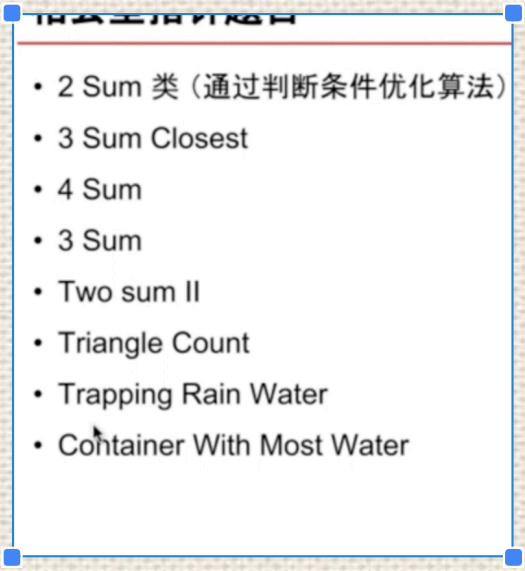
总结：题越做越少

排序时记录下原始位置：可以把原始数组封装成一个两元组，两个一起移动

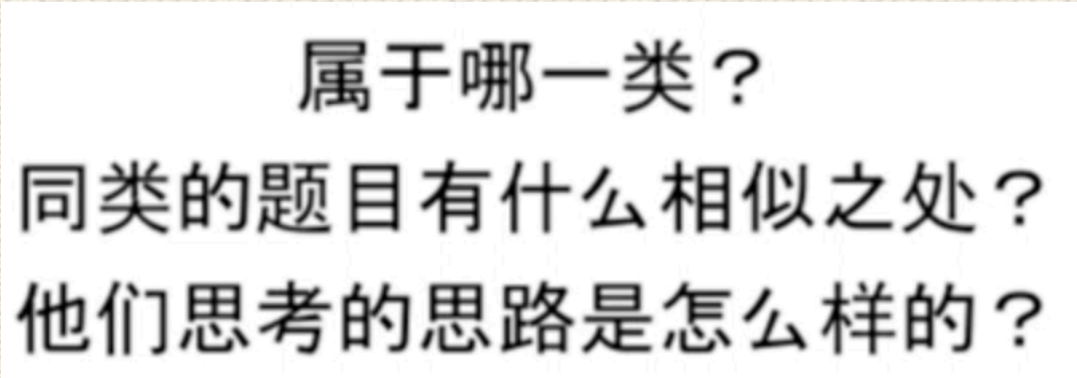
关于查找：

哈希表实现定点查找，找x

平衡二叉树/线段树，查找，大于/小于x




三问




689. Two Sum IV - Input is a BST 中文 English

Given a binary search tree and a number `n`, find two numbers in the tree that sums up to `n`.

## two sum II

这道题首先是最基本的两道做法

-利用哈希表（set）和二叉树的遍历。

-层序遍历

1. 把节点一个一个放到一个list里面。

2. 把节点一个个放到queue里面。外层while循环，控制层遍历。内层for循环，控制每层节点遍历。

-深度优先遍历

采用递归。

1. 函数内写了一个函数要先写该函数，再调用

2. set没有索引

3. python的空是None，空列表[]和None不是一回事。但是空列表[]是False,None 也是False

-先中序遍历排好序，然后两个指针滑动

中序遍历非递归实现自己感觉没问题，但是超时。

-对每个值，用二分法找另一个值。此方法利用了BST特性，但是时间复杂度最高。

1. 注意二叉排序树中没有键值相等的元素

2. 这里二分查找的递归要注意递归的结构。首先错误返回（root==None），然后是正确返回（k==root.val）最后

才是递归查找。自己调用自己也不要忘了类方法加上self。

# trangle count

定一个动两个

**第一步先选择固定哪个。nums[i]是被固定的**

**若nums[lo] + nums[hi] > nums[i],此时 lo<hi<i**

**若nums[lo] + nums[hi] < nums[i],此时 i<lo<hi**

这道题套用two sum II的解法，这里两个指针移动的时候必须是这样的格式

nums[lo] + nums[hi] ? k也就是lo和hi的移动。这样才能在太大时hi向前移动。太小时lo向后移动。其他格式都有歧义

```
def trangleCount(nums):
    list.sort(nums)
    sum = 0
    for i in range(2, len(nums)):
        lo, hi = 0, i-1
        while lo < hi:
            while lo < hi and nums[lo] + nums[hi] <= nums[i]:
                lo += 1
            sum += hi - lo
            hi -= 1
    return sum
```

# three sum closest

找三个数，也是定一个动两个。

动的这一个满足的条件是之和和定的这一个最相近。

这种情况不能指望一次性靠循环挪好。而是走一步看一步，打了hi--,小了lo++,但是没移动一次就要进行一个新的判断。

```
def threeSumClosest(self, numbers, target):
    # write your code here
    list.sort(numbers)
    diff = float('inf')

    for i in range(2, len(numbers)):
        lo, hi = 0, i - 1

        while lo < hi:
            temp = numbers[i] + numbers[lo] + numbers[hi]
            if diff > abs(target - temp):
                diff = abs(target - temp)
                ans = temp
                if diff == 0:
                    return ans

            if numbers[lo] + numbers[hi] < target - numbers[i]:
                lo += 1
            elif numbers[lo] + numbers[hi] > target - numbers[i]:
                hi -= 1

    return ans
```

# three sum smaller

```python
def threeSumSmaller(self, nums, target):
    # Write your code here
    list.sort(nums)
    sum = 0
    for i in range(len(nums)-2):
        lo, hi = i+1, len(nums)-1
        while lo < hi:
            while lo < hi and nums[lo] + nums[hi] >= target - nums[i]:
                hi -= 1
            sum += hi - lo
            lo += 1
    return sum
```

# three sum

-2，1，1 i从1定因为1（2）和1（1）都是1 所以1（2）会被跳过。

```python
def threeSum(self, numbers):
    # write your code here
    list.sort(numbers)
    ans = []
    for i in range(len(numbers)-2):
        if i!=0 and numbers[i] == numbers[i-1]:
            continue
        lo, hi = i+1, len(numbers)-1
        while lo < hi:
            if numbers[lo] + numbers[hi] == -numbers[i]:
                ans.append([numbers[i],numbers[lo], numbers[hi]])
                lo += 1
                while lo < hi and numbers[lo] == numbers[lo-1]:
                    lo +=  1
                hi -= 1
                while lo < hi and numbers[hi] == numbers[hi+1]:
                    hi -=  1
            elif numbers[lo] + numbers[hi] < -numbers[i]:
                lo += 1
                while lo < hi and numbers[lo] == numbers[lo-1]:
                    lo +=  1
            else:
                hi -= 1
                while lo < hi and numbers[hi] == numbers[hi+1]:
                    hi -=  1
    return ans
```

小结：找三个数满足某个条件

1. 定一动二

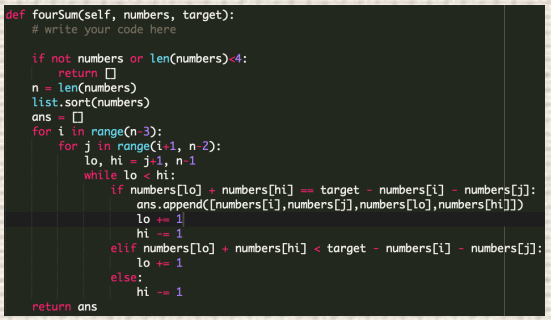
口诀：大于定大，小于等于定小，相近走一步看一步

等于去重：一定二动都得去重，最好走一步看一步

有sort不用hash，没有sort用hash

# 4sum:定2动2（n-sum就是定n-2动2）

下面是未去重版本



```
def fourSum(self, numbers, target):
    # write your code here

    if not numbers or len(numbers)<4:
        return []
    n = len(numbers)
    list.sort(numbers)
    ans = []
    for i in range(n-3):
        for j in range(i+1, n-2):
            lo, hi = j+1, n-1
            while lo < hi:
                if numbers[lo] + numbers[hi] == target - numbers[i] - numbers[j]:
                    ans.append([numbers[i],numbers[j],numbers[lo],numbers[hi]])
                    lo += 1
                    hi -= 1
                elif numbers[lo] + numbers[hi] < target - numbers[i] - numbers[j]:
                    lo += 1
                else:
                    hi -= 1
    return ans
```

去重后

```
def fourSum(self, numbers, target):
    if not numbers or len(numbers)<4:
        return []
    n = len(numbers)
    list.sort(numbers)
    ans = []
    for i in range(n-3):
        if i!=0 and numbers[i] == numbers[i-1]:
            continue
        for j in range(i+1, n-2):
            if j != i+1 and numbers[j] == numbers[j-1]:
                continue
            lo, hi = j+1, n-1
            while lo < hi:
                if numbers[lo] + numbers[hi] == target - numbers[i] - numbers[j]:
                    ans.append([numbers[i],numbers[j],numbers[lo],numbers[hi]])
                    lo += 1
                    while lo<hi and numbers[lo] == numbers[lo-1]:
                        lo += 1
                    hi -= 1
                    while lo<hi and numbers[hi] == numbers[hi+1]:
                        hi -= 1
                elif numbers[lo] + numbers[hi] < target - numbers[i] - numbers[j]:
                    lo += 1
                    while lo<hi and numbers[lo] == numbers[lo-1]:
                        lo += 1
                else:
                    hi -= 1
                    while lo<hi and numbers[hi] == numbers[hi+1]:
                        hi -= 1
    return ans
```

# 4sumII

```python
def fourSumCount(self, A, B, C, D):
    # Write your code here

    di = dict()
    ans = 0
    for i in range(len(A)):
        for j in range(len(B)):
            if A[i]+B[j] in di:
                di[A[i]+B[j]] += 1
            else:
                di[A[i]+B[j]] = 1
    for i in range(len(C)):
        for j in range(len(D)):
            if -(C[i]+D[j]) in di:
                ans += di[-(C[i]+D[j])]
    return ans
```

小结：sum类：

1. 排序+lo,hi
2. hash

# trap rain water

自己的解决方案（超时）：类似于一层一层从下到上扫描，扫描一次所有楼层高度就低一层

1. 左(lo)右(hi)定界（左指针右移到第一个非零元素；右指针左移到第一个非零元素）
2. 从lo-hi遍历，零则bulk+=1，否则大小减一
3. 重复1，2 指导lo,hi相遇。

参考解决方案：每次加的是一列元素

左右两边各维护leftmax, rightmax.两者较小的一定会被包含住。可以放心加上。

leftmax 所在位置永远不会超过lo，rightmax 同理。 计算方向为[ --> <-- ]

1. 计算leftmax, rightmax
2. if leftmax<rightmax: bulk += leftmax-heights[lo], lo += 1

# Trapping Rain Water II

**Description** 中文 English

Given *n* x *m* non-negative integers representing an elevation map 2d where the area of each cell is *1* x *1*, compute how much water it is able to trap after raining.

自己的解决方案（递归超时）：基本思想参考上一题“自己解决方案”也是从‘地’到‘天’一层层扫描。

如果该单元格是墙，且它‘包不住’里面的时候会有一个‘连通’效应。

即把不能成为墙的所有连通分量变成一个灰色边界。临近灰色边界的格子则变成了新的墙。

一层一层减楼层，遇到0就bulk加一。

```
def trapRainWater(self, heights):
    n, m = len(heights), len(heights[0])
    vol = 0
    flag = True
    dir = [[-1, 0], [1, 0], [0,-1], [0, 1]]
    def lianTong(i,j):
        for d in range(4):
            if i+dir[d][0]>=0 and j+dir[d][1]>=0 and i+dir[d][0]<n and j+dir[d][1]<m and
                heights[i+dir[d][0]][j+dir[d][1]] == 0:
                heights[i+dir[d][0]][j+dir[d][1]] = -1
                lianTong(i+dir[d][0], j+dir[d][1])
    def isQiang(i,j):
        if i==0 or j==0 or i==n-1 or j == m-1 or heights[i-1][j] == -1 or heights[i+1][j]
            == -1 or heights[i][j-1] == -1 or heights[i][j+1] == -1:
            return True
        return False
    while flag:
        flag = False
        for i in range(n):
            for j in range(m):
                if isQiang(i, j) and heights[i][j] == 0:
                        heights[i][j] = -1
                        lianTong(i, j)
        for i in range(n):
            for j in range(m):
                if heights[i][j] == -1: continue
                if heights[i][j] == 0: vol += 1
                else: heights[i][j] -= 1
                flag = True
    return vol
```

参考解决方案：基本思想参考上一题。圈水肯定也是从最外层到最里层。

借助堆找到当前最低点，由于是从外向里找，这个点成为‘当前’短板。

放入堆的是，当前短板（短板是最外层最矮的那一个）长度和当

前元素位置。

每个位置只会被访问一次。原因也是因为是由最外层到最里层访问的。最外层最短就能决定当前储水量

```
class Solution:
    """
    @param heights: a matrix of integers
    @return: an integer
    """
    def trapRainWater(self, heights):
        # write your code here
        if not heights or not heights[0]:
            return 0
        n, m = len(heights), len(heights[0])
        vol = 0
        visited = [[0 for j in range(m)] for i in range(n)]
        heap = []
        import heapq

        for i in range(n):
            for j in range(m):
                if i==0 or j==0 or i==n-1 or j==m-1:
                    heapq.heappush(heap, (heights[i][j],i,j))
                    visited[i][j] = 1
        while heap:
            height, i, j = heapq.heappop(heap)
            for (x, y) in ((i-1,j), (i+1,j), (i,j-1), (i,j+1)):
                if x>=0 and y>=0 and x<n and  y<m and not visited[x][y]:
                    vol += max(0, height-heights[x][y])
                    heapq.heappush(heap, (max(height,heights[x][y]),x,y))
                    visited[x][y] = 1
        return vol
```

小结：最外最短。

这类题都是由外层向内层考虑。考虑外层时又是考虑最短的也就是短板效应。

# container with most water

基本思想：从左到右，谁短挪谁，挪动有变好的希望，不挪永远不会改变。

```
def maxArea(self, heights):
    # write your code here
    lo, hi = 0, len(heights)-1
    area = 0
    while lo < hi:
        if heights[lo] == heights[hi]:
            area = max(area, heights[lo] * (hi-lo))
            lo += 1
            hi -= 1
        elif heights[lo] < heights[hi]:
            area = max(area, heights[lo] * (hi-lo))
            lo += 1
        else:
            area = max(area, heights[hi] * (hi-lo))
            hi -= 1
    return area
```