leetcode刷题记录

Dict里存的是key，找的是value！

Array篇

1. 3 sum

利用两个指针，分别从左右两侧向内移动。注意要用continue去除不符合的数据，以此提升速度。

class Solution(object):

def threeSum(self, nums):

"""

:type nums: List[int]

:rtype: List[List[int]]

"""

solution = []

nums.sort()

for i in range(len(nums)-2):

if i >0 and nums[i]==nums[i-1]:

continue

j=i+1

k=len(nums)-1

while j<k:

s=nums[i]+nums[j]+nums[k]

if s>0:

k-=1

elif s<0:

j+=1

else:

solution.append([nums[i],nums[j],nums[k]])

while nums[k]==nums[k-1] and j<k:

k-=1

while nums[j]==nums[j+1] and j<k:

j+=1

k-=1

j+=1

return solution

1. Array Partition

学习了两个函数：

1.sorted() sorted内可接受各种对象，正序或是倒序，或是根据数据的第i项进行排序，例如：sorted(nums, key=lambda s: s[1]):即为对第二项的排序。sort只能接受list[]

2.sum()，当sum(::2)出现时，表示每隔两个再相加，即为第0项+第2项+第4项。。。

切片用法

1. Two sum

两种方法

1.用index的方法，直接找到对应的位置，需注意的是：倘若出现两个相同的数字，不能让两个数字的位置index相同，所以需要用list.index(xxx,i+1),即从第i+1个数字开始检索，避免重复

2.用字典的方法，先给出一个空的字典，然后判断余数是否在字典里（最初肯定不在），如果不在的话赋值进字典，便于后来查取。然后再利用list后面的数值来字典里面找，即利用后面的数据来查取前面的内容。

class Solution(object):

def twoSum(self, nums, target):

"""

:type nums: List[int]

:type target: int

:rtype: List[int]

"""

d={}

for i,number in enumerate(nums):

if target-number in d:

return [d[target-number],i]

else:

d[number]=i

167.Two sum II

def twoSum1(self, numbers, target):

l, r = 0, len(numbers)-1

while l < r:

s = numbers[l] + numbers[r]

if s == target:

return [l+1, r+1]

elif s < target:

l += 1

else:

r -= 1

这种方法叫two-pointer

两头分别向中间缩进

3 sum

class Solution(object):

def threeSumClosest(self, nums, target):

"""

:type nums: List[int]

:type target: int

:rtype: int

"""

nums.sort()

dev1=sum(nums[:3])

for i in range(len(nums)-2):

if i>0 and nums[i]==nums[i-1]:

continue

j,k=i+1,len(nums)-1

while j<k:

s=nums[i]+nums[j]+nums[k]

if s-target>0:

k-=1

elif s-target<0:

j+=1

else:

return s

if abs(s-target)<=abs(dev1-target):

dev1=s

return dev1

利用双指针方法。这样直接比较要比放在list里面快一些；同样dict也比list快一点点，但是感觉差别不算很大。

4 sum

class Solution(object):

def fourSum(self, nums, target):

"""

:type nums: List[int]

:type target: int

:rtype: List[List[int]]

"""

d = {}

nums.sort()

res=set()

for i in range(len(nums) - 1):

for j in range(i + 1, len(nums) - 1):

if nums[i] + nums[j] not in d:

d[nums[i] + nums[j]] = [(i, j)]

else:

d[nums[i] + nums[j]].append((i, j))

for p in range(len(nums) - 1):

for q in range(p + 1, len(nums)):

diff = target - nums[p] - nums[q]

if diff in d:

for k in d[diff]:

if k[0] == p or k[1]==q or k[1]==p or k[0]==q:

continue

num=[nums[p], nums[q], nums[k[0]], nums[k[1]]]

num.sort()

numbers=tuple(num)

res.add(numbers)

return [list(i) for i in res]

这道题继续应用dict的原理，两个数为一组，利用两个sum来做。需要注意的一个是[()]的用法：它使得k形同一个坐标，两个index同时出现，所以可以用k[0]来调用；

另外set的用法中：如果是这种的list [[1,2],[2,3].......]，此时set无法使用因为该list里面的元素unhashable。所以要先将其转化成tuple。然后最后return的时候再换回list。

565.Array Nesting

1). class Solution(object):

def arrayNesting(self, nums):

"""

:type nums: List[int]

:rtype: int

"""

d={}

n,length=0,0

for i in range(len(nums)):

while nums[nums[i]] not in d:

d[nums[nums[i]]]={}

n+=1

i=nums[i]

length=max(n,length)

n=0

return length

2). class Solution(object):

def arrayNesting(self, nums):

"""

:type nums: List[int]

:rtype: int

"""

ans, step, n = 0, 0, len(nums)

seen = [False] \* n

for i in range(n):

while not seen[i]:

seen[i] = True

i, step = nums[i], step + 1

ans = max(ans, step)

step = 0

return ans

714.Best Time to Buy and Sell Stock with Transaction Fee

Dp 动态规划问题，这一步的决定会影响到下一步，走一步看一步的感觉。

class Solution(object):

def maxProfit(self, prices, fee):

"""

:type prices: List[int]

:type fee: int

:rtype: int

"""

buy=-prices[0] #分别定义买和卖

sell=[0]\*len(prices)

for i in range(1,len(prices)):

sell[i]=max(sell[i-1],prices[i]-fee+buy) #在第i-1天卖能获得的利润与第i天卖能获得利润对比

buy=max(buy,sell[i-1]-prices[i]) #决定哪天买入比较划算

return sell[len(prices)-1] #返回最大值，最后一个就是最大值

#注意不能光注意卖出时的利润，同时应该更加注意买入后的利润，毕竟卖完以后为了赚钱还是要继续买的，不是一锤子买卖。

1. Beautiful Arrangements

class Solution(object):

def constructArray(self, n, k):

"""

:type n: int

:type k: int

:rtype: List[int]

"""

ans = range(1, n - k)

for d in xrange(k+1):

if d % 2 == 0:

ans.append(n-k + d/2)

else:

ans.append(n - d/2)

return ans #利用摆动序列求值。

Combination Sum

class Solution(object):

def combinationSum(self, candidates, target):

"""

:type candidates: List[int]

:type target: int

:rtype: List[List[int]]

"""

res=[]

candidates.sort()

i=0

self.dfs(candidates,target,0,[],res)

def dfs(self,nums,target,index,solution,res):

if target==0:

res.append(solution)

return

else:

for i in xrange(index,len(nums)):

target=target-nums[i]

self.dfs(nums,target,i,nums[i],res)

1. Combination Sum II

class Solution(object):

def combinationSum2(self, candidates, target):

"""

:type candidates: List[int]

:type target: int

:rtype: List[List[int]]

"""

res=set()

candidates.sort()

self.dfs(candidates,target,0,[],res)

return [list(i) for i in res] #把tuple的内容输入出来，用list来表达

def dfs(self,nums,target,index,path,res):

if target<0:

return

elif target==0:

res.add(tuple(path))

return

for i in xrange(index,len(nums)):

If i>index and nums[i]==nums[i-1]:

continue #很重要！把重复的数据略过。

self.dfs(nums,target-nums[i],i+1,path+[nums[i]],res)

1. Combination sum III

from itertools import combinations #首先引入combinations函数：作用是选取k个数字，itertools是个迭代模块

class Solution(object):

def combinationSum3(self, k, n):

"""

:type k: int

:type n: int

:rtype: List[List[int]]

"""

return[i for i in combinations(range(1,10),k) if sum(i)==n] #注意用法：可以这样写，方便

01. matrix

class Solution(object):

def updateMatrix(self, matrix):

"""

:type matrix: List[List[int]]

:rtype: List[List[int]]

"""

r=len(matrix)

c=len(matrix[0])

numbers0=[]

for x in xrange(r): #此处可以改为numbers0=[ [x,y] for x in xrange(r) for y in xrange(c) if matrix[x][y]==0]

for y in xrange(c):

if matrix[x][y]==0:

numbers0.append((x,y))

else: matrix[x][y]=10000 #此步是关键所在，必须要让非0位变得很大，才能在之后区分出来。

for p,q in numbers0:

for i,j in [[p+1,q],[p,q+1],[p-1,q],[p,q-1]]:

z=matrix[p][q] + 1

if 0 <= i< r and 0 <= j < c and matrix[i][j]>z:

matrix[i][j]=z

numbers0.append([i,j])

return matrix

重点重点重点，做了很久的题！

1. Accounts Merge

class Solution(object):

def accountsMerge(self, accounts):

"""

:type accounts: List[List[str]]

:rtype: List[List[str]]

"""

from collections import defaultdict #此处是引入defaultdict函数，放在最上端或者此处均可，意义是：给dict一个初始值，这样我就可以在字典为null的情况下直接append，否则不可，参照下面的用法.....

d=defaultdict(list)

res=[]

visited=[False]\*len(accounts) #注意！！这种写法才是对的，生成的是[False,False,False.........]，不能写成[False\*len(accounts)]

for i in xrange(len(accounts)):

for j in xrange(1,len(accounts[i])):

d[accounts[i][j]].append(i) #此处为具体应用，可见:相当于d[xxx]=[]，然后进行append

def addemail(i,email):

if visited[i]:

return #巧妙，访问过的就会变成true，从而跳过。

visited[i]=1

for j in xrange(1,len(accounts[i])):

email.add(accounts[i][j])

for a in d[accounts[i][j]]:

addemail(a,email)

for i in xrange(len(accounts)):

if visited[i]:

continue

email=set() #定义email，之所以放在后面是因为每次调用完addemail函数后都要清空email

addemail(i,email)

res.append([accounts[i][0]]+sorted(email)) #对于set也可以直接排序，用sorted

return res

#关于函数执行问题：第一次函数从第一段的for语句执行完后，先来到第三段执行for，然后再for里找到了addemail函数，从而再执行第二段函数。

1. Average of Levels in Binary Tree

两种做法：

# 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 averageOfLevels(self, root):

sum1=[]

def sumabc(node, depth=0):

if node: #如果这层已经没有数据的话，那么root为null

if len(sum1)<=depth:

sum1.append([0,0])

sum1[depth][0]+=node.val #当前节点的value

sum1[depth][1]+=1

sumabc(node.left, depth+1)

sumabc(node.right, depth+1)

sumabc(root) #注意函数嵌套的用法，函数内部嵌套时，在后面调用。

return [s/float(c) for s,c in sum1]

# 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 averageOfLevels(self, root):

sum1=[]

level=[root] #此步很有必要，因为root本身是treenode type，不是list，不能进行各种运算与迭代

while level:

sum1.append(sum(node.val for node in level)\*1.0/len(level))

level=[x for node in level for x in (node.left,node.right) if x] #level由一个list逐渐变成内部含有多个list的list如list[0],list[i]....

return sum1

划重点！

关于self：它是一个参数，第一个作用是在调用函数时，需要写self.xxxx();如果函数内部无self这个参数，那么调用时也不需要写。

第二点：self在函数内部表示自身的意思，所以会出现self.left; self.val等表达方式，意思就是自身的调用。比如A=self.TreeNode(), self.val就要表达为A.val

关于树：

Val意思应该就是value，表示的是在当前的树下，所获得的当前的节点（node）。

比如如果root是一个完整的树的话，那么root.val就表示树的根；如果是一颗二叉树的左子树（root.left），那么就表示该左子树的根节点。

\_\_init\_\_表示自发，如果在函数的头部，那么无论是什么命令都要先执行init。其中\_\_ \_\_(前后各两条的下划线）表示特殊的意思。