剑指offer

剑指offer Python实现

1. 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
        if not array:
            return False
        row = len(array)
        col = len(array[0])
        i = 0
        j = col-1
        while i < row and j >= 0:
            if array[i][j] > target:
                j -= 1
            elif array[i][j] < target:
                i += 1
            else:
                return True
        return False
```

```
1. 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。
2. class Solution:
3.     # s 源字符串
4.     def replaceSpace(self, s):
5.         # write code here
6.         """
7.         s = s.replace(" ", "%20")
8.         return s
9.         """
10.         # 先统计空格的数量
11.         num_space = 0
12.         for i in s:
13.             if i == " ":
14.                 num_space += 1
15.         l = len(s)
16.         replace_s = [None for _ in range(l + 2 * num_space)]
17.         i = l-1
18.         j = len(replace_s)-1
19.         while i >= 0:
20.             if s[i] != " ":
21.                 replace_s[j] = s[i]
22.                 j -= 1
23.             else:
24.                 replace_s[j] = '0'
25.                 j -= 1
26.                 replace_s[j] = '2'
27.                 j -= 1
28.                 replace_s[j] = '%'
29.                 j -= 1
30.             i -= 1
31.         return "".join(replace_s)
```

```
输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。
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
        # 采用栈存储数据
        res = []
        while listNode:
            res.insert(0, listNode.val)
            listNode = listNode.next
        return res
        """
        '''
        # 直接采用递归的方法
        res = []
        def add_list(listNode):
            if not listNode:
                return
            add_list(listNode.next)
            res.append(listNode.val)
        add_list(listNode)
        return res
        '''
        # 直接采用递归的方法
        def add_list(listNode, res):
            if not listNode:
                return res
            res = add_list(listNode.next, res)
            res.append(listNode.val)
            return res
        return add_list(listNode, [])
```

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。

```
class Solution:
    # 返回构造的TreeNode根节点
    def reConstructBinaryTree(self, pre, tin):
        # write code here
        if len(pre)>0:
            # 根节点必定在pre的第一个数
            root = TreeNode(pre[0])
            # 在中序遍历数列中找到树根节点位置
            root_id = tin.index(pre[0])
            # 由于先序遍历的第一个是中序遍历的最后一个（子树中）， 所以pre从1开始，tin从root_id-1结束
            root.left = self.reConstructBinaryTree(pre[1:1+root_id], tin[:root_id])
            root.right = self.reConstructBinaryTree(pre[1+root_id:], tin[root_id+1:])
            return root
```

```
1.  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。
2.  class Solution:
3.      def __init__(self):
4.          self.s1 = []
5.          self.s2 = []
6.      def push(self, node):
7.          # write code here
8.          self.s1.append(node)
9.      def pop(self):
10.         # return xx
11.         if self.s2:
12.             return self.s2.pop()
13.         else:
14.             # 将s1中的数据依次取出放入s2中
15.             while self.s1:
16.                 self.s2.append(self.s1.pop())
17.             return self.s2.pop()
```

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。

```
class Solution:
    def minNumberInRotateArray(self, rotateArray):
        # write code here
        def getMin(low,high,arr):
            if low > high:
                return 0
            if low == high:
                return arr[low]
            mid = (low+high)//2
            if arr[mid-1] > arr[mid]:
                return arr[mid]
            elif arr[mid] > arr[mid+1]:
                return arr[mid+1]
            elif arr[mid] < arr[high]:
                return getMin(low, mid-1, arr)
            elif arr[mid] > arr[low]:
                return getMin(mid+1, high, arr)
            else:
                return min(getMin(mid+1, high, arr), getMin(low, mid-1, arr))
        return getMin(0, len(rotateArray)-1, rotateArray)
```

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

```
class Solution:
    def Fibonacci(self, n):
        if n == 0:
            return 0
        if n == 1:
            return 1
        # write code here
        fibNMinusOne = 0
        fibNMinusTwo = 1
        res = 0
        for i in range(2, n+1):
            res = fibNMinusOne + fibNMinusTwo
            fibNMinusOne = fibNMinusTwo
            fibNMinusTwo = res
        return res
```

```
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）。
class Solution:
    def jumpFloor(self, number):
        # write code here
        if number <= 0:
            return 0
        res = [1, 2]
        if number == 1:
            return res[0]
        if number == 2:
            return res[1]
        minus1 = 1
        minus2 = 2
        for i in range(3, number+1):
            res = minus1 + minus2
            minus1 = minus2
            minus2 = res
        return res
```

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。

```
class Solution:
    def jumpFloorII(self, number):
        # write code here
        if number <= 0:
            return 0
        if number == 1:
            return 1
        if number == 2:
            return 2
        minus = 2
        for i in range(3, number+1):
            res = 2*minus
            minus = res
        return res
```

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？

```
class Solution:
    def rectCover(self, number):
        # write code here
        if number <= 0:
            return 0
        if number == 1:
            return 1
        if number == 2:
            return 2
        minus1 = 1
        minus2 = 2
        res = 0
        for i in range(3, number+1):
            res = minus1 + minus2
            minus1 = minus2
            minus2 = res
        return res
```

```
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。
class Solution:
    def NumberOf1(self, n):
        # write code here
        '''
        count = 0
        # 由于这里有32,所以不会陷入死循环的状态
        for i in range(32):
            count += (1 & n>>i)
        return count

        #return sum([(n>>i & 1) for i in range(0,32)])
        '''
        count = 0
        # 由于python是动态语言，所以数的大小会根据需要自动改变，这里需要控制为32位
        while (n & 0xffffffff):
            count += 1
            n = (n-1) & n
        return count
```

```
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。
class Solution:
    def Power(self, base, exponent):
        # write code here
        # return base**exponent
        if base == 0 and exponent < 0:
            # 无意义
            return 0
        res = 1
        i = 1
        abs_exponent = abs(exponent)
        while i <= abs_exponent:
            res *= base
            i += 1
        if exponent <= 0:
            return 1/res
        else:
            return res
```

```
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。
class Solution:
    def reOrderArray(self, array):
        # write code here
        '''
        # 采用冒泡算法排序
        for i in range(len(array)-1):
            for j in range(0, len(array)-i-1):
                if array[j] % 2 == 0 and array[j+1] % 2 == 1:
                    array[j], array[j+1] = array[j+1], array[j]
        return array
        '''
        # 采用额外数组法
        res = []
        l = len(array)
        for i in range(len(array)):
            if array[l-i-1] % 2 != 0:
                res.insert(0, array[l-i-1])
            if array[i] % 2 == 0:
                res.append(array[i])
        return res
```

```
输入一个链表，输出该链表中倒数第k个结点。
class Solution:
    def FindKthToTail(self, head, k):
        # write code here
        if not head:
            return
        # python中的数据是没有头结点的
        l1 = head
        l2 = l1
        i = 0
        while i < k:
            l2 = l2.next
            i += 1
            if not l2 and i < k:
                return
        while l2:
            l1 = l1.next
            l2 = l2.next
        return l1
```

```
输入一个链表，反转链表后，输出新链表的表头。
class Solution:
    # 返回ListNode
    def ReverseList(self, pHead):
        # write code here
        # 就地转换法
        if not pHead:
            return
        pre = pHead
        cur = pHead.next
        pHead.next = None
        while cur:
            next = cur.next
            cur.next = pre
            pre = cur
            cur = next
        return pre
```

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。

```
class Solution:
    # 返回合并后列表
    def Merge(self, pHead1, pHead2):
        # write code here
        if not pHead1:
            return pHead2
        if not pHead2:
            return pHead1
        head = pHead1
        pre = pHead1
        next_p = pHead2.next
        while pHead1 and pHead2:
            if pHead1.val <= pHead2.val:
                pre = pHead1
                pHead1 = pHead1.next
            else:
                pre.next = pHead2
                pHead2.next = pHead1
                pre = pHead1
                pHead1 = pHead1.next
                pHead2 = next_p
                if pHead2:
                    next_p = pHead2.next
        # 判断是哪个指针为None
        if not pHead1:
            pre.next = pHead2
            return head
        if not pHead2:
            return head
```

```
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）
class Solution:
    def HasSubtree(self, pRoot1, pRoot2):
        # write code here
        if not pRoot2 or not pRoot1:
            return False
        return self.is_subtree(pRoot1, pRoot2) or self.HasSubtree(pRoot1.left, pRoot2) or self.HasSubtree(pRoot1.right, pRoot2)

    def is_subtree(self, A, B):
        if not B:
            return True
        if not A or A.val != B.val:
            return False
        return self.is_subtree(A.left, B.left) and self.is_subtree(A.right, B.right)
```

```
操作给定的二叉树，将其变换为源二叉树的镜像。
class Solution:
    # 返回镜像树的根节点
    def Mirror(self, root):
        # write code here
        if root:
            tmp = root.left
            root.left = root.right
            root.right = tmp
            self.Mirror(root.left)
            self.Mirror(root.right)
```

```
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.
class Solution:
    # matrix类型为二维列表，需要返回列表
    def printMatrix(self, matrix):
        # write code here
        if not matrix:
            return matrix
        res = []
        startX = 0
        while len(matrix) > 2*startX and len(matrix[0]) > 2*startX:
            res.extend(self.print_matrix_incircle(matrix, startX))
            startX += 1
        return res

    def print_matrix_incircle(self, matrix, start):
        res = []
        endX = len(matrix[0]) - 1 -start
        endY = len(matrix) - 1 - start

        # 从左往右打印,没有约束条件
        for i in range(start, endX+1):
            print(matrix[start][i])
            res.append(matrix[start][i])

        # 从上往下打印，至少有两行
        if endY > start:
            for i in range(start+1, endY+1):
                print(matrix[i][endX])
                res.append(matrix[i][endX])

        # 从右往左打印，至少有两行两列
        if start < endX and endY > start:
            for i in range(endX-1, start-1, -1):
                print(matrix[endY][i])
                res.append(matrix[endY][i])
        # 从下往上打印，至少有三行两列
        if start < endY-1 and start<endX:
            for i in range(endY-1, start, -1):
                print(matrix[i][start])
                res.append(matrix[i][start])
        return res
```

```
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 Solution:
    def __init__(self):
        self.stack = []
        self.stack2 = []
    def push(self, node):
        # write code here
        self.stack.append(node)
        if not self.stack2:
            self.stack2.append(node)
        elif node <= self.stack2[-1]:
            self.stack2.append(node)

    def pop(self):
        # write code here
        if self.stack[-1] == self.stack2[-1]:
            self.stack2.pop()
        return self.stack.pop()
    def top(self):
        # write code here
        return self.stack[-1]
    def min(self):
        # write code here
        return self.stack2[-1]
```

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）

```
class Solution:
    def IsPopOrder(self, pushV, popV):
        # write code here
        if len(pushV) != len(popV):
            return False
        i = 0
        j = 0
        stack = []
        while i < len(pushV):
            stack.append(pushV[i])
            i += 1

            while stack and stack[-1] == popV[j]:
                stack.pop()
                j += 1
        #return not stack and j == len(popV)
        return not stack
```

```
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。
from collections import deque
class Solution:
    # 返回从上到下每个节点值列表，例：[1,2,3]
    def PrintFromTopToBottom(self, root):
        # write code here
        if not root:
            return []
        res = []
        queue = deque()
        queue.append(root)
        while queue:
            root = queue.popleft()
            res.append(root.val)
            if root.left:
                queue.append(root.left)
            if root.right:
                queue.append(root.right)
        return res
```

```
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。
class Solution:
    def VerifySquenceOfBST(self, sequence):
        # write code here
        if not sequence:
            return False
        if len(sequence) == 1:
            return True
        root = sequence[-1]
        i = 0
        while i < len(sequence)-1:
            if sequence[i] > root:
                break
            i += 1
        j = i
        # 判断后面的数是否都大于root值
        while j < len(sequence)-1:
            if sequence[j] < root:
                return False
            j += 1
        if i == 0:
            return self.VerifySquenceOfBST(sequence[i:len(sequence)-1])
        elif i == len(sequence) - 1:
            return self.VerifySquenceOfBST(sequence[:i])
        else:
            return self.VerifySquenceOfBST(sequence[:i]) and self.VerifySquenceOfBST(sequence[i:len(sequence)-1])
```

```
输入一颗二叉树的跟节点和一个整数，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）
class Solution:
    # 返回二维列表，内部每个列表表示找到的路径
    def FindPath(self, root, expectNumber):
        # write code here
        res = []
        def equal_path(root, expectNumber, path, res):
            if not root:
                return
            path.append(root.val)
            if not root.left or not root.right:
                if sum(path) == expectNumber:
                    res.append([i for i in path])
            equal_path(root.left, expectNumber, path, res)
            equal_path(root.right, expectNumber, path, res)
            path.pop()
        equal_path(root, expectNumber, [], res)
        return res
```

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）

```
class Solution:
    # 返回 RandomListNode
    def Clone(self, pHead):
        # write code here
        '''
        # 空间换时间的方法
        if not pHead:
            return pHead
        source_copy = {}
        copy_head = RandomListNode(pHead.label)
        source_copy[id(pHead)] = copy_head
        copy_normal_p = copy_head
        normal_p = pHead.next
        # 对节点进行复制
        while normal_p:
            node = RandomListNode(normal_p.label)
            source_copy[id(normal_p)] = node
            copy_normal_p.next = node
            copy_normal_p = node
            normal_p = normal_p.next
        # 对特殊指针进行复制
        special_p = pHead
        copy_special_p = copy_head
        while special_p:
            if special_p.random:
                copy_special_p.random = source_copy[id(special_p.random)]
            special_p = special_p.next
            copy_special_p = copy_special_p.next
        return copy_head
        '''
        if not pHead:
            return pHead
        self.insert(pHead)
        self.dup_random(pHead)
        return self.split(pHead)
    # 嵌入-复制-分离法
    # 将复制主干内容嵌入原链表
    def insert(self, pHead):
```

```
        pre = pHead
        while pre:
            back = pre.next
            node = RandomListNode(pre.label)
            pre.next = node
            node.next = back
            pre = back
    # 复制随机指针
    def dup_random(self, pHead):
        pre = pHead
        dup = pHead.next
        while dup.next:
            if pre.random:
                dup.random = pre.random.next
            if dup.next:
                dup = dup.next.next
                pre = pre.next.next
     # 分离链表
    def split(self, pHead):
        '''
        pNode=pHead
        #pClonedHead=None
        #pCloned=None
        if pHead:
            pClonedHead=pCloned=pNode.next
            pNode.next=pCloned.next
            pNode=pCloned.next
        while pNode:
            pCloned.next=pNode.next
            pCloned=pCloned.next
            pNode.next=pCloned.next
            pNode=pNode.next
        return pClonedHead
    '''
        # 在分开链表的时候，必须把最后一个连接也要去掉，不去掉所以出了很大的问题
        pNode = pHead
        pCloneHead= pClone = pNode.next
        while pClone.next:
            pNode.next = pClone.next
            pNode = pClone.next
            pClone.next = pNode.next
```

```
81.             pClone = pNode.next
82.         # 切断最后一个连接
83.         pNode.next = None
84.         return pCloneHead
```

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。

```
class Solution:
    def __init__(self):
        self.pHead = None
        self.pEnd = None
    def Convert(self, pRootOfTree):
        # write code here
        if not pRootOfTree:
            return
        self.Convert(pRootOfTree.left)
        pRootOfTree.left = self.pEnd
        if not self.pHead:
            self.pHead = pRootOfTree
        else:
            self.pEnd.right = pRootOfTree
        self.pEnd = pRootOfTree
        self.Convert(pRootOfTree.right)
        return self.pHead
```

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。

```
class Solution:
    def Permutation(self, ss):
        # write code here
        if not ss:
            return []
        ss = list(ss)
        res = []
        self.perm(ss, res, 0)
        uniq = list(set(res))
        return sorted(uniq)

    def perm(self, ss, res, start):
        if start == len(ss)-1:
            res.append("".join(ss))
        else:
            i = start
            while i < len(ss):
                ss[i], ss[start] = ss[start], ss[i]
                self.perm(ss, res, start+1)
                ss[i], ss[start] = ss[start], ss[i]
                i += 1
```

```
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。
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        # write code here
        '''
        # 方法一：采用空间换时间
        count_dic = {}
        for i in range(len(numbers)):
            count_dic[numbers[i]] = count_dic.get(numbers[i], 0) + 1
            if count_dic[numbers[i]] > len(numbers) >> 1:
                return numbers[i]
        return 0
        '''
        # 方法二：采用partition方法，类快排的方法，找到中位数
        begin = 0
        end = len(numbers) - 1
        return self.get_more_than_half(numbers, begin, end)

    def get_more_than_half(self, numbers, begin, end):
        index = self.partition(numbers, begin, end)
        while index != len(numbers) >> 1:
            if index < len(numbers) >> 1:
                index = self.partition(numbers,index+1, end)
            elif index > len(numbers) >> 1:
                index = self.partition(numbers, begin, index-1)

        count = 0
        for i in range(len(numbers)):
            if numbers[i] == numbers[index]:
                count += 1
        return numbers[index] if count > len(numbers) >> 1 else 0

    def partition(self, numbers, begin, end):
        index = begin
        key = numbers[index]
        while begin < end:
            while begin < end and numbers[end] >= key:
                end -= 1
            numbers[begin] = numbers[end]
            while begin < end and numbers[begin] <= key:
```

```
                begin += 1
            numbers[end] = numbers[begin]
        numbers[begin] = key
        return begin
```

```
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,。
class Solution:
    def GetLeastNumbers_Solution(self, tinput, k):
        # write code here
        # 采用partition的方法
        if k == 0 or len(tinput) < k:
        #if not tinput or k == 0 or len(tinput) < k:
            return []
        start = 0
        end = len(tinput) -1
        index = self.partition(tinput, start, end)
        while index != k-1:
            if index > k-1:
                index = self.partition(tinput, start, index-1)
            elif index < k-1:
                index = self.partition(tinput, index+1, end)
        res = sorted(tinput[:k])
        return res

    def partition(self, tinput, start, end):
        key = tinput[start]
        while start < end:
            if start < end and tinput[end] >= key:
                end -= 1
            tinput[start] = tinput[end]
            if start < end and tinput[start] <= key:
                start += 1
            tinput[end] = tinput[start]
        tinput[start] = key
        return start
```

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）

```
class Solution:
    def FindGreatestSumOfSubArray(self, array):
        # write code here
        if not array:
            return 0
        nEnd = array[0]
        nAll = array[0]
        for i in range(1, len(array)):
            if nEnd + array[i] < array[i]:
                nEnd = array[i]
            else:
                nEnd += array[i]
            nAll = max(nEnd, nAll)
        return nAll
```

```
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）。
class Solution:
    def NumberOf1Between1AndN_Solution(self, n):
        # write code here
        res = 0
        while True:
            # 首先判断n的位数
            n_copy = n
            count = 0
            while n_copy:
                n_copy //= 10
                count += 1

            middle_num = 0
            for i in range(count - 1):
                middle_num += n // (10 ** i) % 10 * 10**i

            res += self.get_count_1(n, count, middle_num)

            n = middle_num
            if n < 2:
                break
        return res

    def get_count_1(self, n, count, middle_num):
        if n > 0 and n < 10:
            return 1
        if n < 1:
            return 0
        num = 0
        copy_n = n
        count_copy = count
        while count_copy > 1:
            copy_n //= 10
            count_copy -= 1
        if copy_n > 1:
            num += 10 ** (count - 1)
        else:
            num += middle_num+1
        num += copy_n * (count - 1) * 10 ** (count - 2)
        return num
```

```
1. 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。
2. class Solution:
3.     def PrintMinNumber(self, numbers):
4.         # write code here
5.         return "".join(sorted([str(i) for i in numbers], cmp = lambda x, y: int(x+y) - int(y+x)))
```

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。

```
class Solution:
    def GetUglyNumber_Solution(self, index):
        # write code here
        if index < 1:
            return 0
        ugly_number = [1]
        if index == 1:
            return 1

        m2 = 0
        m3 = 0
        m5 = 0
        i = 1
        while i < index:
            num = min(ugly_number[m2]*2, ugly_number[m3]*3, ugly_number[m5]*5)
            ugly_number.append(num)
            # 这里的if不能用elif，有可能该数为2或者3或者5的最小公倍数
            if ugly_number[m2]*2 <= num:
                m2 += 1
            if ugly_number[m3]*3 <= num:
                m3 += 1
            if ugly_number[m5]*5 <= num:
                m5 += 1
            i += 1
        return ugly_number[-1]
```

```
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）.
class Solution:
    def FirstNotRepeatingChar(self, s):
        # write code here
        # 建立hash表，第一次遍历s添加次数到hash表，第二次遍历s找到第一个只出现一次的字符
        res = {}
        for i in s:
            res[i] = res.get(i, 0) + 1
        for i in s:
            if res[i] == 1:
                return s.index(i)
        return -1
```

```
在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数P。并将P对1000000007取模的结果输出。 即输出P%1000000007
class Solution:
    def InversePairs(self, data):
        # write code here
        if not data:
            return 0
        data_copy = [i for i in data]
        count = self.InversePairsCore(data, data_copy, 0, len(data)-1)
        return count% 1000000007

    def InversePairsCore(self, data, data_copy, start, end):
        if start == end:
            data_copy[start] = data[start]
            return 0
        length = (end-start)//2
        left = self.InversePairsCore(data_copy, data, start, start+length)
        right = self.InversePairsCore(data_copy, data, start+length+1, end)
        i = start+length
        j = end
        indexCopy = end
        count = 0
        while i>=start and j >=start+length+1:
            if data[i]>data[j]:
                data_copy[indexCopy] = data[i]
                indexCopy -= 1
                i -= 1
                count += j-start-length
            else:
                data_copy[indexCopy] = data[j]
                indexCopy -= 1
                j -= 1
        while i >= start:
            data_copy[indexCopy] = data[i]
            i -= 1
            indexCopy -= 1
        while j >= start+length+1:
            data_copy[indexCopy] = data[j]
            j -= 1
            indexCopy -= 1
        return left + right + count
```

```
输入两个链表，找出它们的第一个公共结点。
class Solution:
    def FindFirstCommonNode(self, pHead1, pHead2):
        # write code here
        if not pHead1 or not pHead2:
            return None
        l1 = 0
        l2 = 0
        p1 = pHead1
        p2 = pHead2
        while p1:
            p1 = p1.next
            l1 += 1
        while p2:
            p2 = p2.next
            l2 += 1
        if l1 > l2:
            i = 0
            while i < abs(l1-l2):
                pHead1= pHead1.next
                i += 1
            while pHead1 and pHead2:
                if pHead1 == pHead2:
                    return pHead1
                else:
                    pHead1 = pHead1.next
                    pHead2 = pHead2.next
        else:
            i = 0
            while i < abs(l1-l2):
                pHead2 = pHead2.next
                i += 1
            while pHead1 and pHead2:
                if pHead1 == pHead2:
                    return pHead2
                else:
                    pHead1 = pHead1.next
                    pHead2 = pHead2.next
        return None
```

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
        if not data:
            return 0
        start = 0
        # end指针为数组下标
        end = len(data)-1
        flag = 0
        while start <= end:
            mid = (start+end) >> 1
            if data[mid] > k:
                end = mid-1
            elif data[mid] < k:
                start = mid+1
            else:
                flag = 1
                break
        if flag == 0:
            return 0
        count = 0
        i = mid
        while i >= 0 and data[i] == k:
            count += 1
            i -= 1
        j = mid+1
        while j < len(data) and data[j] == k:
            count += 1
            j += 1
        return count
```

```
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。
class Solution:
    def TreeDepth(self, pRoot):
        # write code here
        if not pRoot:
            return 0
        return max(1 + self.TreeDepth(pRoot.left), 1 + self.TreeDepth(pRoot.right))
```

```
输入一棵二叉树，判断该二叉树是否是平衡二叉树。
class Solution:
    def IsBalanced_Solution(self, pRoot):
        # write code here
        '''
        if not pRoot:
            return True
        return self.IsBanlanced(pRoot) and self.IsBalanced_Solution(pRoot.left) and self.IsBalanced_Solution(pRoot.right)

    def IsBanlanced(self, pRoot):
        if abs(self.height(pRoot.left) - self.height(pRoot.right)) <= 1:
            return True
        else:
            return False
    def height(self, pRoot):
        if not pRoot:
            return 0
        return max(1+self.height(pRoot.left), 1+self.height(pRoot.right))
        '''
        # 通过后续遍历自下往上进行遍历
        return self.dfs(pRoot) != -1

    def dfs(self, pRoot):
        if not pRoot:
            return 0
        left = self.dfs(pRoot.left)
        if left == -1:
            return -1
        right = self.dfs(pRoot.right)
        if right == -1:
            return -1
        if abs(left-right) > 1:
            return -1
        return max(left, right) + 1
```

```
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。
class Solution:
    # 返回[a,b] 其中ab是出现一次的两个数字
    def FindNumsAppearOnce(self, array):
        # write code here
        c = 0
        #for i in array:
         # c ^= i
        c = reduce(lambda x, y: x^y, array)
        k = 0
        d = c
        while d & 1 != 1:
            k += 1
            d = d >> 1
        s = []
        for i in array:
            if i >> k & 1 == 1:
                s.append(i)
        e = 0
        e = reduce(lambda x, y: x^y, s)
        return [c^e, e]
```

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!

```
class Solution:
    def FindContinuousSequence(self, tsum):
        # write code here
        # 建立二维数组
        sequence_sum = [[0] * (tsum / 2 + 1) for _ in range(tsum / 2 + 1)]
    # 为对角线数据赋予初始值
        for i in range(tsum / 2 + 1):
            sequence_sum[i][i] = i + 1
    # 动态规划填充数组信息，并记录为100的数量
        res = []
        for i in range(1, tsum / 2 + 1):
            for j in range(i):
                sequence_sum[i][j] = sequence_sum[i - 1][j] + (i + 1)
                if sequence_sum[i][j] == tsum:
                    res.append(range(j+1, i + 2))
        return res
```

```
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。
class Solution:
    def FindNumbersWithSum(self, array, tsum):
        # write code here
        if not isinstance(array, list):
            return []
        res = []
        begin = 0
        end = len(array)-1
        while begin < end:
            if array[begin] + array[end] > tsum:
                end -= 1
            elif array[begin] + array[end] < tsum:
                begin += 1
            else:
                # 通过数学公式证明，边缘的两个数字乘积一定小于中间两个数字乘积
                return [array[begin], array[end]]
        return []
```

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！

```
class Solution:
    def LeftRotateString(self, s, n):
        # write code here
        '''
        # 空间换时间
        if not s:
            return ""
        n %= len(s)
        return s[n:]+s[:n]
        '''
        # 翻转法
        if not s:
            return ""
        s = list(s)
        n %= len(s)
        begin = 0
        end = n-1
        while begin < end:
            s[begin], s[end] = s[end], s[begin]
            begin += 1
            end -= 1
        begin = n
        end = len(s)-1
        while begin < end:
            s[begin], s[end] = s[end], s[begin]
            begin += 1
            end -= 1
        begin = 0
        end = len(s)- 1
        while begin < end:
            s[begin], s[end] = s[end], s[begin]
            begin += 1
            end -= 1
        return "".join(s)
```

1. 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?

```
class Solution:
    def ReverseSentence(self, s):
        # write code here
        return " ".join([i[::-1] for i in s.split(" ")])[::-1]
```

1. 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。

```
2.  class Solution:
3.      def IsContinuous(self, numbers):
4.          # write code here
5.          if not numbers:
6.              return False
7.          numbers = sorted(numbers)
8.          count_0 = 0
9.          count_blank = 0
10.
11.         for i in range(len(numbers)):
12.             if numbers[i] == 0:
13.                 count_0 += 1
14.             elif numbers[i] != 0:
15.                 break
16.         for j in range(i, len(numbers)-1):
17.             if numbers[j+1] - numbers[j] != 1:
18.                 if numbers[j+1] - numbers[j] == 0:
19.                     return False
20.                 count_blank += numbers[j+1] - numbers[j]-1
21.         if count_0 >= count_blank:
22.             return True
23.         else:
24.             return False
```

1. 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)

```
2. class Solution:
3.     def LastRemaining_Solution(self, n, m):
4.         # write code here
5.
6.         # 动态规划，寻找规律
7.         if n < 1 or m < 1:
8.             return -1
9.         last = 0
10.        for i in range(2, n+1):
11.            last = (last+m)%i
12.        return last
13.
14.        # 循环链表模拟法
15.        # 依次循环m次，到达则删除该节点，同时判断node.next == node,如果True返回该节点
```

```
1. 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）。
2. class Solution:
3.     def Sum_Solution(self, n):
4.         # write code here
5.         # and 运算，如果n为False， 返回False， 如果True，返回后面的数
6.         return n and (n + self.Sum_Solution(n-1))
```

```
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。
class Solution:
    def Add(self, num1, num2):
        # write code here
        while True:
            res = (num1 ^ num2) & 0xffffffff
            carry = ((num1 & num2) << 1) & 0xffffffff
            if carry == 0:
                return res if res <= 0x7fffffff else ~(res ^ 0xffffffff)
            num1 = res
            num2 = carry
```

```
1. 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。
2. class Solution:
3.     def StrToInt(self, s):
4.         # write code here
5.         if not s:
6.             return 0
7.         if s[0] == '+':
8.             flag = 1
9.             s = s[1:]
10.        elif s[0] == '-':
11.            flag = 0
12.            s = s[1:]
13.        else:
14.            flag = -1
15.        asc_0 = ord('0')
16.        asc_9 = ord('9')
17.        res = 0
18.        jinwei = 1
19.        for i in range(len(s)-1, -1,-1):
20.            if ord(s[i]) >= asc_0 and ord(s[i]) <= asc_9:
21.                res += (ord(s[i]) - asc_0) * jinwei
22.                jinwei *= 10
23.            else:
24.                return 0
25.        return -res if flag == 0 else res
```

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。

```
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
        i = 0
        while True:
            if i >= len(numbers):
                return False
            if numbers[i] != i:
                if numbers[i] == numbers[numbers[i]]:
                    duplication[0] = numbers[i]
                    return True
                index = numbers[i]
                numbers[i], numbers[index] = numbers[index], numbers[i]
            else:
                i += 1
```

```
1. 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。
2. class Solution:
3.     def multiply(self, A):
4.         # write code here
5.         b = [1]*len(A)
6.         multi = 1
7.         for i in range(1, len(A)):
8.             multi *= A[i-1]
9.             b[i] = multi
10.        multi = 1
11.        for i in range(len(A)-2, -1, -1):
12.            multi *= A[i+1]
13.            b[i] *= multi
14.        return b
```

```
1. 请实现一个函数用来匹配包括'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符，而'*'表示它前面的字符可以出现任意次（包含0次）。 在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但是与"aa.a"和"ab*a"均不匹配
2. class Solution:
3.     # s, pattern都是字符串
4.     def match(self, s, pattern):
5.         # write code here
6.         # 两个字符串都为空，返回True
7.         if len(s) == 0 and len(pattern) == 0:
8.             return True
9.         if len(s) > 0 and len(pattern) == 0:
10.            return False
11.        # 考虑第二个模式字符串是否为*的情况
12.        if len(pattern) > 1 and pattern[1] == '*':
13.            if len(s) > 0 and (s[0] == pattern[0] or pattern[0] == '.'):
14.                return self.match(s, pattern[2:]) or self.match(s[1:], pattern[2:]) or self.match(s[1:], pattern)
15.            else:
16.                return self.match(s, pattern[2:])
17.        if len(s)>0 and (pattern[0] == '.' or pattern[0] == s[0]):
18.            return self.match(s[1:], pattern[1:])
19.        return False
```

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

```
class Solution:
    # s字符串
    def isNumeric(self, s):
        # write code here
        allow_char = ['+','-','0','1','2','3','4','5','6','7','8','9','e','E','.']
        already_action = 0
        after_e = 0
        count_dot = 0
        count_plus_mins = 0
        for i in range(len(s)):
            if s[i] not in allow_char:
                return False
            if s[i] in ['+','-'] and i != 0 and after_e == 0:
                return False
            if s[i] == '.' and after_e == 1:
                return False
            if s[i] == '.' and after_e == 0:
                count_dot += 1
                if count_dot > 1:
                    return False
            if s[i] in ['e', 'E'] and after_e == 1:
                return False
            if s[i] in ['+', '-'] and after_e == 1:
                count_plus_mins += 1
                if count_plus_mins > 1:
                    return False
            if s[i] in ['e', 'E']:
                after_e = 1
                if i == len(s)-1:
                    return False
        return True
```

```
1. 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"。
2. class Solution:
3.     def __init__(self):
4.         self.s=''
5.         self.count_dic = {}
6.     # 返回对应char
7.     def FirstAppearingOnce(self):
8.         # write code here
9.         # 利用hash表存储每个字符出现的次数，时间复杂度为O(n)
10.        for i in range(len(self.s)):
11.            if self.count_dic[self.s[i]] == 1:
12.                return self.s[i]
13.        return '#'
14.
15.    def Insert(self, char):
16.        # write code here
17.        self.s += char
18.        self.count_dic[char] = self.count_dic.get(char,0)+1
```

```
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。
class Solution:
    def EntryNodeOfLoop(self, pHead):
        # write code here
        if not pHead or not pHead.next:
            return None
        low = pHead
        fast = pHead
        while fast:
            low = low.next
            fast = fast.next.next
            if low == fast:
                break
        if not fast:
            return None
        low = pHead
        while low != fast:
            low= low.next
            fast = fast.next
        return low
```

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

```
class Solution:
    def deleteDuplication(self, pHead):
        # write code here
        '''
        if not pHead or not pHead.next:
            return pHead
        newHead = ListNode("a")
        newHead.next = pHead
        pre = newHead
        cur = pHead
        flag = 0
        while cur.next:
            if cur.val == cur.next.val:
                flag = 1
                cur.next = cur.next.next
            else:
                if flag == 1:
                    pre.next = cur.next
                    cur = cur.next
                    flag = 0
                else:
                    pre = pre.next
                    cur = cur.next
        if flag == 1:
            pre.next = cur.next
        return newHead.next
        '''
        # 采用递归的方式删除
        if not pHead or not pHead.next:
            return pHead
        pNext = pHead.next
        if pNext.val != pHead.val:
            pHead.next = self.deleteDuplication(pHead.next)
        else:
            while pNext and pNext.val == pHead.val:
                pNext = pNext.next
            if not pNext:
                return None
            else:
```

```
                pHead = self.deleteDuplication(pNext)
        return pHead
```

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。

```
class Solution:
    def GetNext(self, pNode):
        # write code here
        if not pNode:
            return pNode
        if pNode.right:
            r_node = pNode.right
            while r_node.left:
                r_node = r_node.left
            return r_node
        while pNode.next:
            father_node = pNode.next
            if father_node.left == pNode:
                return father_node
            pNode = father_node
```

```
1. 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。
2. class Solution:
3.     def isSymmetrical(self, pRoot):
4.         # write code here
5.         return self.judge_symmetry(pRoot, pRoot)
6. 
7.     def judge_symmetry(self, pRoot1, pRoot2):
8.         if not pRoot1 and not pRoot2:
9.             return True
10.        if not pRoot1 or not pRoot2:
11.            return False
12.        if pRoot1.val != pRoot2.val:
13.            return False
14.        return self.judge_symmetry(pRoot1.left, pRoot2.right) and self.judge_symmetry(pRoot1.right, pRoot2.left)
```

```
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。
from collections import deque
class Solution:
    def Print(self, pRoot):
        # write code here
        if not pRoot:
            return []
        res = deque([pRoot])
        tmp = []
        ret = []
        last = pRoot
        flag = 1
        while res:
            node = res.popleft()
            tmp.append(node.val)
            if node.left:
                res.append(node.left)
            if node.right:
                res.append(node.right)
            if node == last:
                ret.append(tmp if flag == 1 else tmp[::-1])
                tmp = []
                flag = -flag
                if res:
                    last = res[-1]
        return ret
```

```
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。
class Solution:
    # 返回二维列表[[1,2],[4,5]]
    def Print(self, pRoot):
        # write code here
        if not pRoot:
            return []
        from collections import deque
        q = deque([pRoot])
        ret = []
        tmp = []
        last = pRoot
        while q:
            node = q.popleft()
            tmp.append(node.val)
            if node.left:
                q.append(node.left)
            if node.right:
                q.append(node.right)
            if node == last:
                ret.append(tmp)
                tmp = []
                if q: last=q[-1]
        return ret
```

请实现两个函数，分别用来序列化和反序列化二叉树

```
class Solution:
    flag = -1
    def Serialize(self, root):
        # write code here
        if not root:
            return '$'
        return str(root.val)+','+self.Serialize(root.left)+','+self.Serialize(root.right)
    def Deserialize(self, s):
        # write code here
        self.flag += 1
        l = s.split(',')
        if self.flag >= len(s):
            return None
        root = None
        if l[self.flag] != '$':
            root = TreeNode(int(l[self.flag]))
            root.left = self.Deserialize(s)
            root.right = self.Deserialize(s)
        return root
```

```
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。
class Solution:
    # 返回对应节点TreeNode
    def KthNode(self, pRoot, k):
        # write code here
        if k <= 0 or not pRoot:
            return None
        buffer = self.find_k_num(pRoot, [])
        if len(buffer) < k:
            return None
        else:
            return buffer[k-1]

    def find_k_num(self, pRoot, buffer):
        if not pRoot:
            return buffer
        self.find_k_num(pRoot.left, buffer)
        buffer.append(pRoot)
        self.find_k_num(pRoot.right, buffer)
        return buffer
```

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。

```
class Solution:
    def __init__(self):
        self.data = []
    def Insert(self, num):
        # write code here
        # 采用二分法实现数据的有序插入
        if not self.data:
            self.data.append(num)
            return
        pre = 0
        back = len(self.data)-1
        while pre <= back:
            mid = (back+pre)//2
            if mid == 0:
                if num < self.data[0]:
                    self.data.insert(0, num)
                else:
                    self.data.insert(1, num)
                return
            if mid == len(self.data)-1:
                if num > self.data[-1]:
                    self.data.append(num)
                else:
                    self.data.insert(-1, num)
                return
            if num < self.data[mid]:
                back = mid-1
            elif num > self.data[mid]:
                pre = mid+1
            else:
                self.data.insert(mid, num)

    def GetMedian(self, data):
        # write code here
        if not self.data:
            return
        if len(self.data) % 2 == 1:
            return self.data[len(self.data)//2]
```

```
        else:
            return (self.data[len(self.data)//2-1] + self.data[len(self.data)//2])/2.0
```

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]}。

```
from collections import deque
class Solution:
    def maxInWindows(self, num, size):
        # write code here
        if not num or size > len(num) or size < 1:
            return []
        res = []
        window = deque()
        for i in range(size):
            while window and num[i] >= num[window[-1]]:
                window.pop()
            window.append(i)

        for i in range(size, len(num)):
            res.append(num[window[0]])
            while window and num[i] >= num[window[-1]]:
                window.pop()
            if window and i - window[0] >= size:
                window.popleft()
            window.append(i)
        res.append(num[window[0]])
        return res
```

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，则之后不能再次进入这个格子。 例如 a b c e s f c s a d e e 这样的3 X 4 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

```
class Solution:
    def hasPath(self, matrix, rows, cols, path):
        # write code here
        for i in range(rows):
            for j in range(cols):
                if matrix[i*cols+j] == path[0]:
                    if self.find(list(matrix), rows, cols, path[1:], i, j):
                        return True
        return False

    def find(self, matrix, rows, cols, path, i, j):
        if not path:
            return True
        matrix[i*cols+j] = '0'
        if j+1 < cols and matrix[i*cols+j+1] == path[0]:
            return self.find(matrix, rows, cols, path[1:],i,j+1)
        elif j-1 >= 0 and matrix[i*cols+j-1] == path[0]:
            return self.find(matrix, rows, cols, path[1:],i,j-1)
        elif i+1 < rows and matrix[(i+1)*cols+j] == path[0]:
            return self.find(matrix, rows, cols, path[1:],i+1,j)
        elif i-1 >= 0 and matrix[(i-1)*cols+j] == path[0]:
            return self.find(matrix, rows, cols,path[1:],i-1,j)
        else:
            return False
```

```
1. 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。 例如，当k为18时，机器人能够进入方格（35,37），因为3+5+3+7 = 18。但是，它不能进入方格（35,38），因为3+5+3+8 = 19。请问该机器人能够达到多少个格子?
2. class Solution:
3.     def movingCount(self, threshold, rows, cols):
4.         # write code here
5.
6.         visited = [1] * rows * cols
7.         if threshold < 0:
8.             return 0
9.         return self.get_moving_count(threshold, rows, cols, 0, 0, 1, visited)
10.
11.
12.     def get_moving_count(self, threshold, rows, cols, i, j, count, visited):
13.         visited[i * cols + j] = 0
14.         if j + 1 < cols and visited[i * cols + j + 1] and self.getDigitSum(i, j + 1) <= threshold:
15.             count = self.get_moving_count(threshold, rows, cols, i, j + 1, count+1, visited)
16.         if j - 1 >= 0 and visited[i * cols + j - 1] and self.getDigitSum(i, j - 1) < threshold:
17.             count = self.get_moving_count(threshold, rows, cols, i, j - 1, count+1, visited)
18.         if i + 1 < rows and visited[(i + 1) * cols + j] and self.getDigitSum(i+1, j) <= threshold:
19.             count = self.get_moving_count(threshold, rows, cols, i + 1, j, count+1, visited)
20.         if i - 1 >= 0 and visited[(i - 1) * cols + j] and self.getDigitSum(i-1, j) <= threshold:
21.             count = self.get_moving_count(threshold, rows, cols, i - 1, j, count+1, visited)
22.         return count
23.
24.     def getDigitSum(self,num1, num2):
25.         sum = 0
26.         while num1 > 0:
27.             sum += num1%10
28.             num1 /= 10
29.         while num2 > 0:
30.             sum += num2%10
31.             num2 /= 10
32.         return sum
```