ASSIGNMENT-5 (12-06-24)

1. Convert the Temperature

```
def temperature(celsius):
    kelvin = celsius + 273.15
    fahrenheit = celsius * 1.80 + 32.00
    return [round(kelvin, 5), round(fahrenheit, 5)]

celsius = 36.50
print("Temperature:", temperature(celsius))
```

OUTPUT:
Temperature: [309.65, 97.7]

2. Number of Subarrays With LCM Equal to K

```
import math
from functools import reduce

def lcm(a, b):
    return abs(a * b) // math.gcd(a, b)

def subarraylcm(nums, k):
    count = 0
    for i in range(len(nums)):
        currentlcm = nums[i]
        for j in range(i, len(nums)):
            currentlcm = lcm(currentlcm, nums[j])
            if currentlcm == k:
                count += 1
            elif currentlcm > k:
                break
    return count

nums = [3, 6, 2, 7, 1]
k = 6
print("Subarrays with LCM Equal to K:", subarraylcm(nums, k))
```

OUTPUT:
Subarrays with LCM Equal to K: 4

3. Minimum Number of Operations to Sort a Binary Tree by Level

```
from collections import deque, defaultdict

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

def min (root):
    if not root:
        return 0

    def minswaps (arr):
```

```
        n = len(arr)
        arrpos = list(enumerate(arr))
        arrpos.sort(key=lambda it: it[1])
        vis = {k: False for k in range(n)}
        ans = 0
        for i in range(n):
            if vis[i] or arrpos[i][0] == i:
                continue
            cyclesize = 0
            j = i
            while not vis[j]:
                vis[j] = True
                j = arrpos[j][0]
                cyclesize += 1
            if cyclesize > 0:
                ans += (cyclesize - 1)
        return ans

    queue = deque([(root, 0)])
    leveldict = defaultdict(list)
    while queue:
        node, level = queue.popleft()
        leveldict[level].append(node.val)
        if node.left:
            queue.append((node.left, level + 1))
        if node.right:
            queue.append((node.right, level + 1))

    operations = 0
    for level in leveldict:
        operations += minswaps(leveldict[level])

    return operations

root = TreeNode(1)
root.left = TreeNode(4)
root.right = TreeNode(3)
root.left.left = TreeNode(7)
root.left.right = TreeNode(6)
root.right.left = TreeNode(8)
root.right.right = TreeNode(5)
root.left.left.left = None
root.left.left.right = None
root.left.right.left = None
root.left.right.right = None
root.right.left.left = TreeNode(9)
root.right.left.right = None
root.right.right.left = TreeNode(10)

print("Minimum Number of Operations to Sort Levels:", min (root))
```

OUTPUT:
Minimum Number of Operations to Sort Levels: 3

4. Maximum Number of Non-overlapping Palindrome Substrings

```
def max (s, k):
    def Palindrome(sub):
        return sub == sub[::-1]
```

```
    n = len(s)
    count = 0
    i = 0

    while i <= n - k:
        found = False
        for j in range(i + k, n + 1):
            if Palindrome(s[i:j]):
                count += 1
                i = j
                found = True
                break
        if not found:
            i += 1

    return count

s = "abaccdbbd"
k = 3
print("Max Palindrome Substrings:", max(s, k))
```

OUTPUT:
Max Palindrome Substrings: 2

5. Minimum Cost to Buy Apples

```
import heapq

def mincostapple(n, roads, applecost, k):
    graph = [[] for _ in range(n)]
    for u, v, cost in roads:
        graph[u-1].append((v-1, cost))
        graph[v-1].append((u-1, cost))

    def dijkstra(start):
        distances = [float('inf')] * n
        distances[start] = 0
        pq = [(0, start)]
        while pq:
            currentdistance, u = heapq.heappop(pq)
            if currentdistance > distances[u]:
                continue
            for v, weight in graph[u]:
                distance = currentdistance + weight
                if distance < distances[v]:
                    distances[v] = distance
                    heapq.heappush(pq, (distance, v))
        return distances

    initialdistances = [dijkstra(i) for i in range(n)]
    mincosts = []

    for i in range(n):
        mincost = float('inf')
        for j in range(n):
            cost = initialdistances[i][j] + applecost[j]
            if i != j:
                cost += initialdistances[i][j] * k
```

```
            mincost = min(mincost, cost)
        mincosts.append(mincost)

    return mincosts

n = 4
roads = [[1, 2, 4], [2, 3, 2], [2, 4, 5], [3, 4, 1], [1, 3, 4]]
apple_cost = [56, 42, 102, 301]
k = 2
print("Minimum Cost to Buy Apples:", mincostapple(n, roads, apple_cost, k))
```

OUTPUT:
Minimum Cost to Buy Apples: [54, 42, 48, 51]

6. Customers With Strictly Increasing Purchases

```
SELECT customer_id
FROM (
    SELECT customer_id, YEAR(order_date) as year, SUM(price) as total
    FROM Orders
    GROUP BY customer_id, year
) as yearly_totals
GROUP BY customer_id
HAVING MIN(yearly_totals.total) < MAX(yearly_totals.total)
ORDER BY customer_id;
```

7. Number of Unequal Triplets in Array

```
def unequaltriplets(nums):
    n = len(nums)
    count = 0
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                if nums[i] != nums[j] and nums[i] != nums[k] and nums[j] != nums[k]:
                    count += 1
    return count

nums = [4, 4, 2, 4, 3]
print("Unequal Triplets:", unequaltriplets(nums))
```

OUTPUT:
Unequal Triplets: 3

8. Closest Nodes Queries in a Binary Search Tree

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

def closestnodes(root, queries):
    def inordertraversal(node):
        if node:
            inordertraversal(node.left)
            values.append(node.val)
```

```
            inordertraversal(node.right)

    values = []
    inordertraversal(root)

    def findclosest(query):
        low, high = -1, -1
        for value in values:
            if value <= query:
                low = value
            if value >= query and high == -1:
                high = value
                break
        return [low, high]

    return [findclosest(query) for query in queries]

root = TreeNode(6)
root.left = TreeNode(2)
root.right = TreeNode
```