```
import java.util.*;
public class Solution {
    public int GetUglyNumber_Solution(int index) {
        if(index<7)return index;
        int[] res = new int[index];
        res[0] = 1;
        int t2 = 0, t3 = 0, t5 = 0, i;
        for(i=1;i<index;i++){
            res[i] = min(res[t2]*2,min(res[t3]*3,res[t5]*5));
            if(res[i] == res[t2]*2)t2++;
```

```
            if(res[i] == res[t3]*3)t3++;
            if(res[i] == res[t5]*5)t5++;
        }
        return res[index-1];
    }
    private int min(int a,int b){
        return (a>b)? b:a;
    }
}
```

十五、数组中的逆序对（剑36）

在数组中的两个数字如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。例如在数组{7，5，6，4}中，一共存在5个逆序对，分别是（7，6）、（7，5）、（7，4）、（5，4）和（6，4）。

可以按照归并排序的思路，先把数组分隔成子数组，先统计出子数组内部的逆序对的数目，然后再统计出两个相邻子数组之间的逆序对的数目。在统计逆序对的过程中，还需要对数组进行排序。

```
public class Solution {
    int cnt;

    public int InversePairs(int[] array) {
        cnt = 0;
        if (array != null)
            mergeSortUp2Down(array, 0, array.length - 1);
        return cnt;
    }
    /*
     * 归并排序(从上往下)
     */
    public void mergeSortUp2Down(int[] a, int start, int end) {
        if (start >= end)
            return;
        int mid = (start + end) >> 1;

        mergeSortUp2Down(a, start, mid);
        mergeSortUp2Down(a, mid + 1, end);

        merge(a, start, mid, end);
    }
    /*
     * 将一个数组中的两个相邻有序区间合并成一个
```

```
     */
    public void merge(int[] a, int start, int mid, int end) {
        int[] tmp = new int[end - start + 1];

        int i = start, j = mid + 1, k = 0;
        while (i <= mid && j <= end) {
            if (a[i] <= a[j])
                tmp[k++] = a[i++];
            else {
                tmp[k++] = a[j++];
                cnt += mid - i + 1;  //关键的一步，统计逆序对.....
                cnt%=1000000007;
            }
        }
        while (i <= mid)
            tmp[k++] = a[i++];
        while (j <= end)
            tmp[k++] = a[j++];
        for (k = 0; k < tmp.length; k++)
            a[start + k] = tmp[k];
    }
}
```

十六、数字在排序数组中出现的次数（剑38）

统计一个数字在排序数组中出现的次数。

利用二分查找直接找到第一个K和最后一个K。以下代码使用递归方法找到第一个K，使用循环方法最后一个K。

> java

```
public class Solution {
    public int GetNumberOfK(int [] array , int k) {
        int length = array.length;
        if(length == 0){
            return 0;
        }
        int firstK = getFirstK(array, k, 0, length-1);
        int lastK = getLastK(array, k, 0, length-1);
        if(firstK != -1 && lastK != -1){
             return lastK - firstK + 1;
        }
        return 0;
    }
```

```
    //递归写法
    private int getFirstK(int [] array , int k, int start, int end){
        if(start > end){
            return -1;
        }
        int mid = (start + end) >> 1;
        if(array[mid] > k){
            return getFirstK(array, k, start, mid-1);
        }
        else if (array[mid] < k){
            return getFirstK(array, k, mid+1, end);
        }
        else if(mid-1 >=0 && array[mid-1] == k){
            return getFirstK(array, k, start, mid-1);
        }
        else{
            return mid;
        }
    }
    //循环写法
    private int getLastK(int [] array , int k, int start, int end){
        int length = array.length;
        int mid = (start + end) >> 1;
        while(start <= end){
            if(array[mid] > k){
                end = mid-1;
                }
            else if(array[mid] < k){
                start = mid+1;
                }
            else if(mid+1 <= length-1 && array[mid+1] == k){
                start = mid+1;
                }

            else{
                return mid;
                }
            mid = (start + end) >> 1;
            }
        return -1;
     }
}
```

十七、数组中只出现一次的数字（剑40）

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次

的数字。要求时间复杂度是$O(n)$，空间复杂度是$O(1)$。

首先我们考虑这个问题的一个简单版本：一个数组里除了一个数字之外，其他的数字都出现了两次。请写程序找出这个只出现一次的数字。

这个题目的突破口在哪里？题目为什么要强调有一个数字出现一次，其他的出现两次？我们想到了异或运算的性质：任何一个数字异或它自己都等于0 。也就是说，如果我们从头到尾依次异或数组中的每一个数字，那么最终的结果刚好是那个只出现一次的数字，因为那些出现两次的数字全部在异或中抵消掉了。

有了上面简单问题的解决方案之后，我们回到原始的问题。如果能够把原数组分为两个子数组。在每个子数组中，包含一个只出现一次的数字，而其它数字都出现两次。如果能够这样拆分原数组，按照前面的办法就是分别求出这两个只出现一次的数字了。
我们还是从头到尾依次异或数组中的每一个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果。因为其它数字都出现了两次，在异或中全部抵消掉了。由于这两个数字肯定不一样，那么这个异或结果肯定不为0 ，也就是说在这个结果数字的二进制表示中至少就有一位为1 。我们在结果数字中找到第一个为1 的位的位置，记为第N 位。现在我们以第N 位是不是1 为标准把原数组中的数字分成两个子数组，第一个子数组中每个数字的第N 位都为1 ，而第二个子数组的每个数字的第N 位都为0 。
现在我们已经把原数组分成了两个子数组，每个子数组都包含一个只出现一次的数字，而其它数字都出现了两次。因此到此为止，所有的问题我们都已经解决。

> java

```java
//num1,num2分别为长度为1的数组。传出参数
//将num1[0],num2[0]设置为返回结果
public class Solution {
    public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {
       if(array==null ||array.length<2)
           return ;
       int temp = 0;
       for(int i=0;i<array.length;i++)
           temp ^= array[i];

       int indexOf1 = findFirstBitIs(temp);
       for(int i=0;i<array.length;i++){
           if(isBit(array[i], indexOf1))
               num1[0]^=array[i];
           else
               num2[0]^=array[i];
       }
   }
    //在正数num的二进制表示中找到最右边是1的位
   public int findFirstBitIs(int num){
```

```
        int indexBit = 0;
        while(((num & 1)==0) && (indexBit)<8*4){
            num = num >> 1;
            ++indexBit;
        }
        return indexBit;
    }
    //判断在num的二进制表示中从右边数起的indexBit位是不是1.
    public boolean isBit(int num,int indexBit){
        num = num >> indexBit;
        return (num & 1) == 1;
    }
}
```

十八、 和为s的两个数字（剑41.1）

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

数列满足递增，设两个头尾两个指针i和j，

- 若ai + aj == sum，就是答案（相差越远乘积越小）
- 若ai + aj > sum，aj肯定不是答案之一（前面已得出 i 前面的数已是不可能），j -= 1
- 若ai + aj < sum，ai肯定不是答案之一（前面已得出 j 后面的数已是不可能），i += 1

时间复杂度为O(n)。

```
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> FindNumbersWithSum(int [] array,int sum) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        if(array==null||array.length<2){
            return list;
        }
        int i=0,j=array.length-1;
        while(i<j){
            if(array[i]+array[j]==sum){
                list.add(array[i]);
                list.add(array[j]);
                break;
            }
            else if(array[i]+array[j]>sum){
                j--;
            }
```

```
            else
                i++;
        }
        return list;
    }
}
```

十九、和为s的连续正数序列（剑41.2）

输入一个正数s，打印出所有和为s的连续正数序列（至少含有两个数）。例如输入15，由于1+2+3+4+5=4+5+6=7+8=15，所以结果打印出三个连续序列1~5、4~6和7~8。

考虑用两个数small和big分别表示序列的最小值和最大值。首先把small初始化为1，big初始化为2，如果从small到big的序列和大于s，我们可以从序列中去掉较小的值，也就是增大small的值。如果从small到big的序列和小于s，我们可以增大big，让这个序列包含更多的数字。因为这个序列至少要有两个数字，我们一直增加small到（1+s）/2为止。

> java

```
import java.util.ArrayList;
/*
*初始化small=1，big=2;
*small到big序列和小于sum，big++;大于sum，small++;
*当small增加到(1+sum)/2是停止
*/
public class Solution {
    public ArrayList<ArrayList<Integer>> FindContinuousSequence(int sum) {
        ArrayList<ArrayList<Integer>> lists=new ArrayList<ArrayList<Integer>>();
        if(sum<=1){return lists;}
        int small=1;
        int big=2;
        while(small!=(1+sum)/2){          //当small==(1+sum)/2的时候停止
            int curSum=sumOfList(small,big);
            if(curSum==sum){
                ArrayList<Integer> list=new ArrayList<Integer>();
                for(int i=small;i<=big;i++){
                    list.add(i);
                }
                lists.add(list);
                small++;big++;
            }else if(curSum<sum){
                big++;
            }else{
                small++;
```

```
            }
        }
        return lists;
    }
    public int sumOfList(int head,int leap){        //计算当前序列的和
        int sum=head;
        for(int i=head+1;i<=leap;i++){
            sum+=i;
        }
        return sum;
    }
}
```

二十、扑克牌的顺子（剑44）

从扑克牌中随机抽5张牌，判断是不是顺子，即这5张牌是不是连续的。2~10为数字本身，A为1，J为11，Q为12，K为13，而大小王可以看做是任意数字，这里定为0.

> java

```
import java.util.*;
public class Solution {
    public boolean isContinuous(int [] numbers) {
        int length = numbers.length;
        if(numbers==null||length==0)return false;//特殊情况
        Arrays.sort(numbers);//排序
        //统计数组中0的个数
        int numberOfZero = 0;
        for(int i =0;i<length&&numbers[i]==0;i++){
            ++numberOfZero;
        }

        int numberOfGap = 0;
        int small = numberOfZero;
        int big = small+1;
        while(big<length){
            //含有对子，不可能是顺子
            if(numbers[small]==numbers[big]){
                return false;
            }
            //统计数组中的间隔数目
            numberOfGap += numbers[big]-numbers[small]-1;
            small=big;
            big++;
        }
```

```
        //如果间隔数小于等于零的数量则可以组成顺子，否则不行。
        if(numberOfGap<=numberOfZero){
            return true;
        }else{return false;}
    }
}
```

二十一、求1+2+.....+n（剑46）

求1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

1. 需利用逻辑与的短路特性实现递归终止。
2. 当n==0时，(n>0)&&((sum+=Sum_Solution(n-1))>0)只执行前面的判断，为false，然后直接返回0；
3. 当n>0时，执行sum+=Sum_Solution(n-1)，实现递归计算Sum_Solution(n)。

> java

```
public class Solution {
    public int Sum_Solution(int n) {
        int sum=n;
        boolean ans = (n>0)&&((sum+=Sum_Solution(n-1))>0);
        return sum;

    }
}
```

二十二、数组中重复的数字（剑51）

在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中任意一个重复的数字。 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是重复的数字2或者3。

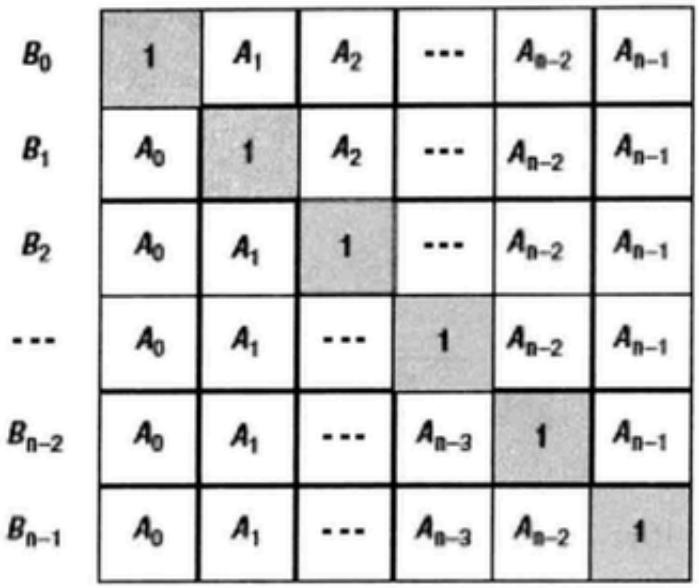
> java

```
public class Solution {
    public boolean duplicate(int numbers[],int length,int [] duplication) {
        if(numbers==null||length==0){return false;}//空指针或空数组
        // 判断数组是否合法,即每个数都在0~n-1之间
        for(int i=0;i<length;i++){
```

```
            if(numbers[i]>length-1||numbers[i]<0){
                return false;
            }
        }
        //若数值与下标不同，则调换位置；
        //比较位置下标为数值(numbers[i])的数值(numbers[numbers[i]])与该数值(numbers[i])是否一致，若一致，则说明有重复数字
        for(int i=0;i<length;i++){
            while(numbers[i]!=i){
                if(numbers[i]==numbers[numbers[i]]){
                    duplication[0] = numbers[i];
                    return true;
                }
                int temp=numbers[i];
                numbers[i]=numbers[temp];
                numbers[temp]=temp;
            }
        }
        return false;
    }
}
```

二十三、构建乘积数组（剑52）

给定一个数组A[0,1,...,n-1],请构建一个数组$B[0, 1, \ldots, n-1]$,其中B中的元素$B[i] = A[0] * A[1] * \ldots * A[i-1] * A[i+1] * \ldots * A[n-1]$。不能使用除法。



B_0	1	A_1	A_2	---	A_{n-2}	A_{n-1}
B_1	A_0	1	A_2	---	A_{n-2}	A_{n-1}
B_2	A_0	A_1	1	---	A_{n-2}	A_{n-1}
---	A_0	A_1	---	1	A_{n-2}	A_{n-1}
B_{n-2}	A_0	A_1	---	A_{n-3}	1	A_{n-1}
B_{n-1}	A_0	A_1	---	A_{n-3}	A_{n-2}	1

图 8.1　把矩阵 B 看成由一个矩形来创建

不妨定义 C[i]=A[0]×A[1]×…×A[i-1]，D[i]=A[i+1]×…×A[n-2]×A[n-1]。C[i]可以用自上而下的顺序计算出来，即 C[i]=C[i-1] ×A[i-1]。类似的，D[i]可以用自下而上的顺序计算出来，即 D[i]=D[i+1] ×A[i+1]。

> java

```java
import java.util.ArrayList;
public class Solution {
    public int[] multiply(int[] A) {
        int length = A.length;
        int[] B = new int[length];
        if(length!=0){
            B[0]=1;
            for(int i=1;i<length;i++){
                B[i]=B[i-1]*A[i-1];
            }
            int temp=1;
            for(int j=length-2;j>=0;j--){
                temp = temp*A[j+1];
                B[j]=temp*B[j];
            }
        }
        return B;
    }
}
```

二十四、 滑动窗口的最大值（剑65）

给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}； 针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下6个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。

表 8.4　找出数组{2, 3, 4, 2, 6, 2, 5, 1}中大小为 3 的滑动窗口的最大值的步骤。在“队列中的下标”一列中，小括号前面的数字表示一个数字在输入数组中的下标。为了方便读者理解，下标对应的在数组中的数字在后面的小括号中标出。

步骤	插入数字	滑动窗口	队列中的下标	最大值
1	2	2	0(2)	N/A
2	3	2, 3	1(3)	N/A
3	4	2, 3, 4	2(4)	4
4	2	3, 4, 2	2(4), 3(2)	4
5	6	4, 2, 6	4(6)	6
6	2	2, 6, 2	4(6), 5(2)	6
7	5	6, 2, 5	4(6), 6(5)	6
8	1	2, 5, 1	6(5), 7(1)	5

> java

```java
import java.util.ArrayList;
import java.util.LinkedList;
public class Solution {

    public ArrayList<Integer> maxInWindows(int [] num, int size)
    {
        ArrayList<Integer> ret = new ArrayList<>();
        if (num == null) {
            return ret;
        }
        if (num.length < size || size < 1) {
            return ret;
        }

        LinkedList<Integer> indexDeque = new LinkedList<>();
        //前size-1个中，前面比num[i]小的，对应下标从下标队列移除；
        for (int i = 0; i < size - 1; i++) {
```

```
            if (!indexDeque.isEmpty() && num[i] > num[indexDeque.getLast()]) {
                indexDeque.removeLast();
            }
            indexDeque.addLast(i);
        }
        //从第size-1个开始；前面比num[i]小的，对应下标从下标队列移除；
        for (int i = size - 1; i < num.length; i++) {
            while(!indexDeque.isEmpty() && num[i] > num[indexDeque.getLast()]) {
                indexDeque.removeLast();
            }
            //把下一个下标加入队列中
            indexDeque.addLast(i);
            //当第一个数字的下标与当前处理的数字的下标之差大于或者等于滑动窗口的大小时，这个数字已经从窗口划出，可以移除了；
            if (i - indexDeque.getFirst() + 1 > size) {
                indexDeque.removeFirst();
            }
            //下标队列的第一个是滑动窗口最大值对应的下标；
            ret.add(num[indexDeque.getFirst()]);
        }
        return ret;
    }
}
```

二十五、矩阵中的路径（剑66）

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。 例如下面的矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

a b c e
s f c s
a d e e

> java

```
public class Solution {
  public int movingCount(int threshold, int rows, int cols)
    {
        boolean[] visited=new boolean[rows*cols];
        return movingCountCore(threshold, rows, cols, 0,0,visited);
```

```
    }
    private int movingCountCore(int threshold, int rows, int cols,
            int row,int col,boolean[] visited) {
        if(row<0||row>=rows||col<0||col>=cols) return 0;
        int i=row*cols+col;
        if(visited[i]||!checkSum(threshold,row,col)) return 0;
        visited[i]=true;
        return 1+movingCountCore(threshold, rows, cols,row,col+1,visited)
                +movingCountCore(threshold, rows, cols,row,col-1,visited)
                +movingCountCore(threshold, rows, cols,row+1,col,visited)
                +movingCountCore(threshold, rows, cols,row-1,col,visited);
    }
    private boolean checkSum(int threshold, int row, int col) {
        int sum=0;
        while(row!=0){
            sum+=row%10;
            row=row/10;
        }
        while(col!=0){
            sum+=col%10;
            col=col/10;
        }
        if(sum>threshold) return false;
        return true;
    }
}
```

二十六、寻找某个值的区间（leetcode 34 Search for a Range）

主要考查二分查找，如果读者对二分查找不是很熟悉，这里推荐一篇博客：你真的会二分查找吗？上面讲得非常详细。

这题要求在一个排好序可能有重复元素的数组里面找到包含某个值的区间范围。要求使用O(log n)的时间，所以我们采用两次二分查找。

主要实现两个方法：

- searchRightIndex：查找并返回target出现在nums数组最右边的index。注意一点，如果target比数组最小值还小，那么返回-1
- searchLeftIndex：查找并返回target出现在nums数组最左边的index。如果target比数组最大值还大，那么返回nums.length

```
public class Solution {
    public int[] searchRange(int[] nums, int target) {
    int[] result = {-1, -1};
    int index = searchRightIndex(nums, 0, nums.length - 1, target);
    if (index < 0 || nums[index] != target)
        return result;
    result[0] = searchLeftIndex(nums, 0, index, target);
    result[1] = index;
    return result;
    }
    //查找并返回target出现在nums数组最右边的index
    public int searchRightIndex(int[] nums, int left, int right, int target) {
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] > target) right = mid - 1;
            else left = mid + 1;
        }
        return right;
    }
    //查找并返回target出现在nums数组最左边的index
    public int searchLeftIndex(int[] nums, int left, int right, int target) {
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] < target) left = mid + 1;
            else right = mid - 1;
        }
        return left;
    }
}
```

二十七、第K个数的问题

27.1 无序数组寻找第K大的数

这题是一道很好的面试题目，首先题目短小，很快就能说清题意而且有很多种解法。从简单到复杂的解法都有，梯度均匀。解决它不需要预先知道特殊领域知识。

这题有很多思路：

1. 按从大到小全排序，然后取第k个元素，时间复杂度O(nlogn)，空间复杂度O(1)
2. 利用堆进行部分排序。维护一个大根堆，将数组元素全部压入堆，然后弹出k次，第k个就是答案。时间复杂度$O(klogn)$，空间复杂度$O(n)$
3. 选择排序，第k次选择后即可得到第k大的数，时间复杂度O(nk)，空间复杂度O(1)

以上三种方法时间复杂度太高。下面介绍两种更好的方法：

> 第一种

利用快速排序中的partition思想，从数组中随机选择一个基准pivot，把数组划分为左右两部分，左边部分元素小于pivot，右边部分元素大于或等于pivot。

```
public class Solution {
    public static int findKthLargest(int[] nums, int k) {
        k = nums.length-k;//找前K大的
        int low = 0;
        int high = nums.length-1;
        while(low < high){
            int pivotloc = partition(nums,low,high);
            if(pivotloc==k)
                break;
            else if(k<pivotloc){//左边
                high = pivotloc-1;
            }else if(k>pivotloc){
                low = pivotloc+1;//在右边
            }
        }
        return nums[k];
    }
    public static int partition(int[] nums, int low, int high){
    int pivot = nums[low];
    while(low<high){
        while(low<high && nums[high]>=pivot) high--;
        nums[low]=nums[high];
        while(low<high && nums[low]<=pivot) low++;
        nums[high] =nums[low];
        }
    nums[low]=pivot;
    return low;
    }
}
```

> 第二种方法：

遍历数组时将数字加入优先队列（堆），一旦堆的大小大于k就将堆顶元素去除，确保堆的大小为k。遍历完后堆顶就是返回值。

```
public class Solution {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> p = new PriorityQueue<Integer>();
```

```
        for(int i = 0 ; i < nums.length; i++){
            p.add(nums[i]);
            if(p.size()>k) p.poll();
        }
        return p.poll();
    }
}
```

27.2 两个有序数组的第K大的数

两个有序数组，寻找归并排序后数组的中位数/第 k 大数字（与二分有关）；

27.3 N个有序数组求最大的K个数

27.4 求两个有序数组的中位数

面经中出现过的数组题

已整理

- 一、二维数组，每行递增，每列递增,实现查找；二维数组，每行递增，每列递增，求第 k 大的数；
- 三、旋转数组的查找；有序数组 从中间某点隔开，右边的放到左边，然后问在这个数组中怎么进行二分查找。讲了思路后手写代码；一维有序数组，经过循环位移后，最小的数出现在数列中间，如果原数组严格递增，如何找这个最小数；如果原数组严格递增或递减，如何找这个最小数；如果原数组非严格递增或递减，如何找这个最小数；
- 五、调整数组顺序使奇数位于偶数前面；让一个数组中的所有奇数在前，偶数在后。
- 六、顺时针打印矩阵
- 九、数组中超过一半的数字；找出数组中出现次数超过一半的数，现在有一个数组，已知一个数出现的次数超过了一半，请用O(n)的复杂度的算法找出这个数我说了一个最简答的，直接遍历数组，用map存储《数，出现的次数》这个键-值对，然后找出超过一半的即可。继续优化，，，，没答上来
- 十、最小的K个数；数列中找第 k 大的数字（与快排或堆排序有关）；编程题最少时间复杂度求数组中第k大的数，
- 十二、从1到n整数中1出现的次数；给你一个数N，问1-N这N个整数里面，每个位上一共出现多少次数字1. 这个是编程之美上的原题。当时看的时候觉得好复杂，最后我也没写出来，当然，面试的技巧就是，无论这道题你会还是不会，尽量把你的思考过程说出来，一方面防止冷场，另一方面可以让他知道你的思考过程。最后面试官让我不用最优的，我就写了个最

笨的。

- 十五、数组中逆序对计算，打印逆序对儿，如输入数组[1,4,2,5,3]，输出(4,2),(4,3),(5,3)；统计数列中的逆序对（归并排序有关）；
- 十八、和为S的两个数字VS和为s的连续正数序列；有序数组寻找和为某数的一对数字；寻找和为定值的多个数；寻找和为定值的两个数
- 二十四、滑动窗口的最大值；coding：数组滑动窗口得到最大的窗口，O（n）复杂度、扩展：这个滑动窗口中必须有m个不同才可以，目标不变，如果改，并分析时间复杂度；
- 二十六、寻找某个值的区间（leetcode 34 Search for a Range）

未整理

数组中后面的数减前面的数的差的最大值，要求时间、空间复杂度尽可能底
多个有序数组的归并
多个有序数组求交集
两个有序数组求差集
两个数组，求差集
两个集合如何求并集，交集；
- leetcode 89 Gray Code
leetcode 42 trapping rain water 墙里能装多少水

有一个数组，让找到两个不重复的连续子序列A,B ，求Max(Sum(A)-Sum(B)。 3分钟解出，10分钟写完代码

LeetCode原题： 有一个集合A包含了一些数，输入N，求元素个数最小的集合B，使得A并B后内的数组合相加能够组成1到N中的所有数

给定一个数组，长度已知为N（中等），求中位数，要求时间复杂度尽可能小：快排+丢弃长度小于N/2的部分

给定数组，要求以最小的时间复杂度求得最大最小值：维护一个小数组，只存放两个数，预定义为min和max，后将剩余N-2个数和它们依次比较，大于max的覆盖max，小于min的覆盖min，其余情形的不更新数组。时间复杂度o(n)。

然后考了个数据结构，给个数组如何建堆http://xfhnever.com/categories/算法与数据结构/page/2/

一维数组，swap 其中的几对数字（每个数字只属于一次 swap 操作），实现查找（与二分有关）；
一个有序数组，其中一个数字发生变异，但不知道变异后会不会影响整体序，如何实现查找；
任意交换其中的两数，发现并恢复；

给定数组，寻找 next big（堆排序有关）；

数组可能是递增、递减、递减后递增、递增后递减四种情况，递增递减都是非严格的，如果有转折点，返回转折点的值，否则返回-1；

给出一个数组，返回数组中满足类似a > b < c > d 这种情况的连续子数组的最大长度。时间复杂度当然是O（n），比较有意思的题目。

同样是一个数组，去掉其中一个数，得到与这个数相邻中比较小的那个数的值（如果是最左侧或者最右侧则返回０）。不断重复上面的操作，直到数组为空，返回累加的最大值。

已知一个数组，有n+2个不同的数，其中n个数出现了偶数次，2个数出现了奇数次，设计算法找出这2个数又只想出一个简单的，用栈，偶数的进出进出，最后在栈中没有了，奇数的进出进，最后会留在栈中。就找到了（这个空间复杂度为O（n）继续优化，，又没想出来。

写递推公式给定整数n和m，问能不能找出整数x，使得x以后的所有整数都可以由整数n和m组合而成

区间查询最大值，要求查询复杂度为O(1)，正解为st表，我敲的线段树，也过了