# 字符串

## 3.\*无重复最长子串

给定字符串，找出不含重复字符的最长子串长度

“abcabcbabc” -> 3

#利用滑动窗口

class Solution:

def lengthOfLongestSubstring(self, s: str) -> int:

if not s:return 0 #空集返回0

left = 0 #字串最左边元素，不断的弹出左边元素直到满足要求

lookup = set()

max\_len = 0

cur\_len = 0

for i in range(len(s)):

cur\_len += 1

while s[i] in lookup:

lookup.remove(s[left])

left += 1

cur\_len -= 1

if cur\_len > max\_len:max\_len = cur\_len

lookup.add(s[i]) #集合的remove和add操作

return max\_len

## 76.最小覆盖子串

给两个字符串S、T，在S中找出包含T所有字符最小子串

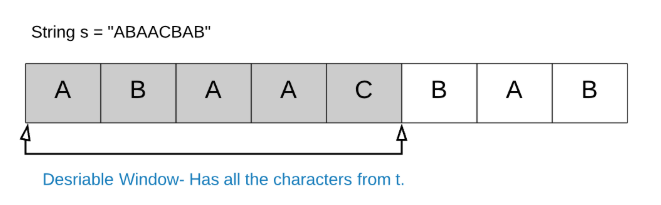
S = “ADOBECODEBANK”， T = “ABC” -> “ABC”

1.初始，left指针和 right 指针都指向 S的第一个元素.

2.将 right 指针右移，扩张窗口，直到得到一个可行窗口，亦即包含 T 的全部字母的窗口。

3.得到可行的窗口后，将 left指针逐个右移，若得到的窗口依然可行，则更新最小窗口大小。

4.若窗口不再可行，则跳转至 2

优化：建个列表，找出T在S出现所有字符和下标

filtered\_S = [(0, 'A'), (1, 'B'), (2, 'C'), (11, 'A'), (14, 'B'), (15, 'C')]

from collections import Counter

class Solution:

def minWindow(self, s: str, t: str) -> str:

if not t or not s:

return ""

dict\_t = Counter(t)#生成计数字典

dict\_tlen = len(dict\_t)

filter\_s = []

for i, c in enumerate(s):

if c in dict\_t:

filter\_s.append((i, c))

window = {}

cur\_len = 0

ans = float('inf'),None,None

l = r = 0

while r < len(filter\_s):

charactor = filter\_s[r][1]

window[charactor] = window.get(charactor,0) + 1

if window[charactor] == dict\_t[charactor]:

cur\_len += 1

while l <= r and cur\_len == dict\_tlen:

charact = filter\_s[l][1]

start = filter\_s[l][0]

end = filter\_s[r][0]

if end-start+1 < ans[0]:

ans = (end - start + 1, start, end)

window[charact] -= 1

l += 1

if window[charact] < dict\_t[charact]:

cur\_len -= 1

r += 1

return "" if ans[0] == float('inf') else s[ans[1] : ans[2]+1]

## 165.比较版本号

比较两个版本号 version1 和 version2。

如果 version1 > version2 返回 1，如果 version1 < version2 返回 -1， 除此之外返回 0。

你可以假设版本字符串非空，并且只包含数字和 . 字符。

 . 字符不代表小数点，而是用于分隔数字序列。

例如，2.5 不是“两个半”，也不是“差一半到三”，而是第二版中的第五个小版本。

你可以假设版本号的每一级的默认修订版号为 0。例如，版本号 3.4 的第一级（大版本）和第二级（小版本）修订号分别为 3 和 4。其第三级和第四级修订号均为 0。

**输入:** *version1* = "0.1", *version2* = "1.1"

**输出:** -1

class Solution:

def compareVersion(self, version1: str, version2: str) -> int:

nums1 = version1.split('.')

nums2 = version2.split('.')

n1, n2 = len(nums1), len(nums2)

# compare versions

for i in range(max(n1, n2)):

i1 = int(nums1[i]) if i < n1 else 0#补0

i2 = int(nums2[i]) if i < n2 else 0

if i1 != i2:

return 1 if i1 > i2 else -1

# the versions are equal

return 0

# 哈希表

## 166. 分数到小数

给定两个整数，分别表示分数的分子 numerator 和分母 denominator，以字符串形式返回小数。

如果小数部分为循环小数，则将循环的部分括在括号内。

**输入: numerator = 2, denominator = 3**

**输出: "0.(6)"**

class Solution:

def fractionToDecimal(self, numerator: int, denominator: int) -> str:

if numerator == 0: return "0"

res = []

# 首先判断结果正负, 异或作用就是 两个数不同 为 True 即 1 ^ 0 = 1 或者 0 ^ 1 = 1

if (numerator > 0) ^ (denominator > 0):

res.append("-")

numerator, denominator = abs(numerator), abs(denominator)

# 判读到底有没有小数

a, b = divmod(numerator, denominator)

res.append(str(a))

# 无小数

if b == 0:

return "".join(res)

res.append(".")

# 处理余数

# 把所有出现过的余数记录下来

loc = {b: len(res)}

while b:

b \*= 10

a, b = divmod(b, denominator)

res.append(str(a))

# 余数前面出现过,说明开始循环了,加括号

if b in loc:

res.insert(loc[b], "(")

res.append(")")

break

# 在把该位置的记录下来

loc[b] = len(res)

return "".join(res)

## 202.快乐数

编写一个算法来判断一个数 n 是不是快乐数。

「快乐数」定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和，然后重复这个过程直到这个数变为 1，也可能是 无限循环 但始终变不到 1。如果 可以变为  1，那么这个数就是快乐数。

如果 n 是快乐数就返回 True ；不是，则返回 False 。

**输入：**19

**输出：**true

**解释：**

12 + 92 = 82

82 + 22 = 68

62 + 82 = 100

12 + 02 + 02 = 1

class Solution:

def isHappy(self, n: int) -> bool:

def next\_value(x):

value = 0

while x:

x, cur = divmod(x,10)

value += cur\*\*2

return value

seen = set()

while n!= 1 and n not in seen:

seen.add(n)

n = next\_value(n)

return n == 1

# 双指针

## 15.三数之和

给一个包含n个整数的数组，判断nums是否存在a，b，c使得a+b+c=0

给定数组 nums = [-1, 0, 1, 2, -1, -4]，

满足要求的三元组集合为：

[[-1, 0, 1],[-1, -1, 2]]

class Solution:

def threeSum(self, nums: List[int]) -> List[List[int]]:

#选择一个C位k，再双指针

nums.sort()

res, k = [], 0

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

if nums[k] > 0: break # 1. because of j > i > k.

if k > 0 and nums[k] == nums[k - 1]: continue # 2. 跳过重复 i, j = k + 1, len(nums) - 1

while i < j: # 3. 双指针

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

if s < 0:

i += 1

while i < j and nums[i] == nums[i - 1]: i += 1

elif s > 0:

j -= 1

while i < j and nums[j] == nums[j + 1]: j -= 1

else:

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

i += 1

j -= 1

while i < j and nums[i] == nums[i - 1]: i += 1

while i < j and nums[j] == nums[j + 1]: j -= 1

return res

## 18.四数之和

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，b，c 和 d ，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。

给定数组 nums = [1, 0, -1, 0, -2, 2]，和 target = 0。

满足要求的四元组集合为：

[

[-1, 0, 0, 1],

[-2, -1, 1, 2],

[-2, 0, 0, 2]

]

class Solution:

def fourSum(self, nums: List[int], target: int) -> List[List[int]]:

result = []

if not nums or len(nums) < 4:

return result

nums.sort()# 必须排序

length = len(nums)

for k in range(length - 3):

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

continue

min1 = nums[k] + nums[k+1] + nums[k+2] + nums[k+3]

if min1 > target:

break

max1 = nums[k]+nums[length-1]+nums[length-2]+nums[length-3]

if max1 < target:

continue

for i in range(k+1, length-2):# 三数之和

if i > k + 1 and nums[i] == nums[i - 1]:

continue

j = i + 1

h = length - 1

min2 = nums[k] + nums[i] + nums[j] + nums[j + 1]

if min2 > target:

continue

max2 = nums[k] + nums[i] + nums[h] + nums[h - 1]

if max2 < target:

continue

while j < h:

curr = nums[k] + nums[i] + nums[j] + nums[h]

if curr == target:

result.append([nums[k],nums[i], nums[j], nums[h]])

j += 1

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

j += 1

h -= 1

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

h -= 1

elif curr > target:

h -= 1

elif curr < target:

j += 1

return result

## 75.颜色分类

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

**输入:** [2,0,2,1,1,0]

**输出:** [0,0,1,1,2,2]

class Solution:

def sortColors(self, nums: List[int]) -> None:

'''

荷兰三色旗问题解，0指针的左边界，2指针的右边界

'''

# 对于所有 idx < p0 : nums[idx < p0] = 0

# curr是当前考虑元素的下标

p0 = curr = 0

# 对于所有 idx > p2 : nums[idx > p2] = 2

p2 = len(nums) - 1

while curr <= p2:

if nums[curr] == 0:

nums[p0], nums[curr] = nums[curr], nums[p0]

p0 += 1

curr += 1

elif nums[curr] == 2:

nums[curr], nums[p2] = nums[p2], nums[curr]

p2 -= 1

else:

curr += 1

## 163.缺失的区间

给定一个排序的整数数组 ***nums***，其中元素的范围在 **闭区间** **[lower, upper]** 当中，返回不包含在数组中的缺失区间。

输入: nums = [0, 1, 3, 50, 75], lower = 0 和 upper = 99,

输出: ["2", "4->49", "51->74", "76->99"]

class Solution:

def findMissingRanges(self, nums: List[int], lower: int, upper: int) -> List[str]:

res = []

low = lower - 1

nums.append(upper + 1)

for num in nums:

dif = num - low

if dif == 2: res.append(str(low+1))

elif dif > 2: res.append(str(low+1) + "->" + str(num-1))

low = num

return res

# 回溯法（深度优先搜索）

## 22.括号生成

输入：n = 3

输出：[

"((()))",

"(()())",

"(())()",

"()(())",

"()()()"

]

class Solution:

def generateParenthesis(self, n: int) -> List[str]:

ans = []

def backtrack(S, left, right):

if len(S) == 2 \* n:

ans.append(''.join(S))

return

if left < n:

S.append('(')

backtrack(S, left+1, right)

#可以缩写backtrack(S+’(’, left+1, right)，就不用pop了

S.pop()

if right < left:

S.append(')')

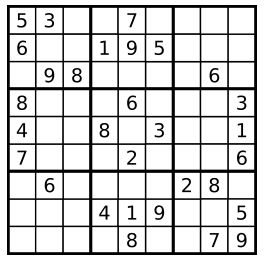
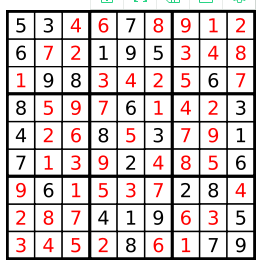
backtrack(S, left, right+1)

S.pop()

backtrack([], 0, 0)

return ans

## 37.解数独

* 给定的数独序列只包含数字 1-9 和字符 '.' 。
* 你可以假设给定的数独只有唯一解。
* 给定数独永远是 9x9 形式的

class Solution:

def solveSudoku(self, board: List[List[str]]) -> None:

"""

Do not return anything, modify board in-place instead.

"""

row = [set(range(1, 10)) for \_ in range(9)] # 行剩余可用数字

col = [set(range(1, 10)) for \_ in range(9)] # 列剩余可用数字

block = [set(range(1, 10)) for \_ in range(9)] # 块剩余可用数字

empty = [] # 收集需填数位置

for i in range(9):

for j in range(9):

if board[i][j] != '.': # 更新可用数字

val = int(board[i][j])

row[i].remove(val)

col[j].remove(val)

block[(i // 3)\*3 + j // 3].remove(val)

else:

empty.append((i, j))

def backtrack(iter=0):

if iter == len(empty): # 处理完empty代表找到了答案

return True

i, j = empty[iter]

b = (i // 3)\*3 + j // 3

for val in row[i] & col[j] & block[b]:#求集合的交集&

row[i].remove(val)

col[j].remove(val)

block[b].remove(val)

board[i][j] = str(val)

if backtrack(iter+1):

return True

row[i].add(val) # 回溯

col[j].add(val)

block[b].add(val)

return False

backtrack()

## 40.组合总和

给定一个数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次。

输入: candidates = [10,1,2,7,6,1,5], target = 8,

所求解集为:

[

[1, 7],

[1, 2, 5],

[2, 6],

[1, 1, 6]

]

class Solution:

def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:

lens = len(candidates)

cur = []

res = []

if lens == 0:

return []

candidates.sort()

def get(cur,res,begin,target):

if target == 0:

res.append(cur[:])

for index in range(begin, lens):

remain = target - candidates[index]

if remain < 0:

break

if index > begin and candidates[index - 1] == candidates[index]:#防止重复

continue

cur.append(candidates[index])

#print(cur)

get(cur,res,index+1,remain)

cur.pop()

get(cur,res,0,target)

return res

## 47.全排列

给定一个可包含重复数字的序列，返回所有不重复的全排列。

输入: [1,1,2]

输出:

[

[1,1,2],

[1,2,1],

[2,1,1]

]

class Solution:

def permuteUnique(self, nums: List[int]) -> List[List[int]]:

lens = len(nums)

if lens == 0:

return []

depth = 0 #表示当前深度

used = [False for i in range(lens)]#表示当前位置是否用过

res = []

path = [] #当前的路径

nums.sort()

def dsf(depth, path, used, res):

if depth == lens:

res.append(path[:])

return

for i in range(lens):

if not used[i]:

if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:#

continue

used[i] = True

path.append(nums[i])

dsf(depth+1, path, used, res)

used[i] = False

path.pop()

dsf(depth, path, used, res)

return res

## 79.单词搜索

给定一个二维网格和一个单词，找出该单词是否存在于网格中。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

board =

[

['A','B','C','E'],

['S','F','C','S'],

['A','D','E','E']

]

给定 word = "ABCCED", 返回 true

给定 word = "SEE", 返回 true

给定 word = "ABCB", 返回 false

class Solution:

# 定义上下左右四个行走方向

directs = [(0, 1), (0, -1), (1, 0), (-1, 0)]

def exist(self, board: List[List[str]], word: str) -> bool:

m = len(board)

if m == 0:

return False

n = len(board[0])

mark = [[0]\*n for \_ in range(m)]

for i in range(len(board)):

for j in range(len(board[0])):

if board[i][j] == word[0]:

# 将该元素标记为已使用

mark[i][j] = 1

if self.backtrack(i, j, mark, board, word[1:]) == True:

return True

else:

# 回溯

mark[i][j] = 0

return False

def backtrack(self, i, j, mark, board, word):

if len(word) == 0:

return True

for direct in self.directs:

cur\_i = i + direct[0]

cur\_j = j + direct[1]

if 0 <= cur\_i < len(board) and 0 <= cur\_j < len(board[0]) and board[cur\_i][cur\_j] == word[0]:

# 如果是已经使用过的元素，忽略

if mark[cur\_i][cur\_j] == 1:

continue

# 将该元素标记为已使用

mark[cur\_i][cur\_j] = 1

if self.backtrack(cur\_i,cur\_j,mark,board,word[1:])== True:

return True

else:

# 回溯

mark[cur\_i][cur\_j] = 0

return False

## 93.复原IP地址

给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。

有效的 IP 地址正好由四个整数（每个整数位于 0 到 255 之间组成），整数之间用 '.' 分隔。

**输入:** "25525511135"

**输出:** ["255.255.11.135", "255.255.111.35"]

class Solution:

def restoreIpAddresses(self, s: str) -> List[str]:

res = []

n = len(s)

def backtrack(i, tmp, flag):

if i == n and flag == 0:

res.append(tmp[:-1])

return

if flag < 0:

return

for j in range(i, i + 3):

if j < n:

if i == j and s[j] == "0":

backtrack(j + 1, tmp + s[j] + ".", flag - 1)

break #0开头是0只能单独占位

if 0 < int(s[i:j + 1]) <= 255:

backtrack(j + 1, tmp + s[i:j + 1] + ".", flag - 1)

backtrack(0, "", 4)

return res

## 130.被围绕的区域

给定一个二维的矩阵，包含 'X' 和 'O'（字母 O）。

找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充

X X X X

X O O X

X X O X

X O X X

运行你的函数后，矩阵变为：

X X X X

X X X X

X X X X

X O X X

class Solution:

def solve(self, board: List[List[str]]) -> None:

if not board or not board[0]:

return

row = len(board)

col = len(board[0])

def dfs(i,j):

board[i][j] = 'B'

for x, y in [(-1,0), (1,0), (0,-1), (0,1)]:

cur\_i = i + x

cur\_j = j + y

if 1 <= cur\_i <= row-2 and 1 <= cur\_j <= col-2 and board[cur\_i][cur\_j] == 'O':

dfs(cur\_i,cur\_j)

for i in range(row):

if board[i][0] == 'O':

dfs(i,0)

if board[i][col-1] == 'O':

dfs(i,col-1)

for j in range(col):

if board[0][j] == 'O':

dfs(0,j)

if board[row-1][j] == 'O':

dfs(row-1,j)

for i in range(row):

for j in range(col):

if board[i][j] == 'O':

board[i][j] = 'X'

if board[i][j] == 'B':

board[i][j] = 'O'

## 200.岛屿数量

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

**输入:**

11000

11000

00100

00011

**输出:** 3

**解释:** 每座岛屿只能由水平和/或竖直方向上相邻的陆地连接而成。

class Solution:

def numIslands(self, grid: List[List[str]]) -> int:

def dfs(r,c):

grid[r][c] = '0'

for x,y in {(r-1,c), (r+1,c), (r,c-1), (r,c+1)}:

if 0 <= x < nr and 0 <= y < nc and grid[x][y] == '1':

dfs(x,y)

nr = len(grid)

if nr == 0:

return 0

nc = len(grid[0])

res = 0

for i in range(nr):

for j in range(nc):

if grid[i][j] == '1':

res += 1

dfs(i,j)

return res

## 131.分割回文串

给定一个字符串 *s*，将*s*分割成一些子串，使每个子串都是回文串。

返回 *s* 所有可能的分割方案。

**输入:** "aab"

**输出:**

[

["aa","b"],

["a","a","b"]

]

class Solution:

def partition(self, s: str) -> List[List[str]]:

n = len(s)

dp = [[False]\*n for \_ in range(n)]

for r in range(n):

for l in range(r+1):

if s[l] == s[r] and (r-l < 2 or dp[l+1][r-1]):

dp[l][r] = True

res = []

def helper(i,tmp):

if i == n:

res.append(tmp)

for j in range(i,n):

if dp[i][j]:

helper(j+1, tmp+[s[i:j+1]])

helper(0, [])

return res

# 广度优先搜索

## 127.单词接龙

给定两个单词（beginWord 和 endWord）和一个字典，找到从 beginWord 到 endWord 的最短转换序列的长度。转换需遵循如下规则：

每次转换只能改变一个字母。

转换过程中的中间单词必须是字典中的单词。

输入:

beginWord = "hit",

endWord = "cog",

wordList = ["hot","dot","dog","lot","log","cog"]

输出: 5

解释: 一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog",

返回它的长度 5。

from collections import defaultdict

class Solution:

def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:

if not beginWord or not endWord or not wordList or endWord not in wordList:

return 0

all\_combo\_dict = defaultdict(list)

L = len(beginWord)

for word in wordList:

for i in range(L):

all\_combo\_dict[word[:i] + '\*' + word[i+1:]].append(word)

queue = [(beginWord,1)]

visited = {beginWord: True}

while queue:

current\_word, level = queue.pop(0)

for i in range(L):

intermedia = current\_word[:i] + '\*' + current\_word[i+1:]

for word in all\_combo\_dict[intermedia]:

if word == endWord:

return level + 1

if word not in visited:

queue.append((word, level+1))

visited[word] = True

all\_combo\_dict[intermedia] = []

return 0

# 贪心算法

## 134.加油站

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

输入:

gas = [1,2,3,4,5]

cost = [3,4,5,1,2]

输出: 3

解释:

从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油

开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油

开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油

开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油

开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油

开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。

因此，3 可为起始索引。

class Solution:

def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:

n = len(gas)

total, cur = 0, 0

start = 0

for i in range(n):

total += gas[i] - cost[i]

cur += gas[i] - cost[i]

if cur < 0:

cur = 0

start = i + 1

return start if total >=0 else -1

# DP动态规划问题

## 5.最长回文子串

给定一个字符串，找到s的最长回文子串

“abccbauuu” -> “abccba”

class Solution:

def longestPalindrome(self, s: str) -> str:

size = len(s)

if size <= 1:

return s

# 二维 dp 问题，动态规划问题

# 状态：dp[l,r]: s[l:r] 包括 l，r ，表示的字符串是不是回文串

dp = [[False for \_ in range(size)] for \_ in range(size)] longest\_l = 1

res = s[0] #置初始状态

# 因为只有 1 个字符的情况在最开始做了判断

# 左边界一定要比右边界小，因此右边界从 1 开始

for r in range(1, size):

for l in range(r):

# 状态转移方程：如果头尾字符相等并且中间也是回文

# 还要考虑长度很短的情况下

if s[l] == s[r] and (r - l <= 2 or dp[l + 1][r - 1]):

dp[l][r] = True

cur\_len = r - l + 1

if cur\_len > longest\_l:

longest\_l = cur\_len

res = s[l:r + 1]

return res

## 72.编辑距离

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符

删除一个字符

替换一个字符

输入：word1 = "horse", word2 = "ros"

输出：3

解释：

horse -> rorse (将 'h' 替换为 'r')

rorse -> rose (删除 'r')

rose -> ros (删除 'e')

class Solution:

def minDistance(self, word1: str, word2: str) -> int:

n1 = len(word1)

n2 = len(word2)

dp = [[0] \* (n2 + 1) for \_ in range(n1 + 1)]

# 第一行

for j in range(1, n2 + 1):

dp[0][j] = dp[0][j-1] + 1

# 第一列

for i in range(1, n1 + 1):

dp[i][0] = dp[i-1][0] + 1

for i in range(1, n1 + 1):

for j in range(1, n2 + 1):

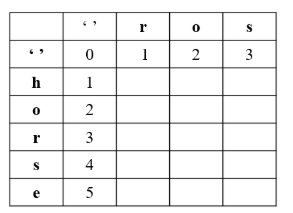
if word1[i-1] == word2[j-1]:

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

else:

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

return dp[-1][-1]



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

## 121.买卖股票最佳时机

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

输入: [7,1,5,3,6,4]

输出: 5

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。

注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

class Solution:

def maxProfit(self, prices: List[int]) -> int:

minprice = float("inf")

maxprofit = 0

for price in prices:

maxprofit = max(price - minprice, maxprofit)

minprice = min(price, minprice)

return maxprofit

#通过记录当前最低价时最大收益

## 122.买卖股票最佳时机II

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

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

输入: [7,1,5,3,6,4]

输出: 7

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

  随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = 6-3 = 3 。

class Solution:

def maxProfit(self, prices: List[int]) -> int:

profit = 0

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

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

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

return profit

#累加

## 123.买卖股票最佳时机III

给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 两笔 交易。

输入: [3,3,5,0,0,3,1,4]

输出: 6

解释: 在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。

  随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天 （股票价格 = 4）的时候卖出，这笔交易所能获得利润 = 4-1 = 3 。

## 152.乘积最大子数组

给你一个整数数组 nums ，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

**输入:** [2,3,-2,4]

**输出:** 6

**解释:** 子数组 [2,3] 有最大乘积 6。

#可能存在负数，所以维护一个最小值

class Solution:

def maxProduct(self, nums: List[int]) -> int:

if not nums: return

res = nums[0]

pre\_max = nums[0]

pre\_min = nums[0]

for num in nums[1:]:

cur\_max = max(pre\_max \* num, pre\_min \* num, num)

cur\_min = min(pre\_max \* num, pre\_min \* num, num)

res = max(res, cur\_max)

pre\_max = cur\_max

pre\_min = cur\_min

return res

## 213.打家劫舍II

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

**输入:** [2,3,2]

**输出:** 3

**解释:** 你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2）, 因为他们是相邻的。

class Solution:

def rob(self, nums: List[int]) -> int:

def my\_rob(nums):

cur, pre = 0, 0

for num in nums:

cur, pre = max(pre + num, cur), cur#涉及到dp只和前两个有关，可以设置双变量

return cur

return max(my\_rob(nums[:-1]),my\_rob(nums[1:])) if len(nums) != 1 else nums[0]#把抢第一家和最后一个单独考虑

# 位运算

## 29.两数相除

给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符

输入: dividend = 10, divisor = 3

输出: 3

解释: 10/3 = truncate(3.33333..) = truncate(3) = 3

假设我们的环境只能存储 32 位有符号整数，其数值范围是 [−231,  231− 1]。本题中，如果除法结果溢出，则返回 231– 1

class Solution:

def divide(self, dividend: int, divisor: int) -> int:

sign = (dividend > 0) ^ (divisor > 0)#异或

dividend = abs(dividend)

divisor = abs(divisor)

count = 0

#把除数不断左移，直到它大于被除数

while dividend >= divisor:

count += 1 #几进位

divisor <<= 1

result = 0

while count > 0:

count -= 1

divisor >>= 1

if divisor <= dividend:

result += 1 << count

#这里的移位运算是把二进制（第count+1位上的1）转换为十进制

dividend -= divisor

if sign: result = -result

return result if -(1<<31) <= result <= (1<<31)-1 else (1<<31)-1

## 50．Pow（X, n）

实现 pow(*x*, *n*)，即计算 x 的 n 次幂函数。

输入: 2.10000, 3

输出: 9.26100

class Solution:

def myPow(self, x: float, n: int) -> float:

a = abs(n)

final = 1

while a > 0:

if a % 2 == 0:

x \*= x

a /= 2

final \*= x

a -= 1

return final if n>0 else 1/final

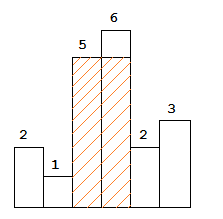
#例如n\*\*77：n\*(n\*\*4)\*(n\*\*8)\*(n\*\*64)

# 栈

## 84. 柱状图中最大的矩形

**输入:** [2,1,5,6,2,3]

**输出:** 10



class Solution:

def largestRectangleArea(self, heights: List[int]) -> int:

#哨兵单调栈

res = 0

stack = []

heights = [0] + heights + [0]

stack.append(0)

lens = len(heights)

for i in range(1, lens):

while heights[i] < heights[stack[-1]]:

h = heights[stack.pop()]

w = i - stack[-1] - 1

res = max(res, h\*w)

stack.append(i)

return res

## 85. 最大矩形

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

输入:

[

["1","0","1","0","0"],

["1","0","1","1","1"],

["1","1","1","1","1"],

["1","0","0","1","0"]

]

输出: 6

class Solution:

def maximalRectangle(self, matrix: List[List[str]]) -> int:

def ans84(heights):

#哨兵单调栈

res = 0

stack = []

heights = [0] + heights + [0]

stack.append(0)

lens = len(heights)

for i in range(1, lens):

while heights[i] < heights[stack[-1]]:

h = heights[stack.pop()]

w = i - stack[-1] - 1

res = max(res, h\*w)

stack.append(i)

return res

row = len(matrix)

if row == 0:

return 0

col = len(matrix[0])

height = [0]\*col

max\_area = 0

for i in range(row):

for j in range(col):

if matrix[i][j] == '1':

height[j] += 1

else:

height[j] = 0

cur = ans84(height)

max\_area = max(max\_area, cur)

return max\_area

# 图

## 207.课程表

你这个学期必须选修 numCourse 门课程，记为 0 到 numCourse-1 。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们：[0,1]

给定课程总量以及它们的先决条件，请你判断是否可能完成所有课程的学习？

**输入:** 2, [[1,0]]

**输出:** true

**解释:** 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。所以这是可能的。

#有向无环图，入度表

from collections import deque

class Solution:

def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:

indegrees = [0 for \_ in range(numCourses)] #入度

adjacency = [[] for \_ in range(numCourses)] #入度表

queue = deque()

# Get the indegree and adjacency of every course.

for cur, pre in prerequisites:

indegrees[cur] += 1

adjacency[pre].append(cur)

# Get all the courses with the indegree of 0.

for i in range(len(indegrees)):

if not indegrees[i]: queue.append(i) #0进入队列表示可直接学习

# BFS TopSort.

while queue:

pre = queue.popleft() #弹出一个

numCourses -= 1 #总课程减一

for cur in adjacency[pre]:

indegrees[cur] -= 1 #对应的所有后续课程入度减一

if not indegrees[cur]: queue.append(cur)#0进入队列可学习

return not numCourses#课程为0表示全部学完

## 332.重新安排行程（欧拉回路）

给定一个机票的字符串二维数组 [from, to]，子数组中的两个成员分别表示飞机出发和降落的机场地点，对该行程进行重新规划排序。所有这些机票都属于一个从JFK（肯尼迪国际机场）出发的先生，所以该行程必须从 JFK 出发。

说明:

如果存在多种有效的行程，你可以按字符自然排序返回最小的行程组合。例如，行程 ["JFK", "LGA"] 与 ["JFK", "LGB"] 相比就更小，排序更靠前

所有的机场都用三个大写字母表示（机场代码）。

假定所有机票至少存在一种合理的行程。

输入: [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]

输出: ["JFK", "MUC", "LHR", "SFO", "SJC"]

class Solution:

def findItinerary(self, tickets: List[List[str]]) -> List[str]:

from collections import defaultdict

graph = defaultdict(list)

res = []

for x, y in sorted(tickets)[::-1]:

graph[x].append(y)

def dfs(tmp):

while graph[tmp]:

dfs(graph[tmp].pop())

res.append(tmp)

dfs("JFK")

return res[::-1]

# 形状变换

## 6.Z字形变换

将一个给定字符串根据给定的行数，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 "LEETCODEISHIRING" 行数为 3 时，排列如下：

L C I R

E T O E S I I G

E D H N

结果："LCIRETOESIIGEDHN"

#巧用flag

class Solution:

def convert(self, s: str, numRows: int) -> str:

if numRows < 2: return s

res = ["" for \_ in range(numRows)]

i, flag = 0, -1#正常flag往下逐渐加一，对应索引加一；反转flag往上减一，对应索引减一

for c in s:

res[i] += c

if i == 0 or i == numRows - 1:

flag = -flag

i += flag

return "".join(res)

## 48.旋转图像

给定一个 *n*× *n* 的二维矩阵表示一个图像。

将图像顺时针旋转 90 度。

给定 matrix =

[

[1,2,3],

[4,5,6],

[7,8,9]

],

原地旋转输入矩阵，使其变为:

[

[7,4,1],

[8,5,2],

[9,6,3]

]

class Solution:

def rotate(self, matrix: List[List[int]]) -> None:

"""

Do not return anything, modify matrix in-place instead.

"""

n = len(matrix[0])

# transpose matrix

for i in range(n):

for j in range(i, n):

matrix[j][i], matrix[i][j] = matrix[i][j], matrix[j][i]

# reverse each row

for i in range(n):

matrix[i].reverse()

#matrix[:] = zip(\*matrix[::-1])#反转，转置

## 54.螺旋矩阵

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

**输入:**

[

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

]

**输出:** [1,2,3,6,9,8,7,4,5]

class Solution:

def spiralOrder(self, matrix: List[List[int]]) -> List[int]:

if not matrix: return []

R, C = len(matrix), len(matrix[0])

seen = [[False] \* C for \_ in range(R)]#表示是否被访问过了

ans = []

dr = [0, 1, 0, -1]

dc = [1, 0, -1, 0]#右下左上

r = c = di = 0

for \_ in range(R \* C):

ans.append(matrix[r][c])

seen[r][c] = True

cr, cc = r + dr[di], c + dc[di]

if 0 <= cr < R and 0 <= cc < C and not seen[cr][cc]:

r, c = cr, cc

else:

di = (di + 1) % 4

r, c = r + dr[di], c + dc[di]

return ans

# 特殊算法

## 169.多数元素（摩尔投票法）

给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于 ⌊ n/2 ⌋ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

**输入:** [2,2,1,1,1,2,2]

**输出:** 2

## 229.求众数（摩尔投票法）

给定一个大小为 n 的数组，找出其中所有出现超过 ⌊ n/3 ⌋ 次的元素。

说明: 要求算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

**输入:** [1,1,1,3,3,2,2,2]

**输出:** [1,2]

class Solution:#设置两个候选人

def majorityElement(self, nums: List[int]) -> List[int]:

can1, can2 = 0, 0

count1, count2 = 0, 0

res = []

for num in nums:

if num == can1:

count1 += 1

elif num == can2:

count2 += 1

elif count1 == 0:

can1 = num

count1 += 1

elif count2 == 0:

can2 = num

count2 += 1

else:

count1 -= 1

count2 -= 1

std = len(nums) // 3

if count1 > 0 and nums.count(can1) > std:

res.append(can1)

if count2 > 0 and nums.count(can2) > std:

res.append(can2)

return res

## 204.计数质数（埃式筛）

统计所有小于非负整数 n 的质数的数量。

**输入:** 10

**输出:** 4

**解释:** 小于 10 的质数一共有 4 个, 它们是 2, 3, 5, 7 。

class Solution:

def countPrimes(self, n: int) -> int:

# 最小的质数是 2

if n < 2:

return 0

isPrime = [1] \* n

isPrime[0] = isPrime[1] = 0 # 0和1不是质数，先排除掉

# 埃式筛，把不大于根号n的所有质数的倍数剔除

for i in range(2, int(n \*\* 0.5) + 1):

if isPrime[i]:

isPrime[i \* i:n:i] = [0] \* ((n - 1 - i \* i) // i + 1)

#设置步长i

return sum(isPrime)

# 常用模块和方法

## # itertools系列

from itertools import permutations, combinations

a = [1,2,3]

pe = list(permutations(a))

#排列[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]

com = list(combinations(a,2))

#组合[(1, 2), (1, 3), (2, 3)]

## # collections系列

from collections import Counter, deque, defaultdict

#计数器(Counter)

a = [1,2,1,1,2,3,3,3,3]

a\_count = Counter(a)

#Counter({3: 4, 1: 3, 2: 2})

#双向队列(deque)

a\_deque = deque(a)

a\_deque.popleft()

#deque([2, 1, 1, 2, 3, 3, 3, 3])

#默认字典(defaultdict)

a = [1,2,1]

hash\_list = defaultdict(list)#列表字典

for i in range(3):

hash\_list[a[i]].append('\*')

#defaultdict(<class 'list'>, {1: ['\*', '\*'], 2: ['\*']})

hash\_int = defaultdict(int)#用来计数

for k in a:

hash\_int[k] += 1

#defaultdict(<class 'int'>, {1: 2, 2: 1})

hash\_set = defaultdict(set)#集合字典

for i in range(3):

hash\_set[a[i]].add('\*')

#defaultdict(<class 'set'>, {1: {'\*'}, 2: {'\*'}})