Assignment - 6

1.
```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def find_maximum_XOR(root):
    def calculate_subtree_sums(node):
        if not node:
            return 0
        left_sum = calculate_subtree_sums(node.left)
        right_sum = calculate_subtree_sums(node.right)
        subtree_sum = node.val + left_sum + right_sum
        subtree_sums.append(subtree_sum)
        return subtree_sum

    def insert_trie(num):
        node = trie
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            if bit not in node:
                node[bit] = {}
            node = node[bit]

    def find_max_xor(num):
        node = trie
        max_xor = 0
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            toggle_bit = 1 - bit
            if toggle_bit in node:
                max_xor = (max_xor << 1) | 1
                node = node[toggle_bit]
```

```
            else:
                max_xor = (max_xor << 1)
                node = node[bit]
        return max_xor

    subtree_sums = []
    calculate_subtree_sums(root)

    trie = {}
    max_xor = 0
    for sum_value in subtree_sums:
        insert_trie(sum_value)
        current_xor = find_max_xor(sum_value)
        max_xor = max(max_xor, current_xor)

    return max_xor

# Example usage
# Creating a binary tree
#       1
#      / \
#     2   3
#    / \   \
#   4   5   6

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

print(find_maximum_XOR(root))  # Output will depend on the actual subtree sums
```

2. class Atom:

```
    def __init__(self, element, electrons):
        self.element = element
        self.electrons = electrons
        self.charge = 0

    def donate_electron(self):
        if self.electrons > 0:
            self.electrons -= 1
            self.charge += 1

    def accept_electron(self):
        self.electrons += 1
        self.charge -= 1

def form_ionic_bond(atom1, atom2):
    if atom1.electrons > 0:
        atom1.donate_electron()
        atom2.accept_electron()
    return atom1, atom2

# Example usage
sodium = Atom('Sodium', 1)
chlorine = Atom('Chlorine', 7)
sodium, chlorine = form_ionic_bond(sodium, chlorine)

print(f"{sodium.element}: Electrons = {sodium.electrons}, Charge = {sodium.charge}")
print(f"{chlorine.element}: Electrons = {chlorine.electrons}, Charge = {chlorine.charge}")
```

3. def minimum_cuts_to_divide_circle(n):

```
    return n - 1 if n > 1 else 0

# Example usage
n = 5
```

```
print(minimum_cuts_to_divide_circle(n))  # Output: 4
4. def difference_ones_zeros(matrix):
    rows = len(matrix)
    cols = len(matrix[0])

    row_diff = [0] * rows
    col_diff = [0] * cols

    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == 1:
                row_diff[i] += 1
                col_diff[j] += 1
            elif matrix[i][j] == 0:
                row_diff[i] -= 1
                col_diff[j] -= 1

    return row_diff, col_diff

# Example usage
matrix = [
    [1, 0, 1],
    [0, 1, 0],
    [1, 1, 0]
]

row_diff, col_diff = difference_ones_zeros(matrix)
print(f"Row differences: {row_diff}")  # Output: Row differences: [1, -1, 1]
print(f"Column differences: {col_diff}")  # Output: Column differences: [1, 1, -1]

5. def minimum_penalty(hours):
    n = len(hours)

    # Calculate the cumulative sums for opening and closing penalties
```

```
    open_penalty = [0] * (n + 1)
    close_penalty = [0] * (n + 1)

    for i in range(n):
        open_penalty[i + 1] = open_penalty[i] + (1 if hours[i] == 'N' else 0)
        close_penalty[i + 1] = close_penalty[i] + (1 if hours[i] == 'Y' else 0)

    # Find the minimum penalty
    min_penalty = float('inf')
    best_hour = 0
    for i in range(n + 1):
        penalty = open_penalty[i] + (close_penalty[n] - close_penalty[i])
        if penalty < min_penalty:
            min_penalty = penalty
            best_hour = i

    return best_hour

# Example usage
hours = "YYNYNY"
print(minimum_penalty(hours))  # Output: 2
```

6. def count_palindromic_subsequences(s):

```
    n = len(s)
    mod = 10**9 + 7

    # dp[i][j] will store the count of palindromic subsequences in s[i..j]
    dp = [[0] * n for _ in range(n)]

    for i in range(n):
        dp[i][i] = 1  # Single character is a palindromic subsequence

    for length in range(2, n + 1):  # length of the substring
        for i in range(n - length + 1):
            j = i + length - 1
```

```
            if s[i] == s[j]:
                dp[i][j] = (dp[i + 1][j] + dp[i][j - 1] + 1) % mod
            else:
                dp[i][j] = (dp[i + 1][j] + dp[i][j - 1] - dp[i + 1][j - 1]) % mod

    return dp[0][n - 1]

# Example usage
s = "abcb"
print(count_palindromic_subsequences(s))  # Output: 6
```

7.
```
def find_pivot(arr):
    total_sum = sum(arr)
    left_sum = 0

    for i in range(len(arr)):
        # total_sum - left_sum - arr[i] is the right sum
        if left_sum == (total_sum - left_sum - arr[i]):
            return i
        left_sum += arr[i]

    return -1  # Return -1 if no pivot index is found

# Example usage
arr = [1, 7, 3, 6, 5, 6]
print(find_pivot(arr))  # Output: 3

arr = [1, 2, 3]
print(find_pivot(arr))  # Output: -1

arr = [2, 1, -1]
print(find_pivot(arr))  # Output: 0
```

8.
```
class ListNode:
```

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

def remove_nodes(head, val):
    # Handle case where head node itself needs to be removed
    while head and head.val == val:
        head = head.next

    current = head

    while current:
        # Skip nodes with the specified value
        while current.next and current.next.val == val:
            current.next = current.next.next
        current = current.next

    return head

def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

# Example usage
# Create a linked list: 1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(6)
head.next.next.next = ListNode(3)
head.next.next.next.next = ListNode(4)
head.next.next.next.next.next = ListNode(5)
```

```
head.next.next.next.next.next.next = ListNode(6)

print("Original linked list:")
print_linked_list(head)

# Remove all nodes with value 6
head = remove_nodes(head, 6)

print("After removal:")
print_linked_list(head)
```

9. def count_subarrays_with_median_k(nums, k):

```
    def count_valid_subarrays(left, right):
        nonlocal count
        count += right - left + 1

    n = len(nums)
    count = 0

    for i in range(n):
        left = i
        right = i
        while right < n and nums[right] <= k:
            if nums[right] == k:
                count_valid_subarrays(left, right)
            right += 1

    return count

# Example usage
nums = [3, 1, 2, 3]
k = 2
print(count_subarrays_with_median_k(nums, k))  # Output: 3
```

10. def count_subarrays_with_median_k(nums, k):

```
    def count_valid_subarrays(left, right):
        nonlocal count
        count += right - left + 1

    n = len(nums)
    count = 0

    for i in range(n):
        left = i
        right = i
        while right < n and nums[right] <= k:
            if nums[right] == k:
                count_valid_subarrays(left, right)
            right += 1

    return count

# Example usage
nums = [3, 1, 2, 3]
k = 2
print(count_subarrays_with_median_k(nums, k))  # Output: 3
```