1. Height of Binary Tree After Subtree Removal Queries:

Algorithm:

This problem involves modifying a binary tree by removing subtrees and calculating the height of the resulting tree.

You can approach this by performing a depth-first search (DFS) on the tree, keeping track of the height of each subtree.

When a subtree is removed, you need to update the height of the parent node accordingly.

The time complexity of this solution would be O(n + m), where n is the number of nodes in the tree and m is the number of queries.

Code:

```
class Solution:
    def heightOfBinaryTreeAfterSubtreeRemovalQueries(self, root, queries):
        def dfs(node):
            if not node:
                return 0
            left_height = dfs(node.left)
            right_height = dfs(node.right)
            return max(left_height, right_height) + 1
        def remove_subtree(node, val):
            if not node:
                return
            if node.val == val:
                node.left, node.right = None, None
            else:
                remove_subtree(node.left, val)
                remove_subtree(node.right, val)
        result = []
        for query in queries:
            remove_subtree(root, query)
            result.append(dfs(root))
        return result
```

---

2. Sort Array by Moving Items to Empty Space:

Algorithm:

This problem involves rearranging the elements in the array to sort it, using the empty space (0) to move the elements around.

You can approach this by iterating through the array and swapping the current element with the element at the index corresponding to its value.

The time complexity of this solution would be O(n), where n is the size of the input array.

Code:

```
def min_operations_to_sort(nums):
    operations = 0
    while 0 in nums:
        for i in range(len(nums) - 1, -1, -1):
            if nums[i] == 0:
                nums.pop(i)
                break
        for i in range(len(nums) - 1):
            if nums[i] > nums[i + 1]:
                nums[i], nums[i + 1] = nums[i + 1], nums[i]
                operations += 1
    return operations
```

---

3. Apply Operations to an Array:

Algorithm:

This problem involves applying a specific operation to each element of the array, based on its index.

You can approach this by iterating through the array and applying the operation to each element.

The time complexity of this solution would be O(n), where n is the size of the input array.

COde:

```
def apply_operations(nums):
    for i in range(len(nums) - 1):
        if nums[i] == nums[i + 1]:
            nums[i] *= 2
            nums[i + 1] = 0
    nums = [x for x in nums if x != 0] + [0] * (len(nums) - len(nums))
    return nums
```

---

4. Maximum Sum of Distinct Subarrays With Length K:

Algorithm:

This problem involves finding the maximum sum of a subarray of length k, where all the elements in the subarray are distinct.

You can approach this using a sliding window technique, keeping track of the distinct elements in the current window.

The time complexity of this solution would be O(n), where n is the size of the input array.

Code;

```
def max_sum_of_distinct_subarrays(nums, k):
    n = len(nums)
    max_sum = 0
    for i in range(n - k + 1):
        subarray = set(nums[i:i + k])
        if len(subarray) == k:
            max_sum = max(max_sum, sum(subarray))
    return max_sum
```

---

5. Total Cost to Hire K Workers:

Algorithm:

This problem involves finding the minimum total cost to hire k workers from a given set of workers with different costs.

You can approach this by sorting the costs array and then selecting the k workers with the lowest costs, while ensuring that the number of selected workers does not exceed the given candidates.

The time complexity of this solution would be O(n log n), where n is the size of the input array.

Code:

```
def total_cost_to_hire_workers(costs, k, candidates):
    costs.sort()
    total_cost = 0
    for _ in range(k):
        if candidates > 0:
            total_cost += costs[0]
            costs.pop(0)
            candidates -= 1
        else:
            total_cost += costs[-1]
            costs.pop()
    return total_cost
```

---

6. Minimum total distance travelled:

Algorithm:

Sort the robots by their positions.

Initialize the total distance to 0.

For each robot:

Find the nearest factory that has not reached its limit.

Calculate the distance from the robot to the factory.

Add the distance to the total distance.

Update the factory's limit.

Code:

```
def min_total_distance_traveled(robots, factories):
    robots.sort()
    total_distance = 0
```

```
    for robot in robots:
        factory = min((factory for factory in factories if factory[1] > 0), key=lambda x: abs(x[0] - robot))
        distance = abs(factory[0] - robot)
        total_distance += distance
        factory[1] -= 1
    return total_distance
```

---

7. Minimum Subarrays in a Valid Split:

Algorithm;

Algorithm:

Initialize the minimum number of subarrays to infinity.

For each possible split:

Calculate the GCD of the first and last elements of each subarray.

If the GCD is greater than 1, update the minimum number of subarrays

Code:

```
def min_subarrays_in_valid_split(nums):
    min_subarrays = float('inf')
    for i in range(1, len(nums)):
        for j in range(i, len(nums)):
            subarray = nums[i:j+1]
            gcd = 1
            for num in subarray:
                for div in range(2, num + 1):
                    if num % div == 0:
                        gcd = div
                        break
            if gcd > 1:
                min_subarrays = min(min_subarrays, j - i + 1)
    return min_subarrays if min_subarrays != float('inf') else -1
```

---

8. Number of Distinct Avererages:

Algorithm:

Algorithm:

Initialize the set of averages.

While the array is not empty:

Remove the minimum and maximum numbers.

Calculate the average of the removed numbers.

Add the average to the set.

Code;

```
def num_distinct_averages(nums):
    averages = set()
    while len(nums) > 0:
        nums.sort()
        min_num = nums.pop(0)
        max_num = nums.pop()
        average = (min_num + max_num) // 2
        averages.add(average)
    return len(averages)
```

---

9. Count Ways To Build Good Strings:

Algorithm:

Algorithm:

Initialize the count of good strings.

For each length from low to high:

Calculate the number of ways to construct a good string of the current length.

Add the count to the total count.

Code:

```
def count_ways_to_build_good_strings(low, high, zero, one):
    MOD = 10**9 + 7
    count = 0
    for length in range(low, high + 1):
        count += (zero + 1) ** (length // 2) * (one + 1) ** (length // 2)
        count %= MOD
    return count
```

---

10. Most Profitable Path in a Tree:

Algorithm:

Algorithm:

Initialize the maximum net income to negative infinity.

For each node:

Calculate the net income if Alice moves to the node.

Update the maximum net income if the calculated net income is higher.

Code:

```
def most_profitable_path_in_tree(edges, bob, amount):
    graph = {}
    for edge in edges:
        if edge[0] not in graph:
            graph[edge[0]] = []
        if edge[1] not in graph:
            graph[edge[1]] = []
        graph[edge[0]].append(edge[1])
        graph[edge[1]].append(edge[0])

    def dfs(node, parent):
        if node == bob:
            return amount[node]
```

```
    income = 0
    for neighbor in graph[node]:
        if neighbor != parent:
            income += dfs(neighbor, node)
    return income

return dfs(0, -1)
```