//面试题1：赋值运算符函数  
/\*\*  
 \* 赋值运算符函数  
 \* 1.对于传入的参数，不应该被修改，使用final修饰；  
 \* 2.如果两个对象相同或值相等，不进行操作，直接返回；  
 \* 3.返回值最好为this，这样可以使赋值链接起来。  
 \* 一个缺点：此赋值从左到右进行，a=b=c等价于a=c，b不会被赋值；  
 \* 而如果是String的=运算，a,b都会被赋成c的值。  
 \*/  
public class P25\_AssignmentOperator {  
    public static class MyString{  
        private String data;  
        public MyString(String data) {  
            this.data = data;  
        }  
        public MyString assign(final MyString another){  
            if(this==another || this.data.equals(another.data))  
                return this;  
            else{  
                this.data = another.data;  
                return this;  
            }  
        }  
        @Override  
        public String toString() {  
            return "MyString{" +  
                    "data='" + data + '\'' +  
                    '}';  
        }  
    }  
    public static void main(String[] args) {  
        MyString s1 = new MyString("a");  
        MyString s2 = new MyString("b");  
        MyString s3 = new MyString("c");  
        System.out.println(s1.assign(s2).assign(s3));  
        System.out.println("s1:" + s1);  
        System.out.println("s2:" + s2);  
        System.out.println("s3:" + s3);  
    }  
}  
//面试题2：实现Singleton模式  
/\*\*  
 \* Created by ryder on 2017/6/7.  
 \* 单例模式  
 \* 定义：指实现了特殊模式的类，该类仅能被实例化一次，产生唯一的一个对象  
 \* 应用举例：windows的任务管理器，回收站，web应用的配置对象，spring中的bean默认也是单例  
 \* 分类：饿汉式，懒汉式，双检锁，静态内部类，枚举  
 \* 评价指标有：单例（必须），线程安全，延迟加载，防止反序列化产生新对象，防止反射攻击  
 \* 实现方法的选择：一般情况下直接使用饿汉式就好了，要求延迟加载时倾向于用静态内部类，涉及到反序列化创建对象或反射问题最好选择枚举  
 \*/  
public class P32\_Singleton {  
    public static void main(String[] args){  
        //调用方式  
        Singleton1 singleton1 = Singleton1.getInstance();  
        Singleton2 singleton2 = Singleton2.getInstance();  
        Singleton3 singleton3 = Singleton3.getInstance();  
        Singleton4 singleton4 = Singleton4.getInstance();  
        Singleton5 singleton5 = Singleton5.getInstance();  
        Singleton6 singleton6 = Singleton6.getInstance();  
        Singleton7 singleton7 = Singleton7.instance;  
        singleton7.setAttribute("aaa");  
    }  
}

//版本一：饿汉式  
//特点：线程安全；在类初始化执行到静态属性时就分配了资源，有资源浪费问题；  
class Singleton1{  
    //或者将私有静态final成员设为公有成员，可省去getInstance公有函数  
    private static final Singleton1 instance = new Singleton1();  
    private Singleton1(){}  
    public static Singleton1 getInstance(){  
        return instance;  
    }  
}

//版本二：懒汉式(非线程安全)  
//特点：在第一次调用获取实例方法时分配内存，实现了懒加载；非线程安全；  
class Singleton2{  
    private static Singleton2 instance= null;  
    private Singleton2(){}  
    public static Singleton2 getInstance(){  
        if(instance==null){  
            instance = new Singleton2();  
        }  
        return instance;  
    }  
}

//版本三：懒汉式变种（synchronized同步方法，支持多线程）  
//特点：线程安全；synchronized而造成的阻塞致使效率低，而且很多的阻塞都是没必要的。  
class Singleton3{  
    private static Singleton3 instance = null;  
    private Singleton3(){}  
    public static synchronized Singleton3 getInstance(){  
        if(instance == null)  
            instance = new Singleton3();  
        return instance;  
    }  
}

//版本四：懒汉式变种（synchronized同步块，支持多线程）  
//特点：写法不同，但与版本三有一样的问题  
class Singleton4{  
    private static Singleton4 instance = null;  
    private Singleton4(){}  
    public static Singleton4 getInstance(){  
        synchronized(Singleton4.class) {  
            if (instance == null)  
                instance = new Singleton4();  
        }  
        return instance;  
    }  
}

//版本五：双检锁DCL，支持多线程-懒汉式  
//特点：线程安全；多进行一次if判断，加入volatile修饰,优点是只有在第一次实例化时加锁，之后不会加锁，提升了效率，缺点写法复杂  
//不加入volatile，可能出现第一个if判断不为null，但还并未执行构造函数的情况，因为java编译器会进行指令重排;  
//volatile的两大作用:  
//1防止编译器对被修饰变量相关代码进行指令重排；2读写操作都不会调用工作内存而是直接取主存，保证了内存可见性  
//指令重排：  
//instance = new Singleton5()可主要分为三步：1分配内存，2调用构造函数，3instance指向被分配的内存（此时instance不为null了）  
//正常顺序为123，指令重排可能执行顺序为132，会造成已不为null但未执行构造函数的问题  
//内存可见性：  
//如果字段是被volatile修饰的，Java内存模型将在写操作后插入一个写屏障指令，在读操作前插入一个读屏障指令。  
//这意味着：1一旦完成写入，任何访问这个字段的线程将会得到最新的；2在写入前，任何更新过的数据值是可见的，因为内存屏障会把之前的写入值都刷新到缓存。  
//因此volatile可提供一定的线程安全，但不适用于写操作依赖于当前值的情况，如自增，自减  
//简单来说，volatile适合这种场景：一个变量被多个线程共享，线程直接给这个变量赋值。  
//还能在双检锁上进行优化，引入一个局部变量，但个人觉得效率提成并不大，不再赘述。  
//volatile参考：http://blog.csdn.net/qq\_29923439/article/details/51273812  
class Singleton5{  
    private volatile static Singleton5 instance = null;  
    private Singleton5(){}  
    public  static Singleton5 getInstance(){  
        if(instance==null){  
            synchronized (Singleton5.class){  
                if(instance==null)  
                    instance = new Singleton5();  
            }  
        }  
        return instance;  
    }  
}

//版本六：静态内部类，支持多线程-懒汉式  
//特点：利用静态内部类（只有在出现它的引用时才被加载），完成懒加载；final保证线程安全;  
//类的加载顺序：http://blog.csdn.net/u012123160/article/details/53224469  
//final的作用:  
//1. 在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。  
//2. 初次读一个包含final域的对象的引用，与随后读这个final域，这两个操作之间不能重排序。  
//扩展：static变量初始化遵循以下规则:  
//1.静态变量会按照声明的顺序先依次声明并设置为该类型的默认值，但不赋值为初始化的值。  
//2.声明完毕后,再按声明的顺序依次设置为初始化的值，如果没有初始化的值就跳过。  
//static变量初始化参考：http://www.jb51.net/article/86629.htm  
class Singleton6{  
    private Singleton6(){}  
    public static Singleton6 getInstance(){  
        return Singleton6Holder.instance;  
    }  
    private static class Singleton6Holder{  
        public static final Singleton6 instance = new Singleton6();  
    }  
}

//版本七：通过枚举实现  
//一个完美的单例需要做到：单例，懒加载，线程安全，防止反序列化产生新对象，防止反射攻击  
//而枚举的特性保证了以上除了懒加载以外的所有要求，而且实现代码及其简单  
//Enum的单例模式参考：http://www.jianshu.com/p/83f7958b0944  
enum Singleton7{  
    instance;  
    private String attribute;  
    void setAttribute(String attribute){  
        this.attribute = attribute;  
    }  
    String getAttribute(){  
        return this.attribute;  
    }  
}

//面试题3：数组中重复的数字  
public class Solution {  
    // Parameters:  
    //    numbers:     an array of integers  
    //    length:      the length of array numbers  
    //    duplication: (Output) the duplicated number in the array number,length of duplication array is 1,so using duplication[0] = ? in implementation;  
    //                  Here duplication like pointor in C/C++, duplication[0] equal \*duplication in C/C++  
    //    这里要特别注意~返回任意重复的一个，赋值duplication[0]  
    // Return value:       true if the input is valid, and there are some duplications in the array number  
    //                     otherwise false  
    public boolean duplicate(int numbers[],int length,int [] duplication) {  
    boolean[] k=new boolean[length];  
        for(int i=0;i<k.length;i++){  
            if(k[numbers[i]]==true){  
                duplication[0]=numbers[i];  
                return true;  
            }  
           k[numbers[i]] = true;  
              
        }  
        return false;  
    }  
}  
/\*\*  
最简单的方法：我最直接的想法就是构造一个容量为N的辅助数组B，原数组A中每个数对应B中下标，首次命中，B中对应元素+1。如果某次命中时，B中对应的不为0，说明，前边已经有一样数字了，那它就是重复的了。  
举例：A{1,2,3,3,4,5}，刚开始B是{0,0,0,0,0,0}，开始扫描A。  
A[0] = 1  {0,1,0,0,0,0}  
A[1] = 2 {0,1,1,0,0,0}  
A[2] = 3 {0,1,1,1,0,0}  
A[3] = 3 {0,1,1,2,0,0}，到这一步，就已经找到了重复数字。  
A[4] = 4 {0,1,1,2,1,0}  
A[5] = 5 {0,1,1,2,1,1}  
时间复杂度O（n），空间复杂度O（n），算法优点是简单快速，比用set更轻量更快，不打乱原数组顺序。  
如果不能用辅助空间，可以参照剑指。  
\*/  
public class Solution {  
    public boolean duplicate(int numbers[],int length,int [] duplication) {  
        int[] assist = new int [length];  
        for(int i = 0; i < length; i++){  
            if(assist[numbers[i]] == 0){  
                assist[numbers[i]] ++;  
            }else{  
                duplication[0] = numbers[i];  
                return true;  
            }  
        }  
        return false;  
    }  
}

//面试题4：二维数组中的查找  
public class Solution {  
    public boolean Find(int target, int [][] array) {  
       int row=array.length;  
        int col=array[0].length;  
        int i=row-1;  
        int j=0;  
        while(i>=0&&j<col){  
            if(target<array[i][j]) i--;  
            else if(target>array[i][j]) j++;  
            else return true;  
        }  
        return false;  
    }  
}

//面试题5：替换空格  
public class Solution {  
    public String replaceSpace(StringBuffer str) {  
        return str.toString().replaceAll("\\s","%20");  
    }  
}  
public class Solution {  
    public String replaceSpace(StringBuffer str) {  
        int spacenum=0;  
        for(int i=0;i<str.length();i++){  
            if(str.charAt(i)==' ')  
                spacenum++;  
        }  
            int indexold=str.length()-1;  
            int newlength=str.length()+spacenum\*2;  
            int indexnew=newlength-1;  
            str.setLength(newlength);  
            for(;indexold>=0&&indexold<newlength;--indexold){  
                if(str.charAt(indexold)==' '){  
                    str.setCharAt(indexnew--,'0');  
                    str.setCharAt(indexnew--,'2');  
                    str.setCharAt(indexnew--,'%');  
                }else{  
                    str.setCharAt(indexnew--,str.charAt(indexold));  
                }  
            }  
            return str.toString();  
        }  
    }

//面试题6：从尾到头打印链表  
/\*\*  
\*    public class ListNode {  
\*        int val;  
\*        ListNode next = null;  
\*  
\*        ListNode(int val) {  
\*            this.val = val;  
\*        }  
\*    }  
\*  
\*/  
import java.util.ArrayList;  
public class Solution {  
    ArrayList<Integer> arrayList=new ArrayList<Integer>();  
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
        if(listNode!=null){  
            this.printListFromTailToHead(listNode.next);  
            arrayList.add(listNode.val);  
        }  
        return arrayList;  
    }  
}  
//用堆栈的后进后出实现，时间复杂度低  
/\*\*  
\*    public class ListNode {  
\*        int val;  
\*        ListNode next = null;  
\*  
\*        ListNode(int val) {  
\*            this.val = val;  
\*        }  
\*    }  
\*  
\*/  
import java.util.ArrayList;  
import java.util.Stack;  
public class Solution {  
   // ArrayList<Integer> arrayList=new ArrayList<Integer>();  
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
        Stack<Integer> stack=new Stack<Integer>();  
            while(listNode!=null){  
                stack.push(listNode.val);  
                listNode=listNode.next;  
            }  
        ArrayList<Integer> list=new ArrayList<Integer>();  
        while(!stack.isEmpty()){  
            list.add(stack.pop());  
        }  
        return list;  
}  
}

//面试题7:重建二叉树  
public class Solution {  
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {  
        TreeNode root=reConstructBinaryTree(pre,0,pre.length-1,in,0,in.length-1);  
        return root;  
    }  
    //前序遍历{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}  
    private TreeNode reConstructBinaryTree(int [] pre,int startPre,int endPre,int [] in,int startIn,int endIn) {  
           
        if(startPre>endPre||startIn>endIn)  
            return null;  
        TreeNode root=new TreeNode(pre[startPre]);  
           
        for(int i=startIn;i<=endIn;i++)  
            if(in[i]==pre[startPre]){  
                root.left=reConstructBinaryTree(pre,startPre+1,startPre+i-startIn,in,startIn,i-1);  
                root.right=reConstructBinaryTree(pre,i-startIn+startPre+1,endPre,in,i+1,endIn);  
                      break;  
            }  
                   
        return root;  
    }  
}  
//面试题8：二叉树的下一个结点  
/\*  
public class TreeLinkNode {  
    int val;  
    TreeLinkNode left = null;  
    TreeLinkNode right = null;  
    TreeLinkNode next = null;

    TreeLinkNode(int val) {  
        this.val = val;  
    }  
}  
\*/  
public class Solution {  
    public TreeLinkNode GetNext(TreeLinkNode node)  
    {  
        if(node==null) return null;  
        if(node.right!=null) {  
            node=node.right;  
            while(node.left!=null)  
                node=node.left;  
            return node;  
    }  
        while(node.next!=null) {  
            if(node.next.left==node)  
                return node.next;  
            node=node.next;  
        }  
        return null;  
      
    }  
}  
//面试题9:用两个栈实现队列  
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() {  
        if(stack1.empty()&&stack2.empty()){  
            throw new RuntimeException("Queue is empty");  
        }  
        if(stack2.empty()){  
            while(!stack1.empty()){  
                stack2.push(stack1.pop());  
            }  
        }  
        return stack2.pop();  
   
    }  
}  
//面试题10：斐波那契数列  
public class Solution {  
    public int Fibonacci(int n) {  
       int a=1;  
        int b=0;  
        int result=0;  
          
        if(n==0) return 0;  
        if(n==1) return 1;  
        for(int i=2;i<=n;i++){  
            result=a+b;  
            b=a;  
            a=result;  
        }  
        return result;  
    }  
}  
//面试题11：旋转数组的最小数字  
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];  
    }  
}  
//面试题12：矩阵中的路径  
public class Solution {  
   
    public boolean hasPath(char[] matrix,int rows,int cols,char[] str) {  
        int flag[]=new int[matrix.length];  
        for(int i=0;i<rows;i++) {  
            for(int j=0;j<cols;j++) {  
                if(helper(matrix,rows,cols,i,j,str,0,flag))  
                    return true;  
            }  
        }  
        return false;  
    }  
    public boolean helper(char[] matrix,int rows,int cols,int i,int j,char[] str,int k,int[] flag) {  
        int index=i\*cols+j;  
        if(i<0||i>=rows||j<0||j>=cols||matrix[index]!=str[k]||flag[index]==1)  
            return false;  
        if(k==str.length-1) return true;  
        flag[index]=1;  
        if(helper(matrix,rows,cols,i-1,j,str,k+1,flag)  
            ||helper(matrix,rows,cols,i+1,j,str,k+1,flag)  
            ||helper(matrix,rows,cols,i,j-1,str,k+1,flag)  
            ||helper(matrix,rows,cols,i,j+1,str,k+1,flag)){  
                return true;  
            }  
            flag[index]=0;  
            return false;

    }

}  
//面试题13：机器人的运动范围  
public class Solution {  
    public int movingCount(int threshold,int rows,int cols) {  
        int flag[][]=new int[rows][cols];  
        return helper(0,0,rows,cols,flag,threshold);  
          
    }  
    private int helper(int i,int j,int rows,int cols,int [][] flag,int threshold) {  
        if(i<0||i>=rows||j<0||j>=cols||numSum(i)+numSum(j)>threshold||flag[i][j]==1)  
            return 0;  
        flag[i][j]=1;  
        return helper(i-1,j,rows,cols,flag,threshold)  
                +helper(i+1,j,rows,cols,flag,threshold)  
                +helper(i,j-1,rows,cols,flag,threshold)  
                +helper(i,j+1,rows,cols,flag,threshold)  
                +1;  
    }  
    private int numSum(int i) {  
        int sum=0;  
        do {  
            sum+=i%10;  
        }while((i=i/10)>0);  
        return sum;  
    }  
}  
//面试题14：剪绳子  
/\*  
 \* 面试题14：剪绳子  
 \* 题目：给你一根长度为n的绳子，请把绳子剪成m段(m和n都是整数，n>1并且m>1)每段绳子的长度记为k[0],k[1],...,k[m].  
 \* 请问k[0]\*k[1]\*...\*k[m]可能的最大乘积是多少？  
 \* 例如，当绳子的长度为8时，我们把它剪成长度分别为2,3,3的三段，此时得到的最大乘积是18.  
 \*/  
 /\*  
 外层循环从绳子长度为4开始，由题意可知最少剪一刀，所以内层循环从j= 1开始，  
 分别计算f(j)\*f(i-j)的值，并且与当前记录的最大值max进行比较。  
本题是一道典型的动态规划算法题，为了避免递归产生的重复计算，采用了从下而上的计算顺序实现。  
 \*/  
public class Test{  
    public static void main(String[] args) {  
        System.out.println(maxAfterCutting(8));  
    }  
    /\*\*  
     \* 常规的需要O(n2)的时间复杂度和O(n)的空间复杂度的动态规划思路  
     \* 题目的意思是:绳子至少是2米，并且必须最少剪一刀。  
     \*/  
    public static int maxAfterCutting(int length){  
        if(length<2)  
            return 0;  
        if(length==2)  
            return 1;  
        if(length==3)  
            return 2;  
        // 子问题的最优解存储在f数组中，数组中的第i个元素表示把长度为i的绳子剪成若干段后各段长度乘积的最大值。  
        int[] f = new int[length+1];  
        f[0] = 0;  
        f[1] = 1;  
        f[2] = 2;  
        f[3] = 3;  
        int result = 0;  
        for(int i = 4;i<=length;i++){  
            int max = 0;  
            for(int j = 1;j<=i/2;j++){  
                int num = f[j]\*f[i-j];  
                if(max<num)  
                    max = num;  
                f[i] = max;  
            }  
        }  
        result = f[length];  
        return result;  
    }  
}

//面试题15：二进制中1的个数  
public class Solution {  
    public int NumberOf1(int n) {  
        int count=0;  
        while(n!=0){  
            ++count;  
            n=(n-1)&n;  
        }  
        return count;  
    }  
}  
//面试题16：数值的整数次方  
public class Solution {  
   public double Power(double base, int n) {  
        double res=1;  
        double curr=base;  
        int exponent;  
        if(n>0) {  
            exponent=n;  
        }else if(n<0) {  
            if(base==0)  
                throw new RuntimeException("分母不能为0");  
            exponent=-n;  
        }else {  
            return 1;  
        }while(exponent!=0) {  
            if((exponent&1)==1)  
                res\*=curr;  
            curr\*=curr;  
            exponent>>=1;  
        }  
        return n>=0?res:(1/res);  
          
  }  
}  
//面试题17：打印从1到最大的n位数  
public static void main(String[] args) {  
        // TODO Auto-generated method stub  
          
        Main test = new Main();  
        test.printToMax(3);  
        //test.printToMax2(3);  
          
    }  
    /\*\*字符串上模拟加法  
     \*  
     \* @param n  
     \*/  
    public void printToMax(int n){  
        if(n < 0)  
            return;  
        char[] number = new char[n];  
//        初始化          
//        for(int i = 0; i < n; i++)  
//            number[i] = '0';  
        Arrays.fill(number, '0');  
        while(!increment(number)){  
            printNumber(number);  
        }  
        return;  
          
    }  
    public boolean increment(char[] num){  
        boolean isOverflow = false;  
        int size = num.length;  
        int carry = 0;  
        for(int i = size - 1; i >= 0; i--){  
            int temp = num[i] - '0' + carry;  
            if(i == size - 1)  
                temp++;  
            if(temp >= 10){  
                if(i == 0) //最高位溢出  
                    isOverflow = true;  
                else{  
                    temp -= 10;  
                    carry = 1;  
                    num[i] = (char) ('0' + temp);  
                }  
            }else{  
                num[i] = (char)('0' + temp);  
                break;  
            }  
        }  
        return isOverflow;  
    }  
    public void printNumber(char[] num){  
        int size = num.length;  
        int i = 0;  
        while(i < size && num[i] == '0') //i < size在前，否则越界  
            i++;  
        //char[] printNum = new char[size - i];  
        //System.arraycopy(num, i, printNum, 0, size - i);//复制数组  
        if(i == size)//不打印全0  
            return;  
        char[] printNum = Arrays.copyOfRange(num, i, size);//复制数组  
        System.out.println(printNum);  
    }  
    /\*\*字符每一位进行全排列  
     \*  
     \* @param n  
     \*/  
    public void printToMax2(int n){  
        if(n <= 0) return;  
        char[] number = new char[n];  
        Arrays.fill(number, '0');  
        printOrder(number,n,0);  
    }  
    public void printOrder(char[] number, int n, int loc){  
        if(loc == n) return;  
        for(int i = 0; i <= 9; i++){  
            number[loc] = (char)('0' + i);  
            if(loc == n - 1){  
                printNumber(number);  
            }  
            printOrder(number,n,loc + 1);  
        }  
    }  
//面试题18：删除链表的节点  
/\*  
 public class ListNode {  
    int val;  
    ListNode next = null;

    ListNode(int val) {  
        this.val = val;  
    }  
}  
\*/  
public class Solution {  
         public ListNode deleteDuplication(ListNode pHead){  
        if(pHead==null||pHead.next==null) {  
            return pHead;  
        }  
        if(pHead.val==pHead.next.val) {  
            ListNode pNode=pHead.next;  
            while(pNode!=null&&pNode.val==pHead.val) {  
                pNode=pNode.next;  
            }  
            return deleteDuplication(pNode);  
        }else {  
            pHead.next=deleteDuplication(pHead.next);  
            return pHead;  
        }

        }  
}  
//面试题19：正则表达式匹配  
public class Solution {  
     public boolean match(char[] str, char[] pattern) {  
        if (str == null || pattern == null) {  
            return false;  
        }  
        int strIndex = 0;  
        int patternIndex = 0;  
        return matchCore(str, strIndex, pattern, patternIndex);  
    }

    public boolean matchCore(char[] str, int strIndex, char[] pattern, int patternIndex) {  
        if (strIndex == str.length && patternIndex == pattern.length) {  
            return true;  
        }  
        if (strIndex != str.length && patternIndex == pattern.length) {  
            return false;  
        }  
          
        if (patternIndex + 1 < pattern.length && pattern[patternIndex + 1] == '\*') {  
            if ((strIndex != str.length && pattern[patternIndex] == str[strIndex])  
                    || (pattern[patternIndex] == '.' && strIndex != str.length)) {  
                return matchCore(str, strIndex, pattern, patternIndex + 2)  
                        || matchCore(str, strIndex + 1, pattern, patternIndex + 2)  
                        || matchCore(str, strIndex + 1, pattern, patternIndex);  
            } else {  
                return matchCore(str, strIndex, pattern, patternIndex + 2);  
            }  
        }  
        if ((strIndex != str.length && pattern[patternIndex] == str[strIndex])  
                || (pattern[patternIndex] == '.' && strIndex != str.length)) {  
            return matchCore(str, strIndex + 1, pattern, patternIndex + 1);  
        }  
        return false;  
    }  
}  
//面试题20：表示数值的字符串  
//正则表达式解法  
public class Solution {  
    public boolean isNumeric(char[] str) {  
        String string = String.valueOf(str);  
        return string.matches("[\\+\\-]?\\d\*(\\.\\d+)?([eE][\\+\\-]?\\d+)?");  
    }  
}  
/\*  
以下对正则进行解释:  
[\\+\\-]?            -> 正或负符号出现与否  
\\d\*                 -> 整数部分是否出现，如-.34 或 +3.34均符合  
(\\.\\d+)?           -> 如果出现小数点，那么小数点后面必须有数字；  
                        否则一起不出现  
([eE][\\+\\-]?\\d+)? -> 如果存在指数部分，那么e或E肯定出现，+或-可以不出现，  
                        紧接着必须跟着整数；或者整个部分都不出现  
\*/  
   
   
//参见剑指offer  
public class Solution {  
    private int index = 0;  
   
    public boolean isNumeric(char[] str) {  
        if (str.length < 1)  
            return false;  
           
        boolean flag = scanInteger(str);  
           
        if (index < str.length && str[index] == '.') {  
            index++;  
            flag = scanUnsignedInteger(str) || flag;  
        }  
           
        if (index < str.length && (str[index] == 'E' || str[index] == 'e')) {  
            index++;  
            flag = flag && scanInteger(str);  
        }  
           
        return flag && index == str.length;  
           
    }  
       
    private boolean scanInteger(char[] str) {  
        if (index < str.length && (str[index] == '+' || str[index] == '-') )  
            index++;  
        return scanUnsignedInteger(str);  
           
    }  
       
    private boolean scanUnsignedInteger(char[] str) {  
        int start = index;  
        while (index < str.length && str[index] >= '0' && str[index] <= '9')  
            index++;  
        return start < index; //是否存在整数  
    }  
}  
//面试题21:调整数组顺序使奇数位于偶数前面  
public class Solution {  
    /\*  
整体思路：  
首先统计奇数的个数  
然后新建一个等长数组，设置两个指针，奇数指针从0开始，偶数指针从奇数个数的末尾开始 遍历，填数  
\*/  
public void reOrderArray(int [] array) {  
        if(array.length==0||array.length==1) return;  
        int oddCount=0,oddBegin=0;  
        int[] newArray=new int[array.length];  
        for(int i=0;i<array.length;i++){  
            if((array[i]&1)==1) oddCount++;  
        }  
        for(int i=0;i<array.length;i++){  
            if((array[i]&1)==1) newArray[oddBegin++]=array[i];  
            else newArray[oddCount++]=array[i];  
        }  
        for(int i=0;i<array.length;i++){  
            array[i]=newArray[i];  
        }  
    }  
}  
//面试题22：链表中倒数第k个节点  
//代码思路如下：两个指针，先让第一个指针和第二个指针都指向头结点  
//然后再让第一个指正走(k-1)步，到达第k个节点。  
//然后两个指针同时往后移动，当第一个结点到达末尾的时候，第二个结点所在位置就是倒数第k个节点了。。  
/\*  
public class ListNode {  
    int val;  
    ListNode next = null;

    ListNode(int val) {  
        this.val = val;  
    }  
}\*/  
public class Solution {  
    public ListNode FindKthToTail(ListNode head,int k) {  
        if(head==null||k<=0){  
            return null;  
        }  
        ListNode pre=head;  
        ListNode last=head;         
        for(int i=1;i<k;i++){  
            if(pre.next!=null){  
                pre=pre.next;  
            }else{  
                return null;  
            }  
        }  
        while(pre.next!=null){  
            pre = pre.next;  
            last=last.next;  
        }  
        return last;  
    }  
}  
//面试题23：链表中环的入口节点  
/\*  
 public class ListNode {  
    int val;  
    ListNode next = null;

    ListNode(int val) {  
        this.val = val;  
    }  
}  
\*/  
//第一步，找环中相汇点  
//分别用p1，p2指向链表头部，p1每次走一步，p2每次走二步，直到p1==p2找到在环中的相汇点。  
//第二步，找环的入口  
//当p1==p2时，p2所经过节点数为2x,p1所经过节点数为x  
//设环中有n个节点,p2比p1多走一圈有2x=n+x; n=x  
//可以看出p1实际走了一个环的步数，再让p2指向链表头部，p1位置不变，p1,p2每次走一步直到p1==p2; 此时p1指向环的入口。  
public class Solution {  
public ListNode EntryNodeOfLoop(ListNode pHead){  
        if(pHead == null || pHead.next == null)  
            return null;  
        ListNode p1 = pHead;  
        ListNode p2 = pHead;  
        while(p2 != null && p2.next != null ){  
            p1 = p1.next;  
            p2 = p2.next.next;  
            if(p1 == p2){  
                p2 = pHead;  
                while(p1 != p2){  
                    p1 = p1.next;  
                    p2 = p2.next;  
                }  
                if(p1 == p2)  
                    return p1;  
            }  
        }  
        return null;  
    }  
}  
//面试题24：反转链表  
/\*  
public class ListNode {  
    int val;  
    ListNode next = null;

    ListNode(int val) {  
        this.val = val;  
    }  
}\*/  
public class Solution {  
public ListNode ReverseList(ListNode head) {  
    ListNode pre = null;  
    ListNode next = null;  
    while (head != null) {  
        next = head.next;  
        head.next = pre;  
        pre = head;  
        head = next;  
    }  
    return pre;  
}  
}  
//面试题25：合并两个排序的链表  
/\*  
public class ListNode {  
    int val;  
    ListNode next = null;

    ListNode(int val) {  
        this.val = val;  
    }  
}\*/  
public class Solution {  
    public ListNode Merge(ListNode list1,ListNode list2) {  
if(list1 == null){  
            return list2;  
        }  
        if(list2 == null){  
            return list1;  
        }  
        ListNode mergeHead = null;  
        ListNode current = null;       
        while(list1!=null && list2!=null){  
            if(list1.val <= list2.val){  
                if(mergeHead == null){  
                   mergeHead = current = list1;  
                }else{  
                   current.next = list1;  
                   current = current.next;  
                }  
                list1 = list1.next;  
            }else{  
                if(mergeHead == null){  
                   mergeHead = current = list2;  
                }else{  
                   current.next = list2;  
                   current = current.next;  
                }  
                list2 = list2.next;  
            }  
        }  
        if(list1 == null){  
            current.next = list2;  
        }else{  
            current.next = list1;  
        }  
        return mergeHead;     
    }  
}  
//面试题26：树的子结构  
/\*\*  
public class TreeNode {  
    int val = 0;  
    TreeNode left = null;  
    TreeNode right = null;

    public TreeNode(int val) {  
        this.val = val;

    }

}  
\*/  
public class Solution {  
 public static boolean HasSubtree(TreeNode root1, TreeNode root2) {  
        boolean result = false;  
        //当Tree1和Tree2都不为零的时候，才进行比较。否则直接返回false  
        if (root2 != null && root1 != null) {  
            //如果找到了对应Tree2的根节点的点  
            if(root1.val == root2.val){  
                //以这个根节点为为起点判断是否包含Tree2  
                result = doesTree1HaveTree2(root1,root2);  
            }  
            //如果找不到，那么就再去root的左儿子当作起点，去判断时候包含Tree2  
            if (!result) {  
                result = HasSubtree(root1.left,root2);  
            }  
               
            //如果还找不到，那么就再去root的右儿子当作起点，去判断时候包含Tree2  
            if (!result) {  
                result = HasSubtree(root1.right,root2);  
               }  
            }  
            //返回结果  
        return result;  
    }  
   
    public static boolean doesTree1HaveTree2(TreeNode node1, TreeNode node2) {  
        //如果Tree2已经遍历完了都能对应的上，返回true  
        if (node2 == null) {  
            return true;  
        }  
        //如果Tree2还没有遍历完，Tree1却遍历完了。返回false  
        if (node1 == null) {  
            return false;  
        }  
        //如果其中有一个点没有对应上，返回false  
        if (node1.val != node2.val) {    
                return false;  
        }  
           
        //如果根节点对应的上，那么就分别去子节点里面匹配  
        return doesTree1HaveTree2(node1.left,node2.left) && doesTree1HaveTree2(node1.right,node2.right);  
    }  
}  
//面试题27：二叉树的镜像  
/\*\*  
public class TreeNode {  
    int val = 0;  
    TreeNode left = null;  
    TreeNode right = null;

    public TreeNode(int val) {  
        this.val = val;

    }

}  
\*/  
public class Solution {  
public void Mirror(TreeNode root){  
    if(root==null) return;  
    if(root.left==null&&root.right==null) return;  
    TreeNode temp=root.left;  
    root.left=root.right;  
    root.right=temp;  
      
    if(root.left!=null) Mirror(root.left);  
    if(root.right!=null) Mirror(root.right);  
      
}  
}  
//面试题28：对称的二叉树  
/\*  
public class TreeNode {  
    int val = 0;  
    TreeNode left = null;  
    TreeNode right = null;

    public TreeNode(int val) {  
        this.val = val;

    }

}  
\*/  
public class Solution {  
//递归  
boolean isSymmetrical(TreeNode pRoot)  
    {  
        if(pRoot == null) return true;  
        return isSymmetrical(pRoot.left, pRoot.right);  
    }  
    private boolean isSymmetrical(TreeNode left, TreeNode right) {  
        if(left == null && right == null) return true;  
        if(left == null || right == null) return false;  
        return left.val == right.val //为镜像的条件：左右节点值相等  
                && isSymmetrical(left.left, right.right) //2.对称的子树也是镜像  
                && isSymmetrical(left.right, right.left);  
    }  
}  
//面试题29:顺时针打印矩阵  
//主体循环部分才5行。其实是有规律可循的。将每一层的四个边角搞清楚就可以打印出来了  
   
import java.util.ArrayList;  
public class Solution {  
    public ArrayList<Integer> printMatrix(int [][] array) {  
        ArrayList<Integer> result = new ArrayList<Integer> ();  
        if(array.length==0) return result;  
        int n = array.length,m = array[0].length;  
        if(m==0) return result;  
        int layers = (Math.min(n,m)-1)/2+1;//这个是层数  
        for(int i=0;i<layers;i++){  
            for(int k = i;k<m-i;k++) result.add(array[i][k]);//左至右  
            for(int j=i+1;j<n-i;j++) result.add(array[j][m-i-1]);//右上至右下  
            for(int k=m-i-2;(k>=i)&&(n-i-1!=i);k--) result.add(array[n-i-1][k]);//右至左  
            for(int j=n-i-2;(j>i)&&(m-i-1!=i);j--) result.add(array[j][i]);//左下至左上  
        }  
        return result;         
    }  
}

//面试题30：包含min函数的栈  
import java.util.Stack;  
/\*  
思路：用一个栈data保存数据，用另外一个栈min保存依次入栈最小的数  
比如，data中依次入栈，5,  4,  3, 8, 10, 11, 12, 1  
则min依次入栈，5,  4,  3，no,no, no, no, 1  
no代表此次不如栈  
每次入栈的时候，如果入栈的元素比min中的栈顶元素小或等于则入栈，否则不如栈。  
 \*/  
public class Solution {  
    Stack<Integer> data = new Stack<Integer>();  
    Stack<Integer> min = new Stack<Integer>();  
    Integer temp = null;  
    public void push(int node) {  
        if(temp != null){  
            if(node <= temp ){  
                temp = node;  
                min.push(node);  
            }  
            data.push(node);  
        }else{  
            temp = node;  
            data.push(node);  
            min.push(node);  
        }  
    }  
       
    public void pop() {  
        int num = data.pop();  
        int num2 = min.pop();  
        if(num != num2){  
           min.push(num2);  
        }  
    }  
       
    public int top() {  
        int num = data.pop();  
        data.push(num);  
        return num;  
    }  
       
    public int min() {  
        int num = min.pop();  
        min.push(num);  
        return num;  
    }  
}  
//面试题31：栈的压入、弹出序列  
/\*  
【思路】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。  
\*/  
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>();  
        //用于标识弹出序列的位置  
        int popIndex = 0;  
        for(int i = 0; i< pushA.length;i++){  
            s.push(pushA[i]);  
            //如果栈不为空，且栈顶元素等于弹出序列  
            while(!s.empty() &&s.peek() == popA[popIndex]){  
                //出栈  
                s.pop();  
                //弹出序列向后一位  
                popIndex++;  
            }  
        }  
        return s.empty();  
    }  
}  
//面试题32：从上到下打印二叉树  
/\*\*  
思路是用arraylist模拟一个队列来存储相应的TreeNode  
\*/  
import java.util.\*;  
public class Solution {  
    public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {  
        ArrayList<Integer> list = new ArrayList<>();  
        ArrayList<TreeNode> queue = new ArrayList<>();  
        if (root == null) {  
            return list;  
        }  
        queue.add(root);  
        while (queue.size() != 0) {  
            TreeNode temp = queue.remove(0);  
            if (temp.left != null){  
                queue.add(temp.left);  
            }  
            if (temp.right != null) {  
                queue.add(temp.right);  
            }  
            list.add(temp.val);  
        }  
        return list;  
    }  
}  
//面试题33：二叉搜索树的后序遍历序列  
public class Solution {  
    public boolean VerifySquenceOfBST(int [] sequence) {  
        if(sequence.length==0)  
            return false;  
        if(sequence.length==1)  
            return true;  
        return ju(sequence, 0, sequence.length-1);  
           
    }  
       
    public boolean ju(int[] a,int star,int root){  
        if(star>=root)  
            return true;  
        int i = root;  
        //从后面开始找  
        while(i>star&&a[i-1]>a[root])  
            i--;//找到比根小的坐标  
        //从前面开始找 star到i-1应该比根小  
        for(int j = star;j<i-1;j++)  
            if(a[j]>a[root])  
                return false;;  
        return ju(a,star,i-1)&&ju(a, i, root-1);  
    }  
}  
//面试题34：二叉树中和为某一值的路径  
import java.util.\*;  
public class Solution {  
    private ArrayList<ArrayList<Integer>> listAll = new ArrayList<ArrayList<Integer>>();  
    private ArrayList<Integer> list = new ArrayList<Integer>();  
    public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {  
        if(root == null) return listAll;  
        list.add(root.val);  
        target -= root.val;  
        if(target == 0 && root.left == null && root.right == null)  
            listAll.add(new ArrayList<Integer>(list));  
        FindPath(root.left, target);  
        FindPath(root.right, target);  
        list.remove(list.size()-1);  
        return listAll;  
    }  
}  
//面试题35：复杂链表的复制  
/\*  
public class RandomListNode {  
    int label;  
    RandomListNode next = null;  
    RandomListNode random = null;

    RandomListNode(int label) {  
        this.label = label;  
    }  
}  
\*/  
/\*  
\*解题思路：  
\*1、遍历链表，复制每个结点，如复制结点A得到A1，将结点A1插到结点A后面；  
\*2、重新遍历链表，复制老结点的随机指针给新结点，如A1.random = A.random.next;  
\*3、拆分链表，将链表拆分为原链表和复制后的链表  
\*/  
public class Solution {  
    public RandomListNode Clone(RandomListNode pHead) {  
        if(pHead == null) {  
            return null;  
        }  
           
        RandomListNode currentNode = pHead;  
        //1、复制每个结点，如复制结点A得到A1，将结点A1插到结点A后面；  
        while(currentNode != null){  
            RandomListNode cloneNode = new RandomListNode(currentNode.label);  
            RandomListNode nextNode = currentNode.next;  
            currentNode.next = cloneNode;  
            cloneNode.next = nextNode;  
            currentNode = nextNode;  
        }  
           
        currentNode = pHead;  
        //2、重新遍历链表，复制老结点的随机指针给新结点，如A1.random = A.random.next;  
        while(currentNode != null) {  
            currentNode.next.random = currentNode.random==null?null:currentNode.random.next;  
            currentNode = currentNode.next.next;  
        }  
           
        //3、拆分链表，将链表拆分为原链表和复制后的链表  
        currentNode = pHead;  
        RandomListNode pCloneHead = pHead.next;  
        while(currentNode != null) {  
            RandomListNode cloneNode = currentNode.next;  
            currentNode.next = cloneNode.next;  
            cloneNode.next = cloneNode.next==null?null:cloneNode.next.next;  
            currentNode = currentNode.next;  
        }  
           
        return pCloneHead;  
    }  
}  
//面试题36:二叉搜索树与双向链表  
//直接用中序遍历  
public class Solution {  
    TreeNode head = null;  
    TreeNode realHead = null;  
    public TreeNode Convert(TreeNode pRootOfTree) {  
        ConvertSub(pRootOfTree);  
        return realHead;  
    }  
       
    private void ConvertSub(TreeNode pRootOfTree) {  
        if(pRootOfTree==null) return;  
        ConvertSub(pRootOfTree.left);  
        if (head == null) {  
            head = pRootOfTree;  
            realHead = pRootOfTree;  
        } else {  
            head.right = pRootOfTree;  
            pRootOfTree.left = head;  
            head = pRootOfTree;  
        }  
        ConvertSub(pRootOfTree.right);  
    }  
}  
//面试题37:序列化二叉树  
/\*  
public class TreeNode {  
    int val = 0;  
    TreeNode left = null;  
    TreeNode right = null;  
   
    public TreeNode(int val) {  
        this.val = val;  
   
    }  
   
}  
\*/  
public class Solution {  
    public int index = -1;  
    String Serialize(TreeNode root) {  
        StringBuffer sb = new StringBuffer();  
        if(root == null){  
            sb.append("#,");  
            return sb.toString();  
        }  
        sb.append(root.val + ",");  
        sb.append(Serialize(root.left));  
        sb.append(Serialize(root.right));  
        return sb.toString();  
  }  
    TreeNode Deserialize(String str) {  
        index++;  
       int len = str.length();  
        if(index >= len){  
            return null;  
        }  
        String[] strr = str.split(",");  
        TreeNode node = null;  
        if(!strr[index].equals("#")){  
            node = new TreeNode(Integer.valueOf(strr[index]));  
            node.left = Deserialize(str);  
            node.right = Deserialize(str);  
        }  
           
        return node;  
  }  
}  
//面试题38:字符串的排列  
import java.util.List;  
import java.util.Collections;  
import java.util.ArrayList;  
   
public class Solution {  
    public static void main(String[] args) {  
        Solution p = new Solution();  
        System.out.println(p.Permutation("abc").toString());  
    }  
   
    public ArrayList<String> Permutation(String str) {  
        List<String> res = new ArrayList<>();  
        if (str != null && str.length() > 0) {  
            PermutationHelper(str.toCharArray(), 0, res);  
            Collections.sort(res);  
        }  
        return (ArrayList)res;  
    }  
   
    public void PermutationHelper(char[] cs, int i, List<String> list) {  
        if (i == cs.length - 1) {  
            String val = String.valueOf(cs);  
            if (!list.contains(val))  
                list.add(val);  
        } else {  
            for (int j = i; j < cs.length; j++) {  
                swap(cs, i, j);  
                PermutationHelper(cs, i+1, list);  
                swap(cs, i, j);  
            }  
        }  
    }  
   
    public void swap(char[] cs, int i, int j) {  
        char temp = cs[i];  
        cs[i] = cs[j];  
        cs[j] = temp;  
    }  
}  
//面试题39:数组中出现次数超过一半的数字  
public class Solution {  
    public int MoreThanHalfNum\_Solution(int [] array) {  
        HashMap<Integer,Integer> map = new HashMap<Integer,Integer>();  
           
        for(int i=0;i<array.length;i++){  
               
            if(!map.containsKey(array[i])){  
               map.put(array[i],1);  
            }else{  
                int count = map.get(array[i]);  
                map.put(array[i],++count);  
            }  
        }  
        Iterator iter = map.entrySet().iterator();  
        while(iter.hasNext()){  
            Map.Entry entry = (Map.Entry)iter.next();  
            Integer key =(Integer)entry.getKey();  
            Integer val = (Integer)entry.getValue();  
            if(val>array.length/2){  
                return key;  
            }  
        }  
        return 0;  
    }  
//面试题40:最小的k个数  
//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。  
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 Comparator<Integer>() {  
   
            @Override  
            public int compare(Integer o1, Integer o2) {  
                return o2.compareTo(o1);  
            }  
        });  
        for (int i = 0; i < length; i++) {  
            if (maxHeap.size() != k) {  
                maxHeap.offer(input[i]);  
            } else if (maxHeap.peek() > input[i]) {  
                Integer temp = maxHeap.poll();  
                temp = null;  
                maxHeap.offer(input[i]);  
            }  
        }  
        for (Integer integer : maxHeap) {  
            result.add(integer);  
        }  
        return result;  
    }  
}

//面试题41:数据流中的中位数  
import java.util.PriorityQueue;  
import java.util.Comparator;  
public class Solution {  
    /\*\*\*\*\*\*\*\*\*\*\*方式一、用两个优先队列来模拟两个堆---主要思路\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
    1.先用java集合PriorityQueue来设置一个小顶堆和大顶堆，大顶堆需要先重写一下里面的比较器  
       
    2.主要的思想是：因为要求的是中位数，那么这两个堆，大顶堆用来存较小的数，从大到小排列；  
                                              小顶堆存较大的数，从小到大的顺序排序，  
                                              显然中位数就是大顶堆的根节点与小顶堆的根节点和的平均数。  
                                                 
    保证：小顶堆中的元素都大于等于大顶堆中的元素，所以每次塞值，并不是直接塞进去，而是从另一个堆中poll出一个最大（最小）的塞值  
       
    3.当数目为偶数的时候，将这个值插入大顶堆中，再将大顶堆中根节点（即最大值）插入到小顶堆中；  
      当数目为奇数的时候，将这个值插入小顶堆中，再讲小顶堆中根节点（即最小值）插入到大顶堆中；  
         
      这样就可以保证，每次插入新值时，都保证小顶堆中值大于大顶堆中的值，并且都是有序的。  
       
    4.由于第一个数是插入到小顶堆中的，所以在最后取中位数的时候，若是奇数，就从小顶堆中取即可。  
    这样，当count为奇数的时候，中位数就是小顶堆的根节点；当count为偶数的时候，中位数为大顶堆和小顶堆两个根节点之和的平均数  
       
    5.例如，传入的数据为：[5,2,3,4,1,6,7,0,8],那么按照要求，输出是"5.00 3.50 3.00 3.50 3.00 3.50 4.00 3.50 4.00 "  
        a.那么，第一个数为5，count=0,那么存到小顶堆中，  
            步骤是：先存到大顶堆；然后弹出大顶堆root，就是最大值给小顶堆，第一次执行完，就是小顶堆为5，count+1=1；      
            此时若要输出中位数，那么就是5.0，因为直接返回的是小顶堆最小值(第一次塞入到小顶堆中，是从大顶堆中找到最大的给他的)  
               
               
               
        b.继续传入一个数为2，那么先存到小顶堆中，将小顶堆最小值弹出给大顶堆，即2，那么这次执行完，小顶堆为5，大顶堆为2，count+1=2  
            此时若要输出中位数，因为是偶数，那么取两个头的平均值，即(5+2)/2=3.5(第二次塞入到大顶堆中，是从小顶堆中找到最小的给他的)  
               
        c.继续传入一个数为3，那么此时count为偶数，那么执行第一个if，先存到大顶堆中，大顶堆弹出最大值，那么3>2，就是弹出3  
            3存到小顶堆中，那么此时小顶堆为3,5，大顶堆为2，count+1=3(第三次塞入到小顶堆中，是从大顶堆中找到最大的给他的)  
            此时若要输出中位数，因为是奇数，那么取小顶堆的最小值，即3.0  
               
        d.继续传入一个数为4，先存到小顶堆中，小顶堆此时为3,4，5,弹出最小值为3，给大顶堆  
            此时大顶堆为3,2,小顶堆为4,5，(第四次塞入到小顶堆中，是从大顶堆中找到最大的给他的)  
            此时若要输出中位数，因为是偶数，那么取两个头的平均值,即(3+4)/2=3.5  
               
        e.依次类推。。。  
       
       
    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
       
    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*方式二、ArrayList\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
    用ArrayList来存输入的数据流，然后每次用Collections.sort(list)来保证数据流有序，然后再取中位数  
    思想非常简单，但是每次都要进行排序，时间复杂度可想而知  
    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
       
    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*方式三、插入排序，插入到对应的位置\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
    LinkedList<Integer> data = new LinkedList<Integer>();  
    public void Insert(Integer num) {  
        for (int i = data.size() - 1; i >= 0 ; i--) {  
            if (num >= data.get(i)){  
                data.add(i+1,num);  
                return;  
            }  
        }  
        data.addFirst(num);  
    }  
    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
       
    int count = 0;  
    private PriorityQueue<Integer> minHeap = new PriorityQueue<>();//默认是小根堆  
    private PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(15, new Comparator<Integer>() {  
        @Override  
        public int compare(Integer o1, Integer o2) {  
            return o2 - o1;  
        }  
    });  
    public void Insert(Integer num) {  
        if(count%2 == 0){  
            //数目为偶数时，插入到小根堆中  
            maxHeap.offer(num);  
            int filteredMaxNum = maxHeap.poll();  
            minHeap.offer(filteredMaxNum);  
        }else{  
            //数目为奇数时，插入到大根堆中  
            minHeap.offer(num);  
            int filteredMinNum = minHeap.poll();  
            maxHeap.offer(filteredMinNum);  
        }  
        count++;  
    }  
   
    public Double GetMedian() {  
        if (count %2 == 0) {  
            return new Double((minHeap.peek() + maxHeap.peek())) / 2;  
        } else {  
            return new Double(minHeap.peek());  
        }  
    }  
   
}  
//面试题42：连续子数组的最大和  
public class Solution {  
    public int FindGreatestSumOfSubArray(int[] array) {  
         //记录当前所有子数组的和的最大值  
        int res=array[0];  
        //包含array[i]的连续数组最大值  
        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;  
    }  
}  
//面试题43:1-n整数中1出现的次数(整数中1出现的次数)  
/\*  
设N = abcde ,其中abcde分别为十进制中各位上的数字。  
如果要计算百位上1出现的次数，它要受到3方面的影响：百位上的数字，百位以下（低位）的数字，百位以上（高位）的数字。  
① 如果百位上数字为0，百位上可能出现1的次数由更高位决定。比如：12013，则可以知道百位出现1的情况可能是：100~199，1100~1199,2100~2199，，...，11100~11199，一共1200个。可以看出是由更高位数字（12）决定，并且等于更高位数字（12）乘以 当前位数（100）。  
② 如果百位上数字为1，百位上可能出现1的次数不仅受更高位影响还受低位影响。比如：12113，则可以知道百位受高位影响出现的情况是：100~199，1100~1199,2100~2199，，....，11100~11199，一共1200个。和上面情况一样，并且等于更高位数字（12）乘以 当前位数（100）。但同时它还受低位影响，百位出现1的情况是：12100~12113,一共114个，等于低位数字（113）+1。  
③ 如果百位上数字大于1（2~9），则百位上出现1的情况仅由更高位决定，比如12213，则百位出现1的情况是：100~199,1100~1199，2100~2199，...，11100~11199,12100~12199,一共有1300个，并且等于更高位数字+1（12+1）乘以当前位数（100）。  
\*/  
public class Solution {  
    public int NumberOf1Between1AndN\_Solution(int n) {  
        int count = 0;//1的个数  
        int i = 1;//当前位  
        int current = 0,after = 0,before = 0;  
        while((n/i)!= 0){             
            current = (n/i)%10; //高位数字  
            before = n/(i\*10); //当前位数字  
            after = n-(n/i)\*i; //低位数字  
            //如果为0,出现1的次数由高位决定,等于高位数字 \* 当前位数  
            if (current == 0)  
                count += before\*i;  
            //如果为1,出现1的次数由高位和低位决定,高位\*当前位+低位+1  
            else if(current == 1)  
                count += before \* i + after + 1;  
            //如果大于1,出现1的次数由高位决定,//（高位数字+1）\* 当前位数  
            else{  
                count += (before + 1) \* i;  
            }      
            //前移一位  
            i = i\*10;  
        }  
        return count;  
    }  
}  
//面试题44: 数字序列中某一位的数字  
public class DigitsInSequence  
{  
    public static void main(String[] args)  
    {  
        System.out.println(digitAtIndex(9)); //9  
        System.out.println(digitAtIndex(189)); //数字99的最后一位：9  
        System.out.println(digitAtIndex(190)); //数字100的第一位：1  
    }

    private static int digitAtIndex(int index)  
    {  
        if (index < 0) return -1;  
        int digits = 1;  
        while (true)  
        {  
            int digitNumbers = countOfNumbersFor(digits); //当前位数的数值个数  
            //数值乘上它的位数等于数字个数，  
            //比如，两位数有90个（10~99），每个数值有2个数字，总数字个数为180  
            int countOfNumbers = digitNumbers \* digits;  
            if (index < countOfNumbers)  
            {  
                return digitAtIndex(index, digits);  
            } else  
            {  
                //在下一位中查找  
                index -= countOfNumbers;  
                digits++;  
            }  
        }  
    }

    //digits位数的数字个数，  
    //两位数有9\*10=90个（10~99），三位数有9\*100=900个（100~999）  
    private static int countOfNumbersFor(int digits)  
    {  
        if (digits == 1)  
            return 10;

        int count = (int) Math.pow(10, digits - 1);  
        return 9 \* count;  
    }

    private static int digitAtIndex(int index, int digits)  
    {  
        //对应的数值  
        int number = beginNumberFor(digits) + index / digits;  
        //从数值右边开始算的位置  
        int indexFromRight = digits - index % digits;  
        //去除右边的indexFromRight-1个数字  
        for (int i = 1; i < indexFromRight; i++)  
            number /= 10;  
        //求个位数字  
        return number % 10;  
    }

    //digits位数的第一个数字，两位数从10开始，三位数从100开始  
    private static int beginNumberFor(int digits)  
    {  
        if (digits == 1)  
            return 0;

        return (int) Math.pow(10, digits - 1);  
    }  
}  
   
//面试题45:把数组排成最小的数  
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]);  
     
  }  
  Collections.sort(list, new Comparator<Integer>(){  
   
  public int compare(Integer str1,Integer str2){  
   String s1=str1+""+str2;  
   String s2=str2+""+str1;  
         return s1.compareTo(s2);  
     }  
  });  
   
  for(int j:list){  
   s+=j;  
  }  
        return s;

    }  
}  
//面试题46:把数字翻译成字符串  
/\*  
给定一个数字，我们按照如下规则把它翻译为字符串：0翻译成”a”，1翻译成”b”，……，11翻译成”l”，……，25翻译成”z”  
一个数字可能有多个翻译。例如12258有5种不同的翻译，它们分别是”bccfi”、”bwfi”、”bczi”、”mcfi”和”mzi”  
请编程实现一个函数用来计算一个数字有多少种不同的翻译方法。

主要思路：可以选一个数字或两个连续的数字（10~25）翻译成一个字符。  
定义f(i)f(i)f(i)：从第i位数字开始的不同翻译数目  
1）若第i个数字和第i+1个数字拼接成的数字在10~25范围内，则递归式子为：  
f(i)=f(i+1)+f(i+2)f(i)=f(i+1)+f(i+2)f(i) =f(i+1) + f(i+2)  
2）否则  
f(i)=f(i+1)f(i)=f(i+1)f(i) =f(i+1)

为了防止重复计算，按i从大到小计算。

关键点：递归  
   
\*/  
public class NumberToString  
{  
    public static void main(String[] args)  
    {  
        System.out.println(getTranslationCount(10)); //2种:bc(1,0)或k(10)  
        System.out.println(getTranslationCount(125)); //3  
        System.out.println(getTranslationCount(12258)); //5  
    }

    private static int getTranslationCount(int number)  
    {  
        if (number < 0) return 0; //负数默认可翻译个数为0  
        return translationCount(String.valueOf(number));  
    }

    private static int translationCount(String number)  
    {  
        int length = number.length();  
        int[] countRecords = new int[length];  
        //只有一个数字，则只有一种翻译方式  
        countRecords[length - 1] = 1;  
        int count;  
        for (int i = length - 2; i >= 0; i--)  
        {  
            //f(i+1)  
            count = countRecords[i + 1];  
            int digit1 = number.charAt(i) - '0';  
            int digit2 = number.charAt(i + 1) - '0';  
            int connectedNumber = digit1 \* 10 + digit2; //拼接两个数字  
            //拼接的数字在10~25范围内  
            if (connectedNumber >= 10 && connectedNumber <= 25)  
            {  
                if (i < length - 2)  
                {  
                    //f(i) = f(i+1) + f(i+2)  
                    count += countRecords[i + 2];  
                } else if (i == length - 2)  
                {  
                    count += 1;  //拼接的数字只有一种翻译方式  
                }  
            }  
            countRecords[i] = count;  
        }  
        count = countRecords[0];  
        return count;  
    }  
}  
   
//面试题47:礼物的最大价值  
/\*\*  
思路1：动态规划  
申请一个与原矩阵行列数一样的二维数组dp[][]，初始化dp[0][i] = data[0][i]；  
然后依次更新dp的每一行即可。由于第m行的值与第m-1行和第m行有关，  
因此可以对dp进行简化，仅用两行的dp，即dp[2][]即可完成状态的记录与更新。

思路2：广度优先遍历  
这个棋盘其实可以看成一个有向图，起点为左上角，终点为右下角，  
每一点仅仅指向右侧和下侧。因此我们可以从左上角开始进行广度优先遍历。  
此外，遍历过程中可以进行剪枝，最终移动到右下角时会仅剩下一个枝，该路径所经的点的数值之和即为所求  
\*/  
package chapter5;

import java.util.LinkedList;  
import java.util.Queue;

/\*\*  
 \* Created with IntelliJ IDEA  
 \* Author: ryder  
 \* Date  : 2017/8/11  
 \* Time  : 13:03  
 \* Description:礼物的最大价值  
 \*\*/  
public class P233\_MaxValueOfGifts {  
    //方法一：动态规划  
    public static int getMaxVaule(int[][] data){  
        if(data.length==0||data[0].length==0)  
            return 0;  
        int[][] dp = new int[2][data[0].length];  
        int dpCurRowIndex = 0,dpPreRowIndex = 0;  
        for(int row=0;row<data.length;row++){  
            dpCurRowIndex = row&1;  
            dpPreRowIndex = 1 - dpCurRowIndex;  
            for(int col=0;col<data[0].length;col++){  
                if(col==0)  
                    dp[dpCurRowIndex][col] = dp[dpPreRowIndex][col]+data[row][col];  
                else{  
                    if(dp[dpPreRowIndex][col]>=dp[dpCurRowIndex][col-1])  
                        dp[dpCurRowIndex][col] = dp[dpPreRowIndex][col]+data[row][col];  
                    else  
                        dp[dpCurRowIndex][col] = dp[dpCurRowIndex][col-1]+data[row][col];  
                }  
            }  
        }  
        return dp[(data.length-1)&1][data[0].length-1];  
    }

    //方法二：有向图的遍历（广度优先，可再剪枝进行优化）  
    public static int getMaxVaule2(int[][] data){  
        if(data.length==0||data[0].length==0)  
            return 0;  
        int maxRowIndex = data.length-1;  
        int maxColIndex = data[0].length-1;  
        Queue<Node> queue = new LinkedList<>();  
        queue.offer(new Node(0,0,data[0][0]));  
        Node nodeCur = null;  
        while (!(queue.peek().row==maxRowIndex && queue.peek().col==maxColIndex)){  
            nodeCur = queue.poll();  
            if(nodeCur.row!=maxRowIndex)  
                queue.offer(new Node(nodeCur.row+1,nodeCur.col,nodeCur.sum+data[nodeCur.row+1][nodeCur.col]));  
            if(nodeCur.col!=maxColIndex)  
                queue.offer(new Node(nodeCur.row,nodeCur.col+1,nodeCur.sum+data[nodeCur.row][nodeCur.col+1]));  
        }  
        int maxSum = 0,temp = 0;  
        while (!queue.isEmpty()){  
            temp = queue.poll().sum;  
            if(temp>maxSum)  
                maxSum = temp;  
        }  
        return maxSum;  
    }  
    public static class Node{  
        int row;  
        int col;  
        int sum;  
        public Node(int r,int c,int s){  
            row = r;col = c;sum = s;  
        }  
    }  
    public static void main(String[] args){  
        int[][] data = {  
                {1,10,3,8},  
                {12,2,9,6},  
                {5,7,4,11},  
                {3,7,16,5}};  
        System.out.println(getMaxVaule(data));  
        System.out.println(getMaxVaule2(data));  
    }  
}  
//面试题48：最长不含重复字符的子字符串  
/\*\*  
题目：最长不含重复字符的子字符串  
请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。  
假设字符串中只包含从’a’到’z’的字符。例如，在字符串中”arabcacfr”，最长非重复子字符串为”acfr”，长度为4。

主要思路：使用动态规划，记录当前字符之前的最长非重复子字符串长度f(i-1)，  
其中i为当前字符的位置。每次遍历当前字符时，分两种情况：

1）若当前字符第一次出现，则最长非重复子字符串长度f(i) = f(i-1)+1。  
2）若当前字符不是第一次出现，则首先计算当前字符与它上次出现位置之间的距离d。  
若d大于f(i-1)，即说明前一个非重复子字符串中没有包含当前字符，则可以添加当前字符到前一个非重复子字符串中，  
所以，f(i) = f(i-1)+1。若d小于或等于f(i-1)，即说明前一个非重复子字符串中已经包含当前字符，  
则不可以添加当前字符，所以，f(i) = d。

关键点：动态规划，两个重复字符的距离

\*/

public class LongestSingleSubstring  
{  
    public static void main(String[] args)  
    {  
        System.out.println(findLongestSubstringLength("arabcacfr")); //4  
        System.out.println(findLongestSubstringLength("bbb")); //1  
        System.out.println(findLongestSubstringLength("")); //0  
    }

    private static int findLongestSubstringLength(String string)  
    {  
        if (string == null || string.equals("")) return 0;  
        int maxLength = 0;  
        int curLength = 0;  
        int[] positions = new int[26];  
        for (int i = 0; i < positions.length; i++)  
        {  
            positions[i] = -1; //初始化为-1，负数表示没出现过  
        }  
        for (int i = 0; i < string.length(); i++)  
        {  
            int curChar = string.charAt(i) - 'a';  
            int prePosition = positions[curChar];  
            //当前字符与它上次出现位置之间的距离  
            int distance = i - prePosition;  
            //当前字符第一次出现，或者前一个非重复子字符串中没有包含当前字符  
            if (prePosition < 0 || distance > curLength)  
            {  
                curLength++;  
            } else  
            {  
                //更新最长非重复子字符串的长度  
                if (curLength > maxLength)  
                {  
                    maxLength = curLength;  
                }  
                curLength = distance;  
            }  
            positions[curChar] = i; //更新字符出现的位置  
        }  
        if (curLength > maxLength)  
        {  
            maxLength = curLength;  
        }  
        return maxLength;  
    }  
}  
   
//面试题49：丑数  
/\*  
说下思路，如果p是丑数，那么p=2^x \* 3^y \* 5^z  
那么只要赋予x,y,z不同的值就能得到不同的丑数。  
如果要顺序找出丑数，要知道下面几个特（fei）点（hua）。  
对于任何丑数p：  
（一）那么2\*p,3\*p,5\*p都是丑数，并且2\*p<3\*p<5\*p  
（二）如果p<q, 那么2\*p<2\*q,3\*p<3\*q,5\*p<5\*q  
现在说说算法思想：  
    由于1是最小的丑数，那么从1开始，把2\*1，3\*1，5\*1，进行比较，得出最小的就是1  
的下一个丑数，也就是2\*1，  
    这个时候，多了一个丑数‘2’，也就又多了3个可以比较的丑数，2\*2，3\*2，5\*2，  
这个时候就把之前‘1’生成的丑数和‘2’生成的丑数加进来也就是  
(3\*1,5\*1,2\*2，3\*2，5\*2)进行比较，找出最小的。。。。如此循环下去就会发现，  
每次选进来一个丑数，该丑数又会生成3个新的丑数进行比较。  
    上面的暴力方法也应该能解决，但是如果在面试官用这种方法，估计面试官只会摇头吧  
。下面说一个O（n）的算法。  
    在上面的特（fei）点（hua）中，既然有p<q, 那么2\*p<2\*q，那么  
“我”在前面比你小的数都没被选上，你后面生成新的丑数一定比“我”大吧，那么你乘2  
生成的丑数一定比我乘2的大吧，那么在我选上之后你才有机会选上。  
其实每次我们只用比较3个数：用于乘2的最小的数、用于乘3的最小的数，用于乘5的最小的  
数。也就是比较(2\*x , 3\*y, 5\*z) ，x>=y>=z的，  
重点说说下面代码中p的作用：int p[] = new int[] { 0, 0, 0 }; p[0]表示最小用于  
乘2比较数在数组a中的【位置】。 \*/  
public class Solution {  
    final int d[] = { 2, 3, 5 };  
    public int GetUglyNumber\_Solution(int index) {  
        if(index == 0) return 0;  
        int a[] = new int[index];  
        a[0] = 1;  
        int p[] = new int[] { 0, 0, 0 };  
        int num[] = new int[] { 2, 3, 5 };  
        int cur = 1;  
   
        while (cur < index) {  
            int m = finMin(num[0], num[1], num[2]);  
            if (a[cur - 1] < num[m])  
                a[cur++] = num[m];  
            p[m] += 1;  
            num[m] = a[p[m]] \* d[m];  
        }  
        return a[index - 1];  
    }  
   
    private int finMin(int num2, int num3, int num5) {  
        int min = Math.min(num2, Math.min(num3, num5));  
        return min == num2 ? 0 : min == num3 ? 1 : 2;  
    }  
}  
//面试题50：第一个只出现一次的字符  
import java.util.LinkedHashMap;  
// use linkedhashmap to keep the order  
public class Solution {  
    public int FirstNotRepeatingChar(String str) {  
        LinkedHashMap <Character, Integer> map = new LinkedHashMap<Character, Integer>();  
        for(int i=0;i<str.length();i++){  
            if(map.containsKey(str.charAt(i))){  
                int time = map.get(str.charAt(i));  
                map.put(str.charAt(i), ++time);  
            }  
            else {  
                map.put(str.charAt(i), 1);  
            }  
        }  
        int pos = -1;    
        int i=0;  
        for(;i<str.length();i++){  
            char c = str.charAt(i);  
            if (map.get(c) == 1) {  
                return i;  
            }  
        }  
        return pos;  
    }  
}  
//面试题51：数组中的逆序对  
/\*归并排序的改进，把数据分成前后两个数组(递归分到每个数组仅有一个数据项)，  
合并数组，合并时，出现前面的数组值array[i]大于后面数组值array[j]时；则前面  
数组array[i]~array[mid]都是大于array[j]的，count += mid+1 - i  
参考剑指Offer，但是感觉剑指Offer归并过程少了一步拷贝过程。  
还有就是测试用例输出结果比较大，对每次返回的count mod(1000000007)求余  
\*/  
   
public class Solution {  
    public int InversePairs(int [] array) {  
        if(array==null||array.length==0)  
        {  
            return 0;  
        }  
        int[] copy = new int[array.length];  
        for(int i=0;i<array.length;i++)  
        {  
            copy[i] = array[i];  
        }  
        int count = InversePairsCore(array,copy,0,array.length-1);//数值过大求余  
        return count;  
           
    }  
    private int InversePairsCore(int[] array,int[] copy,int low,int high)  
    {  
        if(low==high)  
        {  
            return 0;  
        }  
        int mid = (low+high)>>1;  
        int leftCount = InversePairsCore(array,copy,low,mid)%1000000007;  
        int rightCount = InversePairsCore(array,copy,mid+1,high)%1000000007;  
        int count = 0;  
        int i=mid;  
        int j=high;  
        int locCopy = high;  
        while(i>=low&&j>mid)  
        {  
            if(array[i]>array[j])  
            {  
                count += j-mid;  
                copy[locCopy--] = array[i--];  
                if(count>=1000000007)//数值过大求余  
                {  
                    count%=1000000007;  
                }  
            }  
            else  
            {  
                copy[locCopy--] = array[j--];  
            }  
        }  
        for(;i>=low;i--)  
        {  
            copy[locCopy--]=array[i];  
        }  
        for(;j>mid;j--)  
        {  
            copy[locCopy--]=array[j];  
        }  
        for(int s=low;s<=high;s++)  
        {  
            array[s] = copy[s];  
        }  
        return (leftCount+rightCount+count)%1000000007;  
    }  
}  
//面试题52：两个链表的第一个公共结点  
/\*  
public class ListNode {  
    int val;  
    ListNode next = null;

    ListNode(int val) {  
        this.val = val;  
    }  
}\*/  
import java.util.HashMap;  
public class Solution {  
    public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {  
        ListNode current1 = pHead1;  
        ListNode current2 = pHead2;  
   
   
        HashMap<ListNode, Integer> hashMap = new HashMap<ListNode, Integer>();  
        while (current1 != null) {  
            hashMap.put(current1, null);  
            current1 = current1.next;  
        }  
        while (current2 != null) {  
            if (hashMap.containsKey(current2))  
                return current2;  
            current2 = current2.next;  
        }  
   
        return null;  
   
    }  
}  
//面试题53：在排序数组中查找数字  
//统计一个数字在排序数组中出现的次数。  
public class Solution {  
    public int GetNumberOfK(int [] array , int k) {  
      int num = 0;  
        if (array != null && array.length > 0) {  
            int firstKIndex = getFirstK(array, k, 0, array.length - 1);  
            int lastKIndex = getLastK(array, k, 0, array.length - 1);  
            if (firstKIndex > -1 && lastKIndex > -1)  
                num = lastKIndex - firstKIndex + 1;  
        }  
        return num;  
    }

    /\*  
    \* 找到第一个出现的数字的下标  
    \*/  
    public  int getFirstK(int[] array, int k, int start, int end) {  
        if (start > end)  
            return -1;  
        int middleIndex = start + (end - start) / 2;  
        int middleData = array[middleIndex];  
        //先判断中间的值是否等于k  
        if (middleData == k) {  
        //判断是不是第一个K，前一个不等于K，就是第一个K  
            if (middleIndex > 0 && array[middleIndex - 1] != k || middleIndex == 0) {  
                return middleIndex;  
            } else  
                end = middleIndex - 1;  
        } else if (middleData > k) {  
            end = middleIndex - 1;  
        } else  
            start = middleIndex + 1;  
        return getFirstK(array, k, start, end);  
    }

     /\*  
    \* 找到最后一个出现的数字的下标  
    \*/  
    public  int getLastK(int array[], int k, int start, int end) {  
        if (start > end) {  
            return -1;  
        }  
        int middleIndex = (start + end) / 2;  
        int middleData = array[middleIndex];  
        if (middleData == k) {  
         //判断是不是最后一个K，后一个不等于K，就是最后一个K  
            if (middleIndex < array.length - 1 && array[middleIndex + 1] != k || middleIndex == array.length - 1)  
                return middleIndex;  
            else  
                start = middleIndex + 1;  
        } else if (middleData < k) {  
            start = middleIndex + 1;  
        } else  
            end = middleIndex - 1;  
        return getLastK(array, k, start, end);

    }  
}

//面试题54: 二叉搜索树的第k大节点  
/\*  
public class TreeNode {  
    int val = 0;  
    TreeNode left = null;  
    TreeNode right = null;

    public TreeNode(int val) {  
        this.val = val;

    }

}  
\*/

//思路：二叉搜索树按照中序遍历的顺序打印出来正好就是排序好的顺序。  
//     所以，按照中序遍历顺序找到第k个结点就是结果。  
public class Solution {  
   int index = 0; //计数器  
    TreeNode KthNode(TreeNode root, int k)  
    {  
        if(root != null){ //中序遍历寻找第k个  
            TreeNode node = KthNode(root.left,k);  
            if(node != null)  
                return node;  
            index ++;  
            if(index == k)  
                return root;  
            node = KthNode(root.right,k);  
            if(node != null)  
                return node;  
        }  
        return null;  
    }  
}  
//面试题55：二叉树的深度  
/\*\*  
public class TreeNode {  
    int val = 0;  
    TreeNode left = null;  
    TreeNode right = null;

    public TreeNode(int val) {  
        this.val = val;

    }

}  
\*/  
public class Solution {  
    public int TreeDepth(TreeNode root){  
        if(root==null){  
            return 0;  
        }  
        int left=TreeDepth(root.left);  
        int right=TreeDepth(root.right);  
        return Math.max(left,right)+1;  
    }  
}  
//面试题56:数组中数字出现的次数  
//二分查找  
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 && array[mid+1] == k){  
                start = mid+1;  
            }else{  
                return mid;  
            }  
            mid = (start + end) >> 1;  
        }  
        return -1;  
    }  
}  
//面试题57：和为s的数字  
//左右夹逼  
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]);  
                return list;  
           }else if(array[i]+array[j]>sum){  
                j--;  
            }else{  
                i++;  
            }  
               
        }  
        return list;  
    }  
}  
//面试题58:翻转字符串(翻转单词顺序列)  
public class Solution {  
    public String ReverseSentence(String str) {  
        if(str.length() == 0){  
            return "";  
        }  
        char[] chars = str.toCharArray();  
        reverse(chars, 0, chars.length - 1);//整个字符串翻转  
        int start = 0;  
        for(int i = 0; i <= chars.length; i++){//最后一个没有空格，用越界判断  
            if(i == chars.length || chars[i] == ' '){  
                reverse(chars, start, i - 1);  
                start = i + 1;  
            }  
        }  
        return String.valueOf(chars);  
    }  
      
    public void reverse(char[] chars, int start, int end){//翻转字符数组  
        char temp = ' ';  
        while(start < end){  
            temp = chars[start];  
            chars[start] = chars[end];  
            chars[end] = temp;  
            start++;  
            end--;  
        }  
    }  
}  
//面试题59：队列的最大值  
/\*\*  
题目描述：给定一个数组和滑动窗口k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]}。  
     分别找出这6个滑动窗口中的最大值，并保存到一个列表中。  
     最简单的思路就是分别找到这些滑动窗口，并用一个数组来保存，  
     然后找出每个数组的最大值，但是因为比较每个数组需要时间复杂度为O（k），  
     滑动窗口滑动的时间复杂度为O（n），所以总的时间复杂度为O（nk）；  
     那么我们希望在窗口滑动的同时，进行最大值的更新，  
     那么可以使用一个双向的队列来保存滑动窗口中的值，  
     使最大值总是在双向队列的头部，新的元素和对尾的元素比较，如果队尾的元素小于新的元素，  
     就把队尾的元素删除，然后从队尾加入新的元素。但是怎么判断在队列头的最大值是否过期呢，  
     那么这就需要在双向队列中保存的是元素的下标，  
     当发现队列头部的元素下标和当前下标的差值大于滑动窗口的大小的时候，  
     说明该最大值过期，应该把他从队列头部删除掉。  
\*/  
public class Solution {  
    public ArrayList<Integer> maxInWindows(int [] num, int size)  
    {  
        //这道题的解题思路主要有两点：  
        //1、用一个双端队列ArrayDeque来保存滑动窗口中的元素  
        //2、双端队列里面保存的是元素的下标的值，而不是元素本身的值，这点非常重要，用于判断元素是否过期  
          
        ArrayList<Integer> aList = new ArrayList<>();  
        if(num == null || size<1 || num.length<size){  
            return aList;  
        }  
        ArrayDeque<Integer> aDeque = new ArrayDeque<>();  
        int begin;  
        for(int i = 0; i<num.length; i++){  
            begin = i-size+1;  
            if (aDeque.isEmpty()){  
                aDeque.add(i);  
            }  
            else if(begin>aDeque.peekFirst()){  
                aDeque.pollFirst();  
            }  
            while(!aDeque.isEmpty() && num[i]>=num[aDeque.peekLast()]){  
                aDeque.pollLast();  
            }  
            aDeque.add(i);  
            if(begin >= 0){  
                aList.add(num[aDeque.peekFirst()]);  
            }  
        }  
        return aList;  
    }  
}

//面试题60：n个骰子的点数  
/\*\*  
题目：把n个骰子仍在地上，所有骰子朝上一面的点数之和为s，输入n，打印出s的所有可能的值出现的概率。  
骰子一共6个面，每个面上都有一个点数，对应的是1-6之间的一个数字。  
所以n个骰子的点数和的最小值是n,最大值为6n.另外根据排列组合的知识，  
我们还知道n个骰子的所有点数的排列数为6^n.要解决这个问题，  
我们需要先统计出每一个点数出现的次数，然后把每一个点数出现的次数除以6^n,  
就能求出每个点数出现的概率。

解法一：基于递归求骰子的点数，时间效率不够高

    现在我们考虑如何统计每一个点数出现的次数。  
要向求出n个骰子的点数和，可以先把n个骰子分为两堆：  
第一堆只有一个，另一个有n-1个。单独的那一个有可能出现从1到6的点数。  
我们需要计算从1到6的每一种点数和剩下的n-1个骰子来计算点数和。  
接下来把剩下的n-1个骰子还是分成两堆，第一堆只有一个，第二堆有n-2个。  
我们把上一轮哪个单独骰子的点数和这一轮单独骰子的点数相加，再和n-2个骰子来计算点数和。  
分析到这里，我们不难发现这是一种递归的思路，递归结束的条件就是最后只剩下一个骰子。  
    我们可以定义一个长度为6n-n+1的数组保存n个骰子的点数和，  
和为sum的点数出现的次数保存到第sum-n个元素里。  
\*/  
int g\_maxValue = 6;    
public void printProbability(int number) {    
        if (number < 1)    
            return;    
        int maxSum=number\*g\_maxValue;  
        int[] pProbabilities= new int[maxSum-number + 1];    
        Probability(number,pProbabilities);  
      int total=Math.pow((Double)g\_maxValue,number);  
      for(int i=number;i<=maxSum;++i){  
   
            double ratio = (double) probabilities[flag][i] / total;    
            System.out.println(i);    
            System.out.println(ratio);    
        }    
 delete pProbalities;  
    }    
public void Probability(int number,int[] pProbabilities);  
}  
public void Probability(int original,int current,int sum,int[] pProbabilities){  
  if(current==1){  
     pProbabilities[sum-orginal]++;  
  }else{  
      for(int i=1;i<g\_maxValue;++i){  
         Probability(orginal,current-1,i+sum,pProbabilities);  
      }  
    }  
}  
/\*\*  
解法二：基于循环求骰子的点数，时间性能好  
   
可以换一个思路来解决这个问题，我们可以考虑用两个数组来存储骰子点数的每一个总数出现的次数。  
在一次循环中，每一个数组中的第n个数字表示骰子和为n出现的次数。  
在下一轮循环中，我们加上一个新的骰子，此时和为n出现的次数。  
下一轮中，我们加上一个新的骰子，  
此时和为n的骰子出现的次数应该等于上一次循环中骰子点数和为n-1,n-2,n-3,n-4,n-5,n-6的次数之和，  
所以我们把另一个数组的第n个数字设为前一个数组对应的第n-1，n-2，n-3，n-4，n-5与n-6之和。  
\*/  
 public void printProbability(int number) {    
        if (number < 1)    
            return;    
        int g\_maxValue = 6;    
        int[][] probabilities = new int[2][];    
        probabilities[0] = new int[g\_maxValue \* number + 1];    
        probabilities[1] = new int[g\_maxValue \* number + 1];    
        int flag = 0;    
        for (int i = 1; i <= g\_maxValue; i++)    
            probabilities[0][i] = 1;    
        for (int k = 2; k <= number; ++k) {    
            for (int i = 0; i < k; ++i)    
                probabilities[1 - flag][i] = 0;    
            for (int i = k; i <= g\_maxValue \* k; ++i) {    
                probabilities[1 - flag][i] = 0;    
                for (int j = 1; j <= i && j <= g\_maxValue; ++j)    
                    probabilities[1 - flag][i] += probabilities[flag][i - j];    
            }    
            flag = 1 - flag;    
        }    
        double total = Math.pow(g\_maxValue, number);    
        for (int i = number; i <= g\_maxValue \* number; i++) {    
            double ratio = (double) probabilities[flag][i] / total;    
            System.out.println(i);    
            System.out.println(ratio);    
        }    
    }

//面试题61: 扑克牌的顺子  
/\*  
1、排序  
2、计算所有相邻数字间隔总数  
3、计算0的个数  
4、如果2、3相等，就是顺子  
5、如果出现对子，则不是顺子  
\*/  
import java.util.Arrays;  
public class Solution {  
    public boolean isContinuous(int[] numbers) {  
        int numOfZero = 0;  
        int numOfInterval = 0;  
        int length = numbers.length;  
        if(length == 0){  
           return false;  
        }  
        Arrays.sort(numbers);  
        for (int i = 0; i < length - 1; i++) {  
            // 计算癞子数量  
            if (numbers[i] == 0) {  
                numOfZero++;  
                continue;  
            }  
            // 对子，直接返回  
            if (numbers[i] == numbers[i + 1]) {  
                return false;  
            }  
            numOfInterval += numbers[i + 1] - numbers[i] - 1;  
        }  
        if (numOfZero >= numOfInterval) {  
            return true;  
        }  
        return false;  
    }  
}  
//面试题62:圆圈中最后剩下的数(孩子们的游戏)  
import java.util.LinkedList;  
   
public class Solution {  
    public int LastRemaining\_Solution(int n, int m) {  
        LinkedList<Integer> list = new LinkedList<Integer>();  
        for (int i = 0; i < n; i ++) {  
            list.add(i);  
        }  
           
        int bt = 0;  
        while (list.size() > 1) {  
            bt = (bt + m - 1) % list.size();  
            list.remove(bt);  
        }  
           
        return list.size() == 1 ? list.get(0) : -1;  
    }  
}  
//面试题63:股票的最大利润  
/\*\*  
题目要求：  
求买卖股票一次能获得的最大利润。  
例如，输入{9，11，8，5，7，12，16，14}，5的时候买入，  
16的时候卖出，则能获得最大利润11。

解题思路：  
遍历过程中记录最小值min，然后计算当前值与min的差值diff，  
并更新maxDiff，maxDiff=max(diff)。

\*/package chapter6;

public class P304\_MaximalProfit {  
    public static int maxDiff(int[] data){  
        if(data==null||data.length<2)  
            return 0;  
        int min = data[0];  
        int maxDiff = data[1] - min;  
        if(data[1]<min)  
            min = data[1];  
        for(int i=2;i<data.length;i++){  
            if(data[i]-min>maxDiff)  
                maxDiff = data[i]-min;  
            if(data[i]<min)  
                min = data[i];  
        }  
        return maxDiff;  
    }  
    public static void main(String[] args){  
        int[] data1 = new int[]{9,11,8,5,7,12,16,14};  
        int[] data2 = new int[]{9,8,7,6,5,4,3,1};  
        System.out.println(maxDiff(data1)); //11  
        System.out.println(maxDiff(data2)); //-1  
    }  
}

//面试题64:求1+2+3..+n  
public class Solution {  
    public int Sum\_Solution(int n) {  
         return (int) (Math.pow(n, 2) + n) >> 1;  
    }  
}

//面试题65:不用加减乘除做加法  
public class Solution {  
    public int Add(int num1,int num2) {  
  return Integer.sum(num1,num2);  
    }  
}  
//面试题66：构建乘积数组  
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 \*=A[j+1];  
                B[j] \*=temp;  
            }  
        }  
        return B;  
    }  
}

//调整数组顺序使奇数位于偶数前面  
public class Solution {  
public void reOrderArray(int [] array) {  
        if(array.length==0||array.length==1) return;  
        int oddCount=0,oddBegin=0;  
        int[] newArray=new int[array.length];  
        for(int i=0;i<array.length;i++){  
            if((array[i]&1)==1) oddCount++;  
        }  
        for(int i=0;i<array.length;i++){  
            if((array[i]&1)==1) newArray[oddBegin++]=array[i];  
            else newArray[oddCount++]=array[i];  
        }  
        for(int i=0;i<array.length;i++){  
            array[i]=newArray[i];  
        }  
    }  
}  
//数组中出现次数超过一半的数字  
import java.util.\*;  
public class Solution {  
public int MoreThanHalfNum\_Solution(int [] array) {  
        HashMap<Integer,Integer> hashmap=new HashMap<>();  
        for(int i=0;i<array.length;i++){  
            Integer tmp=hashmap.get(array[i]);  
            if(tmp==null){  
                hashmap.put(array[i], 1);  
                tmp=1;  
            }else{  
                hashmap.put(array[i], tmp+1);  
            }  
            if(tmp+1>array.length/2) return array[i];  
        }  
        /\*  
        Iterator iter=hashmap.entrySet().iterator();  
        while(iter.hasNext()){  
            Map.Entry<Integer,Integer> entry=(Entry<Integer, Integer>) iter.next();  
            if(entry.getValue()>array.length/2){  
                return entry.getKey();  
            }  
        }  
        \*/  
        return 0;  
    }  
}  
//和为s的连续正数序列  
import java.util.ArrayList;  
public class Solution {  
    public ArrayList<ArrayList<Integer> > FindContinuousSequence(int sum) {  
        ArrayList<ArrayList<Integer>> ans = new ArrayList<>();  
        for (int n = (int) Math.sqrt(2 \* sum); n >= 2; n--) {  
            if ((n & 1) == 1 && sum % n == 0 || (sum % n) \* 2 == n) {  
                ArrayList<Integer> list = new ArrayList<>();  
                for (int j = 0, k = (sum / n) - (n - 1) / 2; j < n; j++, k++) {  
                    list.add(k);  
                }  
                ans.add(list);  
            }  
        }  
        return ans;  
    }  
}  
//左旋转字符串  
public class Solution {  
public String LeftRotateString(String str,int n) {  
int length = str.length();  
if(length<=0){  
return "";  
}  
StringBuffer sb = new StringBuffer(str.substring(0, n));  
StringBuffer sb1 = new StringBuffer(str.substring(n, str.length()));  
sb1.append(sb);  
//System.out.println(sb1);  
return sb1.toString();  
          
    }  
}  
//数组中只出现一次的数字

public class Solution {  
    public void FindNumsAppearOnce(int[] array, int[] num1, int[] num2)    {  
        int length = array.length;  
        if(length == 2){  
            num1[0] = array[0];  
            num2[0] = array[1];  
            return;  
        }  
        int bitResult = 0;  
        for(int i = 0; i < length; ++i){  
            bitResult ^= array[i];  
        }  
        int index = findFirst1(bitResult);  
        for(int i = 0; i < length; ++i){  
            if(isBit1(array[i], index)){  
                num1[0] ^= array[i];  
            }else{  
                num2[0] ^= array[i];  
            }  
        }  
    }  
       
    private int findFirst1(int bitResult){  
        int index = 0;  
        while(((bitResult & 1) == 0) && index < 32){  
            bitResult >>= 1;  
            index++;  
        }  
        return index;  
    }  
       
    private boolean isBit1(int target, int index){  
        return ((target >> index) & 1) == 1;  
    }  
}  
//把字符串转换成整数  
public class Solution {  
    public static boolean flag;  
    public static int StrToInt(String str) {  
        flag = false;  
        //判断输入是否合法  
        if (str == null || str.trim().equals("")) {  
            flag = true;  
            return 0;  
        }  
        // symbol=0,说明该数为正数;symbol=1，该数为负数;start用来区分第一位是否为符号位  
        int symbol = 0;  
        int start = 0;  
        char[] chars = str.trim().toCharArray();  
        if (chars[0] == '+') {  
            start = 1;  
        } else if (chars[0] == '-') {  
            start = 1;  
            symbol = 1;  
        }  
        int result = 0;  
        for (int i = start; i < chars.length; i++) {  
            if (chars[i] > '9' || chars[i] < '0') {  
                flag = true;  
                return 0;  
            }  
            int sum= result \* 10 + (int) (chars[i] - '0');  
               
               
            if((sum-(int) (chars[i] - '0'))/10!=result){  
                flag=true;  
                return 0;  
            }  
               
            result=result \* 10 + (int) (chars[i] - '0');  
            /\*  
             \* 本人认为java热门第一判断是否溢出是错误的，举个反例  
             \* 当输入为value=2147483648时，在计算机内部的表示应该是-2147483648  
             \* 显然value>Integer.MAX\_VALUE是不成立的  
            \*/  
        }  
        // 注意：java中-1的n次方不能用：(-1)^n .'^'异或运算  
        // 注意，当value=-2147483648时，value=-value  
        result = (int) Math.pow(-1, symbol) \* result;  
        return result;  
    }  
}  
//字符流中第一个不重复的字符

public class Solution {  
    //Insert one char from stringstream  
    private int[] occurence = new int[256];  
    private int index;  
      
    public Solution(){  
        for(int i=0;i<256;i++){  
            occurence[i] = -1;  
        }  
        index = 0;  
    }  
    void Insert(char ch)  
    {  
        if(occurence[ch] == -1)  
            occurence[ch] = index;  
        else if(occurence[ch] >= 0)  
            occurence[ch] = -2;  
          
        index++;  
    }  
  //return the first appearence once char in current stringstream  
    char FirstAppearingOnce()  
    {  
        char ch = '\0';  
        int minIndex = Integer.MAX\_VALUE;  
        for(int i=0;i<256;i++){  
            if(occurence[i] >=0 && occurence[i]<minIndex){  
                ch = (char)i;  
                minIndex = occurence[i];  
            }  
        }  
        if(ch == '\0')  
            return '#';  
        return ch;  
    }  
}  
//对称的二叉树  
/\*  
public class TreeNode {  
    int val = 0;  
    TreeNode left = null;  
    TreeNode right = null;

    public TreeNode(int val) {  
        this.val = val;

    }

}  
\*/  
public class Solution {  
boolean isSymmetrical(TreeNode pRoot)  
    {  
        if(pRoot == null) return true;  
        return isSymmetrical(pRoot.left, pRoot.right);  
    }  
    private boolean isSymmetrical(TreeNode left, TreeNode right) {  
        if(left == null && right == null) return true;  
        if(left == null || right == null) return false;  
        return left.val == right.val //为镜像的条件：左右节点值相等  
                && isSymmetrical(left.left, right.right) //2.对称的子树也是镜像  
                && isSymmetrical(left.right, right.left);  
    }  
}  
//按之字形顺序打印二叉树  
import java.util.\*;

/\*  
public class TreeNode {  
    int val = 0;  
    TreeNode left = null;  
    TreeNode right = null;

    public TreeNode(int val) {  
        this.val = val;

    }

}  
\*/  
public class Solution {

/\*\*  
 \* 大家的实现很多都是将每层的数据存进ArrayList中，偶数层时进行reverse操作，  
 \* 在海量数据时，这样效率太低了。  
 \* （我有一次面试，算法考的就是之字形打印二叉树，用了reverse，  
 \* 直接被鄙视了，面试官说海量数据时效率根本就不行。）  
 \*  
 \* 下面的实现：不必将每层的数据存进ArrayList中，偶数层时进行reverse操作，直接按打印顺序存入  
 \* 思路：利用Java中的LinkedList的底层实现是双向链表的特点。  
 \*     1)可用做队列,实现树的层次遍历  
 \*     2)可双向遍历,奇数层时从前向后遍历，偶数层时从后向前遍历  
 \*/  
public ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {  
    ArrayList<ArrayList<Integer>> ret = new ArrayList<>();  
    if (pRoot == null) {  
        return ret;  
    }  
    ArrayList<Integer> list = new ArrayList<>();  
    LinkedList<TreeNode> queue = new LinkedList<>();  
    queue.addLast(null);//层分隔符  
    queue.addLast(pRoot);  
    boolean leftToRight = true;  
       
    while (queue.size() != 1) {  
        TreeNode node = queue.removeFirst();  
        if (node == null) {//到达层分隔符  
            Iterator<TreeNode> iter = null;  
            if (leftToRight) {  
                iter = queue.iterator();//从前往后遍历  
            } else {  
                iter = queue.descendingIterator();//从后往前遍历  
            }  
            leftToRight = !leftToRight;  
            while (iter.hasNext()) {  
                TreeNode temp = (TreeNode)iter.next();  
                list.add(temp.val);  
            }  
            ret.add(new ArrayList<Integer>(list));  
            list.clear();  
            queue.addLast(null);//添加层分隔符  
            continue;//一定要continue  
        }  
        if (node.left != null) {  
            queue.addLast(node.left);  
        }  
        if (node.right != null) {  
            queue.addLast(node.right);  
        }  
    }  
       
    return ret;  
}

}  
//把二叉树打印成多行  
import java.util.ArrayList;

/\*  
public class TreeNode {  
    int val = 0;  
    TreeNode left = null;  
    TreeNode right = null;

    public TreeNode(int val) {  
        this.val = val;

    }

}  
\*/

//用递归做的  
public class Solution {  
    ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {  
        ArrayList<ArrayList<Integer>> list = new ArrayList<>();  
        depth(pRoot, 1, list);  
        return list;  
    }  
       
    private void depth(TreeNode root, int depth, ArrayList<ArrayList<Integer>> list) {  
        if(root == null) return;  
        if(depth > list.size())  
            list.add(new ArrayList<Integer>());  
        list.get(depth -1).add(root.val);  
           
        depth(root.left, depth + 1, list);  
        depth(root.right, depth + 1, list);  
    }  
}  
//滑动窗口的最大值  
import java.util.\*;  
/\*\*  
用一个双端队列，队列第一个位置保存当前窗口的最大值，当窗口滑动一次  
1.判断当前最大值是否过期  
2.新增加的值从队尾开始比较，把所有比他小的值丢掉  
\*/  
public class Solution {  
   public ArrayList<Integer> maxInWindows(int [] num, int size)  
    {  
        ArrayList<Integer> res = new ArrayList<>();  
        if(size == 0) return res;  
        int begin;  
        ArrayDeque<Integer> q = new ArrayDeque<>();  
        for(int i = 0; i < num.length; i++){  
            begin = i - size + 1;  
            if(q.isEmpty())  
                q.add(i);  
            else if(begin > q.peekFirst())  
                q.pollFirst();  
           
            while((!q.isEmpty()) && num[q.peekLast()] <= num[i])  
                q.pollLast();  
            q.add(i);    
            if(begin >= 0)  
                res.add(num[q.peekFirst()]);  
        }  
        return res;  
    }  
}