1.赋值运算函数

思路：
将返回值类型声明为该类型的引用
把传入的参数类型声明为常量引用
释放实例自身已有的内存
判断传入的参数和当前的实例是不是同一个实例

2.单例设计模式

3.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：从右上角或左下角开始找，逐行删除，或者用二分法查找

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

4.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：从后往前复制，数组长度会增加，或使用 StringBuilder、StringBuffer 类

```
public String replaceSpace(String str) {
    if (str == null)
        return null;
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < str.length(); i++) {
        if (String.valueOf(str.charAt(i)).equals(" ")) {
            sb.append("%20");
        }else {
            sb.append(str.charAt(i));
        }
    }
    return String.valueOf(sb);
}
```

5. 输入一个链表，从尾到头打印链表每个节点的值。

思路：借助栈实现，或使用递归的方法。
代码实现：

```
public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
    ArrayList<Integer> list = new ArrayList<>();
    if (listNode == null)
        return list;
    Stack<ListNode> stack = new Stack<>();
    while (listNode != null) {
        stack.push(listNode);
        listNode = listNode.next;
    }

    while (!stack.isEmpty()) {
        list.add(stack.pop().val);
    }
    return list;
}
```

6. 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：先找出根节点，然后利用递归方法构造二叉树
代码实现：

```
public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;
        TreeNode(int x) { val = x; }
 }
public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
      if (pre == null || in == null) {
            return null;
      }
      if (pre.length == 0 || in.length == 0) {
            return null;
      }
      if (pre.length != in.length) {
            return null;
      }
     TreeNode root = new TreeNode(pre[0]);
     for (int i = 0; i < pre.length; i++) {
           if (pre[0] == in[i]) {
              root.left = reConstructBinaryTree(
              Arrays.copyOfRange(pre,1,i+1),Arrays.copyOfRange(in,0,i));
              root.right = reConstructBinaryTree(
              Arrays.copyOfRange(pre,i+1,pre.length),Arrays.copyOfRange(in,i+1,i
        n.length));
            }
      }
      return root;
}
```

7.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：一个栈压入元素，而另一个栈作为缓冲，将栈 1 的元素出栈后压入栈 2 中。也可以将栈 1 中的最后一个元素直接出栈，而不用压入栈 2 中再出栈。

```
public void push(int node) {
      stack1.push(node);
}
public int pop() throws Exception {
      if (stack1.isEmpty() && stack2.isEmpty()) {
            throw new Exception("栈为空！ ");
      }
```

```
        if (stack2.isEmpty()) {
            while(!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }
```

8.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：利用二分法，找到中间的数，然后和最左边的值进行比较，若大于最左边的数，则最左边从 mid 开始，若小于最右边值，则最右边从 mid 开始。若左中右三值相等，则取 mid 前后值中较小的数。

代码实现：

```
    public int minNumberInRotateArray(int [] array) {
        if (array == null || array.length == 0)
            return 0;
        int left = 0;
        int right = array.length - 1;
        int mid = 0;

        while (array[left] >= array[right]) {
            if(right - left <= 1) {
                mid = right;
                break;
            }
            mid = (left + right)/2;
            if (array[left] == array[mid] && array[mid] == array[right]) {
                if (array[left+1] != array[right-1]) {
                    mid = array[left+1] < array[right-1] ? left+1:right-1;
                } else {
                   left++;
                   right--;
                }
            } else {
               if (array[left] <= array[mid]) {
                    left = mid;
               } else {
                    right = mid;
```

```
            }
        }
    }

    return array[mid];
}
```

9.1 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

思路：递归的效率低，使用循环方式。
代码实现：

```
public long fibonacci(int n) {
    long result=0;
    long preOne=1;
    long preTwo=0;
    if(n==0) {
        return preTwo;
    }
    if(n==1) {
        return preOne;
    }
    for (int i = 2; i <= n; i++) {
        result = preOne+preTwo;
        preTwo = preOne;
        preOne = result;
    }
    return result;
}
```

9.3 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：斐波那契数列思想
代码实现：

```
public int Fibonaccik(int n) {
    int number = 1;
    int sum = 1;
    if (n <= 0)
        return 0;
```

```
        if (n == 1 ) {
            return 1;
        }

        while (n-- >= 2) {
            sum += number;
            number = sum - number;
        }
        return sum;
    }
```

9.4 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：2^(n-1)
代码实现：

```
    public int JumpFloor2(int target) {
        return (int) Math.pow(2,target-1);
    }
```

10.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。

思路：a&(a-1)的结果会将 a 最右边的 1 变为 0，直到 a = 0，还可以先将 a&1 != 0，然后右移 1 位，但不能计算负数的值.
代码实现：

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

11.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。不得使用库函数，不需要考虑大数问题
思路：不能用==比较两个浮点数是否相等，因为有误差。考虑输入值的多种情况。
代码实现：

```
public double Power(double base, int exponent) {
    double res = 0;
    if (equal(base,0)) {
        return 0;
    }
    if (exponent == 0) {
        return 1.0;
    }
    if (exponent > 0) {
        res = mutiply(base,exponent);
    }else {
        res = mutiply(1/base,-exponent);
    }
    return res;
}
public double mutiply(double base, int e) {
    double sum = 1;
    for (int i = 0; i < e; i++) {
        sum = sum * base;
    }
    return sum;
}
public boolean equal(double a, double b) {
    if (a - b < 0.000001 && a - b > -0.000001) {
        return true;
    }
    return false;
}
```

12.打印 1 到最大的 n 位数

思路：考虑大数问题，使用字符串或数组表示。
代码实现：

```
public void printToMaxOfNDigits(int n) {
    int[] array=new int[n];
```

```
        if(n <= 0)
            return;
        printArray(array, 0);
    }
    private void printArray(int[] array,int n) {
        for(int i = 0; i < 10; i++) {
            if(n != array.length) {
                array[n] = i;
                printArray(array, n+1);
            } else {
                boolean isFirstNo0 = false;
                for(int j = 0; j < array.length; j++) {
                    if(array[j] != 0) {
                        System.out.print(array[j]);
                        if(!isFirstNo0)
                            isFirstNo0 = true;
                    } else {
                        if(isFirstNo0)
                            System.out.print(array[j]);
                    }
                }
                System.out.println();
                return ;
            }
        }
    }
```

13.O(1)时间删除链表节点

思路：将要删除节点的下一个节点的值赋给要删除的节点，然后指向下下一个节点

代码实现：

```
    public void deleteNode(ListNode head, ListNode deListNode) {
        if (deListNode == null || head == null)
            return;
        if (head == deListNode) {
            head = null;
        } else {
            // 若删除节点是末尾节点，往后移一个
            if (deListNode.nextNode == null) {
                ListNode pointListNode = head;
                while (pointListNode.nextNode.nextNode != null) {
                    pointListNode = pointListNode.nextNode;
                }
```

```
                pointListNode.nextNode = null;
            } else {
                deListNode.data = deListNode.nextNode.data;
                deListNode.nextNode = deListNode.nextNode.nextNode;
            }
        }
    }
```

14.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：每次只和前面一个数交换位置。或者利用辅助数组
代码实现：

```
    // 解法一：移动偶数位置，时间复杂度 O（n²），空间复杂度 O（1)
     public void reOrderArray(int[] array) {
        if (array == null || array.length == 0) {
            return ;
        }
        for (int i = 1; i < array.length; i++) {
            int j = i - 1;
            if (array[i] % 2 != 0) {
                while (j >= 0) {
                    if (array[j] % 2 != 0) {
                        break;
                    }
                    if (array[j] % 2 == 0) {
                        int t = array[j + 1];
                        array[j + 1] = array[j];
                        array[j] = t;
                        j--;
                    }
                }
            }
        }
    }
    // 解法二：双指针法，时间复杂度 O（n)，空间复杂度 O（1)
     public void reOrderArray(int[] array) {
        if (array == null || array.length == 0) {
            return;
```

```
        }
        int left = 0;
        int right = array.length - 1;
        while (left < right) {
            while (left < right && array[left] % 2 != 0) {
                left++;
            }
            while (left < right && array[right] % 2 == 0) {
                right--;
            }
            if (left < right) {
                int tmp = array[left];
                array[left] = array[right];
                array[right] = tmp;
            }
        }
    }
```

15.输入一个链表，输出该链表中倒数第 k 个结点。

扩展题：找中间节点，使用两个指针，一个走一步，一个走两步。找到中间节点
思路：定义一快一慢两个指针，快指针走 K 步，然后慢指针开始走，快指针到尾时，慢指针就找到了倒数第 K 个节点。
代码实现：

```
    public ListNode FindKthToTail(ListNode head,int k) {
        if (head == null || k <= 0) {
            return null;
        }
        ListNode fast = head;
        ListNode slow = head;
        while(k-- > 1) {
            if (fast.next != null)
                fast = fast.next;
            else
                return null;
        }
        while (fast.next != null) {
            fast = fast.next;
            slow = slow.next;
        }
        return slow;
    }
```

16.输入一个链表，反转链表后，输出链表的所有元素。

扩展题：输出反转后链表的头节点，定义三个指针反向输出。
思路：定义两个指针，反向输出
代码实现：

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

17.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。

思路：递归与非递归求解，小数放在前面。
代码实现：

```
public ListNode Merge(ListNode list1,ListNode list2) {
    if (list1 == null) {
        return list2;
    }
    if (list2 == null) {
        return list1;
    }
    ListNode newHead = null;
    if (list1.val <= list2.val) {
        newHead = list1;
        newHead.next = Merge(list1.next,list2);
    }else {
        newHead = list2;
        newHead.next = Merge(list1,list2.next);
    }

    return newHead;
}
```

18.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：若根节点相等，利用递归比较他们的子树是否相等，若根节点不相等，则利用递归分别在左右子树中查找。
代码实现：

```
public boolean HasSubtree(TreeNode root1,TreeNode root2) {
    boolean result = false;
    if (root2 != null && root1 != null) {
        if(root1.val == root2.val){
            result = doesTree1HaveTree2(root1,root2);
        }
        if (!result)
            return HasSubtree(root1.left,root2) ||
HasSubtree(root1.right,root2);
    }
    return result;
}
public boolean doesTree1HaveTree2(TreeNode node1, TreeNode node2) {
    if (node2 == null) {
        return true;
    }
    if (node1 == null) {
        return false;
    }
    if (node1.val != node2.val) {
        return false;
    }
    return doesTree1HaveTree2(node1.left,node2.left) &&
                doesTree1HaveTree2(node1.right,node2.right);
}
```

19.操作给定的二叉树，将其变换为源二叉树的镜像。

思路：使用递归或非递归方式交换每个节点的左右子树位置。
代码实现：

```
public void Mirror(TreeNode root) {
    if (root == null) {
        return;
    }
```

```
        Stack<TreeNode> stack = new Stack<>();
        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                TreeNode temp = root.left;
                root.left = root.right;
                root.right = temp;
                stack.push(root);
                root = root.left;
            }
            if (!stack.isEmpty()) {
                root = stack.pop();
                root = root.right;
            }
        }
    }
```

20.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：终止行号大于起始行号，终止列号大于起始列号，

代码实现：

```
    public ArrayList<Integer> printMatrix(int[][] matrix) {
        ArrayList<Integer> list = new ArrayList<>();
        if(matrix == null)
            return list;
        int start = 0;
        while(matrix[0].length > start*2 && matrix.length > start*2) {
            printOneCircle(matrix,start,list);
            start++;
        }
        return list;
    }
    private void printOneCircle(int[][] matrix,int start,ArrayList<Integer> list) {
        int endX = matrix[0].length - 1 - start; // 列
        int endY = matrix.length - 1 - start;  // 行
        // 从左往右
        for (int i = start; i <= endX; i++)
            list.add(matrix[start][i]);
        // 从上往下
        if (start < endY) {
            for (int i = start + 1; i <= endY; i++)
```

```
                list.add(matrix[i][endX]);
        }
        // 从右往左（判断是否会重复打印）
        if (start < endX && start < endY) {
            for (int i = endX - 1; i >= start; i--)
                list.add(matrix[endY][i]);
        }
        // 从下往上（判断是否会重复打印）
        if (start < endX && start < endY - 1) {
            for (int i = endY - 1; i >= start + 1; i--)
                list.add(matrix[i][start]);
        }
    }
```

21.定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的 min 函数。

思路：定义两个栈，一个存放入的值。另一个存最小值。
代码实现：

```
public void push(int node) {
    stack1.push(node);
    if (stack2.isEmpty()) {
        stack2.push(node);
    }else {
        if (stack2.peek() > node) {
            stack2.push(node);
        }
    }
}
public void pop() {
    if (stack1.pop() == stack2.peek()) {
        stack2.pop();
    }
}
public int top() {
    return stack1.peek();
}
public int min() {
    return stack2.peek();
}
```

22.输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。

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：用栈来压入弹出元素，相等则出栈。
代码实现：

```
public boolean IsPopOrder(int [] pushA,int [] popA) {
    if (pushA == null || popA == null) {
        return false;
    }
    Stack<Integer> stack = new Stack<>();
    int index = 0;

    for (int i = 0; i < pushA.length; i++) {
        stack.push(pushA[i]);
        while (!stack.isEmpty() && stack.peek() == popA[index]) {
            stack.pop();
            index++;
        }
    }
    return stack.isEmpty();
}
```

23.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。

思路：利用队列（链表）辅助实现。
代码实现：

```
public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
    ArrayList<Integer> list = new ArrayList<>();
    if (root == null) {
        return list;
    }
    LinkedList<TreeNode> queue = new LinkedList<>();
    queue.add(root);

    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();
        list.add(node.val);
```

```
            if (node.left != null) {
                queue.addLast(node.left);
            }
            if (node.right != null) {
                queue.addLast(node.right);
            }
        }
        return list;
}
```

24.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：先找到右子树的开始位置，然后分别进行左右子树递归处理。
代码实现：

```
    public boolean VerifySquenceOfBST(int[] sequence) {
        if (sequence == null || sequence.length == 0)
            return false;
        int rstart = 0;
        int length = sequence.length;
        for (int i = 0; i < length - 1; i++) {
            if (sequence[i] < sequence[length - 1])
                rstart++;
        }
        if (rstart == 0) {
            VerifySquenceOfBST(Arrays.copyOfRange(sequence,0,length-1));
        }else {
            for (int i = rstart; i < length - 1; i++) {
                if (sequence[i] <= sequence[length - 1]) {
                    return false;
                }
            }
            VerifySquenceOfBST(Arrays.copyOfRange(sequence,0,rstart));
            VerifySquenceOfBST(Arrays.copyOfRange(sequence,rstart,length - 1));
        }
        return true;
}
```

25.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：先保存根节点，然后分别递归在左右子树中找目标值，若找到即到达叶子节点，打印路径中的值
代码实现：

```
public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {
    if(root == null)
        return resultList;
    list.add(root.val);
    target -= root.val;
    if(target == 0 && root.left == null && root.right == null){
        resultList.add(new ArrayList<>(list));
    }else {
        FindPath(root.left,target);
        FindPath(root.right,target);
    }
    //每返回上一层一次就要回退一个节点
    list.remove(list.size()-1);
    return resultList;
}
```

26.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）
思路：先复制链表的 next 节点，将复制后的节点接在原节点后，然后复制其它的节点，最后取偶数位置的节点（复制后的节点）。
代码实现：

```
public RandomListNode Clone2(RandomListNode pHead) {
    if(pHead == null)
        return null;
    RandomListNode head = new RandomListNode(pHead.label) ;
    RandomListNode temp = head ;
    while(pHead.next != null) {
        temp.next = new RandomListNode(pHead.next.label) ;
        if(pHead.random != null) {
```

```
                temp.random = new RandomListNode(pHead.random.label) ;
            }
            pHead = pHead.next ;
            temp = temp.next ;
        }
        return head ;
}
```

27.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：定义一个链表的尾节点，递归处理左右子树，最后返回链表的头节点
代码实现：

```
public TreeNode Convert(TreeNode pRootOfTree) {
        TreeNode lastlist = covertNode(pRootOfTree,null);
        TreeNode pHead = lastlist;
        while (pHead != null && pHead.left != null) {
            pHead = pHead.left;
        }
        return pHead;
}
public TreeNode covertNode(TreeNode root, TreeNode lastlist) {
        if (root == null)
            return null;
        TreeNode cur = root;
        if (cur.left != null) {
            lastlist = covertNode(cur.left,lastlist);
        }
        cur.left = lastlist;
        if (lastlist != null) {
            lastlist.right = cur;
        }
        lastlist = cur;
        if (cur.right != null) {
            lastlist = covertNode(cur.right,lastlist);
        }
        return lastlist;
}
```

28.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：将当前位置的字符和前一个字符位置交换，递归。
代码实现：

```
public ArrayList<String> Permutation(String str) {
        ArrayList<String> result = new ArrayList<String>() ;
        if(str == null || str.length() == 0)
                return result;
        char[] chars = str.toCharArray() ;
        TreeSet<String> temp = new TreeSet<>() ;
        Permutation(chars, 0, temp);
        result.addAll(temp) ;
        return result ;
}
public void Permutation(char[] chars, int index, TreeSet<String> result) {
        if(chars == null || chars.length == 0 )
                return ;
        if (index < 0 || index > chars.length - 1)
                return;
        if(index == chars.length-1) {
                result.add(String.valueOf(chars)) ;
        }else {
                for(int i=index ; i<=chars.length-1 ; i++) {
                        swap(chars, index, i) ;
                        Permutation(chars, index+1, result);
                        //  回退
                        swap(chars, index, i) ;
                }
        }
}
public void swap(char[] c, int a,int b) {
        char temp = c[a];
        c[a] = c[b];
        c[b] = temp;
}
```

29.数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字

思路：将首次出现的数 count+1，与之后的数进行比较，相等则+1，否则—1，最后进行校验是否超过长度的一半。
代码实现：

```
public int MoreThanHalfNum_Solution(int [] array) {
	int maxCount = array[0];
	int number = array[0];
	int count = 1;
	for (int i = 1; i < array.length; i++) {
		if (number != array[i]) {
			if (count == 0) {
				number = array[i];
				count = 1;
			}else {
				count--;
			}
		}else {
			count++;
		}
		if (count == 1) {
			maxCount = number;
		}
	}
	// 验证
	int num = 0;
	for (int j = 0; j < array.length; j++) {
		if (array[j] == maxCount) {
			num++;
		}
	}
	if (num * 2 > array.length) {
		return maxCount;
	}
	return 0;
}
```

30.输入 n 个整数，找出其中最小的 K 个数。

思路：先将前 K 个数放入数组，进行堆排序，若之后的数比它还小，则进行调整
代码实现：

```
public ArrayList<Integer> GetLeastNumbers_Solution(int [] input, int k) {
        ArrayList<Integer> list = new ArrayList<>();
        if (input == null || k <= 0 || k > input.length) {
                return list;
        }
        int[] kArray = Arrays.copyOfRange(input,0,k);
        // 创建大根堆
        buildHeap(kArray);
        for(int i = k; i < input.length; i++) {
                if(input[i] < kArray[0]) {
                        kArray[0] = input[i];
                        maxHeap(kArray, 0);
                }
        }
        for (int i = kArray.length - 1; i >= 0; i--) {
                list.add(kArray[i]);
        }
        return list;
}
public void buildHeap(int[] input) {
        for (int i = input.length/2 - 1; i >= 0; i--) {
                maxHeap(input,i);
        }
}
private void maxHeap(int[] array,int i) {
        int left=2*i+1;
        int right=left+1;
        int largest=0;
        if(left < array.length && array[left] > array[i])
                largest=left;
        else
                largest=i;
        if(right < array.length && array[right] > array[largest])
                largest = right;
        if(largest != i) {
                int temp = array[i];
                array[i] = array[largest];
                array[largest] = temp;
                maxHeap(array, largest);
        }
}
```

31.求连续子数组（包含负数）的最大和

思路：若和小于 0，则将最大和置为当前值，否则计算最大和。
代码实现：

```
public int FindGreatestSumOfSubArray(int[] array) {
    if (array == null || array.length == 0)
        return 0;
    int cur = array[0];
    int greast = array[0];
    for (int i = 1; i < array.length; i++) {
        if (cur < 0) {
            cur = array[i];
        }else {
            cur += array[i];
        }
        if (cur > greast) {
            greast = cur;
        }
    }
    return greast;
}
```

32.从 1 到非负整数 n 中 1 出现的次数

思路：若百位上数字为 0，百位上可能出现 1 的次数由更高位决定；若百位上数字为 1，百位上可能出现 1 的次数不仅受更高位影响还受低位影响；若百位上数字大于 1，则百位上出现 1 的情况仅由更高位决定

代码实现：

```
public long CountOne2(long n) {
    long count = 0; // 1 的个数
    long i = 1;    //  当前位
    long current = 0,after = 0,before = 0;
    while((n / i) != 0) {
        before = n / (i * 10); //  高位
        current = (n / i) % 10; //  当前位
        after = n - (n / i) * i;    //  低位
        if (current == 0)
            //如果为 0,出现 1 的次数由高位决定,等于高位数字 *  当前位数
            count = count + before * i;
        else if(current == 1)
```

```
                //如果为 1,出现 1 的次数由高位和低位决定,高位*当前位+低位+1
                count = count + before * i + after + 1;
            else if (current > 1)
                // 如果大于 1,出现 1 的次数由高位决定,（高位数字+1）* 当前位数
                count = count + (before + 1) * i;
            //前移一位
            i = i * 10;
        }
        return count;
    }
```

33.输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个

思路：先将整型数组转换成 String 数组，然后将 String 数组排序，最后将排好序的字符串数组拼接出来。关键就是制定排序规则。或使用比较和快排的思想，将前面的数和最后的数比较，若小则放到最前面，最后再递归调用。

代码实现：

```
    public String PrintMinNumber(int [] numbers) {
        if(numbers == null || numbers.length == 0)
            return "";
        int len = numbers.length;
        String[] str = new String[len];
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < len; i++){
            str[i] = String.valueOf(numbers[i]);
        }
        Arrays.sort(str,new Comparator<String>(){
            @Override
            public int compare(String s1, String s2) {
                String c1 = s1 + s2;
                String c2 = s2 + s1;
                return c1.compareTo(c2);
            }
        });
        for(int i = 0; i < len; i++){
            sb.append(str[i]);
        }
        return sb.toString();
    }
```

34.求从小到大的第 N 个丑数。丑数是只包含因子 2、3 和 5 的数，习惯上我们把 1 当做是第一个丑数。

思路：乘 2 或 3 或 5，之后比较取最小值。
代码实现：

```
public int GetUglyNumber_Solution(int index) {
    if (index <= 0)
        return 0;
    int[] arr = new int[index];
    arr[0] = 1;
    int multiply2 = 0;
    int multiply3 = 0;
    int multiply5 = 0;
    for (int i = 1; i < index; i++) {
        int min = Math.min(arr[multiply2] * 2,Math.min(arr[multiply3] *
3,arr[multiply5] * 5));
        arr[i] = min;
        if (arr[multiply2] * 2 == min)
            multiply2++;
        if (arr[multiply3] * 3 == min)
            multiply3++;
        if (arr[multiply5] * 5 == min)
            multiply5++;
    }
    return arr[index - 1];
}
```

35.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

思路：利用 LinkedHashMap 保存字符和出现次数。
代码实现：

```
public int FirstNotRepeatingChar(String str) {
    if (str == null || str.length() == 0)
        return -1;
    char[] c = str.toCharArray();
    LinkedHashMap<Character,Integer>                    hash=new
LinkedHashMap<Character,Integer>();
    for(char item : c) {
        if(hash.containsKey(item))
            hash.put(item, hash.get(item)+1);
```

```
            else
                hash.put(item, 1);
        }
        for(int i = 0;i < str.length(); i++){
            if (hash.get(str.charAt(i)) == 1) {
                return i;
            }
        }
        return -1;
    }
```

36.在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数 P

思路：本质是归并排序，在比较时加入全局变量 count 进行记录逆序对的个数，若 data[start] >= data[index] ，则 count 值为 mid+1-start

代码实现：

```
    int count = 0;
    public int InversePairs(int [] array) {
        if(array==null)
            return 0;
        mergeSort(array,0,array.length-1);
        return count;
    }
    private void mergeSort(int[] data,int start,int end) {
        int mid = (start + end) / 2;
        if (start < end) {
            mergeSort(data, start, mid);
            mergeSort(data, mid + 1, end);
            merge(data, start, mid, end);
        }
    }
    public void merge(int[] data,int start,int mid,int end) {
        int arr[] = new int[end - start + 1];
        int c = 0;
        int s = start;
        int index = mid + 1;
        while (start <= mid && index <= end) {
            if (data[start] < data[index]) {
                arr[c++] = data[start++];
            } else {
```

```
                arr[c++] = data[index++];
                count += mid +1 - start;
                count %= 1000000007;
            }
        }
        while (start <= mid) {
            arr[c++] = data[start++];
        }
        while (index <= end) {
            arr[c++] = data[index++];
        }
        for (int d : arr) {
            data[s++] = d;
        }
    }
```

37.输入两个链表，找出它们的第一个公共结点。

思路：先求出链表长度，然后长的链表先走多出的几步，然后两个链表同时向下走去寻找相同的节点，代码量少的方法需要将两个链表遍历两次，然后从头开始相同的节点。

代码实现：

```
    // 不需要遍历链表的解法
    public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {
        ListNode p1 = pHead1;
        ListNode p2 = pHead2;
        while (p1 != p2){
            p1 = (p1 != null ? p1.nextNode : pHead2);
            p2 = (p2 != null ? p2.nextNode : pHead1);
        }
        return p1;
    }
```

38.统计一个数字在排序数组中出现的次数。

思路：利用二分查找+递归思想，进行寻找。当目标值与中间值相等时进行判断

代码实现：

```
    public int GetNumberOfK(int[] array,int k) {
        int result=0;
        int mid = array.length/2;
        if(array==null || array.length == 0)
            return 0;
        if(array.length == 1) {
```

```
            if(array[0] == k)
                return 1;
            else
                return 0;
        }
        if(k < array[mid])
            result += GetNumberOfK(Arrays.copyOfRange(array, 0, mid),k);
        else if(k > array[mid])
            result += GetNumberOfK(Arrays.copyOfRange(array, mid,
array.length),k);
        else {
            for(int i = mid;i < array.length;i++) {
                if(array[i] == k)
                    result++;
                else
                    break;
            }
            for(int i = mid - 1;i >= 0;i--) {
                if(array[i] == k)
                    result++;
                else
                    break;
            }
        }
        return result;
    }
```

39.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：利用递归遍历分别返回左右子树深度

代码实现：

```
    public int TreeDepth(TreeNode root) {
        if (root == null)
            return 0;
        int left = TreeDepth(root.left);
        int right = TreeDepth(root.right);
        return left > right ? left + 1 : right + 1;
    }
```

39.输入一棵二叉树，判断该二叉树是否是平衡二叉树。

思路：平衡因子的绝对值<= 1.

代码实现：

```
    public boolean isBalanced(TreeNode root) {
        if(root == null) {
```

```
            return true;
        }
        boolean condition = Math.abs(maxDepth(root.left) - maxDepth(root.right))
<= 1;
        return condition && isBalanced(root.left) && isBalanced(root.right);
    }
    public int treePath(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int left = treePath(root.left);
        int right = treePath(root.right);
        return left > right ? (left + 1) : (right + 1);
    }
```

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

思路：两个相同的数异或后为 0，将所有数异或后得到一个数，然后求得 1 在该数最右边出现的 index，然后判断每个数右移 index 后是不是 1。
代码实现：

```
    public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {
        if (array == null)
            return;
        num1[0] = 0;
        num2[0] = 0;
        int number = array[0];
        for (int i = 1; i < array.length; i++)
            number ^= array[i];
        // 异或后的数 1 出现在第几位
        int index = 0;
        while ((number & 1) == 0) {
            number = number >> 1;
            index++;
        }
        for (int i = 0; i < array.length; i++) {
            // 判断第 index 位是不是 0
            boolean isBit = ((array[i] >> index) & 1) == 0;
            if (isBit) {
                num1[0] ^= array[i];
            } else {
                num2[0] ^= array[i];
            }
```

```
        }
    }
```

41.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

思路：定义两个指针，分别递增，寻找和为 s 的序列。
代码实现：

```
    public ArrayList<ArrayList<Integer>> FindContinuousSequence(int sum) {
        ArrayList<ArrayList<Integer>> arrayList = new ArrayList<>();
        ArrayList<Integer> list = new ArrayList<>();
        if (sum < 3)
            return arrayList;
        int small = 1;
        int big = 2;
        while (small < (sum + 1) / 2) {
            int s = 0;
            for (int i = small; i <= big; i++) {
                s += i;
            }
            if (s == sum) {
                for (int i = small; i <= big; i++) {
                    list.add(i);
                }
                arrayList.add(new ArrayList<>(list));
                list.clear();
                small++;
            } else {
                if (s > sum) {
                    small++;
                } else {
                    big++;
                }
            }
        }
        return arrayList;
    }
```

41.1 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：定义两个指针，分别从前面和后面进行遍历。间隔越远乘积越小，所以是最先出现的两个数乘积最小
代码实现：

```
public ArrayList<Integer> FindNumbersWithSum(int [] array,int sum) {
    ArrayList<Integer> list = new ArrayList<>();
    if (array == null )
        return list;
    int left = 0;
    int right = array.length - 1;
    while (left < right) {
        int s = array[left] + array[right];
        if (s == sum) {
            list.add(array[left]);
            list.add(array[right]);
            return list;
        }else {
            if (s > sum) {
                right--;
            }else {
                left++;
            }
        }
    }
    return list;
}
```

42.翻转字符串

思路：先将整个字符串翻转，然后将每个单词翻转。
代码实现：

```
public String ReverseSentence(String str) {
    if (str == null || str.length() == 0)
        return str;
    if (str.trim().length() == 0)
        return str;
    StringBuilder sb = new StringBuilder();
    String re = reverse(str);
```

```
        String[] s = re.split(" ");
        for (int i = 0; i < s.length - 1; i++) {
            sb.append(reverse(s[i]) + " ");
        }
        sb.append(reverse(s[s.length-1]));
        return String.valueOf(sb);
    }
    public String reverse(String str) {
        StringBuilder sb = new StringBuilder();
        for (int i = str.length() - 1; i >= 0 ; i--) {
            sb.append(str.charAt(i));
        }
        return String.valueOf(sb);
    }
```

42.1 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

思路：拼接或反转三次字符串
代码实现：

```
public String LeftRotateString(String str,int n) {
        if (str == null || str.length() == 0)
            return str;
        String s1 = reverse(str.substring(0,n));
        String s2 = reverse(str.substring(n,str.length()));
        return reverse(s2)+reverse(s1);
    }
```

43.把 n 个骰子扔在地上，所有骰子朝上一面的点数之和为 s,输入 n,打印出 s 的所有可能出现的概率

思路：递归一般是自顶向下的分析求解，而循环则是自底向上，占用更少的空间和更少的时间，性能较好。定义一个二维数组，第一次掷骰子有 6 种可能，第一个骰子投完的结果存到 probabilities[0]；第二次开始掷骰子，在下一循环中，我们加上一个新骰子，此时和为 n 的骰子出现次数应该等于上一次循环中骰子点数和为 n-1,n-2,n-3, n-4,n-5，n-6 的次数总和，所以我们把另一个数组的第 n 个数字设为前一个数组对应 n-1,n-2,n-3,n-4,n-5，n-6 之和
代码实现：

```
public void printProbability(int number) {
        if(number<1)
            return ;
```

```
    int g_maxValue=6;
    int[][] probabilities=new int[2][];
    probabilities[0]=new int[g_maxValue*number+1];
    probabilities[1]=new int[g_maxValue*number+1];
    int flag=0;
    // 当第一次抛掷骰子时，有 6 种可能，每种可能出现一次
    for(int i=1;i<=g_maxValue;i++)
        probabilities[0][i]=1;
    //从第二次开始掷骰子，假设第一个数组中的第 n 个数字表示骰子和为 n 出现的次数，
    for(int k=2;k<=number;++k) {
        for(int i=0;i<k;++i)
            // 第 k 次掷骰子，和最小为 k，小于 k 的情况是不可能发生的,令不可能发生的次数设置为 0！
            probabilities[1-flag][i]=0;
        // 第 k 次掷骰子，和最小为 k，最大为 g_maxValue*k
        for(int i=k;i<=g_maxValue*k;++i) {
            // 初始化，因为这个数组要重复使用，上一次的值要清 0
            probabilities[1-flag][i]=0;
            for(int j=1;j<=i && j<=g_maxValue;++j)
                probabilities[1-flag][i]+=probabilities[flag][i-j];
        }
        flag=1-flag;
    }
    double total=Math.pow(g_maxValue, number);
    for(int i=number;i<=g_maxValue*number;i++) {
        double ratio=(double) probabilities[flag][i]/total;
        System.out.println(i);
        System.out.println(ratio);
    }
}
```

44.扑克牌的顺子

思路：用数组记录五张扑克牌，将数组调整为有序的，若 0 出现的次数>=顺子的差值，即为顺子。

代码实现：

```
public boolean isContinuous(int [] numbers) {
    if (numbers == null || numbers.length == 0)
        return false;
    int count = 0;
    int diff = 0;
    Arrays.sort(numbers);
```

```
        for (int i = 0; i < numbers.length - 1; i++) {
            if (numbers[i] == 0) {
                count++;
                continue;
            }
            if (numbers[i] != numbers[i+1]) {
                diff += numbers[i+1] - numbers[i] - 1;
            } else {
                return false;
            }
        }
        if (diff <= count)
            return true;
        return false;
}
```

45.圆圈中最后剩下的数字（约瑟夫环）

思路：利用循环链表实现
代码实现：

```
public int LastRemaining_Solution(int n, int m) {
        LinkedList<Integer> list = new LinkedList<Integer>();
        int bt = 0;
        for (int i = 0; i < n; i ++) {
            list.add(i);
        }
        while (list.size() > 1) {
            bt = (bt + m - 1) % list.size();
            list.remove(bt);
        }
        return list.size() == 1 ? list.get(0) : -1;
}
```

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

思路：巧用递归（返回值类型为 Boolean）
代码实现：

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

```
}
```

47.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：利用位运算
代码实现：

```
public int Add(int num1,int num2) {
        while (num2 != 0) {
                // 计算个位
                int temp = num1 ^ num2;
                // 计算进位（1+1）
                num2 = (num1 & num2) << 1;
                num1 = temp;
        }
        return num1;
}
```

48.不能被继承的类

思路：私有构造器的类不能继承

49.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：若为负数，则输出负数，字符 0 对应 48,9 对应 57，不在范围内则返回 false。
代码实现：

```
public int StrToInt(String str) {
        if (str == null || str.length() == 0)
                return 0;
        int mark = 0;
        int number = 0;
        char[] chars = str.toCharArray();
        if (chars[0] == '-')
                mark = 1;
        for (int i = mark; i < chars.length; i++) {
                if (chars[i] == '+') {
                        continue;
```

```
            }
            if (chars[i] < 48 || chars[i] > 57) {
                return 0;
            }
            number = number * 10 + chars[i] - 48;
        }
        return mark == 0 ? number : -number;
}
```

50.求树中两个节点的最低公共祖先

（1）树是二叉搜索树
思路：从树的根节点开始遍历，如果根节点的值大于其中一个节点，小于另外一个节点，则根节点就是最低公共祖先。否则如果根节点的值小于两个节点的值，则递归求根节点的右子树，如果大于两个节点的值则递归求根的左子树。如果根节点正好是其中的一个节点，那么说明这两个节点在一条路径上，所以最低公共祖先则是根节点的父节点，时间复杂度是 O(logn)，空间复杂度是 O(1)
代码实现：

```
public static BinaryTreeNode getLowestCommonAncestor(BinaryTreeNode rootParent,BinaryTreeNode root,
                                                      BinaryTreeNode node1,BinaryTreeNode node2){
        if(root == null || node1 == null || node2 == null){
            return null;
        }
        if((root.value - node1.value)*(root.value - node2.value) < 0){
            return root;
        }else if((root.value - node1.value)*(root.value - node2.value) > 0){
            BinaryTreeNode newRoot = ((root.value > node1.value) && (root.value > node2.value))
                        ? root.leftNode : root.rightNode;
            return getLowestCommonAncestor(root,newRoot, node1, node2);
        }else{
            return rootParent;
        }
}
```

（2）若树是普通树，但有指向父节点的指针
思路：两个节点如果在两条路径上，类似于求两个链表的第一个公共节点。由于每个节点的深度最多为 logn，所以时间复杂度为 O(logn),空间复杂度 O(1)
代码实现：

```
public static BinaryTreeNode getLowestCommonAncestor1(BinaryTreeNode root,BinaryTreeNode node1,
                                                       BinaryTreeNode node2){
```

```
        if(root == null || node1 == null || node2 == null){
            return null;
        }
        int depth1 = findTheDepthOfTheNode(root, node1, node2);
        if(depth1 == -1){
            return node2.parentNode;
        }
        int depth2 = findTheDepthOfTheNode(root, node2, node1);
        if(depth2 == -1){
            return node1.parentNode;
        }
        //p 指向较深的节点 q 指向较浅的节点
        BinaryTreeNode p = depth1 > depth2 ? node1 : node2;
        BinaryTreeNode q = depth1 > depth2 ? node2 : node1;
        int depth =    Math.abs(depth1 - depth2);
        while(depth > 0){
            p = p.parentNode;
            depth --;
        }
        while(p != q){
            p = p.parentNode;
            q = q.parentNode;
        }
        return p;
    }
    //求 node1 的深度，如果 node1 和 node2 在一条路径上，则返回-1，否则返回 node1 的深度
    public static int findTheDepthOfTheNode(BinaryTreeNode root,BinaryTreeNode node1,
                                                    BinaryTreeNode node2){
        int depth = 0;
        while(node1.parentNode != null){
            node1 = node1.parentNode;
            depth ++;
            if(node1 == node2){
                return -1;
            }
        }
        return depth;
    }
```

（3）若树是普通树，并没有指向父节点的指针

思路：用栈来实现类似于指向父节点指针的功能，获取 node 节点的路径时间复杂度为 O(n),所以总的时间复杂度是 O(n),空间复杂度是 O(logn)

代码实现：

```
public static BinaryTreeNode getLowestCommonAncestor2(BinaryTreeNode root, BinaryTreeNode node1,
                                                       BinaryTreeNode node2){
    if(root == null || node1 == null || node2 == null){
        return null;
    }
    Stack<BinaryTreeNode> path1 = new Stack<BinaryTreeNode>();
    boolean flag1 = getThePathOfTheNode(root, node1,path1);
    if(!flag1){//树上没有 node1 节点
        return null;
    }
    Stack<BinaryTreeNode> path2 = new Stack<BinaryTreeNode>();
    boolean flag2 = getThePathOfTheNode(root, node2,path2);
    if(!flag2){//树上没有 node2 节点
        return null;
    }
    if(path1.size() > path2.size()){ //让两个路径等长
        while(path1.size() !=    path2.size()){
            path1.pop();
        }
    }else{
        while(path1.size() !=    path2.size()){
            path2.pop();
        }
    }

    if(path1 == path2){//当两个节点在一条路径上时
        path1.pop();
        return path1.pop();
    }else{
        BinaryTreeNode p = path1.pop();
        BinaryTreeNode q = path2.pop();
        while(q != p){
            p = path1.pop();
            q = path2.pop();
        }
        return p;
    }
}
//获得根节点到 node 节点的路径
public static boolean getThePathOfTheNode(BinaryTreeNode root,BinaryTreeNode node,
                                          Stack<BinaryTreeNode> path){
```

```
        path.push(root);
        if(root == node){
            return true;
        }
        boolean found = false;
        if(root.leftNode != null){
            found = getThePathOfTheNode(root.leftNode, node, path);
        }
        if(!found && root.rightNode != null){
            found = getThePathOfTheNode(root.rightNode, node, path);
        }
        if(!found){
            path.pop();
        }
        return found;
}
```

51.在一个长度为 n 的数组里的所有数字都在 0 到 n-1 的范围内，找出数组中任意一个重复的数字。

思路：若下标大于 length，则减去 length，最后再加上 length，若下标的数组值大于 length，则返回 true。或使用辅助空间（HashSet）

代码实现：

```
public boolean duplicate(int numbers[],int length,int [] duplication) {
        if (numbers == null || length == 0 || length == 1)
            return false;
        for (int i = 0; i < length; i++) {
            int index = numbers[i];
            if (index >= length)
                index -= length;
            if (numbers[index] >= length) {
                duplication[0] = index;
                return true;
            }
            numbers[index] += length;
        }
        return false;
}
```

52.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]。其中 A[i] = 1。不能使用除法

思路：使用矩阵法求解，将矩阵分为上三角矩阵和下三角矩阵，分别求乘积
代码实现：

```
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;
        //计算上三角连乘
        for(int j = length-2; j >= 0; j--){
            temp *= A[j+1];
            B[j] *= temp;
        }
    }
    return B;
}
```

53.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）

思路：当字符串只有一个字符时，进行判断，否则就有两种递归情况，（1）当模式中的第二个字符不是“”时：如果字符串第一个字符和模式中的第一个字符相匹配或是点那么字符串和模式都后移一个字符，然后匹配剩余的；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。（2）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或是点，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；
代码实现：

```
public boolean match(char[] str, char[] pattern) {
    if (str == null || pattern == null)
        return false;
    // 若字符串的长度为 1
    if (str.length == 1) {
        if (pattern.length == 1){
            if (str[0] == pattern[0] || pattern[0] == '.')
                return true;
            return false;
        }
    }
    int sindex = 0;
    int pindex = 0;
    return matchIndex(str,sindex,pattern,pindex);
}

public boolean matchIndex(char[] str,int sindex, char[] pattern, int pindex) {
    // str 和 pattern 同时到达末尾，则匹配成功
    if (sindex == str.length && pindex == pattern.length)
        return true;
    // 若 pattern 先到尾，而 str 没有到达末尾，则匹配失败
    if (sindex != str.length && pindex == pattern.length)
        return false;
    // 若 pattern 第二个字符是*
    if (pindex + 1 < pattern.length && pattern[pindex + 1] == '*') {
        if (sindex != str.length && pattern[pindex] == str[sindex] ||
                sindex != str.length && pattern[pindex] == '.') {
            return matchIndex(str,sindex+1,pattern,pindex+2)
                    || matchIndex(str,sindex,pattern,pindex+2)
                    || matchIndex(str,sindex+1,pattern,pindex);
        } else {
            return matchIndex(str,sindex,pattern,pindex+2);
        }
    }
    // 若 pattern 第二个字符不是*
    if (sindex != str.length && pattern[pindex] == str[sindex] ||
            sindex != str.length && pattern[pindex] == '.')
        return matchIndex(str,sindex+1,pattern,pindex+1);
    return false;
}
```

54.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）

思路：逐个字符进行判断，e 或 E 和小数点最多出现一次，而 e 或 E 的前一个必须是数字，且不能是第一个或最后一个字符，符号的前一个字符不能是 e 或 E。也可用正则表达式判断！

代码实现：

```
public boolean isNumeric(char[] str) {
    if (str == null)
        return false;
    int index = 0;
    int ecount = 0;
    int point = 0;
    // 如果第一个字符是符号就跳过
    if (str[0] == '-' || str[0] == '+')
        index++;
    for (int i = index; i < str.length; i++) {
        if (str[i] == '-' || str[i] == '+') {
            if (str[i-1] != 'e' && str[i-1] != 'E')
                return false;
            continue;
        }
        if (str[i] == 'e' || str[i] == 'E') {
            ecount++;
            if (ecount > 1)
                return false;
            if (i == 0 || str[i-1] < 48 || str[i-1] > 57 || i == str.length-1)
                return false;
            point++;
            continue;
        }
        if (str[i] == '.') {
            point++;
            if (point > 1)
                return false;
            continue;
        }
        // 出现非数字且不是 e/E 则返回 false（小数点和符号用 continue 跳过了）
        if ((str[i] < 48 || str[i] > 57) && (str[i] != 'e') && (str[i] != 'E'))
            return false;
    }
```

```
        return true;
}
```

55.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。

思路：借助辅助空间进行判断，如字符数组。
代码实现：

```
char[] chars = new char[256];
StringBuilder sb = new StringBuilder();
public void Insert(char ch) {
        sb.append(ch);
        chars[ch]++;
}
public char FirstAppearingOnce() {
        char[] str = sb.toString().toCharArray();
        for (char c : str) {
                if (chars[c] == 1) {
                        return c;
                }
        }
        return '#';
}
```

56.一个链表中包含环，请找出该链表的环的入口结点。

思路：定义快慢两个指针，相遇后（环中相汇点）将快指针指向 pHead 然后一起走，每次往后挪一位，相遇的节点即为所求。详细分析：相遇即 p1==p2 时，p2 所经过节点数为 2x,p1 所经过节点数为 x,设环中有 n 个节点,p2 比 p1 多走一圈有 2x=n+x; n=x;可以看出 p1 实际走了一个环的步数，再让 p2 指向链表头部，p1 位置不变，p1,p2 每次走一步直到 p1==p2; 此时 p1 指向环的入口。
代码实现：

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

```
                while (fast != slow) {
                    fast = fast.next;
                    slow = slow.next;
                }
                if (fast == slow)
                    return slow;
            }
        }
        return null;
    }
```

57.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。

思路：先新建一个头节点，然后向后查找值相同的节点，重复查找后删除
代码实现：

```
public ListNode deleteDuplication(ListNode pHead) {
        if (pHead == null)
            return null;
        // 新建一个节点，防止头结点被删除
        ListNode first = new ListNode(-1);
        first.next = pHead;
        ListNode p = pHead;
        // 指向前一个节点
        ListNode preNode = first;
        while (p != null && p.next != null) {
            if (p.val == p.next.val) {
                int val = p.val;
                // 向后重复查找
                while (p != null && p.val == val) {
                    p = p.next;
                }
                // 上个非重复值指向下一个非重复值：即删除重复值
                preNode.next = p;
            }else {
                // 如果当前节点和下一个节点值不等，则向后移动一位
                preNode = p;
                p = p.next;
            }
        }
        return first.next;
    }
```

58.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：若节点右孩子存在，则设置一个指针从该节点的右孩子出发，一直沿着指向左子结点的指针找到的叶子节点即为下一个节点；若节点不是根节点。如果该节点是其父节点的左孩子，则返回父节点；否则继续向上遍历其父节点的父节点，重复之前的判断，返回结果

代码实现：

```
public TreeLinkNode GetNext(TreeLinkNode pNode) {
    if (pNode == null)
        return null;
    if (pNode.right != null) {
        pNode = pNode.right;
        while (pNode.left != null) {
            pNode = pNode.left;
        }
        return pNode;
    }
    while (pNode.next != null) {
        // 找第一个当前节点是父节点左孩子的节点
        if (pNode.next.left == pNode)
            return pNode.next;
        pNode = pNode.next;
    }
    return null;
}
```

59.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：利用递归进行判断，若左子树的左孩子等于右子树的右孩子且左子树的右孩子等于右子树的左孩子，并且左右子树节点的值相等，则是对称的。

代码实现：

```
public boolean isSymmetrical(TreeNode pRoot){
    if (pRoot == null)
        return true;
    return isCommon(pRoot.left, pRoot.right);
```

```
}
public boolean isCommon(TreeNode leftNode, TreeNode rightNode) {
    if (leftNode == null && rightNode == null)
        return true;

    if (leftNode != null && rightNode != null)
        return leftNode.val == rightNode.val &&
                isCommon(leftNode.left,rightNode.right) &&
                isCommon(leftNode.right,rightNode.left);
    return false;
}
```

60.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，依此类推。

思路：利用两个栈的辅助空间分别存储奇数偶数层的节点，然后打印输出。或使用链表的辅助空间来实现，利用链表的反向迭实现逆序输出。
代码实现：

```
public ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<>();
    if (pRoot == null)
        return res;
    Stack<TreeNode> s1 = new Stack<>();   // s1 表示奇数，从右向左输出
    Stack<TreeNode> s2 = new Stack<>();   // s2 表示偶数，从左向右输出
    s1.push(pRoot);
    int level = 1;
    while (!s1.empty() || !s2.empty()) {
        if (level % 2 != 0) {
            ArrayList<Integer> list = new ArrayList<>();
            while (!s1.empty()) {
                TreeNode node = s1.pop();
                if (node != null) {
                    list.add(node.val);
                    s2.push(node.left);
                    s2.push(node.right);
                }
            }
            if (!list.isEmpty()) {
                res.add(list);
                level++;
            }
```

```
            } else {
                ArrayList<Integer> list = new ArrayList<>();
                while (!s2.empty()) {
                    TreeNode node = s2.pop();
                    if (node != null) {
                        list.add(node.val);
                        s1.push(node.right);
                        s1.push(node.left);
                    }
                }
                if (!list.isEmpty()) {
                    res.add(list);
                    level++;
                }
            }
        }
        return res;
}
```

61.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：利用辅助空间链表或队列来存储节点，每层输出。
代码实现：

```
public ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {
        ArrayList<ArrayList<Integer>> res = new ArrayList<>();
        if (pRoot == null)
            return res;
        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.add(pRoot);
        ArrayList<Integer> list = new ArrayList<>();
        int start = 0;
        int end = 1;
        while (!queue.isEmpty()) {
            TreeNode node = queue.pop();
            list.add(node.val);
            start++;
            if (node.left != null)
                queue.offer(node.left);
            if (node.right != null)
                queue.offer(node.right);
            if (start == end) {
                start = 0;
```

```
                end = queue.size();
                res.add(new ArrayList<>(list));
                list.clear();
            }
        }
        return res;
}
```

62.请实现两个函数，分别用来序列化和反序列化二叉树

思路：序列化：前序遍历二叉树存入字符串中；反序列化：根据前序遍历重建二叉树。

代码实现：

```
public String Serialize(TreeNode root) {
        StringBuffer sb = new StringBuffer();
        if (root == null){
            sb.append("#,");
            return sb.toString();
        }
        sb.append(root.val + ",");
        sb.append(Serialize(root.left));
        sb.append(Serialize(root.right));
        return sb.toString();
}
public int index = -1;
public TreeNode Deserialize(String str) {
        index++;
        int len = str.length();
        String[] strr = str.split(",");
        TreeNode node = null;
        if (index >= len)
            return null;
        if (!strr[index].equals("#")){
            node = new TreeNode(Integer.valueOf(strr[index]));
            node.left = Deserialize(str);
            node.right = Deserialize(str);
        }
        return node;
}
```

63.给定一颗二叉搜索树，请找出其中的第 k 大的结点

思路：二叉搜索树按照中序遍历的顺序打印出来正好就是排序好的顺序，第 k 个

结点就是第 K 大的节点，分别递归查找左右子树的第 K 个节点，或使用非递归借用栈的方式查找，当 count=k 时返回根节点。
代码实现：

```
int count = 0;
public TreeNode KthNode(TreeNode pRoot, int k) {
    if (pRoot == null || k < 1)
        return null;
    count++;
    if (count == k) {
        return pRoot;
    }
    TreeNode leftNode = KthNode(pRoot.left,k);
    if (leftNode != null)
        return leftNode;
    TreeNode rightNode = KthNode(pRoot.right,k);
    if (rightNode != null)
        return rightNode;
    return null;
}
```

64.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：创建优先级队列维护大顶堆和小顶堆两个堆，并且小顶堆的值都大于大顶堆的值，2 个堆个数的差值小于等于 1，所以当插入个数为奇数时：大顶堆个数就比小顶堆多 1，中位数就是大顶堆堆头；当插入个数为偶数时，使大顶堆个数跟小顶堆个数一样，中位数就是 2 个堆堆头平均数。也可使用集合类的排序方法。
代码实现：

```
int count = 0;
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(16, new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2.compareTo(o1);
    }
});
public void Insert(Integer num) {
    count++;
    // 当数据的个数为奇数时，进入大根堆
    if ((count & 1) == 1) {
```

```
            minHeap.offer(num);
            maxHeap.offer(minHeap.poll());
        } else {
            maxHeap.offer(num);
            minHeap.offer(maxHeap.poll());
        }
    }
    public Double GetMedian() {
        if (count == 0)
            return null;
        // 当数据个数是奇数时，中位数就是大根堆的顶点
        if ((count & 1) == 1) {
            return Double.valueOf(maxHeap.peek());
        } else {
            return Double.valueOf((minHeap.peek() + maxHeap.peek())) / 2;
        }
    }
```

65.给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值

思路：两个 for 循环，第一个 for 循环滑动窗口，第二个 for 循环滑动窗口中的值，寻找最大值。还可以使用时间复杂度更低的双端队列求解。
代码实现：

```
public ArrayList<Integer> maxInWindows(int [] num, int size) {
        ArrayList<Integer> list = new ArrayList<>();
        if (num == null || size < 1 || num.length < size)
            return list;
        int length = num.length - size + 1;
        for (int i = 0; i < length; i++) {
            int current = size + i;
            int max = num[i];
            for (int j = i; j < current; j++) {
                if (max < num[j]) {
                    max = num[j];
                }
            }
            list.add(max);
        }
        return list;
    }
```

66.请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。

路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。

思路：回溯法，双层 for 循环，判断每一个点，每次递归调用上下左右四个点，用 flag 标志是否已经匹配（return），进行判断点的位置是否越界，是否已经正确匹配，判断矩阵的路径与模式串的第 index 个字符是否匹配。

代码实现：

```
public boolean hasPath(char[] matrix, int rows, int cols, char[] str) {
	int flag[] = new int[matrix.length];
	for (int i = 0; i < rows; i++) {
		for (int j = 0; j < cols; j++) {
			if (helper(matrix, rows, cols, i, j, str, 0, flag))
				return true;
		}
	}
	return false;
}
private boolean helper(char[] matrix,int rows,int cols,int i,int j,char[] str,int k,int[] flag)
{
	int index = i * cols + j;
	if (i < 0 || i >= rows || j < 0 || j >= cols || matrix[index] != str[k] || flag[index] ==
1)
		return false;
	if(k == str.length - 1)
		return true;
	flag[index] = 1;
	if (helper(matrix, rows, cols, i - 1, j, str, k + 1, flag)
			|| helper(matrix, rows, cols, i + 1, j, str, k + 1, flag)
			|| helper(matrix, rows, cols, i, j - 1, str, k + 1, flag)
			|| helper(matrix, rows, cols, i, j + 1, str, k + 1, flag)) {
		return true;
	}
	flag[index] = 0;
	return false;
}
```

67.地上有一个 m 行和 n 列的方格。一个机器人从坐标 0,0 的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于 k 的格子。

思路：利用递归实现，每次只能走上下左右四个点，进行判断点的位置是否越界，点数之和是否大于 K，是否已经走过了。

代码实现：

```
public int movingCount(int threshold, int rows, int cols) {
	int flag[][] = new int[rows][cols]; //记录是否已经走过
	return helper(0, 0, rows, cols, flag, threshold);
}
private int helper(int i, int j, int rows, int cols, int[][] flag, int threshold) {
	if (i < 0 || i >= rows || j < 0 || j >= cols ||
			numSum(i) + numSum(j) > threshold || flag[i][j] == 1)
		return 0;
	flag[i][j] = 1;
	return helper(i - 1, j, rows, cols, flag, threshold)
			+ helper(i + 1, j, rows, cols, flag, threshold)
			+ helper(i, j - 1, rows, cols, flag, threshold)
			+ helper(i, j + 1, rows, cols, flag, threshold) + 1;
}
private int numSum(int i) {
	int sum = 0;
	while (i > 0) {
		sum += i % 10;
		i = i / 10;
	}
	return sum;
}
```