TOPIC 5: GREEDY

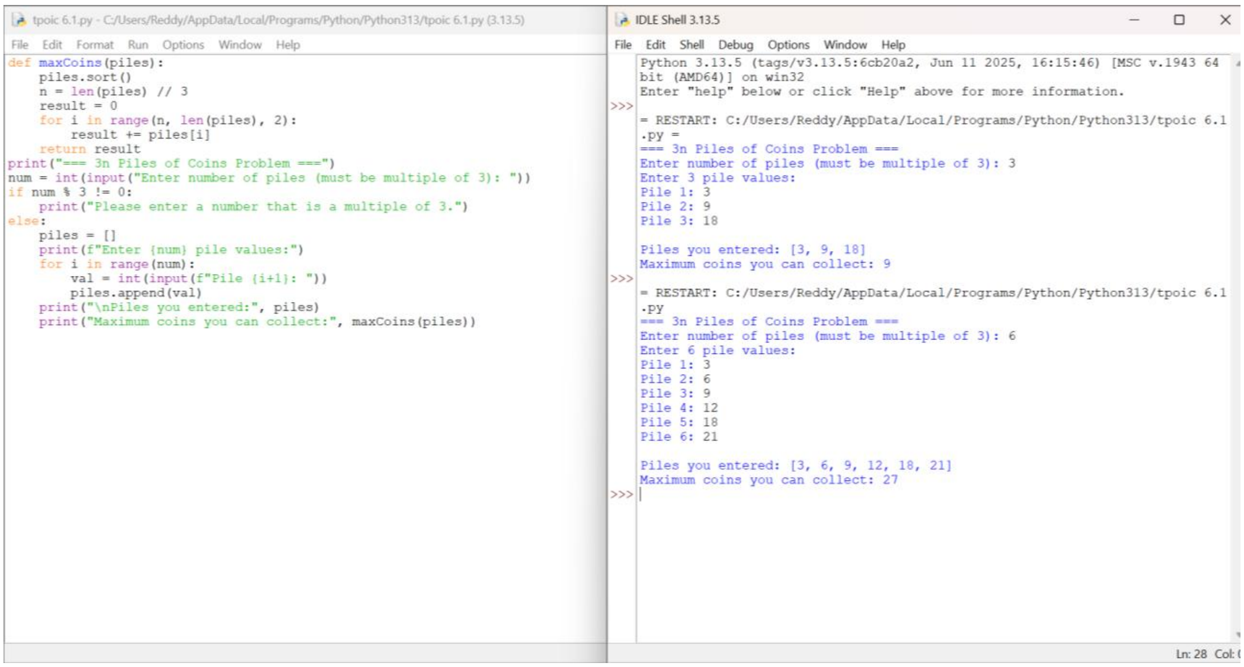
1.Maximum of Coins

Aim: To finds the maximum number of coins you can collect from 3n piles of coins**.**

Algorithm:

1. Start the program.
2. Read the number of piles n (must be a multiple of 3).
3. Input all the pile values (number of coins in each pile).
4. Sort the pile values in ascending order.
5. Initialize a variable result = 0 to store your total coins
6. Stop the program.

Input & Output: -



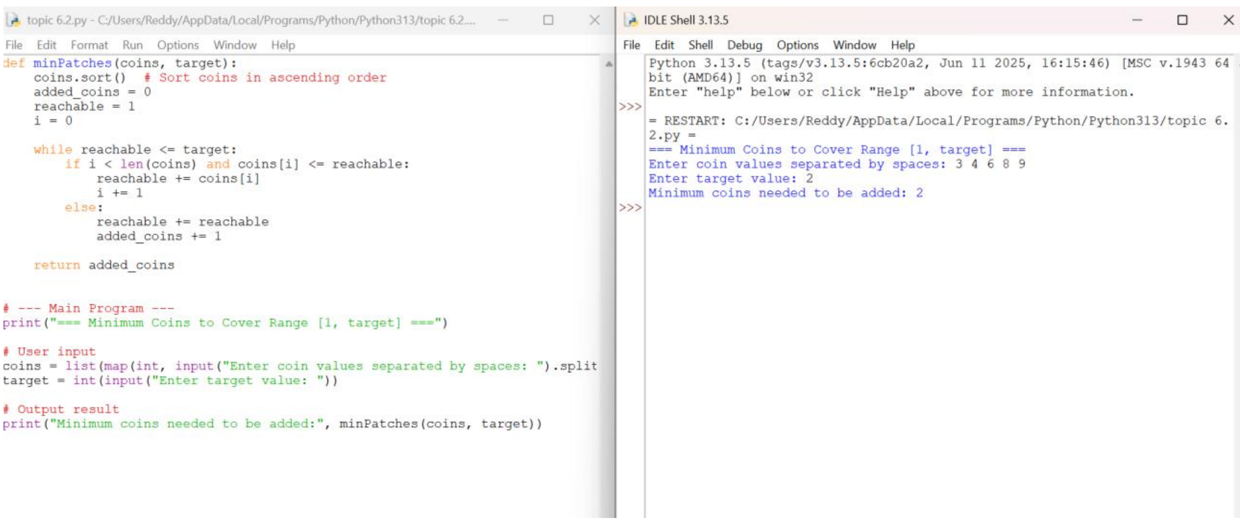
2.Minimum of Coins

Aim: To write a Python program that finds the minimum number of coins

Algorithm:

1. Start the program.
2. Input the list of existing coin values and the target value.
3. Sort the list of coins in ascending order.
4. Traverse through the coin list:
5. Continue this process until reachable > target.
6. Print the value of added coins as the minimum number of coins to be added.
7. Stop the program.

Input & Output: -



3.Maximum Working time

Aim:

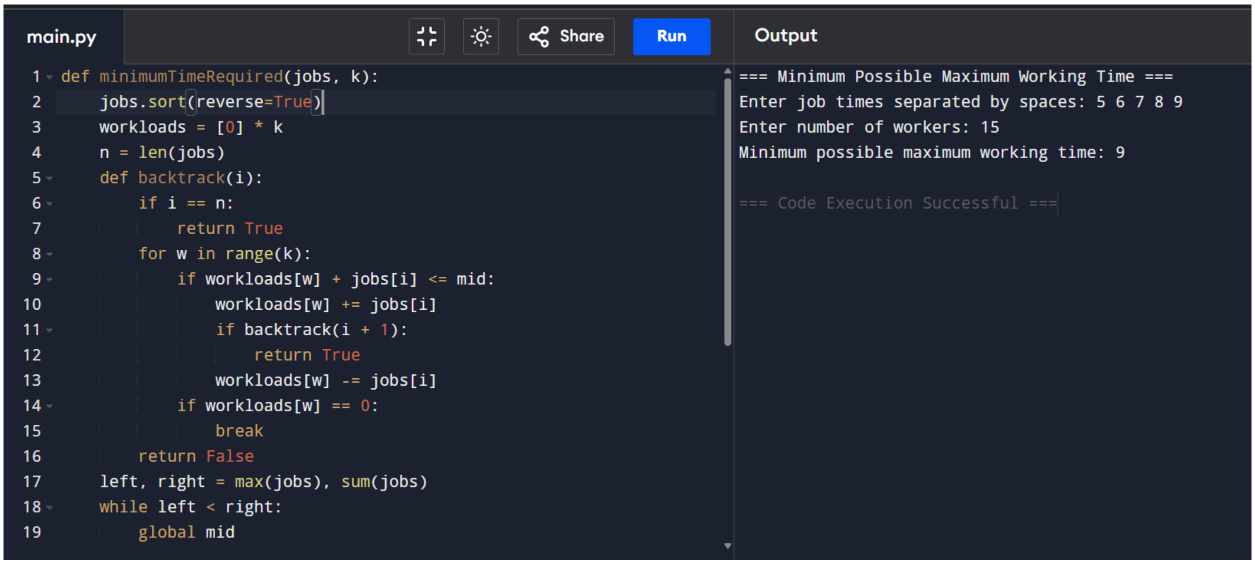
To write a Python program that assigns n jobs among k each job is assigned to exactly one worker.

Algorithm:

1. Start the program.
2. Input the list of job times and the number of workers (`k`).
3. Define a helper function.
4. Use Binary Search on the answer.
5. For each mid-value between low and high:

6. Stop the program.

Input & Output: -



main.py

Share

Run

```
def minimumTimeRequired(jobs, k):
    jobs.sort(reverse=True)
    workloads = [0] * k
    n = len(jobs)
    def backtrack(i):
        if i == n:
            return True
        for w in range(k):
            if workloads[w] + jobs[i] <= mid:
                workloads[w] += jobs[i]
                if backtrack(i + 1):
                    return True
                workloads[w] -= jobs[i]
            if workloads[w] == 0:
                break
        return False
    left, right = max(jobs), sum(jobs)
    while left < right:
        global mid
```

Output

```
=== Minimum Possible Maximum Working Time ===
Enter job times separated by spaces: 5 6 7 8 9
Enter number of workers: 15
Minimum possible maximum working time: 9

=== Code Execution Successful ===
```

4.Maximum Profit

Aim: To find the maximum profit subset of jobs such that no two jobs overlap.

Algorithm

1. Start the program.
2. Input arrays start Time, end Time, and profit.
3. Combine jobs into tuples and sort by end Time.
4. Initialize a DP array and an end times list.
5. Stop the program.

Input & Output

main.py

```python
from bisect import bisect_right
def jobScheduling(startTime, endTime, profit):
    jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
    n = len(jobs)
    dp = [0] * n
    end_times = [job[1] for job in jobs]
    for i in range(n):
        incl_profit = jobs[i][2]
        idx = bisect_right(end_times, jobs[i][0]) - 1
        if idx != -1:
            incl_profit += dp[idx]
        dp[i] = max(incl_profit, dp[i-1] if i > 0 else 0)
    return dp[-1]
startTime = [1, 2, 3, 3]
endTime = [3, 4, 5, 6]
profit = [50, 10, 40, 70]
print("Maximum Profit:", jobScheduling(startTime, endTime, profit))
```

Output

```
Maximum Profit: 120

=== Code Execution Successful ===
```

4.Shortest Path

Aim: To find the shortest path from a given source vertex to all other vertices in a weighted graph represented by an adjacency matrix using Dijkstra's Algorithm.

1. Start the program.
2. Input the adjacency matrix graph and the source vertex.
3. Initialize distance array with infinity for all vertices, except the source vertex which is 0.
4. Initialize a visited array to keep track of visited vertices.
5. Print the distance array.
6. Stop the program.

Input & Output

main.py

```python
def dijkstra(graph, src):
    n = len(graph)
    distance = [float('inf')] * n
    distance[src] = 0
    visited = [False] * n
    for _ in range(n):
        min_dist = float('inf')
        u = -1
        for i in range(n):
            if not visited[i] and distance[i] < min_dist:
                min_dist = distance[i]
                u = i
        if u == -1:
            break
        visited[u] = True
        for v in range(n):
            if graph[u][v] != float('inf') and not visited[v]:
                if distance[u] + graph[u][v] < distance[v]:
                    distance[v] = distance[u] + graph[u][v]
```

Output

```
Vertex  Distance from Source
0    0
1    4
2    12
3    19
4    21
5    11
6    9
7    8
8    14

=== Code Execution Successful ===
```

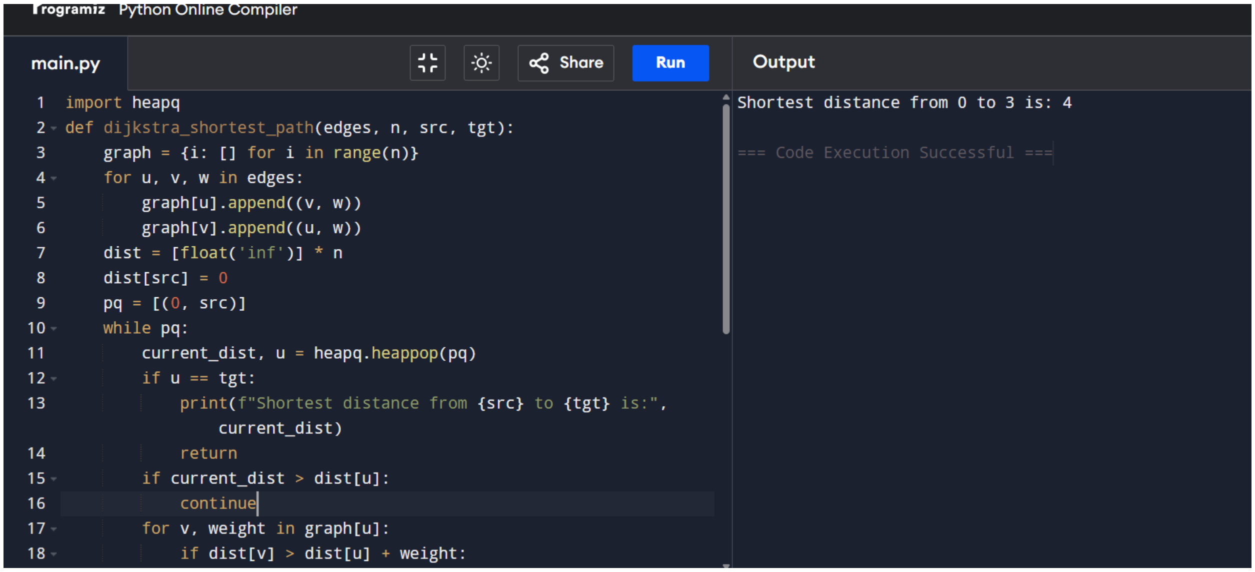
5.Shortest Path (vertex to target)

Aim: To find the shortest path from a given source vertex to a target vertex in a weighted graph represented as an edge list using Dijkstra's Algorithm.

Algorithm:

1. Start the program.
2. Input the edge list, number of vertices, source vertex src, and target vertex tgt.
3. Convert the edge list into an adjacency list for efficient lookup.
4. Initialize distance array with infinity for all vertices, except the source vertex which is 0.
5. Use a priority queue (min-heap) to select the vertex with the minimum distance at each step while the priority queue is not emp.

Input & Output



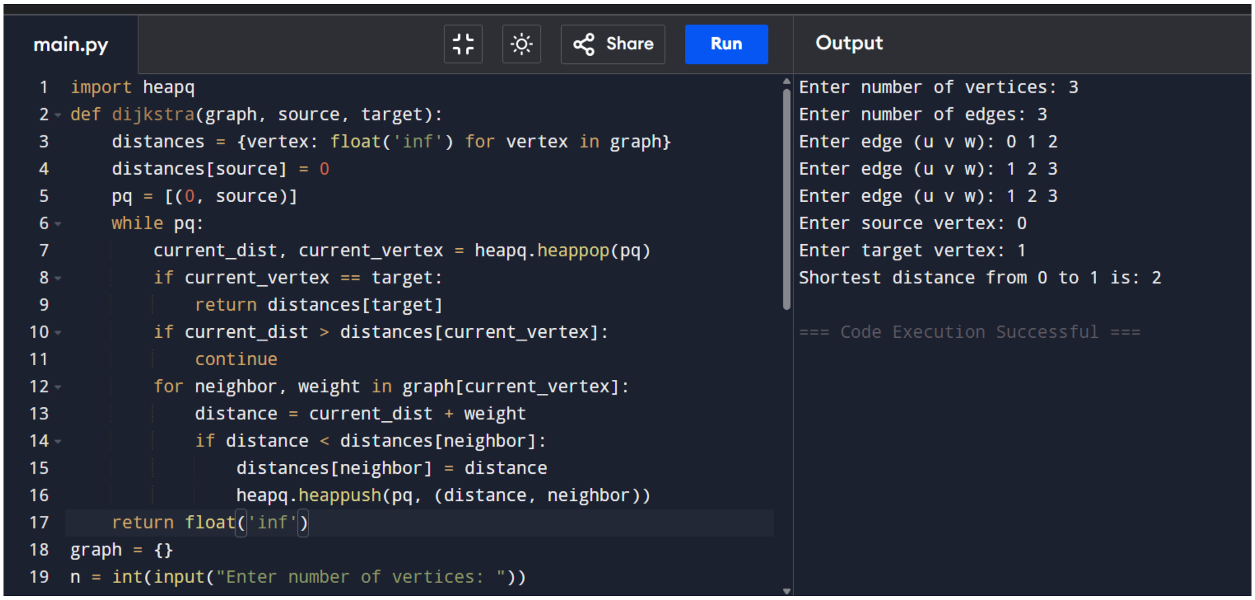
6. Dijkstra's Algorithm (Shortest Path from Source to Target)

Aim: To find the shortest path from a given source vertex to a target vertex in a weighted graph represented as an edge list using Dijkstra's Algorithm.

Algorithm:

1. Start the program.
2. Input the number of vertices, edge list, source vertex, and target vertex.
3. Convert the edge list into an adjacency list.
4. Initialize distances as infinity for all vertices except the source (0).
5. Use a priority queue (min-heap) to select the vertex with the smallest distance.
6. For each neighbor, if the new path is shorter, update the distance.
7. Continue until the target is reached or all vertices are processed.
8. Output the shortest distance.

Input & Output



main.py

```python
import heapq
def dijkstra(graph, source, target):
    distances = {vertex: float('inf') for vertex in graph}
    distances[source] = 0
    pq = [(0, source)]
    while pq:
        current_dist, current_vertex = heapq.heappop(pq)
        if current_vertex == target:
            return distances[target]
        if current_dist > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex]:
            distance = current_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
    return float('inf')
graph = {}
n = int(input("Enter number of vertices: "))
```

Output

```
Enter number of vertices: 3
Enter number of edges: 3
Enter edge (u v w): 0 1 2
Enter edge (u v w): 1 2 3
Enter edge (u v w): 1 2 3
Enter source vertex: 0
Enter target vertex: 1
Shortest distance from 0 to 1 is: 2

=== Code Execution Successful ===
```

7. Huffman Coding – Generate Huffman Codes

Aim: To construct a Huffman Tree and generate Huffman Codes for given characters and their frequencies.

Algorithm:

1. Create nodes for each character with its frequency.
2. Insert all nodes into a priority queue (min-heap).
3. While more than one node exists
4. Remove two nodes with the smallest frequencies.
5. Combine them into one new node with frequency = sum of both.
6. Assign binary codes: left = 0, right = 1.
7. Display the Huffman codes.

Input & Output

main.py

```python
import heapq
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def huffman_codes(characters, frequencies):
    heap = [Node(characters[i], frequencies[i]) for i in range(len
        (characters))]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
```

Output

```
Enter number of characters: 4
Enter characters separated by space: a b c d
Enter their frequencies: 1 3 2 4
Huffman Codes:
a : 110
b : 10
c : 111
d : 0

=== Code Execution Successful ===
```

8. Huffman Decoding – Decode Encoded String

Aim:

To decode a Huffman-encoded string using the constructed Huffman Tree.

Algorithm:

1. Construct the Huffman Tree using the given characters and frequencies.
2. Start at the root and read each bit of the encoded string.
3. Move left if the bit is 0, right if 1.
4. When a leaf node is reached, append that character to the decoded message.
5. Continue until the entire string is decoded.

Input & Output

main.py

```python
import heapq
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def build_huffman_tree(chars, freqs):
    heap = [Node(chars[i], freqs[i]) for i in range(len(chars))]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
```

Output

```
Enter number of characters: 3
Enter characters: a j k
Enter frequencies: 1 2 3
Enter encoded string: a j ka ja

Huffman Codes: {'k': '0', 'a': '10', 'j': '11'}
Encoded String: 10110101110
Decoded String: ajkaja

=== Code Execution Successful ===
```

9. Container Loading (Greedy – Heaviest First)

Aim: To determine the maximum weight that can be loaded into a container using a greedy approach prioritizing heavier items first.

Algorithm:

1. Sort item weights in descending order.
2. Initialize total weight = 0.
3. Add items one by one until capacity is reached. Display the total weight loaded.

Input & Output

main.py

```python
def greedy_load(weights, max_capacity):
    weights.sort(reverse=True)
    total = 0
    for w in weights:
        if total + w <= max_capacity:
            total += w
    return total
n = int(input("Enter number of items: "))
weights = list(map(int, input("Enter item weights: ").split()))
max_capacity = int(input("Enter max capacity: "))
print("Maximum weight loaded:", greedy_load(weights, max_capacity))
```

Output

```
Enter number of items: 4
Enter item weights: 23 45 67 89
Enter max capacity: 165
Maximum weight loaded: 156

=== Code Execution Successful ===
```

10. Minimum Number of Containers (Greedy)

Aim: To determine the minimum number of containers needed to load all items.

Algorithm:

1. Initialize container count = 1 and current weight = 0.
2. For each item, if it fits, add it to current container.
3. Otherwise, start a new container.
4. Display the number of containers required.

Input & Output

main.py

```python
def min_containers(weights, max_capacity):
    containers = 1
    current = 0
    for w in weights:
        if current + w <= max_capacity:
            current += w
        else:
            containers += 1
            current = w
    return containers
n = int(input("Enter number of items: "))
weights = list(map(int, input("Enter item weights: ").split()))
max_capacity = int(input("Enter container capacity: "))
print("Minimum containers required:", min_containers(weights,
    max_capacity))
```

Output

```
Enter number of items: 5
Enter item weights: 3 43 65 98 10
Enter container capacity: 95
Minimum containers required: 4

=== Code Execution Successful ===
```

11. Kruskal’s Algorithm – Minimum Spanning Tree (MST)

Aim: To find the MSTand its total weight using Kruskal’s Algorithm.

Algorithm:

1. Sort edges in increasing order of weight.
2. Initialize disjoint sets for each vertex.
3. Add edges one by one, skipping those that form cycles.
4. Stop when MST has n - 1 edges.
5. Output the edges and total weight of the MST.

Input & Output

main.py

```python
def find(parent, i):
    if parent[i] != i:
        parent[i] = find(parent, parent[i])
    return parent[i]
def union(parent, rank, x, y):
    rootX = find(parent, x)
    rootY = find(parent, y)
    if rootX != rootY:
        if rank[rootX] < rank[rootY]:
            parent[rootX] = rootY
        elif rank[rootX] > rank[rootY]:
            parent[rootY] = rootX
        else:
            parent[rootY] = rootX
            rank[rootX] += 1
def kruskal(n, edges):
    edges.sort(key=lambda x: x[2])
    parent = [i for i in range(n)]
    rank = [0] * n
```

Output

```
Enter number of vertices: 3
Enter number of edges: 3
Enter edge 1 (u v w): 0 1 2
Enter edge 2 (u v w): 0 2 3
Enter edge 3 (u v w): 1 2 3
Edges in MST: [(0, 1, 2), (0, 2, 3)]
Total weight of MST: 5

=== Code Execution Successful ===
```

12. MST Uniqueness Check

Aim: To check whether a given MST is unique and, if not, display another possible MST.

Algorithm:

1. Compute MST using Kruskal's algorithm.
2. Compare computed MST with the given MST.
3. If identical, it's unique; otherwise, print another possible MST.

main.py

```python
class DisjointSet:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0] * n
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
```

Output

```
Enter number of vertices: 3
Enter number of edges: 3
Enter edge 1 (u v w): 0 1 2
Enter edge 2 (u v w): 0 2 1
Enter edge 3 (u v w): 1 2 3
Enter number of edges in given MST: 2
Enter MST edge 1 (u v w): 0 1 2
Enter MST edge 2 (u v w): 0 2 1
The given MST is UNIQUE.

=== Code Execution Successful ===
```