**ASSIGNMENT 6**

**1 . Height of Binary Tree After Subtree Removal Queries** CODE : class TreeNode:

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

def tree\_height(root):

if not root:

return -1

left\_height = tree\_height(root.left) right\_height = tree\_height(root.right) return 1 + max(left\_height, right\_height)

def remove\_subtree(root, val):

if not root:

return None if root.val == val:

return None

root.left = remove\_subtree(root.left, val) root.right = remove\_subtree(root.right, val) return root

def process\_queries(root, queries):

answer = [] for query in queries: original\_left = root.left original\_right = root.right root.left = remove\_subtree(original\_left, query) root.right = remove\_subtree(original\_right, query) height\_after\_removal = tree\_height(root) answer.append(height\_after\_removal)

root.left = original\_left

root.right = original\_right

return answer

root = TreeNode(1) root.left = TreeNode(2) root.right = TreeNode(3) root.left.left = TreeNode(4) root.left.right = TreeNode(5) root.right.left = TreeNode(6) root.right.right = TreeNode(7) queries = [2, 3] print(process\_queries(root, queries))

OUTPUT :



1. **. Sort Array by Moving Items to Empty Space** CODE : def min\_operations\_to\_sort(nums):
   1. = len(nums) zero\_index = nums.index(0) moves\_to\_start = zero\_index

additional\_moves\_start = sum(1 for i in range(1, n) if nums[(zero\_index + i) % n] != i) moves\_to\_end = (n - 1) - zero\_index

additional\_moves\_end = sum(1 for i in range(1, n) if nums[(zero\_index - i) % n] != i)

total\_moves\_start = moves\_to\_start + additional\_moves\_start total\_moves\_end = moves\_to\_end + additional\_moves\_end

return min(total\_moves\_start, total\_moves\_end)

nums = [3, 0, 1, 2] print(min\_operations\_to\_sort(nums))

OUTPUT :



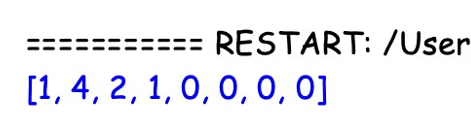
1. **. Apply Operations to an Array** CODE : def apply\_operations(nums):
   1. = len(nums) for i in range(n - 1):

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

nums[i] \*= 2 nums[i + 1] = 0

result = [num for num in nums if num != 0] # Collect all non-zero elements zero\_count = nums.count(0) # Count the number of zeros result.extend([0] \* zero\_count) # Append zeros to the end return result

nums = [1, 2, 2, 1, 1, 0, 0, 1] print(apply\_operations(nums)) # Output: [1, 4, 2, 1, 0, 0, 0, 0] OUTPUT :



1. **. Maximum Sum of Distinct Subarrays With Length K** CODE : def max\_subarray\_sum\_distinct(nums, k):
   1. = len(nums) if k > n:

return 0

max\_sum = 0 current\_sum = 0 for i in range(k):

current\_sum += nums[i]

if len(set(nums[:k])) == k:

max\_sum = current\_sum for i in range(k, n):

current\_sum += nums[i] - nums[i - k] if len(set(nums[i - k + 1:i + 1])) == k:

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

return max\_sum

nums = [1, 2, 1, 2, 3, 4, 5] k = 3

print(max\_subarray\_sum\_distinct(nums, k))

OUTPUT :



**5 . Total Cost to Hire K Workers** CODE : import heapq def total\_hiring\_cost(costs, k, candidates):

n = len(costs) if k > n:

return 0

left\_heap = [] right\_heap = []

for i in range(min(candidates, n)):

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

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

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

total\_cost = 0 left\_index = candidates right\_index = n - candidates - 1

for \_ in range(k):

if left\_heap and (not right\_heap or left\_heap[0] <= right\_heap[0]):

cost, idx = heapq.heappop(left\_heap) total\_cost += cost if left\_index <= right\_index:

heapq.heappush(left\_heap, (costs[left\_index], left\_index)) left\_index += 1

else:

cost, idx = heapq.heappop(right\_heap) total\_cost += cost

if left\_index <= right\_index:

heapq.heappush(right\_heap, (costs[right\_index], right\_index)) right\_index -= 1

return total\_cost

costs = [3, 2, 7, 7, 1, 2]

k = 3

candidates = 2

print(total\_hiring\_cost(costs, k, candidates))

OUTPUT :



**6 . Minimum Total Distance Travelled** CODE : def min\_total\_distance(robot, factory):

robot.sort()

factory.sort()

total\_distance = 0 factory\_index = 0

robots\_assigned = [0] \* len(factory) for r in robot:

while factory\_index < len(factory) and robots\_assigned[factory\_index] >=

factory[factory\_index][1]:

factory\_index += 1

if factory\_index < len(factory):

total\_distance += abs(r - factory[factory\_index][0]) robots\_assigned[factory\_index] += 1

return total\_distance

robot = [1, 3, 5] factory = [[2, 2], [4, 1]] print(min\_total\_distance(robot, factory)) OUTPUT :



**7 . Minimum Subarrays in a Valid Split** CODE :

from math import gcd from itertools import combinations

def min\_subarrays(nums):

n = len(nums) if n == 0:

return -1

dp = [float('inf')] \* n dp[0] = 1 for i in range(1, n):

for j in range(i + 1): if gcd(nums[j], nums[i]) > 1: if j == 0: dp[i] = min(dp[i], 1)

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

return dp[-1] if dp[-1] != float('inf') else -1 nums = [2, 3, 4, 9, 6] print(min\_subarrays(nums)) # Output: 2 (split as [2,3,4], [9,6])

OUTPUT :



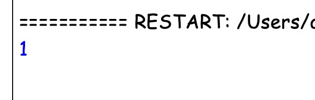
**8 . Number of Distinct Averages** CODE : def distinctAverages(nums): nums.sort() distinct\_averages = set() left, right = 0, len(nums) - 1

while left < right: min\_val = nums[left] max\_val = nums[right] average = (min\_val + max\_val) / 2 distinct\_averages.add(average) left += 1 right -= 1 return len(distinct\_averages)

nums = [1, 3, 2, 5, 4, 6]

print(distinctAverages(nums))

OUTPUT :



**9 . Count Ways To Build Good Strings**

CODE : 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 >= zero: dp[i] = (dp[i] + dp[i - zero]) % MOD

if i >= one:

dp[i] = (dp[i] + dp[i - one]) % MOD

result = 0 for i in range(low, high + 1):

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

return result zero = 1 one = 1 low = 3

high = 5

print(countGoodStrings(zero, one, low, high))

OUTPUT :



**10 . Most Profitable Path in a Tree** CODE :

from collections import defaultdict, deque def maxNetIncome(n, edges, amount, bob): MOD = 10\*\*9 + 7

tree = defaultdict(list) for a, b in edges: tree[a].append(b) tree[b].append(a)

dist\_from\_root = [-1] \* n dist\_from\_root[0] = 0 queue = deque([0]) while queue:

node = queue.popleft() for neighbor in tree[node]:

if dist\_from\_root[neighbor] == -1: dist\_from\_root[neighbor] = dist\_from\_root[node] + 1 queue.append(neighbor)

def dfs(node, parent): max\_income = -float('inf') is\_leaf = True for neighbor in tree[node]:

if neighbor != parent: is\_leaf = False income = dfs(neighbor, node) max\_income = max(max\_income, income)

alice\_income = 0 bob\_income = 0 if dist\_from\_root[node] < dist\_from\_root[bob]: alice\_income = amount[node]

elif dist\_from\_root[node] > dist\_from\_root[bob]:

bob\_income = amount[node] else:

alice\_income = amount[node] // 2 bob\_income = amount[node] // 2 if is\_leaf: return alice\_income

else:

return alice\_income + max\_income return dfs(0, -1)

n = 7

edges = [[0,1],[0,2],[1,3],[1,4],[2,5],[2,6]]

amount = [0, -2, 3, 4, -3, 2, 1] bob = 4

print(maxNetIncome(n, edges, amount, bob))

OUTPUT :

