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 * 1.8 + 32.00

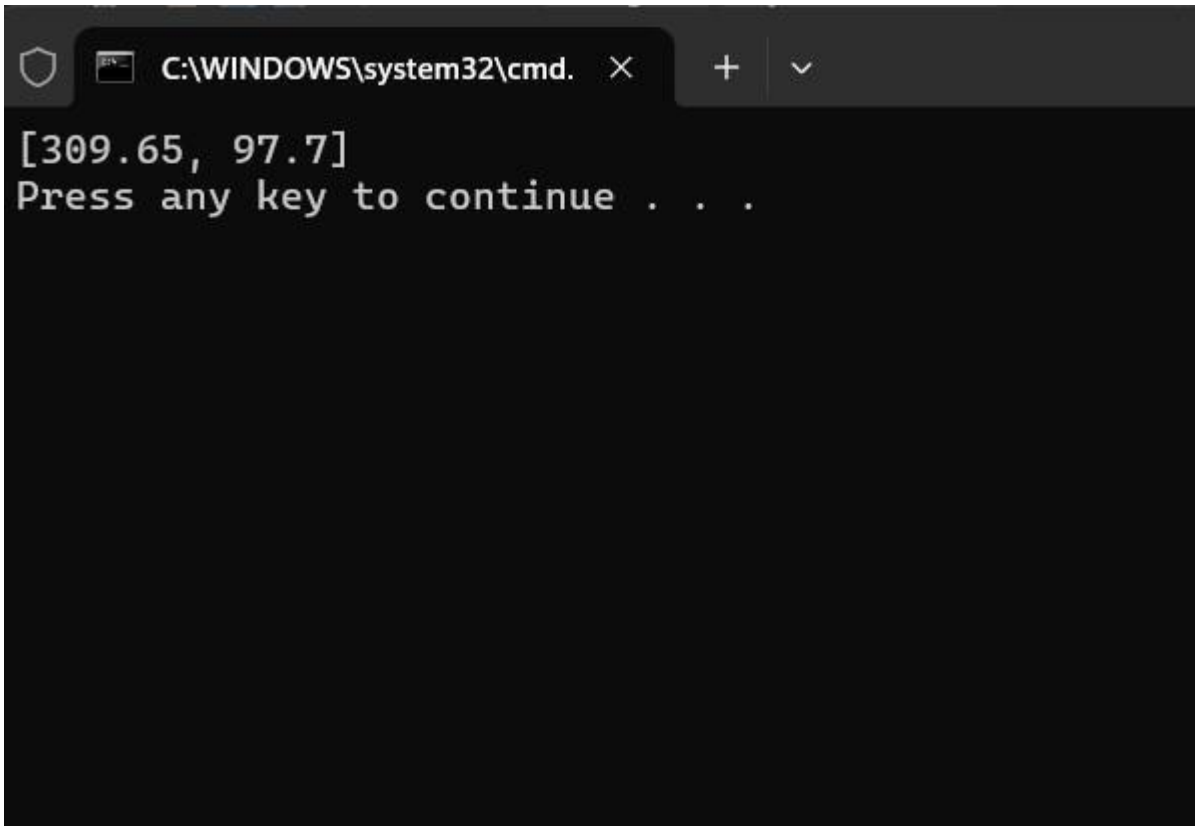
Example 1: Input: celsius = 36.50

Output: [309.65000,97.70000]

PROGRAM:

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

celsius = 36.50  result =
convert_temperature(celsius)
print(result)
```

OUTPUT:



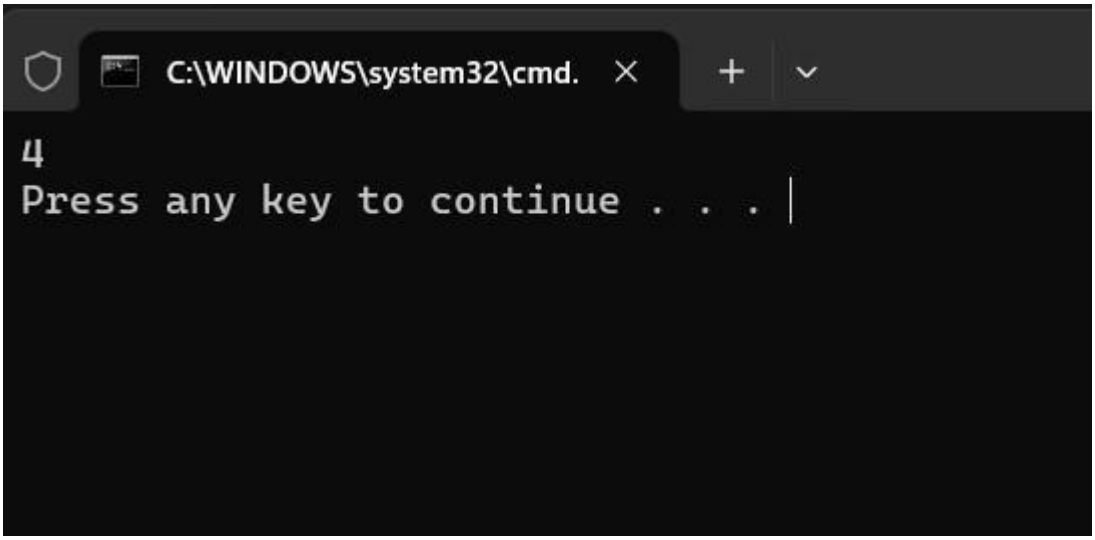
2) Number of Subarrays With LCM Equal to K Given an integer array nums and an integer k, return the number of subarrays of nums wherethe least common multiple of the subarray's elements is k.A subarray is a contiguous non- empty sequence of elements within an array.The least common multiple of an array is thesmallest positive integer that is divisible by all the array elements.

PROGRAM:

```
from math import gcd from
functools import reduce
 def lcm(a, b):     return abs(a * b) //
gcd(a, b)
```

```
def lcm_list(numbers):     return
reduce(lcm, numbers)
def count_subarrays_with_lcm_k(nums, k):
count = 0     n = len(nums)
        for i in range(n):
       current_lcm = nums[i]         for j
in range(i, n):
          current_lcm = lcm(current_lcm, nums[j])            if
current_lcm == k:
              count += 1            elif
current_lcm > k:                  break

   return count

nums1 = [3, 6, 2, 7, 1] k1 = 6
print(count_subarrays_with_lcm_k(nums1, k1))
```

OUTPUT:



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

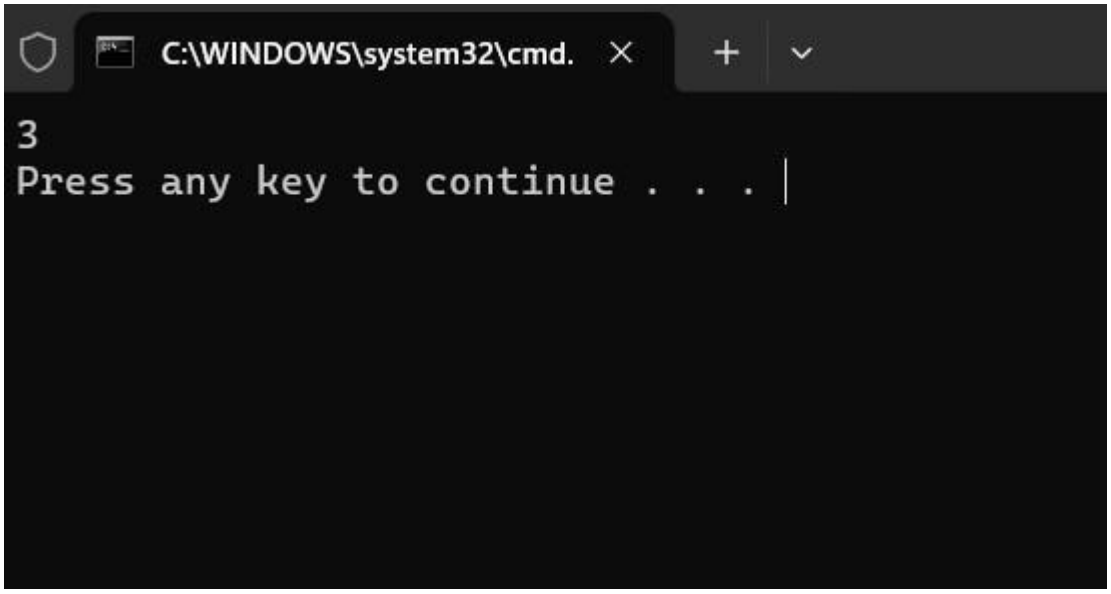
their values.Return the minimumnumber of operations needed to make the values at each level sorted in a strictly increasing order.

PROGRAM:

```
from collections import deque
 class TreeNode:     def __init__(self, val=0, left=None,
right=None):         self.val = val         self.left = left
self.right = right
 def minSwaps(arr):
n = len(arr)
    arrpos = [(arr[i], i) for i in range(n)]
arrpos.sort()      visited = {k: False for k in
range(n)}     swaps = 0     for i in range(n):
if visited[i] or arrpos[i][1] == i:
continue         cycle_size = 0         x = i
while not visited[x]:             visited[x] =
True             x = arrpos[x][1]
cycle_size += 1         if cycle_size > 0:
swaps += (cycle_size - 1)     return swaps
 def
minOperationsToSortTree(root):
if not root:         return 0


    queue = deque([root])     operations
=0          while queue:
        level_size = len(queue)
current_level = []                  for _ in
range(level_size):
node = queue.popleft()
current_level.append(node.
val)             if node.left:
queue.append(node.left)
if node.right:
                queue.append(node.right)


        operations += minSwaps(current_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.right.left.left =
TreeNode(9)
root.right.right.left = TreeNode(10)

print(minOperationsToSortTree(root))
```

OUTPUT:

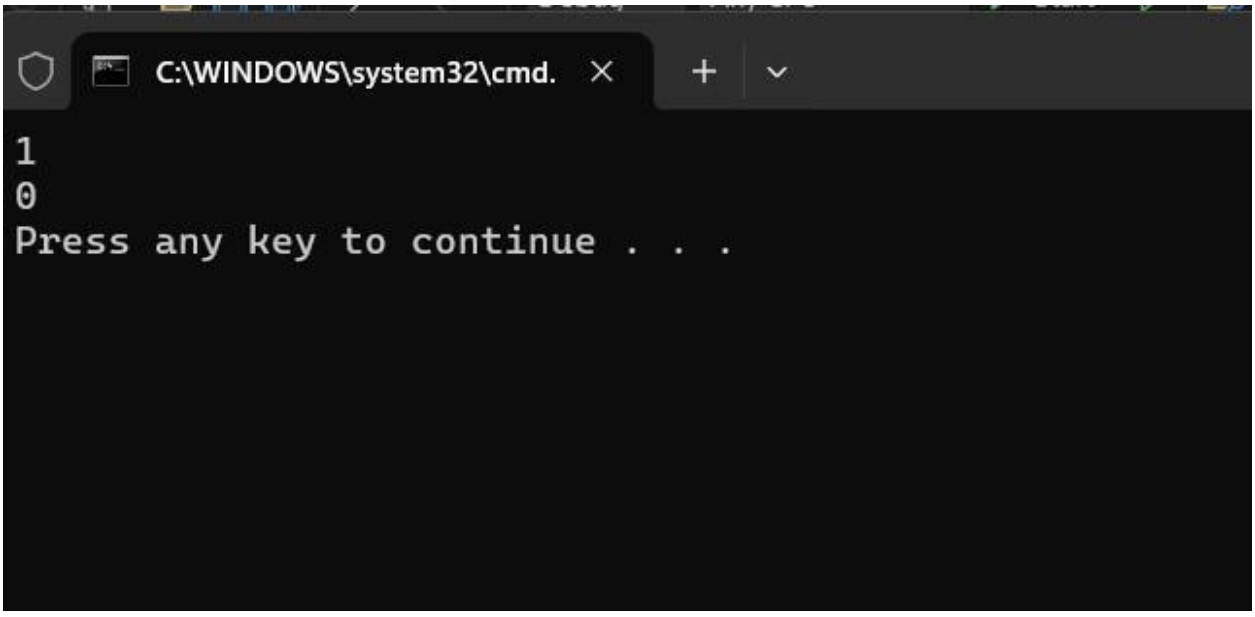


4) Maximum Number of Non-overlapping Palindrome Substrings You are given a string s and a positive integer k.Select a set of non-overlapping substrings from the string s that satisfy the following conditions: ● The length of each substring is at least k. ● Each substring is a palindrome.

PROGRAM:

```
def is_palindrome(s, left, right):
   while left < right:        if
s[left] != s[right]:
          return False
left += 1        right -= 1
return True
 def
max_non_overlapping_palindromes(s,
k):     n = len(s)     if n < k:        return 0

   dp = [0] * n          for i in range(n):
for j in range(i - k + 1, -1, -1):
if is_palindrome(s, j, i):               if j
== 0:                  dp[i] = max(dp[i], 1)
else:                  dp[i] = max(dp[i],
dp[j - 1] +
1)

   return max(dp)

s1 = "abaccdbbd" k1
= 3
print(max_non_overlapping_palindrom
es(s1, k1))     s2 = "adbcda" k2 = 2
print(max_non_overlapping_palindrom
es(s2, k2))
```

OUTPUT:



5) Minimum Cost to Buy Apples You are given a positive integer n representing n cities numbered from 1

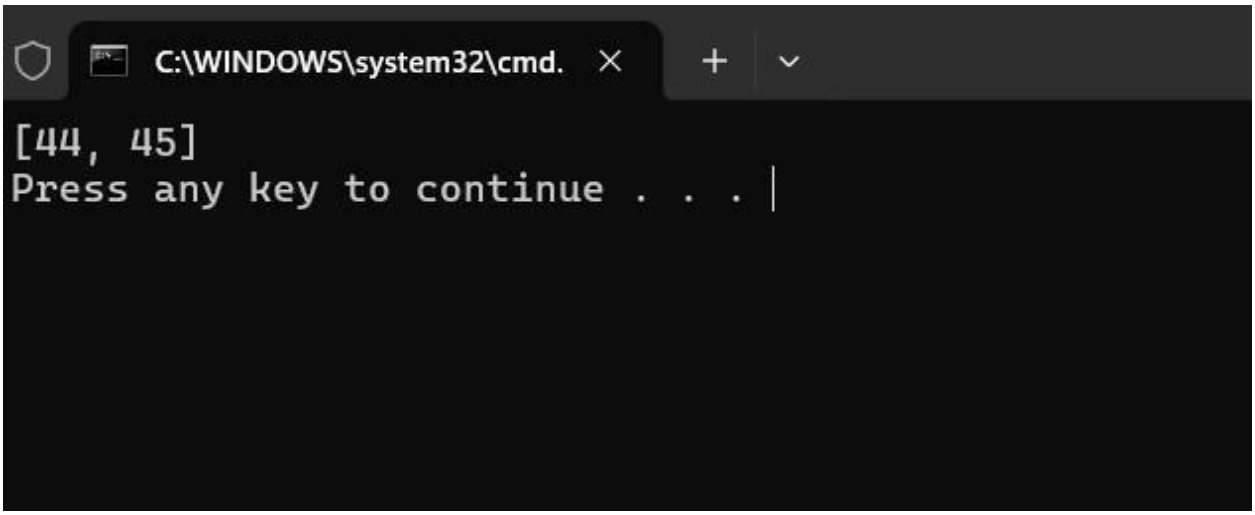
to n. You are alsogiven a 2D array roads, where roads[i] = [ai, bi, costi] indicates that there is a bidirectional road between cities ai and bi with a cost of traveling equal to costi.

PROGRAM:

```
import heapq   def dijkstra(n,
graph, start):     distances =
[float('inf')] * (n + 1)
distances[start] = 0
priority_queue = [(0, start)]
       while priority_queue:        current_distance, current_node =
heapq.heappop(priority_queue)
             if current_distance > distances[current_node]:
          continue

      for          neighbor,          weight          in
graph[current_node]:                  distance  =
current_distance + weight           if distance <
distances[neighbor]:
distances[neighbor] = distance
            heapq.heappush(priority_queue, (distance, neighbor))
```

```
   return distances   def
min_cost_to_buy_apples(n, roads,
appleCost, k):
graph = [[] for _ in range(n + 1)]
        for a, b, cost in roads:
       graph[a].append((b, cost))        graph[b].append((a,
cost))

   min_costs = []          for i in range(1, n + 1):
distances = dijkstra(n, graph, i)
for j in range(1, n + 1):           if i != j:                total_cost =
distances[j] + appleCost[j-1]
min_costs.append(total_cost)

   min_costs = sorted(min_costs)[:k]

   return min_costs   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_cost_to_buy_apples(n, roads,
appleCost, k))
```

OUTPUT:



6.Customers With Strictly Increasing

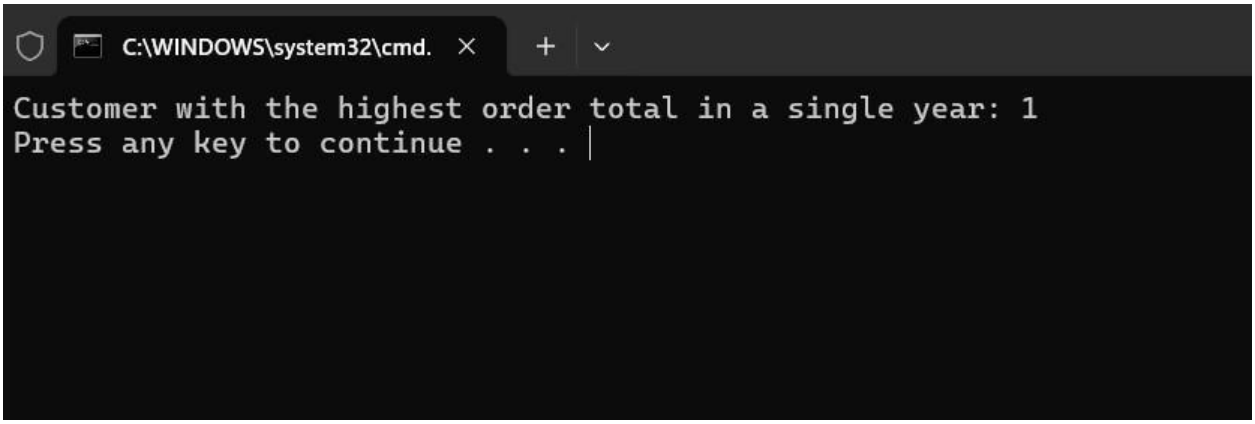
Purchases

PROGRAM:

```
from collections import
defaultdict   orders = [
   {"order_id": 1, "customer_id": 1, "order_date": "2019-07-01", "price":
1100},
   {"order_id": 2, "customer_id": 1, "order_date": "2019-11-01", "price":
1200},
   {"order_id": 3, "customer_id": 1, "order_date": "2020-05-26", "price":
3000},
   {"order_id": 4, "customer_id": 1, "order_date": "2021-08-31", "price":
3100},
   {"order_id": 5, "customer_id": 1, "order_date": "2022-12-07", "price":
4700},
   {"order_id": 6, "customer_id": 2, "order_date": "2015-01-01", "price":
700},
   {"order_id": 7, "customer_id": 2, "order_date": "2017-11-07", "price":
1000},
   {"order_id": 8, "customer_id": 3, "order_date": "2017-01-01", "price":
900},
   {"order_id": 9, "customer_id": 3, "order_date": "2018-11-07", "price":
900}
]
customer_yearly_totals = defaultdict(lambda:
defaultdict(int))   for order in orders:     customer_id
=
order["customer_id"]     year =
order["order_date"].split("-")[0]     price =
order["price"]
```

```
 customer_yearly_totals[customer_id][year] += price

customer_max_yearly_total = {}    for customer_id,
yearly_totals in customer_yearly_totals.items():
max_total = max(yearly_totals.values())
    customer_max_yearly_total[customer_id] = max_total

max_customer_id = max(customer_max_yearly_total,
key=customer_max_yearly_total.get)   print(f"Customer with the highest
order total in a single year: {max_customer_id}")
```

OUTPUT:

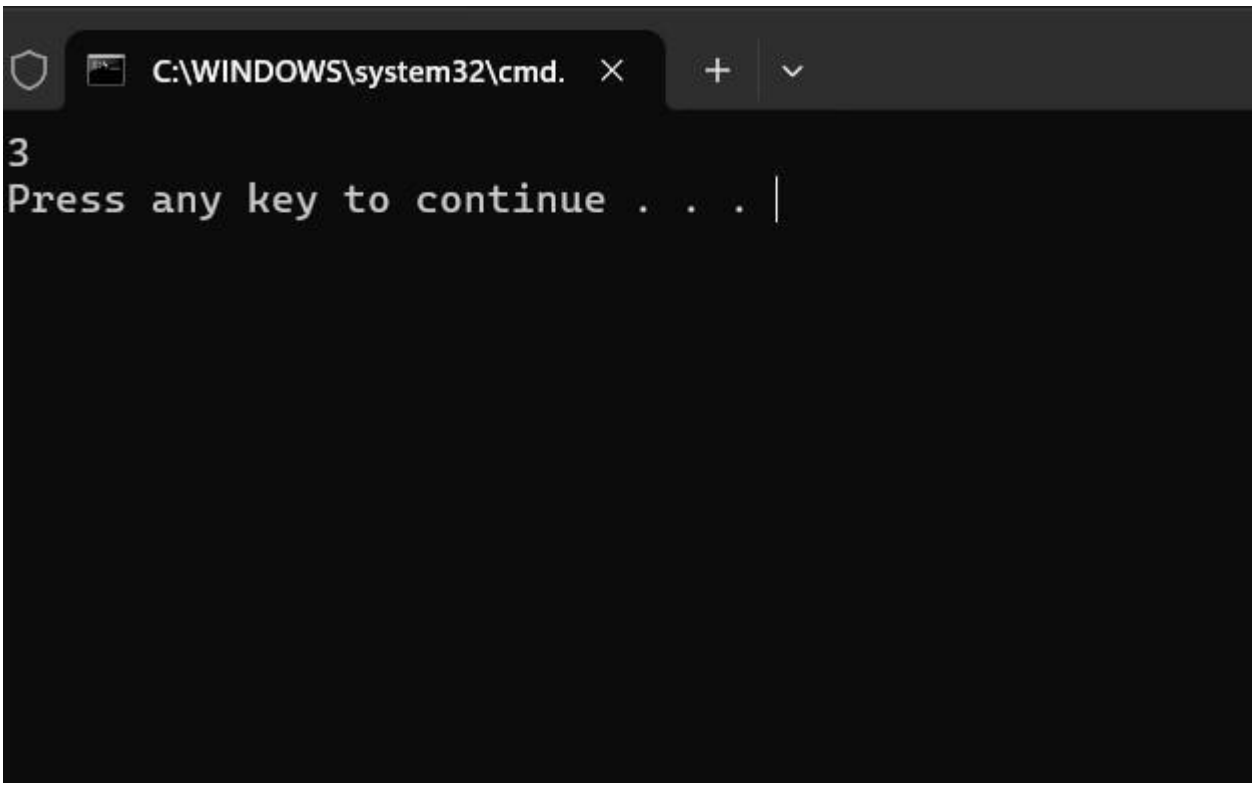


7.Number of Unequal Triplets in Array You are given a 0-indexed array of positive integers nums. Find the number of triplets (i, j, k)that meet the following conditions: ● 0 <= i < j < k < nums.length ● nums[i], nums[j], and nums[k] are pairwise distinct. ○ In other words, nums[i] != nums[j], nums[i] != nums[k], and nums[j] !=nums[k].

PROGRAM:

```
def count_valid_triplets(nums):     n = len(nums)     count = 0          for i in
range(n - 2):          for j in range(i + 1, n - 1):               for k in range(j + 1, n):
if nums[i] != nums[j] and nums[j] != nums[k] and nums[i] != nums[k]:
                    count += 1

    return count

nums = [4, 4, 2, 4, 3] print(count_valid_triplets(nums))
```

OUTPUT:



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.

PROGRAM:

```
class TreeNode:     def __init__(self, val=0, left=None,
right=None):         self.val = val         self.left = left
self.right = right
 def find_closest_values(root,
queries):     results = []     for
query in queries:          L =
find_floor(root, query)         R =
find_ceiling(root, query)
results.append([L, R])


   return results
def find_floor(root,
x):     if not root:
return -1
        if root.val == x:         return root.val
elif root.val > x:         return
find_floor(root.left, x)

   floor = find_floor(root.right, x)     return floor if
floor != -1 else root.val
 def find_ceiling(root, x):
if not root:         return -1
        if root.val == x:
return root.val     elif
root.val < x:
       return find_ceiling(root.right, x)
```

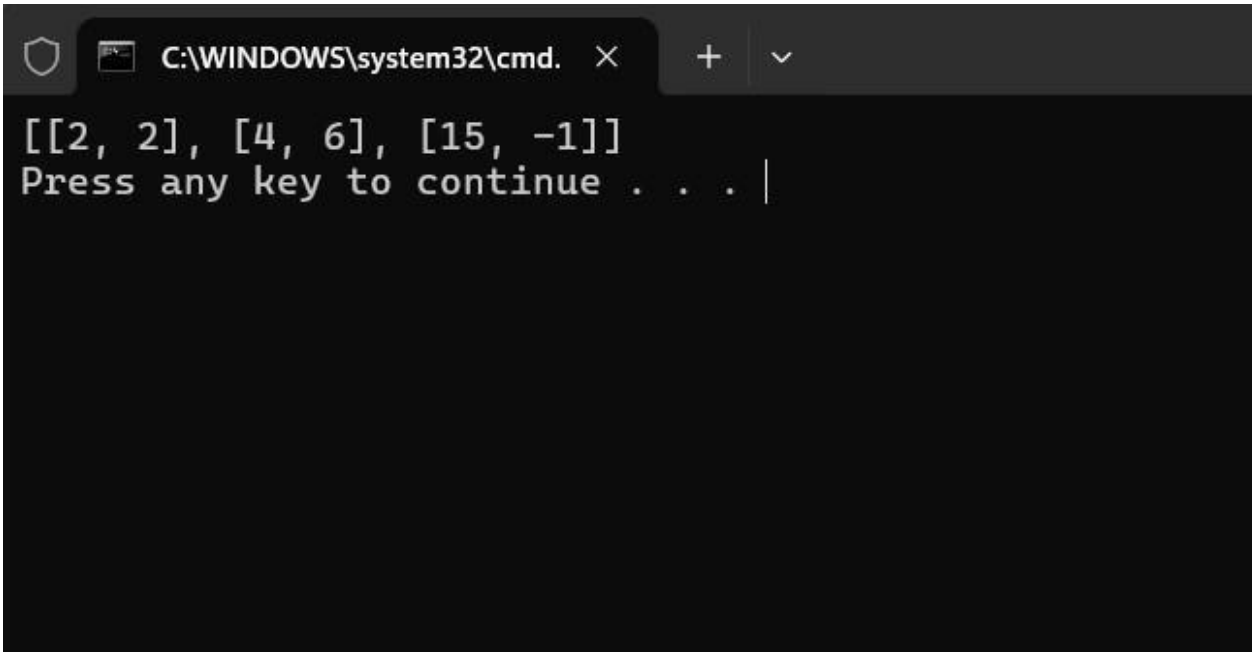
ceil = find_ceiling(root.left, x) return ceil if ceil != -1 else root.val

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(find_closest_values(root, queries))

OUTPUT:



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

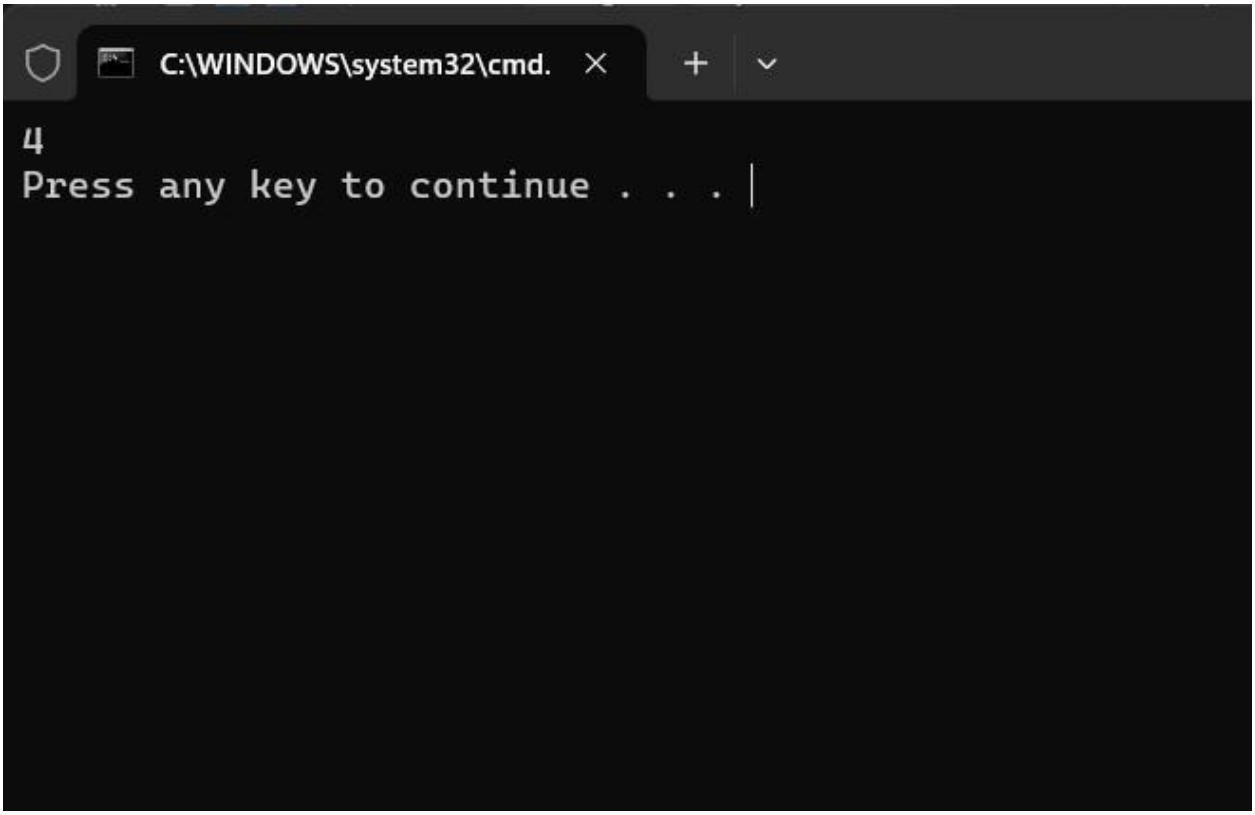
roads where roads[i] = [ai, bi] denotes that there existsabidirectional road connecting cities ai and bi. There is a meeting for the representatives of each city. The meeting is in the capital city.There is a car in each city. You are given an integer seats that indicates the number of seats in each 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.

PROGRAM:

```
from collections import defaultdict,
deque   def
minFuelCostToCapital(roads, seats):
   n = len(roads) + 1
if n == 1:        return 0

   graph = defaultdict(list)     for u, v
in roads:        graph[u].append(v)
graph[v].append(u)

   queue = deque([(0, 1)])
visited = [False] * n     visited[0] =
True     fuel_cost = 0
        while queue:
size = len(queue)
level_fuel_cost =
0                for _ in range(size):
current, seats_used = queue.popleft()

          level_fuel_cost += 1
for neighbor in graph[current]:
if not visited[neighbor]:
visited[neighbor] = True
                queue.append((neighbor, seats_used + 1))
```

```
        fuel_cost += level_fuel_cost

    return fuel_cost

roads = [[0,1],[0,2],[0,3]] seats = 5
print(minFuelCostToCapital(roads, seats))
```

OUTPUT:



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

nonintersecting 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 nonprime.

PROGRAM:

```
def is_prime_digit(char):     return char in
{'2', '3', '5', '7'}
 def count_beautiful_partitions(s, k, minLength):     n =
len(s)     memo = {}
    def count_beautiful_partitions_recursive(pos, k):
if (pos, k) in memo:
          return memo[(pos,
k)]                if k == 0 and
pos == n:            return 1
              if k == 0 or pos == n:
return 0
              count = 0        for end in range(pos + minLength, n + 1):
substring =
s[pos:end]            if is_prime_digit(substring[0]) and not
is_prime_digit(substring[-1]):
count += count_beautiful_partitions_recursive(end, k - 1)

       memo[(pos, k)] = count        return
count

   return count_beautiful_partitions_recursive(0, k)

s = "23542185131"
k = 3 minLength =
2  print(count_beautiful_partitions(s, k,
minLength))
```

OUTPUT: