*Tools->Preference->not capitalize, not smart quota*

*Then add some tap-left stops*

**368 Largest Divisible Subset**

*Given a set of distinct positive integers, find the largest subset such that every pair (Si, Sj) of elements in this subset satisfies: Si % Sj = 0 or Sj % Si = 0. If there are multiple solutions, return any subset is fine.*

def largestDivisibleSubset(self, nums):

        nums.sort()

        pairs = {-1: set()}  *# every n % -1 is 0*

        for num in nums:

# max(obj)  obj must be not None

            pairs[num] = max((pairs[k] for k in pairs if num % k == 0),

                                    key=len) | {num}

        return list(max(pairs.values(), key=len))

**316 Remove Duplicate Letters**

*Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.*

def removeDuplicateLetters(self, s):

       if not s: ***# basic return***

return ''

chars = sorted(set(s))

for char in chars:

idx = s.find(char)

suffix = s[idx:]

if set(suffix) == set(s):

return char + self.removeDulicateLetters(suffix.replace(char, '')

**542 01 Matrix**

*Given a matrix consists of 0 and 1, find the distance of the nearest 0 for each cell. The distance between two adjacent cells is 1.*

from collections import deque

def updateMatrix(self, matrix):

if not matrix or not matrix[0]:

return matrix

rows, cols = len(matrix), len(matrix[0])

dirs = ((1, 0), (-1, 0), (0, -1), (0, 1))

q = deque()

for i, row in enumerate(matrix):

for j, item in enumerate(row):

if matrix[i][j] == 0:

q.append((i, j))

else:

matrix[i][j] = float('inf')

while q:

x, y = q.popleft()

dis = matrix[x][y]

for dx, dy in dirs:

nx, ny, ndis = x + dx, y + dy, dis + 1

if 0 <= nx < rows and 0 <= ny < cols and matrix[nx][ny] > ndis:

matrix[nx][ny] = ndis

q.append((nx, ny))

return matrix

**140 Word Break II**

*Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences.*

def wordBreak(self, s, wordDict):

ans = []

       def rbreak(rs, pres):

            if not rs:

                ans.append(' '.join(pres))

            else:

                for word in wordDict:

                    if rs.startswith(word):

                        remain = rs[len(word):]

                        pres.append(word)

                        rbreak(remain, pres)

                        pres.pop()

        rbreak(s, [])

        return ans

*This solution can solve basic test cases ,but not for aaaaaaa, so we have to add a memory dictionary. However, this kind of back-track solution can not get the intermedia result enoughly. We must write another one:*

def wordBreak(self, s, wordDict):

memory = {}

def re\_break(start=0):

if start >= len(s):

return [''] *# here we must put a str in it, then we can it use to expand other word*

elif start in memory:

return memory[start]

else:

res = []

for word in wordDict:

now\_str = s[start:]

if now\_str.startswith(word):

next\_start = start + len(word)

rest\_str = now\_str[next\_start:]

rest\_res = re\_break(next\_start)

for rest in rest\_res:

*# mention the inner parenthesis, if else have lower priority*

      res.append(word + (' '  if rest else '') + rest)

memory[start] = res

return res

return re\_break()

**541 Reverse String II**

*Given a string and an integer k, you need to reverse the first k characters for every 2k characters counting from the start of the string. If there are less than k characters left, reverse all of them.*

def reverseString(self, s, k):

slist = list(s)

for i in range(0, len(s), 2\*k):

*# slist[i:i+k].reverse()  this would create a new list then reverse the new list, slist not changed*

*# slist[i+k-1:i-1:-1] also does not work because if i is 0, -1 means the last one*

slist[i:i+k] = reversed(slist[i:i+k])

return ''.join(slist)

**679 24-Game**

*You have 4 cards each containing a number from 1 to 9. You need to judge whether they could operated through \*, /, +, -, (, ) to get the value of 24.*

from operator import truediv, add, sub, mul

class Solution:

    def judgePoint24(self, nums):

        if not nums:

            return False

        if len(nums) == 1:

            return abs(nums[0] - 24) < 1e-6

        L = len(nums)

        ops = (truediv, add, sub, mul)

        for i in range(L):

            for j in range(L):

                if i != j:

                    rest = [nums[k] for k in range(L) if i != k != j]

                    for op in ops:

                        if op is not truediv or nums[j] != 0:

                            rest.append(op(nums[i], nums[j]))

                            if self.judgePoint24(rest):

                                return True

                            rest.pop()

        return False

**406 Queue Reconstruction by Height**

*Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers (h, k), where h is the height of the person and k is the number of people in front of this person who have a height greater than or equal to h. Write an algorithm to reconstruct the queue.*

from collections import deque

def reconstructQueue(self, people):

    people.sort(key=lambda p: (-p[0], p[1]))

    ans = deque()

    for p in people:

        ans.insert(p[1], p)

    return list(ans)   *# deque is not list, if the return type is list, we have to make a change*

**483 Smallest Good Base (Math problem)**

*For an integer n, we call k>=2 a good base of n, if all digits of n base k are 1. Now given a string representing n, you should return the smallest good base of n in string format.*

from math import log

def smallestGoodBase(self, n):

n = int(n)

max\_length = log(n, 2)

for i in range(max\_length, 1, -1):

k = int(n\*\*i\*\*-1)

if (k\*\*(i + 1) - 1) // (k - 1) == n:

return str(k)

return str(n-1)

**302 Smallest Rectangel Enclosing Black Pixels (brute force or BFS are accepted, here is bisect)**

*An image is represented by a binary matrix with 0 as a white pixel and 1 as a black pixel. The black pixels are connected, i.e., there is only one black region. Pixels are connected horizontally and vertically. Given the location (x, y) of one of the black pixels, return the area of the smallest (axis-aligned) rectangle that encloses all black pixels.*

def minArea(self, image, x, y):

    if not image or not image[0]:

        return 0

    def searchUpDown(low, high, up=True):  # the default attributes must all at end!

        while low < high:

            mid = (low + high) // 2

            if ('1' in image[mid]) == up:  # when meet 1 or 0, ask it's number or charactor

                high = mid

            else:

                low = mid + 1

        return low

    def searchLeftRight(left, right, top, down, lside=True):

        while left < right:

            mid = (left + right) // 2

            if any(image[row][mid] == '1' for row in range(top, down)) == lside:

                right = mid

            else:

                left = mid + 1

        return left

    top = searchUpDown(0, x, up=True)

    bottom = searchUpDown(x+1, len(image), up=False)

    left = searchLeftRight(0, y, top, bottom, True)

    right = searchLeftRight(y+1, len(image[0]), top, bottom, False)

    return (right - left) \* (bottom - top)

**377 Combination Sum IV**

*Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.*

def combinationSum4(self, nums, target):

memory = {}

def re\_sum(pre=0):

if pre == target:

return 1

elif pre in memory:

return memory[pre]

else:

temp = 0

for num in nums:

if pre + num <= target:

temp += re\_sum(pre + num)

memory[pre] = temp

return temp

return re\_sum()

**345 Reverse Vowels of a String**

*Write a function that takes a string as input and reverse only the vowels of a string.*

def reverseVowels(self, s):

vs = 'aeouiAEOUI'

vowels = [(i, c) for i, c in enumerate(s) if c in vs]

reversed\_vowels = vowels[::-1]

ss = list(s)

for i, (idx, c) in enumerate(vowels):

n\_c = reversed\_vowels[i][-1]

ss[idx] = n\_c

*# we can also only use one list to finish*

*for i in range(len(vowels) // 2):*

*o\_idx = vowels[i][0]*

*n\_idx = vowels[-i-1][0]*

*ss[o\_idx], ss[n\_idx] == ss[n\_idx], ss[o\_idx]*

return ''.join(ss)

**329 Longest Increasing Path in a Matrix *# lost lot of ' ) ' and ' ] '***

*Given an integer matrix, find the length of the longest increasing path. From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary.*

def longest IncreasingPath(self, matrix):

if not matrix or not matrix[0]: return 0

rows, cols = len(martix), len(matrix[0])

dist = [[0] \* cols for \_ in range(rows)]

dirs = ((0, 1), (0, -1), (1, 0), (-1, 0))

ans = 0

def dfs(x, y):

            if dist[x][y]: return dist[x][y]

            for dx, dy in dirs:

                nx, ny = x + dx, y + dy

                if 0 <= nx < rows and 0 <= ny < cols:

                    if matrix[nx][ny] > matrix[x][y]:

                        dist[x][y] = max(dist[x][y], dfs(nx, ny) + 1)

                    else:

                        dist[x][y] = max(dist[x][y], 1)

            return dist[x][y]

for i in range(rows):

for j in range(cols):

ans = max(ans, dfs(i, j))

* *# this basic dfs can not finish the work, because to remember the result of every point, we should run dfs in every point, and the iterative method is not very good. we should use recursive one!*

*if dist[i][j] == 0:*

*q = [(i, j)]*

*dist[i][j] = 1*

*while q:*

*x, y = q.pop()*

*for dx, dy in dirs:*

*nx, ny = x + dx, y + dy*

*if 0 <= nx < rows and 0 <= ny < cols and matrix[nx][ny] > matrix[x][y] and dist[nx][ny] < dist[x][y] + 1:*

*dist[nx][ny] = dist[x][y] + 1*

*ans = max(ans, dist[nx][ny])*

*q.append((nx, ny))*

return ans

**482 License Key Formatting**

*Input: S = "5F3Z-2e-9-w", K = 4  Output: "5F3Z-2E9W"*

def licenseKeyFormatting(self, S, K):

if not S:

return ''

s = S.replace('-', '').upper()

start = len(s) % K

res = s[:start]

for i in range(start, len(s), K):

res += ('-' if i > 0 else '') + s[i:i+K]

return res

**114 Flatten Binary Tree to Linked List**

*Given a binary tree, flatten it to a linked list in-place.*

def flatten(self, root):

        def flatten\_node(node):

            if not node:

                return None, None

            left\_head, left\_tail = flatten\_node(node.left)

            right\_head, right\_tail = flatten\_node(node.right)

            node.left = None

            node.right = left\_head or right\_head

            if left\_tail and right\_head:

                left\_tail.right = right\_head

            tail = right\_tail if right\_tail else left\_tail

            return node, tail if tail else node

        flatten\_node(root)

***# We can also use a recursive way like this, it's hard to understand, but very clean:***

class Solution:

    def \_\_init\_\_(self):

        self.head = None

    def flatten(self, root):

        """

        :type root: TreeNode

        :rtype: void Do not return anything, modify root in-place instead.

        """

        if not root:

            return

        self.flatten(root.right)

        self.flatten(root.left)

        root.right = self.head

        root.left = None

        self.head = root

**330 Patching Array**

*Given a sorted positive integer array nums and an integer n, add/patch elements to the array such that any number in range [1, n] inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required.*

def minPatches(self, nums, n):

cover = 1

idx = 0

ans = 0

while cover <= n:

if idx < len(nums) and nums[i] <= cover:

cover = cover + nums[i]

idx += 1

else:

ans += 1

cover = cover + cover

return ans

**42 Trapping Rain Water**

*Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.*

def trap(self, height):

        left\_wall, right\_wall = 0, 0

        left, right = 0, len(height) - 1

        water = 0

        while left <= right: ***# note here is <= not only <***

            if left\_wall < right\_wall:

                if height[left] < left\_wall:

                    water += left\_wall - height[left]

                else:

                    left\_wall = height[left]

                left += 1

            else:

                if height[right] < right\_wall:

                    water += right\_wall - height[right]

                else:

                    right\_wall = height[right]

                right -= 1

        return water

**407 Trapping Rain Water II**

*Given an m x n matrix of positive integers representing the height of each unit cell in a 2D elevation map, compute the volume of water it is able to trap after raining.*

***#  error: ny was wrote to xy & height -> heigth & didn't check the validity /və'lɪdəti/ at the begining***

from queue import PriorityQueue

def trapRainWater(self, heightMap):

        if not heightMap or not heightMap[0]:

            return 0

        pq = PriorityQueue()

        rows, cols = len(heightMap), len(heightMap[0])

***# used visited if they don't ask you to in-place, don't put yourself in trouble***

        visited = [[False]\*cols for \_ in range(rows)]

        for x, row in enumerate(heightMap):

            for y, height in enumerate(row):

                if x == 0 or x == rows -1 or y == 0 or y == cols - 1:

                    pq.put((height, x, y))

                    visited[x][y] = True

        water = 0

        dirs = ((-1, 0), (1, 0), (0, 1), (0, -1))

        while pq.qsize():

            height, x ,y = pq.get()

            for dx, dy in dirs:

                nx, ny = x + dx, y + dy

                if 0 < nx < rows - 1 and 0 < ny < cols - 1 and not visited[nx][ny]:

                    if heightMap[nx][ny] < height:

                        water += height - heightMap[nx][ny]

                        heightMap[nx][ny] = height

                    pq.put((heightMap[nx][ny], nx, ny))

                    visited[nx][ny] = True

        return water

**341 Flatten Nested List Iterator**

*Given a nested list of integers, implement an iterator to flatten it. Each element is either an integer, or a list -- whose elements may also be integers or other lists.*

***#  1. we can't use len(stack) to check the hasNext(), because if the stack has a empty list like [[]], the len(list) is 1, however, there is no item in the inner list. So we have to expand the list within hasNext()***

***#  2. for the count, we should use 0 -> n, every time we add one, not n -> 0. because we need the count to get the value: val = list[count]***

***#  error: forget self.stack, wrote to stack, this mistake happened several times!***

class NestedIterator(object):

    def \_\_init\_\_(self, nestedList):

        self.stack = []

        self.stack.append([nestedList, 0])

    def next(self):

***# after change append [list, count] above, forget to change the order here.***

        nest\_list, count = self.stack[-1]

        self.stack[-1][1] += 1

        return nest\_list[count].getInteger()

    def hasNext(self):

        while self.stack:

            nest\_list, count = self.stack[-1]

            if count == len(nest\_list):

                self.stack.pop()

            else:

                item = nest\_list[count]

                if item.isInteger():

                    return True

                self.stack[-1][1] += 1

                self.stack.append([item.getList(), 0])

**447. Number of Boomerangs**

*Given n points in the plane that are all pairwise distinct, a "boomerang" is a tuple of points (i, j, k) such that the distance between i and j equals the distance between i and k (the order of the tuple matters).*

from collections import defaultdict

def numberOfBoomerangs(self, points):

def points\_dis(A, B):

return sum((a-b)\*\*2 for a, b zip(A, B)) ***# error: there no \* in front of zip***

count = 0

for i in range(len(points)):

dis = defaultdict(int)  
 for j in range(len(points)):

if i != j:

d = points\_dis(points[i], points[j])

count += dis[d]

dis[d] += 1

return count\*2  ***# arithmetic progression sum(A) = n(an+a1)/2***

**448. Find All Numbers Disappeared in an Array**

*Given an array of integers where 1 ≤ a[i] ≤ n (n = size of array), some elements appear twice and others appear once. Find all the elements of [1, n] inclusive that do not appear in this array.*

def findDisappearedNumbers(self, nums):

     for num in nums:

***# error: forgot to compute the abs at first, since the num may be changed to negative***

            idx = abs(num)

            if nums[idx-1] > 0:

                nums[idx-1] = -nums[idx-1]

     res = [i+1 for i, num in enumerate(nums) if num > 0]

     return res

**616. Add Bold Tag in String**

*Given a string s and a list of strings dict, you need to add a closed pair of bold tag <b> and </b> to wrap the substrings in s that exist in dict. If two such substrings overlap, you need to wrap them together by only one pair of closed bold tag. Also, if two substrings wrapped by bold tags are consecutive, you need to combine them.*

***# error: we don't need to use recursive and memory, since we loop through the while s. If we use recursive, it will lead to a TLE***

***# of course, we can still use memery, but the memery should in the i loop, not the start of addBold***

def addBoldTag(self, s, dict):

L = len(s)

bold = [False]\*L

~~memory = set()~~

def addBold(start=0):

if start >= L ~~or start in memory~~:

return

for i in range(start, L):

*# if i in memory: return* ***# if we want to use memory***

suffix = s[i:]

for word in dict:

if suffix.startswith(word):

new\_start = i + len(word)

for j in range(i, new\_start):

bold[j] = True

~~addBold(new\_start)~~

*# memory.add(i)* ***# if we want to use memory***

~~memory.add(start)~~

addBold()

pre = 0

ans = ''

for i in range(L):

if bold[i] != pre:

pre = bold[i]

ans += '<b>' if pre else '</b>'

ans += s[i]

ans += '</b>' if pre else ''

***# for this loop, we can use groupby like that:***

*for tag, members in itertools.groupby(zip(s, bold), lambda z: z[1]):*

*if tag == 1:*

*ans += '<b>'*

*ans += ''.join(pair[0] for pair in members)*

*if tag == 1:*

*ans += '</b>'*

return ans

**604. Design Compressed String Iterator**

*Design and implement a data structure for a compressed string iterator. It should support the following operations: next and hasNext. The given compressed string will be in the form of each letter followed by a positive integer representing the number of this letter existing in the original uncompressed string.*

***# of course, we can use only cur\_char and cur\_num to replace queue to save space, this just for practice.***

from queue import deque

class StringIterator(object):

def \_\_init\_\_(self, compressedString):

self.q = deque()

i = 0

while i < len(compressedString):

char = compressedString[i]

i += 1

start = i

while i < len(compressedString) and compressedString[i].isdigit():

i += 1

num = int(compressedString[start:i])

self.q.append([char, num])***# error: tuple is not mutable! should be [ ] not ( )***

def next(self): *# if there is a letter, return it, otherwise return a blank*

if self.hasNext():

temp, count = self.q[0]

if count == 1:

self.q.popleft()

else:

self.q[0][1] -= 1

return temp

else:

return ' '

def hasNext(self): *# return true or false*

return bool(self.q)  ***# error: should use bool(), not return self.q directly, since we should return bool***

**638. Shopping Offers**

*In LeetCode Store, there are some kinds of items to sell. Each item has a price. However, there are some special offers, and a special offer consists of one or more different kinds of items with a sale price. Each special offer is represented in the form of an array, the last number represents the price you need to pay for this special offer, other numbers represents how many specific items you could get if you buy this offer. You could use any of special offers as many times as you want.*

def shoppingOffers(self, price, special, needs):

*# memory = {} # we can also use a memory, but if not, it's also fine*

def buy(need):  # return the cost for need

*# if tuple(need) in memory:*

*#    return memory[tuple(need)]*

if not any(need):  ***# error, not need == needs, it's "not any(need)"***

return 0  
 min\_cost = sum(p\*c for p, c in zip(price, need))

for offer in special:

if all(n >= s for n, s in zip(need, offer)): ***# error: here is >= not >***

new\_need = [need[i] - offer[i] for i in range(len(need))]

cost = offer[-1] + buy(new\_need)

min\_cost = min(min\_cost, cost)

*# memory[tuple(need)] = min\_cost*

return min\_cost

return buy(needs) ***# error: here is needs not need***

**587. Erect the Fence**

*There are some trees, where each tree is represented by (x,y) coordinate in a two-dimensional garden. Your job is to fence the entire garden using the minimum length of rope as it is expensive. The garden is well fenced only if all the trees are enclosed. Your task is to help find the coordinates of trees which are exactly located on the fence perimeter.*

def ourterTrees(self, points):

        def oritation(A, B, C):

            return (B.x - A.x) \* (C.y-B.y) - (B.y - A.y) \* (C.x - B.x)

        hull = []

***# lambda item: (item.x, item,y, item.z ..) must has the (), since it is sorted according a tuple, not two items***

        points.sort(key=lambda tree: (tree.x, tree.y))

        for i in range(len(points)):

            while len(hull) >= 2 and oritation(hull[-2], hull[-1], points[i]) < 0:

                hull.pop()

            hull.append(points[i])

        for i in reversed(range(len(points))):

            while len(hull) >= 2 and oritation(hull[-2], hull[-1], points[i]) < 0:

                hull.pop()

            hull.append(points[i])

        return list(set(hull)) ***#  error: hull may has duplicate items, so can't return directly***

**643. Maximum Average Subarray I**

*Given an array consisting of n integers, find the contiguous subarray of given length k that has the maximum average value. And you need to output the maximum average value.*

def findMaxAverage(self, nums, k):

        if len(nums) < k:

            return 0

        ans = 0

        sum\_ = 0

        for i in range(len(nums)):

            sum\_ += nums[i]  ***# error: forgot a 's', nums[i] -> num[i]***

            if i == k-1:

                ans = sum\_ / k ***# error: forgot a \_, sum\_ -> sum, don't use this kind of method later***

            elif i > k-1:

                sum\_ -= nums[i-k]

                ans = max(ans, sum\_ / k)

        return ans

**644. Maximum Average Subarray II**

*Given an array consisting of n integers, find the contiguous subarray whose length is greater than or equal to k that has the maximum average value. And you need to output the maximum average value.*

***# a TLE solution, and several errors.***

def findMaxAverage(self, nums, k):

if len(nums) < k:

return

~~ans = 0~~  *# this should be ans = float('-inf') since there may be negative values*

for i in range(len(nums)):  *# here can be optimaized to i in range(len(nums) - k + 1)*

~~sum = nums[i]~~  *# this should be sum = 0, and j start from i not i + 1, since if there only one element, i+1 is out.*

~~for j in range(i+1, len(nums)):~~  *# for j in range(i, len(nums)):*

sum += nums[j]

~~if j - i >= k:~~  *# here is j - i + 1, since the sum contains i and j, there are j - i + 1 items*

ans = max(ans, sum / ~~(j - i)~~)  *# also here is j - i + 1*

return ans

***# here is the right solution***

from itertools import accumulate

def findMaxAverage(self, nums, k):

N = len(nums)

S = list(accumulate(nums))

def density(start, end):

return (S[end] - (S[start-1] if start>0 else 0)) / (end - start + 1)

q = []

ans = float('-inf')

for i in range(k-1, N):

while len(q) >= 2 and density(q[-2], q[-1]-1) >= density(q[-2]), i-k):

q.pop()

q.append(i-k+1)

while len(q) >= 2 and density(q[0], q[1]-1) <= density(q[0], i):

q.pop(0)

ans = max(ans, density(q[0], i))

return ans

**581. Shortest Unsorted Continuous Subarray**

*Given an integer array, you need to find one continuous subarray that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order, too. You need to find the shortest such subarray and output its length.*

def findUnsortedSubarray(self, nums):

right\_max = []

rmax = float('-inf') ***# error: here is float('-inf') not float('inf')***

for num in nums:

rmax = max(rmax, num)

right\_max.append(rmax)

left\_min = []

lmin = float('inf')

for num in reversed(nums):

lmin = min(lmin, num)

left\_min.insert(0, lmin)

left = len(nums) - 1  ***# error: the left should start from right, not 0***

for i in range(len(nums)):

if nums[i] > left\_min[i]:

left = i

break

right = 0  ***# error: same with left, start from left, not len(nums)-1***

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

if nums[i] < right\_max[i]:

right = i

break

return right - left + 1 if right > left else 0 ***# error:  if else judge if no found, just return 0, not negative***

**657. Judge Route Circle**

*Initially, there is a Robot at position (0, 0). Given a sequence of its moves, judge if this robot makes a circle, which means it moves back to the original place. The move sequence is represented by a string. And each move is represent by a character. The valid robot moves are R (Right), L (Left), U (Up) and D (down). The output should be true or false representing whether the robot makes a circle.*

def judgeCircle(self, moves):

right, up = 0, 0

for c in moves:

if c == 'U':

up += 1

elif c == 'D':

up -= 1

elif c == 'R':  ***# error: elif wrote to elfi...***

right += 1

else:

right -= 1

return right == 0 and up == 0

**651. 4 Keys Keyboard**

*Imagine you have a special keyboard with the following keys:*

*Key 1: (A): Print one 'A' on screen.*

*Key 2: (Ctrl-A): Select the whole screen.*

*Key 3: (Ctrl-C): Copy selection to buffer.*

*Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.*

*Now, you can only press the keyboard for N times (with the above four keys), find out the maximum numbers of 'A' you can print on screen.*

def maxA(self, N):

best = [0, 1]

for k in range(2, N+1):

count = 0

for i in range(k-1):

count = max(count, best[i] \* (k - i - 1))

best.append(count)

best[-1] = max(best[-1], best[-2] + 1)

return best[-1]

**583. Delete Operation for Two Strings**

*Given two words word1 and word2, find the minimum number of steps required to make word1 and word2 the same, where in each step you can delete one character in either string.*

***# for string, if we need compair in normal algorithm, it's better to use idx parameter other than substrings***

***# for dp solution, it's already index. Be care of base case and i-1, j-1 for dp is usually larger than original strs***

***# approach I,*** *longest common str, then L1 + L2 - 2\*common*

def minDistance(self, word1, word2):

m, n = len(word1), len(word2)

dp = [[0] \* (n+1) for \_ in range(m + 1)]

for i in range(1, m+1):

for j in range(1, n+1):

if word1[i] == word2[j]:

dp[i][j] = 1 + dp[i-1][j-1] ***# error: forgot to add 1***

else:

dp[i][j] = max(dp[i-1][j], dp[i][j-1])

return m + n - 2 \* dp[-1][-1]

***# approach II,*** *use dp for distance directly*

def minDistance(self, word1, word2):

m, n = len(word1), len(word1)

dp = [[0] \* (n + 1) for \_ in range(m + 1)]

for i in range(0, m+1):

for j in range(0, n+1):

if i == 0 or j == 0:

dp[i][j] = max(i, j)

elif word1[i-1] == word2[j-1]:  ***# error: forgot to minus 1***

dp[i][j] = dp[i-1][j-1]

else:

dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1])

return dp[-1][-1]

**658. Find K Closest Elements**

*Given a sorted array, two integers k and x, find the k closest elements to x in the array. The result should also be sorted in ascending order. If there is a tie, the smaller elements are always preferred.*

from bisect import bisect\_left

def findClosestElements(self, arr, k, x):

        if k >= len(arr):

            return arr

        idx = bisect\_left(arr, x)

***# error: note the max and min here, don't upside down***

        left = max(idx - k, 0)

        right = min(len(arr)-1, idx + k)

        while right - left + 1 > k:

            if abs(arr[left] - x) <= abs(arr[right] -x):

                right -= 1

            else:

                left += 1

        return arr[left:right+1]

**568 Maximum Vacation Days**

*You're given the flights matrix and days matrix, output the maximum vacation days you could take during K weeks.*

*1. You can only travel among N cities, represented by indexes from 0 to N-1. Initially, you are in the city 0 on Monday.*

*2. The flights are represented as a N\*N matrix (not necessary symmetrical). If there is no flight from the city i to the city j, flights[i][j] = 0; Otherwise, flights[i][j] = 1. Also, flights[i][i] = 0 for all i.*

*3. You totally have K weeks (each week has 7 days) to travel. You can only take flights at most once per day and can only take flights on each week's Monday morning. We don't consider the impact of flight time.*

*4. For each city, you can only have restricted vacation days in different weeks, given an N\*K matrix called days representing this relationship. For the value of days[i][j], it represents the maximum days you could take vacation in the city i in the week j.*

***# two things we should pay attention:***

***# 1. if we use pre -> now, we should consider the pre is exist or not.***

***# 2. we can use '-inf' to avoid the pre problem, since if it not exist, it will still 'inf'***

***# 3. we can also use now -> next to solve this problem, it's kind of same.***

***# 4. the code is right, since it's TLE on leetcode. we can use 1D dp to solve this problem***

***# Approach 1, using float('-inf')***

def maxVacationDays(self, flights, days):

        N, W = len(days), len(days[0])

        dp = [[float('-inf')] \* N for \_ in range(W+1)]

        dp[0][0] = 0

        for week in range(1, W+1):

            for city in range(N):

                for pre\_city in range(N):

                    if flights[pre\_city][city] or pre\_city == city:

                        dp[week][city] = max(dp[week][city], dp[week-1][pre\_city] + days[city][week-1])

        return max(dp[-1])

***# Approach 2, using -1 to identify where pre\_city can be achieved or not***

def maxVacationDays(self, flights, days):

        N, D = len(days), len(days[0])

        dp = [days[i][0] if flights[0][i] else -1 for i in range(N)]

        dp[0] = days[0][0]

        for week in range(1, D):

            cur\_dp = [-1] \* N

            for pre\_city, vactions in enumerate(dp):

                if vactions != -1:

                    for city, available in enumerate(flights[pre\_city]):

                        if available or city == pre\_city:

                            cur\_dp[city] = max(cur\_dp[city] , vactions + days[city][week])

            dp = cur\_dp

        return max(dp)

**357. Count Numbers with Unique Digits**

*Given a non-negative integer n, count all numbers with unique digits, x, where 0 ≤ x < 10n.*

*Given n = 2, return 91. (the total numbers in the range of 0 ≤ x < 100, excluding [11,22,33,44,55,66,77,88,99])*

def countNumbersWithUniqueDigits(self, n):

if n == 0:

return 1

res = 10

unique\_combination = 9

unique\_remain = 9

while n > 1 and unique\_remain > 0:

unique\_combination = unique\_combination \* unique\_remain

res += unique\_remain

unique\_remain -= 1

n -= 1

return res

**317. Shortest Distance from All Buildings**

*You want to build a house on an empty land which reaches all buildings in the shortest amount of distance. You can only move up, down, left and right. You are given a 2D grid of values 0, 1 or 2, where:*

*Each 0 marks an empty land which you can pass by freely.*

*Each 1 marks a building which you cannot pass through.*

*Each 2 marks an obstacle which you cannot pass through.*

*There will be at least one building. If it is not possible to build such house according to the above rules, return -1.*

*# there are two import notes:*

***# 1. we should have a matrix to remember how many homes can reach this cell***

***# 2. to reduce the time complexity, we have to ensure all the homes are connected***

***!!! Very Important !!! The BFS, set the visited after add to queue, not after pop to queue! Otherwise, there will be lots of duplicate cells in the queue!  same with the DFS, the dfs has smaller duplicate since it pop on the end, no chance to add lot of same cells, but still have about ½ duplicate! The BFS will have more than 10 times duplicate! For example, if we have a 10\*10 matrix, it should have 100 visiting in total. With correct method (set the visited after add to queue), it exactly 100 visiting. With wrong method, DFS is 181, BFS is 2075!!!!***

from queue import deque

def shortestDistance(self, grid):

        rows, cols = len(grid), len(grid[0])

        homes = sum(1 for i in range(rows) for j in range(cols) if grid[i][j] == 1)

        dis = [[[0] \* 2 for c in range(cols)] for r in range(rows)]  ***# three dimensions***

        dirs = ((0, 1), (0, -1), (1, 0), (-1, 0))

        for i in range(rows):

            for j in range(cols):

                if grid[i][j] == 1:

                    available = 1 ***# get the number of connected homes***

                    q = deque()

                    q.append((i, j, 0))

                    visited = [[False]\*cols for \_ in range(rows)]  ***# every BFS has its own visited***

                    visited[i][j] = True

                    while q:

                        x, y, d = q.popleft()

                        for dx, dy in dirs:

                            nx, ny, nd = x + dx, y + dy, d + 1

                            if 0 <= nx < rows and 0 <= ny < cols and not visited[nx][ny]:

                                visited[nx][ny] = True  ***# set visited here, not after pop***

                                if grid[nx][ny] == 0:

                                    dis[nx][ny][0] += nd

                                    dis[nx][ny][1] += 1

                                    q.append((nx, ny, nd))

                                elif grid[nx][ny] == 1:

                                    available += 1

                    if available != homes:

                        return -1

        ans = float('inf')

        for i in range(rows):

            for j in range(cols):

                if dis[i][j][0] != 0 and dis[i][j][1] == homes:

                    ans = min(ans, dis[i][j][0])

        return ans if ans != float('inf') else -1

**360. Sort Transformed Array**

*Given a sorted array of integers nums and integer values a, b and c. Apply a quadratic function of the form f(x) = ax2 + bx + c to each element x in the array. The returned array must be in sorted order. Expected time complexity: O(n)*

***# error: If we use bisect and start from the middle to sides, we should start at idx-1 or idx according a<0 or > 0***

***# error: corner case: a == 0***

***# it's hard to set all the cases if we just use the index of bisect, because it's easy to miss one element, whether the idx or the idx + 1 or idx - 1. So, it's better to start from the 0 to n-1***

def sortTransformedArray(self, nums, a, b, c):

        def quadratic(x):

            return a\*x\*\*2 + b\*x + c

***# actually, we can delete the a == 0 part, but I think it's also good to seperate it out***

*if a == 0:*

*if b > 0:*

*return [quadratic(x) for x in nums]*

*else:*

*return [quadratic(x) for x in reversed(nums)]*

        L = len(nums)

        res = [0] \* L

        idx = 0 if a < 0 else L - 1

        left, right = 0, L - 1 ***# error: here is L - 1***

        while left <= right:  ***# error: here is <= not <***

            q\_left, q\_right = quadratic(nums[left]), quadratic(nums[right])

            if a < 0:

                if q\_left < q\_right:

                    res[idx] = q\_left

                    left += 1

                else:

                    res[idx] = q\_right

                    right -= 1

                idx += 1

            else:

                if q\_left < q\_right:

                    res[idx] = q\_right

                    right -= 1

                else:

                    res[idx] = q\_left

                    left += 1

                idx -= 1

        return res

**315. Count of Smaller Number After Self**

*You are given an integer array nums and you have to return a new counts array. The counts array has the property where counts[i] is the number of smaller elements to the right of nums[i].*

***# note: it's better to use reversed(range(N)) or range(N)[::-1], since range(N-1, -1, -1) is ugly and easy to wrong***

***# range(5)[::-1] => range(4, -1, -1) python will do it for you, but Need More Time, about two times!***

***# the priority of comparison: or < and < not***

def countSmaller(self, nums):

        ans = [0] \* len(nums)

        def divcon(enums):

            mid = len(enums) // 2

            if mid:

                left, right = divcon(enums[:mid]), divcon(enums[mid:])

                for i in reversed(range(len(enums))):

                    if not right or left and left[-1][1] > right[-1][1]:  ***# the priority of comparison: or < and < not***

                        idx = left[-1][0]

                        ans[idx] += len(right)

                        enums[i] = left.pop()

                    else:

                        enums[i] = right.pop()

            return enums

        divcon(list(enumerate(nums)))

        return ans

**543. Diameter of Binary Tree**

*Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.*

def diameterOfBinaryTree(self, root):

self.diameter = 0

def findMax(node):

if not node:  ***# error: fogot to change from root to node***

return 0

left = findMax(node.left)

right = findMax(node.right)

self.diameter = max(left + right, self.diameter)

return 1 + max(left, right)

findMax(root)

return self.diameter

**544. Output Contest Matchese**

*During the NBA playoffs, we always arrange the rather strong team to play with the rather weak team, like make the* ***rank 1*** *team play with the* ***rank nth*** *team, which is a good strategy to make the contest more interesting. Now, you're given n teams, you need to output their final contest matches in the form of a string.*

def findContestMatch(self, n):

def pairContest(teams):

L = len(teams)

mid = L // 2

res = []

for i in range(mid):

res.append('(' + teams[i] + ',' + teams[-i-1] + ')')

return res

ans = [str(i + 1) for i in range(n)]

while len(ans) > 1:

ans = pairContest(ans)

return ans[0] if ans else ''

**668. Kth Smallest Number in Multiplication Table**

*Nearly every one have used the Multiplication Table. But could you find out the k-th smallest number quickly from the multiplication table?*

***# In binary search, the left and right is continuous, but the multiply table is sparse. Thus the result may not in the table. So, we can't use count == k to return, we have to use count >= k to let the function converge by itself.***

def findKthNumber(self, m, n, k):

        left, right = 1, m\*n

        while left < right:

            mid = (left + right) // 2  ***# error, here must be int, so it's // not /***

            count = 0

            for i in range(m):

                count += min(mid//(i+1), n)  ***# error, forgot to plus 1***

            if count >= k: # use this to converge

                right = mid

            else:

                left = mid + 1

        return left

**361. Bomb Enemy**

*Given a 2D grid, each cell is either a wall 'W', an enemy 'E' or empty '0', return the maximum enemies you can kill using one bomb. The bomb kills all the enemies in the same row and column from the planted point until it hits the wall.*

***# logistic error: at first, I think we should compute the kills for each cell. Actually, we only need to recompute when we meet a 'Wall'. Thus, it requires us to have variables to store the row and col kills. Since we loop row by row, every time we start row\_kill from 0, we only need to remember this row. But for the col\_kill, it's not start from 0, so we have to have a list to store the pre\_col\_kill to add up.***

def maxKilledEnemies(self, grid):

        if not grid or not grid[0]:

            return 0

        rows, cols = len(grid), len(grid[0])

        ans = 0

        cols\_kill =  [0]\*cols

        row\_kill = 0

        for i in range(rows):

            for j in range(cols):

***# only when we meet a 'W', we need to recalculate the kills***

                if j == 0 or grid[i][j-1] == 'W':

                    row\_kill = 0

                    for k in range(j, cols):

                        if grid[i][k] == 'W':

                            break

                        row\_kill += grid[i][k] == 'E'

***# only when we meet a 'W', we need to recalculate the kills***

                if i == 0 or grid[i-1][j] == 'W':

                    cols\_kill[j] = 0

                    for k in range(i, rows):

                        if grid[k][j] == 'W':

                            break

                        cols\_kill[j] += grid[k][j] == 'E'

***# when we meet a '0', just add row kills and col kills together***

                if grid[i][j] == '0':

                    ans = max(ans, row\_kill + cols\_kill[j])

        return ans

**314. Binary Tree Vertical Order Traversal**

*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.*

def verticalOrder(self, root):

if not root:

return []

res = collections.defaultdict(list)

q = collections.deque([(0, root)])

while q:

idx, node = q.popleft()

res[idx].append(node.val)

if node.left:

q.append((idx-1, node.left))  ***# error: wrote to q.left***

if node.right:

q.append((idx+1, node.right))

*# we can also use one line:*

*return [res[key] for key in sorted(res)]  # when we use iter funs, res is equal to res.keys()*

ans = []

for key in sorted(res.keys()):

ans.append(res[key])

return ans

**94. Binary Tree Traversal (in-order & pre-order & post-order)**

*Given a binary tree, return its nodes' values.*

***# Approach I, recursive.***

def inorderTraversal(self, root):

ans = []

def dfs(node):

if not node:

return

# *ans.append(node.val)* ***# pre-order***

if node.left:

dfs(node.left)

ans.append(node.val)  ***# in-order***

if node.right:

dfs(node.right)

*# ans.append(node.val)* ***# post-order***

dfs(root)  ***# error: forgot to call dfs*** return ans

***# Approach II iterative. Actually, it's same with recursive, just use a stack to obvious store the depth***

***# iterative pre-order.***

def preorderTraversal(self, root):

    if not root:

        return []

    q = [root]

    ans = []

    while q:

        node = q.pop()

        ans.append(node.val)

        if node.right:

            q.append(node.right)

        if node.left:

            q.append(node.left)

    return ans

***# iterative in-order.***

def inorderTraversal(self, root):

    if not root:

        return []

    q = []

    node = root

    ans = []

    while q or node:

        if node:

            q.append(node)

            node = node.left

        else:

            node = q.pop()

            ans.append(node.val)

            node = node.right

    return ans

***# iterative post-order.***

def postorderTraversal(self, root):

    q = []

    pre\_node = None

    node = root

    ans = []

    while node or q:

        if node:

            q.append(node)

            node = node.left

        else:

            cur = q[-1]

            if cur.right and cur.right is not pre\_node:

                node = cur.right

            else:

                ans.append(cur.val)

                pre\_node = q.pop()

    return ans

**545. Boundary of Binary Tree**

*Given a binary tree, return the values of its boundary in anti-clockwise direction starting from root. Boundary includes left boundary, leaves, and right boundary in order without duplicate nodes.*

***# note: we can also use recursive sub methods to traverse, here is iterative for practice.***

def boundaryOfBinaryTree(self, root):

    if not root:

        return []

    boundary = [root.val]  ***# note: at first, put the root in to boundary to reduce corner case***

    node = root.left  ***# we can only start from the left one***

    while node:

        if node.left or node.right:

            boundary.append(node.val)

        node = node.left if node.left else node.right  ***# note: if no left, we can go right***

    q = [root]  # begin to deal with all the leaves

    while q:

        node = q.pop()

        if node != root and not node.left and not node.right:  ***# except the root***

            boundary.append(node.val)

            continue

        if node.right:

            q.append(node.right)

        if node.left:

            q.append(node.left)

    q = []

    node = root.right  ***# only deal with the right***

    while node or q:

        if node:

            if node.right or node.left:

                q.append(node)

            node = node.right if node.right else node.left  ***# note: same with the left boundary***

        else:

            temp = q.pop()

            boundary.append(temp.val)

    return boundary

**549. Binary Tree Longest Consecutive Sequence II**

*Given a binary tree, you need to find the length of Longest Consecutive Path in Binary Tree. Especially, this path can be either increasing or decreasing. For example, [1,2,3,4] and [4,3,2,1] are both considered valid, but the path [1,2,4,3] is not valid. On the other hand, the path can be in the child-Parent-child order, where not necessarily be parent-child order.*

***# node: kind of trivial, think carefully. Where to recursive, where to compare, increase, and how to return***

def longestConsecutive(self, root):

self.ans = 0

def subPath(node):

if not node:

return 0, 0

inc, dec = 1, 1

if node.left:

left\_inc, left\_dec = subPath(node.left)

if node.left.val == node.val + 1:

inc = left\_inc + 1

elif node.left.val == node.val - 1:

dec = left\_dec + 1

if node.right:

right\_inc, right\_dec = subPath(node.right)

if node.right.val == node.val + 1:

inc = max(inc, right\_inc + 1)

elif node.right.val == node.val - 1:

dec = max(dec, right\_dec + 1)

self.ans = max(self.ans, dec + inc -  1) ***# error: forgot to sum up***

return inc, dec

subPath(root)

return self.ans

**298. Binary Tree Longest Consecutive Sequence**

*Given a binary tree, find the length of the longest consecutive sequence path. The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).*

***# note: this kind of question, if we compare children to this node, we have to take care the two child like below. So there are lot of if else. We can also do it with put parent as a parameter, then the dfs only needs to compare the node itself with the parent, it may be easier.***

***# with depth parameter***

def longestConsecutive(self, root):

    if not root:

        return 0

    self.ans = 0

    def dfs(node, long):

        self.ans = max(self.ans, long)

        if not node:

            return

        if node.left:

            if node.left.val == node.val + 1:

                dfs(node.left, long+1)

            else:

                dfs(node.left, 1)

        if node.right:

            if node.right.val == node.val + 1:

                dfs(node.right, long+1)

            else:

                dfs(node.right, 1)

    dfs(root, 1)

    return self.ans

***# more clear way***

def longestConsecutive(self, root):

    self.ans = 0

    def subPath(node):

        if not node:

            return 0

        sub = 1

        if node.left:

            left\_sub = subPath(node.left)

            if node.left.val == node.val + 1:

                sub = left\_sub + 1

        if node.right:

            right\_sub = subPath(node.right)

            if node.right.val == node.val + 1:

                sub = max(sub, right\_sub + 1)

        self.ans = max(self.ans, sub)

        return sub

    subPath(root)

    return self.ans

**665. Non-decreasing Array**

*Given an array with n integers, your task is to check if it could become non-decreasing by modifying at most 1 element. We define an array is non-decreasing if array[i] <= array[i + 1] holds for every i (1 <= i < n).*

def checkPossibility(self, nums):

    L = len(nums)

    count = 0

    for i in range(1, L): ***# error: we should loop all the L, not L-1, since we need add count***

        if nums[i] < nums[i-1]:

            count += 1

*# Actually, we can check count > 1 here, just: if count > 1: return False and return True at last*

            pre\_num = nums[i-2] if i > 1 else float('-inf')

  next \_num  = nums[i+1] if i+1 < L else float('inf')

if nums[i-1] > next\_num and nums[i] < pre\_num:  ***# error: it's AND, wrote to OR***

                return False

    return count <= 1

**526. Beautiful Arrangement**

*Suppose you have N integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these N numbers successfully if one of the following is true for the ith position (1 <= i <= N) in this array:*

*The number at the ith position is divisible by i.*

*i is divisible by the number at the ith position.*

*Now given N, how many beautiful arrangements can you construct?*

***# note: i in [1, N], so we have to use range(1, N+1) to fit.***

***# the whole method is a simple backtracing***

def countArrangement(self, N):

self.count = 0

visited = [False] \* (N + 1)

def addBeauty(pos):

if pos > N:  
 self.count += 1

for i in range(1, N+1):

if not visited[i] and (pos % i == 0 or i % pos == 0):

visited[i] = True

addBeauty(pos+1)

visited[i] = False

addBeauty(1)

return self.count

**667. Beautiful Arrangement II**

*Given two integers n and k, you need to construct a list which contains n different positive integers ranging from 1 to n and obeys the following requirement:*

*Suppose this list is [a1, a2, a3, ... , an], then the list [|a1 - a2|, |a2 - a3|, |a3 - a4|, ... , |an-1 - an|] has exactly k distinct integers. If there are multiple answers, print any of them.*

***# the idea is: we first get 1 distinct |a-b| that A = [1,2,3,4….N]***

***# then get k-1 distinct |a-b| that B = [1, k-1, 2, k-2…]***

***# then add N to every elements of B, and combine A, B together, we can get the result***

def constructArray(self, n, k):

ans = [i for i in range(1, n-k)]

start = n - k

for i in range(k+1):

if i % 2 == 0:

ans.append(i // 2 + start)

else:

ans.append(k - i//2 + start)

return ans

**676. Implement Magic Dictionary**

*Implement a magic directory with buildDict, and search methods. For the method buildDict, you'll be given a list of non-repetitive words to build a dictionary. For the method search, you'll be given a word, and judge whether if you modify exactly one character into another character in this word, the modified word is in the dictionary you just built.*

class MagicDictionary(object):

    def \_\_init\_\_(self):

        self.dic = collections.defaultdict(list)

    def buildDict(self, dict):

        for word in dict:

            self.dic[len(word)].append(word)

    def search(self, word):

*# it can compress to one column, but it's hard to understand*

*# return any(sum(c1 != c2 for c1, c2 in zip(w, word)) == 1 for w in self.dic[len(word)])*

        for w in self.dic[len(word)]:

            count = 0

            for c1, c2 in zip(w, word):

                if c1 != c2:

                    count += 1

            if count == 1:

                return True

        return False

**551. Student Attendance Record I**

You are given a string representing an attendance record for a student. The record only contains the following three characters: 'A' : Absent. 'L' : Late. 'P' : Present. A student could be rewarded if his attendance record doesn't contain more than one 'A' (absent) or more than two continuous 'L' (late).

def checkRecord(self, s):

late = 0

absent = 0

for c in s:

if c == 'L':

late += 1

else:

late = 0

if c == 'A':

absent += 1

if absent > 1 or late > 2:

return False

return True

**552. Student Attendance Record II**

*Given a positive integer n, return the number of all possible attendance records with length n, which will be regarded as rewardable. The answer may be very large, return it after mod 109 + 7. A record is regarded as rewardable if it doesn't contain more than one 'A' (absent) or more than two continuous 'L' (late).*

***# note: the idea is, if without 'A', then dp[i] = ('P' + dp[i-1]) + ('PL' + dp[i-2]) + ('PLL' + dp[i-3])***

***# Add 'A', if we add to i, so there are i-1 digits before, and n-i digits behind, so: res += dp[i-1] \* dp[n-i]***

def checkRecord(self, n):

    if n == 0:

        return 0

    if n == 1:

        return 3

    M = int(1e9 + 7) ***# note: here we should int(), otherwise it's float, and the result may change***

    dp = [1, 2 ,4]

    for i in range(3, n+1):

        dp.append((dp[i-1] + dp[i-2] + dp[i-3]) % M)

    res = dp[-1]

    for i in range(1, n+1):

        res += dp[i-1] \* dp[n-i] % M

    return res % M

**121. Best Time to Buy and Sell Stock**

*Say you have an array for which the ith element is the price of a given stock on day i. If you were only permitted to complete* ***at most one*** *transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit. Note that you cannot sell a stock before you buy one.*

def maxProfit(self, prices):

profit = 0

low = float('inf')

for price in prices:

profix = max(profit, price - low)

low = min(low, price)

return profit

**122. Best Time to Buy and Sell Stock II**

*Say you have an array for which the ith element is the price of a given stock on day i. Design an algorithm to find the maximum profit. You may complete* ***as many transactions as you like*** *(i.e., buy and sell stock multiple times).*

def maxProfit(self, prices):

profit = 0

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

if prices[i] > prices[i-1]:

profit += prices[i] - prices[i-1]

return profit

**123. Best Time to Buy and Sell Stock III**

*Say you have an array for which the ith element is the price of a given stock on day i. Design an algorithm to find the maximum profit. You may complete* ***at most two*** *transactions. Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).*

def maxProfit(self, prices):

buy1, buy2, sell1, sell2 = float('-inf'), float('-inf'), 0, 0

for price in prices:

sell2 = max(sell2, buy2 + price)

buy2 = max(buy2, sell1 - price)

sell1 = max(sell1, buy1 + price)

buy1 = max(buy1, -price)  
 return sell2

**188. Best Time to Buy and Sell Stock IV**

*Say you have an array for which the ith element is the price of a given stock on day i. Design an algorithm to find the maximum profit. You may complete* ***at most k*** *transactions.*

def maxProfit(self, k, prices):

L = len(prices)

if k >= L/2:

return bestTimeBuySellStock\_II

dp = [[0]\*(L+1) for i in range(k+1)]  *# dp[i][j] stands for with i transactions, in jth day, the max profit*

for i in range(1, k+1):  *# why k in the outer loop? because if we do i transaction on the base of i-1*

profit = float('-inf')

for j, price in enumerate(prices, 1):

dp[i][j] = max(dp[i][j-1], price + profit)  *# sell or not sell*

profit = max(profit, dp[i-1][j-1] - price)  *# buy or not buy*

return dp[-1][-1]

**309. Best Time to Buy and Sell Stock with Cooldown**

*Design an algorithm to find the maximum profit. You may complete as many transactions as you like. But after you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)*

def maxProfit(self, prices):

    if len(prices) < 2:

        return 0

    sell, buy, prev\_sell, prev\_buy = 0, -prices[0], 0, 0

    for price in prices:

        prev\_buy = buy

        buy = max(prev\_sell - price, prev\_buy)

        prev\_sell = sell

        sell = max(prev\_buy + price, prev\_sell)

    return sell

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

*Your are given an array of integers prices, for which the i-th element is the price of a given stock on day i; and a non-negative integer fee representing a transaction fee. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction. You may not buy more than 1 share of a stock at a time (ie. you must sell the stock share before you buy again.) Return the maximum profit you can make.*

def maxProfit(self, prices, fee):

    buy, sell = float('-inf'), 0

    for price in prices:

        sell = max(sell, buy + price - fee)

        buy = max(buy, sell - price)

    return sell

**General Thinking about Stock problems:**

**DP:**

For every day, we have three action: buy, sell, rest

For action buy and sell, it depends on two status: hold, no hold

DP[day][trans times][no hold / hold]  # only hold can sell. no hold can buy

**Base cases:**

T[-1][k][0] = 0, T[-1][k][1] = -Infinity  *# if there is no stock*

T[i][0][0] = 0, T[i][0][1] = -Infinity  *# if there is no transaction*

**Recurrence relations:**

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])

T[i][k][1] = max(T[i-1][k][1], T[i-1][k-1][0] - prices[i])

**Case I: k = 1**

T[i][1][0] = max(T[i-1][1][0], T[i-1][1][1] + prices[i])

T[i][1][1] = max(T[i-1][1][1], T[i-1][0][0] - prices[i])  # since all T[i][0][0] = 0

  = max(T[i-1][1][1], -prices[i])

*# code*

*T\_i10, T\_i11 = 0, float('-inf')   
for price in prices:  
     T\_i10 = max(T\_i10, T\_i11 + price)  
     T\_i11 = max(T\_i11, -price)  
return T\_i10*

**Case II: k = inf**

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])

T[i][k][1] = max(T[i-1][k][1], T[i-1][k-1][0] - prices[i])  # since T[i-1][k-1][0] = T[i-1][k][0]

              = max(T[i-1][k][1], T[i-1][k][0] - prices[i])

*# code*

*T\_ik0, T\_ik1 = 0, float('-inf')*

*for price in prices:*

*T\_ik0\_old = T\_ik0*

*T\_ik0 = max(T\_ik0, T\_ik1 + price)*

*T\_ik1 = max(T\_ik1, T\_ik0\_old - price)*

*return T\_ik0*

**Case III: k = 2**

T[i][2][0] = max(T[i-1][2][0], T[i-1][2][1] + prices[i])

T[i][2][1] = max(T[i-1][2][1], T[i-1][1][0] - prices[i])

T[i][1][0] = max(T[i-1][1][0], T[i-1][1][1] + prices[i])

T[i][1][1] = max(T[i-1][1][1], -prices[i])

*# code*

*T\_i10, T\_i11 = 0,  float('-inf')*

*T\_i20, T\_i21 = 0,float('-inf')*

*for price in prices:*

*T\_i20 = max(T\_i20, T\_i21 + price)*

*T\_i21 = max(T\_i21, T\_i10 - price)*

*T\_i10 = max(T\_i10, T\_i11 + price)*

*T\_i11 = max(T\_i11, -price)*

*return T\_i20*

**Case IV: k is arbitrary**

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])

T[i][k][1] = max(T[i-1][k][1], T[i-1][k-1][0] - prices[i])

*# code*

*T\_ik0 = [0] \* (k + 1])*

*T\_ik1 = [float('-inf')] \* (k + 1])*

*for price in prices:*

*for j in range(k, 0, -1):*

*T\_ik0[j] = max(T\_ik0[j], T\_ik1[j] + price)*

*T\_ik1[j] = max(T\_ik1[j], T\_ik0[j - 1] - price);*

*return T\_ik0[k]*

***# another approach***

*dp = [[0]\*(L+1) for i in range(k+1)]*

*for i in range(1, k+1):*

*profit = float('-inf')*

*for j, price in enumerate(prices, 1):*

*dp[i][j] = max(dp[i][j-1], price + profit)*

*profit = max(profit, dp[i-1][j-1] - price)*

**Case V: k = inf with cooldown**

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])

T[i][k][1] = max(T[i-1][k][1], T[i-2][k][0] - prices[i])

*# reduce to O(1) space*

*T\_ik0\_pre, T\_ik0, T\_ik1 = 0, 0, float('-inf')*

*for price in prices:*

*T\_ik0\_old = T\_ik0*

*T\_ik0 = max(T\_ik0, T\_ik1 + price)*

*T\_ik1 = max(T\_ik1, T\_ik0\_pre - price)*

*T\_ik0\_pre = T\_ik0\_old*

*return T\_ik0*

**Case VI: k = inf with transaction fee**

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])

T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i] - fee)

or

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i] - fee)

T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i])

*# code*

*T\_ik0, T\_ik1 = 0, float('-inf')*

*for price in prices:*

*T\_ik0\_old = T\_ik0*

*T\_ik0 = max(T\_ik0, T\_ik1 + price)*

*T\_ik1 = max(T\_ik1, T\_ik0\_old - price - fee)*

*or*

*T\_ik0 = max(T\_ik0, T\_ik1 + price - fee)*

*T\_ik1 = max(T\_ik1, T\_ik0\_old - price)*

*return T\_ik0*

**269. Alien Dictionary**

*There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of non-empty words from the dictionary, where words are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.*

***# error:  we should take care all the members, not only the useful ones. We have two methods to use:***

***# 1. initial all of them at the beginning.***

***# 2. after the initialization, deal with the idle members at one time.***

***# we can use a class or only a tuple, here we use class.***

class Node(object):

    def \_\_init\_\_(self):

***# here we should take care another thing.***

*# since we don't use the details of parent, we can also use number to store parent, but must make other changes!*

*# change children from* ***set*** *to* ***list*** *OR check whether this parent already* ***exist*** *before parent += 1!*

        self.parent = set()

        self.children = set()

class Solution:

    def alienOrder(self, words):

        if not words:

            return ''

*# method 1, initialize all the members*

*# cs = {}*

*# for word in words:*

*#     for c in word:*

*#         cs[c] = Node()*

***# here we use method 2, only initialize useful members***

        cs = collections.defaultdict(Node)

        L = len(words)

        for i in range(1, L):

            for c1, c2 in zip(words[i-1], words[i]):

                if c1 != c2:

                    cs[c1].children.add(c2)

                    cs[c2].parent.add(c1)

                    break

        p\_nodes = collections.deque([c for c in cs if len(cs[c].parent)==0])

*# if we use method 1, just:*

*res = ''*

***# method 2 need to take care idle members at here.***

        res = ''.join(set(''.join(words)) - set(cs))

        while p\_nodes:

            c = p\_nodes.popleft()

            res += c

            c\_node = cs.pop(c)

            for child in c\_node.children:

                cs[child].parent.remove(c)

                if len(cs[child].parent) == 0:

                    p\_nodes.append(child)

        return res if len(cs) == 0 else ''

**659. Split Array into Consecutive Subsequences**

*You are given an integer array sorted in ascending order (may contain duplicates), you need to split them into several subsequences, where each subsequences consist of at least 3 consecutive integers. Return whether you can make such a split.*

def isPossible(self, nums):

counts = collections.Counter(nums)

tails = collections.Counter()

for num in nums:

if counts[num] == 0:

continue

elif tails[num] > 0:

tails[num] -= 1

tails[num+1] += 1

elif counts[num+1] > 0 and counts[num+2] > 0:

counts[num+1] -= 1

counts[num+2] -= 1

tails[num+3] += 1

else:

return False

counts[num] -= 1

return True

**656. Coin Path**

*Given an array A (index starts at 1) consisting of N integers: A1, A2, ..., AN and an integer B(steps you can jump). If you step on the index i, you have to pay Ai coins. If Ai is -1, it means you can’t jump to this place. Now, you start from A1, find the path to reach AN with the minimum coins.  If there are multiple paths with the same cost, return the lexicographically smallest path. If it's not possible to reach the place indexed N then you need to return an empty array.*

***# there are lot of trick in this question.***

***# 1. To keep it in lexicographically, we have to loop from end to start. Of course, we can also from start to end, but we need more operations.***

***# 2. For each loop, we should use a min\_cost to remember the cost from this idx. If we init the cost to 'inf' directly, we cann't compare it with 'inf' + A[idx]***

***# 3. At last, If we want to get the path in one while loop, we should init the path[-1] = L, it's stand for the end.***

def cheapestJump(self, A, B):

    if not A:

        return []

    L = len(A)

    cost = [0] \* L

    path = [-1] \* L

    path[-1] = L

    for idx in range(L)[::-1]:

        min\_cost = float('inf')

        for step in range(B):

            n\_idx = idx + step + 1

            if n\_idx < L and A[n\_idx] != -1 and A[idx] + cost[n\_idx] < min\_cost:

                    min\_cost = cost[idx] = A[idx] + cost[n\_idx]

                    path[idx] = n\_idx

    res = []

    idx = 0

    while idx < L and path[idx] > 0:

        res.append(idx+1)

        idx = path[idx]

    return res if idx == L else []

**562. Longest Line of Consecutive One in Matrix**

*Given a 01 matrix M, find the longest line of consecutive ones The line could be horizontal, vertical, diagonal or anti-diagonal.*

def longestLine(self, M):

    if not M or not M[0]:

        return 0

    rows, cols = len(M), len(M[0])

*~~# dp = [[0]\*4 for \_ in range(rows) for \_ in range(cols)]~~****#******two error here: the order of cols and rows, the inner [] for cols***

    dp = [[[0]\*4 for \_ in range(cols)] for \_ in range(rows)]

    ans = 0

    for i in range(rows):

        for j in range(cols):

            if M[i][j] == 1:

                dp[i][j][0] = dp[i][j-1][0] + 1 if j > 0 else 1

                dp[i][j][1] = dp[i-1][j][1] + 1 if i > 0 else 1

                dp[i][j][2] = dp[i-1][j-1][2] + 1 if i > 0 and j > 0 else 1

                dp[i][j][3] = dp[i-1][j+1][3] + 1 if i > 0 and j+1 < cols else 1

                ans = max(ans, max(dp[i][j]))

    return ans

**652. Find Duplicate Subtrees**

*Given a binary tree, return all duplicate subtrees. For each kind of duplicate subtrees, you only need to return the root node of any one of them. Two trees are duplicate if they have the same structure with same node values.*

***# Approach I: time O(n2), space O(n2)***

def findDuplicateSubtrees(self, root):

    ans = []

    count = collections.Counter()

    def find(node):

        if not node: return '#'

        serial = '{},{},{}'.format(node.val, find(node.left), find(node.right))

        count[serial] += 1

        if count[serial] == 2:

            ans.append(node)

        return serial

    find(root)

    return ans

***# Approach II: time O(n), space O(n)***

def findDuplicateSubtrees(self, root):

trees = collections.defaultdict()

trees.default\_factory = trees.\_\_len\_\_

count = collections.Counter()

ans = []

def find(node):

if node:

sign = node.val, find(node.left), find(node.right)

nid = trees[sign]

count[key] += 1

if count[key] == 2:

ans.append(node)

return nid

find(root)

return ans

**560. Subarray Sum Equals K**

*Given an array of integers and an integer k, you need to find the total number of continuous subarrays whose sum equals to k.*

def subarraySum(self, nums, k):

    memo = collections.defaultdict(int)

    memo[0] = 1  ***# error: forgot to add base case***

    asum = 0

    ans = 0

    for num in nums:

        asum += num

        if (asum - k) in memo:

            ans += memo[asum-k]

        memo[asum] += 1

    return ans

**214. Shortest Palindrome** (**KMP** only generate **PMT***(Partial Match Table)*)

*Given a string s, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.*

def shortestPalindrome(self, s):

    def getTable(s):

        table = [-1] \* len(s)

        i, j = 0, -1

        while i < len(s)-1:

            if j == -1 or s[i] == s[j]:

                i += 1

                j += 1

                table[i] = j

            else:

                j = table[j]

        return table

    T = getTable(s + '#' + s[::-1])

    start = T[-1]+1

    return s[start:][::-1] + s

def shortestPalindrome(self, s):

def getTable(s):

table = [0]

j = 0

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

while j > 0 and s[j] != s[i]:

j = table[j - 1]

j = j + 1 if s[i] == s[j] else 0

table.append(j)

return table

T = getTable(s + '#' + s[::-1])

start = T[-1]

return s[start:][::-1] + s

def shortestPalindrome(self, s):

def getTable(s):

L = len(s)

table = [0] \* L

for i in range(1, L):

j = table[i-1]

while j > 0 and s[j] != s[i]:

j = table[j-1]

if s[j] == s[i]:

j += 1

table[i] = j

return table

    T = getTable(s + '#' + s[::-1])

    start = T[-1]

    return s[start:][::-1] + s

**397. Integer Replacement**

*Given a positive integer n and you can do operations as follow:*

*If n is even, replace n with n/2.*

*If n is odd, you can replace n with either n + 1 or n - 1.*

*What is the minimum number of replacements needed for n to become 1?*

def integerReplacement(self, n):

ans = 0

while n > 1:

ans += 1

if n % 2 == 0:                       *# f(1) = 0, f(2n) = 1 + f(n), f(2n + 1) = min(f(2n) + 1, f(2n + 2) + 1)*

n = n / 2                          *# f(2n + 1) = min(f(2n) + 1, f(n + 1) + 2)*

elif n % 4 == 1 or n == 3:    *# If n % 4 = 3 and n != 3, then f(n) = f(n + 1) + 1.*

n -= 1                             *# If n % 4 = 1 or n = 3, then f(n) = f(n - 1) + 1.*

else:

n += 1

return ans

**803. Bricks Falling When Hit**

*We have a grid of 1s (bricks) and 0s.  A brick will not drop if it is directly connected to the top of the grid or at least one of its (4-way) adjacent bricks will not drop. Now, we will do some erasures sequentially. Each time we will erase (i, j), and then some other bricks may drop. Return an array representing the number of bricks that will drop after each erasure in sequence.*

***# we cann't solve it only with DFS or BFS, cause if only suit hit the bricks from near to far. If we hit a far one, it only influence further ones, however we don't know which one of its neighbors is further or not. We can use additional matrix to store the level, but it may become very complex.***

def hitBricks(self, grid, hits):

    if not grid or not grid[0] or not hits:

        return []

    rows, cols = len(grid), len(grid[0])

    dirs = ((0, 1), (0, -1), (-1, 0), (1, 0))

***# used to set connected with top***

    def dfs(x, y):

        grid[x][y] = 2

        hold = 1

        for dx, dy in dirs:

            nx, ny = x + dx, y + dy

            if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 1:

                hold += dfs(nx, ny)

        return hold

***# check whether this cell is connected***

    def is\_connected(x, y):

        if x == 0:  ***# error: forgot this condition***

            return True

        for dx, dy in dirs:

            nx, ny = x + dx, y + dy

            if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 2:

                return True

        return False

***# begin the main logsitic***

    for x, y in hits: ***# first hit all the bricks at the beginning***

        grid[x][y] -= 1

    for y in range(cols): ***# connect the remain to the top***

        if grid[0][y] == 1:

            dfs(0, y)

***# add the hits one by one reversely, and check how many cells will be connected by this one***

    res = []

    for x, y in hits[::-1]:

        grid[x][y] += 1***# add the cell***

        if grid[x][y] and is\_connected(x, y):

            res.insert(0, dfs(x, y)-1)

        else:

            res.insert(0, 0) ***# error: forgot this condition***

    return res

**493. Reverse Paris**

*Given an array nums, we call (i, j) an important reverse pair if i < j and nums[i] > 2\*nums[j]. You need to return the number of important reverse pairs in the given array.*

def reversePairs(self, nums):

    self.ans = 0

    def merge(temp):

        L = len(temp)

        mid = L // 2

        if not mid:

            return temp

        l, r = merge(temp[:mid]), merge(temp[mid:])

        idx = 0

        i, j, n, ll, lr = 0, 0, 0, len(l), len(r)

        while i < ll:  ***# here is i < left\_length***

            while n < lr and l[i] > 2 \* r[n]:

                n += 1

            self.ans += n

            while j < lr and l[i] > r[j]:

                temp[idx] = r[j]

                j += 1

                idx += 1

            temp[idx] = l[i]

            i += 1  ***# mainly care i, the left part***

            idx += 1

        while idx < L:  ***# the right remains, add to res***

            temp[idx] = r[j]

            j += 1

            idx +=1

        return temp

    merge(nums)

    return self.ans

def reversePairs(self, nums):

    self.ans = 0

    def merge(temp):

        L = len(temp)

        mid = L // 2

        if not mid:

            return temp

        l, r = merge(temp[:mid]), merge(temp[mid:])

        i, j = 0, 0

        while i < len(l) and j < len(r):

            if l[i] > 2 \* r[j]:

                self.ans += len(l) - i

                j += 1

            else:

                i += 1

        return sorted(l + r)

    merge(nums)

    return self.ans

**284. Peeking Iterator**

*Given an Iterator class interface with methods: next() and hasNext(), design and implement a PeekingIterator that support the peek() operation -- it essentially peek() at the element that will be returned by the next call to next().*

class PeekingInterator:

def \_\_init\_\_(self, iterator):

self.iter = iterator

self.temp = self.iter.next() if self.iter.hasNext() else None

def peek(self):

return self.temp

def next(self):

t = self.temp

self.temp = self.iter.next() if self.iter.hasNext() else None

return t

def hasNext(self):

return self.temp != None

**288. Unique Word Abbreviation**

*An abbreviation of a word follows the form <first letter><number><last letter>. Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no other word from the dictionary has the same abbreviation.*

***# error: unique mean not exist or exist but only one time***

class ValidWordAbbr:

    def \_\_init\_\_(self, dictionary):

        self.brief = collections.defaultdict(set)

        for word in dictionary:

            bw = word

            if len(word) > 2:

                bw = word[0]+str(len(word)-2)+word[-1]

            self.brief[bw].add(word)

    def isUnique(self, word):

        bw = word

        if len(word) > 2:

            bw = word[0]+str(len(word)-2)+word[-1]

        return bw not in self.brief or len(self.brief[bw]) == 1 and self.brief[bw] == {word}

**66. Plus One**

*Given a non-empty array of digits representing a non-negative integer, plus one to the integer. The digits are stored such that the most significant digit is at the head of the list, and each element in the array contain a single digit. You may assume the integer does not contain any leading zero, except the number 0 itself.*

***# error: only add one, not add two nums***

def plusOne(self, digits):

    L = len(digits)

    idx = L-1

    while idx >= 0:

        if digits[idx] < 9:

            digits[idx] += 1

            return digits

        digits[idx] = 0

        idx -= 1

    res = [1] + [0] \* (L)

    return res

**230. Kth Smallest Element in a BST**

*Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.*

***# note: if we use recursive like this, we must use ans to store the val, since the stack still work after return***

def kthSmallest(self, root, k):

    self.count = 0

    self.ans = None

    def dfs(node):

        if node:

            dfs(node.left)

            self.count += 1

            if self.count == k:

                self.ans = node.val

                return

            dfs(node.right)

    dfs(root)

    return self.ans

def kthSmallest(self, root, k):

    count = 0

    stack, node = [], root

    while stack or node:

        if node:

            stack.append(node)

            node = node.left

        else:

            node = stack.pop()

            count += 1

            if count == k:

                return node.val

*# error: we should not verify if node.right*

            node = node.right  

**289. Game of Life**

*Given a board with m by n cells, each cell has an initial state live (1) or dead (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) with the following four rules:* ***1.*** *Any live cell with fewer than two live neighbors dies, as if caused by under-population.* ***2.*** *Any live cell with two or three live neighbors lives on to the next generation.* ***3.*** *Any live cell with more than three live neighbors dies, as if by over-population.****4.*** *Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction. Write a function to compute the next state (after one update) of the board given its current state. The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously.*

def gameOfLife(self, board):

    rows, cols = len(board), len(board[0])

    def check(x, y):

        live = 0

        for i in range(-1,2):

            for j in range(-1, 2):

                if i == 0 and j == 0: ***# error: can not put this in to below, because it may jump both i == 0 or j == 0!***

                    continue

                nx, ny = x + i, y + j

                if 0 <= nx < rows and 0 <= ny < cols and board[nx][ny] & 1: ***# we can add (i !=0 or j != 0) here***

                    live += 1

        return live

    for i in range(rows):

        for j in range(cols):

            live = check(i, j)

            if board[i][j] == 1 and (live == 2 or live == 3):

                board[i][j] = 3

            elif board[i][j] == 0 and live == 3:

                board[i][j] = 2

    for i in range(rows):

        for j in range(cols):

            board[i][j] >>= 1  ***# error: forgot to add =, here is >>=***

**282. Expression Add Operators**

*Given a string that contains only digits 0-9 and a target value, return all possibilities to add binary operators (not unary) +, -, or \* between the digits so they evaluate to the target value.*

def addOperators(self, num, target):

    ans = []

    def helper(start, cur, pre, path):

        if start == len(num):

            if cur == target:

                ans.append(path)

            return

        for i in range(start+1, len(num)+1):

            if num[start] == '0' and i > start+1: ***# error: here is '0', forgot it's string, wrote to 0***

                break

            val = int(num[start:i])

            if start == 0:

                helper(i, val, val, str(val))

            else:

                helper(i, cur + val, val, path + '+' + str(val))

                helper(i, cur - val, -val, path + '-' + str(val))

                helper(i, cur - pre + pre \* val, pre \* val, path + '\*' + str(val))

    helper(0, 0, 0, '')

    return ans

**10. Regular Expression Matching**

*Given an input string (s) and a pattern (p), implement regular expression matching with support for '.' and '\*'.*

*'.' Matches any single character.*

*'\*' Matches zero or more of the preceding element.*

def isMatch(self, s, p):

    dp = [[False] \* (len(p)+1) for \_ in range(len(s)+1)]

    dp[0][0] = True

    for i, c in enumerate(p, 1):   ***# base case***

        dp[0][i] = c == '\*' and dp[0][i-2]

    for i, ws in enumerate(s, 1):

        for j, wp in enumerate(p, 1):

            if ws == wp or wp == '.':

                dp[i][j] = dp[i-1][j-1]

            elif wp == '\*':

                dp[i][j] = dp[i][j-1] or dp[i][j-2]  ***# \* count as 1 or 0(delete pre one)***

                if p[j-2] == '.' or p[j-2] == ws:  # ***\* count multiple ws***

                    dp[i][j] |= dp[i-1][j]  # same with dp[i][j] = dp[i][j] or dp[i-1][j]

    return dp[-1][-1]

**20. Valid Parentheses**

*Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.*

def isVaild(self, s):

stack = []

pairs = {')':'(', '}':'{', ']':'['}

for c in s:

if c in '[{(':

stack.append(c)

else:

p = pairs[c]

if not stack or p != stack.pop():

return False

return not stack

**22. Generate Parentheses**

*Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.*

def generateParenthsis(self, n):  # time and space: O(22n/n)

res = []

def generate(pre, left, right):

if left:

generate(pre+'(', left-1, right)

if right > left:

generate(pre+')', left, right-1)

if not right:

res.append(pre)

generate('', n, n)

return res

**228. Summary Ranges**

*Given a sorted integer array without duplicates, return the summary of its ranges.*

*Input:  [0,2,3,4,6,8,9,12]*

*Output: ["0","2->4","6","8->9","12"]*

def summaryRanges(self, nums):

    res = []

    idx = 0

    while idx < len(nums):

        start = idx  ***# note: since we start from here, so, we should take care the end***

        while idx+1 < len(nums) and nums[idx+1] == nums[idx] + 1: ***# the end is confirmed by idx + 1***

            idx += 1

        if start == idx:

            res.append(str(nums[idx]))

        else:

            res.append('{}->{}'.format(nums[start], nums[idx]))

        idx += 1  ***# error: forgot this...***

    return res

**23. Merge k Sorted Lists**

*Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.*

***# note, we can also use a priorityqueue to solve this. The length of priority queue is K, because we only need to put each head node into it. Here is divide and conquer method.***

def mergeKLists(self, lists):

    def mergeTwoLists(one, two):

        head = pre = ListNode(0)  ***# error, forgot to generate a dummy node.***

        while one or two:

            if not one:

                pre.next = two

                return head.next

            elif not two:

                pre.next = one

                return head.next

            elif one.val <= two.val:

                pre.next = one

                pre = pre.next  ***# error, forgot this***

                one = one.next

            else:

                pre.next = two

                pre = pre.next ***# error, forgot this***

                two = two.next

        return head.next ***# we can also delete this line, because either one or two must be reach end at first***

    while len(lists) > 1:

        temp = []

        for i in range(0, len(lists), 2):

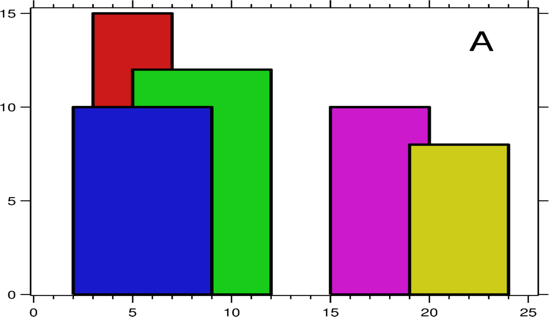
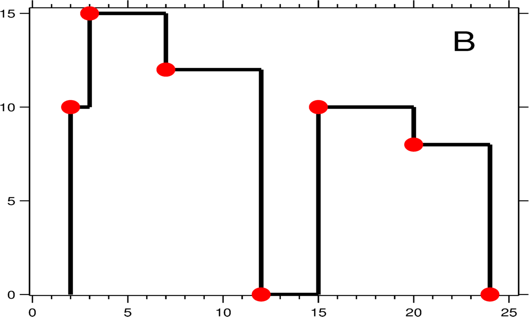
            temp.append(mergeTwoLists(lists[i], lists[i+1] if i+1 < len(lists) else None))

        lists = temp

    return lists[0] if lists else None ***# error, forgot this if else***

**218. The Skyline Problem**

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are **given the locations and height of all the buildings** as shown on a cityscape photo (Figure A), write a program to **output the skyline** formed by these buildings collectively (Figure B).

 [](https://leetcode.com/static/images/problemset/skyline1.jpg)

Each building is represented by a triplet of integers [Li, Ri, Hi], where Li and Ri are the x coordinates of the left and right edge of the ith building, respectively, and Hi is its height. It is guaranteed 0 ≤ Li, Ri ≤ INT\_MAX, 0 < Hi ≤ INT\_MAX, and Ri - Li > 0. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

For instance, Figure A are recorded as: [[2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8]]

The input list is already sorted in ascending order by the left x position Li.

The output list must be sorted by the x position.

**A key point is the left endpoint of a horizontal line segment**. The last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the **key points**  in Figure B should be :[[2 10],[3 15],[7 12],[12 0],[15 10],[20 8],[24, 0]]

***# note: The PriorityQueue, we can use get and put, but no peek(), and delete, we can use pq.queue[0] to get the peek, but can not use pq.queue.remove(item) to delete some element, it will modify the pq without reorder it! So it's better to use heapq on a q (list)***

def getSkyLine(self, buildings):

    points = defaultdict(list)

    for start, end, height in buildings:

        points[start].append(height)***# if height > 0, start***

        points[end].append(-height)  ***# if height < 0, end***

    res = []

    pre = 0

    q = [pre]

    xidx = sorted(points.keys())

    for x in xidx:

        for h in points[x]:

            if h > 0:***# if start, add to q, because heapq can only get min, so we add an '-'.***

                heapq.heappush(q, -h)

            else:  ***# if end, remove from the q, end influence. Here is a trick, we remove before we get highest from the***

                q.remove(h)***# priority queue, it can ensure get the right point without using pre\_idx***

                heapq.heapify(q)

        sky = -q[0]  # the highest one

        if sky != pre:

            res.append([x, sky])

            pre = sky

    return res

**224. Basic Calculator I**

*Implement a basic calculator to evaluate a simple expression string. The expression string may contain "(",  ")", "+", " - ", non-negative integers and empty spaces.*

def calculate(self, s)

    idx, L, res, stack, signs, sign = 0, len(s), 0, [], [], 1

    while idx < L:

        c = s[idx]

        if c == '(':

            stack.append(res)

            signs.append(sign)

            res, sign = 0, 1

        elif c == '-':         elif c == '+':

            sign = -1            sign = 1

        elif c == ')':

            res = stack.pop() + res\*signs.pop()  ***# error: wrote to res += stack.pop()\*signs.pop()***

        elif c.isdigit():

            start = idx

            while idx < L and s[idx].isdigit():

                idx += 1

            res += int(s[start:idx]) \* sign

            idx -= 1

        idx += 1

    return res

**227. Basic Calculator II**

*Implement a basic calculator to evaluate a simple expression string. The expression string contains only non-negative integers, +, -, \*, / operators and empty spaces . The integer division should truncate toward zero.*

***# note: we should use a sign to member the previous sign***

def calculate(self, s):

    idx, L, sign, stack, num =  0, len(s), '+', [], 0

    while idx < L:

        c = s[idx]

        if c.isdigit():

            start = idx

            while idx < L and s[idx].isdigit():

                idx += 1

            num = int(s[start:idx])

            idx -= 1

***# error: here is if!    NOT elif, otherwise idx == L-1 will not get***

**if** c in '+-\*/' or idx == L-1:  ***# error: forgot the idx == L-1***

            if sign == '+':

                stack.append(num)

            elif sign == '-':

                stack.append(-num)

            elif sign == '\*':

                stack.append(num \* stack.pop())

            elif sign == '/':

                stack.append(int(stack.pop() / num))  ***# use int() truncate to 0,  int(3.8/2) = 1, -3//2 = -2, int(-3/2) = -1***

            num = 0 ***# this line can be deleted***

            sign = c

        idx += 1

    return sum(stack) ***# sum([]) default is 0***

**388. Longest Absolute File Path**

*The string "dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsubdir2\n\t\t\tfile2.ext" represents:*

*dir*

*subdir1*

*file1.ext*

*subsubdir1*

*subdir2*

*subsubdir2*

*file2.ext*

*find the longest file's path from the given string*

def lengthLongestPath(self, input):

path = {0:0}

res = 0

for line in input.splitlines():

name = line.lstrip('\t')

deep = len(line) - len(name)

if '.' in name:

res = max(res, path[deep] + len(name))

else:

path[deep+1] = path[deep] + len(name) + 1

return res

**503. Next Greater Element II**

*Given a circular array (the next element of the last element is the first element of the array), print the Next Greater Number for every element. The Next Greater Number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, output -1 for this number.*

def nextGreaterElements(self, nums):

L = len(nums)

stack, res = [], [-1]\*L

for idx in range(2\*L):

r\_idx = idx % L

while stack and nums[stack[-1]] < nums[r\_idx]:

res[stack.pop()] = nums[r\_idx]

stack.append(r\_idx)

return res

**421. Maximum XOR of Two Numbers in an Array**

*Given a non-empty array of numbers, a0, a1, a2, … , an-1, Find the maximum result of ai XOR aj.*

def findMaximumXOR(self, nums):  
 ans, mask = 0, 0

for i in range(31, -1, -1):

mask = mask | (1 << i)

s = set()

for num in nums:

s.add(num & mask)

temp = max | (1 << i)

for h in s:

if temp ^ h in s: ***# if a ^ b = c,  a ^ c = b, then b ^ c = a***

ans = temp

break

return mask

**465. Optimal Account Balancing**

*A group of friends went on holiday and sometimes lent each other money. For example, Alice paid for Bill's lunch for $10. Then later Chris gave Alice $5 for a taxi ride. We can model each transaction as a tuple (x, y, z) which means person x gave person y $z. Assuming Alice, Bill, and Chris are person 0, 1, and 2 respectively (0, 1, 2 are the person's ID), the transactions can be represented as [[0, 1, 10], [2, 0, 5]]. Given a list of transactions between a group of people, return the minimum number of transactions required to settle the debt.*

def minTransfers(self, transactions):

balance = collections.defaultdict(int)

for p1, p2, money in transactions:

balance[p1] += money

balance[p2] -= money

rest = []

for p in balance:

if balance[p] != 0:

rest.append(balance[p])

def dfs(r1):

while r1 < len(rest) and rest[r1] == 0:

r1 += 1

ts = float('inf')

pre = 0 ***# jump the values equal to previous one, if we don't use it, get a TLE***

for r2 in range(r1+1, len(rest)):

if rest[r2] != pre and rest[r2]\*rest[r1] < 0:  ***# if r1 and r2 have different debt***

rest[r2] += rest[r1]  ***# put all r1's debt to r2, so r1 is clear***

ts = min(ts, 1 + dfs(r1 + 1)) ***# then 1(this transaction) + dfs next rest debt***

rest[r2] -= rest[r1] ***# after that, recover the r2 for next operation***

pre = rest[r2]  ***# remember this one for future comparing***

return ts if ts != float('inf') else 0 ***# if we can't make any transaction, return 0***

return dfs(0) ***# start from the first deb***t

**310. Minimum Height Trees**

*For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.*

def findMinheightTrees(self, n, edges):   **# note: excellent method!! from leaves to center**

if n == 1:

return [0]

adj = collections.defaultdict(set)

for a, b in edges:

adj[a].add(b)

adj[b].add(a)

leaves = [node for node in adj if len(adj[node]) == 1]

while n > 2: *# we can't use len(leaves) here, since n is the number of node, 100 nodes may have two leaves*

n -= len(leaves)

new\_leaves = []

for node in leaves:

edge = adj.pop(node).pop() ***# error, set can't be stared, just use pop(), because it has only one edge***

adj[edge].discard(node)

if len(adj[edge]) == 1:

new\_leaves.append(edge)

leaves = new\_leaves

return leaves

**369. Plus One Linked List**

Given a non-negative integer represented as non-empty a singly linked list of digits, plus one to the integer. You may assume the integer do not contain any leading zero, except the number 0 itself. The digits are stored such that the most significant digit is at the head of the list.

def plusOne(self, head):

    def reverse(node):

        pre = None

        while node:

*# node.next, pre, node = pre, node, node.next* ***# we can use this line to replace the follow 4 lines***

            next = node.next

            node.next = pre

            pre = node

            node = next

        return pre

    head = node = reverse(head)

    carry = 1

    while node and carry: ***# we can't use node.next to test here to make the if carry simply***

        carry, node.val = divmod(node.val + carry, 10)

        node = node.next

    head = reverse(head)

    if carry: ***# since the node is already none, we can only add a carry as a head to reversed list***

        node = ListNode(carry)

        node.next = head

        head = node

    return head

**765. Couples Holding Hands**

*N couples sit in 2N seats arranged in a row and want to hold hands. We want to know the minimum number of swaps so that every couple is sitting side by side. A swap consists of choosing any two people, then they stand up and switch seats. The people and seats are represented by an integer from 0 to 2N-1, the couples are numbered in order, the first couple being (0, 1), the second couple being (2, 3), and so on with the last couple being (2N-2, 2N-1). The couples' initial seating is given by row[i] being the value of the person who is initially sitting in the i-th seat.8*

def minSwapsCouples(self, row):

    N = len(row) // 2

    couple = [[] for \_ in range(N)]

    for i, p in enumerate(row):

        couple[p//2].append(i//2)  ***# we don't care the pid, so, only use cid(couple\_id)***

    adj = [[] for \_ in range(N)]

    for c1, c2 in couple:

        adj[c1].append(c2)

        adj[c2].append(c1)

    swaps = N  ***# this N is the key idea, N couples - the number of cycles is the answer.***

***# each cycle is we can keep changing until we match the end one to the first one***

    for c in range(N):

        if not adj[c]:

            continue

        swaps -= 1

        c1, c2 = c, adj[c].pop()

        while c2 != c:  ***# here is while not if    AND here is c2 != c not c2 != c1***

            adj[c2].remove(c1)

            c1, c2 = c2, adj[c2].pop()

    return swaps

**766. Toeplitz Matrix**

*A matrix is Toeplitz if every diagonal from top-left to bottom-right has the same element. Now given an M x N matrix, return True if and only if the matrix is Toeplitz.*

def isToeplitzMatrix(self, matrix):

***# note: it's better to use i,j or r,c rather than row, col since we may use row, col for enumerate or other purpose.***

    return all(r == 0 or c == 0 or matrix[r-1][c-1] == item

                        for r, row in enumerate(matrix)

                        for c, item in enumerate(row))

**239. Sliding Window Maximum**

*Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.*

def maxSildingWindow(self, nums, k):

if k > len(nums):

return []

L = len(nums)

dq = collections.deque()

ans = []

for idx, num in enumerate(nums):

while dq and dq[-1][-1] < num:

dq.pop()

dq.append((idx, num))

if idx >= k - 1:

while dq[0][0] < idx - k + 1:

dq.popleft()

ans.append(dq[0][-1])  ***# error: here is dp[0] not dp[-1] AND this line is in if***

return ans

**4. Median of Two Sorted Arrays**

*There are two sorted arrays nums1 and nums2 of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).*

def findMedianSortedArrays(self, nums1, nums2):

def find(start1, start2, k):

L1, L2 = len(nums1), len(nums2)

if start1 >= L1:

return nums2[start2 + k -1]

if start2 >= L2:

return nums1[start1 + k - 1]

if k == 1:

return min(nums1[start1], nums2[start2])

next\_start1 = start1 + k//2 - 1

half1 = nums1[cut1] if next\_start1 < L1 else float('inf')

next\_start2 = start2 + k//2 - 1

half2 = nums2[cut2] if next\_start1 < L2 else float('inf')

if half1 < half2:

return find(start1 + k//2, start2, k - k//2)

else:

return find(start1, start2 + k//2, k - k //2)

n = len(nums1) + len(nums2)

if n % 2 == 0:

return((find(0,0,n//2) + find(0,0,n//2+1)) / 2)

else:

return find(0, 0, n//2 + 1)

**163. Missing Ranges**

*Given a sorted integer array, where the range of elements are in the inclusive range [lower, upper], return its missing ranges.*

def findMissingRanges(self, nums, lower, upper):

    pre = lower - 1

    nums.append(upper + 1)

    ans = []

    for num in nums:

        if num == pre + 2:

            ans.append(str(num-1))

        elif num > pre + 2:

            ans.append('{}->{}'.format(pre+1, num-1))

        pre = num

    return ans

**31. Next Permutation**

*Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers. If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).*

def nextPermutation(self, nums):

def reverse(start, end):

while start < end:

nums[start], nums[end] = nums[end], nums[start]

start += 1

end -= 1

L = len(nums)

key\_idx =  L - 1

for i in range(L-2, -1, -1):

if nums[i] < nums[i+1]:

key\_idx = i

break  ***# error: forgot to beak***

if key\_idx == L - 1:

reverse(0, key\_idx)

else:

right = L - 1

while nums[right] <= nums[key\_idx]:

right -= 1

nums[key\_idx], nums[right] = nums[right], nums[key\_idx]

reverse(key\_idx + 1, L - 1)

**208.Implement Trie (Prefix Tree)**

*Implement a trie with insert, search, and startsWith methods.*

***# note, we can use an separate class TrieNode to store the dic and isWord, the keep a TrieNode object as a root in Trie. Or we can store the dic and word just in Trie itself. They are kind of same, but there are still some difference.***

1. ***TrieNode has no functions, just data structure.***
2. ***Trie only focus on functions, rather than storage.***
3. ***This separation makes the structure simple.***

class TrieNode(object):

def \_\_init\_\_(self):

self.dic = collections.defaultdict(TrieNode)

self.isWord = False

class Trie(object):

def \_\_init\_\_(self):

self.root = TrieNode()

def insert(self, word):

cur = self.root

for c in word:

cur = cur.dic[c]

cur.isWord = True

def search(self, word):

cur = self.root

for c in word:

if c in cur.dic:

cur = cur.dic[c]

else:

return False

return cur.isWord

def startsWith(self, prefix):  # return bool, if there is any word starts with prefix

cur = self.root

for c in prefix:

if c in cur.dic: ***# error: here is cur.dic not self.dic***

cur = cur.dic[c]

else:

return False

return cur.isWord or len(cur.dic) > 0 ***# error must be add > 0, because we need return bool***

**200. Number of Islands**

*Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.*

def numIslands(self, grid):

    dirs = ((0, 1), (0, -1), (-1, 0), (1, 0))

    def dfs(x, y):

        grid[x][y] = '0'

        for dx, dy in dirs:

            nx, ny = x + dx, y + dy

            if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] == '1':

                dfs(nx, ny)

    ans = 0

    for i in range(len(grid)):

        for j in range(len(grid[i])):

            if grid[i][j] == '1':

                ans += 1

                dfs(i, j)

    return ans

**305. Number of Islands II**

*A 2d grid map of m rows and n columns is initially filled with water. We may perform an addLand operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each addLand operation. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.*

***# note: it's a classical union-find problem, except a count function***

class UNF(object):

    def \_\_init\_\_(self):

        self.base = {}

        self.rank = {}

        self.count = 0

    def add(self, idx):

        self.count += 1

        self.base[idx] = idx

        self.rank[idx] = 1

***# there are 3 way to optimize find complex, path compression, halving, splitting***

    def find(self, idx):

*#         while idx != self.base[idx]:* ***# path halving***

*#             self.base[idx] = self.base[self.base[idx]]*

*#             idx = self.base[idx]*

*#         return idx*

*#         while idx != self.base[idx]:* ***# path splitting, kind of same with path halving***

*#             idx, self.base[idx] = self.base[idx], self.base[self.base[idx]]*

*#         return idx*

        if idx != self.base[idx]:  ***# path compression***

            self.base[idx] = self.find(self.base[idx])

        return self.base[idx]

    def union(self, idx1, idx2):

***# note: metion here, r1 and r2 are root for idx1 and idx2, if r1 != r2, we set one to another, that means the leader of that group betrys to the other group. All the member only follow the leader.***

        r1, r2 = self.find(idx1), self.find(idx2)

        if r1 == r2:

            return

        if self.rank[r1] > self.rank[r2]:

            r1, r2 = r2, r1

        self.base[r1] = r2 ***# error: base not rank***

        if self.rank[r1] == self.rank[r2]:

            self.rank[r2] += 1

        self.count -= 1

    def has(self, idx):

        return idx in self.base

class Solution:

    def numIslands2(self, m, n, positions):

        unf = UNF()

        ans = []

        dirs = ((0, 1), (0, -1), (1, 0), (-1, 0))

        for x, y in positions:

            idx = n \* x + y

            unf.add(idx)

            for dx, dy in dirs:

                nx, ny = x + dx, y + dy

                nidx = n \* nx + ny

                if *0 <= nx < m and 0 <= ny < n and unf.has(nidx):*

                    unf.union(idx, nidx)

            ans.append(unf.count)

        return ans

**463. Island Perimeter**

*You are given a two-dimensional grid where 1 represents land and 0 represents water. Grid cells are connected horizontally/vertically (not diagonally). There is exactly one island (i.e., one or more connected land cells) and the island doesn't have "lakes" (water inside that isn't connected to the water around the island). One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.*

def islandPerimeter(self, grid):

    if not grid or not grid[0]:

        return 0

    rows, cols = len(grid), len(grid[0])

    for i in range(rows):

        for j in range(cols):

            if grid[i][j] == 1:

                count, neighbor = 0, 0

                grid[i][j] = 2

                stack = [(i, j)]

                dirs = ((0, 1), (0, -1), (-1, 0), (1, 0))

                while stack:

                    x, y = stack.pop()

                    count += 1

                    for dx, dy in dirs:

                        nx, ny = x + dx, y + dy

                        if 0 <= nx < rows and 0 <= ny < cols:

                            if grid[nx][ny] != 0:

                                neighbor += 1

                            if grid[nx][ny] == 1:

                                grid[nx][ny] = 2

                                stack.append((nx, ny))

                return 4 \* count - neighbor

**44. Wildcard Matching**

*Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '\*'.*

* *'?' Matches any single character.*
* *'\*' Matches any sequence of characters (including the empty sequence).*

def isMatch(self, s, p)

    M, N = len(s), len(p)

    dp = [[False] \* (N + 1) for \_ in range(M + 1)]

    dp[0][0] = True

    for i, c in enumerate(p, 1):

*# if c == '\*' and dp[0][i-1]:*

*#     dp[0][i] = True*

        if c == '\*': ***# this base and the above one, both are OK***

            dp[0][i] = dp[0][i-1]

    for i, c\_s in enumerate(s, 1):

        for j, c\_p in enumerate(p, 1):

            if c\_p == c\_s or c\_p == '?':

                dp[i][j] = dp[i-1][j-1]  ***# error: wrote to ==***

            if c\_p == '\*':

                dp[i][j] = dp[i-1][j] or dp[i][j-1]

    return dp[-1][-1]

**173. Binary Search Tree Iterator**

class BSTIterator(object):

    def \_\_init\_\_(self, root):

        self.queue = []

        while root:

            self.queue.append(root)

            root = root.left

    def hasNext(self):

        return bool(self.queue)

    def next(self):

        temp = self.queue.pop()

        right = temp.right

        while right:

            self.queue.append(right)

            right = right.left

        return temp.val

**50. Pow(x, n)**

*Implement pow(x, n), which calculates x raised to the power n (xn).*

***# note: it's very intuitive to use recursive. shouldn't use iterative***

def myPow(self, x, n):

    if n == 0:

        return 1

    if n < 0:

        return 1 / self.myPow(x, -n)

    if n % 2:

        return x \* self.myPow(x, n-1)

    return self.myPow(x\*x, n // 2)

**166. Fraction to Recurring Decimal**

*Given two integers representing the numerator and denominator of a fraction, return the fraction in string format. If the fractional part is repeating, enclose the repeating part in parentheses.*

def fractionToDecimal(self, numerator, denominator):

    sign = '-' if numerator \* denominator < 0 else ''

    numerator, denominator = abs(numerator), abs(denominator)  ***# error: forgot this***

    n, remain = divmod(numerator, denominator)

    ans = [sign, str(n), '.']

    stack = []

    while remain not in stack:

        stack.append(remain)

        n, remain = divmod(remain \* 10, denominator)

        ans.append(str(n))

    idx = stack.index(remain)  ***# note, here is stack.index, not ans.index***

    ans.insert(idx+3, '(')  ***# note, here is 3***

    ans.append(')')

    return ''.join(ans).replace('(0)', '').rstrip('.')  ***# here are also some conner case***

**358. Rearrange String k Distance Apart**

*Given a non-empty string s and an integer k, rearrange the string such that the same characters are at least distance k from each other. All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string "".*

def rearrangeString(self, s, k):

    if k == 0:  ***# if no this line, it will be TEL***

        return s

    count = collections.Counter(s)

    chars = [[c, v] for c, v in count.items()]  ***# error: here can't be k, otherwise, this k will overwrite the input k***

    res = list(s)

    idx, L = 0, len(s)

    while idx < L:

        chars.sort(key=lambda item: item[1], reverse=True)

        for i in range(k):

            if i >= len(chars) or chars[i][1] == 0:  ***#error, here is >=, wrote to >***

                return ''.join(res) if idx == L else ''

            res[idx] = chars[i][0]

            chars[i][1] -= 1

            idx += 1

    return ''.join(res)

**323. Number of Connected Components in an Undirected Graph**

*Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.*

class UNF(object):

    def \_\_init\_\_(self, n):

        self.base = [i for i in range(n)]

        self.rank = [1] \* n

        self.count = n

    def find(self, n):

        if self.base[n] != n:

            self.base[n] = self.find(self.base[n])

        return self.base[n]

    def union(self, a, b):

        r1, r2 = self.find(a), self.find(b)

        if r1 == r2:

            return

        self.count -= 1

        if self.rank[r1] > self.rank[r2]:

            r1, r2 = r2, r1

        self.base[r1] = r2

        if self.rank[r1] == self.rank[r2]:

            self.rank[r2] += 1

class Solution:  *# basic union-find problem*

    def countComponents(self, n, edges):

        unf = UNF(n)

        for a, b in edges:

            unf.union(a, b)

        return unf.count

**351. Android Unlock Patterns**

*Given an Android 3x3 key lock screen and two integers m and n, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns of the Android lock screen, which consist of minimum of m keys and maximum n keys.*

*Rules for a valid pattern:*

1. *Each pattern must connect at least m keys and at most n keys.*
2. *All the keys must be distinct.*
3. *If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.*
4. *The order of keys used matters.*

def numberOfPatterns(self, m, n):

skip = [[0]\*10 for \_ in range(10)]

skip[1][3] = skip[3][1] = 2

skip[1][7] = skip[7][1] = 4

skip[3][9] = skip[9][3] = 6

skip[7][9] = skip[9][7] = 8

skip[1][9] = skip[9][1] = skip[2][8] = skip[8][2] = skip[3][7] = skip[7][3] = skip[4][6] = skip[6][4] = 5

visited = [False] \* 10

ans = 0

def dfs(cur, remain):

if remain < 0:

return 0

if remain == 0:

return 1

visited[cur] = True

temp = 0

for i in range(1, 10):

if not visited[i] and (skip[cur][i] == 0 or visited[skip[cur][i]]):

temp += dfs(i, remain - 1)

visited[cur] = False

return temp

for i in range(m, n+1):

ans += dfs(1, i - 1) \* 4  # 1, 3, 7, 9

ans += dfs(2, i - 1) \* 4  # 2, 4, 6, 8

ans += dfs(5, i - 1)  # 5

return ans

**356. Line Reflection**

*Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflect the given points.*

***# note: several cc: 1. points is None; 2. r\_line is not x=0, it can be any vertical line; 3.  may have duplicate points***

def isReflected(self, points):

    if not points:

        return True

    left, right = min(points)[0], max(points)[0]

    r\_sum = left + right

    for x, y in points:

        rp = [r\_sum - x, y]  ***# error: here is list not tuple***

        if rp not in points:

            return False

    return True

**318. Maximum Product of Word Lengths**

*Given a string array words, find the maximum value of length(word[i]) \* length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.*

***# NOTE: this method may be the common method for lots of items but only kind of informations needed problems.***

***# if we loop 2 times, it's O(N2), however, if we preprocess them and only leave useful information,***

***# the N may be reduced significantly.***

***# note: bit method uses 200ms, and normal method uses 1800ms, both are AC***

def maxProduct(self, words):

    dic = collections.defaultdict(int)

    for word in words:

        mask = 0

        for c in word:

            mask |= 1 << (ord(c) - ord('a'))

        dic[mask] = max(dic[mask], len(word))

    ans = 0

    for w1 in dic:

        for w2 in dic:

            if not (w1 & w2):

                ans = max(ans, dic[w1] \* dic[w2])

    return ans

def maxProduct(self, words):

    dic = collections.defaultdict(int)

    for word in words:

***# sorted() not xx.sort(), because sort() has no return***

***# key can't be list, because it's not hashable***

        key = ''.join(sorted(list(set(word))))

        dic[key] = max(dic[key], len(word))

    ans = 0

    for key in dic:

        for key2 in dic:

            if not (set(key) & set(key2)): ***# here is set op***

                ans = max(ans, dic[key] \* dic[key2])

    return ans

**354. Russian Doll Envelopes**

*You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope. What is the maximum number of envelopes can you Russian doll?*

***# note, we can't only sort and begin compute, because sort is only care w or h, and this problem care w and h together***

def maxEnvelopes(self, envelopes):

*if not envelopes:*  ***# note: this line can be omitted, since sort and bisect can be used on none lsit***

*return 0*

    envelopes.sort(key=lambda item: (item[0], -item[1]))

    ans = []                          ***# since we sort the envelopes by w ascending and h descending.***

    for w, h in envelopes:    ***# we only care increasing subsequece of h. this method come from the below one.***

        idx = bisect.bisect\_left(ans, h)

        if idx == len(ans):

            ans.append(h)

        else:

            ans[idx] = h

    return len(ans)

**300. Longest Increasing Subsequence.** O(nlogn)

*Given an unsorted array of integers, find the length of longest increasing subsequence.*

def lengthOfLIS(self, nums):

    ans = []

    for num in nums:

        idx = bisect.bisect\_left(ans, num)

        if idx == len(ans): ***# if num is bigger than previous ones, add to behind***

            ans.append(num)

        else: ***# if it's smaller than someone, replace it, but it doesn't influnce the result, only all the bigger one are***

            ans[idx] = num                  ***# replaced by smaller ones, then, the next coming one can add to behind***

    return len(ans)

**128. Longest Consecutive Sequence**  O(n)

*Given an unsorted array of integers, find the length of the longest consecutive elements sequence.*

def longestConsecutive(self, nums):

    nums = set(nums)

    ans = 0

    for num in nums:

        if num - 1 not in nums: ***# this line guarantee there is no needless/redundant whiles***

            cur = num

            longth = 0

            while cur in nums:

                cur += 1

                longth += 1

            ans = max(ans, longth)

    return ans

**320. Generalized Abbreviation**

*Write a function to generate the generalized abbreviations of a word. "word"->Output:["word","1ord","w1rd", …,"w2d",...,'3d',"4"]*

def generateAbbreviations(self, word):

ans = []

def generate(start, count, temp):

if start >= len(word):

ans.append(temp + (str(count) if count else ''))  ***# error: if xxx else xxx has lower priority than +***

else:

generate(start + 1, count + 1, temp)

generate(start + 1, 0, temp + (str(count) if count else '') + word[start]) ***# error: count is 0***

generate(0, 0, '')

return ans

**313. Super Ugly Number**

*Write a program to find the nth super ugly number. Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of size k.*

def nthSuperUglyNumber(self, n, primes):

    ugly = [0] \* n

    ugly[0] = 1

    pq = queue.PriorityQueue()

    for p in primes:

        pq.put((p, 0, p))  ***# val, idx, p => val = p \* ugly[idx]***

    for i in range(1, n):

        ugly[i] = pq.queue[0][0]

        while pq.queue[0][0] == ugly[i]:

            val, idx, p = pq.get()

            next\_idx = idx + 1  ***# next\_idx = cur\_idx + 1***

            pq.put((p \* ugly[next\_idx], next\_idx, p))

    return ugly[-1]

**363. Max Sum of Rectangle No Larger Than K**

*Given a non-empty 2D matrix matrix and an integer k, find the max sum of a rectangle in the matrix such that its sum is no larger than k.*

def maxSumSubmatrix(self, matrix, k):

if not matrix or not matrix[0]:

        return 0

    m, n = len(matrix), len(matrix[0])

    ans = float('-inf')

    for i in range(n):

        asum = [0] \* m

        for j in range(i, n):

            for r in range(m):

                asum[r] += matrix[r][j]

            temp\_sums = [0]  ***# error: forgot to init with a 0***

            cur\_sum = 0  ***# error: forgot to use this accumulate sum***

            for s in asum:

                cur\_sum += s

                bigger = cur\_sum - k

                idx = bisect.bisect\_left(temp\_sums, bigger)

                if idx < len(temp\_sums):

                    ans = max(ans, cur\_sum - temp\_sums[idx])

                bisect.insort(temp\_sums, cur\_sum)

    return ans

**353. Design Snake Game**

*Design a Snake game that is played on a device with screen size = width x height. The snake is initially positioned at the top left corner (0,0) with length = 1 unit. You are given a list of food's positions in row-column order. When a snake eats the food, its length and the game's score both increase by 1. Each food appears one by one on the screen. For example, the second food will not appear until the first food was eaten by the snake. When a food does appear on the screen, it is guaranteed that it will not appear on a block occupied by the snake.*

class SnakeGame:

    def \_\_init\_\_(self, width, height, food):

        self.rows, self.cols = height, width

        self.foods = food

        self.count = 0

        self.body = collections.deque([(0, 0)])

    def move(self, direction):

        hx, hy = self.body[-1]

        if direction == 'U':

            hx -= 1

        elif direction == 'L':

            hy -= 1

        elif direction == 'R':

            hy += 1

        elif direction == 'D':

            hx += 1

        if hx < 0 or hx >= self.rows or hy < 0 or hy >= self.cols:

            return -1  ***# out of boundary***

        if *self.count < len(self.foods) and hx == self.foods[self.count][0] and hy == self.foods[self.count][1]:*

            self.count += 1  ***# get food  ERROR: forgot to check count < len(food)***

        else:

            self.body.popleft()  ***# just move***

        if (hx, hy) in self.body:  ***# ERROR: add at last, otherwise, head always in body***

            return -1  # hit itself

        self.body.append((hx, hy))

        return self.count

**321. Create Maximum Number**

*Given two arrays of length m and n with digits 0-9 representing two numbers. Create the maximum number of length k <= m + n from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits.*

def maxNumber(self, nums1, nums2, k):

def getK(nums, k):

useless = len(nums) - k

res = []

for num in nums:

while useless > 0 and res and res[-1] < num:

res.pop()

useless -= 1

res.append(num)

return res[:k]

def merge(a, b):

return [max(a, b).pop(0) for \_ in a+b]  ***# here is pop(0)***

ans = []

for i in range(k+1):

if i <= len(nums1) and k-i <= len(nums2):

part1, part2 = getK(nums1, i), getK(nums2, k-i)

cur = merge(part1, part2)

ans = max(ans, cur)

return ans

**769. Max Chunks To Make Sorted**

*Given an array arr that is a permutation of [0, 1, ..., N - 1], we split the array into some "chunks" (partitions), and individually sort each chunk.  After concatenating them, the result equals the sorted array. What is the most number of chunks we can make?*

def maxChunksToSorted(self, arr):

    ans, N = 0, len(arr)

    high = 0

    for idx, num in enumerate(arr):

        high = max(high, num)

        if high == idx:

            ans += 1

    return ans

**768. Max Chunks To Make Sorted II**

*This question is the same as "Max Chunks to Make Sorted" except the integers of the given array are not necessarily distinct, the input array could be up to length 2000, and the elements could be up to 10\*\*8.*

def maxChunksToSorted(self, arr):

count, order = collections.Counter(), []

for num in arr:

count[num] += 1

order.append((num, count[num]))

ans, cur = 0, None

for x, y in zip(order, sorted(order)):

cur = max(cur, x)

if cur == y:

ans += 1

return ans

**286. Walls and Gate**

*You are given a m x n 2D grid initialized with these three possible values. (-1: A wall; 0: A gate; INF: Infinity means an empty room). Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF.*

def wallsAndGates(self, rooms):

    if not rooms or not rooms[0]: return ***# we can use enumerate to get the gates to avoid the base case check***

    rows, cols = len(rooms), len(rooms[0])

    gates = [(r, c, 0) for r in range(rows) for c in range(cols) if rooms[r][c] == 0]

    dirs = ((0, 1), (0, -1), (1, 0), (-1, 0))

    for x, y, d in gates:  ***# we can use gate as stack directly, since for itself is kind of deque***

*# stack = collections.deque([(r, c, 0)])* ***# of course, we can also run dfs as usual***

*# while stack:*

*#     x, y, d = stack.popleft()*

        for dx, dy in dirs:

            nx, ny, nd = x + dx, y + dy, d + 1

            if 0 <= nx < rows and 0 <= ny < cols and rooms[nx][ny] > nd:

                rooms[nx][ny] = nd

                gates.append((nx, ny, nd))

**293. Flip Game**

*You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move and therefore the other person will be the winner. Write a function to compute all possible states of the string after one valid move.*

def generatePossibleNextMoves(self, s):

    ans = []

    for i in range(1, len(s)):

        if s[i-1] == '+' and s[i] == '+':

            ans.append(s[:i-1] + '--' + s[i+1:])

    return ans

**294. Flip Game II**

*Write a function to determine if the starting player can guarantee a win.*

***# note: the logic of this recursive for first player is kind of trick. we don't need to use a bool to check which player***

def canWin(self, s):

    if '++' not in s:

        return False

    for i in range(1, len(s)):

        if s[i-1] == '+' and s[i] == '+':

            rest = s[:i-1] + '--' + s[i+1:]

            if not self.canWin(rest):

                return True

    return False

**280. Wiggle Sort**

*Given an unsorted array nums, reorder it in-place such that nums[0] <= nums[1] >= nums[2] <= nums[3]....*

def wiggleSort(self, nums):

    for i in range(1, len(nums)):

        if i % 2:  ***# note: we can use if (i%2) ^ (nums[i] > nums[i-1]): xxx to substitute all the if else below***

            if nums[i] < nums[i-1]:

                nums[i], nums[i-1] = nums[i-1], nums[i]

        else:

            if nums[i] > nums[i-1]:

                nums[i], nums[i-1] = nums[i-1], nums[i]

**324. Wiggle Sort II**

*Given an unsorted array nums (contains duplicate elements), reorder it such that nums[0] < nums[1] > nums[2] < nums[3]....*

def wiggleSort(self, nums): ***# only three lines, AC***  
    nums.sort()

    half = len(nums[::2])  ***# (len(nums) - 1) // 2 + 1 same***

    nums[::2], nums[1::2] = nums[:half][::-1], nums[half:][::-1]

***# the below functions are for the right method =>>***

def partition(l, r):  ***# the 215 below use additional memory***

    pivot = nums[r]  ***# this one is in-place***

    idx = l

    for i in range(l, r):

        if nums[i] < pivot:

            nums[i], nums[idx] = nums[idx], nums[i]

            idx += 1

    nums[idx], nums[r] = nums[r], nums[idx]

    return idx

def findKthSmallest(l, r, k):

    if l <= r:

        n = partition(l, r)

        if n + 1 == k:

            return nums[n]

        elif n + 1 < k:

            return findKthSmallest(n + 1, r, k)

        else:

            return findKthSmallest(l, n - 1, k)

***# we can also do it with O(N), but it got TLE***

def wiggleSort(self, nums):

    L = len(nums)

    def r\_idx(i):

        return (1+2\*i) % (L|1)

    def swap(a, b):

        nums[a], nums[b] = nums[b], nums[a]

***# mention here is L//2 + 1***

    mid = ***findKthSmallest***(nums, L//2 + 1)***#see left***

    left, i, right = 0, 0, L-1

    while i <= right:

        if nums[r\_idx(i)] > mid:

            swap(r\_idx(left), r\_idx(i))

            left += 1

            i += 1

        elif nums[r\_idx(i)] < mid:

            swap(r\_idx(right), r\_idx(i))

            right -= 1

        else:

            i += 1

**215. Kth Largest Element in an Array**

*Find the kth largest element in an unsorted array.*

class Solution(object):

*def partition(self, nums, l, r):  # actually, the l and r can be omited, and use p=nums[-1], i in range(****L-1****)*

        idx, pivot = l, nums[r]

        for i in range(l, r):

            if nums[i] < pivot:

                nums[i], nums[idx] = nums[idx], nums[i]

                idx += 1

        nums[idx], nums[r] = nums[r], nums[idx]

        return idx

*def findKthSmallest(self, nums, k):*

        if nums:

            n = self.partition(nums, 0, len(nums)-1)  ***# the return n is idx, so idx+1 is number***

            if n + 1 == k:

                return nums[n]

            elif n + 1 < k:  ***# mention here, we should find from n+1 to find the k-n-1***

                return self.findKthSmallest(nums[n+1:], k - n -1)

            else:

                return self.findKthSmallest(nums[:n], k)

*def findKthLargest(self, nums, k):*

        random.shuffle(nums) ***# shuffle can guarantee O(N). needs: import random***

        L = len(nums)

        ks = L - k + 1 ***# find K largest, means L-k+1 smallest***

        return self.findKthSmallest(nums, ks)

**295. Find Median from Data Stream**

from queue import PriorityQueue

class MedianFinder(object):

    def \_\_init\_\_(self):

        self.small, self.large = PriorityQueue(), PriorityQueue()

    def addNum(self, num):

        self.large.put(num)***#ERROR: here, put into larger first, then get the small one put into small***

        self.small.put(-self.large.get())

        if self.small.qsize() > self.large.qsize():

            self.large.put(-self.small.get())

    def findMedian(self):

        if self.small.qsize() < self.large.qsize():

            return self.large.queue[0]

        else:

            return (self.large.queue[0] - self.small.queue[0]) / 2

**279. Perfect Squares**

*Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n.*

def numSquare(self, n):

if n < 2:

return n

squares = []

i = 1

while i\*i <= n:  ***# error: here is <=***

squares.append(i\*i)

i += 1

ans = 0

mids = {n}

while mids:

ans += 1

temp = set()

for num in mids:

for square in squares:

if square > num:

break

if square == num:

return ans

temp.add(num-square)

mids = temp

return ans

**367. Valid Perfect Square**

*Given a positive integer num, write a function which returns True if num is a perfect square else False.*

***# note: if we don't check num==1 at first, we should use while low <= high to contain this situation, and change the high = mid to high = mid - 1 to avoid dead loop, such as 5***

def isPerfectSquare(self, num):

    if num == 1:

        return True

    low, high = 1, num

    while low < high:

        mid = (low + high) // 2

        mid\_s = mid\*\*2

        if mid\_s == num:

            return True

        if mid\_s > num:

            high = mid

        else:

            low = mid + 1

    return False

**485. Max Consecutive Ones**

*Given a binary array, find the maximum number of consecutive 1s in this array.*

def findMaxConsecutiveOnes(self, nums):

    ans, count = 0, 0

    for num in nums:

        if num == 0:

            count = 0

        else:

            count += 1

            ans = max(ans, count)

    return ans

**487. Max Consecutive Ones II**

*Given a binary array, find the maximum number of consecutive 1s in this array if you can flip at most one 0.*

def findMaxConsecutiveOnes(self, nums):  
 ans = 0

left, zeros, k = 0, [], 1  ***# if we can filp k 0s***

for i, num in enumerate(nums):

if num == 0:

zeros.append(i)

if len(zeros) > k:

left = zeros.pop(0) + 1

***# note: every num, we count the dis to left***

ans = max(ans, i - left + 1)

return ans

***# for k == 1, simple version***

def findMaxConsecutiveOnes(self, nums):

    ans = 0

    pre\_zero, start = -1, 0

    for i, num in enumerate(nums):

        if num == 0:

            start = pre\_zero + 1

            pre\_zero = i

        ans = max(ans, i - start + 1)

    return ans

**400. Nth Digit**

*Find the nth digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, …*

def findNthDigit(self, n):

    digits, count, start = 1, 9, 1

    while n > digits \* count:

        n -= digits \* count

        digits += 1

        count \*= 10

        start \*= 10

***# n is how many digits left start from start***

    offset, nth = divmod(n-1, digits) ***# n-1 because start is already counted as 1 digit***

    num = str(start + offset)  ***# start + offset is the num that contains the nth digit***

    return int(num[nth])  ***# error: return type is int***

**401. Binary Watch**

*A binary watch has 4 LEDs on the top which represent the hours (0-11), and the 6 LEDs on the bottom represent the minutes (0-59). Each LED represents a zero or one. Given a non-negative integer n which represents the number of LEDs  on, return all possible times the watch could represent.*

def readBinaryWatch(self, num):

ans = []

for h in range(12): ***# note: it's difficult to solve binary so we can solve reversely.***

for m in range(60):

if (bin(h) + bin(m)).count('1') == num:  ***# bin(num) return a str***

ans.append('{}:{:02d}'.format(h, m)) ***# {:02d} note here is : in {} not %***

return ans

**312. Burst Balloons**

*Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] \* nums[i] \* nums[right] coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent. Find the maximum coins you can collect by bursting the balloons wisely.*

def maxCoins(self, nums):

    nums = [1] + nums + [1]  ***# add the start and end basis***

    L = len(nums)

    dp = [[0] \* L for \_ in range(L)] ***# how much we can get between left & right***

    for d in range(2, L): ***# the distance between left and right***

        for left in range(L-d): ***# the left is at most L - distance***

            right = left + d

            for i in range(left + 1, right): ***# current burst balloon***

***# cur \* left \* right is the score for this brust + already got from left to cur and cur to right***

                dp[left][right] = max(dp[left][right], nums[left]\*nums[i]\*nums[right] + dp[i][right] + dp[left][i])

    return dp[0][-1]

**374. Guess Number Higher or Lower**

*We are playing the Guess Game. The game is as follows: I* ***pick a number from 1 to n****. You have to guess which number I picked. Everytime you can call a pre-defined API guess(int num) which returns 3 possible results (-1: low, 1: high, or 0: correct)*

***# basic binary search***

def guessNumber(self, n):

    low, high = 1, n

    while low < high:

        mid = (low + high) // 2

        r = guess(mid)

        if r == 0:

            return mid

        elif r == -1:

            high = mid

        else:

            low = mid + 1

    return low

**375. Guess Number Higher or Lower II**

*the rule is same with above one, however, when you guess a particular number x, and you guess wrong, you pay $x. Given a particular n ≥ 1, find out how much money you need to have to guarantee a win.*

def getMoneyAmout(self, n):  T: O(N3)  S: O(N2)

    dp = [[0] \* (n + 1) for \_ in range(n + 1)]  ***# dp[i][j] means the cost for i to j***

    def tryNums(start=1, end=n):

        if start >= end:

            return 0

        if dp[start][end]:

            return dp[start][end]

        money = float('inf')

        for num in range(start, end + 1): ***# try every num between start and end***

***# before we got num, we have to from left or right, try all of them, then choose the bigger one***

            temp = num + max(tryNums(start, num-1), tryNums(num+1, end))

            money = min(money, temp)  ***# after try all the numbers in previous line, choose the min cost***

        dp[start][end] = money

        return money

    return tryNums()

**308. Range Sum Query 2D - Mutable**

*Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).*

class NumMatrix(object):

    def \_\_init\_\_(self, matrix):

        if not matrix or not matrix[0]:

            self.sums = []

        else:

            self.matrix = matrix

            rows, cols = len(matrix), len(matrix[0])

            self.sums = [[0]\*(cols+1) for \_ in range(rows+1)] ***# here is len+1, to avoid check out boundary***

            for r in range(rows):

                for c in range(cols):

***# here is several ways to compute the sum, this one is simple***

                    self.sums[r+1][c+1] = self.sums[r+1][c] + self.sums[r][c+1] + matrix[r][c] - self.sums[r][c]

    def update(self, row, col, val):

        if self.sums:

            rows, cols = len(self.matrix), len(self.matrix[0])

            diff = val - self.matrix[row][col]

            self.matrix[row][col] = val  ***# error: forgot to update matrix for further use***

            for r in range(row, rows):

                for c in range(col, cols):

                    self.sums[r+1][c+1] += diff

    def sumRegion(self, row1, col1, row2, col2):

        if not self.sums:

            return 0                          ***# since sums already contain redundant boundaries, it becomes simple***

        return *self.sums[row2+1][col2+1] - self.sums[row1][col2+1] - self.sums[row2+1][col1] + self.sums[row1][col1]*

**373. Find K Pairs with Smallest Sums**

*You are given two integer arrays nums1 and nums2 sorted in ascending order and an integer k. Define a pair (u,v) which consists of one element from the first array and one element from the second array. Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums.*

***# note: we only want to get k, so for each array, we at most choose k.***

*import queue*

def kSmallestPairs(self, nums1, nums2, k):

if not nums1 or not nums2 or not k:

return []

pq = queue.PriorityQueue()

ans = []

for i in range(min(k, len(nums1))):

pq.put((nums1[i]+nums2[0], nums1[i], nums2[0], 0))

while k > 0 and not pq.empty():

ss, n1, n2, i = pq.get()

ans.append((n1, n2))

k -= 1

if i == len(nums2) - 1:  ***# note: here is len - 1***

continue

pq.put((n1+nums2[i+1], n1, nums2[i+1], i+1))

return ans

**687. Longest Univalue Path**

*Given a binary tree, find the length of the longest path where each node in the path has the same value. This path may or may not pass through the root. The length of path between two nodes is represented by the number of edges between them.*

def longestUnivaluePath(self, root):

    self.ans = 0

    def path(node):

        if not node:

            return 0

        left, right = path(node.left), path(node.right) ***# we should search first, not put it under if condition***

        cur\_left ,cur\_right = 0, 0

        if node.left and node.val == node.left.val:

            cur\_left = left + 1  ***# if we put path(left) here to replace left, it will be jumped when cur.val != cur.left.val***

        if node.right and node.val == node.right.val:

            cur\_right = right + 1

        self.ans = max(self.ans, cur\_left + cur\_right)

        return max(cur\_left, cur\_right)

    path(root)

    return self.ans

**689. Maximum Sum of 3 Non-Overlapping Subarrays**

*In a given array nums of positive integers, find three non-overlapping subarrays with maximum sum. Each subarray will be of size k. Return the result as a list of indices representing the starting position. If same, return the lexicographically smallest one.*

def maxSumOfThreeSubarrays(self, nums, k):

    sums, left\_sum, right\_sum = [], [], []

    temp = 0

    for i in range(len(nums)):

        temp += nums[i]

        if i >= k:  ***#error: here is i >= k, forgot =***

            temp -= nums[i-k]

        if i >= k - 1:  ***# error: here is i >= k - 1, forgot =, and append under if, not directly***

            sums.append(temp)

    best = 0  ***# for left***

    for i in range(len(sums)):

        if sums[i] > sums[best]:

            best = i

        left\_sum.append(best)

    best = len(sums) - 1  ***# for right***

    for i in range(len(sums)-1, -1, -1):

        if sums[i] >= sums[best]:  ***# error: here is >=, forgot =***

            best = i

        right\_sum.insert(0, best)

    ans = None  ***# main logic***

    for i in range(k, len(sums)-k):

        l, r = left\_sum[i-k], right\_sum[i+k]

        if not ans:

            ans = [l, i, r]

        else:

            pl, pi, pr = ans

            if sums[l] + sums[i] + sums[r] > sums[pl] + sums[pi] + sums[pr]:

                ans = [l, i, r]

    return ans

**524. Longest Word in Dictionary through Deleting**

*Given a string and a string dictionary, find the longest string in the dictionary that can be formed by deleting some characters of the given string. If there are more than one possible results, return the longest word with the smallest lexicographical order. If there is no possible result, return the empty string.*

def findLongestWord(self, s, d):

    def check(w):

        if len(w) > len(s):

            return False

        i, j = 0, 0

        while j < len(s) and i < len(w): ***# if I loop with j, and while s[j] != w[i] j++. it does not work...***

            if i < len(w) and w[i] == s[j]:  ***# note: here is if, not while***

                i += 1

            j += 1

        return i == len(w)

    ans = ''

    for word in d:

        if check(word):

            if len(word) > len(ans) or (len(word) == len(ans) and word < ans):

                ans = word

    return ans

**527. Word Abbreviation**

*Given an array of* ***N*** *distinct non-empty strings, generate minimal possible abbreviations for every word following rules below.*

1. *Begin with the first character and then the number of characters abbreviated, which followed by the last character.*
2. *If there are any conflict, that is more than one words share the same abbreviation, a longer prefix is used instead of only the first character until making the map from word to abbreviation become unique. In other words, a final abbreviation cannot map to more than one original words.*
3. *If the abbreviation doesn't make the word shorter, then keep it as original.*

def wordsAbbreviation(self, dict)

    brifs = collections.defaultdict(list)

    for i, word in enumerate(dict):

        brifs[word[0]+str(len(word))+word[-1]].append((i, word))

    ans = [None] \* len(dict)

    Trie = lambda: collections.defaultdict(Trie)

    for groups in brifs.values():

        trie = Trie()

        for i, word in groups:  ***# this loop just for generate a trie of this group***

            cur = trie

            for c in word[1:]:  ***# error: here is from [1:]***

                cur['#'] = 1 + cur.get('#', 0)***# use '#' to store the count of this c in this group, it's different with other c***

                cur = cur[c]

        for i, word in groups:  ***# the main logic for this group***

            cur = trie

            for prefix, c in enumerate(word[1:], 1):  ***# the same group may have different birf length***

                if cur['#'] == 1: break  ***# no other word use the same c at this length, then break***

                cur = cur[c]

            if len(word) - prefix - 1 > 1:***# this prefix still can use after above for loop***

                ans[i] = word[:prefix] + str(len(word)-prefix-1) + word[-1]

            else:

                ans[i] = word

    return ans

**684. Redundant Connection**

*# basic Union Find problem. find redundant connection in undirected graph*

def findRedundantConnection(self, edges):

    def find(x):

        if x not in base: base[x] = x

        if x != base[x]:

            base[x] = find(base[x])

        return base[x]

    def union(a, b):

        ra, rb = find(a), find(b)

        if ra != rb: base[ra] = rb

    for a, b in edges:

        if find(a) == find(b): return [a, b]

        union(a, b)

**685. Redundant Connection II**

*# this is a directed graph*

***# note: we can also use DFS to solve this problem***

def findRedundantDirectedConnection(self, edges):

    base = {}  # basic union-find functions

    def find(x):

        if x not in base: base[x] = x

        if x != base[x]:

            base[x] = find(base[x])

        return base[x]

    def union(a, b):

        ra, rb = find(a), find(b)

        if ra != rb:

            base[ra] = rb

***# if there is a small cycle, we need to find which edge should be deleted***

    two\_parent\_node, n = -1, len(edges)

    parent = collections.defaultdict(list)

    for edge in edges:

        u, v = edge[0], edge[1]

        parent[v].append(u)

        if len(parent[v]) == 2:

            two\_parent\_node = v

    can1, can2 = None, None

    if two\_parent\_node != -1:  ***# if we have a two\_p\_node, we find the two candidate edges***

        p1, p2 = parent[two\_parent\_node]

        can1, can2 = [p1, two\_parent\_node], [p2, two\_parent\_node]

    for u, v in edges:

        if [u, v] == can2: ***# delete this edge, don't union them***

            continue

        if find(u) == find(v): ***# u and v are already connected***

            if two\_parent\_node == -1: ***# if no two\_p\_node, means there only a big cycle***

                return [u, v]

            else: ***# there is a small cycle, since we already jump can2, so just return can1***

                return can1

        union(u, v)

    return can2

**531. Lonely Pixel I**

*Given a picture consisting of black and white pixels, find the number of black lonely pixels. The picture is represented by a 2D char array consisting of 'B' and 'W', which means black and white pixels respectively. A black lonely pixel is character 'B' that located at a specific position where the same row and same column don't have any other black pixels.*

def findLonelyPixel(self, picture):

    if not picture or not picture[0]:

        return 0

    rows, cols = len(picture), len(picture[0])

    col\_black = [0] \* cols

    for i in range(rows):

        for j in range(cols):

            if picture[i][j] == 'B':

                col\_black[j] += 1

    count = 0

    for i in range(rows):

        if picture[i].count('B') == 1:

            if col\_black[picture[i].index('B')] == 1:  ***# kind of wind***

                count += 1

    return count

**533. Lonely Pixel II**

*Given a picture consisting of black and white pixels, and a positive integer N, find the number of black pixels located at some specific row R and column C that align with all the following rules:*

1. *Row R and column C both contain exactly N black pixels.*
2. *For all rows that have a black pixel at column C, they should be exactly the same as row R*

def findBlackPixel(self, picture, N):

    if not picture or not picture[0]:

        return 0

    rows, cols = len(picture), len(picture[0])

    col\_black = [0] \* cols

    for i in range(rows):

        for j in range(cols):

            if picture[i][j] == 'B':

                col\_black[j] += 1

    same\_rows = collections.Counter(map(tuple, picture)) ***# find the same rows***

    ans = 0

    for row, count in same\_rows.items():

        if count == N and row.count('B') == N:

            for i, pixel in enumerate(row):

                if pixel == 'B' and col\_black[i] == N:

                    ans += N ***# error: here is + N, not + 1, because N rows are same***

    return ans

**598. Range Addition II**

*Given an m \* n matrix M initialized with all 0's and several update operations. Operations are represented by a 2D array, and each operation is represented by an array with two positive integers a and b, which means M[i][j] should be added by one for all 0 <= i < a and 0 <= j < b. Count the number of maximum integers in the matrix after performing all the operations.*

def maxCount(self, m, n, ops):

    for r, c in ops:  ***# just find the smallest cover range***

        m = min(m, r)

        n = min(n, c)

    return m\*n

**370. Range Addition**

*Assume you have an array of length n initialized with all 0's and are given k update operations. Each operation is represented as a triplet: [startIndex, endIndex, inc] which increments each element of subarray A[startIndex ... endIndex] (startIndex and endIndex inclusive) with inc. Return the modified array after all k operations were executed.*

def getModifiedArray(self, length, updates):

    nums = [0]\*length

    for start, end, inc in updates:

        nums[start] += inc

        if end+1 < length:  ***# note, here is end+1, end only influence items behind***

            nums[end+1] -= inc

    asum = 0

    for i in range(length):

        asum += nums[i]

        nums[i] = asum

    return nums

**346. Moving Average from Data Stream**

*Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.*class MovingAverage:

    def \_\_init\_\_(self, size):

        self.size = size

        self.queue = []

        self.asum = 0

    def next(self, val):

        self.queue.append(val)  ***# error, forgot this line***

        self.asum += val

        if len(self.queue) > self.size:

            self.asum -= self.queue.pop(0)

        return self.asum / len(self.queue)***# don't need check here, since we ken len<k with above line***

**327. Count of Range Sum**

*Given an integer array nums, return the number of range sums that lie in [lower, upper] inclusive. Range sum S(i, j) is defined as the sum of the elements in nums between indices i and j (i ≤ j), inclusive.*

def countRangeSum(self, nums, lower, upper):

    sums =[0]

    for num in nums:

        sums.append(sums[-1]+num)

    def sort(low, hi):

        mid = (low + hi) // 2

        if mid == low: return 0

        count = sort(low, mid) + sort(mid, hi)

        i = j = mid

        for left in sums[low:mid]:

            while i < hi and sums[i] - left < lower:

                i += 1

            while j < hi and sums[j] - left <= upper:

                j += 1

            count += j - i

        sums[low:hi] = sorted(sums[low:hi])

        return count

    return sort(0, len(sums))

**348. Design Tic-Tac-Toe**

*Design a Tic-tac-toe game that is played between two players on a n x n grid. You may assume the following rules:*

1. *A move is guaranteed to be valid and is placed on an empty block.*
2. *Once a winning condition is reached, no more moves is allowed.*
3. *A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.*

class TicTacToe:

    def \_\_init\_\_(self, n):

        self.n = n

        self.rows = [0] \* n

        self.cols = [0] \* n

        self.dia = 0

        self.adia = 0

    def move(self, row, col, player):

        toAdd = 1 if player == 1 else -1

        self.rows[row] += toAdd

        self.cols[col] += toAdd

        if row == col:

            self.dia += toAdd

        if col == (self.n - row - 1):

            self.adia += toAdd

        if *abs(self.rows[row]) == self.n or abs(self.cols[col]) == self.n or abs(self.dia) == self.n or abs(self.adia) == self.n:*

                return player

        return 0

**326. Power of Three**

*Given an integer, write a function to determine if it is a power of three.*

def isPowerOfThree(self, n):

# return Math.log(n) / Math.log(3) % 1 == 0  ***# 3.2 % 1 = 0.2 .  not AC becasue has little remain***

return n > 0 and 1162261467 % n == 0

**451. Sort Characters By Frequency**

*Given a string, sort it in decreasing order based on the frequency of characters.*

def frequencySort(self, s):

    count = collections.Counter(s)

    fres = count.most\_common() ***# this method can sort the count with frequence***

    res = ''

    for c, v in fres:

        res += c \* v

    return res

**459. Repeated Substring Pattern**

*Given a non-empty string check if it can be constructed by taking a substring of it and appending multiple copies of the substring together.*

def repeatedSubstringPattern(self, s):

    if not s:

        return False

    ss = (s + s)[1:-1]

    return ss.find(s) != -1

**444. Sequence Reconstruction**

*Check whether the original sequence org can be uniquely reconstructed from the sequences in seqs. The org sequence is a permutation of the integers from 1 to n, with 1 ≤ n ≤ 104. Reconstruction means building a shortest common supersequence of the sequences in seqs (i.e., a shortest sequence so that all sequences in seqs are subsequences of it). Determine whether there is only one sequence that can be reconstructed from seqs and it is the org sequence.*

def sequenceReconstruction(self, org, seqs):

    adj, parents = collections.defaultdict(list),  collections.defaultdict(int)

    nodes = set()

    for arr in seqs:

        nodes |= set(arr)

        for i in range(len(arr)):

            if i == 0:

                parents[arr[i]] += 0

            if i < len(arr) - 1:

                adj[arr[i]].append(arr[i+1])

                parents[arr[i+1]] += 1

    cur = [k for k in parents if parents[k] == 0]

    res = []

    while len(cur) == 1:

        cur\_num = cur.pop()

        res.append(cur\_num)

        for node in adj[cur\_num]:

            parents[node] -= 1

            if parents[node] == 0:

                cur.append(node)

    if len(cur) > 1:

        return False

    return len(res) == len(nodes) and res == org

**425. Word Squares**

*Given a set of words (without duplicates), find all word squares you can build from them. A sequence of words forms a valid word square if the kth row and column read the exact same string. For example, the word sequence ["ball","area","lead","lady"] forms a word square because each word reads the same both horizontally and vertically.*

1. *All words will have the exact same length.*
2. *Word length is at least 1 and at most 5.*

def wordSquares(self, words):

    self.n, dic = len(words[0]), collections.defaultdict(list)

    for word in words:

        for i in range(self.n):

            dic[word[:i]].append(word)

    res = []

    def build(square):

        if len(square) == self.n:

            res.append(square)

            return

        prefix = ''

        for row in square:

            prefix += row[len(square)]

        for word in dic[prefix]:

            build(square + [word])

    for word in words:

        build([word])

    return res

**422. Valid Word Square**

*Given a sequence of words, check whether it forms a valid word square.*

def validWordSquare(self, words):

    for i, row in enumerate(words):

        for j, c in enumerate(row):  ***# error: forgot to check i, j***

            if j >= len(words) or i >= len(words[j]) or words[i][j] != words[j][i]:

                return False

    return True

**460. LFU Cache (Least Frequently Used)**

***get(key)*** *- Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.*

***put(key, value)*** *- Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least frequently used item before inserting a new item. For the purpose of this problem, when there is a tie (i.e., two or more keys that have the same frequency), the least recently used key would be evicted.*

class LFUCache:

    def \_\_init\_\_(self, capacity):

        self.capacity = capacity

        self.val = {}

        self.count = {}

        self.lists = collections.defaultdict(list)

        self.min = -1

    def get(self, key):

        if key not in self.val: return -1

        count = self.count[key]

        self.count[key] += 1

        self.lists[count].remove(key)

        if count == self.min and len(self.lists[count]) == 0:

            self.min += 1

        self.lists[count+1].append(key)

        return self.val[key]

    def put(self, key, value):

        if not self.capacity: return

        if key in self.val:

            self.val[key] = value

            self.get(key)

            return

        if len(self.val) == self.capacity:

            evict = self.lists[self.min][0]

            self.lists[self.min].remove(evict)

            self.val.pop(evict)

            self.count.pop(evict)

        self.val[key] = value

        self.count[key] = 1

        self.min = 1

        self.lists[1].append(key)

**469. Convex Polygon**

*Given a list of points that form a polygon when joined sequentially, find if this polygon is convex.*

def isConvex(self, points):

    def check(a, b, c):

        (ax, ay), (bx, by), (cx, cy) = a, b, c  ***# note, this kind of from to unpack multiple values***

        return (cy - by) \* (bx - ax) - (cx - bx) \* (by - ay)

    neg, pos = False, False

    N = len(points)

    points += points[:2]

    for i in range(N):

        a, b, c = points[i:i+3]

        convex = check(a, b, c)

        if convex > 0:

            neg = True

        if convex < 0:  # can't use else here, we should jump 0

            pos = True

    return False if (neg and pos) else True

**418. Sentence Screen Fitting**

*Given a rows x cols screen and a sentence represented by a list of non-empty words, find how many times the given sentence can be fitted on the screen.*

def wordsTyping(self, sentence, rows, cols):

s = ' '.join(sentence)

s += ' '

start, L = 0, len(s)

for i in range(rows):

start += cols  ***# first, we add the whole line***

while start > 0 and s[start % L] != ' ':  ***# then find the previous ' '***

start -= 1

start += 1 ***# then move next***

*# we can also check first, then while, it's same*

*if s[start % l] == ' ':  # if end with ' ', move next*

*start += 1*

*else:  # else, go back to find the previous ' '*

*while start > 0 and s[(start-1) % L] != ' ':*

*start -= 1*

return start // L

**249. Group Shifted Strings**

*Given a string, we can "shift" each of its letter to its successive letter, for example: "abc" -> "bcd". We can keep "shifting" which forms the sequence:"abc" -> "bcd" -> ... -> "xyz". Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.*

def groupString(self, strings):

    dic = collections.defaultdict(list)

    for s in strings:

        if not s: continue

        temp = str(len(s)) + '#' ***# we use str to store the key, actually if we use 1<<c-a also AC, but it has some***

        for c in s[1:]:                 ***# other issues, for example, abcd and acbd will be grouped into the same group***

            temp += str((ord(c)-ord(s[0])) % 26) + ','

        dic[temp].append(s)

    return list(dic.values())  ***# error: dic.values() not a list! we need to change it to list***

**270. Closest Binary Search Tree Value**

*Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.*

def closestValue(self, root, target):

    if not root: return

    node, ans = root, float('inf')

    while node:

        ans = node.val if abs(node.val - target) < abs(ans - target) else ans

        node = node.left if node.val > target else node.right

    return ans

**272. Closest Binary Search Tree Value II**

*Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.*

***# note: we can't only use the solution above to get path=>sort=>output path[:k], because K may be greater than depth***

def cloesetKValues(self, root, target, k):

    ans = []

    big, small  = [], [] ***# big store nodes bigger than t, small opposite***

    node = root

    while node:

        if node.val > target:

            big.append(node)

            node = node.left

        else:

            small.append(node)

            node = node.right

    def getNext(bigger=True):  ***# get a bigger one or smaller one***

        node = big.pop() if bigger else small.pop()

        cur = node.right if bigger else node.left  ***# get nodes that bigger than big[-1] or smaller than small[-1]***

        while cur:

            big.append(cur) if bigger else small.append(cur)

            cur = cur.left if bigger else cur.right ***# if get next bigger, it's the left iterative of bigger one, vice-versa***

        return node.val

    while k:

        if not big and not small:

            break

        if not big:

            ans.append(getNext(bigger=False))

        elif not small:

            ans.append(getNext(bigger=True))

        elif abs(target-small[-1].val) < abs(target-big[-1].val):

            ans.append(getNext(bigger=False))  ***# if small is good, get next small***

        else:

            ans.append(getNext(bigger=True))

        k -= 1

    return ans

**530. Minimum Absolute Differece in BST**

*Given a binary search tree with non-negative values, find the minimum absolute difference between values of any two nodes.*

def getMinimumDifference(self, root):

    ans = float('inf')

    stack = [(root, float('-inf'), float('inf'))]

    while stack:

*# l\_b is left boundary, r\_b is right boundary*

        node, l\_b, r\_b = stack.pop()

        ans = min(ans, node.val - l\_b, r\_b - node.val)

        if node.left:

            stack.append((node.left, l\_b, node.val))

        if node.right:

            stack.append((node.right, node.val, r\_b))

    return ans

***# or just store the vals inorder, then get the min difference. if not bst, just sort the vals***

def getMinimumDifference(self, root):

    ans = float('inf')

    vals, node, stack = [], root, []

    while node or stack:

        if node:

            stack.append(node)

            node = node.left

        else:

            node = stack.pop()

            vals.append(node.val)

            node = node.right

    for i in range(1, len(vals)):

        ans = min(ans, vals[i]-vals[i-1])

    return ans

**417. Pacific Atlantic Water Flow**

*Given an m x n matrix of non-negative integers representing the height of each unit cell in a continent, the "Pacific ocean" touches the left and top edges of the matrix and the "Atlantic ocean" touches the right and bottom edges. Water can only flow in four directions (up, down, left, or right) from a cell to another one with height equal or lower. Find the list of grid coordinates where water can flow to both the Pacific and Atlantic ocean.*

***# error: if only one row or only one col, we can't use if … elif … elif … to add all the boundary***

***# we can't add [x,y] while looping, since there may some duplicate, and if we change to set, set.add function must add hashable objects, not list. so if we use set to solve duplicate, we should change to list at last.***

def pacificAtlantic(self, matrix):

    if not matrix or not matrix[0]:

        return []

    rows, cols, ans, p\_s, a\_s = len(matrix), len(matrix[0]),  [], [], []

    p\_v, a\_v = [[False]\*cols for \_ in range(rows)], [[False]\*cols for \_ in range(rows)]

    for i in range(rows):

        p\_s.append((i, 0))

        a\_s.append((i, cols-1))

        p\_v[i][0] = True

        a\_v[i][cols-1] = True

    for j in range(cols):

        p\_s.append((0, j))

        a\_s.append((rows-1, j))

        p\_v[0][j] = True

        a\_v[rows-1][j] = True

    dirs = ((0, 1), (0, -1), (1, 0), (-1, 0))

    def dfs(stack, visited):

        while stack:

            x, y = stack.pop()

            for dx, dy in dirs:

                nx, ny = x + dx, y + dy

                if *0 <= nx < rows and 0 <= ny < cols and not visited[nx][ny] and matrix[nx][ny] >= matrix[x][y]*:

                    visited[nx][ny] = True

                    stack.append((nx, ny))

    dfs(p\_s, p\_v)

    dfs(a\_s, a\_v)

    return [[i, j] for i in range(rows) for j in range(cols) if p\_v[i][j] and a\_v[i][j]]

**471. Encode String with Shortest Length**

*Given a non-empty string, encode the string such that its encoded length is the shortest.*

*The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets is being repeated k times.*

*k will be a positive integer and encoded string will not be empty or have extra space.*

*If an encoding process does not make the string shorter, do not encode it. If there are several solutions, return any of them.*

def encode(self, s):

    L = len(s)

    dp = [['']\*L for \_ in range(L)]

    for w in range(L):  ***# the out layer is w, which is not used in inner loop. w stands for the length of cur substring***

        for i in range(L-w):

            j = i + w

            sub = s[i:j+1]  ***# sub string***

            dp[i][j] = sub

            if j-i < 4:

                continue

            else:

                for k in range(i, j):  ***# pre process***

                    if len(dp[i][j]) > len(dp[i][k]) + len(dp[k+1][j]):

                        dp[i][j] = dp[i][k] + dp[k+1][j]

                for k in range(len(sub)):

                    r\_sb = sub[:k+1]

                    if r\_sb and len(sub) % len(r\_sb) == 0 and len(sub.replace(r\_sb, '')) == 0:

                        ss = str(len(sub) // len(r\_sb)) + '[' + dp[i][i+k] + ']'  ***# note: here is dp[i][i+k] not r\_sb,*** *↓*

                        if len(dp[i][j]) > len(ss):                      ***#  r\_sb can also be encoded, so here is dp[i][i+k]***

                            dp[i][j] = ss

    return dp[0][-1]

**394. Decode String**

*Given an encoded string, return it's decoded string.*

def decodeString(self, s):

    stack = []

    cur\_s, cur\_n = '', 0

    i, L = 0, len(s)

    while i < L:

        if s[i] == '[':

            stack.append((cur\_s, cur\_n))

            cur\_s, cur\_n = '', 0

        elif s[i] == ']':

            pre, num = stack.pop()

            cur\_s = pre + cur\_s \* num

        elif s[i].isdigit():

            start = i

            while i < L and s[i].isdigit():

                i += 1

            cur\_n = int(s[start:i])

            i -= 1

        else:

            cur\_s += s[i]

        i += 1

    return cur\_s

def decodeString1(self, s):

    if '[' not in s: return s

    i = 0

    while i < len(s):

        if s[i].isdigit():

            left, start = s[:i], i

            while i < len(s) and s[i].isdigit():

                i += 1

            num = int(s[start:i])

            count, start = 0, i

            while i < len(s):

                if s[i] == '[':

                    count += 1

                if s[i] == ']':

                    count -= 1

                if count == 0:

                    break

                i += 1  ***# error: forgot i+1 here***

            mid, right = s[start+1:i], s[i+1:]

            return left + self.decodeString(mid) \* num + self.decodeString(right)  ***# error: forgot self. here***

        i += 1  ***# error: forgot i+1 here***

**474. Ones and Zeroes**

*For now, suppose you are a dominator of m 0s and n 1s respectively. On the other hand, there is an array with strings consisting of only 0s and 1s. Now your task is to find the maximum number of strings that you can form with given m 0s and n 1s. Each 0 and 1 can be used at most once.*

def findMaxFrom(self, strs, m, n):  ***# TEL for python, but it's the solution for other language***

    dp = [[0]\*(n+1) for \_ in range(m+1)]

    for s in strs:

        zs, os = s.count('0'), s.count('1')

        for i in range(m, zs-1, -1):

            for j in range(n, os-1, -1):

                dp[i][j] = max(dp[i][j], dp[i-zs][j-os] + 1)

    return dp[-1][-1]

**475. Heaters**

*Your are designing a standard heater with fixed warm radius to warm all the houses. Now, you are given positions of houses and heaters on a horizontal line, find out minimum radius of heaters so that all houses could be covered by those heaters.*  
def findRadius(self, houses, heaters):

    heaters.sort()

    ans = 0  ***# here is 0 we want it as samll as possible, and enlarge it with houses coming***

    for h in houses:

        idx = bisect.bisect\_left(heaters, h)

        dis1 = (h - heaters[idx-1]) if idx > 0 else float('inf')

        dis2 = (heaters[idx] - h) if 0 <= idx < len(heaters) else float('inf')

        ans = max(ans, min(dis1, dis2))

    return ans

**5. Longest Palindromic Substring**

*Given a string s, find the longest palindromic substring in s. You may assume that the maximum length of s is 1000.*

def longestPalindrome(self, s):

    # approach I, we can reveres s to rs, then find longest common substr in s and rs (use dp)

    # approach II, from center to expand

    self.start = 0

    self.end = 0

    def expand(i, j):

        while i >=0 and j < len(s) and s[i] == s[j]:

            i -= 1

            j += 1

        i += 1

        j -= 1

        if j - i > self.end - self.start:

            self.start, self.end = i, j

    for i in range(len(s)):

        expand(i, i)  # odd

        expand(i, i+1)  # even

    return s[self.start:self.end+1]

**411. Minimum Unique Word Abbreviation**

*A string such as "word" contains the following abbreviations:["word", "1ord", "w1rd", "2rd", "w2d", "wo2", "1o1d", "1or1"...]*

*Given a target string and a set of strings in a dictionary, find an abbreviation of this target string with the smallest possible length such that it does not conflict with abbreviations of the strings in the dictionary. Each number or letter in the abbreviation is considered length = 1. For example, the abbreviation "a32bc" has length = 4.*

def minAbbreviation(self, target, dictionary):

    def toNumber(w):

        num = 0

        for i in range(len(target)):

            num <<= 1

            num += w[i] == target[i]

        return num

    wlist = [toNumber(w) for w in dictionary if len(w) == len(target)]

    self.ans = (1 << len(target)) - 1

    self.length = len(target)

    def dfs(num, depth, length):

        if length >= self.length: return

        if depth == len(target):

            if not any(num & w == num for w in wlist):

                self.ans = num

                self.length = length

            return

        dfs((num << 1) + 1, depth + 1, length + 1)

        if length == 0 or num & 1:

            for i in range(2, len(target) - depth + 1):

                dfs(num << i, depth + i, length + 1)

    dfs(0, 0, 0)

    def toWord(num):

        w = ''

        count = 0

        for i in range(len(target)):

            if num & (1 << len(target) - i - 1):

                if count:

                    w += str(count)

                    count = 0

                w += target[i]

            else:

                count += 1

        return w + (str(count) if count else '')  ***# note, we should use () to wrap if else***

    return toWord(self.ans)

**begin 机经 before July**

**340. Longest Substring with At Most K Distinct Characters**

*Given a string, find the length of the longest substring T that contains at most k distinct characters.*

*For example, Given s = “eceba” and k = 2, T is "ece" which its length is 3.*

def lengthOfLongestSubstringKDistinct(self, s, k):

    ans, start, dic = 0, 0, {}

    for i, c in enumerate(s):

        dic[c] = i

        if len(dic) > k:

            pop = min(dic.values())

            dic.pop(s[pop])

            start = pop + 1

        ans = max(ans, i - start + 1)

    return ans

**3. Longest Substring Without Repeating Characters**

*Given a string, find the length of the longest substring without repeating characters.*

def lengthOfLongestSubstring(self, s):

    ans, start, dic = 0, 0, {}

    for i, c in enumerate(s):

        if c not in dic:

            dic[c] = i

        else:

            start = max(start, dic[c] + 1)***# error: forgot max! eg. abba, the last a, we should start from 2, not dic[a]+1***

            dic[c] = i

        ans = max(ans, i - start + 1)

    return ans

**395. Longest Substring with At Least K Repeating Characters**

*Find the length of the longest substring T of a given string (consists of lowercase letters only) such that every character in T appears no less than k times.*

def longestSubstring(self, s, k):

    if len(s) < k: return 0

    count = collections.Counter(s)

    char, least = count.most\_common()[-1]

    if least >= k:

        return len(s)

    return max(self.longestSubstring(t, k) for t in s.split(char))  ***# the return is int, can use max directly***

**734. Sentence Similarity**

*Given two sentences words1, words2 (each represented as an array of strings), and a list of similar word pairs pairs, determine if two sentences are similar. For example, "great acting skills" and "fine drama talent" are similar, if the similar word pairs are pairs = [["great", "fine"], ["acting","drama"], ["skills","talent"]].*

def areSentencesSimilar(self, words1, words2, pairs):

    if len(words1) != len(words2): return False

    pairs = list(map(set, pairs)) ***# if we want to use set(xxx), the elements in xxx needs hashable***

***# or pairs = set(map(tuple, pairs)), since tuple is hashable, but we should use (w1, w2) and (w2, w1)***

    for w1, w2 in zip(words1, words2):

        if w1 == w2: continue

        key = {w1, w2}

        if key not in pairs:

            return False

    return True

**737. Sentence Similarity II**

*Similar with Sentence Similarity I, but the similarity relation is transitive. For example, if "great" and "good" are similar, and "fine" and "good" are similar, then "great" and "fine" are similar.*

def areSentencesSimilarTwo(self, words1, words2, pairs): 

    if len(words1) != len(words2):

        return False

    base = {}

    def find(w):

        if w not in base:

            base[w] = w

        if base[w] != w:

            base[w] = find(base[w])

        return base[w]

    def union(w1, w2):

        r1, r2 = find(w1), find(w2)

        if r1 != r2:

            base[r1] = base[r2]

    for w1, w2 in pairs:

        union(w1, w2)

    for w1, w2 in zip(words1, words2):

        if w1 != w2 and find(w1) != find(w2):

            return False

    return True

**120. Triangle**

*Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below. [1], [2,3], [4,5,6], for 1 you can move 2, 3, for 3, you can move to 5,6*

def minimumTotal(self, triangle):

    L, dp = len(triangle), triangle[-1]

    for i in range(L-2, -1, -1):

        for j in range(len(triangle[i])):  ***# since the j is from left to right, we change dp[j] will not influence next step***

            dp[j] = min(dp[j], dp[j+1]) + triangle[i][j]

    return dp[0]

**854. K-Similar Strings**

*Strings A and B are K-similar (for some non-negative integer K) if we can swap the positions of two letters in A exactly K times so that the resulting string equals B.*

def kSimilarity(self, A, B):

    def neighbors(S):

        i = 0

        while i< len(S) and S[i] == B[i]: ***# find the next different one***

            i += 1

        T = list(S)

        for j in range(i+1, len(S)):

            if S[j] == B[i]:  ***# find a same one, then swap with the previous different one***

                T[i], T[j] = T[j], T[i]

                yield "".join(T)  ***# return one by one***

                T[j], T[i] = T[i], T[j]  ***# when get one, continue run this line***

    queue = collections.deque([A])

    seen = {A: 0}

    while queue:

        S = queue.popleft()

        if S == B: return seen[S]

        for T in neighbors(S):

            if T not in seen:

                seen[T] = seen[S] + 1

                queue.append(T)

**49. Group Anagrams**

*Given an array of strings, group anagrams together.*

*Input: ["eat", "tea", "tan", "ate", "nat", "bat"],  Output: [ ["ate","eat","tea"], ["nat","tan"], ["bat"]]*

def groupAnagrams(self, strs):

    ans = collections.defaultdict(list)

    for s in strs:

        key = tuple(sorted(s))***# can't use key = set(s), because set is unhashable***

*# since sort is klogk, so we can also use a count as a key*

*# key = [0] \* 26*

*# for c in s:*

*#     key[ord(c)-ord(a)] += 1*

*# key = tuple(key)*

        ans[key].append(s)

    return list(ans.values())

**486. Predict the Winner**

*Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. This continues until all the scores have been chosen. The player with the maximum score wins. Given an array of scores, predict whether player 1 is the winner. Every player is smart enough.*

def PredictWinner(self, nums):***# T:O(n^2), S: O(n)***

    L = len(nums)

    dp = [0] \* L***# we can use 2-d dp, but here 1-d to reduce space, dp[i] is from 0 - i, the maximum store a player can get.***

    for start in range(L-2, -1, -1):

        for end in range(start+1, L):

            ps = nums[start] - dp[end]

            pe = nums[end] - dp[end-1]

            dp[end] = max(ps, pe)

    return dp[-1] >= 0

def PredictWinner(self, nums):***# T: O(n^2), S: O(n^2)***

    memo = {}

    def pick(start, end):

        if start == end:

            return nums[start]

        if (start, end) in memo:

            return memo[start, end]

        ps = nums[start] - pick(start+1, end)

        pe = nums[end] - pick(start, end-1)

        memo[start, end] = max(ps, pe)

        return memo[start, end]

    return pick(0, len(nums)-1) >= 0  # can be 0

**862. Shortest Subarray with Sum at Least K**

*Return the length of the shortest, non-empty, contiguous subarray of A with sum at least K. If there is no non-empty subarray with sum at least K, return -1.*

def shortestSubarray(self, A, K):

    asum = [0] + list(itertools.accumulate(A))  *# we should add [0], becasue if we got exact K in i,*

    cds = collections.deque()                          *# we should return i, this i is sum-sum[0], only sum[0] is 0*

    ans = float('inf')

    for i, num in enumerate(asum):

*# if we don't add [0] above, we should add a situation here*

*if num >= K:*

*ans = min(ans, i + 1)*

        while cds and num - asum[cds[0]] >= K:

            ans = min(ans, i - cds.popleft())  ***# don't need to +1 here, because asum contains popleft***

        while cds and num <= asum[cds[-1]]:

            cds.pop()

        cds.append(i)

    return ans if ans < float('inf') else -1

**325. Maximum Size Subarray Sum Equals k**

*Given an array nums and a target value k, find the maximum length of a subarray that sums to k. If there isn't one, return 0.*

def maxSubArrayLen(self, nums, k):

    asum = 0

    dic = {0:-1}  ***# error: forgot the base case***

    ans = 0

    for i, num in enumerate(nums):

        asum += num

        if asum not in dic:

            dic[asum] = i

        if asum - k in dic:

            ans = max(ans, i - dic[asum-k])

    return ans

**797. All Paths From Source to Target**

*Given a directed, acyclic graph of N nodes.  Find all possible paths from node 0 to node N-1.*

def allPathsSourceTarget(self, graph):

    self.N = len(graph)

    ans = []

    def search(node, path):

        if node == self.N - 1:

            ans.append(path)

        for nxt in graph[node]:

            if nxt not in path:

                search(nxt, path+[nxt])

    search(0, [0])  ***# note, should put [0] into path***

    return ans

**79. Word Search**

*Given a 2D board and a word, find if the word exists in the grid.The word can be constructed from letters of sequentially adjacent cell. The same letter cell may not be used more than once.*

def exist(self, board, word):

    if not board or not board[0]:

        return False

    rows, cols = len(board), len(board[0])

    dirs = ((0, 1), (0, -1), (-1, 0), (1, 0))

    visited = set()

    def search(x, y, n=0):

        if n >= len(word):

            return True

        c = word[n]

        for dx, dy in dirs:

            nx, ny = x + dx, y + dy

            if 0 <= nx < rows and 0 <= ny < cols and (nx, ny) not in visited:

                if c == board[nx][ny]:

                    visited.add((nx, ny))

                    if search(nx, ny, n + 1):

                        return True

                    visited.remove((nx, ny))

        return False

    for i in range(rows):

        for j in range(cols):

            if board[i][j] == word[0]:

                visited.add((i, j))

                if search(i, j, 1):

                    return True

                visited.remove((i, j))

    return False

**212. Word Search II**

*Given a 2D board and a list of words from the dictionary, find all words in the board.*

def findWords(self, board, words):

    Trie = lambda: collections.defaultdict(Trie)  ***# use a tire to reduce time complexity***

    root = Trie()

    for word in words:

        cur = root

        for c in word:

            cur = cur[c]

        cur['#'] = word

    rows, cols, dirs = len(board), len(board[0]), ((0, 1), (0, -1), (-1, 0), (1, 0))

    ans, words, visited = [], set(words), set()

    def search(x, y, cur):

        if '#' in cur:

            ans.append(cur.pop('#'))  ***# can use pop to avoid duplicate***

        for dx, dy in dirs:

            nx, ny = x + dx, y + dy

            if 0 <= nx < rows and 0 <= ny < cols and (nx, ny) not in visited and board[nx][ny] in cur:

                visited.add((nx, ny))

                search(nx, ny, cur[board[nx][ny]])

                visited.remove((nx, ny))

    for i in range(rows):

        for j, c in enumerate(board[i]):

            if c in root:

                visited.add((i, j)) ***# don't forgot visited***

                search(i, j, root[c])

                visited.remove((i, j))

    return ans

**215. Kth Largest Element in an Array**

*Find the kth largest element in an unsorted array.*

def findKthLargest(self, nums, k):

    def qsort(l, r, n):

        j, num = l, nums[r]  ***# we have a j = l here, so it's all qsort(x, x, n), don't need to modify n***

        for i in range(l, r):

            if nums[i] < num:  ***# we can find k directly, only needs change here to nums[i] > num***

                nums[i], nums[j] = nums[j], nums[i]

                j += 1

        nums[j], nums[r] = nums[r], nums[j]

        if j + 1 == n: ***# error: here is j + 1***

            return nums[j]

        elif j + 1 > n:  ***# error: here is also j + 1***

            return qsort(l, j-1, n)

        else:

            return qsort(j+1, r, n)

*# return qsort(0, len(nums)-1, k)* ***# we can find k directly, only we should change < to > in qsort function***

    return qsort(0, len(nums)-1, len(nums)-k+1)  ***# if we change largest to smallest, don't forget + 1***

**207. Course Schedule**

*There are a total of n courses you have to take, labeled from 0 to n-1. Some courses may have prerequisites, which is expressed as a pair: [0,1] Given the total number of courses and a list of prerequisite, is it possible for you to finish all courses?*

def canFinish(self, numCourses, prerequisites):

    parents, children = collections.defaultdict(list), collections.defaultdict(list)

    classes = [i for i in range(numCourses)]

    for p, c in prerequisites:

        parents[c].append(p)

        children[p].append(c)

    queue = [c for c in classes if c not in parents]

    num = len(queue)

    for c in queue:

        for child in children[c]:

            parents[child].remove(c)

            if len(parents[child]) == 0:

                queue.append(child)

                num += 1

    return num == numCourses

**210. Course Schedule II**

*Similar with Course Schedule I, return the ordering of courses you should take to finish all courses. If not return a empty list.*

def findOrder(self, numCourses, prerequisites):

    parents = collections.defaultdict(list)

    children = collections.defaultdict(list)

    classes = [i for i in range(numCourses)]

    for c, p in prerequisites:

        parents[c].append(p)

        children[p].append(c)

    queue = [c for c in classes if c not in parents]

    for c in queue:

        for child in children[c]:

            parents[child].remove(c)

            if len(parents[child]) == 0: queue.append(child)

    return queue if len(queue) == numCourses else []   ***# remember to check***

**630. Course Schedule III**

*There are n different online courses numbered from 1 to n. Each course has some duration(course length) t and closed on dth day. A course should be taken continuously for t days and must be finished before or on the dth day. You will start at the 1st day. Given n online courses represented by pairs (t,d), your task is to find the maximal number of courses that can be taken.*

def scheduleCourse(self, courses):

    courses.sort(key=lambda item:item[-1])  ***# we choose the class with earlier end time***

    start, take = 0, PriorityQueue()

    for dur, end in courses:

        if start + dur <= end:

            start += dur

            take.put(-dur)

        else:  ***# if we can't take this class, and this class has shorter time than have taken, then take this one. Since we already taken the longer one, we can definitely take this one***

            if not take.empty() and -take.queue[0] > dur:  ***# can't use if take, pq can't be bool directly***

                start += take.get() + dur

                take.put(-dur)

    return take.qsize()

**334. Increasing Triplet Subsequence**

*Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.*

def increasingTriplet(self, nums):

***# we can only use two variable instead of the q in right***

    first = second = float('inf')

    for num in nums:

        if num <= first:

            first = num

        elif num <= second:

            second = num

        else:

            return True

    return False

def increasingTriplet1(self, nums):

    q = []

    for i, num in enumerate(nums):

        if len(q) >= 3:

            return True

        if not q:

            q.append(num)

        elif num > q[-1]:

            q.append(num)

        else:

            for i in range(len(q)):

                if q[i] >= num:

                    q[i] = num

                    break

    return len(q) >= 3

**582. Kill Process**

*Given n processes, each process has a unique PID (process id) and its PPID (parent process id). Now given the two lists, and a PID representing a process you want to kill, return a list of PIDs of processes that will be killed in the end.*

def killProcess(self, pid, ppid, kill):

    adj = collections.defaultdict(list)

    for p, pp in zip(pid, ppid):

        adj[pp].append(p)

    q, ans = [kill], set()

    while q:

        p = q.pop()

        ans.add(p)

        for nei in adj[p]:

            if nei not in ans:***# since we use 'in', if we use list to ans, it will TLE***

                q.append(nei)

    return list(ans)

**254. Factor Combinations**

*Numbers can be regarded as product of its factors. For example, 8 = 2 x 2 x 2 or 2 x 4. Write a function that takes an integer n and return all possible combinations of its factors.*

def getFactors(self, n):

    ans = []

    def decompose(k=n, f=2, path=[]):

        while f \* f <= k:

            if k % f == 0:

                ans.append(path + [f, k//f])

                decompose(k//f, f, path+[f])

            f += 1

    decompose()

    return ans

**43. Multiply Strings**

*Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2, also represented as a string.*

def multiply(self, num1, num2):

    m, n = len(num1), len(num2)

    res = [0] \* (m + n)

    for i in range(m-1, -1, -1):

        for j in range(n-1, -1, -1):

            c1, c2 = num1[i], num2[j]

            mut = int(c1) \* int(c2)

            i\_carry, i\_cur = i + j, i + j  + 1 ***# the right is bigger index, the carry in left, so idx of carry < inx of cur***

            mut += res[i\_cur]

            res[i\_carry] += mut // 10  ***# carry should +=***

            res[i\_cur] = mut % 10  ***# cur only =, because we add to mut at the beginning***

    return ''.join(map(str, res)).lstrip('0') or '0'

**415. Add Strings**

*Given two non-negative integers num1 and num2 represented as string, return the sum of num1 and num2.*

def addStrings(self, num1, num2):

    carry, i, j = 0, len(num1)-1, len(num2)-1***# don't forget -1***

    res = []

    while carry or i >= 0 or j >= 0:  ***# here is >= 0***

        temp = carry

        if j >= 0:  ***# also >= 0***

            temp += int(num2[j])

            j -= 1

        if i >= 0:

            temp += int(num1[i])

            i -= 1

        carry, cur = divmod(temp, 10)

        res.insert(0, str(cur))

    return ''.join(res) or '0' ***# don't forget or***

**450. Delete Node in a BST**

*Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST. Note: Time complexity should be O(height of tree).*

def deleteNode(self, root, key):

    if not root:

        return None

    if key < root.val:

        root.left = self.deleteNode(root.left, key)

    elif key > root.val:

        root.right = self.deleteNode(root.right, key)

    else:

        if not root.right: return root.left  ***# if only have one brunch, just use it***

        if not root.left: return root.right

        cur = root.left  ***# if have left and right, we can't only use one and discard another one, so we should choose***

        while cur.right:  ***# the biggest from the left OR the smallest on the right, both can do.***

            cur = cur.right

        root.val = cur.val

        root.left = self.deleteNode(root.left, cur.val)  ***# since we change root to cur, we should also delete cur***

    return root

**867. Transpose Matrix**

Given a matrix A, return the transpose of A.

def transpose(self, A):

    # return zip(\*A)  ***# leetcode hide zip function***

    if not A or not A[0]: return []

    rows, cols = len(A), len(A[0])

    res = [[None] \* rows for \_ in range(cols)]

    for i in range(rows):

        for j in range(cols):

            res[j][i] = A[i][j]

    return res

def transpose(self, A):

***# if rows == cols, we can transpose inplace***

     for i in range(rows-1):

         for j in range(i+1, rows):

             A[i][j], A[j][i] = A[j][i], A[i][j]

     return A

**399. Evaluate Division**

*Equations are given in the format A / B = k, where A and B are variables represented as strings, and k is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return -1.0.*

def calcEquation(self, equations, values, queries):

    graph = collections.defaultdict(set)

    for (x, y), t in zip(equations, values):

        graph[x].add((t, y))

        graph[y].add((1/t, x))

    res = []

    for x, y in queries:

        if x not in graph or y not in graph:

            res.append(-1.0)

            continue

        visited = set()

        stack = [(x, 1.0)]***# here, we should also put the result into the stack***

        visited.add(x)

        while stack:

            cur, temp = stack.pop()

            if cur == y:

                res.append(temp)

                break

            for t, z in graph[cur]:

                if z not in visited:

                    stack.append((z, temp \* t))  ***# error: don't use temp \*= t here, since it may cause duplicate multiples***

                    visited.add(z)

        if y not in visited:  ***# the break can only jump out of while, it will still run the code below it, so we should check***

            res.append(-1.0)

    return res

**836. Rectangle Overlap**

*A rectangle is represented as a list [x1, y1, x2, y2], where (x1, y1) are the coordinates of its bottom-left corner, and (x2, y2) are the coordinates of its top-right corner.Given two (axis-aligned) rectangles, return whether they overlap.*

def isRectangleOverlap(self, rec1, rec2):

    (x1, y1, x2, y2), (a1, b1, a2, b2) = rec1, rec2

    return not (x2 <= a1 or x1 >= a2 or y1 >= b2 or y2 <= b1)

**recent problems on LeetCode**

**859. Buddy Strings**

*Given two strings A and B of lowercase letters, return true if and only if we can swap two letters in A so that the result equals B.*

def buddyStrings(self, A, B):

    if len(A) != len(B): return False

    if len(A) < 2: return False

    if A == B:

        count = collections.Counter(A).most\_common()

        return count[0][-1] >= 2   ***# if A==B, must have two same char for exchanging***

    diff = []

    for i in range(len(A)):

        if A[i] != B[i]:

            diff.append([A[i], B[i]])

            if len(diff) > 2:  ***# put check here can guarantee the space***

                return False

    return len(diff) == 2 and diff[0] == diff[-1][::-1] ***#do not use reversed, since reversed return a iterator not a list***

**857. Minimum Cost to Hire K Workers**

*There are N workers.  The i-th worker has a quality[i] and a minimum wage expectation wage[i]. Now we want to hire exactly K workers to form a paid group.  When hiring a group of K workers, we must pay them according to the following rules:*

* *Every worker in the paid group should be paid in the ratio of their quality compared to other workers in the paid group.*
* *Every worker in the paid group must be paid at least their minimum wage expectation.*

*Return the least amount of money needed to form a paid group satisfying the above conditions.*

def mincostToHireWorkers(self, quality, wage, K):

    rates = []

    for i in range(len(wage)):

        rates.append((wage[i]/quality[i], quality[i]))

    rates.sort()  ***# here is from small to large, so we can add larger rate one by one to the pq, and compute***

    ans = float('inf')                                                                 ***# the max\_rate, or use the rate directly***

    pq, sum\_q, max\_rate = PriorityQueue(), 0, 0

    for rate, qua in rates:

        pq.put(-qua)

        sum\_q += qua

        # max\_rate = max(max\_rate, rate)***# since the rate has already sorted, we can use rate directly***

        if pq.qsize() > K:

            sum\_q += pq.get()

        if pq.qsize() == K:

            # ans = min(ans, sum\_q \* max\_rate)

            ans = min(ans, sum\_q \* rate)

    return ans

**835. Image Overlap**

*Two images A and B are given as square matrices of the same size. (only 0s and 1s as values.) We translate one image however we choose (sliding it left, right, up, or down any number of units), and place it on top of the other image.  After, the overlap of this translation is the number of positions that have a 1 in both images. What is the largest possible overlap?*

def largestOverlap(self, A, B):

    A = [(i, j) for i, row in enumerate(A) for j, item in enumerate(row) if item]

    B = [(i, j) for i, row in enumerate(B) for j, item in enumerate(row) if item]

    count = collections.Counter((ax-bx, ay-by) for ax, ay in A for bx, by in B)

    return max(count.values() or [0])  ***# if the input has no 1, count will be None***

**849. Maximize Distance to Closest Person**

*In a row of seats, 1 represents a person sitting in that seat, and 0 represents that the seat is empty. There is at least one empty seat, and at least one person sitting.Alex wants to sit in the seat such that the distance between him and the closest person to him is maximized. Return that maximum distance to closest person.*

def maxDistToClosest(self, seats):

    dis, pre, ans = 0, -1, 0

    for i, seat in enumerate(seats):

        if seat:

            if pre == -1:

                dis = i

            else:

                d = (i - pre) // 2

                if d > dis:

                    dis = d

            pre = i

    return max(dis, len(seats) - 1 - pre)

**855. Exam Room**

*In an exam room, there are N seats in a single row, numbered 0, 1, 2, ..., N-1. When a student enters the room, they must sit in the seat that maximizes the distance to the closest person.  If there are multiple such seats, they sit in the seat with the lowest number. (Also, if no one is in the room, then the student sits at seat number 0.) ExamRoom has two functions:*

* *ExamRoom.seat() returning an int representing what seat the student sat in*
* *ExamRoom.leave(int p) representing that the student in seat number p now leaves the room.*

class ExamRoom:

    def \_\_init\_\_(self, N):

        self.N = N

        self.seats = []

    def seat(self):

        if not self.seats:

            self.seats.append(0)

            return 0

***# here is a important trick to deal with 0th seat: put the new to 0 and let dis = seats[0]***

        ans, dis, seats = 0, self.seats[0], self.seats

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

            d = (seats[i] - seats[i-1]) // 2

            if d > dis:

                dis = d

                ans = seats[i-1] + d

        if self.N - 1 - seats[-1] > dis:

            ans = self.N - 1

        bisect.insort(seats, ans)

        return ans

    def leave(self, p):

        self.seats.remove(p)

**853. Car Fleet**

*N cars are going to the same destination along a one lane road.  The destination is target miles away. Each car i has a constant speed speed[i] (in miles per hour), and initial position position[i] miles towards the target along the road. A car can never pass another car ahead of it, but it can catch up to it, and drive bumper to bumper at the same speed. The distance between these two cars is ignored - they are assumed to have the same position. A car fleet is some non-empty set of cars driving at the same position and same speed.  Note that a single car is also a car fleet. If a car catches up to a car fleet right at the destination point, it will still be considered as one car fleet. How many car fleets will arrive at the destination?*

def carFleet(self, target, position, speed):

    times = [(target - p) / s for p, s in sorted(zip(position, speed))]

    ans = 0

    while len(times) > 1:

        head = times.pop()

        if times[-1] > head:

            ans += 1

        else:

            times[-1] = head

    return ans + bool(times)

**852. Peak Index in a Mountain Array**

*Let's call an array A a mountain if the following properties hold: A.length >= 3. There exists some 0 < i < A.length - 1 such that A[0] < A[1] < ... A[i-1] < A[i] > A[i+1] > ... > A[A.length - 1]. Given an array that is definitely a mountain, return any i is a peak.*

def peakIndexInMountainArray(self, A):

    # O(N)

    # for i in range(len(A)-1):  # this can also AC, since 0 < i < L-1

    #     if A[i] > A[i+1]:

    #         return i

    #O(logN)

    left, right = 0, len(A)-1

    while left < right:

        mid = (left + right) // 2

        if A[mid] > A[mid+1]:

            right = mid

        else:

            left = mid + 1

    return left

**845. Longest Mountain in Array**

*Let's call any (contiguous) subarray B (of A) a mountain if the following properties hold: B.length >= 3. There exists some 0 < i < B.length - 1 such that B[0] < B[1] < ... B[i-1] < B[i] > B[i+1] > ... > B[B.length - 1] (Note that B could be any subarray of A, including the entire array A.) Given an array A, return the length of the longest mountain. Return 0 if there is no mountain.*

def longestMountain(self, A):

    L = len(A)

    ans, up, down = 0, 0, 0

    for i in range(1, L):

        if A[i] == A[i-1] or (down and A[i] > A[i-1]):  ***# if i == 0, down,up are 0, it's fine to go on***

            up = down = 0

        up += A[i] > A[i-1]

        down += A[i] < A[i-1]

        if up and down:

            ans = max(ans, up + down + 1)

    return ans

**847. Shortest Path Visiting All Nodes**

*An undirected, connected graph of N nodes (labeled 0, 1, 2, ..., N-1) is given as graph. graph.length = N, and j != i is in the list graph[i] exactly once, if and only if nodes i and j are connected. Return the length of the shortest path that visits every node. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.*

def shortestPathLength(self, graph):

    N = len(graph)

    dq = collections.deque([(1<<i, i) for i in range(N)])

    dis = collections.defaultdict(lambda: float('inf'))

    for node in dq:

        dis[node] = 0

    while dq:

        cover, cur = dq.popleft()

        d = dis[cover, cur]

        if cover == 2\*\*N - 1:

            return d

        for neighbor in graph[cur]:

            cover2 = cover | (1 << neighbor)

            if d + 1 < dis[cover2, neighbor]:

                dis[cover2, neighbor] = d + 1

                dq.append((cover2, neighbor))

**844. Backspace String Compare**

*Given two strings S and T, return if they are equal when both are typed into empty text editors. # means a backspace character.*

def backspaceCompare(self, S, T):

***# O(M+N), O(M+N)***

      def getStr(B):

          s = []

          for c in B:

              if c == '#':

                  if s:

                      s.pop()

              else:

                  s.append(c)

          return ''.join(s)

      return getStr(S) == getStr(T)

def backspaceCompare(self, S, T):

***# O(M+N), O(1)***

    def getChar(B):

        skip =  0

        for c in reversed(B):

            if c == '#':

                skip += 1

            elif skip:

                skip -= 1

            else:

                yield c

***# zip will stop on the shortest one, itertools.zip\_longest stop on the longest one and fill up the shorter ones with None or set a default fillvalue argument.***

    return all(a == b for a, b in itertools.zip\_longest(getChar(S), getChar(T)))

**832. Flipping an Image**

*Given a binary matrix A, we want to flip the image horizontally, then invert it, and return. To flip an image horizontally means that each row of the image is reversed.  For example, flipping [1, 1, 0] horizontally results in [0, 1, 1]. To invert an image means that each 0 is replaced by 1, and each 1 is replaced by 0. For example, inverting [0, 1, 1] results in [1, 0, 0].*

def flipAndInvertImage(self, A):

    rows, cols = len(A), len(A[0])

    for row in range(rows):

        for col in range((cols + 1) // 2): ***# error: here needs cols + 1***

            A[row][col], A[row][~col] = A[row][~col] ^ 1, A[row][col] ^ 1  ***# remeber ~num***

    return A

**843. Guess the Word**

*We are given a word list of unique words, each word is 6 letters long, and one word in this list is chosen as secret. You may call master.guess(word) to guess a word.  The guessed word should have type string and must be from the original list with 6 lowercase letters. This function returns an integer type, representing the number of exact matches (value and position) of your guess to the secret word.  Also, if your guess is not in the given wordlist, it will return -1 instead. For each test case, you have 10 guesses to guess the word. At the end of any number of calls, if you have made 10 or less calls to master.guess and at least one of these guesses was the secret, you pass the testcase. Besides the example test case below, there will be 5 additional test cases, each with 100 words in the word list.  The letters of each word in those testcases were chosen independently at random from 'a' to 'z', such that every word in the given word lists is unique.*

def findSecretWord(self, wordlist, master):

    N = len(wordlist)

    sames = [[sum(a == b for a, b in zip(wordlist[i], wordlist[j])) for j in range(N)] for i in range(N)]

    possible = [i for i in range(N)]

    guessed = set()

    def findMinPoss():

        nxt, max\_possible = 0, len(possible)

        for i in range(N):

            if i not in guessed:

                temp\_same = [0] \* 7

                for j in possible:

                    if i != j:

                        temp\_same[sames[i][j]] += 1

                max\_p = max(temp\_same)

                if max\_p < max\_possible:

                    nxt, max\_possible = i, max\_p

        return nxt

    while possible:

        guess = findMinPoss()

        guessed.add(guess)

        match = master.guess(wordlist[guess])

        if match == len(wordlist[0]): return

        possible = [i for i in possible if sames[guess][i] == match]

**841. Keys and Rooms**

*There are N rooms and you start in room 0.  Each room has a distinct number in 0, 1, 2, ..., N-1, and each room may have some keys to access the next room.  Formally, each room i has a list of keys rooms[i], and each key rooms[i][j] is an integer in [0, 1, ..., N-1] where N = rooms.length.  A key rooms[i][j] = v opens the room with number v. Initially, all the rooms start locked (except for room 0). You can walk back and forth between rooms freely. Return true if and only if you can enter every room.*

def canVisitAllRooms(self, rooms):

    opened = [False] \* len(rooms)

    queue = [0]

    opened[0] = True

    while queue:

        room = queue.pop()

        for nxt in rooms[room]:

            if not opened[nxt]:

                opened[nxt] = True

                queue.append(nxt)

    return all(opened)

**840. Magic Squares In Grid**

*A 3 x 3 magic square is a 3 x 3 grid filled with distinct numbers from 1 to 9 such that each row, column, and both diagonals all have the same sum. Given an grid of integers, how many 3 x 3 "magic square" subgrids are there?*

def numMagicSquaresInside(self, grid):

    if not grid or not grid[0]: return 0

    rows, cols = len(grid), len(grid[0])

    def check(start\_row, start\_col):

        sums = set()

        col\_s = [0] \* 3

        dias = [0] \* 2

        for i in range(3):

            row = start\_row + i

            s = 0

            for j in range(3):

                col = start\_col + j

                if grid[row][col] > 9 or grid[row][col] < 1:

                    return False

                s += grid[row][col]

                col\_s[j] += grid[row][col]

                if i == j:

                    dias[0] += grid[row][col]

                if i == 2-j:

                    dias[1] += grid[row][col]

            sums.add(s)

        sums.update(col\_s)

        sums.update(dias)

        return len(sums) == 1

    ans = 0

    for i in range(0, rows-3+1):

        for j in range(0, cols-3+1):

            if check(i, j):

                ans += 1

    return ans

**837. New 21 Game**

*Alice plays the following game, loosely based on the card game "21". Alice starts with 0 points, and draws numbers while she has less than K points.  During each draw, she gains an integer number of points randomly from the range [1, W], where W is an integer. Each draw is independent and the outcomes have equal probabilities. Alice stops drawing numbers when she gets K or more points.  What is the probability that she has N or less points?*

def new21Game(self, N, K, W):

    dp = [0] \* (N + W + 1)

    for i in range(K, N+1):

        dp[i] = 1

    S = min(N-K+1, W) ***# S = (dp[K+1] + dp[K+2] + ... + dp[K+W]) / W, and dp[K~N] is 1, so S is min(N-k+1, W)***

    for i in range(K-1, -1, -1):

        dp[i] = S / W

        S = S + dp[i] - dp[i+W]

    return dp[0]

**839. Similar String Groups**

*Two strings X and Y are similar if we can swap two letters (in different positions) of X, so that it equals Y. For example, "tars" and "rats" are similar and "rats" and "arts" are similar, but "star" is not similar to "tars", "rats", or "arts". Together, these form two connected groups by similarity: {"tars", "rats", "arts"} and {"star"}.  Notice "tars" and "arts" are in the same group even though they are not similar. Formally, each group is such that a word is in the group if it is similar to at least one other word in the group. We are given a list A of strings. Every string in A is an anagram of every other string in A. How many groups are there?*

def numSimilarGroups(self, A):

    SA = set(A)  # to delete duplicate items

    A = list(SA) # make it indexable

    L = len(A)

    base = [i for i in range(L)]

    def find(x):

        if base[x] != x:

            base[x] = find(base[x])

        return base[x]

    def union(x, y):

        base[find(x)] = find(y)

    def check(x, y):

        return sum(a!=b for a, b in zip(A[x], A[y])) == 2

    if L < len(A[0])\*\*2:  ***# deal with long word with few items***

        for i1, i2 in itertools.combinations(range(L), 2):

            if check(i1, i2):

                union(i1, i2)

    else: ***# deal with short word with lots of items***

        buckets = collections.defaultdict(set)

        for i, word in enumerate(A):

            for i1, i2 in itertools.combinations(range(len(word)), 2):

                if word[i1] != word[i2]:  ***# filter, otherwise it will exceed memory limit***

                    wl = list(word)

                    wl[i1], wl[i2] = wl[i2], wl[i1]

                    key = "".join(wl)

                    if key in SA:

                        buckets[key].add(i)

        for i, word in enumerate(A):

            for j in buckets[word]:

                union(i, j)

    return sum(base[x] == x for x in range(L))

**812. Largest Triangle Area**

*You have a list of points in the plane. Return the area of the largest triangle that can be formed by any 3 of the points.*

def largestTriangleArea(self, points):

    ans = 0

    def area(a, b, c):

        (ax, ay), (bx, by), (cx, cy) = a, b, c

        return abs((ax\*by + bx\*cy + cx\*ay) - (ay\*bx + by\*cx + cy\*ax)) / 2  ***# error: forgot abs***

    return max(area(a, b, c) for a, b, c in itertools.combinations(points, 3))

**838. Push Dominoes**

*There are N dominoes in a line, and we place each domino vertically upright. In the beginning, we simultaneously push some of the dominoes either to the left or to the right. After each second, each domino that is falling to the left pushes the adjacent domino on the left. Similarly, the dominoes falling to the right push their adjacent dominoes standing on the right. When a vertical domino has dominoes falling on it from both sides, it stays still due to the balance of the forces. For the purposes of this question, we will consider that a falling domino expends no additional force to a falling or already fallen domino. Given a string "S" representing the initial state. S[i] = 'L', if the i-th domino has been pushed to the left; S[i] = 'R', if the i-th domino has been pushed to the right; S[i] = '.', if the i-th domino has not been pushed. Return a string representing the final state.*

def pushDominoes(self, dominoes):

    L = len(dominoes)

    ans = [0] \* L

    f = 0

    for i, d in enumerate(dominoes):

        if d == 'R':

            f = L

        elif d == 'L':

            f = 0

        else:

            f = max(f-1, 0)

        ans[i] = f

    f = 0

    for i in range(L-1, -1, -1):

        d = dominoes[i]

        if d == 'L':

            f = L

        elif d == 'R':

            f = 0

        else:

            f = max(f-1, 0)

        ans[i] -= f

    return ''.join(['.' if f == 0 else 'R' if f > 0 else 'L' for f in ans])

**830. Positions of Large Groups**

*In a string S of lowercase letters, these letters form consecutive groups of the same character. For example, a string like S = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z" and "yy". Call a group large if it has 3 or more characters.  We would like the starting and ending positions of every large group. The final answer should be in lexicographic order.*

def largeGroupPositions(self, S):

    ans, idx = [], 0

    while idx < len(S):

        c = S[idx]

        start = idx

        while idx < len(S) and c == S[idx]:

            idx += 1

        if idx - start >= 3:

            ans.append([start, idx-1])

    return ans

**834. Sum of Distances in Tree**

*An undirected, connected tree with N nodes labelled 0...N-1 and N-1 edges are given. The ith edge connects nodes edges[i][0] and edges[i][1] together. Return a list ans, where ans[i] is the sum of the distances between node i and all other nodes.*

def sumOfDistancesInTree(self, N, edges):

    graph = [[] for \_ in range(N)]

    for a, b in edges:

        graph[a].append(b)

        graph[b].append(a)

    children = [1] \* N

    sum\_distance = [0] \* N

    def compute(node=0, parent=None):

        for neighbor in graph[node]:

            if neighbor != parent:

                compute(neighbor, node)

                children[node] += children[neighbor]

***# +children[neighbor] means how many children it has, so how many times this edge (node->neighbor) has been passed. +sum\_distance[neighbor] is just add the sub\_distance***

                sum\_distance[node] += children[neighbor] + sum\_distance[neighbor]

    def update(node=0, parent=None):

        for neighbor in graph[node]:

            if neighbor != parent:

***# for the neighbor, sum\_distance[node] means its parent sum\_distance, and if this parent moved to children, all the children's (children[neighbor]) path will reduce one, at the meantime, all the path to other nodes (N - children[neighbor]) will increase one.***

                sum\_distance[neighbor] = sum\_distance[node] - children[neighbor] + N - children[neighbor]

                update(neighbor, node)

    compute()

    update()

    return sum\_distance

**817. Linked List Components**

*We are given head, the head node of a linked list containing unique integer values. We are also given the list G, a subset of the values in the linked list. Return the number of connected components in G, where two values are connected if they appear consecutively in the linked list.*

def numComponents(self, head, G):

    G = set(G)

    ans = 0

    while head:

        if head.val in G:

            ans += 1

            while head and head.val in G:

                head = head.next

        else:

            head = head.next

    return ans

**833. Find And Replace in String**

*To some string S, we will replace some groups of letters with new ones (not necessarily the same size). Each replacement operation has 3 parameters: a starting index i, a source word x and a target word y.  The rule is that if x starts at position i in the original string S, then we will replace that x with y. If not, we do nothing. All these operations occur simultaneously.*

def findReplaceString(self, S, indexes, sources, targets):

*# another simple method*

*# S = list(S)*

*# for i, x, y in sorted(zip(indexes, sources, targets), reverse = True):*

*#     if all(i+k < len(S) and S[i+k] == x[k] for k in range(len(x))):*

*#         S[i:i+len(x)] = list(y)*

*# return ''.join(S)*

    if not indexes:

        return S

    indexes += [len(S)]

    indexes = [(index, idx) for idx, index in enumerate(indexes)]

    indexes.sort()

    res = S[:indexes[0][0]]

    for i in range(1, len(indexes)):

        start, idx = indexes[i-1]

        end, \_ = indexes[i]

        substr = S[start:end]

        source, target = sources[idx], targets[idx]

        if substr.startswith(source):

            substr = substr.replace(source, target, 1)

        res += substr

    return res

**818. Race Car**

*Your car starts at position 0 and speed +1 on an infinite number line.  (Your car can go into negative positions.) Your car drives automatically according to a sequence of instructions A (accelerate) and R (reverse). When you get an instruction "A", your car does the following: position += speed, speed \*= 2. When you get an instruction "R", your car does the following: if your speed is positive then speed = -1 , otherwise speed = 1.  (Your position stays the same.) Now for some target position, say the length of the shortest sequence of instructions to get there.*

def racecar(self, target):

***# the sum of geometric progression is a1(1-q\*\*n)/(1-q), here a1 = 1, q = 2, so the sum is 2\*\*n - 1***

***# that means if the target is 2\*\*n - 1, we just need n step to get there***

    dp = {0:0}

    def compute(t):

        if t in dp: return dp[t]

        n = t.bit\_length()

        if 2\*\*n - 1 == t:  ***# if reach t exactly***

            dp[t] = n

            return n

        dp[t] = n + 1 + compute(2\*\*n -1 - t) ***# here (2\*\*n - 1) is the position we go n steps***

        for back in range(n-1): ***# go forward n-1 steps, then R, then go back 'back' steps, then R, the race(t-[2\*\*(n-1)-1]+(2\*\*back-1)) is race(t-2\*\*(n-1)+2\*\*back)***

            dp[t] = min(dp[t], n + 1 + back + compute(t - 2\*\*(n-1) + 2\*\*back)) ***# n-1 + 1R + back + 1R = n+1+back***

        return dp[t]

    return compute(target)

**816. Ambiguous Coordinates**

*We had some 2-dimensional coordinates, like "(1, 3)" or "(2, 0.5)".  Then, we removed all commas, decimal, and spaces, and ended up with the string S.  Return a list of strings representing all possibilities for our original coordinates could have been.*

def ambiguousCoordinates(self, S):

    S = S[1:-1]

    def addDot(s):

        if not s or len(s) > 1 and s[0] == s[-1] == '0':

            return []

        if s[-1] == '0': return [s]

        if s[0] == '0': return ['0.'+s[1:]]

        return [s] + [s[:i] + '.' + s[i:] for i in range(1, len(s))]

    return ['({}, {})'.format(left, right) for i in range(1, len(S)) for left, right in itertools.product(addDot(S[:i]), addDot(S[i:]))]

**815. Bus Routes**

*We have a list of bus routes. Each routes[i] is a bus route that the i-th bus repeats forever. For example if routes[0] = [1, 5, 7], this means that the first bus (0-th indexed) travels in the sequence 1->5->7->1->5->7->1->... forever. We start at bus stop S (initially not on a bus), and we want to go to bus stop T. Travelling by buses only, what is the least number of buses we must take to reach our destination? Return -1 if it is not possible.*

def numBusesToDestination(self, routes, S, T):

    if S == T: return 0  ***# error: forgot this line***

    graph = collections.defaultdict(list) ***# both set and list are OK***

    routes = list(map(set, routes))

    for i, route in enumerate(routes):

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

            route2 = routes[j]

            if any(s in route2 for s in route):  ***# here use s in route2, not any(s1 == s2 for s1 in r1 for s2 in r2)***

                graph[i].append(j)

                graph[j].append(i)

    starts, ends, visited = [], [], [False] \* len(routes)

    for idx, route in enumerate(routes):

        if S in route:

            starts.append(idx)

            visited[idx] = True

        if T in route: ends.append(idx)

    q = deque([(idx, 1) for idx in starts])

    while q:

        idx, depth = q.popleft()

        if idx in ends:  # here is idx in ends, not T in routes[idx]

            return depth

        for nxt in graph[idx]:

            if not visited[nxt]:

                visited[nxt] = True

                q.append((nxt, depth + 1))

    return -1

**813. Largest Sum of Averages**

*We partition a row of numbers A into at most K adjacent (non-empty) groups, then our score is the sum of the average of each group. What is the largest score we can achieve? Note that scores are not necessarily integers.*

def largestSumOfAverages(self, A, K):

    asum = list(itertools.accumulate(A))

    def ave(i, j):

        return (asum[j] - asum[i] + A[i]) / (j - i + 1)

    N = len(A)                                                         ***Actually, we can sue a 2-d array, here use 1-d to reduce space***

    dp = [ave(i, N-1) for i in range(N)]  ***# dp[i] means the result of start from i, base case is k = 1.     ↑***

    for k in range(K-1): ***# every loop, we compute ave(i, j) + ave(j, N), that means k + 1***

        for i in range(N):

            for j in range(i+1, N):

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

    return dp[0]

**809. Expressive Words**

*Sometimes people repeat letters to represent extra feeling, such as "hello" -> "heeellooo", "hi" -> "hiiii".  Here, we have groups, of adjacent letters that are all the same character, and adjacent characters to the group are different.  A group is extended if that group is length 3 or more, so "e" and "o" would be extended in the first example, and "i" would be extended in the second example.  As another example, the groups of "abbcccaaaa" would be "a", "bb", "ccc", and "aaaa"; and "ccc" and "aaaa" are the extended groups of that string. Given a list of query words, return the number of words that are stretchy.*

def expressiveWords(self, S, words):

    ans = []

    s, L = set(S), len(S)

    for word in words:

        if set(word) != s or len(word) > L:***# forgot len(word) > L***

            continue

        i, j, l = 0, 0, len(word)

        while i < l and j < L:

            start\_i, start\_j, count = i, j, 0

            while j < L and word[i] == S[j]:

                count += 1

                j += 1

                if i + 1 < l and word[i+1] == word[i]:

                    i += 1

            if count < 3 and j - start\_j - 1 != i - start\_i:***# here is j - s\_j - 1, forgot -1***

                break

            i += 1

        if i == l and j == L:

            ans.append(word)

    return len(ans)

**779. K-th Symbol in Grammar**

*On the first row, we write a 0. Now in every subsequent row, we look at the previous row and replace each occurrence of 0 with 01, and each occurrence of 1 with 10. Given row N and index K, return the K-th indexed symbol in row N. (The values of K are 1-indexed.) (1 indexed).*

def kthGrammar(self, N, K):

    if K == 1:

        return 0

    return (1 - K % 2) ^ self.kthGrammar(N-1, (K+1) // 2)

**808. Soup Servings**

*There are two types of soup: type A and type B. Initially we have N ml of each type of soup. There are four kinds of operations:*

1. *Serve 100 ml of soup A and 0 ml of soup B*
2. *Serve 75 ml of soup A and 25 ml of soup B*
3. *Serve 50 ml of soup A and 50 ml of soup B*
4. *Serve 25 ml of soup A and 75 ml of soup B*

*When we serve some soup, we give it to someone and we no longer have it.  Each turn, we will choose from the four operations with equal probability 0.25. If the remaining volume of soup is not enough to complete the operation, we will serve as much as we can.  We stop once we no longer have some quantity of both types of soup. Return the probability that soup A will be empty first, plus half the probability that A and B become empty at the same time.*

def soupServings(self, N):

    memo = {}

    if N > 4800: return 1***# the probability of b is used up first is small, so if we can server many times, p near to 1***

    def f(a, b):

        if (a, b) in memo: return memo[a, b]

        if a <= 0 and b <= 0: return 0.5  ***# half of p for a and b used up together***

        if a <= 0: return 1  ***# a used up first***

        if b <= 0: return 0  ***# b used up first***

        memo[(a, b)] = 0.25 \* (f(a - 4, b) + f(a - 3, b - 1) + f(a - 2, b - 2) + f(a - 1, b - 3))***# add up all the choices, then \* 0.25***

        return memo[(a, b)]

    N = math.ceil(N / 25.0)

    return f(N, N)

**802. Find Eventual Safe States**

*In a directed graph, we start at some node and every turn, walk along a directed edge of the graph.  If we reach a node that is terminal (that is, it has no outgoing directed edges), we stop. Now, say our starting node is eventually safe if and only if we must eventually walk to a terminal node.  Which nodes are eventually safe? Return them as an array in sorted order.*

def eventualSafeNodes(self, graph):

    L = len(graph)

    status = [0] \* L  ***# 0 stands for unvisited, 1 stands for safe, -1 stands for unsafe***

    def dfs(node):

        if status[node] != 0:

            return status[node] == 1

        status[node] = -1  ***# we set it to unsafe as default***

        if not all(dfs(nxt) for nxt in graph[node]): ***# if all the children are safe, set to 1(safe), else return False***

            return False

        status[node] = 1

        return True

    return [i for i in range(L) if dfs(i)]  ***# or return list(filter(dfs, range(L))), note that, in Python3, map and filt***

**766. Toeplitz Matrix**

*A matrix is Toeplitz if every diagonal from top-left to bottom-right has the same element. Now given an M x N matrix, return True if and only if the matrix is Toeplitz.*

def isToeplitzMatrix(self, matrix):

    return all(r == 0 or c == 0 or matrix[r-1][c-1] == item

                        for r, row in enumerate(matrix)

                        for c, item in enumerate(row))

**800. Similar RGB Color**

*In the following, every capital letter represents some hexadecimal digit from 0 to f. The red-green-blue color "#AABBCC" can be written as "#ABC" in shorthand.  For example, "#15c" is shorthand for the color "#1155cc". Now, say the similarity between two colors "#ABCDEF" and "#UVWXYZ" is -(AB - UV)^2 - (CD - WX)^2 - (EF - YZ)^2. Given the color "#ABCDEF", return a 7 character color that is most similar to #ABCDEF, and has a shorthand (that is, it can be represented as some "#XYZ"*

def similarRGB(self, color):

    def compute(old):

        old = int(old, 16)

        f, s = divmod(old, 17)***# AA is A \* 16 + A, so the total is A \* 17***

        if s > 8:***# find the similar one, 8 is 16/2, if >8, f+1 is more similar***

            f += 1

        return '{:02x}'.format(f\*17)

    return '#' + compute(color[1:3]) + compute(color[3:5]) + compute(color[5:])

**792. Number of Matching Subsequences**

*Given string S and a dictionary of words words, find the number of words[i] that is a subsequence of S.*

def numMatchingSubseq(self, S, words):

***# if len(S) is very large, we need to run lots of comparations, this method only run 1 time of len(S)***

    waiting = collections.defaultdict(list)

    for it in map(iter, words):

        waiting[next(it)].append(it)

    for c in S:

        for it in waiting.pop(c, ()):

            waiting[next(it, None)].append(it)

    return len(waiting[None])

**778. Swim in Rising Water**

*On an N x N grid, each square grid[i][j] represents the elevation at that point (i,j). Now rain starts to fall. At time t, the depth of the water everywhere is t. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most t. You can swim infinite distance in zero time. Of course, you must stay within the boundaries of the grid during your swim. You start at the top left square (0, 0). What is the least time until you can reach the bottom right square (N-1, N-1)?*

***# there are two ways to solve this problem. but we can't avoid to loop all the grid everytime! so the key is to get the nxt node to begin loop. the first way is to use a priorityqueue to get the nxt node. Or we can always start from (0,0) and loop the grid with specific height. then we can use binary search to get the right height.***

def swimInWater(self, grid):

    N, visited, dirs = len(grid), set(),  ((0, 1), (0, -1), (-1, 0), (1, 0))

    pq = PriorityQueue()

    pq.put((grid[0][0], 0, 0))

    visited.add((0, 0))

    ans = 0

    while not pq.empty():

        h, x, y = pq.get()

        ans = max(ans, h)

        if x == N - 1 and y == N - 1:

            return ans

        for dx, dy in dirs:

            nx, ny = x + dx, y + dy

            if 0 <= nx < N and 0 <= ny < N and (nx, ny) not in visited:

                visited.add((nx, ny))

                pq.put((grid[nx][ny], nx, ny))

**790. Domino and Tromino Tiling**

*We have two types of tiles: a 2x1 domino shape, and an "L" tromino shape. These shapes may be rotated.  Given N, how many ways are there to tile a 2 x N board? Return your answer modulo 10^9 + 7.*

def numTilings(self, N):

    if not N:

        return 0

    base = [1, 2, 5]

    if N <= 3:

        return base[N-1]

    M = 1e9 + 7

    dp1, dp2, dp3 = base

    ans = 0

    for i in range(4, N+1):

        ans = (2\* dp3 + dp1) % M

        dp1, dp2, dp3 = dp2, dp3, ans

    return int(ans)

**789. Escape The Ghosts**

*You start at the point (0, 0), and your destination is (target[0], target[1]). There are several ghosts on the map, the i-th ghost starts at (ghosts[i][0], ghosts[i][1]). Each turn, you and all ghosts simultaneously \*may\* move in one of 4 cardinal directions going from the previous point to a new point 1 unit of distance away. You escape if and only if you can reach the target before any ghost reaches you (for any given moves the ghosts may take.)  If you reach any square (including the target) at the same time as a ghost, it doesn't count as an escape. Return True if and only if it is possible to escape.*

def escapeGhosts(self, ghosts, target):

    def dis(a, b=target):

        (ax, ay), (bx, by) = a, b

        return abs(ax - bx) + abs(ay - by)

    start = (0, 0)

    road = dis(start)

    return all(dis(g) > road for g in ghosts)

**788. Rotated Digits**

*X is a good number if after rotating each digit individually by 180 degrees, we get a valid number that is different from X.  Each digit must be rotated - we cannot choose to leave it alone. A number is valid if each digit remains a digit after rotation. 0, 1, and 8 rotate to themselves; 2 and 5 rotate to each other; 6 and 9 rotate to each other, and the rest of the numbers do not rotate to any other number and become invalid. Now given a positive number N, how many numbers X from 1 to N are good?*

def rotatedDigits(self, N):

    ans = 0

    valid = '0182569'

    good = '2569'

    for i in range(N+1):

        s = str(i)

        if all(c in valid  for c in s):

            if any(c in good for c in s):

                ans += 1

    return ans

**783. Minimum Distance Between BST Nodes**

*Given a Binary Search Tree (BST) with the root node root, return the minimum difference between the values of any two different nodes in the tree.*

def minDiffInBST(self, root):

    self.ans = float('inf')

    def dfs(node=root, l=float('-inf'), r=float('inf')):

        self.ans = min(self.ans, node.val - l, r - node.val)

        if node.left:

            dfs(node.left, l, node.val)

        if node.right:

            dfs(node.right, node.val, r)

    dfs()

    return self.ans

**782. Transform to Chessboard**

*An N x N board contains only 0s and 1s. In each move, you can swap any 2 rows with each other, or any 2 columns with each other. What is the minimum number of moves to transform the board into a "chessboard" - a board where no 0s and no 1s are 4-directionally adjacent? If the task is impossible, return -1.*

def movesToChessboard(self, board):

    N, ans = len(board), 0

***# For each count of lines from {rows, columns}...***

    for count in (collections.Counter(map(tuple, board)), collections.Counter(zip(\*board))):

***# If there are more than 2 kinds of lines, or if the number of kinds is not appropriate ...***

        if len(count) != 2 or sorted(count.values()) != [N//2, (N+1)//2]: return -1

***# If the lines are not opposite each other, impossible***

        line1, line2 = count

        if not all(x ^ y for x, y in zip(line1, line2)): return -1

***# starts = what could be the starting value of line1*. *If N is odd, we have to start with the more* *frequent element***

        starts = [(line1.count(1) \* 2 > N)] if N%2 else [0, 1]

***# To transform line1 into the ideal line [i%2 for i ...], we take the number of differences and divide by two***

        ans += min(sum((i-x) % 2 for i, x in enumerate(line1, start))

                   for start in starts) // 2

    return ans

***# another method***

def movesToChessboard1(self, b):

    N = len(b)

    if any(b[0][0] ^ b[i][0] ^ b[0][j] ^ b[i][j] for i in range(N) for j in range(N)): return -1

    if not N // 2 <= sum(b[0]) <= (N + 1) // 2: return -1

    if not N // 2 <= sum(b[i][0] for i in range(N)) <= (N + 1) // 2: return -1

    col = sum(b[0][i] == i % 2 for i in range(N))

    row = sum(b[i][0] == i % 2 for i in range(N))

    if N % 2:

        if col % 2: col = N - col

        if row % 2: row = N - row

    else:

        col = min(N - col, col)

        row = min(N - row, row)

    return (col + row) // 2

**780. Reaching Points**

*A move consists of taking a point (x, y) and transforming it to either (x, x+y) or (x+y, y). Given a starting point (sx, sy) and a target point (tx, ty), return True if we can move from the point (sx, sy) to (tx, ty). Otherwise, return False.*

def reachingPoints(self, sx, sy, tx, ty):

    while tx > sx and ty > sy:***# here is > not >=***

        if tx > ty:

            tx %= ty

        else:

            ty %= tx

***# the return kind of complex, but easy to understand***

    return tx == sx and (ty - sy) % sx == 0 or ty == sy and (tx - sx) % sy == 0

**777. Swap Adjacent in LR String**

*In a string composed of 'L', 'R', and 'X' characters, like "RXXLRXRXL", a move consists of either replacing one occurrence of "XL" with "LX", or replacing one occurrence of "RX" with "XR". Given the starting string start and the ending string end, return True if and only if there exists a sequence of moves to transform one string to the other.*

def canTransform(self, start, end):

    if start.replace('X', '') != end.replace('X', ''): return False

    j = 0

    for i, c in enumerate(start):

        if c == 'L':

            while end[j] != c:

                j += 1

            if j > i:  return False  ***# L in end must less than in start, so j must < i, if j > i, return False***

            j += 1  ***# forgot +1 here***

    j = 0

    for i, c in enumerate(start):

        if c == 'R':

            while end[j] != c:

                j += 1

            if i > j: return False

            j += 1

    return True

**753. Cracking the Safe**

*There is a box protected by a password. The password is n digits, where each letter can be one of the first k digits 0, 1, ..., k-1. You can keep inputting the password, the password will automatically be matched against the last n digits entered. For example, assuming the password is "345", I can open it when I type "012345", but I enter a total of 6 digits. Please return any string of minimum length that is guaranteed to open the box after the entire string is inputted.*

def crackSafe(self, n, k):

    ans, visited = [], set()

    def dfs(cur):

        for i in range(k):

            nxt = cur + str(i)

            if nxt not in visited:

                visited.add(nxt)

                dfs(nxt[1:])

                ans.append(str(i))

    cur = '0' \* (n - 1)

    dfs(cur)

    return ''.join(ans) + '0' \* (n - 1)

**774. Minimize Max Distance to Gas Station**

*On a horizontal number line, we have gas stations at positions stations[0], stations[1], ..., stations[N-1], where N = stations.length. Now, we add K more gas stations so that D, the maximum distance between adjacent gas stations, is minimized. Return the smallest possible value of D.*

***# we can use a pq to store all the intervals, and insert stations into interval, this need O(KlogN), k is the insert times, logN is the depth of the heap.***

***# here we use binary seach, it needs O(NlogM), N is the num of stations, the check function use O(N), M is the max value of D, it may be the max of the gas station's position, at most 10^8. log(M) is binary search times.***

***# the two method above, since K maybe 10^6, much bigger than N, we'd like to use O(NlogM) rather than O(KlogN)***

def minmaxGasDist(self, stations, K):

***# if the minmum dis between stations is d, that means interval / d < 1, so (s[i] - s[i-1]) // d is the number of stations we can insert, we add them up, if the sum <= K, this d is valid***

    def check(d):

        return sum((stations[i] - stations[i-1]) // d for i in range(1, len(stations))) <= K

    l, r = 0, 1e8

    while r - l > 1e-6:

        mid = (r + l) / 2***# error: result can be float, so don't use // 2***

        if check(mid):

            r = mid

        else:

            l = mid  ***# if mid is not possible, l = mid, don't use l = mid + 1, because mid is float, +1 may exceed the result***

    return l

**769. Max Chunks To Make Sorted**

*Given an array arr that is a permutation of [0, 1, ..., arr.length - 1], we split the array into some number of "chunks" (partitions), and individually sort each chunk.  After concatenating them, the result equals the sorted array. What is the most number of chunks we could have made?*

def maxChunksToSorted(self, arr):

    ans, N = 0, len(arr)

    high = 0

    for idx, num in enumerate(arr):

        high = max(high, num)

        if high == idx:

            ans += 1

    return ans

**768. Max Chunks To Make Sorted II**

*This question is the same as "Max Chunks to Make Sorted" except the integers of the given array are not necessarily distinct, the input array could be up to length 2000, and the elements could be up to 10\*\*8.*

def maxChunksToSorted(self, arr):

    count, order = collections.Counter(), []

    for num in arr:

        count[num] += 1

        order.append((num, count[num]))

    ans, cur = 0, (0, 0)

    for x, y in zip(order, sorted(order)):

        cur = max(cur, x)

        if cur == y:

            ans += 1

    return ans

**556. Next Greater Element III**

*Given a positive 32-bit integer n, you need to find the smallest 32-bit integer which has exactly the same digits existing in the integer n and is greater in value than n. If no such positive 32-bit integer exists, you need to return -1.*

def nextGreaterElement(self, n):

    ans = list(str(n))

    i = len(ans) - 2

    while i >= 0:

        if ans[i] < ans[i+1]:

            break

        i -= 1

    if i < 0:

        return -1

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

        if ans[j] > ans[i]:

            ans[i], ans[j] = ans[j], ans[i]

            break

    ans = int(''.join(ans[:i+1] + ans[i+1:][::-1]))

    return  ans if ans < 2\*\*31 else -1

**681. Next Closest Time**

*Given a time represented in the format "HH:MM", form the next closest time by reusing the current digits. There is no limit on how many times a digit can be reused. You may assume the given input string is always valid. For example, "01:34", "12:09" are all valid. "1:34", "12:9" are all invalid.*

def nextClosestTime(self, time):

    time = time.replace(':', '')

    digits = {int(c) for c in time}

    cur = 60 \* int(time[:2]) + int(time[2:])

    over = 60 \* 24 ***# can't use float('inf'), otherwise, the result is also float***

    for h1, h2, m1, m2 in itertools.product(digits, repeat=4):

        hour = 10 \* h1 + h2

        minute = 10 \* m1 + m2

        if hour < 24 and minute < 60:

            temp = hour \* 60 + minute

            temp\_over = (temp - cur) % (60 \* 24)

            if 0 < temp\_over < over:  ***# error: can't use min(over, cur\_over), because if over is 0, it'll the same time.***

                over = temp\_over

    return '{:02d}:{:02d}'.format(\*divmod((cur + over) % (60 \* 24), 60)) ***# error: forgot %***

**724. Find Pivot Index**

*Given an array of integers nums, write a method that returns the "pivot" index of this array. We define the pivot index as the index where the sum of the numbers to the left of the index is equal to the sum of the numbers to the right of the index. If no such index exists, we should return -1. If there are multiple pivot indexes, you should return the left-most pivot index.*

def pivotIndex(self, nums):

    asum = list(itertools.accumulate(nums))

    for i, s in enumerate(asum):

        if s - nums[i] == asum[-1] - s:

            return i

    return -1

**133. Clone Graph**

*Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.*

def cloneGraph(self, node):

    clones = {}

    def clone(root=node):

        if not root:

            return None

        if root.label in clones:

            return clones[root.label]

        new\_root = UndirectedGraphNode(root.label)

        clones[root.label] = new\_root  ***# error: we should put new to clones here, not after the for***

        for nb in root.neighbors:

            new\_root.neighbors.append(clone(nb))

        return new\_root

    return clone()

**683. K Empty Slots**

*There is a garden with N slots. In each slot, there is a flower. The N flowers will bloom one by one in N days. In each day, there will be exactly one flower blooming and it will be in the status of blooming since then. Given an array flowers consists of number from 1 to N. Each number in the array represents the place where the flower will open in that day. For example, flowers[i] = x means that the unique flower that blooms at day i will be at position x, where* ***i and x will be in the range from 1 to N****. Also given an integer k, you need to output in which day there exists two flowers in the status of blooming, and also the number of flowers between them is k and these flowers are not blooming. If there isn't such day, output -1.*

def kEmptySlots(self, flowers, k):

    blooms = []

    for day, position in enumerate(flowers, 1):   ***# start from 1***

        idx = bisect.bisect(blooms, position)

        for neighbor in blooms[max(0, idx-1):idx+1]:

            if abs(neighbor-position) - 1 == k: ***# error: here needs -1***

                return day

        blooms.insert(idx, position)

    return -1

**271. Encode and Decode Strings**

*Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.*

class Codec:

    def encode(self, strs):

        return ''.join('{:d}:{}'.format(len(s), s) for s in strs)

    def decode(self, s):

        strs = []

        i = 0

        while i < len(s):

            j = s.find(':', i) ***# find('x', start\_idx)***

            i = j + 1 + int(s[i:j])

            strs.append(s[j+1:i])

        return strs

**480. Sliding Window Median**

*Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.*

def medianSlidingWindow(self, nums, k):  *# Time: O(n⋅k⋅log(k))      Space: O(K)*

    win, res = [], []

    for i, n in enumerate(nums):

        if i >= k: win.pop(bisect.bisect(win, nums[i-k])-1)

        bisect.insort(win, nums[i])

        if i >= k - 1:

            res.append(float(win[k//2]) if k % 2 else (win[k//2] + win[k//2 - 1]) / 2.0)  ***# here is float***

    return res

**481. Magical String**

*A magical string S consists of only '1' and '2' and obeys the following rules: The string S is magical because concatenating the number of contiguous occurrences of characters '1' and '2' generates the string S itself.*

*The first few elements of string S is the following: S = "1221121221221121122……"*

*S : 1 22 11 2 1 22 1 22 11 2 11 22 ......  and the occurrences of '1's or '2's in each group are:*

*# of group: 1 2 2 1 1 2 1 2 2 1 2 2 ...... You can see that the occurrence sequence above is the S itself.*

*Given an integer N as input, return the number of '1's in the first N number in the magical string S.*

def magicalString(self, n):

    if n <= 0: return 0

    if n <= 3: return 1

    a = [0] \* (n+1)

    a[0], a[1], a[2] = 1, 2, 2

    head, tail, num, res = 2, 3, 1, 1

    while tail < n:

        for i in range(a[head]):

            a[tail] = num

            if num == 1 and tail < n:

                res += 1

            tail += 1

        num = 3 ^ num  ***# change 1 to 2, 2 to 1***

        head += 1

    return res

**408. Valid Word Abbreviation**

*Given a non-empty string s and an abbreviation abbr, return whether the string matches with the given abbreviation. Given s = "internationalization", abbr = "i12iz4n": Return true.*

def validWordAbbreviation(self, word, abbr):

    i, j, m, n = 0, 0, len(word), len(abbr)

    while i < m and j < n:

        if word[i]==abbr[j]:

            i+=1

            j+=1

        elif abbr[j].isalpha() or abbr[j] == '0':  ***# forgot '0'***

            return False

        else:

            start = j

            while j < n and abbr[j].isdigit():

                j += 1

            num = int(abbr[start:j])

            i += num

    return i == m and j == n

**409. Longest Palindrome**

*Given a string which consists of letters, find the length of the longest palindromes that can be built with those letters.*

def longestPalindrome(self, s):

    odds = sum(n % 2 for n in collections.Counter(s).values())

    return len(s) - odds + bool(odds)

**484. Find Permutation**

*By now, you are given a secret signature consisting of character 'D' and 'I'. 'D' represents a decreasing relationship between two numbers, 'I' represents an increasing relationship between two numbers. And our secret signature was constructed by a special integer array, which contains uniquely all the different number from 1 to n (n is the length of the secret signature plus 1). For example, the secret signature "DI" can be constructed by array [2,1,3] or [3,1,2], but won't be constructed by array [3,2,4] or [2,1,3,4], which are both illegal constructing special string that can't represent the "DI" secret signature. On the other hand, now your job is to find the lexicographically smallest permutation of [1, 2, ... n] could refer to the given secret signature in the input.*

def findPermutation(self, s):

    res = []

    for i in range(len(s)):

        if s[i] == 'I':  ***# when we meet a I, insert a decrease string***

            res.extend(range(i + 1, len(ret), -1))

    res.extend(range(len(s) + 1, len(ret), -1))***# insert the rest***

    return res

**402. Remove K Digits**

*Given a non-negative integer num represented as a string, remove k digits from the number so that the new number is the smallest possible.*

def removeKdigits(self, num, k):

    res = []

    for c in num:

        while k and res and res[-1] > c:

            res.pop()

            k -= 1

        res.append(c)

    return ''.join(res[:-k or len(res)]).lstrip('0') or '0'  ***# if k == 0, join all res***

**494. Target Sum**

*You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols + and -. For each integer, you should choose one from + and - as its new symbol. Find out how many ways to assign symbols to make sum of integers equal to target S.*

def findTargetSumWays(self, nums, S):***# if we put some -, we change some to negative, and ↓***

    def subSum(s):                       ***#                                  sum(P) - sum(N) = target***

        dp = [0] \* (s + 1)***# sum(P) + sum(N) + sum(P) - sum(N) = target + sum(P) + sum(N)***

        dp[0] = 1                             ***#                                            2 \* sum(P) = target + sum(nums)***

        for n in nums:

            for i in range(s, n - 1, -1):

                dp[i] += dp[i-n]

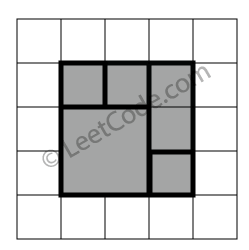
        return dp[s]

    s = sum(nums)

    if s < S or (s + S) % 2:

        return 0

    return subSum((s+S) // 2)



**391. Perfect Rectangle**

*Given N axis-aligned rectangles where N > 0, determine if they all together form an exact cover of a rectangular region. Each rectangle is represented as a bottom-left point and a top-right point. For example, a unit square is represented as [1,1,2,2]. (coordinate of bottom-left point is (1, 1) and top-right point is (2, 2)).*

def isRectangleCover(self, rectangles):

    bl\_x, bl\_y, tr\_x, tr\_y = float('inf'), float('inf'), float('-inf'), float('-inf')

    s, area = set(), 0

    for x1, y1, x2, y2 in rectangles:

        bl\_x = min(x1, bl\_x)

        bl\_y = min(y1, bl\_y)

        tr\_x = max(x2, tr\_x)

        tr\_y = max(y2, tr\_y)

        area += (y2 - y1) \* (x2 - x1)

        p1, p2, p3, p4 = (x1, y1), (x1, y2), (x2, y1), (x2, y2)

        for p in (p1, p2, p3, p4):  ***# every point appear two times except the outer 4 points.***

            if p in s:

                s.remove(p)

            else:

                s.add(p)

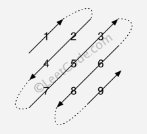
    p1, p2, p3, p4 = (bl\_x, bl\_y), (bl\_x, tr\_y), (tr\_x, tr\_y), (tr\_x, bl\_y)

    return s == {p1, p2, p3, p4} and area == (tr\_y - bl\_y) \* (tr\_x-bl\_x)

**498. Diagonal Traverse**

*Given a matrix of M x N elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image.*

***# just walk over the matrix in the desired order. My d is the diagonal number, i.e., i+j. So I can compute j as d-i.***

***# Why the range range(max(0, d-n+1), min(d+1, m))? Well I need 0 <= i < m and 0 <= j < n. As said above, j is d-i, so I have 0 <= d-i < n. Isolating i gives me i <= d and i > d-n. Since we're dealing with integers, they're equivalent to i < d+1 and i >= d-n+1. So my i needs to be in the range [0, m) as well as in the range [d-n+1, d+1). And my range is simply the intersection of those two ranges.***

def findDiagonalOrder(self, matrix):

    m, n = len(matrix), len(matrix and matrix[0])

    return [matrix[i][d-i]

            for d in range(m+n-1)

            for i in range(max(0, d-n+1), min(d+1, m))[::d%2\*2-1]]

**331. Verify Preorder Serialization of a Binary Tree**

*One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #.*

***# all non-null node provides 2 outdegree and 1 indegree (2 children and 1 parent), except root***

***# all null node provides 0 outdegree and 1 indegree (0 child and 1 parent).***

def isValidSerialization(self, preorder):

    vals = preorder.split(',')

    diff = 1

    for val in vals:

        diff -= 1

        if diff < 0: return False

        if val != '#':

            diff += 2

    return diff == 0

**255. Verify Preorder Sequence in Binary Search Tree**

*Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree. You may assume each number in the sequence is unique.*

def verifyPreorder(self, preorder):

    stack, low = [],  float('-inf')

    for p in preorder:

        if p < low: return False

        while stack and p > stack[-1]:

            low = stack.pop()

        stack.append(p)

    return True

**336. Palindrome Pairs**

*Given a list of unique words, find all pairs of distinct indices (i, j) in the given list, so that the concatenation of the two words, i.e. words[i] + words[j] is a palindrome.*

def palindromePairs(self, words):

    def is\_palindrome(check):

        return check == check[::-1]

    words = {word: i for i, word in enumerate(words)}

    valid\_pals = []

    for word, k in words.items():

        n = len(word)

        for j in range(n+1):

            pref = word[:j]

            suf = word[j:]

            if is\_palindrome(pref):

                back = suf[::-1]

                if back != word and back in words:

                    valid\_pals.append([words[back],  k])

            if j != n and is\_palindrome(suf):

                back = pref[::-1]

                if back != word and back in words:

                    valid\_pals.append([k, words[back]])

    return valid\_pals

**332. Reconstruct Itinerary**

*Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order. All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.*

*1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].*

*2. All airports are represented by three capital letters (IATA code). You may assume all tickets form at least one valid itinerary.*

def findItinerary(self, tickets):

    targets = collections.defaultdict(list)

    for a, b in sorted(tickets)[::-1]:

        targets[a] += b,

    route, stack = [], ['JFK']

    while stack:

        while targets[stack[-1]]:

            stack += targets[stack[-1]].pop(),

        route += stack.pop(),

    return route[::-1]

**378. Kth Smallest Element in a Sorted Matrix**

*Given a n x n matrix where each of the rows and columns are sorted in ascending order, find the kth smallest item in the matrix.*

def kthSmallest(self, matrix, k):

    rows, cols = len(matrix), len(matrix[0])

    lo, hi = matrix[0][0], matrix[-1][-1]

    while lo < hi:

        mid = (hi + lo) // 2

        count, col = 0, cols - 1

        for row in range(rows):

            while col >= 0 and matrix[row][col] > mid:

                col -= 1

            count += (col + 1)

        if count < k:

            lo = mid + 1

        else:

            hi = mid

    return lo

**393. UTF-8 Validation**

*A character in UTF8 can be from 1 to 4 bytes long, subjected to the following rules:*

*For 1-byte character, the first bit is a 0, followed by its unicode code.*

*For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10.*

*0000 0000-0000 007F | 0xxxxxxx*

*0000 0080-0000 07FF | 110xxxxx 10xxxxxx*

*0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx*

*0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx*

def validUtf8(self, data):

    count = 0;

    for c in data:

        if count == 0:

            if (c >> 5) == 0b110:

                count = 1

            elif (c >> 4) == 0b1110:

                count = 2

            elif (c >> 3) == 0b11110:

                count = 3

            elif (c >> 7):

                return False

        else:

            if (c >> 6) != 0b10: return False

            count -= 1

    return count == 0

**686. Repeated String Match**

*Given strings A and B, find the minimum times A has to be repeated so that B is a substring of it. If no such solution, return -1.*

def repeatedStringMatch(self, A, B):  *#O(N∗(N+M)),  O(M+N)*

    q = (len(B) - 1) // len(A) + 1

    for i in range(2):

        if B in A \* (q+i):

            return q + i

    return -1

**501. Find Mode in Binary Search Tree**

*Given a binary search tree (BST) with duplicates, find all the mode(s) (the most frequently occurred element) in the given BST.*

def findMode(self, root):

    if not root:

        return []

    count = collections.Counter()

    def dfs(node):

        if node:

            count[node.val] += 1

            dfs(node.left)

            dfs(node.right)

    dfs(root)

    max\_count = max(count.values())

    return [k for k, v in count.items() if v == max\_count]

**520. Detect Capital**

*Given a word, you need to judge whether the usage of capitals in it is right or not.*

def detectCapitalUse(self, word):

    return word.isupper() or word.islower() or word.istitle()

**521. Longest Uncommon Subsequence I**

*Given a group of two strings, you need to find the longest uncommon subsequence of this group of two strings. The longest uncommon subsequence is defined as the longest subsequence of one of these strings and this subsequence should not be any subsequence of the other strings.*

def findLUSlength(self, a, b):

    return -1 if a == b else max(len(a), len(b))

**522. Longest Uncommon Subsequence II**

*Same with 521, but Given a list of strings instead of two words.*

def findLUSlength(self, strs):

    def subseq(w1, w2): ***# True iff word1 is a subsequence of word2.***

        i = 0

        for c in w2:

            if i < len(w1) and w1[i] == c:

                i += 1

        return i == len(w1)

    strs.sort(key=len, reverse=True)  ***# if we find one, it is the longest***

    for i, word1 in enumerate(strs):

        if all(not subseq(word1, word2) for j, word2 in enumerate(strs) if i != j):

            return len(word1)

    return -1

**506. Relative Ranks**

*Given scores of N athletes, find their relative ranks and the people with the top three highest scores, who will be awarded medals: "Gold Medal", "Silver Medal" and "Bronze Medal".*

def findRelativeRanks(self, nums):

    s = {n: i for i, n in enumerate(sorted(nums, reverse=True))}

    medals = ['Gold', 'Silver', 'Bronze']

    return [str(s[n]+1) if s[n] >= len(medals) else (medals[s[n]] + ' Medal') for n in nums]

**719. Find K-th Smallest Pair Distance**

*Given an integer array, return the k-th smallest distance among all the pairs. The distance of a pair (A, B) is defined as the absolute difference between A and B.*

def smallestDistancePair(self, nums, k):***# O(NlogW+NlogN)  O(1)***

    def possible(guess): ***# Is there k or more pairs with distance <= guess?***

        count = left = 0

        for right, x in enumerate(nums):

            while x - nums[left] > guess:

                left += 1

            count += right - left

        return count >= k

    nums.sort()

    lo = 0

    hi = nums[-1] - nums[0]

    while lo < hi:

        mi = (lo + hi) // 2

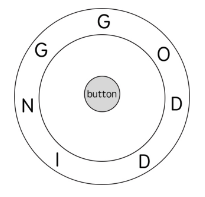
        if possible(mi):

            hi = mi

        else:

            lo = mi + 1

    return lo

**514. Freedom Trail**

*In the video game Fallout 4, the quest "Road to Freedom" requires players to reach a metal dial called the "Freedom Trail Ring", and use the dial to spell a specific keyword in order to open the door.   Input: ring = "godding", key = "gd" Output: 4 Explanation:*

*For the first key character 'g', since it is already in place, we just need 1 step to spell this character.  For the second key character 'd', we need to rotate the ring "godding" anticlockwise by two steps to make it become "ddinggo".  Also, we need 1 more step for spelling. So the final output is 4.*

def findRotateSteps(self, ring, key):

    def dist(i, j):  ***# the distance between two points (i, j) on the ring***

        return min(abs(i - j), len(ring) - abs(i - j))

    pos = {}  ***# build the position list for each character in ring***

    for i, c in enumerate(ring):

        if c in pos:

            pos[c].append(i)

        else:

            pos[c] = [i]

    state = {0: 0}  ***# the current possible state: {position of the ring: the cost}***

    for c in key:

        next\_state = {}

        for j in pos[c]:  ***# every possible target position***

            next\_state[j] = float('inf')

            for i in state:  ***# every possible start position***

                next\_state[j] = min(next\_state[j], dist(i, j) + state[i])

        state = next\_state

    return min(state.values()) + len(key)

**727. Minimum Window Subsequence**

*Given strings S and T, find the minimum (contiguous) substring W of S, so that T is a subsequence of W.*

def minWindow(self, S, T):  ***# O(ST)  O(S)***

    cur = [i if x == T[0] else -1 for i, x in enumerate(S)]

    for j in range(1, len(T)):  ***# At time j when considering T[:j+1], the smallest window [s, e] where S[e] == T[j]***

        last = -1                      ***# is represented by cur[e] = s.***

        new = [-1] \* len(S)

        for i, u in enumerate(S): ***#we calculate the new windows, "new" for T[:j+1].  'last' is the last window seen.***

            if last != -1 and u == T[j]:

                new[i] = last

            if cur[i] != -1:

                last = cur[i]

        cur = new

    ans = 0, len(S)  ***#Looking at the window data cur, choose the smallest length window [s, e].***

    for e, s in enumerate(cur):

        if s >= 0 and e - s < ans[1] - ans[0]:

            ans = s, e

    return S[ans[0]: ans[1]+1] if ans[1] < len(S) else ""

**261. Graph Valid Tree**

*Given n nodes labeled from 0 to n-1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.*

def validTree(self, n, edges):

    base = {}

    def find(x):

        if x not in base:

            base[x] = x

        if base[x] != x:

            base[x] = find(base[x])

        return base[x]

    def union(a, b):

        ra, rb = find(a), find(b)

        if ra != rb:

            base[ra] = rb

        return ra != rb

    return len(edges) == n-1 and all(union(a, b) for a, b in edges)

**266. Palindrome Permutation**

*Given a string, determine if a permutation of the string could form a palindrome.*

def canPermutePalindrome(self, s):

    return sum(v % 2 for v in collections.Counter(s).values()) <= 1

**267. Palindrome Permutation II**

*Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.*

def generatePalindromes(self, s):  ***# O(n/2 + 1)!     O(n)***

    counts = collections.Counter(s)

    odd = ''

    for ch in counts:

        if counts[ch] % 2 == 1:

            if odd != '':

                return []

            odd = ch

        counts[ch] //= 2

    perms = self.get\_permutations(counts)

    return [word + odd + word[::-1] for word in perms]

def get\_permutations(self, counts):

    strs = []

    def helper(s):

        if counts.most\_common(1)[0][1] == 0:  ***# most\_common(n) n is top n***

            strs.append(s)

            return

        for ch in counts:

            if counts[ch] == 0:  ***# count unique character only.  Each time you consider a character the count itself is***

                continue            ***# like an index of the duplicated number. This implies that the 'a' with count (index) 8,***

            counts[ch] -= 1      ***# will never occur after another 'a' with count (index) 6.***

            helper(s + ch)

            counts[ch] += 1

    helper('')

    return strs

**259. 3Sum Smaller**

*Given an array of n integers nums and a target, find the number of index triplets i, j, k with 0 <= i < j < k < n that satisfy the condition nums[i] + nums[j] + nums[k] < target.*

def threeSumSmaller(self, nums, target):

    result = 0

    nums.sort()

    for i in range(len(nums) - 2):

        l, r = i + 1, len(nums) - 1

        while l < r:

            s = nums[i] + nums[l] + nums[r]

            if s < target:

                result += r - l

                l += 1

            else:

                r -= 1

    return result

**257. Binary Tree Paths**

*Given a binary tree, return all root-to-leaf paths.*

def binaryTreePaths(self, root):

    if not root: return []

    ans = []

    def dfs(node=root, path=[]):

        if not node.left and not node.right:

            ans.append(path+[node.val])

        else:

            if node.left:

                dfs(node.left, path+[node.val])

            if node.right:

                dfs(node.right, path+[node.val])

    dfs()

    return ['->'.join(map(str, path)) for path in ans]

**253. Meeting Rooms II**

*Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), find the minimum number of conference rooms required.*

def minMeetingRooms(self, intervals):

    starts, ends = [], []

    for i in intervals:

        starts.append(i.start)

        ends.append(i.end)

    starts.sort()

    ends.sort()

    ans = over = 0

    for start in starts:

        if start < ends[over]:

            ans += 1

        else:

            over += 1

    return ans

**251. Flatten 2D Vector**

*Implement an iterator to flatten a 2d vector.*

class Vector2D(object):

    def \_\_init\_\_(self, vec2d):

        self.col, self.row, self.vec = 0, 0, vec2d

    def next(self):

        result = self.vec[self.row][self.col]

        self.col += 1

        return result

    def hasNext(self):

        while self.row < len(self.vec):

            if self.col < len(self.vec[self.row]): return True

            self.col = 0

            self.row += 1

        return False

**276. Paint Fence**

*There is a fence with n posts, each post can be painted with one of the k colors. You have to paint all the posts such that no more than two adjacent fence posts have the same color. Return the total number of ways you can paint the fence.*

def numWays(self, n, k):

    w = [0, k, k\*k]

    while len(w) <= n:

        w.append(sum(w[-2:]) \* (k-1))

    return w[n]  ***# here is n, not -1, because n may be 0, 1***

**240. Search a 2D Matrix II**

*Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:*

*Integers in each row are sorted in ascending from left to right.*

*Integers in each column are sorted in ascending from top to bottom.*

def searchMatrix(self, matrix, target):

    if len(matrix) == 0 or len(matrix[0]) == 0: return False

    rows, cols = len(matrix), len(matrix[0])

    row, col = rows - 1, 0

    while row >= 0 and col < cols:

        if matrix[row][col] > target:

            row -= 1

        elif matrix[row][col] < target:

            col += 1

        else: *# found it*

            return True

    return False

**17. Letter Combinations of a Phone Number**

*Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent.*

def letterCombinations(self, digits):

    dic = {'2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl', '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'}

    if len(digits) == 0:

        return []

    if len(digits) == 1:

        return list(dic[digits[0]])

    prev = self.letterCombinations(digits[:-1])

    additional = dic[digits[-1]]

    return [s + c for s in prev for c in additional]

**162. Find Peak Element**

*A peak element is an element that is greater than its neighbors. Given an array nums, where nums[i] ≠ nums[i+1], find a peak element and return its index. The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.*

def findPeakElement(self, nums):

    l, r = 0, len(nums) -1

    while l < r:

        mid = (l + r) // 2

        if nums[mid] > nums[mid + 1]:

            r = mid

        else:

            l = mid + 1

    return l

**139. Word Break**

*Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, determine if s can be segmented into a space-separated sequence of one or more dictionary words.*

def wordBreak(self, s, wordDict):

    dp = [True]

    for i in range(1, len(s) + 1):

        res = any(dp[j] and s[j:i] in wordDict for j in range(i))

        dp += [res]

    return dp[-1]

**57. Insert Interval**

*Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary). You may assume that the intervals were initially sorted according to their start times.*

def insert(self, intervals, newInterval):

    s, e, left, right = newInterval.start, newInterval.end, [], []

    for i in intervals:

        if i.end < s:

            left.append(i)

        elif i.start > e:

            right.append(i)

        else:

            s, e = min(s, i.start), max(e, i.end)

    return left + [Interval(s, e)] + right

**56. Merge Intervals**

*Given a collection of intervals, merge all overlapping intervals.*

def merge(self, intervals):

    ans = []

    for i in sorted(intervals, key=lambda interval: interval.start):

        if ans and i.start <= ans[-1].end:

            ans[-1].end = max(ans[-1].end, i.end)

        else:

            ans.append(i)

    return ans

**54. Spiral Matrix**

*Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.*

def spiralOrder(self, matrix):

    ans = []

    while matrix:

        ans.extend(matrix.pop(0))

        if matrix and matrix[0]:

            for row in matrix:

                ans.append(row.pop())

        if matrix:

            ans.extend(matrix.pop()[::-1])

        if matrix and matrix[0]:

            for row in matrix[::-1]:

                ans.append(row.pop(0))

    return ans

**59. Spiral Matrix II**

*Given a positive integer n, generate a square matrix filled with elements from 1 to n2 in spiral order.*

def generateMatrix(self, n):

    res = [[0] \* n for \_ in range(n)]

    x, y, dx, dy = 0, 0, 0, 1

    for k in range(n\*n):

        res[x][y] = k + 1

        if res[(x + dx) % n][(y + dy) % n]:

            dx, dy = dy, -dx

        x += dx

        y += dy

    return res

**155. Min Stack**

*Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.*

class MinStack:

    def \_\_init\_\_(self):

        self.q = []

    def push(self, x):

        curMin = self.getMin()

        if curMin == None or x < curMin:

            curMin = x

        self.q.append((x, curMin));

    def pop(self):

        if self.q:

            self.q.pop()

    def top(self):

        if self.q:

            return self.q[len(self.q) - 1][0]

        return None

    def getMin(self):

        if self.q:

            return self.q[len(self.q) - 1][1]

        return None

**716. Max Stack**

*Design a max stack that supports push, pop, top, peekMax and popMax.*

*push(x) -- Push element x onto stack.*

*pop() -- Remove the element on top of the stack and return it.*

*top() -- Get the element on the top.*

*peekMax() -- Retrieve the maximum element in the stack.*

*popMax() -- Retrieve the maximum element in the stack, and remove it. If you find more than one maximum elements, only remove the top-most one.*

class MaxStack:

    def \_\_init\_\_(self):

        self.stack = []

        self.maxHeap = []

        self.toPop\_heap = {}  *# to keep track of things to remove from the heap*

        self.toPop\_stack = set()  *# to keep track of things to remove from the stack*

    def push(self, x):

        heapq.heappush(self.maxHeap, (-x,-len(self.stack)))

        self.stack.append(x)

    def pop(self):

        self.top()

        x = self.stack.pop()

        key = (-x,-len(self.stack))

        self.toPop\_heap[key] = self.toPop\_heap.get(key,0) + 1

        return x

    def top(self):

        while self.stack and len(self.stack)-1 in self.toPop\_stack:

            x = self.stack.pop()

            self.toPop\_stack.remove(len(self.stack))

        return self.stack[-1]

    def peekMax(self):

        while self.maxHeap and self.toPop\_heap.get(self.maxHeap[0],0):

            x = heapq.heappop(self.maxHeap)

            self.toPop\_heap[x] -= 1

        return -self.maxHeap[0][0]

    def popMax(self):

        self.peekMax()

        x,idx = heapq.heappop(self.maxHeap)

        x,idx = -x,-idx

        self.toPop\_stack.add(idx)

        return x

**158. Read N Characters Given Read4 II - Call multiple times**

*The API: int read4(char \*buf) reads 4 characters at a time from a file. The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file. By using the read4 API, implement the function int read(char \*buf, int n) that reads n characters from the file.*

class Solution(object):

    def \_\_init\_\_(self):

        self.queue = []

    def read(self, buf, n):

        idx = 0

        while True:

            buf4 = [""]\*4

            l = read4(buf4)

            self.queue.extend(buf4)

            curr = min(len(self.queue), n-idx)

            for i in range(curr):

                buf[idx] = self.queue.pop(0)

                idx+=1

            if curr == 0: break

        return idx

**22. Generate Parentheses**

*Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.*

def generateParenthesis(self, n):  ***# 4^n / sqrt(n)***

    res = []

    def generate(pre, left, right):

        if left: generate(pre+'(', left-1, right)

        if right > left: generate(pre+')', left, right-1)

        if not right: res.append(pre)

    generate('', n, n)

    return res

**246. Strobogrammatic Number**

*A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down). Write a function to determine if a number is strobogrammatic. The number is represented as a string.*

def isStrobogrammatic(self, num):

    return all(num[i] + num[~i] in '696 00 11 88' for i in range(len(num)//2+1))

**247. Strobogrammatic Number II**

*Find all strobogrammatic numbers that are of length = n.*

def findStrobogrammatic(self, n):

    evenMidCandidate = ["11","69","88","96", "00"]

    oddMidCandidate = ["0", "1", "8"]

    if n == 1: return oddMidCandidate

    if n == 2: return evenMidCandidate[:-1]

    if n % 2:

        pre, midCandidate = self.findStrobogrammatic(n-1), oddMidCandidate

    else:

        pre, midCandidate = self.findStrobogrammatic(n-2), evenMidCandidate

    premid = (n-1)//2

    return [p[:premid] + c + p[premid:] for c in midCandidate for p in pre]

**146. LRU Cache**

*Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and put.*

*get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.*

*put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.*

class Node:

    def \_\_init\_\_(self, k, v):

        self.key = k

        self.val = v

        self.prev = None

        self.next = None

class LRUCache:

    def \_\_init\_\_(self, capacity):

        self.capacity = capacity

        self.dic = dict()

        self.head = Node(0, 0)

        self.tail = Node(0, 0)

        self.head.next = self.tail

        self.tail.prev = self.head

    def \_add(self, node):

        p = self.tail.prev

        p.next = node

        self.tail.prev = node

        node.prev = p

        node.next = self.tail

 def \_remove(self, node):

        p = node.prev

        n = node.next

        p.next = n

        n.prev = p

    def get(self, key):

        if key in self.dic:

            n = self.dic[key]

            self.\_remove(n)

            self.\_add(n)

            return n.val

        return -1

    def put(self, key, value):

        if key in self.dic:

            self.\_remove(self.dic[key])

        n = Node(key, value)

        self.\_add(n)

        self.dic[key] = n

        if len(self.dic) > self.capacity:

            n = self.head.next

            self.\_remove(n)

            del self.dic[n.key]

**359. Logger Rate Limiter**

*Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is not printed in the last 10 seconds. Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false. It is possible that several messages arrive roughly at the same time.*

class Logger:

    def \_\_init\_\_(self):

        self.ok = {}

    def shouldPrintMessage(self, timestamp, message):

        if timestamp < self.ok.get(message, 0):

            return False

        self.ok[message] = timestamp + 10

        return True

**274. H-Index**

*Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index. According to the definition of h-index on Wikipedia: "A scientist has index h if h of his/her N papers have at least h citations each, and the other N − h papers have no more than h citations each."*

def hIndex(self, citations):  ***# O(N), O(N)***

    n = len(citations)

    citeCount = [0] \* (n+1)

    for c in citations:

        if c >= n:

            citeCount[n] += 1

        else:

            citeCount[c] += 1

    i = n-1

    while i >= 0:

        citeCount[i] += citeCount[i+1]

        if citeCount[i+1] >= i+1:

            return i+1

        i -= 1

    return 0

def hIndex(self, citations): ***# O(NLogN) O(1)***

    citations.sort()

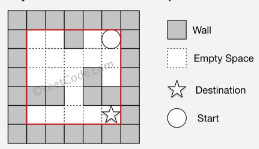
    n = len(citations)

    for i in xrange(n):

        if citations[i] >= (n-i):

            return n-i

    return 0



*left -> down -> left -> down -> right -> down -> right*

**490. The Maze**

*There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction. Given the ball's start position, the destination and the maze, determine whether the ball could stop at the destination. The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.*

def hasPath(self, maze, start, destination):

    q = [start]

    n, m = len(maze), len(maze[0])

    dirs = ((0, 1), (0, -1), (1, 0), (-1, 0))

    while q:

        i, j = q.pop(0)

        maze[i][j] = 2

        if i == destination[0] and j == destination[1]: return True

        for x, y in dirs:

            row, col = i + x,  j + y

            while 0 <= row < n and 0 <= col < m and maze[row][col] != 1:

                row += x

                col += y

            row -= x

            col -= y

            if maze[row][col] == 0:

                q.append([row, col])

    return False

**505. The Maze II**

*There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction. Given the ball's start position, the destination and the maze, find the shortest distance for the ball to stop at the destination. The distance is defined by the number of empty spaces traveled by the ball from the start position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.*

def shortestDistance(self, maze, start, destination):

    res, dest = None, tuple(destination)

    m, n = len(maze), len(maze[0])

    def go(start, direction): *# return the stop position and length*

        i, j = start

        ii, jj = direction

        l=0

        while 0<=i+ii<m and 0<=j+jj<n and maze[i+ii][j+jj]!=1:

            i+=ii

            j+=jj

            l+=1

        return l, (i,j)

    visited={}

    q=[]

    heapq.heappush(q, (0, tuple(start)))

    while q:

        length, cur = heapq.heappop(q)

        if cur in visited and visited[cur]<=length:

            continue

        visited[cur]=length

        if cur == dest:

            return length

        for direction in [(-1, 0), (1, 0), (0,-1), (0,1)]:

            l, np = go(cur, direction)

            heapq.heappush(q, (length+l, np))

    return -1

**299. Bulls and Cows**

*You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number. Write a function to return a hint according to the secret number and friend's guess, use A to indicate the bulls and B to indicate the cows.  Please note that both secret number and friend's guess may contain duplicate digits.*

*Input: secret = "1807", guess = "7810"   Output: "1A3B"*

*Explanation: 1 bull and 3 cows. The bull is 8, the cows are 0, 1 and 7.*

def getHint(self, secret, guess):

    bulls = sum(map(operator.eq, secret, guess))

    both = sum(min(secret.count(x), guess.count(x)) for x in set(guess))

    return '{:d}A{:d}B'.format(bulls, both - bulls)

**756. Pyramid Transition Matrix**

*We are stacking blocks to form a pyramid. Each block has a color which is a one letter string, like `'Z'`. For every block of color `C` we place not in the bottom row, we are placing it on top of a left block of color `A` and right block of color `B`. We are allowed to place the block there only if `(A, B, C)` is an allowed triple. We start with a bottom row of bottom, represented as a single string. We also start with a list of allowed triples allowed. Each allowed triple is represented as a string of length 3. Return true if we can build the pyramid all the way to the top, otherwise false.*

def pyramidTransition(self, bottom, allowed):

    pool = collections.defaultdict(list)

    for x in allowed: pool[x[:2]].append(x[2])

    def dfs(bottom):

        if len(bottom)==1: return True

        for b in itertools.product(\*(pool[x+y] for x,y in zip(bottom[:-1],bottom[1:]))):

            if dfs(b): return True

        return False

    return dfs(bottom)

**380. Insert Delete GetRandom O(1)**

*Design a data structure that supports all following operations in average O(1) time.*

*insert(val): Inserts an item val to the set if not already present.*

*remove(val): Removes an item val from the set if present.*

*getRandom: Returns a random element from current set. Each element must have the same probability of being returned.*

import random

class RandomizedSet(object):

    def \_\_init\_\_(self):

        self.nums, self.pos = [], {}

    def insert(self, val):

        if val not in self.pos:

            self.nums.append(val)

            self.pos[val] = len(self.nums) - 1

            return True

        return False

    def remove(self, val):

        if val in self.pos:

            idx, last = self.pos[val], self.nums[-1]

            self.nums[idx], self.pos[last] = last, idx

            self.nums.pop(); self.pos.pop(val, 0)

            return True

        return False

    def getRandom(self):

        return self.nums[random.randint(0, len(self.nums) - 1)]

**379. Design Phone Directory**

*Design a Phone Directory which supports the following operations:*

*get: Provide a number which is not assigned to anyone.*

*check: Check if a number is available or not.*

*release: Recycle or release a number.*

class PhoneDirectory:

    def \_\_init\_\_(self, maxNumbers):

        self.available = set(range(maxNumbers))

    def get(self):

        return self.available.pop() if self.available else -1

    def check(self, number):

        return number in self.available

    def release(self, number):

        self.available.add(number)

**382. Linked List Random Node**

*Given a singly linked list, return a random node's value from the linked list. Each node must have the same probability of being chosen.*

*Follow up: What if the linked list is extremely large and its length is unknown to you? Could you solve this efficiently without using extra space?*

class Solution:

    def \_\_init\_\_(self, head):

        self.head = head

    def getRandom(self):

        node = self.head

        before = 0

        buffer = [None] \* 100

        while node:

            now = 0

            while node and now < 100:

                buffer[now] = node

                node = node.next

                now += 1

            r = random.randrange(now + before)

            if r < now:

                pick = buffer[r]

            before += now

        return pick.val

**421. Maximum XOR of Two Numbers in an Array**

*Given a non-empty array of numbers, a0, a1, a2, … , an-1, where 0 ≤ ai < 231. Find the maximum result of ai XOR aj.*

***# Build the answer bit by bit from left to right. Let's say we already know the largest first seven bits we can create. How to find the largest first eight bits we can create? Well it's that maximal seven-bits prefix followed by 0 or 1. Append 0 and then try to create the 1 one (i.e., answer ^ 1) from two eight-bits prefixes from nums. If we can, change that 0 to 1.***

def findMaximumXOR(self, nums):

    answer = 0

    for i in range(32)[::-1]:

        answer <<= 1

        prefixes = {num >> i for num in nums}

        answer += any(answer^1 ^ p in prefixes for p in prefixes)

    return answer

**483. Smallest Good Base**

*For an integer n, we call k>=2 a good base of n, if all digits of n base k are 1. Now given a string representing n, you should return the smallest good base of n in string format.*

def smallestGoodBase(self, n):

    n = int(n)

    max\_m = int(math.log(n,2)) *# Refer [7]*

    for m in range(max\_m,1,-1):

        k = int(n\*\*m\*\*-1)  *# Refer [6]*

        if (k\*\*(m+1)-1) // (k-1) == n:

            return str(k)  *# Refer [3]*

    return str(n-1)

**726. Number of Atoms**

*Given a chemical formula (given as a string), return the count of each atom. Input:  formula = "Mg(OH)2" Output: "H2MgO2"*

def countOfAtoms(self, formula):

    i, N, stack = 0, len(formula), [collections.Counter()]

    while i < N:

        if formula[i] == '(':

            stack.append(collections.Counter())

            i += 1

        elif formula[i] == ')':

            top = stack.pop()

            i += 1

            i\_start = i

            while i < N and formula[i].isdigit(): i += 1

            multiplicity = int(formula[i\_start: i] or 1)

            for name, v in top.items():

                stack[-1][name] += v \* multiplicity

        else:

            i\_start = i

            i += 1

            while i < N and formula[i].islower(): i += 1

            name, i\_start = formula[i\_start: i], i

            while i < N and formula[i].isdigit(): i += 1

            multiplicity = int(formula[i\_start: i] or 1)

            stack[-1][name] += multiplicity

    return *"".join(name + (str(stack[-1][name]) if stack[-1][name] > 1 else '') for name in sorted(stack[-1]))*

**729. My Calendar I**

*Implement a MyCalendar class to store your events. A new event can be added if adding the event will not cause a double booking. Your class will have the method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end. A double booking happens when two events have some non-empty intersection (ie., there is some time that is common to both events.) For each call to the method MyCalendar.book, return true if the event can be added to the calendar successfully without causing a double booking. Otherwise, return false and do not add the event to the calendar.*

class MyCalendar:

    def \_\_init\_\_(self):

        self.calendar = []

    def book(self, start, end):

        for s, e in self.calendar:

            if s < end and start < e:

                return False

        self.calendar.append((start, end))

        return True

**731. My Calendar II**

*Same to Calendar I, but a new event can be added if adding the event will not cause a* ***triple*** *booking.*

class MyCalendarTwo:

    def \_\_init\_\_(self):

        self.calendar = []

        self.overlaps = []

    def book(self, start, end):

        for i, j in self.overlaps:

            if start < j and end > i:

                return False

        for i, j in self.calendar:

            if start < j and end > i:

                self.overlaps.append((max(start, i), min(end, j)))

        self.calendar.append((start, end))

        return True

**732. My Calendar III**

*A K-booking happens when K events have some non-empty intersection.*

class MyCalendarThree: *# O(N^2)  O(N)*

    def \_\_init\_\_(self):

        self.delta = collections.Counter()

    def book(self, start, end):

        self.delta[start] += 1

        self.delta[end] -= 1

        active = ans = 0

        for x in sorted(self.delta):

            active += self.delta[x]

            if active > ans: ans = active

        return ans

**739. Daily Temperatures**

*Given a list of daily temperatures, produce a list that, for each day in the input, tells you how many days you would have to wait until a warmer temperature. If there is no future day for which this is possible, put 0 instead. For example, given the list temperatures = [73, 74, 75, 71, 69, 72, 76, 73], your output should be [1, 1, 4, 2, 1, 1, 0, 0]. Note: The length of temperatures will be in the range [1, 30000]. Each temperature will be an integer in the range [30, 100].*

def dailyTemperatures(self, temperatures):  *# O(N) O(W)*

    ans = [0] \* len(T) ***# T is temperatures***

    stack = [] #indexes from hottest to coldest

    for i in range(len(T) - 1, -1, -1):

        while stack and T[i] >= T[stack[-1]]:

            stack.pop()

        if stack:

            ans[i] = stack[-1] - i

        stack.append(i)

    return ans

**387. First Unique Character in a String**

*Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.*

def firstUniqChar(self, s):

    letters='abcdefghijklmnopqrstuvwxyz'

    index=[s.index(l) for l in letters if s.count(l) == 1]

    return min(index) if len(index) > 0 else -1

**748. Shortest Completing Word**

*Find the minimum length word from a given dictionary words, which has all the letters from the string licensePlate. Such a word is said to complete the given string licensePlate. Here, for letters we ignore case. For example, "P" on the licensePlate still matches "p" on the word. It is guaranteed an answer exists. If there are multiple answers, return the one that occurs first in the array. The license plate might have the same letter occurring multiple times. For example, given a licensePlate of "PP", the word "pair" does not complete the licensePlate, but the word "supper" does.*

def shortestCompletingWord(self, licensePlate, words):

    def count(itera):

        ans = [0] \* 26

        for letter in itera:

            ans[ord(letter) - ord('a')] += 1

        return ans

    def dominates(c1, c2):

        return all(x1 >= x2 for x1, x2 in zip(c1, c2))

    ans = None

    target = count(c.lower() for c in licensePlate if c.isalpha())

    for word in words:

        if (not ans or len(word) < len(ans)) and dominates(count(word.lower()), target):

            ans = word

    return ans

**528. Random Pick with Weight**

*Given an array w of positive integers, where w[i] describes the weight of index i, write a function pickIndex which randomly picks an index in proportion to its weight.*

class Solution:

    def \_\_init\_\_(self, w):

        self.w = w

        self.n = len(w)

        self.s = sum(self.w)

        for i in range(1,self.n):

            w[i] += w[i-1]

    def pickIndex(self):

        seed = random.randint(1,self.s)

        l,r = 0, self.n-1

        while l<r:

            mid = (l+r)//2

            if seed <= self.w[mid]:

                r = mid

            else:

                l = mid+1

        return l

**529. Minesweeper**

*Let's play the minesweeper game (Wikipedia, online game)!*

*You are given a 2D char matrix representing the game board. 'M' represents an unrevealed mine, 'E' represents an unrevealed empty square, 'B' represents a revealed blank square that has no adjacent (above, below, left, right, and all 4 diagonals) mines, digit ('1' to '8') represents how many mines are adjacent to this revealed square, and finally 'X' represents a revealed mine.*

*Now given the next click position (row and column indices) among all the unrevealed squares ('M' or 'E'), return the board after revealing this position according to the following rules:*

*If a mine ('M') is revealed, then the game is over - change it to 'X'.*

*If an empty square ('E') with no adjacent mines is revealed, then change it to revealed blank ('B') and all of its adjacent unrevealed squares should be revealed recursively.*

*If an empty square ('E') with at least one adjacent mine is revealed, then change it to a digit ('1' to '8').*

*Return the board when no more squares will be revealed.*

def updateBoard(self, board, click):

    (row, col), directions = click, ((-1, 0), (1, 0), (0, 1), (0, -1), (-1, 1), (-1, -1), (1, 1), (1, -1))

    if 0 <= row < len(board) and 0 <= col < len(board[0]):

        if board[row][col] == 'M':

            board[row][col] = 'X'

        elif board[row][col] == 'E':

            n = sum([board[row + r][col + c] == 'M' for r, c in directions if 0 <= row + r < len(board) and 0 <= col + c < len(board[0])])

            board[row][col] = str(n or 'B')

            for r, c in directions \* (not n): self.updateBoard(board, [row + r, col + c])

    return board

**572. Subtree of Another Tree**

*Given two non-empty binary trees s and t, check whether tree t has exactly the same structure and node values with a subtree of s. A subtree of s is a tree consists of a node in s and all of this node's descendants. The tree s could also be considered as a subtree of itself.*

def isMatch(self, s, t):

    if not(s and t):

        return s is t

    return (s.val == t.val and

            self.isMatch(s.left, t.left) and

            self.isMatch(s.right, t.right))

def isSubtree(self, s, t):

    if self.isMatch(s, t): return True

    if not s: return False

    return self.isSubtree(s.left, t) or self.isSubtree(s.right, t)

**632. Smallest Range**

*You have k lists of sorted integers in ascending order. Find the smallest range that includes at least one number from each of the k lists. We define the range [a,b] is smaller than range [c,d] if b-a < d-c or a < c if b-a == d-c.*

*Input:[[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]*

*Output: [20,24]*

*Explanation:*

*List 1: [4, 10, 15, 24,26], 24 is in range [20,24].*

*List 2: [0, 9, 12, 20], 20 is in range [20,24].*

*List 3: [5, 18, 22, 30], 22 is in range [20,24].*

def smallestRange(self, A):

    pq = [(row[0], i, 0) for i, row in enumerate(A)]

    heapq.heapify(pq)

    ans = -1e9, 1e9

    right = max(row[0] for row in A)

    while pq:

        left, i, j = heapq.heappop(pq)

        if right - left < ans[1] - ans[0]:

            ans = left, right

        if j + 1 == len(A[i]):

            return ans

        v = A[i][j+1]

        right = max(right, v)

        heapq.heappush(pq, (v, i, j+1))

**694. Number of Distinct Islands**

*Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.*

*Count the number of distinct islands. An island is considered to be the same as another if and only if one island can be translated (and not rotated or reflected) to equal the other.*

def numDistinctIslands(self, grid):

    seen = set()

    def explore(r, c, r0, c0):

        if (0 <= r < len(grid) and 0 <= c < len(grid[0]) and

                grid[r][c] and (r, c) not in seen):

            seen.add((r, c))

            shape.add((r - r0, c - c0))

            explore(r+1, c, r0, c0)

            explore(r-1, c, r0, c0)

            explore(r, c+1, r0, c0)

            explore(r, c-1, r0, c0)

    shapes = set()

    for r in range(len(grid)):

        for c in range(len(grid[0])):

            shape = set()

            explore(r, c, r, c)

            if shape:

                shapes.add(frozenset(shape))

    return len(shapes)

**701. Insert into a Binary Search Tree**

*Given the root node of a binary search tree (BST) and a value to be inserted into the tree, insert the value into the BST. Return the root node of the BST after the insertion. It is guaranteed that the new value does not exist in the original BST.*

*Note that there may exist multiple valid ways for the insertion, You can return any of them.*

def insertIntoBST(self, root, val):

    if root == None:

        return TreeNode(val);

    if root.val < val:

        root.right = self.insertIntoBST(root.right, val);

    else:

        root.left = self.insertIntoBST(root.left, val);

    return root

**715. Range Module**

*A Range Module is a module that tracks ranges of numbers. Your task is to design and implement the following interfaces in an efficient manner.*

*addRange(int left, int right) Adds the half-open interval [left, right), tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval [left, right) that are not tracked.*

*queryRange(int left, int right) Returns true if and only if every real number in the interval [left, right) is currently being tracked.*

*removeRange(int left, int right) Stops tracking every real number currently being tracked in the interval [left, right).*

class RangeModule(object):

    def \_\_init\_\_(self):

        self.X = [0, 10\*\*9]

        self.track = [False] \* 2

    def addRange(self, left, right, track=True):

        def index(x):

            i = bisect.bisect\_left(self.X, x)

            if self.X[i] != x:

                self.X.insert(i, x)

                self.track.insert(i, self.track[i-1])

            return i

        i = index(left)

        j = index(right)

        self.X[i:j] = [left]

        self.track[i:j] = [track]

    def queryRange(self, left, right):

        i = bisect.bisect(self.X, left) - 1

        j = bisect.bisect\_left(self.X, right)

        return all(self.track[i:j])

    def removeRange(self, left, right):

        self.addRange(left, right, False)

**721. Accounts Merge**

*Given a list accounts, each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account.*

*Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some email that is common to both accounts. Even if two accounts have the same name, they may belong to different people as people may have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.*

*After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.*

DFS or Union-Find

**720. Longest Word in Dictionary**

*Given a list of strings words representing an English Dictionary, find the longest word in words that can be built one character at a time by other words in words. If there is more than one possible answer, return the longest word with the smallest lexicographical order. If there is no answer, return the empty string.*

def longestWord(self, words):

    wordset = set(words)

    words.sort(key = lambda c: (-len(c), c))

    for word in words:

        if all(word[:k] in wordset for k in xrange(1, len(word))):

            return word

    return ""

Or Trie + BFS/DFS O(sum(w))

**722. Remove Comments**

*Given a C++ program, remove comments from it. The program source is an array where source[i] is the i-th line of the source code. This represents the result of splitting the original source code string by the newline character \n.*

*In C++, there are two types of comments, line comments, and block comments.*

*The string // denotes a line comment, which represents that it and rest of the characters to the right of it in the same line should be ignored.*

*The string /\* denotes a block comment, which represents that all characters until the next (non-overlapping) occurrence of \*/ should be ignored. (Here, occurrences happen in reading order: line by line from left to right.) To be clear, the string /\*/ does not yet end the block comment, as the ending would be overlapping the beginning.*

*The first effective comment takes precedence over others: if the string // occurs in a block comment, it is ignored. Similarly, if the string /\* occurs in a line or block comment, it is also ignored.*

*If a certain line of code is empty after removing comments, you must not output that line*

*There will be no control characters, single quote, or double quote characters. For example, source = "string s = "/\* Not a comment. \*/";" will not be a test case. (Also, nothing else such as defines or macros will interfere with the comments.)*

*It is guaranteed that every open block comment will eventually be closed, so /\* outside of a line or block comment always starts a new comment.*

*Finally, implicit newline characters can be deleted by block comments. Please see the examples below for details.*

*After removing the comments from the source code, return the source code in the same format.*

def removeComments(self, source):

    in\_block = False

    ans = []

    for line in source:

        i = 0

        if not in\_block:

            newline = []

        while i < len(line):

            if line[i:i+2] == '/\*' and not in\_block:

                in\_block = True

                i += 1

            elif line[i:i+2] == '\*/' and in\_block:

                in\_block = False

                i += 1

            elif not in\_block and line[i:i+2] == '//':

                break

            elif not in\_block:

                newline.append(line[i])

            i += 1

        if newline and not in\_block:

            ans.append("".join(newline))

    return ans

**725. Split Linked List in Parts**

*Given a (singly) linked list with head node root, write a function to split the linked list into k consecutive linked list "parts".*

*The length of each part should be as equal as possible: no two parts should have a size differing by more than 1. This may lead to some parts being null.*

*The parts should be in order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal parts occurring later.*

*Return a List of ListNode's representing the linked list parts that are formed.*

*Examples 1->2->3->4, k = 5 // 5 equal parts [ [1], [2], [3], [4], null ]*

def splitListToParts(self, root, k):

    cur = root

    for N in xrange(1001):

        if not cur: break

        cur = cur.next

    width, remainder = divmod(N, k)

    ans = []

    cur = root

    for i in xrange(k):

        head = cur

        for j in xrange(width + (i < remainder) - 1):

            if cur: cur = cur.next

        if cur:

            cur.next, cur = None, cur.next

        ans.append(head)

    return ans

**750. Number Of Corner Rectangles**

*Given a grid where each entry is only 0 or 1, find the number of corner rectangles.*

*A corner rectangle is 4 distinct 1s on the grid that form an axis-aligned rectangle. Note that only the corners need to have the value 1. Also, all four 1s used must be distinct.*

class Solution(object):

    def countCornerRectangles(self, grid):

        rows = [[c for c, val in enumerate(row) if val]

                for row in grid]

        N = sum(sum(row) for row in grid)

        SQRTN = int(N\*\*.5)

        ans = 0

        count = collections.Counter()

        for r, row in enumerate(rows):

            if len(row) >= SQRTN:

                target = set(row)

                for r2, row2 in enumerate(rows):

                    if r2 <= r and len(row2) >= SQRTN:

                        continue

                    found = sum(1 for c2 in row2 if c2 in target)

                    ans += found \* (found - 1) / 2

            else:

                for pair in itertools.combinations(row, 2):

                    ans += count[pair]

                    count[pair] += 1

        return ans

**745. Prefix and Suffix Search**

*Given many words, words[i] has weight i.*

*Design a class WordFilter that supports one function, WordFilter.f(String prefix, String suffix). It will return the word with given prefix and suffix with maximum weight. If no word exists, return -1.*

*Examples: Input:*

*WordFilter(["apple"])*

*WordFilter.f("a", "e") // returns 0*

*WordFilter.f("b", "") // returns -1*

Trie = lambda: collections.defaultdict(Trie)

WEIGHT = False

class WordFilter(object):

    def \_\_init\_\_(self, words):

        self.trie = Trie()

        for weight, word in enumerate(words):

            cur = self.trie

            cur[WEIGHT] = weight

            for i, x in enumerate(word):

                #Put all prefixes and suffixes

                tmp = cur

                for letter in word[i:]:

                    tmp = tmp[letter, None]

                    tmp[WEIGHT] = weight

                tmp = cur

                for letter in word[:-i or None][::-1]:

                    tmp = tmp[None, letter]

                    tmp[WEIGHT] = weight

                #Advance letters

                cur = cur[x, word[~i]]

                cur[WEIGHT] = weight

    def search(self, prefix, suffix):

        cur = self.trie

        for a, b in map(None, prefix, suffix[::-1]):

            if (a, b) not in cur: return -1

            cur = cur[a, b]

        return cur[WEIGHT]

**767. Reorganize String**

*Given a string S, check if the letters can be rearranged so that two characters that are adjacent to each other are not the same.*

*If possible, output any possible result.  If not possible, return the empty string.*

def reorganizeString(self, S):

        N = len(S)

        A = []

        for c, x in sorted((S.count(x), x) for x in set(S)):

            if c > (N+1)/2: return ""

            A.extend(c \* x)

        ans = [None] \* N

        ans[::2], ans[1::2] = A[N/2:], A[:N/2]

        return "".join(ans)

**785. Is Graph Bipartite?**

*Given an undirected graph, return true if and only if it is bipartite.*

*Recall that a graph is bipartite if we can split it's set of nodes into two independent subsets A and B such that every edge in the graph has one node in A and another node in B.*

*The graph is given in the following form: graph[i] is a list of indexes j for which the edge between nodes i and j exists.  Each node is an integer between 0 and graph.length - 1. There are no self edges or parallel edges: graph[i] does not contain i, and it doesn't contain any element twice.*

def isBipartite(self, graph):

    color = {}

    for node in xrange(len(graph)):

        if node not in color:

            stack = [node]

            color[node] = 0

            while stack:

                node = stack.pop()

                for nei in graph[node]:

                    if nei not in color:

                        stack.append(nei)

                        color[nei] = color[node] ^ 1

                    elif color[nei] == color[node]:

                        return False

    return True

**806. Number of Lines To Write String**

*We are to write the letters of a given string S, from left to right into lines. Each line has maximum width 100 units, and if writing a letter would cause the width of the line to exceed 100 units, it is written on the next line. We are given an array widths, an array where widths[0] is the width of 'a', widths[1] is the width of 'b', ..., and widths[25] is the width of 'z'.*

*Now answer two questions: how many lines have at least one character from S, and what is the width used by the last such line? Return your answer as an integer list of length 2.*

def numberOfLines(self, widths, S):

    lines, width = 1, 0

    for c in S:

        w = widths[ord(c) - ord('a')]

        width += w

        if width > 100:

            lines += 1

            width = w

    return lines, width

**807. Max Increase to Keep City Skyline**

*In a 2 dimensional array grid, each value grid[i][j] represents the height of a building located there. We are allowed to increase the height of any number of buildings, by any amount (the amounts can be different for different buildings). Height 0 is considered to be a building as well.*

*At the end, the "skyline" when viewed from all four directions of the grid, i.e. top, bottom, left, and right, must be the same as the skyline of the original grid. A city's skyline is the outer contour of the rectangles formed by all the buildings when viewed from a distance. See the following example.*

*What is the maximum total sum that the height of the buildings can be increased?*

def maxIncreaseKeepingSkyline(self, grid):

    row\_maxes = [max(row) for row in grid]

    col\_maxes = [max(col) for col in zip(\*grid)]

    return sum(min(row\_maxes[r], col\_maxes[c]) - val

               for r, row in enumerate(grid)

               for c, val in enumerate(row))

**850. Rectangle Area II**

*We are given a list of (axis-aligned) rectangles.  Each rectangle[i] = [x1, y1, x2, y2] , where (x1, y1) are the coordinates of the bottom-left corner, and (x2, y2) are the coordinates of the top-right corner of the ith rectangle.*

*Find the total area covered by all rectangles in the plane.  Since the answer may be too large, return it modulo 10^9 + 7.*

class Node(object):

    def \_\_init\_\_(self, start, end):

        self.start, self.end = start, end

        self.total = self.count = 0

        self.\_left = self.\_right = None

    @property

    def mid(self):

        return (self.start + self.end) / 2

    @property

    def left(self):

        self.\_left = self.\_left or Node(self.start, self.mid)

        return self.\_left

    @property

    def right(self):

        self.\_right = self.\_right or Node(self.mid, self.end)

        return self.\_right

    def update(self, i, j, val):

        if i >= j: return 0

        if self.start == i and self.end == j:

            self.count += val

        else:

            self.left.update(i, min(self.mid, j), val)

            self.right.update(max(self.mid, i), j, val)

        if self.count > 0:

            self.total = X[self.end] - X[self.start]

        else:

            self.total = self.left.total + self.right.total

        return self.total

class Solution(object):

    def rectangleArea(self, rectangles):

        OPEN, CLOSE = 1, -1

        events = []

        global X

        X = set()

        for x1, y1, x2, y2 in rectangles:

            events.append((y1, OPEN, x1, x2))

            events.append((y2, CLOSE, x1, x2))

            X.add(x1)

            X.add(x2)

        events.sort()

        X = sorted(X)

        Xi = {x: i for i, x in enumerate(X)}

        active = Node(0, len(X) - 1)

        ans = 0

        cur\_x\_sum = 0

        cur\_y = events[0][0]

        for y, typ, x1, x2 in events:

            ans += cur\_x\_sum \* (y - cur\_y)

            cur\_x\_sum = active.update(Xi[x1], Xi[x2], typ)

            cur\_y = y

        return ans % (10\*\*9 + 7)

**846. Hand of Straights**

*Alice has a hand of cards, given as an array of integers. Now she wants to rearrange the cards into groups so that each group is size W, and consists of W consecutive cards. Return true if and only if she can.*

def isNStraightHand(self, hand, W):

    c = collections.Counter(hand)

    start = collections.deque()

    last\_checked, opened = -1, 0

    for i in sorted(c):

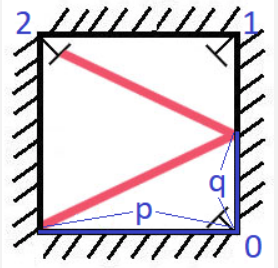
        if opened > c[i] or opened > 0 and i > last\_checked + 1: return False

        start.append(c[i] - opened)

        last\_checked, opened = i, c[i]

        if len(start) == W: opened -= start.popleft()

    return opened == 0

**858. Mirror Reflection**

*There is a special square room with mirrors on each of the four walls.  Except for the southwest corner, there are receptors on each of the remaining corners, numbered 0, 1, and 2.*

*The square room has walls of length p, and a laser ray from the southwest corner first meets the east wall at a distance q from the 0th receptor.*

*Return the number of the receptor that the ray meets first.  (It is guaranteed that the ray will meet a receptor eventually.)*

*Input: p = 2, q = 1*

*Output: 2*

*Explanation: The ray meets receptor 2 the first time it gets reflected back to the left wall.*

def mirrorReflection(self, p, q):

        from fractions import gcd

        g = gcd(p, q)

        p = (p / g) % 2

        q = (q / g) % 2

        return 1 if p and q else 0 if p else 2

def mirrorReflection(self, p, q):

        from fractions import Fraction as F

        x = y = 0

        rx, ry = p, q

        targets = [(p, 0), (p, p), (0, p)]

        while (x, y) not in targets:

            #Want smallest t so that some x + rx, y + ry is 0 or p

            #x + rxt = 0, then t = -x/rx etc.

            t = float('inf')

            for v in [F(-x,rx), F(-y,ry), F(p-x,rx), F(p-y,ry)]:

                if v > 0: t = min(t, v)

            x += rx \* t

            y += ry \* t

            #update rx, ry

            if x == p or x == 0: # bounced from east/west wall, so reflect on y axis

                rx \*= -1

            if y == p or y == 0:

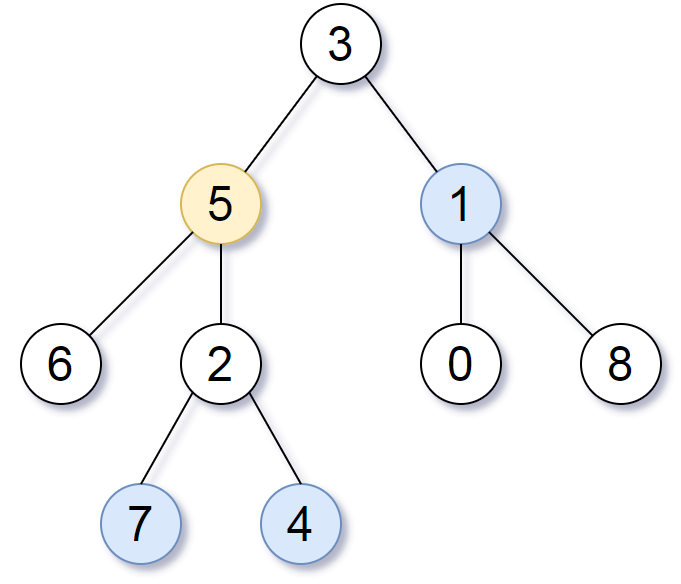
                ry \*= -1

        return 1 if x==y==p else 0 if x==p else 2

**863. All Nodes Distance K in Binary Tree**

*We are given a binary tree (with root node root), a target node, and an integer value K.*

*Return a list of the values of all nodes that have a distance K from the target node.  The answer can be returned in any order.*

*Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2*

*Output: [7,4,1]*

*Explanation:*

*The nodes that are a distance 2 from the target node (with value 5)*

*have values 7, 4, and 1.*

def distanceK(self, root, target, K):

    def dfs(node, par = None):

        if node:

            node.par = par

            dfs(node.left, node)

            dfs(node.right, node)

    dfs(root)

    queue = collections.deque([(target, 0)])

    seen = {target}

    while queue:

        if queue[0][1] == K:

            return [node.val for node, d in queue]

        node, d = queue.popleft()

        for nei in (node.left, node.right, node.par):

            if nei and nei not in seen:

                seen.add(nei)

                queue.append((nei, d+1))

    return []

**900. RLE Iterator**

*Write an iterator that iterates through a run-length encoded sequence.*

*The iterator is initialized by RLEIterator(int[] A), where A is a run-length encoding of some sequence.  More specifically, for all even i, A[i] tells us the number of times that the non-negative integer value A[i+1] is repeated in the sequence.*

*The iterator supports one function: next(int n), which exhausts the next n elements (n >= 1) and returns the last element exhausted in this way.  If there is no element left to exhaust, next returns -1 instead.*

*For example, we start with A = [3,8,0,9,2,5], which is a run-length encoding of the sequence [8,8,8,5,5].  This is because the sequence can be read as "three eights, zero nines, two fives".*

class RLEIterator(object):

    def \_\_init\_\_(self, A):

        self.A = A

        self.i = 0

        self.q = 0

    def next(self, n):

        while self.i < len(self.A):

            if self.q + n > self.A[self.i]:

                n -= self.A[self.i] - self.q

                self.q = 0

                self.i += 2

            else:

                self.q += n

                return self.A[self.i+1]

        return -1

**457. Circular Array Loop**

*You are given an array of positive and negative integers. If a number n at an index is positive, then move forward n steps. Conversely, if it's negative (-n), move backward n steps. Assume the first element of the array is forward next to the last element, and the last element is backward next to the first element. Determine if there is a loop in this array. A loop starts and ends at a particular index with more than 1 element along the loop. The loop must be "forward" or "backward'.*

*Example 1: Given the array [2, -1, 1, 2, 2], there is a loop, from index 0 -> 2 -> 3 -> 0.*

*Example 2: Given the array [-1, 2], there is no loop.*

*Note: The given array is guaranteed to contain no element "0".*

*Can you do it in O(n) time complexity and O(1) space complexity?*

def circularArrayLoop(self, nums):

    if not nums or len(nums) < 2:

        return False

    n = len(nums)

    for i in range(n):

        if type(nums[i]) != int: # visited element

            continue

        if nums[i] % n == 0: # self-loop

            continue

        direction = (nums[i] > 0) # loop direction, cannot be changed midway

        mark = str(i)

        while (type(nums[i]) == int) and (direction ^ (nums[i] < 0)) and (nums[i] % n != 0):

            jump = nums[i]

            nums[i] = mark

            i = (i + jump) % n

        if nums[i] == mark:

            return True

    return False

**424. Longest Repeating Character Replacement**

*Given a string that consists of only uppercase English letters, you can replace any letter in the string with another letter at most k times. Find the length of a longest substring containing all repeating letters you can get after performing the above operations.*

def characterReplacement(self, s, k):

    res = lo = hi = 0

    counts = collections.Counter()

    for hi in range(1, len(s)+1):

        counts[s[hi-1]] += 1

        max\_char\_n = counts.most\_common(1)[0][1]

        if hi - lo - max\_char\_n > k:

            counts[s[lo]] -= 1

            lo += 1

    return hi - lo

**85. Maximal Rectangle**

*Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.*

Example:

Input:                         Output: 6

[ ["1","0","1","0","0"],

  ["1","0","1","1","1"],

  ["1","1","1","1","1"],

  ["1","0","0","1","0"]]

**248. Strobogrammatic Number III**

*A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).*

*Write a function to count the total strobogrammatic numbers that exist in the range of low <= num <= high.*

*Example:*

*Input: low = "50", high = "100" Output: 3 Explanation: 69, 88, and 96 are three strobogrammatic numbers.*

def strobogrammaticInRange(self, low, high):

    maps={"0":"0","1":"1","6":"9","8":"8","9":"6"}

    cl,ch=len(low), len(high)

    if cl>ch or (cl==ch and low>high): return 0

    ans=["","0","1","8"]

    count=0

    while ans:

        tmp=[]

        for w in ans:

            if len(w)<ch or (len(w)==ch and w<=high):

                if len(w)>cl or (len(w)==cl and w>=low):

                    if len(w)>1 and w[0]=="0":  # leading zeros

                        pass

                    else:

                        count+=1

                if ch-len(w)>=2:

                    for key in maps:

                        res=key+w+maps[key]

                        tmp.append(res)

        ans=tmp

    return count

**428. Serialize and Deserialize N-ary Tree**

def serialize(self, root):

    serial = []

    def preorder(node):

        if not node: return

        serial.append(str(node.val))

        for child in node.children:

            preorder(child)

        serial.append("#")

    preorder(root)

    return " ".join(serial)

def deserialize(self, data):

    if not data: return None

    tokens = deque(data.split())

    root = Node(int(tokens.popleft()), [])

    def helper(node):

        if not tokens: return

        while tokens[0] != "#": # add child nodes with subtrees

            value = tokens.popleft()

            child = Node(int(value), [])

            node.children.append(child)

            helper(child)

        tokens.popleft()        # discard the "#"

    helper(root)

    return root

**489. Robot Room Cleaner**

*Given a robot cleaner in a room modeled as a grid.*

*Each cell in the grid can be empty or blocked.*

*The robot cleaner with 4 given APIs can move forward, turn left or turn right. Each turn it made is 90 degrees.*

*When it tries to move into a blocked cell, its bumper sensor detects the obstacle and it stays on the current cell.*

*Design an algorithm to clean the entire room using only the 4 given APIs shown below.*

class Solution:

    def cleanRoom(self, robot):

        def nxt(x, y, degree=0):

            if degree == 0: return x - 1, y

            if degree == 90: return x, y  + 1

            if degree == 180: return x + 1, y

            if degree == 270: return x, y - 1

        def go\_back():

            robot.turnLeft()

            robot.turnLeft()

            robot.move()

            robot.turnLeft()

            robot.turnLeft()

        visited = set()

        def dfs(x, y, cur):

            visited.add((x, y))

            robot.clean()

            for i in range(4): # 4 = 360 // 90

                nx, ny = nxt(x, y, cur)

                if (nx, ny) not in visited:

                    if robot.move():

                        dfs(nx, ny, cur)

                        go\_back()

                robot.turnRight()

                cur = (cur + 90) % 360

        dfs(0, 0, 0)

**156. Binary Tree Upside Down**

*Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.*

def upsideDownBinaryTree(self, root):

    if not root or not root.left:

        return root

    lRoot = self.upsideDownBinaryTree(root.left)

    rMost = lRoot

    while rMost.right:

        rMost = rMost.right

    root, rMost.left, rMost.right = lRoot, root.right, TreeNode(root.val)

    return root

**398. Random Pick Index**

*Given an array of integers with possible duplicates, randomly output the index of a given target number. You can assume that the given target number must exist in the array.*

*Note: The array size can be very large. Solution that uses too much extra space will not pass the judge.*

class Solution(object):

    def \_\_init\_\_(self, nums):

        self.nums = nums

    def pick(self, target):

        return random.choice([k for k, v in enumerate(self.nums) if v == target])

**871. Minimum Number of Refueling Stops**

*A car travels from a starting position to a destination which is target miles east of the starting position.  
Along the way, there are gas stations.  Each station[i] represents a gas station that is station[i][0] miles east of the starting position, and has station[i][1] liters of gas. The car starts with an infinite tank of gas, which initially has startFuel liters of fuel in it.  It uses 1 liter of gas per 1 mile that it drives.  
When the car reaches a gas station, it may stop and refuel, transferring all the gas from the station into the car.  
What is the least number of refueling stops the car must make in order to reach its destination?  If it cannot reach the destination, return -1.  
Note that if the car reaches a gas station with 0 fuel left, the car can still refuel there.  If the car reaches the destination with 0 fuel left, it is still considered to have arrived.*

def minRefuelStops(self, target, tank, stations):  
    pq = []  # A maxheap is simulated using negative values  
    stations.append((target, float('inf')))  
  
    ans = prev = 0  
    for location, capacity in stations:  
        tank -= location - prev  
        while pq and tank < 0:  # must refuel in past  
            tank += -heapq.heappop(pq)  
            ans += 1  
        if tank < 0: return -1  
        heapq.heappush(pq, -capacity)  
        prev = location  
  
    return ans

**117. Populating Next Right Pointers in Each Node II*Given a binary tree***

*struct TreeLinkNode { TreeLinkNode \*left; TreeLinkNode \*right; TreeLinkNode \*next;}*

*Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL. Initially, all next pointers are set to NULL.*

def connect(self, node):

    tail = dummy = TreeLinkNode(0)

    while node:

        tail.next = node.left

        if tail.next: tail = tail.next

        tail.next = node.right

        if tail.next: tail = tail.next

        node = node.next

        if not node:

            tail = dummy

            node = dummy.next

**70. Climbing Stairs**

*You are climbing a stair case. It takes n steps to reach to the top.*

*Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?*

# Top down - TLE

def climbStairs1(self, n):

    if n == 1:

        return 1

    if n == 2:

        return 2

    return self.climbStairs(n-1)+self.climbStairs(n-2)

# Bottom up, O(n) space

def climbStairs2(self, n):

    if n == 1:

        return 1

    res = [0 for i in xrange(n)]

    res[0], res[1] = 1, 2

    for i in xrange(2, n):

        res[i] = res[i-1] + res[i-2]

    return res[-1]

# Bottom up, constant space

def climbStairs3(self, n):

    if n == 1:

        return 1

    a, b = 1, 2

    for i in xrange(2, n):

        tmp = b

        b = a+b

        a = tmp

    return b

# Top down + memorization (list)

def climbStairs4(self, n):

    if n == 1:

        return 1

    dic = [-1 for i in xrange(n)]

    dic[0], dic[1] = 1, 2

    return self.helper(n-1, dic)

def helper(self, n, dic):

    if dic[n] < 0:

        dic[n] = self.helper(n-1, dic)+self.helper(n-2, dic)

    return dic[n]

# Top down + memorization (dictionary)

def \_\_init\_\_(self):

    self.dic = {1:1, 2:2}

def climbStairs(self, n):

    if n not in self.dic:

        self.dic[n] = self.climbStairs(n-1) + self.climbStairs(n-2)

    return self.dic[n]

**91. Decode Ways**

*A message containing letters from A-Z is being encoded to numbers using the following mapping:*

*'A' -> 1   'B' -> 2 ….  'Z' -> 26*

*Given a non-empty string containing only digits, determine the total number of ways to decode it.*

*Example 1:*

*Input: "12"  Output: 2 Explanation: It could be decoded as "AB" (1 2) or "L" (12).*

def numDecodings(self, s):

    if s == "": return 0

    dp = [0 for x in range(len(s)+1)]

    dp[0] = 1

    for i in range(1, len(s)+1):

        if s[i-1] != "0":

            dp[i] += dp[i-1]

        if i != 1 and s[i-2:i] < "27" and s[i-2:i] > "09":  #"01"ways = 0

            dp[i] += dp[i-2]

    return dp[len(s)]

**920. Number of Music Playlists**

*Your music player contains N different songs and she wants to listen to L (not necessarily different) songs during your trip.  You create a playlist so that:*

*Every song is played at least once*

*A song can only be played again only if K other songs have been played*

*Return the number of possible playlists.  As the answer can be very large, return it modulo 10^9 + 7.*

from functools import lru\_cache

class Solution:

    def numMusicPlaylists(self, N, L, K):

        @lru\_cache(None)

        def dp(i, j):

            if i == 0:

                return +(j == 0)

            ans = dp(i-1, j-1) \* (N-j+1)

            ans += dp(i-1, j) \* max(j-K, 0)

            return ans % (10\*\*9+7)

        return dp(L, N)

class Solution(object):

    def numMusicPlaylists(self, N, L, K):

        # dp[S] at time P = <S, P> as discussed in article

        dp = [1] \* (L-N+1)

        for p in xrange(2, N-K+1):

            for i in xrange(1, L-N+1):

                dp[i] += dp[i-1] \* p

        # Multiply by N!

        ans = dp[-1]

        for k in xrange(2, N+1):

            ans \*= k

        return ans % (10\*\*9 + 7)

**150. Evaluate Reverse Polish Notation**

*Evaluate the value of an arithmetic expression in Reverse Polish Notation.*

*Valid operators are +, -, \*, /. Each operand may be an integer or another expression.*

*Division between two integers should truncate toward zero.*

*The given RPN expression is always valid. That means the expression would always evaluate to a result and there won't be any divide by zero operation.*

def evalRPN(self, tokens):

    stack = []

    for t in tokens:

        if t not in ["+", "-", "\*", "/"]:

            stack.append(int(t))

        else:

            r, l = stack.pop(), stack.pop()

            if t == "+":

                stack.append(l+r)

            elif t == "-":

                stack.append(l-r)

            elif t == "\*":

                stack.append(l\*r)

            else:

                # here take care of the case like "1/-22",

                # in Python 2.x, it returns -1, while in

                # Leetcode it should return 0

                if l\*r < 0 and l % r != 0:

                    stack.append(l/r+1)

                else:

                    stack.append(l/r)

    return stack.pop()

**135. Candy**

*here are N children standing in a line. Each child is assigned a rating value. You are giving candies to these children subjected to the following requirements: Each child must have at least one candy. Children with a higher rating get more candies than their neighbors. What is the minimum candies you must give?*

def candy(self, ratings):

        if not ratings: return 0

        min\_rating, L = min(ratings), len(ratings)

        start\_points = []

        for i, r in enumerate(ratings):

            if i == 0 or i == L - 1:

                start\_points.append(i)

            elif ratings[i-1] >= r and ratings[i+1] >= r:

                start\_points.append(i)

        ans = [1] \* L

        for i in start\_points:

            for dx in (-1, 1):

                ni = i + dx

                if 0 <= ni < L and ratings[ni] > ratings[i]:

                    ans[ni] = max(ans[ni], ans[i] + 1)

                    start\_points.append(ni)

        return sum(ans)

**90. Subsets II**

*Given a collection of integers that might contain duplicates, nums, return all possible subsets (the power set). Note: The solution set must not contain duplicate subsets.*

def subsetsWithDup(self, S):

        res = [[]]

        S.sort()

        for i in range(len(S)):

            if i == 0 or S[i] != S[i - 1]:

                l = len(res)

            for j in range(len(res) - l, len(res)):

                res.append(res[j] + [S[i]])

        return res

**96. Unique Binary Search Trees**

*Given n, how many structurally unique BST's (binary search trees) that store values 1 ... n?*

def numTrees1(self, n):

    res = [0] \* (n+1)

    res[0] = 1

    for i in range(1, n+1):

        for j in range(i):

            res[i] += res[j] \* res[i-1-j]

    return res[n]

**736. Parse Lisp Expression**

*Input: (let x 2 (add (let x 3 (let x 4 x)) x))*

*Output: 6*

*Explanation: Even though (let x 4 x) has a deeper scope, it is outside the context*

*of the final x in the add-expression.  That final x will equal 2.*

def evaluate(self, expression):

    st, d, tokens = [], {}, ['']

    def getval(x):

        return d.get(x, x)

    def evaluate(tokens):

        if tokens[0] in ('add', 'mult'):

            tmp = map(int, map(getval, tokens[1:]))

            return str(tmp[0] + tmp[1] if tokens[0] == 'add' else tmp[0] \* tmp[1])

        else: #let

            for i in xrange(1, len(tokens)-1, 2):

                if tokens[i+1]:

                    d[tokens[i]] = getval(tokens[i+1])

            return getval(tokens[-1])

    for c in expression:

        if c == '(':

            if tokens[0] == 'let':

                evaluate(tokens)

            st.append((tokens, dict(d)))

            tokens =  ['']

        elif c == ' ':

            tokens.append('')

        elif c == ')':

            val = evaluate(tokens)

            tokens, d = st.pop()

            tokens[-1] += val

        else:

            tokens[-1] += c

    return int(tokens[0])