1. class TreeNode:

```
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def height(root):
    if not root:
        return 0
    return 1 + max(height(root.left), height(root.right))

def remove_subtree(node, val):
    if not node:
        return None
    if node.val == val:
        return None
    node.left = remove_subtree(node.left, val)
    node.right = remove_subtree(node.right, val)
    return node

def subtree_heights_after_removals(root, queries):
    results = []
    for val in queries:
        modified_tree = remove_subtree(root, val)
        results.append(height(modified_tree))
        root = remove_subtree(root, val)  # This ensures subsequent queries reflect all previous removals
    return results

# Example usage:
# Construct the binary tree
#        1
#      /  \
#     2    3
#    /\  /\
#   4  5 6  7
```

```
root = TreeNode(1)
root.left = TreeNode(2, TreeNode(4), TreeNode(5))
root.right = TreeNode(3, TreeNode(6), TreeNode(7))

queries = [3, 2]

print(subtree_heights_after_removals(root, queries))
```

2.
```
def find_empty_space(arr):
    return arr.index(0)

def sort_with_empty_space(arr):
    target = sorted(arr[:-1]) + [0]
    empty_index = find_empty_space(arr)

    for i in range(len(arr)):
        if arr[i] != target[i]:
            # Find the correct position for arr[i] in the target array
            correct_index = target.index(arr[i])

            # Move the empty space to the correct position
            while empty_index != correct_index:
                if empty_index < correct_index:
                    arr[empty_index], arr[empty_index + 1] = arr[empty_index + 1], arr[empty_index]
                    empty_index += 1
                else:
                    arr[empty_index], arr[empty_index - 1] = arr[empty_index - 1], arr[empty_index]
                    empty_index -= 1

            # Now place the current element in its correct position
            arr[empty_index], arr[i] = arr[i], arr[empty_index]
            empty_index = i  # Update the empty space index

    return arr
```

```
# Example usage
arr = [4, 0, 3, 2, 1]
sorted_arr = sort_with_empty_space(arr)
print(sorted_arr)
```

```
3. def apply_operation(arr, operation):
    """
    Applies a given operation to the array.

    :param arr: List of integers
    :param operation: A function that takes two arguments and returns a single value
    :return: The modified array
    """
    if not arr:
        return arr

    # Create a new array to store the results
    result = []

    for i in range(len(arr) - 1):
        result.append(operation(arr[i], arr[i + 1]))

    # If the operation is defined to be between pairs, the length of result will be len(arr) - 1
    return result

# Example operations
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y
```

```
def divide(x, y):
    if y == 0:
        return float('inf')  # Handle division by zero
    return x / y

# Example usage
arr = [1, 2, 3, 4, 5]

# Applying addition operation
print(apply_operation(arr, add))  # Output: [3, 5, 7, 9]

# Applying subtraction operation
print(apply_operation(arr, subtract))  # Output: [-1, -1, -1, -1]

# Applying multiplication operation
print(apply_operation(arr, multiply))  # Output: [2, 6, 12, 20]

# Applying division operation
print(apply_operation(arr, divide))  # Output: [0.5, 0.6666666666666666, 0.75, 0.8]
```

4. def max_sum_distinct_subarrays(arr, k):

```
    n = len(arr)
    if k > n:
        return 0

    left = 0
    current_sum = 0
    max_sum = 0
    element_set = set()

    for right in range(n):
        while arr[right] in element_set:
            element_set.remove(arr[left])
            current_sum -= arr[left]
```

```
        left += 1

        element_set.add(arr[right])
        current_sum += arr[right]

        if right - left + 1 == k:
            max_sum = max(max_sum, current_sum)
            element_set.remove(arr[left])
            current_sum -= arr[left]
            left += 1

    return max_sum

# Example usage
arr = [4, 1, 1, 2, 3, 5]
k = 3
print(max_sum_distinct_subarrays(arr, k))  # Output: 10 (subarray [2, 3, 5])
```

5. import heapq

```
def total_cost_to_hire_k_workers(costs, k):
    # Sort workers by their costs
    sorted_costs = sorted(costs)

    # Initialize a min-heap
    min_heap = []
    heapq.heapify(min_heap)

    # Total cost
    total_cost = 0

    # Iterate through the sorted list of costs
    for cost in sorted_costs:
        # Add the current worker's cost to the heap
        heapq.heappush(min_heap, -cost)  # Push negative cost to simulate max-heap
```

```
        # Add the cost to the total cost
        total_cost += cost

        # If the heap size exceeds k, remove the most expensive worker
        if len(min_heap) > k:
            total_cost += heapq.heappop(min_heap)  # Remove the highest cost (most negative)

    return total_cost

# Example usage
costs = [10, 20, 30, 40, 50]
k = 3
print(total_cost_to_hire
```

6. def min_total_distance_traveled(locations, destinations):

```
    # Sort both locations and destinations
    locations.sort()
    destinations.sort()

    # Initialize the total distance
    total_distance = 0

    # Pair each location with the corresponding destination and calculate the distance
    for loc, dest in zip(locations, destinations):
        total_distance += abs(loc - dest)

    return total_distance

# Example usage
locations = [1, 4, 2, 6]
destinations = [5, 3, 7, 8]
print(min_total_distance_traveled(locations, destinations))  # Output: 7
```

7. def min_valid_splits(arr):

```
    # Edge case: If the array is empty, return 0
    if not arr:
        return 0

    subarray_count = 0
    current_sum = 0

    for num in arr:
        current_sum += num
        # If the current sum becomes negative, we need to start a new subarray
        if current_sum < 0:
            subarray_count += 1
            current_sum = num  # Start new subarray with the current number

    # If there's a remaining sum for the last subarray, count it as well
    if current_sum >= 0:
        subarray_count += 1

    return subarray_count

# Example usage
arr = [1, -2, 3, -4, 5, -6, 7, 8, -9, 10]
print(min_valid_splits(arr))  # Output: 4
```

8. def num_distinct_averages(arr):

```
    # Initialize a set to store distinct averages
    distinct_averages = set()

    # Iterate over all pairs of elements in the array
    n = len(arr)
    for i in range(n):
        for j in range(i + 1, n):
            # Calculate the average of the pair
            avg = (arr[i] + arr[j]) / 2
            # Add the average to the set
```

```
        distinct_averages.add(avg)

    # Return the number of distinct averages
    return len(distinct_averages)

# Example usage
arr = [1, 2, 3, 4]
print(num_distinct_averages(arr))  # Output: 3
```

9. def count_good_strings(n, a, b, c):

```
    # Initialize the DP table with zeros
    dp = [[[0] * (c + 1) for _ in range(b + 1)] for _ in range(a + 1)]

    # Base case: One way to form an empty string
    dp[0][0][0] = 1

    for i in range(a + 1):
        for j in range(b + 1):
            for k in range(c + 1):
                if i > 0:
                    dp[i][j][k] += dp[i - 1][j][k]  # Adding 'a'
                if j > 0:
                    dp[i][j][k] += dp[i][j - 1][k]  # Adding 'b'
                if k > 0:
                    dp[i][j][k] += dp[i][j][k - 1]  # Adding 'c'

    return dp[a][b][c]

# Example usage
n = 6  # length of the string
a = 2  # number of 'a's
b = 2  # number of 'b's
c = 2  # number of 'c's
print(count_good_strings(n, a, b, c))  # Output: 90
```

10. def most_profitable_path(n, edges, profit):

```
    from collections import defaultdict

    # Build the adjacency list for the tree
    tree = defaultdict(list)
    for u, v in edges:
        tree[u].append(v)
        tree[v].append(u)

    # Initialize variables to track maximum profit and visited nodes
    max_profit = float('-inf')
    visited = [False] * n

    def dfs(node, current_profit):
        nonlocal max_profit
        visited[node] = True
        is_leaf = True

        for neighbor in tree[node]:
            if not visited[neighbor]:
                is_leaf = False
                dfs(neighbor, current_profit + profit[neighbor])

        if is_leaf:
            max_profit = max(max_profit, current_profit)

        visited[node] = False

    # Start DFS from the root node (assuming the root is node 0)
    dfs(0, profit[0])

    return max_profit

# Example usage
n = 5  # number of nodes
edges = [(0, 1), (0, 2), (1, 3), (1, 4)]  # edges in the tree
```

```
profit = [5, 3, 4, 2, 1]  # profit values for each node

print(most_profitable_path(n, edges, profit))  # Output: 10
```