1. Height of Binary Tree After Subtree Removal Queries:-

Time Complexity: O(n+q)O(n + q)O(n+q)

Space Complexity: O(n)O(n)O(n)

```
def subtree_heights(root):
    heights = {}

    def dfs(node):
        if not node:
            return 0
        left_height = dfs(node.left)
        right_height = dfs(node.right)
        height = 1 + max(left_height, right_height)
        heights[node] = height
        return height

    dfs(root)
    return heights

def height_after_removal(root, queries):
    heights = subtree_heights(root)
    results = []
    for query in queries:
        node = find_node(root, query)
        if node in heights:
            results.append(heights[root] - heights[node])
        else:
            results.append(heights[root])
    return results
```

2. Sort Array by Moving Items to Empty Space:-

- Time Complexity: O(nlog n)O(n \log n)O(nlogn)
- Space Complexity: O(1)O(1)O(1)

```
def min_operations_to_sort(nums):
    n = len(nums)
    zero_index = nums.index(0)
    sorted_nums = sorted(nums)
    ops = 0

    for i in range(n):
        if nums[i] != sorted_nums[i]:
            ops += 1
            swap_index = nums.index(sorted_nums[i])
            nums[zero_index], nums[swap_index] = nums[swap_index],
nums[zero_index]
            zero_index = swap_index
```

```
    return ops

nums = [1, 0, 2, 4, 3]
print(min_operations_to_sort(nums))
```

3. Apply Operations to an Array:-

- Time Complexity: O(n)O(n)O(n)
- Space Complexity: O(1)O(1)O(1)

```
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
    nums.sort(key=lambda x: x == 0)
    return nums

nums = [1, 2, 2, 1, 1, 0]
print(apply_operations(nums))
```

4. Maximum Sum of Distinct Subarrays With Length K:-

- Time Complexity: O(n)O(n)O(n)
- Space Complexity: O(k)O(k)O(k)

```
def max_sum_of_distinct_subarrays(nums, k):
    if len(nums) < k:
        return 0

    max_sum = 0
    current_sum = 0
    window = set()

    for i in range(len(nums)):
        if nums[i] in window:
            return 0
        current_sum += nums[i]
        window.add(nums[i])

        if i >= k:
            current_sum -= nums[i - k]
            window.remove(nums[i - k])

        if len(window) == k:
            max_sum = max(max_sum, current_sum)
```

```
    return max_sum

nums = [1, 5, 4, 2, 9, 9, 9]
k = 3
print(max_sum_of_distinct_subarrays(nums, k))
```

5. **Total Cost to Hire K Workers:-**

- Time Complexity: O(nlog n)O(n \log n)O(nlogn)
- Space Complexity: O(n)O(n)O(n)

```
import heapq

def total_cost_to_hire_k_workers(costs, k, candidates):
    n = len(costs)
    left = costs[:candidates]
    right = costs[-candidates:] if candidates <= n else costs[candidates:]
    heapq.heapify(left)
    heapq.heapify(right)
    total_cost = 0

    for _ in range(k):
        if left and (not right or left[0] <= right[0]):
            total_cost += heapq.heappop(left)
        else:
            total_cost += heapq.heappop(right)

    return total_cost

costs = [3, 2, 7, 7, 1, 2]
k = 3
candidates = 2
print(total_cost_to_hire_k_workers(costs, k, candidates))
```

6. **Minimum Total Distance Traveled:-**

- Time Complexity: O(nlog n)O(n \log n)O(nlogn)
- Space Complexity: O(1)O(1)O(1)

```
python
Copy code
def min_total_distance(robot, factory):
    robot.sort()
    factory.sort(key=lambda x: x[0])
    total_distance = 0

    for r in robot:
        min_distance = float('inf')
        for f, limit in factory:
```

```
            if limit > 0:
                min_distance = min(min_distance, abs(r - f))
                limit -= 1
                break

        total_distance += min_distance

    return total_distance

robot = [1, -1]
factory = [[-2, 1], [2, 1]]
print(min_total_distance(robot, factory))
```

7. Minimum Subarrays in a Valid Split

- Time Complexity: O(nlog n)O(n \log n)O(nlogn)
- Space Complexity: O(1)O(1)O(1)

```
import math

def min_subarrays_in_valid_split(nums):
    n = len(nums)
    subarray_count = 0
    start = 0

    while start < n:
        end = start
        while end < n and math.gcd(nums[start], nums[end]) > 1:
            end += 1

        if start == end:
            return -1

        subarray_count += 1
        start = end

    return subarray_count

nums = [2, 6, 3, 4, 3]
print(min_subarrays_in_valid_split(nums))
```

8. Number of Distinct Averages:-

- Time Complexity: O(nlog n)O(n \log n)O(nlogn)
- Space Complexity: O(n)O(n)O(n)

```
def number_of_distinct_averages(nums):
    nums.sort()
    averages = set()
```

```
    while nums:
        min_val = nums.pop(0)
        max_val = nums.pop()
        avg = (min_val + max_val) / 2
        averages.add(avg)

    return len(averages)

nums = [4, 1, 4, 0, 3, 5]
print(number_of_distinct_averages(nums))
```

9. Count Ways To Build Good Strings:-

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

  dp0 = [0] * (high + 1)
  dp1 = [0] * (high + 1)

  dp0[0] = 1
  dp1[0] = 1

  for i in range(1, high + 1):
    dp0[i] = (dp0[i-1] + dp1[i-1]) % MOD
    dp1[i] = dp0[i-1]


  cumulative_sum = [0] * (high + 1)
  cumulative_sum[0] = 1  # dp0[0] and dp1[0] both contribute 1 to cumulative_sum[0]

  for i in range(1, high + 1):
    cumulative_sum[i] = (cumulative_sum[i-1] + dp0[i] + dp1[i]) % MOD
  if low == 0:
    return (cumulative_sum[high] - 1) % MOD
  else:
    return (cumulative_sum[high] - cumulative_sum[low-1]) % MOD
```

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

```
def max_profit_in_tree(n, edges, amount, bob):

    import collections




    graph = collections.defaultdict(list)

    for a, b in edges:
```

```
        graph[a].append(b)
        graph[b].append(a)



    alice_profit = [-float('inf')] * n  # Alice's max profit starting from
each node
    bob_arrival = [-1] * n  # Bob's arrival time at each node

    def dfs_bob(node, parent):
        for neighbor in graph[node]:
            if neighbor == parent:
                continue
            bob_arrival[neighbor] = bob_arrival[node] + 1
            dfs_bob(neighbor, node)



    bob_arrival[bob] = 0
    dfs_bob(bob, -1)

    def dfs_alice(node, parent):
        if amount[node] >= 0:
            alice_profit[node] = amount[node]  # Reward node
        else:
            alice_profit[node] = 0  # Cost node, assume 0 profit initially

        for neighbor in graph[node]:
            if neighbor == parent:
                continue
         neighbor
            if bob_arrival[neighbor] > bob_arrival[node]:
                # Bob reaches neighbor before Alice
                profit_to_neighbor = alice_profit[node] + amount[neighbor]
            else:
simultaneously
```

```
                profit_to_neighbor = alice_profit[node] + amount[neighbor]
// 2


to neighbor

            alice_profit[node] = max(alice_profit[node],
dfs_alice(neighbor, node))


        return alice_profit[node]


    dfs_alice(0, -1)


    max_net_income = max(alice_profit)


    return max_net_income


n = 6

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

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

bob = 3


max_income = max_profit_in_tree(n, edges, amount, bob)

print(f"Maximum net income for Alice: {max_income}")
```