1.二维数组中的查找

```
'''
从右上角开始查找
如果当前数 > target，那就左移一位
如果当前数 < target，那就下移一位
'''
class Solution:
    def Find(self, target,array):
        if array == []:
            return False
        num_row = len(array)
        num_col = len(array[0])

        i = 0
        j = num_col -1

        while i<=num_row-1 and j>=0:
            if array[i][j]>target:
                j -= 1
            elif array[i][j]<target:
                i += 1
            else:
                return True
        return False
```

2.替换空格

```
'''
再开一个list，遇到空格就append['%20']，否则append本身
'''
class Solution:
    def replaceSpace(self,s):
        return ''.join(c if c!=' ' else '%20' for c in s)

    #也可以这样写replace
        #return s.replace(' ','%20')
```

3.从尾到头打印链表

```
'''
使用栈，用列表模拟
就是使用pop从尾到头打印
'''
class ListNode:
    def __init__(self,x):
        self.val = x
        self.next = None

class Solution:
    def printListFromTailToHead(self, listNode):
        stack = []
        while listNode:
            stack.append(listNode.val)
            listNode = listNode.next
        while stack:
            print(stack.pop())
#方法二：使用递归
    '''
    这个递归真的很有意思
    '''
    def printListFromTail2Head(self, listNode):
        if listNode:
            printListFromTail2Head(listNode.next)
            print(listNode.val)
```

4.重建二叉树

```
'''
输入前序和中序，重建出二叉树并返回

条件：前序遍历的第一个值一定是根节点，对那个中序遍历中间的阶段，再中序的此节点的左侧是左子树，右侧是右子树
使用递归：前序的[0]是root，对应中序的[i];前序的[1:i+1]和中序的[:i]作为对应的左子树继续上一个过程；前序的[i+1:]和中序的[i+1:]对应右子树继续
'''
class Solution:
    def reConstructBinaryTree(self,pre,tin):
        if not pre or not tin:
            return None
        root = TreeNode(pre[0])
        if set(pre) != set(tin):
            return None
        i = tin.index(pre[0])
        root.left = self.reConstructBinaryTree(pre[1:i+1], tin[:i])
        root.right = self.reConstructBinaryTree(pre[i+1:], tin[i+1:])
        return  root
```

5.用两个栈实现队列

```
'''
一个stack入队，一个出队
出栈为空则从入栈导入道出栈中
push：直接push金stack1，
pop：需要判断stack1和stack2的情况，如果stack2不是空，则直接从stack2中pop，如果stack2为空，把stack1中的值push进stack2中，再popstack2，达到前后反序的目的
'''
class Solution:
    def __init__(self):
        self.stack1 = []
        self.stack2 = []

    def push(self,node):
        self.stack1.append(node)
    def pop(self):
        if len(self.stack1)==0 and len(self.stack2)==0:
            return None
        elif len(self.stack2)==0:
            while len(self.stack1) > 0:
                self.stack2.append(self.stack1.pop())
        return self.stack2.pop()     #所有前面的只是在stack2中反序，return了一个反序后的pop
```

6.旋转数组的最小数字

```
'''
非减排序的数组的一个旋转，输出旋转数组的最小元素

二分查找，首元素肯定>=尾元素，找一个中间点，如果它比大的大，说明最小数字再中间点的后面，比小的小，说明最小数字再中间点的前面

'''
class Solution:
    def minNumberInRotateArray(self,nums):
        l, r = 0, len(nums)-1
        if nums[l] < nums[r]:
            return nums[l]
        while l <= r:
            mid = (l+r)//2
            if nums[mid] > nums[l]:
                l = mid
            elif nums[mid] < nums[r]:
                r = mid
            else:
                return nums[r]  #因为右边一直是更小的一方，所以循环完毕，return mid[r]
```

7.费波切纳数列

```
'''
从0开始，第0项为0
输入整数n，输出第n项
'''
class Solution:
    def Fib(self,n):
        a = 0
        b = 1
        for _ in range(n):
            a, b = b, a+b
        return a
```

8.跳台阶

```
'''
每次1级或者2级
求跳上n级总共有集中跳法

f(1)=1,f(2)=2,f(n)=f(n-1)+f(n-2)
因为n级可以先减去1就是f(n-1),还可以先减去2
'''
class Solution:
    def jumpFloor(self,n):
        a, b = 1, 2
        res = 0
        if n <= 0:
            res = 0
        elif n == 1:
            res = a
        elif n == 2:
            res = b
        else:
            for i in range(n-2):
                res = a+b
                a = b
                b = res
                #这里跟fib一样，完全可以写成 a， b = b， a+b
        return res
```

9.变态跳台阶

```
'''
可以跳1，2，3.。。n阶
总共有几种不同跳法？
这种情况下加一级，就会增加一倍
思考下f(n) = 2*f(n-1)
'''
class Solution:
    def jumpFloor2(self,number):
        ans = 1
        if number >= 2:
            for i in range(number-1):
                ans = ans * 2
        return ans
```

10.矩形覆盖

```
'''
2*1的矩形，使用n个2*1的矩形无重叠覆盖2*n的矩形
总共有几种方法？
可以先除去一个竖着的2*1，f(n-1),或者除去2个横着的1*2，f(n-2)
f(n) = f(n-1)+f(n-2)
'''
class Solution:
    def rectCover(self,number):
        a ,b = 1, 1
        if number <3:
            ans = number
        else:
            for i in range(number-1):
                ans = a+b
                a=b
                b=ans
        return  ans
```

11.二进制中1的个数

```
'''
思路：  n 和 n-1 按位与，最右边的1会变成0 经过几次运算变成0，就是有几个1
'''
class Solution:
    def NemberOf1(self, n):
        count = 0
        for _ in range(32):
            count += (n&1==1)
            n >>= 1
        return count

    #或者
    def Nember_of_1(self, num):
        count = 0
        if num < 0:
            n = n&0xffffffff
            while n!=0:
                count += 1
                n = (n-1)&n
        return count
```

12.数值的整数次方

```
'''
给定double类型的浮点数base和int类型的整数exponent。求出base的exponent次方

利用指数右移一位，实现的是除2操作
再&1判断是否是奇数，是奇数就再×base
'''
class Solution:
    def Power(self, base, exponent):
        def PowerUnsign(base, exponent):
            if exponent==0:
                return 1
            if exponent==1:
                return  base
            ans = PowerUnsign(base, exponent>>1)
            ans *= ans
            if exponent & 1 == 1:
                ans *= base
            return ans

        if exponent < 0:
            return 1.0/PowerUnsign(base, abs(exponent))
        else:
            return PowerUnsign(base, abs(exponent))
```

13.调整数组顺序，奇数位于偶数前

```
'''

'''
class Solution:
    def reOrderArray(self, array):
        odd , even = [], []
        for i in array:
            odd.append(i) if i%2==1 else even.append(i)
        return odd+even


    #用lambda表达式也可以
    def reOrder(self, array):
        return sorted(array, key = lambda c:c%2, reverse=True)
```

14.链表的倒数第K个节点

```
'''
双指针，fast先走k步，然后再一起走
当fast走到尾节点时，slow在倒数第k个基点
'''
class Solution:
    def FindKthToTail(self, head, k):
        fast = slow = head
        for _ in range(k):
            if not fast:
                return None
            fast = fast.next
            while fast:
                slow, fast = slow.next, fast.next
            return slow
```

15.反转链表

```
'''
需要考虑空链表和只有一个节点的链表
'''
class Solution:
    def ReverseList(self, pHead):
        if not pHead or not pHead.next:
            return pHead
        then = pHead.next
        pHead.next = None
        last = then.next
        while then:
            then.next = pHead
            pHead = then
            then = last
            if then :
                last = then.next
        return  pHead

    #或者，只是简单移动指针
    def reverse(self, pHead):
        prev = None
        while pHead:
            pHead.next, prev, pHead = prev, pHead, pHead.next
        return prev
```

16.合并两个排序的链表

```
'''
两个单调递增的链表  输出两个链表合成后的链表  合成后单调不减
两个指针指向两个链表的头节点，取小的放进合并后的，剩余部分再比较
'''

#使用递归
class Solution:
    def Merge(self, l1,l2):
        if not l1 or not l2:
            return l1 or l2     #l1或者l2有None则跳出递归
        if l1.val < l2.val:
            l1.next = self.Merge(l1.next, l2)
            return l1
        else:
            l2.next = self.Merge(l1, l2.next)
            return l2
#使用迭代
class Solution:
    def Merge(self, pHead1, pHead2):
        l = head = ListNode(0)  #新建一个虚拟节点
        while pHead1 and pHead2:
            if pHead1.val <= pHead2.val:
                l.next, pHead1 = pHead1, pHead1.next
            else:
                l.next, pHead2 = pHead2, pHead2.next
            l = l.next
        l.next = pHead1 or pHead2       #跳出while之后，多余的部分直接塞到后面尾部
        return head.next
```

17.树的子结构

```
'''
判断B是不是A的子结构（空树不是任何树的子结构）
思路：在A中查找B根节点一致的值，然后判断A中以该节点为根的子树，是不是和B有相同的结构
递归
'''
class Solution:
    def HasSubTree(self, pRoot1, pRoot2):
        result = False
        if pRoot1!=None and pRoot2!=None:
            if pRoot1.val == pRoot2.val:
                result = self.DoesTree1haveTree2(pRoot1, pRoot2)
            if not result:
                result = self.HasSubTree(pRoot1.left, pRoot2)
            if not result:
                result = self.HasSubTree(pRoot1.right, pRoot2)
        return  result

    def DoesTree1haveTree2(self, pRoot1, pRoot2):
        if pRoot2==None:
            return True
        if pRoot1==None:
            return False
        if pRoot1.val != pRoot2.val:
            return False
        return self.DoesTree1haveTree2(pRoot1.left, pRoot2.left) and
self.DoesTree1haveTree2(pRoot1.right, pRoot2.right)

#写的太模糊了，递归看不懂
#重写一个
class Solution:
    def HasSubTree(self, s, t):
        def is_same(s, t):
            if s and t:     #根节点都不为空
                equal = (s.val==t.val)  #bool
                if not t.left and not t.right:  # t左右子树都为空，就只是一个点，判断equal即可，否则，判断root相等且左右相等
                    return equal
                else:
                    return (equal and is_same(s.left, t.left) and is_same(s.right, t.right))
            else:
                return s is t   #bool

        stack = s and [s]
        while stack:
            node = stack.pop()
            if node:
                res = is_same(node, t)  #判断s的节点子树与t是否相等
                if res:
                    return True
                stack.append(node.right)
                stack.append(node.left)
        return False

#再有，可以取巧，将Tree换成str
class Solution:
    def HasSub(self, pRoot1, pRoot2):
        def convert(p):
            if p:
                return str(p.val)+convert(p.left)+convert(p.right)
            else:
                return ''
        return convert(pRoot2) in convert(pRoot1) if pRoot2 else False
#这个可以有更简单的递归
class Solution:
    def HasSub(self, pRoot1, pRoot2):
        if pRoot1 and pRoot2:
            if pRoot1.val == pRoot2.val:
                return self.HasSub(pRoot1.left, pRoot2.left) and self.HasSub(pRoot1.right, pRoot2.right)
            else:
                return self.HasSub(pRoot1.left, pRoot2) or self.HasSub(pRoot1.right, pRoot2)
        if not pRoot1 or not pRoot2:
            return False
        return  True
```

18.二叉树的镜像

```
'''
把给定的二叉树转换成它的镜像
'''

#使用递归
class Solution:
    def Mirror(self, root):
        if root:
            root.left, root.right = root.right, root.left
            self.Mirror(root.left)
            self.Mirror(root.right)
#使用迭代
```

```
class Solution:
    def Mirror(self, root):
        stack = root and [root]
        while stack:
            node = stack.pop()
            if node:
                node.left, node.right = node.right, node.left
                stack += node.right, node.left
```

19.顺时针打印矩阵

```
'''
从外往里 螺旋形 顺时针打印矩阵
'''
class Solution:
    def printMatrix(self,matrix):
        return (
                matrix and list(matrix.pop(0)) +
                self.printMatrix(list(zip(*matrix))[::-1])
        )
#如果逆时针
def anti_clock_wise(self, matrix):
    if not matrix:
        return []
    clock_wise = list(zip(*(matrix[::-1])))
    a = list(clock_wise.pop(0))[::-1]
    b = self.anti_clock_wise(clock_wise)
    return a+b
```

20.包含min函数的栈

```
'''
时间复杂度 O(1)
实现栈的min
思路：建立辅助栈，每次最小值压入辅助栈，辅助栈顶一直就是最小元素，
当数据栈中，最小值被弹出时，同样弹出辅助栈中的栈顶元素
'''
class Solution:
    def __init__(self):
        self.stack = []
        self.minStack = []
    def push(self, node):
        self.stack.append(node)
        if self.minStack==[] or node < self.min():
            self.minStack.append(node)
        else:
            temp = self.min()
            self.minStack.append(temp)
    def pop(self):
        if self.stack==[] or self.minStack==[]:
            return None
        self.minStack.pop()
        self.stack.pop()
    def top(self):
        return self.stack[-1]
    def min(self):
        return self.minStack[-1]
```

21.栈的压入和弹出

```
'''
输入两个整数序列，第一个序列表示栈的压入顺序，判断第二个是否可能是栈的弹出
'''
class Solution:
    def IsPopOrder(self, pushV, popV):
        if pushV == [] or popV==[]:
            return False
        stack = []
        for i in pushV:
            stack.append(i)
            while len(stack) and stack[-1]==popV[0]:
                stack.pop()
                popV.pop(0) #每次都是辅助栈的栈顶和popV的第一个值判断是否相等，相等就弹出
        if len(stack):
            return False
        else:
            return True
```

22.从上往下打印二叉树

```
'''
思路：  引入一个队列，每次打印一个节点的时候，
如果该节点存在子节点，就把该节点的子节点放入队列的末尾，取出队列头部的最早进入队列的节点
'''
class Solution:
    def PrintFromTopToBottom(self, root):
        queue = []
        if not root:
            return []
        result = []
        queue.append(root)
        while len(queue)>0:
            currentRoot = queue.pop(0)
            result.append(currentRoot.val)
            if currentRoot.left:
                queue.append(currentRoot.left)
            if currentRoot.right:
                queue.append(currentRoot.right)
        return result
```

23.二叉搜索树的后续遍历序列

```
'''
输入一个整数数组，判断是不是某个二叉搜索树的后序遍历
思路：  根据后序遍历特点，尾元素一定是root，小于尾元素的值是左子树，大于尾元素的值是右子树
且序列前半部分小于尾元素，后半部分大于尾元素，将序列分为左子树和右子树，递归
'''
class Solution:
    def VerifySquenceOfBST(self, seq):
        if seq == []:
            return False
        length = len(seq)
        root = seq[-1]
        for i in range(length):
            if seq[i] > root:
                break
        for j in range(i, length):
            if seq[j] < root:
                return False

        left = True
        if i>0:
            left = self.VerifySquenceOfBST(seq[:i])
        right = True
        if j<length-1:
            right = self.VerifySquenceOfBST(seq[i:length-1])

        return left and right
```

24.二叉树中和为某一值的路径

```
'''
输入根节点和一个整数，打印出二叉树中节点值的和为输入整数的所有路径
思路： 用前序遍历访问二叉树，当访问道节点时，加入路径中，并累加节点值，直到访问到符合要求的节点或者访问到叶节点，
然后，递归访问该节点的父节点，函数退出时删除当前节点，并减去当前系欸但那的值，相当于出栈入栈过程
'''
#迭代
class Solution:
    def FindPath(self, root, total):
        stack = root and [(root, [root.val], total)]
        ans = []
        while stack:
            n, v, t = stack.pop()
            if not n.left and not n.right and n.val==t: #若左右子树都是空且root.val=total，则[root.val]是一个路径
                ans.append(v)
            if n.right:
                stack.append((n.right, v+[n.right.val], t-n.val))
            if n.left:
                stack.append((n.left, v+[n.left.val], t-n.val))
        return ans
#递归
#先找出所有路径，再过滤
class Solution:
    def FindPath(self, root, sum_val):
        paths = self.all_paths(root)
        return [path for path in paths if sum(path)==sum_val]
    def all_paths(self, root):
        if not root:
            return []
        return [
            [root.val]+path
            for kid in (root.left, root.right) if kid
            for path in self.all_paths(kid)
        ] or [[root.val]]

#递归
class Solution:
    def FindPath(self, root, sum):
        if not root:
            return []
        val, *kids = root.val, root.left, root.right
        if any(kids):
            return [
                [val]+path
                for kid in kids if kid
                for path in self.FindPath(kid,sum-val)
            ]        #这个语句太复杂了，改成更容易理解的下一种
        return [[val]] if val==sum else []

#递归
class Solution:
    def FindPath(self, root, sum):
        if not root: return []
        if root.left or root.right:
            a = self.FindPath(root.left, sum-root.val) +
self.FindPath(root.right, sum-root.val)
            return [[root.val]+i for i in a]
        return [[root.val]] if sum==root.val else []
```

25. 复杂链表的复制

```
'''
输入复杂链表，节点有节点值和两个指针，一个指向下一节点，另一个指向任意一个节点
返回结果为复制后的复杂链表的head
思路：遍历两次，第一次复制到字典中，第二次关联
'''
class Solution:
    def Clone(self, head):
        cp = {None:None}
        m = n = head
        #复制
        while m:
            cp[m] = RandomListNode(m.label)
            m = m.next
        #关联
        while n:
            cp[n].next = cp[n.next]
            cp[n].random = cp[n.random]
            n = n.next
        return cp[head]
```

26. 二叉搜索树和双向链表

```
'''
输入二叉搜索树，将它转换成一个排序的双向链表，要求不能创建任何新节点，只能调整树中节点指针
的指向

思路： 分治，左右子树，递归实现。根节点的左边连接左子树最右边的节点，根节点的右边连接右子树
最左边的节点
'''
class Solution:
    def Convert(self, root):
        def convert_tree(node):
            if not node:
                return None
            if node.left:
                left = convert_tree(node.left)
                while left.right:
                    left = left.right
                left.right = node
                node.left = left
            if node.right:
                right = convert_tree(node.right)
                while right.left:
                    right = right.left
                right.left = node
                node.right = right
            return node

        if not root:
            return root
        root = convert_tree(root)
        while root.left:
            root= root.left
        return root
```

27. 字符串的排列

```
'''
输入一个字符串，按英文字母顺序打印所有可能的排列
递归，不断固定第一个，求之后的可能排列
'''
class Solution:
    def Pernutation(self,ss):
        if not ss:
            return []
        return self.permute(ss)
    def permute(self,ss):
        return sorted(list(set(
            [h+p
             for i,h in enumerate(ss)
             for p in self.permute(ss[:i]+ss[i+1:])]
        ))) or ['']
#使用迭代
def Permutation(ss):
    ans = ['']
    for s in ss:
        ans = [p[:i] + s + p[i:]
               for p in ans for i in range((p+s).index(s)+1)]
    return sorted(ans) if ss else []
#循环可以展开写
class Solution:
    def Permutation(self, ss):
        if not ss:
            return []
        ret = []
        for i in range(ss):
            for j in self.Permutation(ss[:i]+ss[i+1:]):
                ret.append(ss[i]+j)
        return sorted(list(set(ret)))
```

28. 数组中出现此处超过一半的数字

```
'''
数组张有一个数字出现的次数超过数组长度的一半，找出这个数字，不存在就输出0
'''
#把这个 叫做波亦尔摩尔投票算法
#这个答案是错的
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        count = 0
        candidate = None
        for num in numbers:
            if count == 0:
                candidate = num
            count += (1 if num == candidate else -1)
        return candidate
s = Solution()
c = s.MoreThanHalfNum_Solution(['a','a','b','c'])
'''
使用hash，key是数字，value是次数
'''
class Solution:
    def MoreThanHalfNuma_Solution(self, nums):
        hashs = dict()
        length = len(nums)
        for n in nums:
            hashs[n] = hashs[n]+ 1 if hashs.get(n) else 1
            if hashs[n] > length/2:
                return n
        return 0
```

29. 最小的k个数

```
'''
输入n个整数，找出其中最小的k个数
基于划分，使比第k个数小的都在左边，大的都在右边，递归构建快排
'''
class Solution:
    def GetLeastNumbers_Solution(self, tinput, k):
        if not tinput or k>len(tinput):
            return []
        tinput = self.quick_sort(tinput)
        return tinput[:k]
    def quick_sort(self, lst):
        if not lst:
            return []
        pivot = lst[0]
        left = self.quick_sort([x for x in lst[1:] if x<pivot])
        right = self.quick_sort([x for x in lst[1:] if x>=pivot])
        return left+[pivot]+right
s = Solution()
ls = s.GetLeastNumbers_Solution([3,5,2,4,1],2)
lst = s.quick_sort([3,5,2,4,1])
```

30. 连续子数组的最大和

```
'''
给定数组有正负，求出连续子数组的最大和
'''
class Solution:
    def FindGreatestSumOfSubArray(self, nums):
        cp_nums = nums[:]
        for i in range(1, len(nums)): #从index==1开始，判断前一个值是否大于0，大则累加
            if cp_nums[i-1] > 0:
                cp_nums[i] += cp_nums[i-1]
        return max(cp_nums)
lst = [1,-2,3,-2]
s = Solution()
res = s.FindGreatestSumOfSubArray(lst)
```

31. 整数中1出现的次数

```
'''
求出1-n中1出现的次数
'''
class Solution:
    def NumberOf1Between1AndN_Solution(self, n):
        countr, i = 0, 1
        while i < n:
            divider = i*10
            countr += (n // divider) * i + min(max(n % divider -i + 1, 0), i)
            i *= 10
        return countr
s = Solution()
res = s.NumberOf1Between1AndN_Solution(156)
#完全看不懂，去一边去吧
class Solution:
    def NumberOf1Between1AndN_Solution(self, n):
        count = 0
        for i in range(1, n+1):
            while i:
                if i%10==1:
                    count += 1
                i /= 10
        return count
```

32. 把数组排成最小的数

```
'''
输入正整数数组，打印能拼出来的数字中最小的一个

'''
#使用冒泡排序
class Solution:
    def PrintMinNumber(self, numbers):
        if numbers==None or len(numbers)<=0:
            return ''
        strNum = [str(m) for m in numbers]
        for i in range(len(numbers)-1):
            for j in range(i+1, len(numbers)):
                if strNum[i]+strNum[j] > strNum[j]+strNum[i]:
                    strNum[i], strNum[j] = strNum[j], strNum[i]
        return int(''.join(strNum))
```

33. 丑数

```
'''
把只包含质因子2/3/5的数叫做丑数 把1作为第一个丑数
求从小到大顺序的第N个丑数
'''
class Solution:
    def GetUglyNumbers_Solution(self, n):
        if n == 0:
            return 0
        q = [1]
        t2 = t3 = t5 = 0
        for _ in range(n-1):
            a2, a3, a5 = q[t2]*2, q[t3]*3, q[t5]*5
            to_add = min(a2, a3, a5)
            q.append(to_add)
            if a2 == to_add:
                t2 += 1
            if a3 == to_add:
                t3 += 1
            if a5 == to_add:
                t5 += 1
        return q[-1]
```

34. 第一次支持先一次的字符

```
'''
字符串全部由字母组成，找到第一个只出现一次的字符的位置，没有返回-1
遍历两次，第一次用hash存放字符和出现的次数，第二次找到hash等于1的值
'''
class Solution:
    def FirstNotRepeatingChar(self, s):
        if s == None or len(s) <= 0:
            return -1
        alphabet = dict()
        lst = ''.join(s)
        for i in lst:
            if i not in alphabet.keys():
                alphabet[i] = 1
            alphabet[i] += 1
        for i in lst:
            if alphabet[i] == 1:
                return lst.index(i)
```

35. 数组中的逆序对

```
'''
前面的数字大于后面的数字，称为一个逆序对，求总数
'''
class Solution:
    def InversePairs(self, data):
        count = 0
        copy = []
        for _ in data:
            copy.append(_)
        copy.sort()

        for i in range(len(copy)):
            count += data.index(copy[i])
            data.remove(copy[i])
        return count % 1000000007
```

36. 两个链表的第一个公共节点

```
'''
输入两个链表，找出他们的第一个公共节点
'''
class Solution:
    def FindFirstCommonNode(self, pHead1, pHead2):
        p1, p2 = pHead1, pHead2
        while p1 != p2:
            p1 = p1.next if p1 else pHead2
            p2 = p2.next if p2 else pHead1
        return p1
```

37. 数字在排序数组中出现的次数

```
'''
统计一个数字在排序数组中出现的次数
二分查找？
'''
class Solution:
    def GetNumberOfK(self, data, k):
        def search(n):
            lo, hi = 0, len(data)
            while lo < hi:
                mid = (lo + hi) // 2
                if data[mid] >= n:
                    hi = mid
                else:
                    lo = mid + 1
            return lo
        lo = search(k)
        if k in data[lo : lo+1]:
            return search(k+1)-lo
        else:
            return 0
```

38. 二叉树的深度

```
'''
求二叉树的深度
递归，只有一个root，深度为1，存在左子树或右子树，深度为左右子树中深度较深的+1
'''
class Sloution:
    def TreeDepth(self, pRoot):
        if not pRoot:
            return 0
        return max(self.TreeDepth(pRoot.left), self.TreeDepth(pRoot.right)) + 1
```

39. 平衡二叉树

```
'''
平衡二叉树：空树或者左右两个子树高度差<=1，且子树都是平衡二叉树
递归，在遍历节点时记录深度，一边遍历一边判断
'''
class Solution:
    def __init__(self):
        self.flag = True
    def IsBalanced_Solution(self, pRoot):
        self.getDepth(pRoot)
        return self.flag
    def getDepth(self, root):
        if not root:
            return 0
        left = self.getDepth(root.left) + 1
        right = self.getDepth(root.right) + 1
        if abs(left-right) > 1:
            self.flag = False
        return left if left>right else right
```

40. 数组中只出现一次的数字

```
'''
一个整型数组中除了两个数字以外，其他数字都出现了偶数次 找出这两个只出现一次的数字
最简单的hashmap，但是空间复杂度太高
考虑：把这两个元素分到两个组，由于两数不等，所以异或结果不为0，按异或结果二进制中1的所在位，
可以把他们分到两个子数组。子数组的异或结果就是这两个数
'''
class Solution:
    def FindNumsAppearOnce(self, array):
        if array == None:
            return []
        xor = 0
        for i in array:
            xor ^= i
        idxOf1 = self.getFirstIdx(xor)
        num1 = num2 = 0
        for j in range(len(array)):
            if self.IsBit(array[j], idxOf1):
                num1 ^= array[j]
            else:
                num2 ^= array[j]
        return [num1, num2]
    def getFirstIdx(self, num):
        idx = 0
        while num & 1 == 0 and idx <= 32:
            idx += 1
            num = num >> 1
            return idx

    def IsBit(self, num, indexBit):
        num = num >> indexBit
        return num & 1
```

41. 和为s的连续正数序列

```
'''
超过target一半的肯定不行，从1，2开始移动指针
'''
class Solution:
    def FindContinuousSequence(self, tsum):
        end = (tsum + 1)//2
        lo, hi, cur_sum = 1, 2, 3
        ans = []
        while lo < end:
            if cur_sum < tsum:
                hi += 1
                cur_sum += hi
            else:
                if cur_sum == tsum:
                    ans.append(list(range(lo, hi+1)))
                cur_sum -= lo
                lo += 1
        return ans
```

42. 和为s的两个数

```
'''
输入一个递增排序的数组和一个数字s，找出数组中的两个数，使和为s，如果有多对，输出积最小的
双指针起点终点移动
'''
class Solution:
    def FindNumberWithSum(self, array, tsum):
        l, r = 0, len(array)-1
        while l < r:
            if array[l]+array[r] < tsum:
                l += 1
            elif array[l]+array[r] > tsum:
                r -= 1
            else:
                return array[l], array[r]
        return []
```

43. 左旋转字符串

```
'''
就是循环左移
'''
class Solution:
    def LeftRotateString(self, s, n):
        if not s:
            return  ''
        n = n % len(s)
        return s[n:] + s[:n]
```

44. 翻转单词顺序列

```
'''
i am a student --> student a am i

'''
class Solution:
    def ReverseSentence(self, s):
        return ' '.join(reversed(s.split(' ')))

#展开写
class Solution:
    def ReverseSentence(self, s):
        def reverse(s):
            s = s.split(' ')
            for i in range(len(s)//2):
                s[i], s[~i] = s[~i], s[i]
            return ' '.join(s)
        s = reverse(s)
        return s
```

45. 扑克牌顺子

```
'''
2个大王2个小王可以当作任意牌
'''
class Solution:
    def IsContinous(self, numbers):
        if not numbers:
            return False
        joker_count = numbers.count(0)
        left_cards = sorted(numbers)[joker_count:] #剩下的牌中非joker的list
        need_joker = 0
        for i in range(len(left_cards)-1):
            if left_cards[i+1] == left_cards[i]:
                return False
            need_joker += (left_cards[i+1] - left_cards[i] - 1)
        return need_joker <= joker_count
```

46. 孩子们的游戏(圆圈中剩下的数)

```
'''
约瑟夫环
fn = [(fn-1)+m] % n 其中，fn是场上有n个人时在场的人的编号
f1 = 0
```

```
'''
class Solution:
    def LastRemaining_Solution(self, n ,m):
        if n <= 0 or m <= 0:
            return -1
        last_num = 0
        for i in range(2, n+1):
            last_num = (last_num + m)%2
        return last_num
#或者list旋转数组也可以?
class Solution:
    def LastRemaining_Solution(self, n, m):
        if n<=0 or m<=0:
            return -1
        seats = range(n)
        while seats:
            rot = (m-1) % len(seats)
            seats, last = seats[rot+1:] + seats[:rot], seats[rot]
        return last
```

47. 求1+2+。。。+n

```
'''
要求不能用乘除法，for while if else switch case等判断和条件语句
'''
#写个递归，不过终止条件也类似于判断语句了
class Solution:
    def Sum_solution(self, n):
        return n and (n + self.Sum_solution(n-1))
```

48. 不用加减乘除做加法

```
'''
两数异或：相当于每一位相加不考虑进位
两数相与并左移一位：相当于求进位
python中负数会有问题
'''
class Solution:
    def Add(self, num1, num2):
        while num2 != 0:
            temp = num1 ^ num2
            num2 = (num1 & num2) << 1
            num1 = temp & 0xFFFFFFFF
        return num1 if num1 >> 31 == 0 else num1-4294967296
```

49. 字符串转化成整数

```
'''
str --> int 当str不符合数字要求时，返回0
数值为0或者字符串不是合法数值返回0
'''
class Solution:
    def StrToInt(self, s):
        flag = False
        if not s or len(s) < 1:
            return 0
        num = []
        numdict =
{'0':0,'1':1,'2':2,'3':3,'4':4,'5':5,'6':6,'7':7,'8:'8,'9':9}
        for i in s:
            if i in numdict.keys():
                num.append(numdict[i])
            elif i=='+' or i=='-':
                continue
            else:
                return 0
        ans = 0
        if len(num)==1 and num[0]==0:
            flag = True
            return 0
        for i in num:
            ans = ans*10 + i
        if s[0] == '-':
            ans=0-ans
            return  ans
```

50. 数组中重复的数字

```
'''
在长度为n的数组中所有数字都在0-n-1的范围内，有重复。
找出数组中任意一个重复的数字
思路：先排序，再遍历数组查找重复的数字 O(nlgn)
或者建立哈希表，在O(n)查找到
'''
class Solution:
    def duplicate(self, numbers, duplication):
        for i, num in enumerate(numbers):
            while i != num:
                if numbers[num] == num:
                    duplication[0] = numbers[i]
                    return  True
                else:
                    numbers[i], numbers[num] = numbers[num], numbers[i]
                    num = numbers[i]
        return False
```

51. 构建成绩数组

```
'''
给定数组A[0,1,...,n-1] 构建数组B[0,1,2...,n-1] 其中B[i]=A[0]*....A[i-
1]*A[i+1]*...*A[n-1]
'''
class Solution:
    def multiply(self, A):
        C =[1]
        for i in range(len(A)-1):
            C.append(C[-1] * A[i])
        D =[1]
        for j in range(len(A)-1, 0, -1):
            D.append(D[-1] * A[j])
        D.reverse()
        return [C[i] * D[i] for i in range(len(A))]
```

52. 正则表达式匹配

```python
class Solution:
    def match(self, s, patten):
        if not patten: return not s
        f_match = bool(s) and patten[0] in {s[0], '.'}
        if len(patten) > 1 and patten[1] == '*'
            return (self.match(s, patten[2:]) or
                    (f_match and self.match(s[1:], patten))
                    )
        else:
            return f_match and self.match(s[1:], patten[1:])
```

53. 表示数值的字符串

```python
'''
判断字符串是否表示数值
注意判断E和e后面跟一个整数，正负均可，不能没有，也不能是小数
'''
class Solution:
    def isNumberic(self, s):
        if s == None or len(s) <= 0:
            return  False
        aList = [w.lower() for w in s]
        if 'e' in aList:
            indexE = aList.index('e')
            front = aList[:indexE]
            behind = aList.[indexE+1:]
            if '.' in behind or len(behind) == 0:
                return False
            isFront = self.scanDigit(front)
            isBehind = self.scanDigit(behind)
            return isBehind and isFront
        else:
            isNum = self.scanDigit(aList)
            return isNum

        def scanDigit(self, alist):
            dotNum = 0
            allowVal = ['0', '1', '2','3','4','5','6','7','8','9','+','-
','e']
            for i in range(len(alist)):
                if alist[i] not in allowVal:
                    return False
                if alist[i] == '.':
                    dotNum += 1
                if alist[i] in '+-' and i!=0:
                    return False
            if dotNum > 1:
                return False
            return True

#使用try也可以弄个捷径
def isNumeric(self, s):
    try:
        float(s)
        if s[0:2] != '+-' and s[0:2] != '-+':
            return False
        else:
            return True
    except:
        return False
```

54. 字符流中第一个不重复的字符

```
'''
找出字符流中第一个只出现一次的字符
不存在只出现一次的字符时，返回#

引入存储空间，一个dict存储当前字符和出现的次数，一个list存储当前出现的字符，
每次比较list的第一个字符再dict中对应的次数
'''
class Solution:
    def __init__(self):
        self.adict = {}
        self.alist = []
    def FirstAppearingOnce(self):
        while len(self.alist) > 0 and self.adict[self.alist[0]] == 2:
            self.alist.pop(0)
        if len(self.alist) == 0:
            return '#'
        else:
            return self.alist[0]
    def Insert(self, char):
        if char not in self.adict.keys():
            self.adict[char] = 1
            self.alist.append(char)
        elif self.adict[char]:
            self.adict[char] = 2
```

55. 链表中环的入口节点

```
'''
给一个链表，如果包含环，找出链表环的入口节点，否则，输出null

思路：双指针；当fast走到末端，说明没有环；fast==slow,跳出循环
然后head和slow一起走，假设头部走到环a步，环b长，
能够推出 ，相遇时slow走了nb，fast走了2nb，
所以从头再走，head走a，slow走a+nb，相遇，所以都指向入口
'''
class Solution:
    def EntryNodeOfLoop(self, head):
        fast = slow = head
        while fast and fast.next:
            slow, fast = slow.next, fast.next.next
            if slow is fast:
                break
        else:
            return None
        while head is not slow:
            head, slow = head.next, slow.next
        return head
```

56. 删除链表中重复的节点

```
'''
删除重复节点，返回链表头指针
思路：先找重复节点，头节点也可能重复，所以需要新建虚拟节点
遍历链表，同时需要把前一个节点与之后不重复的系欸但项链(last负责把前一节点和当前不重复节点相连)
'''
class Solution:
    def deletDuplication(self, pHead):
        if pHead is None or pHead.next is None:
            return pHead
        first = ListNode(-1)    #新建虚拟节点
        first.next = pHead
        last = first
        while pHead and pHead.next:
            if pHead.val == pHead.next.val:
                val = pHead.val
                while pHead and val == pHead.val:
                    pHead = pHead.next  #删除节点
                last.next = pHead
            else:
                last = pHead
                pHead =pHead.next
        return first.next
```

57. 二叉树的下一个节点

```
'''
找出中序遍历的下一个节点并返回
'''
class Solution:
    def GetNext(self, pNode):
        if pNode == None:
            return None
        #当前给定节点是root时，没有下一节点，先假定pNext=None
        pNext = None
        #如果输入节点有右子树，下一系欸但那是右子树的最左节点
        if pNode.right:
            pNode = pNode.right
            while pNode.left:
                pNode = pNode.left
            pNext = pNode
        else:
            #如果有父节点，且当前节点是父节点的左子节点，下一节点就是父节点
            if pNode.next and pNode.next.left == pNode:
                pNext = pNode.next
            #如果有father且是father的右，则向上遍历
            #当遍历道以当前节点为father的左子节点时，输入系欸但的下一系欸但那时当前节点的父节点
            elif pNode.next and pNode.next.right == pNode:
                pNode = pNode.next
                while pNode.next and pNode.next.right == pNode:
                    pNode = pNode.next
                    if pNode.next:
                        pNext = pNode.next
            return pNext
```

```
'''
写个窍门，把root开始的中序遍历保存，然后直接找pNode的下一个
'''
class Solution:
    def GetNext(self, pNode):
        dummy = pNode
        while dummy.next:
            dummy = dummy.next
        self.result = []
        self.midTraversal(dummy)
        return self.result[self.result.index(pNode)+1] if
self.result.index(pNode) != len(self.result)-1 else None
    def midTraversal(self, root):
        if not root:
            return
        self.midTraversal(root.left)
        self.result.append(root)
        self.midTraversal(root.right)
```

58. 对称的二叉树

```
'''
判断二叉树是不是对称的
'''
class Solution:
    def isSymmetrical(self, root):
        def symmetric(p1, p2):
            if p1 and p2:
                return(p1.val == p2.val and
                       symmetric(p1.left, p2.right) and
                       symmetric(p1.right, p2.left))
            else:
                return p1 is p2
        if not root:
            return True
        return symmetric(root.left, root.right)
```

59. 之字形打印二叉树

```
'''
第一行从左到右，第二行从右到左...
'''
class Solution:
    def Print(self, root):
        ans, level, order = [], root and [root], 1
        while level:
            ans.append([n.val for n in level][::order])
            order *= -1
            level = [kid for n in level for kid in (n.left, n.right) if kid]
        return ans
```

60. 二叉树打印成多行

```
class Solution:
    def Print(self, root):
        ans , level = [], root and []
        while level:
            ans.append([n.val for n in level])
            level = [kid for n in level for kid in (n.left,n.right) if kid]
        return ans
```

61. 序列化二叉树

```
'''
实现二叉树的序列化和反序列化
序列化：把二叉树按某种遍历方式某种格式保存为字符串
通过#表示空节点，！表示节点结束（value!）
反序列化：根据str，重建二叉树
'''
class Solution:
    def __init__(self):
        self.flag = -1
    def Serialize(self, root):
        if not root:
            return '#'
        return
str(root.val)+','+self.Serialize(root.left)+self.Serialize(root.right)

    def Deserialize(self, s):
        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
```

62. 二叉搜索树的第k个节点

```
'''
返回第k小的节点,二叉搜索树的中序就是递增排序好的
'''
class Solution:
    def KthNode(self, pRoot, k):
        if not pRoot or not k:
            return None
        res = []
        def traverse(node):
            if len(res) >= k or not node:
                return None
            traverse(node.left)
            res.append(node)
            traverse(node.right)
        traverse(pRoot)
        if len(res) < k:
            return None
        return res[k-1]
```

63. 数据流中的中位数

```
'''

'''
class Solution:
    def __init__(self):
        self.arr = []
    def Insert(self,num):
        self.arr.append(num)
        self.arr.sort()
    def GetMedian(self, num):
        if len(self.arr)%2 == 1:
            return self.arr[len(self.arr)/2]
        elif len(self.arr)%2 == 0:
            return (self.arr[len(self.arr)/2] + self.arr[len(self.arr)/2-1])
/ 2.0
```

64. 滑动窗口的最大值

```
'''
输入数列和窗口大小，输出滑动窗口的最大值
'''
class Solution:
    def maxInWindows(self, nums, size):
        return [max(nums[i: i+size])
                for i in range(len(nums)-size+1) if size != 0]
```

65. 矩阵中的路径

```
'''
从矩阵中能否找到包含某字符串的所有字符的路径，路径可以从矩阵中任意各自开启
每一步可以选择四个方向，但是不能重复选择各自
'''
class Solution:
    def hasPath(self, matrix, rows, cols, path):
        for i in range(rows):
            for j in range(cols):
                if matrix[i*cols+j] == path[0] and 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'  #使用的格子置零，防止重复使用
        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
```

66. 机器人的运动范围

```
'''
m行n列方格，从(0,0)开始移动，每次移动一格，四个方向，但是不能进入行列坐标个位数字之和大于k的格子
返回能达到的格子的数量
'''
class Solution:
    def movingCount(self, threshold, rows, cols):
        visited = [[False]*cols for _ in range(rows)]
        def get_sum(x, y):
            return sum(map(int, str(x)+str(y)))

        def movingCore(threshold, rows, cols, i, j):
            if get_sum(i, j) <= threshold:
                visited[i][j] = True
                for x, y in ((i-1, j), (i+1,j), (i,j-1), (i,j+1)):
                    if 0<=x<rows and 0<=y<cols and not visited[x][y]:
                        movingCore(threshold, rows, cols, x, y)
        movingCore(threshold, rows, cols, 0, 0)
        return sum(sum(visited, []))
```