牛客在线编程试题详解

剑指offer--Java篇

牛客资料库出品
nowcoder.com

目录

剑指 offer 题目题解：Java 篇

在线编程地址：https://www.nowcoder.com/ta/coding-interviews?from=EDjob

1.二维数组的查找

题目描述

在一个二维数组中，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。

思路分析

二维数组的查找是一个经典的关于数组的算法题目，主要是利用数组的特性进行查找。在这里我们采用从右上角开始查找的思路：即从右上角开始，如果当前的数字比目标数字大，则列减 1，否则行加 1。查找结束的时候如果还未返回，即可认为不存在。

代码实现

```
public class Solution {

    public boolean Find(int[][] array, int target) {

        //定义多维数组的行数
        int row = 0;

        //定义多维数组的列数
        int col = array[0].length - 1;

        // 通过 while 循环控制
        while (row < array.length && col >= 0) {
            if (array[row][col] > target)
                col--;
            else if (array[row][col] < target)
                row++;
            else
                return true;
        }
        return false;
    }
```

```
    public static void main(String[] args) {
        Solution s = new Solution();
        int[][] a = {{1, 2, 3}, {4, 5, 6}};
        //测试用例
        for (int i = 0; i < 10; i++) {
            System.out.println(s.Find(a, i));
        }
    }
}
```

2. 替换空格

题目描述

请实现一个函数，将一个字符串中的空格替换成“%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。

思路分析

字符串中空格替换有两种方式，一种是遍历一遍，将源字符串直接复制到新字符串，遇到空格替换之即可，这种方法会增加空间复杂度。另一种方法是，先统计源字符串中空格数，获得替换后字符串的长度，再从后向前进行替换。

```
class Solution {
public:
void replaceSpace(char *str,int length) {
if(str == ""){
 return;
 }
 //统计字符串 str 中空格数，以及字符串的总长度，由于是通过指针移动统计，因而统计结束之后需要将指针复位
int originLength = 0, blankCount = 0;
while(*str != '\0'){
 originLength ++;
if(*str == ' '){
 blankCount ++;
}
str ++;
}
// 指针复位
str -= originLength;
 int totalLength = originLength + 2 * blankCount;
```

```
if(totalLength > length)
return;
// 从最后一个字符开始替换，当两个字符串指针相遇时，替换结束
while(originLength != totalLength){
 char a = str[originLength];
 if ( a == ' '){
str[totalLength--] = '0';
str[totalLength--] = '2';
str[totalLength--] = '%';
originLength--;
 }
else{
str[totalLength] = str[originLength]; originLength--;
totalLength--;
}
 }
 }
};
```

3. 从尾到头打印链表

题目描述

输入一个链表，从尾到头打印链表每个节点的值。

思路分析

通过递归的方法很容易求解。

```
/**
*  struct ListNode {
*        int val;
*        struct ListNode *next;
*        ListNode(int x) :
*              val(x), next(NULL) {
*        }
*  };
*/class Solution {public:
    vector<int> printListFromTailToHead(ListNode* head) {
        vector<int> v;
        _printListFromTailToHead(head,v);
        return v;
```

```
    }

    void _printListFromTailToHead(ListNode* head,vector<int> &v){
        // 遍历到尾结点，返回
        if(head == NULL){
            return;
        }
        // 递归获取下一个结点
        _printListFromTailToHead(head->next,v);
        // 打印结点值
        v.push_back(head->val);
    }
};
/**
*    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>();
        _printListFromTailToHead(listNode,arr);
        return arr;
    }

    private void _printListFromTailToHead(ListNode
listNode,ArrayList<Integer> arr){
        if(listNode == null){
            return;
        }
        // 先行递归
        _printListFromTailToHead(listNode.next,arr);
        arr.add(listNode.val);
    }
}
```

4.重建二叉树

题目描述

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。
e

思路分析

重建二叉树是一个经典的二叉树问题，主要是考察二叉树的遍历特点，根据前序遍历和中序遍历构建二叉树。

在二叉树的前序遍历中，第一个数字总是树的根节点。在中序遍历中，树的根节点在序列的中间，左子树的节点的值位于根节点的左边，右子树节点的值位于根节点值的右边。

因此需要扫描中序遍历序列才能找到很结点的值，由此可以找到左子树的节点的个数和右子树节点的个数，然后在前序遍历序列中找到左子树的根节点，再到中序遍历序列中找到左子树的左子树和右子树。依次递归。由于二叉树的构造本身就是用递归实现的，所以重建二叉树也用递归进行实现实很简单的。

代码实现

```
public class Solution {

    public static void main(String[] args) {
        int[] preOrder = {1, 2, 4, 7, 3, 5, 6, 8};
        int[] inOrder = {4, 7, 2, 1, 5, 3, 8, 6};
        BinaryTreeNode node = reConstruct(preOrder, inOrder);
        printTree(node);
    }

    //二叉树节点
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }

    /**
     * 判断输入合法性
     *
```

```
     * @param preOrder
     * @param inOrder
     * @return
     */
    public static BinaryTreeNode reConstruct(int[] preOrder, int[]
inOrder) {
        if (preOrder == null || inOrder == null || preOrder.length !=
inOrder.length || preOrder.length < 1)
            return null;
        return construct(preOrder, 0, preOrder.length - 1, inOrder, 0,
inOrder.length - 1);
    }

    /**
     * 根据前序遍历和中序遍历构建二叉树
     *
     * @param preOrder 前序遍历序列
     * @param ps       前序遍历开始位置
     * @param pe       前序遍历结束位置
     * @param inOrder  中序遍历序列
     * @param is       中序遍历开始位置
     * @param ie       中序遍历结束位置
     * @return 构建的树的根节点
     */
    public static BinaryTreeNode construct(int[] preOrder, int ps, int
pe, int[] inOrder, int is, int ie) {
        //开始位置大于结束位置说明已经处理到叶节点了
        if (ps > pe)
            return null;
        ///前序遍历第一个数字为当前的根节点
        int value = preOrder[ps];
        //index 为根节点在中序遍历序列中的索引
        int index = is;
        while (index <= ie && inOrder[index] != value) {
            index++;
        }

        //如果在整个中序遍历中没有找到根节点说明输入的数据是不合法的
        if (index > ie) {
            throw new RuntimeException("invalid input" + index);
        }
        BinaryTreeNode node = new BinaryTreeNode();
        node.value = value;
        //当前节点的左子树的个数为 index-is
```

```
        //左子树对应的前序遍历的位置在 preOrder[ps+1,ps+index-is]
        //左子树对应的中序遍历的位置在 inOrder[is,index-1]
        node.left = construct(preOrder, ps + 1, ps + index - is, inOrder,
is, index - 1);
        //当前节点的右子树的个数为 ie-index
        //右子树对应的前序遍历位置在 preOrder[ps+index-is+1,pe]
        //右子树对应的中序遍历位置在 inOrder[index+1,ie]
        node.right = construct(preOrder, ps + index - is + 1, pe, inOrder,
index + 1, ie);
        return node;
    }

    //中序遍历递归打印
    public static void printTree(BinaryTreeNode node) {
        if (node != null) {
            printTree(node.left);
            System.out.print(node.value + " ");
            printTree(node.right);
        }
    }

}
```

5.用两个栈实现队列

题目描述

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。

```
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() {
       // 保证元素都是一次性从 stack1 转移到 satck2
       if(stack2.isEmpty() && !stack1.isEmpty()){
```

```
            while(!stack1.isEmpty()){
                stack2.push(stack1.pop());
            }
        }
        if(!stack2.isEmpty()){
            return stack2.pop();
        }
         throw new RuntimeException("queue is empty....");
    }
}
```

注意事项

1.用第一个栈 stack1 进行 push()操作。
2.pop()操作时都是从 stack2 中进行的，因为 stack2 中元素的顺序恰好是队列出队的顺序，所以先检查队列是否为空，如果不为空，保证元素都是一次性从 stack1 转移到 satck2 中，然后让 stack2 进行 pop()操作。

6. 旋转数组的最小数字

题目描述

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。 输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。
例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为 1。 NOTE：给出的所有元素都大于 0，若数组大小为 0，请返回 0。

思路分析

1.由于数组是非递减的排序数组，因而旋转后的数组，旋转部分依然会保持着这样的特性。
2.设置一个 maxValue，从第一个开始检测，当发现后一个比前一个大或等于时，更新 maxValue，否则说明找到了，将最小值保存在 minValue 中，结束查找即可。

代码实现

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

```
        int maxValue = 0;
        for(int i = 0; i < array.length; i ++){
            if(array[i] >= maxValue){
                maxValue = array[i];
            }else{
                minValue = array[i];
                break;
            }
        }
        return minValue;
    }
}
```

7.斐波那契数列

题目描述

大家都知道斐波那契数列，现在要求输入一个整数 n，请你输出斐波那契数列的第 n 项。n<=39。

解决方法

```
public class Solution { public int Fibonacci(int n) { // 索引位置 ： 1 2 3 4 5 6 7 // 数列形式 ： 0 1 1 2 3 5 8 13 if(n == 1){ return 1; } if(n == 2){ return 1; } // 由于 n <= 39 ,因而可以采用递归方法，实际上还是栈溢出，因而用迭代吧 int first = 1; int second = 1; int result = 0; for(int i = 3; i <= n; i ++){ result = first + second; first = second; second = result; } return result; } }
```

8. 跳台阶

题目描述

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。

解决方法

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

9.变态跳台阶

题目描述

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。
解决思路
f(1) = 1
f(2) = f(2-1) + f(2-2) //f(2-2) 表示 2 阶一次跳 2 阶的次数。
f(3) = f(3-1) + f(3-2) + f(3-3)
...
f(n) = f(n-1) + f(n-2) + f(n-3) + ... + f(n-(n-1)) + f(n-n)
说明：
1）这里的 f(n) 代表的是 n 个台阶有一次 1,2,...n 阶的 跳法数。
2）n = 1 时，只有 1 种跳法，f(1) = 1
3）n = 2 时，会有两个跳得方式，一次 1 阶或者 2 阶，这回归到了问题（1），f(2) = f(2-1) + f(2-2)

4）n = 3 时，会有三种跳得方式，1 阶、2 阶、3 阶，

那么就是第一次跳出 1 阶后面剩下：f(3-1);第一次跳出 2 阶，剩下 f(3-2)；第一次 3 阶，那么剩下 f(3-3)

因此结论是 f(3) = f(3-1)+f(3-2)+f(3-3)
5）n = n 时，会有 n 中跳的方式，1 阶、2 阶...n 阶，得出结论：

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

6）由以上已经是一种结论，但是为了简单，我们可以继续简化：

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

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

可以得出：

f(n) = 2*f(n-1)

7）得出最终结论,在 n 阶台阶，一次有 1、2、...n 阶的跳的方式时，总得跳法为：

```
        | 1        ,(n=0 )
f(n) = | 1 ,(n=1 )
          | 2*f(n-1),(n>=2)
```

解决方案

```
public class Solution { public int JumpFloorII(int target) { if(target < 0){ throw new RuntimeException("Bad arguments"); } int result = 1; if(target == 1){ return result; } for(int i = 1; i <= target-1; i ++){ result *= 2; } return result; } }
```

10. 矩阵覆盖

题目描述

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？

思路分析

通过总结发现规律符合斐波那契数列。用迭代法快速求解。

代码实现

```
public class Solution {
    public int RectCover(int target) {
       if(target <= 0){
            return 0;
        }
        if(target == 1){
            return 1;
        }
        if(target == 2){
            return 2;
        }
```

```
        int front = 1;
        int current = 2;
        int result = 0;
        for(int i = 3; i <= target; i++){
            result = front + current;
            front = current;
            current = result;
        }
        return result;
    }
}
```

11. 二进制中 1 的位数

题目描述

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。

代码实现

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

这里使用了一个小技巧，即如果 n 的二进制数中至少含有一个 1，n&(n-1)的运算结果会把数字 n 的二进制数的最右面的 1 变为 0，因而一直运算到 n 等于 0 时的运算次数即为数字 n 的二进制表示中 1 的个数。

```
public class Solution {
    public int NumberOf1(int n) {
        int numOfOne = 0;
       for(int i = 0; i < 32; i ++){
            if(((n >> i ) & 1) == 1){
                numOfOne ++;
            }
```

```
        }
        return numOfOne;
    }
}
```

这里采用常规的计算方法，通过每次将数字 n 除以 2 来判断每一位是否为 1，最后得到 1 的总数。

12. 数值的整数次方

题目描述

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。

```
/**
    * 原始方法：时间复杂度为 O(exponent)
    *
    * @param base
    * @param exponent
    * @return
    */
   public static 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;
   }
/**
    * 优化方法：时间复杂度为 log(exponent)
    * 1.全面考察指数的正负、底数是否为零等情况。
    * 2.写出指数的二进制表达，例如 13 表达为二进制 1101。
    * 3.举例:10^1101 = 10^0001*10^0100*10^1000。
```

```
     * 4.通过&1 和>>1 来逐位读取 1101，为 1 时将该位代表的乘数累乘到最终
结果。
     * @param base
     * @param exponent
     * @return
     */
    public static double OptimizePower(double base, int exponent) {
        if (base == 0.0d) {
            return 0.0;
        }
        int e = exponent > 0 ? exponent : -exponent;
        double result = 1.0;
        double current = base;

        while (e != 0) {
            // 如果当前位为 1
            if ((exponent & 1) == 1) {
                result *= current;
            }
            current *= base;
            e >>= 1;
        }
        return exponent > 0 ? result : 1 / result;
}
```

13.调整数组顺序使奇数位于偶数前面

题目描述

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。

思路分析

扫描数组，若发现偶数在奇数前面，则交换它们的位置。使用两个指针，第一个指针从数组第一个元素开始向后扫描，第二个指针从数组最后一个元素开始向前扫描。若第一个指针指向偶数，第二个指针指向奇数，则交换它们的顺序，如此循环，直到第一个指针和第二个指针相遇，或者在第二指针的后面。

代码实现

```
import java.util.Arrays;

public class Solution {

    public static void main(String[] args) {
        int[] array = new int[10];
        for (int i = 0; i < 10; i++) {
            array[i] = i;
        }
        reorder(array);
        System.out.println(Arrays.toString(array));
    }

    private static void reorder(int[] array) {
        if (array == null || array.length == 0) return;
        int begin = 0;
        int end = array.length - 1;
        while (begin < end) {
            //找到数组前面的偶数
            while (begin < end && !isEven(array[begin])) begin++;
            //找到数组后面的奇数
            while (begin < end && isEven(array[end])) end--;
            //交换奇偶数
            if (begin < end) {
                int temp = array[begin];
                array[begin] = array[end];
                array[end] = temp;
            }
        }
    }

    /**
     * 查找条件
     *
     * @param n
     * @return
     */
    private static boolean isEven(int n) {
        return (n & 1) == 0;
    }

}
```

14. 链表中倒数第 k 个结点

题目描述

输入一个链表，输出该链表中倒数第 k 个结点。

思路分析

1. 先遍历链表，获得链表中的总结点数。
2. 倒数第 k 个元素，即为顺数第 size - k + 1 个元素。
3. 遍历链表，获得第 size - k + 1 的元素。

```
/*
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){
            return null;
        }
        // 统计链表中所有节点数
        ListNode current = head;
        int count = 0;
        while(current != null){
            current = current.next;
            count ++;
        }
        // 判断 k 是否合法
        if(k > count){
            return null;
        }
        int offset = count - k + 1;
        current = head;
        // 当 offset 变为 1 的时候，就找到了这个结点
        while(offset != 1){
            current = current.next;
            offset --;
        }
```

```
            return current;
        }
    }
```

15. 反转链表

题目描述

输入一个链表，反转链表后，输出链表的所有元素。

参考资料

Java 实现单链表翻转　看图理解单链表的反转

```
public class Solution {

    static class Node {
        int val;
        Node next;

        public Node(int val, Node next) {
            this.val = val;
            this.next = next;
        }
    }

    /**
     * 非递归方式实现
     *
     * @param head
     * @return
     */
    public static Node reverseList(Node head) {
        Node prev = null;
        while (head != null) {
            // 保存 head 的下一个结点
            Node next = head.next;
            // 将结点 head 反转
            head.next = prev;
            // 移动 prev 、head 结点
            prev = head;
            head = next;
```

```
        }
        // 当 head = null 时，prev 恰好是原始链表的尾结点
        return prev;
    }

    /**
     * 用递归的方法反转链表
     *
     * @param head
     * @return
     */
    public static Node reverseList_(Node head) {
        if (head == null || head.next == null) {
            return head;
        }
        Node reHead = reverseList_(head.next);
        // 倒数第二个节点
        head.next.next = head;
        // 断开原始链接
        head.next = null;
        return reHead;
    }

    public static void main(String[] args) {
        Node head = new Node(1, null);
        Node current = head;
        for (int i = 2; i < 11; i++) {
            Node node = new Node(i, null);
            current.next = node;
            current = current.next;
        }
        Node reHead = reverseList(head);
        while (reHead != null) {
            System.out.print(reHead.val + " ");
            reHead = reHead.next;
        }
        System.out.println();
        current = head;
        for (int i = 2; i < 11; i++) {
            Node node = new Node(i, null);
            current.next = node;
            current = current.next;
        }
        Node reHead_ = reverseList_(head);
```

```
        while (reHead_ != null) {
            System.out.print(reHead_.val + " ");
            reHead_ = reHead_.next;
        }
    }
}
```

16.合并两个排序的链表

题目描述

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。

代码实现

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/public class Solution {
    public ListNode Merge(ListNode list1,ListNode list2) {
        ListNode head;
        // 如果 list1 为空，返回 list2
        if(list1 == null){
            return list2;
        }
        // 如果 list2 为空，返回 list1
        if(list2 == null){
            return list1;
        }
        // 比较递归
        if(list1.val <= list2.val){
            head = list1;
            head.next = Merge(list1.next,list2);
        }else{
            head = list2;
            head.next = Merge(list1,list2.next);
        }
```

```
        return head;
    }
}
```

17.树的子结构

题目描述

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

思路分析

假如有两棵树，即 A 树与 B 树。要查找树 A 中是否存在和树 B 结构一样的子树，思路是使用递归去比较每个元素。递归终止条件是我们到达树 A 或者树 B 的叶结点。

代码实现

```
class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

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

    }

}


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 = check(root1, root2);
        }

        if (!result) {
```

```
            result = HasSubtree(root1.right, root2);
        }

        if (!result) {
            result = HasSubtree(root1.left, root2);
        }

        return result;
    }

    private boolean check(TreeNode node1, TreeNode node2) {
        //如果 root2 都检查完了，都匹配，则返回 true
        if (node2 == null) return true;
        //如果 root1 都检查完了，不匹配，则返回 false
        if (node1 == null) return false;

        if (node1.val != node2.val) return false;

        return check(node1.left, node2.left) && check(node1.right,
node2.right);
    }

}
```

18.二叉树的镜像

题目描述

操作给定的二叉树，将其变换为源二叉树的镜像。

思路分析

如果当前节点为空，返回，否则交换该节点的左右节点，递归的对其左右节点进行交换处理。

代码实现

```
class TreeNode {
    int val;
    TreeNode left = null;
    TreeNode right = null;
```

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


public class Solution {

    public static void mirrorTree(TreeNode root) {
        if (root == null)
            return;
        //交换该节点指向的左右节点。
        TreeNode temp = root.left;
        root.left = root.right;
        root.right = temp;
        //对其左右孩子进行镜像处理。
        mirrorTree(root.left);

        mirrorTree(root.right);
    }

}
```

19.顺时针打印矩阵

题目描述

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.

思路分析

1.分析发现，我们可以通过循环的每一次打印矩阵中的一个圈（从外到内）。
2.接下来分析循环结束的条件：假设这个矩阵的行数是 rows，列数是 columns。打印第一圈的左上角的坐标是(0,0),第二圈的左上角的坐标是(1,1),依次类推。我们注意到，左上角的行坐标和列坐标总是相同的，于是可以在矩阵中选取左上角为(start,start)的一圈作为我们分析的目标。
3.条件是：columns > startX × 2 并且 rows > startY × 2
4.显然我们可以把打印一圈分为四步：第一步从左到右打印一行，第二步从上到下打印一列，第三步从右到左打印一行，第四步从下到上打印一列。

5.不过，最后一圈有可能会退化成只有一行，只有一列，甚至只有一个数字。因此打印也退化成了三步或两步或一步。因此打印前要分析好每一步的前提条件。
6.第一步总是需要的。第二步需要判断终止行号大于起始行号。第三步需要终止行号大于起始行号以及终止列小于起始列号。第四步需要至少有三行两列，因此要求终止行号比起始行号至少大
7.同时终止行号大于起始列号。

代码实现

```
import java.util.List;import java.util.ArrayList;public class Solution
{
    public ArrayList<Integer> printMatrix(int[][] matrix) {
        //  1  2  3 4
        //  5  6  7 8
        //  9 10 11 12
        // 13 14 15 16
        ArrayList<Integer> list = new ArrayList<>();
        int m = matrix.length;
        int n = 0;
        if (m <= 0) {
            return list;
        } else {
            n = matrix[0].length;
        }
        // 从左上角开始打印
        int start = 0;
        // 当 start * 2 小于行数并且小于列数时，说明存在小矩阵可以继续
打印
        while (start * 2 < m && start * 2 < n) {
            printMatrixByCircle(matrix, start, list);
            start++;
        }
        return list;
    }

    private void printMatrixByCircle(int[][] matrix, int start,
List<Integer> res) {
        int rows = matrix.length;
        int cols = matrix[0].length;
        // 计算结束位置
        int endX = cols - 1 - start;
        int endY = rows - 1 - start;

        // 横着打印肯定需要
```

```
            for (int i = start; i <= endX; i++) {
                res.add(matrix[start][i]);
            }

            // 从上到下打印一列
            if (start < endY) {
                for (int i = start + 1; i <= endY; i++) {
                    res.add(matrix[i][endX]);
                }
            }

            // 从右至左打印一行
            if (start < endY && start < endX) {
                for (int i = endX - 1; i >= start; i--) {
                    res.add(matrix[endY][i]);
                }
            }

            // 从下至上打印
            if (start < endY - 1 && start < endX) {
                for (int i = endY - 1; i >= start + 1; i--) {
                    res.add(matrix[i][start]);
                }
            }
        }
    }
```

20. 包含 min 函数的栈

题目描述

定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的 min 函数。

思路分析

对于 push 和 pop 操作来说，都很简单，无论是数组实现栈，还是链表实现栈都很容易。但是唯独 min 函数不好做。
首先对于栈这个数据结构来说，我们只能获取第一个元素，也就是栈顶的元素。我们不能访问到别的元素，所以我们不行也不可能去遍历获取栈的最小值。
那么我们肯定需要去保存栈的最小值，但是每当一个元素出栈之后，这个最小值需要变动，那么怎么变动呢？这又是一个问题。

还有就是如何去保存这个最小值，如果我们维护一个优先队列，或者别的数据结构， 使得维护的结构是已经排序好的，那么当然可以，但是这对于空间和时间的开销来说都不行。
既然是栈，那么肯定有栈自己的解决方式。
只要维护另一个栈，就能完成这个任务。

思路

1、定义一个栈这里我们称为最小栈，原来的栈我们称为数据栈。
2、最小栈和数据栈元素个数一定相同。最小栈的栈顶元素为数据栈的所有元素的最小值。
3、数据栈入栈一个元素 A，最小栈需要拿这个元素与最小栈栈顶元素 B 比较，如果 A 小于 B，则最小栈入栈 A。否则最小栈入栈 B。
4、当数据栈出栈一个元素时，最小栈也同时出栈一个元素。

代码实现

```
import java.util.Stack;
import java.util.Stack;
public class Solution {

    // 数据栈
    private Stack<Integer> stack = new Stack();
    // 最小栈
    private Stack<Integer> minStack = new Stack();


    public void push(int node) {
        // 如果最小栈非空，并且栈顶元素小于当前值，那么最小栈将栈顶元素继续入栈，代表当前这个栈的最小值没有变化
        if(!minStack.isEmpty() && minStack.peek() < node){
            minStack.push(minStack.peek());
        }else{
            minStack.push(node);
        }
        // 数据栈直接入栈
        stack.push(node);
    }

    public void pop() {
        // 数据栈与最小栈都出栈
        stack.pop();
        minStack.pop();
    }
```

```
    public int top() {
        return stack.peek();
    }

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

21.栈的压入、弹出序列

题目描述

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

思路分析

*因为有入栈序列，所以从入栈开始，遍历入栈序列
* 遍历入栈序列时,每遍历一个,就判断是否是出栈序列中的需要弹出的值，
* 如果是则弹出，然后出栈序列加一，接着继续根据入栈序列添加（for 循环继续）
* 如果不是，则继续根据入栈序列添加，直到匹配出栈序列需要的数字为止
* 如果遍历完都发现没有匹配的，则该出栈序列不是合法出栈序列

代码实现

```
public class Solution {

    public static boolean isPopValid(int[] pushOrder, int[] popOrder) {
        if (pushOrder.length == 0 || popOrder.length == 0 ||
pushOrder.length != popOrder.length)
            return false;
        Stack<Integer> stack = new Stack<Integer>();
        int j = 0;
        for (int current : pushOrder) {
            stack.push(current);
            while ((!stack.empty()) && (stack.peek() == popOrder[j])) {
                stack.pop();
```

```
                j++;
            }
        }
        if (stack.empty()) {
            return true;
        } else {
            return false;
        }
    }

    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<Integer>();
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);
        stack.push(5);
        int push[] = {1, 2, 3, 4, 5};
        int pop[] = {4, 5, 3, 2, 1};

        boolean is = isPopValid(push, pop);
        System.out.println("is valid : " + is);
    }

}
```

22. 从上往下打印二叉树

题目描述

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。

题目分析

层序遍历直接想到的应该是基于队列实现。从上到下打印二叉树的规律：每一次打印一个结点的时候，如果该结点有子结点，则把该结点的子结点放到一个队列的末尾。接下来到队列头部取出最早进入队列的结点，重复前面的打印操作，直到队列中所有的结点都被打印出来为止。

代码实现

```
import java.util.ArrayList;import java.util.Queue;import
java.util.ArrayDeque;/**
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

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

    }

}
*/public class Solution {
    // 使用队列保存遍历到的结点
    private Queue<TreeNode> queue = new ArrayDeque<>();

    public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
        ArrayList<Integer> arr = new ArrayList();
        // 首先将根节点加入队列
        if(root != null){
            queue.add(root);
        }
        while (!queue.isEmpty()){
            // 遍历结点
            TreeNode node = queue.poll();
            arr.add(node.val);
            if (node.left != null){
                queue.add(node.left);
            }
            if (node.right != null){
                queue.add(node.right);
            }
        }
        return arr;
    }
}
```

23.二叉搜索树的后序遍历序列

题目描述

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。

思路分析

二叉搜索树遍历结果中最后一个节点是二叉树的根节点，前面一些节点小于这个节点，后面节点大于这个节点
如果后面一些节点有小于这个节点值的节点，那么就不是二叉搜索树的后序遍历，作为程序的退出点找到临界点点，递归的执行这个函数

代码实现

```
public class Solution {
    public static void main(String[] args) {
//        int[] array = {5, 7, 6, 9, 11, 10, 8, 111};
        int[] array = {7, 4, 6, 5};
//      int[] array={6,7,8,5};
        System.out.println(verfiySequenceOfBST(array));
    }

    public static boolean verfiySequenceOfBST(int[] input) {
        if (input == null || input.length == 0) {
            return false;
        }
        int last = input.length - 1;
        int cut = 0;
        for (int i = 0; i < input.length - 1; i++) {
            if (input[i] > input[last]) {
                break;
            }
            cut = i + 1;
        }
        for (int i = cut; i < input.length - 1; i++) {
            if (input[i] < input[last]) {
                return false;
            }
        }
        boolean left = true;
        boolean right = true;
```

```
        if (cut > 0) {
            left = verfiySequenceOfBST(Arrays.copyOfRange(input, 0,
cut));
        }
        if (cut < input.length - 1) {
            right = verfiySequenceOfBST(Arrays.copyOfRange(input, cut,
input.length - 1));
        }
        return left && right;
    }
}
```

24.二叉树中和为某一值的路径

题目描述

输入一颗二叉树和一个整数，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。

思路分析

路径从根结点开始，应该用类似于前序遍历的方式访问树节点。我们需要整个路径，就需要一个容器保存经过路径上的节点，以及一个变量记录当前已有节点元素的和。当前序遍历到某一个节点时，添加该节点到路径，累加节点值。如果该节点为叶子节点并节点值累计等于目标整数，则找到一条路径。如果不是叶子节点，则继续访问子节点。一个节点访问结束后，递归函数自动回到其父节点。

代码实现

```
class TreeNode {

    TreeNode left;
    TreeNode right;
    Integer val;

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

public class Solution {
```

```
    // pathList 存所有可能路径
    // path 用来保存路径的数据结构 这儿用 ArrayList 模拟了栈的数据结构
    private ArrayList<ArrayList<Integer>> pathList = new
ArrayList<ArrayList<Integer>>();
    private ArrayList<Integer> path = new ArrayList<Integer>();

    public ArrayList<ArrayList<Integer>> findPath(TreeNode root, int
sum) {
        if (root == null)
            return pathList;
        path.add(root.val);
        sum -= root.val;
        // 路径值等于 0，且当前节点是叶子节点 则找到一条路径
        if (sum == 0 && root.left == null && root.right == null)
            // 每次找到路径都需要添加一个新的 path 不可以直接加 path 成
员变量 这是个引用，不然所有 pathList 的值都指向同一个 path
            pathList.add(new ArrayList<Integer>(path));
        if (root.left != null)
            findPath(root.left, sum);

        if (root.right != null)
            findPath(root.right, sum);
        // 访问完当前节点 需要删除路径中最后一个节点，回退至父节点
        path.remove(path.size() - 1);
        return pathList;
    }

    public static void main(String[] args) {
        TreeNode node1 = new TreeNode(10);
        TreeNode node2 = new TreeNode(5);
        TreeNode node3 = new TreeNode(12);
        TreeNode node4 = new TreeNode(4);
        TreeNode node5 = new TreeNode(7);

        node1.left = node2;
        node1.right = node3;

        node2.left = node4;
        node2.right = node5;

        Solution s = new Solution();
        System.out.println(s.pathList);
        s.findPath(node1, 22);
```

```
        for (ArrayList<Integer> list : s.pathList) {
            for (int i : list) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }

}
```

25.复杂链表的复制

题目描述

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的 head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

题目分析

如果是普通链表，那么将原链表遍历一遍，创建出新链表即可。但是这个链表中每个节点中有一个指向任意结点的指针，这就增加了实现的难度。对于这样的问题，有三种思路。
1.将原链表复制后，对于新链表中的每一个结点，在原链表中寻找其特殊指针指向的结点，然后在新链表中找到，并指向那个结点。时间复杂度为 O(n^2)。

2.将原链表的结点以及新链表的结点一一对应的存储在 Hash 表中，这样通过两次遍历链表的操作即可将链表复制。时间复杂度为 O(n),空间复杂度为 O(n)。

3.将原链表的结点复制后，插入到原结点后面，这样复制完之后就会发现，新链表中每一个节点特殊指针即为原链表结点特殊指针后面的那个结点，这样的时间复杂度为 O(n)。

代码实现

```
RandomListNode* RandomList::cloneByHashTable(RandomListNode*
pHead){ if (pHead == NULL) { return NULL; }
// 定义一个 hash 表
unordered_multimap<RandomListNode*, RandomListNode*> table;// 开辟一个
头结点
RandomListNode* pCloneHead = new RandomListNode(0);
pCloneHead->next = NULL;
```

```
pCloneHead->random = NULL;
//将头节点放入 table 中
table.insert(make_pair(pHead,pCloneHead));
// 设置操作指针
RandomListNode* pNode = pHead->next;
RandomListNode*pCloneNode = pCloneHead;
// 第一遍，先简单复制链表 while (pNode != NULL)
{
    RandomListNode* pCloneTail = new RandomListNode(pNode->label);
    pCloneTail->next = pNode->next;
    pCloneTail->random = NULL;

    // 将复制的节点链接到复制链表的尾部
    pCloneNode->next = pCloneTail;
    // 更新当前结点
    pCloneNode = pCloneTail;

    // 将新的对应关系插入到 Hash 表
    table.insert(make_pair(pNode,pCloneTail));

    // 移动原链表操作指针
    pNode = pNode->next;
}
// 需从头设置特殊指针，重新设置操作指针
pNode = pHead;
pCloneNode = pCloneHead;
while (pNode != NULL){
    // 如果存在特殊指针，则设置之
    if (pNode->random != NULL)
    {
        pCloneNode->random = table.find(pNode->random)->second;
    }
    // 移动原链表以及克隆链表
    pNode = pNode->next;
    pCloneNode = pCloneNode->next;
}return pCloneHead;
}
class RandomListNode {
    int label;
    RandomListNode next = null;
    RandomListNode random = null;

    RandomListNode(int label) {
        this.label = label;
```

```
    }
}
public class Solution {
    public RandomListNode Clone(RandomListNode pHead)
    {
        copyList(pHead);
        setSibling(pHead);
        return disconnectedList(pHead);
    }

    // 将复制后的结点放在原链表结点的尾部
    private void copyList(RandomListNode head){
        if(head == null)
            return;
        RandomListNode node = head;
        while(node != null){
            RandomListNode copyNode = new RandomListNode(node.label);
            copyNode.next = node.next;
            copyNode.random = null;

            node.next = copyNode;
            node = copyNode.next;
        }
    }

    // 将链表中复制后的结点的 random 属性设置好
    private void setSibling(RandomListNode head){
        if(head == null)
            return;
        RandomListNode node = head;
        while(node != null){
            RandomListNode copyNode = node.next;
            if(node.random != null){
                copyNode.random = node.random.next;
            }
            node = copyNode.next;
        }
    }

    private RandomListNode disconnectedList(RandomListNode head){
        RandomListNode node = head;
        RandomListNode copyHead = null;
        RandomListNode copyNode = null;
        if(node != null){
```

```
            copyHead = node.next;
            copyNode = node.next;

            node.next = copyNode.next;
            node = node.next;
        }

        while(node != null){
            copyNode.next = node.next;
            copyNode = copyNode.next;

            node.next = copyNode.next;
            node = node.next;
        }
        return copyHead;
    }
}
```

26.二叉搜索树与双向链表

题目描述

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

思路分析

二叉搜索树的中序遍历是一个有序的数组，在中序遍历的时候，用 Pre 指针保存前一个节点，当访问到当前节点的时候，将 Pre 节点右指针，指向当前节点，当前节点的左指针指向 Pre。 这样中序遍历完二叉搜索树，就产生了一个双向链表。

代码实现

```
class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

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

```
    }

}

public class Solution {

    public TreeNode ConvertWithStack(TreeNode pRootOfTree) {
        if (pRootOfTree == null)
            return null;
        //如果根节点为 null 返回 null 否则当前节点 cur 指向根节点
pRootOfTree
        TreeNode cur = pRootOfTree;
        TreeNode pre = null;
        //pre 用于指向当前节点的前一个节点，开始时为 null。
        Stack<TreeNode> stack = new Stack<TreeNode>();
        while (!stack.isEmpty() || cur != null) {
            //将左子树上的节点入栈
            while (cur != null) {
                stack.push(cur);
                cur = cur.left;
            }
            //如果已经到了树的最左边，则将栈中的节点出栈。
            cur = stack.pop();
            if (pre == null) {
                //刚开始的时候 pre==null 将最左边的 节点赋值给
pRootOfTree。当前节点复值给 pre
                pRootOfTree = cur;
                pre = cur;

            } else {
                pre.right = cur;
                cur.left = pre;
                pre = cur;
                //前一个节点的右指针指向 cur 当前节点，当前节点的左指
针指向 前一个节点 pre
            }
            cur = cur.right;
        }
        return pRootOfTree;
    }

}
```

27. 字符串的排列

题目描述

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。
输入描述
输入一个字符串,长度不超过 9(可能有字符重复),字符只包括大小写字母。

思路分析

递归的思路暂时还存在问题！！！！

代码实现

```
import java.util.List;import java.util.ArrayList;import
java.util.Collections;public class Solution {

    public ArrayList<String> Permutation(String str) {
        ArrayList<String> res = new ArrayList();
       if(str == null || str.length() == 0){
           return res;
       }
        PermutationHelper(str.toCharArray(),0,res);
        Collections.sort(res);
        return res;
    }

    private void PermutationHelper(char[] str,int n,List<String> res){
        if(n == str.length){
            res.add(String.valueOf(str));
        }
        int i = n;
        for(int j = i; j < str.length; j++){
            if(i != j && str[i] == str[j]){
                continue;
            }
            swap(str,i,j);
            PermutationHelper(str,i+1,res);
            swap(str,i,j);
        }
    }
```

```
    /**
    * 交换字符串中的两个字符
    */
    private void swap(char[] str,int i,int j){
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}
```

28. 数组中出现次数超过一半的数字

题目描述

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。

思路分析

最直观的思路就是统计出每个数字出现的次数，当发现某一个数字出现次数大于数组长度的一半时，返回结果即可，这样的时间复杂度为 O(n^2)。

代码实现

```
public class Solution {
    public int MoreThanHalfNum_Solution(int [] array) {
        if(array == null){
            return 0;
        }
        int length = array.length;
        int times = length >> 1;
        for(int i=0; i < length; i ++){
            int count = 0;
            for(int j=0; j < length; j ++){
                if(array[i] == array[j]){
                    count += 1;
                }
            }

            if(count > times){
```

```
                return array[i];
            }
        }
        return 0;
    }
}
```

优化方法

将数组排序，那么排序后的数组中间的数字必定为寻找的数字，这样时间复杂度就降低成排序算法的时间复杂度了，由于还需进一步确认中间的那个数字是否真的超过了数组长度的一半，再统计一遍即可。

时间复杂度为（nLog(n) + n）。

代码实现

```
import java.util.Arrays;
public class Solution {
    public int MoreThanHalfNum_Solution(int [] array) {
        if (array == null || array.length < 1) {
            return 0;
        }
        Arrays.sort(array);
        int length = array.length;
        int target = array[length / 2];
        int count = 0;
        for (int i = 0; i < length; i++) {
            if (target == array[i])
                count++;
        }
        if (count > (length >> 1)){
            return target;
        }
        return 0;
    }
}
```

29. 最小的 K 个数

题目描述

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,。

题目分析

这是一个典型的 topK 问题，即求一组数字中最小的 k 个数，因而经典的解法也有很多，比如，维护一个大小为 k 的最大堆，当堆满后，再有新的数字入堆时，就与堆顶的数字进行比较，如果新的数字小于堆顶的数字，就删除堆顶的数字，并将新的数字放入堆中，这样，遍历完所有的数字的时候，堆里的元素就是最小的 k 个数。其时间复杂度大致为 O(nlog(n))

代码实现

```
public class Solution {

    // 最大堆
    private static PriorityQueue<Integer> maxHeap = new
PriorityQueue<>(new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return o2 - o1;
        }
    });

    /**
     * @param input
     * @param k
     * @return
     */
    public static ArrayList<Integer> GetLeastNumbers_Solution(int[]
input, int k) {
        ArrayList<Integer> result = new ArrayList<>();
        if (input == null || input.length == 0 ||input.length < k) {
            return result;
        }
        if (input.length <= k) {
            for (int i = 0; i < input.length; i++) {
                result.add(input[i]);
            }
```

```
            return result;
        }
        for (int i = 0; i < input.length; i++) {
            if (maxHeap.size() < k) {
                maxHeap.offer(input[i]);
            } else {
                // 如果新元素比堆顶的元素还要小，替换之
                if (maxHeap.element() > input[i]){
                    maxHeap.poll();
                    maxHeap.offer(input[i]);
                }
            }
        }
        Integer[] ele = new Integer[k];
        maxHeap.toArray(ele);
        for (int i = 0; i < ele.length; i++) {
            result.add(ele[i]);
        }
        return result;
    }

    public static void main(String[] args) {
        int[] arr = {4,5,1,6,2,7,3,8};
        ArrayList<Integer> list = GetLeastNumbers_Solution(arr,1);
        System.out.println(list);
    }
}
```

30. 连续子数组的最大和

题目描述

HZ 偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后,他又发话了:在古老的一维模式识别中,常常需要计算连续子向量的最大和,当向量全为正数的时候,问题很好解决。但是,如果向量中包含负数,是否应该包含某个负数,并期望旁边的正数会弥补它呢？例如:{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为 8(从第 0 个开始,到第 3 个为止)。你会不会被他忽悠住？(子向量的长度至少是 1)

思路分析

我们试着从头到尾累加数组中的每个数字。初始化和为 0。 第一步加上第一个数字 1，此时和为 1。 第二步加上第二个数字-2，此时和为-1。小于 0，那么用-1 加上 3 为 2，比 3 本身还小。也就是说从第一个数字开始的子数组比从第三个数字开始的子数组小。因此不必考虑从第一个数字开始的子数组。 第三步，从第三个数开始累加。继续第一步，第二步，直到最后。

代码实现

```
public class Solution {
    public int FindGreatestSumOfSubArray(int[] array) {
        if(array == null || array.length < 1){
            return 0;
        }
        int currentSum = 0;
        int greateSum = 0x80000000;
        int length = array.length;
        for(int i=0; i < length; i++){
            // 如果当前和小于等于 0,那么将当前值更新为将会访问的元素
            if(currentSum <= 0){
                currentSum = array[i];
            }else{
                // 如果当前值大于 0，加上将要访问的元素
                currentSum += array[i];
            }
            // 比较当前值与当前最大值，更新当前最大值
            if(currentSum > greateSum){
                greateSum = currentSum;
            }
        }
        return greateSum;
    }
}
```

31.整数中 1 出现的次数

题目描述

求出 1~13 的整数中 1 出现的次数,并算出 100~1300 的整数中 1 出现的次数？为此他特别数了一下 1~13 中包含 1 的数字有 1、10、11、12、13 因此共出现 6 次,但是对于后面问题他就没辙了。ACMer 希望你们帮帮他,并把问题更加普遍化,可以很快的求出任意非负整数区间中 1 出现的次数。

思路分析

1、如果第 i 位（从低位向高位开始）上的数字是 0，那么第 i 位可能出现 1 的次数仅由更高位决定（如果没有高位，则视高位为 0），等于更高位数字*当前位数的权重 10^(i-1)；

2、如果第 i 位上的数字为 1，则第 i 位上可能出现 1 的次数不仅受更高位影响还受低位影响（如果没有低位，则视低位为 0），等于更高位数字*当前位数的权重 10^(i-1) + (低位数字+1)；

3、如果第 i 位上的数字大于 1，则第 i 位上可能出现 1 的次数仅由更高位决定（如果没有高位，则视高位为 0），等于（更高位数字+1）*当前位数的权重 10^(i-1)。

注：（这里的 X∈[1,9]，因为 X=0 不符合下列规律，需要单独计算）。

首先要知道以下的规律：

从 1 至 10，在它们的个位数中，任意的 X 都出现了 1 次。

从 1 至 100，在它们的十位数中，任意的 X 都出现了 10 次。

从 1 至 1000，在它们的百位数中，任意的 X 都出现了 100 次。

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

以 21354 为例，寻找 1 出现的次数：

个位：从 1 至 21350 中包含了 2135 个 10，因此 1 出现了 2135 次，21351，21352，21353，21354 其中 21351 包含了一个 1，故个位出现 1 的次数为：2135*10(1-1) + 1 = 2136 次；

公式：(2135+1) * 10^(1-1) = 2136；

十位：从 1 到 21300 中包含了 213 个 100，因此 1 出现了 213 * 10^(2-1) = 2130 次，剩下的数字是 21301 到 21354，它们的十位数字是 5 > 1；因此它会包含 10 个 1；故总数为 2130 + 10 = 2140 次；

公式：（213 + 1）* 10^(2-1) = 2140 次；

百位：从 1 到 21000 中包含了 21 个 1000，因此 1 出现了 21 * 10^(3-1) = 2100 次，剩下的数字是 21001 到 21354，它们的百位数字是 3 > 1；因此它会包含 100 个 1；故总数为 2100 + 100 = 2200 次；

公式：（21 + 1）* 10^(3-1) = 2200 次；

千位：从 1 到 20000 中包含了 2 个 10000，因此 1 出现了 2 * 10^(4-1) = 2000 次，剩下的数字是 20001 到 21354，它们的千位数字是 1 = 1；情况稍微复杂些，354 + 1 = 355；故 1 的总数为 2000 + 355 = 2355 次；

公式：2 * 10^(4-1) + 354 + 1 = 2355 次；

万位：万位是 2 > 1，没有更高位；因此 1 出现的次数是 1 * 10^(5-1) = 10000 次；

公式：（0 + 1）*10^(5-1) = 10000 次；

故总共为：2136+2140+2200+2355+10000=18831 次；

故总结：

* 取第 i 位左边的数字（高位），乘以 10 ^(i-1) ，得到基础值 a 。
* 取第 i 位数字，计算修正值：
  * 如果大于 X，则结果为 a+ 10 ^(i-1) 。
  * 如果小于 X，则结果为 a 。
  * 如果等 X，则取第 i 位右边（低位）数字，设为 b ，最后结果为 a+b+1 。

代码实现

```
public class Solution {

    public int numberOf1Between1AndN(int n, int x) {
        if (n < 0 || x < 1 || x > 9) {
            return 0;
        }
        int curr, low, temp, high = n, i = 1, total = 0;
        while (high != 0) {
            high = n / (int) Math.pow(10, i); //获取第 i 位的高位
            temp = n % (int) Math.pow(10, i); //
            curr = temp / (int) Math.pow(10, i - 1); //获取第 i 位
            low = temp % (int) Math.pow(10, i - 1); //获取第 i 位的低位
            if (curr == x) { //等于 x
                total += high * (int) Math.pow(10, i - 1) + low + 1;
            } else if (curr < x) { //小于 x
                total += high * (int) Math.pow(10, i - 1);
            } else { //大于 x
                total += (high + 1) * (int) Math.pow(10, i - 1);
            }
```

```
            i++;
        }
        return total;
    }

    public static void main(String[] args) {
        Solution s = new Solution();
        int nums = s.numberOf1Between1AndN(111, 1);
        System.out.println(nums);
    }

}
```

32. 把数组排成最小的数

题目描述

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。

思路分析

这个题目类似于字典序的问题，而在这里，是比较两个数字组合之后的大小。即对于数字 a 和 b，它们之间的组合有 ab 和 ba 两种，要求我们打印出最小的一个组合数字出来； 如果 ab 小于 ba，则 a 是在 b 的前面; 如果 ab 大于 ba，则 b 在 a 的前面;按照这种思路，对于一个输入数组中的所有数字，我们将通过特殊处理后将其排序即可得到结果。

代码实现

```
import java.util.List;import java.util.ArrayList;import
java.util.Comparator;import java.util.Collections;
    public class Solution {
    public String PrintMinNumber(int [] numbers) {
       if (numbers == null || numbers.length < 1) {
            return "";
        }
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < numbers.length; i++) {
            list.add(numbers[i]);
        }
```

```
        Collections.sort(list, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                String str1 = o1 + "" + o2;
                String str2 = o2 + "" + o1;
                return str1.compareTo(str2);
            }
        });
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < numbers.length; i++) {
            sb.append(list.get(i));
        }
        return sb.toString();

    }
}
```

33. 找出第 n 个抽数

题目描述

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

代码实现

```
public class Solution {
    // 假定 N 的最大值为 1500
    private int[] data = new int[1500];

    public int GetUglyNumber_Solution(int index) {
        if(index <= 0){
            return 0;
        }
        // 第一个数为丑数
       data[0] = 1;
        // 设置三个指针，分别表示当前可以获得下个丑数的基数的位置
        int i2 = 0;
        int i3 = 0;
        int i5 = 0;
```

```
        // 获取下一个丑数
        int currentIndex = 1;
        int next = data[0];
        while(currentIndex < index){
            // 计算下一个丑数
            next = min(data[i2] * 2,data[i3]* 3, data[i5] * 5);
            if(data[i2] * 2 <= next){
                i2 ++;
            }
            if(data[i3] * 3 <= next){
                i3 ++;
            }
            if(data[i5] * 5 <= next){
                i5 ++;
            }
            data[currentIndex ++] = next;
        }
        return next;
    }

    private int min(int a,int b, int c){
        int min = a < b ? a : b;
        min = min < c ? min : c;
        return min;
    }
}
```

34. 第一个只出现一次的字符

题目描述

在一个字符串(1<=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

题目分析

1.HashMap 记录字符出现频数
2.数组记录字符的顺序以及字符的位置

代码实现

```
import java.util.Map;import java.util.HashMap;public class Solution {
```

```
    private Map<Character,Integer> map = new HashMap();
    private char[] data = new char[10000];

    public int FirstNotRepeatingChar(String str) {
        // 统计字符出现次数以及记录字符位置
        for(int i = 0; i < str.length(); i ++){
            char cur = str.charAt(i);
            if(map.get(cur) == null){
                map.put(cur,1);
            }else{
                map.put(cur,map.get(cur) + 1);
            }
            data[i] = cur;
        }
        // 寻找第一个只出现一次字符，并返回位置
        for(int i = 0; i < str.length(); i ++){
            if(map.get(data[i]) == 1){
                return i;
            }
        }
        return -1;
    }
}
```

35.数组中的逆序对

题目描述

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。并将 P 对 1000000007 取模的结果输出。 即输出 P%1000000007

思路分析

利用归并排序的思想,先求前面一半数组的逆序数,再求后面一半数组的逆序数,然后求前面一半数组比后面一半数组中大的数的个数（也就是逆序数），这三个过程加起来就是整体的逆序数目了。

代码实现

```
public class Solution {
```

```
    public static int iPairs(int[] array) {
        if (array == null)
            throw new IllegalArgumentException();
        // 创建辅助数组
        int length = array.length;
        int[] copy = new int[length];
        System.arraycopy(array, 0, copy, 0, length);
        int numberOfInversePairs = iPairs(array, copy, 0, length - 1);
        return numberOfInversePairs;
    }

    /**
     * @param array 未归并数组
     * @param copy  用于存储归并后数据的数组
     * @param begin 起始位置
     * @param end   结束位置
     * @return 逆序数
     * @author Thanos
     */
    public static int iPairs(int[] array, int[] copy, int begin, int end)
{
        if (begin == end)
            return 0;
        int mid = (begin + end) / 2;
        // 递归调用
        int left = iPairs(copy, array, begin, mid);
        int right = iPairs(copy, array, mid + 1, end);
        // 归并
        int i = mid, j = end, pos = end;
        int count = 0; // 记录相邻子数组间逆序数

        while (i >= begin && j >= mid + 1) {
            if (array[i] > array[j]) {
                copy[pos--] = array[i--];
                count += j - mid;
            } else
                copy[pos--] = array[j--];
        }

        while (i >= begin)
            copy[pos--] = array[i--];
        while (j >= mid + 1)
            copy[pos--] = array[j--];
```

```
        return left + right + count;
    }

    public static void main(String... args) {
        int test[] = {7, 5, 1, 6, 4};
        int count = iPairs(test);
        System.out.println(count + " ");
    }

}
```

36. 两个链表的第一个公共结点

题目描述

输入两个链表，找出它们的第一个公共结点。

题目分析

有两个链表，首先判断链表是否为空，如果都不为空，以其中一个链表为基础，遍历另一个链表，逐个进行比较。时间复杂度为 O(n^2)。

代码实现

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/public class Solution {
    public ListNode FindFirstCommonNode(ListNode pHead1, ListNode
pHead2) {
        // 如果存在空链表，说明没有公共结点
       if(pHead1 == null || pHead2 == null){
            return null;
        }
        // 以链表 pHead1 作为基本链表，取一个结点与 pHead2 中的所有结点
进行比较
```

```
        ListNode target = pHead1;
        while(target != null){
            // 避免修改 pHead2
            ListNode current = pHead2;
            while(current != null){
                if(target == current){
                    return target;
                }
                current = current.next;
            }
            target = target.next;
        }
        return null;
    }
}
```

优化方法 1：hash 法

初级方法虽然可以完成任务，但是时间复杂度较高，为了降低时间复杂度，我们采用 Hash 法。即通过将第二个链表存入 HashMap 中，遍历第一个链表，查看该结点是否在 HashMap 中来判断。时间复杂度为 O(m+n)，空间复杂度为 O(n).

代码实现

```
import java.util.Map;import java.util.HashMap;/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/public class Solution {
    public ListNode FindFirstCommonNode(ListNode pHead1, ListNode
pHead2) {
        if (pHead1 == null || pHead2 == null){
            return null;
        }
        // 将第二个链表存入 HashMap 中
        Map<ListNode,Integer> map = new HashMap<>();
        while (pHead2 != null){
            map.put(pHead2,2);
            pHead2 = pHead2.next;
        }
```

```
        // 遍历第一个链表，查找相同结点
        while (pHead1 != null){
            if (map.get(pHead1) != null){
                return pHead1;
            }
            pHead1= pHead1.next;
        }
        return null;
    }
}
```

37.数字在排序数组中出现的次数

题目描述

统计一个数字在排序数组中出现的次数。

思路分析

最简单的方法就是遍历一边数组，比较每一个数字，统计指定数字出现的次数。这样的解决办法时间复杂度为O(n)，空间复杂度为O(1)。

代码实现

```
public class Solution {
    public int GetNumberOfK(int [] array , int k) {
       int count = 0;
       for(int i = 0; i < array.length; i ++){
           if(k == array[i]){
               count ++;
           }
       }
        return count;
    }
}
```

优化方法

上面的方法虽然简单，但是不是最优的。既然是有序的数组，我们应该很容易想到二分查找的办法，通过二分查找找到该数字的起始地址以及结束地址，然后运算即可获取答案。

优化实现

```
public static int getNumberOfK(int[] array, int k) {
        if (array == null || array.length == 0) {
            return 0;
        }
        int startIndex = -1;
        int endIndex = -1;
        startIndex = getStartIndex(array, k, 0, array.length - 1);
        endIndex = getEndIndex(array, k, 0, array.length - 1);
        if (startIndex != -1 && endIndex != -1) {
            return (endIndex - startIndex + 1);
        }
        return 0;
    }

    /**
     * 用递归的方法查找首次出现 k 的位置
     *
     * @param array
     * @param k
     * @param start
     * @param end
     * @return
     */
    public static int getStartIndex(int[] array, int k, int start, int
end) {
        if (start > end) {
            return -1;
        }
        // 防止相加后溢出
        int mid = start + ((end - start) >> 1);
        if (array[mid] < k) {
            return getStartIndex(array, k, mid + 1, end);
        } else if (array[mid] > k) {
            return getStartIndex(array, k, start, mid - 1);
        } else if (mid - 1 >= 0 && array[mid - 1] == k) {
            // 防止找到的不是第一个
            return getStartIndex(array, k, start, mid - 1);
        } else {
            return mid;
        }
    }
```

```
    /**
     * 用循环的方法查找最后出现 k 的位置
     *
     * @param array
     * @param k
     * @param start
     * @param end
     * @return
     */
    public static int getEndIndex(int[] array, int k, int start, int end)
{
        int length = array.length;
        int mid = start + ((end - start) >> 1);
        while (start <= end) {
            if (array[mid] < k) {
                start = mid + 1;
            } else if (array[mid] > k) {
                end = mid- 1;
            } else if (mid + 1 < length && array[mid + 1] == k) {
                // 防止找到的不是最后一个
                start = mid + 1;
            } else {
                return mid;
            }
            mid = start + ((end - start) >> 1);
        }
        return -1;
    }
```

38.二叉树的深度

题目描述

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。

思路分析

通过递归，分别求出树的左子树与右子树的深度，然后比较获得最大的深度并返回之即可。

```
/**
```

```
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 leftDepth = TreeDepth(root.left);
        int rightDepth = TreeDepth(root.right);

        return 1 + (leftDepth > rightDepth ? leftDepth : rightDepth);
    }
}
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
           val(x), left(NULL), right(NULL) {
    }
};*/class Solution {public:
    int TreeDepth(TreeNode* pRoot)
    {
       if(pRoot == nullptr){
            return 0;
        }

        int leftDepth = TreeDepth(pRoot -> left);
        int rightDepth = TreeDepth(pRoot -> right);

        return 1 + (leftDepth > rightDepth ? leftDepth : rightDepth);
    }
};
```

39.平衡二叉树

题目描述

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

思路分析

通过递归逐层检查二叉树的子树是否为平衡二叉树。

代码实现

```
class Solution {
public:
 bool IsBalanced_Solution(TreeNode* pRoot) {
int depth = 0;
return IsBalanced(pRoot,depth);
} bool IsBalanced(TreeNode *root, int & dep){
 if(root == NULL){
return true;
}
int left = 0;
int right = 0;
if(IsBalanced(root->left,left) && IsBalanced(root->right, right)){
 int dif = left - right;
 if(dif<-1 || dif >1)
return false;
 dep = (left > right ? left : right) + 1;
 return true;
}
 return false;
 }
};
```

40.数组中只出现一次的数字

题目描述

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。

思路分析

* 除了有两个数字只出现了一次，其他数字都出现了两次。异或运算中，任何一个数字和自己本身异或都是 0，任何一个数字和 0 异或都是本身。
* 如果尝试把原数组分成两个子数组，且刚好每个子数组中各自包含一个只出现一次的数字。则在该前提下，每个子数组中，只有一个数字出现了一次，其他数字都出现了两次。
* 针对每个子数组，从头到尾依次异或每个数字，则最后留下来的就是只出现了一次的数字。因为出现两次的都抵消掉了。
* 怎样实现子数组的划分呢。若对原数组从头到尾的进行异或，则最后得到的结果就是两个只出现一次的数字的异或运算结果。这个结果的二进制表示中，至少有一位为 1，因为这两个数不相同。该位记为从最低位开始计数的第 n 位。
* 则分组的标准定为从最低位开始计数的第 n 位是否为 1。因为出现两次的同一个数字，各个位数上都是相同的，所以一定被分到同一个子数组中，且每个子数组中只包含一个出现一次的数字。

代码实现

```
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;
    }

}
```

41.和为 S 的连续正数序列

题目描述

小明很喜欢数学,有一天他在做数学作业时,要求计算出 9~16 的和,他马上就写出了正确答案是 100。但是他并不满足于此,他在想究竟有多少种连续的正数序列的和为 100(至少包括两个数)。没多久,他就得到另一组连续正数和为 100 的序列:18,19,20,21,22。现在把问题交给你,你能不能也很快的找出所有和为 S 的连续正数序列? Good Luck!

算法分析

假设序列的起始值为 small,序列的结束值为 big。那么存在一个中间值 middle = (sum + 1) / 2 使得 small < middle 一定成立。如果当前序列的和比 sum 小的时候，那么就依次增加 big(big ++),使之趋近于 sum。如果当前值比 sum 大，那么就依次减掉 small，也使之趋近于 sum。这样一直循环就可以找到所有的序列。

算法实现

```
import java.util.ArrayList;public class Solution {
    public ArrayList<ArrayList<Integer> > FindContinuousSequence(int
sum) {
        ArrayList<ArrayList<Integer> > result = new ArrayList<>();
        if(sum < 3){
            return result;
        }
        int small = 1;
        int big = 2;
```

```
            int middle = (sum + 1) / 2;
            int curSum = small + big;
             // 当最小值不超过中间值的时候，一直查找
             while(small < middle){
                 // 如果当前的和就是，保存当前序列
                 if(curSum == sum){
                     result.add(getSequeuence(small,big));
                 }
                 // 如果当前值比 sum 大，依次减掉较小的值
                 while(curSum > sum && small < middle){
                     curSum -= small;
                     small ++;
                     if(curSum == sum){
                         result.add(getSequeuence(small,big));
                     }
                 }
                 // 正常情况下，增加较大的值
                 big ++;
                 curSum += big;
             }
             return result;
        }

        /**
        * 根据最小值与最大值，返回连续的数字序列
        */
        private ArrayList<Integer> getSequeuence(int small,int big){
            ArrayList<Integer> arr = new ArrayList();
            for(int i=small; i<=big; i++){
                arr.add(i);
            }
            return arr;
        }
    }
```

42.和为 S 的两个数字

题目描述

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，是的他们的和正好是 S，如果有多对数字的和等于 S，输出两个数的乘积最小的。

思路分析

由于是递增排序的数组，所以，我们可以使用两个指针，一个从前往后指向较小的数，一个从后往前指向较大的数，这样找到的两个数的乘积必然最小。当两数的和大于 S 时，较大的数字指针向左移动，当两数的和小于 S 时，较小的数字指针向右移动，当两数靠近时，查找结束。

代码实现

```
import java.util.ArrayList;public class Solution {
    public ArrayList<Integer> FindNumbersWithSum(int [] array,int sum)
{
        ArrayList<Integer> result = new ArrayList();
        if(array == null || array.length < 2){
            return result;
        }
        // small 指向小数字位置的指针
        int small = 0;
        // big 指向大数字位置的指针
        int big = array.length - 1;
        int curSum = array[small] + array[big];
        // 让两个数字的位置不断靠近
        while(small < big){
            if(curSum == sum){
                result.add(array[small]);
                result.add(array[big]);
                return result;
            }
            if(curSum > sum){
                big--;
            }
            if(curSum < sum){
                small ++;
            }
            curSum = array[small] + array[big];
        }
        return result;
    }
}
```

43.左旋转字符串

题目描述

汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。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”。是不是很简单？OK，搞定它！

思路分析

n 作为字符串左旋的位置，其实就是将字符串[0,n)放置在字符串[n,length)的后面，从而组成新的字符串。方法有很多种。

代码实现

```
public class Solution {
    public String LeftRotateString(String str,int n) {
        if(str == null || str.length() <= 1){
            return str;
        }
        int length = str.length();
        StringBuilder sb = new StringBuilder();
        sb.append(str.substring(n,length));
        sb.append(str.substring(0,n));
        return sb.toString();
    }
}
```

简洁方法

```
public class Solution {
    public String LeftRotateString(String str,int n) {
        if(str == null || str.length() <= 1){
            return str;
        }
        str += str;
        return str.substring(n,length + n);
    }
}
```

44.翻转单词顺序列

题目描述

牛客最近来了一个新员工 Fish，每天早晨总是会拿着一本英文杂志，写些句子在本子上。同事 Cat 对 Fish 写的内容颇感兴趣，有一天他向 Fish 借来翻看，但却读不懂它的意思。例如，“student. a am I”。后来才意识到，这家伙原来把句子单词的顺序翻转了，正确的句子应该是“I am a student.”。Cat 对一一的翻转这些单词顺序可不在行，你能帮助他么？

题目分析

1.判断字符串是否为 null。
2.判断字符串是否全为空字符。
3.按照空字符将字符串分离成字符串数组。
4.将数组从后向前重新排列成一个字符串。

代码实现

```
public class Solution {
    public String ReverseSentence(String str) {
        if(str == null){
            return null;
        }
        // 如果字符串中全部都是空字符，直接返回
        if(str.trim().equals("")){
            return str;
        }
        StringBuilder sb = new StringBuilder();
        String[] strArr = str.split(" ");
        for (int i = strArr.length - 1; i >= 0; i--) {
            // 最后一个字符末尾无需加空格
            if(i == 0){
                sb.append(strArr[i]);
            }else{
                sb.append(strArr[i]).append(" ");
            }
        }
        return sb.toString();
    }
}
```

45.扑克牌顺子

题目描述

LL 今天心情特别好,因为他去买了一副扑克牌,发现里面居然有 2 个大王,2 个小王(一副牌原本是 54 张^_^)...他随机从中抽出了 5 张牌,想测测自己的手气,看看能不能抽到顺子,如果抽到的话,他决定去买体育彩票,嘿嘿！！“红心 A,黑桃 3,小王,大王,方片 5”,“Oh My God!”不是顺子.....LL 不高兴了,他想了想,决定大\小 王可以看成任何数字,并且 A 看作 1,J 为 11,Q 为 12,K 为 13。上面的 5 张牌就可以变成“1,2,3,4,5”(大小王分别看作 2 和 4),“So Lucky!”。LL 决定去买体育彩票啦。 现在,要求你使用这幅牌模拟上面的过程,然后告诉我们 LL 的运气如何。为了方便起见,你可以认为大小王是 0。

思路分析

由于确定了是 5 张扑克牌，并且判定是否为顺子，那么我们可以利用 5 个连续数字的特性来求解。为了处理方便，我们需要先将数字进行排序，并且统计出 0 的个数，然后分情况讨论即可。
1.当 zeroNum = 0 时，说明五个数字要想连续，则 numbers[4]-numbers[0]=4 必然成立，否则不是连续的。
2.当 zeroNum = 1 时，说明有一个 0，这个 0 可以放在剩余数字的两头，也可以放在剩余数字中间任意位置。当 0 放在两头时,此时又 numbers[4]-numbers[1]=3 成立。当 0 放在剩余数字中间时，numbers[4]-numbers[1]=4 成立。得出范围即 numbers[4]-numbers[1] >=3 && numbers[4]-numbers[1]<=4 成立。
3.同理，当 zeroNum=2 时，可得出范围 numbers[4]-numbers[1] >=2 && numbers[4]-numbers[1]<=4 成立。
4.同理，当 zeroNum=3 时，可得出范围 numbers[4]-numbers[1] >=1 && numbers[4]-numbers[1]<=4 成立。
5.当 zeroNum >= 4 时，必定可以。

代码实现

```
import java.util.Arrays;
public class Solution {
    public boolean isContinuous(int [] numbers) {
        if(numbers == null || numbers.length < 5){
            return false;
        }
        int length = numbers.length;
       // 将数字进行排序
        Arrays.sort(numbers);
        int start = -1;
        for(int i=0; i < length; i++){
```

```
            if(numbers[i] == 0){
                start = i;
            }else{
                break;
            }
        }
        // 0 的数量
        int zeroNum = start + 1;
        if(zeroNum == 0){
            if( numbers[4] - numbers[0] == 4){
                return true;
            }else{
                return false;
            }
        }else if(zeroNum == 1){
            if((numbers[4] - numbers[1] >= 3) && (numbers[4] - numbers[1]
<= 4)){
                return true;
            }else{
                return false;
            }
        }else if(zeroNum == 2){
            if((numbers[4] - numbers[2] >= 2) && (numbers[4] - numbers[2]
<= 4) ){
                return true;
            }else{
                return false;
            }
        }else if(zeroNum == 3){
            if((numbers[4] - numbers[3] >= 1) && (numbers[4] - numbers[3]
<= 4)){
                return true;
            }else{
                return false;
            }
        }else if(zeroNum >= 4){
            return true;
        }
        return false;
    }
}
```

优化思路

上面虽然解决了问题，但是需要排序，并且后面判断复杂。通过总结规律，我们知道，这 5 个数字要想组成顺子，首先除了 0 之外，不可重数；其次非零的最大值与最小值之差最大可为 4。因而这个题目变得很简单。

代码实现

```
public class Solution {

    public static boolean isContinuous(int [] numbers){
        if(numbers == null || numbers.length < 5){
            return false;
        }
        int maxn = -1;
        int minn = 14;
        Set<Integer> set = new HashSet();
        for(Integer num : numbers){
            if(!set.add(num) && num !=0){
                // 说明有非零的对子存在，肯定不是顺子
                return false;
            }
            if (num != 0){
                if(maxn < num){
                    maxn = num;
                }
                if(minn > num){
                    minn = num;
                }
            }
        }
        if(maxn - minn <= 4){
            return true;
        }else{
            return false;
        }
    }

    public static void main(String[] args) {
        int[] num = {0,3,2,6,4};
        System.out.println(isContinuous(num));
    }
}
```

46.孩子们的游戏(圆圈中最后剩下的数)

题目描述

每年六一儿童节,牛客都会准备一些小礼物去看望孤儿院的小朋友,今年亦是如此。HF 作为牛客的资深元老,自然也准备了一些小游戏。其中,有个游戏是这样的: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唱首歌,然后可以在礼品箱中任意的挑选礼物,并且不再回到圈中,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,可以不用表演,并且拿到牛客名贵的“名侦探柯南”典藏版(名额有限哦!!^_^)。请你试着想下,哪个小朋友会得到这份礼品呢?(注：小朋友的编号是从 0 到 n-1)

思路分析

1.使用链表将 n 个孩子用 0~n 表示出来。
2.用一个指针指向当前报数的孩子，如果当前孩子报数是 m-1，那么将这个孩子移除队列。
3.如果当前孩子是最后一个孩子，就将下一个孩子位置重置为队首的孩子。

代码实现

```
import java.util.List;import java.util.LinkedList;
public class Solution {
    public int LastRemaining_Solution(int n, int m) {
        //
        if(n < 1 || m < 1){
            return -1;
        }
        List<Integer> list = new LinkedList();
        for(int i=0; i < n; i++){
            list.add(i);
        }
        // 用 current 指示当前小朋友的位置,并且从 0 号小朋友开始报数,这里可以改成从任意一个小朋友开始报数
        int current = 0;
        while(list.size() > 1){
            // 寻找下一个需要出队的小朋友
            for(int i = 1; i < m; i++){
                current ++;
                // 下一个报数的小朋友在队头
                if(current == list.size()){
                    current = 0;
```

```
                }
            }
            // 找到了一个小朋友，出队
            list.remove(current);
            // 如果 current 代表的小朋友是排在最后面的那个小朋友，那么
下一个报数的小朋友在队头
            if(current == list.size()){
                current = 0;
            }
        }
        // 寻找完毕
        return list.get(0);
    }
}
```

47.求 1+2+3+…+n

题目描述

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）。

思路分析

用逻辑与(&&)递归实现。

代码实现

```
public class Solution {

    public int sum(int n) {
        int sum = n;
        boolean ans = (n > 0) && ((sum += sum(n - 1)) > 0);
        return sum;
    }

    public static void main(String[] args) {
        Solution s = new Solution();
        System.out.println(s.sum(5));
    }

}
```

48.不用加减乘除求和

题目描述

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。

思路解析

异或运算示例
 0000 0011
^0000 0001
 0000 0010

1.首先将两个数字进行异或运算，得到的结果是如果两数对应位不同为 1 且不同为 0，那么该位上对应的值相加后被保留。 与运算实例
0000 0011 &0000 0001 0000 0001

2.再将两个数字进行与运算，运算结果是，如果对应位都为 1，那么该位在异或运算中变为 0，但是需要进位。 根据上面的运算结果，我们可以将第二步运算结果向左移动 1 位，与第一步运算结果继续进行第一步第二步。直到第二步运算结果为 0 时，结束即可。

代码实现

```
public class Solution {
    public int Add(int num1,int num2) {
        if(num2 == 0){
            return num1;
        }
        // 对应位为 01，10 运算后结果为 1，对应位为 00，11 运算后结果为 0
        int xOrResult = num1 ^ num2;
        // 对应位 11 的抑或后需要进位
        int carry = (num1 & num2) << 1;
        return Add(xOrResult,carry);
    }
}
```

49.把字符串转换成整数

题目描述

将一个字符串转换成一个整数，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

思路分析

1.逐个字符判断，如果是数字字符，则继续，否则返回 0.
2.注意首字符有可能为 '+'、'-' 符号，特殊处理，并标记，返回的时候区分正负数。

代码实现

```
public class Solution {

    public static int StrToInt(String str) {
        if(str == null || str.length() == 0){
            return 0;
        }
        int length = str.length();
        int result = 0;
        boolean positive = true;
        for(int i=0; i < length; i++){
            if(i == 0 && str.charAt(i) == '+'){
                continue;
            }
            if(i == 0 && str.charAt(i) == '-'){
                positive = false;
                continue;
            }
            if(!isDigit(str.charAt(i))){
                return 0;
            }
            result = result * 10 + (str.charAt(i) - '0');
        }
        return positive ? result : -result;
    }

    private static boolean isDigit(char num){
        if(num >= '0' && num <= '9'){
            return true;
        }
```

```
        return false;
    }

    public static void main(String[] args) {
        System.out.println(StrToInt("-123"));
    }
}
```

50.数组中重复的数字

题目描述

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。

算法分析

这是一个数组问题，题目也要求输出第一个重复的数字，因而我们不妨用冒泡排序的思想，从第一个开始检查，当遇到重复的数字时就停止检查，并保存该数字。

算法实现

```
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)
{
        if(length <= 1){
            return false;
        }
```

```
        for(int i = 0; i < length - 1; i++ ){
            for(int j = i + 1; j < length; j ++){
                if(numbers[i] == numbers[j]){
                    duplication[0] = numbers[i];
                    return true;
                }
            }
        }
        return false;
    }
}
```

算法优化

利用冒泡排序的思想虽然可以解决，但是时间复杂度较高为O(n^2)，我们可以考虑借助一个辅助数组，由于数字的取值范围是0-n-1,因而我们可以对每个出现的数字进行标记，这样在遍历时检查标记即可实现获取第一个重复数字的需求。其时间复杂度为O(n)，空间复杂度为O(n)。

优化实现

```
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)
{
        if(length <= 1){
            return false;
        }
        boolean[] tag = new boolean[length];
        for(int i = 0; i < length; i ++){
            if(tag[numbers[i]] == true){
                duplication[0] = numbers[i];
                    return true;
            }
            tag[numbers[i]] = true;
```

```
        }
        return false;
    }
}
```

51.构建乘积数组

题目描述

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。

思路分析

根据书中提供的思路，可以将 n 个 A 数组组成如图所示的矩阵。这样 B[i]的计算可分成两部分，即矩阵的下三角部分和矩阵的上三角部分。 下三角部分计算: 令 B[0]=1 则 B[1]=B[0]*A[0] B[2]=B[1]*A[1] B[i] = B[i-1] * A[i-1] i 取值范围是[1,length) 下三角部分计算：令 right=1 则当 j = length-2 时,right *= A[j+1]， B[j] *= right; j = length-3 时， right *= A[j+1]， B[j] *= right; j = 0 时， right *= A[1],B[0] *= right; 如此循环两遍即可计算出数组 B。

代码实现

```
import java.util.ArrayList;public class Solution {
    // 记得画图啊
    public int[] multiply(int[] A) {
        // 根据矩阵的特点求解
        int length = A.length;
        int[] B = new int[length];
        // 计算下三角矩阵
        for(int i = 1; i < length; i++){
            B[0] = 1;
            B[i] = B[i - 1] * A[i-1];
        }
        // 计算下三角所有的值
        int right = 1;
        for(int j = (length - 2); j >=0; j--){
            right *= A[ j +1];
            B[j] *= right;
        }
        return B;
    }
```

```
}
```

52.正则表达式匹配

题目描述

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"abaca"匹配，但是与"aa.a"和"ab*a"均不匹配

思路分析

每次分别在 str 和 pattern 中取一个字符进行匹配，如果匹配，则匹配下一个字符，否则，返回不匹配。 设匹配递归函数 match(str, pattern)。 如果模式匹配字符的下一个字符是‘’： 如果 pttern 当前字符和 str 的当前字符匹配，：有以下三种可能情况 （1）pttern 当前字符能匹配 str 中的 0 个字符：match(str, pattern+2) （2）pttern 当前字符能匹配 str 中的 1 个字符：match(str+1, pattern+2) （3）pttern 当前字符能匹配 str 中的 多 个字符：match(str+1, pattern) 如果 pttern 当前字符和和 str 的当前字符不匹配 pttern 当前字符能匹配 str 中的 0 个字符：(str, pattern+2) 如果模式匹配字符的下一个字符不是‘’，进行逐字符匹配。 对于 ‘.’ 的情况比较简单，'.'和一个字符匹配 match(str+1, pattern+1) 另外需要注意的是：空字符串”” 和 “.*” 是匹配的

代码实现

```
public class Solution {

    public static boolean match(char[] str, char[] pattern) {
        if (str == null || pattern == null) {
            return false;
        }
        return matchCore(str, 0, pattern, 0);
    }

    private static boolean matchCore(char[] str, int s, char[] pattern,
int p) {
        if (s == str.length && p == pattern.length) {
            // 匹配结束
            return true;
        }
```

```
        if (s != str.length && p == pattern.length) {
            // pattern 不够
            return false;
        }

        // 当 pattern 的下一个字符是 *
        if (p + 1 < pattern.length && pattern[p + 1] == '*') {
            // 如果当前的字符串相同或者 pattern 中的字符为'.'
            if (s != str.length && str[s] == pattern[p] || s != str.length
&& pattern[p] == '.') {
                return matchCore(str, s, pattern, p + 2)/*匹配 0 个*/
                        || matchCore(str, s + 1, pattern, p + 2)/*匹配
1 个*/
                        || matchCore(str, s + 1, pattern, p)/*匹配任意
个*/;
            } else {
                return matchCore(str, s, pattern, p + 2);
            }
        }
        if (s != str.length && str[s] == pattern[p] || s != str.length
&& pattern[p] == '.') {
            return matchCore(str, s + 1, pattern, p + 1);
        }
        return false;
    }

    public static void main(String[] args) {
        System.out.println(match("".toCharArray(),
".*".toCharArray()));
    }
}
```

53.表示数值的字符串

题目描述

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。
输出描述
true false

思路分析

由**题目描述**可知，一个数值字符串可能是常规的字符串，也可能是科学记数法表示的数值型字符串。就科学计数法表示可以发现，在字符 'E'、'e' 的左侧表示的与常规数值型字符串一样，在字符 'E'、'e' 的右侧，是整数。因而可以将数值型字符串按照科学计数法表示的分成两半分别检查即可。
辅助函数

boolean contains(char[] str, char ch)：判断字符串中是否包含某个字符

```
    public boolean contains(char[] str, char ch) {
        for (int i = 0; i < str.length; i++) {
            if (str[i] == ch) {
                return true;
            }
        }
        return false;
    }
```

char[] subChars(char[] str, int startIndex, int endIndex)：截取字符数组中指定[startIndex,endIndex)中的字符

```
    public char[] subChars(char[] str, int startIndex, int endIndex) {
        char[] ret = new char[endIndex - startIndex];
        for (int i = startIndex; i < endIndex; i++) {
            ret[i - startIndex] = str[i];
        }
        return ret;
    }
```

boolean isDigit(char[] num)：判断一个字符数组是否是纯数字

```
    public boolean isDigit(char[] num) {
        for (int i = 0; i < num.length; i++) {
            if (num[i] < '0' || num[i] > '9') {
                return false;
            }
        }
        return true;
    }
```

boolean starWith(char[] str, char ch)：字符数组以字符 ch 开头

```
    public boolean starWith(char[] str, char ch) {
        if (str[0] == ch)
            return true;
        return false;
    }
```

boolean isDecimal(char[] str)：判断是否为合法的数值（非科学记数法表示）

```
    public boolean isDecimal(char[] str) {
        if (starWith(str, '-') || starWith(str, '+')) {
            str = subChars(str, 1, str.length);
        }
        if (contains(str, '.')) {// 如果是小数
            int posE = -1;
            for (int i = 0; i < str.length; i++) {
                if (str[i] == '.') {
                    posE = i;
                    break;
                }
            }
            if (posE == 0 || posE == str.length - 1) {
                return true;
            }
            char[] left = new char[posE];
            char[] right = new char[str.length - posE - 1];
            left = subChars(str, 0, posE);
            right = subChars(str, posE + 1, str.length);
            return isDigit(left) && isDigit(right);

        } else {// 如果不是小数
            return isDigit(str);
        }
    }
```

代码实现

```
    public boolean isNumeric(char[] str) {
      // 如果这个数使用科学计数法表示，将数字分成两部分判断
      int posE = -1;
      if (contains(str, 'E') || contains(str, 'e')) {
          for (int i = 0; i < str.length; i++) {
              if (str[i] == 'E' || str[i] == 'e') {
                  posE = i;
                  break;
              }
          }
          // 如果 'E' || 'e'在开始与结尾的位置，说明错误
          if (posE == 0 || posE == str.length - 1) {
              return false;
          }
```

```
            char[] left = new char[posE];
            char[] right = new char[str.length - posE - 1];
            left = subChars(str, 0, posE);
            right = subChars(str, posE + 1, str.length);
            // left 判断与非科学计数法一样

            // right 判断必须为整数
            if (starWith(right, '-') || starWith(right, '+')) {
                right = subChars(right, 1, right.length);
            }
            return isDecimal(left) && isDigit(right);
        } else {
            return isDecimal(str);
        }
    }
```

54.字符流中第一个不重复的字符

题目描述

一个链表中包含环，请找出该链表的环的入口结点。

思路分析

这道题目很简单，由于单链表有环，因而遍历单链表时，会回到环入口。即如果遍历时第一次发现遍历过的结点必定就是环入口结点。java 中 HashSet 的 add 方法可轻松判断。当 add 新的元素时，返回 true，否则返回 false。

代码实现

```
import java.util.Set;
import java.util.HashSet;/*
 public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}
*/public class Solution {
```

```
    // 灵机一动，HashSet 真是个好东西
    public ListNode EntryNodeOfLoop(ListNode pHead)
    {
        if(pHead == null || pHead.next == null){
            return null;
        }
        ListNode current = pHead;
        Set<ListNode> set = new HashSet();
        // 当 set 中不存在当前结点，就把当前结点保存在 Set 中
        while(set.add(current)){
            current = current.next;
        }
        // 当放不进去了，说明该结点 Set 中已经存在，即链表的环入口结点
        return current;
    }
}
```

55.链表中环的入口结点

题目描述

一个链表中包含环，请找出该链表的环的入口结点。

思路分析

使用 Java 中的 HashSet 轻松找到环的入口结点。

代码实现

```
import java.util.HashSet;

class ListNode {
    ListNode next;
    Integer val;
}

public class Solution {

    public ListNode EntryNodeOfLoop(ListNode pHead) {
        HashSet<ListNode> set = new HashSet<ListNode>();
        while (pHead != null) {
```

```
            if (!set.add(pHead)) {
                return pHead;
            }
            pHead = pHead.next;
        }
        return null;
    }

}
```

56.删除链表中重复的结点

题目描述

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

思路分析

1.两个指针，一个指向前一个节点 preNode，另一个指向当前节点 node，如果遇到相等的节点，node 向后移动，preNode 不动，存下 node.val 方便后面的比较，直到遇到 node 和 node.next 不相等，preNode 就可以指向 node.next 注意：链表开头可能就开始有重复的节点，所以默认 preNode=null，在后面的给 preNode 赋值的时候，若 preNode 为 null，那就把 pHead 设置为 node.next

代码实现

```
import java.util.Set;import java.util.LinkedHashSet;/*
 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;
```

```
        }
        ListNode node = pHead;
        ListNode preNode = null;
        while(node != null){
            // 如果当前结点和下一个结点相同
            if(node.next != null && node.val == node.next.val){
                int val = node.val;
                //继续寻找，直到找打一个不同的结点
                while(node.next != null && node.next.val == val){
                    node = node.next;
                }
                // 此时 node 为相同结点的最后一个结点
                if(preNode == null){
                    pHead = node.next;
                }else{
                    preNode.next = node.next;
                }
            }else{
                preNode = node;
            }
            node = node.next;
        }
        return pHead;
    }
}
```

57.二叉树的下一个结点

题目描述

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。

思路分析

对于一棵二叉树，我们可以知道：
* 有右子树的，那么下个结点就是右子树最左边的点；
* 没有右子树的，也可以分成两类，如果是父节点左孩子 ，那么父节点就是下一个节点 ；如果是父节点的右孩子找他的父节点的父节点的父节点…直到当前结点是其父节点的左孩子位置。如果没有，那么他就是尾节点。

代码实现

```
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;
        //如果有右子树，则找右子树的最左节点
        if (node.right != null) {
            node = node.right;
            while (node.left != null) node = node.left;
            return node;
        }
        //没右子树，则找第一个当前节点是父节点左孩子的节点
        while (node.next != null) {
            if (node.next.left == node) return node.next;
            node = node.next;
        }
        return null;   //退到了根节点仍没找到，则返回 null
    }
}
```

58.对称的二叉树

题目描述

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。

思路分析

判断每个节点是否相等，每个节点的左子树是否与右子树相等即可。

代码实现

```
class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

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

    }

}

public class Solution {
    boolean isSymmetrical(TreeNode r1, TreeNode r2) {
        if (r1 == null && r2 == null)
            return true;
        if (r1 == null || r2 == null)
            return false;
        return r1.val == r2.val && isSymmetrical(r1.left, r2.right) &&
isSymmetrical(r1.right, r2.left);
    }

    boolean isSymmetrical(TreeNode pRoot) {
        return isSymmetrical(pRoot, pRoot);
    }
}
```

59.按之字形顺序打印二叉树

题目描述

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。

思路分析

* 使用两个栈来分别存储奇数层节点和偶数层节点。
* 注意两个栈的插入顺序是不同的。

* 对于奇数层来说，也就是从左往右的顺序，先添加左子树，然后添加右子树。对于偶数层，刚好相反，先添加右子树，然后添加左子树。

代码实现

```
class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

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

    }

}

public class Solution {
    boolean isSymmetrical(TreeNode r1, TreeNode r2) {
        if (r1 == null && r2 == null)
            return true;
        if (r1 == null || r2 == null)
            return false;
        return r1.val == r2.val && isSymmetrical(r1.left, r2.right) &&
isSymmetrical(r1.right, r2.left);
    }

    boolean isSymmetrical(TreeNode pRoot) {
        return isSymmetrical(pRoot, pRoot);
    }
}
```

60.把二叉树打印成多行

题目描述

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。

思路分析

通过队列保存遍历到的结点，依次打印出每层节点，遇到空节点跳过。

代码实现

```
class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

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

    }

}

public class Solution {
    ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {
        ArrayList<ArrayList<Integer>> arrs = new
ArrayList<ArrayList<Integer>>();
        if (pRoot == null) {
            return arrs;
        }
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.add(pRoot);
        int last = q.size(), count = 0;
        while (!q.isEmpty()) {
            count = 0;
            last = q.size();
            ArrayList<Integer> arr = new ArrayList<Integer>();
            while (count < last) {
                TreeNode tr = q.poll();
                count++;
                arr.add(tr.val);
                if (tr.left != null) {
                    q.add(tr.left);
                }
                if (tr.right != null) {
                    q.add(tr.right);
                }
            }
            arrs.add(arr);
        }
        return arrs;
    }
}
```

61.序列化二叉树

题目描述

请实现两个函数，分别用来序列化和反序列化二叉树。

思路分析

* 利用前序遍历序列化二叉树，然后从记录的字符串中反序列化二叉树。
* 遇到空节点需要用特殊字符加以标记。如“#”

代码实现

```
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) {
        StringBuilder s = new StringBuilder();
        if (root == null) {
            s.append("#,");
            return s.toString();
        }
        s.append(root.val + ",");
        s.append(serialize(root.left));
        s.append(serialize(root.right));
        return s.toString();
    }

    TreeNode deserialize(String str) {
        index++;
        String[] DLRseq = str.split(",");
```

```
        TreeNode leave = null;
        if (!DLRseq[index].equals("#")) {
            leave = new TreeNode(Integer.valueOf(DLRseq[index]));
            leave.left = deserialize(str);
            leave.right = deserialize(str);
        }
        return leave;
    }
}
```

62.二叉搜索树的第 k 个结点

题目描述

给定一颗二叉搜索树，请找出其中的第 k 大的结点。例如， 5 / \ 3 7 /\ /\ 2 4 6 8 中，按结点数值大小顺序第三个结点的值为 4。

思路分析

中序遍历过程中，累加计算访问过的节点数目，当计数器等于要求的 k 时，则返回该节点。

代码实现

```
class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

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

    }

}

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;
    }
}
```

63.数据流中的中位数

题目描述

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

思路分析

由于是数据流，也就是说输入是动态变化的，所以数据结构最好选用动态数组，ArrayList 比较合适。在计算中位数前对集合进行排序，然后返回中位数即可。

代码实现

```
public class Solution {

    private List<Integer> data = new ArrayList<>();

    public void Insert(int num) {
        data.add(num);
    }

    public Double GetMedian() {
        Collections.sort(data);
        System.out.println(data);
        if ((data.size() & 1) == 1) {
            return data.get(data.size() / 2).doubleValue();
        }else {
```

```
            Double median = (data.get(data.size() / 2 - 1).doubleValue()
+ data.get(data.size() / 2).doubleValue())/ 2.0;
            return median;
        }
    }
}
```

优化方法

上述方法虽然简单，但是对于每一次计算中位数都需要将数组进行排序，因而时间复杂度较高，即 n * nlog(n)。 那么，有什么办法可以使得排序的时间复杂度变低呢？我们想到了堆这个数据结构。当数据流中的数为偶数个时，入最小堆，但是先入最大堆，将最大堆中堆顶的数据取出，放入最小堆中。当数据流中的数据为奇数个时，入最大堆，但是先入最小堆，将最小堆堆顶的数据取出，放入最大堆中。 这样，最大堆中的数据都比最小堆中的数据小，并且最大堆中堆顶元素与最小堆堆顶元素是排序后数据中间相邻的两个数。当数据流中的数据为奇数个时，直接取最小堆堆顶的元素，为偶数个时，取最大堆与最小堆堆顶元素的平均值。

代码实现

```
public class OptimizeSolution {

    private PriorityQueue<Integer> minHeap = new PriorityQueue<>();//
小顶堆
    private PriorityQueue<Integer> maxHeap = new PriorityQueue<>(15, new
Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return o2 - o1;
        }
    });
    private int count = 0;

    public void Insert(int num) {
        // 偶数个数
        if ((count & 1) == 0) {
            // 先入大根堆
            maxHeap.offer(num);
            // 获取大根堆中最大值
            Integer maxNum = maxHeap.poll();
            // 将大根堆中的最大值放入小根堆
            minHeap.offer(maxNum);
        } else {
```

```
            // 奇数时，先入小根堆
            minHeap.offer(num);
            // 获取小根堆中的最小值
            Integer minNum = minHeap.poll();
            // 将这个最小值放入大根堆
            maxHeap.offer(minNum);
        }
        count++;
    }

    public Double GetMedian() {
        if ((count & 1) == 0) {
            int min = minHeap.peek();
            int max = maxHeap.peek();
            return (min + max) / 2.0;
        } else {
            return minHeap.peek() / 1.0;
        }
    }

    public static void main(String[] args) {
        OptimizeSolution s = new OptimizeSolution();
        s.Insert(5);
        s.Insert(3);
        s.Insert(2);
        s.Insert(1);
        System.out.println(s.GetMedian());
    }
}
```

64.滑动窗口的最大值

题目描述

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]}。

思路分析

根据题意可知，我们需要一个可以保存 size 个数字，并且方便的删除最先进来的数字，以及将新的数字添加到末尾的数据结构，很容易就想到了队列这个数据结构。 即循环遍历数组的同时维护一个 size 大小的队列，当队列的大小为 size 时获取队列中的最大值，并将队头元素删除，遍历结束即可。

代码实现

```
import java.util.ArrayDeque;import java.util.ArrayList;import
java.util.Queue;
public class Solution {

    private static Queue<Integer> queue = new ArrayDeque<>();

    public static ArrayList<Integer> maxInWindows(int[] num, int size)
{
        ArrayList<Integer> list = new ArrayList();
        if (num == null || num.length < size) {
            return list;
        }
        int length = num.length;
        for (int i = 0; i < length; i++) {
            queue.offer(num[i]);
            if (queue.size() == size) {
                // 获取队列中的最大值
                int maxValue = 0x80000000;
                for (Integer number : queue) {
                    if (maxValue < number) {
                        maxValue = number;
                    }
                }
                list.add(maxValue);
                // 打印当前的窗口信息以及最大值
                System.out.println(queue + ":" + maxValue);

                // 删除队列头部的数字，并再添加一个呀
                queue.poll();
            }
        }
        return list;
    }

    public static void main(String[] args) {
```

```
        int[] number = {2, 3, 4, 2, 6, 2, 5, 1};
        System.out.println(maxInWindows(number, 3));
    }
}
```

65.矩阵中的路径

题目描述

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。 例如[a b c e s f c s a d e e]是 3*4 矩阵，其包含字符串”bcced”的路径，但是矩阵中不包含“abcb”路径，因为字符串的第一个字符 b 占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

思路分析

这题典型的回溯法，当前位置能不能走，要看后面的路径是否满足条件，当上下左右都不能走时，说明该位置不能走，要回退一步，如果回退到出发点，说明这个点作为起点行不通。代码如下，这里发现剑指 offer 的一个错误，剑指 offer 中存当前已经走的路程 pathLength 是用的引用类型，不满足时--pathLength，实际上用普通局部变量就行，返回上一层调用的函数时，局部变量会回到上一层函数的值，也不需要自减。

代码实现

```
public class Solution {
    public boolean hasPath(char[] matrix, int rows, int cols, char[] str)
{
        if (matrix == null || str == null || matrix.length < 1 ||
matrix.length < str.length) return false;
        boolean[] visited = new boolean[rows * cols];
        int curLength = 0;
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (coreHasPath(matrix, rows, cols, i, j, str, visited,
curLength)) return true;
            }
        }
        return false;
    }
```

```
    private boolean coreHasPath(char[] matrix, int rows, int cols, int
row, int col, char[] str, boolean[] visited, int curLength) {
        if (curLength == str.length) return true;
        boolean hasPath = false;
        if (row >= 0 && row < rows && col >= 0 && col < cols && !visited[row
* cols + col] && matrix[row * cols + col] == str[curLength]) {
            curLength++;
            visited[row * cols + col] = true;
            hasPath = coreHasPath(matrix, rows, cols, row - 1, col, str,
visited, curLength) ||
                    coreHasPath(matrix, rows, cols, row + 1, col, str,
visited, curLength) ||
                    coreHasPath(matrix, rows, cols, row, col - 1, str,
visited, curLength) ||
                    coreHasPath(matrix, rows, cols, row, col + 1, str,
visited, curLength);
            if (!hasPath) {
                visited[row * cols + col] = false; //return hasPath 回
到上一层调用，curLength 的值会自动回到上一层调用时的值
            }
        }
        return hasPath;
    }


    public static void main(String[] args) {
        char[] matrix = new char[]{'A', 'B', 'C', 'E', 'S', 'F', 'C', 'S',
'A', 'D', 'E', 'E'};
        char[] str = new char[]{'A', 'B', 'C', 'C', 'E', 'D'};
        Solution solution = new Solution();
        System.out.println(solution.hasPath(matrix, 3, 4, str));
    }
}
```

66.机器人的运动范围

题目描述

地上有一个 m 行 n 列的方格。一个机器人从坐标（0,0）的格子开始移动，它每次可以向左，向右，向上，向下移动一格，但不能进入行坐标和列坐标的位数之和大于 k 的格子。例如：当 k 为 18 时，机器人能够进入方格（35,37），因为 3+5+3+7 = 18；但它不能进入方格（35,38），因为 3 + 5+3+8 = 19.请问该机器人最多能到达多少个格子？

思路分析

这个方格也可以看出一个 m*n 的矩阵。同样在这个矩阵中，除边界上的格子之外其他格子都有四个相邻的格子。 机器人从坐标(0,0)开始移动。当它准备进入坐标为(i,j)的格子时，通过检查坐标的数位和来判断机器人是否能够进入。如果机器人能够进入坐标为(i,j)的格子，我们接着再判断它能否进入四个相邻的格子(i,j-1)、(i-1,j),(i,j+1)和(i+1,j)。

代码实现

```
public class Solution {
    public  int movingCount(int threshold, int rows, int cols) {
        // 定义一个指示数组，判断当前位置是否已经访问过
        boolean[][] visited = new boolean[rows][cols];
        int movingCount = movingCountCore(threshold, rows, cols, 0, 0,
visited);
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (visited[i][j]){
                    System.out.print(visited[i][j] + "  ");
                }else {
                    System.out.print(visited[i][j] + " ");
                }
            }
            System.out.println();
        }
        return movingCount;
    }

    /**
     * @param threshold
     * @param rows
     * @param cols
```

```
     * @param row
     * @param col
     * @return
     */
    private  int movingCountCore(int threshold, int rows, int cols, int
row, int col, boolean[][] visited) {
        int count = 0;
        if (check(threshold, rows, cols, row, col, visited)) {
            // 访问后做出标记
            visited[row][col] = true;
            count += 1 + movingCountCore(threshold, rows, cols, row, col
- 1, visited)/*左*/
                    + movingCountCore(threshold, rows, cols, row, col +
1, visited)/*右*/
                    + movingCountCore(threshold, rows, cols, row - 1,
col, visited)/*上*/
                    + movingCountCore(threshold, rows, cols, row + 1,
col, visited)/*下*/;
        }
        return count;
    }

    private boolean check(int threshold, int rows, int cols, int row, int
col, boolean[][] visited) {
        if (row >= 0 && row < rows && col >= 0 && col < cols
                && positionSum(row, col) <= threshold
                && !visited[row][col]) {
            // 说明可以访问这个点
            return true;
        } else {
            return false;
        }
    }

    /**
     * 计算当前坐标各数位之和
     *
     * @param row
     * @param col
     * @return
     */
    private int positionSum(int row, int col) {
        int sum = 0;
        while (row != 0) {
```

```
            sum += row % 10;
            row /= 10;
        }
        while (col != 0) {
            sum += col % 10;
            col /= 10;
        }
        return sum;
    }
}
```

感谢牛友征程（牛客 id：135186）提供此部分题解！