题目 数值的整数次方

考点 代码的完整性 热点指数 79138 通过率 31.19%

具体题目

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。

/*剑指书中细节： *1.当底数为0且指数<0时 *会出现对0求倒数的情况，需进行错误处理，设置一个全局变量； *2.判断底数是否等于0 *由于base为double型，不能直接用==判断 *3.优化求幂函数 *当n为偶数，a^n =（a^n/2）*（a^n/2）*当n为奇数，a^n = a^[(n-1)/2] * a^[(n-1)/2] * a *时间复杂度O(logn) */

```
public class Solution {
      boolean invalidInput=false;
      public double Power(double base, int exponent) {
        if(equal(base,0.0)&&exponent<0){
            invalidInput=true;
            return 0.0;
        }
        int absexponent=exponent;
        if(exponent<0)
            absexponent=-exponent;
        double res=getPower(base,absexponent);
        if(exponent<0)
            res=1.0/res;
        return res;
  }
    boolean equal(double num1,double num2){
        if(num1-num2>-0.000001&&num1-num2<0.000001)
            return true;
        else
            return false;
    }
    double getPower(double b,int e){
        if(e==0)
            return 1.0;
        if(e==1)
            return b;
        double result=getPower(b,e>>1);
        result*=result;
        if((e&1)==1)
            result*=b;
        return result;
    }
}
```

第一种方法：使用递归，时间复杂度O(logn) 当n为偶数，a^n =（a^n/2）*（a^n/2） 当n为奇数，a^n = a^[(n-1)/2] * a^[(n-1)/2] * a 举例： 2^11 = 2^1 * 2^2 * 2^8 2^1011 = 2^0001 * 2^0010 * 2^1000 第二种方法：累乘，时间复杂度为O(n) 【参考代码】

```
package javaTest;
public class JavaTest {
    public static void main(String[] args) {
        System.out.println(power(3, 3));
        System.out.println(powerAnother(3, 3));
    }
    // 使用递归
    public static double power(double base, int exponent) {
```

```
        int n = Math.abs(exponent);
        double result = 0.0;
        if (n == 0)
            return 1.0;
        if (n == 1)
            return base;

        result = power(base, n >> 1);
        result *= result;
        if ((n & 1) == 1) // 如果指数n为奇数，则要再乘一次底数base
            result *= base;
        if (exponent < 0) // 如果指数为负数，则应该求result的倒数
            result = 1 / result;

        return result;
    }
    // 使用累乘
    public static double powerAnother(double base, int exponent) {
        double result = 1.0;
        for (int i = 0; i < Math.abs(exponent); i++) {
            result *= base;
        }
        if (exponent >= 0)
            return result;
        else
            return 1 / result;
    }

}
```

题目 调整数组顺序使奇数位于偶数前面

考点 代码的完整性 热点指数 79796 通过率 25.80%

具体题目

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

时间复杂度为O（n），空间复杂度为O（n）的算法

```
/*
整体思路：
首先统计奇数的个数
然后新建一个等长数组，设置两个指针，奇数指针从0开始，偶数指针从奇数个数的末尾开始  遍历，填数
*/

public class Solution {
    public void reOrderArray(int [] array) {
        if(array.length==0||array.length==1) return;
        int oddCount=0,oddBegin=0;
        int[] newArray=new int[array.length];
        for(int i=0;i<array.length;i++){
            if((array[i]&1)==1) oddCount++;
        }
        for(int i=0;i<array.length;i++){
            if((array[i]&1)==1) newArray[oddBegin++]=array[i];
            else newArray[oddCount++]=array[i];
        }
        for(int i=0;i<array.length;i++){
            array[i]=newArray[i];
        }
    }
}
```

从题目得出的信息： 相对位置不变--->保持稳定性；奇数位于前面，偶数位于后面 --->存在判断，挪动元素位置； 这些都和内部排序算法相似，考虑到具有稳定性的排序算法不多，例如插入排序，归并排序等；这里采用插入排序的思想实现。

```
public class Solution {
    public void reOrderArray(int [] array) {
        //相对位置不变，稳定性
        //插入排序的思想
        int m = array.length;
        int k = 0;//记录已经摆好位置的奇数的个数
        for (int i = 0; i < m; i++) {
            if (array[i] % 2 == 1) {
                int j = i;
                while (j > k) {//j >= k+1
                    int tmp = array[j];
                    array[j] = array[j-1];
                    array[j-1] = tmp;
                    j--;
                }
                k++;
            }
        }
    }
}
```

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

题目 链表中倒数第k个结点

考点 代码的鲁棒性 热点指数 81656 通过率 20.49%

具体题目

输入一个链表，输出该链表中倒数第k个结点。

时间复杂度O(n),一次遍历即可

```
public class Solution {
    public ListNode FindKthToTail(ListNode head,int k) {
        ListNode pre=null,p=null;
        //两个指针都指向头结点
        p=head;
        pre=head;
        //记录k值
        int a=k;
        //记录节点的个数
        int count=0;
        //p指针先跑，并且记录节点数，当p指针跑了k-1个节点后，pre指针开始跑，
        //当p指针跑到最后时，pre所指指针就是倒数第k个节点
        while(p!=null){
            p=p.next;
            count++;
            if(k<1){
                pre=pre.next;
            }
            k--;
        }
        //如果节点个数小于所求的倒数第k个节点，则返回空
        if(count<a) return null;
        return pre;

    }
}
```

```
public ListNode FindKthToTail(ListNode head,int k) {
        ListNode p = head;
        while(k-- != 0){
            if(p == null)
                return null;
            p = p.next;
        }
        while(p != null){
            p = p.next;
            head = head.next;
        }
        return head;
    }
```

```
public class Solution {
    public ListNode FindKthToTail(ListNode head,int k) {
        if(head == null)
            return null;
        int count = 0;
        ListNode temp = head;
        for (int i = 0; temp != null; temp = temp.next) {
            count++;
```

```
        }
        if(k>count)
            return null;
        System.out.println(count);
        //一共有count个，倒数第k个就是正数第count-k+1,下标是count-k
        for(int i = 0;i<count-k;i++){
            head = head.next;
        }
        return head;
    }
}
```

题目 反转链表

考点 代码的鲁棒性 热点指数 79927 通过率 28.80%

具体题目

输入一个链表，反转链表后，输出新链表的表头。

容易出现的问题：

- 输入的链表头指针为NULL 或整个链表只有一个结点时，程序立即崩溃
- 反转后的链表出现断裂
- 返回的反转之后的头结点不是原始链表的尾结点。

Java　循环操作　详细思路

```
public class Solution {
    public ListNode ReverseList(ListNode head) {

        if(head==null)
            return null;
        //head为当前节点，如果当前节点为空的话，那就什么也不做，直接返回null；
        ListNode pre = null;
        ListNode next = null;
        //当前节点是head，pre为当前节点的前一节点，next为当前节点的下一节点
        //需要pre和next的目的是让当前节点从pre->head->next1->next2变成pre<-head next1->next2
        //即pre让节点可以反转所指方向，但反转之后如果不用next节点保存next1节点的话，此单链表就此断开了
        //所以需要用到pre和next两个节点
        //1->2->3->4->5
        //1<-2<-3 4->5
        while(head!=null){
            //做循环，如果当前节点不为空的话，始终执行此循环，此循环的目的就是让当前节点从指向next到指向pre
            //如此就可以做到反转链表的效果
            //先用next保存head的下一个节点的信息，保证单链表不会因为失去head节点的原next节点而就此断裂
            next = head.next;
            //保存完next，就可以让head从指向next变成指向pre了，代码如下
            head.next = pre;
            //head指向pre后，就继续依次反转下一个节点
            //让pre，head，next依次向后移动一个节点，继续下一次的指针反转
            pre = head;
            head = next;
        }
        //如果head为null的时候，pre就为最后一个节点了，但是链表已经反转完毕，pre就是反转后链表的第一个节点
        //直接输出pre就是我们想要得到的反转后的链表
        return pre;
    }
}
```

递归的方法其实是非常巧的，它利用递归走到链表的末端，然后再更新每一个node的next 值 ，实现链表的反转。而newhead 的值没有发生改变，为该链表的最后一个结点，所以，反转后，我们可以得到新链表的head。 注意关于链表问题的常见注意点的思考：
1、如果输入的头结点是 NULL，或者整个链表只有一个结点的时候
2、链表断裂的考虑

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

```
// 这里采用一种递归的方式，从链表节点的尾部进行反转指针即可。仔细体会，递归的简练。代码如下：
public class Solution {
    public ListNode ReverseList(ListNode head) {
           if(head == null || head.next == null) {
             return head;
         }
         ListNode preNode = ReverseList(head.next);
         head.next.next = head;
         head.next = null;
         return preNode;
    }
}
```

```
/**
 * 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。
 *
 * @author shijiacheng
 * @date 2018/2/23
 */
public class ReverseListSolution {
    /**
     * 依次遍历所有节点，将所有节点的next指向前一个节点
     */
    public ListNode ReverseList(ListNode head) {
        ListNode pre = null;
        ListNode next = null;
        while (head != null) {
            next = head.next;//持有下一个节点的引用
            head.next = pre;//将当前节点对下一个节点的引用指向前一个节点
            pre = head;//将前一个节点指向当前节点
            head = next;//将当前节点指向下一个节点
        }
        return pre;
    }
}
```

题目 合并两个排序的链表

考点 代码的鲁棒性 热点指数 74132 通过率 26.89%

具体题目

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。

容易犯的两个错误：

- 在写代码之前没有对合并的过程想清楚，最终合并出来的链表要么中间断开了要么没有做到递增排序
- 代码在鲁棒性方面存在问题，程序一旦有了特殊的输入（如空链表），就会崩溃

非递归版本：

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

}
```

```
//递归解法 参考高票答案
public class Solution {
    public ListNode Merge(ListNode list1,ListNode list2) {
        if (list1 == null) return list2;
        if (list2 == null) return list1;
        if (list1.val < list2.val) {
            list1.next = Merge(list1.next, list2);
            return list1;
        } else {
            list2.next = Merge(list1, list2.next);
            return list2;
        }
    }
}
```

```
/**
```

```
 * 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。
    *
 * @author shijiacheng
 * @date 2018/2/23
    */
    public class MergeSortedListsSolution {
    /**
     * 链表1的头结点的值小于链表2的头结点的值，因此链表1的头结点是合并后链表的头结点。

     * 在剩余的结点中，链表2的头结点的值小于链表1的头结点的值，因此链表2的头结点是剩

     * 余结点的头结点，把这个结点和之前已经合并好的链表的尾结点链接起来。
        */
        public ListNode Merge(ListNode list1, ListNode list2) {
        if (list1 == null) {
            return list2;
        } else if (list2 == null) {
            return list1;
        }
        ListNode mergeNode = null;
        if (list1.val < list2.val) {
            mergeNode = list1;
            mergeNode.next = Merge(list1.next, list2);
        } else {
            mergeNode = list2;
            mergeNode.next = Merge(list1, list2.next);
        }
        return mergeNode;
        }
}
```

题目 树的子结构

考点 代码的鲁棒性 热点指数 64057 通过率 23.05%

具体题目

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）

结题步骤

```
public class Solution {
    public static boolean HasSubtree(TreeNode root1, TreeNode root2) {
        boolean result = false;
        //当Tree1和Tree2都不为零的时候，才进行比较。否则直接返回false
        if (root2 != null && root1 != null) {
            //如果找到了对应Tree2的根节点的点
            if(root1.val == root2.val){
                //以这个根节点为为起点判断是否包含Tree2
                result = doesTree1HaveTree2(root1,root2);
            }
            //如果找不到，那么就再去root的左儿子当作起点，去判断时候包含Tree2
            if (!result) {
                result = HasSubtree(root1.left,root2);
            }


            //如果还找不到，那么就再去root的右儿子当作起点，去判断时候包含Tree2
            if (!result) {
                result = HasSubtree(root1.right,root2);
               }
            }
            //返回结果
        return result;
    }
    public static boolean doesTree1HaveTree2(TreeNode node1, TreeNode node2) {
        //如果Tree2已经遍历完了都能对应的上，返回true
        if (node2 == null) {
            return true;
        }
        //如果Tree2还没有遍历完，Tree1却遍历完了。返回false
        if (node1 == null) {
            return false;
        }
        //如果其中有一个点没有对应上，返回false
        if (node1.val != node2.val) {
                return false;
        }

        //如果根节点对应的上，那么就分别去子节点里面匹配
        return doesTree1HaveTree2(node1.left,node2.left) &&
doesTree1HaveTree2(node1.right,node2.right);
    }

/*思路：参考剑指offer
1、首先设置标志位result = false，因为一旦匹配成功result就设为true，
剩下的代码不会执行，如果匹配不成功，默认返回false
2、递归思想，如果根节点相同则递归调用DoesTree1HaveTree2（），
如果根节点不相同，则判断tree1的左子树和tree2是否相同，
再判断右子树和tree2是否相同
```

```
3、注意null的条件，HasSubTree中，如果两棵树都不为空才进行判断，
DoesTree1HasTree2中，如果Tree2为空，则说明第二棵树遍历完了，即匹配成功，
tree1为空有两种情况（1）如果tree1为空&&tree2不为空说明不匹配，
（2）如果tree1为空，tree2为空，说明匹配。
*/
public class Solution {
    public boolean HasSubtree(TreeNode root1,TreeNode root2) {
        boolean result = false;
            if(root1 != null && root2 != null){
                if(root1.val == root2.val){
                    result = DoesTree1HaveTree2(root1,root2);
                }
                if(!result){result = HasSubtree(root1.left, root2);}
                if(!result){result = HasSubtree(root1.right, root2);}
            }
            return result;
    }
    public boolean DoesTree1HaveTree2(TreeNode root1,TreeNode root2){
            if(root1 == null && root2 != null) return false;
            if(root2 == null) return true;
            if(root1.val != root2.val) return false;
            return DoesTree1HaveTree2(root1.left, root2.left) &&
DoesTree1HaveTree2(root1.right, root2.right);
        }
}
```

其中需要注意的是：

1. 测试用例如果pRoot2为空的话，返回的false而不是我们认为的空树应该是所有树的子树
2. 再判断是否子树的过程中，应该先判断pRoot2是否为空，为空则表明子树的所有节点都比较完了，应该是子树返回True
3. 要养成一个习惯，对任何一个树节点进行访问时，一定要提前检测该节点是否为空

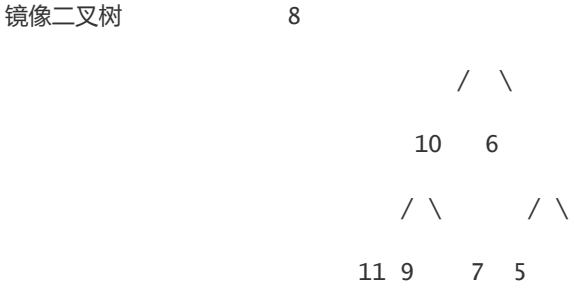
题目 二叉树的镜像

考点 面试思路 热点指数 69995 通过率 42.52%

具体题目

操作给定的二叉树，将其变换为源二叉树的镜像。

输入描述:二叉树的镜像定义：




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

```
/* 先前序遍历这棵树的每个结点，如果遍历到的结点有子结点，就交换它的两个子节点，
当交换完所有的非叶子结点的左右子结点之后，就得到了树的镜像 */
/**
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;
    public TreeNode(int val) {
        this.val = val;
    }
}
*/
public class Solution {
    public void Mirror(TreeNode root) {
        if(root == null)
```

```
            return;
        if(root.left == null && root.right == null)
            return;


        TreeNode pTemp = root.left;
        root.left = root.right;
        root.right = pTemp;

        if(root.left != null)
            Mirror(root.left);
        if(root.right != null)
            Mirror(root.right);
        }
    }
```

```
import java.util.Stack;
public class Solution {
    public void Mirror(TreeNode root) {
        if(root == null){
            return;
        }
        Stack<TreeNode> stack = new Stack<TreeNode>();
        stack.push(root);
        while(!stack.isEmpty()){
            TreeNode node = stack.pop();
            if(node.left != null||node.right != null){
                TreeNode temp = node.left;
                node.left = node.right;
                node.right = temp;
            }
            if(node.left!=null){
                                stack.push(node.left);
            }
            if(node.right!=null){
                stack.push(node.right);
            }
        }
    }
}
```

题目描述 解题思路 我们或许还记得递归的终极思想是数学归纳法，我们思考递归的时候一定不要去一步一步看它执行了啥，只会更绕。我们牢牢记住，思考的方式是我们首先假设子问题都已经完美处理，我只需要处理一下最终的问题即可，子问题的处理方式与最终那个处理方式一样，但是问题规模一定要以1的进制缩小。最后给一个递归出口条件即可。 对于本题，首先假设root的左右子树已经都处理好了，即左子树自身已经镜像了，右子树自身也镜像了，那么最后一步就是交换左右子树，问题解决。所以我只需要将root.left和root.right交换即可。下面进入递归，就是处理子问题。子问题的处理方式与root一样，只是要缩小问题规模，所以只需要考虑root.left是什么情况，root.right是什么情况即可。 我的答案

```
public class Solution {
        public void Mirror(TreeNode root) {
            reverseTree(root);
        }
        private void reverseTree(TreeNode root){
            //为空则结束
            if(root == null){
                return;
            }
```

```
            //假设root两边的子树自己都已经翻转成功了，那么只需要再将左右子树互换一下就成功了
            //交换root的左右子树
            swap(root);
            //左右子树翻转自己去处理就行了，我们规定每个子树的root都跟最终的root处理方式一样即可
            reverseTree(root.left);
            reverseTree(root.right);
        }
        private void swap(TreeNode root){
            TreeNode node = null;
            node = root.left;
            root.left = root.right;
            root.right = node;
        }
    }
```

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

题目 顺时针打印矩阵

考点 画图让抽象形象化 热点指数 58075 通过率 17.54%

具体题目

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

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

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

```
import java.util.ArrayList;
public class Solution {
    ArrayList  a=new ArrayList();    new一个数组 以便下面函数能调用
    public ArrayList printMatrix(int [][] matrix) {
       int tR=0;
       int tC=0;
       int dR=matrix.length-1;
       int dC=matrix[0].length-1;
        while(tR<=dR&&tC<=dC){ 左上边界最多到达右下边界 用于判断是否还是剥圈打印
       printEdge(matrix,tR++,tC++,dR--,dC--);
        }
      return a;
    }
    public  void printEdge(int [][] m,int tR,int tC,int dR,int dC){
        if(tR==dR){     先判断是否只是一横行 如果是 打印该横行的列（通常用于内圈）
            for(int i=tC;i<=dC;i++){
                a.add(m[tR][i]);
            }
        }
        else if(tC==dC){   再判断是否只是一竖列 如果是 打印该横行的列（通常用于内圈
            for(int i=tR;i<=dR;i++){
                a.add(m[i][tC]);
            }
        }
        else {
            int curC=tC;用2个变量储存 用于判断当前位置
            int curR=tR;
            while(curC!=dC){       当前位置未到达当前行的最右列 --》往右去
                a.add(m[tR][curC]);
            curC++;
            }
            while(curR!=dR){       当前位置未到达当前列的最底行 --》往下去
                a.add(m[curR][dC]);
                curR++;
            }
            while(curC!=tC){       当前位置未到达当前行的最左列 --》往左去
                a.add(m[dR][curC]);
                curC--;
            }
            while(curR!=tR){       当前位置未到达当前列的最顶行 --》往上去
                a.add(m[curR][tC]);
```

```
                curR--;
            }
        }
    }
}
```

1.每次都是一个圈，所以定义四个变量限定每次循环的界限：
  startRow,endRow,startCol,endCol；
2.分别把首行，末列，末行，首列的数据依次加入list；
3.注意不要重复加入某个点，每次都要限定界限。
代码如下：

```
    public ArrayList<Integer> printMatrix(int [][] matrix) {
        int row = matrix.length;
        if(row==0)
            return null;
        int col = matrix[0].length;
        if(col==0)
            return null;
        ArrayList<Integer> list = new ArrayList<Integer>();

        int startRow = 0;
        int endRow = row-1;
        int startCol = 0;
        int endCol = col-1;
        while(startRow<=endRow&&startCol<=endCol){
            //如果就剩下一行
            if(startRow==endRow){
                for(int i=startCol;i<=endCol;i++)
                    list.add(matrix[startRow][i]);
                return list;
            }
            //如果就剩下一列
            if(startCol==endCol){
                for(int i=startRow;i<=endRow;i++)
                    list.add(matrix[i][startCol]);
                return list;
            }
            //首行
            for(int i=startCol;i<=endCol;i++)
                list.add(matrix[startRow][i]);
            //末列
            for(int i=startRow+1;i<=endRow;i++)
                list.add(matrix[i][endCol]);
            //末行
            for(int i=endCol-1;i>=startCol;i--)
                list.add(matrix[endRow][i]);
            //首列
            for(int i=endRow-1;i>=startRow+1;i--)
                list.add(matrix[i][startCol]);

            startRow = startRow + 1;
            endRow = endRow - 1;
            startCol = startCol + 1;
            endCol = endCol - 1;
        }
        return list;
    }
```

```
/**
 * @description 顺时针打印矩阵
 * @author GongchuangSu
```

```
 * @since 2016.09.03
 * @explain 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.
 */
import java.util.*;
public class Solution{
    ArrayList<Integer> list = new ArrayList<>();

    public ArrayList<Integer> printMatrix(int [][] matrix) {
        int rows = matrix.length;
        int columns = matrix[0].length;
        int start = 0;
        while(rows > start*2 && columns > start*2){
            printMatrixInCircle(matrix, rows, columns, start);
            start++;
        }
        return list;
    }

    /**
     * 功能：打印一圈
     */
    public void printMatrixInCircle(int [][] matrix, int rows, int columns, int start){
        // 从左到右打印一行
        for(int i = start; i < columns - start; i++)
            list.add(matrix[start][i]);
        // 从上到下打印一列
        for(int j = start + 1; j < rows - start; j++)
            list.add(matrix[j][columns - start - 1]);
        // 从右到左打印一行
        for(int m = columns - start - 2; m >= start && rows - start - 1 > start; m--)
            list.add(matrix[rows - start - 1][m]);
        // 从下到上打印一列
        for(int n = rows - start - 2; n >= start + 1 && columns - start - 1 > start; n--)
            list.add(matrix[n][start]);
    }
}
```

```
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> printMatrix(int [][] matrix) {
       ArrayList<Integer> ls = new ArrayList<Integer>();
        int colStart = 0;
        int colEnd = matrix[0].length;
        int lineStart = 0;
        int lineEnd = matrix.length;
        int count = lineEnd * colEnd;
        if (matrix == null)
            return ls;
        while (count != 0) {
            for(int i = colStart;i<colEnd;i++){
                ls.add(matrix[lineStart][i]);
                count--;
            }
            lineStart++;
            if(count==0)
```

```
                break;
            for(int i = lineStart;i<lineEnd;i++){
                ls.add(matrix[i][colEnd-1]);
                count--;
            }
            colEnd--;
            if(count==0)
                break;
            for(int i = colEnd-1;i>=colStart;i--){
                ls.add(matrix[lineEnd-1][i]);
                count--;
            }
            lineEnd--;
            if(count==0)
                break;
            for(int i = lineEnd-1;i>=lineStart;i--){
                ls.add(matrix[i][colStart]);
                count--;
            }
            colStart++;
            if(count==0)
                break;
        }
        return ls;
    }
}
```

题目 包含min函数的栈

考点 举例让抽象具体化 热点指数 59617 通过率 30.98%

具体题目

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

```
/*
   解题思路：
   我们可以设计两个栈：StackDate和StackMin，一个就是普通的栈，另外一个存储push进来的最小值。
首先是push操作：
   每次压入的数据newNum都push进StackDate中，然后判断StackMin是否为空，如果为空那也把newNum同步压入
   StackMin里；如果不为空，就先比较newNum和StackMin中栈顶元素的大小，如果newNum较大，那就不压入StackMin
里，否则
   就同步压入StackMin里。如：
接着是pop操作
   先将StackDate中取出的数据value与StackMin的栈顶元素比较，因为对应push操作，value不可能小于
   StackMin中的栈顶元素，最多是相等。如果相等，那么StackMin中也取出数据，同时返回value，否则只是返回value就
可以了。
最后是getMin操作
    由上就可知，StackMin中存储的数据就是当前最小的，所以只要返回StackMin中的栈顶元素就可以了。
*/

import java.util.Stack;
public class Solution {
    private Stack<Integer> stackDate;
    private Stack<Integer> stackMin;

    public Solution(){
        stackDate = new Stack<>();
        stackMin = new Stack<>();
    }
    public void push(int node) {
        stackDate.push(node);
        if(stackMin.isEmpty()){
            stackMin.push(node);
        }else if(node <= stackMin.peek()){
            stackMin.push(node);
        }
    }

    public void pop() {
         if(stackDate.isEmpty()){
            throw new RuntimeException("This stack is empty!");
         }
         if(stackDate.peek() == stackMin.peek()){
             stackMin.pop();
         }
         stackDate.pop();
    }

    public int top() {
        if(stackDate.isEmpty()){
            throw new RuntimeException("This stack is empty!");
         }
        int value = stackDate.pop();
         if(value == stackMin.peek()){
             stackMin.pop();
```

```
         }
        return value;
    }

    public int min() {
        if(stackMin.isEmpty()){
            throw new RuntimeException("This stack is empty!");
        }else{
            return stackMin.peek();
        }
    }
}
```

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中所含最小元素的min函数（时间复杂度应为O（1））。 解题思路 思路：利用一个辅助栈来存放最小值 栈 3，4，2，5，1 辅助栈 3，3，2，2，1 每入栈一次，就与辅助栈顶比较大小，如果小就入栈，如果大就入栈当前的辅助栈顶； 当出栈时，辅助栈也要出栈 这种做法可以保证辅助栈顶一定都当前栈的最小值 我的答案

```
import java.util.Stack;
public class Solution {
    //存放元素
    Stack<Integer> stack1 = new Stack<Integer>();
    //存放当前stack1中的最小元素
    Stack<Integer> stack2 = new Stack<Integer>();
    //stack1直接塞，stack2要塞比栈顶小的元素，要不然就重新塞一下栈顶元素
    public void push(int node) {
        stack1.push(node);
        if(stack2.isEmpty() || stack2.peek() > node){
            stack2.push(node);
        }else{
            stack2.push(stack2.peek());
        }
    }
    //都要pop一下
    public void pop() throws Exception{
        if(stack1.isEmpty()){
           throw new Exception("no element valid");
        }
        stack1.pop();
        stack2.pop();
    }
    public int top(){
        if(stack1.isEmpty()){
           return 0;
        }
        return stack1.peek();
    }
    public int min(){
        if(stack2.isEmpty()){
           return 0;
        }
        return stack2.peek();
    }
}
```

题目 栈的压入、弹出序列

考点 举例让抽象具体化 热点指数 60348 通过率 28.69%

具体题目

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

【思路】借用一个辅助的栈，遍历压栈顺序，先讲第一个放入栈中，这里是1，然后判断栈顶元素是不是出栈顺序的第一个元素，这里是4，很显然1≠4，所以我们继续压栈，直到相等以后开始出栈，出栈一个元素，则将出栈顺序向后移动一位，直到不相等，这样循环等压栈顺序遍历完成，如果辅助栈还不为空，说明弹出序列不是该栈的弹出顺序。

举例：

入栈1,2,3,4,5

出栈4,5,3,2,1

首先1入辅助栈，此时栈顶1≠4，继续入栈2

此时栈顶2≠4，继续入栈3

此时栈顶3≠4，继续入栈4

此时栈顶4＝4，出栈4，弹出序列向后一位，此时为5，,辅助栈里面是1,2,3

此时栈顶3≠5，继续入栈5

此时栈顶5=5，出栈5,弹出序列向后一位，此时为3，,辅助栈里面是1,2,3

….

依次执行，最后辅助栈为空。如果不为空说明弹出序列不是该栈的弹出顺序。

```
 import java.util.ArrayList;
import java.util.Stack;
public class Solution {
    public boolean IsPopOrder(int [] pushA,int [] popA) {
        if(pushA.length == 0 || popA.length == 0)
            return false;
        Stack<Integer> s = new Stack<Integer>();
        //用于标识弹出序列的位置
        int popIndex = 0;
        for(int i = 0; i< pushA.length;i++){
            s.push(pushA[i]);
            //如果栈不为空，且栈顶元素等于弹出序列
            while(!s.empty() &&s.peek() == popA[popIndex]){
                //出栈
                s.pop();
                //弹出序列向后一位
                popIndex++;
            }
        }
        return s.empty();
    }
}
```

```
/*思路：先循环将pushA中的元素入栈，遍历的过程中检索popA可以pop的元素
**如果循环结束后栈还不空，则说明该序列不是pop序列。
**文字有点难说明白，看代码。
*/
import java.util.ArrayList;
import java.util.Stack;
public class Solution {
    public boolean IsPopOrder(ArrayList<Integer> pushA,ArrayList<Integer> popA) {
        Stack stack = new Stack();
        if( pushA.size() == 0 && popA.size() == 0 ) return true;
        for( int i=0,j=0; i < pushA.size(); i++ ){
```

```
            stack.push( pushA.get(i) );
            while( ( !stack.empty() )&& ( stack.peek() == popA.get(j) ) ){
                stack.pop();
                j ++;
            }
        }

        return stack.empty() == true;
    }
}
```

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可能为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，序列4,5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的） 解题思路 一开始都看不懂题目... 后来才好像明白是什么个意思... 假设有一串数字要将他们压栈: 1 2 3 4 5 如果这个栈是很大很大，那么一次性全部压进去，再出栈：5 4 3 2 1 但是，如果这个栈高度为4，会发生什么？ 1 2 3 4都顺利入栈，但是满了，那么要先出栈一个，才能入栈，那么就是先出4，然后压入5，随后再全部出栈：4 5 3 2 1 那么我总结了所有可能的出栈情况: 5 4 3 2 1//栈高度为5 4 5 3 2 1//栈高度为4 3 4 5 2 1//栈高度为3 2 3 4 5 1//栈高度为2 1 2 3 4 5//栈高度为1 借助一个辅助的栈，遍历压栈的顺序，依次放进辅助栈中。 对于每一个放进栈中的元素，栈顶元素都与出栈的popIndex对应位置的元素进行比较，是否相等，相等则popIndex++，再判断，直到为空或者不相等为止。 我的答案

```
import java.util.ArrayList;
import java.util.Stack;
public class Solution {
    public boolean IsPopOrder(int [] pushA,int [] popA) {
        //数组为空的情况
        if(pushA.length == 0 || popA.length == 0){
            return false;
        }
        //弹出序列的下表索引
        int popIndex = 0;
        //辅助栈
        Stack<Integer> stack = new Stack<Integer>();
        for(int i=0;i<pushA.length;i++){
            //不停地将pushA中的元素压入栈中，一旦栈顶元素与popA相等了，则开始出栈
            //不相等则继续入栈
            stack.push(pushA[i]);
            while(!stack.isEmpty() && stack.peek()==popA[popIndex]){
                stack.pop();
                popIndex++;
            }
        }
        //栈中没有元素了说明元素全部一致，并且符合弹出顺序，那么返回true
        return stack.isEmpty();
    }
}
```