**ASSIGNMENT – 7**

**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**

def convert\_temperature(celsius):

kelvin = celsius + 273.15

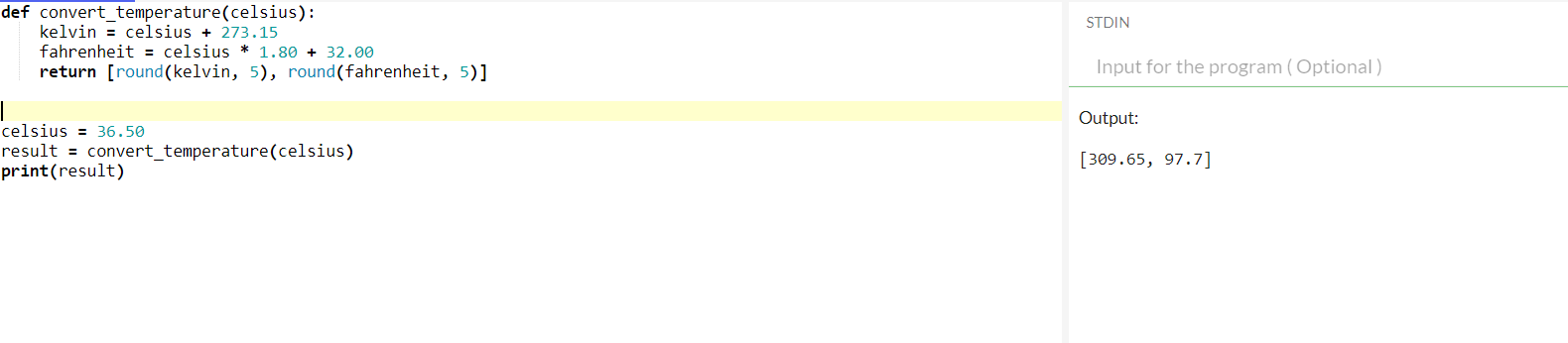
fahrenheit = celsius \* 1.80 + 32.00

return [round(kelvin, 5), round(fahrenheit, 5)]

# Example usage

celsius = 36.50

result = convert\_temperature(celsius)

print(result)

**2) Given an integer array nums and an integer k, return the number of subarrays of nums where the 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 the smallest positive integer that is divisible by all the array elements.**

import math

from functools import reduce

from math import gcd

def lcm(x, y):

return abs(x \* y) // gcd(x, y)

def lcm\_of\_list(lst):

return reduce(lcm, lst)

def count\_subarrays\_with\_lcm(nums, k):

count = 0

n = len(nums)

for i in range(n):

current\_lcm = 1

for j in range(i, n):

current\_lcm = lcm(current\_lcm, nums[j])

if current\_lcm == k:

count += 1

if current\_lcm > k:

break

return count

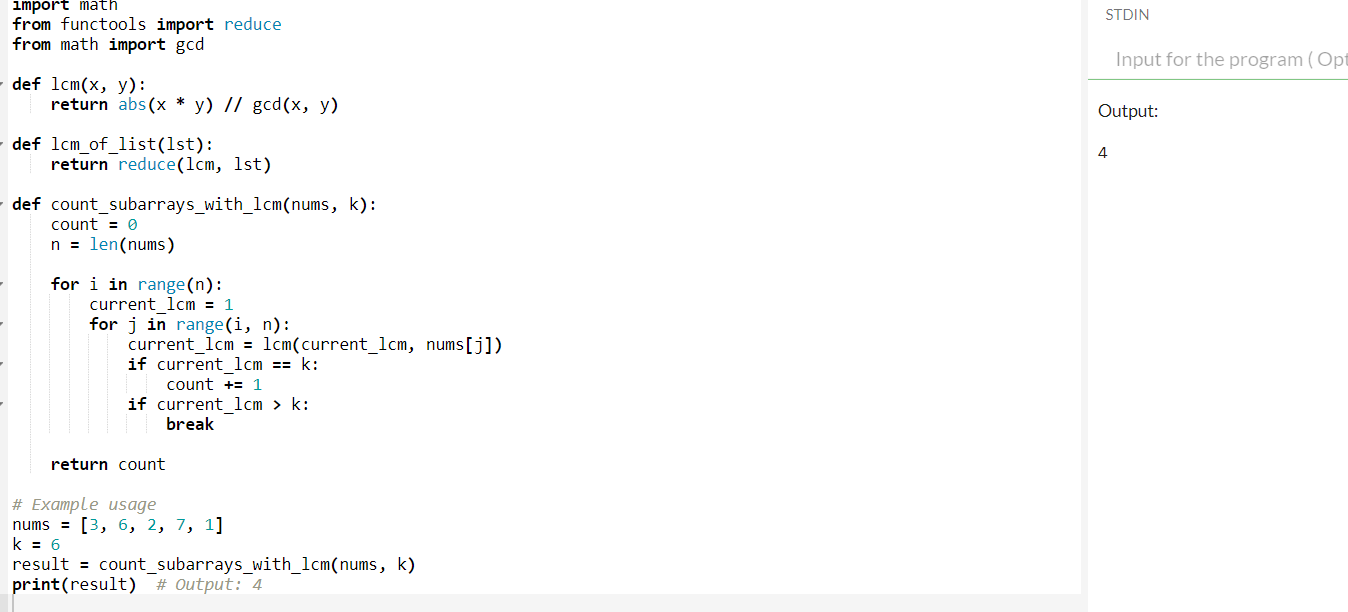
# Example usage

nums = [3, 6, 2, 7, 1]

k = 6

result = count\_subarrays\_with\_lcm(nums, k)

print(result) # Output: 4



**3) You are given the root of a binary tree with unique values.In one operation, you can choose any two nodes at the same level and swap their values.Return the minimum number 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.**

from collections import deque

# Definition for a binary tree node.

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def minSwapsToSort(arr):

n = len(arr)

arrpos = [(value, index) for index, value in enumerate(arr)]

arrpos.sort()

vis = {k: False for k in range(n)}

ans = 0

for i in range(n):

if vis[i] or arrpos[i][1] == i:

continue

cycle\_size = 0

x = i

while not vis[x]:

vis[x] = True

x = arrpos[x][1]

cycle\_size += 1

if cycle\_size > 0:

ans += (cycle\_size - 1)

return ans

def minOperationsToSortLevels(root):

if not root:

return 0

queue = deque([root])

min\_operations = 0

while queue:

level\_size = len(queue)

level\_nodes = []

for \_ in range(level\_size):

node = queue.popleft()

level\_nodes.append(node.val)

if node.left:

queue.append(node.left)

if node.right:

queue.append(node.right)

min\_operations += minSwapsToSort(level\_nodes)

return min\_operations

# Example usage:

# Construct the binary tree from the given example

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)

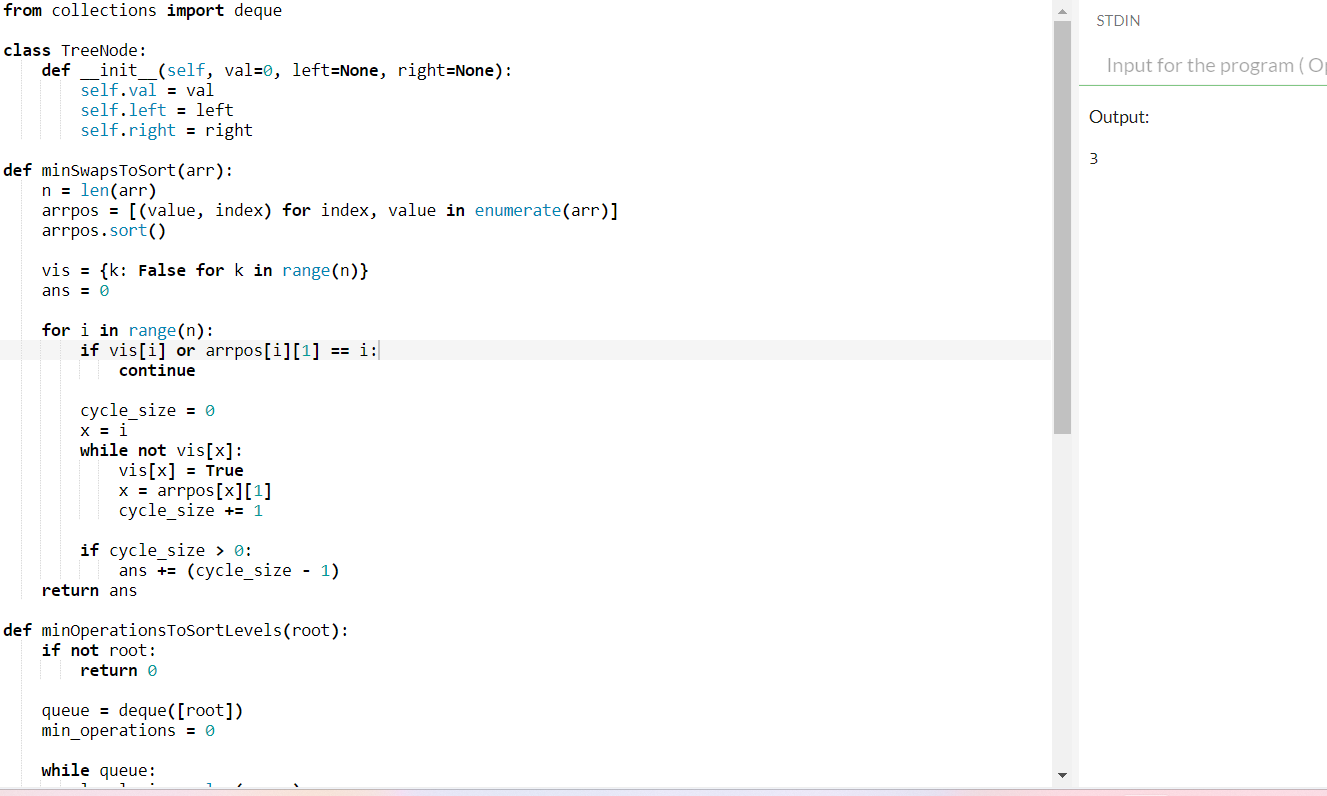
root.right.left.left = TreeNode(9)

root.right.left.right = TreeNode(10)

# Calculate the minimum number of operations

result = minOperationsToSortLevels(root)

print(result)



**4) 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. ● Eachsubstring is a palindrome. Return the maximum number of substrings in an optimal selection.A substring is a contiguous sequence of characters within a string.**

def maxPalindromicSubstrings(s, k):

n = len(s)

dp = [[False] \* n for \_ in range(n)]

# Populate dp table for palindromic substrings

for length in range(1, n+1):

for i in range(n-length+1):

j = i + length - 1

if length == 1:

dp[i][j] = True

elif length == 2:

dp[i][j] = (s[i] == s[j])

else:

dp[i][j] = (s[i] == s[j] and dp[i+1][j-1])

palindromic\_substrings = []

for i in range(n):

for j in range(i + k - 1, n):

if dp[i][j]:

palindromic\_substrings.append((i, j))

# Sort by end index to facilitate non-overlapping selection

palindromic\_substrings.sort(key=lambda x: x[1])

count = 0

last\_end = -1

for start, end in palindromic\_substrings:

if start > last\_end:

count += 1

last\_end = end

return count

# Example usage

s = "abaccdbbd"

k = 3

print(maxPalindromicSubstrings(s, k))



**5) 5. Minimum Cost to Buy Apples You are given a positive integer n representing n cities numbered from 1 to n. You are also given 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 from city i.**

import heapq

def dijkstra(n, graph, start):

dist = [float('inf')] \* (n + 1)

dist[start] = 0

min\_heap = [(0, start)]

while min\_heap:

current\_dist, u = heapq.heappop(min\_heap)

if current\_dist > dist[u]:

continue

for v, weight in graph[u]:

distance = current\_dist + weight

if distance < dist[v]:

dist[v] = distance

heapq.heappush(min\_heap, (distance, v))

return dist

def min\_total\_cost(n, roads, appleCost, k):

graph = [[] for \_ in range(n + 1)]

for u, v, cost in roads:

graph[u].append((v, cost))

graph[v].append((u, cost))

all\_costs = []

for i in range(1, n + 1):

dist\_from\_i = dijkstra(n, graph, i)

min\_cost = float('inf')

for j in range(1, n + 1):

if i != j:

cost = dist\_from\_i[j] + appleCost[j - 1] + dist\_from\_i[j] \* k

min\_cost = min(min\_cost, cost)

else:

min\_cost = min(min\_cost, appleCost[j - 1])

all\_costs.append(min\_cost)

return all\_costs

# Example usage

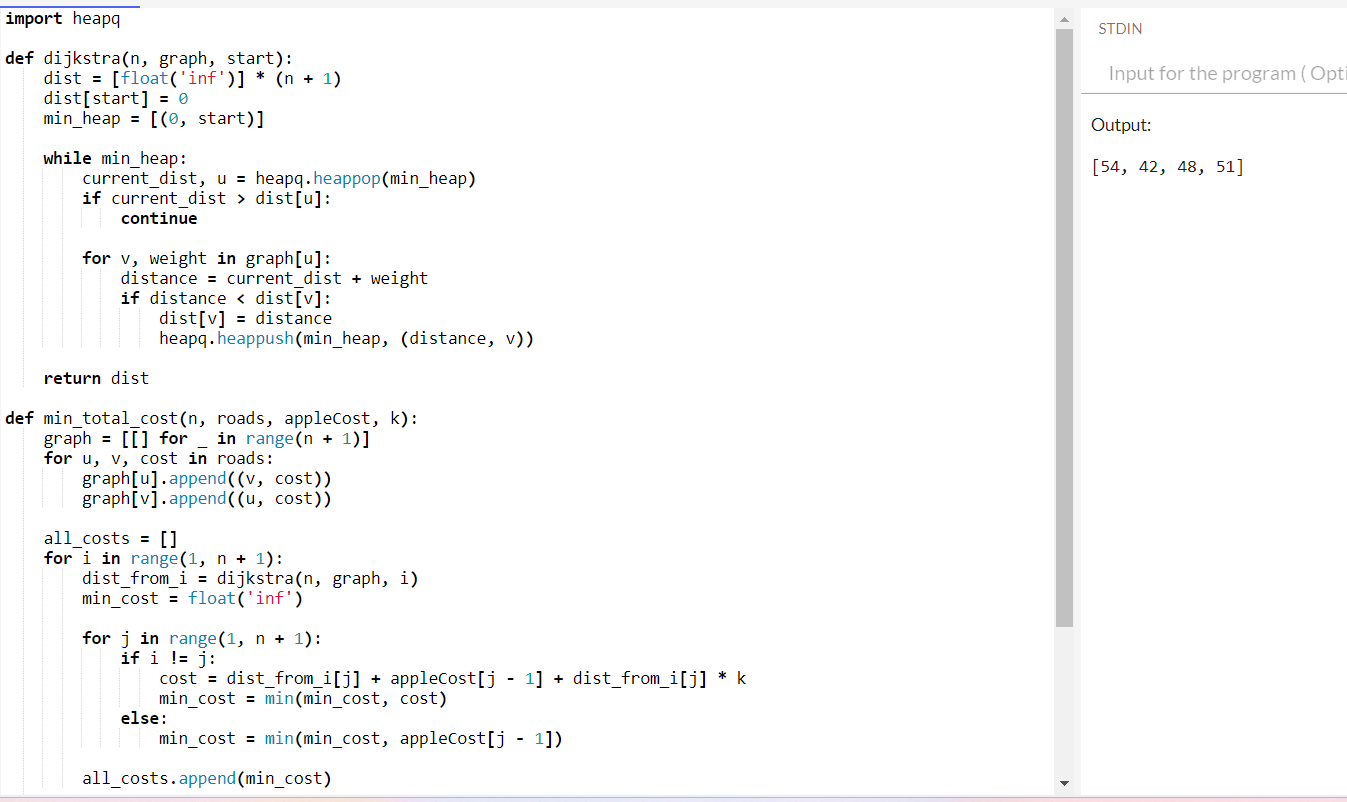
n = 4

roads = [[1, 2, 4], [2, 3, 2], [2, 4, 5], [3, 4, 1], [1, 3, 4]]

appleCost = [56, 42, 102, 301]

k = 2

print(min\_total\_cost(n, roads, appleCost, k))



**6) 6. Customers With Strictly Increasing Purchases SQLSchema 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 strictly increasing yearly. ● Thetotal purchases of a customer in one year is the sum of the prices of their orders in that year. If for some year the customer did not make any order, we consider the total purchases 0. ● Thefirst year to consider for each customer is the year of their first order. ● Thelast year to consider for each customer is the year of their last order. Return the result table in any order.**

import sqlite3

# Create a new SQLite database (or connect to an existing one)

conn = sqlite3.connect(':memory:')

cursor = conn.cursor()

# Create the Orders table

cursor.execute('''

CREATE TABLE Orders (

order\_id INTEGER PRIMARY KEY,

customer\_id INTEGER,

order\_date DATE,

price INTEGER

)

''')

# Insert sample data into the Orders table

orders = [

(1, 1, '2019-07-01', 1100),

(2, 1, '2019-11-01', 1200),

(3, 1, '2020-05-26', 3000),

(4, 1, '2021-08-31', 3100),

(5, 1, '2022-12-07', 4700),

(6, 2, '2015-01-01', 700),

(7, 2, '2017-11-07', 1000),

(8, 3, '2017-01-01', 900),

(9, 3, '2018-11-07', 900)

]

cursor.executemany('INSERT INTO Orders (order\_id, customer\_id, order\_date, price) VALUES (?, ?, ?, ?)', orders)

# Define and execute the SQL query

query = '''

WITH YearlyPurchases AS (

SELECT

customer\_id,

strftime('%Y', order\_date) AS year,

SUM(price) AS total\_price

FROM Orders

GROUP BY customer\_id, year

),

CustomerYears AS (

SELECT

customer\_id,

MIN(year) AS start\_year,

MAX(year) AS end\_year

FROM YearlyPurchases

GROUP BY customer\_id

),

YearRange AS (

SELECT

customer\_id,

year

FROM CustomerYears

JOIN (

SELECT DISTINCT strftime('%Y', order\_date) AS year

FROM Orders

) AS Yrs

ON year BETWEEN start\_year AND end\_year

),

PurchasesWithGaps AS (

SELECT

yr.customer\_id,

yr.year,

COALESCE(yp.total\_price, 0) AS total\_price

FROM YearRange yr

LEFT JOIN YearlyPurchases yp

ON yr.customer\_id = yp.customer\_id AND yr.year = yp.year

),

RankedPurchases AS (

SELECT

customer\_id,

year,

total\_price,

ROW\_NUMBER() OVER (PARTITION BY customer\_id ORDER BY year) AS rn

FROM PurchasesWithGaps

),

Comparison AS (

SELECT

a.customer\_id,

a.total\_price,

b.total\_price AS prev\_total\_price

FROM RankedPurchases a

LEFT JOIN RankedPurchases b

ON a.customer\_id = b.customer\_id AND a.rn = b.rn + 1

)

SELECT DISTINCT customer\_id

FROM Comparison

GROUP BY customer\_id

HAVING COUNT(\*) = COUNT(CASE WHEN total\_price > prev\_total\_price THEN 1 END);

'''

# Execute the query and fetch results

cursor.execute(query)

result = cursor.fetchall()

# Print the result

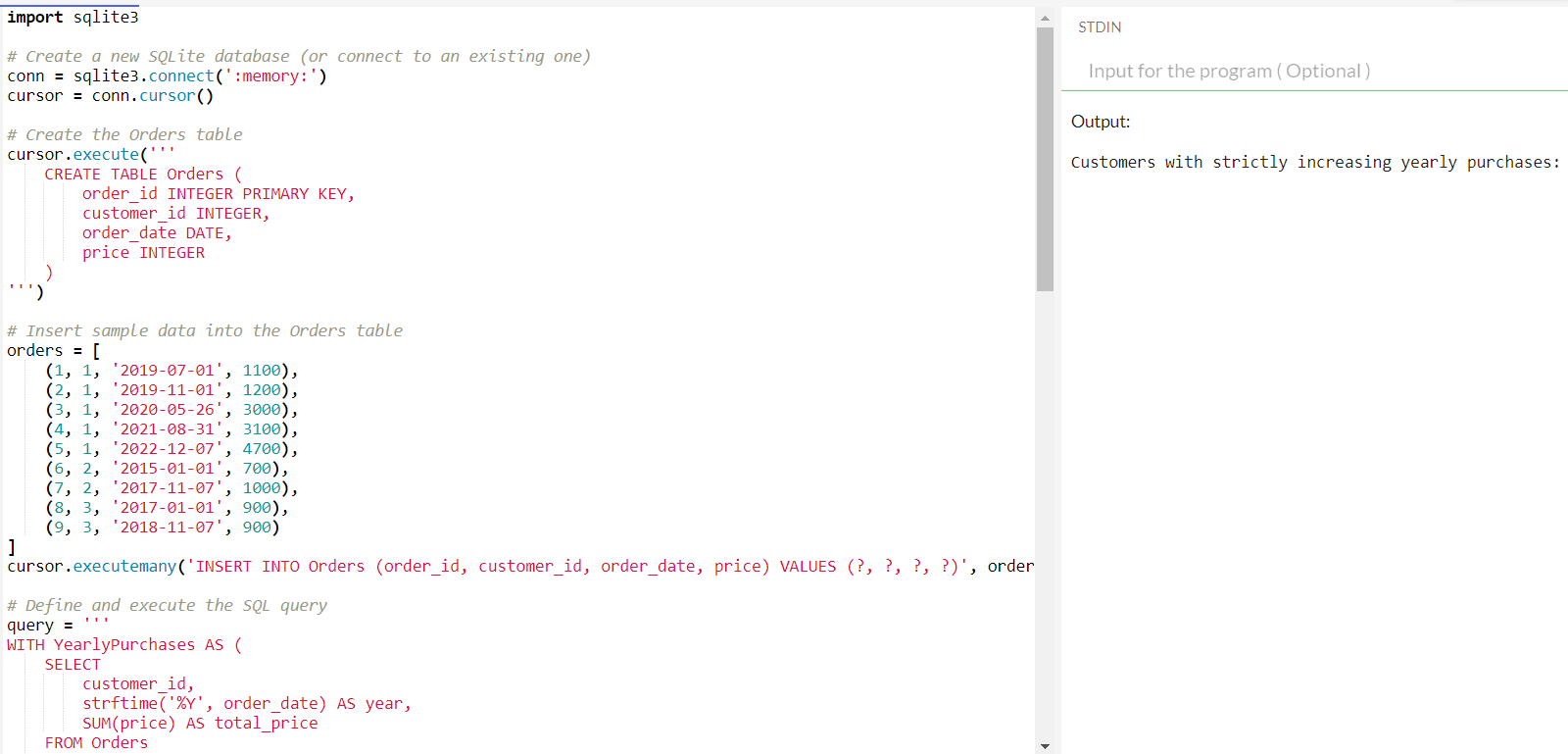
print("Customers with strictly increasing yearly purchases:")

for row in result:

print(row[0])

# Close the connection

conn.close()



**7) 7.NumberofUnequalTripletsinArray Youaregivena0-indexedarrayofpositiveintegersnums.Findthenumberoftriplets(i,j,k) thatmeetthefollowingconditions: ● 0<=i<j<k**

def countUnequalTriplets(nums):

count = 0

n = len(nums)

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]:

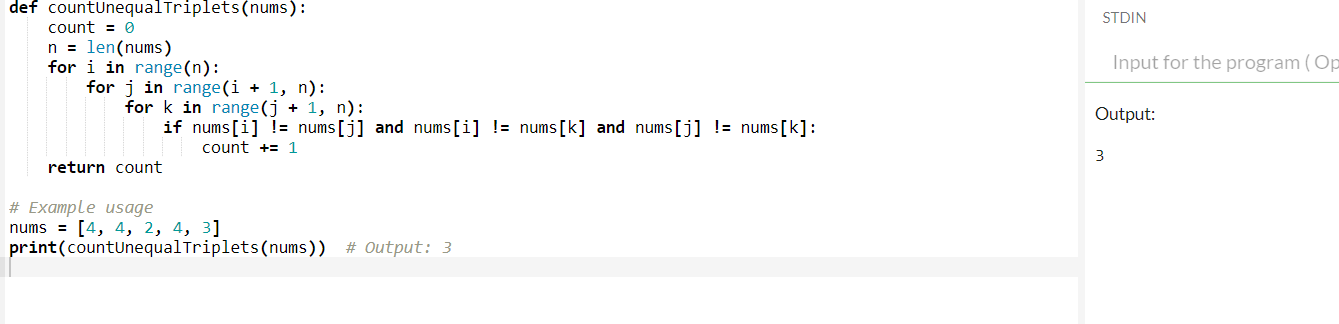
count += 1

return count

# Example usage

nums = [4, 4, 2, 4, 3]

print(countUnequalTriplets(nums))



8) 8**.ClosestNodesQueriesinaBinarySearchTree Youaregiventherootofabinarysearchtreeandanarrayqueriesofsizenconsistingof positiveintegers. Finda2Darrayanswerofsizenwhereanswer[i]=[mini,maxi]: ● miniisthelargestvalueinthetreethatissmallerthanorequaltoqueries[i].Ifasuch valuedoesnotexist,add-1instead. ● maxiisthesmallestvalueinthetreethatisgreaterthanorequaltoqueries[i].Ifa suchvaluedoesnotexist,add-1instead.**

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def inorder\_traversal(root):

# Perform an inorder traversal to get the node values in sorted order

result = []

stack = []

current = root

while stack or current:

while current:

stack.append(current)

current = current.left

current = stack.pop()

result.append(current.val)

current = current.right

return result

def closest\_nodes(root, queries):

# Get the sorted node values from the BST

sorted\_values = inorder\_traversal(root)

def find\_closest\_values(query):

import bisect

pos = bisect.bisect\_left(sorted\_values, query)

mini = -1 if pos == 0 else sorted\_values[pos - 1]

maxi = -1 if pos == len(sorted\_values) else sorted\_values[pos]

if pos < len(sorted\_values) and sorted\_values[pos] == query:

mini = maxi = sorted\_values[pos]

return [mini, maxi]

# For each query, find the closest values

return [find\_closest\_values(query) for query in queries]

# Example usage

root = TreeNode(6)

root.left = TreeNode(2)

root.right = TreeNode(13)

root. left.left = TreeNode(1)

root.left.right = TreeNode(4)

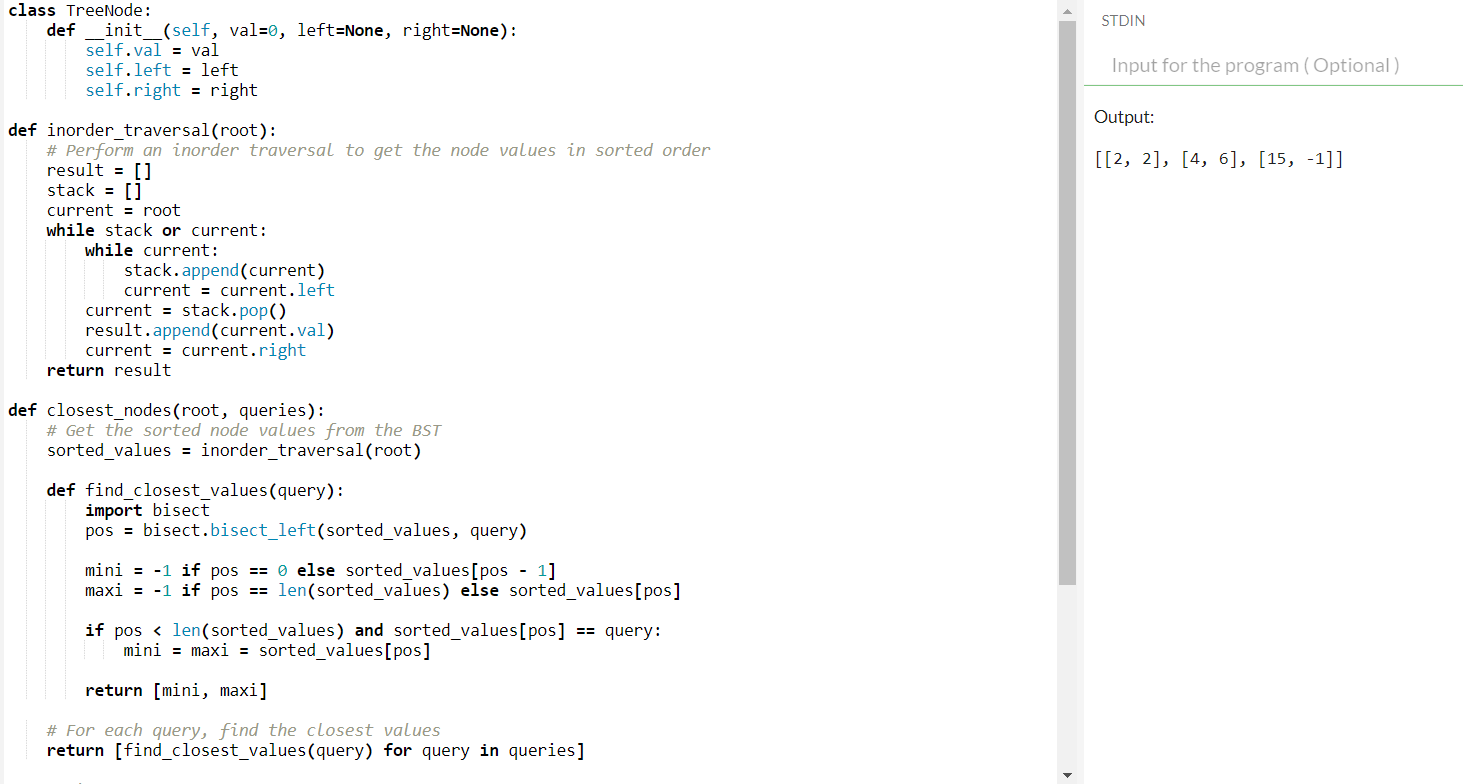
root.right.left = TreeNode(9)

root.right.right = TreeNode(15)

root.right.right.left = TreeNode(14)

queries = [2, 5, 16]

print(closest\_nodes(root, queries))



9) **9.MinimumFuelCosttoReporttotheCapital Thereisatree(i.e.,aconnected,undirectedgraphwithnocycles)structurecountrynetwork consistingofncitiesnumberedfrom0ton-1andexactlyn-1roads.Thecapitalcityiscity 0.Youaregivena2Dintegerarrayroadswhereroads[i]=[ai,bi]denotesthatthereexistsa bidirectionalroadconnectingcitiesaiandbi.**

from collections import deque, defaultdict

def minFuel(roads, seats):

n = len(roads) + 1 # number of cities

graph = defaultdict(list)

# Build the adjacency list representation of the graph

for road in roads:

u, v = road[0], road[1]

graph[u].append(v)

graph[v].append(u)

# BFS initialization

queue = deque([0])

visited = [False] \* n

visited[0] = True

fuel\_needed = [0] \* n

while queue:

city = queue.popleft()

for neighbor in graph[city]:

if not visited[neighbor]:

visited[neighbor] = True

queue.append(neighbor)

# Calculate fuel needed to reach neighbor

if seats[neighbor] >= len(graph[neighbor]) + 1:

fuel\_needed[neighbor] = fuel\_needed[city]

else:

fuel\_needed[neighbor] = fuel\_needed[city] + 1

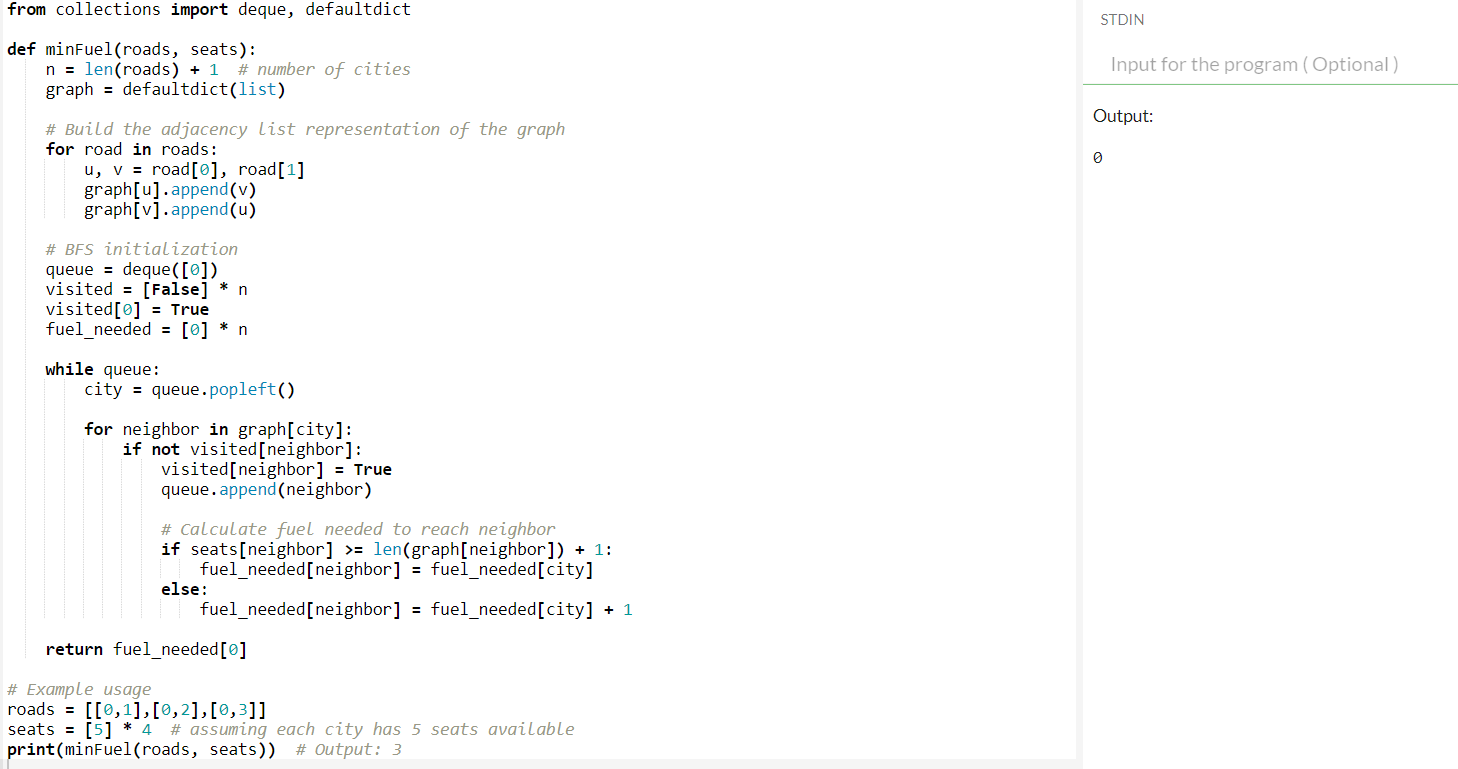
return fuel\_needed[0]

# Example usage

roads = [[0,1],[0,2],[0,3]]

seats = [5] \* 4 # assuming each city has 5 seats available

print(minFuel(roads, seats)) # Output: 3



10) You are given a string s that consists of the digits '1' to '9' and two integers k and minLength. Apartition of s is called beautiful if: ● s is partitioned into k non-intersecting substrings. ● Each substring has a length of at least minLength. ● Eachsubstring starts with a prime digit and ends with a non-prime digit. Prime digits are '2', '3', '5', and '7', and the rest of the digits are non-prime.

def count\_beautiful\_partitions (s, k, minLength):

MOD = 10\*\*9 + 7

primes = {'2', '3', '5', '7'}

def is\_prime(char):

return char in primes

def is\_non\_prime(char):

return not is\_prime(char)

def is\_valid\_partition(partition):

return is\_prime(partition[0]) and is\_non\_prime(partition[-1]) and len(partition) >= minLength

def count\_partitions(s, k, start, current\_partition):

if k == 0:

return 1 if start == len(s) else 0

count = 0

for i in range(start, len(s)):

current\_partition += s[i]

if is\_valid\_partition(current\_partition):

count = (count + count\_partitions(s, k - 1, i + 1, "")) % MOD

return count

return count\_partitions(s, k, 0, "")

# Example

s = "23542185131"

k = 3

minLength = 2

print(count\_beautiful\_partitions(s, k, minLength))

