作者：半情调

1.二维数组中的查找

在一个二维数组中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

思路

类似于二分查找，根据题目，如果拿数组中任意一个元素与目标数值进行比较，如果该元素小于目标数值，那么目标数值一定是在该元素的下方或右方，如果大于目标数值，那么目标数值一定在该元素的上方或者左方。 在二维数组的查找中，两个指针是一个上下方向移动，一个是左右方向移动。两个指针可以从同一个角出发。本题从右上角出发寻找解题思路。

代码

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

2.替换空格

请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

代码

```
public class Solution {
    public String replaceSpace(StringBuffer str) {
        StringBuffer res = new StringBuffer();
        int len = str.length() -1;
        for(int i=0; i<=len; i++){
             if(str.charAt(i) == ' ')
                res.append("%20");
            else
                res.append(str.charAt(i));
        }
```

```
        return res.toString();
    }
}
//charAt()方法用于返回指定索引处的字符
```

3.从尾到头打印链表

输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。

思路：使用栈从头到尾push链表的元素，然后pop所有的元素到一个list中并返回。

代码

```
/**
*    public class ListNode {
*        int val;
*        ListNode next = null;
*        ListNode(int val) {
*            this.val = val;
*        }
*    }
*/
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
        ArrayList<Integer> arr = new ArrayList<Integer>();
        ListNode p = listNode;
        ArrayList<Integer> stack = new ArrayList<Integer>();
        while(p!=null){
            stack.add(p.val);
            p = p.next;
        }
        int n = stack.size()-1;
        for(int i=n;i>=0;i--){
            arr.add(stack.get(i));
        }
        return arr;
    }
}
```

4.重建二叉树

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

思路：先序遍历和中序遍历的关系，先序遍历的第一个值是根节点的值。在中序遍历中，根节点左边的值是左子树，右边的值是右子树上的值。

代码

```
/**
 * Definition for binary tree
 * public class TreeNode {
```

```
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        if(pre.length == 0 || in.length == 0)
            return null;
        return buildTree(pre, in, 0, pre.length - 1, 0, in.length - 1);
    }
    public TreeNode buildTree(int[] pre, int[] in, int preStart, int preEnd, int inStart, int inEnd){
        TreeNode root = new TreeNode(pre[preStart]);
        int i = 0;
        for(; i < in.length; i++){
            if(in[i] == root.val)
                break;
        }
        int leftLength = i - inStart;
        int rightLength = inEnd - i;
        if(leftLength > 0)
            root.left = buildTree(pre, in, preStart+1, preStart+leftLength, inStart, i-1);
        if(rightLength > 0)
            root.right = buildTree(pre, in, preStart+leftLength+1, preEnd, i+1, inEnd);
        return root;
    }
}
```

分析

根据中序遍历和前序遍历可以确定二叉树，具体过程为：

1. 根据前序序列第一个结点确定根结点
2. 根据根结点在中序序列中的位置分割出左右两个子序列
3. 对左子树和右子树分别递归使用同样的方法继续分解

例如：
前序序列{1,2,4,7,3,5,6,8} = pre
中序序列{4,7,2,1,5,3,8,6} = in

1. 根据当前前序序列的第一个结点确定根结点，为 1
2. 找到 1 在中序遍历序列中的位置，为 in[3]
3. 切割左右子树，则 in[3] 前面的为左子树， in[3] 后面的为右子树
4. 则切割后的**左子树前序序列**为：{2,4,7}，切割后的**左子树中序序列**为：{4,7,2}；切割后的**右子树前序序列**为：{3,5,6,8}，切割后的**右子树中序序列**为：{5,3,8,6}
5. 对子树分别使用同样的方法分解

```
import java.util.Arrays;
public class Solution {
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        if (pre.length == 0 || in.length == 0) {
            return null;
```

```
        }
        TreeNode root = new TreeNode(pre[0]);
        // 在中序中找到前序的根
        for (int i = 0; i < in.length; i++) {
            if (in[i] == pre[0]) {
                // 左子树，注意 copyOfRange 函数，左闭右开
                root.left =
reConstructBinaryTree(Arrays.copyOfRange(pre, 1, i + 1),
Arrays.copyOfRange(in, 0, i));
                // 右子树，注意 copyOfRange 函数，左闭右开
                root.right =
reConstructBinaryTree(Arrays.copyOfRange(pre, i + 1, pre.length),
Arrays.copyOfRange(in, i + 1, in.length));
                break;
            }
        }
        return root;
    }
}
```

5.用两个栈实现一个队列

用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

思路

定义两个stack，分别是stack1和stack2，队列的push和pop是在两侧的，push操作很简单，只需要在stack1上操作，而pop操作时，先将stack1的所有元素push到stack2中，然后stack2的pop返回的元素即为目标元素，然后把stack2中的所有元素再push到stack1中。

代码

```
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() {
        int temp;
        while(!stack1.empty()){
            temp = stack1.pop();
            stack2.push(temp);
        }
        int res = stack2.pop();
        while(!stack2.empty()){
            temp = stack2.pop();
            stack1.push(temp);
        }
        return res;
    }
}

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 (stack2.size() <= 0) {
            while (stack1.size() != 0) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }
}
```

6.旋转数组中的最小数字

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。 NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

思路：这个题很简单，题目说的不明白，意思是一个递增排序的数组做了一次旋转，给你旋转后的数组，找到最小元素。输入{3,4,5,1,2}输出1。

两个方法：1.遍历数组元素，如果前一个元素大于后一个元素，则找到了最小的元素。如果前一个一直小于后一个元素，说明没有旋转，返回第一个元素。

2.二分查找，如果中间元素位于递增元素，那么中间元素>最右边元素，最小元素在后半部分。否则，最小元素在前半部分。

代码

1.时间复杂度O(n)

```
import java.util.ArrayList;
public class Solution {
    public int minNumberInRotateArray(int [] array) {
        if(array.length==0)
            return 0;
        for(int i=0;i<array.length-1;i++){
            if(array[i] > array[i+1])
                return array[i+1];
        }
        return array[0];
    }
}
```

2.二分查找时间复杂度O(logn)

```
import java.util.ArrayList;
public class Solution {
    public int minNumberInRotateArray(int [] array) {
        if(array.length==0)
            return 0;
```

```
        int l=0;
        int r=array.length-1;
        while(l<r){
            int mid=(l+r)/2;
            if(array[mid]>array[r])
                l = mid+1;
            else
                r = mid;
        }
        return array[l];
    }
}
```

7.斐波那契数列

要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0），n<=39。

思路：菲波那切数列：F(1)=1，F(2)=1, F(n)=F(n-1)+F(n-2)（n>=3，n∈N*）

只需定义两个整型变量，b表示后面的一个数字，a表示前面的数字即可。每次进行的变换是:temp = a，a=b，b=temp + b

代码

```
public class Solution {
    public int Fibonacci(int n) {
        if(n<=0)
            return 0;
        int a=1, b=1;
        int temp;
        for(int i=2;i<n;i++){
            temp = a;
            a = b;
            b = temp+b;
        }
        return b;
    }
}

public class Solution {
    public int Fibonacci(int n) {
        if(n<=1){
            return n;
        }
        return Fibonacci(n-1)+Fibonacci(n-2);
    }
}
```

8.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级的台阶总共有多少种跳法（先后次序不同算不同的结果）。

思路：典型的动态规划问题，对于第n阶台阶来说，有两种办法，一种是爬一个台阶，到第n-1阶；第二种是爬两个台阶，到第n-2阶。

得出动态规划递推式：$F(n) = F(n-1) + F(n-2)$

代码

```
public class Solution {
    public int JumpFloor(int target) {
        if(target<=0)
            return 0;
        if(target == 1)
            return 1;
        int a=1,b=2;
        int temp;
        for(int i=3;i<=target;i++){
            temp = a;
            a = b;
            b = temp+b;
        }
        return b;
    }
}

public class Solution {
    public int JumpFloor(int n) {
        if (n == 1) return 1;
        if (n == 2) return 2;
        return JumpFloor(n - 1) + JumpFloor(n - 2);
    }
}
```

9.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。

思路： n=0时,f(n)=0；n=1时,f(n)=1；n=2时,f(n)=2；假设到了n级台阶，我们可以n-1级一步跳上来，也可以不经过n-1级跳上来，所以f(n)=2*f(n-1)。

推公式也能得出：

n = n时：f(n) = f(n-1)+f(n-2)+...+f(n-(n-1)) + f(n-n) = f(0) + f(1) + f(2) + ... + f(n-1)

由于f(n-1) = f(0)+f(1)+f(2)+ ... + f((n-1)-1) = f(0) + f(1) + f(2) + f(3) + ... + f(n-2)

所以f(n) = f(n-1)+f(n-1)=2*f(n-1)

代码

```
public class Solution {
    public int JumpFloorII(int target) {
        if(target<=0)
            return 0;
        if (target == 1) return 1;
        if (target == 2) return 2;
        int[] result=new int[target];
        result[0]=1;
        result[1]=2;
        for(int i=2;i<target;i++){
```

```
            result[i]=2*result[i-1];
        }
        return result[target-1];
    }
}
```

10.矩阵覆盖

我们可以用*21的小矩形横着或者竖着去覆盖更大的矩形。请问用n个*21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?

思路： n = 1: f(n) = 1; n=2 : f(n) = 2;

假设到了n，那么上一步就有两种情况，在n-1的时候，竖放一个矩形，或着是在n-2时，横放两个矩形（这里不能竖放两个矩形，因为放一个就变成了n-1，那样情况就重复了），所以总数是f(n)=f(n-1)+f(n-2)。时间复杂度O(n)。和跳台阶题一样。

代码

```
public class Solution {
    public int RectCover(int target) {
        if(target<=0)
            return 0;
        if(target==1)
            return 1;
        if(target==2)
            return 2;
        int[] res=new int[target];
        res[0]=1;
        res[1]=2;
        for(int i=2;i<=target-1;i++){
            res[i]=res[i-1]+res[i-2];
        }
        return res[target-1];
    }
}
```

11.二进制中1的个数

输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。例如，9表示1001，因此输入9，输出2。

思路： 如果整数不等于0，那么该整数的二进制表示中至少有1位是1。

先假设这个数的最右边一位是1，那么该数减去1后，最右边一位变成了0，其他位不变。

再假设最后一位不是1而是0，而最右边的1在第m位，那么该数减去1，第m位变成0，m右边的位变成1，m之前的位不变。

上面两种情况总结，一个整数减去1，都是把最右边的1变成0，如果它后面还有0，那么0变成1。那么我们把一个整数减去1,与该整数做位运算，相当于把最右边的1变成了0，比如1100与1011做位与运算，得到1000。那么一个整数中有多少个1就可以做多少次这样的运算。

代码

```java
public class Solution {
    public int NumberOf1(int n) {
        int count=0;
        while(n!=0){
            count +=1;
            n = (n-1)&n;
        }
        return count;
    }
}
```

12.数值的整数次方

给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。

代码：考虑base=0/exponent=0/exponent<0的情况 。

```java
public class Solution {
    public double Power(double base, int exponent) {
        if(exponent==0)
            return 1;
        if(base==0)
            return 0;
        int flag=1;
        double res=1;
        if(exponent<0){
            flag =-1;
            exponent = -exponent;
        }
        while(exponent!=0){
            res = res*base;
            exponent -= 1;
        }
        if(flag==-1){
            res = 1/res;
        }
        return res;
  }
}

public class Solution {
    public double Power(double base, int exponent) {
        if (base == 0.0){
            return 0.0;
        }
        // 前置结果设为1.0，即当exponent=0 的时候，就是这个结果
        double result = 1.0d;
        // 获取指数的绝对值
        int e = exponent > 0 ? exponent : -exponent;
        // 根据指数大小，循环累乘
        for(int i = 1 ; i <= e; i ++){
            result *= base;
        }
        // 根据指数正负，返回结果
        return exponent > 0 ? result : 1 / result;
  }
```

```
}
```

13.调整数组顺序使奇数位于偶数前面

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

```java
import java.util.ArrayList;
import java.util.List;
public class Solution {
    public void reOrderArray(int [] array) {
        List arr1=new ArrayList<Integer>();
        List arr2 = new ArrayList<Integer>();
        for(int i=0;i<array.length;i++){
            if(array[i]%2!=0)
                arr1.add(array[i]);
            else
                arr2.add(array[i]);
        }
        List list =new ArrayList<Integer>();
        list.addAll(arr1);
        list.addAll(arr2);
        for(int i=0;i<list.size();i++){
            array[i]=(Integer)list.get(i);
        }
    }
}
```

14.链表中的倒数第K个节点

输入一个链表，输出该链表中倒数第k个结点。

思路：假设链表中的节点数大于等于k个，那么一定会存在倒数第k个节点，首先使用一个快指针先往前走k步，然后两个指针每次走一步，两个指针之间始终有k的距离，当快指针走到末尾时，慢指针所在的位置就是倒数第k个节点。

代码

```java
public class Solution {
    public ListNode FindKthToTail(ListNode head,int k) {
        if(head == null)
            return null;
        ListNode fast = head;
        ListNode slow = head;
        int t=0;
        while(fast!=null && t<k){
            t += 1;
            fast = fast.next;
        }
        if(t<k)
            return null;  //考虑链表的长度<k
        while(fast != null){
            fast = fast.next;
            slow = slow.next;
        }
```

```
        return slow;
    }
}
```

15.反转链表

```
public class Solution {
    public ListNode ReverseList(ListNode head) {
        if(head==null)
            return null;
        ListNode p = head;
        ListNode q = head.next;
        while(q!=null){
            head.next = q.next;
            q.next = p;
            p = q;
            q = head.next;
        }
        return p;
    }
}
```

16.合并两个排序的链表

输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

```
public class Solution {
    public ListNode Merge(ListNode list1,ListNode list2) {
        if(list1==null)
            return list2;
        if(list2==null)
            return list1;
        ListNode head = new ListNode(-1);
        ListNode list3 = head;
        while(list1 != null && list2 != null){
            if(list1.val <= list2.val){
                head.next = list1;
                list1 = list1.next;
            }
            else{
                head.next = list2;
                list2 = list2.next;
            }
            head = head.next;
        }
        while(list1 != null){
            head.next = list1;
            list1=list1.next;
            head = head.next;
        }
        while(list2 != null){
            head.next = list2;
            list2=list2.next;
            head = head.next;
```

```
        }
        return list3.next;
    }
}
```

17.树的子结构

输入两棵二叉树A，B，判断B是不是A的子结构。（空树不是任意一个树的子结构）

思路：采用递归的思路，单独定义一个函数判断B是不是从当前A的根节点开始的子树，这里判断是不是子树也需要一个递归的判断。如果是，则返回True，如果不是，再判断B是不是从当前A的根节点的左子节点或右子节点开始的子树。

```
public class Solution {
    public boolean HasSubtree(TreeNode root1,TreeNode root2) {
        if(root1==null || root2==null)
            return false;
        boolean result = false;
        if(root1.val==root2.val)
            result = isSubtree(root1,root2);
        if(!result)
            result = HasSubtree(root1.left,root2);
        if(!result)
            result = HasSubtree(root1.right,root2);
        return result;
    }
    public boolean isSubtree(TreeNode root1,TreeNode root2){
        if(root2==null)
            return true;
        if(root1==null)
            return false;
        if(root1.val != root2.val)
            return false;
        return isSubtree(root1.left,root2.left) &&
isSubtree(root1.right,root2.right);
    }
}
```

18.二叉树的镜像

操作给定的二叉树，将其变换为源二叉树的镜像。

```
二叉树的镜像定义：源二叉树                    镜像二叉树
            8                                  8
          /  \                               /  \
         6   10                             10   6
        / \  / \                           / \  / \
       5  7 9  11                         11 9  7  5
```

代码：递归交换左右结点

```java
public class Solution {
    public void Mirror(TreeNode root) {
        if(root == null)
            return ;
        TreeNode temp;
        temp = root.left;
        root.left = root.right;
        root.right = temp;
        Mirror(root.left);
        Mirror(root.right);
    }
}
```

19.顺时针打印矩阵

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下4 X 4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.

思路：输出第一行后逆时针翻转矩阵。

代码

```java
import java.util.ArrayList;
public class Solution {
    ArrayList<Integer> arr = new ArrayList<Integer>();
    public ArrayList<Integer> printMatrix(int [][] matrix) {
        if(matrix.length == 0)
           return arr;
        int rows = matrix.length;
        int cols = matrix[0].length;
        int start = 0;
        while(rows > start * 2 && cols > start * 2){
            printOneCircle(matrix,start,rows,cols);
            start += 1;
        }
        return arr;
    }

    public void printOneCircle(int[][] matrix,int start,int rows,int cols){
        int endrow = rows - start - 1;
        int endcol = cols - start - 1;
        for(int i=start;i<=endcol;i++){
            arr.add(matrix[start][i]);
        }
        if(endrow > start)
            for(int i = start+1;i<=endrow;i++){
                arr.add(matrix[i][endcol]);
            }
        if(endrow > start && endcol > start)
            for(int i=endcol - 1;i>=start;i--){
                arr.add(matrix[endrow][i]);
            }
        if(endrow > start + 1 && endcol > start)
            for(int i = endrow - 1;i>start;i--){
                arr.add(matrix[i][start]);
            }
```

```
    }
}
```

20.包含min函数的栈

定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的min函数（时间复杂度应为O（1））。

```
import java.util.Stack;
public class Solution {
    Stack<Integer> stack = new Stack<Integer>();
    Stack<Integer> minstack = new Stack<Integer>();
    public void push(int node) {
        stack.push(node);
        if(minStack.empty() || node < minStack.peek())
            minStack.push(node);
        else
            minStack.push(minStack.peek());
    }
    public void pop() {
        if(!stack.empty()){
            stack.pop();
            minStack.pop();
        }
    }
    public int top() {
        if(!stack.empty()){
            return stack.peek();    //peek()返回栈顶元素但不弹出
        }
        else
            return -1;
    }
    public int min() {
        if(!minStack.empty())
            return minStack.peek();
        else
            return -1;
    }
}
```

21.栈的压入、弹出

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可能为该栈的弹出顺序。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）

思路：栈的压入顺序是指1,2,3,4,5是依次push到栈的，但并不是说只有push的过程，也可能有pop的操作，比如push 1，2，3，4之后，把4pop出去，然后再push5，再pop5，然后依次pop3,2,1。弹出序列是指每次pop出去的元素都是当时栈顶的元素。

那么就可以构造一个辅助栈来判断弹出序列是不是和压栈序列对应。首先遍历压栈序列的元素push到辅助栈，判断是不是弹出序列的首元素，如果是，则弹出序列pop首元素（指针后移），如果不是，则继续push，再接着判断；直到遍历完了压栈序列，如果辅助栈或者弹出序列为空，则返回True，否则返回False

代码

```
import java.util.ArrayList;
import java.util.Stack;
public class Solution {
    public boolean IsPopOrder(int [] pushA,int [] popA) {
        Stack<Integer> stack = new Stack<Integer>();
        int index = 0;
        for(int i=0;i<pushA.length;i++){
            if(pushA[i]==popA[index]){
                index += 1;
            }
            else
                stack.push(pushA[i]);
        }
        while(index<popA.length){
            if(popA[index] != stack.pop())
                return false;
            index +=1;
        }
        return true;
    }
}
```

22.从上到下打印二叉树

从上往下打印出二叉树的每个节点，同层节点从左至右打印。

```
//二叉树的层次遍历
import java.util.*;
public class Solution {
    public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        Deque<TreeNode> deque = new LinkedList<TreeNode>();
        if(root==null)
            return list;
        deque.add(root);
        while(!deque.isEmpty()){
            TreeNode t = deque.pop();
            list.add(t.val);
            if(t.left != null)
                deque.add(t.left);
            if(t.right != null)
                deque.add(t.right);
        }
        return list;
    }
}
```

23.二叉搜索树的后序遍历序列

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。

思路：递归判断。如果序列的长度小于2，那一定是后序遍历的结果。根据BST和后序遍历的性质，遍历结果的最后一个一定是根节点，那么序列中前面一部分小于根节点的数是左子树，后一部分是右子树，递归进行判断。

代码

```
public class Solution {
    public boolean VerifySquenceOfBST(int[] sequence) {
        if(sequence.length==0)
            return false;
        return isSequenceOfBST(sequence,0,sequence.length-1);
    }
    public boolean isSequenceOfBST(int[] sequence,int start,int end){
        if(end-start <2)
            return true;
        int flag = sequence[end];
        int i=start;
        while(sequence[i]<flag)
            i+=1;
        for(int j=i;j<end;j++){
            if(sequence[j]<flag)
                return false;
        }
        return isSequenceOfBST(sequence,start,i-1) &&
isSequenceOfBST(sequence,i,end-1);
    }
}
```

24.二叉树中和为某一值的路径

输入一颗二叉树的根节点和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。(注意: 在返回值的list中，数组长度大的数组靠前)

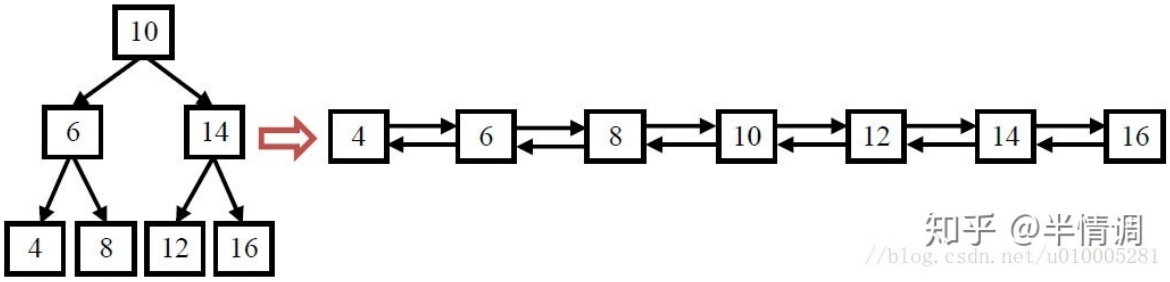
思路：定义一个子函数，输入的是当前的根节点、当前的路径以及还需要满足的数值，同时在子函数中运用回溯的方法进行判断。

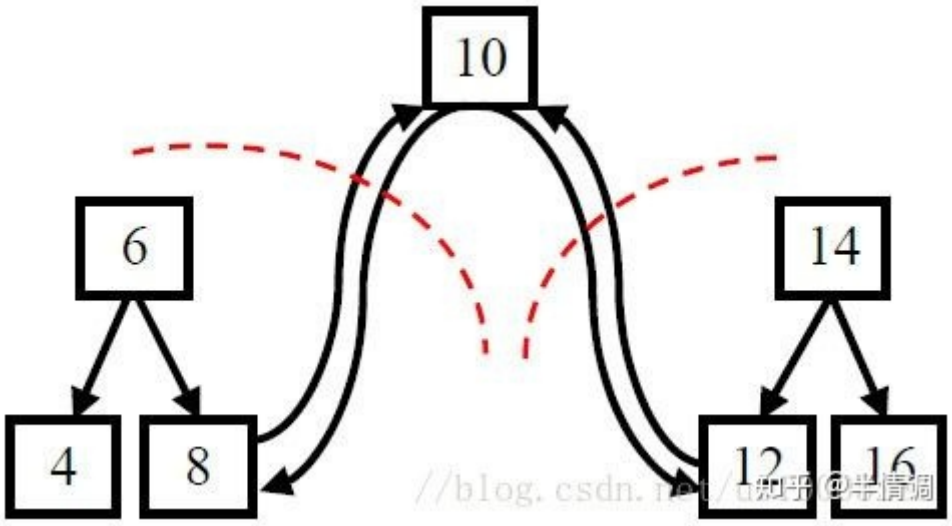
代码

```
import java.util.ArrayList;
public class Solution {
    public ArrayList<ArrayList<Integer>> res = new ArrayList<>();
    public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target)
{
         if(root==null)
            return res;
        ArrayList<Integer> arr = new ArrayList<Integer>();
        subPath(root,arr,target);
        return res;
    }
    public void subPath(TreeNode node,ArrayList<Integer> arr,int target){
        if(node.left==null && node.right==null && target==node.val){
            arr.add(node.val);
            res.add(arr);
            return;
        }
```

```
            arr.add(node.val);
            ArrayList<Integer> left = (ArrayList<Integer>)arr.clone();
            ArrayList<Integer> right = (ArrayList<Integer>)arr.clone();
            arr = null;
            if(node.left!=null){
                subPath(node.left,left,target-node.val);
            }
            if(node.right!=null){
                subPath(node.right,right,target-node.val);
            }
        }
    }
```

25.复杂链表的复制





26.二叉搜索树与双向链表

27.字符串的排列

输入一个字符串,按字典序打印出该字符串中字符的所有排列。例如输入字符串abc,则打印出由字符a,b,c所能排列出来的所有字符串abc,acb,bac,bca,cab和cba。

思路：递归。把字符串分为两个部分： 字符串的第一个字符，第一个字符后面的所有字符。1.求所有可能出现在第一个位置的字符，用索引遍历。2.求第一个字符以后的所有字符的全排列。将后面的字符又分成第一个字符以及剩余字符。

```
import java.util.ArrayList;
import java.util.TreeSet;
public class Solution {
    public ArrayList<String> Permutation(String str) {
        ArrayList<String> res = new ArrayList();
```

```
        if(str == null || str.length() == 0)
            return res;
        TreeSet<String> set = new TreeSet();
        generate(str.toCharArray(), 0, set);
        res.addAll(set);
        return res;
    }
    public void generate(char[] arr, int index, TreeSet<String> res){
        if(index == arr.length)
            res.add(new String(arr));
        for(int i = index ; i < arr.length ; i++){
            swap(arr, index, i);
            generate(arr, index + 1, res);

            swap(arr, index, i);
        }
    }
    public void swap(char[] arr, int i, int j){
        if(arr == null || arr.length == 0 || i < 0 || j > arr.length - 1)
            return;
        char tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}
```

28.数组中出现次数超过一半的数字

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。

解法1：对数组进行排序，如果该数存在，那么就是排序数组中间的数，判断这个数的个数是否大于一半，如果是，返回这个数，否则返回0。时间复杂度：O(nlogn)；空间复杂度：O(1)。

解法2：在遍历数组时保存两个值：一是数组中一个数字，一是次数。遍历下一个数字时，若它与之前保存的数字相同，则次数加1，否则次数减1；若次数为0，则保存下一个数字，并将次数置为1。遍历结束后，所保存的数字即为所求。最后验证这个数是否出现了一半以上。

```
public class Solution {
    public int MoreThanHalfNum_Solution(int [] array) {
        int res = array[0];
        int count = 1;
        for(int i=1;i<array.length;i++){
            if(res==array[i])
                count += 1;
            else
                count -= 1;
            if(count==0){
                res = array[i];
                count = 1;
            }
        }
        count = 0;
        for(int j=0;j<array.length;j++){
            if(array[j]==res)
```

```
                count += 1;
        }
        if(count>array.length/2)
            return res;
        else
            return 0;
    }
}
```

29.最小的K个数

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,。

思路：堆排序，使用PriorityQueue或者自行构建最小堆。

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;
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;
    }
}
```

30.连续子数组的最大和

给定一个整数数组 nums ， 找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

```
输入：[-2,1,-3,4,-1,2,1,-5,4],
输出：6
解释：连续子数组 [4,-1,2,1] 的和最大，为 6。
```

思路：需要两个变量，一个是*global_max,从全局来看，每次最大的是什么组合，另一个是local_max*，和*global_max*相比，更新*global_max*。

代码

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

31.从1到n的整数中1出现的个数

比如，1-13中，1出现6次，分别是1，10，11，12，13。

```
public class Solution {
    public int NumberOf1Between1AndN_Solution(int n) {
        int count = 0;
        for(int i=0;i<=n;i++){
            int a=i;
            while(a>0){
                if(a%10==1)
                    count +=1;
                a=a/10;
            }
        }
        return count;
    }
}
```

32.把数组排成最小的数

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。

思路： 根据题目的要求，两个数字m和n能拼接称数字mn和nm。如果mn<nm，也就是m应该拍在n的前面，我们定义此时m小于n；反之，如果nm<mn，我们定义n小于m。如果mn=nm,m等于n。

```
import java.util.ArrayList;

public class Solution {
    public String PrintMinNumber(int [] numbers) {
        int n;
        StringBuilder s = new StringBuilder();
```

```
        ArrayList<Integer> list = new ArrayList<>();
        n = numbers.length;
        for (int i = 0; i < n; i++) {
            list.add(numbers[i]);
        }
        list.sort((str1, str2) -> {
            String s1 = str1 + "" + str2;
            String s2 = str2 + "" + str1;
            return s1.compareTo(s2);
        });
        list.forEach(s::append);
        return s.toString();
    }
}
```

31.丑数

把只包含质因子2、3和5的数称作丑数。例如6、8都是丑数，但14不是，因为它包含质因子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。

思路：动态规划的解法。 一个丑数一定由另一个丑数乘以2或者乘以3或者乘以5得到，那么我们从1开始乘以2,3,5，就得到2,3,5三个丑数，在从这三个丑数出发乘以2,3,5就得到4,6,10; 6,9,15;10,15,25九个丑数，这种方法会得到重复且无序的丑数，而且我们题目要求第N个丑数，这样的方法得到的丑数也是无序的，我们可以维护三个索引。

代码：

```
import java.util.ArrayList;
public class Solution {
    public int GetUglyNumber_Solution(int index) {
        if(index<=0)
            return 0;
        ArrayList<Integer> arr = new ArrayList<Integer>();
        arr.add(1);
        int a = 0;
        int b = 0;
        int c = 0;
        int nextMin = 1;
        for(int i=2;i<=index;i++){
            nextMin =
Math.min(Math.min(arr.get(a)*2,arr.get(b)*3),arr.get(c)*5);
            arr.add(nextMin);
            if(nextMin>=arr.get(a)*2)
                a += 1;
            if(nextMin>=arr.get(b)*3)
                b += 1;
            if(nextMin>=arr.get(c)*5)
                c += 1;
        }
        return nextMin;
    }
}
```

32.第一个只出现一次的字符

在一个字符串(0<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符,并返回它的位置, 如果没有则返回 -1（需要区分大小写）。

思路：创建哈希表，下标为ACII值，值为出现次数。

代码

```
import java.util.*;
public class Solution {
    public int FirstNotRepeatingChar(String str) {
        if(str.length()==0)
            return -1;
        HashMap<Character,Integer> map=new HashMap<Character,Integer>();
        for(int i=0;i<str.length();i++)
        {
            char c=str.charAt(i);
            if(map.containsKey(c))
            {
                int time=map.get(c);
                time++;
                map.put(c,time);
            }
            else
                map.put(c,1);
        }
       for(int i=0;i<str.length();i++)
       {
          char c=str.charAt(i);
          if(map.get(c)==1)
              return i;
       }
       return -1;
    }
}
```

33.数组中的逆序对

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P。例如{7,5,6,4}，存在5个逆序对，分别是(7,5),(7,6),(7,4),(6,4),(5,4)。并将P对1000000007取模的结果输出。 即输出P%1000000007

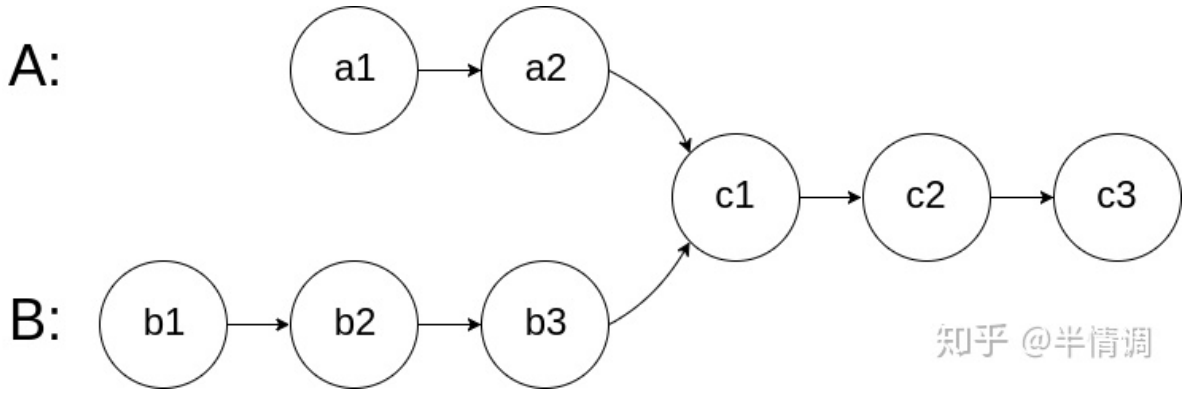
```
//使用归并排序的思路求解
public class Solution {
    public int InversePairs(int [] array) {
        if(array==null || array.length==0)
            return 0;
        int[] copy = new int[array.length];
        for(int i=0;i<array.length;i++){
            copy[i]=array[i];
        }
        int count = InversePairsCore(array, copy, 0, array.length-1);
        return count;
    }
    private  static 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;
    }
}
```

34.两个链表的第一个公共结点

(leetcode160) 编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

注意：

- 如果两个链表没有交点，返回 `null`.
- 在返回结果后，两个链表仍须保持原有的结构。
- 可假定整个链表结构中没有循环。
- 程序尽量满足 O(*n*) 时间复杂度，且仅用 O(*1*) 内存。

分析

设置两个指针，一个从headA开始遍历，遍历完headA再遍历headB，另一个从headB开始遍历，遍历完headB再遍历headA，如果有交点，两个指针会同时遍历到交点处。

代码

```
public class Solution {
    public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {
        if(pHead1==null || pHead2==null)
            return null;
        ListNode p1 = pHead1;
        ListNode p2 = pHead2;
        while(p1 != p2){
            if(p1 != null)
                p1 = p1.next;
            else
                p1 = pHead2;
            if(p2 != null)
                p2 = p2.next;
            else
                p2 = pHead1;
        }
        return p1;
    }
}
```

35.统计一个数字在排序数组中的出现的次数

思路：考虑数组为空的情况，直接返回0；用二分查找法，找到i和j的位置。

```
public class Solution {
    public int GetNumberOfK(int [] array , int k) {
        if(array.length == 0)
            return 0;
        int i = 0;
        int j = array.length-1;
```

```
        while(i<j && array[i] != array[j]){
            if(array[i]<k)
                i++;
            if(array[j]>k)
                j--;
        }
        if(array[i] != k)
            return 0;
        return j-i+1;
    }
}
```

36.二叉树的深度

(同leetcode104)输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

示例：
给定二叉树 `[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

思路

递归的方法，比较左边路径和右边路径哪边最长，选择最长的一边路径，加上root结点本身的长度。

```
public class Solution {
    public int TreeDepth(TreeNode root) {
        if(root == null)
            return 0;
        return Math.max(TreeDepth(root.left),TreeDepth(root.right))+1;
    }
}
```

37.平衡二叉树

（同leetcode110）输入一个二叉树，判断是否是平衡二叉树。

> 平衡二叉树：一个二叉树*每个节点*的左右两个子树的高度差的绝对值不超过1。

示例：

给定二叉树 `[3,9,20,null,null,15,7]`

```
    3
   / \
  9  20
    /  \
   15   7
```

返回 `true` 。

思路

利用104题中判断二叉树最大深度的函数，左子树和右子树的深度差小于等于1即为平衡二叉树。

```
public class Solution {
    public boolean IsBalanced_Solution(TreeNode root) {
        if(root==null)
            return true;
        if(Math.abs(height(root.left)-height(root.right))>1)
            return false;
        else
            return IsBalanced_Solution(root.left) &&
IsBalanced_Solution(root.right);
    }
    public int height(TreeNode root){
        if(root == null)
            return 0;
        return Math.max(height(root.left),height(root.right))+1;
    }
}

public boolean IsBalanced_Solution(TreeNode root) {
        if(root==null){
            return true;
        }
        if(Math.abs(deep(root.left)-deep(root.right))!=1){
            return false;
        }else{
            return
IsBalanced_Solution(root.left)&&IsBalanced_Solution(root.right);
        }
    }
    public int deep(TreeNode root){
        if(root==null){
            return 0;
        }
        return Math.max(deep(root.left),deep(root.right))+1;
    }
```

38.数组中只出现一次的数字

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。

思路：如果数组中只有一个数字出现了一次，对数组所有数求一次异或，两个相同的数的异或是0。那么如果数组中有两个数出现了一次，其他出现了两次，将这数组分成两个子数组，这两个数字分别出现在这两个子数组中，那么就转换成了前面所说的求异或的问题。那么怎么分呢，这里的思路是根据要求的这两个数的异或之后最右边不为1的这一位进行划分的。

```
//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[]) {
        int res = 0;
```

```
        for(int x:array)
            res ^= x;
        int splitBit = 1;
        while((res & splitBit)==0)
            splitBit = splitBit << 1;
        int res1 = 0;
        int res2 = 0;
        for(int x:array){
            if((x & splitBit) != 0)
                res1 ^= x;
            else
                res2 ^= x;
        }
        num1[0] = res1;
        num2[0] = res2;
    }
}
```

39.和为S的连续正数序列

输出所有和为S的连续正数序列。序列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序。例如连续正数和为100的序列:18,19,20,21,22。

思路：判断每一个起始位置，然后往后遍历，如果和大于目标的话，就进行下一次循环，如果等于目标，将arraylist添加到返回结果。另外，连续的数肯定小于等于和的一半。

```
import java.util.ArrayList;
public class Solution {
    public ArrayList<ArrayList<Integer> > FindContinuousSequence(int sum) {
        ArrayList<ArrayList<Integer>> aList=new
ArrayList<ArrayList<Integer>>();
        if(sum<2)
            return aList;
        for(int i=1;i<=sum/2;i++){
            ArrayList<Integer> aList2=new ArrayList<Integer>();
            int count=0;
            for(int j=i;j<sum;j++){
                count+=j;
                aList2.add(j);
                if(count>sum)
                    break;
                else if(count==sum){
                    aList.add(aList2);
                    break;
                }
            }
        }
        return aList;
    }
}
```

40.和为S的两个数字

输入一个递增排序的数组和一个数字S，在数组中查找两个数，使得他们的和正好是S，如果有多对数字的和等于S，输出两个数的乘积最小的。

思路：由于是排好序的数组，因此对于和相等的两个数来说，相互之间的差别越大，那么乘积越小，因此我们使用两个指针，一个从前往后遍历，另一个从后往前遍历数组即可。

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

41.左旋转字符串

对于一个给定的字符序列S，请你把其循环左移K位后的序列输出。例如，字符序列S=”abcXYZdef”,要求输出循环左移3位后的结果，即“XYZdefabc”。

思路：分割法。

```
public class Solution {
    public String LeftRotateString(String str,int n) {
        if(str==null || str.length()==0)
            return "";
        String str1=str.substring(0,n);
        String str2=str.substring(n,str.length());
        return str2+str1;
    }
}
```

42.翻转单词顺序列

例如，“student. a am I”翻转为“I am a student.”。

思路：按空格切分为数组，从尾到头遍历数组，依次拼接起来。

```
public class Solution {
    public String ReverseSentence(String str) {
        if(str.trim().length() <= 0){
            return str;
        }
        String[] strArr = str.split(" ");
        String res = "";
        for(int i=strArr.length-1;i>=0;i--){
```

```
            if(i != 0){
                res += strArr[i] + " ";
            }else{
                res += strArr[i];
            }
        }
        return res;
    }
}
```

43.扑克牌顺子

一副扑克牌,里面有2个大王，2个小王，从中随机抽出5张牌，如果牌能组成顺子就输出true，否则就输出false。为了方便起见，大小王是0，大小王可以当作任何数字。

思路:

1、将数组排序； 2、统计数组中0的个数，即判断大小王的个数；3、统计数组中相邻数字之间的空缺总数，如果空缺数小于等于大小王的个数，可以组成顺子，否则不行。如果数组中出现了对子，那么一定是不可以组成顺子的。

代码:

```
import java.util.Arrays;
public class Solution {
    public boolean isContinuous(int [] numbers) {
        int m = numbers.length;
        if(m==0)
            return false;
        Arrays.sort(numbers);
        int count = 0;
        for(int i=0;i<m;i++){
            if(numbers[i]==0)
                count += 1;
        }
        for(int i=count;i<m-1;i++){
            if(numbers[i+1]==numbers[i])
                return false;
            else if((numbers[i+1]-numbers[i]-1)>count)
                return false;
            else
                count -= (numbers[i+1]-numbers[i]-1);
        }
        return true;
    }
}
```

44.孩子们的游戏（圆圈中最后剩下的数）

游戏是这样的：首先，让小朋友们围成一个大圈。然后，他随机指定一个数m，让编号为0的小朋友开始报数。每次喊到m-1的那个小朋友要出列，不再回到圈中，从他的下一个小朋友开始，继续0...m-1报数....这样下去....直到剩下最后一个小朋友获胜，获胜的小朋友编号多少？(注：小朋友的编号是从0到n-1)

分析：约瑟夫环问题，可以用数组模拟，但需要维护是否出列的状态。使用LinkedList模拟一个环cycle，出列时删除对应的位置。 1. 报数起点为start（初始为0），则出列的位置为out = (start + m - 1) % cycle.size()，删除out位置； 2. 更新起点start = out，重复1直到只剩下一个元素。 时间复杂性、空间复杂度均为O(n)。

```
import java.util.LinkedList;
public class Solution {
    public int LastRemaining_Solution(int n, int m) {
        if (n < 1 || m < 1) {
            return -1;
        }
        LinkedList<Integer> cycle = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            cycle.add(i);
        }
        int start = 0;
        while (cycle.size() > 1) {
            int out = (start + m - 1) % cycle.size();
            cycle.remove(out);
            start = out;
        }
        return cycle.remove();
    }
}
```

45.求1+2+3+...+n

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）。

思路：将加法问题转化为递归进行求解即可。

代码：

```
public class Solution {
    public int sum = 0;
    public int Sum_Solution(int n) {
        getSum(n);
        return sum;
    }
    private void getSum(int n){
        if(n<=0)
            return;
        sum += n;
        getSum(n-1);
    }
}
```

46.不用加减乘除做加法

写一个函数，求两个整数之和，要求在函数体内不得使用+、-、*、/四则运算符号。

思路：

对数字做运算，除了加减乘除外，还有**位运算**，位运算是针对二进制的，二进制的运算有“三步走”策略：

例如5的二进制是101，17的二进制10001。
第一步：各位相加但不计进位，得到的结果是10100。
第二步：计算进位值，只在最后一位相加时产生一个进位，结果是二进制10。
第三步：把前两步的结果相加，得到的结果是10110。转换成十进制正好是22。

接着把二进制的加法用位运算替代：
（1）不考虑进位对每一位相加，0加0、1加1的结果都是0，1加0、0加1的结果都是1。这和异或运算相同。（2）考虑进位，只有1加1的时候产生进位。 位与运算只有两个数都是1的时候结果为1。考虑成两个数都做位与运算，然后向左移一位。（3）相加的过程依然重复前面两步，直到不产生进位为止。

```
public class Solution {
    public int Add(int num1,int num2) {
         while (num2!=0) {
            int temp = num1^num2;
            num2 = (num1&num2)<<1;
            num1 = temp;
        }
        return num1;
    }
}
```

47.把字符串转换成整数

将一个字符串转换成一个整数(实现Integer.valueOf(string)的功能，但是string不符合数字要求时返回0)，要求不能使用字符串转换整数的库函数。 数值为0或者字符串不是一个合法的数值则返回0。

示例1：

```
输入：+2147483647 ，输出：2147483647；输入：1a33，输出：0。
```

思路：考虑溢出。

```
public class Solution {
    public int StrToInt(String str) {
        if (str == null)
            return 0;
        int result = 0;
        boolean negative = false;//是否负数
        int i = 0, len = str.length();
        /**
         * limit 默认初始化为 负的 最大正整数 ，假如字符串表示的是正数
         * 那么result(在返回之前一直是负数形式)就必须和这个最大正数的负数来比较，
         * 判断是否溢出
         */
        int limit = -Integer.MAX_VALUE;
        int multmin;
        int digit;
        if (len > 0) {
            char firstChar = str.charAt(0);//首先看第一位
            if (firstChar < '0') { // Possible leading "+" or "-"
                if (firstChar == '-') {
                    negative = true;
                    limit = Integer.MIN_VALUE;//在负号的情况下，判断溢出的值就变成了 整数的 最小负数了
                } else if (firstChar != '+')//第一位不是数字和-只能是+
```

```
                    return 0;
                if (len == 1) // Cannot have lone "+" or "-"
                    return 0;
                i++;
            }
            multmin = limit / 10;
            while (i < len) {
                // Accumulating negatively avoids surprises near MAX_VALUE
                digit = str.charAt(i++)-'0';//char转int
                if (digit < 0 || digit > 9)//0到9以外的数字
                    return 0;
                //判断溢出
                if (result < multmin) {
                    return 0;
                }
                result *= 10;
                if (result < limit + digit) {
                    return 0;
                }
                result -= digit;
            }
        } else {
            return 0;
        }
        //如果是正数就返回-result（result一直是负数）
        return negative ? result : -result;
    }
}
```

48.数组中重复的数字

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}，那么对应的输出是第一个重复的数字2。

思路：一个简单的方法是先排序再查找，时间复杂度是O(nlogn)。还可以用哈希表来解决，遍历每个数字，每扫描到一个数字可以用O(1)的时间来判断哈希表中是否包含了这个数字，如果没有包含，则加到哈希表，如果包含了，就找到了一个重复的数字。时间复杂度O(n)。

我们注意到数组中的数字都在0~n-1范围内，如果这个数组中没有重复的数字，那么当数组排序后数字i在下标i的位置，由于数组中有重复的数字，有些位置可能存在多个数字，同时有些位置可能没有数字。遍历数组，当扫描到下标为i 的数字m时，首先看这个数字是否等于i，如果是，继续扫描，如果不是，拿它和第m个数字进行比较。如果它和第m个数字相等，就找到了一个重复的数字，如果不相等，就交换两个数字。继续比较。

```
 // 这里要特别注意~找到任意重复的一个值并赋值到duplication[0]
 // 函数返回true/false
public class Solution {
    public boolean duplicate(int numbers[],int length,int[] duplication) {
        for(int i=0;i<length;i++){
            while(numbers[i]!=i){
                int m=numbers[i];
                if(numbers[m]==numbers[i]){
                    duplication[0]=m;
                    return true;
                }
```

```
                else{
                    numbers[i]=numbers[m];
                    numbers[m]=m;
                }
            }
        }
        return false;
    }
}
```

49.构建乘积数组

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

思路：如果没有不能使用除法的限制，可以直接用累乘的结果除以A[i]。由于题目有限制，一种直观的解法是连乘n-1个数字，但时间复杂度是O(n^2)。可以把B[i]=A[0]*A[1]...A[i-1]*A[i+1]...A[n-1]分成A[0]*A[1]...A[i-1]和A[i+1]...*A[n-1]两部分的乘积。

```
public class Solution {
    public int[] multiply(int[] A) {
        int length = A.length;
        int[] B = new int[length];
        if(length != 0 ){
            B[0] = 1;
            //计算下三角连乘
            for(int i = 1; i < length; i++){
                B[i] = B[i-1] * A[i-1];
            }
            int temp = 1;
            //计算上三角
            for(int j = length-2; j >= 0; j--){
                temp *= A[j+1];
                B[j] *= temp;
            }
        }
        return B;
    }
}
```

50.正则表达式匹配

请实现一个函数用来匹配包括'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符，而'*'表示它前面的字符可以出现任意次（包含0次）。 在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但是与"aa.a"和"ab*a"均不匹配。

思路：如果 s和pattern都为空，匹配成功。

当模式中的第二个字符不是“*”时：

1、如果字符串第一个字符和模式中的第一个字符相匹配，那么字符串和模式都后移一个字符，然后匹配剩余的。

2、如果字符串第一个字符和模式中的第一个字符相不匹配，直接返回false。

而当模式中的第二个字符是“*”时：

如果字符串第一个字符跟模式第一个字符不匹配，则模式后移2个字符，继续匹配。如果字符串第一个字符跟模式第一个字符匹配，可以有3种匹配方式：

1、模式后移2字符，相当于x*被忽略；

2、字符串后移1字符，模式后移2字符；

3、字符串后移1字符，模式不变，即继续匹配字符下一位，因为*可以匹配多位；

```
public class Solution {
    public boolean match(char[] str, char[] pattern) {
    if (str == null || pattern == null)
        return false;
    return matchCore(str, 0, pattern, 0);
}
    public  boolean matchCore(char[] str,int i,char[] pattern,int j) {
        //str到尾，pattern到尾，匹配成功
        if(str.length == i && j == pattern.length)
            return true;
        //pattern先到尾，匹配失败
        if(str.length != i && j == pattern.length)
            return false;
       //注意数组越界问题，一下情况都保证数组不越界
       if(j < pattern.length-1 && pattern[j+1] == '*') {//下一个是*
           if(str.length != i && (str[i] == pattern[j] || pattern[j] == '.')) //匹配
               return matchCore(str,i,pattern,j+2)||matchCore(str,i+1,pattern,j);
           else//当前不匹配
               return matchCore(str,i,pattern,j + 2);
       }
       //下一个不是“*”，当前匹配
       if(str.length != i && (str[i] == pattern[j] || pattern[j] == '.'))
           return matchCore(str,i + 1,pattern,j + 1);
        return false;
    }
}
```

51.表示数值的字符串

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100","5e2","-123","3.1416"和"-1E-16"都表示数值。 但是"12e","1a3.14","1.2.3","+-5"和"12e+4.3"都不是。

思路：数字的格式可以用A[.[B]][E|eC]或者.B[E|eC]表示，其中A和C都是整数（可以有符号也可以没有），B是一个无符号数。

如果遍历到e或E，那么之前不能有e或E，并且e或E不能在末尾；

如果遍历到小数点,那么之前不能有小数点，并且之前不能有e或E；

如果遍历到正负号，那么如果之前有正负号，只能够出现在e或E的后面，如果之前没符号，那么符号只能出现在第一位，或者出现在e或E的后面；

如果遍历到不是上面所有的符号和0~9，返回False。

代码：

```java
public class Solution {
    public boolean isNumeric(char[] str) {
        boolean hasE = false;
        boolean hasDot = false;
        boolean hasSign = false;
        for(int i=0;i<str.length;i++){
            if(str[i]=='E' || str[i]=='e'){
                if(hasE || i==str.length-1)
                    return false;
                hasE = true;
            }
            else if(str[i]=='.'){
                if(hasE || hasDot)
                    return false;
                hasDot = true;
            }
            else if(str[i]=='+' || str[i]=='-'){
                if(hasSign && (str[i-1]!='E' || str[i-1]!='e'))
                    return false;
                if(!hasSign && i!=0 && str[i-1]!='E' && str[i-1]!='e')
                    return false;
            }
            else{
                if(str[i]<'0' || str[i]>'9')
                    return false;
            }
        }
        return true;
    }
}


import java.util.regex.Pattern;

public class Solution {
    public static boolean isNumeric(char[] str) {
        String pattern = "^[-+]?\\d*(?:\\.\\d*)?(?:[eE][+\\-]?\\d+)?$";
        String s = new String(str);
        return Pattern.matches(pattern,s);
    }
}
```

52.字符流中第一个不重复的字符

请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符"go"时，第一个只出现一次的字符是"g"。当从该字符流中读出前六个字符“google"时，第一个只出现一次的字符是"l"。

```
如果当前字符流没有存在出现一次的字符，返回#字符。
```

思路：用一个字典保存下出现过的字符，以及字符出现的次数。

除保存出现的字符之外，我们用一个字符数组保存出现过程字符顺序，如果不保存插入的char的话，我们可以遍历ascii码中的字符。

代码：

```java
import java.util.*;
public class Solution {
    public ArrayList<Character> charlist = new ArrayList<Character>();
    public HashMap<Character,Integer> map = new HashMap<Character,Integer>
();
    public void Insert(char ch)
    {
        if(map.containsKey(ch))
            map.put(ch,map.get(ch)+1);
        else
            map.put(ch,1);
        charlist.add(ch);
    }
    public char FirstAppearingOnce()
    {
        char c='#';
        for(char key : charlist){
            if(map.get(key)==1){
                c=key;
                break;
            }
        }
        return c;
    }
}
```

53.链表中环的入口节点

给一个链表，若其中包含环，请找出该链表的环的入口结点，否则，输出null。

思路：快慢指针，快指针一次走两步，慢指针一次走一步。如果链表中存在环，且环中假设有n个节点，那么当两个指针相遇时，快的指针刚好比慢的指针多走了环中节点的个数，即n步。从另一个角度想，快的指针比慢的指针多走了慢的指针走过的步数，也是n步。相遇后，快指针再从头开始走，快慢指针再次相遇时，所指位置就是入口。

https://cyc2018.github.io/CS-Notes/#/notes/23.%20%E9%93%BE%E8%A1%A8%E4%B8%AD%E7%8E%AF%E7%9A%84%E5%85%A5%E5%8F%A3%E7%BB%93%E7%82%B9

代码：

```java
public class Solution {
    public ListNode EntryNodeOfLoop(ListNode pHead)
    {
        if(pHead==null || pHead.next==null || pHead.next.next==null)
            return null;
        ListNode fast=pHead.next.next;
        ListNode slow = pHead.next;
        while(fast != slow){
            if(fast.next==null || fast.next.next==null)
                return null;
            fast = fast.next.next;
            slow = slow.next;
        }
        fast = pHead;
        while(fast != slow){
            fast = fast.next;
```

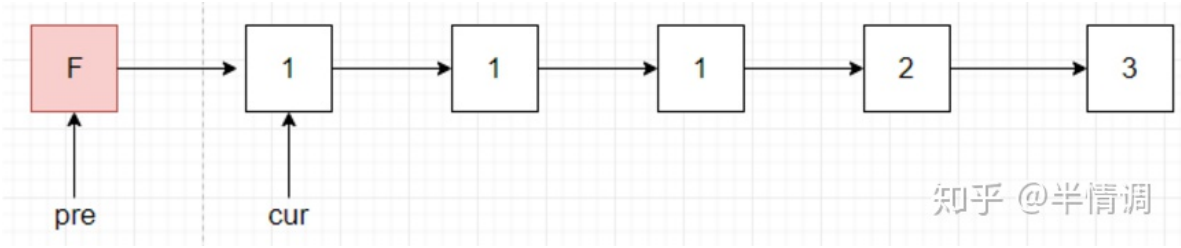
```
            slow = slow.next;
        }
        return fast;
    }
}
```

54.删除链表中重复的结点

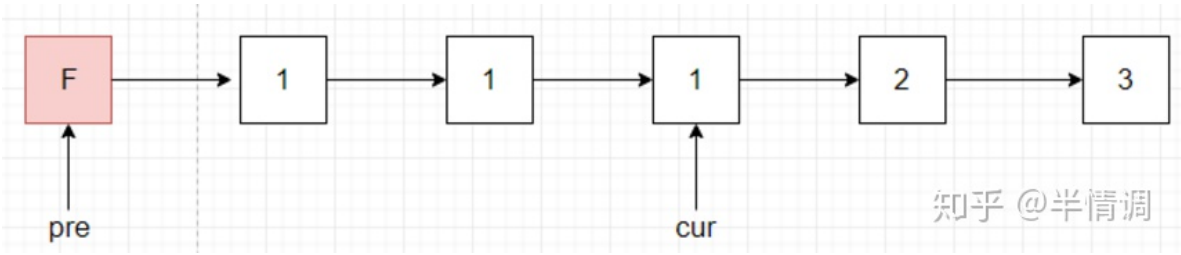
在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。 例如，链表1->2->3->3->4->4->5 处理后为 1->2->5。(leetcode82)

思路

1.设置一个虚拟头结点，设置两个指针，pre指向虚拟头结点，cur指向头结点。

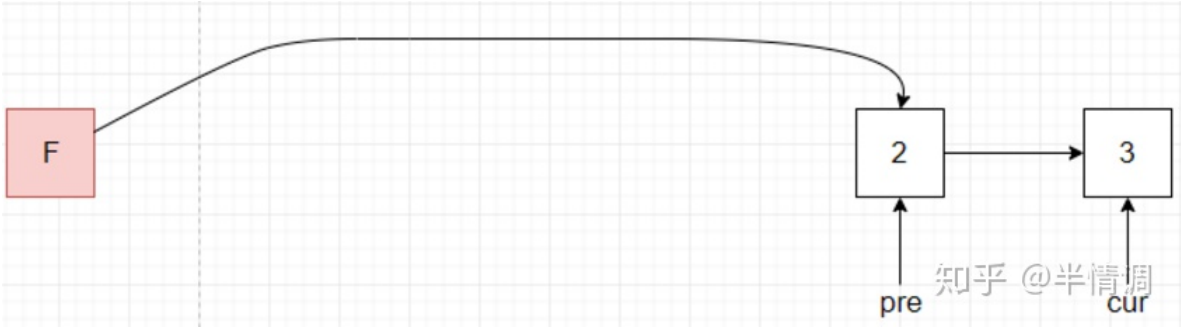


2.判断下一个节点的值和cur的值是否相等，若相等cur后移，直到下个节点的值和cur的值不同。



3.此时执行pre.next= cur.next。

4.继续走直到结尾.



代码

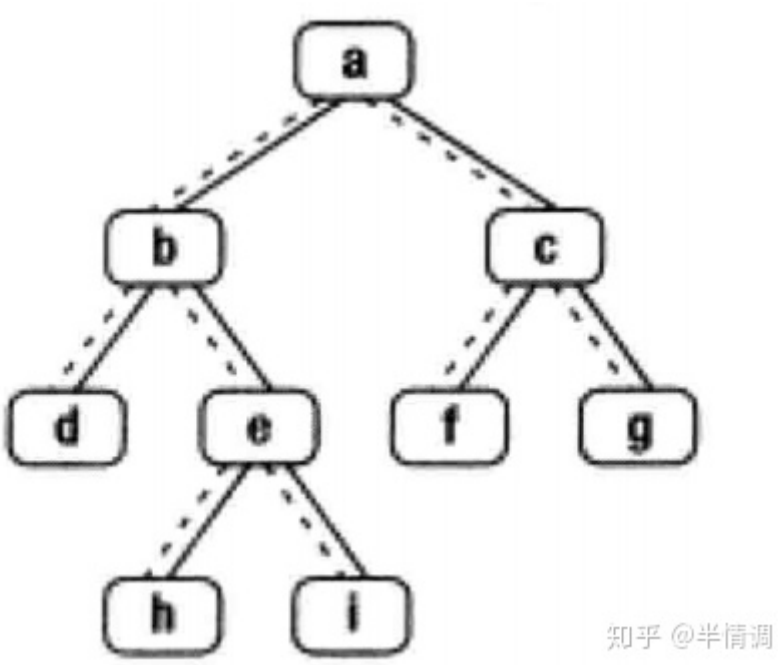
```
public class Solution {
```

```
    public ListNode deleteDuplication(ListNode pHead)
    {
        ListNode dummy = new ListNode(0);
        dummy.next = pHead;
        ListNode pre = dummy;
        ListNode cur = pHead;
        while(cur != null){
            while(cur.next != null && cur.next.val==cur.val)
                cur = cur.next;
            if(pre.next==cur)
                pre=pre.next;
            else
                pre.next = cur.next;
            cur = cur.next;
        }
        return dummy.next;
    }
}
```

55.二叉树的下一个结点

给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。

思路：如下图所示，二叉树的中序遍历序列是{d,b,h,e,i,a,f,c,g}。



1、如果该节点有右子树，那么它的下一个节点就是它的右子树的最左侧子节点；

2、如果该节点没有右子树且是父节点的左子树，那么下一节点就是父节点；

3、如果该节点没有右子树且是父节点的右子树，比如i节点，那么我们往上找父节点，找到一个节点满足： 它是它的父节点的左子树的节点。

```
public class Solution {
    public TreeLinkNode GetNext(TreeLinkNode pNode) {
        if (pNode == null)
            return pNode;
        if (pNode.right != null) { // 节点有右子树
```

```
            pNode = pNode.right;
            while (pNode.left != null) {
                pNode = pNode.left;
            }
            return pNode;
        }
        // 节点无右子树且该节点为父节点的左子节点
        else if ( pNode.next != null && pNode.next.left == pNode) {
            return pNode.next;
        }
        // 节点无右子树且该节点为父节点的右子点
        else if (pNode.next != null && pNode.next .right == pNode) {
            while(pNode.next != null && pNode .next .left != pNode){
                pNode = pNode.next ;
            }
            return pNode.next ;
        }
        else
            return pNode.next ;//节点无父节点 ，即节点为根节点
    }
}
```

56.对称的二叉树

请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。（leetcode101题）

例如，二叉树 `[1,2,2,3,4,4,3]` 是对称的。

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

思路

递归的思想，首先判断头结点是否为空。然后将根节点的左右两个节点假设成两个独立的树，如果左右两个树都为空，返回True。然后看左子树的左结点和右子树的右结点、左子树的右结点和右子树的左结点是否相同，都相同返回True.

代码

```
public class Solution {
    boolean isSymmetrical(TreeNode pRoot)
    {
        if(pRoot==null)
            return true;
        return isSymmetricalTree(pRoot.left,pRoot.right);
    }
    private boolean isSymmetricalTree(TreeNode left,TreeNode right){
        if(left==null && right==null)
            return true;
        else if(left==null || right==null)
            return false;
        else if(left.val != right.val)
```

```
            return false;
        else{
            return isSymmetricalTree(left.left,right.right)
                && isSymmetricalTree(left.right,right.left);
        }
    }
}
```

57.把二叉树打印成多行

从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。（leetcode102题）

给定二叉树: `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其层次遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

思路

用队列实现，root为空，返回空；队列不为空，记下此时队列中的节点个数end，end个节点出队列的同时，记录节点值，并把节点的左右子节点加入队列中。

代码

```
import java.util.*;
public class Solution {
    ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {
        ArrayList<ArrayList<Integer>> result = new
ArrayList<ArrayList<Integer>>();
        if(pRoot == null){
            return result;
        }
        Queue<TreeNode> layer = new LinkedList<TreeNode>();
        ArrayList<Integer> layerList = new ArrayList<Integer>();
        layer.add(pRoot);
        int start = 0, end = 1;
        while(!layer.isEmpty()){
            TreeNode cur = layer.remove();
            layerList.add(cur.val);
            start++;
            if(cur.left!=null){
                layer.add(cur.left);
            }
            if(cur.right!=null){
                layer.add(cur.right);
```

```
            }
            if(start == end){
                end = layer.size();
                start = 0;
                result.add(layerList);
                layerList = new ArrayList<Integer>();
            }
        }
        return result;
    }
}
```

58.按之字形顺序打印二叉树

请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。

例如：
给定二叉树 `[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回锯齿形层次遍历如下：

```
[
  [3],
  [20,9],
  [15,7]
]
```

思路

用两个栈实现，栈s1与栈s2交替入栈出栈。reverse方法时间复杂度比较高，两个栈以空间换时间。

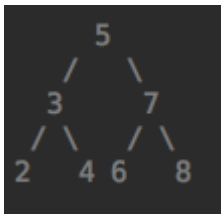
代码

```
public class Solution {
    public ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {
        ArrayList<ArrayList<Integer> > listAll = new ArrayList<>();
        if(pRoot==null)return listAll;
        Stack<TreeNode> s1 = new Stack<>();
        Stack<TreeNode> s2 = new Stack<>();
        int level = 1;
        s1.push(pRoot);
        while(!s1.isEmpty()||!s2.isEmpty()){
            ArrayList<Integer> list = new ArrayList<>();
            if(level++%2!=0){
                while(!s1.isEmpty()){
                    TreeNode node = s1.pop();
                    list.add(node.val);
                    if(node.left!=null)s2.push(node.left);
                    if(node.right!=null)s2.push(node.right);
```

```
                }
            }
            else{
                while(!s2.isEmpty()){
                    TreeNode node = s2.pop();
                    list.add(node.val);
                    if(node.right!=null)s1.push(node.right);
                    if(node.left!=null)s1.push(node.left);
                }
            }
            listAll.add(list);
        }
        return listAll;
    }
}
```

59.二叉搜索树的第K个节点

给定一棵二叉搜索树，请找出其中的第k小的结点。例如，（5，3，7，2，4，6，8）中，按结点数值大小顺序第三小结点的值为4。



思路：如果是按中序遍历二叉搜索树的话，遍历的结果是递增排序的。所以只需要中序遍历就很容易找到第K个节点。

```
public class Solution {
    int count=0;
    TreeNode KthNode(TreeNode pRoot, int k)
    {
        if(pRoot==null || k<=0)
            return null;
        TreeNode res = KthNode(pRoot.left, k);
        if(res!=null)
            return res;
        count++;
        if(count==k)
            return pRoot;
        res = KthNode(pRoot.right, k);
        return res;
    }
}
```

60.滑动窗口的最大值

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]}。

思路：双向队列，queue存入num的位置，时间复杂度O(n)

我们用双向队列可以在O(N)时间内解决这题。当我们遇到新的数时，将新的数和双向队列的末尾比较，如果末尾比新数小，则把末尾扔掉，直到该队列的末尾比新数大或者队列为空的时候才住手。这样，我们可以保证队列里的元素是从头到尾降序的，由于队列里只有窗口内的数.因此一个新数进来：1、判断队列头部的数的下标是否还在窗口中；2、将新数加入到队列；3、如果index已经达到窗口的大小了，则将队列头部的值加入到返回结果中

```
import java.util.*;
public class Solution {
    public ArrayList<Integer> maxInWindows(int [] num, int size)
    {
        ArrayList<Integer> res = new ArrayList<Integer>();
        if(size <= 0)
            return res;
        Deque<Integer> queue = new ArrayDeque<Integer>();
        for(int i=0;i<num.length;i++){
            if(queue.size()>0 && (i - queue.getFirst()) >= size)
                queue.removeFirst();
            while(queue.size()>0 && num[i] > num[queue.getLast()])
                queue.removeLast();
            queue.addLast(i);
            if(i+1>=size)
                res.add(num[queue.getFirst()]);
        }
        return res;
    }
}
```