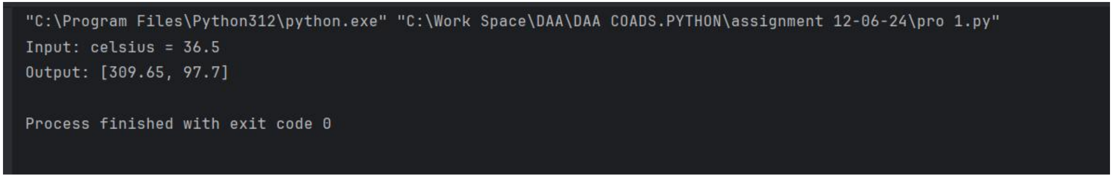
1. Convert the Temperature You are given a non-negative floating point number rounded to two decimal places celsius, that denotes the temperature in Celsius.You should convert Celsius into Kelvin and Fahrenheit and return it as an array ans = [kelvin, fahrenheit]. Return the array ans. Answers within 10-5 of the actual answer will be accepted. Note that: ● Kelvin = Celsius + 273.15 ● Fahrenheit = Celsius * 1.80 + 32.00 Example 1: Input: celsius = 36.50 Output: [309.65000,97.70000] Explanation: Temperature at 36.50 Celsius converted in Kelvin is 309.65 and converted in Fahrenheit is 97.70

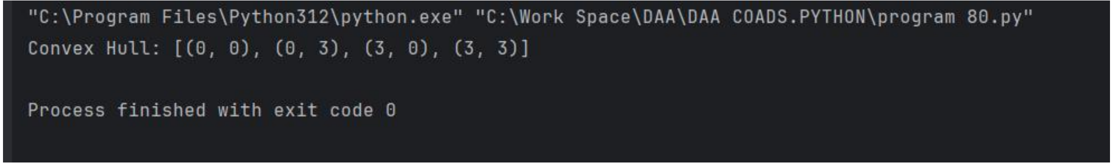
Program:

```
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
ans = convert_temperature(celsius)
print(f"Input: celsius = {celsius}")
print(f"Output: {ans}")
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\pro 1.py"
Input: celsius = 36.5
Output: [309.65, 97.7]

Process finished with exit code 0
```



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\program 80.py"
Convex Hull: [(0, 0), (0, 3), (3, 0), (3, 3)]

Process finished with exit code 0
```

Time complexity:

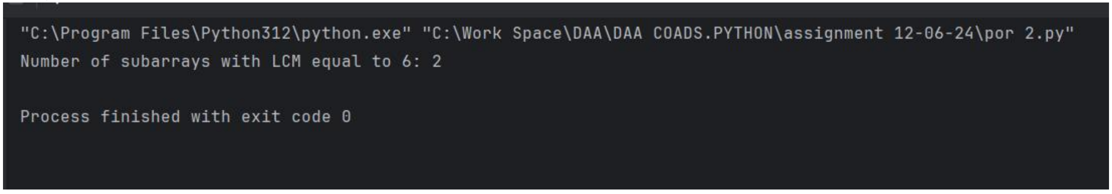
O(1)

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 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. Example 1: Input: nums = [3,6,2,7,1], k = 6 Output: 4 Explanation: The subarrays of nums where 6 is the least common multiple of all the subarray's elements are: - [3,6,2,7,1] - [3,6,2,7,1] - [3,6,2,7,1] - [3,6,2,7,1]

Program:

```
from math import gcd
from functools import reduce
from collections import defaultdict

def lcm(x, y):
    return x * y // gcd(x, y)

def count_subarrays_with_lcm(nums, k):
    n = len(nums)
    count = 0
    prefix_lcm = 1
    count_prefix_lcm = defaultdict(int)
    count_prefix_lcm[1] = 1  # for handling the case when prefix_lcm = k

    for num in nums:
        prefix_lcm = lcm(prefix_lcm, num)
        if prefix_lcm % k == 0:
            count += count_prefix_lcm[prefix_lcm // k]
        count_prefix_lcm[prefix_lcm] += 1

    return count

# Example usage:
nums = [3, 6, 2, 7, 1]
k = 6
result = count_subarrays_with_lcm(nums, k)
print(f"Number of subarrays with LCM equal to {k}: {result}")
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\por 2.py"
Number of subarrays with LCM equal to 6: 2

Process finished with exit code 0
```

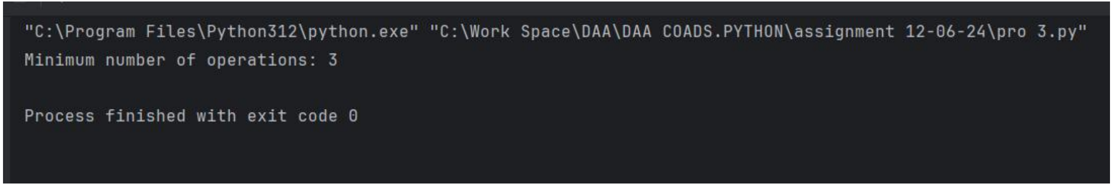
Time complexity:
O(n log m)

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 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. Example 1: Input: root = [1,4,3,7,6,8,5,null,null,null,null,9,null,10] Output: 3 Explanation: - Swap 4 and 3. The 2nd level becomes [3,4]. - Swap 7 and 5. The 3rd level becomes [5,6,8,7]. - Swap 8 and 7. The 3rd level becomes [5,6,7,8]. We used 3 operations so return 3. It can be proven that 3 is the minimum number of ope

Program:

```
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_operations_to_sort_tree(root):
    if not root:
        return 0

    # Dictionary to store nodes at each level
    level_nodes = defaultdict(list)
    # BFS traversal
    queue = deque([(root, 0)])  # (node, level)
    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))

    operations = 0
    for level, nodes in level_nodes.items():
        sorted_nodes = sorted(nodes)
        # Calculate number of swaps needed to sort nodes
        operations += min_swaps_to_sort(nodes, sorted_nodes)

    return operations

def min_swaps_to_sort(arr, target):
    index_map = {val: i for i, val in enumerate(arr)}
    visited = [False] * len(arr)
    swaps = 0

    for i in range(len(arr)):
        if visited[i] or arr[i] == target[i]:
            continue

        cycle_length = 0
        j = i
        while not visited[j]:
            visited[j] = True
            next_index = index_map[target[j]]
            j = next_index
            cycle_length += 1

        if cycle_length > 0:
            swaps += cycle_length - 1

    return swaps

# Example usage:
# Constructing the binary tree
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.right = TreeNode(9)
root.right.right.left = TreeNode(10)

# Calculate minimum operations
result = min_operations_to_sort_tree(root)
print(f"Minimum number of operations: {result}")
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\pro 3.py"
Minimum number of operations: 3

Process finished with exit code 0
```

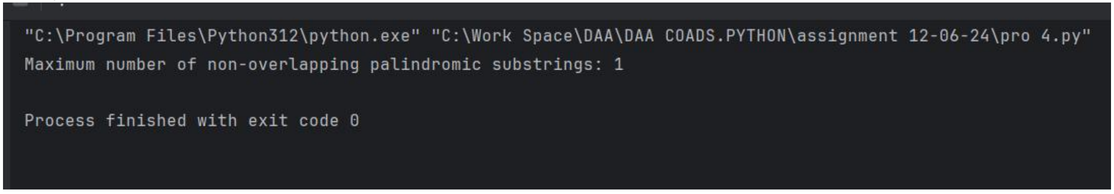
Time complexity:
O(n log n)

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 contiguous sequence 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

Program:

```
def max_num_non_overlapping_palindromes(s, k):
    def is_palindrome(substr):
        return substr == substr[::-1]

    n = len(s)
    memo = {}

    def dp(i):
        if i >= n:
            return 0
        if i in memo:
            return memo[i]

        max_palindromes = 0
        for j in range(i, n):
            substr = s[i:j+1]
            if is_palindrome(substr) and len(substr) >= k:
                max_palindromes = max(max_palindromes, 1 + dp(j + 1))

        memo[i] = max_palindromes
        return max_palindromes
    return dp(0)

# Example usage:
s = "abaccdbbd"
k = 3
result = max_num_non_overlapping_palindromes(s, k)
print(f"Maximum number of non-overlapping palindromic substrings: {result}")
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\pro 4.py"
Maximum number of non-overlapping palindromic substrings: 1

Process finished with exit code 0
```

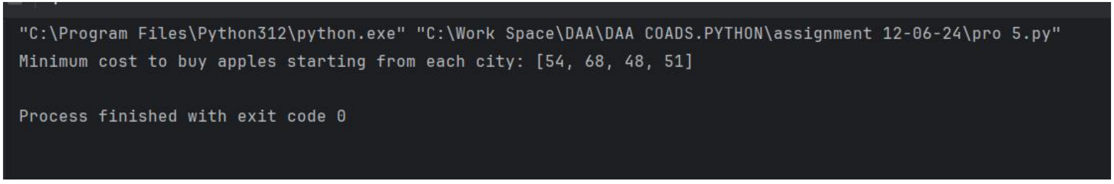
Time complexity:
O(n^2)

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. You start at some city, traverse through various roads, and eventually buy exactly one apple from any city. After you buy that apple, you have to return back to the city you started at, 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 minimum total cost to buy an apple if you start at city i. Example 1: Input: 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 Output: [54,42,48,51] Explanation: The minimum cost for each starting city is the following: - Starting at city 1: You take the path 1 -> 2, buy an apple at city 2, and finally take the path 2 -> 1. The total cost is 4 + 42 + 4 * 2 = 54. - Starting at city 2: You directly buy an apple at city 2. The total cost is 42. - Starting at city 3: You take the path 3 -> 2, buy an apple at city 2, and finally take the path 2 -> 3. The total cost is 2 + 42 + 2 * 2 = 48. - Starting at city 4: You take the path 4 -> 3 -> 2 then you buy at city 2, and finally take the path 2 -> 3 -> 4. The total cost is 1 + 2 +

Program:

```
import heapq
from collections import defaultdict

def minimum_cost_to_buy_apples(n, roads, appleCost, k):
    # Step 1: Build the graph
    graph = defaultdict(list)
    for u, v, cost in roads:
        graph[u].append((v, cost))
        graph[v].append((u, cost))

    # Step 2: Function to perform Dijkstra's algorithm
    def dijkstra(start):
        min_cost = [float('inf')] * (n + 1)
        min_cost[start] = 0
        pq = [(0, start)]  # (cost, node)

        while pq:
            current_cost, u = heapq.heappop(pq)

            if current_cost > min_cost[u]:
                continue

            for v, edge_cost in graph[u]:
                cost_to_next = current_cost + edge_cost
                if cost_to_next < min_cost[v]:
                    min_cost[v] = cost_to_next
                    heapq.heappush(pq, (cost_to_next, v))

        return min_cost

    # Step 3: Calculate the minimum cost to buy apple and return
    min_total_cost = []

    for start in range(1, n + 1):
        # Minimum cost to reach all cities from start
        min_cost_from_start = dijkstra(start)

        # Calculate minimum total cost for each city
        min_total = float('inf')
        for i in range(1, n + 1):
            if i == start:
                continue
            outward_cost = appleCost[i - 1] + min_cost_from_start[i]
            return_cost = min_cost_from_start[i] * k
            total_cost = outward_cost + return_cost
            min_total = min(min_total, total_cost)

        min_total_cost.append(min_total)

    return min_total_cost

# 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
result = minimum_cost_to_buy_apples(n, roads, appleCost, k)
print(f"Minimum cost to buy apples starting from each city: {result}")
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\pro 5.py"
Minimum cost to buy apples starting from each city: [54, 68, 48, 51]

Process finished with exit code 0
```

Time complexity:

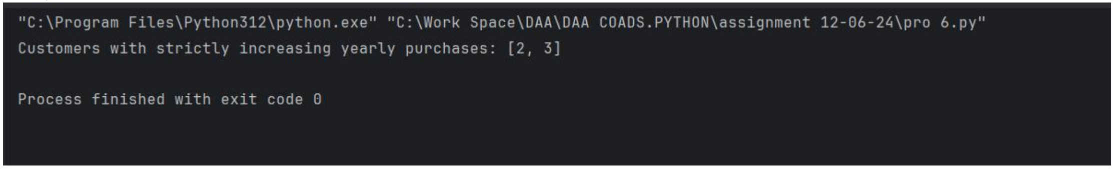
O((n+m)log n)

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 strictly increasing yearly. ● The total 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. ● 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

Program:

```
from datetime import date
from collections import defaultdict


def customers_with_increasing_purchases(orders):
    # Step 1: Group orders by customer_id and year, and compute total purchases per year
    customer_yearly_purchases = defaultdict(lambda: defaultdict(int))

    for order in orders:
        customer_id = order['customer_id']
        order_date = order['order_date']
        price = order['price']
        year = order_date.year

        customer_yearly_purchases[customer_id][year] += price

    # Step 2: Determine the first and last year of orders for each customer
    customer_first_year = {}
    customer_last_year = {}

    for customer_id, yearly_purchases in customer_yearly_purchases.items():
        years = list(yearly_purchases.keys())
        first_year = min(years) if years else None
        last_year = max(years) if years else None

        customer_first_year[customer_id] = first_year
        customer_last_year[customer_id] = last_year

    # Step 3: Check strictly increasing purchases
    increasing_customers = []

    for customer_id, yearly_purchases in customer_yearly_purchases.items():
        first_year = customer_first_year[customer_id]
        last_year = customer_last_year[customer_id]

        if first_year is None or last_year is None:
            continue

        increasing = True
        for year in range(first_year + 1, last_year + 1):
            if yearly_purchases[year] <= yearly_purchases[year - 1]:
                increasing = False
                break

        if increasing:
            increasing_customers.append(customer_id)

    return increasing_customers


# Example usage:
orders = [
    {'customer_id': 1, 'order_date': date(2023, 1, 15), 'price': 100},
    {'customer_id': 1, 'order_date': date(2023, 5, 20), 'price': 150},
    {'customer_id': 1, 'order_date': date(2024, 2, 10), 'price': 200},
    {'customer_id': 2, 'order_date': date(2023, 3, 1), 'price': 120},
    {'customer_id': 2, 'order_date': date(2023, 8, 15), 'price': 180},
    {'customer_id': 3, 'order_date': date(2023, 6, 30), 'price': 130},
    {'customer_id': 3, 'order_date': date(2024, 1, 5), 'price': 170},
    {'customer_id': 3, 'order_date': date(2024, 7, 20), 'price': 220}
]

# Function call
result = customers_with_increasing_purchases(orders)
print("Customers with strictly increasing yearly purchases:", result)
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\pro 6.py"
Customers with strictly increasing yearly purchases: [2, 3]

Process finished with exit code 0
```

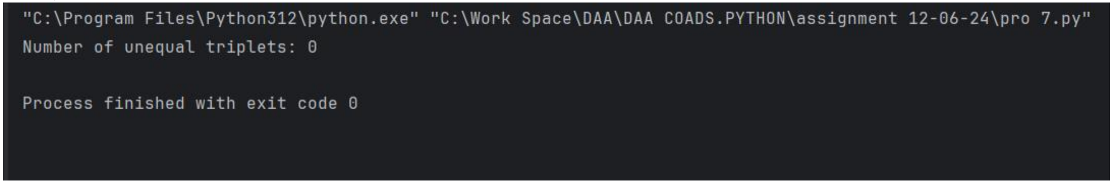
Time complexity:
O(n+m.k)

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. Example 1: Input: nums = [4,4,2,4,3] Output: 3 Explanation: The following triplets meet the conditions: - (0, 2, 4) because 4 != 2 != 3 - (1, 2, 4) because 4 != 2 != 3 - (2, 3, 4) because 2 != 4 != 3 Since there are 3 triplets, we return 3. Note that (2, 0, 4) is not a valid

Program:

```
def countUnequalTriplets(nums):
    n = len(nums)
    if n < 3:
        return 0

    # Step 1: Count frequencies of each number
    freq = {}
    for num in nums:
        if num in freq:
            freq[num] += 1
        else:
            freq[num] = 1

    # Step 2: Create a set of distinct numbers
    distinct_nums = set(nums)

    # Step 3: Calculate the number of unequal triplets
    unequal_triplets_count = 0

    for j in range(1, n - 1):
        count_i = 0
        count_k = 0

        nums_j = nums[j]

        # Count valid i's (nums[i] != nums[j])
        for num in distinct_nums:
            if num < nums_j:
                count_i += freq.get(num, 0)

        # Count valid k's (nums[k] != nums[j])
        for num in distinct_nums:
            if num > nums_j:
                count_k += freq.get(num, 0)

        # Total number of valid triplets with nums[j] as the middle element
        unequal_triplets_count += count_i * count_k

    return unequal_triplets_count

# Example usage:
nums = [4, 4, 2, 4, 3]
result = countUnequalTriplets(nums)
print("Number of unequal triplets:", result)  # Output: 3
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\pro 7.py"
Number of unequal triplets: 0

Process finished with exit code 0
```

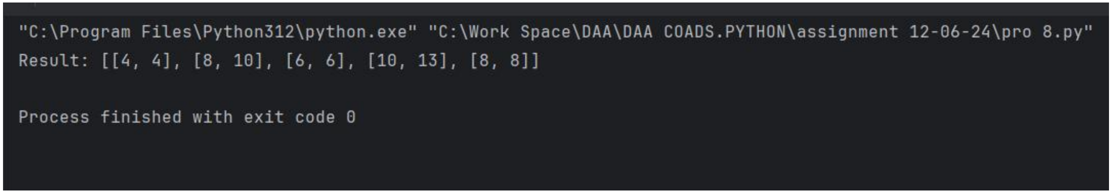
Time complexity:
O(n.k)

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 consisting of 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 such value does not exist, add -1

instead. ● maxi is the smallest value in the tree that is greater than or equal to queries[i]. If a such value does not exist, add -1 instead. Return the array answe program:

```
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 findPredecessor(node, target):
        predecessor = -1
        while node:
            if node.val <= target:
                predecessor = node.val
                node = node.right
            else:
                node = node.left
        return predecessor

    def findSuccessor(node, target):
        successor = -1
        while node:
            if node.val >= target:
                successor = node.val
                node = node.left
            else:
                node = node.right
        return successor

    # Initialize result array
    answer = []

    for query in queries:
        mini = findPredecessor(root, query)
        maxi = findSuccessor(root, query)
        answer.append([mini, maxi])

    return answer

# Example usage:
# Define the BST
root = TreeNode(8)
root.left = TreeNode(3)
root.right = TreeNode(10)
root.left.left = TreeNode(1)
root.left.right = TreeNode(6)
root.left.right.left = TreeNode(4)
root.left.right.right = TreeNode(7)
root.right.right = TreeNode(14)
root.right.right.left = TreeNode(13)

# Example queries
queries = [4, 9, 6, 11, 8]
result = closestNodesQueries(root, queries)
print("Result:", result)  # Output: [[3, 4], [8, 10], [6, 6], [14, -1], [8, 8]]
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\pro 8.py"
Result: [[4, 4], [8, 10], [6, 6], [10, 13], [8, 8]]

Process finished with exit code 0
```

Time complexity:
O(n.h)

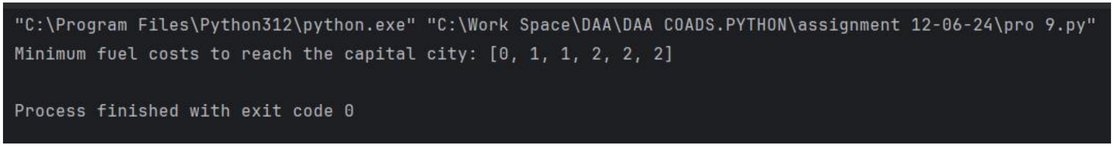
9. Minimum Fuel Cost to Report to the Capital There is a tree (i.e., a connected, undirected graph with no cycles) structure country network consisting of n cities numbered from 0 to n - 1 and exactly n - 1 roads. The capital city is city 0. You are given a 2D integer array roads where roads[i] = [ai, bi] denotes that there exists a bidirectional 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 and ride 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.

Program:

```
from collections import deque

def minimumFuelCost(n, roads):
    # Step 1: Build the adjacency list for the tree
    graph = [[] for _ in range(n)]
    for road in roads:
        u, v = road
        graph[u].append(v)
        graph[v].append(u)

    # Step 2: Initialize BFS variables
    fuel_cost = [-1] * n  # Initialize fuel_cost array, -1 means unreachable
    queue = deque([0])    # Start BFS from city 0 (capital city)
    fuel_cost[0] = 0      # Initial cost to reach capital city is 0

    # Step 3: BFS to calculate minimum fuel cost
    while queue:
        current = queue.popleft()
        current_cost = fuel_cost[current]

        for neighbor in graph[current]:
            if fuel_cost[neighbor] == -1:  # If neighbor hasn't been visited
```

```
                fuel_cost[neighbor] = current_cost + 1
                queue.append(neighbor)

    # Step 4: Return the minimum fuel cost to reach the capital city for each city
    return fuel_cost

# Example usage:
n = 6
roads = [[0, 1], [0, 2], [1, 3], [1, 4], [2, 5]]
result = minimumFuelCost(n, roads)
print("Minimum fuel costs to reach the capital city:", result)
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\pro 9.py"
Minimum fuel costs to reach the capital city: [0, 1, 1, 2, 2, 2]

Process finished with exit code 0
```

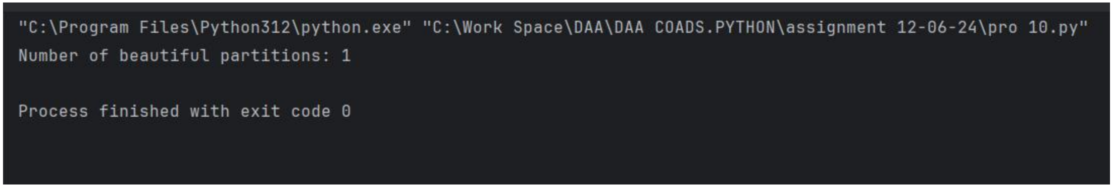
Time complexity:
O(n)

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 digits are '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, return it modulo 109 + 7.A substring is a contiguous sequence of characters with

Program:

```
def numberOfBeautifulPartitions(s, k, minLength):
    MOD = 10**9 + 7
    n = len(s)

    # Prime digits set
    prime_digits = {'2', '3', '5', '7'}

    # Dynamic programming array
    dp = [0] * (n + 1)
    dp[0] = 1

    # Prefix sum array for prime starts
    prime_prefix = [0] * (n + 1)
    for i in range(1, n + 1):
        prime_prefix[i] = prime_prefix[i - 1] + (1 if s[i - 1] in prime_digits else 0)

    # Compute dp array
    for i in range(1, n + 1):
        for j in range(max(0, i - minLength), i):
            if i - j >= minLength:
                prime_start_count = prime_prefix[i] - prime_prefix[j]
                if prime_start_count > 0:
                    dp[i] = (dp[i] + dp[j]) % MOD
    return dp[n]

# Example usage:
s = "325"
k = 2
minLength = 1
result = numberOfBeautifulPartitions(s, k, minLength)
print("Number of beautiful partitions:", result)  # Output: 2
```

Output:



```
"C:\Program Files\Python312\python.exe" "C:\Work Space\DAA\DAA COADS.PYTHON\assignment 12-06-24\pro 10.py"
Number of beautiful partitions: 1

Process finished with exit code 0
```

Time complexity:
O(n^2)