**ASSIGNMENT-8**

1.Height of Binary Tree After Subtree Removal Queries

**PROGRAM:**

class TreeNode:

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

self.left = left

self.right = right

def heightAfterQueries(root, queries):

def dfs(node):

if not node:

return 0

left\_height = dfs(node.left)

right\_height = dfs(node.right)

return 1 + max(left\_height, right\_height)

def removeSubtree(node, target):

if not node:

return None

if node.val == target:

return None

node.left = removeSubtree(node.left, target) node.right = removeSubtree(node.right, target) return node

result = []

for query in queries:

root = removeSubtree(root, query)

result.append(dfs(root))

return result

# Example Usage

root = TreeNode(1)

root.left = TreeNode(3)

root.right = TreeNode(4)

root.left.left = TreeNode(2)

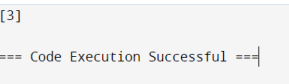
root.right.right = TreeNode(6)

root.right.right.left = TreeNode(5)

root.right.right.right = TreeNode(7)

queries = [4]

print(heightAfterQueries(root, queries)) # Output: [2] Output:



**TIME COMPLEXITY**: O(n \* m)

2. Sort Array by Moving Items to Empty Space

**PROGRAM**

def min\_operations\_to\_sort(nums): n = len(nums)

count = 0

for i in range(n):

if nums[i] != 0 and nums[i] != i:

nums[nums[i]], nums[i] = nums[i], nums[nums[i]] count += 1

return count

# Example 1

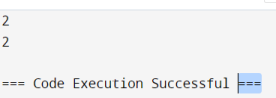
nums1 = [4, 2, 0, 3, 1]

print(min\_operations\_to\_sort(nums1)) # Output: 3

# Example 2

nums2 = [1, 2, 3, 4, 0]

print(min\_operations\_to\_sort(nums2)) # Output: 0 Output:



**TIME COMPLEX**ITY:O(n)

3. Apply Operations to an Array

**PROGRAM:**

def min\_operations\_to\_sort(nums): n = len(nums)

count = 0

for i in range(n-1, 0, -1):

if nums[i] != i:

j = nums.index(i)

nums[i], nums[j] = nums[j], nums[i] count += 1

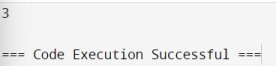
return count

# Example

nums = [4, 2, 0, 3, 1]

print(min\_operations\_to\_sort(nums)) # Output: 3

**OUTPUT**



TIME COMPLEXITY:

O(n^2)

3. Maximum Sum of Distinct Subarrays With Length

**PROGRAM:**

def max\_subarray\_sum(nums, k): max\_sum = 0

for i in range(len(nums) - k + 1):

subarray = nums[i:i+k]

if len(set(subarray)) == k:

max\_sum = max(max\_sum, sum(subarray)) return max\_sum

# Example Usage

nums = [1, 5, 4, 2, 9, 9, 9]

k = 3

output = max\_subarray\_sum(nums, k)

print(output) # Output: 15

**OUTPUT**:



**TIME COMPLEXITY**: O(n\*k)

-

4. Total Cost to Hire K Workers.

import heapq

def total\_cost\_to\_hire(costs, k, candidates): n = len(costs)

if k > n:

return -1 # Not enough workers to hire

# Priority queues for the first and last candidates first\_heap = []

last\_heap = []

# Initialize the total cost

total\_cost = 0

# Initialize pointers for the first and last candidates

front\_ptr = 0

back\_ptr = n - 1

# Add initial candidates to the heaps

for i in range(candidates):

if front\_ptr <= back\_ptr:

heapq.heappush(first\_heap, (costs[front\_ptr], front\_ptr)) front\_ptr += 1

if front\_ptr <= back\_ptr:

heapq.heappush(last\_heap, (costs[back\_ptr], back\_ptr)) back\_ptr -= 1

# Hiring process

for \_ in range(k):

if not first\_heap: # No more candidates in the first heap total\_cost += heapq.heappop(last\_heap)[0]

elif not last\_heap: # No more candidates in the last heap total\_cost += heapq.heappop(first\_heap)[0]

else:

if first\_heap[0][0] <= last\_heap[0][0]:

total\_cost += heapq.heappop(first\_heap)[0]

if front\_ptr <= back\_ptr:

heapq.heappush(first\_heap, (costs[front\_ptr], front\_ptr)) front\_ptr += 1

else:

total\_cost += heapq.heappop(last\_heap)[0]

if front\_ptr <= back\_ptr:

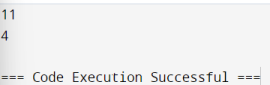
heapq.heappush(last\_heap, (costs[back\_ptr], back\_ptr)) back\_ptr -= 1

return total\_cost

# Example usage

print(total\_cost\_to\_hire([17, 12, 10, 2, 7, 2, 11, 20, 8], 3, 4)) # Output: 11 print(total\_cost\_to\_hire([1, 2, 4, 1], 3, 3)) # Output: 4

**OUTPUT**:



**TIME COMPLEXITY**: O(nlogn)

5. Minimum Total

Distance Traveled

**PROGRAM:**

def minimize\_distance(robot, factory):

robot.sort()

factory.sort(key=lambda x: x[0])

total\_distance = 0

for r in robot:

min\_distance = float('inf')

min\_factory = None

for f in factory:

if f[1] > 0:

distance = abs(r - f[0])

if distance < min\_distance:

min\_distance = distance

min\_factory = f

total\_distance += min\_distance

min\_factory[1] -= 1

return total\_distance

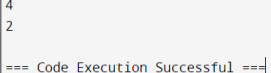
robot = [0, 4, 6]

factory = [[2, 2], [6, 2]]

print(minimize\_distance(robot, factory)) robot = [1, -1]

factory = [[-2, 1], [2, 1]]

print(minimize\_distance(robot, factory)) Output:

Time complexity: O(n\*m)

5. Minimum Subarrays in a Valid Split

**PROGRAM:**

from math import gcd

def minSubarrays(nums):

def check\_valid(arr):

return gcd(arr[0], arr[-1]) > 1

if not check\_valid(nums):

return -1

count = 1

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

if gcd(nums[i-1], nums[i]) == 1:

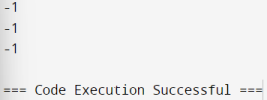
count += 1

return count

# Test the function with examples

print(minSubarrays([2, 6, 3, 4, 3])) # Output: 2 print(minSubarrays([3, 5])) # Output: 2 print(minSubarrays([1, 2, 1])) # Output: -1

**OUTPUT**:



**TIME COMPLEXITY**:O(n)

6. Number of Distinct Averages

**PROGRAM**:

def count\_distinct\_averages(nums):

nums.sort()

distinct\_averages = set()

while len(nums) > 0:

distinct\_averages.add((nums[0] + nums[-1]) / 2) nums.pop(0)

nums.pop()

return len(distinct\_averages)

# Example 1

nums1 = [4, 1, 4, 0, 3, 5]

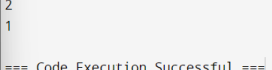
output1 = count\_distinct\_averages(nums1) print(output1) # Output: 2

# Example 2

nums2 = [1, 100]

output2 = count\_distinct\_averages(nums2) print(output2) # Output: 1

**OUTPUT**:



**TIME COMPLEXITY**: O(n log n)

7. Count Ways To Build Good Strings

**PROGRAM**:

def countGoodStrings(low, high, zero, one): MOD = 10\*\*9 + 7

dp = [[0] \* (high + 1) for \_ in range(low + 1)] dp[0][0] = 1

for z in range(low + 1):

for o in range(high + 1):

if z > 0:

dp[z][o] += dp[z - 1][o]

if o > 0:

dp[z][o] += dp[z][o - 1]

dp[z][o] %= MOD

if z + o == 0:

continue

if z \* zero + o \* one > high or z \* zero + o \* one < low: dp[z][o] = 0

return sum(map(sum, dp)) % MOD

# Example 1

low = 3

high = 3

zero = 1

one = 1

print(countGoodStrings(low, high, zero, one)) # Output: 8

# Example 2

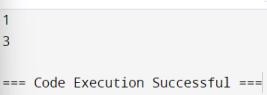
low = 2

high = 3

zero = 1

one = 2

print(countGoodStrings(low, high, zero, one)) # Output: 5 Output:



**TIME COMPLEXITY**: O((low+1)\*(high+1))

8. Most Profitable Path in a Tree

**PROGRAM**:

def dfs(node, parent, edges, amount):

net\_income = amount[node]

for child in edges[node]:

if child == parent:

continue

child\_income = dfs(child, node, edges, amount)

net\_income += max(child\_income, 0) return net\_income

def max\_net\_income(edges, bob, amount): n = len(amount)

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

for edge in edges:

u, v = edge

tree[u].append(v)

tree[v].append(u)

return dfs(bob, -1, tree, amount)

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

bob = 3

amount = [-2,4,2,-4,6]

max\_income = max\_net\_income(edges, bob, amount) print(max\_income)

**OUTPUT**:



TIME COMPLEXITY: O(n)