ASSIGNMENT-06

DATE: 11/06/2024

1.Maximum XOR

```
from collections import defaultdict
class Tree:
    def __init__(self, n, edges, values):
        self.n = n
        self.edges = edges
        self.values = values
        self.tree = defaultdict(list)
        self.subtree_sum = [0] * n
        self.construct_tree()
    def construct_tree(self):
        for u, v in self.edges:
            self.tree[u].append(v)
            self.tree[v].append(u)
    def calculate_subtree_sums(self, node, parent):
        subtree_sum = self.values[node]
        for neighbor in self.tree[node]:
            if neighbor != parent:
                subtree_sum += self.calculate_subtree_sums(neighbor, node)
        self.subtree_sum[node] = subtree_sum
        return subtree_sum
    def max_xor_of_two_subtrees(self):
        self.calculate_subtree_sums(0, -1)
        max_xor = 0
        subtree_sums = set(self.subtree_sum[1:])
        for sum_value in subtree_sums:
            for other_sum in subtree_sums:
                if sum_value != other_sum:
                    max_xor = max(max_xor, sum_value ^ other_sum)
        return max_xor
```

```
def max_xor_of_two_non_overlapping_subtrees(n, edges, values):
    if n < 2:
        return 0
    tree = Tree(n, edges, values)
    return tree.max_xor_of_two_subtrees()
n = 6
edges = [[0, 1], [0, 2], [1, 3], [1, 4], [2, 5]]
values = [2, 8, 3, 6, 2, 5]
print(max_xor_of_two_non_overlapping_subtrees(n, edges, values))
```

2.Create table

```
CREATE TABLE Elements (
    symbol VARCHAR PRIMARY KEY,
    type ENUM('Metal', 'Nonmetal', 'Noble'),
    electrons INT
);
```

3.Minimum cuts divide circle

```
def min_cuts_to_divide_circle(n):
    if n == 1:
        return 0
    return n if n % 2 == 1 else n // 2
print(min_cuts_to_divide_circle(4))
```

4.Difference between ones and zeros in row and column

```
def bestClosingTime(customers: str) -> int:
    n = len(customers)
    penalty_open = 0
    penalty_close = customers.count('Y')
    min_penalty = penalty_open + penalty_close
    best_hour = 0
```

```
    for i in range(1, n + 1):
        if customers[i - 1] == 'Y':
            penalty_close -= 1
        else:
            penalty_open += 1
        current_penalty = penalty_open + penalty_close
        if current_penalty < min_penalty:
            min_penalty = current_penalty
            best_hour = i
    return best_hour
print(bestClosingTime("YYNY"))
```

5.Minimum penalty for a shop

```
def minimum_penalty(customers):
    n = len(customers)
    min_penalty = float('inf')
    min_hour = 0
    left_N = [0] * (n + 1)
    right_Y = [0] * (n + 1)
    for i in range(1, n + 1):
        left_N[i] = left_N[i - 1] + (1 if customers[i - 1] == 'N' else 0)
    for i in range(n - 1, -1, -1):
        right_Y[i] = right_Y[i + 1] + (1 if customers[i] == 'Y' else 0)
    for j in range(n + 1):
        penalty = left_N[j] + right_Y[j]
        if penalty < min_penalty:
            min_penalty = penalty
            min_hour = j
    return min_hour
print(minimum_penalty("YYNY"))
```

6.count palindrome subsequence

```
def count_palindromic_subsequences(s):
    MOD = 10**9 + 7
    n = len(s)
    if n < 5:
        return 0
    count = 0
    for i in range(n):
        for j in range(i+1, n):
            for k in range(j+1, n):
                for l in range(k+1, n):
                    for m in range(l+1, n):
                        if s[i] == s[m] and s[j] == s[l]:
                            count = (count + 1) % MOD
    return count
print(count_palindromic_subsequences("103301"))
```

7.Pivot integer

```
def find_pivot_integer(n):
    total_sum = (n * (n + 1)) // 2
    running_sum = 0
    for x in range(1, n + 1):
        running_sum += x
        if running_sum == total_sum - running_sum + x:
            return x
    return -1
print(find_pivot_integer(8))
```

8.Append characters

```
def append_characters(s: str, t: str) -> int:
    m, n = len(s), len(t)
    j = 0
    for i in range(m):
```

```
        if j < n and s[i] == t[j]:
            j += 1
    return n - j
print(append_characters("coaching", "coding"))
```

9.Remove nodes from linked list

```
class ListNode:
    def _init_(self, val=0, next=None):
        self.val = val
        self.next = next
def remove_nodes(head: ListNode) -> ListNode:
    if not head or not head.next:
        return head
    prev, current = None, head
    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node
    max_val = float('-inf')
    current = prev
    new_head = None
    while current:
        if current.val >= max_val:
            max_val = current.val
            if new_head is None:
                new_head = ListNode(current.val)
                new_tail = new_head
            else:
                new_tail.next = ListNode(current.val)
                new_tail = new_tail.next
        current = current.next
```

```
    prev, current = None, new_head
    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node
    return prev
def print_list(head):
    while head:
        print(head.val, end=" -> ")
        head = head.next
    print("None")
head = ListNode(5, ListNode(2, ListNode(13, ListNode(3, ListNode(8)))))
new_head = remove_nodes(head)
print_list(new_head)
```

10.Count subarrays with median k

```
def count_subarrays_with_median_k(nums, k):
    n = len(nums)
    k_index = nums.index(k)
    left_counts = {0: 1}
    balance = 0
    count = 0
    for i in range(k_index, -1, -1):
        if nums[i] < k:
            balance -= 1
        elif nums[i] > k:
            balance += 1
        left_counts[balance] = left_counts.get(balance, 0) + 1
    balance = 0
    for i in range(k_index, n):
        if nums[i] < k:
```

```
            balance -= 1
        elif nums[i] > k:
            balance += 1
        count += left_counts.get(-balance, 0)
        count += left_counts.get(-balance + 1, 0)
    return count
print(count_subarrays_with_median_k([3, 2, 1, 4, 5], 4))
```