链表

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class ListNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.next = None  
  
  
class Solution:  
 def FindFirstCommonNode(self, pHead1, pHead2):  
 """  
 思路：其实利用栈主要解决就是同时到达第一个结点的问题。那么从链表头出发如何同时到达第一个相同的结点呢? 链表的长度相同就  
 可以，其实就是走的结点数目相同。所以可以让其中长的链表先走几步，剩余的长度到短链表的长度相同。  
 """  
 # write code here  
 # if pHead1 is None or pHead2 is None:  
 # return  
 #  
 # cur1 = pHead1  
 # count1 = 1  
 # while cur1.next is not None:  
 # cur1 = cur1.next  
 # count1 += 1  
 #  
 # cur2 = pHead2  
 # count2 = 1  
 # while cur2.next is not None:  
 # cur2 = cur2.next  
 # count2 += 1  
 #  
 # if cur2 != cur1:  
 # return  
 #  
 # if count1 < count2:  
 # n = count2 - count1  
 # cur1 = pHead1  
 # cur2 = pHead2  
 # while cur1 != cur2:  
 # if n > 0:  
 # cur2 = cur2.next  
 # n -= 1  
 # else:  
 # cur2 = cur2.next  
 # cur1 = cur1.next  
 # else:  
 # n = count1 - count2  
 # cur1 = pHead1  
 # cur2 = pHead2  
 # while cur1 != cur2:  
 # if n > 0:  
 # cur1 = cur1.next  
 # n -= 1  
 # else:  
 # cur2 = cur2.next  
 # cur1 = cur1.next  
 # return cur1  
 """  
 有个思路，不需要存储链表的额外空间。也不需要提前知道链表的长度。看下面的链表例子：  
 0-1-2-3-4-5-null  
 a-b-4-5-null  
 代码的ifelse语句，对于某个指针p1来说，其实就是让它跑了连接好的的链表，长度就变成一样了。  
 如果有公共结点，那么指针一起走到末尾的部分，也就一定会重叠。看看下面指针的路径吧。  
 p1： 0-1-2-3-4-5-null(此时遇到ifelse)-a-b-4-5-null  
 p2: a-b-4-5-null(此时遇到ifelse)0-1-2-3-4-5-null  
 因此，两个指针所要遍历的链表就长度一样了  
 """  
 # cur1 = pHead1  
 # cur2 = pHead2  
 # while cur1 != cur2:  
 # cur1 = cur1.next if cur1 else pHead2  
 # cur2 = cur2.next if cur2 else pHead1  
 # return cur1  
 """  
 思路：两条相交的链表呈Y型。可以从两条链表尾部同时出发，最后一个相同的结点就是链表的第一个相同的结点。可以利用栈来实现。  
 时间复杂度有O(m + n), 空间复杂度为O(m + n)   
 """  
 stack1, stack2 = [], []  
 cur1, cur2 = pHead1, pHead2  
 while cur1 is not None:  
 stack1.append(cur1)  
 cur1 = cur1.next  
  
 while cur2 is not None:  
 stack2.append(cur2)  
 cur2 = cur2.next  
  
 first = None  
 while stack1 and stack2:  
 top1 = stack1.pop()  
 top2 = stack2.pop()  
 if top1 is top2:  
 first = top1  
 else:  
 break  
 return first

链表中环的入口结点

"""  
题目描述  
给一个链表，若其中包含环，请找出该链表的环的入口结点，否则，输出null。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class ListNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.next = None  
  
   
class Solution:  
 def EntryNodeOfLoop(self, pHead):  
 # write code here  
 if pHead is None:  
 return None  
  
 head = ListNode(0)  
 head.next = pHead  
  
 fast = head  
 low = head  
 meetNode = None  
 while fast is not None and fast.next is not None:  
 fast = fast.next.next  
 low = low.next  
 if fast == low:  
 meetNode = low  
 break  
  
 if meetNode is None:  
 return None  
  
 cur = head  
 while meetNode != cur:  
 meetNode = meetNode.next  
 cur = cur.next  
 return cur

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class ListNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.next = None  
  
  
class Solution:  
 def FindKthToTail(self, head, k):  
 # write code here  
 if head is None or k == 0:  
 return  
  
 fast = head  
 count = 1  
 while count < k:  
 if fast.next is None:  
 return  
 fast = fast.next  
 count += 1  
  
 cur = head  
 while fast.next is not None:  
 cur = cur.next  
 fast = fast.next  
 return cur

从尾到头打印链表

"""  
题目描述  
输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class ListNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.next = None  
  
  
class Solution:  
 # 返回从尾部到头部的列表值序列，例如[1,2,3]  
 def printListFromTailToHead(self, listNode):  
 # write code here  
 if listNode is None:  
 return []  
 if listNode.next is None:  
 return listNode  
  
 head = ListNode(0)  
 head.next = listNode  
  
 cur = head.next.next  
 head.next.next = None  
 head.next.next = None  
  
 while cur.next is not None:  
 next = cur.next  
 cur.next = head.next  
 head.next = cur  
 cur = next  
  
 cur.next = head.next  
 head.next = cur  
  
 li = []  
 while head.next is not None:  
 head = head.next  
 li.append(head.val)  
 return li

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  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class ListNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.next = None  
  
  
class Solution:  
 def deleteDuplication(self, pHead):  
 # write code here  
 if pHead is None or pHead.next is None:  
 return pHead  
  
 head = ListNode(0)  
 head.next = pHead  
  
 cur = head.next  
 pre = head  
 flag = False  
 while cur is not None and cur.next is not None:  
 if cur.val == cur.next.val:  
 cur = cur.next  
 flag = True  
 else:  
 if flag:  
 pre.next = cur.next  
 cur = cur.next  
 flag = False  
 else:  
 pre = cur  
 cur = cur.next  
 # 判断1，2，3，4，4，5和1，1，1，1  
 if flag:  
 cur = cur.next  
 pre.next = cur  
 return head.next

反转链表

"""  
题目描述  
输入一个链表，反转链表后，输出新链表的表头。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class ListNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.next = None  
  
  
class Solution:  
 # 返回ListNode  
 def ReverseList(self, pHead):  
 # write code here  
 if pHead is None:  
 return  
 if pHead.next is None:  
 return pHead  
  
 head = ListNode(0)  
 head.next = pHead  
  
 cur = head.next.next  
 head.next.next = None  
  
 while cur.next is not None:  
 next = cur.next  
 cur.next = head.next  
 head.next = cur  
 cur = next  
 cur.next = head.next  
 head.next = cur  
 return head.next

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class ListNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.next = None  
  
  
class Solution:  
 # 返回合并后列表  
 def Merge(self, pHead1, pHead2):  
 # write code here  
 head = ListNode(0)  
 cur = head  
 while pHead1 is not None and pHead2 is not None:  
 if pHead1.val <= pHead2.val:  
 cur.next = pHead1  
 cur = cur.next  
 pHead1 = pHead1.next  
 else:  
 cur.next = pHead2  
 cur = cur.next  
 pHead2 = pHead2.next  
  
 if pHead1 is not None:  
 cur.next = pHead1  
 if pHead2 is not None:  
 cur.next = pHead2  
 return head.next

复杂链表的复制

"""  
1. 把复制的结点链接在原始链表的每一对应结点后面  
2. 把复制的结点的random指针指向被复制结点的random指针的下一个结点  
3. 拆分成两个链表，奇数位置为原链表，偶数位置为复制链表，注意复制链表的最后一个结点的next指针不能跟原链表指向同一个空结点None，  
next指针要重新赋值None(判定程序会认定你没有完成复制）  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class RandomListNode:  
 def \_\_init\_\_(self, x):  
 self.label = x  
 self.next = None  
 self.random = None  
  
  
class Solution:  
 # 返回 RandomListNode  
 def Clone(self, pHead):  
 # write code here  
 if pHead is None:  
 return  
  
 cur = pHead  
 while cur is not None:  
 cur\_node = RandomListNode(cur.label)  
 cur\_node.next = cur.next  
 cur.next = cur\_node  
 cur = cur\_node.next  
  
 cur = pHead  
 while cur is not None:  
 cur\_node = cur.next  
 if cur.random is not None:  
 cur\_node.random = cur.random.next  
 cur = cur\_node.next  
  
 cur = pHead  
 head = cur.next  
 while cur is not None:  
 cur\_node = cur.next  
 cur.next = cur\_node.next  
 if cur\_node.next is None:  
 cur\_node.next = None  
 else:  
 cur\_node.next = cur\_node.next.next  
 cur = cur.next  
 return head

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)  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class LNode:  
 def \_\_init\_\_(self, item=None):  
 self.val = item  
 self.next = None  
  
  
class Solution:  
 def LastRemaining\_Solution(self, n, m):  
 # write code here  
 if n == 0 or m == 0:  
 return -1  
 head = LNode()  
 cur = head  
 for i in range(n):  
 node = LNode(i)  
 cur.next = node  
 cur = cur.next  
 cur.next = head.next  
  
 # 注意第一次寻找第m个小朋友时，从head结点开始，后边的寻找不从头结点开始，count=1  
 cur = head  
 count = 0  
 while cur.next != cur:  
 for i in range(count, m):  
 pre = cur  
 cur = cur.next  
 if pre.next == head.next:  
 head.next = cur.next  
 pre.next = head.next  
 cur = cur.next  
 else:  
 pre.next = cur.next  
 cur = cur.next  
 count = 1  
 return cur.val

栈与队列

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Stack:  
 def \_\_init\_\_(self):  
 self.stack = []  
  
 def length(self):  
 return len(self.stack)  
  
 def is\_empty(self):  
 return self.length() == 0  
  
 def push(self, item):  
 self.stack.append(item)  
  
 def pop(self):  
 if self.is\_empty():  
 return  
 else:  
 return self.stack.pop()  
  
 def peek(self):  
 if self.is\_empty():  
 return  
 else:  
 return self.stack[-1]  
  
  
class Solution:  
 def \_\_init\_\_(self):  
 self.enqueue = Stack()  
 self.dequeue = Stack()  
  
 def push(self, node):  
 # write code here  
 self.enqueue.push(node)  
  
 def pop(self):  
 # return xx  
 if self.enqueue.is\_empty() and self.dequeue.is\_empty():  
 return  
 if self.dequeue.is\_empty():  
 while not self.enqueue.is\_empty():  
 self.dequeue.push(self.enqueue.pop())  
 return self.dequeue.pop()  
 else:  
 return self.dequeue.pop()

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中所含最小元素的min函数（时间复杂度应为O（1））。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Stack:  
 def \_\_init\_\_(self):  
 self.stack = []  
  
 def length(self):  
 return len(self.stack)  
  
 def is\_empty(self):  
 return self.length() == 0  
  
 def push(self, item):  
 self.stack.append(item)  
  
 def pop(self):  
 if self.is\_empty():  
 return  
 else:  
 return self.stack.pop()  
  
 def peek(self):  
 if self.is\_empty():  
 return  
 else:  
 return self.stack[-1]  
  
  
class Solution:  
 def \_\_init\_\_(self):  
 self.min\_stack = Stack()  
 self.data\_stack = Stack()  
  
 def push(self, node):  
 # write code here  
 if self.min\_stack.is\_empty():  
 self.min\_stack.push(node)  
 elif self.min\_stack.peek() >= node:  
 self.min\_stack.push(node)  
 self.data\_stack.push(node)  
  
 def pop(self):  
 # write code here  
 if self.data\_stack.peek() == self.min\_stack.peek():  
 self.min\_stack.pop()  
 return self.data\_stack.pop()  
  
 def top(self):  
 # write code here  
 return self.data\_stack.peek()  
  
 def min(self):  
 # write code here  
 return self.min\_stack.peek()

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）  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def IsPopOrder(self, pushV, popV):  
 # write code here  
 # stack = []  
 # for i in pushV:  
 # stack.append(i)  
 # while len(stack) != 0:  
 # if stack[-1] == popV[0]:  
 # stack.pop()  
 # popV.pop(0)  
 # else:  
 # break  
 # return len(popV) == 0  
 stack = []  
 while popV:  
 if pushV and pushV[0] == popV[0]:  
 pushV.pop(0)  
 popV.pop(0)  
 elif stack and stack[-1] == popV[0]:  
 stack.pop()  
 popV.pop(0)  
 elif pushV:  
 stack.append(pushV.pop(0))  
 else:  
 return False  
 return True

树

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 def \_\_init\_\_(self):  
 self.pHead = None  
 self.pEnd = None  
  
 def Convert(self, pRootOfTree):  
 # write code here  
 if pRootOfTree is None:  
 return  
 self.Convert(pRootOfTree.left)  
 if self.pHead is None:  
 self.pHead = pRootOfTree  
 else:  
 pRootOfTree.left = self.pEnd  
 self.pEnd.right = pRootOfTree  
  
 self.pEnd = pRootOfTree  
 self.Convert(pRootOfTree.right)  
 return self.pHead

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def VerifySquenceOfBST(self, sequence):  
 # write code here  
 """  
 思路：  
 已知条件：后序序列最后一个值为root；二叉搜索树左子树值都比root小，右子树值都比root大。  
 1、确定root；  
 2、遍历序列（除去root结点），找到第一个大于root的位置，则该位置左边为左子树，右边为右子树；  
 3、遍历右子树，若发现有小于root的值，则直接返回false；  
 4、分别判断左子树和右子树是否仍是二叉搜索树（即递归步骤1、2、3）。  
  
 if sequence is None or sequence == []:  
 return False  
  
 length = len(sequence)  
 for i in range(length):  
 if sequence[i] > sequence[length - 1]:  
 break  
  
 for j in range(i, length):  
 if sequence[j] < sequence[length - 1]:  
 return False  
  
 left = True  
 if i > 0:  
 left = self.VerifySquenceOfBST(sequence[:i])  
 right = True  
 if i < length - 1:  
 right = self.VerifySquenceOfBST(sequence[i:-1])  
 return left and right  
 """  
  
 length = len(sequence)  
 if length == 0:  
 return False  
  
 while length:  
 index = 0  
 length -= 1  
 while sequence[index] < sequence[length]:  
 index += 1  
 while sequence[index] > sequence[length]:  
 index += 1  
  
 if index < length: # 找到大于最后元素的位置，然后后边的元素如果有小于的，则返回False  
 return False  
 return True

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Stack:  
 def \_\_init\_\_(self):  
 self.stack = []  
  
 def length(self):  
 return len(self.stack)  
  
 def is\_empty(self):  
 return self.length() == 0  
  
 def push(self, item):  
 self.stack.append(item)  
  
 def pop(self):  
 if self.is\_empty():  
 return  
 else:  
 return self.stack.pop()  
  
 def peek(self):  
 if self.is\_empty():  
 return  
 else:  
 return self.stack[-1]  
  
  
class Solution:  
 def \_\_init\_\_(self):  
 self.enqueue = Stack()  
 self.dequeue = Stack()  
  
 def push(self, node):  
 # write code here  
 self.enqueue.push(node)  
  
 def pop(self):  
 # return xx  
 if self.enqueue.is\_empty() and self.dequeue.is\_empty():  
 return  
 if self.dequeue.is\_empty():  
 while not self.enqueue.is\_empty():  
 self.dequeue.push(self.enqueue.pop())  
 return self.dequeue.pop()  
 else:  
 return self.dequeue.pop()

二叉搜索树的第k个结点

"""  
题目描述  
给定一棵二叉搜索树，请找出其中的第k小的结点。例如， （5，3，7，2，4，6，8） 中，按结点数值大小顺序第三小结点的值为4。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 # 返回对应节点TreeNode  
 def \_\_init\_\_(self):  
 self.res = []  
  
 def KthNode(self, pRoot, k):  
 # write code here  
 if pRoot is None:  
 return  
  
 self.mid\_order(pRoot)  
 if 0 < k <= len(self.res):  
 return self.res[k - 1]  
  
 def mid\_order(self, pRoot):  
 if pRoot is None:  
 return  
 self.mid\_order(pRoot.left)  
 self.res.append(pRoot)  
 self.mid\_order(pRoot.right)  
  
  
class Solution:  
 # 返回对应节点TreeNode  
 def \_\_init\_\_(self):  
 self.index = 0  
  
 def KthNode(self, pRoot, k):  
 # write code here  
 if pRoot is None:  
 return  
  
 node = self.KthNode(pRoot.left, k)  
 if node is not None:  
 return node  
  
 self.index += 1  
 if self.index == k:  
 return pRoot  
  
 node = self.KthNode(pRoot.right, k)  
 if node is not None:  
 return node

二叉树中和为某一值的路径

"""  
题目描述  
输入一颗二叉树的根节点和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经  
过的结点形成一条路径。(注意: 在返回值的list中，数组长度大的数组靠前)  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 # 返回二维列表，内部每个列表表示找到的路径  
 def \_\_init\_\_(self):  
 self.path = []  
 self.result = []  
  
 def FindPath(self, root, expectNumber):  
 # write code here  
 if root is None:

return []  
 self.path.append(root.val)  
 if root.left is None and root.right is None and sum(self.path) == expectNumber:  
 self.result.append(self.path[:])  
 self.FindPath(root.left, expectNumber)  
 self.FindPath(root.right, expectNumber)  
 self.path.pop()  
 return self.result

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeLinkNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
 self.next = None  
class Solution:  
 def GetNext(self, pNode):  
 # write code here  
 if pNode is None:  
 return None  
  
 # 当前结点有右结点，且该右结点没有左结点，则右结点即为下一个结点；若该右结点有左结点，则最后的左结点为下一个结点。  
 if pNode.right is not None:  
 p = pNode.right  
 while p.left is not None:  
 p = p.left  
 return p  
  
 # 当前结点没有右结点，寻找第一个父结点使的该父结点的父结点的左结点等于该父结点，则该父结点的父节点即为下一个结点  
 while pNode.next is not None:  
 if pNode.next.left == pNode:  
 return pNode.next  
 else:  
 pNode = pNode.next  
 return None

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 def TreeDepth(self, pRoot):  
 # write code here  
 # 广度优先搜索  
 # if pRoot is None:  
 # return 0  
 # queue = []  
 # queue.append(pRoot)  
 # deep = 0  
 # while queue:  
 # length = len(queue)  
 # deep += 1  
 # for i in range(length):  
 # cur\_node = queue.pop(0)  
 # if cur\_node.left is not None:  
 # queue.append(cur\_node.left)  
 # if cur\_node.right is not None:  
 # queue.append(cur\_node.right)  
 # return deep  
  
 # 递归  
 if pRoot is None:  
 return 0  
 return max(self.TreeDepth(pRoot.left), self.TreeDepth(pRoot.right)) + 1

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。  
输入描述:  
  
二叉树的镜像定义：源二叉树  
 8  
 / \  
 6 10  
 / \ / \  
 5 7 9 11  
镜像二叉树  
 8  
 / \  
 10 6  
 / \ / \  
 11 9 7 5  
  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 # 返回镜像树的根节点  
 def Mirror(self, root):  
 # write code here  
 if root is None:  
 return  
 self.Mirror(root.right)  
 self.Mirror(root.left)  
 root.left, root.right = root.right, root.left  
 return root

对称二叉树

"""  
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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 def isSymmetrical(self, pRoot):  
 # write code here  
 if pRoot is None:  
 return True  
 return self.isEqual(pRoot.left, pRoot.right)  
  
 def isEqual(self, lRoot, rRoot):  
 if lRoot is None and rRoot is None:  
 return True  
 if lRoot is not None and rRoot is None:  
 return False  
 if lRoot is None and rRoot is not None:  
 return False  
 if lRoot.val == rRoot.val:  
 return self.isEqual(lRoot.left, rRoot.right) and self.isEqual(lRoot.right, rRoot.left)  
 else:  
 return False

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 def IsBalanced\_Solution(self, pRoot):  
 # write code here  
 if pRoot is None:  
 return True  
 if abs(self.depth(pRoot.left) - self.depth(pRoot.right)) > 1:  
 return False  
 return self.IsBalanced\_Solution(pRoot.left) and self.IsBalanced\_Solution(pRoot.right)  
  
 def depth(self, pRoot):  
 if pRoot is None:  
 return 0  
 return max(self.depth(pRoot.left), self.depth(pRoot.right)) + 1

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  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 def \_\_init\_\_(self):  
 self.index = -1  
  
 def Serialize(self, root):  
 # write code here  
 if root is None:  
 return '#,'  
 # self.s += root.val  
 # self.Serialize(root.left)  
 # self.Serialize(root.right)  
 return str(root.val) + ',' + self.Serialize(root.left) + self.Serialize(root.right)  
  
 def Deserialize(self, s):  
 # write code here  
 self.index += 1  
 li = s.split(',')  
 if self.index > len(li):  
 return None  
  
 root = None  
 if li[self.index] != '#':  
 root = TreeNode(int(li[self.index]))  
 root.left = self.Deserialize(s)  
 root.right = self.Deserialize(s)  
 return root

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 # 返回二维列表[[1,2],[4,5]]  
 def Print(self, pRoot):  
 # write code here  
 if pRoot is None:  
 return []  
  
 sub = []  
 queue = []  
 queue.append(pRoot)  
  
 while queue:  
 length = len(queue)  
 res = []  
 for i in range(length):  
 cur\_node = queue.pop(0)  
 res.append(cur\_node.val)  
 if cur\_node.left is not None:  
 queue.append(cur\_node.left)  
 if cur\_node.right is not None:  
 queue.append(cur\_node.right)  
 sub.append(res)  
 return sub

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
   
class Solution:  
 def Print(self, pRoot):  
 # write code here  
 if pRoot is None:  
 return []  
  
 stack1 = []  
 stack2 = []  
 sub = []  
 flag = True  
  
 stack1.append(pRoot)  
 while stack1 or stack2:  
 res = []  
 if flag:  
 length1 = len(stack1)  
 for i in range(length1):  
 cur\_node = stack1.pop()  
 res.append(cur\_node.val)  
 if cur\_node.left is not None:  
 stack2.append(cur\_node.left)  
 if cur\_node.right is not None:  
 stack2.append(cur\_node.right)  
 else:  
 length2 = len(stack2)  
 for i in range(length2):  
 cur\_node = stack2.pop()  
 res.append(cur\_node.val)  
 if cur\_node.right is not None:  
 stack1.append(cur\_node.right)  
 if cur\_node.left is not None:  
 stack1.append(cur\_node.left)  
 sub.append(res)  
 flag = not flag  
 return sub

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）  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
  
class Solution:  
 def HasSubtree(self, pRoot1, pRoot2):  
 # write code here  
 result = False  
 if pRoot1 is not None and pRoot2 is not None:  
 result = self.is\_equal(pRoot1, pRoot2)  
 if not result:  
 result = self.HasSubtree(pRoot1.left, pRoot2)  
 if not result:  
 result = self.HasSubtree(pRoot1.right, pRoot2)  
 return result  
  
 def is\_equal(self, pRoot1, pRoot2):  
 if pRoot2 is None:  
 return True  
 if pRoot1 is None:  
 return False  
 if pRoot1.val == pRoot2.val:  
 return self.is\_equal(pRoot1.left, pRoot2.left) and

self.is\_equal(pRoot1.right, pRoot2.right)  
 else:  
 return False

数组与字符串

二维数组中的查找

"""  
题目描述  
在一个二维数组中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一  
个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 # array 二维列表  
 def Find(self, target, array):  
 # write code here  
 i = 0  
 j = len(array[0]) - 1  
 while i < len(array) and j > -1:  
 if array[i][j] < target:  
 i += 1  
 elif array[i][j] > target:  
 j -= 1  
 else:  
 return True  
 return False

不用加减乘除做加法

"""  
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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def Add(self, num1, num2):  
 # write code here  
 # 由于题目要求不能使用四则运算，那么就需要考虑使用位运算  
 # 两个数相加可以看成两个数的每个位先相加，但不进位，然后在加上进位的数值  
 # 如12+8可以看成1+0=1 2+8=0，由于2+8有进位，所以结果就是10+10=20  
 # 二进制中可以表示为1000+1100 先每个位置相加不进位，  
 # 则0+0=0 0+1=1 1+0=1 1+1=0这个就是按位异或运算  
 # 对于1+1出现进位，我们可以使用按位与运算然后在将结果左移一位  
 # 最后将上面两步的结果相加，相加的时候依然要考虑进位的情况，直到不产生进位  
 # 注意python没有无符号右移操作，所以需要越界检查  
 # 按位与运算：相同位的两个数字都为1，则为1；若有一个不为1，则为0。  
 # 按位异或运算：相同位不同则为1，相同则为0。  
 while num2 != 0:  
 sums = (num1 ^ num2) & 0xffffffff  
 carry = (num1 & num2) << 1 & 0xffffffff  
 num1 = sums  
 num2 = carry  
 if num1 <= 0x7fffffff:  
 return num1  
 else:  
 num1 = ~(num1 ^ 0xffffffff)  
 return num1

丑数

"""  
题目描述  
把只包含质因子2、3和5的数称作丑数（Ugly Number）。例如6、8都是丑数，但14不是，因为它包含质因子7。 习惯上我们把1当做是第一个丑数。求按从小到大的顺序的第N个丑数。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def GetUglyNumber\_Solution(self, index):  
 # write code here  
 """  
 实现思路：  
 只需要记录三个指针显示到达哪一步；“|”表示指针,arr表示丑数数组；  
 （1）1  
 |2  
 |3  
 |5  
 目前指针指向0,0,0，队列头arr[0] \* 2 = 2, arr[0] \* 3 = 3, arr[0] \* 5 = 5  
 （2）1 2  
 2 |4  
 |3 6  
 |5 10  
 目前指针指向1,0,0，队列头arr[1] \* 2 = 4, arr[0] \* 3 = 3, arr[0] \* 5 = 5  
 （3）1 2 3  
 2| 4 6  
 3 |6 9  
 |5 10 15  
 目前指针指向1,1,0，队列头arr[1] \* 2 = 4, arr[1] \* 3 = 6, arr[0] \* 5 = 5  
 """  
 if index < 7:  
 return index  
  
 p2 = p3 = p5 = 0  
 res = [1]  
 while len(res) < index:  
 min\_num = min(res[p2] \* 2, min(res[p3] \* 3, res[p5] \* 5))  
 if res[p2] \* 2 == min\_num:  
 p2 += 1  
 if res[p3] \* 3 == min\_num:  
 p3 += 1  
 if res[p5] \* 5 == min\_num:  
 p5 += 1  
 res.append(min\_num)  
 return res[-1]

二进制中1的个数

"""  
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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def NumberOf1(self, n):  
 # write code here  
 """  
 因为负数是以补码的形式存在的，python负数与0xffffffff按位与  
 正好求得该负数的补码状态  
 """  
 if n < 0:  
 n = n & 0xffffffff  
  
 num = 0  
 while n != 0:  
 n = n & (n - 1)  
 num += 1  
 return num  
 # return bin(n & 0xffffffff).count('1')

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（先后次序不同算不同的结果）。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def jumpFloor(self, number):  
 # write code here  
 res = [1, 1, 2]  
 while len(res) <= number:  
 res.append(res[-1] + res[-2])  
 return res[number]

变态跳

"""  
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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def jumpFloorII(self, number):  
 # write code here  
 return 2 \*\* (number - 1)

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，使得他们的和正好是S，如果有多对数字的和等于S，输出两个数的乘积最小的。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def FindNumbersWithSum(self, array, tsum):  
 # write code here  
 # 从两边向中间移动  
 begin = 0  
 end = len(array) - 1  
  
 while begin < end:  
 sums = array[begin] + array[end]  
 if sums > tsum:  
 end -= 1  
 elif sums < tsum:  
 begin += 1  
 else:  
 return [array[begin], array[end]]  
 return []  
  
 # 哈希  
 # for i in array:  
 # j = tsum - i  
 # if j in array:  
 # return [i, j]  
 # return []

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!  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def FindContinuousSequence(self, tsum):  
 # write code here  
 if tsum == 1:  
 return []  
 begin = 1  
 end = 2  
 res = []  
  
 while begin < end:  
 sums = 0  
 queue = []  
 for i in range(begin, end + 1):  
 sums += i  
 queue.append(i)  
  
 if sums < tsum:  
 end += 1  
 elif sums > tsum:  
 begin += 1  
 else:  
 res.append(queue)  
 begin += 1  
 end += 1  
   
 if end > tsum // 2 + 1:  
 break  
 return res

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  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def Permutation(self, ss):  
 # write code here  
 if len(ss) == 0:  
 return []  
 if len(ss) == 1:  
 return [ss]  
 res = set()  
 for i in range(len(ss)):  
 for j in self.Permutation(ss[:i] + ss[i + 1:]):  
 res.add(ss[i] + j)  
 return sorted(res)

字符流中第一个不重复的字符

"""  
题目描述  
请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符"go"时，第一个只出现一次的字符是"g"。当从该字符流中读出前六个字符“google"时，第一个只出现一次的字符是"l"。  
输出描述:  
如果当前字符流没有存在出现一次的字符，返回#字符。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 # 返回对应char  
 def \_\_init\_\_(self):  
 self.in\_stream = []  
 self.has\_map = {}  
  
 def FirstAppearingOnce(self):  
 # write code here  
 for i in self.in\_stream:  
 if self.has\_map[i] == 1:  
 return i  
 return '#'  
  
 def Insert(self, char):  
 # write code here  
 self.in\_stream.append(char)  
 if char not in self.has\_map:  
 self.has\_map[char] = 1  
 else:  
 self.has\_map[char] = 2

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！  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def LeftRotateString(self, s, n):  
 # write code here  
 # if s == '':  
 # return ''  
 # li = list(s)  
 # for i in range(n):  
 # li.append(li.pop(0))  
 # return ''.join(li)  
 # 原理：YX = (XTY T)T   
 return (s[:n][::-1] + s[n:][::-1])[::-1]

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， 如果牌能组成顺子就输出  
true，否则就输出false。为了方便起见,你可以认为大小王是0。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def IsContinuous(self, numbers):  
 # write code here  
 if numbers == []:  
 return False  
 numbers = sorted(numbers)  
 count\_0 = numbers.count(0)  
 length = len(numbers) - 1  
 for i in range(length, count\_0, -1):  
 if numbers[i] == numbers[i - 1]:  
 return False  
 else:  
 count\_0 = count\_0 - (abs(numbers[i] - numbers[i - 1]) - 1)  
 if count\_0 < 0:  
 return False  
 else:  
 return True

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(实现Integer.valueOf(string)的功能，但是string不符合数字要求时返回0)，要求不能使用字符串转换整数的库  
函数。 数值为0或者字符串不是一个合法的数值则返回0。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def StrToInt(self, s):  
 # write code here  
 if s == '':  
 return 0  
  
 for i in s[1:]:  
 if i < '0' or i > '9':  
 return 0  
  
 if s[0] != '-' and s[0] != '+' and s[0] < '0' and s[0] > '9':  
 return 0  
  
 sums = 0  
 for index, value in enumerate(s[::-1]):  
 if value == '-':  
 return -sums  
 if value == '+':  
 return sums  
 sums += (10 \*\* index) \* (ord(value) - 48)  
 return sums

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def PrintMinNumber(self, numbers):  
 # write code here  
 if numbers == []:  
 return ''  
 length = len(numbers)  
 for i in range(length - 1):  
 for j in range(length - 1 - i):  
 if str(numbers[j]) + str(numbers[j + 1]) > str(numbers[j + 1]) + str(numbers[i]):  
 numbers[j + 1], numbers[j] = numbers[j], numbers[j + 1]  
 return int(''.join(map(str, numbers)))

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def Power(self, base, exponent):  
 # write code here  
 # return base \*\* exponent  
 flip = exponent  
 if exponent == 0:  
 return 1  
 if exponent <= 0:  
 exponent = -exponent  
  
 num = 1  
 while exponent != 0:  
 if exponent & 1 == 1:  
 num \*= base  
 base \*= base  
 exponent >>= 1  
  
 if flip < 0:  
 return 1 / num  
 else:  
 return num

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def GetNumberOfK(self, data, k):  
 # write code here  
 # 先找到最左侧的  
 begin = 0  
 end = len(data) - 1  
 first = None  
 while begin <= end:  
 mid = (begin + end) // 2  
 if data[mid] < k:  
 begin = mid + 1  
 elif data[mid] > k:  
 end = mid - 1  
 elif begin == mid or data[mid - 1] != k:  
 first = mid  
 break  
 else:  
 end -= 1  
  
 # 在找到最右侧的  
 begin = 0  
 end = len(data) - 1  
 second = None  
 while begin <= end:  
 mid = (begin + end) // 2  
 if data[mid] < k:  
 begin = mid + 1  
 elif data[mid] > k:  
 end = mid - 1  
 elif mid == end or data[mid + 1] != k:  
 second = mid  
 break  
 else:  
 begin += 1  
  
 # 个数等于左侧-右侧+1  
 if first is None:  
 return 0  
 else:  
 return second - first + 1

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。我们使用Insert()方法读取数据流，使用GetMedian()方法获取当前读取数  
据的中位数。  
"""  
  
  
# 利用堆，有待改进  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def \_\_init\_\_(self):  
 self.res = []  
  
 def Insert(self, num):  
 # write code here  
 self.res.append(num)  
  
 def GetMedian(self):  
 # write code here  
 res = sorted(self.res)  
 length = len(res) // 2  
  
 if len(res) % 2 == 0:  
  
 return (res[length - 1] + res[length]) / 2  
 else:  
 return res[length]

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def MoreThanHalfNum\_Solution(self, numbers):  
 # write code here  
 count = 1  
 num = numbers[0]  
 length = len(numbers)  
 for i in range(1, length):  
 if num == numbers[i]:  
 count += 1  
 else:  
 count -= 1  
  
 if count == 0:  
 num = numbers[i]  
 count = 1  
  
 count = 0  
 for i in numbers:  
 if num == i:  
 count += 1  
 if count > length // 2:  
 return num  
 else:  
 return 0  
  
 # 哈希  
 # di = {}  
 # for i in numbers:  
 # if i in di:  
 # di[i] += 1  
 # else:  
 # di[i] = 1  
 #  
 # length = len(numbers) // 2  
 # for key, value in di.items():  
 # if value > length:  
 # return key  
 # return 0

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 # 哈希  
 # 返回[a,b] 其中ab是出现一次的两个数字  
 def FindNumsAppearOnce(self, array):  
 # write code here  
 # has\_map = {}  
 # for i in array:  
 # if i not in has\_map:  
 # has\_map[i] = 1  
 # else:  
 # has\_map[i] += 1  
 #  
 # res = []  
 # for key, value in has\_map.items():  
 # if value == 1:  
 # res.append(key)  
 # return res  
  
 # 异或  
 """  
 思路: 就是使用异或，但是与在成对出现的数字中查找一个单独的数字不同的是需要利用异或结果的最低位为1的flag将数组中的数字分为两类，一类是与flag按位与为0，另一类为不为0，这样再分别异或一次就能够找出这两个数。很是巧妙。其中有一个语法上容易忽略的坑：==的优先级比&高，所以&时需要加括号。  
 """  
 if array == []:  
 return  
 tmp = 0  
 for i in array:  
 tmp ^= i  
  
 idx = 0  
 while (tmp & 1) == 0:  
 tmp >>= 1  
 idx += 1  
  
 a = 0  
 b = 0  
 for i in array:  
 if self.isBit(i, idx):  
 a = a ^ i  
 else:  
 b = b ^ i  
  
 return [a, b]  
  
 def isBit(self, s, idx):  
 s >>= idx  
 return (s & 1) == 1

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 # 这里要特别注意~找到任意重复的一个值并赋值到duplication[0]  
 # 函数返回True/False  
 def duplicate(self, numbers, duplication):  
 # write code here  
  
 for j, i in enumerate(numbers):  
 c = numbers[:j] + numbers[j + 1:]  
 if i in c:  
 duplication[0] = i  
 return True  
 return False

整数中1出现的次数(从1到n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（从1 到 n 中1出现的次数）。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def NumberOf1Between1AndN\_Solution(self, n):  
 # write code here  
 # return ''.join(map(str, range(1, n + 1))).count('1')  
 num = 0  
 for i in range(1, n + 1):  
 while i:  
 if i % 10 == 1:  
 num += 1  
 i /= 10  
 return num

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（从0开始，第0项为0）。  
n<=39  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def Fibonacci(self, n):  
 # write code here  
 # if n == 0:  
 # return 0  
 # if n == 1:  
 # return 1  
 # return self.Fibonacci(n - 1) + self.Fibonacci(n - 2)  
 if n == 0:  
 return 0  
 if n == 1:  
 return 1  
 i = 0  
 j = 1  
 while n > 1:  
 i, j = j, i + j  
 n -= 1  
 return j

旋转数组的最小值

"""  
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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def minNumberInRotateArray(self, rotateArray):  
 # write code here  
 # if rotateArray == []:  
 # return 0  
 # length = len(rotateArray)  
 # for i in range(length - 1):  
 # if rotateArray[i] > rotateArray[i + 1]:  
 # return rotateArray[i + 1]  
 # return rotateArray[0]  
 return self.findMin(rotateArray, 0, len(rotateArray) - 1)  
  
 def findMin(self, rotateArray, low, high):  
 if low > high:  
 return 0  
 mid = (low + high) // 2  
  
 if rotateArray[mid] > rotateArray[mid + 1]:  
 return rotateArray[mid + 1]  
 elif rotateArray[mid] < rotateArray[mid - 1]:  
 return rotateArray[mid]  
 elif rotateArray[mid] > rotateArray[low]:  
 return self.findMin(rotateArray, mid + 1, high)  
 elif rotateArray[mid] < rotateArray[low]:  
 return self.findMin(rotateArray, low, mid - 1)  
 elif rotateArray[mid] == rotateArray[low] and rotateArray[mid] == rotateArray[high]:  
 return min(self.findMin(rotateArray, low, mid - 1), self.findMin(rotateArray, mid + 1, high))

替换空格

"""  
题目描述  
请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 # s 源字符串  
 def replaceSpace(self, s):  
 # write code here  
 li = list(s)  
 for index, key in enumerate(li):  
 if key == ' ':  
 li[index] = '%20'  
 return ''.join(li)

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,。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def GetLeastNumbers\_Solution(self, tinput, k):  
 # write code here  
 length = len(tinput)  
 if k > length or k == 0:  
 return []  
 res = []  
 for i in range(length):  
 min = i  
 for j in range(i + 1, length):  
 if tinput[j] < tinput[min]:  
 min = j  
 tinput[i], tinput[min] = tinput[min], tinput[i]  
 res.append(tinput[i])  
 if len(res) == k:  
 return res

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。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def multiply(self, A):  
 # write code here  
 """  
 A = [5,2,3,4]  
 left = [1, 5x1, 2x5x1, 3x2x5x1]  
 right = [1, 4x1, 3x4x1, 2x3x4x1]  
 """  
 length = len(A) - 1  
 left = [1]  
 right = [1]  
  
 for i in range(length):  
 left.append(A[i] \* left[i])  
 right.append(A[-i - 1] \* right[i])  
 return [left[i] \* right[-i - 1] for i in range(len(left))]

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）。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def Sum\_Solution(self, n):  
 # write code here  
 return n and n + self.Sum\_Solution(n - 1)

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]}。  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def maxInWindows(self, num, size):  
 # write code here  
 if size == 0:  
 return []  
 return [max(num[i:i + size]) for i in range(0, len(num) - size + 1)]

矩形覆盖

"""  
题目描述  
我们可以用2\*1的小矩形横着或者竖着去覆盖更大的矩形。请问用n个2\*1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？  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def rectCover(self, number):  
 # write code here  
 res = [0, 1, 2]  
 while len(res) <= number:  
 res.append(res[-1] + res[-2])  
 return res[number]

第一个只出现一次的字符

"""  
题目描述  
在一个字符串(0<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符,并返回它的位置, 如果没有则返回 -1（需要区分大小写）.  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def FirstNotRepeatingChar(self, s):  
 # write code here  
 di = {}  
 for i in s:  
 if i not in di:  
 di[i] = 1  
 else:  
 di[i] += 1  
  
 for index, value in enumerate(s):  
 if di[value] == 1:  
 return index  
 return -1

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？  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def ReverseSentence(self, s):  
 # write code here  
 # return ' '.join(s.split(' ')[::-1])  
 return ' '.join([i[::-1] for i in s.split(' ')])[::-1]

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  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def reOrderArray(self, array):  
 # write code here  
 # return [i for i in array if i % 2 == 1] + [j for j in array if j % 2 == 0]  
 length = len(array)  
 jishu = 0  
 for i in range(0, length):  
 if array[i] % 2 == 1:  
 for j in range(i, jishu, -1):  
 array[j], array[j - 1] = array[j - 1], array[j]  
 jishu += 1  
 return array

连续子数组的最大和

"""  
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)。给一个数组，返回它的最大连续子序列的和，你会不会被他忽悠住？(子向量的长度至少是1)  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 def FindGreatestSumOfSubArray(self, array):  
 # write code here  
 # 为正则加  
 for i in range(1, len(array)):  
 if array[i - 1] > 0:  
 array[i] += array[i - 1]  
 return max(array)

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.  
"""  
  
  
# -\*- coding:utf-8 -\*-  
class Solution:  
 # matrix类型为二维列表，需要返回列表  
 def printMatrix(self, matrix):  
 # write code here  
 result = []  
 while matrix:  
 for i in matrix.pop(0):  
 result.append(i)  
 matrix = list(zip(\*matrix))[::-1]  
 return result

排序

# 每次比较相邻的值，一次寻找出的是最大值  
def bubble\_sorted(array):  
 """  
 最好情况时间复杂度为O(n)  
 平均情况时间复杂度为O(n2)  
 最坏情况时间复杂度为O(n2)  
 空间复杂度为O(1)  
 稳定  
 """  
 length = len(array)  
 for i in range(length - 1):  
 for j in range(length - i - 1):  
 if array[j] > array[j + 1]:  
 array[j], array[j + 1] = array[j + 1], array[j]  
  
  
def bubble\_sorted\_k(array, k):  
 length = len(array)  
 for i in range(length): # 前k小的需要全比较  
 for j in range(length - 1 - i):  
 if array[j] < array[j + 1]:  
 array[j], array[j + 1] = array[j + 1], array[j]  
 if i == k - 1:  
 return array[-k]  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 arr0 = [54, 26, 93, 17, 77, 31, 44, 55, 20]  
 bubble\_sorted(arr0)  
 print(arr0)  
  
 arr1 = [54, 26, 93, 17, 77, 31, 44, 55, 20]  
 print(bubble\_sorted\_k(arr1, 5))

# 每次找到后续的最小值索引，与当前的外层循环的值交换  
def select\_sorted(array):  
 """  
 最好情况时间复杂度为O(n2)  
 平均情况时间复杂度为O(n2)  
 最坏情况时间复杂度为O(n2)  
 空间复杂度为O(1)  
 不稳定  
 """  
 length = len(array)  
 for i in range(length - 1):  
 min = i  
 for j in range(i + 1, length):  
 if array[j] < array[min]:  
 min = j  
 array[min], array[i] = array[i], array[min]  
  
  
def select\_sorted\_k(array, k):  
 length = len(array)  
 for i in range(length): # 前k小的要全比较  
 min = i  
 for j in range(i + 1, length):  
 if array[j] < array[min]:  
 min = j  
 array[min], array[i] = array[i], array[min]  
 if i == k - 1:  
 return array[i]  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 arr0 = [54, 26, 93, 17, 77, 31, 44, 55, 20]  
 select\_sorted(arr0)  
 print(arr0)  
  
 arr1 = [54, 26, 93, 17, 77, 31, 44, 55, 20]  
 print(select\_sorted\_k(arr1, 4))

# 当前元素从索引1开始与前面有序的数列，从后往前进行比较  
def insert\_sorted(array):  
 """  
 最好情况时间复杂度为O(n)  
 平均情况时间复杂度为O(n2)  
 最坏情况时间复杂度为O(n2)  
 空间复杂度为O(1)  
 稳定  
 """  
 length = len(array)  
  
 for i in range(length - 1):  
 for j in range(i + 1, 0, -1):  
 if array[j] < array[j - 1]:  
 array[j], array[j - 1] = array[j - 1], array[j]  
 else:  
 break  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 arr0 = [54, 26, 93, 17, 77, 31, 44, 55, 20]  
 insert\_sorted(arr0)  
 print(arr0)

def shell\_sorted(array, k):  
 """  
 最好情况时间复杂度为O(n1.3)  
 平均情况时间复杂度为O(nlogn)  
 最坏情况时间复杂度为O(n2)  
 空间复杂度为O(1)  
 不稳定  
 """  
 length = len(array)  
 gap = k  
 while gap > 0:  
 for i in range(length - gap):  
 for j in range(i + gap, gap - 1, -gap):  
 if array[j] < array[j - gap]:  
 array[j], array[j - gap] = array[j - gap], array[j]  
 else:  
 break  
 gap //= 2  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 arr0 = [54, 26, 93, 17, 77, 31, 44, 55, 20]  
 shell\_sorted(arr0, 8)  
 print(arr0)

# 每次找到第一个元素应该在的位置，左侧小于，右侧大于，然后分别递归左侧，和右侧  
def quick\_sorted(array, start, end):  
 """  
 最好情况时间复杂度为O(nlogn)  
 平均情况时间复杂度为O(nlogn)  
 最坏情况时间复杂度为O(n2)  
 空间复杂度为O(logn)  
 不稳定  
 """  
 if start >= end: # 只有左指针小于右指针才进行比较，递归终止条件  
 return array  
  
 mid = array[start] # 每次第一个元素作为所要寻找的位置元素  
 low = start  
 high = end  
 while low < high: # 只用low<high才进行循环  
 while low < high and array[high] > mid: # high值大于中间值，high向左移动  
 high -= 1  
 array[low] = array[high] # 否则，high值给low值  
  
 while low < high and array[low] < mid: # low值小于中间值，low向右移动  
 low += 1  
 array[high] = array[low] # 否则，low值给high值  
  
 array[low] = mid # 当low=high时退出循环，middle给low值  
 quick\_sorted(array, start, low - 1) # 小于middle的值，进行比较  
 quick\_sorted(array, low + 1, end) # 大于middle的值，进行比较  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 arr0 = [54, 26, 93, 17, 77, 31, 44, 55, 20]  
 quick\_sorted(arr0, 0, len(arr0) - 1)  
 print(arr0)

# 分而治之，先整个数组分到最小，然后按照大小组合数组  
def merger\_sorted(array):  
 """  
 最好情况时间复杂度为O(nlogn)  
 平均情况时间复杂度为O(nlogn)  
 最坏情况时间复杂度为O(nlogn)  
 空间复杂度为O(n)  
 稳定  
 """  
 n = len(array)  
 if n <= 1:  
 return array  
  
 mid = n // 2  
 left = merger\_sorted(array[:mid])  
 right = merger\_sorted(array[mid:])  
  
 res = []  
 left\_pointer, right\_pointer = 0, 0  
 while left\_pointer < len(left) and right\_pointer < len(right):  
 if left[left\_pointer] < right[right\_pointer]:  
 res.append(left[left\_pointer])  
 left\_pointer += 1  
 else:  
 res.append(right[right\_pointer])  
 right\_pointer += 1  
  
 res += left[left\_pointer:]  
 res += right[right\_pointer:]  
  
 return res  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 arr0 = [54, 26, 93, 17, 77, 31, 44, 55, 20]  
 print(merger\_sorted(arr0))

# 堆通常指的是一颗完全二叉树  
def heap\_sorted(array):  
 length = len(array)  
 # 创建小根堆  
 for i in range((length - 1) // 2, -1, -1):  
 adjust\_heap(array, i, length)  
 # 堆排序  
 for j in range(length - 1, -1, -1):  
 array[0], array[j] = array[j], array[0]  
 adjust\_heap(array, 0, j)  
  
  
# 创建的是小根堆，输出的是升序，因为调整的时候，需要将父结点至于数组尾部  
def adjust\_heap(array, index, length):  
 lchild = 2 \* index + 1  
 rchild = 2 \* index + 2  
 if lchild < length:  
 if rchild < length and array[rchild] > array[lchild]:  
 k = rchild  
 else:  
 k = lchild  
  
 if array[index] < array[k]:  
 array[index], array[k] = array[k], array[index]  
 adjust\_heap(array, k, length)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 arr0 = [54, 26, 93, 17, 77, 31, 44, 55, 20]  
 heap\_sorted(arr0)  
 print(arr0)