今天你刷题了吗

OJ平台

常见术语

简写	全称	中文
OJ	Online Judge	在线判题系统
AC	Accepted	通过
WA	Wrong Answer	答案错误
TLE	Time Limit Exceed	超时
OLE	Output Limit Exceed	超过输出限制
MLE	Memory Limit Exceed	超内存
RE	Runtime Error	运行时错误
PE	Presentation Error	格式错误
CE	Compile Error	无法编译

输入问题

牛客网的OJ答题平台，需要自己写输入部分。

```
arr = input("");     #输入一个一维数组，每个数之间使空格隔开
num = [int(n) for n in arr.split()];    #将输入每个数以空格键隔开做成数组
print(num[0]+num[1]);         #打印数组
# encoding: utf-8
'''
Python的输入是野生字符串，所以要自己转类型
strip去掉左右两端的空白符，返回str
slipt把字符串按空白符拆开，返回[str]
map把list里面的值映射到指定类型，返回[type]
EOF用抓异常
print后面加逗号就不会换行，否则反之，当然3.x君自行传参
题目细节没看太细，可能有的地方不对，不要在意这些细节啦
これは以上です'''

# 有多组输入数据，但没有具体的告诉你有多少组，只是让你对应每组输入，应该怎样输出。

while True:
    try:
        a, b = map(int, raw_input().strip().split())
        print
        a + b,
    except EOFError:
        break
```

```
# 输入一个整数，告诉我们接下来有多少组数据，然后在输入每组数据的具体值。

tcase = int(raw_input().strip())
for case in range(tcase):
    a, b = map(int, raw_input().strip().split())
    print
    a + b,

# 有多组输入数据，没有具体的告诉你有多少组,但是题目却告诉你遇见什么结束

while True:
    a, b = map(int, raw_input().strip().split())
    if a == 0 and b == 0:
        break
    print
    a + b,

# 输入有多组，并却题目告诉你每组输入遇见什么结束，与第三种不同之处在于，每组输入都有相应的细化。

tcase = int(raw_input().strip())
for case in range(tcase):
    a, b = map(int, raw_input().strip().split())
    if a == 0 and b == 0:
        break
    print
    a + b,

# 这次的输入实现输入一个整数，告诉我们有多少行，在输入每一行。对于每一行的输入，有划分为第一个数和其他的数，第一个数代表那一组数据一共有多少输入。

tcase = int(raw_input().strip())
for case in range(tcase):
    data = map(int, raw_input().strip().split())
    n, array = data[0], data[1:]

    sum = 0
    for i in range(n):
        sum += array[i]
    print
    sum,

# 有多种输入数据，对于每组输入数据的第一个数代表该组数据接下来要输入数据量

while True:
    try:
        data = map(int, raw_input().strip().split())
        n, array = data[0], data[1:]

        sum = 0
        for i in range(n):
            sum += array[i]
        print
        sum,
    except EOFError:
        raise

# 这道题的输出只是简单的在每组输出后边多加一个换行而已！
```

```
while True:
    try:
        a, b = map(int, raw_input().strip().split())
        print
        a + b
    except EOFError:
        break

# 这种类型的输出注意的就是换行，这类题目说在输出样例中，每组样例之间有什么什么，所以我们在对应输出的同时要判断一下是否是最后一组输出，如果不是，就 将题目所说的东西输出（一般是换行或空格），如果是，就直接结束。

while True:
    data = raw_input().strip()
    if data.isspace():
        break
    else:
        data = map(int, data)
        n, array = data[0], data[1:]

        sum = 0
        for i in range(n):
            sum += array[i]
        print
        sum,
```

数组

1.两数之和

1. 两数之和

给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那 **两个** 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

示例:

```
给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

```
class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        '''
        #1.O(n^2)
        if not nums:
            return None
        n=len(nums)
        for i in range(n):
```

```
            for j in range(i+1,n):
                if nums[j]==target-nums[i]:
                    return [i,j]
        '''
        #2.哈希表 O(n),空间O(n)
        hashmap={}
        for i in range(len(nums)):
            dif=target-nums[i]
            if hashmap.get(dif)!=None and hashmap.get(dif)!=i:
                return [hashmap.get(dif),i]
            hashmap[nums[i]]=i
```

[167. 两数之和 II - 输入有序数组]

给定一个已按照*升序排列*的有序数组，找到两个数使得它们相加之和等于目标数。

函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

说明:

- 返回的下标值（index1 和 index2）不是从零开始的。
- 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

示例:

```
输入: numbers = [2, 7, 11, 15], target = 9
输出: [1,2]
解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。
```

```
class Solution(object):
    def twoSum(self, numbers, target):
        """
        :type numbers: List[int]
        :type target: int
        :rtype: List[int]
        """
        #双指针法：时间效率O(n),空间效率O(1)
        p1=0
        p2=len(numbers)-1
        while p1<=p2:
            if numbers[p1]+numbers[p2]==target:
                return [p1+1,p2+1]
            elif numbers[p1]+numbers[p2]<target:
                p1=p1+1
            if numbers[p1]+numbers[p2]>target:
                p2=p2-1
        return None
```

[170. 两数之和 III - 数据结构设计]

设计并实现一个 TwoSum 的类，使该类需要支持 `add` 和 `find` 的操作。

`add` 操作 - 对内部数据结构增加一个数。
`find` 操作 - 寻找内部数据结构中是否存在一对整数，使得两数之和与给定的数相等。

示例 1:

```
add(1); add(3); add(5);
find(4) -> true
find(7) -> false
```

示例 2:

```
add(3); add(1); add(2);
find(3) -> true
```

```
class TwoSum(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.nums=[]
        self.dict={}

    def add(self, number):
        """
        Add the number to an internal data structure..
        :type number: int
        :rtype: None
        """
        self.nums.append(number)
        self.dict[number]=len(self.nums)-1


    def find(self, value):
        """
        Find if there exists any pair of numbers which sum is equal to the
value.
        :type value: int
        :rtype: bool
        """
        for i in range(len(self.nums)):
            k=value-self.nums[i]
            if k in self.dict and self.dict[k]!=i:
                return True
        return False



# Your TwoSum object will be instantiated and called as such:
# obj = TwoSum()
# obj.add(number)
# param_2 = obj.find(value)
```

653. 两数之和 IV - 输入 BST

给定一个二叉搜索树和一个目标结果，如果 BST 中存在两个元素且它们的和等于给定的目标结果，则返回 true。

案例 1:

输入:

```
    5
   / \
  3   6
 / \   \
2   4   7

Target = 9
```

输出: True

案例 2:

输入:

```
    5
   / \
  3   6
 / \   \
2   4   7

Target = 28
```

输出: False

```python
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def findTarget(self, root, k):
        """
        :type root: TreeNode
        :type k: int
        :rtype: bool
        """
        #二叉搜索树，中序遍历得到有序数组 O(n)
        path=[]
        self.inorder(root,path)
        p1=0
        p2=len(path)-1
        while p1<p2:
            if path[p1]+path[p2]==k:
                return True
            elif path[p1]+path[p2]<k:
                p1=p1+1
            else:
                p2=p2-1
        return False

    def inorder(self,root,path):
        if not root:
```

```
            return
        self.inorder(root.left,path)
        path.append(root.val)
        self.inorder(root.right,path)
```

2.寻找重复数

287. 寻找重复数

给定一个包含 *n* + 1 个整数的数组 *nums*，其数字都在 1 到 *n* 之间（包括 1 和 *n*），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

示例 1:

```
输入: [1,3,4,2,2]
输出: 2
```

示例 2:

```
输入: [3,1,3,4,2]
输出: 3
```

说明:

1. **不能**更改原数组（假设数组是只读的）。
2. 只能使用额外的 *O*(1) 的空间。
3. 时间复杂度小于 $O(n^2)$ 。
4. 数组中只有一个重复的数字，但它可能不止重复出现一次。

思路：重点在于二分查找值，统计个数

```
class Solution(object):
    def findDuplicate(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #类二分查找，O(nlogn)，二分的不是数组下标，而是数值，比较的是数值个数
        start=1
        end=len(nums)-1
        while start<=end:
            mid=(start+end)//2
            count_o=self.countRange(nums,start,mid)
            if start==end:
                if count_o>1:
                    return start
                else:
                    break
            if count_o>(mid-start)+1:
                end=mid
            else:
                start=mid+1


    def countRange(self,nums,start,end):
        if not nums:
            return 0
        count=0
        for i in range(len(nums)):
            if nums[i]>=start and nums[i]<=end:
```

```
                count+=1
        return count
```

3.删除重复数字

[26. 删除排序数组中的重复项](#)

给定一个排序数组，你需要在**原地**删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在**原地修改输入数组**并在使用 O(1) 额外空间的条件下完成。

示例 1:

```
给定数组 nums = [1,1,2],

函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。

你不需要考虑数组中超出新长度后面的元素。
```

示例 2:

```
给定 nums = [0,0,1,1,1,2,2,3,3,4],

函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。

你不需要考虑数组中超出新长度后面的元素。
```

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以**“引用”**方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

```
class Solution(object):
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #双指针
        if nums==None:
            return 0
        if len(nums)==1:
            return 1

        p1=1 #快指针
        p2=0 #慢指针
        while(p1<len(nums)):     #快指针的边界
```

```
            if nums[p2]!=nums[p1]: #判断**不相等继续
                p2=p2+1
                nums[p2]=nums[p1] #***精华--直接删除
            p1=p1+1                #相等就移动快指针
        return p2+1
```

80. 删除排序数组中的重复项 II

给定一个排序数组，你需要在**原地**删除重复出现的元素，使得每个元素最多出现两次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在**原地修改输入数组**并在使用 O(1) 额外空间的条件下完成。

示例 1:

```
给定 nums = [1,1,1,2,2,3],

函数应返回新长度 length = 5, 并且原数组的前五个元素被修改为 1, 1, 2, 2, 3 。

你不需要考虑数组中超出新长度后面的元素。
```

示例 2:

```
给定 nums = [0,0,1,1,1,1,2,3,3],

函数应返回新长度 length = 7, 并且原数组的前五个元素被修改为 0, 0, 1, 1, 2, 3, 3 。

你不需要考虑数组中超出新长度后面的元素。
```

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以**“引用”**方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中**该长度范围内**的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

```
class Solution(object):
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #双指针
        if len(nums)<=2:
            return len(nums)
        p1=2        #前两个一定会保留
        p2=2
        while p1<len(nums):
            if nums[p1]!=nums[p2-2]:
                nums[p2]=nums[p1]
                p2+=1
            p1+=1
        return p2
```

总结：保留k个重复数字

```
class Solution(object):

    # 模板写法

    def removeDuplicates(self, nums, k):
        """
        :type nums: List[int]
        :rtype: int
        """
        size = len(nums)
        if size <= k:
            return size
        # counter 表示下一个要覆盖的索引
        counter = k
        # 索引为 0 和 1 的数一定会被保留，因此遍历从索引 k 开始
        for i in range(k, size):
            if nums[i] != nums[counter - k]:
                nums[counter] = nums[i]
                counter += 1
        return counter
```

4.无重复字符的最长子串

[3. 无重复字符的最长子串]

给定一个字符串，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

输入: "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:

输入: "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

示例 3:

输入: "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
请注意，你的答案必须是 **子串** 的长度，"pwke" 是一个*子序列*，不是子串。

思路：滑动窗口

```
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
```

```
        #滑动窗口,队列
        if len(s)<=1:
            return len(s)
        result=[]
        cur_len=0
        max_len=0
        for i in range(len(s)):
            cur_len+=1
            while s[i] in result:
                result.pop(0)
                cur_len-=1
            result.append(s[i])
            if cur_len>max_len:
                max_len=cur_len
        return max_len
```

5.最短无序连续子数组

581. 最短无序连续子数组

给定一个整数数组，你需要寻找一个**连续的子数组**，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。

你找到的子数组应是**最短**的，请输出它的长度。

示例 1:

```
输入: [2, 6, 4, 8, 10, 9, 15]
输出: 5
解释: 你只需要对 [6, 4, 8, 10, 9] 进行升序排序，那么整个表都会变为升序排序。
```

说明:

1. 输入的数组长度范围在 [1, 10,000]。
2. 输入的数组可能包含**重复**元素 ，所以**升序**的意思是<=。

```
class Solution(object):
    def findUnsortedSubarray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #1.排序，对应找出第一个及最后一个不同索引
        #2.从前往后记录最大索引区间，从后往前记录最小索引区间
        #数组长度[1,10000]
        high=0
        low=1
        maxValue=0
        minValue=0
        for i in range(len(nums)):
            if i==0:
                maxValue=nums[i]
            if nums[i]<maxValue:
                high=i
            maxValue=max(maxValue,nums[i])
        for i in range(len(nums)-1,-1,-1):
            if i==len(nums)-1:
                minValue=nums[i]
            if nums[i]>minValue:
                low=i
            minValue=min(minValue,nums[i])
```

```
        return high-low+1
```

6.一手顺子

[846. 一手顺子]

爱丽丝有一手（hand）由整数数组给定的牌。

现在她想把牌重新排列成组，使得每个组的大小都是 W，且由 W 张连续的牌组成。

如果她可以完成分组就返回 true，否则返回 false。

示例 1：

```
输入：hand = [1,2,3,6,2,3,4,7,8], W = 3
输出：true
解释：爱丽丝的手牌可以被重新排列为 [1,2,3]，[2,3,4]，[6,7,8]。
```

示例 2：

```
输入：hand = [1,2,3,4,5], W = 4
输出：false
解释：爱丽丝的手牌无法被重新排列成几个大小为 4 的组。
```

提示：

1. 1 <= hand.length <= 10000
2. 0 <= hand[i] <= 10^9
3. 1 <= W <= hand.length

思路：

1.首先判断列表元素个数，不是W的整数倍肯定False 2.元素从小到大排序.sort()方法仅列表类型具备，sorted()方法适用于多种数据类型 3.列表的.remove()方法仅删除遇到的第一个匹配元素，因此从列表的第一个元素开始，相继删除list[0]至list[0]+W的所有元素，删完第一轮代表找出了第一套顺子，接下来不断循环，如果发现list[0]至list[0]+W的某元素不在列表中，表明缺牌，返回FALSE 即可，如果最终列表被删空，说明满足要求，返回True。

```
class Solution(object):
    def isNStraightHand(self, hand, W):
        """
        :type hand: List[int]
        :type W: int
        :rtype: bool
        """
        if len(hand)%W!=0:
            return False
        hand.sort()
        while len(hand)>0:
            No1=hand[0]
            for j in range(W):
                if No1+j in hand:
                    hand.remove(No1+j)
                    if hand==[]:
                        return True
                else:
                    return False
```

7.移动零

283. 移动零

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

```
输入: [0,1,0,3,12]
输出: [1,3,12,0,0]
```

说明:

1. 必须在原数组上操作，不能拷贝额外的数组。
2. 尽量减少操作次数。

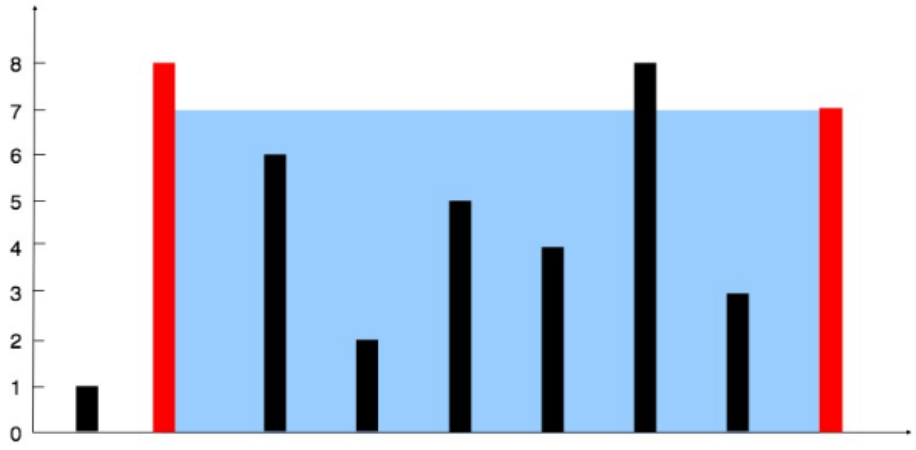
```
class Solution(object):
    def moveZeroes(self, nums):
        """
        :type nums: List[int]
        :rtype: None Do not return anything, modify nums in-place instead.
        """
        #快慢指针
        slow=0
        for fast in range(len(nums)):
            if nums[fast]!=0:
                nums[slow],nums[fast]=nums[fast],nums[slow]
                slow+=1
        return nums
```

8.盛最多水的容器

11. 盛最多水的容器

给定 n 个非负整数 a_1，a_2，…，a_n，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明： 你不能倾斜容器，且 n 的值至少为 2。



图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例:

```
输入：[1,8,6,2,5,4,8,3,7]
输出：49
```

思路：这种方法背后的思路在于，两线段之间形成的区域总是会受到其中较短那条长度的限制。此外，两线段距离越远，得到的面积就越大。

我们在由线段长度构成的数组中使用两个指针，一个放在开始，一个置于末尾。 此外，我们会使用变量 maxareamaxarea 来持续存储到目前为止所获得的最大面积。 在每一步中，我们会找出指针所指向的两条线段形成的区域，更新 maxareamaxarea，并将指向较短线段的指针向较长线段那端移动一步。

```python
class Solution(object):
    def maxArea(self, height):
        """
        :type height: List[int]
        :rtype: int
        """
        #双指针
        p1=0
        p2=len(height)-1
        maxnum=0
        while p1<p2:
            s=(p2-p1)*min(height[p2],height[p1])
            maxnum=max(maxnum,s)
            if height[p1]<=height[p2]:
                p1+=1
            else:
                p2-=1
        return maxnum
```

9.找到消失的数字

448. 找到所有数组中消失的数字

给定一个范围在 1 ≤ a[i] ≤ n (n = 数组大小) 的 整型数组，数组中的元素一些出现了两次，另一些只出现一次。

找到所有在 [1, n] 范围之间没有出现在数组中的数字。

您能在不使用额外空间且时间复杂度为O(n)的情况下完成这个任务吗? 你可以假定返回的数组不算在额外空间内。

示例:

```
输入:
[4,3,2,7,8,2,3,1]

输出:
[5,6]
```

思路：将所有正数作为数组下标，置对应数组值为负值。那么，仍为正数的位置即为（未出现过）消失的数字。

举个例子：

- 原始数组：[4,3,2,7,8,2,3,1]
- 重置后为：[-4,-3,-2,-7,`8`,`2`,-3,-1]

结论：[8,2] 分别对应的index为[5,6]（消失的数字）

```python
class Solution(object):
    def findDisappearedNumbers(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        #将出现的数字对应位置对应负数，没出现的位置仍未正数
        if not nums:
            return nums
        for i in range(len(nums)):
            nums[abs(nums[i])-1]=-abs(nums[abs(nums[i])-1])
        result=[]
        for i in range(len(nums)):
            if nums[i]>0:
                result.append(i+1)
        return result
```

10.反转字符串

[344. 反转字符串]

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 `char[]` 的形式给出。

不要给另外的数组分配额外的空间，你必须**原地修改输入数组**、使用 O(1) 的额外空间解决这一问题。

你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

示例 1:

```
输入: ["h","e","l","l","o"]
输出: ["o","l","l","e","h"]
```

示例 2:

```
输入: ["H","a","n","n","a","h"]
输出: ["h","a","n","n","a","H"]
```

```python
class Solution(object):
    def reverseString(self, s):
        """
        :type s: List[str]
        :rtype: None Do not return anything, modify s in-place instead.
        """
        #双指针
        if not s:
            return s
        p1=0
        p2=len(s)-1
        while p1<=p2:
            s[p1],s[p2]=s[p2],s[p1]
            p1+=1
            p2-=1
        return s
```

11.三数之和

15. 三数之和

给定一个包含 n 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 a，b，c，使得 a + b + c = 0？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

```
例如，给定数组 nums = [-1, 0, 1, 2, -1, -4]，

满足要求的三元组集合为：
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

思路：

- 暴力法搜索为 $O(N^3)$ 时间复杂度，可通过双指针动态消去无效解来优化效率。
- **双指针法铺垫：** 先将给定 `nums` 排序，复杂度为 $O(NlogN)$。
- **双指针法思路：** 固定 3 个指针中最左（最小）数字的指针 `k`，双指针 `i`，`j` 分设在数组索引 $(k, len(nums))$ 两端，通过双指针交替向中间移动，记录对于每个固定指针 `k` 的所有满足 `nums[k] + nums[i] + nums[j] == 0` 的 `i`, `j` 组合：
    1. 当 `nums[k] > 0` 时直接 `break` 跳出：因为 `nums[j] >= nums[i] >= nums[k] > 0`，即 3 个数字都大于 0，在此固定指针 `k` 之后不可能再找到结果了。
    2. 当 `k > 0` 且 `nums[k] == nums[k - 1]` 时即跳过此元素 `nums[k]`：因为已经将 `nums[k - 1]` 的所有组合加入到结果中，本次双指针搜索只会得到重复组合。
    3. `i`，`j` 分设在数组索引 $(k, len(nums))$ 两端，当 `i < j` 时循环计算 `s = nums[k] + nums[i] + nums[j]`，并按照以下规则执行双指针移动：
        - 当 `s < 0` 时，`i += 1` 并跳过所有重复的 `nums[i]`；
        - 当 `s > 0` 时，`j -= 1` 并跳过所有重复的 `nums[j]`；
        - 当 `s == 0` 时，记录组合 `[k, i, j]` 至 `res`，执行 `i += 1` 和 `j -= 1` 并跳过所有重复的 `nums[i]` 和 `nums[j]`，防止记录到重复组合。
- **复杂度分析：**
    - 时间复杂度 $O(N^2)$：其中固定指针 `k` 循环复杂度 $O(N)$，双指针 `i`，`j` 复杂度 $O(N)$。
    - 空间复杂度 $O(1)$：指针使用常数大小的额外空间。

```python
class Solution(object):
    def sum3(self, nums):
        nums=sorted(nums)
```

```
        result=[]
        #三指针
        for k in range(len(nums)-2):
            if nums[k]>0:
                break
            if k>0 and nums[k]==nums[k-1]:
                continue
            i=k+1
            j=len(nums)-1
            while i<j:
                s = nums[k] + nums[i] + nums[j]
                if s==0:
                    result.append([nums[k],nums[i],nums[j]])
                    i+=1
                    j-=1
                    while i<j and nums[i]==nums[i-1]:
                        i+=1
                    while i<j and nums[j]==nums[j+1]:
                        j-=1
                if s<0:
                    i+=1
                    while i<j and nums[i]==nums[i-1]:
                        i+=1
                if s>0:
                    j-=1
                    while i<j and nums[j]==nums[j+1]:
                        j-=1
        return result
if __name__=='__main__':
    S=Solution()
    nums=[-1,0,1,2,-1,-4]
    print(S.sum3(nums))
```

12.数组交集

[349. 两个数组的交集]

给定两个数组，编写一个函数来计算它们的交集。

示例 1:

```
输入: nums1 = [1,2,2,1], nums2 = [2,2]
输出: [2]
```

示例 2:

```
输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出: [9,4]
```

说明:

- 输出结果中的每个元素一定是唯一的。
- 我们可以不考虑输出结果的顺序。

```
class Solution(object):
    def intersection(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        set1=set(nums1)
        set2=set(nums2)
        return list(set1&set2)
```

[350. 两个数组的交集 II]

给定两个数组，编写一个函数来计算它们的交集。

示例 1:

```
输入: nums1 = [1,2,2,1], nums2 = [2,2]
输出: [2,2]
```

示例 2:

```
输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出: [4,9]
```

说明:

- 输出结果中每个元素出现的次数，应与元素在两个数组中出现的次数一致。
- 我们可以不考虑输出结果的顺序。

进阶:

- 如果给定的数组已经排好序呢？你将如何优化你的算法？
- 如果 *nums1* 的大小比 *nums2* 小很多，哪种方法更优？
- 如果 *nums2* 的元素存储在磁盘上，磁盘内存是有限的，并且你不能一次加载所有的元素到内存中，你该怎么办？

```
class Solution(object):
    def intersect(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        '''
         #排序+双指针
        nums1=sorted(nums1)
        nums2=sorted(nums2)
        result=[]
        i=j=0
        while i<len(nums1) and j<len(nums2):
            if nums1[i]==nums2[j]:
                result.append(nums1[i])
                i+=1
                j+=1
            elif nums1[i]<nums2[j]:
                i+=1
            else:
                j+=1
        return result
        '''
```

```
        #哈希表
        count=dict()
        res=[]
        for i in nums1:
            if i in count:
                count[i]+=1
            else:
                count[i]=1
        for j in nums2:
            if j in count and count[j]!=0:
                res.append(j)
                count[j]-=1
        return res
```

矩阵

1.有序矩阵中的查找

74. 搜索二维矩阵

编写一个高效的算法来判断 *m* x *n* 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

- 每行中的整数从左到右按升序排列。
- 每行的第一个整数大于前一行的最后一个整数。

示例 1:

```
输入:
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 3
输出: true
```

示例 2:

```
输入:
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 13
输出: false
```

思路：二分查找

```
class Solution(object):
    def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """
        #递增排序，二分查找
        #二维矩阵，mid坐标(mid//n,mid%n)
        m=len(matrix)
        if m==0:
```

```
            return False
        n=len(matrix[0])
        left=0
        right=m*n-1
        while left<=right:
            mid=(left+right)//2
            mid_value=matrix[mid//n][mid%n]
            if target==mid_value:
                return True
            if target<mid_value:
                right=mid-1
            if target>mid_value:
                left=mid+1
        return False
```

240. 搜索二维矩阵 II

编写一个高效的算法来搜索 *m* x *n* 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

示例:

现有矩阵 matrix 如下：

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

给定 target = `5`，返回 `true`。

给定 target = `20`，返回 `false`。

```
class Solution(object):
    def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """
        #去行去列查找：以右上角点为基准
        m=len(matrix)
        if m==0:
            return False
        n=len(matrix[0])
        row=0
        col=n-1
        while row<m and col>=0:
            if target==matrix[row][col]:
                return True
            if target<matrix[row][col]:
                col=col-1
            if target>matrix[row][col]:
                row=row+1
        return False
```

2.螺旋矩阵

54. 螺旋矩阵

给定一个包含 *m* x *n* 个元素的矩阵（*m* 行, *n* 列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。

示例 1:

```
输入:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
输出: [1,2,3,6,9,8,7,4,5]
```

示例 2:

```
输入:
[
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9,10,11,12]
]
输出: [1,2,3,4,8,12,11,10,9,5,6,7]
```

```python
class Solution(object):
    def spiralOrder(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[int]
        """
        row=len(matrix)
        if row==0:
            return None
        col=len(matrix[0])
        left=0
        right=col-1
        top=0
        bottom=row-1
        result=[]
        while left<=right and top<=bottom:
            #form left-to-right
            for i in range(left,right+1):
                result.append(matrix[top][i])

            #form top-to-bottom
            for i in range(top+1,bottom+1):
                result.append(matrix[i][right])
            #避免单行
            if top!=bottom:
                #from right-to-left
                for i in range(right-1,left-1,-1):
                    result.append(matrix[bottom][i])
            #避免单列
            if left!=right:
                #from bottom-to-top
                for i in range(bottom-1,top,-1):
                    result.append(matrix[i][left])
            left+=1
```

```
            right-=1
            top+=1
            bottom-=1
        return result
```

深度优先遍历

1.岛屿数量

200. 岛屿数量

给定一个由 '1'（陆地）和 '0'（水）组成的的二维网格，计算岛屿的数量。一个岛被水包围，并且它是通过水平方向或垂直方向上相邻的陆地连接而成的。你可以假设网格的四个边均被水包围。

示例 1:

```
输入:
11110
11010
11000
00000

输出: 1
```

示例 2:

```
输入:
11000
11000
00100
00011

输出: 3
```

思路：遍历岛这个二维数组，如果当前数为1，则进入感染函数并将岛个数+1

```
class Solution(object):
    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        #DFS：深度优先
        x_size=len(grid)
        if x_size==0:
            return 0
        y_size=len(grid[0])
        number=0
        for i in range(x_size):
            for j in range(y_size):
                if grid[i][j]=="1":
                    number+=1
                    self.Infect(i,j,grid)
        return number
    def Infect(self,x,y,grid):
        if x<0 or x>=len(grid) or y<0 or y>=len(grid[0]) or grid[x][y]!="1":
            return
        grid[x][y]=0     #设置flag
        #四个方向探索
        self.Infect(x-1,y,grid)
```

```
            self.Infect(x+1,y,grid)
            self.Infect(x,y-1,grid)
            self.Infect(x,y+1,grid)
```

2.字母大小写全排列

784. 字母大小写全排列

给定一个字符串 `S`，通过将字符串 `S` 中的每个字母转变大小写，我们可以获得一个新的字符串。返回所有可能得到的字符串集合。

示例：
输入： `S = "a1b2"`
输出： `["a1b2", "a1B2", "A1b2", "A1B2"]`

输入： `S = "3z4"`
输出： `["3z4", "3Z4"]`

输入： `S = "12345"`
输出： `["12345"]`

注意：

- `S` 的长度不超过 `12`。
- `S` 仅由数字和字母组成。

```
class Solution(object):
    def letterCasePermutation(self, S):
        """
        :type S: str
        :rtype: List[str]
        """
        length=len(S)
        result=[]
        if length==0:
            return result
        def DFS(start,tmp):
            if start>=length or len(tmp)==length: ##找到一种情况
                result.append(tmp)
                return
            if S[start].isdigit(): #数字
                DFS(start+1,tmp+S[start])
            elif S[start].isupper(): #大写字母
                DFS(start+1,tmp+S[start])
                DFS(start+1,tmp+S[start].lower())
            elif S[start].islower(): #小写字母
                DFS(start+1,tmp+S[start])
                DFS(start+1,tmp+S[start].upper())
        DFS(0,'')
        return result
```

3.子集

78. 子集

给定一组**不含重复元素**的整数数组 *nums*，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

```
输入: nums = [1,2,3]
输出:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

```python
class Solution(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
         #回溯法
        result=[]
        self.DFS(nums,0,[],result)
        return result
    def DFS(self, nums, index, path, result):
        result.append(path)
        for i in range(index, len(nums)):
            self.DFS(nums, i+1, path+[nums[i]], result)
```

查找与排序

1.二分查找

给定一个 `n` 个元素有序的（升序）整型数组 `nums` 和一个目标值 `target` ，写一个函数搜索 `nums` 中的 `target`，如果目标值存在返回下标，否则返回 `-1`。

示例 1:

```
输入: nums = [-1,0,3,5,9,12], target = 9
输出: 4
解释: 9 出现在 nums 中并且下标为 4
```

示例 2:

```
输入: nums = [-1,0,3,5,9,12], target = 2
输出: -1
解释: 2 不存在 nums 中因此返回 -1
```

提示：

1. 你可以假设 `nums` 中的所有元素是不重复的。
2. `n` 将在 `[1, 10000]` 之间。
3. `nums` 的每个元素都将在 `[-9999, 9999]` 之间。

```
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        low=0
        high=len(nums)-1
        while low<=high:
            mid=(low+high)//2
            if nums[mid]==target:
                return mid
            if nums[mid]<target:
                low=mid+1
            if nums[mid]>target:
                high=mid-1
        return -1
```

进阶：在排序数组中查找元素的第一个和最后一个位置

题号：53.在排序数组中查找数字 34. 在排序数组中查找元素的第一个和最后一个位置

题目描述：

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 O(log *n*) 级别。

如果数组中不存在目标值，返回 `[-1, -1]`。

示例 1:

```
输入: nums = [5,7,7,8,8,10], target = 8
输出: [3,4]
```

示例 2:

```
输入: nums = [5,7,7,8,8,10], target = 6
输出: [-1,-1]
```

```
class Solution(object):
    def searchRange(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        if nums==[]:
            return [-1,-1]
        first=self.FindFirstK(nums,target)
        last=self.FindLastK(nums,target)
        return [first,last]

    def FindFirstK(self, nums, target):
        low=0
        high=len(nums)-1
        while low<=high:
            mid=(low+high)//2
            if nums[mid]==target:
```

```
                if mid==0 or (mid>0 and nums[mid-1]!=target):
                    return mid
                else:
                    high=mid-1
            if nums[mid]<target:
                low=mid+1
            if nums[mid]>target:
                high=mid-1
        return -1
    def FindLastK(self, nums, target):
        low=0
        high=len(nums)-1
        length=len(nums)-1
        while low<=high:
            mid=(low+high)//2
            if nums[mid]==target:
                if mid==length or (mid<length and nums[mid+1]!=target):
                    return mid
                else:
                    low=mid+1
            if nums[mid]<target:
                low=mid+1
            if nums[mid]>target:
                high=mid-1
        return -1
```

2.Top K问题

Top K问题在数据分析中非常普遍的一个问题（在面试中也经常被问到），比如：

> 从20亿个数字的文本中，找出最大的前100个。

解决Top K问题有两种思路，

- 最直观：小顶堆（大顶堆 -> 最小100个数）；
- 较高效：Quick Select算法。

215.数组中第K大的元素

在未排序的数组中找到第 **k** 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

```
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5
```

示例 2:

```
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4
```

```
class Solution(object):
    def findKthLargest(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
```

```
        :rtype: int
        """
        #快排
        self.Quicksort(nums,0,len(nums)-1)
        print(nums)
        return nums[k-1]
    def Quicksort(self, nums, low, high):
        if low<high:
            left=low
            right=high
            base=nums[low]
            while left<right:
                while left<right and nums[right]<base:
                    right-=1
                while left<right and nums[left]>=base:
                    left+=1
                if left<right:
                    nums[left],nums[right]=nums[right],nums[left]
            nums[low],nums[right]=nums[right],nums[low]
            self.Quicksort(nums,low,right-1)
            self.Quicksort(nums,right+1,high)
```

39.数组中出现次数超过一半的数字

题目：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。

思路：**快排之后**，处于中间的那个数，如果出现次数超过一半，即为要求的数

```
# -*- coding:utf-8 -*-
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        # write code here
        if not numbers:
            return 0
        self.Quicksort(numbers,0,len(numbers)-1)
        k=numbers[len(numbers)/2]
        flag=self.MorethanHalf(numbers,k)
        if flag:
            return k
        else:
            return 0
    def Quicksort(self,numbers,low,high):
        if low<high:
            left=low
            right=high
            base=numbers[low]
            while left<right:
                while left<right and numbers[right]>base:
                    right-=1
                while left<right and numbers[left]<=base:
                    left+=1
                if left<right:
                    numbers[left],numbers[right]=numbers[right],numbers[left]
            numbers[low],numbers[right]=numbers[right],numbers[low]
            self.Quicksort(numbers,low,right-1)
```

```
            self.Quicksort(numbers,right+1,high)
    def MorethanHalf(self,numbers,k):
        flag=0
        num=0
        for i in range(len(numbers)):
            if numbers[i]==k:
                num+=1
        mid=len(numbers)/2
        if num>mid:
            flag=1
        return flag
```

40.最小的k个数

思路：排序，快排O(nlogn),海量数据时候，选择堆排序(nlogk)。

堆排序思路（大）

(1).将待排序的数组构建出一个大根堆

(2).取出这个大根堆的堆顶节点（最大值）与堆的最下最右的元素进行交换，然后把剩下的元素再构造一个大根堆。

(3).重复第二步，直到这个大根堆的长度为 1 ， 此时完成排序。

left=2*root+1

right=2*root+2

```
# -*- coding:utf-8 -*-
class Solution:
    def GetLeastNumbers_Solution(self, tinput, k):
        # write code here
        #堆排序——大顶堆
        mlist=[]
        if not tinput or len(tinput)<k or k<=0:
            return mlist
        for i in range(k):
            mlist.append(tinput[i])
        ## 先建立大根堆
        self.BuildHeap(mlist,k)

        ## 继续往后搜索，找到k个小的数
        for i in range(k,len(tinput)):
            if tinput[i]<mlist[0]:  ##堆顶是最大的
                mlist[0]=tinput[i]
                self.AdjustHeap(mlist,k,0)       ##调整使得堆顶最大
        ## 把大顶堆排序
        for i in range(k-1,0,-1):
            ##最大值一个个取出来
            mlist[0],mlist[i]=mlist[i],mlist[0]
            self.AdjustHeap(mlist,i,0)

        return mlist


    def BuildHeap(self,heap,n):
        for i in range((n-2)/2,-1,-1): ##从有左右孩子的节点开始
            self.AdjustHeap(heap,n,i)
```

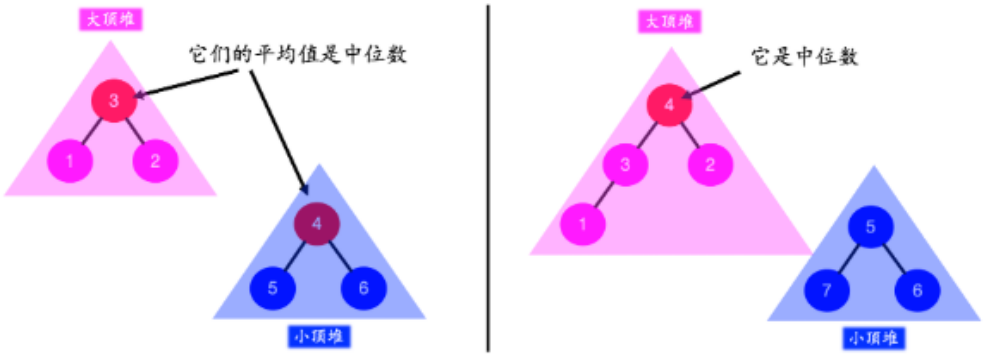
```
    def AdjustHeap(self,heap,n,root):  ##堆调整
        left=2*root+1
        right=left+1
        larger=root                          ##根节点大于左右子节点
        if left<n and heap[left]>heap[larger]:
            larger=left
        if right<n and heap[right]>heap[larger]:
            larger=right
        if larger!=root:    ##左右节点之一
            heap[larger],heap[root]=heap[root],heap[larger]
            self.AdjustHeap(heap,n,larger)            ##继续往下调整
```

3.数据流中的中位数

41.数据流中的中位数（TODO）

4、由于我们只关心这两个“有序数组”中的最值，有一个数据结构可以帮助我们快速找到这个最值，这就是“优先队列”。具体来说：
（1）“前有序数组”因为关心最大值，可以“动态地”放置在一个“大顶堆”中；
（2）“后有序数组”因为关心最小值，可以“动态地”放置在一个“小顶堆”中。

5、当从数据流中读出的数的个数为偶数的时候，让两个堆中的元素个数相等，两个堆顶元素的平均值就是所求的中位数（如下左图）；当从数据流中读出的数的个数为奇数的时候，只要保证大顶堆的元素个数永远比小顶堆的元素个数多 1 个，那么大顶堆的堆顶元素就是所求的中位数（如下右图）。



4.寻找两个有序数组的中位数

给定两个大小为 m 和 n 的有序数组 `nums1` 和 `nums2`。

请你找出这两个有序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。

你可以假设 `nums1` 和 `nums2` 不会同时为空。

示例 1:

```
nums1 = [1, 3]
nums2 = [2]

则中位数是 2.0
```

示例 2:

```
nums1 = [1, 2]
nums2 = [3, 4]

则中位数是 (2 + 3)/2 = 2.5
```

```
class Solution(object):
    def findMedianSortedArrays(self, nums1, nums2):
        """
        :type nums1: List[int]
```

```
        :type nums2: List[int]
        :rtype: float
        """
        if not nums1:
            if len(nums2)%2==1:
                return nums2[len(nums2)//2]
            else:
                return (nums2[len(nums2)//2-1]+nums2[len(nums2)//2])/2.0
        if not nums2:
            if len(nums1)%2==1:
                return nums1[len(nums1)//2]
            else:
                return (nums1[len(nums1)//2-1]+nums1[len(nums1)//2])/2.0
        p1=0
        p2=0
        length1=len(nums1)
        length2=len(nums2)
        nums=[]
        while p1<length1 and p2<length2:
            if nums1[p1]<=nums2[p2]:
                nums.append(nums1[p1])
                p1+=1
            else:
                nums.append(nums2[p2])
                p2+=1
        if p1==length1:
            while p2<length2:
                nums.append(nums2[p2])
                p2+=1
        if p2==length2:
            while p1<length1:
                nums.append(nums1[p1])
                p1+=1
        if (length1+length2)%2==1:
            return nums[(length1+length2)//2]
        else:
            return (nums[(length1+length2)//2-1]+nums[(length1+length2)//2])/2.0
```

5.合并两个有序数组

思路：两个指针，从前往后，时间效率O(m+n)，空间效率O(m+n)需要借助额外空间

如果，两个指针从后往前，不需要额外的空间。

给定两个有序整数数组 *nums1* 和 *nums2*，将 *nums2* 合并到 *nums1* 中，使得 *num1* 成为一个有序数组。

说明:

- 初始化 *nums1* 和 *nums2* 的元素数量分别为 *m* 和 *n*。
- 你可以假设 *nums1* 有足够的空间（空间大小大于或等于 *m + n*）来保存 *nums2* 中的元素。

示例:

输入:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],   n = 3

输出: [1,2,2,3,5,6]

```
class Solution(object):
```

```
    def merge(self, nums1, m, nums2, n):
        """
        :type nums1: List[int]
        :type m: int
        :type nums2: List[int]
        :type n: int
        :rtype: None Do not return anything, modify nums1 in-place instead.
        """
        #从后往前
        p1=m-1
        p2=n-1
        p=m+n-1
        while p1>=0 and p2>=0:
            if nums1[p1]>=nums2[p2]:
                nums1[p]=nums1[p1]
                p1-=1
            else:
                nums1[p]=nums2[p2]
                p2-=1
            p-=1
        if p2>=0:
            nums1[:p2+1]=nums2[:p2+1]
        return nums1
```

6.旋转排序数组中的最小值

153. 寻找旋转排序数组中的最小值

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

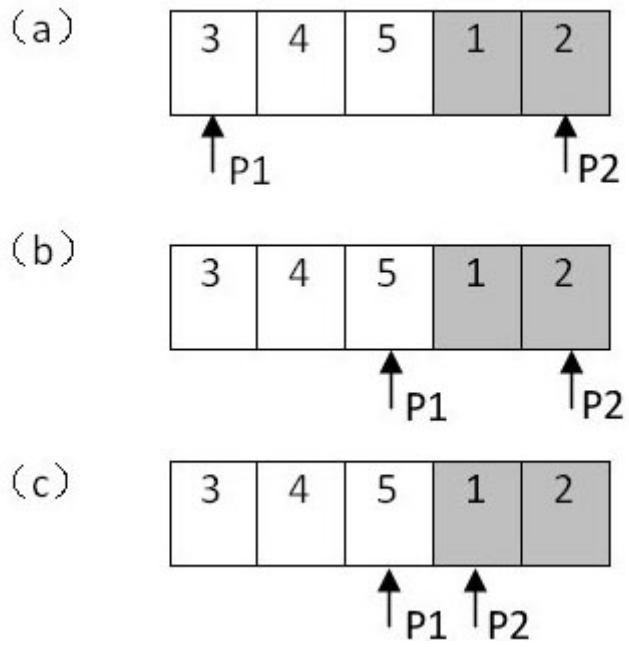
示例 1:

```
输入: [3,4,5,1,2]
输出: 1
```

示例 2:

```
输入: [4,5,6,7,0,1,2]
输出: 0
```

思路：二分法




```python
class Solution(object):
    def findMin(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #二分查找，O(logn)
        #不存在重复元素
        if not nums:
            return -1
        if len(nums)==1:
            return nums[0]
        p1=0                #前半段
        p2=len(nums)-1      #后半段
        while nums[p1]>nums[p2]: #发生旋转
            if p2-p1==1:
                return nums[p2]
            mid=(p1+p2)//2
            if nums[mid]>nums[p1]: #mid在前半段
                p1=mid
            if nums[mid]<nums[p2]: #mid在后半段
                p2=mid
        return nums[0]
```

154. 寻找旋转排序数组中的最小值 II

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

请找出其中最小的元素。

注意数组中可能存在重复的元素。

示例 1:

```
输入: [1,3,5]
输出: 1
```

示例 2:

```
输入: [2,2,2,0,1]
输出: 0
```

说明:

- 这道题是 寻找旋转排序数组中的最小值 的延伸题目。
- 允许重复会影响算法的时间复杂度吗？会如何影响，为什么？

思路：二分查找法，但需要考虑特殊，如[2,2,2,2,0,1,2]

```
class Solution(object):
    def findMin(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #二分查找O(logn)，特殊情况，顺序查找O(n)
        if not nums:
            return nums
        if len(nums)==1:
            return nums[0]
        p1=0
        p2=len(nums)-1
        while nums[p1]>=nums[p2]:
            if p2-p1==1:
                return nums[p2]
            mid=(p1+p2)//2
            if nums[p1]==nums[mid]==nums[p2]: ##重复很多O(n)
                return min(nums)
            if nums[mid]>=nums[p1]:
                p1=mid
            if nums[mid]<=nums[p2]:
                p2=mid
        return nums[0] #没有旋转
```

7.搜索旋转排序数组

33. 搜索旋转排序数组

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 `-1` 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 *O*(log *n*) 级别。

示例 1:

```
输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4
```

示例 2:

```
输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1
```

思路：二分查找，先找有序范围，再判断有序范围内有没有，更新left或right指针，不断缩小范围。O(logn)

直接使用二分法，

判断那个二分点有几种可能性

1. 直接等于 `target`
2. 在左半边的递增区域
   a. `target` 在 `left` 和 `mid` 之间
   b.不在之间
3. 在右半边的递增区域
   a. `target` 在 `mid` 和 `right` 之间
   b. 不在之间

```
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        #mid总会出现在有序的一侧
        if not nums:
            return -1
        if len(nums)==1:
            if nums[0]==target:
                return 0
            else:
                return -1
        p1=0
        p2=len(nums)-1
        while p1<=p2:
            mid=(p1+p2)//2
            if nums[mid]==target:
                return mid
            if nums[mid]>=nums[p1]: #在左侧有序区
```

```
                if target>=nums[p1] and target<nums[mid]:
                    p2=mid-1
                else:
                    p1=mid+1
            if nums[mid]<=nums[p2]: #z在右侧有序区
                if target>nums[mid] and target<=nums[p2]:
                    p1=mid+1
                else:
                    p2=mid-1
        return -1
```

[81. 搜索旋转排序数组 II]

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,0,1,2,2,5,6]` 可能变为 `[2,5,6,0,0,1,2]`)。

编写一个函数来判断给定的目标值是否存在于数组中。若存在返回 `true`，否则返回 `false`。

示例 1:

```
输入: nums = [2,5,6,0,0,1,2], target = 0
输出: true
```

示例 2:

```
输入: nums = [2,5,6,0,0,1,2], target = 3
输出: false
```

进阶:

- 这是 搜索旋转排序数组 的延伸题目，本题中的 `nums` 可能包含重复元素。
- 这会影响到程序的时间复杂度吗？会有怎样的影响，为什么？

思路：方法：二分查找，先找有序范围，再判断有序范围内有没有，更新left或right指针，不断缩小范围

如果a[mid]与a[right]相等，则让right--直到不相等，其余部分与问题Search in Rotated Sorted Array I相同

O(logn)

```
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: bool
        """
        if not nums:
            return False
        if len(nums)==1:
            if nums[0]==target:
                return True
            else:
                return False
        p1=0
        p2=len(nums)-1
        while p1<=p2:
            mid=(p1+p2)//2
            if target==nums[mid]:
```

```
                return True
            elif nums[mid]<nums[p2]:
                if target>nums[mid] and target<=nums[p2]:
                    p1=mid+1
                else:
                    p2=mid-1
            elif nums[mid]>nums[p2]:
                if target>=nums[p1] and target<nums[mid]:
                    p2=mid-1
                else:
                    p1=mid+1
            else:           ###重复元素处理
                p2-=1
        return False
```

8.搜索峰值

[162. 寻找峰值]

峰值元素是指其值大于左右相邻值的元素。

给定一个输入数组 `nums`，其中 `nums[i] ≠ nums[i+1]`，找到峰值元素并返回其索引。

数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。

你可以假设 `nums[-1] = nums[n] = -∞`。

示例 1:

```
输入: nums = [1,2,3,1]
输出: 2
解释: 3 是峰值元素，你的函数应该返回其索引 2。
```

示例 2:

```
输入: nums = [1,2,1,3,5,6,4]
输出: 1 或 5
解释: 你的函数可以返回索引 1，其峰值元素为 2;
     或者返回索引 5， 其峰值元素为 6。
```

思路：二分查找 过程： 首先要注意题目条件，在题目描述中出现了 nums[-1] = nums[n] = -∞，这就代表着 只要数组中存在一个元素比相邻元素大，那么沿着它一定可以找到一个峰值 根据上述结论，我们就可以使用二分查找找到峰值 查找时，左指针p1，右指针 p2，以其保持左右顺序为循环条件 根据左右指针计算中间位置 m，并比较 m 与 m+1 的值，如果 m 较大，则左侧存在峰值，r = m，如果 m + 1 较大，则右侧存在峰值，l = m + 1 时间复杂度：O(logN)O(logN)

```
class Solution(object):
    def findPeakElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #二分法：条件是，往旁边大的一侧缩进
        if not nums:
            return nums
        if len(nums)==1:
            return 0
        p1=0
        p2=len(nums)-1
        while p1<p2:
```

```
            mid=(p1+p2)//2
            if nums[mid]<nums[mid+1]:
                p1=mid+1
            else:
                p2=mid
        return p1
```

9.第一个错误版本

278. 第一个错误的版本

你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最新版本没有通过质量检测。由于每个版本都是基于之前的版本开发的，所以错误的版本之后的所有版本都是错的。

假设你有 `n` 个版本 `[1, 2, ..., n]`，你想找出导致之后所有版本出错的第一个错误的版本。

你可以通过调用 `bool isBadVersion(version)` 接口来判断版本号 `version` 是否在单元测试中出错。实现一个函数来查找第一个错误的版本。你应该尽量减少对调用 API 的次数。

示例:

```
给定 n = 5，并且 version = 4 是第一个错误的版本。

调用 isBadVersion(3) -> false
调用 isBadVersion(5) -> true
调用 isBadVersion(4) -> true

所以，4 是第一个错误的版本。
```

思路：二分查找

```
# The isBadVersion API is already defined for you.
# @param version, an integer
# @return a bool
# def isBadVersion(version):

class Solution(object):
    def firstBadVersion(self, n):
        """
        :type n: int
        :rtype: int
        """
        #二分查找，O(logn)
        p1=0
        p2=n
        while p1<=p2:
            mid=(p1+p2)//2
            if isBadVersion(mid)==False:
                p1=mid+1
            else:
                p2=mid-1
        return p1
```

10.会议室

252. 会议室

给定一个会议时间安排的数组，每个会议时间都会包括开始和结束的时间 `[[s1,e1],[s2,e2],...]` ($s_i < e_i$)，请你判断一个人是否能够参加这里面的全部会议。

示例 1:

```
输入: [[0,30],[5,10],[15,20]]
输出: false
```

示例 2:

```
输入: [[7,10],[2,4]]
输出: true
```

```
class Solution(object):
    def canAttendMeetings(self, intervals):
        """
        :type intervals: List[List[int]]
        :rtype: bool
        """
        #先把开始的时间排序，再比较后一个开始的时间是不是在前一个结束之前
        intervals=sorted(intervals,key=lambda s:s[0])      #按关键词排序
        for i in range(1,len(intervals)):
            if intervals[i][0]<intervals[i-1][1]:
                return False
        return True
```

字符串

1.第一次只出现一次的字符

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

案例:

```
s = "leetcode"
返回 0.

s = "loveleetcode",
返回 2.
```

注意事项: 您可以假定该字符串只包含小写字母。

```
class Solution(object):
    def firstUniqChar(self, s):
        """
        :type s: str
        :rtype: int
        """
        #haspmap
        result={}
        for i in s:
```

```
            if i in result:
                result[i]+=1
            else:
                result[i]=1
        for i in range(len(s)):
            if result[s[i]]==1:
                return i
        return -1
```

2.最长公共前缀

14. 最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 `""`。

示例 1:

```
输入: ["flower","flow","flight"]
输出: "fl"
```

示例 2:

```
输入: ["dog","racecar","car"]
输出: ""
解释: 输入不存在公共前缀。
```

```
class Solution(object):
    def longestCommonPrefix(self, strs):
        """
        :type strs: List[str]
        :rtype: str
        """
        #只需要比较最短与最长的字符串
        if len(strs)==0:
          return ""
        str1=min(strs)
        str2=max(strs)
        for i,c in enumerate(str1):
          if c!=str2[i]:
            return str1[:i]
        return str1 #防止一个空串
```

树

1.二叉树的遍历

递归遍历

```
#前序：根-左-右
def preorder(self,root,path):
    if root==None:
        return
    path.append(root.val)
    self.preorder(root.left,path)
```

```
    self.preorder(root.right,path)
#中序：左-根-右
def inorder(self,root,path):
    if root==None:
        return
    self.inorder(root.left,path)
    path.append(root.val)
    self.inorder(root.right,path)
#后序：左-右-根
def postorder(self,root,path):
    if root==None:
        return
    self.postorder(root.left,path)
    self.postorder(root.right,path)
    path.append(root.val)
```

非递归遍历

144. 二叉树的前序遍历

思路：

- 前序遍历的顺序为根-左-右，具体算法为：
- 把根节点push到栈中
- 循环检测栈是否为空，若不空，则取出栈顶元素，保存其值
- 看其右子节点是否存在，若存在则push到栈中
- 看其左子节点，若存在，则push到栈中

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def preorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        #根-左-右 利用栈
        stack=[]
        path=[]
        if not root:
            return path
        stack.append(root)
        while stack:
            node=stack.pop()
            path.append(node.val)
            if node.right:
                stack.append(node.right)  #先进后出
            if node.left:
                stack.append(node.left)
        return path
```

94. 二叉树的中序遍历

给定一个二叉树，返回它的*中序*遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [1,3,2]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

思路：

- 中序遍历的顺序为左-根-右，具体算法为：
- 从根节点开始，先将根节点压入栈
- 然后再将其所有左子结点压入栈，取出栈顶节点，保存节点值
- 再将当前指针移到其右子节点上，若存在右子节点，则在下次循环时又可将其所有左子结点压入栈中

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def inorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        stack=[]
        path=[]
        if not root:
            return path
        node=root
        while stack or node:
            if node:
                stack.append(node)
                node=node.left
            else:
                node=stack.pop()
                path.append(node.val)
                node=node.right
        return path
```

[145. 二叉树的后序遍历]

给定一个二叉树，返回它的 *后序* 遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [3,2,1]
```

思路：两个栈-联系前序遍历

后续遍历为左右根，而先序遍历为根左右，可以像先序遍历的那样，做出根右左的结构，先左压左节点进栈，后右节点进栈1，但是不打印，将根右左的结构再压入栈2中，变成左右根，弹出打印，便是后序遍历

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def postorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        #后序遍历：左-右-根
        #借助两个栈
        path=[]
        stack1=[]
        stack2=[]
        if not root:
            return path
        node=root
        stack1.append(node)
        while stack1:
            node=stack1.pop()
            stack2.append(node)
            if node.left:
                stack1.append(node.left)
            if node.right:
                stack1.append(node.right)
        while stack2:
            path.append(stack2.pop().val)
        return path
```

102. 二叉树的层次遍历

给定一个二叉树，返回其按层次遍历的节点值。 （即逐层地，从左到右访问所有节点）。

例如:
给定二叉树: `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其层次遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

思路：利用队列的先进先出(FIFO)特征。每次从队列头部获取一个节点，就将该节点的左右子节点加入队列尾部，如此反复，直到队列为空。分层需要增加行存储

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def levelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        path=[]
        if not root:
            return path
        curLayer=[]
        curLayer.append(root)
        while curLayer:
            row=[]
            nextLayer=[]
            for node in curLayer:  #队列：先进先出
                row.append(node.val)
                if node.left:
                    nextLayer.append(node.left)
                if node.right:
                    nextLayer.append(node.right)
            path.append(row)
            curLayer=nextLayer
        return path
```

103. 二叉树的锯齿形层次遍历(之字形)

给定一个二叉树，返回其节点值的锯齿形层次遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：
给定二叉树 `[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回锯齿形层次遍历如下：

```
[
  [3],
  [20,9],
  [15,7]
]
```

思路：偶数行反转python 倒序row[::-1]

```python
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def zigzagLevelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        path=[]
        if not root:
            return path
        curLayer=[]
        curLayer.append(root)
        Iseven=True   #偶数行标志
        while curLayer:
            row=[]
            nextLayer=[]
            Iseven=not Iseven
            for node in curLayer:  #队列：先进先出
                row.append(node.val)
                if node.left:
                    nextLayer.append(node.left)
                if node.right:
                    nextLayer.append(node.right)
            if Iseven:
                path.append(row[::-1])
            else:
                path.append(row)
            curLayer=nextLayer
        return path
```

二叉树的垂直遍历(带锁) Given a binary tree, return the vertical order traversal of its nodes' values. (ie, from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from left to right.

例子：

Given binary tree [3,9,20,null,null,15,7],

```
   3
  / \
 9  20
   /  \
  15   7
```

return its vertical order traversal as:

```
[
  [9],
  [3,15],
  [20],
  [7]
]
```

登录后复制

Given binary tree [3,9,20,4,5,2,7],

```
    _3_
   /   \
  9    20
 / \   / \
4   5 2   7
```

return its vertical order traversal as:

```
[
  [4],
  [9],
  [3,5,2],
  [20],
  [7]
]
```

思路：

- 以根结点的标签为0作为基础，每个节点的左子结点-1，右子节点+1，相同标签的都存在一个vector中；
- 利用map来映射相同标签的vector；
- 利用队列来对每个节点进行处理，同时入队的为绑定标签的节点，用pair来绑定。

```
struct BinaryTree {
    int val;
    BinaryTree *left;
    BinaryTree *right;
    BinaryTree(int value) :
        val(value), left(nullptr), right(nullptr) { }
};

vector<vector<int> > verticalOrder(BinaryTree *root) {
    vector<vector<int> > res;
    if (root == nullptr)
        return res;

    map<int, vector<int>> m;
    queue<pair<int, BinaryTree*>> q;
    q.push({ 0, root });
    while (!q.empty()) {
```

```
        auto a = q.front();
        q.pop();
        m[a.first].push_back(a.second->val);
        if (a.second->left)
            q.push({a.first-1, a.second->left});
        if (a.second->right)
            q.push({a.first+1, a.second->right});
    }
    for (auto a : m) {
        res.push_back(a.second);
    }

    return res;
}
```

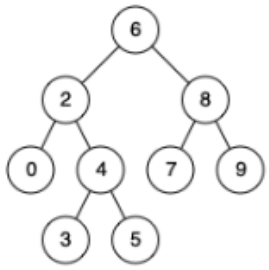
2.二叉搜索树的最近公共祖先

235.二叉搜索树的最近公共祖先

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。

示例 2:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
解释: 节点 2 和节点 4 的最近公共祖先是 2，因为根据定义最近公共祖先节点可以为节点本身。

思路：这个分割点就是能让节点 p和节点 q 不能在同一颗子树上的那个节点，或者是节点 p 和节点 q中的一个，这种情况下其中一个节点是另一个节点的父亲节点。

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def lowestCommonAncestor(self, root, p, q):
```

```
        """
        :type root: TreeNode
        :type p: TreeNode
        :type q: TreeNode
        :rtype: TreeNode
        """
        #迭代,利用二叉搜索树的特性
        if not root:
            return None
        p_val=p.val
        q_val=q.val
        node=root
        while node:
            parent_val=node.val
            if p_val<parent_val and q_val<parent_val:
                node=node.left
            elif p_val>parent_val and q_val>parent_val:
                node=node.right
            else:
                return node
        return None
```

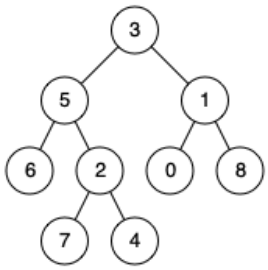
二叉树的最近公共祖先

[236. 二叉树的最近公共祖先]

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出: 3
解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出: 5
解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

思路：在左、右子树中分别查找是否包含p或q，如果（两种情况：左子树包含p，右子树包含q/左子树包含q，右子树包含p），那么此时的根节点就是最近公共祖先 如果左子树包含p和q，那么到root->left中查找，最近公共祖先在左子树里面 如果右子树包含p和q，那么到root->right中查找，最近公共祖先在右子树里面

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def lowestCommonAncestor(self, root, p, q):
        """
        :type root: TreeNode
        :type p: TreeNode
        :type q: TreeNode
        :rtype: TreeNode
        """
        #p,q出现在左、右子树的情况
        if not root or root==p or root==q:
            return root
        left=self.lowestCommonAncestor(root.left,p,q)
        right=self.lowestCommonAncestor(root.right,p,q)
        if not left:
            return right      #p、q在右子树
        if not right:
            return left       #p、q在左子树
        return root
```

3.合并二叉树

617. 合并二叉树

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则**不为** NULL 的节点将直接作为新二叉树的节点。

示例 1:

输入:

```
        Tree 1                     Tree 2
          1                         2
         / \                       / \
        3   2                     1   3
       /                           \   \
      5                             4   7
```

输出:

合并后的树:

```
             3
            / \
           4   5
          / \   \
         5   4   7
```

注意: 合并必须从两个树的根节点开始。

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def mergeTrees(self, t1, t2):
        """
        :type t1: TreeNode
        :type t2: TreeNode
        :rtype: TreeNode
        """
        #递归合并
        if not t1:
            return t2
        if not t2:
            return t1
        newTree=TreeNode(t1.val+t2.val)
        newTree.left=self.mergeTrees(t1.left,t2.left)
        newTree.right=self.mergeTrees(t1.right,t2.right)
        return newTree
```

4.翻转二叉树

226. 翻转二叉树

翻转一棵二叉树。

示例：

输入：

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

输出：

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

备注：
这个问题是受到 Max Howell 的 原问题 启发的：

> 谷歌：我们90%的工程师使用您编写的软件(Homebrew)，但是您却无法在面试时在白板上写出翻转二叉树这道题，这太糟糕了。

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def invertTree(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        if not root:
            return None
        newTree=TreeNode(root.val)
        newTree.right=self.invertTree(root.left)
        newTree.left=self.invertTree(root.right)
        return newTree
```

5.二叉树的最大深度

104. 二叉树的最大深度

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

示例：
给定二叉树 `[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

思路：标签：DFS 找出终止条件：当前节点为空 找出返回值：节点为空时说明高度为0，所以返回0；节点不为空时则分别求左右子树的高度的最大值，同时加1表示当前节点的高度，返回该数值 某层的执行过程：在返回值部分基本已经描述清楚 时间复杂度：O(n)

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        #DFS:递归
        if not root:
            return 0
        else:
            left=self.maxDepth(root.left)
            right=self.maxDepth(root.right)
            return max(left,right)+1
```

6.平衡二叉树

110. 平衡二叉树

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

> 一个二叉树*每个节点*的左右两个子树的高度差的绝对值不超过1。

示例 1:

给定二叉树 `[3,9,20,null,null,15,7]`

```
    3
   / \
  9  20
    /  \
   15   7
```

返回 `true` 。

示例 2:

给定二叉树 `[1,2,2,3,3,null,null,4,4]`

```
       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
```

返回 `false` 。

思路：左子树是平衡二叉树

右子树是平衡二叉树

左右子树深度不超过1

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def isBalanced(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        if root==None:
            return True
        if not self.isBalanced(root.left) or not self.isBalanced(root.right):
            return False

        left=self.maxDepth(root.left)
        right=self.maxDepth(root.right)
        if abs(left-right)<2:
            return True
        else:
            return False
    def maxDepth(self, root):
        if root==None:
            return 0
        left=self.maxDepth(root.left)
        right=self.maxDepth(root.right)
        return max(left,right)+1
```

7.把二叉搜索树转换为累加树

538. 把二叉搜索树转换为累加树

题目描述 | 评论 (78) | 题解(21) New | 提交记录

给定一个二叉搜索树（Binary Search Tree），把它转换成为累加树（Greater Tree)，使得每个节点的值是原来的节点值加上所有大于它的节点值之和。

例如：

输入： 二叉搜索树：

```
              5
            /   \
           2     13
```

输出： 转换为累加树：

```
             18
            /   \
          20     13
```

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def convertBST(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        #DFS:逆中序遍历
        '''
         #1.递归
        self.sum=0
        def DFS(root):
            if not root:
                return
            DFS(root.right)
            self.sum=self.sum+root.val
            root.val=self.sum
            DFS(root.left)
            return root
        return DFS(root)

        '''
        #2.非递归，栈
        nsum=0
        node=root
        stack=[]
        while node or len(stack):
            while node:
                stack.append(node)
                node=node.right
            node=stack.pop()
            nsum=nsum+node.val
            node.val=nsum
            node=node.left
        return root
```

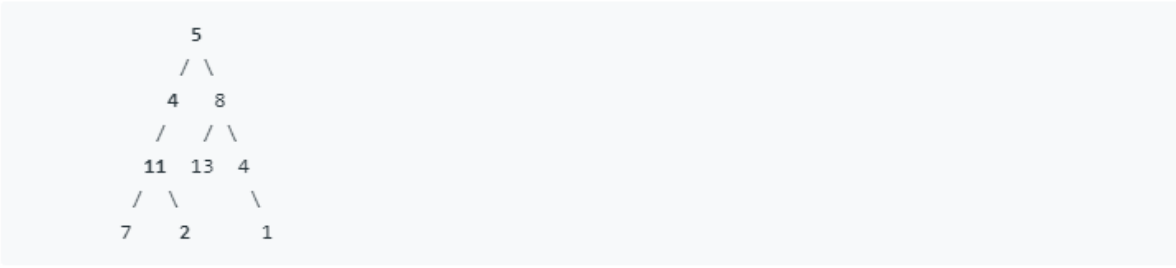
8.路径总和

112. 路径总和

给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

说明: 叶子节点是指没有子节点的节点。

示例:
给定如下二叉树，以及目标和 `sum = 22`，



返回 `true`，因为存在目标和为 22 的根节点到叶子节点的路径 `5->4->11->2`。

递归/迭代

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        #递归:DFS
        if not root:
            return False
        sum=sum-root.val
        if not root.left and not root.right and sum==0:
            return True
        return self.hasPathSum(root.left,sum) or self.hasPathSum(root.right,sum)
```

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        #迭代：栈
        if not root:
            return False
```

```
        stack=[]
        stack.append((root,sum-root.val))
        while stack:
            node,cur_sum=stack.pop()
            if not node.left and not node.right and cur_sum==0:
                return True
            if node.left:
                stack.append((node.left,cur_sum-node.left.val))
            if node.right:
                stack.append((node.right,cur_sum-node.right.val))
        return False
```

[113. 路径总和 II]

给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

说明: 叶子节点是指没有子节点的节点。

示例:
给定如下二叉树，以及目标和 `sum = 22`，

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \    / \
        7    2  5   1
```

返回:

```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

思路：递归

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def pathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: List[List[int]]
        """
        #DFS:递归
        result=[]
        if not root:
            return result
        def DFS(root,sum,path):
            if not root:
                return
            sum=sum-root.val
```

```
            if not root.left and not root.right and sum==0:
                path+=[root.val]
                result.append(path)
                return
            DFS(root.left,sum,path+[root.val])
            DFS(root.right,sum,path+[root.val])
        DFS(root,sum,[])
        return result
```

437. 路径总和 III

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过1000个节点，且节点数值范围是 [-1000000,1000000] 的整数。

示例：

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

返回 3。和等于 8 的路径有：

1.  5 -> 3
2.  5 -> 2 -> 1
3.  -3 -> 11
```

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def __init__(self):
        self.pathnumber=0
    def pathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: int
        """
        #双重递归，先递归所有根节点的路径，再递归每一个条路上的节点
        if not root:
            return self.pathnumber
        self.Sum(root,sum)
        self.pathSum(root.left,sum)
        self.pathSum(root.right,sum)
        return self.pathnumber
    def Sum(self,root,sum):
            #计算该节点处的路径条数
```

```
            if not root:
                return
            sum=sum-root.val
            if sum==0:
                self.pathnumber+=1
            self.Sum(root.left,sum)
            self.Sum(root.right,sum)
```

9.对称二叉树

101. 对称二叉树

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 `[1,2,2,3,4,4,3]` 是对称的。

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

但是下面这个 `[1,2,2,null,3,null,3]` 则不是镜像对称的:

```
    1
   / \
  2   2
   \   \
   3    3
```

说明:

如果你可以运用递归和迭代两种方法解决这个问题，会很加分。

思路：左右对称。

```
class Solution(object):
    def isSymmetric(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        #镜像
        #递归
        return self.isMirror(root,root)
    def isMirror(self,p1,p2):
        if not p1 and not p2:
            return True
        if not p1 or not p2:
            return False
        if p1.val!=p2.val:
            return False
        else:
            return self.isMirror(p1.left,p2.right) and
self.isMirror(p1.right,p2.left)
```

```
        #迭代：队列，连续两个值相等
        queue=[]
        queue.append(root)
```

```
        queue.append(root)
        while len(queue)!=0:
            p1=queue.pop()
            p2=queue.pop()
            if not p1 and not p2:
                continue
            if not p1 or not p2:
                return False
            if p1.val!=p2.val:
                return False
            else:
                queue.append(p1.left)
                queue.append(p2.right)
                queue.append(p1.right)
                queue.append(p2.left)
        return True
```

10.前序中序构建二叉树

105. 从前序与中序遍历序列构造二叉树

根据一棵树的前序遍历与中序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

例如，给出

```
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]
```

返回如下的二叉树：

```
    3
   / \
  9  20
    /  \
   15   7
```

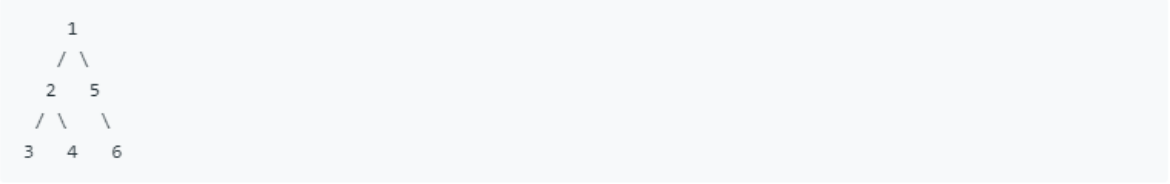
```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def buildTree(self, preorder, inorder):
        """
        :type preorder: List[int]
        :type inorder: List[int]
        :rtype: TreeNode
        """
        if not inorder: return
        root = TreeNode(preorder.pop(0))
        i = inorder.index(root.val)
        root.left = self.buildTree(preorder, inorder[:i])
        root.right = self.buildTree(preorder, inorder[i+1:])
        return root
```

11.二叉树展开链表

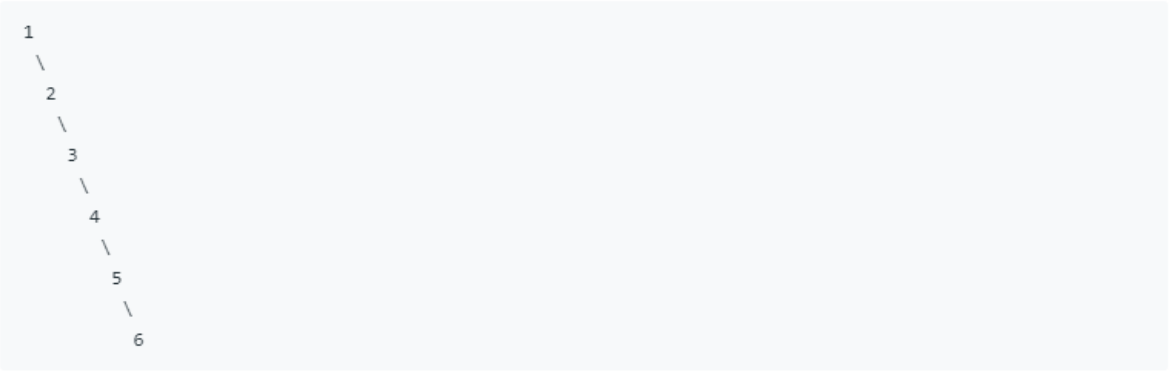
[114. 二叉树展开为链表]

给定一个二叉树，原地将它展开为链表。

例如，给定二叉树



将其展开为：



```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def flatten(self, root):
        """
        :type root: TreeNode
        :rtype: None Do not return anything, modify root in-place instead.
        """
        if not root:
            return
        #左子树拉直
        self.flatten(root.left)
        #右子树拉直
        self.flatten(root.right)
        #记录右子树
        tmp=root.right
        #左子树转移
        root.right=root.left
        #左子树置null
        root.left=None
        #找到最右点
        while root.right!=None:
            root=root.right
        #把之前的右串拼接上
        root.right=tmp
```

12.二叉树直径

543. 二叉树的直径

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过根结点。

示例：
给定二叉树

```
          1
         / \
        2   3
       / \
      4   5
```

返回 **3**, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def __init__(self):
        self.max=0
    def diameterOfBinaryTree(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        self.maxDepth(root)
        return self.max
    #左子树最大深度+右子树最大深度
    def maxDepth(self,node): #求最大深度
        if node==None:
            return 0
        left=self.maxDepth(node.left)
        right=self.maxDepth(node.right)
        self.max=max(left+right,self.max)  #记录
        return max(left,right)+1
```

13.二叉搜索树的搜索

700. 二叉搜索树中的搜索

给定二叉搜索树（BST）的根节点和一个值。 你需要在BST中找到节点值等于给定值的节点。 返回以该节点为根的子树。 如果节点不存在，则返回 NULL。

例如，

```
给定二叉搜索树：

        4
       / \
      2   7
     / \
    1   3

和值：2
```

你应该返回如下子树:

```
      2
     / \
    1   3
```

在上述示例中，如果要找的值是 `5`，但因为没有节点值为 `5`，我们应该返回 `NULL`。

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def searchBST(self, root, val):
        """
        :type root: TreeNode
        :type val: int
        :rtype: TreeNode
        """
        if not root:
            return None
        if val==root.val:
            return root
        elif val<root.val:
            return self.searchBST(root.left,val)
        else:
            return self.searchBST(root.right,val)
```

链表

1.反转链表

题号：14 反转链表 206. 反转链表1

题目描述：输入一个链表，反转链表后，输出新链表的表头。

思路：主要为了防止链表在i点断开，需要记录前一个指针，当前指针，后一个指针

当前节点是pCur，pPre为当前节点的前一节点，pNext为当前节点的下一节点 需要pPre和pNext的目的是让当前节点从pPre->pCur->PNext1->pNext2变成pPre<-pCur PNext1->pNext2 即pPre让节点可以反转所指方向，但反转之后如果不用PNext节点保存next1节点的话，此单链表就此断开了 所以需要用到pPre和pNext两个节点 1->2->3->4->5 1<-2<-3 4->5

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    # 返回ListNode
    def ReverseList(self, pHead):
        # write code here
        if pHead==None or pHead.next==None:
            return pHead

        pPre=None
        pCur=pHead
        pNext=None

        while(pCur!=None):
            pNext=pCur.next  ##记录下一个节点
            pCur.next=pPre   ##改变指向

            pPre=pCur        ##移动一个指针
            pCur=pNext
        return pPre
```

反转链表 2

题号：92 反转链表2

题目描述：反转从位置 *m* 到 *n* 的链表。请使用一趟扫描完成反转。

说明: 1 ≤ m ≤ n ≤ 链表长度。

示例:

输入: 1->2->3->4->5->NULL, m = 2, n = 4 输出: 1->4->3->2->5->NULL

题目思路：需要记录m-1，m处的指针。中间部分正常反转，然后改变next方向

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def reverseBetween(self, head, m, n):
        """
        :type head: ListNode
        :type m: int
        :type n: int
        :rtype: ListNode
        """
        if m==n:
            return head
        mCur=None #m值
        mPre=None #m前一个值
        newHead=ListNode(0)
        newHead.next=head
```

```
        cur=newHead
        i=0
        while i<m:
            mPre=cur
            cur=cur.next
            i+=1
        mCur=cur
        pre=cur
        cur=cur.next

        nNext=None #n的下一个值
        while i<n:  #开始反转
            nNext=cur.next
            cur.next=pre

            pre=cur
            cur=nNext
            i+=1
        mPre.next=pre  #第n个节点
        mCur.next=nNext #第n+1个节点
        return newHead.next
```

2.从尾到头打印链表

题号：6 剑指offer

题目描述：输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。

思路：栈的思想，先入后出

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # 返回从尾部到头部的列表值序列，例如[1,2,3]
    def printListFromTailToHead(self, listNode):
        # write code here
        ## 入栈，出栈
        stack1=[]
        stack2=[]
        if listNode==None:
            return stack2
        p=listNode
        while p!=None:
            stack1.append(p.val)
            p=p.next
        while len(stack1)!=0:
            stack2.append(stack1.pop())
        return stack2
```

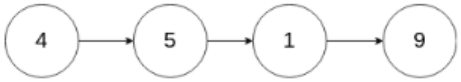
3.删除链表中的节点

题号：18 删除链表中的节点 237. 删除链表中的节点

237.删除链表中的节点

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为:



示例 1:

```
输入: head = [4,5,1,9], node = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2:

```
输入: head = [4,5,1,9], node = 1
输出: [4,5,9]
解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

说明:

- 链表至少包含两个节点。
- 链表中所有节点的值都是唯一的。
- 给定的节点为非末尾节点并且一定是链表中的一个有效节点。
- 不要从你的函数中返回任何结果。

注意：只给了要求被删除的节点

思路：把要删除那个节点的值换成下一个节点值，next也换next.next

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def deleteNode(self, node):
        """
        :type node: ListNode
        :rtype: void Do not return anything, modify node in-place instead.
        """
        ## 只给要删除的节点(非尾节点)
        if node.next!=None:
            node.val=node.next.val
            if node.next.next==None:
                node.next=None
            else:
                node.next=node.next.next
```

18 删除链表的节点

在O(1)时间内删除链表节点。给定单向链表的头指针和一个节点指针，定义一个函数在O(1)时间内删除该节点。

思路：(1)O(n)遍历找到要删除的节点的前一个节点，再把下一个节点链接过来。。

（2）O(1)把要删除的节点的下一个节点的内容复制过来，再把下一个节点的下一个节点链接过来。

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    def deleteNode(self, pHead, pDNode):
        ##只有一个节点
        if pHead==pDNode:
            pHead==None
            return pHead
        ##要删除是尾节点，只能顺序查找，把前一个节点的下一个节点指向None
        if pDNode.next==None:
            pNode=pHead
            while(pNode.next!=pDNode):
                pNode=pNode.next
            pNode.next=None
            return pHead
        pDNode.val=pDNode.next.val
        if pDnode.next.next==None:
                pDnode.next=None
            else:
                pDnode.next=pDnode.next.next
        return pHead
```

4.删除列表中的重复节点

83删除排序链表中的重复元素1

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1:

```
输入: 1->1->2
输出: 1->2
```

示例 2:

```
输入: 1->1->2->3->3
输出: 1->2->3
```

思路：保留重复的节点，两指针

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        #保留重复节点
        if head==None or head.next==None:
            return head
```

```
        cur=head
        nex=None
        while cur!=None and cur.next!=None:
            nex=cur.next
            if cur.val==nex.val: #当前值和下一个值相同
                while(nex!=None and cur.val==nex.val):#继续搜索
                    nex=nex.next
            cur.next=nex
            cur=nex
        return head
```

82.删除排序链表中的重复元素 II

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 *没有重复出现* 的数字。

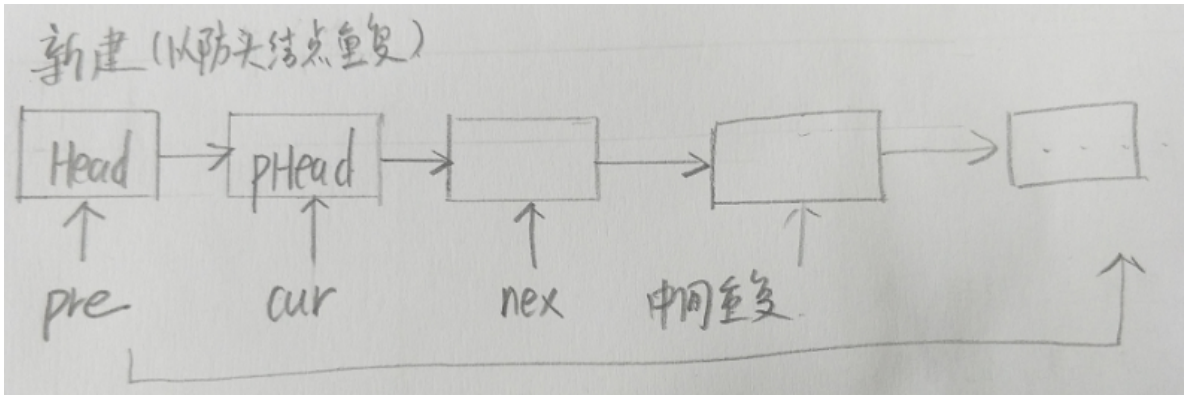
示例 1:

```
输入: 1->2->3->3->4->4->5
输出: 1->2->5
```

示例 2:

```
输入: 1->1->1->2->3
输出: 2->3
```

思路：重复的点不保留，三指针法




```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        #重复链表不保留
        #三指针
        if head==None or head.next==None:
            return head
        newhead=ListNode(0)
        newhead.next=head
        pre=newhead
        cur=head
```

```
        nex=None
        while cur!=None and cur.next!=None:
            nex=cur.next
            if cur.val==nex.val: #找到第一个重复节点
                while nex!=None and cur.val==nex.val: #重复查找
                    nex=nex.next
                pre.next=nex
                cur=nex
            else:
                pre=cur
                cur=nex
        return newhead.next
```

5.删除链表中倒数第k个节点

题号 18 删除链表中倒数第k个节点 19. 删除链表的倒数第N个节点

剑指offer,找到倒数k(简单)

题目描述：输入一个链表，输出该链表中倒数第k个结点

			倒数k		
1	2	3	4	5	6

倒数第k个，是正数第n-k+1

思路：(1)把倒数变成正数，先遍历一次，得到n,再遍历n-k+1步

(2) 快慢指针问题：只需遍历一次，两个指针，如果一个走到最后一个，一个走到了倒数第k个，则两者相差n-(n-k+1)=k-1个步长。所以一个先走k-1步皆可。(注意：k不能比链表长度大)

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def FindKthToTail(self, head, k):
        # write code here
        if head==None or k==0:
            return None
        ##两个相差k-1的指针
        pAhead=head
        pBehind=None
        ##第一个指针走k-1步
        for i in range(0,k-1,1):
            if(pAhead.next!=None):  ##保证k不能比链表长度长
                pAhead=pAhead.next
            else:
                return None
        pBehind=head
        while(pAhead.next!=None):
            pAhead=pAhead.next
            pBehind=pBehind.next
        return pBehind
```

leetcode:删除

给定一个链表，删除链表的倒数第 *n* 个节点，并且返回链表的头结点。

示例：

```
给定一个链表: 1->2->3->4->5, 和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.
```

说明：

给定的 *n* 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

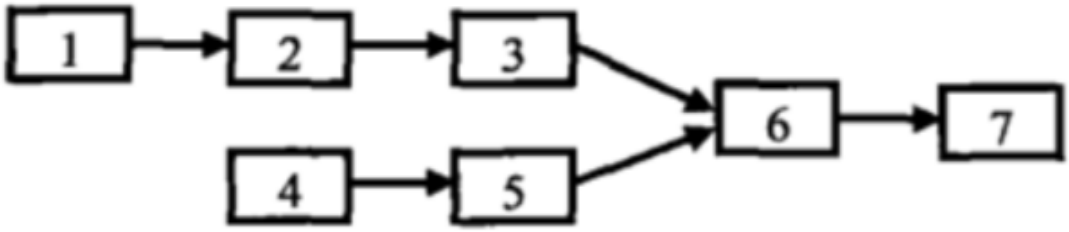
思路：删除倒数第N个节点

```
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def removeNthFromEnd(self, head, n):
        """
        :type head: ListNode
        :type n: int
        :rtype: ListNode
        """
        #要求一遍扫描
        #快慢指针，快走n步
        #倒数第n个数,正数应该是N-n+1
        #两指针相差N-(N-n+1)=n-1 删除n,希望得到其前面一个数，相差n
        if head==None or n==0 or head.next==None:
            return None
        newhead=ListNode(0)
        newhead.next=head    ##防止删除第一个节点
        pFast=newhead
        pSlow=newhead
        for i in range(n):
            if pFast!=None:  #保证n合理
                pFast=pFast.next
            else:
                return None
        while(pFast.next!=None):
            pFast=pFast.next
            pSlow=pSlow.next

        pSlow.next=pSlow.next.next
        return newhead.next
```

6.两个链表的第一个公共节点

题号：52.两个链表的第一个公共节点

题目：输入两个链表，找到它们的第一个公共节点。



思路：(1)暴力解法：在第一个链表上顺序遍历每一个节点，每遍历一个节点，就在第二个链表上顺序遍历每一个节点。找到相等的节点，即为公共节点。时间复杂度O(m*n)。

(2)栈：后进先出。从链表尾部考虑，重合以后节点相同。故从尾部弹出，相同继续，不相同就是公共节点。时间复杂度O(m+n),空间复杂度O(m+n)。

(3)指针法：先各自遍历得到链表的长度，然后长的先走差的长度，再一起走，相同点就是公共节点。

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    def FindFirstCommonNode(self, pHead1, pHead2):
        # write code here
        if pHead1==None or pHead2==None:
            return None
        length1=self.Getlength(pHead1)
        length2=self.Getlength(pHead2)
        if length1>=length2:
            diff=length1-length2
            p1=pHead1
            p2=pHead2
        else:
            diff=length2-length1
            p1=pHead2
            p2=pHead1
        #长的先走
        for i in range(diff):
            p1=p1.next
        #一起走
        while p1!=None and p2!=None and p1!=p2:
            p1=p1.next
            p2=p2.next
        return p1

    def Getlength(self,pHead):
        length=0
        while(pHead!=None):
            length+=1
            pHead=pHead.next
        return length
```

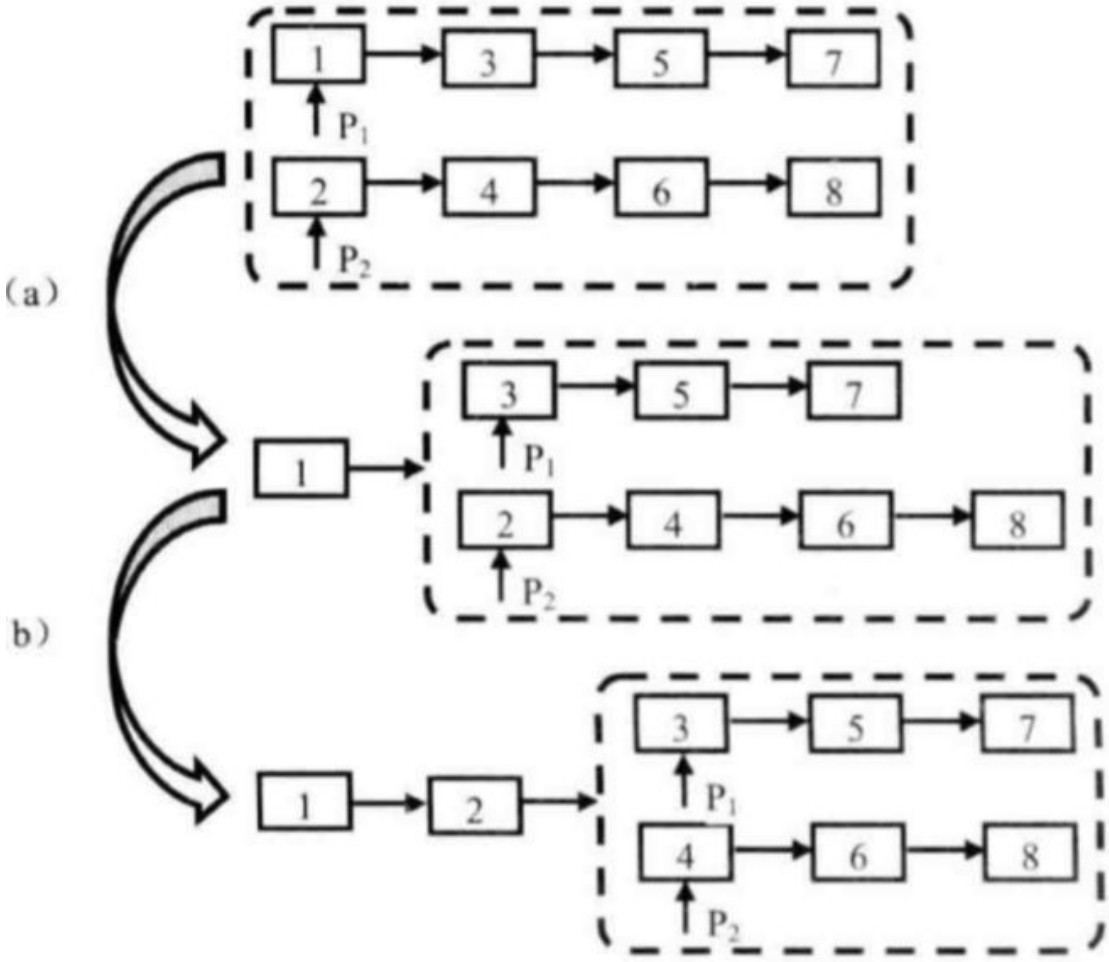
7.合并两个排序链表

将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例:

```
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

思路：两个指针




```python
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def mergeTwoLists(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        newhead=ListNode(0)
        cur=newhead
        p1=l1
        p2=l2
        while p1!=None and p2!=None:
            if p1.val<=p2.val:
                cur.next=p1
                p1=p1.next
```

```
            else:
                cur.next=p2
                p2=p2.next
            cur=cur.next
        if p1==None:
            cur.next=p2
        if p2==None:
            cur.next=p1
        return newhead.next
```

拓展：23. 合并K个排序链表

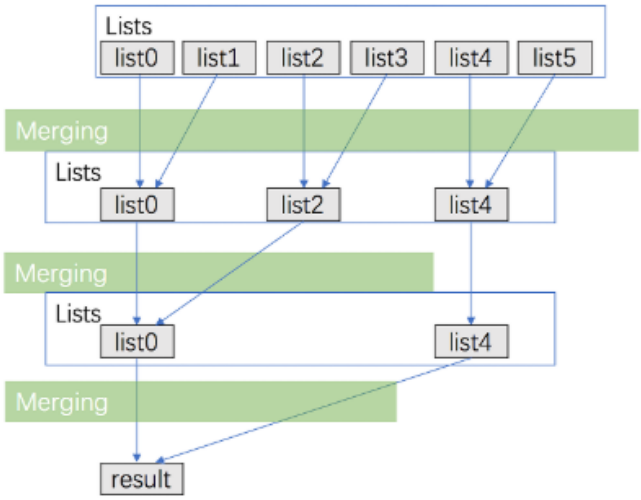
思路：分-治思想,时间复杂度O(NlogK),空间复杂度O(1)

复杂度分析

- 时间复杂度：$O(N \log k)$，其中 k 是链表的数目。
  - 我们可以在 $O(n)$ 的时间内合并两个有序链表，其中 n 是两个链表中的总节点数。
  - 将所有的合并进程加起来，我们可以得到：$O\left(\sum_{i=1}^{log_2 k} N\right) = O(N \log k)$。
- 空间复杂度：$O(1)$
  - 我们可以用 $O(1)$ 的空间实现两个有序链表的合并。

- 将 k 个链表配对并将同一对中的链表合并。
- 第一轮合并以后， k 个链表被合并成了 $\frac{k}{2}$ 个链表，平均长度为 $\frac{2N}{k}$，然后是 $\frac{k}{4}$ 个链表， $\frac{k}{8}$ 个链表等等。
- 重复这一过程，直到我们得到了最终的有序链表。

因此，我们在每一次配对合并的过程中都会遍历几乎全部 N 个节点，并重复这一过程 $\log_2 K$ 次。




```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """
```

```
        ##采用分-治
        length=len(lists)
        if length==0:
            return None
        if length==1:
            return lists[0]
        if length==2:
            return self.partLists(lists[0],lists[1])
        mid=int(length/2)
        left=self.mergeKLists(lists[0:mid])
        right=self.mergeKLists(lists[mid:length])
        result=self.partLists(left,right)
        return result
    def partLists(self,l1,l2):  #两个排序链表合并
        newhead=ListNode(0)
        cur=newhead
        p1=l1
        p2=l2
        while p1!=None and p2!=None:
            if p1.val<=p2.val:
                cur.next=p1
                p1=p1.next
            else:
                cur.next=p2
                p2=p2.next
            cur=cur.next
        if p1==None:
            cur.next=p2
        if p2==None:
            cur.next=p1
        return newhead.next
```

8.有环链表问题

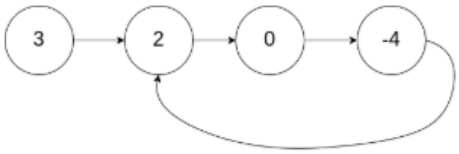
题号：23.链表中环的入口节点 141 142环形链表

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。

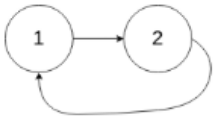
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3：

```
输入：head = [1], pos = -1
输出：false
解释：链表中没有环。
```

思路：快慢指针，快的追上慢的有环

```python
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def hasCycle(self, head):
        """
        :type head: ListNode
        :rtype: bool
        """
        #快慢指针-快的追上慢的指针，有环
        if head==None:
            return False
        pSlow=head.next
        if pSlow==None:
            return False
        pFast=pSlow.next
        while pFast!=None and pSlow!=None:
            if pFast==pSlow:
                return True
            pSlow=pSlow.next
            pFast=pFast.next
```

```
            if pFast!=None:  #多走一步
                pFast=pFast.next
        return False
```

142.链表中环的入口节点

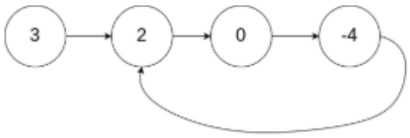
给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 `null`。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。

说明： 不允许修改给定的链表。

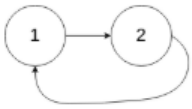
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：tail connects to node index 1
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：tail connects to node index 0
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3：

```
输入：head = [1], pos = -1
输出：no cycle
解释：链表中没有环。
```

思路：(1)先判断有没有环,一快一慢指针，如果相遇，有环，且相遇点在环内

(2)计算环的长度

(3)再次两个指针，一个先走环的长度，则一起走后，相遇点就是入口。

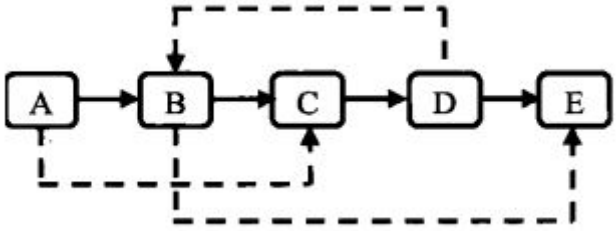
```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def detectCycle(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        #三步法
        #是否有环
        #环的长度
        #环的第一个节点
```

```
        meetnode=self.hasCycle(head)
        if meetnode==None:
            return None
        #计算环的长度，相遇节点在环内
        cycleLength=1
        p1=meetnode.next
        while p1!=meetnode:
            cycleLength+=1
            p1=p1.next
        #环的第一个节点，快先走环的长度，再相遇
        p1=head
        p2=head
        for i in range(cycleLength):
            p1=p1.next
        while p1!=p2:
            p1=p1.next
            p2=p2.next
        return p1

    def hasCycle(self,head):
        if head==None:
            return None
        pSlow=head.next
        if pSlow==None:
            return None
        pFast=pSlow.next
        while pFast!=None and pSlow!=None:
            if pFast==pSlow:
                return pFast
            pSlow=pSlow.next
            pFast=pFast.next
            if pFast!=None:
                pFast=pFast.next
        return None
```

9.复杂链表的复制

题号：35 复杂链表的复制 138复制带随机指针的链表

题目描述：输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）



class RandomListNode:

def **init**(self, x):

self.label = x

self.next = None

self.random = None

(1)直观想法：第一步：复制原始链表的每个节点，用next连接；第二步：设置每个节点random。需要遍历链表两遍，时间复杂度O(n^2)

(2)空间换时间，hashmap,遍历一次的时候，记录random, 空间复杂度O(n)

(3)只需遍历一次，而且不需要额外空间。分三步。

第一步：创建新的节点----长链表

第二步：设置random

第三步：长链表拆分

具体分为三步：

（1）在旧链表中创建新链表，此时不处理新链表的兄弟结点

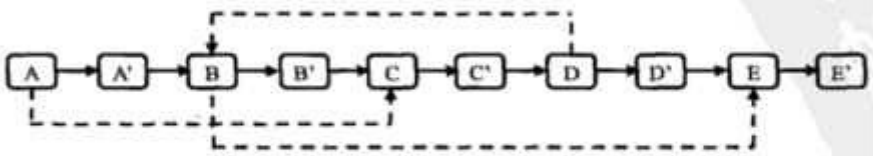


图 4.9 复制复杂链表的第一步

（2）根据旧链表的兄弟结点，初始化新链表的兄弟结点

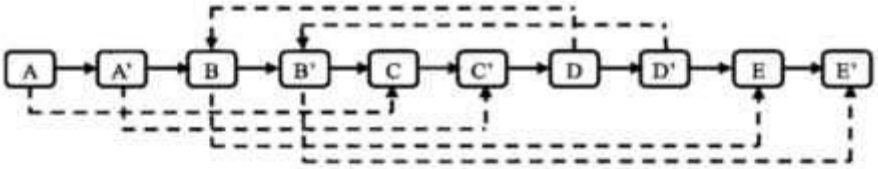


图 4.10 复制复杂链表的第二步

（3）从旧链表中拆分得到新链表

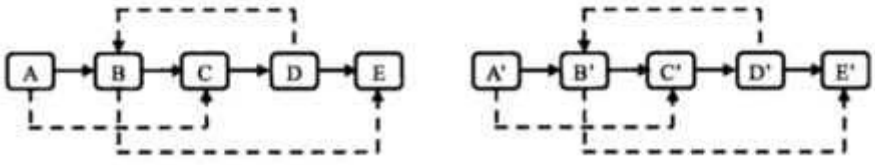


图 4.11 复制复杂链表的第三步


```
# -*- coding:utf-8 -*-
# class RandomListNode:
#     def __init__(self, x):
#         self.label = x
#         self.next = None
#         self.random = None
class Solution:
    # 返回 RandomListNode
    def Clone(self, pHead):
        # write code here
        self.CloneNodes(pHead)
        self.ConnectRandom(pHead)
        return self.DisConnect(pHead)
    def CloneNodes(self,pHead):
```

```
        pNode=pHead
        #新建复杂节点
        while(pNode!=None):
            #复制
            cloneNode=RandomListNode(0)
            cloneNode.label=pNode.label
            cloneNode.next=pNode.next
            cloneNode.random=None
            #链接
            pNode.next=cloneNode
            #下一个节点
            pNode=cloneNode.next
    def ConnectRandom(self,pHead):
        pNode=pHead
        #复制Random
        while(pNode!=None):
            cloneNode=pNode.next
            if pNode.random!=None:
                cloneNode.random=pNode.random.next
            pNode=cloneNode.next
    def DisConnect(self,pHead):
        #复制的头节点
        pNode=pHead
        pCloneHead=None
        pCloneNode=None
        if pNode!=None:
            pCloneHead=pCloneNode=pNode.next
            pNode.next=pCloneNode.next
            pNode=pNode.next
        while(pNode!=None):
            pCloneNode.next=pNode.next
            pCloneNode=pCloneNode.next
            pNode.next=pCloneNode.next
            pNode=pNode.next
        return pCloneHead
```

10.两数相加

题号：2.两数相加

题目描述：逆序

给出两个 **非空** 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 **逆序** 的方式存储的，并且它们的每个节点只能存储 **一位** 数字。

如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例：

```
输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 0 -> 8
原因：342 + 465 = 807
```

思路：初等数学

我们使用变量来跟踪进位，并从包含最低有效位的表头开始模拟逐位相加的过程。

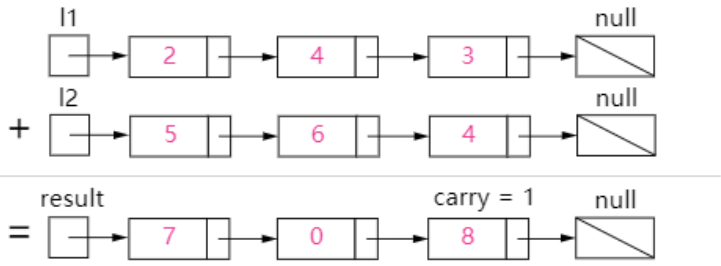


图1，对两数相加方法的可视化：$342 + 465 = 807$，每个结点都包含一个数字，并且数字按位逆序存储。

算法

就像你在纸上计算两个数字的和那样，我们首先从最低有效位也就是列表 $l1$ 和 $l2$ 的表头开始相加。由于每位数字都应当处于 $0 \dots 9$ 的范围内，我们计算两个数字的和时可能会出现“溢出”。例如，$5 + 7 = 12$。在这种情况下，我们会将当前位的数值设置为 2，并将进位 $carry = 1$ 带入下一次迭代。进位 $carry$ 必定是 0 或 1，这是因为两个数字相加（考虑到进位）可能出现的最大和为 $9 + 9 + 1 = 19$。

伪代码如下：

- 将当前结点初始化为返回列表的哑结点。
- 将进位 $carry$ 初始化为 0。
- 将 p 和 q 分别初始化为列表 $l1$ 和 $l2$ 的头部。
- 遍历列表 $l1$ 和 $l2$ 直至到达它们的尾端。
    - 将 x 设为结点 p 的值。如果 p 已经到达 $l1$ 的末尾，则将其值设置为 0。
    - 将 y 设为结点 q 的值。如果 q 已经到达 $l2$ 的末尾，则将其值设置为 0。
    - 设定 $sum = x + y + carry$。
    - 更新进位的值，$carry = sum / 10$。
    - 创建一个数值为 $(sum \bmod 10)$ 的新结点，并将其设置为当前结点的下一个结点，然后将当前结点前进到下一个结点。
    - 同时，将 p 和 q 前进到下一个结点。
- 检查 $carry = 1$ 是否成立，如果成立，则向返回列表追加一个含有数字 1 的新结点。
- 返回哑结点的下一个结点。

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        #低位相加
        newhead=ListNode(0)
        pNode=newhead
        p1=l1
        p2=l2
        flag=0
        while p1!=None or p2!=None:
            if p1==None:
                x=0
            else:
                x=p1.val
            if p2==None:
                y=0
```

```
            else:
                y=p2.val
            sum_n=flag+x+y
            flag=sum_n/10
            pNode.next=ListNode(sum_n%10)
            pNode=pNode.next
            if p1!=None:
                p1=p1.next
            if p2!=None:
                p2=p2.next
        if flag:
            pNode.next=ListNode(flag)
        return newhead.next
```

题目：正序

给定两个非空链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储单个数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

进阶:

如果输入链表不能修改该如何处理？换句话说，你不能对列表中的节点进行翻转。

示例:

输入: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
输出: 7 -> 8 -> 0 -> 7

思路：借用栈的先进后出变为逆序

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        #利用栈解决
        p1=l1
        p2=l2
        stack1=[]
        stack2=[]
        stack3=[]
        while(p1!=None):
            stack1.append(p1.val)
            p1=p1.next
        while(p2!=None):
            stack2.append(p2.val)
            p2=p2.next
        newhead=ListNode(0)
        pNode=newhead
```

```
        flag=0
        while len(stack1)!=0 or len(stack2)!=0:
            if len(stack1)==0:
                x=0
            else:
                x=stack1.pop()

            if len(stack2)==0:
                y=0
            else:
                y=stack2.pop()
            sum_n=flag+x+y
            flag=sum_n/10
            stack3.append(sum_n%10)
        if flag:
            stack3.append(flag)
        while len(stack3)!=0:
            pNode.next=ListNode(stack3.pop())
            pNode=pNode.next
        return newhead.next
```

11.奇偶链表

题目描述：

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

示例 1:

```
输入: 1->2->3->4->5->NULL
输出: 1->3->5->2->4->NULL
```

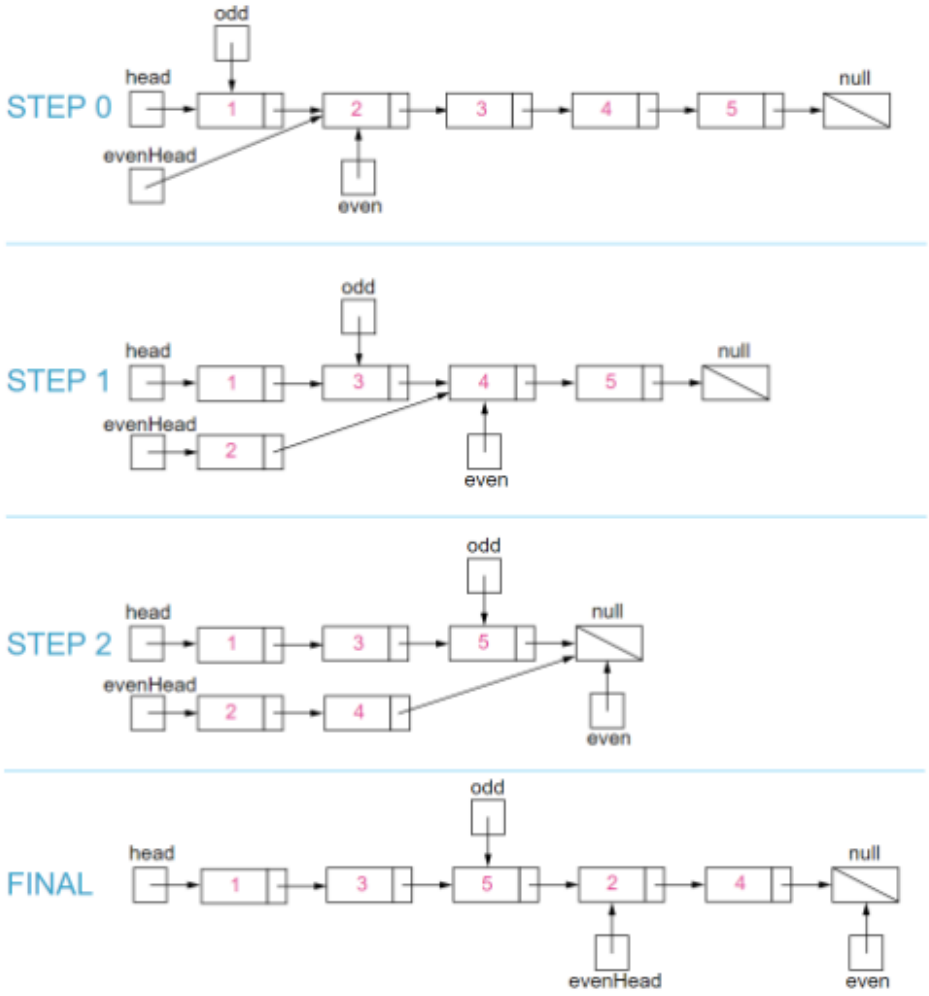
示例 2:

```
输入: 2->1->3->5->6->4->7->NULL
输出: 2->3->6->7->1->5->4->NULL
```

说明:

- 应当保持奇数节点和偶数节点的相对顺序。
- 链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

思路：奇偶分离




```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def oddEvenList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if head==None or head.next==None:
            return head
        podd=head
        pevenhead=peven=podd.next

        while podd.next!=None and peven.next!=None:
                podd.next=peven.next
                podd=podd.next
                peven.next=podd.next
                peven=peven.next
        podd.next=pevenhead
        return head
```

12.回文链表

题号：234.回文链表

题目描述：

请判断一个链表是否为回文链表。

示例 1:

```
输入: 1->2
输出: false
```

示例 2:

```
输入: 1->2->2->1
输出: true
```

进阶:
你能否用 O(n) 时间复杂度和 O(1) 空间复杂度解决此题?

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def isPalindrome(self, head):
        """
        :type head: ListNode
        :rtype: bool
        """
        '''
        #简单粗暴，栈
        if head==None or head.next==None:
            return True
        stack=[]
        pNode=head
        while pNode!=None:
            stack.append(pNode.val)
            pNode=pNode.next
        mid=len(stack)/2
        pNode=head
        for i in range(mid):
            if pNode.val==stack.pop():
                pNode=pNode.next
            else:
                return False
        return True
        '''
        #进阶:快慢指针，找到中点
        if head==None or head.next==None:
            return True
        pFast=head
        pSlow=head
        while pFast!=None:
            pFast=pFast.next
            if pFast!=None: ##奇偶长度
                pFast=pFast.next
            pSlow=pSlow.next
```

```
        #后半部分要反转
        pSlow=self.reverse(pSlow)
        p=head
        while pSlow!=None:
            if pSlow.val==p.val:
                pSlow=pSlow.next
                p=p.next
            else:
                return False
        return True
    def reverse(self,head):
        cur=head
        pre=None
        nex=None
        while cur!=None:
            nex=cur.next
            cur.next=pre

            pre=cur
            cur=nex
        return pre
```

13.排序链表

148. 排序链表

在 O(*n* log *n*) 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1:

```
输入: 4->2->1->3
输出: 1->2->3->4
```

示例 2:

```
输入: -1->5->3->4->0
输出: -1->0->3->4->5
```

在真实的面试中遇到过这道题? 是 否

思路：归并排序

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def sortList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        #归并排序：先二分再合并
        #1.找到中点 2.一分为二 3.排序结合
        #递归终止条件
        if not head or not head.next:
```

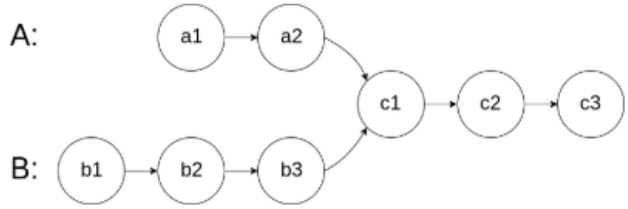
```
            return head
        #找到中点，而且要断开
        pre,slow,fast=head,head,head
        while fast and fast.next:
            pre=slow
            slow=slow.next
            fast=fast.next.next
        pre.next=None #断开
        left=self.sortList(head)
        right=self.sortList(slow)
        return self.merge(left,right)

    def merge(self,p1,p2):
        if not p1:
            return p2
        if not p2:
            return p1
        if p1.val<p2.val:
            head=ListNode(p1.val)
            head.next=self.merge(p1.next,p2)
        else:
            head=ListNode(p2.val)
            head.next=self.merge(p1,p2.next)
        return head
```

14.相交链表

160. 相交链表

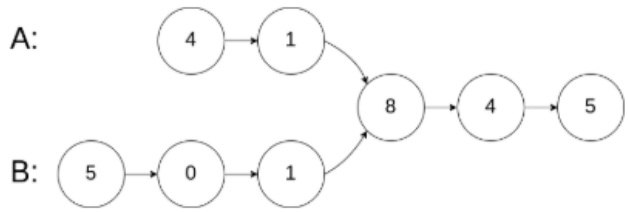
编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

示例 1：



输入： intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出： Reference of the node with value = 8
输入解释： 相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
```

```
#         self.next = None

class Solution(object):
    def getIntersectionNode(self, headA, headB):
        """
        :type head1, head1: ListNode
        :rtype: ListNode
        """
        length1=self.getLength(headA)
        length2=self.getLength(headB)
        if length1>length2:
            diff=length1-length2
            pfast=headA
            pslow=headB
        else:
            diff=length2-length1
            pfast=headB
            pslow=headA

        for i in range(diff):
                pfast=pfast.next

        while pfast!=None and pslow!=None:
            if pfast==pslow:
                return pfast
            pfast=pfast.next
            pslow=pslow.next
        return None

    def getLength(self,head):
        length=0
        while head!=None:
            length+=1
            head=head.next
        return length
```

栈与队列

1.每日温度

[739. 每日温度]

根据每日 气温 列表，请重新生成一个列表，对应位置的输入是你需要再等待多久温度才会升高超过该日的天数。如果之后都不会升高，请在该位置用 0 来代替。

例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。

提示： 气温 列表长度的范围是 [1, 30000]。每个气温的值的均为华氏度，都是在 [30, 100] 范围内的整数。

思路：维护一个递减栈，栈记录的是下标

如果新的元素大于栈顶：pop,

否则入栈

```
class Solution(object):
    def dailyTemperatures(self, T):
        """
        :type T: List[int]
```

```
        :rtype: List[int]
        """
        #维护递减栈，记录的是下标
        stack=[]
        res=[0]*len(T)
        for key,value in enumerate(T):
            if stack:
                while stack and T[stack[-1]]<value:
                    res[stack[-1]]=key-stack[-1]
                    stack.pop()
            stack.append(key)
        return res
```

贪婪算法

贪婪算法：用数学方式证明每一步选择都是最优的。时间，空间复杂度O(1)

1.股票买卖

122. 买卖股票的最佳时机 II

给定一个数组，它的第 *i* 个元素是一支给定股票第 *i* 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 7
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
     随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6-3 = 3 。
```

示例 2:

```
输入: [1,2,3,4,5]
输出: 4
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
     注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
     因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
```

示例 3:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下，没有交易完成，所以最大利润为 0。
```

思路：多次交易~

贪心算法，一次遍历，只要今天价格小于明天价格就在今天买入然后明天卖出，时间复杂度`O(n)`

```
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        #贪婪算法
```

```
        result=0
        if len(prices)<=0:
            return result
        for i in range(1,len(prices)):
            if prices[i]>prices[i-1]:
                result+=prices[i]-prices[i-1]
        return result
```

2.任务调度器

621. 任务调度器

给定一个用字符数组表示的 CPU 需要执行的任务列表。其中包含使用大写的 A - Z 字母表示的26 种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。CPU 在任何一个单位时间内都可以执行一个任务，或者在待命状态。

然而，两个**相同种类**的任务之间必须有长度为 **n** 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。

你需要计算完成所有任务所需要的**最短时间**。

示例 1：

```
输入：tasks = ["A","A","A","B","B","B"], n = 2
输出：8
执行顺序：A -> B -> (待命) -> A -> B -> (待命) -> A -> B.
```

注：

1. 任务的总个数为 [1, 10000]。
2. n 的取值范围为 [0, 100]。

```
class Solution(object):
    def leastInterval(self, tasks, n):
        """
        :type tasks: List[str]
        :type n: int
        :rtype: int
        """
        #贪心算法：(maxtimes-1)*(n+1)+number_max
        length=len(tasks)
        nums=[0]*26
        for i in tasks:
            nums[ord(i)-ord('A')]+=1
        numsB=sorted(nums)
        maxtimes=numsB[25]
        number_max=0
        for i in range(25,-1,-1):
            if numsB[i]==maxtimes:
                number_max+=1
            if numsB[i]<maxtimes:
                break
        result=(maxtimes-1)*(n+1)+number_max
        return result if result >= length else length
```

动态规划

动态规划特征： ①求一个问题的最优解； ②整体的问题的最优解是依赖于各个子问题的最优解； ③小问题之间还有相互重叠的更小的子问题； ④从上往下分析问题，从下往上求解问题；

应用动态规划时候，每一步都可能面临若干选择。时间复杂度O（n^2），空间复杂度O(n)

1.爬楼梯

70. 爬楼梯

假设你正在爬楼梯。需要 *n* 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

注意：给定 *n* 是一个正整数。

示例 1：

```
输入： 2
输出： 2
解释： 有两种方法可以爬到楼顶。
1.  1 阶 + 1 阶
2.  2 阶
```

示例 2：

```
输入： 3
输出： 3
解释： 有三种方法可以爬到楼顶。
1.  1 阶 + 1 阶 + 1 阶
2.  1 阶 + 2 阶
3.  2 阶 + 1 阶
```

```python
class Solution(object):
    def climbStairs(self, n):
        """
        :type n: int
        :rtype: int
        """
        #一维DP
        dp=[0]*(n+1)
        dp[0]=0
        dp[1]=1
        if n==1:
            return dp[1]
        dp[2]=2
        for i in range(3,n+1):
            dp[i]=dp[i-1]+dp[i-2]
        return dp[n]
```

青蛙变态跳台阶问题

题目描述：一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

思路：数学归纳法

```python
# -*- coding:utf-8 -*-
class Solution:
    def jumpFloorII(self, number):
        # write code here
        if number<=0:
            return 0
        else:
            return pow(2,number-1)
```

矩阵覆盖

题目描述：我们可以用2x1的小矩形横着或者竖着去覆盖更大的矩形。请问用n个2x1的小矩形无重叠地覆盖一个2*n的大矩形，总共有多少种方法？

思路：

```
# -*- coding:utf-8 -*-
class Solution:
    def rectCover(self, number):
        # write code here
        ##斐波那契数列
        if number<=2:
            return number
        else:
            firstNumber=1
            twoNumber=2
            for i in range(3,number+1):
                rectnumber=twoNumber+firstNumber
                firstNumber=twoNumber
                twoNumber=rectnumber
            return rectnumber
```

2.最大子序和

53. 最大子序和

给定一个整数数组 `nums` ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例:

```
输入: [-2,1,-3,4,-1,2,1,-5,4],
输出: 6
解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。
```

进阶:

如果你已经实现复杂度为 O(*n*) 的解法，尝试使用更为精妙的分治法求解。

```
class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #一维DP
        #dp[i]前i处的最大和
        dp=[0]*len(nums)
        dp[0]=nums[0]
        dpmax=dp[0]
        for i in range(1,len(nums)):
            if nums[i]<dp[i-1]+nums[i]:
                dp[i]=dp[i-1]+nums[i]
            else:
                dp[i]=nums[i]
            if dp[i]>dpmax:
                dpmax=dp[i]
        return dpmax
```

3.使用最小花费爬楼梯

746. 使用最小花费爬楼梯

数组的每个索引做为一个阶梯，第 `i` 个阶梯对应着一个非负数的体力花费值 `cost[i]`(索引从0开始)。

每当你爬上一个阶梯你都要花费对应的体力花费值，然后你可以选择继续爬一个阶梯或者爬两个阶梯。

您需要找到达到楼层顶部的最低花费。在开始时，你可以选择从索引为 0 或 1 的元素作为初始阶梯。

示例 1:

```
输入: cost = [10, 15, 20]
输出: 15
解释: 最低花费是从cost[1]开始，然后走两步即可到阶梯顶，一共花费15。
```

示例 2:

```
输入: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]
输出: 6
解释: 最低花费方式是从cost[0]开始，逐个经过那些1，跳过cost[3]，一共花费6。
```

注意:

1. `cost` 的长度将会在 `[2, 1000]`。
2. 每一个 `cost[i]` 将会是一个Integer类型，范围为 `[0, 999]`。

```python
class Solution(object):
    def minCostClimbingStairs(self, cost):
        """
        :type cost: List[int]
        :rtype: int
        """
        #dp[i]代表到i位置的最小花费
        #状态转移：dp[i]=min(dp[i-1],dp[i-2])+cost[i]
        n=len(cost)
        dp=[0]*n
        dp[0]=cost[0]
        dp[1]=cost[1]
        for i in range(2,n):
            dp[i]=min(dp[i-1],dp[i-2])+cost[i]
        return min(dp[n-2],dp[n-1])
```

4.比特位计算

338. 比特位计数

给定一个非负整数 **num**。对于 **0 ≤ i ≤ num** 范围中的每个数字 **i** ，计算其二进制数中的 1 的数目并将它们作为数组返回。

示例 1:

```
输入: 2
输出: [0,1,1]
```

示例 2:

```
输入: 5
输出: [0,1,1,2,1,2]
```

进阶:

- 给出时间复杂度为**O(n*sizeof(integer))**的解答非常容易。但你可以在线性时间**O(n)**内用一趟扫描做到吗？
- 要求算法的空间复杂度为**O(n)**。
- 你能进一步完善解法吗？要求在C++或任何其他语言中不使用任何内置函数（如 C++ 中的 **__builtin_popcount**）来执行此操作。

```python
class Solution(object):
    def countBits(self, num):
        """
        :type num: int
        :rtype: List[int]
        """
        #找规律：
        #偶数：dp[i]=dp[i//2]
        #奇数：dp[i]=dp[i//2]+1
        dp=[0]*(num+1)
        for i in range(1,num+1):
            if i%2==0:
                dp[i]=dp[i//2]
            else:
                dp[i]=dp[i//2]+1
        return dp
```

5.大礼包

638. 大礼包

在LeetCode商店中， 有许多在售的物品。

然而，也有一些大礼包，每个大礼包以优惠的价格捆绑销售一组物品。

现给定每个物品的价格，每个大礼包包含物品的清单，以及待购物品清单。请输出**确切**完成待购清单的最低花费。

每个大礼包的由一个数组中的一组数据描述，最后一个数字代表大礼包的价格，其他数字分别表示内含的其他种类物品的数量。

任意大礼包可无限次购买。

示例 1:

```
输入: [2,5], [[3,0,5],[1,2,10]], [3,2]
输出: 14
解释:
有A和B两种物品，价格分别为¥2和¥5。
大礼包1，你可以以¥5的价格购买3A和0B。
大礼包2， 你可以以¥10的价格购买1A和2B。
你需要购买3个A和2个B， 所以你付了¥10购买了1A和2B（大礼包2），以及¥4购买2A。
```

示例 2:

```
输入: [2,3,4], [[1,1,0,4],[2,2,1,9]], [1,2,1]
输出: 11
解释:
A，B，C的价格分别为¥2，¥3，¥4.
你可以用¥4购买1A和1B，也可以用¥9购买2A，2B和1C。
你需要买1A，2B和1C，所以你付了¥4买了1A和1B（大礼包1），以及¥3购买1B， ¥4购买1C。
你不可以购买超出待购清单的物品，尽管购买大礼包2更加便宜。
```

说明:

1. 最多6种物品， 100种大礼包。
2. 每种物品，你最多只需要购买6个。
3. 你不可以购买超出待购清单的物品，即使更便宜。

思路：DFS

状态转移：curprice=self.shoppingOffers(price,special,needs_remain)+m_bag[n]

```python
class Solution(object):
    def shoppingOffers(self, price, special, needs):
        """
        :type price: List[int]
        :type special: List[List[int]]
        :type needs: List[int]
        :rtype: int
        """
        if not needs:
            return 0
        #定义minprice
        minprice=0
        n=len(needs)
        #全部单价购买
        for i in range(n):
            minprice+=price[i]*needs[i]
        #大礼包分类购买
        for m_bag in special:
            if self.check(m_bag,needs): #可以购买礼包
                needs_remain=[0]*n
                for i in range(n):
                    needs_remain[i]=needs[i]-m_bag[i]
                #当前花费=剩余需求的最小花费+本次大礼包的钱
```

```
curprice=self.shoppingOffers(price,special,needs_remain)+m_bag[n]
                #记录最少花费
                minprice=min(minprice,curprice)
        return minprice

    #大礼包的数量不可以超出需求
    def check(self,bag,needs):
        n=len(needs)
        for i in range(n):
            if bag[i]>needs[i]:
                return False
        return True
```

6.最小路径和

64. 最小路径和

给定一个包含非负整数的 *m* x *n* 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明： 每次只能向下或者向右移动一步。

示例:

```
输入:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
输出: 7
解释: 因为路径 1→3→1→1→1 的总和最小。
```

```
class Solution(object):
    def minPathSum(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        #二维DP,填表
        #只能向下或向右：dp[i][j]=min(dp[i-1][j],dp[i][j-1])+grid[i][j]
        row=len(grid) #行
        col=len(grid[0]) #列
        dp=[[0 for j in range(col)] for i in range(row)]
        #初始化行列
        dp[0][0]=grid[0][0]
        for i in range(1,row):
            dp[i][0]=dp[i-1][0]+grid[i][0]
        for j in range(1,col):
            dp[0][j]=dp[0][j-1]+grid[0][j]

        for i in range(1,row):
            for j in range(1,col):
                dp[i][j]=min(dp[i-1][j],dp[i][j-1])+grid[i][j]
        return dp[-1][-1]
```

7.不同的二叉搜索树

96. 不同的二叉搜索树

给定一个整数 *n*，求以 1 ... *n* 为节点组成的二叉搜索树有多少种?

示例:

```
输入: 3
输出: 5
解释:
给定 n = 3, 一共有 5 种不同结构的二叉搜索树:

   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

思路:

标签：动态规划 假设n个节点存在二叉排序树的个数是G(n)，令f(i)为以i为根的二叉搜索树的个数，则G(n) = f(1) + f(2) + f(3) + f(4) + ... + f(n)G(n)=f(1)+f(2)+f(3)+f(4)+...+f(n)

当i为根节点时，其左子树节点个数为i-1个，右子树节点为n-i，则 f(i) = G(i-1)*G(n-i)f(i)=G(i−1)∗G(n−i)

综合两个公式可以得到 卡特兰数 公式 G(n) = G(0)*G(n-1)*+*G(1)*(n-2)+...+G(n-1)*G(0)G(n)=G(0)∗G(n−1)+G(1)∗(n−2)+...+G(n−1)∗G(0)

```python
class Solution(object):
    def numTrees(self, n):
        """
        :type n: int
        :rtype: int
        """
        #DFS
        #G(n)=f(1)+f(2)+...+f(n)
        #f(j)=G(j-1)*G(n-j)
        dp=[0]*(n+1)
        dp[0]=1
        dp[1]=1
        for i in range(2,n+1):
            for j in range(1,i+1):
                dp[i]+=dp[j-1]*dp[i-j]
        return dp[-1]
```

8.最长回文子串

5. 最长回文子串

给定一个字符串 `s`，找到 `s` 中最长的回文子串。你可以假设 `s` 的最大长度为 1000。

示例 1:

```
输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。
```

示例 2:

```
输入: "cbbd"
输出: "bb"
```

```python
class Solution(object):
    def countSubstrings(self, s):
        """
        :type s: str
        :rtype: int
        """
        #中心扩展法
        if len(s)<1:
            return 0
        result=0
        for i in range(len(s)):
            res1=self.count(s,i,i)     #奇数串
            res2=self.count(s,i,i+1)  #偶数串
            result+=res1
            result+=res2
        return result
    def count(self,s,p1,p2):
        num=0
        while p1>=0 and p2<len(s) and s[p1]==s[p2]:
            num+=1
            p1-=1
            p2+=1
        return num
```

9.编辑距离

72. 编辑距离

给定两个单词 *word1* 和 *word2*，计算出将 *word1* 转换成 *word2* 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

1. 插入一个字符
2. 删除一个字符
3. 替换一个字符

示例 1:

```
输入: word1 = "horse", word2 = "ros"
输出: 3
解释:
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

示例 2:

```
输入: word1 = "intention", word2 = "execution"
输出: 5
解释:
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')
```

思路：

动态规划：

`dp[i][j]` 代表 `word1` 到 `i` 位置转换成 `word2` 到 `j` 位置需要最少步数

所以，

当 `word1[i] == word2[j]`， `dp[i][j] = dp[i-1][j-1]`；

当 `word1[i] != word2[j]`， `dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1`

其中，`dp[i-1][j-1]` 表示替换操作，`dp[i-1][j]` 表示删除操作，`dp[i][j-1]` 表示插入操作。

注意，针对第一行，第一列要单独考虑，我们引入 `''` 下图所示：

	''	r	o	s
''	0	1	2	3
h	1			
o	2			
r	3			
s	4			
e	5			

第一行，是 `word1` 为空变成 `word2` 最少步数，就是插入操作

第一列，是 `word2` 为空，需要的最少步数，就是删除操作

```
class Solution(object):
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """
        #dp[i][j]:表示word1中i到word2中j的最短距离
        length1=len(word1)
        length2=len(word2)
        dp=[[0 for j in range(length2+1)] for i in range(length1+1)]
        #边界条件：字母前面加空格
        dp[0][0]=0
        for j in range(1,length2+1):
            dp[0][j]=dp[0][j-1]+1
        for i in range(1,length1+1):
            dp[i][0]=dp[i-1][0]+1
        #状态转移
        for i in range(1,length1+1):
            for j in range(1,length2+1):
                if word1[i-1]==word2[j-1]:
                    dp[i][j]=dp[i-1][j-1]
                else:
                    dp[i][j]=min(min(dp[i-1][j],dp[i][j-1]),dp[i-1][j-1])+1
        return dp[-1][-1]
```

10.戳气球

312. 戳气球

有 n 个气球，编号为0 到 n-1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。每当你戳破一个气球 i 时，你可以获得 nums[left] * nums[i] * nums[right] 个硬币。 这里的 left 和 right 代表和 i 相邻的两个气球的序号。注意当你戳破了气球 i 后，气球 left 和气球 right 就变成了相邻的气球。

求所能获得硬币的最大数量。

说明:

- 你可以假设 nums[-1] = nums[n] = 1，但注意它们不是真实存在的所以并不能被戳破。
- 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

示例:

```
输入: [3,1,5,8]
输出: 167
解释: nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
     coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1   = 167
```

思路：

dp[i][j] 表示戳破 [i+1...j-1] 号气球的最大收益。

假设 k 号气球 (i+1 <= k <= j-1) 是 [i+1...j-1] 中最后一个被戳破的，则

dp[i][j] = max {for k = range(i+1, j -1) nums[i] * nums[k] * nums[j] + dp[i][k] + dp[k][j]}

```
class Solution(object):
    def maxCoins(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #动态规划
        #考虑边界
        nums.insert(0,1)
        nums.append(1)
        n=len(nums)
        #dp[i][j]:代表nums[i+1]到nums[j-1]最大硬币数,k是最后一个戳破的气球
        dp=[[0 for i in range(n)] for j in range(n)]
        #循环是从长度length开始
        for length in range(2,n):
            for i in range(0,n-length):
                j=i+length
                for k in range(i+1,j):
                    dp[i][j]=max(dp[i][j],nums[i]*nums[k]*nums[j]+dp[i][k]+dp[k]
[j])
        return dp[0][n-1]
```

11.乘积最大子序列

152. 乘积最大子序列

给定一个整数数组 `nums` ，找出一个序列中乘积最大的连续子序列（该序列至少包含一个数）。

示例 1:

```
输入: [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。
```

示例 2:

```
输入: [-2,0,-1]
输出: 0
解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。
```

思路：遍历数组时计算当前最大值，不断更新 令imax为当前最大值，则当前最大值为 imax = max(imax * nums[i], nums[i]) 由于存在负数，那么会导致最大的变最小的，最小的变最大的。因此还需要维护当前最小值imin，imin = min(imin * nums[i], nums[i])。当负数出现时则imax与imin进行交换再进行下一步计算 时间复杂度：O(n)

```python
class Solution(object):
    def maxProduct(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #DP
        imax=[0]*len(nums)
        imin=[0]*len(nums)
        imax[0]=nums[0]
        imin[0]=nums[0]
        nmax=nums[0]
        for i in range(1,len(nums)):
            if nums[i]<0:                #出现负数情况
                imax,imin=imin,imax
            imax[i]=max(imax[i-1]*nums[i],nums[i])
            imin[i]=min(imin[i-1]*nums[i],nums[i])
            nmax=max(nmax,imax[i])
        return nmax
```

12.预测赢家

[486. 预测赢家]

给定一个表示分数的非负整数数组。 玩家1从数组任意一端拿取一个分数，随后玩家2继续从剩余数组任意一端拿取分数，然后玩家1拿，……。每次一个玩家只能拿取一个分数，分数被拿取之后不再可取。直到没有剩余分数可取时游戏结束。最终获得分数总和最多的玩家获胜。

给定一个表示分数的数组，预测玩家1是否会成为赢家。你可以假设每个玩家的玩法都会使他的分数最大化。

示例 1:

```
输入：[1, 5, 2]
输出：False
解释：一开始，玩家1可以从1和2中进行选择。
如果他选择2（或者1），那么玩家2可以从1（或者2）和5中进行选择。如果玩家2选择了5，那么玩家1则只剩下1（或者2）可选。
所以，玩家1的最终分数为 1 + 2 = 3，而玩家2为 5。
因此，玩家1永远不会成为赢家，返回 False。
```

示例 2:

```
输入：[1, 5, 233, 7]
输出：True
解释：玩家1一开始选择1。然后玩家2必须从5和7中进行选择。无论玩家2选择了哪个，玩家1都可以选择233。
最终，玩家1（234分）比玩家2（12分）获得更多的分数，所以返回 True，表示玩家1可以成为赢家。
```

注意:

1. 1 <= 给定的数组长度 <= 20.
2. 数组里所有分数都为非负数且不会大于10000000。
3. 如果最终两个玩家的分数相等，那么玩家1仍为赢家。

```python
class Solution(object):
    def PredictTheWinner(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        #DP[i][j]:表示在区间i到j的范围内,比对手多的分数
        n=len(nums)
        dp=[[0 for i in range(n)] for j in range(n)]
        for i in range(n):
            dp[i][i]=nums[i]
        #状态转移：dp[i][j]=max(nums[i]-dp[i+1][j],nums[j]-dp[i][j-1])
        for i in range(n-1,-1,-1):
            for j in range(i+1,n):
                dp[i][j]=max(nums[i]-dp[i+1][j],nums[j]-dp[i][j-1])

        if dp[0][n-1]>=0:
            return True
        else:
            return False
```

13.完全平方数

279. 完全平方数

给定正整数 *n*，找到若干个完全平方数（比如 `1, 4, 9, 16, ...`）使得它们的和等于 *n*。你需要让组成和的完全平方数的个数最少。

示例 1:

```
输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4.
```

示例 2:

```
输入: n = 13
输出: 2
解释: 13 = 4 + 9.
```

```python
class Solution(object):
    def numSquares(self, n):
        """
        :type n: int
        :rtype: int
        """
        #DP:状态转移 dp[n]=1+min(dp[n-1^1],dp[n-2^2],dp[n-3^2]....)
        dp=[0]*(n+1)
        dp[0]=0
        dp[1]=1
        for i in range(2,n+1):
            j=1
            minval=i ##最坏情况
            while j*j<=i:
                minval=min(minval,dp[i-j*j])
                j=j+1
            dp[i]=1+minval
        return dp[n]
```

14.回文子串

[647. 回文子串]

给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被计为是不同的子串。

示例 1:

```
输入: "abc"
输出: 3
解释: 三个回文子串: "a", "b", "c".
```

示例 2:

```
输入: "aaa"
输出: 6
说明: 6个回文子串: "a", "a", "a", "aa", "aa", "aaa".
```

注意:

1. 输入的字符串长度不会超过1000。

```python
class Solution(object):
    def countSubstrings(self, s):
        """
```

```
        :type s: str
        :rtype: int
        """
        #中心扩展法
        if len(s)<1:
            return 0
        result=0
        for i in range(len(s)):
            res1=self.count(s,i,i)    #奇数串
            res2=self.count(s,i,i+1)  #偶数串
            result+=res1
            result+=res2
        return result
    def count(self,s,p1,p2):
        num=0
        while p1>=0 and p2<len(s) and s[p1]==s[p2]:
            num+=1
            p1-=1
            p2+=1
        return num
```

15.数组中最近距离

给定一个数组，数组中含有重复元素，给定两个数字num1和num2，求这两个数字在数组中出现的位置的最小距离。

时间复杂度：O(n)

```
class Solution(object):
    def twoNumDistance(self, nums, number1, number2):
        if not nums:
            return -1
        num1post=-1
        num2post=-1
        minDistance=len(nums)
        for i in range(len(nums)):
            if nums[i]==number1:
                num1post=i
                if num2post!=-1:
                    minDistance=min(minDistance, num1post-num2post)
            if nums[i]==number2:
                num2post=i
                if num1post!=-1:
                    minDistance=min(minDistance, num2post-num1post)
        return minDistance
if __name__=='__main__':
    S=Solution()
    nums=[1,3,2,4,5,6,7,89,0,1,23,4,6,6]
    print(S.twoNumDistance(nums,6,1))
```

16.目标和

494. 目标和

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 - 中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

示例 1:

```
输入: nums: [1, 1, 1, 1, 1], S: 3
输出: 5
解释:

-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1+1-1 = 3

一共有5种方法让最终目标和为3。
```

注意:

1. 数组的长度不会超过20，并且数组中的值全为正数。
2. 初始的数组的和不会超过1000。
3. 保证返回的最终结果为32位整数。

```python
class Solution(object):
    def findTargetSumWays(self, nums, S):
        """
        :type nums: List[int]
        :type S: int
        :rtype: int
        """
        ##问题转换：正号sum(P)，负号sum(N)
        ##sum(P)-sum(N)=S, sum(P)+sum(N)=sum(nums),转化为sum(P)=(S+sum(nums))/2
        ##0-1背包问题
        #dp[i][j]前i个值，和为j最大的方法数
        sum_n = sum(nums)
        if (sum_n+S)%2!=0 or sum_n<S:
            return 0
        return self.sub_set(nums,(S+sum_n)//2)

    def sub_set(self,nums,S):
        dp = [0]*(S+1)
        dp[0] = 1
        for num in nums:
            #每一遍循环得到和值为i的方法数，以次更新
            for i in range(S,num-1,-1):
                dp[i] += dp[i-num]
        return dp[S]
```

17.矩阵连乘的最优乘法顺序

18.计算各个位数不同的数字个数

357. 计算各个位数不同的数字个数

给定一个**非负**整数 n，计算各位数字都不同的数字 x 的个数，其中 $0 \le x < 10^n$ 。

示例:

```
输入：2
输出：91
解释：答案应为除去 11,22,33,44,55,66,77,88,99 外，在 [0,100) 区间内的所有数字。
```

```
class Solution(object):
    def countNumbersWithUniqueDigits(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n==0:
            return 1
        dp=[0]*(n+1)
        dp[0]=1
        dp[1]=10
        num=9
        for i in range(2,n+1):
            num*=(10-i+1)
            dp[i]=num+dp[i-1]
        return dp[n]
```

19.最大正方形

221. 最大正方形

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

示例:

```
输入：

1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0

输出：4
```

```
class Solution(object):
    def maximalSquare(self, matrix):
        """
        :type matrix: List[List[str]]
        :rtype: int
        """
        #dp转化为最大边长问题
        #转移方程：dp[i][j]=min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1
        if matrix==[]:
            return 0
        M=len(matrix)
        N=len(matrix[0])
        dp=[[0 for j in range(N)] for i in range(M)]
        maxsize=0
        #初始化
```

```
        for i in range(M):
            dp[i][0]=int(matrix[i][0])
            maxsize=max(maxsize,dp[i][0])
        for j in range(N):
            dp[0][j]=int(matrix[0][j])
            maxsize=max(maxsize,dp[0][j])
        for i in range(1,M):
            for j in range(1,N):
                if matrix[i][j]=='1':
                    dp[i][j]=min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1
                else:
                    dp[i][j]=0
                maxsize=max(maxsize,dp[i][j])
        return maxsize**2
```

20.拆分字符串

[139. 单词拆分]

给定一个**非空**字符串 *s* 和一个包含**非空**单词列表的字典 *wordDict*，判定 *s* 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

```
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
```

示例 2：

```
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
     注意你可以重复使用字典中的单词。
```

示例 3：

```
输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false
```

```
class Solution(object):
    def wordBreak(self, s, wordDict):
        """
        :type s: str
        :type wordDict: List[str]
        :rtype: bool
        """
        #动态规划：dp[i]代表字符串0-i-1结尾的是否可以拆分
        dp=[0]*(len(s)+1)
        #空串一定可以
        dp[0]=1
        #把以i-1结尾的串分为0~j-1,j~i-1两部分是否在wordDict
        for i in range(1,len(s)+1):
            for j in range(i):
                if dp[j] and s[j:i] in wordDict:
                    dp[i]=1
```

```
                    break
        return dp[len(s)]
```

21.分割等和子集

[416. 分割等和子集]

给定一个**只包含正整数**的**非空**数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

注意:

1. 每个数组中的元素不会超过 100
2. 数组的大小不会超过 200

示例 1:

```
输入: [1, 5, 11, 5]

输出: true

解释: 数组可以分割成 [1, 5, 5] 和 [11].
```

示例 2:

```
输入: [1, 2, 3, 5]

输出: false

解释: 数组不能分割成两个元素和相等的子集.
```

```
class Solution(object):
    def canPartition(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        ##0-1背包
        sum_nums=sum(nums)
        #和为奇数
        if sum_nums % 2==1:
            return False
        target=sum_nums//2
        #dp[i][j] 代表0~i-1和为j的情况
        dp=[[0 for j in range(target+1)] for i in range(len(nums))]
        #初始化
        for j in range(target+1):
            if nums[0]==j:
                dp[0][j]=1
            else:
                dp[0][j]=0
        for i in range(1,len(nums)):
            for j in range(target+1):
                if nums[i]<=j:
                    dp[i][j]=dp[i-1][j-nums[i]] or dp[i-1][j]
```

```
                else:
                    dp[i][j]=dp[i-1][j]
        if dp[-1][-1]==1:
            return True
        else:
            return False
```

位运算

1.汉明距离

461. 汉明距离

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 `x` 和 `y`，计算它们之间的汉明距离。

注意：
$0 \le x, y < 2^{31}$.

示例：

```
输入: x = 1, y = 4

输出: 2

解释:
1   (0 0 0 1)
4   (0 1 0 0)
      ↑   ↑

上面的箭头指出了对应二进制位不同的位置。
```

思路：汉明重量：二进制中1的个数

汉明距离：两个二进制中位数不同的个数

汉明重量计算： **n &= n - 1**

这个操作对比当前操作位高的位没有影响，对低位则完全清零。

拿6（110）来做例子，

第一次 110&101=100，这次操作成功的把从低位起第一个1消掉了，同时计数器加1。

第二次100&011=000，同理又统计了高位的一个1，此时n已变为0，不需要再继续了，于是110中有2个1。

汉明距离计算：

先将两个数进行异或^（相同为0不同为1）运算 再将异或完的数 计算汉明重量，也就是计算有多少个不同的数， 计算共多少个1

```
class Solution(object):
    def hammingDistance(self, x, y):
        """
        :type x: int
        :type y: int
        :rtype: int
        """
        y=x^y
        count=0
        while y:
            count+=1
            y=y&(y-1)
        return count
```

2.汉明距离总和

477. 汉明距离总和

两个整数的 汉明距离 指的是这两个数字的二进制数对应位不同的数量。

计算一个数组中，任意两个数之间汉明距离的总和。

示例:

输入: 4, 14, 2

输出: 6

解释: 在二进制表示中，4表示为0100，14表示为1110，2表示为0010。（这样表示是为了体现后四位之间关系）
所以答案为:
HammingDistance(4, 14) + HammingDistance(4, 2) + HammingDistance(14, 2) = 2 + 2 + 2 = 6.

注意:

1. 数组中元素的范围为从 0 到 10^9。
2. 数组的长度不超过 10^4。

思路：用两两循环的，会超时

转化为各个位上的汉明距离

```
class Solution(object):
    def totalHammingDistance(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #10^9=2^32
        num1=[0]*32
        #统计各个位上1的个数
        for num in nums:
            x=1
            for i in range(32):
                if x&num:      #对应位是否位1
                    num1[i]+=1
                x=x<<1         #左移1位
        result=0
        n=len(nums)
        #各个位数上的汉明距离
        for i in range(32):
            result+=num1[i]*(n-num1[i])
```

```
        return result
```

3.只出现一次的数字

136. 只出现一次的数字

给定一个**非空**整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

```
输入: [2,2,1]
输出: 1
```

示例 2:

```
输入: [4,1,2,1,2]
输出: 4
```

思路：

思路

- 标签：位运算
- 本题根据题意，线性时间复杂度O(n)，很容易想到使用Hash映射来进行计算，遍历一次后结束得到结果，但是在空间复杂度上会达到O(n)，需要使用较多的额外空间
- 既满足时间复杂度又满足空间复杂度，就要提到位运算中的异或运算XOR，主要因为异或运算有以下几个特点：
  - 一个数和0做XOR运算等于本身：a⊕0 = a
  - 一个数和其本身做XOR运算等于0：a⊕a = 0
  - XOR运算满足交换律和结合律：a⊕b⊕a = (a⊕a)⊕b = 0⊕b = b
- 故而在以上的基础条件上，将所有数字按照顺序做抑或运算，最后剩下的结果即为唯一的数字
- 时间复杂度：O(n)，空间复杂度：O(1)

```
class Solution(object):
    def singleNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        #hashmap:空间复杂度O(n)
        #异或：a xor 0 =a a xor a=0
        a=0
        for num in nums:
            a^=num
        return a
```

滑动窗口

1.

滑动窗口题目:

3. 无重复字符的最长子串

30. 串联所有单词的子串

76. 最小覆盖子串

159. 至多包含两个不同字符的最长子串

209. 长度最小的子数组

239. 滑动窗口最大值

567. 字符串的排列

632. 最小区间

727. 最小窗口子序列

回溯法

1.全排列

46. 全排列

> 给定一个**没有重复**数字的序列，返回其所有可能的全排列。
>
> **示例:**
>
> ```
> 输入: [1,2,3]
> 输出:
> [
>   [1,2,3],
>   [1,3,2],
>   [2,1,3],
>   [2,3,1],
>   [3,1,2],
>   [3,2,1]
> ]
> ```

```python
class Solution(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        #回溯法
        if not nums:
            return []
        result=[]
        self.DFS(nums, 0, result)
        return result
```

```
    def DFS(self, nums, index, result):
        #回溯条件
        if index==len(nums)-1:
            result.append(nums[:]) #nums[:] 修改时有效，深拷贝，等同于nums.copy()
        for i in range(index,len(nums)):
            #避免重复
            if nums[index]==nums[i] and index!=i:
                continue
            else:
                nums[index],nums[i]=nums[i],nums[index]
                self.DFS(nums,index+1,result)
                nums[index],nums[i] = nums[i],nums[index]
```

2.括号生成

22. 括号生成

给出 *n* 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且**有效的**括号组合。

例如，给出 *n* = 3，生成结果为：

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

```
class Solution(object):
    def generateParenthesis(self, n):
        """
        :type n: int
        :rtype: List[str]
        """
        #回溯法
        result=[]
        self.generate(n,n,'',result)
        return result
    def generate(self, left, right, string, result):
        if left==right and left==0:
            result.append(string)
        ##右边剩余大于左边
        if right>left:
            self.generate(left, right-1, string+')', result)
        ##放左括号
        if left>0:
            self.generate(left-1, right, string+'(', result)
```

单调栈

1.右侧第一个大的数字

题目：给定一个整型数组，数组元素随机无序的，要求打印出所有元素右边第一个大于该元素的值。

要求时间效率O(n)

思路：stack记录下标，重点

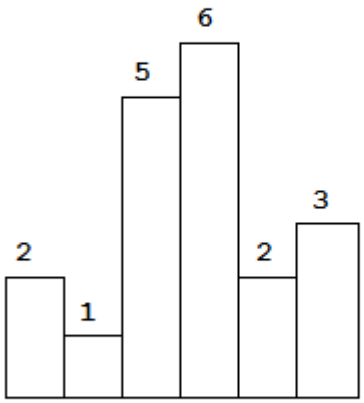
```
class Solution(object):
    def firstMaxNumber(self, nums):
        '''
        要求时间复杂度为：O(n),单调栈，空间复杂度O(n)
        :param nums:
        :return:
        '''
        if not nums or len(nums)<2:
            return -1
        stack=[]
        result=[0]*len(nums)
        stack.append(0)
        i=1
        while i<len(nums):
            if len(stack)!=0 and nums[i]>nums[stack[-1]]:
                result[stack.pop(-1)]=nums[i]
            else:
                stack.append(i)
                i+=1
        while len(stack)!=0:
            result[stack.pop(-1)]=-1
        return result
if __name__=='__main__':
    S=Solution()
    nums=[8, 2, 5, 4, 3, 9, 7, 2, 5]
    print(S.firstMaxNumber(nums))
```

2.柱状图的最大矩形

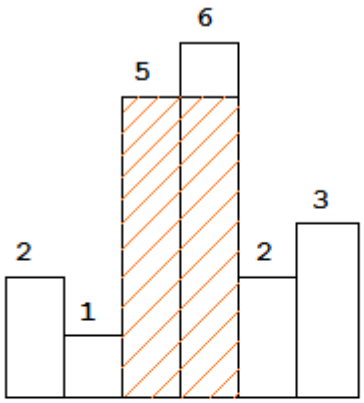
84. 柱状图中最大的矩形

给定 *n* 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 `[2,1,5,6,2,3]`。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 `10` 个单位。

示例:

```
输入: [2,1,5,6,2,3]
输出: 10
```

```python
class Solution(object):
    def largestRectangleArea(self, heights):
        """
        :type heights: List[int]
        :rtype: int
        """
        #单调栈：维持一个单调递增栈.Area=height[top]*(i-pop(-1)-1)
        #前面就是比他小的左边界，右边是他的右边界
        #左右为0
        heights=[0]+heights+[0]
        stack=[]
        area=0
        for i in range(len(heights)):
            while stack and heights[stack[-1]]>heights[i]:
```

```
                top=stack.pop(-1)
                area=max(area,heights[top]*(i-stack[-1]-1))
            stack.append(i)
        return area
```

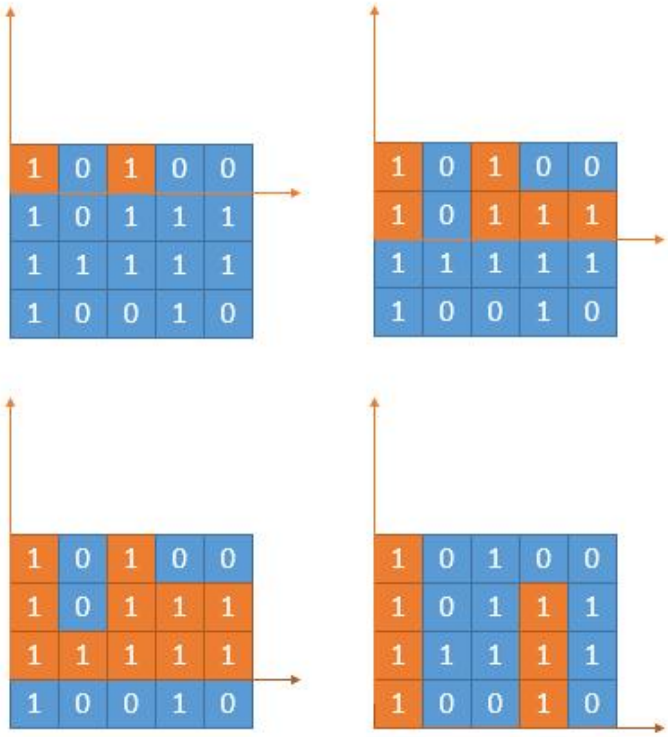
3.最大矩阵

85. 最大矩形

给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

示例:

```
输入:
[
  ["1","0","1","0","0"],
  ["1","0","1","1","1"],
  ["1","1","1","1","1"],
  ["1","0","0","1","0"]
]
输出: 6
```




```
class Solution(object):
    def maximalRectangle(self, matrix):
        """
        :type matrix: List[List[str]]
        :rtype: int
        """
        #本题是每一层的高度传给单调栈的问题
        if len(matrix)==0:
            return 0
        N=len(matrix)
        M=len(matrix[0])
```

```
        maxarea=0
        rows=[0]*M
        for i in range(N):
            for j in range(M):
                if matrix[i][j]=='0':
                    rows[j]=0
                else:
                    rows[j]=rows[j]+1
            maxarea=max(maxarea,self.maxRowRectangle(rows))
        return maxarea


    def maxRowRectangle(self, rows):
        rows=[0]+rows+[0]
        stack=[]
        area=0
        for i in range(len(rows)):
            while stack and rows[stack[-1]]>rows[i]:
                top=stack.pop()
                area=max(area,rows[top]*(i-stack[-1]-1))
            stack.append(i)
        return area
```

其他

1.有效括号

[20. 有效的括号]

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

```
输入: "()"
输出: true
```

示例 2:

```
输入: "()[]{}"
输出: true
```

示例 3:

```
输入: "(]"
输出: false
```

示例 4:

```
输入: "([)]"
输出: false
```

示例 5:

```
输入: "{[]}"
输出: true
```

```
class Solution(object):
    def isValid(self, s):
        """
        :type s: str
        :rtype: bool
        """
        ##栈
        left=['(','{','[']
        right=[')','}',']']
        #对应上
        mapping=dict(zip(left,right))
        stack=[]
        for i in s:
            if not stack or i in left: #如果栈为空或者是左括号
                stack.append(i)
            elif (stack[-1] in left) and (mapping[stack[-1]]==i): #栈最后一个是左括号且与之对应右
                stack.pop()
            else:
                return False
        if not stack:
            return True
        else:
            return False
```

2.大数相加

对于Python而言，自带了大数系统，所以直接相加即可；

顺便说一下Python的数据类型：

（1）None ： 即为空类型

（2）Boolean类型：布尔类型，只有两个值， True（1）和False（0）

（3）int类型：整形 这里的整形对于C语言而言就包含了long long 甚至更大的数字

（4）float类型：浮点型， 同上包含了double甚至更大的

（5）String类型：字符串类型

```
a = int(input())
b = int(input())
print(a+b)
```

3.直线上最多的点数

149. 直线上最多的点数

给定一个二维平面，平面上有 *n* 个点，求最多有多少个点在同一条直线上。

示例 1:

```
输入: [[1,1],[2,2],[3,3]]
输出: 3
解释:
^
|
|        o
|     o
|  o
+------------->
0  1  2  3  4
```

示例 2:

```
输入: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
输出: 4
解释:
^
|
|  o
|     o        o
|        o
|  o        o
+------------------->
0  1  2  3  4  5  6
```

```python
import numpy as np  #除法精度的问题  np.longdouble(1)
class Solution(object):
    def maxPoints(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        #哈希表保存精度
        l=len(points)
        if l<3:
            return l
        result=0
        slope_map={}
        for i in range(l):
            slope_map.clear()
            max_slope=0
            same,vertical=1,0 #相同点，斜率垂直
            for j in range(i+1,l):
                dx=points[i][0]-points[j][0]
                dy=points[i][1]-points[j][1]
                if dx==0 and dy==0:
                    same+=1
                elif dx==0:
                    vertical+=1
                else:
                    slope=(dy*np.longdouble(1))/dx
                    slope_map[slope]=slope_map.get(slope,0)+1
                    max_slope=max(max_slope,slope_map[slope])
            result=max(result,max(max_slope,vertical)+same)
        return result
```