# 数组模板

from typing import List

class Solution:

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

# while (left < right): # 有序数组-二分法

# while i < n: # 二重遍历-快慢指针

# while i < n: # 二重遍历-滑动窗口

# float('-inf') # 更新最大值-初始化最小值

# range(0, 9) # 循环不变量-左开右闭

return 0

def ACM():

obj = Solution()

while True:

try:

nums = list(map(int, input().split(" ")))

target = nums.pop(0)

x = obj.function(nums, target)

print(x)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

nums = [1, 2, 3, 5, 8]

target = 5

obj = Solution()

x = obj.function(nums, target)

print(x)

# ACM()

# 02. 链表模板

class ListNode:

def \_\_init\_\_(self, val=None, next=None):

self.val = val

self.next = next

class Solution:

def function(self, head: ListNode, val: int) -> ListNode:

dummy\_head = ListNode(next=head) # 创建虚拟头部节点以简化删除过程

cur = dummy\_head

while cur.next: # 遍历链表

cur = cur.next

temp = cur.next # 要改变cur->next的指向之前，请用temp保存

slow = fast = cur # 快慢指针：需要固定间距N，fast和slow先达到间距N再同时移动。快慢指针可分析比例列方程

return dummy\_head.next

def NumsToListNode(self, nums):

dummy = ListNode(None)

root = dummy

for i in range(len(nums)):

node = ListNode(nums[i])

root.next = node

root = root.next

return dummy.next

def ListNodeToNums(self, root):

nums = []

while root:

nums.append(root.val)

root = root.next

return nums

def ACM():

obj = Solution()

while True:

try:

nums = list(map(int, input().split(" ")))

val = nums.pop(0)

obj = Solution()

root = obj.NumsToListNode(nums)

x = obj.function(root, val)

# print(x)

nums = obj.ListNodeToNums(x)

print(nums)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

nums = [1, 0, 0, 1, 1]

val = 1

obj = Solution()

root = obj.NumsToListNode(nums)

x = obj.function(root, val)

print(x)

nums = obj.ListNodeToNums(x)

print(nums)

# ACM()

# 04. 哈希表模板

from typing import List

class Solution:

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

# res = [0] \* 26 # 统计有限个数-数组

# res[ord(i) - ord("a")] += 1

# res = set() # 统计无限个数-集合

# res.add(i)

# res = {} # 统计无限个数-字典

# res[i] = res.get([i], 0) + 1

# 获取int每位数字

# n = 31010

# res = []

# while n:

# n, r = divmod(n, 10)

# res.append(r)

# res.reverse()

# 不求索引-可排序

# while left < right and nums[left] == nums[left + 1]: # 3数4数求和-abcd分别去重

return 0

def ACM():

obj = Solution()

while True:

try:

nums = list(map(int, input().split(" ")))

target = nums.pop(0)

x = obj.function(nums, target)

print(x)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

nums = [1, 2, 3, 5, 8]

target = 5

obj = Solution()

x = obj.function(nums, target)

print(x)

# ACM()

# 05. 二叉树模板

from typing import List

class Solution:

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

nums = nums[::-1] # 双重翻转-整体和局部

return 0

# KMP

def getNext(self, next, s):

j = -1

next[0] = j

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

while j >= 0 and s[i] != s[j + 1]:

j = next[j]

if s[i] == s[j + 1]:

j += 1

next[i] = j

def strStr(self, haystack: str, needle: str) -> int: # KMP

if not needle:

return 0

next = [0] \* len(needle)

self.getNext(next, needle)

j = -1

for i in range(len(haystack)):

while j >= 0 and haystack[i] != needle[j + 1]:

j = next[j]

if haystack[i] == needle[j + 1]:

j += 1

if j == len(needle) - 1:

return i - j

return -1

def ACM():

obj = Solution()

while True:

try:

haystack, needle = input().split(' ')

x = obj.strStr(haystack, needle)

print(x)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

haystack = "abbabbabc"

needle = "abbabc"

obj = Solution()

x = obj.strStr(haystack, needle)

print(x)

ACM() # abbabbabc abbabc

# 06. 栈队列模板

from typing import List

from collections import deque

import heapq

class MyQueue: # 单调队列（从大到小

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

self.queue = deque()

def pop(self, value):

if self.queue and value == self.queue[0]:

self.queue.popleft()

def push(self, value):

while self.queue and value > self.queue[-1]:

self.queue.pop()

self.queue.append(value)

def front(self):

return self.queue[0]

class Solution:

# 小顶堆（前k大

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

# pri\_que = [] # 小顶堆，弹小留大。若要实现大顶堆，将数据取反后push到堆中，pop的时候再取反即可

# heapq.heappush(pri\_que, (freq, key)) # 元组按靠前元素排序

# if len(pri\_que) > k: # 堆的大小大于了K弹出，保证堆的大小一直为k

# heapq.heappop(pri\_que)

#

# for i in range(k - 1, -1, -1): # 弹出剩余的大元素，先弹小，所以逆序

# result[i] = heapq.heappop(pri\_que)[0]

return 0

def ACM():

obj1 = Solution()

while True:

try:

nums = list(map(int, input().split(" ")))

target = nums.pop(0)

outcome = obj1.function(nums, target)

print(outcome)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

# ACM()

nums = [1, 2, 3, 5, 8]

target = 5

obj1 = Solution()

outcome = obj1.function(nums, target)

print(outcome)

# 06. 二叉树模板

'''

https://leetcode.cn/problems/average-of-levels-in-binary-tree/

给定一个非空二叉树的根节点 root , 以数组的形式返回每一层节点的平均值。与实际答案相差 10-5 以内的答案可以被接受。

'''

from typing import List, Optional

from collections import deque

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

class Solution:

def averageOfLevels(self, root: Optional[TreeNode]) -> List[float]:

if not root:

return []

queue = deque([root])

result = []

while queue:

l = len(queue)

sum = 0

for \_ in range(l):

cur = queue.popleft()

sum += cur.val

if cur.left:

queue.append(cur.left)

if cur.right:

queue.append(cur.right)

result.append(sum / l)

return result

# 根据数组构建二叉树

def NumsToBinarytree(self, nums: [], nonenode='null') -> TreeNode:

if not nums:

return None

# 用于存放构建好的节点

root = TreeNode(-1)

Tree = []

# 将数组元素全部转化为树节点

for i in range(len(nums)):

if nums[i] != nonenode:

node = TreeNode(nums[i])

else:

node = None

Tree.append(node)

if i == 0:

root = node

# 直接判断2\*i+2<len(Tree)会漏掉2\*i+1=len(Tree)-1的情况

for i in range(len(Tree)):

if Tree[i] and 2 \* i + 1 < len(Tree):

Tree[i].left = Tree[2 \* i + 1]

if 2 \* i + 2 < len(Tree):

Tree[i].right = Tree[2 \* i + 2]

return root

def BinarytreeToNums(self, root: Optional[TreeNode], nonenode='null') -> List[int]:

if not root:

return None

nums = [root.val]

queue = deque([root])

while queue:

l = len(queue)

for \_ in range(l):

cur = queue.popleft()

if cur.left:

queue.append(cur.left)

nums.append(cur.left.val)

else:

nums.append(nonenode)

if cur.right:

queue.append(cur.right)

nums.append(cur.right.val)

else:

nums.append(nonenode)

return nums

def ACM():

obj1 = Solution()

while True:

try:

nums = list(map(int, input().split(" ")))

obj1 = Solution()

root = obj1.NumsToBinarytree(nums)

outcome = obj1.averageOfLevels(root)

print(outcome)

nums = obj1.BinarytreeToNums(root)

print(nums)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

# ACM()

nums = [0, 1, 2, 'null', 'null', 1, 1]

obj1 = Solution()

root = obj1.NumsToBinarytree(nums)

outcome = obj1.averageOfLevels(root)

print(outcome)

nums = obj1.BinarytreeToNums(root)

print(nums)

# 07. 回溯模板

'''

https://leetcode.cn/problems/combinations/

给定两个整数 n 和 k，返回范围 [1, n] 中所有可能的 k 个数的组合。

你可以按 任何顺序 返回答案。

'''

from typing import List

class Solution:

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

self.path = []

self.result = []

def combine(self, n: int, k: int) -> List[List[int]]:

self.backtracking(n, k, 1)

return self.result

def backtracking(self, n, k, startIndex):

if len(self.path) == k:

self.result.append(self.path[:]) # path为浅复制，必须要重新用一个列表path[:]，否则后续path改变，result会随之变化

return

# for i in range(startIndex, n - (k - len(path)) + 2): # 优化的地方

for i in range(startIndex, n + 1):

if startIndex + (k - len(self.path)) > n + 1: # 优化的地方从for范围中剥离出来写成if

return

self.path.append(i) # 处理节点

self.backtracking(n, k, i + 1)

self.path.pop() # 回溯，撤销处理的节点

def ACM():

obj1 = Solution()

while True:

try:

n, k = list(map(int, input().split(" ")))

outcome = obj1.combine(n, k)

print(outcome)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

# ACM()

n = 4

k = 3

obj1 = Solution()

outcome = obj1.combine(n, k)

print(outcome)

# 09. 动规模板

'''

https://leetcode.cn/problems/climbing-stairs/

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

'''

from typing import List

class Solution:

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

# 排除 Corner Case

if n == 1:

return 1

# 创建 dp table

dp = [0] \* (n + 1)

# 初始化 dp 数组

dp[1] = 1

dp[2] = 2

# 遍历顺序

for i in range(3, n + 1):

dp[i] = dp[i - 1] + dp[i - 2]

# 返回答案

return dp[n]

def ACM():

obj1 = Solution()

while True:

try:

n = int(input())

outcome = obj1.climbStairs(n)

print(outcome)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

# ACM()

n = 2

obj1 = Solution()

outcome = obj1.climbStairs(n)

print(outcome)

# 09. 01背包

'''

https://leetcode.cn/problems/partition-equal-subset-sum/description/

给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

'''

from typing import List

class Solution:

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

# 排除 Corner Case

s = sum(nums)

if s % 2 != 0:

return False

# 创建 dp table

bagnums = s // 2

dp = [0] \* (bagnums + 1)

# 初始化 dp 数组

for j in range(nums[0], bagnums + 1):

dp[j] = nums[0]

# 遍历顺序

for i in range(1, len(nums)): # 内层容量，内层背包均正确

# for j in range(nums[i], bagnums + 1): # 容量顺序遍历，错误

for j in range(bagnums, nums[i] - 1, -1): # 容量逆序遍历

dp[j] = max(dp[j], nums[i] + dp[j - nums[i]])

if dp[-1] == bagnums:

return True

# 返回答案

return False

def ACM():

obj1 = Solution()

while True:

try:

nums = list(map(int, input().split(" ")))

outcome = obj1.canPartition(nums)

print(outcome)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

# ACM()

nums = [1, 3, 2, 4]

obj1 = Solution()

outcome = obj1.canPartition(nums)

print(outcome)

# 09. 完全背包

'''

https://leetcode.cn/problems/coin-change-ii/description/

给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。

请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0 。

假设每一种面额的硬币有无限个。

题目数据保证结果符合 32 位带符号整数。

'''

from typing import List

class Solution:

def change(self, amount: int, coins: List[int]) -> int:

# 排除 Corner Case

# 创建 dp table

dp = [0] \* (amount + 1)

# 初始化 dp 数组

dp[0] = 1

# 遍历顺序

for i in coins: # 本题求组合：内层容量；若求排列：内存物品

for j in range(i, amount + 1): # 完全背包放入多次，顺序遍历

dp[j] += dp[j - i]

# 返回答案

return dp[-1]

def ACM():

obj1 = Solution()

while True:

try:

nums = list(map(int, input().split(" ")))

amount = nums.pop(0)

outcome = obj1.change(amount, nums)

print(outcome)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

# ACM()

amount = 5

coins = [1, 2, 5]

obj1 = Solution()

outcome = obj1.change(amount, coins)

print(outcome)

# 11. 搜索模板

'''

https://leetcode.cn/problems/number-of-islands/description/

岛屿总是被水包围，并且每座岛屿只能由水平方向和/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。

'''

from typing import List

from collections import deque

class SolutionDFS:

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

self.count = 0

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

m, n = len(grid), len(grid[0])

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

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

result = 0

def dfs(x, y): # 内方法省去了变量传递

for d in dirs: # dfs从任意一方向（四方向之一）一直递归

nextx = x + d[0]

nexty = y + d[1]

if nextx < 0 or nextx >= m or nexty < 0 or nexty >= n: # 越界了，直接跳过

continue

if not visited[nextx][nexty] and grid[nextx][nexty] == 1: # 没有访问过的同时是陆地的

visited[nextx][nexty] = True

dfs(nextx, nexty)

for i in range(m):

for j in range(n):

if not visited[i][j] and grid[i][j] == 1:

visited[i][j] = True

result += 1 # 遇到没访问过的陆地，+1

dfs(i, j) # 将与其链接的陆地都标记上 true

return result

class SolutionBFS:

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

self.count = 0

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

m = len(grid)

n = len(grid[0])

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

dirs = [[0, 1], [1, 0], [-1, 0], [0, -1]]

result = 0

def bfs(i, j):

q = deque()

q.append((i, j)) # 加入队列即标记，防止重复加入，append必定伴随visited

visited[i][j] = True

while q: # bfs从任意一节点（队尾）一直发散，入队出队

x, y = q.popleft()

for d in dirs:

next\_i = x + d[0]

next\_j = y + d[1]

if next\_i < 0 or next\_i >= m or next\_j < 0 or next\_j >= n:

continue

if not visited[next\_i][next\_j] and grid[next\_i][next\_j] == 1:

q.append((next\_i, next\_j))

visited[next\_i][next\_j] = True

for i in range(m):

for j in range(n):

if not visited[i][j] and grid[i][j] == 1:

result += 1

bfs(i, j)

return result

def ACM():

obj1 = SolutionDFS()

while True:

try:

n = int(input())

grid = []

for \_ in range(n):

grid.append(list(map(int, input().split(" "))))

outcome = obj1.numIslands(grid)

print(outcome)

except:

break

if \_\_name\_\_ == "\_\_main\_\_":

# ACM()

grid = [

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

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

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

[0, 0, 0, 1, 1]

]

obj1 = SolutionDFS()

outcome = obj1.numIslands(grid)

print(outcome)

obj2 = SolutionBFS()

outcome = obj2.numIslands(grid)

print(outcome)