

剑指Offer

- 【数组】 1. 二维数组中的查找
- 【字符串】 2. 替换空格 — string本身可以历遍
- 【链表】 3. 从尾到头打印链表 — list的翻转
- 【数】 4. 重建二叉树
- 【栈和队列】 5. 用两个栈 【先进后出】 实现队列 【先进先出】
- 【查找和排序】 6. 旋转数组的最小数字
- 【递归和循环】 7. 斐波那契数列
- 【递归和循环】 8. 跳台阶
- 【递归和循环】 9. 变态跳台阶
- 【递归和循环】 10. 矩形覆盖
- 【位运算】 11. 二进制中1的个数
- 【代码的完整性】 12. 数值的整数次方
- 【代码的完整性】 13. 调整数组顺序使奇数位于偶数前面
- 【代码的鲁棒性】 14. 链表中倒数第k个结点
- 【代码的鲁棒性】 15. 反转链表
- 【代码的鲁棒性】 16. 合并两个排序的链表 — 重复比较的话应该往递归想
- * 【代码的鲁棒性】 17. 树的子结构
- 【面试思路】 18. 二叉树的镜像
- 【画图让抽象形象化】 19. 顺时针打印矩阵
- 【举例让抽象具体化】 20. 包含min函数的栈
- * 【举例让抽象具体化】 21. 栈的压入、弹出序列
- 【队列实现BFS】 22. 从上往下打印二叉树
- * 【举例让抽象具体化】 23. 二叉搜索树的后序遍历序列
- * 【举例让抽象具体化】 24. 二叉树中和为某一值的路径
- 【分解让复杂问题简单】 25. 复杂链表的复制
- 【分解让复杂问题简单】 27. 字符串的排列
- 【时间效率】 28. 数组中出现次数超过一半的数字
- 【时间效率】 29. 最小的K个数
- 【时间效率】 30. 连续子数组的最大和
- 【时间效率】 31. 整数中1出现的次数 (从1到n整数中1出现的次数)
- 【时间效率】 32. 把数组排成最小的数
- 【时间空间效率的平衡】 33. 丑数
- 【时间空间效率的平衡】 34. 第一个只出现一次的字符位置
- 35. 数组中的逆序对
- 36. 两个链表的第一个公共结点
- 37. 数字在排序数组中出现的次数
- 38. 二叉树的深度
- 39. 平衡二叉树
- 40. 数组中只出现一次的数字
- 41. 和为S的连续正数序列
- 42. 和为S的两个数字
- 43. 左旋转字符串
- 44. 翻转单词顺序列
- 45. 扑克牌顺子
- 46. 圆圈中最后剩下的数
- 【发散思维能力】 47. 求 $1+2+3+\dots+n$
- 48. 不用加减乘除做加法
- 49. 把字符串转换成整数

- [50. 数组中重复的数字](#)
- [51. 构建乘积数组](#)
- [52. 正则表达式匹配](#)
- [53. 表示数值的字符串](#)
- [54. 字符流中第一个不重复的字符](#)
- [55. 链表中环的入口结点](#)
- [56. 删除链表中重复的结点](#)
- [57. 二叉树的下一个结点](#)
- [58. 对称的二叉树](#)
- [59. 按之字形顺序打印二叉树](#)
- [60. 把二叉树打印成多行](#)
- [61. 序列化二叉树](#)
- [62. 二叉搜索树的第k个结点](#)
- [63. 数据流中的中位数](#)
- [64. 滑动窗口的最大值](#)
- [65. 矩阵中的路径](#)
- [66. 机器人的运动范围](#)
- [67. 剪绳子](#)

【数组】1. 二维数组中的查找

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。

题解



记得考虑空数组！

- 左下/右上元素移动法：复杂度为 $O(N)$

```
# 左下/右上元素移动法
class Solution:
    # array 二维列表
    def Find(self, target, array):
        # 考虑空数组
        if len(array) == 0 or len(array[0]) == 0:
            return False
        # 右上走策略，从左下角开始
        row = len(array)-1
        col = 0
        while target != array[row][col]:
            if target < array[row][col]:
                if row == 0:
                    return False
                row -= 1
            else:
                if col == len(array[0])-1:
                    return False
                col += 1
        return True
```

- N行二分查找：复杂度为 $O(N \log N)$

```
# N行二分查找
class Solution:
```

```

# array 二维列表
def Find(self, target, array):
    # 考虑空数组
    if len(array) == 0 or len(array[0]) == 0:
        return False
    row = len(array)-1
    col = 0
    for row in range(len(array)):
        if array[row][0] <= target <= array[row][len(array[0])-1]:
            l = 0
            r = len(array[0])-1
            while l<=r:
                mid = l + (r-1)//2 #必须是// 因为要整除
                if target == array[row][mid]:
                    return True
                elif target < array[row][mid]:
                    r = mid-1
                else:
                    l = mid+1
    return False

```

【字符串】2. 替换空格 — string本身可以历遍

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。

题解



string本身可以历遍

```

class Solution:
    # s 源字符串
    def replaceSpace(self, s):
        # write code here
        a=''
        for l in s:
            if l == ' ':
                a += '%20'
            else:
                a += l
        return a

```

【链表】3. 从尾到头打印链表 — list的翻转

输入一个链表，按链表从尾到头的顺序返回一个ArrayList。



1. 注意空值啊！！



**2. list翻转就是list[n::-1] — 从第n个位置坐标开始 截取顺序相反，n为空则是整个反转
:endBefore :skip]**

x[startAt

题解

```

# 遍历，新创一个数组保存
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):
        # 注意空值
        if listNode is None:
            return []
        res=[]
        while listNode is not None:
            res.append(listNode.val)
            listNode=listNode.next
        return res[::-1]

    # 递归
    def printListFromTailToHead(self, listNode):
        # write code here
        if listNode is None:
            return []
        return self.printListFromTailToHead(listNode.next) + [listNode.val]

func(next1) + [next0.val]
func(next2) + [next1.val] + [next0.val]
nextn == None --> [] + [nextn-1.val] + ... + [next0.val]

# 利用栈先入后出的特性完成（待补充）

```

【数】4. 重建二叉树

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。

题解

根据中序遍历和前序遍历可以确定二叉树，具体过程为：

1. 根据前序序列第一个结点确定根结点
2. 根据根结点在中序序列中的位置分割出左右两个子序列
3. 对左子树和右子树分别递归使用同样的方法继续分解

例如：前序序列{1,2,4,7,3,5,6,8} = pre 中序序列{4,7,2,1,5,3,8,6} = in

1. 根据当前前序序列的第一个结点确定根结点，为 1
2. 找到 1 在中序遍历序列中的位置，index 为 3，为 in[3]
3. 切割左右子树，则 in[3] 前面的为左子树，in[3] 后面的为右子树
4. 则切割后的**左子树前序序列**为：{2,4,7}，切割后的**左子树中序序列**为：{4,7,2}；
 - 切割后的**右子树前序序列**为：{3,5,6,8}，切割后的**右子树中序序列**为：{5,3,8,6}
5. 对子树分别使用同样的方法分解

```

class Solution:
    def reConstructBinaryTree(self, pre, tin):
        if not pre or not tin:
            return None
        # 前序序列第一个为root
        root = TreeNode(pre.pop(0)) #要不断pop根部达到trimming的效果
        # 找中序序列中root位置
        index = tin.index(root.val)
        # 先递归找左子树
        root.left = self.reConstructBinaryTree(pre, tin[:index])
        # 再递归找右子树

```

```

root.right = self.reConstructBinaryTree(pre, tin[index + 1:])
return root

```

定根：root = **TreeNode**(pre.pop(0))
取值：**root.val**

【栈和队列】5. 用两个栈【先进后出】实现队列【先进先出】

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。

题解

创两个栈，一个压入，一个弹出

压入时正常操作

弹出时先把stackpush的数据再压入stackpop中，顺序就能反过来。不过要先判断stackpop中是否有元素，不为空时直接弹出stackpop中数据

```

class Solution:
    def __init__(self):
        self.stackpush = []
        self.stackpop = []
    def push(self, node):
        self.stackpush.append(node)
    def pop(self):
        if len(self.stackpop) == 0:
            # FIFO 先进先出
            while len(self.stackpush) > 0:
                self.stackpop.append(self.stackpush.pop()) # FIFO stackpush的第一个应该被pop out
            return self.stackpop.pop()
        else:
            return self.stackpop.pop()

```

【查找和排序】6. 旋转数组的最小数字

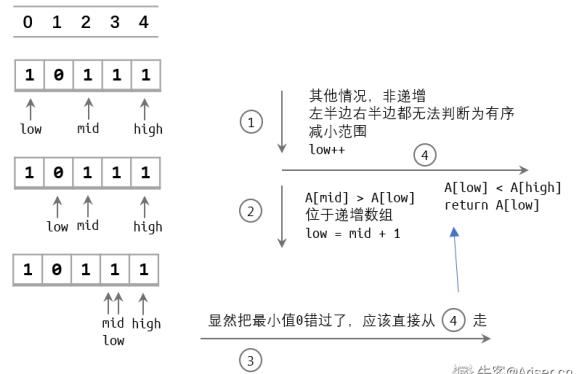
把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。

输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。

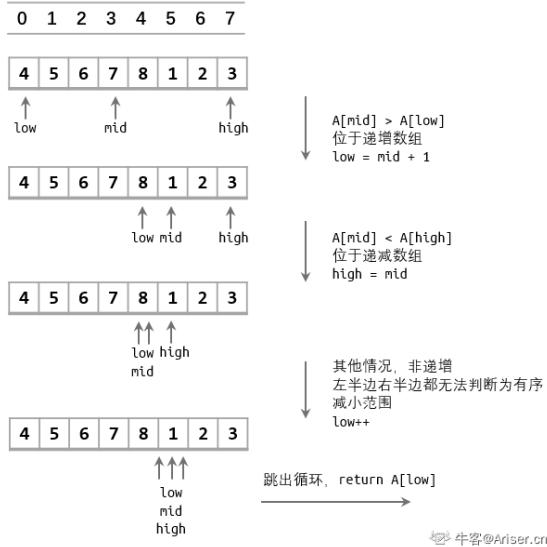
例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。

NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

题解



特殊情况



牛客@Ariser.cn

二分法思想，从中间开始判断左边/右边哪个是单调递增数列

```
# 二分查找
class Solution:
    def minNumberInRotateArray(self, rotateArray):
        # write code here
        if len(rotateArray) == 0:
            return 0
        if len(rotateArray) == 1:
            return rotateArray[0]
        l = 0 #array的头部
        r = len(rotateArray)-1 #array尾巴
        while r >= l:
            mid = l+(r-1)//2
            if rotateArray[mid] < rotateArray[mid-1]: #mid正好小于前一个值，正好是节点
                return rotateArray[mid]
            if rotateArray[mid] < rotateArray[0]: #mid小于头头，证明已经过了节点，r应该变为mid-1
                r = mid-1
            else: #mid大于头头，证明还在单调递增当中，l应该改为mid+1
                l = mid+1
# 单边查找
class Solution:
    def minNumberInRotateArray(self, rotateArray):
        # write code here
        pre = -7e20 # 先设置一个很小的数
        for num in rotateArray:
            if num < pre :
                return num
            pre = num
        if len(rotateArray) == 0:
            return 0
        return rotateArray[0]
```

【递归和循环】7. 斐波那契数列

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

题解

迭代 (Iteration) 会重复计算

```

# 时间复杂度:O(n)    空间复杂度:O(1)
class Solution:
    def Fibonacci(self, n):
        # write code here
        a = 0
        b = 1
        if n <= 1:
            return n
        for i in range(n):
            a, b = b, b+a
        return a

```

【递归和循环】8. 跳台阶

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）

题解：中心思想就是 $F(N)=F(N-1)+F(N-2)$

- 常规递归解法 $F(N)=F(N-1)+F(N-2)$ ，时间复杂度太过于昂贵 $O(2^n)$
- Fibonacci数列（迭代） $O(n)$

```

# 递归:O(2^n)
def jumpFloor(number):
    if number <= 0:
        return
    elif number == 1:
        return 1
    elif number == 2:
        return 2
    res = 0
    res = res + jumpFloor(number-1) + jumpFloor(number-2)
    return res

# Fibonacci数列:O(n)
class Solution:
    def jumpFloor(self, number):
        # write code here
        # 也可以初始化l=1, r=2, 然后loop in range (number-1) , 可以cover n从1到n的情况
        l = 0
        r = 1
        for _ in range(number):
            l, r = r, l+r
        return r

```

- 二阶递推数列，可用矩阵乘法的形式表示 $O(\log N)$

状态矩阵为 2×2 的矩阵：

$$(F(n), F(n-1)) = (F(n-1), F(n-2)) \times \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

代入 $F(1)=1, F(2)=2, F(3)=3, F(4)=5$ 得

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}$$

$n>2$ 时，公式可简化为

$$(F(n), F(n-1)) = (F(n-1), F(n-2)) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix} = (2, 1) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^{n-2}$$

变成了求解矩阵N次方的问题

```
class Solution:
    def jumpFloor(self, number):
        # write code here
    def matrixPower(m, p):
        # 单位矩阵
        res = [[0 if i != j else 1 for i in range(len(m[0]))] for j in range(len(m))]
        tmp = m
        while p > 0:
            if p & 1 != 0:
                res = muliMatrix(res, tmp)
            tmp = muliMatrix(tmp, tmp)
            p >>= 1
        return res

    def muliMatrix(m1, m2):
        res = [[0 for i in range(len(m2[0]))] for j in range(len(m1))]
        for i in range(len(m1)):
            for j in range(len(m2[0])):
                for k in range(len(m1[0])):
                    res[i][j] += m1[i][k] * m2[k][j]
        return res

    if number < 1:
        return 0
    if number == 1 or number == 2:
        return number
    base = [[1, 1], [1, 0]]
    res = matrixPower(base, number-2)
    return 2*res[0][0] + res[0][1]
```

【递归和循环】9. 变态跳台阶

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。

题解

观察规律可得 $2^{(n-1)}$

两种解释：

- 每个台阶可以看作一块木板，让青蛙跳上去，n个台阶就有n块木板，最后一块木板是青蛙到达的位子，必须存在，其他(n-1)块木板可以任意选择是否存在，则每个木板有存在和不存在两种选择，(n-1)块木板就有 $[2^{(n-1)}]$ 种跳法，可以直接得到结果。
- 因为n级台阶，第一步有n种跳法：跳1级、跳2级、到跳n级 跳1级，剩下n-1级，则剩下跳法是f(n-1) 跳2级，剩下n-2级，则剩下跳法是f(n-2) 所以 $f(n)=f(n-1)+f(n-2)+\dots+f(1)$ 因为 $f(n-1)=f(n-2)+f(n-3)+\dots+f(1)$
所以 $f(n)=2*f(n-1)$

```
# 常规
class Solution:
    def jumpFloorII(self, number):
        # write code here
        if number < 1:
            return 0
        else:
            return 2***(number-1)
```

```
# 移位
return 1<<--number-1
```

【递归和循环】10. 矩形覆盖

我们可以用21的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个21的小矩形无重叠地覆盖一个 $2 \times n$ 的大矩形，总共多少种方法？

题解

Fibonacci数列

- 当 $n=1$ 时，只能竖着覆盖， $f(1)=1$ ；
- 当 $n=2$ 时，既可以横着覆盖，也可以竖着覆盖， $f(2)=2$ ；
- 当 $n=N$ 时，只需要考虑第一块如何覆盖即可，详见下图：

2*1的小矩阵



2*N的大矩阵



$n=1$ 时, $f(1)=1$



$n=N$ 时, 只有以下两种情况:



$n=2$ 时, $f(2)=2$



$$f(n) = f(n-2) + f(n-1)$$

```
class Solution:
    def rectCover(self, number):
        # write code here
        if number < 1:
            return 0
        else:
            l=1
            r=2
            for _ in range(number-1):
                l,r = r,l+r #(或者r+l也是一样的)
            return l
```

【位运算】11. 二进制中1的个数

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。

题解

如果一个整数不为0，那么这个整数至少有一位是1。如果我们把这个整数减1，那么原来处在整数最右边的1就会变为0，原来在1后面的所有的0都会变成1(如果最右边的1后面还有0的话)。其余所有位将不会受到影响。

举个例子：一个二进制数1100，从右边数起第三位是处于最右边的一个1。减去1后，第三位变成0，它后面的两位0变成了1，而前面的1保持不变，因此得到的结果是1011.我们发现减1的结果是把最右边的一个1开始的所有位都取反了。这个时候如果我们再把原来的整数和减去1之后的结果做与运算，从原来整数最右边一个1那一开始所有位都会变成0。如**1100&1011=1000**.也就是说，把一个整数减去1，再和原整数做与运算，会把该整数最右边一个1变成0.那么一个整数的二进制有多少个1，就可以进行多少次这样的操作。

```
class Solution:
    def NumberOf1(self, n):
        # write code here
        count = 0
        if n < 0:
            # Python需要与0xffffffff位与才能变成补码形式
            n = n & 0xffffffff
        while n:
            count += 1
            n = (n - 1) & n #二进制经典 与 操作
        return count
# python带的一个bin()方程
def NumberOf1(n):
    if n >= 0:
        bi = bin(n)
    else:
        bi = bin(n & 0xffffffff)
    return bi.count('1')
```

1100 & 1011 = 1000

1100和1011之间的关系：

1100-1=1011

12 11

0+0=0 0+1=1 1+1=10

0 0

1 1

2 10

3 11

4 100

5 101

6 110

7 111

8 1000

9 1001

10 1010

11

+ 1

100

【代码的完整性】12. 数值的整数次方

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。

保证base和exponent不同时为0

题解

快速求幂算法_hkdgjqr的专栏-CSDN博客

直接乘要做998次乘法。但事实上可以这样做，先求出 2^k 次幂： $3^{999} = 3^{(512 + 256 + 128 + 64 + 32 + 4 + 2 + 1)} = (3^{512}) * (3^{256}) * (3^{128}) * (3^{64}) * (3^{32}) * (3^4) * (3^2) * 3$ 这样只要做16次乘法。即使加上一些辅助的存储和运算，也比直接乘高效得多（尤其如果这里底数

 <https://blog.csdn.net/hkdgjqr/article/details/5381028>



x << y
Returns x with the bits shifted to the left by y places (and new bits on the right-hand-side are zeros). This is the same as multiplying x by 2^{**y} .

x >> y
Returns x with the bits shifted to the right by y places. This is the same as //ing x by 2^{**y} .

x & y
Does a "bitwise and". Each bit of the output is 1 if the corresponding bit of x AND of y is 1, otherwise it's 0.

x | y
Does a "bitwise or". Each bit of the output is 0 if the corresponding bit of x AND of y is 0, otherwise it's 1.

~x
Returns the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as $-x - 1$.

x ^ y
Does a "bitwise exclusive or". Each bit of the output is the same as the corresponding bit in x if that bit in y is 0, and it's the complement of the bit in x if that bit in y is 1.

Just remember about that infinite series of 1 bits in a negative number, and these should all make sense.

```
# 暴力连乘：时间复杂度是 O(N), 空间复杂度是 O(1)
class Solution:
    def Power(self, base, exponent):
        # write code here
        t = 1
        if exponent == 0:
            return 1
        elif exponent > 0:
            for i in range(exponent):
                t*=base
        else:
            for i in range(-exponent):
                t/=base
        return t
# 快速幂算法
class Solution:
    def fast_power(self, base, exponent):
        # 永远第一时间补上特殊情况
        if base == 0:
            return 0
        if exponent == 0:
            return 1
        # coding开始
        e = abs(exponent)
        tmp = base
        res = 1
        while(e > 0):
            #如果最后一位为1，那么给res乘上这一位的结果 (&判断e的奇偶，e为偶结果为0；e为奇结果为1)
            if e%2 == 1: # e&1是一个二进制 与 的操作比较
                res = res * tmp
            # 符号>>代表e/(2^1)的整数部分，找整除数
            e = e//2
            tmp = tmp * tmp
        return res if exponent > 0 else 1/res
3**100
100 = 2^6+2^5+2^2
3^(64+32+4)

>> 向右位移
bin(100):'0b1100100' # 1出现的位置就是需要res保存进去的位置
bin(50): '0b110010'
bin(25): '0b11001'

e 100 -> 50 -> 25 -> 12 -> 6 -> 3 -> 1 -> 0
tmp 3 -> 3^2 -> 3^4 -> 3^8 -> 3^16 -> 3^32 -> 3^64 -> 3^128 #tmp初始值是base 3
res 1 -> 1 -> 1 -> 3^4 -> 3^4 -> 3^4 -> 3^(4+32) -> 3^(4+32+64)
```

【代码的完整性】13. 调整数组顺序使奇数位于偶数前面

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。

题解

```
# 新建数组 时间复杂度为O(n), 空间复杂度为O(n)
class Solution:
    def reOrderArray(self, array):
        # write code here
        o=[]
        e=[]
        for i in array:
            if i%2 == 1:
                o.append(i)
            else:
                e.append(i)
        return o+e
# lambda表达式
def reOrderArray(array):
    return sorted(array, key=lambda c:c%2, reverse=True)
# 类似冒泡算法，前偶后奇数交换
def reOrderArray(array):
    # write code here
    for i in range(len(array)):
        for j in range(len(array)-1,i,-1): #j为顺序10, 9, 8, ..., i这样的数
            if array[j]%2 == 1 and array[j-1]%2 == 0: #前一个是奇数, 后一个是偶数
                array[j],array[j-1]=array[j-1],array[j]
    return array
```

【代码的鲁棒性】14. 链表中倒数第k个结点

输入一个链表，输出该链表中倒数第k个结点。

题解

- 用list存住链表值
- 设置两个指针，p1, p2，先让p1走k-1步，然后再一起走，直到p1为最后一个时，p2即为倒数第k个节点

```
# 剑一个list存起来
class Solution:
    def FindKthToTail(self, head, k): #这里head就是listnode的head, 可以直接用head.next得到下一个点
        # write code here
        if len(temp)<k or k<1: #一直都还是要考虑空值/异常情况
            return
        temp=[]
        while head:
            temp.append(head)
            head=head.next #访问下一个节点用next, 取值用va
        return temp[-k]
# 双指针
class Solution:
    def FindKthToTail(self, head, k):
        # write code here
        if head==None or k<1: #一直都还是要考虑空值
            return
        p1=head
        p2=head
        for _ in range(k): #所以p1在实际中走k步, 比理论多走一步
            if p1==None:
                return
            p1=p1.next #理论：先让p1走k-1步
        while p1: #因为p1需要走到None, 所以要在上面多走一步
            p1=p1.next
            p2=p2.next
        return p2
```

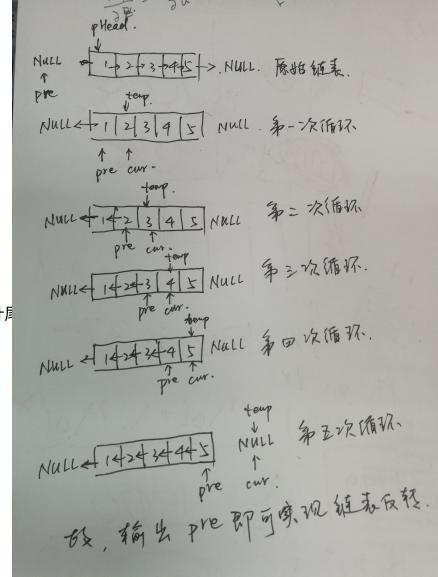
【代码的鲁棒性】15. 反转链表

输入一个链表，反转链表后，输出新链表的表头。

题解

三指针实现：1→2→3→4→5，遍历链表，把1的next置为None，2的next置为1，以此类推，5的next置为4。得到反转链表。需要考虑链表只有1个元素的情况。图中有具体的每步迭代的思路，最后输出pre而不是cur是因为最后一次迭代后cur已经指向None了，而pre是完整的反向链表。

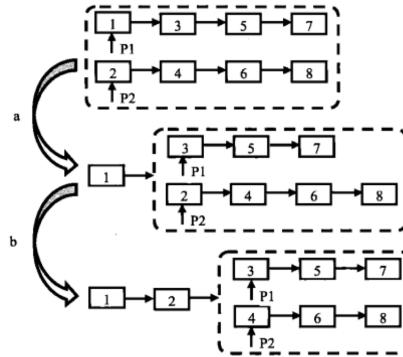
```
class Solution:  
    # 返回ListNode  
    def ReverseList(self, pHead):  
        # write code here  
        if pHead == None or pHead.next == None:  
            #None的情况要考虑完整  
            return pHead #如果链表只有一个值就只能直接phead, 所以不能写return None  
        pre = None  
        cur = pHead  
        while cur:  
            tmp = cur.next #先用一个tmp保持cur.next  
            cur.next = pre #反指针设定尾巴为pre  
            pre = cur #反指针cur的头重新赋予pre, 这样pre就拥有上一个cur.next的反指针  
            cur = tmp #再把tmp重新返回给cur, 继续下一个循环  
        return pre
```



【代码的鲁棒性】16. 合并两个排序的链表 — 重复比较的话应该往递归想

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。

题解



```
# 非递归版本  
class Solution:  
    # 返回合并后列表  
    def Merge(self, pHead1, pHead2):  
        # write code here  
        oHead = ListNode(None)  
        res = oHead  
        while pHead1 and pHead2:  
            if pHead1.val >= pHead2.val:  
                oHead.next = pHead2
```

```

        pHead2 = pHead2.next
    else:
        oHead.next = pHead1
        pHead1 = pHead1.next
        oHead = oHead.next
    if pHead1:
        oHead.next = pHead1
    elif pHead2:
        oHead.next = pHead2
    return res.next
# 递归版本
class Solution:
    # 返回合并后列表
    def Merge(self, pHead1, pHead2):
        # write code here
        if pHead1 is None:
            return pHead2
        if pHead2 is None:
            return pHead1
        if pHead1.val < pHead2.val:
            pHead1.next = self.Merge(pHead1.next, pHead2)
            return pHead1
        else:
            pHead2.next = self.Merge(pHead1, pHead2.next)
            return pHead2

```

* 【代码的鲁棒性】17. 树的子结构

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）

题解

- 在树A中找到和B的根结点的值一样的节点R

查找树中的值，即为树的遍历。**可用递归的方法**，也可以用循环的方法。因为递归的代码简洁，所以一般使用递归的方式。

- 判断A中以R为根结点的子树是不是包含和树B一样的结构

```

class Solution:
    # 遍历树A
    def HasSubtree(self, pRoot1, pRoot2):
        # write code here
        result = False
        if pRoot1 and pRoot2: # 经常会用if A and B去判断是否还有内容
            if pRoot1.val == pRoot2.val:
                result = self.IsSubtree(pRoot1,pRoot2)
            if not result:
                result = self.HasSubtree(pRoot1.left,pRoot2) or self.HasSubtree(pRoot1.right,pRoot2)
        return result
    # IsSubtree用于递归判断树的每个节点是否相同
    # 前两个if语句不可以颠倒顺序；如果2已经遍历完成，那么为True；颠倒的话会先判断1为None
    def IsSubtree(self, pRoot1, pRoot2):
        if pRoot2 == None: #如果root2先遍历完成发现都满足，那就是true
            return True
        if pRoot1 == None:
            return False
        if pRoot1.val != pRoot2.val:
            return False
        return self.IsSubtree(pRoot1.left,pRoot2.left) and self.IsSubtree(pRoot1.right,pRoot2.right)

```

【面试思路】18. 二叉树的镜像

操作给定的二叉树，将其变换为源二叉树的镜像。

题解

```

class Solution:
    # 返回镜像树的根节点
    def Mirror(self, root):
        # write code here
        if root == None:
            return root #或者None也可以
        # 交换左右子树
        tmp = root.left
        root.left = root.right
        root.right = tmp
        # 递归
        self.Mirror(root.left)
        self.Mirror(root.right)

```

源二叉树

```

8
/ \
6 10
/ \ / \
5 7 9 11

```

镜像二叉树

```

8
/ \
10 6
/ \ / \
11 9 7 5

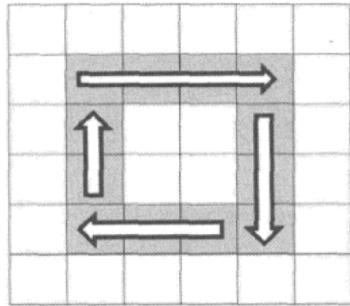
```

【画图让抽象形象化】19. 顺时针打印矩阵

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 4×4 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.

题解

- 外圈 → 内圈
- 魔方逆时针旋转



外圈到内圈

```

1 2 3
4 5 6
7 8 9
删除第一行后，逆时针旋转
6 9
5 8
4 7

```

```

# 魔方逆时针旋转1
class Solution:
    # matrix类型为二维列表，需要返回列表
    def printMatrix(self, matrix):
        result = []
        while(matrix):
            # 加上最上方一行
            result+=matrix.pop(0)
            if not matrix or not matrix[0]:
                break
            matrix = self.turn(matrix)
        return result
    def turn(self, matrix):
        num_r = len(matrix)
        num_c = len(matrix[0])
        newmat = []
        for i in range(num_c):

```

```

newmat2 = []
    for j in range(num_r):
        newmat2.append(matrix[j][i])
    newmat.append(newmat2)
    newmat.reverse()
    return newmat
# 魔方逆时针旋转2
class Solution:
    def printMatrix(self, matrix):
        res = []
        while matrix:
            res += matrix.pop(0)
            if matrix and matrix[0]:
                for row in matrix:
                    res.append(row.pop()) #这个是把右边的值打出来
            if matrix:
                res += matrix.pop()[-1:] #这个是最后一行
            if matrix and matrix[0]:
                for row in matrix[0:-1]: #这个是左边的值
                    res.append(row.pop(0))
        return res
#Mia的解答~~~~
class Solution:
    def printMatrix(self, matrix):
        if matrix == None:
            return None
        if len(matrix) == 1:
            return matrix[0]
        if len(matrix[0]) == 1:
            return [i[0] for i in matrix]
        if len(matrix[0])%2==0: k = len(matrix[0])/2
        else: k=(len(matrix)+1)/2
        n = 1
        new = []
        while n <= k:
            if len(matrix) > 1:
                for j in range(2*len(matrix)-2):
                    if j == 0:
                        new.extend(matrix[j])
                    elif j < len(matrix)-1:
                        new.append(matrix[j][-1])
                    elif j == len(matrix)-1:
                        new.extend(matrix[j][:-1])
                    else:
                        new.append(matrix[j-2*(j-len(matrix))+1][0])
            else:
                new.extend(matrix[0])
            matrix = [i[1:-1] for i in matrix[1:-1]]
            n += 1
        return new

```

【举例让抽象具体化】20. 包含min函数的栈

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））。

题解

创建一个辅助栈，每次把最小元素压入。可以保证data的最小值一直在辅助栈的顶端

```

class Solution:
    def __init__(self):
        self.datastack = []
        self.minstack = []
    def push(self, node): #只需要加node 不需要return
        self.datastack.append(node)
        if self.minstack:
            self.minstack.append(min(node, self.minstack[-1]))
        else:

```

```

        self.minstack.append(node)
    def pop(self):
        if self.datastack:
            self.minstack.pop() #也需要同样在minstack pop一个出来
            return self.datastack.pop()
    def top(self): #出栈
        if self.datastack:
            return self.datastack[-1] #需要返回一个值
    def min(self): # 注意题目条件时间复杂度要求是1
        if self.datastack:
            return self.minstack[-1]

```

* 【举例让抽象具体化】21. 栈的压入、弹出序列

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）

题解

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

```

举例：
入栈1, 2, 3, 4, 5
出栈4, 5, 3, 2, 1
首先1入辅助栈，此时栈顶1≠4，继续入栈2
此时栈顶2≠4，继续入栈3
此时栈顶3≠4，继续入栈4
此时栈顶4=4，出栈4，弹出序列向后一位，此时为5，，辅助栈里面是1, 2, 3
此时栈顶3≠5，继续入栈5
此时栈顶5=5，出栈5，弹出序列向后一位，此时为3，，辅助栈里面是1, 2, 3

```

```

class Solution:
    def IsPopOrder(self, pushV, popV):
        # stack中存入pushV中取出的数据
        stack = []
        while popV:
            # 如果stack的最后一个元素与popV中第一个元素相等，将两个元素都弹出
            if stack and stack[-1] == popV[0]:
                stack.pop()
                popV.pop(0)
            # 如果pushV中有数据，压入stack
            elif pushV:
                stack.append(pushV.pop(0))
            # 上面情况都不满足，直接返回false
            else: #pushV已经全部进去辅助栈，且辅助栈最后一个跟popV[0]也不相等
                return False
        return True

```

【队列实现BFS】22. 从上往下打印二叉树

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**。

题解

广度优先搜索 BFS，借助一个队列就可以实现（层次遍历）

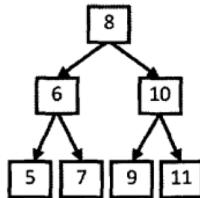


图 4.5 一棵二叉树，从上往下按层打印的顺序为 8、6、10、5、7、9、11

表 4.4 按层打印图 4.5 中的二叉树的过程

步骤	操作	队列
1	打印结点 8	结点 6、结点 10
2	打印结点 6	结点 10、结点 5、结点 7
3	打印结点 10	结点 5、结点 7、结点 9、结点 11
4	打印结点 5	结点 7、结点 9、结点 11
5	打印结点 7	结点 9、结点 11
6	打印结点 9	结点 11
7	打印结点 11	

```

class Solution:
    # 返回从上到下每个节点值列表，例：[1, 2, 3]
    def PrintFromTopToBottom(self, root):
        if not root:
            return [] # none return结果根据题目具体而改
        res = []
        queue = []
        queue.append(root)
        while len(queue) > 0:
            res.append(queue[0].val) #第一次循环root是8那个点
            if queue[0].left:
                queue.append(queue[0].left)
            if queue[0].right:
                queue.append(queue[0].right)
            queue.pop(0)
        return res

```

* 【举例让抽象具体化】23. 二叉搜索树的后序遍历序列

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。

题解

递归：BST的后序序列的合法序列是，对于一个序列S，最后一个元素是x（也就是根），如果去掉最后一个元素的序列为T，那么T满足：T可以分成两段，前一段（左子树）小于x，后一段（右子树）大于x，且这两段（子树）都是合法的后序序列。



BTS的性质：右>根>左

```

class Solution:
    def VerifySequenceOfBST(self, sequence):
        if not sequence or len(sequence) <= 0:

```

```

        return False
    # 左子树
    for i in range(len(sequence)):
        if sequence[i] > sequence[-1]:
            break
    # 右子树
    for j in range(i, len(sequence)): #range from i to len(sequence) !
        if sequence[j] < sequence[-1]: #不是小于等于，等于很可能在走到最后一个点的时候就return False
            return False
    res1=True
    if i>0: #排除i为0
        res1=self.VerifySquenceofBST(sequence[0:i])
    res2=True
    if i<len(sequence)-1:
        res2=self.VerifySquenceofBST(sequence[i:-1])
    return res1 and res2

```

* 【举例让抽象具体化】24. 二叉树中和为某一值的路径

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)

题解

非递归法：后序遍历

1. 进栈时候，把值同时压入路径的向量数组，修正路径的和
2. 出栈时候，先判断和是否相等，且该节点是否是叶节点，判断完成后保持和栈一致，抛出路径，修改路径的和
3. 向量数组和栈的操作要保持一致

注意把一个列表加到另一个列表中作为另一个列表的元素，一定要这样写list2.append(list1[:])（浅拷贝），不然加的是空的，这涉及到Python的可变对象、不可变对象等机制

```

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):
        res, val = [], []
        def goPath(root):
            if root:
                val.append(root.val)
                if root.left==None and root.right==None and sum(val)==expectNumber:
                    res.append(val[:])
                else:
                    goPath(root.left)
                    goPath(root.right)
                val.pop()
        goPath(root)
        return res

```

```

#题外话：找所有路径 - DFS
class TreeNode:
    def __init__(self, value):
        self.val = value
        self.left = None
        self.right = None

class Solution:
    def binaryTreePaths(self, root):
        if root == None:
            return []
        result = []
        self.DFS(root, result, [root.val])
        return result

```

```

def DFS(self, root, result, path):
    if root.left == None and root.right == None:
        result.append(path)
    if root.left != None:
        self.DFS(root.left, result, path + [root.left.val])
    if root.right != None:
        self.DFS(root.right, result, path + [root.right.val])

```

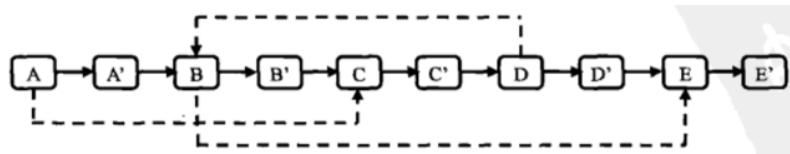
【分解让复杂问题简单】25. 复杂链表的复制

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）

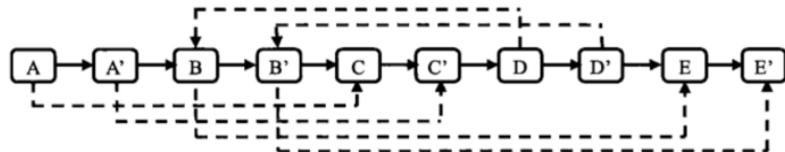
题解

- 递归法
- 哈希表
- 三步法

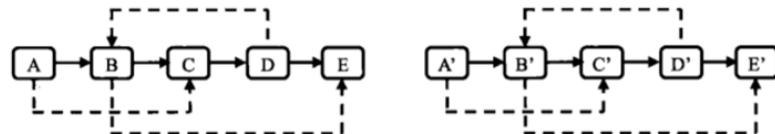
1. 把复制的结点链接在原始链表的每一对应结点后面



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

# 递归
class Solution:
    def Clone(self, head):

```

```

if not head: return
newNode = RandomListNode(head.label)
newNode.random = head.random #任意指针
newNode.next = self.Clone(head.next)
return newNode

# 哈希表
class Solution:
    def Clone(self, head):
        nodeList = []      #存放各个节点
        randomList = []    #存放各个节点指向的random节点。没有则为None
        labelList = []     #存放各个节点的值

        while head:
            randomList.append(head.random)
            nodeList.append(head)
            labelList.append(head.label)
            head = head.next
        #random节点的索引，如果没有则为1
        labelIndexList = map(lambda c: nodeList.index(c) if c else -1, randomList)

        dummy = RandomListNode(0)
        pre = dummy
        #节点列表，只要把这些节点的random设置好，顺序串起来就ok了。
        nodeList=map(lambda c:RandomListNode(c),labelList)
        #把每个节点的random绑定好，根据对应的index来绑定
        for i in range(len(nodeList)):
            if labelIndexList[i]!=-1:
                nodeList[i].random=nodeList[labelIndexList[i]]
        for i in nodeList:
            pre.next=i
            pre=pre.next
        return dummy.next

# 三步法
class Solution:
    # 返回 RandomListNode
    def Clone(self, pHead):
        if not pHead:
            return None

        dummy = pHead

        # first step, N' to N next
        while dummy:
            dummysnext = dummy.next
            copynode = RandomListNode(dummy.label)
            copynode.next = dummysnext
            dummy.next = copynode
            dummy = dummysnext

        dummy = pHead

        # second step, random' to random
        while dummy:
            dummyrandom = dummy.random
            copynode = dummy.next
            if dummyrandom:
                copynode.random = dummyrandom.next
            dummy = copynode.next

        # third step, split linked list
        dummy = pHead
        copyHead = pHead.next
        while dummy:
            copyNode = dummy.next
            dummysnext = copyNode.next
            dummy.next = dummysnext
            if dummysnext:
                copyNode.next = dummysnext.next
            else:
                copyNode.next = None

```

```

dummy = dummy.next
return copyHead

```

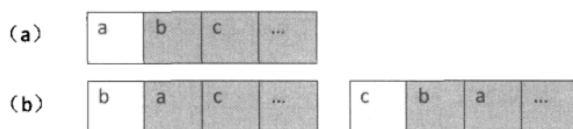
【分解让复杂问题简单】27. 字符串的排列

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。

题解

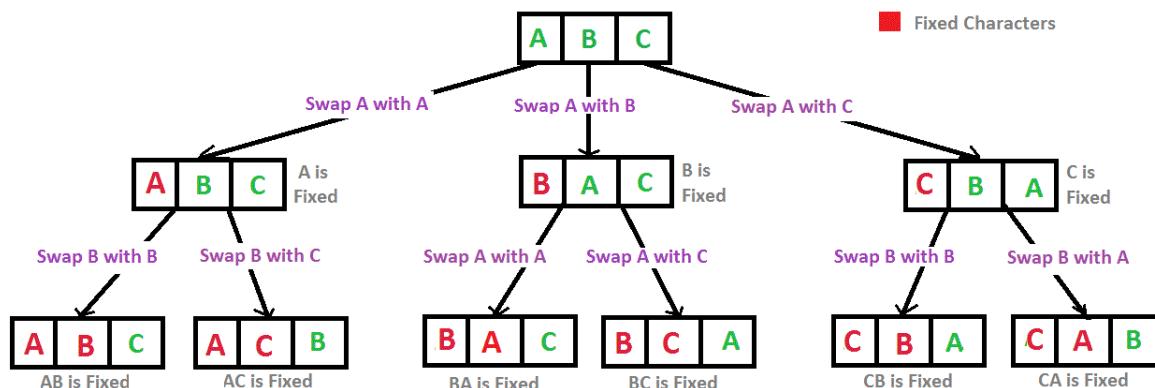
- 用递归的想法去做

我们求整个字符串的排列，可以看成两步：首先求所有可能出现在第一个位置的字符，即把第一个字符和后面所有的字符交换。图 4.14 就是分别把第一个字符 a 和后面的 b、c 等字符交换的情形。首先固定第一个字符（如图 4.14 (a) 所示），求后面所有字符的排列。这个时候我们仍把后面的所有字符分成两部分：后面字符的第一个字符，以及这个字符之后的所有字符。然后把第一个字符逐一和它后面的字符交换（如图 4.14 (b) 所示）……



- 回溯法

利用树去尝试不同的可能性，不断地去字符串数组里面拿一个字符出来拼接字符串，当字符串数组被拿空时，就把结果添加进结果数组里，然后回溯上一层。（通过往数组加回去字符以及拼接的字符串减少一个来回溯。）



Recursion Tree for Permutations of String "ABC"

```

# 递归
class Solution:
    def Permutation(self, ss):
        if len(ss) <= 1:
            return ss
        res = set() #用set避免重复
        for i in range(len(ss)):

```

```

# 每一个j是Permutation(ss[:i]+ss[i+1:])这个list中不同排列组合的一个string
    for j in self.Permutation(ss[:i]+ss[i+1:]):
        res.add(ss[i]+j) #set用add
    return sorted(res)

# 回溯
class Solution:
    def Permutation(self, ss):
        if len(ss) <= 1:
            return ss
        res=set()
        tmp=[]
        # 用一个dict记住每个单词出现次数
        n_dict = dict((x,0) for x in ss)
        for s in ss:
            n_dict[s]+=1
        self.Back(res,tmp,ss,n_dict)
        return sorted(list(res))

    def Back(self,res,tmp,ss,counter):
        if len(tmp) == len(ss):
            print(tmp)
            print(len(ss))
            res.add(''.join(tmp))
        else:
            for i in ss:
                if counter[i] == 0:
                    continue
                # 用一次就减一，回溯时记得加回来
                counter[i] -= 1
                tmp.append(i)
                self.Back(res,tmp,ss,counter)
                counter[i] += 1
                tmp.pop()

```

【时间效率】28. 数组中出现次数超过一半的数字

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。

题解

- 创dict记录次数
时间复杂度O(n)?
- 数组排序后，如果符合条件的数存在，则一定是数组中间那个数
快排sort，时间复杂度O(NlogN)
- 如果有符合条件的数字，则它出现的次数比其他所有数字出现的次数和还要多
在遍历数组时保存两个值：一是数组中一个数字，一是次数。遍历下一个数字时，若它与之前保存的数字相同，则次数加1，否则次数减1；若次数为0，则保存下一个数字，并将次数置为1。遍历结束后，所保存的数字即为所求。
然后再判断它是否符合条件即可。

```

# dict/counter
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        n_dict = dict((x,0) for x in numbers)
        for i in numbers:
            n_dict[i]+=1
            if n_dict[i] > len(numbers)//2:
                return i
        return 0

# 排序
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        # 对列表进行快排

```

```

left = 0
right = len(numbers) - 1
stack = [right, left]
while stack:
    low = stack.pop()
    high = stack.pop()
    if low >= high:
        continue
    less = low - 1
    mid = numbers[high]
    for i in range(low, high):
        if numbers[i] <= mid:
            less += 1
            numbers[less], numbers[i] = numbers[i], numbers[less]
    numbers[less + 1], numbers[high] = numbers[high], numbers[less + 1]
    stack.extend([high, less+2, less, low])
# 验证
count = 0
length = len(numbers) // 2
for i in numbers:
    if i == numbers[length // 2]:
        count += 1
return numbers[length // 2] if count > length / 2.0 else 0
# +1-1
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        if not numbers:
            return 0
        num = numbers[0]
        count = 1
        for i in range(1, len(numbers)):
            if numbers[i] == num:
                count += 1
            else:
                count -= 1
            if count == 0:
                num = numbers[i]
                count = 1
        count = 0
        for i in numbers:
            if i == num:
                count += 1
        return num if count > len(numbers) / 2.0 else 0
# Mia
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        if len(numbers)==1:
            return numbers[0]
        elif numbers.count(sorted(numbers)[int(len(numbers)/2)+1]) > len(numbers)/2:
            return sorted(numbers)[int(len(numbers)/2)+1]
        else:
            return 0

```

【时间效率】29. 最小的K个数

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

题解

- 快排 时间复杂度O(nlogn)
- partition 时间复杂度O(n)
使比第k个元素小的元素放左边，比其大的元素放右边
- 最小堆
用最小堆保存这k个数，每次只和堆顶比，如果比堆顶小，删除堆顶，新数入堆

表 5.1 两种算法的特点比较

	基于Partition函数的思路	基于堆或者红黑树的思路
时间复杂度	$O(n)$	$O(n * \log k)$
是否需要修改输入数组	是	否
是否适用于海量数据	否	是

```

# 作弊方法-排序
class Solution:
    def GetLeastNumbers_Solution(self, tinput, k):
        if k <= len(tinput): #注意条件
            return sorted(tinput)[:k]
        else:
            return []
# partition
import random
class Solution:
    def GetLeastNumbers_Solution(self, tinput, k):
        # write code here
        n = len(tinput)
        if n<=0 or k>n:
            return []
        if k==0:
            return []
        start = 0
        end = n-1
        index = self.partition(tinput,start,end)
        while index != k-1:
            if index >k-1:
                end = index - 1
                index = self.partition(tinput,start,end)
            else:
                start = index +1
                index = self.partition(tinput,start,end)
        res = tinput[:k]
        res=sorted(res)
        return res

    def partition(self,arr,start,end):
        if start==end:
            p=start
        else:
            p = random.randrange(start,end)
        arr[p],arr[end]=arr[end],arr[p]
        small = start-1
        for i in range(start,end):
            if arr[i]<arr[end]:
                small+=1
                if small != i:
                    arr[small],arr[i]=arr[i],arr[small]
        small +=1
        arr[small],arr[end]=arr[end],arr[small]
        return small

# 最小堆
class Solution:
    def minFixHeap(self,t,i,n):
        tmp = t[i]
        j = i*2+1
        while j < n:
            if j+1 < n and t[j+1] < t[j]:
                j+=1
            if t[j] >= tmp:
                break
            t[i] = t[j]
            i = j

```

```

        j = i*2 +1
        t[i] = tmp

    def GetLeastNumbers_Solution(self, tinput, k):
        lens = len(tinput)
        if k>lens or k <0 or lens ==0:
            return []
        for i in range(lens/2,-1,-1):
            self.minFixHeap(tinput,i,lens)

        res = []
        for i in range(lens-1,lens-k-1,-1):
            res.append(tinput[0])
            tinput[0] = tinput[i]
            self.minFixHeap(tinput,0,i)
        return res

```

【时间效率】30. 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)

题解

- 一个loop $O(n)$
- 动态规划

$dp[i]$ 表示以元素array[i]结尾的最大连续子数组和。

以[-2,-3,4,-1,-2,1,5,-3]为例

可以发现,

$dp[0] = -2$
 $dp[1] = -3$
 $dp[2] = 4$
 $dp[3] = 3$

以此类推,会发现

$dp[i] = \max\{dp[i-1]+array[i], array[i]\}.$

```

# 一个loop
class Solution:
    def FindGreatestSumOfSubArray(self, array):
        res = -7e20
        tmp = 0
        for i in range(len(array)):
            tmp += array[i]
            if tmp > res:
                res = tmp
            if tmp < 0:
                tmp = 0
        return res

# 动态规划
class Solution:
    def FindGreatestSumOfSubArray(self, array):
        n = len(array)
        dp = [ i for i in array]
        for i in range(1,n):

```

```

dp[i] = max(dp[i-1]+array[i], array[i])

return max(dp)

```

【时间效率】31. 整数中1出现的次数（从1到n整数中1出现的次数）

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）。

题解

像类似这样的问题，我们可以通过归纳总结来获取相关的东西。

首先可以先分类：

▼ 个位

我们知道在个位数上，1会每隔10出现一次，例如1、11、21等等，我们发现以10为一个阶梯的话，每一个完整的阶梯里面都有一个1，例如数字22，按照10为间隔来分三个阶梯，在完整阶梯0-9，10-19之中都有一个1，但是19之后有一个不完整的阶梯，我们需要去判断这个阶梯中会不会出现1，易推断知，如果最后这个露出来的部分小于1，则不可能出现1（这个归纳换做其它数字也成立）。

我们可以归纳个位上1出现的个数为：

$$\lfloor n/10 \rfloor * 1 + (n \% 10 == 0 ? 1 : 0)$$

▼ 十位

现在说十位数，十位数上出现1的情况应该是10-19，依然沿用分析个位数时候的阶梯理论，我们知道10-19这组数，每隔100出现一次，这次我们的阶梯是100，例如数字317，分析有阶梯0-99，100-199，200-299三段完整阶梯，每一段阶梯里面都会出现10次1（从10-19），最后分析露出来的那段不完整的阶梯。我们考虑如果露出来的数大于19，那么直接算10个1就行了，因为10-19肯定会出现；如果小于10，那么肯定不会出现十位数的1；如果在10-19之间的，我们计算结果应该是 $k - 10 + 1$ 。例如我们分析300-317，17个数字，1出现的个数应该是 $17 - 10 + 1 = 8$ 个。

那么现在可以归纳：十位上1出现的个数为：

$$\begin{aligned} \text{设 } k = n \% 100, \text{ 即为不完整阶梯段的数字} \\ \text{归纳式为: } (n / 100) * 10 + (\text{if}(k > 19) 10 \\ \text{else if}(k < 10) 0 \text{ else } k - 10 + 1) \end{aligned}$$

▼ 百位

现在说百位1，我们知道在百位，100-199都会出现百位1，一共出现100次，阶梯间隔为1000，100-199这组数，每隔1000就会出现一次。这次假设我们的数为2139。跟上述思想一致，先算阶梯数 * 完整阶梯中1在百位出现的个数，即 $n/1000 * 100$ 得到前两个阶梯中1的个数，那么再算漏出来的部分139，沿用上述思想，不完整阶梯数199，得到100个百位1， $100 \leq k \leq 199$ 则得到 $k - 100 + 1$ 个百位1。

那么继续归纳百位上出现1的个数：

$$\begin{aligned} \text{设 } k = n \% 1000 \text{ 归纳式为: } (n / 1000) * 100 + (\text{if}(k > 199) 100 \\ \text{else if}(k < 100) 0 \text{ else } k - 100 + 1) \end{aligned}$$

后面的依次类推....

▼ 再次回顾个位

我们把个位数上算1的个数的式子也纳入归纳式中

| $k = n \% 10$ 个位数上1的个数为: $n / 10 * 1 + (\text{if}(k > 1) 1 \text{ else if}(k < 1) 0 \text{ else } k - 1 + 1)$

完美! 归纳式看起来已经很规整了。来一个更抽象的归纳, 设 i 为计算1所在的位数, $i=1$ 表示计算个位数的1的个数, 10 表示计算十位数的1的个数等等。

| $k = n \% (i * 10)$ $\text{count}(i) = (n / (i * 10)) * i + (\text{if}(k > i * 2 - 1) i \text{ else if}(k < i) 0 \text{ else } k - i + 1)$

好了, 这样从10到10的 n 次方的归纳就完成了。

| $\text{sum1} = \text{sum}(\text{count}(i)), i = \text{Math.pow}(10, j), 0 <= j <= \log_{10}(n)$

但是有一个地方值得我们注意的, 就是代码的简洁性来看, 有多个ifelse不太好, 能不能进一步简化呢? 我们可以把后半段简化成这样, 我们不去计算 $i * 2 - 1$ 了, 我们只需保证 $k - i + 1$ 在 $[0, i]$ 区间内就行了, 最后后半段可以写成这样

| $\min(\max((n \bmod (i * 10)) - i + 1, 0), i)$

对于**824883294**, 先求 $0 - 800000000$ 之间(不包括 800000000)的, 再求 $0 - 24883294$ 之间的。

如果等于1, 如1244444, 先求 $0 - 1000000$ 之间, 再求 $1000000 - 1244444$, 那么只需要加上 $244444 + 1$, 再求 $0 - 244444$ 之间的1

如果大于1, 例: $0 - 800000000$ 之间1的个数为8个 100000000 的1的个数加上 100000000 , 因为从 $1000000000 - 2000000000$ 共有 1000000000 个数且最高位都为1。

对于最后一位数, 如果大于1, 直接加上1即可。

```
class Solution:
    def NumberOf1Between1AndN_Solution(self, n):
        result = 0
        if n < 0:
            return 0
        length = len(str(n))
        listN = list(str(n))
        for i, v in enumerate(listN):
            a = length - i - 1 # a为10的幂
            if i == length - 1 and int(v) >= 1:
                result += 1
                break
            if int(v) > 1:
                result += int(10 ** a * a / 10) * int(v) + 10 ** a
            if int(v) == 1:
                result += (int(10 ** a * a / 10) + int("".join(listN[i+1:]))) + 1
        return result
```

【时间效率】32. 把数组排成最小的数

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 。

题解

1. 不够位数的补齐比较即可

2. 注意string可以起到拼接+比较大小的作用

```
[In [3]: '332'>'313'
Out[3]: True

[In [4]: '3'>'4'
Out[4]: False

[In [5]: '3'+'1'
Out[5]: '31'
```

```
# 补齐位数
class Solution:
    def PrintMinNumber(self, numbers):
        if not numbers:
            return ''
        maxl = len(str(max(numbers)))
        tmp = [str(x) for x in numbers]
        for i in range(len(tmp)):
            while len(tmp[i]) < maxl:
                tmp[i] += tmp[i][-1]
        tmp = [int(x) for x in tmp]
        zlist = list(zip(tmp, numbers))
        zlist = sorted(zlist, key = lambda x:x[0])
        res = ''
        for i in zlist:
            res += str(i[1])
        return res.lstrip('0') or 0
# 比较两个字符串加起来大小
# python3没有cmp
class Solution:
    def PrintMinNumber(self, numbers):
        # write code here
        if not numbers: return ""
        numbers = list(map(str, numbers))
        numbers.sort(cmp=lambda x, y: cmp(x + y, y + x))
        return ''.join(numbers).lstrip('0') or '0'
# Mia --- 网上看冒泡思路后自己写的：先转成string，利用string不断拼接找到最小的数
def PrintMinNumber(self, numbers):
    # write code here
    if numbers is None or numbers == []:
        return ""
    if len(numbers)==1:
        return numbers[0]
    numbers = [str(i) for i in numbers]
    for i in range(len(numbers)):
        for j in range(i, len(numbers)):
            if numbers[j]+numbers[i]<numbers[i]+numbers[j]:
                numbers[i], numbers[j]=numbers[j], numbers[i]
    return int(''.join(numbers))
```

【时间空间效率的平衡】33. 丑数

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。

题解

- 逐个判断每个整数是否为丑数，时间复杂度高
- 创建数组存储已找到丑数，用空间换时间

说下思路，如果p是丑数，那么 $p=2^x * 3^y * 5^z$

那么只要赋予x,y,z不同的值就能得到不同的丑数。

如果要顺序找出丑数，要知道下面几个特点：

对于任何丑数p：

- (一) $2*p, 3*p, 5*p$ 都是丑数，并且 $2*p < 3*p < 5*p$
- (二) 如果 $p < q$, 那么 $2*p < 2*q, 3*p < 3*q, 5*p < 5*q$

现在说说算法思想：

由于1是最小的丑数，那么从1开始，把 $2*1, 3*1, 5*1$ ，进行比较，得出最小的就是1的下一个丑数，也就是 $2*1$ ，

这个时候，多了一个丑数'2'，也就又多了3个可以比较的丑数， $2*2, 3*2, 5*2$ ，这个时候就把之前'1'生成的丑数和'2'生成的丑数加进来也就是 $(3*1, 5*1, 2*2, 3*2, 5*2)$ 进行比较，找出最小的。。。如此循环下去就会发现，每次选进来一个丑数，该丑数又会生成3个新的丑数进行比较。

下面说一个O(n)的算法。

既然有 $p < q$, 那么 $2*p < 2*q$, 那么“我”在前面比你小的数都没被选上，你后面生成新的丑数一定比“我”大吧，那么你乘2生成的丑数一定比我乘2的大吧，那么在我选上之后你才有机会选上。

其实每次我们只用比较3个数：

用于乘2的最小的数、用于乘3的最小的数，用于乘5的最小的数。也就是比较 $(2*x, 3*y, 5*z)$ ， $x \geq y \geq z$ 的

另一个解释：

找出最小值之后的替换 是指之前顺序排列的丑数数组的每个值（!）都要乘以2,3,5的比较，一开始 $1*2$ 和 $1*3$ 和 $1*5$ 比较，找出最小的是2，把2放进a数组，这时候替换2的数就是 $2*2$; 比较 $2*2, 1*3, 1*5$ ，找到最小的是3，把3放进a数组，替换3的是 $2*3$; 比较 $2*2, 2*3, 1*5$ ，找到最小是4,4放进a数组，这时候替换的用来乘以2的数就是3即 $3*2$

```
# 逐个判断
class Solution:
    def GetUglyNumber_Solution(self, index):
        n = 1
        c = 1
        while c < index:
            if self.isUgly(n):
                n+=1
                c+=1
            else:
                n+=1
        return n
    def isUgly(self, number):
        while number%2 == 0:
            number/=2
        while number%3 == 0:
            number/=3
        while number%5 == 0:
            number/=5
        return True if number == 1 else False
# 空间换时间
def GetUglyNumber_Solution(self, index):
    if index < 7: #不包含7, 7是单纯质数
        return index
    res = [1] #初始化有1
    t2 = t3 = t5 = 0
    for i in range(1,index):
        res.append(min(res[t2]*2,min(res[t3]*3,res[t5]*5)))
        if res[i] == res[t2]*2 : t2+=1
        if res[i] == res[t3]*3 : t3+=1
        if res[i] == res[t5]*5 : t5+=1
    return res[-1]
```

【时间空间效率的平衡】34. 第一个只出现一次的字符位置

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)

题解

- 一次循环初始化cDict, 一次循环生成记录字符出现次数的dict, 一次循环找到count为1的字符, 一次循环找第一个字符

```
# 时间复杂度O(n)
class Solution:
    def FirstNotRepeatingChar(self, s):
        cDict = dict((x, 0) for x in set(s))
        for i in s:
            cDict[i] += 1
        sList = []
        for sString, sCount in cDict.items():
            if sCount == 1:
                sList.append(sString)
        for i in range(len(s)):
            if s[i] in sList:
                return i
        return -1
# 一行写法 (用count method)
def FirstNotRepeatingChar(s):
    return [i for i in range(len(s)) if s.count(s[i]) == 1][0] if s else -1
```

35. 数组中的逆序对

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 $P \% 1000000007$

输入描述:

题目保证输入的数组中没有相同的数字

数据范围:

对于%50的数据, size $\leq 10^4$

对于%75的数据, size $\leq 10^5$

对于%100的数据, size $\leq 2 \times 10^5$

示例1

输入: 1,2,3,4,5,6,7,0

输出: 7

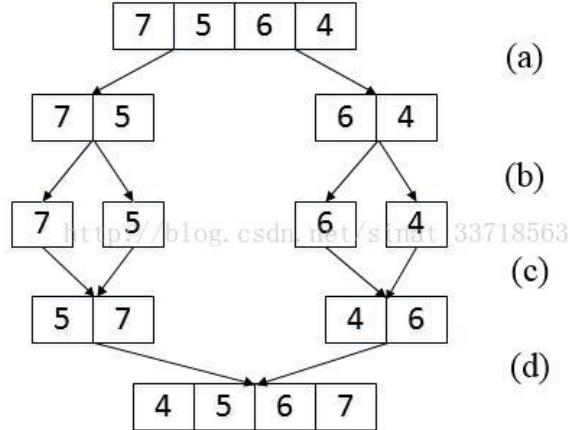
题解

- 暴力求解法, 时间复杂度为 $O(n^2)$, 空间复杂度 $O(1)$
- 归并排序的改进, 把数据分成前后两个数组(递归分到每个数组仅有一个数据项), 合并数组, 合并时, 出现前面的数组值 $array[i]$ 大于后面数组值 $array[j]$ 时; 则前面数组 $array[i] \sim array[mid]$ 都是大于 $array[j]$ 的, $count += mid + 1 - i$

时间复杂度 $O(n \log n)$, 空间复杂度 $O(n)$

看到这个题目, 我们的第一反应是顺序扫描整个数组。每扫描到一个数组的时候, 逐个比较该数字和它后面的数字的大小。如果后面的数字比它小, 则这两个数字就组成了一个逆序对。假设数组中含有n个数字。由于每个数字都要和 $O(n)$ 这个数字比较, 因此这个算法的时间复杂度为 $O(n^2)$ 。

我们以数组{7,5,6,4}为例来分析统计逆序对的过程。每次扫描到一个数字的时候，我们不拿ta和后面的每一个数字作比较，否则时间复杂度就是 $O(n^2)$ ，因此我们可以考虑先比较两个相邻的数字。



(a) 把长度为4的数组分解成两个长度为2的子数组；

(b) 把长度为2的数组分解成两个成都为1的子数组；

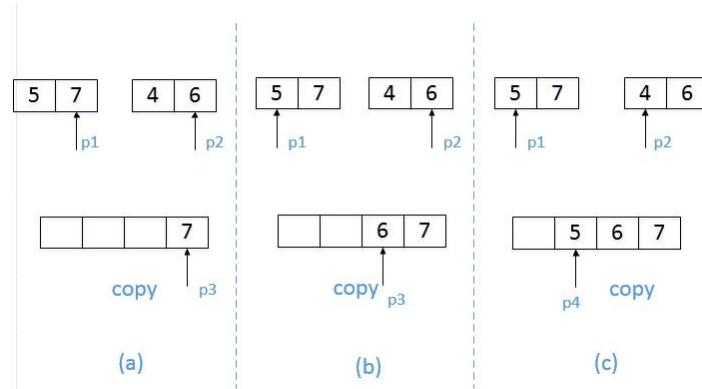
(c) 把长度为1的子数组 合并、排序并统计逆序对；

(d) 把长度为2的子数组合并、排序，并统计逆序对；

在上图（a）和（b）中，我们先把数组分解成两个长度为2的子数组，再把这两个子数组分别拆成两个长度为1的子数组。接下来一边合并相邻的子数组，一边统计逆序对的数目。在第一对长度为1的子数组{7}、{5}中7大于5，因此(7,5)组成一个逆序对。同样在第二对长度为1的子数组{6}、{4}中也有逆序对 (6,4)。由于我们已经统计了这两对子数组内部的逆序对，因此需要把这两对子数组 排序 如上图（c）所示，以免在以后的统计过程中再重复统计。

接下来我们统计两个长度为2的子数组子数组之间的逆序对。合并子数组并统计逆序对的过程如下图所示。

我们先用两个指针分别指向两个子数组的末尾，并每次比较两个指针指向的数字。如果第一个子数组中的数字大于第二个子数组中的数字，则构成逆序对，并且逆序对的数目等于第二个子数组中剩余数字的个数，如下图（a）和（c）所示。如果第一个数组的数字小于或等于第二个数组中的数字，则不构成逆序对，如图b所示。每一次比较的时候，我们都把较大的数字从后面往前复制到一个辅助数组中，确保 **辅助数组（记为copy）** 中的数字是递增排序的。在把较大的数字复制到辅助数组之后，把对应的指针向前移动一位，接下来进行下一轮比较。



过程：先把数组分割成子数组，先统计出子数组内部的逆序对的数目，然后再统计出两个相邻子数组之间的逆序对的数目。在统计逆序对的过程中，还需要对数组进行排序。如果对排序算法很熟悉，我们不难发现这个过程实际上就是归并排序。

```
count = 0
class Solution:
    def InversePairs(self, data):
        global count
        def MergeSort(lists):
            global count
            if len(lists) <= 1:
                return lists
            num = int(len(lists)/2)
            left = MergeSort(lists[:num])
            right = MergeSort(lists[num:])
            r, l=0, 0
            result=[]
            while l<len(left) and r<len(right):
                if left[l] < right[r]:
                    result.append(left[l])
                    l += 1
                else:
                    result.append(right[r])
                    r += 1
                    count += len(left)-l
            result += right[r:]
            result += left[l:]
            return result
        MergeSort(data)
        return count%1000000007
```

36. 两个链表的第一个公共结点

输入两个链表，找出它们的第一个公共结点。（注意因为传入数据是链表，所以错误测试数据的提示是用其他方式显示的，保证传入数据是正确的）

题解

- 暴力法，时间复杂度 $O(mn)$
- 辅助栈，时间复杂度 $O(m+n)$ ，空间复杂度 $O(m+n)$

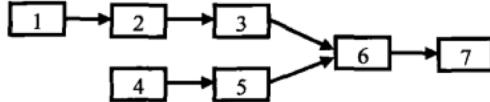


图 5.3 两个链表在值为 6 的结点处交汇

经过分析我们发现，如果两个链表有公共结点，那么公共结点出现在两个链表的尾部。如果我们从两个链表的尾部开始往前比较，最后一个相同的结点就是我们要找的结点。可问题是在单向链表中，我们只能从头结点开始按顺序遍历，最后才能到达尾结点。最后到达的尾结点却要最先被比较，这听起来是不是像“后进先出”？于是我们就能想到用栈的特点来解决这个问题：分别把两个链表的结点放入两个栈里，这样两个链表的尾结点就位于两个栈的栈顶，接下来比较两个栈顶的结点是否相同。如果相同，则把栈顶弹出接着比较下一个栈顶，直到找到最后一个相同的结点。

- 两链表连接，时间复杂度 $O(m+n)$

0-1-2-3-4-5-null

a-b-4-5-null

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！

```
# 暴力法
class Solution:
    def FindFirstCommonNode(self, pHead1, pHead2):
        while pHead1:
            tmp = pHead2
            while tmp:
                if pHead1 == tmp:
                    return pHead1
                tmp = tmp.next
            pHead1 = pHead1.next
        return None

# 辅助栈
class Solution:
    def FindFirstCommonNode(self, pHead1, pHead2):
        if not pHead1 or not pHead2:
            return None
        stack1 = []
        stack2 = []
        while pHead1:
            stack1.append(pHead1)
            pHead1 = pHead1.next
        while pHead2:
            stack2.append(pHead2)
            pHead2 = pHead2.next
        res = None
        while stack1 and stack2:
            if stack1[-1] == stack2.pop():
                res = stack1.pop()
            else:
                break
        return res

# 两链表连接
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。

题解

- 二分法+找周边相同值，时间复杂度 $O(n)$
- 二分法找第一个k和最后一个k（递归）
- 如果数字都为整数，二分法找 $k-0.5$ 和 $k+0.5$ 的位置相减即可
- `return data.count(k)`

```
# 二分法+找周边相同值，考虑升序和降序情况
class Solution:
    def GetNumberOfK(self, data, k):
```

```

        if not data or len(data) == 0:
            return 0
        if data[0] > data[-1]:
            flag = True
        elif data[0] < data[-1]:
            flag = False
        else:
            if data[0] == k:
                return len(data)
            else:
                return 0
    l = 0
    r = len(data)-1
    exist = False
    while l <= r:
        mid = l + (r-l)//2
        if data[mid] == k:
            exist = True
            break
        elif data[mid] > k:
            if flag:
                l = mid+1
            else:
                r = mid-1
        else:
            if flag:
                r = mid-1
            else:
                l = mid+1
    if exist:
        res = -1
        for i in range(mid,-1,-1):
            if data[i] == k:
                res += 1
            else:
                break
        for i in range(mid,len(data)):
            if data[i] == k:
                res += 1
            else:
                break
        return res
    else:
        return 0
# 二分法找第一个k和最后一个k（递归），只能升序
class Solution:
    def GetFirstK(self, data, k):
        low = 0
        high = len(data) - 1
        while low <= high:
            mid = (low + high) // 2
            if data[mid] < k:
                low = mid + 1
            elif data[mid] > k:
                high = mid - 1
            else:
                #当到list[0]或不为k的时候跳出函数
                if mid == low or data[mid - 1] != k:
                    return mid
                else:
                    high = mid - 1
        return -1

    def GetLastK(self, data, k):
        low = 0
        high = len(data) - 1
        while low <= high:
            mid = (low + high) // 2
            if data[mid] > k:
                high = mid - 1
            elif data[mid] < k:
                low = mid + 1

```

```

        else:
            if mid == high or data[mid + 1] != k:
                return mid
            else:
                low = mid + 1
    return -1

def GetNumberOfK(self, data, k):
    if not data:
        return 0
    if self.GetLastK(data, k) == -1 and self.GetFirstK(data, k) == -1:
        return 0
    return self.GetLastK(data, k) - self.GetFirstK(data, k) + 1

```

38. 二叉树的深度

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。

题解

- 递归（层次遍历）
- 层次遍历也可以用队列完成

```

# 递归
class Solution:
    def TreeDepth(self, pRoot):
        if not pRoot:
            return 0
        depth = self.dtree(pRoot, 0)
        return depth
    def dtree(self, root, depth):
        if not root:
            return depth
        depth += 1
        return max(self.dtree(root.left, depth), self.dtree(root.right, depth))

# 递归改进版
class Solution:
    def TreeDepth(self, pRoot):
        if pRoot is None:
            return 0
        count = max(self.TreeDepth(pRoot.left), self.TreeDepth(pRoot.right)) + 1
        return count

```

39. 平衡二叉树

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

题解

- 左右子树判断深度差是否不超过1
- 后续遍历，一边遍历一边判断是否平衡

```

# 左右子树判断深度差是否不超过1
class Solution:
    def IsBalanced_Solution(self, pRoot):
        if not pRoot:
            return True
        ldepth = self.dtree(pRoot.left, 0)
        rdepth = self.dtree(pRoot.right, 0)
        diff = abs(ldepth - rdepth)
        if diff > 1:

```

```

        return False
    return self.IsBalanced_Solution(pRoot.left) & self.IsBalanced_Solution(pRoot.right)
def dtree(self, root, depth):
    if not root:
        return depth
    depth += 1
    return max(self.dtree(root.left, depth), self.dtree(root.right, depth))
# 每个结点只遍历一次
class Solution:
    res = True
    def IsBalanced_Solution(self, pRoot):
        # write code here
        self.helper(pRoot)
        return self.res
    def helper(self,root):
        if not root:
            return 0
        if not self.res : return 1
        left = 1 + self.helper(root.left)
        right = 1 + self.helper(root.right)
        if abs(left-right)>1:
            self.res = False
        return max(left,right)
# 另一种写法
class Solution:
    def IsBalanced_Solution(self, p):
        return self.dfs(p) != -1
    def dfs(self, p):
        if p is None:
            return 0
        left = self.dfs(p.left)
        if left == -1:
            return -1
        right = self.dfs(p.right)
        if right == -1:
            return -1
        if abs(left - right) > 1:
            return -1
        return max(left, right) + 1

```

40. 数组中只出现一次的数字

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。

题解

- Dictionary
- 异或运算（只有一个数字的话更简单）

位运算中异或的性质：两个相同数字异或=0，一个数和0异或还是它本身

我们还是从头到尾依次异或数组中的每一个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果。因为其他数字都出现了两次，在异或中全部抵消了。由于这两个数字肯定不一样，那么异或的结果肯定不为 0，也就是说在这个结果数字的二进制表示中至少就有一位为 1。我们在结果数字中找到第一个为 1 的位的位置，记为第 n 位。现在我们以第 n 位是不是 1 为标准把原数组中的数字分成两个子数组，第一个子数组中每个数字的第 n 位都是 1，而第二个子数组中每个数字的第 n 位都是 0。由于我们分组的标准是数字中的某一位是 1 还是 0，那么出现了两次的数字肯定被分配到同一个子数组。因为两个相同的数字的任意一位都是相同的，我们不可能把两个相同的数字分配到两个子数组中去，于是我们已经把原数组分成了两个子数组，每个子数组都包含一个只出现一次的数字，而其他数字都出现了两次。我们已经知道如何在数组中找出唯一一个只出现一次数字，因此到此为止所有的问题都已经解决了。

举个例子，假设输入数组 {2, 4, 3, 6, 3, 2, 5, 5}。当我们依次对数组中的每一个数字做异或运算之后，得到的结果用二进制表示是 0010。异或得到结果中的倒数第二位是 1，于是我们根据数字的倒数第二位是不是 1 分为两个数组。第一个子数组 {2, 3, 6, 3, 2} 中所有数字的倒数第二位都是 1，而第二个子数组 {4, 5, 5} 中所有数字的倒数第二位都是 0。接下来只要分别对这两个子数组求异或，就能找出第一个子数组中只出现一次的数字是 6，而第二个子数组中只出现一次的数字是 4。

```
# 自己写的O(n)解法
class Solution:
    # 返回[a,b] 其中ab是出现一次的两个数字
    def FindNumsAppearOnce(self, array):
        if not array or len(array) < 2:
            return []
        cDict = dict((x, 0) for x in array)
        for i in array:
            cDict[i] += 1
        res = []
        for val, cnt in cDict.items():
            if cnt == 1:
                res.append(val)
        return res

# 异或
class Solution:
    def FindNumsAppearOnce(self, array):
        if not array or len(array) < 2:
            return []
        a = 0
        for i in array:
            a^=i
        idx = 0
        while a&1 == 0:
            a = a>>1
            idx += 1
        r1, r2 = 0, 0
        for i in array:
            if self.isBit(i, idx) == 0:
                r1^=i
            else:
                r2^=i
        return [r1, r2]
    def isBit(self, num, idx):
        num = num>>idx
        return num&1
```

41. 和为S的连续正数序列

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的序列为: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!

输出描述:

输出所有和为S的连续正数序列。序列内按照从小至大的顺序,序列间按照开始数字从小到大的顺序

题解

- 从1到S一个个数字试,时间复杂度 $O(n^2)$
- 两个指针在数组上滑动

有了解决前面问题的经验,我们也考虑用两个数 small 和 big 分别表示序列的最小值和最大值。首先把 small 初始化为 1, big 初始化为 2。如果从 small 到 big 的序列的和大于 s, 我们可以从序列中去掉较小的值, 也就是增大 small 的值。如果从 small 到 big 的序列的和小于 s, 我们可以增大 big, 让这个序列包含更多的数字。因为这个序列至少要有两个数字, 我们一直增加 small 到 $(1+s)/2$ 为止。

以求和为 9 的所有连续序列为为例,我们先把 small 初始化为 1, big 初始化为 2。此时介于 small 和 big 之间的序列是{1, 2}, 序列的和为 3, 小于 9, 所以我们下一步要让序列包含更多的数字。我们把 big 增加 1 变成 3, 此时序列为{1, 2, 3}。由于序列的和是 6, 仍然小于 9, 我们接下来再增加 big 变成 4, 介于 small 和 big 之间的序列也随之变成{1, 2, 3, 4}。由于序列的和 10 大于 9, 我们要删去去序列中的一些数字, 于是我们增加 small 变成 2, 此时得到的序列是{2, 3, 4}, 序列的和正好是 9。我们找到了第一个和为 9 的连续序列,把它打印出来。接下来我们再增加 big, 重复前面的过程,可以找到第二个和为 9 的连续序列{4, 5}。可以用表 6.2 总结整个过程。

```
# 从1到S一个个数字试
class Solution:
    def FindContinuousSequence(self, tsum):
        tlist = list(range(1,tsum))
        res = []
        for i in range(tsum-1):
            a = []
            b = 0
            for j in range(i,tsum-1):
                a.append(j+1)
                b += (j+1)
                if b == tsum:
                    res.append(a)
                    break
                elif b > tsum:
                    break
            return res
# 两个指针在数组上滑动
class Solution:
    def FindContinuousSequence(self, tsum):
```

```

if tsum < 3:
    return []
res = []
small,big = 1,2
mid = (1 + tsum) / 2
curSum = small + big

while small < mid:
    if curSum == tsum:
        res.append(range(small, big + 1))
        small += 2
        big += 1
        curSum -= small * 2 - 3
        curSum += big
    elif curSum < tsum:
        big += 1
        curSum += big
    else:
        curSum -= small
        small += 1
return res

```

42. 和为S的两个数字

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。

输出两个数，小的先输出。

题解

前后两个指针

```

class Solution:
    def FindNumbersWithSum(self, array, tsum):
        if not array or len(array)<2:
            return []
        a,b = 0,len(array)-1
        while a < b:
            if array[a]+array[b] == tsum:
                return [array[a],array[b]]
            elif array[a]+array[b] > tsum:
                b-=1
            else:
                a+=1
        return []

```

43. 左旋转字符串

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！

题解

- 作弊法（slicing）
- 设X=abc，Y=def，所以字符串可以表示成XY，如题干，问如何求得YX。假设X的翻转为XT，XT=cba，同理YT=fed，那么YX=(XTYT)T，三次翻转后可得结果

$$YX = (X^T Y^T)^T$$

```

# 作弊
class Solution:
    def LeftRotateString(self, s, n):
        if not s or len(s) < 1:
            return ""
        n = n%len(s)
        return s[n:]+s[:n]

# 反转
class Solution:
    def LeftRotateString(self, s, n):
        res, length = list(s), len(s)
        if n > length : return ""
        for i in range(int(n/2)):
            res[i], res[n-1-i] = res[n-1-i], res[i]
        for i in range(n, int((n+length)/2)):
            res[i], res[length-1-i+n] = res[length-1-i+n], res[i]
        for i in range(int(length/2)):
            res[i], res[length-1-i] = res[length-1-i], res[i]
        return "".join(res)

```

44. 翻转单词顺序列

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？

题解

- 作弊
- 两次翻转，第一次整体翻转，第二次每个单词再翻转（时间O(n),空间O(1)）

```

# 作弊
class Solution:
    def ReverseSentence(self, s):
        s = s.split(' ')
        return ' '.join(s[::-1])

# 作弊2
class Solution:
    def ReverseSentence(self, s):
        s = s.split(' ')
        a,b = 0,len(s)-1
        while a < b:
            s[a],s[b] = s[b],s[a]
            a+=1
            b-=1
        return ' '.join(s)

# 堆，无聊，复制了一个来
def ReverseSentence(self, s):
    alist = s.split(" ")
    stacklist = []
    for i in alist:
        stacklist.append(i)
    result = []
    while len(stacklist) > 0:
        result.append(stacklist.pop())
    return " ".join([x for x in result])

```

45. 扑克牌顺子

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。

题解

- max-min<5即可, 不允许重复
- 先统计王的数量, 再把牌排序, 如果后面一个数比前面一个数大于1以上, 那么中间的差值就必须用王来补了。看王的数量够不够, 如果够就返回true, 否则返回false

```
# max-min<5
class Solution:
    def IsContinuous(self, numbers):
        if not numbers:
            return False
        if numbers.count(0) < 4:
            numbers = [x for x in numbers if x!= 0]
            if len(numbers) == len(set(numbers)):
                if max(numbers)-min(numbers) <= 4:
                    return True
                return False
            elif numbers.count(0) == 4:
                return True
        # 用王补差
        def IsContinuous(self, numbers):
            if not numbers: return False
            numbers.sort()
            zeroNum = numbers.count(0)
            for i, v in enumerate(numbers[:-1]):
                if v != 0:
                    if numbers[i+1]==v: return False
                    zeroNum = zeroNum - (numbers[i + 1] - v) + 1
                    if zeroNum < 0:
                        return False
            return True
```

46. 圆圈中最后剩下的数

0,1,...,n-1这n个数字排成一个圆圈, 从数字0开始每次从这个圆圈里删除第m个数字。求出这个圆圈里剩下的最后一个数字。如果没有, 请返回-1

题解

- 迭代剪掉第m个数字, 把剩下的数组反过来拼起来, 时间复杂度高O(mn)
- 使用list模拟循环链表, 用cur作为指向list的下标位置, 当到尾部时移到头部
- 动态规划, 时间复杂度O(n)

首先我们定义一个关于 n 和 m 的方程 $f(n, m)$, 表示每次在 n 个数字 $0, 1, \dots, n-1$ 中每次删除第 m 个数字最后剩下的数字。

在这 n 个数字中, 第一个被删除的数字是 $(m-1)\%n$ 。为了简单起见, 我们把 $(m-1)\%n$ 记为 k , 那么删除 k 之后剩下的 $n-1$ 个数字为 $0, 1, \dots, k-1, k+1, \dots, n-1$, 并且下一次删除从数字 $k+1$ 开始计数。相当于在剩下的序列中, $k+1$ 排在最前面, 从而形成 $k+1, \dots, n-1, 0, 1, \dots, k-1$ 。该序列最后剩下的数字也应该是关于 n 和 m 的函数。由于这个序列的规律和前面最初的序列不一样 (最初的序列是从 0 开始的连续序列), 因此该函数不同于前面的函数, 记为 $f'(n-1, m)$ 。最初序列最后剩下的数字 $f(n, m)$ 一定是删除一个数字之后的序列最后剩下的数字, 即 $f(n, m) = f'(n-1, m)$ 。

接下来我们把剩下的这 $n-1$ 个数字的序列 $k+1, \dots, n-1, 0, 1, \dots, k-1$ 做

一个映射, 映射的结果是形成一个从 0 到 $n-2$ 的序列:

$$\begin{array}{rcl} k+1 & \rightarrow & 0 \\ k+2 & \rightarrow & 1 \\ \cdots & & \\ n-1 & \rightarrow & n-k-2 \\ 0 & \rightarrow & n-k-1 \\ 1 & \rightarrow & n-k \\ \cdots & & \\ k-1 & \rightarrow & n-2 \end{array}$$

我们把映射定义为 p , 则 $p(x) = (x - k - 1) \% n$ 。它表示如果映射前的数字是 x , 那么映射后的数字是 $(x - k - 1) \% n$ 。该映射的逆映射是 $p^{-1}(x) = (x + k + 1) \% n$ 。

由于映射之后的序列和最初的序列具有同样的形式, 即都是从 0 开始的连续序列, 因此仍然可以用函数 f 来表示, 记为 $f(n-1, m)$ 。根据我们的映射规则, 映射之前的序列中最后剩下的数字 $f'(n-1, m) = p^{-1}[f(n-1, m)] = [f(n-1, m) + k + 1] \% n$, 把 $k = (m-1)\%n$ 代入得到 $f(n, m) = f'(n-1, m) = [f(n-1, m) + m] \% n$ 。

经过上面复杂的分析, 我们终于找到了一个递归公式。要得到 n 个数字的序列中最后剩下的数字, 只需要得到 $n-1$ 个数字的序列中最后剩下的数字, 并以此类推。当 $n=1$ 时, 也就是序列中开始只有一个数字 0, 那么很显然最后剩下的数字就是 0。我们把这种关系表示为:

$$f(n, m) = \begin{cases} 0 & n=1 \\ [f(n-1, m) + m] \% n & n>1 \end{cases}$$

这个公式无论用递归还是用循环, 都很容易实现。下面是一段基于循

```
# 迭代剪掉第m个数字
class Solution:
    def LastRemaining_Solution(self, n, m):
        if not n or n <= 0:
            return -1
        tmp = [x for x in range(n)]
        while len(tmp) > 1:
            a = m%len(tmp)
            if a == 0:
                tmp.pop()
            else:
```

```

        tmp = tmp[a:] + tmp[:a-1]
    return tmp[0]
# 动态规划
class Solution:
    def LastRemaining_Solution(self, n, m):
        if n < 1 or m < 1:
            return -1
        last = 0
        for i in range(2, n+1):
            last = (last+m)%i
        return last

```

【发散思维能力】47. 求 $1+2+3+\dots+n$

求 $1+2+3+\dots+n$, 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）。

题解

```

# 实际上是循环
return sum(range(n+1))
# 短路求值
## python中逻辑运算符的用法, a and b, a为False, 返回a, a为True, 就返回b
return n and n + self.Sum_Solution(n-1)

```

48. 不用加减乘除做加法

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。

题解

- 二进制位运算

解析一

首先补充知识，二进制算法是用补码计算的！补码！补码！重要的事情三遍！

首先正数举例， $5+6$:

$\text{bin}(5)=0101, \text{bin}(6)=0110$

首先我们在计算十进制的时候思路是这样的， $5+6=11$ ，首先看1，之后发现需要左移也就是所谓的进1，变成11，二进制是类似的；

第一次， $0101 \wedge 0110 = 0011$

$0101 \& 0111 = 0100$ ，发现 $0100 \neq 0000$ ，所以是需要进位左移 $0100 \ll 1 = 1000$

第二次， $0011 \wedge 1000 = 1011$

$0011 \& 1000 = 0000$ ，发现 $0000 == 0000$ ，所以返回1011，也就是11

之后负数举例， $5-6$: (以8位举例)

$\text{bin}(5)=00000101, \text{bin}(-6)=10000110$, 反码:11111001, 补码:11111010

开始计算：

第一次， $00000101 \wedge 11111010 = 11111111$

$00000101 \& 11111010 = 00000000 == 00000000$ ，注意11111111是补码，需要转换回去, 反码11111110，源码10000001=-1, $\text{num}1-\text{pow}(2,32)$ 作用就是类似 $255-256=-1$

解析二

让我们实现两数相加，但是不能用加号或者其他什么数学运算符号，那么我们只能回归计算机运算的本质，位操作 Bit Manipulation，我们在做加法运算的时候，每位相加之后可能会有进位 Carry 产生，然后在下一位计算时需要加上进位一起运算，那么我们能不能将两部分拆开呢，我们来看一个例子 $759+674$

1. 如果我们不考虑进位，可以得到 323
2. 如果我们只考虑进位，可以得到 1110
3. 我们把上面两个数字假期 $323+1110=1433$ 就是最终结果了

然后我们进一步分析，如果得到上面的第一第二种情况，我们在二进制下来看，不考虑进位的加， $0+0=0$ ， $0+1=1$, $1+0=1$, $1+1=0$ ，这就是异或的运算规则，如果只考虑进位的加 $0+0=0$, $0+1=0$, $1+0=0$, $1+1=1$ ，而这其实这就是'与'的运算，而第三步在将两者相加时，我们再递归调用这个算法，终止条件是当进位为0时，直接返回第一步的结果。一切都是如此的美好，突然有一天，博主的所有方法都无法通过 OJ 了，不知为何，原因不明。在热心网友 [GGGGITFK](#) 的提示下，终于知道了错误的原因：

runtime error: left shift of negative value -2147483648，对INT_MIN左移位。

就是 LeetCode 自己的编译器比较 strict，不能对负数进行左移，就是说最高位符号位必须要为0，才能左移（此处应有尼克杨问号脸？！），好吧，你赢了。那么我们在a和b相'与'之后，再'与'上一个最高位为0，其余位都为1的数 $0x7fffffff$ ，这样可以强制将最高位清零，然后再进行左移。

```
# 垃圾版，只能操作正整数
class Solution:
    def Add(self, num1, num2):
        num1,num2 = min(num1,num2),max(num1,num2)
        num1 = bin(num1)[2:][::-1]
        num2 = bin(num2)[2:][::-1]
        for _ in range(len(num2)-len(num1)):
            num1+= '0'
        res = []
        tmp = 0
        for i in range(len(num2)):
            if num1[i]=='0' and num2[i]=='0' and tmp==0:
                res.append('0')
            elif num1[i]=='0' and num2[i]=='0' and tmp==1:
                res.append('1')
                tmp = 0
            elif num1[i]=='1' and num2[i]=='1' and tmp==0:
                res.append('0')
                tmp = 1
            elif num1[i]=='1' and num2[i]=='1' and tmp==1:
                res.append('1')
            elif tmp==0:
                res.append('1')
            else:
                res.append('0')
        if tmp==1:
            res.append('1')
        res = int(''.join(res[::-1]),2)
        return res

# 牛逼版
## 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。
class Solution:
    def Add(self, a, b):
        while(b):
            a,b = (a^b) & 0xFFFFFFFF, ((a&b)<<1) & 0xFFFFFFFF
        return a if a<=0x7FFFFFFF else ~(a^0xFFFFFFFF)

# 位运算2
class Solution:
    def Add(self, a, b):
        if b == 0:
            return a
        sum = a^b
```

```
carry = (a & b & 0x7fffffff) << 1
return self.Add(sum, carry)
```

49. 把字符串转换成整数

将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数。 数值为0或者字符串不是一个合法的数值则返回0

输入描述:

输入一个字符串,包括数字字母符号,可以为空

输出描述:

如果是合法的数值表达则返回该数字，否则返回0

```
输入
+2147483647
1a33
输出
2147483647
0
```

题解

需要考虑：

- 数据上下溢出
- 空字符串
- 只有正负号
- 有无正负号
- 错误标志输出

```
class Solution:
    def StrToInt(self, s):
        if s is None or len(s) == 0:
            return 0
        res = 0
        start = 0
        flag = True
        if s[0]=='+' or s[0]=='-':
            start += 1
            if s[0]=='-':
                flag = False
        for i in range(start,len(s)):
            if s[i]>='0' and s[i]<='9':
                res = res*10 + (ord(s[i])-ord('0'))
            else:
                return 0
        if flag:
            if res > 0x7FFFFFFF:
                return 0
            else:
                if res > 0x80000000:
                    return 0
                res *= -1
        return res
```

50. 数组中重复的数字

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。

题解

- 用了dict
 - 利用了哈希的特性，但不需要额外的存储空间。因此时间复杂度为O(n)，不需要额外空间
- 把当前序列当成是一个下标和下标对应值是相同的数组
 - 遍历数组，判断当前位置的值和下标是否相等： 2.1. 若相等，则遍历下一位； 2.2. 若不等，则将当前位置i上的元素和a[i]位置上的元素比较：若它们相等，则成功！若不等，则将它们两交换。换完之后a[i]位置上的值和它的下标是对应的，但i位置上的元素和下标并不一定对应；重复2.2的操作，直到当前位置i的值也为i，将i向后移一位，再重复2.

```
# dict
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):
        cDict = {}
        for i in numbers:
            try:
                cDict[i] += 1
                duplication[0] = i
            except:
                cDict[i] = 1
            return True
        except:
            cDict[i] = 1
        return False
# 不用额外空间
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):
        index = 0
        while index < len(numbers):
            if numbers[index] == index:
                index += 1
            elif numbers[index] == numbers[numbers[index]]:
                duplication[0] = numbers[index]
                return True
            else:
                index_2 = numbers[index]
                numbers[index], numbers[index_2] = numbers[index_2], numbers[index]
        return False
```

51. 构建乘积数组

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。（注意：规定B[0] = A[1] * A[2] * ... * A[n-1]，B[n-1] = A[0] * A[1] * ... * A[n-2];）

题解

- 笨拙O(n^2)
- B[i]的值可以看作下图的矩阵中每行的乘积

下三角用连乘可以很容易求得，上三角，从下向上也是连乘。

因此我们的思路就很清晰了，先算下三角中的连乘，即我们先算出B[i]中的一部分，然后反过来按上三角中的分布规律，把另一部分也乘进去。

B_0	1	A_1	A_2	...	A_{n-2}	A_{n-1}
B_1	A_0	1	A_2	...	A_{n-2}	A_{n-1}
B_2	A_0	A_1	1	...	A_{n-2}	A_{n-1}
...	A_0	A_1	...	1	A_{n-2}	A_{n-1}
B_{n-2}	A_0	A_1	...	A_{n-3}	1	A_{n-1}
B_{n-1}	A_0	A_1	...	A_{n-3}	A_{n-2}	1

```
# O(n^2)
class Solution:
    def multiply(self, A):
        B = []
        A.append(1)
        for i in range(len(A)-1):
            tmp = 1
            for j in A[:i]+A[i+1:]:
                tmp *= j
            B.append(tmp)
        return B

# 把每行被1分割的两部分乘积都计算出来，从首尾分别用累乘算出两个列表，两个列表首尾相乘就是B的元素
class Solution:
    def multiply(self, A):
        head = [1]
        tail = [1]
        for i in range(len(A)-1):
            head.append(A[i]*head[-1])
            tail.append(A[-i-1]*tail[-1])
        return [head[j]*tail[-j-1] for j in range(len(head))]
```

52. 正则表达式匹配

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"abaca"匹配，但是与"aa.a"和"ab*a"均不匹配

题解

```
class Solution:
    # s, pattern都是字符串
    def match(self, s, pattern):
        # 如果s与pattern都为空，则True
        if len(s) == 0 and len(pattern) == 0:
            return True
        # 如果s不为空，而pattern为空，则False
        elif len(s) != 0 and len(pattern) == 0:
            return False
        # 如果s为空，而pattern不为空，则需要判断
        elif len(s) == 0 and len(pattern) != 0:
            # pattern中的第二个字符为*，则pattern后移两位继续比较
            if len(pattern) > 1 and pattern[1] == '*':
```

```

        return self.match(s, pattern[2:])
    else:
        return False
# s与pattern都不为空的情况
else:
    # pattern的第二个字符为*的情况
    if len(pattern) > 1 and pattern[1] == '*':
        # s与pattern的第一个元素不同，则s不变，pattern后移两位，相当于pattern前两位当成空
        if s[0] != pattern[0] and pattern[0] != '.':
            return self.match(s, pattern[2:])
        else:
            # 如果s[0]与pattern[0]相同，且pattern[1]为*，这个时候有三种情况
            # pattern后移2个，s不变；相当于把pattern前两位当成空，匹配后面的
            # pattern后移2个，s后移1个；相当于pattern前两位与s[0]匹配
            # pattern不变，s后移1个；相当于pattern前两位，与s中的多位进行匹配，因为*可以匹配多位
            return self.match(s, pattern[2:]) or self.match(s[1:], pattern[2:]) or self.match(s[1:], pattern)
    # pattern第二个字符不为*的情况
    else:
        if s[0] == pattern[0] or pattern[0] == '.':
            return self.match(s[1:], pattern[1:])
        else:
            return False

```

53. 表示数值的字符串

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。

题解

```

class Solution:
    # s字符串
    def isNumeric(self, s):
        if s is None or len(s) == 0:
            return False
        curr = 0
        e = 0
        d = 0
        if s[0] == '+' or s[0] == '-':
            if len(s) == 1:
                return False
            curr = 1
        while curr < len(s):
            if '0' <= s[curr] and s[curr] <= '9':
                curr += 1
            elif s[curr] in ['e', 'E']:
                e += 1
                if curr == len(s)-1 or e > 1:
                    return False
                if s[curr+1] in ['-','+']:
                    if curr == len(s)-2:
                        return False
                    curr += 1
                    curr += 1
                else:
                    return False
            elif s[curr] == '.':
                d += 1
                if e == 1 or d > 1 or curr == len(s)-1:
                    return False
                if s[curr+1] in ['+', '-','E','e']:
                    curr += 1
                else:
                    return False
        return True

```

54. 字符流中第一个不重复的字符

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。

题解

- 字典
- 利用一个int型数组表示256个字符（时间复杂度O(1)，空间复杂度O(n)）
 1. 利用一个int型数组表示256个字符，这个数组初值置为-1
 2. 每读出一个字符，将该字符的位置存入字符对应数组下标中
 3. 若值为-1表示第一次读入，不为-1且>0表示不是第一次读入，将值改为-2
 4. 之后在数组中找到>0的最小值，该数组下标对应的字符为所求。
 5. 在python中，ord(char)是得到char对应的ASCII码；chr(idx)是得到ASCII位idx的字符

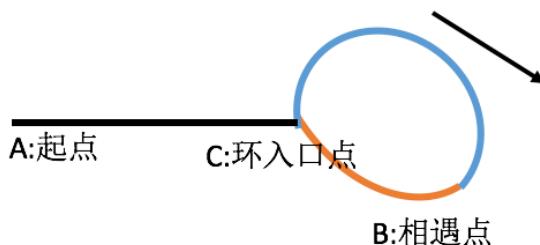
```
# 字典
class Solution:
    def __init__(self):
        self.s = ''
    def FirstAppearingOnce(self):
        if not self.s:
            return '#'
        cDict = dict((x, 0) for x in self.s)
        for i in self.s:
            cDict[i] += 1
        for i in self.s:
            if cDict[i] == 1:
                return i
        return '#'
    def Insert(self, char):
        self.s += char
# int数组
class Solution:
    def __init__(self):
        self.char_list = [-1 for i in range(256)]
        self.index = 0 # 记录当前字符的个数，可以理解为输入的字符串中的下标
    def FirstAppearingOnce(self):
        min_value = 500
        min_idx = -1
        for i in range(256):
            if self.char_list[i] > -1:
                if self.char_list[i] < min_value:
                    min_value = self.char_list[i]
                    min_idx = i
        if min_idx > -1:
            return chr(min_idx)
        else:
            return '#'
    def Insert(self, char):
        # 如果是第一次出现，则将对应元素的值改为下边
        if self.char_list[ord(char)] == -1:
            self.char_list[ord(char)] = self.index
        # 如果已经出现过两次了，则不修改
        elif self.char_list[ord(char)] == -2:
            pass
        # 如果出现过一次，则进行修改，修改为-2
        else:
            self.char_list[ord(char)] = -2
        self.index += 1
```

55. 链表中环的入口结点

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。

题解

- list
- 快慢指针 (时间复杂度: $O(n)$, 空间复杂度: $O(1)$)
 - 1、设置快慢指针，假如有环，他们最后一定相遇。
 - 2、两个指针分别从链表头和相遇点继续出发，每次走一步，最后一定相遇于环入口。



假设x为环前面的路程（黑色路程），a为环入口到相遇点的路程（蓝色路程，假设顺时针走），c为环的长度（蓝色+橙色路程）

当快慢指针相遇的时候：

此时慢指针走的路程为 $S_{slow} = x + m * c + a$

快指针走的路程为 $S_{fast} = x + n * c + a$

$$2 S_{slow} = S_{fast}$$

$$2 * (x + m * c + a) = (x + n * c + a)$$

$$\text{从而可以推导出: } x = (n - 2 * m) * c - a = (n - 2 * m - 1) * c + c - a$$

即环前面的路程 = 数个环的长度 (可能为0) + $c - a$

什么是 $c - a$? 这是相遇点后，环后面部分的路程。（橙色路程）

所以，我们可以让一个指针从起点A开始走，让一个指针从相遇点B开始继续往后走，2个指针速度一样，那么，当从原点的指针走到环入口点的时候（此时刚好走了x）从相遇点开始走的那个指针也一定刚好到达环入口点。所以2者会相遇，且恰好相遇在环的入口点。

```
# list保存nodes
class Solution:
    def EntryNodeOfLoop(self, pHead):
        tempList = []
        p = pHead
        while p:
            if p in tempList:
                return p
            else:
                tempList.append(p)
            p = p.next
# 快慢指针
class Solution:
    def EntryNodeOfLoop(self, pHead):
        slow, fast=pHead, pHead
```

```

        while fast and fast.next:
            slow=slow.next
            fast=fast.next.next
            if slow==fast:
                slow2=pHead
                while slow!=slow2:
                    slow=slow.next
                    slow2=slow2.next
                return slow

```

56. 删除链表中重复的结点

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

题解

- 两个指针，一个在后面记录不重复结点，另一个扫描链表
- 递归

```

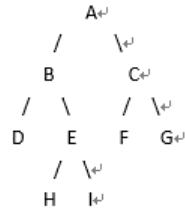
# 双指针，pHead指向newHead开头，newHead扫描，遇到重复就用tmpNode向前判断重复终止
class Solution:
    def deleteDuplication(self, pHead):
        if pHead == None or pHead.next == None:
            return pHead
        newHead = ListNode(None)
        newHead.next = pHead
        pHead = newHead
        while newHead.next and newHead.next.next:
            if newHead.next.val == newHead.next.next.val:
                tmpNode = newHead.next.next
                while tmpNode and tmpNode.val == newHead.next.next.val:
                    tmpNode = tmpNode.next
                # newHead下一个指向了新数值
                newHead.next = tmpNode
            else:
                newHead = newHead.next
        return pHead.next
# 递归
class Solution:
    def deleteDuplication(self, pHead):
        if pHead is None or pHead.next is None:
            return pHead
        head1 = pHead.next
        if head1.val != pHead.val:
            pHead.next = self.deleteDuplication(pHead.next)
        else:
            while pHead.val == head1.val and head1.next is not None:
                head1 = head1.next
            if head1.val != pHead.val:
                pHead = self.deleteDuplication(head1)
            else:
                return None
        return pHead

```

57. 二叉树的下一个结点

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。

题解



1. 若该节点存在右子树：则下一个节点为右子树最左子节点（如图节点 B）

2. 若该节点不存在右子树：这时分两种情况：

2.1 该节点为父节点的左子节点，则下一个节点为其父节点（如图节点 D）

2.2 该节点为父节点的右子节点，则沿着父节点向上遍历，知道找到一个节点的父节点的左子

节点为该节点，则该节点的父节点下一个节点（如图节点 I，沿着父节点一直向上查找找到 B（B 为其父节点的左子节点），则 B 的父节点 A 为下一个节点）。

```

# class TreeLinkNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
#         self.next = None
# 可将tmp全部替换为pNode，空间复杂度更低
class Solution:
    def GetNext(self, pNode):
        if pNode == None:
            return None
        if pNode.right != None:
            tmp = pNode.right
            while tmp.left != None:
                tmp = tmp.left
            return tmp
        else:
            if pNode.next != None and pNode == pNode.next.left:
                return pNode.next
            elif pNode.next != None and pNode == pNode.next.right:
                tmp = pNode.next
                while tmp.next != None and tmp == tmp.next.right:
                    tmp = tmp.next
                return tmp.next
            else:
                return pNode.next

```

58. 对称的二叉树

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。

题解

- 递归比较

```

class Solution:
    def isSymmetrical(self, pRoot):
        if not pRoot:
            return True
        return self.compare(pRoot.left, pRoot.right)
    def compare(self, pRoot1, pRoot2):
        if not pRoot1 and not pRoot2:
            return True

```

```

if not pRoot1 or not pRoot2:
    return False
if pRoot1.val == pRoot2.val:
    if self.compare(pRoot1.left, pRoot2.right) and self.compare(pRoot1.right, pRoot2.left):
        return True
return False

```

59. 按之字形顺序打印二叉树

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。

题解

- 从左到右存结点，偶数行reverse
- 一个队列，一个栈

```

# 从左到右存结点
class Solution:
    def Print(self, pRoot):
        if not pRoot:
            return []
        nodeStack = [pRoot]
        res = []
        # 存结点，统一从左向右存
        while nodeStack:
            tmp = []
            nextStack = []
            for i in nodeStack:
                tmp.append(i.val)
                if i.left:
                    nextStack.append(i.left)
                if i.right:
                    nextStack.append(i.right)
            nodeStack = nextStack
            res.append(tmp)
        # 遍历res，每一个元素为二叉树一层的结点值（从左到右）
        result = []
        for i, v in enumerate(res):
            if i%2 == 0:
                result.append(v)
            else:
                result.append(v[::-1])
        return result

```

60. 把二叉树打印成多行

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。

题解

- 上题代码
- 递归

```

# 数组存值
class Solution:
    # 返回二维列表[[1,2],[4,5]]
    def Print(self, pRoot):
        if not pRoot:
            return []
        nextNode = [pRoot]
        res = []

```

```

while nextNode:
    tmp = []
    tValue = []
    for i in nextNode:
        tValue.append(i.val)
        if i.left:
            tmp.append(i.left)
        if i.right:
            tmp.append(i.right)
    nextNode = tmp
    res.append(tValue)
return res

# 递归
class Solution:
    # 返回二维列表[[1,2],[4,5]]
    def Print(self, pRoot):
        if not pRoot:
            return []
        def depth(node, d):
            if node:
                if d > len(res):
                    res.append([])
                res[d-1].append(node.val)
                depth(node.left, d+1)
                depth(node.right, d+1)
        res = []
        depth(pRoot, 1)
        return res

```

61. 序列化二叉树

请实现两个函数，分别用来序列化和反序列化二叉树

二叉树的序列化是指：把一棵二叉树按照某种遍历方式的结果以某种格式保存为字符串，从而使得内存中建立起来的二叉树可以持久保存。序列化可以基于先序、中序、后序、层序的二叉树遍历方式来进行修改，序列化的结果是一个字符串，序列化时通过某种符号表示空节点（#），以！表示一个结点值的结束（value!）。

二叉树的反序列化是指：根据某种遍历顺序得到的序列化字符串结果str，重构二叉树。

题解

```

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):
        l = s.split(',')
        self.flag += 1
        # 判断字符串是否读完
        if self.flag >= len(s):
            return None
        node = None
        # 构建结点
        if l[self.flag] != '#':
            node = TreeNode(int(l[self.flag]))
            node.left = self.Deserialize(s)
            node.right = self.Deserialize(s)
        return node

```

62. 二叉搜索树的第k个结点

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。

题解

- 中序遍历后找结点，空间复杂度高

```
class Solution:  
    # 返回对应节点TreeNode  
    def KthNode(self, pRoot, k):  
        if not pRoot:  
            return None  
        if k <= 0:  
            return None  
        def recurTree(node):  
            if node:  
                recurTree(node.left)  
                l.append(node)  
                recurTree(node.right)  
        l = []  
        recurTree(pRoot)  
        if k > len(l):  
            return None  
        return l[k-1]
```

63. 数据流中的中位数

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。

题解

一个小顶堆和大顶堆

大顶堆用来存较小的数，从大到小排列；小顶堆存较大的数，从小到大的顺序排序

- 保证：小顶堆中的元素都大于等于大顶堆中的元素，所以每次塞值，并不是直接塞进去，而是从另一个堆中poll出一个最大（最小）的塞值
- 当数目为偶数的时候，将这个值插入大顶堆中，再将大顶堆中根节点（即最大值）插入到小顶堆中；
- 当数目为奇数的时候，将这个值插入小顶堆中，再讲小顶堆中根节点（即最小值）插入到大顶堆中；
- 取中位数的时候，如果当前个数为偶数，显然是取小顶堆和大顶堆根结点的平均值；如果当前个数为奇数，显然是取小顶堆的根节点

例如，传入的数据为：[5,2,3,4,1,6,7,0,8]，输出是"5.00 3.50 3.00 3.50 3.00 3.50 4.00 3.50 4.00 "

那么整个程序的执行流程应该是（用min表示小顶堆，max表示大顶堆）：

- 5先进入大顶堆，然后将大顶堆中最大值放入小顶堆中，min=[5], max=[None], med=5.00
- 2先进入小顶堆，然后将小顶堆中最小值放入大顶堆中，min=[5], max=[2], med=(5+2)/2=3.50
- 3先进入大顶堆，然后将大顶堆中最大值放入小顶堆中，min=[3,5], max=[2], med=3.00
- 4先进入小顶堆，然后将小顶堆中最小值放入大顶堆中，min=[4,5], max=[3,2], med=(4+3)/2=3.50
- 1先进入大顶堆，然后将大顶堆中最大值放入小顶堆中，min=[3,4,5], max=[2,1], med=3.00
- 6先进入小顶堆，然后将小顶堆中最小值放入大顶堆中，min=[4,5,6], max=[3,2,1], med=(4+3)/2=3.50
- 7先进入大顶堆，然后将大顶堆中最大值放入小顶堆中，min=[4,5,6,7], max=[3,2,1], med=4.00

- 0先进入小顶堆，然后将小顶堆中最大值放入小顶堆中，min=[4,5,6,7], max=[3,2,1,0], med=(4+3)/2=3.50
- 8先进入大顶堆，然后将大顶堆中最小值放入大顶堆中，min=[4,5,6,7,8], max=[3,2,1,0], med=4.00

```

class Solution:
    def __init__(self):
        self.minNums=[]
        self.maxNums=[]

    def maxHeapInsert(self,num):
        self.maxNums.append(num)
        lens = len(self.maxNums)
        i = lens - 1
        while i > 0:
            if self.maxNums[i] > self.maxNums[(i - 1) / 2]:
                t = self.maxNums[(i - 1) / 2]
                self.maxNums[(i - 1) / 2] = self.maxNums[i]
                self.maxNums[i] = t
                i = (i - 1) / 2
            else:
                break

    def maxHeapPop(self):
        t = self.maxNums[0]
        self.maxNums[0] = self.maxNums[-1]
        self.maxNums.pop()
        lens = len(self.maxNums)
        i = 0
        while 2 * i + 1 < lens:
            nexti = 2 * i + 1
            if (nexti + 1 < lens) and self.maxNums[nexti + 1] > self.maxNums[nexti]:
                nexti += 1
            if self.maxNums[nexti] > self.maxNums[i]:
                tmp = self.maxNums[i]
                self.maxNums[i] = self.maxNums[nexti]
                self.maxNums[nexti] = tmp
                i = nexti
            else:
                break
        return t

    def minHeapInsert(self,num):
        self.minNums.append(num)
        lens = len(self.minNums)
        i = lens - 1
        while i > 0:
            if self.minNums[i] < self.minNums[(i - 1) / 2]:
                t = self.minNums[(i - 1) / 2]
                self.minNums[(i - 1) / 2] = self.minNums[i]
                self.minNums[i] = t
                i = (i - 1) / 2
            else:
                break

    def minHeapPop(self):
        t = self.minNums[0]
        self.minNums[0] = self.minNums[-1]
        self.minNums.pop()
        lens = len(self.minNums)
        i = 0
        while 2 * i + 1 < lens:
            nexti = 2 * i + 1
            if (nexti + 1 < lens) and self.minNums[nexti + 1] < self.minNums[nexti]:
                nexti += 1
            if self.minNums[nexti] < self.minNums[i]:
                tmp = self.minNums[i]
                self.minNums[i] = self.minNums[nexti]
                self.minNums[nexti] = tmp
                i = nexti
            else:
                break

```

```

        break
    return t

def Insert(self, num):
    if (len(self.minNums)+len(self.maxNums))&1 == 0:
        if len(self.maxNums)>0 and num < self.maxNums[0]:
            self.maxHeapInsert(num)
            num = self.maxHeapPop()
            self.minHeapInsert(num)
    else:
        if len(self.minNums)>0 and num > self.minNums[0]:
            self.minHeapInsert(num)
            num = self.minHeapPop()
            self.maxHeapInsert(num)

def GetMedian(self):
    allLen = len(self.minNums) + len(self.maxNums)
    if allLen == 0:
        return -1
    if allLen&1 == 1:
        return self.minNums[0]
    else:
        return (self.maxNums[0] + self.minNums[0])/2.0

```

64. 滑动窗口的最大值

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{4,4,6,6,5}；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]}。

题解

- max, 时间复杂度O(n*size)
- 双向队列，queue存入num的位置，时间复杂度O(n)

用一个双端队列，队列第一个位置保存当前窗口的最大值，当窗口滑动一次

- 1.判断当前最大值是否过期
- 2.新增加的值从队尾开始比较，把所有比他小的值丢掉

```

# max
class Solution:
    def maxInWindows(self, num, size):
        if not num or size <= 0:
            return []
        n = len(num)
        if size > n:
            return []
        res = []
        for i in range(n-size+1):
            tmp = num[i:i+size]
            res.append(max(tmp))
        return res

# 双向队列
class Solution:
    def maxInWindows(self, num, size):
        queue, res, i = [], [], 0
        while size>0 and i<len(num):
            #若最大值queue[0]位置过期，则弹出
            if len(queue)>0 and i-size+1 > queue[0]:
                queue.pop(0)
            #每次弹出所有比num[i]小的数字
            while len(queue)>0 and num[queue[-1]]<num[i]:
                queue.pop()
            queue.append(i)
            res.append(queue[0])
            i += 1
        return res

```

```

if i>=size-1:
    res.append(num[queue[0]])
    i += 1
return res

```

65. 矩阵中的路径

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。例如

$$\begin{bmatrix} a & b & c & e \\ s & f & c & s \\ a & d & e & e \end{bmatrix}$$

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。

题解

回溯法

```

class Solution:
    def hasPath(self, matrix, rows, cols, path):
        # write code here
        if not matrix:
            return False
        if not path:
            return True
        x = [list(matrix[cols*i:cols*i+cols]) for i in range(rows)]
        for i in range(rows):
            for j in range(cols):
                if self.exist_helper(x, i, j, path):
                    return True
        return False
    def exist_helper(self, matrix, i, j, p):
        if matrix[i][j] == p[0]:
            if not p[1:]:
                return True
            matrix[i][j] = ''
            if i > 0 and self.exist_helper(matrix, i-1, j, p[1:]):
                return True
            if i < len(matrix)-1 and self.exist_helper(matrix, i+1, j, p[1:]):
                return True
            if j > 0 and self.exist_helper(matrix, i, j-1, p[1:]):
                return True
            if j < len(matrix[0])-1 and self.exist_helper(matrix, i, j+1, p[1:]):
                return True
            matrix[i][j] = p[0]
            return False
        else:
            return False
# 不懂错在哪
class Solution():
    def hasPath(self, matrix, rows, cols, path):
        x = [list(matrix[cols*row:cols*row+cols]) for row in range(rows)]
        matrix = x
        position = []
        for i in path:
            tmp = []
            for row in range(rows):
                for col in range(cols):
                    if matrix[row][col] == i:
                        tmp.append([row, col])
            if len(tmp) == 0:

```

```

        return False
    position.append(tmp)
print(position)
flag = False
if self.back(position, [], 0, flag):
    return True
return False
def back(self, position, tmp, step, flag):
    if step == len(position):
        trueCount = 0
        for j in range(len(tmp)-1):
            if tmp[j]==tmp[j+1] or abs(tmp[j][0]-tmp[j+1][0])>1 or abs(tmp[j][1]-tmp[j+1][1])>1:
                break
            else:
                trueCount += 1
        if trueCount == len(tmp)-1:
            return True
    if step < len(position):
        for i in position[step]:
            tmp.append(i)
            self.back(position, tmp, step+1, flag)
            tmp.pop()

```

66. 机器人的运动范围

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？

题解

```

# 递归
class Solution:
    def movingCount(self, threshold, rows, cols):
        if threshold < 0:
            return 0
        record = []
        def move(row, col):
            record.append([row, col])
            def sumNumber(n):
                return sum(map(int,[i for i in str(n)]))
            # left
            if col>0 and sumNumber(row)+sumNumber(col-1)<=threshold:
                if [row, col-1] not in record:
                    move(row, col-1)
            # right
            if col<cols-1 and sumNumber(row)+sumNumber(col+1)<=threshold:
                if [row, col+1] not in record:
                    move(row, col+1)
            # up
            if row>0 and sumNumber(row-1)+sumNumber(col)<=threshold:
                if [row-1, col] not in record:
                    move(row-1, col)
            # down
            if row<rows-1 and sumNumber(row+1)+sumNumber(col)<=threshold:
                if [row+1, col] not in record:
                    move(row+1, col)
        move(0, 0)
        return len(record)
# 递归精简版
class Solution:
    def __init__(self):
        self.count = 0

    def movingCount(self, threshold, rows, cols):
        # write code here
        arr = [[1 for i in range(cols)] for j in range(rows)]

```

```

        self.findway(arr, 0, 0, threshold)
        return self.count

    def findway(self, arr, i, j, k):
        if i < 0 or j < 0 or i >= len(arr) or j >= len(arr[0]):
            return
        tmpi = list(map(int, list(str(i))))
        tmpj = list(map(int, list(str(j))))
        if sum(tmpi) + sum(tmpj) > k or arr[i][j] != 1:
            return
        arr[i][j] = 0
        self.count += 1
        self.findway(arr, i + 1, j, k)
        self.findway(arr, i - 1, j, k)
        self.findway(arr, i, j + 1, k)
        self.findway(arr, i, j - 1, k)

```

67. 剪绳子

给你一根长度为n的绳子，请把绳子剪成整数长的m段（m、n都是整数，n>1并且m>1），每段绳子的长度记为k[0],k[1],...,k[m]。请问k[0]xk[1]x...xk[m]可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

输入描述:

输入一个数n，意义见题面。（ $2 \leq n \leq 60$ ）

示例

输入：8

输出：18

题解

- 动态规划

问题的最优解一定是由子问题的最优解构成：

如果一个长度为n的绳子可以分为 $n = s[1] + s[2] + s[3] + s[4]$ 这么几个段，那么一定有 $\max = s[1]*s[2]*s[3]*s[4]$ ，且 $s[1] * s[2]$ 为长度是 $s[1]+s[2]$ 的子问题的最优解，同理 $s[3]*s[4]$ 一定是 $s[3]+s[4]$ 的最优解，所以我们可以知道 n 其实可以表示为上述两段的最优解。

当绳子长度大于等于4的时候，不管对绳子割几刀，总可以把问题看成，先割一刀（长度在1-n）之间，考虑对称性，第一刀范围在1-n/2之间

- 贪婪算法、乘方运算

先举几个例子，可以看出规律来。

4 : 2*2

5 : 2*3

6 : 3*3

7 : 2*2*3 或者4*3

8 : 2*3*3

9 : 3*3*3

10: 2*2*3*3 或者4*3*3

11: 2*3*3*3

12: 3*3*3*3

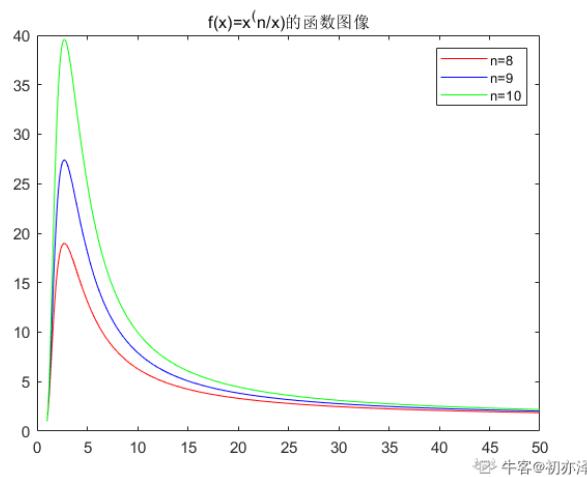
13: 2*2*3*3*3 或者4*3*3*3

下面是分析：

首先判断 $k[0]$ 到 $k[m]$ 可能有哪些数字，实际上只可能是2或者3。
当然也可能有4，但是 $4=2*2$ ，我们就简单些不考虑了。
 $5 < 2*3, 6 < 3*3$, 比6更大的数字我们就更不用考虑了，肯定要继续分。
其次看2和3的数量，2的数量肯定小于3个，为什么呢？因为 $2*2*2 < 3*3$ ，那么题目就简单了。
直接用n除以3，根据得到的余数判断是一个2还是两个2还是没有2就行了。
由于题目规定 $m > 1$ ，所以2只能是 $1*1$ ，3只能是 $2*1$ ，这两个特殊情况直接返回就行了。

乘方运算的复杂度为： $O(\log n)$ ，用动态规划来做会耗时比较多。

$f(x) = (x)^{n/x}$ ，最大时x为e也即2.718。取整的话 $f(3) > f(2)$



```
# 动态规划
class Solution:
    def cutRope(self, number):
        # 给定初始值
        if number<=1:
            return 0
        elif number<=2:
            return 1
        elif number<=3:
            return 2
        # 定义最大值数组
        prod=[0,1,2,3]
        for i in range(4,number+1):
            # max 初始化为 一分的结果(本身)
            max=0
            # 对于最少二分做解
            for j in range(1,i//2+1):
                pro=prod[j]*prod[i-j]
                if pro>max:
                    max=pro
            prod.append(max)
        return prod[number]

# 贪婪算法
class Solution:
    def cutRope(self, number):
        res=1
        if number<=1:
            return 0
        elif number<=2:
            return 1
        elif number<=3:
```

```
    return 2
elif number>3:
    if number%3==0:
        res=3** (number//3)
    elif number%3==1:
        res=3** (number//3-1)*4
    else:
        res=3** (number//3)*(number%3)
return res
```