**ASSIGNMENT 13.04.2024**

1.**Height of Binary Tree After Subtree Removal Queries You are given the root of a binary tree with n nodes. Each node is assigned a unique value from 1 to n. You are also given an array queries of size m.You have to perform m independent queries on the tree where in the ith query you do the following: ● Remove the subtree rooted at the node with the value queries[i] from the tree. It is guaranteed that queries[i] will not be equal to the value of the root. Return an array answer of size m where answer[i] is the height of the tree after performing the ith query. Note: ● The queries are independent, so the tree returns to its initial state after each query. ● The height of a tree is the number of edges in the longest simple path from the root to some node in the tree.**

class TreeNode:

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

self.value = value

self.left = left

self.right = right

def compute\_heights(node, height):

if node is None:

return -1

left\_height = compute\_heights(node.left, height)

right\_height = compute\_heights(node.right, height)

height[node.value] = 1 + max(left\_height, right\_height)

return height[node.value]

def remove\_subtree\_and\_compute\_height(root, remove\_val, height):

if root is None:

return 0

if root.value == remove\_val:

return 0

if root.left and root.left.value == remove\_val:

return height[root.value] - (height[root.left.value] + 1)

if root.right and root.right.value == remove\_val:

return height[root.value] - (height[root.right.value] + 1)

new\_height = height[root.value]

return new\_height

def height\_after\_queries(root, queries):

height = {}

compute\_heights(root, height)

result = []

for query in queries:

new\_height = remove\_subtree\_and\_compute\_height(root, query, height)

result.append(new\_height)

return result

2. **Sort Array by Moving Items to Empty Space You are given an integer array nums of size n containing each element from 0 to n - 1 (inclusive). Each of the elements from 1 to n - 1 represents an item, and the element 0 represents an empty space. In one operation, you can move any item to the empty space. nums is considered to be sorted if the numbers of all the items are in ascending order and the empty space is either at the beginning or at the end of the array. For example, if n = 4, nums is sorted if: ● nums = [0,1,2,3] or ● nums = [1,2,3,0] ...and considered to be unsorted otherwise.Return the minimum number of operations needed to sort nums**

from collections import deque

def min\_operations\_to\_sort(nums):

n = len(nums)

sorted1 = tuple(range(n)) # (0, 1, 2, ..., n-1)

sorted2 = tuple(range(1, n)) + (0,) # (1, 2, ..., n-1, 0)

start = tuple(nums)

if start == sorted1 or start == sorted2:

return 0

def swap(nums, i, j):

nums = list(nums)

nums[i], nums[j] = nums[j], nums[i]

return tuple(nums)

queue = deque([(start, nums.index(0), 0)]) # (current state, index of 0, number of moves)

visited = set([start])

while queue:

current, zero\_index, moves = queue.popleft()

for i in range(n):

if i != zero\_index:

new\_state = swap(current, i, zero\_index)

if new\_state in visited:

continue

if new\_state == sorted1 or new\_state == sorted2:

return moves + 1

queue.append((new\_state, i, moves + 1))

visited.add(new\_state)

return -1 # In case no solution is found (should not happen for valid input)

3**. Apply Operations to an Array You are given a 0-indexed array nums of size n consisting of non-negative integers.You need to apply n - 1 operations to this array where, in the ith operation (0-indexed), you will apply the following on the ith element of nums: ● If nums[i] == nums[i + 1], then multiply nums[i] by 2 and set nums[i + 1] to 0. Otherwise, you skip this operation. After performing all the operations, shift all the 0's to the end of the array. ● For example, the array [1,0,2,0,0,1] after shifting all its 0's to the end, is [1,2,1,0,0,0]. Return the resulting array.Note that the operations are applied sequentially, not**

def apply\_operations(nums):

n = len(nums)

for i in range(n - 1):

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

nums[i] \*= 2

nums[i + 1] = 0

result = [num for num in nums if num != 0]

zeros = [0] \* (n - len(result))

return result + zeros

4. **Maximum Sum of Distinct Subarrays With Length K You are given an integer array nums and an integer k. Find the maximum subarray sum of all the subarrays of nums that meet the following conditions: ● The length of the subarray is k, and ● All the elements of the subarray are distinct. Return the maximum subarray sum of all the subarrays that meet the conditions. If no subarray meets the conditions, return 0. A subarray is a contiguous non-empty sequence of elements within an array**

def max\_sum\_of\_distinct\_subarrays(nums, k):

n = len(nums)

if k > n:

return 0

max\_sum = 0

current\_sum = 0

window\_start = 0

seen = {}

for window\_end in range(n):

num = nums[window\_end]

if num in seen:

while nums[window\_start] != num:

del seen[nums[window\_start]]

current\_sum -= nums[window\_start]

window\_start += 1

current\_sum -= nums[window\_start]

window\_start += 1

seen[num] = True

current\_sum += num

if window\_end - window\_start + 1 > k:

del seen[nums[window\_start]]

current\_sum -= nums[window\_start]

window\_start += 1

if window\_end - window\_start + 1 == k:

max\_sum = max(max\_sum, current\_sum)

return max\_sum

5. **Total Cost to Hire K Workers You are given a 0-indexed integer array costs where costs[i] is the cost of hiring the ith worker.You are also given two integers k and candidates. We want to hire exactly k workers according to the following rules: ● You will run k sessions and hire exactly one worker in each session. ● In each hiring session, choose the worker with the lowest cost from either the first candidates workers or the last candidates workers. Break the tie by the smallest index. ○ For example, if costs = [3,2,7,7,1,2] and candidates = 2, then in the first hiring session, we will choose the 4th worker because they have the lowest cost [3,2,7,7,1,2]. ○ In the second hiring session, we will choose 1st worker because they have the same lowest cost as 4th worker but they have the smallest index [3,2,7,7,2]. Please note that the indexing may be changed in the process. ● If there are fewer than candidates workers remaining, choose the worker with the lowest cost among them. Break the tie by the smallest index. ● A worker can only be chosen once. Return the total cost to hire**

import heapq

def total\_cost\_to\_hire\_k\_workers(costs, k, candidates):

n = len(costs)

total\_cost = 0

min\_heap\_start = []

min\_heap\_end = []

for i in range(candidates):

heapq.heappush(min\_heap\_start, (costs[i], i))

for i in range(max(candidates, n - candidates), n):

heapq.heappush(min\_heap\_end, (costs[i], i))

start\_ptr = candidates

end\_ptr = n - candidates - 1

for \_ in range(k):

# Get the smallest cost from both heaps, breaking ties by index

if min\_heap\_start and (not min\_heap\_end or min\_heap\_start[0] <= min\_heap\_end[0]):

cost, idx = heapq.heappop(min\_heap\_start)

total\_cost += cost

if start\_ptr <= end\_ptr:

heapq.heappush(min\_heap\_start, (costs[start\_ptr], start\_ptr))

start\_ptr += 1

else:

cost, idx = heapq.heappop(min\_heap\_end)

total\_cost += cost

if start\_ptr <= end\_ptr:

heapq.heappush(min\_heap\_end, (costs[end\_ptr], end\_ptr))

end\_ptr -= 1

return total\_cost

6. **Minimum Total Distance Traveled There are some robots and factories on the X-axis. You are given an integer array robot where robot[i] is the position of the ith robot. You are also given a 2D integer array factory where factory[j] = [positionj, limitj] indicates that positionj is the position of the jth factory and that the jth factory can repair at most limitj robots. The positions of each robot are unique. The positions of each factory are also unique. Note that a robot can be in the same position as a factory initially. All the robots are initially broken; they keep moving in one direction. The direction could be the negative or the positive direction of the X-axis. When a robot reaches a factory that did not reach its limit, the factory repairs the robot, and it stops moving. At any moment, you can set the initial direction of moving for some robot. Your target is to minimize the total distance traveled by all the robots. Return the minimum total distance traveled by all the robots. The test cases are generated such that all the robots can be repaired. Note that ● All robots move at the same speed. ● If two robots move in the same direction, they will never collide. ● If two robots move in opposite directions and they meet at some point, they do not collide. They cross each other. ● If a robot passes by a factory that reached its limits, it crosses it as if it does not exist. ● If the robot moved from a position x to a position y, the distance it moved is |y - x|.**def min\_total\_distance(robot, factory):

robot.sort()

factory.sort()

n = len(robot)

m = len(factory)

current\_direction = 1 # 1 for right, -1 for left

total\_distance = 0

factory\_index = 0

for pos, limit in factory:

if current\_direction == 1:

robots\_to\_move = [r for r in robot if r <= pos]

else:

robots\_to\_move = [r for r in robot if r >= pos]

for r in robots\_to\_move:

total\_distance += abs(r - pos)

for r in robots\_to\_move:

robot.remove(r)

current\_direction \*= -1

return total\_distance

7. **Minimum Subarrays in a Valid Split You are given an integer array nums.Splitting of an integer array nums into subarrays is valid if: ● the greatest common divisor of the first and last elements of each subarray is greater than 1, and ● each element of nums belongs to exactly one subarray. Return the minimum number of subarrays in a valid subarray splitting of nums. If a valid subarray splitting is not possible, return -1. Note that: ● The greatest common divisor of two numbers is the largest positive integer that evenly divides both numbers. ● A subarray is a contiguous non-empty part of an array.**  
import math

from collections import defaultdict

def minSubarrays(nums):

n = len(nums)

if n == 0:

return 0

dp = [float('inf')] \* (n + 1)

dp[0] = 0

gcd\_dp = defaultdict(lambda: float('inf'))

gcd\_dp[0] = 0

def gcd(a, b):

while b:

a, b = b, a % b

return a

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

current\_gcd = 0

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

current\_gcd = gcd(current\_gcd, nums[k])

if current\_gcd > 1:

dp[j] = min(dp[j], gcd\_dp[current\_gcd] + 1)

gcd\_dp[current\_gcd] = min(gcd\_dp[current\_gcd], dp[j])

return dp[n] if dp[n] != float('inf') else -1

8. **Number of Distinct Averages You are given a 0-indexed integer array nums of even length. As long as nums is not empty, you must repetitively: ● Find the minimum number in nums and remove it. ● Find the maximum number in nums and remove it. ● Calculate the average of the two removed numbers. The average of two numbers a and b is (a + b) / 2. ● For example, the average of 2 and 3 is (2 + 3) / 2 = 2.5. Return the number of distinct averages calculated using the above process.Note that when there is a tie for a minimum or maximum number, any can be removed.**def numberOfDistinctAverages(nums):

nums.sort()

n = len(nums)

averages = set()

left, right = 0, n - 1

while left < right:

average = (nums[left] + nums[right]) / 2.0

averages.add(average)

left += 1

right -= 1

return len(averages)

9. **Count Ways To Build Good Strings Given the integers zero, one, low, and high, we can construct a string by starting with an empty string, and then at each step perform either of the following: ● Append the character '0' zero times. ● Append the character '1' one times. This can be performed any number of times.A good string is a string constructed by the above process having a length between low and high (inclusive). Return the number of dif erent good strings that can be constructed satisfying these properties. Since the answer can be large, return it modulo 109 + 7**

def countGoodStrings(zero, one, low, high):

MOD = 10\*\*9 + 7

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

dp[0] = 1

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

if i >= low:

dp[i] = (dp[i-1] + dp[i-2]) % MOD

else:

dp[i] = dp[i-1]

result = 0

for i in range(low, high + 1):

result = (result + dp[i]) % MOD

return result

10. **Most Profitable Path in a Tree There is an undirected tree with n nodes labeled from 0 to n - 1, rooted at node 0. You are given a 2D integer array edges of length n - 1 where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the tree. At every node i, there is a gate. You are also given an array of even integers amount, where amount[i] represents: ● the price needed to open the gate at node i, if amount[i] is negative, or, ● the cash reward obtained on opening the gate at node i, otherwise. The game goes on as follows: ● Initially, Alice is at node 0 and Bob is at node bob. ● At every second, Alice and Bob each move to an adjacent node. Alice moves towards some leaf node, while Bob moves towards node 0. ● For every node along their path, Alice and Bob either spend money to open the gate at that node, or accept the reward. Note that: ○ If the gate is already open, no price will be required, nor will there be any cash reward. ○ If Alice and Bob reach the node simultaneously, they share the price/reward for opening the gate there. In other words, if the price to open the gate is c, then both Alice and Bob pay c / 2 each. Similarly, if the reward at the gate is c, both of them receive c / 2 each. ● If Alice reaches a leaf node, she stops moving. Similarly, if Bob reaches node 0, he stops moving. Note that these events are independent of each other. Return the maximum net income Alice can have if she travels towards the optimal leaf node.**

def maxIncomeDFS(node, parent, edges, amount, dp):

# Initialize dp arrays for the current node

dp[node] = [0, 0]

if amount[node] >= 0:

dp[node][0] = amount[node] // 2

else:

dp[node][0] = amount[node] \* (-1) // 2

dp[node][1] = dp[node][0]

for child in edges[node]:

if child != parent:

maxIncomeDFS(child, node, edges, amount, dp)

dp[node][0] = max(dp[node][0], dp[child][0])

dp[node][1] = max(dp[node][1], dp[child][1])

def maxIncome(edges, amount):

n = len(amount)

if n == 1

return 0

tree = [[] for \_ in range(n)]

for a, b in edges:

tree[a].append(b)

tree[b].append(a)

dp = [[0, 0] for \_ in range(n)]

maxIncomeDFS(0, -1, tree, amount, dp)

return dp[0][0]