Amazon

HashTable(dictionary & set)

两个数组元素是否相同

如果没有重复元素：

先判断长度； 如果长度相同，把一个数组转化为hashset, 对另外一个数组进行遍历在hashset里查找，这样o(n)时间内可以完成。

如果有重复元素：用 HashMap 记录 (num, freq) 数组 A 的freq, 然后遍历另一个数组，相应 num 的 freq 减一，最后遍历 HashMap, 看所有的数的 freq 是否都是 0

1.Two Sum

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:

        d = {}
        for i, num in enumerate(nums):
            if target - num in d:
                return [i, d[target - num]]
            else:
                d[num] = i
        return []

        """
        Data Structure:
            d:  [key: num, val: index]
                store index of num of all previous nums

        Algorithm:
            iterate each num:
                (1)if target - num exists in d --> we have a complete pair
                        return [i, d[target - i]]
                (2)else not exist in d: --> put it into d

        """
```

14.Longest Common Prefix

```
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        res = ""
        for chars in zip(*strs):
            s = set(chars)
            if len(s) > 1:
                break
            else:
                res += chars[0]
        return res

        """
        Data Strucutre:
            1.zip(*strs): str --> list
             i      0  1  2  3  4  5

             0      f  l  o  w  e  r
             1      f  l  o  w
             2      f  l  i  g  h  t

        Algorithm:
            1. convert all strings into list and zip them together

            2. for each chars in one postion of zip:
                    (1) if num of distinct chars > 1:
                        this char cannot be in prefix
                        break

                    (2)else: if num of different chars is 1:
                        add it to prefix

            3. return prefix
        """
```

828.Count Unique Characters of All Substrings of a Given String

```
class Solution:
    def uniqueLetterString(self, s: str) -> int:
        d = {}
        for c in string.ascii_uppercase:
            d[c] = [-1, -1]

        res = 0
        for i, c in enumerate(s):
            i1, i2 = d[c]
            i3 = i
            res += (i3 - i2) * (i2 - i1)
            d[c] = [i2, i3]
        # after iteration
        for c in string.ascii_uppercase:
            i1, i2 = d[c]
            i3 = len(s)
            res += (i3 - i2) * (i2 - i1)

        return res
        """
            Explanation:
            A1 B C A2 C B A3

            how many substring can unique A2 show up

            Imagine to insert '(' and ')'

            --> times = number of '(' * number of ')'

            image now move to char C and we have last two postions of this Char
            d[C] = {i1, i2}, current index is i3

            In this way, we can set char C on pos i2 as unique char
            number of subarray where char C is unique == (i2 - i1) *(i3 - i2)

            After Iteration,
            we have d[C] = {p1, p2}
            char C on pos p2 has not been set unique yet
            so we now need to deal with it by setting i1 = p1, i2 = p2, i3 =
len(s)
            res += (i2 - i1) * (i3 - i2)

        Data Structure:
            1.  d[c] = {last but one pos, last pos}
                For cur A3,     A1           A2
                when we move to A3, we can compute how many times can A2 show up

        Algorithm:
            1.  initialize d[each letter] = [-1, -1]

            2.  for each c in s:
                    (1) (i1, i2), i3 = d[c], current index i
                    (2) res += (i2 - i1) * (i3 - i2)
                    (3) update last two positions:
                        d[c] = {i2, i3}

            3.  for each letter in d:
                    i1, i2 = d[letter]
                    i3 = n
                    res += (i2 - i1) * (i3 - i2)
        """
```

1152.Analyze User Website Visit Pattern

```
from collections import defaultdict
from itertools import combinations
class Solution:
    def mostVisitedPattern(self, username: List[str], timestamp: List[int],
website: List[str]) -> List[str]:
        A = sorted(zip(timestamp, username, website), key = lambda x : x[0])
        d1 = defaultdict(int)
        d2 = defaultdict(list)
        for t, u, w in A:
            d2[u] += [w]

        for ws in d2.values():
            ps = combinations(ws, 3)
            for p in set(ps):
                d1[p] += 1

        res = sorted(d1.keys(), key = lambda x : (-d1[x], x))
        return res[0]
        """
        Explanation:
            1.  return pattern with max freq(number of user)
                --> d[pattern] = number of users

            2.  d[user] = list of websites
                pattern = combiantions(websites, 3)
                --> de-duplicate --> set

            3.  zip(time, user, websites) sorted by time
                d[user] += [website]

        Data Structure:
            1. d1: key: pattern(tuple), val: number of users (int)
            2. d2: key: user(string) val: list of websites(list)

        Algorithm:
            1.  get zip(time, user, website) sorted by time

            2.  for each t, u, w in zip(,,,):
                    d2[u] += [w]

            3.  for each list of ws (values() of d2):
                    (1)patterns = set(combinations(list of ws, 3))
                    (2)for each pattern p
                            d1[p] += 1

            4.  sort pattern by (1) freq (2) lexicographically(if freq is equal)
                res = sorted(d1.keys(), key = lambda x : (-d1[x], x))
                return res[0]
        """
```

1481.Least Number of Unique Integers after K Removals

```
class Solution:
    def findLeastNumOfUniqueInts(self, arr: List[int], k: int) -> int:
        d = collections.Counter(arr)
        res = len(d)

        for freq in sorted(d.values()):
            if k > freq:
                res -= 1
                k -= freq
            elif k == freq:
                res -= 1
                return res
            else:
                return res
        return res
        """
        Data Structure:
            1.  d: {key: num, val: freq} sorted by freq

        Algorithm:
            1.  count freq of each num
            2.  extract values (freq) and sort it
            3.  total number of unique number
                for v in sorted values:
                    if freq < k: current number can be completely deleted and we
still need to delete other number
                        k -= freq
                        total -= 1
                    elif freq == k: current number can be completely deleted and
stop
                        total -= 1
                        return total
                    else: we cannot delete all of current numbers so stop
                        return total
        """
```

Union Find

1135.Connecting Cities With Minimum Cost

```
from heapq import heappop, heappush
class Solution:
    def minimumCost(self, n: int, connections: List[List[int]]) -> int:
        p = [0] * n
        for i in range(n):
            p[i] = i

        def find(node):
            if p[node] == node:
                return node
            p[node] = find(p[node])
            return p[node]

        h = []
        for a, b, c in connections:
            heappush(h, (c, a - 1, b - 1))

        res = 0
        while h:
            c, a, b = heappop(h)
            p1, p2 = find(a), find(b)
            if p1 == p2:
                continue
            else:
                res += c
                p[p1] = p2

        root = 0
        for i in range(n):
            if p[i] == i:
                root += 1
        return res if root == 1 else -1
        """
        Explanation:
            if we can reach any vertex from any vertex
            then this graph is connected

            for a graph of n vertex, we need at least n - 1 unique edges
            (if edges >= n, there must be circle)

            For this question, if we want to connect n cities
            we need n - 1 edges without circle

        Data Struture:
            1.  h : min heap [cost, a, b]

            2.  union find: parent, find

        Algorithm:
            1.  initialize h with each edges
            2.  pop out edges with min cost
                    p1, p2 = find(a), find(b)
                    (1)if p1 == p2:
                            a and b have been connected, they are in the same
tree
                            we do not use this edge
                    (2)if p1 != p2:
                            connect a and b
                            p[p1] = p2

            3.  how to judge whether this is only one tree?
                if p[node] == node: node is root
                check how many roots are in parent
        """
```

Binary Search

数组中和最接近0的两个数

binary search

TC: O(nlogn)

```python
def findclosest(arr):
    arr.sort()
    n = len(arr)
    res = [10000, 10000]
    for i in range(n):
        num = arr[i]
        l, r = 0, n - 1
        while l + 1 < r:
            m = l + (r - l) // 2
            total = num + arr[m]
            if total == 0 and m != i:
                return [arr[i], arr[m]]
            elif total < 0:
                l = m
            else:
                r = m
            #decide which is closer to 0 ? arr[i] + arr[l] / arr[i] + arr[l]
        i1 = i; i2 = -1
        if l == i or r == i:
            i2 = l if l != i else r
        else:
            i2 = l if abs(arr[i] + arr[l]) < abs(arr[i] + arr[r]) else r
        if abs(sum(res)) > abs(arr[i1] + arr[i2]):
            res = [arr[i1], arr[i2]]
    return res
```

```
arr = [-50, -40, 100, 200]
print(findclosest(arr))
```

33. Search in Rotated Sorted Array

image-20211003193453310

---

image-20211005144656432

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return mid
            if nums[mid] >= nums[left]:
                if target >= nums[left] and target <= nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                if target > nums[mid] and target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
        return -1
```

81.Search in Rotated Sorted Array II

本题含有重复元素

这个题目就比33. Search in Rotated Sorted Array的不含重复元素的题目多了一个去除某些重复元素的情况，当 nums[mid] == nums[left] 时，让 left += 1，并退出本次循环（continue），其余部分完全相同。

image-20211003220428657

---

image-20211003221205842

```
class Solution:
    def search(self, nums: List[int], target: int) -> bool:
        n = len(nums)
        left, right = 0, n - 1
        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return True
            if nums[left] == nums[mid]:
                left += 1
                continue
            if nums[mid] > nums[left]:
                if target >= nums[left] and target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                if target > nums[mid] and target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
        return False
```

注 Attention：nums[left] > nums[mid] 说明后半部分有序，反之前半部分有序。恰好相等的时候，你就不知道哪半部分有序了，所以需要 left ++，因为 nums[mid] == nums[left]，所以也不用害怕，left ++把需要找的数加没了。

Stack

907.Sum of Subarray Minimums

```
class Solution:
    def sumSubarrayMins(self, arr: List[int]) -> int:
        mod = 10 ** 9 + 7
        res = 0
        stack = []
        arr = [0] + arr + [0]

        for i in range(len(arr)):
            while stack and arr[stack[-1]] > arr[i]:
                i2 = stack.pop()
                i1 = stack[-1]
                i3 = i
                res += (i2 - i1) * (i3 - i2) * arr[i2]
            else:
                stack += [i]
        return res % mod
        """
        LC 84, 85, 907
        Explanation:
            1.
            j     <      i    >       k
            i1          i2            i3
            j is first num < i on the left
            k is first num < i on the right

            num of subarray whose minimum is i --> (i - j) * (k - i)

            2.stack
                stack[num1, num2, num3...]
                num1 is first num < num2
                num2 is first num < num3
                ...
                stack[numi, numj]  now given numk
                if numj > numk --> numk is first num smaller than numj on the right
                we can update res with numi, numj, numk
            3. modulo: 取余数
        Algorirhtm:
            1.  arr = [0] + arr + [0]

            2.  iterate each num from left to right:
                    (1) while (stack[-2] <) stack[-1] > num:
                        i = stack.pop()
                        j = stack[-1]
                        k = cur index
                        update res with i, j, k and num

                    (2) when we cannot pop stack anymore
                        (stack[-2] <) stack[-1] < num
                        stack[-1] will be first num smaller than cur num:

                        add cur num to stack

        """
```

735.Asteroid Collision

https://leetcode.com/problems/asteroid-collision/

image-20220118172654025

image-20220118172707116

```
class Solution:
```

```python
    def asteroidCollision(self, asteroids: List[int]) -> List[int]:
        stack = []
        res = []
        for a in asteroids:
            if a > 0:
                stack += [a]
            else:
                flag = True
                while stack:
                    if stack[-1] > abs(a):
                        flag = False
                        break
                    elif stack[-1] == abs(a):
                        stack.pop()
                        flag = False
                        break
                    else:
                        stack.pop()

                if flag:
                    res += [a]

        res += stack
        return res

        '''
        Explanation:
            we use a stack to store all remaining asteroids moving right

        Data Structure:
            stack: all remaining asteroids moving right
            flag: true if current asteroid exists

        Algorithm:
            1.  iterate each asteriod:
                (1)  if moving right:
                        add it to stack
                (2)  if moving left:
                        while stack is not empty:
                            if top of stack > cur: cur will die and break loop
                            elif top of stack == cur: cur will die, top of stack
will die too, break
                            else top of stack < cur: top of stack will die and
continue
                        if cur is still alive, add it to res

            2.  after iteration, add all asteroids in stack to res
        '''
```

BFS

102.Binary Tree Level Order Traversal

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
from collections import deque
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []
        res = []
        queue = deque()
        queue += [root]

        while queue:
            size = len(queue)
            cur = []
            for i in range(size):
                node = queue.popleft()
                if node.left:
                    queue += [node.left]
                if node.right:
                    queue += [node.right]
                cur += [node.val]
            res += [cur]
        return res
```

127.Word Ladder

```
from collections import deque
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) ->
int:
        wordList = set(wordList)
        if endWord not in wordList:
            return 0
        q = deque()
        q += [beginWord]
        level = 1

        while q: # q 不为空
            size = len(q)
            for _ in range(size):
                w = q.popleft()
                if w == endWord:
                    return level
                for i in range(len(w)):
                    for j in range(26): # 遍历26个字母
                        c = chr(ord('a') + j)
                        nw = w[:i] + c + w[i + 1:] # w[i] is replaced by c
                        if nw in wordList:
                            q += [nw]
                            wordList.remove(nw)
            level += 1
        return 0
        """
        HashTable(set); BFS

        Data Structure:
            q, queue: all the words in the same level
            v, visited: --> delete from wordList

        Algorithnm: BFS
            1. add beginWord to q:
            2. while queue is not empty:
                    check each word in the same level / q

                    (1) if word is endWord:
                            return level

                    (2) not:
                        for each char in word:
                            char --> 'a' ~ 'z'
                            if new word in wordList
                                add newWord to queue
                                delete newWord from wordList
                    (3) after checking all words:
                            level += 1
        """
```

1730.Shortest Path to Get Food

```
from collections import deque
class Solution:
    def getFood(self, grid: List[List[str]]) -> int:
        m, n = len(grid), len(grid[0])
        queue = deque()

        for i in range(m):
            for j in range(n):
                if grid[i][j] == '*':
                    queue += [[i, j]]
                    break
        step = 0
        while queue:
            size = len(queue)
            for _ in range(size):
                i, j = queue.popleft()
                for di, dj in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    ni, nj = i + di, j + dj
                    if 0 <= ni < m and 0 <= nj < n and grid[ni][nj] in ['#',
'O']:
                        if grid[ni][nj] == '#':
                            return step + 1
                        grid[ni][nj] = '|'
                        queue += [[ni, nj]]
            step += 1
        return -1
    '''
    Data Structure:
        q: lastly added nodes of same level

    '''
```

994.Rotting Oranges

```
from collections import deque
class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        if not grid: # if grid is None
            return -1
        m, n = len(grid), len(grid[0])
        fresh = 0
        q = deque()

        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    fresh += 1
                elif grid[i][j] == 2:
                    q += [(i, j)]

        time = 0
        while q and fresh > 0:
            size = len(q)
            for _ in range(size):
                i, j = q.popleft()
                for di, dj in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                    ni, nj = i + di, j + dj
                    if ni < 0 or ni >= m or nj < 0 or nj >= n or grid[ni][nj] in
[0, 2]:
                        continue
                    fresh -= 1
                    grid[ni][nj] = 2
                    q += [(ni, nj)]
            time += 1
        return time if fresh == 0 else -1

# Time complexity: O(rows * cols) -> each cell is visited at least once
# Space complexity: O(rows * cols) -> in the worst case if all the oranges are
rotten they will be added to the queue
```

DFS

79.Word Search DFS

```
class Solution:
    def exist(self, grid: List[List[str]], word: str) -> bool:
        m, n = len(grid), len(grid[0])
        def dfs(i, j, w):
            if len(w) == 0:
                return True

            temp = grid[i][j]
            grid[i][j] = '#'

            res = False
            for di, dj in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                ni, nj = i + di, j + dj
                if 0 <= ni < m and 0 <= nj < n and grid[ni][nj] == w[0]:
                    res |= dfs(ni, nj, w[1:])
            grid[i][j] = temp
            return res

        res = False
        for i in range(m):
            for j in range(n):
                if grid[i][j] == word[0]:
                    res |= dfs(i, j, word[1:])

        return res

        """
        Explanation:
            注: dfs(i, j, word): 是指以(i, j)为起点找到 word
            for searching islands:
                we just convert land into water (visited)

            But for this question,
                though we still need to mark current letter as visited

                current letter can be used by other dfs function

                so we need to convert it back after recursion

        Data Structure:
            dfs(i, j, word):
                (1) char on pos (i, j) is found (word[0])
                (2) we are searching for word[1] on pos(ni, nj)
                grid[i][j] == 'A'(word[0]), word[1:] == 'BCCED' (target is
'ABCCED')

                starting from current pos, whether or not we can find a complete
word

        Algorithm:
            1.  initialize res = False
            2.  for each pos:
                    if grid[i][j] == word[0]:
                        res |= dfs(i, j, word[1:])

            3.  dfs(i, j, w):
                    (1) Base Case
                        if len(w) == 0:
                            return True

                    (2) temp = grid[i][j]
                        grid[i][j] = '#'

                        for di, dj in [(0, 1), (0, -1), (1, 0), (-1, 0)]
                            ni, nj = i + di, j + dj
                            if ni, nj not out of boundary and grid[ni][nj] ==
w[0]
                                res |= dfs(ni, nj, w[1:])

                    (3) convert i, j back
                        grid[i][j] = temp

                    (4) return res
        """
```

212.Word Search II

```
from collections import defaultdict
class TrieNode:
    def __init__(self):
        self.children = defaultdict(TrieNode)
        self.word = None

    def addword(self, w):
        cur = self
        for c in w:
            cur = cur.children[c] #build and move
        cur.word = w #cur.word = True


class Solution:
    def findWords(self, grid: List[List[str]], words: List[str]) -> List[str]:
        root = TrieNode()
        for w in words:
            root.addword(w)

        res = []
        m, n = len(grid), len(grid[0])
        def dfs(i, j, node):
            nonlocal res
            if node.word:
                res += [node.word]

            temp = grid[i][j]
            grid[i][j] = '#'

            for di, dj in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                ni, nj = i + di, j + dj
                if 0 <= ni < m and 0 <= nj < n and grid[ni][nj] in 
node.children:
                    dfs(ni, nj, node.children[grid[ni][nj]])

            grid[i][j] = temp

        for i in range(m):
            for j in range(n):
                if grid[i][j] in root.children:
                    dfs(i, j, root.children[grid[i][j]])

        return set(res)



        """
        Explanation:
            if we search words by word Search I
            TC will be O(m * n * m * n * k)(number of words to search)
            we want to solve it within O(m * n * m * n) by Trie

        Data Structure:
            1.  Tire:
                (1) children: {key: letter, val: TireNode}
                (2) word:   the word formed by from root to current node
                            default as None

            2.  dfs(i, j, TrieNode):
                if we find a complete word, add it to self.res

                (1) char on current pos is found (TrieNode)
                (2) we are searching for children of TrieNode


        Algorithm:
            1.  initialize TrieNode
                (1) build a new class TrieNode
                (2) add all words to Trie
                now we have root of Trie (t)

            2.  (1) for each pos:
                        if cur char(grid[i][j]) on current pos in children of t:
                            dfs(i, j, t.children[cur char])

                (2) de-duplicate res


            3.  dfs(i, j, node):
                    (1) if node.word is not None:
                            add its corresponging word to result

                        #Notice that we cannot return / end here
                        #eat --> eateat
                        #so we need to conitinue to search

                    (2) <1> convert it into '#' as visited
                        <2> for new pos ni, nj:
                            if not out of boundary and grid[ni][nj] in children
of node:
                                dfs(ni, nj, node.children[grid[ni][nj](next
char)])
                        <3> convert it back
        """
```

200.Number of Islands

```python
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid or not grid[0]:
            return 0
        m, n = len(grid), len(grid[0])
        res = 0

        def dfs(x, y):
            if x < 0 or x >= m or y < 0 or y >= nr grid[x][y] == '0':
                return
            grid[x][y] = '0'
            for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                nx, ny = x + dx, y + dy
                dfs(nx, ny)

        for i in range(m):
            for j in range(n):
                if grid[i][j] == '1':
                    res += 1
                    dfs(i, j)
        return res

        """
        Data Structure:
            dfs --> turn all lands into water
            x, y


        Algorithm:
            1.  for each cell:
                    if cell is land:
                        res += 1
                        dfs(land)

            2.  dfs:
                (1) p --> r:
                    if x, y out of boundary or water
                        return
                    grid[x][y] = 0(陆地沉没法) / 2  --> visited

                (2) r --> c:
                    for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]
                        nx, ny = x + dx, y + dy
                        dfs(nx, ny)

        """
```

472.Concatenated Words Hard 没看

```
from functools import lru_cache
class Solution:
    def findAllConcatenatedWordsInADict(self, words: List[str]) -> List[str]:
        v = set(words)
        @lru_cache(None)
        def dfs(w):
            res = False
            for i in range(1, len(w)):
                w1, w2 = w[:i], w[i:]
                res |= (w1 in v) and (w2 in v or dfs(w2))
                if res:
                    return True
            return res

        res = []
        for w in words:
            if dfs(w):
                res += [w]
        return res

        """
        Data Structure:
            dfs(w): check if division parts of w can be found in words

        Algorithm:
            1.initialize v = set(words)

            2.  dfs(w):
                (1)we do not judge if w is in v, we only care about its division
parts

                (2) w is divided into w1 and w2
                    for i in range(1, n):
                        w1 = w[:i]
                        w2 = w[i:]
                        res |= (w1 in v or dfs(w1)) and (w2 in v or dfs(w2))

                        #   but notice that if dfs(w1) is duplicated
                            we just delete it

                        -->
                        res |= (w1 in v) and (w2 in v or dfs(w2))

            3.  for each w in words:
                    if dfs(w):
                        add it to res(list of words)
        """
```

Tree

103.Binary Tree Zigzag Level Order Traversal

Tree + BFS(flag)

image-20220114183627973

image-20220114183649202

```python
from collections import deque
class Solution:
    def zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []
        res = []
        q = deque()
        q += [root]
        flag = 1

        while q:
            size = len(q)
            temp = []
            for _ in range(size):
                cur = q.popleft()
                if cur.left:
                    q += [cur.left]
                if cur.right:
                    q += [cur.right]
                temp += [cur.val]
            res += [temp[::flag]]
            flag = -flag

        return res
```

236.Lowest Common Ancestor of a Binary Tree

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

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
        if root is None:
            return None

        if root.val == p.val or root.val == q.val:
            return root

        l, r = self.lowestCommonAncestor(root.left, p, q),
self.lowestCommonAncestor(root.right, p, q)
        if l and r:
            return root
        if not l:
            return r
        if not r:
            return l

        """
        Data Structure:
            dfs --> return p or q or node of other val
            p / q : (1) p / q is lowest ancestor (2) p / q is found
            node of other val: node is loweset ancestor

        Algorithm:
            (1) if current node is p / q:
                    return p / q

            (2) find p / q / lowest ancestor in left / right subtree
                    l, r = dfs(root.left), dfs(root.right)

                <1> if l and r are not None --> p and q are found in different
subtrees:
                        root is loweset ancestor
                        return root
                <2> if l or r is None
                        return the not None one (p / q / node of other val)
                <3> if l and r are both None:
                        return None
        """
```

101.Symmetric Tree

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if root is None:
            return True
        return self.isMirror(root.left, root.right)
    def isMirror(self, root1, root2):
        if root1 is None and root2 is None:
            return True
        elif root1 is None or root2 is None:
            return False
        return root1.val == root2.val and self.isMirror(root1.left, root2.right)
and self.isMirror(root1.right, root2.left)
```

863.All Nodes Distance K in Binary Tree

https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

image-20220118172506822

image-20220118172523015

BFS + DFS

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
from collections import deque
class Solution:
    def distanceK(self, root: TreeNode, target: TreeNode, k: int) -> List[int]:
        d = collections.defaultdict(list)
        def dfs(p, c):
            if p and c:
                d[p.val] += [c.val]
                d[c.val] += [p.val]
            if c.left:
                dfs(c, c.left)
            if c.right:
                dfs(c, c.right)

        dfs(None, root)

        q = deque()
        v = set()

        q += [target.val]
        v.add(target.val)
        level = 0

        while q:
            if level == k:
                return list(q)
            size = len(q)
            for _ in range(size):
                cur = q.popleft()
                for node in d[cur]:
                    if node not in v:
                        q += [node]
                        v.add(node)
            level += 1
        return []

        '''
        Explanation:
            we want convert original tree into a new tree whose root is target,
            then we can use bfs

            1.  target: d{key: node, val: set of neighbor nodes}

            2.  start with target(new root)
                bfs it

        Data Structure:
            1.  d{key: node, val: set of neighbor nodes}
            2.  q: lastly added nodes of same level
            3.  v: all visited nodes
            4.  helper(parent, child):
                    (1) put connection of parent and child into d
                    (2) put all connections in subtree rooting at child into d

        Algorithm:
            1.  helper(None, root)

            2.  initialize q, v with target, level = 0

            3.  while q:
                    1.  if level == dist, q is res and return it
                    2.  for each node in q:
                        put all unvisited neighbor nodes into q and v
                    3.  level += 1
        '''
```

Sliding Window

239.Sliding Window Maximum

```
from collections import deque
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        dq = deque()
        res = []

        for i, n in enumerate(nums):
            while dq and nums[dq[-1]] < n:
                dq.pop() # 后进先出
            while dq and dq[0] <= i - k:
                dq.popleft()
            dq += [i]
            if i >= k - 1:
                res += [nums[dq[0]]]

        return res
    """
    Explanation:
        Assume now we move to num
        And we also have a deque [n1, n2, n3 ... nk] storing candidates for max
num in current sliding window

        Before we adding nk, all nums from n1 ~ nk- 1 > nk
        in this logic
        n1 > n2 > n3 > ... nk

    Data Structure:
        deque: storing all candidates(index of num) for max num in cur window

    Algorithm:
        1.  iterate each num:
                (1) pop out all candidates < num
                    --> pop 从后面开始，所以是后进先出
                (2) pop out all candidates whose index <= current index - k
                    --> popleft

                (3) add index of the num to deque

                (4) if i >= k - 1:
                        add deque[0] to res

    """
```

3.Longest Substring Without Repeating Characters

```python
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        count = collections.defaultdict(int)
        res = 0
        start = 0
        for i in range(len(s)):
            count[s[i]] += 1
            while count[s[i]] > 1:
                count[s[start]] -= 1
                start += 1
            res = max(res, i - start + 1)
        return res
```

1151.Minimum Swaps to Group All 1's Together

```python
class Solution:
    def minSwaps(self, data: List[int]) -> int:
        n = len(data)
        k = sum(data)
        one = 0
        res = inf

        for i in range(k):
            if data[i] == 1:
                one += 1
        res = min(res, k - one)

        for i in range(k, n):
            j = i - k
            if data[i] == 1:
                one += 1
            if data[j] == 1:
                one -= 1
            res = min(res, k - one)
        return res
        """
        Sliding window with fixed length

        Explanation:
            we want all elements in window are '1'
            length of window = total number of '1' in data
            we will swap '0' in window with '1' outside window

        Data Structure:
            1.  k --> total number of '1' in data
            2.  i, j: end and start of window
            3.  one: number of '1' in window

        Algorithm:
            1.  (1) for first k elements:
                        if '1': one += 1
                (2) update res

            2.  for i in range(k, n):
                    j = i - k
                    if data[i] == '1':
                        one += 1
                    if data[j] == '1'
                        one -= 1
                    update res
        """
```

heap

973.K Closest Points to Origin

```
from heapq import heappush, heappop
class Solution:
    def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
        h = []
        for x, y in points[:k]:
            d = x ** 2 + y ** 2
            heappush(h, (-d, [x, y]))

        for x, y in points[k:]:
            d = x ** 2 + y ** 2
            heappush(h, (-d, [x, y]))
            heappop(h)

        res = []
        while h:
            _, [x, y] = heappop(h)
            res += [[x, y]]
        return res
        """
        res = []
        for _, (x, y) in h:
            res += [[x, y]]
        return res
        """

        """
        K-smallest numbers

        Data Structure:
            h:  stores points with min dist
            pop out points with longer dist
            --> max heap

    Algorithm:
        (1) for first k points:
                just push them into heap
        (2) for points[k:]
                (1) push current point into heap
                (2) pop out point with max dist
        (3) return
    """
```

1167.Minimum Cost to Connect Sticks

heap + greedy

```python
from heapq import heappop, heappush
class Solution:
    def connectSticks(self, sticks: List[int]) -> int:
        h = []
        for s in sticks:
            heappush(h, s)

        res = 0
        while len(h) >= 2:
            s1, s2 = heappop(h), heappop(h)
            res += (s1 + s2)
            heappush(h, s1 + s2)
        return res
        """
        Explanation:
            each time we just connect two smallest sticks
            add it to heap
            until there is only one stick

        Data Structure:
            h: min heap

        Algorithm:
            while len(h) >= 2:
                pop out two smallest sticks
                add cost of connection to res
                add new stick to heap
        """
```

1353.Maximum Number of Events That Can Be Attended 没写

Design

146.LRU Cache

```
class LRUCache:
    def __init__(self, capacity: int):
        self.d = collections.OrderedDict()
        self.k = capacity

    def get(self, key: int) -> int:
        if key not in self.d:
            return -1
        else:
            val = self.d[key]
            del self.d[key]
            self.d[key] = val
            return val

    def put(self, key: int, value: int) -> None:
        if key in self.d:
            del self.d[key]
        else:
            if len(self.d) == self.k:
                self.d.popitem(last = False)
        self.d[key] = value


# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
    """
    Explanation:
        cache --> [d] capacity k --> only k [key, val]
        LRU cache   --> keys are sorted by insertion time

        LRU cache --> orderedict


    Data Structure:
        OrderedDict: d
        capacity: k

    Algorithm:
        def get(key): (we do not need to consider capacity here)
            1. if key not exists in d:
                    return -1
            2. else if it does exist:
                    (1) delete item [key, val]
                    (2) put [key, val] into d again
                    (3) return val

        def put(key, val):
            1. if key exists in d:
                    (1) delete original [key, old val]
                    (2) put back [key, new val]
            2.  else key does not exist:
                    (1) if size == capacity:
                            pop out the least recenlty used item --> popitem(last
= False)
                            put [key, new val]
                    (2) else:
                            put [key, new val]
    """
```

348.Design Tic-Tac-Toe

```
class TicTacToe:

    def __init__(self, n: int):
        self.rows = [0] * n
        self.cols = [0] * n
        self.diag = 0
        self.anti_diag = 0
        self.n = n

    def move(self, i: int, j: int, player: int) -> int:
        self.rows[i] += (1 if player == 1 else -1)

        self.cols[j] += (1 if player == 1 else -1)

        if i == j:
            self.diag += (1 if player == 1 else -1)

        if i == self.n - 1 - j:
            self.anti_diag += (1 if player == 1 else -1)

        if abs(self.rows[i]) == self.n or abs(self.cols[j]) == self.n or
abs(self.diag) == self.n or abs(self.anti_diag) == self.n:
            return player
        else:
            return 0
```

```
# Your TicTacToe object will be instantiated and called as such:
# obj = TicTacToe(n)
# param_1 = obj.move(row,col,player)

    '''
    Explanation:
        we need two arrays(rows, cols) and two variables(diag, anti_diag)

        two player + 1 / - 1

        if the absolute value of rows[i] / cols[j] / diag / anti diag == n:
            current player wins

    Data Structure:
        1.  init:
            two arrays(rows, cols) and two variables(diag, anti_diag)

        2.  move:
            (1) update value in init
            (2) return 0 if nobody wins
                return id of cur player if he/her wins

    Algorithm:
        1.  initialize rows, cols, diag, anti_diag

        2.  given i, j, player
            (1) rows[i] += (1 if player == 1 else -1)
                check if abs(rows[i]) == n

            (2) cols[j] += (1 if player == 1 else -1)
                check if abs(cols[i]) == n

            (3) if i == j, update diag
                check if abs(diag[i]) == n

            (4) if i == n - 1 - j, update anti diag
                check if abs(anti_diag[i]) == n

    '''
```

1275.Find Winner on a Tic Tac Toe Game

https://leetcode.com/problems/find-winner-on-a-tic-tac-toe-game/

image-20220117232626071

image-20220117232704096

```
class Solution:
    def tictactoe(self, moves: List[List[int]]) -> str:
        n = 3
        row = [0] * n
        col = [0] * n
        player = 1
        diag = antiDiag = 0

        for i, j in moves:
            row[i] += player
            col[j] += player

            if i == j:
                diag += player

            if i + j == n - 1:
                antiDiag += player

            if abs(row[i]) == n or abs(col[j]) == n or abs(diag) == n or
abs(antiDiag) == n:
                return "A" if player == 1 else "B"

            player = -player

        return "Draw" if len(moves) == n * n else "Pending"
        '''
        348 --> Design

        Data Structure:
            row[m]
            col[n]
            diag
            antidiag

            player = 1 / -1

        Algorithm:
            iterate each move i, j
                (1) update row[i], col[j], diag, antidiag with player
                (2) if abs(row[i]) / abs(col[j]) / diag / antidiag == N
                        return current player
                (3) change player
        '''
```

bit

136.Single Number

https://leetcode.com/problems/single-number/

image-20220112113710568

思路

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

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        res = 0
        for num in nums:
            res ^= num
        return res

        """
        a ^ a == 0
        a ^ 0 == a
        """
```

Intervals & Meeting Room

56.Merge Intervals

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        res = []
        intervals = sorted(intervals) + [[10 ** 4 + 1, 10 ** 4 + 1]]
        s, e = intervals[0]

        for ns, ne in intervals[1:]:
            if ns <= e:
                e = max(e, ne)
            else:
                res += [[s, e]]
                s, e = ns, ne

        return res

        """
        Data Structure:
            list of (start, end) sorted by start
            (s, e): old
            (ns, ne): cur

        Algorithm:
            1. sort all intervals with start

            2.  iterate each intervals:
                    Assume we have a new interval with new start and new end,
[newS, newE]
                    check if we can merge current interval with that new
interval


                    if we can merge it
                        (1)
                        |___________|

                            |___________|

                        (2)
                        |______________|

                            |_____|
                        update old End with max(ne, e)
                    if we cannot:
                        add old interval to res
                        update interval = cur interval
        """
```

252.Meeting Rooms

```
class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        intervals.sort()
        #intervals.sort(key=lambda x: x[0])
        for i in range(len(intervals) - 1):
            if intervals[i + 1][0] < intervals[i][1]:
                return False

        return True
```

本题中下一个的 start time 可以和上一个的 end time 一样（面试中可以问一下：if the start time of the next interval is the same as the end time of the current interval, could the person attend all meetings?)

253.Meeting Rooms II

return *the minimum number of conference rooms required*

```
class Solution:
    import heapq
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:
        intervals.sort()
        heap = [] # stores the end time of the intervals
        for i in intervals:
                # If the new start time is greater than or equal to the exist end
time, means the room has been released, replace the previous time with the new
ending time
            if heap and heap[0] <= i[0]:
                heapq.heapreplace(heap, i[1])
            else:
                # The room is till in use, add (push a new end time to min heap)
a new room
                heapq.heappush(heap, i[1])
        return len(heap)
    """
    minHeap
    """
```

```
import heapq
# 将x压入堆中
heapq.heappush(heap, x)
# 从堆中弹出最小的元素
heapq.heappop(heap)
# 让列表具备堆特征
heapq.heapify(heap)
# 弹出最小的元素，并将x压入堆中
heapq.heapreplace(heap, x)
# 返回iter中n个最大的元素
heapq.nlargest(n, iter)
# 返回iter中n个最小的元素
heapq.nsmallest(n, iter)
```

Math & Tricky

370.Range Addition

store every start index for each value and at end index + 1 minus it

for example it will look like:

[1 , 3 , 2] , [2, 3, 3] (length = 5)

res[0, 2, 0, 0, -2]

res[0 ,2, 3, 0, -5]

sum 0, 2, 5, 5, 0

res[0, 2, 5, 5, 0]

Explanation: We update the `value` at `start index`, because it will be used in the future when we are adding up the values for the sum at each index between `start index` and `end index` (both inclusive). We update the `negative value` at the `end index + 1`, because the `positive value` of it should be only added at its previous indices (from start index to end index). Thus, when we accumulate the sum at the end for each index, we will get the correct values for each index. If the `end index` is the last index in the resulting array, we don't have to do the `end index + 1` part, because there is no more index after the last index and there will be no error when we accumulate the sum.

```
class Solution:
    def getModifiedArray(self, length: int, updates: List[List[int]]) ->
List[int]:
        arr = [0] * length
        res = [0] * length

        for s, e, i in updates:
            arr[s] += i
            if e + 1 <= length - 1:
                arr[e + 1] -= i

        s = 0 # sum
        for i in range(length):
            s += arr[i]
            res[i] = s

        return res
        """
        Explanation:
            Brute force: TC: O(lengt of updates * length of range)

            when we meet start index: number of meeting rooms += inc
            when we meet end index + 1: meeting is over, number of meeting rooms
-= inc

            for meeting room, we use cur to mark how many meeting rooms are
needed now
            for this question, we use cur to mark sum of inc now

        Data Structure:
            1.  arr[i]: how many meeting rooms are occupied or released

            2.  res[i]: res of i

        Algorithm:
            1.  for start, end, inc in updates:
                    arr[start] += inc
                    if end + 1 <= n - 1
                        arr[end + 1] -= inc

            2.  for i in range(len(arr)):
                    cur += arr[i]
                    res[i] = cur
        """
```

696.Count Binary Substrings

```
class Solution:
    def countBinarySubstrings(self, s: str) -> int:
        s += '2'
        n = len(s)
        prev = 0
        cur = 1
        res = 0

        for i in range(1, n):
            if s[i] == s[i - 1]:
                cur += 1
            else:
                res += min(prev, cur)
                prev = cur
                cur = 1
        return res
        """
        Explanation:
            1.  000 11111
                if there are two groups, res += min(len1, len2)

        Data Strucutre:
            1.  cur: length of current all '1' or all '0'
            2.  prev: length of previous all '0' or '1'

        Algorithm:
            1.  initialize prev = 0, cur = 1
            2.  iterate each index i from 1
                    if s[i] == s[i - 1]:
                        cur += 1
                    else: s[i] != s[i - 1]
                        current substring is complete
                        <1> res += min(cur, prev)
                        <2> update cur and prev:
                            cur = 1, prev = cur

            3.  for last cur:
                    to update res with it, just add '2' to end of s
        """
```

926.Flip String to Monotone Increasing

```
class Solution:
    def minFlipsMonoIncr(self, s: str) -> int:
        one, zero = 0, 0
        res = inf

        for c in s:
            if c == '0':
                zero += 1

        for c in s:
            if c == '0':
                zero -= 1
            res = min(res, one + zero)
            if c == '1':
                one += 1
        return res
        """
        greedy，也可以用dp做，但是没有 greedy 好
        Explanation:
            1.
            xxxx    0      xxx   or   xxx      1       xxx
            prev   cur    after       prev    cur     after

            we assume current number is threhold
            all left should be '0', all right should be '1'
            convert all prev '1' into '0' and all after '0' into '1'

            2.
            if we iterate num from left to right:
            we can get how many '1' on the left

            we need to iterate at first to get total number of '0'
            the number of '0' after cur position = total number of '0' - prev
'0'

        Data Structure:
            when we stand on index i(cur)
            1. one before cur: number of '1' from 0 to i - 1
            2. zero after cur: number of '0' from i + 1 to n - 1

        Algorithm:
            (1) get total number of '0'
                intialize zero, one = 0, 0
            (2) iterate each num:
                    <1> update zero to make it range from i + 1 to n - 1
                        if num == '0':
                        zero -= 1

                    <2> update res with one and zero

                    <3> if num == '1'
                        one += '1'
        """
```

937.Reorder Data in Log Files

https://leetcode.com/problems/reorder-data-in-log-files/

image-20211221174506313

image-20211221174524747

```
class Solution:
    def reorderLogFiles(self, logs: List[str]) -> List[str]:
        letters, digits = [], []
        for log in logs:
            cur = log.split()
            if cur[1].isdigit():
                digits += [log]
            else:
                letters += [log]

        letters.sort(key=lambda x:(x.split()[1:], x.split()[0]))
        return letters + digits

        """
        x --> each element in letters
        log --> string
        x.split() --> list of [0 identifier, 1: contents]
        sort by (1) contents (2)identifier
        lambda x : (x.split()[1:], x.split()[0])
        """


    '''
    Explanation:
        log --> (1) identifier (2) contents

    Data Structure:
        1.  list of [digital logs]

        2.  list of tuple of (contents, corresponding letter logs)

    Algorithm:
        1.  for each logs:
            (1) if digital logs:
                    add it to list
            (2) if letter log:
                    add it to list

        2.  sort tuples of letters by (1) contents (2) identifier
        3.  return sorted letters + digits
    '''
```

1268.Search Suggestions System

https://leetcode.com/problems/search-suggestions-system/

image-20211221183717547

image-20211221183741266

```python
class Solution:
    def suggestedProducts(self, products: List[str], searchWord: str) ->
List[List[str]]:
        products.sort()
        n = len(products)
        res = []
        t = ''

        for c in searchWord:
            t += c
            l, r = 0, n
            while l < r:
                m = l + (r - l) // 2
                if products[m] >= t:
                    r = m
                else:
                    l = m + 1
            temp = []
            for i in range(3):
                if l + i <= n - 1 and products[l + i].startswith(t):
                    temp += [products[l + i]]
            res += [temp]
        return res

    """
    Explanation:
        (1) give suggestions every time we type a char of searchWord

        (2) a, b, c, cd, cde
            if target is 'cd'
            all string < target lexigraphically cannot be prefix of 'cd'

    Data Structure:
        (1) products: stores all strings to suggest and sort them
lexicographically
        (2) target: current str

    Algorithm:
        (1) sort products and initialize target = ''
        (2) for each c in searchWord:
                <1> update target += c
                <2> find index i of first word in products >= target
lexicographically ----> binary search
                    try to add i, i + 1, i + 2 to temp (list of words to
suggest)
                        if target is prefix of those words
                <3> add temp to res
    """
```

1492.The kth Factor of n

```python
class Solution:
    def kthFactor(self, n: int, k: int) -> int:
        prev = []
        for i in range(1, int(sqrt(n)) + 1):
            if n % i == 0:
                if i * i != n:
                    prev += [i]
                k -= 1
                if k == 0:
                    return i

        if k > len(prev):
            return -1
        else:
            return n // (prev[-k])


        """
        Explanation:
            we only need to iterate from 1 to sqrt(n)
            a, b, c, d, e
            n = c * c
            --> d and e can be computed by a and b
                we only need one c so we do not consider c as num in previous
numbers

            By doing so, TC decreases from O(n) to O(sqrt(n))

        Data Structure:
            1.  k: number of remaining factors
            2.  prev: list of smaller factors

        Algorithm:
            1.  for i in range(1, int(sqrt(n))):
                    if n % i == 0:
                        (1) if i * i != n:
                                prev += [i]
                        (2) if i * i == n:
                                do nothing
                        (3) k -= 1
                            if k == 0
                            return i

            2.  Now there are not enough smaller factors
                number of bigger factor == len(prev)

                (1) if k > len(prev): not enough factors
                        return -1
                (2) else k <= len(prev)
                        return n // prev[-k]
        """
```

1710.Maximum Units on a Truck

```
class Solution:
    def maximumUnits(self, boxTypes: List[List[int]], truckSize: int) -> int:
        A = sorted(boxTypes, key = lambda x: -x[1])
        # boxTypes.sort(key=lambda x : -x[1])
        k = truckSize
        res = 0
        for c, u in A:
            if c < k:
                res += c * u
                k -= c
            else:
                res += k * u
                return res
        return res

        """
        Data Structure:
            A: (number of box, unit of box) sorted by unit from big to small
            k:  remaining boxes to put on truck

        Algorithm:
            (1) sort A, initialize k = truckSize
            (2) for each box in A:
                    if number of box < k:
                        res += count * unit
                        k -= number of box
                    else: number of box >= k
                        count = k
                        res += count * unit
                        return res
            (3) return res
        """
```

1648.Sell Diminishing-Valued Colored Balls 没写

42.Trapping Rain Water

```
class Solution:
    def trap(self, A: List[int]) -> int:
        n = len(A)
        maxL, maxR = A[0], A[n - 1]
        l, r = 1, n - 2

        res = 0

        while l <= r:
            if maxL < maxR:
                if A[l] > maxL:
                    maxL = A[l]
                else:
                    res += maxL - A[l]
                l += 1
            else:
                if A[r] > maxR:
                    maxR = A[r]
                else:
                    res += maxR - A[r]
                r -= 1

        return res
        """
        Explanation:

            maxLeft  left   ...   right maxRight

            result of bars from left to right have not been computed yet
            I do not know how much water can they trap

            maxLeft is height of tallest bar whose idx < idx of left
            maxright is height of tallest bar whose idx > idx of right

            now we want to know left / right can trap how much water

            if maxLeft < maxRight:
                we can compute result of left
                because it is determined by smaller boundary
                and smaller boundary must be maxLeft

                if left is higher than maxLeft --> there will be not water
                    --> only update maxleft and left += 1

                else: trap water and make current bar has same height with
maxLeft
                    --> only update res(trap water) and left += 1

        Data Structure:
            1. maxLeft and maxRight
            2. left and right
        Algorithm:
            1. initialize maxLeft and maxRight = nums[0] and [n - 1]
                left should be 1 and right = n - 2

            2.  while left <= right:
                    (1) if maxLeft < maxRight;
                            deal with left
                        <1> if height of left >= maxLeft:
                                update maxLeft
                                left += 1

                        <2> if height of left < maxLeft:
                                update res
                                left += 1

                    (2) else deal with right
        """
```

283.Move Zeroes

https://leetcode.com/problems/move-zeroes/

image-20211228202509871

```python
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        cur = 0
        n = len(nums)
        for i in range(n):
            if nums[i] != 0:
                nums[cur] = nums[i]
                cur += 1
        for i in range(cur, n):
            nums[i] = 0
```