1. Convert the Temperature

You are given a non-negative floating point number rounded to two decimal places celsius, that denotes the temperature in Celsius.You should convert Celsius into Kelvin and

Fahrenheit and return it as an array

ans = [kelvin, fahrenheit]. Return the array ans. Answers within 10-5 of the actual answer

will be accepted. Note that:

● Kelvin = Celsius + 273.15

● Fahrenheit = Celsius * 1.80 + 32.00

Code:

```
def convert_temperature(celsius):
    kelvin = celsius + 273.15
    fahrenheit = celsius * 1.80 + 32.00
    return [kelvin, fahrenheit]
celsius = 25.75
ans = convert_temperature(celsius)
print(ans)
```

Output:

```
[298.9, 78.35]
```

2. Number of Subarrays With LCM Equal to K

Given an integer array nums and an integer k, return the number of subarrays of nums wherethe least common multiple of the subarray's elements is k.A subarray is a contiguous non- empty sequence of elements within an array.The least common multiple of an array is thesmallest positive integer that is divisible by all the array elements. Example 1: Input: nums = [3,6,2,7,1], k = 6

Code:

```
from math import gcd
from functools import reduce

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

def lcm_array(arr):
    return reduce(lcm, arr)

def count_subarrays_with_lcm(nums, k):
    count = 0
    n = len(nums)

    for start in range(n):
        current_lcm = 1
        for end in range(start, n):
            current_lcm = lcm(current_lcm, nums[end])
            if current_lcm == k:
                count += 1
    return count
nums = [3, 6, 2, 7, 1]
k = 6
result = count_subarrays_with_lcm(nums, k)
print(result)
```

Output:

```
4
```

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

You are given the root of a binary tree with unique values.In one operation, you can chooseany two nodes at the same level and swap their values.Return the minimumnumber of

operations needed to make the values at each level sorted in a strictly increasing order. The level of a node is the number of edges along the path between it and the root node

Code:

```
from collections import defaultdict, deque
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
def min_operations_to_sort(root):
    if not root:
        return 0
    level_nodes = defaultdict(list)
    queue = deque([(root, 0)])
    while queue:
        node, level = queue.popleft()
        level_nodes[level].append(node.val)

        if node.left:
            queue.append((node.left, level + 1))
        if node.right:
            queue.append((node.right, level + 1))

    min_operations = 0
    for level, nodes in level_nodes.items():
        sorted_nodes = sorted(nodes)
        for i in range(len(nodes)):
            if nodes[i] != sorted_nodes[i]:
                min_operations += 1

    return min_operations
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(8)
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
root.right.left = TreeNode(7)
root.right.right = TreeNode(9)

result = min_operations_to_sort(root)
print(result)
```

Output:

```
0
```

4. Maximum Number of Non-overlapping Palindrome Substrings

You are given a string s and a positive integer k.Select a set of non-overlapping substrings

from the string s that satisfy the following conditions:

● The length of each substring is at least k. ● Each substring is a palindrome. Return the maximum number of substrings in an optimal selection.A substring is a contiguoussequence of characters within a string. Example 1:

Input: s = "abaccdbbd", k = 3

Output: 2

Explanation: We can select the substrings underlined in s = "abaccdbbd". Both "aba" and"dbbd" are palindromes and have a length of at least k = 3. It can be shown that we cannot find a selection with more than two valid substrings

Code:

```
def max_num_of_palindromes(s: str, k: int) -> int:
    n = len(s)
    dp = [[False] * n for _ in range(n)]
    for i in range(n):
        dp[i][i] = True

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j] and (length == 2 or dp[i + 1][j - 1]):
                dp[i][j] = True
    count = 0
    i = 0
    while i < n:
        max_len = 0
        for j in range(i, n):
            if dp[i][j] and (j - i + 1) >= k:
                max_len = max(max_len, j - i + 1)

        if max_len > 0:
            count += 1
            i += max_len
        else:
            i += 1
    return count
s = "abaccdbbd"
k = 3
print(max_num_of_palindromes(s, k))
```

Output:

```
2
```

5. Minimum Cost to Buy Apples

You are given a positive integer n representing n cities numbered from 1 to n. You are alsogiven a 2D array roads, where roads[i] = [ai, bi, costi] indicates that there is a bidirectional

road between cities ai and bi with a cost of traveling equal to costi. You can buy apples in any city you want, but some cities have different costs to buy apples. You are given the array appleCost where appleCost[i] is the cost of buying one apple fromcity i. You start at some city, traverse through various roads, and eventually buy exactly one applefrom any city. After you buy that apple, you have to return back to the city you startedat, but

now the cost of all the roads will be multiplied by a given factor k. Given the integer k, return an array answer of size n where answer[i] is the minimumtotal

cost to buy an apple if you start at city i.

Code:

```
import heapq
from collections import defaultdict, deque
def minimumCostToBuyApples(n, roads, appleCost, k):
    graph = defaultdict(list)
    for a, b, cost in roads:
        graph[a].append((b, cost))
        graph[b].append((a, cost))
    def dijkstra(start):
        pq = [(0, start)]
        dist = [float('inf')] * (n + 1)
        dist[start] = 0

        while pq:
            curr_dist, u = heapq.heappop(pq)
            if curr_dist > dist[u]:
                continue
            for v, cost in graph[u]:
                if dist[u] + cost < dist[v]:
                    dist[v] = dist[u] + cost
                    heapq.heappush(pq, (dist[v], v))

        return dist
    answer = [float('inf')] * n

    for i in range(1, n + 1):
        dist_from_i = dijkstra(i)
        for j in range(1, n + 1):
            if i != j:
                total_cost = appleCost[i - 1] + dist_from_i[j] * k
                answer[i - 1] = min(answer[i - 1], total_cost)

    return answer
n = 4
roads = [[1, 2, 1], [3, 4, 1], [2, 3, 1]]
appleCost = [1, 2, 3, 4]
k = 2
print(minimumCostToBuyApples(n, roads, appleCost, k))
```

Output:

6. Customers With Strictly Increasing Purchases

SQL Schema

Table: Orders

+--------------+------+

| Column Name | Type |

+--------------+------+

| order_id | int |

| customer_id | int |

| order_date | date |

| price | int |

+--------------+------+

order_id is the primary key for this table. Each row contains the id of an order, the id of customer that ordered it, the date of the order, and its price. Write an SQL query to report the IDs of the customers with the total purchases strictlyincreasing yearly. ● The total purchases of a customer in one year is the sum of the prices of their ordersinthat year. If for some year the customer did not make any order, we consider the total

purchases 0. ● The first year to consider for each customer is the year of their first order. ● The last year to consider for each customer is the year of their last order. Return the result table in any order. The query result format is in the following example.

Code:

```
import sqlite3
conn = sqlite3.connect('your_database.db')
cursor = conn.cursor()
query = """
WITH CustomerFirstLastYear AS (
    SELECT
        customer_id,
        MIN(YEAR(order_date)) AS first_order_year,
        MAX(YEAR(order_date)) AS last_order_year
    FROM
        Orders
    GROUP BY
        customer_id
),

CustomerTotalPurchases AS (
    SELECT
        customer_id,
        YEAR(order_date) AS order_year,
        SUM(price) AS total_purchases
    FROM
        Orders
    GROUP BY
        customer_id,
        YEAR(order_date)
),

CustomerYearlyIncrease AS (
    SELECT
        c.customer_id
    FROM
        CustomerFirstLastYear c
        LEFT JOIN CustomerTotalPurchases p ON c.customer_id = p.customer_id
                                          AND p.order_year >= c.first_order_year
                                          AND p.order_year <= c.last_order_year
    GROUP BY
        c.customer_id
    HAVING
        COUNT(DISTINCT p.order_year) = COUNT(DISTINCT CASE WHEN LEAD(p.total_purchases) OVER (PARTITION BY c.customer_id ORDER BY p.order_year) > p.total_purchases THEN p.order_year END)
)

SELECT
    customer_id
FROM
    CustomerYearlyIncrease;
"""
```

Output:

7. Number of Unequal Triplets in Array

You are given a 0-indexed array of positive integers nums. Find the number of triplets (i, j, k)that meet the following conditions:

- **0 <= i < j < k < nums.length**
- **nums[i], nums[j], and nums[k] are pairwise distinct. ○ In other words, nums[i] != nums[j], nums[i] != nums[k], and nums[j] !=nums[k]. Return the number of triplets that meet the conditions.**

Code:

```
def numUnequalTriplets(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 = [1, 2, 3, 4, 5]
print(numUnequalTriplets(nums))
```

Output:

```
10
```

8. Closest Nodes Queries in a Binary Search Tree

You are given the root of a binary search tree and an array queries of size n consistingof

positive integers. Find a 2D array answer of size n where answer[i] = [mini, maxi]:

● mini is the largest value in the tree that is smaller than or equal to queries[i]. If a suchvalue does not exist, add -1 instead. ● maxi is the smallest value in the tree that is greater than or equal to queries[i]. If asuch value does not exist, add -1 instead. Return the array answer.

Code:

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

def closestNodesQueries(root, queries):
    def findClosest(root, target):
        mini = -1
        maxi = -1

        while root:
            if root.val == target:
                return root.val, root.val
            elif root.val < target:
                mini = root.val
                root = root.right
            else:
                maxi = root.val
                root = root.left

        return mini, maxi

    result = []

    for query in queries:
        mini, maxi = findClosest(root, query)
        result.append([mini, maxi])

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

queries = [3, 1, 5, 6]
print(closestNodesQueries(root, queries))
```

Output:

```
[[3, 3], [1, 1], [5, 5], [5, -1]]
```

9. Minimum Fuel Cost to Report to the Capital

There is a tree (i.e., a connected, undirected graph with no cycles) structure country networkconsisting of n cities numbered from 0 to n - 1 and exactly n - 1 roads. The capital cityis city0. You are given a 2D integer array roads where roads[i] = [ai, bi] denotes that there existsabidirectional road connecting cities ai and bi. There is a meeting for the representatives of each city. The meeting is in the capital

city.There is a car in each city. You are given an integer seats that indicates the number of

seats in each car.A representative can use the car in their city to travel or change the car andride with another representative. The cost of traveling between two cities is one liter of fuel. Return the minimum number of liters of fuel to reach the capital city.

Code:

```
from collections import deque

def minFuelCostToCapital(n, roads):
    graph = [[] for _ in range(n)]
    for a, b in roads:
        graph[a].append(b)
        graph[b].append(a)

    queue = deque([0])
    visited = set([0])
    fuel = 0
    while queue:
        size = len(queue)
        for _ in range(size):
            current = queue.popleft()
            for neighbor in graph[current]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
        fuel += 1

    return fuel - 1
n = 6
roads = [[0, 1], [0, 2], [1, 3], [1, 4], [2, 5]]
print(minFuelCostToCapital(n, roads))
```

Output:

```
2
```

10. Number of Beautiful Partitions

You are given a string s that consists of the digits '1' to '9' and two integers k and minLength. A partition of s is called beautiful if:

● s is partitioned into k non-intersecting substrings. ● Each substring has a length of at least minLength. ● Each substring starts with a prime digit and ends with a non-prime digit. Prime digitsare '2', '3', '5', and '7', and the rest of the digits are non-prime. Return the number of beautiful partitions of s. Since the answer may be very large, returnit

modulo 109 + 7.A substring is a contiguous sequence of characters within a string.

Code:

```
def countBeautifulPartitions(s, k, minLength):
    MOD = 10**9 + 7
    n = len(s)
    prime_digits = {'2', '3', '5', '7'}
    non_prime_digits = {'1', '4', '6', '8', '9'}
    dp = [0] * (n + 1)
    dp[0] = 1
    for i in range(1, n + 1):
        for j in range(1, min(i, k) + 1):
            if i >= minLength:
                if s[i - 1] in prime_digits and s[i - minLength] in non_prime_digits:
                    dp[i] = (dp[i] + dp[i - minLength]) % MOD
            dp[i] = (dp[i] + dp[i - 1]) % MOD

    return dp[n]
s = "325"
k = 2
minLength = 1
print(countBeautifulPartitions(s, k, minLength))
```

Output:

```
4
```