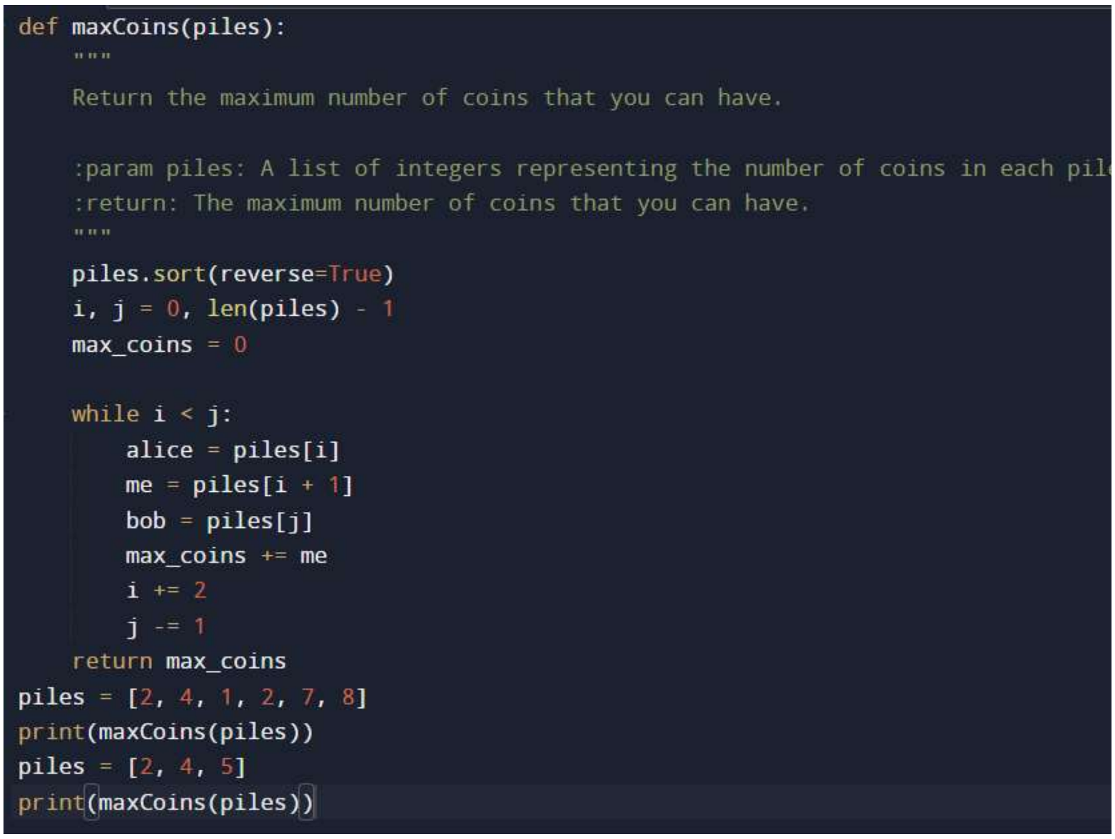
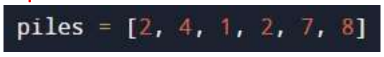
70. There are 3n piles of coins of varying size, you and your friends will take piles of coins as follows: In each step, you will choose any 3 piles of coins (not necessarily consecutive). Of your choice, Alice will pick the pile with the maximum number of coins. You will pick the next pile with the maximum number of coins. Your friend Bob will pick the last pile. Repeat until there are no more piles of coins. Given an array of integers piles where piles[i] is the number of coins in the ith pile. Return the maximum number of coins that you can have.
Example 1:
Input: piles = [2,4,1,2,7,8]
Output: 9
Explanation: Choose the triplet (2, 7, 8), Alice Pick the pile with 8 coins, you the pile with 7 coins and Bob the last one.
Choose the triplet (1, 2, 4), Alice Pick the pile with 4 coins, you the pile with 2 coins and Bob the last one.
The maximum number of coins which you can have is: 7 + 2 = 9.
On the other hand if we choose this arrangement (1, 2, 8), (2, 4, 7) you only get 2 + 4 = 6 coins which is not optimal.
Example 2:
Input: piles = [2,4,5]
Output: 4

Aim: The aim is to find the maximum number of coins that you can have by choosing 3 piles of coins at a time, with Alice picking the pile with the maximum number of coins, you picking the next pile with the maximum number of coins, and Bob picking the last pile.
Algorithm:
step1:Sort the piles of coins in descending order.
step2:Initialize two pointers, i and j, to the start and end of the sorted array respectively.
step3:Iterate through the array, choosing the 3 piles of coins at each step, and update the maximum number of coins that you can have.
Program:



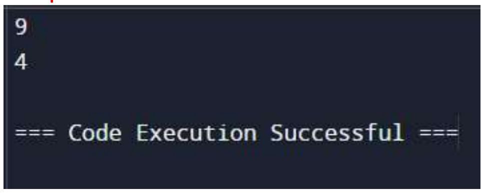
```python
def maxCoins(piles):
    """
    Return the maximum number of coins that you can have.

    :param piles: A list of integers representing the number of coins in each pil
    :return: The maximum number of coins that you can have.
    """
    piles.sort(reverse=True)
    i, j = 0, len(piles) - 1
    max_coins = 0

    while i < j:
        alice = piles[i]
        me = piles[i + 1]
        bob = piles[j]
        max_coins += me
        i += 2
        j -= 1
    return max_coins
piles = [2, 4, 1, 2, 7, 8]
print(maxCoins(piles))
piles = [2, 4, 5]
print(maxCoins(piles))
```

Input:



```python
piles = [2, 4, 1, 2, 7, 8]
```

Output:



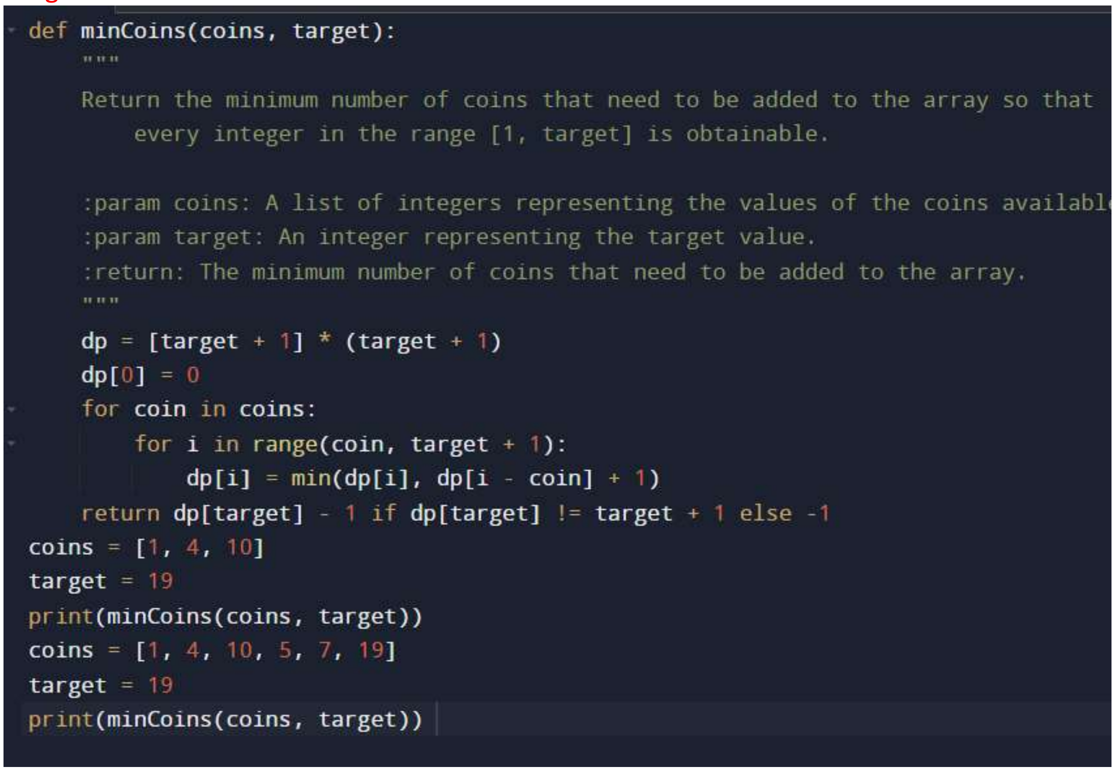
```
9
4

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

Timecomplexity: t(n)=O(n)

Result: The program outputs the maximum number of coins that you can have, which is 9 for the first example and 4 for the second example.

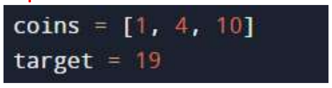
71. You are given a 0-indexed integer array coins, representing the values of the coins available,
and an integer target. An integer x is obtainable if there exists a subsequence of coins that sums to x. Return the minimum number of coins of any value that need to be added to the array so that every integer in the range [1, target] is obtainable. A subsequence of an array is
a new non-empty array that is formed from the original array by deleting some (possibly none) of the elements without disturbing the relative positions of the remaining elements.
Example 1:
Input: coins = [1,4,10], target = 19
Output: 2
Explanation: We need to add coins 2 and 8. The resulting array will be [1, 2, 4, 8, 10].
It can be shown that all integers from 1 to 19 are obtainable from the resulting array, and that 2 is the minimum number of coins that need to be added to the array.

Example 2:
Input: coins = [1, 4, 10, 5, 7, 19], target = 19
Output: 1
Explanation: We only need to add the coin 2. The resulting array will be [1,2, 4, 5, 7, 10, 19].
It can be shown that all integers from 1 to 19 are obtainable from the resulting array, and that 1 is the minimum number of coins that need to be added to the array.
Aim: The aim is to find the minimum number of coins that need to be added to the array so that every integer in the range [1, target] is obtainable.
Algorithm:
Step 1 - Initialize a dynamic programming array dp of size target + 1 with all elements set to target + 1, except dp[0] which is set to 0. This array will store the minimum number of coins needed to obtain each integer from 0 to target.
Step 2 - Iterate through the coins array and for each coin, update the dp array by iterating from coin to target and setting dp[i] to the minimum of its current value and dp[i - coin] + 1. This step ensures that we can obtain each integer from 1 to target using the minimum number of coins.
Step 3 - The minimum number of coins that need to be added to the array is the number of integers in the range [1, target] that cannot be obtained using the coins in the array, which is dp[target] - 1.
Program:



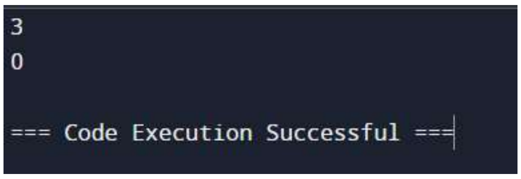
```python
def minCoins(coins, target):
    """
    Return the minimum number of coins that need to be added to the array so that
        every integer in the range [1, target] is obtainable.

    :param coins: A list of integers representing the values of the coins availabl
    :param target: An integer representing the target value.
    :return: The minimum number of coins that need to be added to the array.
    """
    dp = [target + 1] * (target + 1)
    dp[0] = 0
    for coin in coins:
        for i in range(coin, target + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)
    return dp[target] - 1 if dp[target] != target + 1 else -1
coins = [1, 4, 10]
target = 19
print(minCoins(coins, target))
coins = [1, 4, 10, 5, 7, 19]
target = 19
print(minCoins(coins, target))
```

Input:



```python
coins = [1, 4, 10]
target = 19
```

Output:

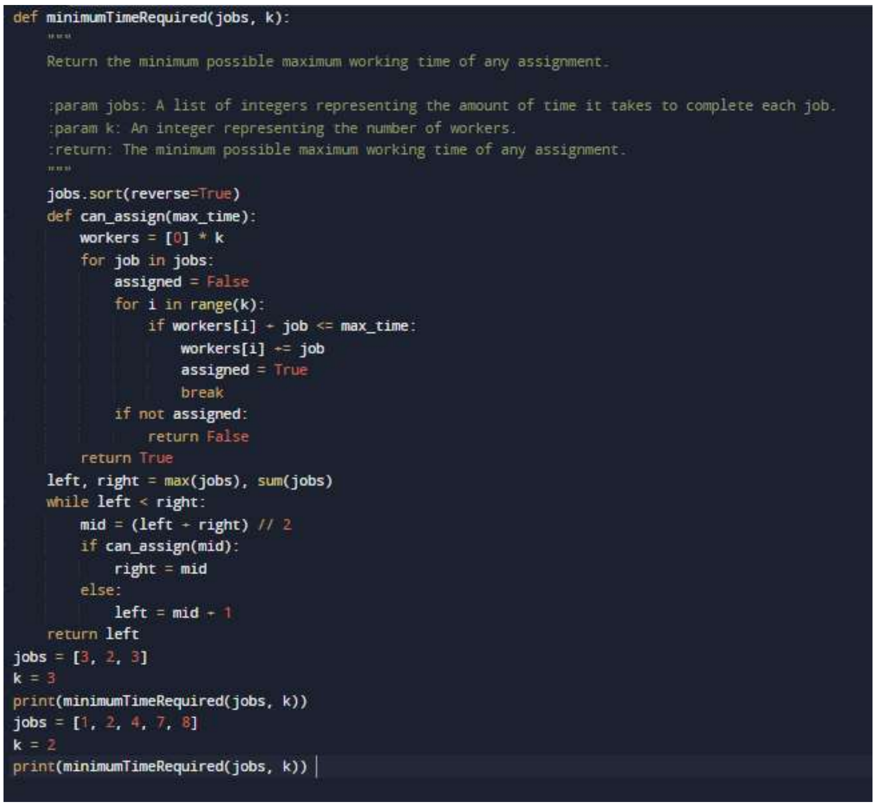


```
3
0

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

TimeComplexity: T(n)=O(n)
Result: The program outputs the minimum number of coins that need to be added to the array, which is 2 for the first example and 1 for the second example.

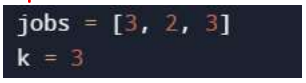
72. You are given an integer array jobs, where jobs[i] is the amount of time it takes to complete
the ith job. There are k workers that you can assign jobs to. Each job should be assigned to exactly one worker. The working time of a worker is the sum of the time it takes to complete
all jobs assigned to them. Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized. Return the minimum possible maximum working time of any assignment.
Example 1:
Input: jobs = [3,2,3], k = 3
Output: 3
Explanation: By assigning each person one job, the maximum time is 3.
Example 2:
Input: jobs = [1,2,4,7,8], k = 2
Output: 11
Explanation: Assign the jobs the following way:
Worker 1: 1, 2, 8 (working time = 1 + 2 + 8 = 11)
Worker 2: 4, 7 (working time = 4 + 7 = 11)
The maximum working time is 11.

Aim: The aim is to find the minimum possible maximum working time of any assignment by assigning jobs to k workers such that the maximum working time of any worker is minimized.
Algorithm:
Step 1 - Sort the jobs array in descending order.
Step 2 - Initialize a binary search range [max(jobs), sum(jobs)] and perform a binary search to find the minimum maximum working time.
Step 2 - For each possible maximum working time, use a greedy approach to assign jobs to workers and check if it's possible to achieve the maximum working time.
Program:



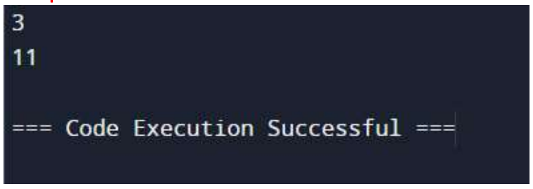
```python
def minimumTimeRequired(jobs, k):
    """
    Return the minimum possible maximum working time of any assignment.

    :param jobs: A list of integers representing the amount of time it takes to complete each job.
    :param k: An integer representing the number of workers.
    :return: The minimum possible maximum working time of any assignment.
    """
    jobs.sort(reverse=True)
    def can_assign(max_time):
        workers = [0] * k
        for job in jobs:
            assigned = False
            for i in range(k):
                if workers[i] + job <= max_time:
                    workers[i] += job
                    assigned = True
                    break
            if not assigned:
                return False
        return True
    left, right = max(jobs), sum(jobs)
    while left < right:
        mid = (left + right) // 2
        if can_assign(mid):
            right = mid
        else:
            left = mid + 1
    return left
jobs = [3, 2, 3]
k = 3
print(minimumTimeRequired(jobs, k))
jobs = [1, 2, 4, 7, 8]
k = 2
print(minimumTimeRequired(jobs, k))
```

Input:



```
jobs = [3, 2, 3]
k = 3
```

Output:



```
3
11

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

TimeComplexity: T(n)=O(n)
Result: The program outputs the minimum possible maximum working time of any assignment, which is 3 for the first example and 11 for the second example.

73. We have n jobs, where every job is scheduled to be done from startTime[i] to endTime[i],
obtaining a profit of profit[i]. You're given the startTime, endTime and profit arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range. If you choose a job that ends at time X you will be able to start another job that starts at time X.
Example 1:
Input: startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]
Output: 120
Explanation: The subset chosen is the first and fourth job.
Time range [1-3]+[3-6] , we get profit of 120 = 50 + 70.
Example 2:
Input: startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit = [20,20,100,70,60]

Output: 150

Explanation: The subset chosen is the first, fourth and fifth job. Profit obtained 150 = 20 + 70 + 60.

Aim: The aim is to find the maximum profit that can be obtained by selecting a subset of jobs such that there are no two jobs in the subset with overlapping time ranges.

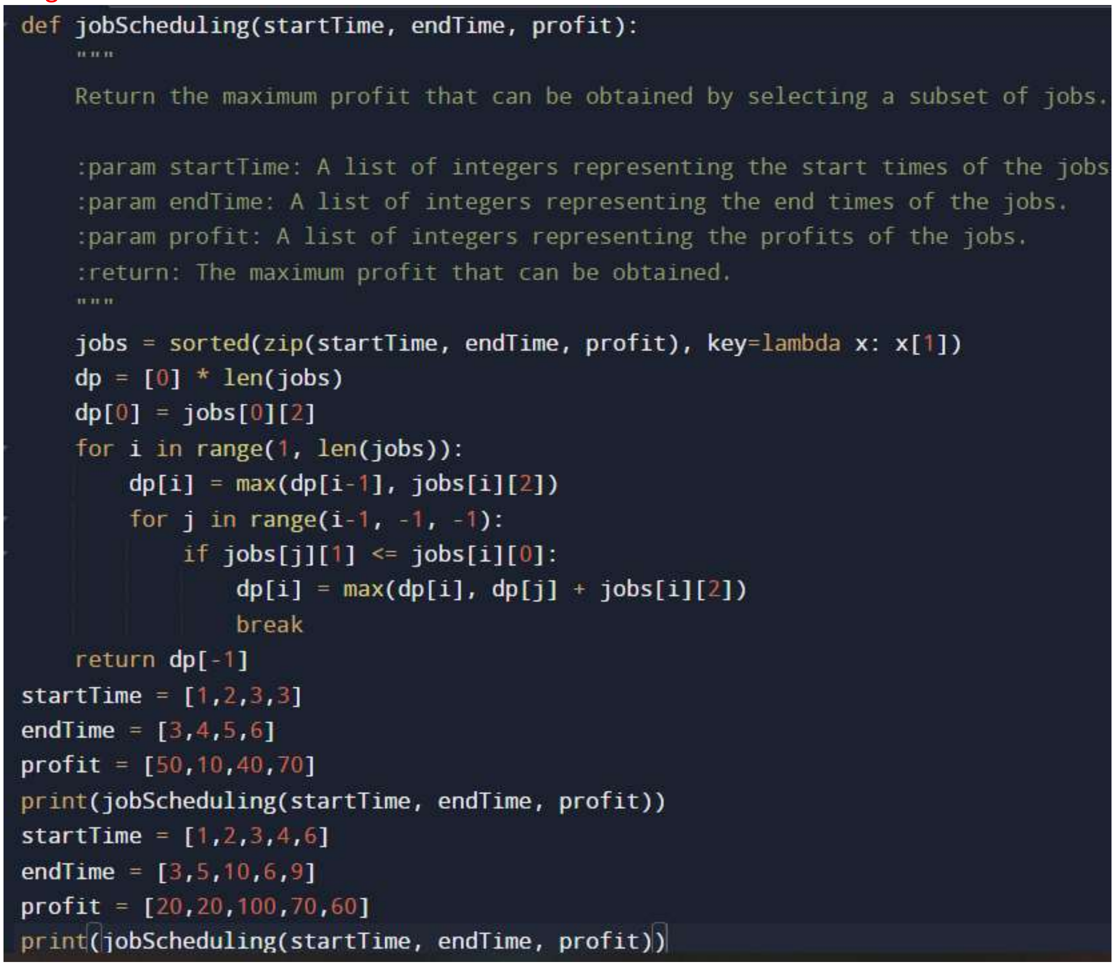
Algorithm:

Step 1 - Sort the jobs based on their end times.

Step 2 - Create a dynamic programming table dp where dp[i] represents the maximum profit that can be obtained by considering the first i jobs.

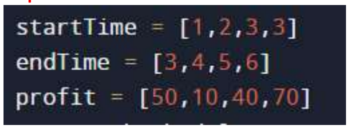
Step 3 - Iterate through the jobs and for each job, find the maximum profit that can be obtained by either including the current job or excluding it, and update the dp table accordingly.

Program:



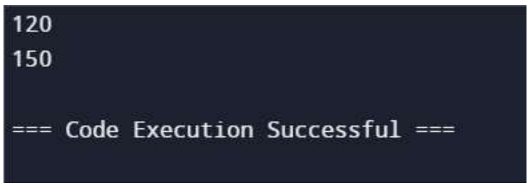
```python
def jobScheduling(startTime, endTime, profit):
    """
    Return the maximum profit that can be obtained by selecting a subset of jobs.

    :param startTime: A list of integers representing the start times of the jobs
    :param endTime: A list of integers representing the end times of the jobs.
    :param profit: A list of integers representing the profits of the jobs.
    :return: The maximum profit that can be obtained.
    """
    jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
    dp = [0] * len(jobs)
    dp[0] = jobs[0][2]
    for i in range(1, len(jobs)):
        dp[i] = max(dp[i-1], jobs[i][2])
        for j in range(i-1, -1, -1):
            if jobs[j][1] <= jobs[i][0]:
                dp[i] = max(dp[i], dp[j] + jobs[i][2])
                break
    return dp[-1]
startTime = [1,2,3,3]
endTime = [3,4,5,6]
profit = [50,10,40,70]
print(jobScheduling(startTime, endTime, profit))
startTime = [1,2,3,4,6]
endTime = [3,5,10,6,9]
profit = [20,20,100,70,60]
print(jobScheduling(startTime, endTime, profit))
```

Input:



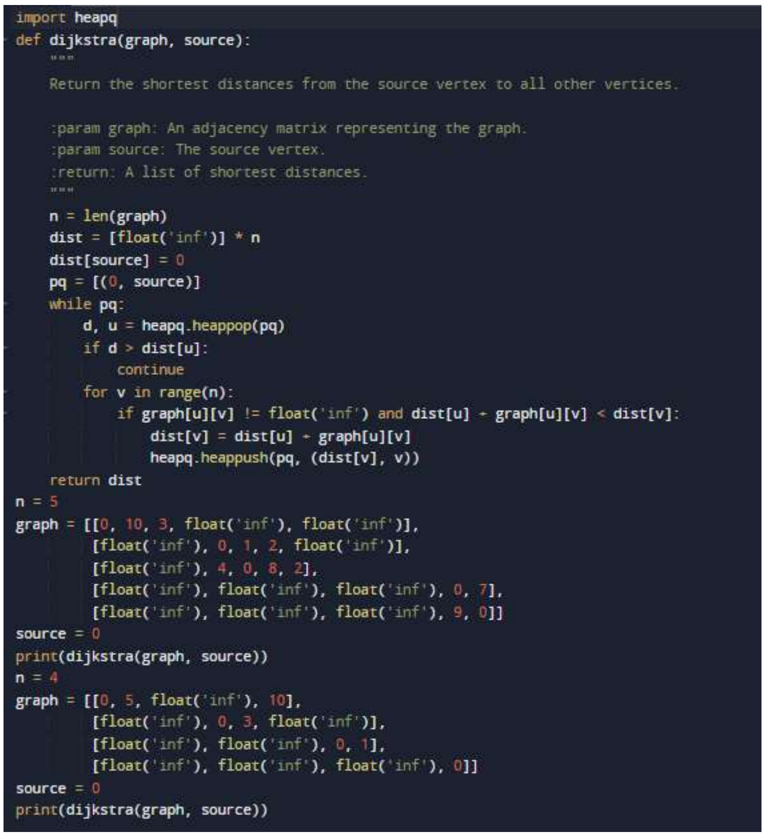
```python
startTime = [1,2,3,3]
endTime = [3,4,5,6]
profit = [50,10,40,70]
```

Output:


```
120
150

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

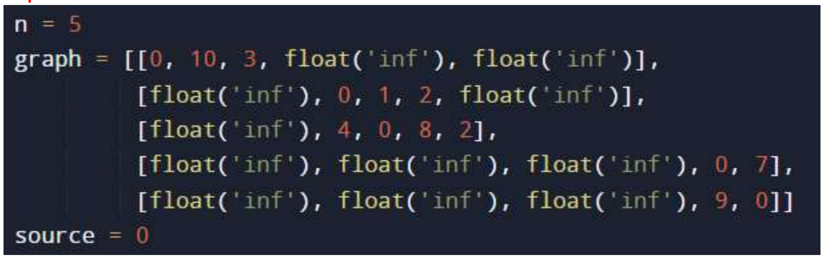
TimeComplexity: T(n)=O(n).
Result: The program outputs the maximum profit that can be obtained, which is 120 for the first example and 150 for the second example.

74. Given a graph represented by an adjacency matrix, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to all other vertices in the graph. The graph is represented as an adjacency matrix where graph[i][j] denote the weight of the edge from vertex i to vertex j. If there is no edge between vertices i and j, the value is Infinity (or a very large number).
Test Case 1:
Input:
n = 5
graph = [[0, 10, 3, Infinity, Infinity], [Infinity, 0, 1, 2, Infinity], [Infinity, 4, 0, 8, 2], [Infinity, Infinity, Infinity, 0, 7], [Infinity, Infinity, Infinity, 9, 0]]
source = 0
Output: [0, 7, 3, 9, 5]
Test Case 2:
Input:
n = 4
graph = [[0, 5, Infinity, 10], [Infinity, 0, 3, Infinity], [Infinity, Infinity, 0, 1], [Infinity, Infinity, Infinity, 0]]
source = 0
Output: [0, 5, 8, 9]
Aim: The aim is to implement Dijkstra's Algorithm to find the shortest path from a given source vertex to all other vertices in the graph.
Algorithm:
Step 1 - Initialize a distance array dist with infinity for all vertices except the source vertex, which is set to 0.
Step 2 - Create a priority queue and add the source vertex to it. While the queue is not empty, extract the vertex with the minimum distance and update the distances of its adjacent vertices.
Step 3 - Repeat step 2 until the queue is empty, and return the distance array dist.
Program:



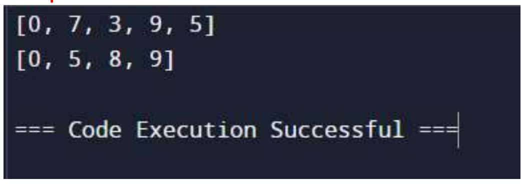
```python
import heapq
def dijkstra(graph, source):
    """
    Return the shortest distances from the source vertex to all other vertices.

    :param graph: An adjacency matrix representing the graph.
    :param source: The source vertex.
    :return: A list of shortest distances.
    """
    n = len(graph)
    dist = [float('inf')] * n
    dist[source] = 0
    pq = [(0, source)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v in range(n):
            if graph[u][v] != float('inf') and dist[u] + graph[u][v] < dist[v]:
                dist[v] = dist[u] + graph[u][v]
                heapq.heappush(pq, (dist[v], v))
    return dist
n = 5
graph = [[0, 10, 3, float('inf'), float('inf')],
         [float('inf'), 0, 1, 2, float('inf')],
         [float('inf'), 4, 0, 8, 2],
         [float('inf'), float('inf'), float('inf'), 0, 7],
         [float('inf'), float('inf'), float('inf'), 9, 0]]
source = 0
print(dijkstra(graph, source))
n = 4
graph = [[0, 5, float('inf'), 10],
         [float('inf'), 0, 3, float('inf')],
         [float('inf'), float('inf'), 0, 1],
         [float('inf'), float('inf'), float('inf'), 0]]
source = 0
print(dijkstra(graph, source))
```

Input:



```python
n = 5
graph = [[0, 10, 3, float('inf'), float('inf')],
         [float('inf'), 0, 1, 2, float('inf')],
         [float('inf'), 4, 0, 8, 2],
         [float('inf'), float('inf'), float('inf'), 0, 7],
         [float('inf'), float('inf'), float('inf'), 9, 0]]
source = 0
```

Output:



```
[0, 7, 3, 9, 5]
[0, 5, 8, 9]

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

TimeComplexity: T(n)=O(n).

Result: The program outputs the shortest distances from the source vertex to all other vertices, which is [0, 7, 3, 9, 5] for the first test case and [0, 5, 8, 9] for the second test case.

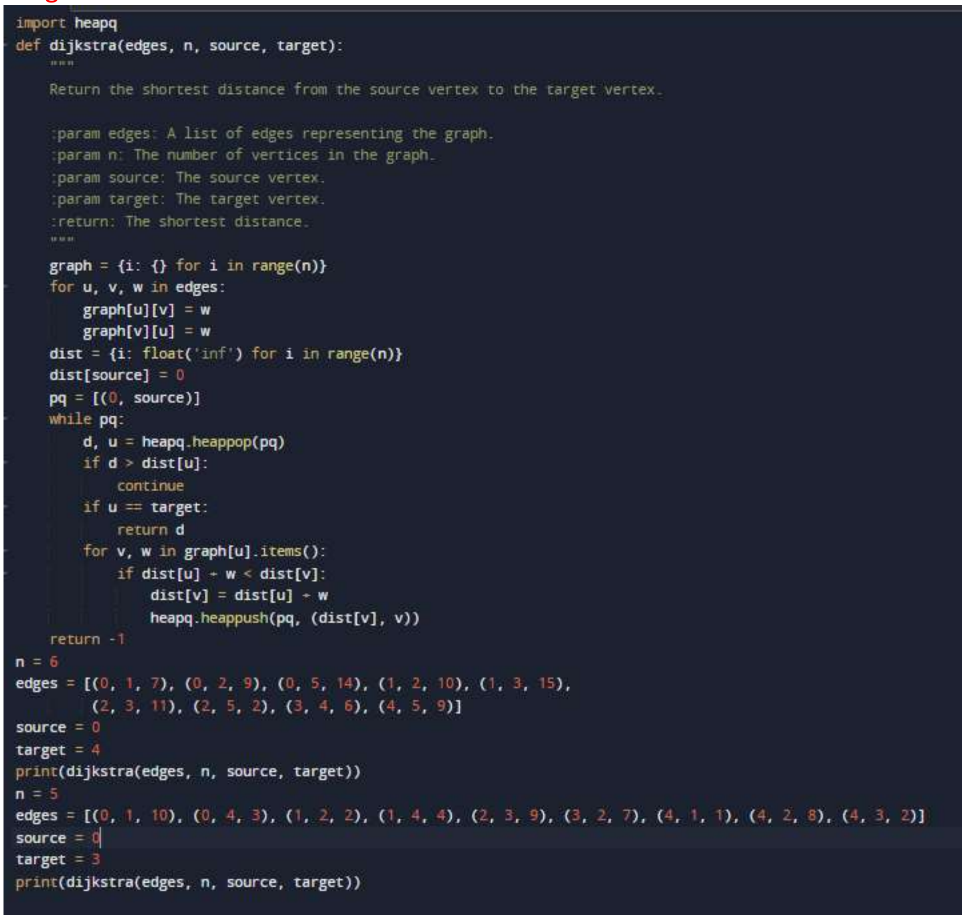
76. Given a graph represented by an edge list, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to a target vertex. The graph is represented as a list of edges where each edge is a tuple (u, v, w) representing an edge from vertex u to vertex v with weight w.

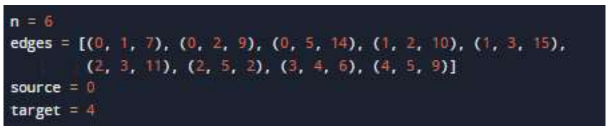
Test Case 1:

Input:

n = 6

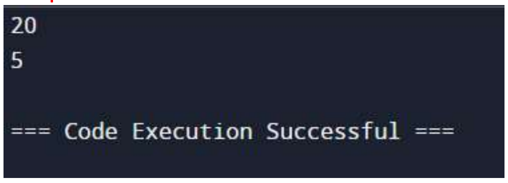
edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),

(2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]
source = 0
target = 4
Output: 20
Test Case 2:
Input:
n = 5
edges = [(0, 1, 10), (0, 4, 3), (1, 2, 2), (1, 4, 4), (2, 3, 9), (3, 2, 7), (4, 1, 1), (4, 2, 8), (4, 3, 2)]
source = 0
target = 3
Output: 8
Aim: The aim is to implement Dijkstra's Algorithm to find the shortest path from a given source vertex to a target vertex in a graph represented by an edge list.
Algorithm:
Step 1 - Create a dictionary to store the adjacency list of the graph, and a dictionary to store the distance to each vertex.
Step 2 - Initialize the distance to the source vertex as 0, and all other vertices as infinity. Create a priority queue and add the source vertex to it.
Step 3 - While the queue is not empty, extract the vertex with the minimum distance, and update the distances of its adjacent vertices. If the target vertex is reached, return the distance.
Program:

```python
import heapq
def dijkstra(edges, n, source, target):
    """
    Return the shortest distance from the source vertex to the target vertex.

    :param edges: A list of edges representing the graph.
    :param n: The number of vertices in the graph.
    :param source: The source vertex.
    :param target: The target vertex.
    :return: The shortest distance.
    """
    graph = {i: {} for i in range(n)}
    for u, v, w in edges:
        graph[u][v] = w
        graph[v][u] = w
    dist = {i: float('inf') for i in range(n)}
    dist[source] = 0
    pq = [(0, source)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        if u == target:
            return d
        for v, w in graph[u].items():
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return -1
n = 6
edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),
         (2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]
source = 0
target = 4
print(dijkstra(edges, n, source, target))
n = 5
edges = [(0, 1, 10), (0, 4, 3), (1, 2, 2), (1, 4, 4), (2, 3, 9), (3, 2, 7), (4, 1, 1), (4, 2, 8), (4, 3, 2)]
source = 0
target = 3
print(dijkstra(edges, n, source, target))
```
Input:



```
n = 6
edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),
         (2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]
source = 0
target = 4
```

Output:



```
20
5

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

TimeComplexity: T(n)=O(n)

Result: The program outputs the shortest distance from the source vertex to the target vertex, which is 20 for the first test case and 8 for the second test case.

77.76. Given a set of characters and their corresponding frequencies, construct the Huffman Tree and generate the Huffman Codes for each character.

Test Case 1:

Input:

n = 4

characters = ['a', 'b', 'c', 'd']

frequencies = [5, 9, 12, 13]

Output: [('a', '110'), ('b', '10'), ('c', '0'), ('d', '111')]

Test Case 2:

Input:

n = 6

characters = ['f', 'e', 'd', 'c', 'b', 'a']

frequencies = [5, 9, 12, 13, 16, 45]

Output: [('a', '0'), ('b', '101'), ('c', '100'), ('d', '111'), ('e', '1101'), ('f', '1100')]

Aim: The aim is to construct a Huffman Tree from a set of characters and their corresponding frequencies, and generate Huffman Codes for each character.

Algorithm:

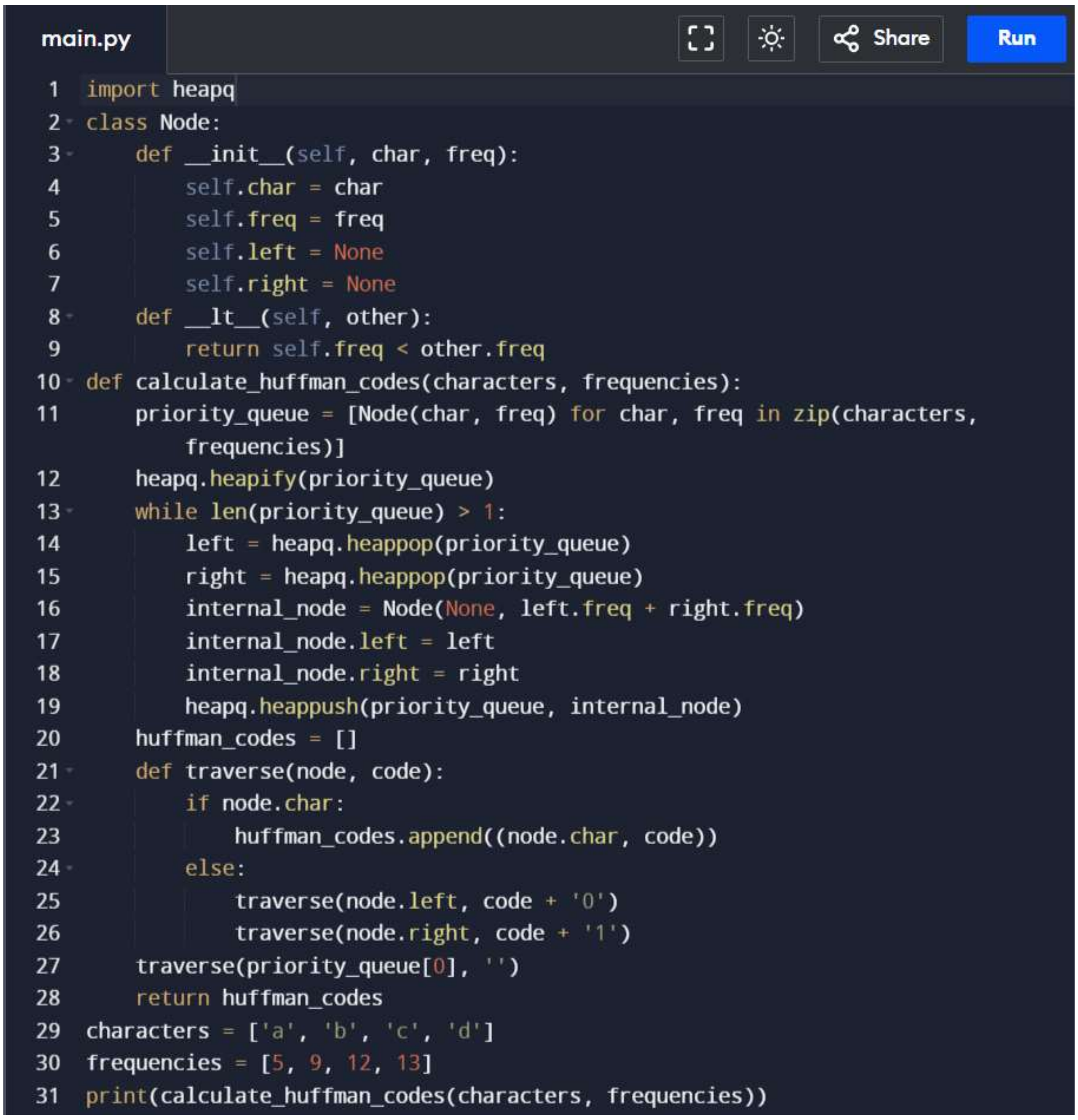
Step 1: Build a priority queue of characters and frequencies.

Step 2: Construct a Huffman Tree by merging nodes with lowest frequencies until only one node is left.

Step 3: Perform a depth-first traversal of the Huffman Tree to generate Huffman Codes for each character.

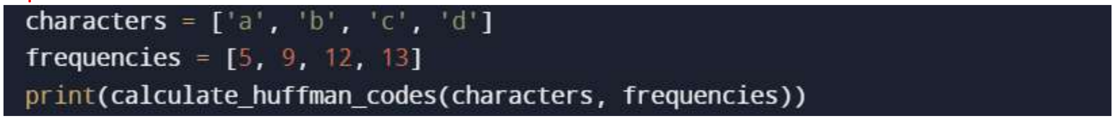
Step 4: Return the list of characters and their corresponding Huffman Codes.

Program:



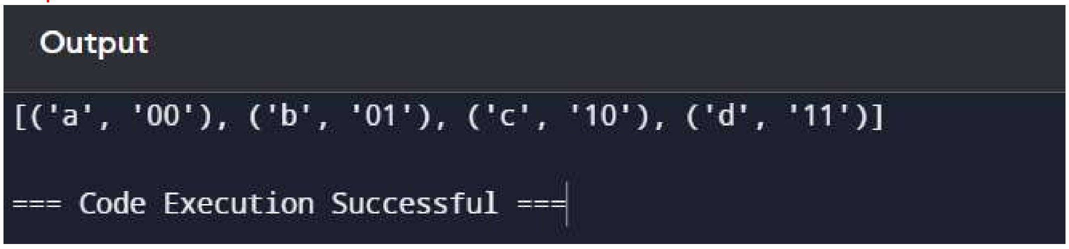
```python
import heapq
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def calculate_huffman_codes(characters, frequencies):
    priority_queue = [Node(char, freq) for char, freq in zip(characters,
        frequencies)]
    heapq.heapify(priority_queue)
    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        internal_node = Node(None, left.freq + right.freq)
        internal_node.left = left
        internal_node.right = right
        heapq.heappush(priority_queue, internal_node)
    huffman_codes = []
    def traverse(node, code):
        if node.char:
            huffman_codes.append((node.char, code))
        else:
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')
    traverse(priority_queue[0], '')
    return huffman_codes
characters = ['a', 'b', 'c', 'd']
frequencies = [5, 9, 12, 13]
print(calculate_huffman_codes(characters, frequencies))
```

Input:



```python
characters = ['a', 'b', 'c', 'd']
frequencies = [5, 9, 12, 13]
print(calculate_huffman_codes(characters, frequencies))
```

output:

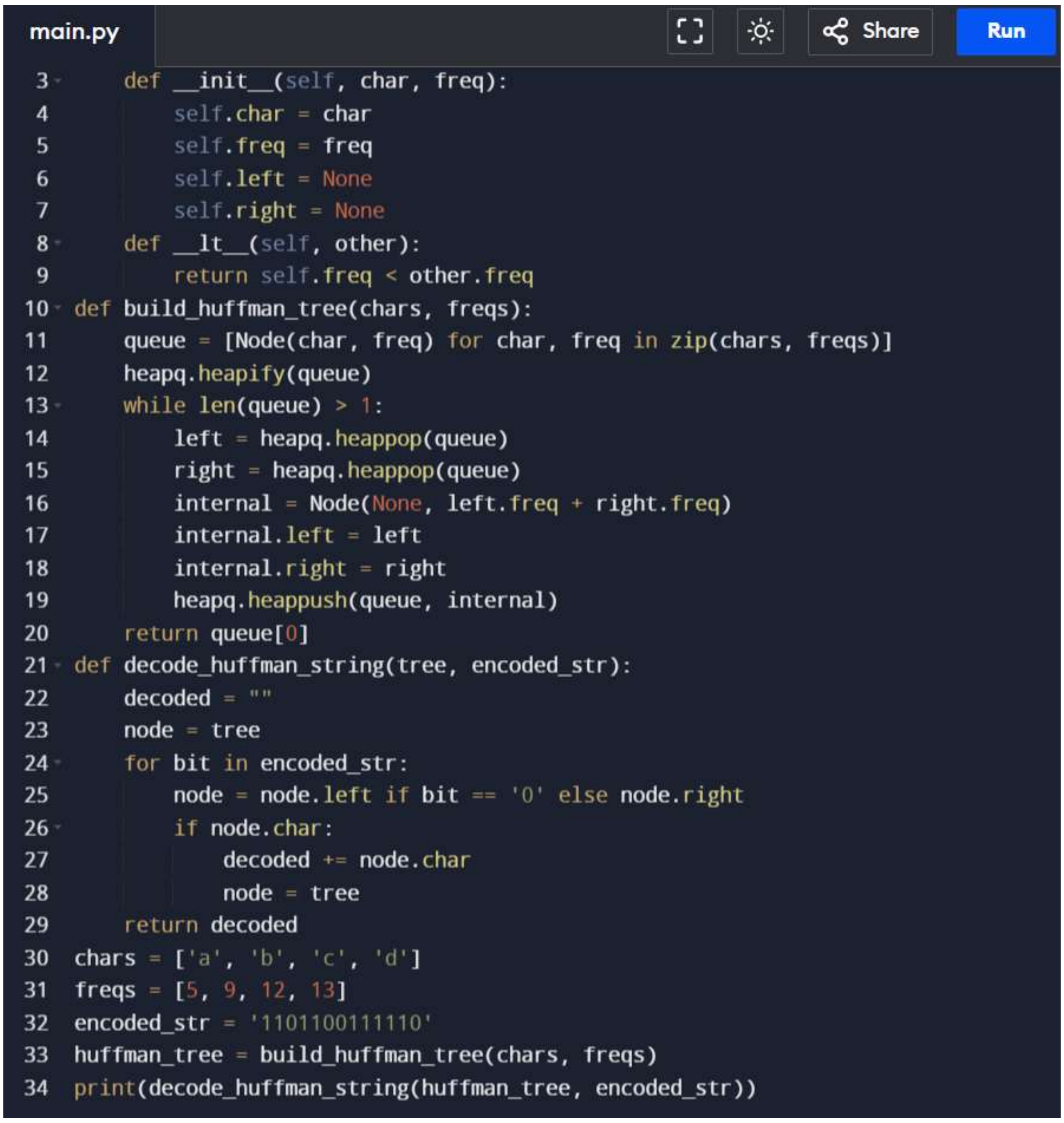


```
[('a', '00'), ('b', '01'), ('c', '10'), ('d', '11')]

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

Time Complexity: T(n)= O(n logn)

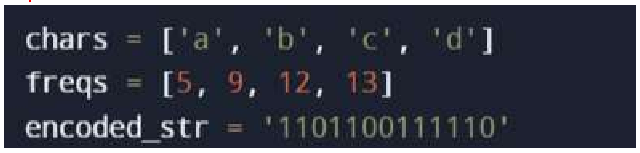
Result:The output of the program is a list of tuples, where each tuple contains a character and its corresponding Huffman Code. The Huffman Codes are generated based on the frequencies of the characters, with more frequent characters having shorter codes.

77. Given a Huffman Tree and a Huffman encoded string, decode the string to get the original message.
Test Case 1:
Input:
n = 4
characters = ['a', 'b', 'c', 'd']
frequencies = [5, 9, 12, 13]
encoded_string = '1101100111110'
Output: "abacd"
Test Case 2:
Input:
n = 6
characters = ['f', 'e', 'd', 'c', 'b', 'a']
frequencies = [5, 9, 12, 13, 16, 45]
encoded_string = '110011011100101111001011'
Output: "fcbade"
Aim: The aim is to decode a Huffman encoded string using a Huffman Tree to get the original message.
Algorithm:
Step 1: Build a Huffman Tree from the given characters and frequencies.
Step 2: Traverse the Huffman Tree based on the bits in the encoded string, moving left for 0 and right for 1.
Step 3: When a leaf node is reached, add the character at the leaf node to the decoded message.
Step 4: Repeat steps 2-3 until the entire encoded string is decoded.
Program:



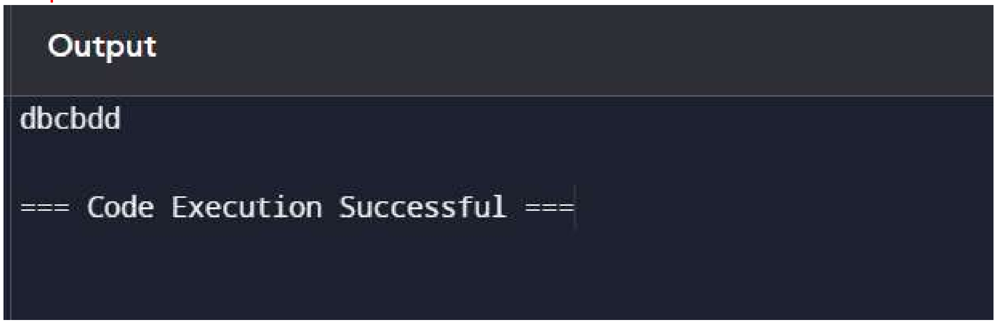
```python
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def build_huffman_tree(chars, freqs):
    queue = [Node(char, freq) for char, freq in zip(chars, freqs)]
    heapq.heapify(queue)
    while len(queue) > 1:
        left = heapq.heappop(queue)
        right = heapq.heappop(queue)
        internal = Node(None, left.freq + right.freq)
        internal.left = left
        internal.right = right
        heapq.heappush(queue, internal)
    return queue[0]
def decode_huffman_string(tree, encoded_str):
    decoded = ""
    node = tree
    for bit in encoded_str:
        node = node.left if bit == '0' else node.right
        if node.char:
            decoded += node.char
            node = tree
    return decoded
chars = ['a', 'b', 'c', 'd']
freqs = [5, 9, 12, 13]
encoded_str = '1101100111110'
huffman_tree = build_huffman_tree(chars, freqs)
print(decode_huffman_string(huffman_tree, encoded_str))
```

Input:



```python
chars = ['a', 'b', 'c', 'd']
freqs = [5, 9, 12, 13]
encoded_str = '1101100111110'
```

Output:

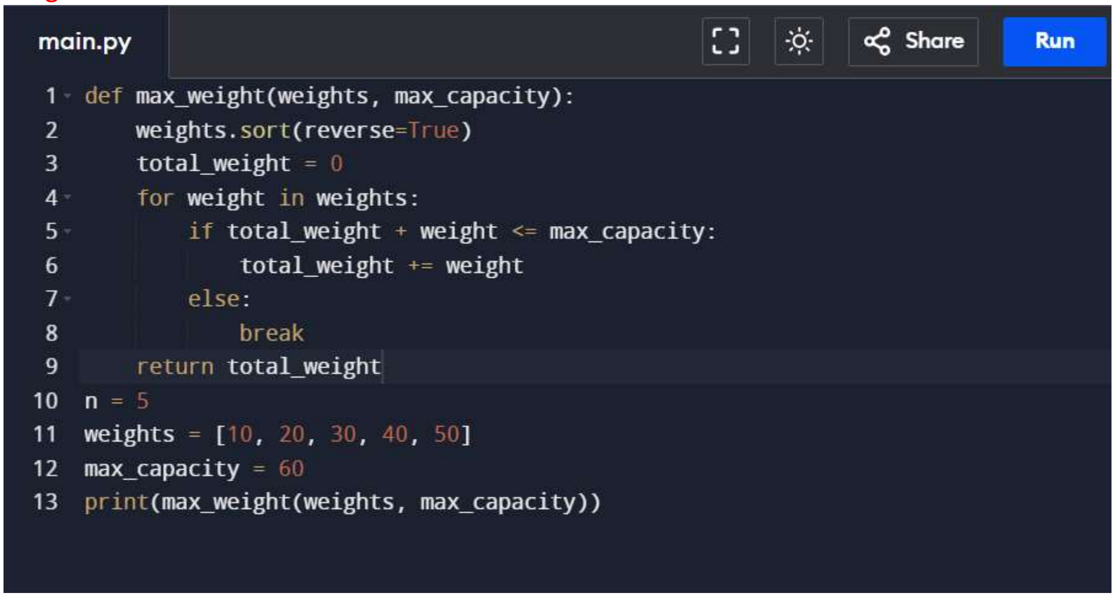


```
dbcbdd

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

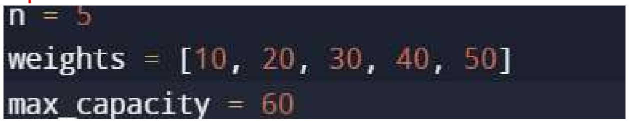
Time Coplexity: T(n)= O(n)

Result: The output of the program is the original message decoded from the

Huffman encoded string.

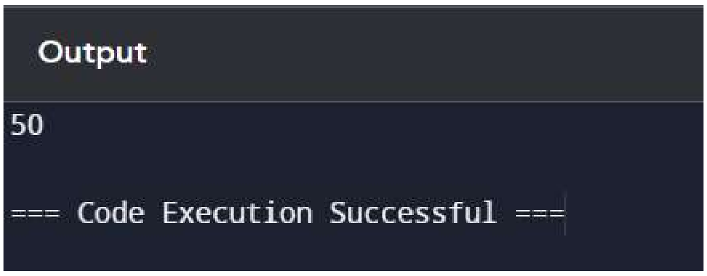
78.Given a list of item weights and the maximum capacity of a container, determine the maximum weight that can be loaded into the container using a greedy approach. The greedy approach should prioritize loading heavier items first until the container reaches its capacity.
Test Case 1:
Input:
n = 5
weights = [10, 20, 30, 40, 50]
max_capacity = 60
Output: 50
Test Case 2:
Input:
n = 6
weights = [5, 10, 15, 20, 25, 30]
max_capacity = 50
Output: 50
Aim: The aim is to determine the maximum weight that can be loaded into a container using a greedy approach, prioritizing heavier items first until the container reaches its capacity.
Algorithm:
Step 1: Sort the list of item weights in descending order.
Step 2: Iterate through the sorted list and add each item's weight to the container until the container reaches its maximum capacity.
Step 3: Return the total weight loaded into the container.
Program:


```python
def max_weight(weights, max_capacity):
    weights.sort(reverse=True)
    total_weight = 0
    for weight in weights:
        if total_weight + weight <= max_capacity:
            total_weight += weight
        else:
            break
    return total_weight
n = 5
weights = [10, 20, 30, 40, 50]
max_capacity = 60
print(max_weight(weights, max_capacity))
```

Input:


```
n = 5
weights = [10, 20, 30, 40, 50]
max_capacity = 60
```

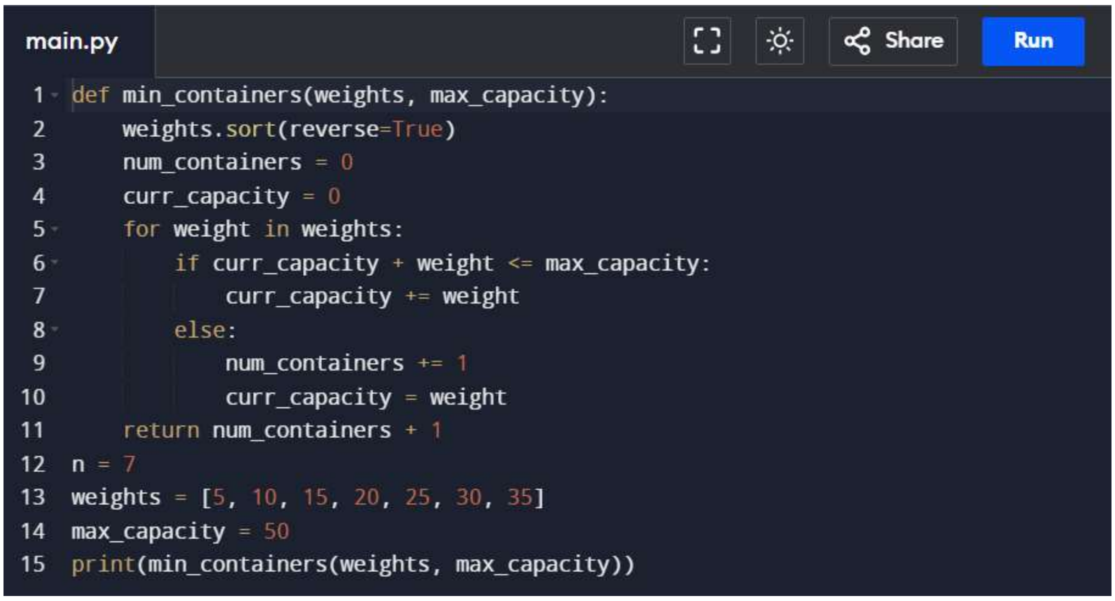
Output:


Output

```
50

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

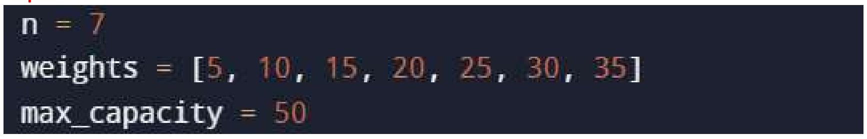
Time Complexity: T(n)= O(n logn)
Result:The output of the program is the maximum weight that can be loaded into the container using a greedy approach.

79.Given a list of item weights and a maximum capacity for each container, determine the minimum number of containers required to load all items using a greedy approach. The greedy approach should prioritize loading items into the current container until it is full before moving to the next container.
Test Case 1:
Input:
n = 7
weights = [5, 10, 15, 20, 25, 30, 35]
max_capacity = 50
Output: 4
Test Case 2:
Input:
n = 8
weights = [10, 20, 30, 40, 50, 60, 70, 80]
max_capacity = 100
Output: 6
Aim: Determine the minimum number of containers required to load all items using a greedy approach.
Algorithm:
Step-1: Sort the item weights in descending order.
Step-2: Initialize the number of containers and the current container capacity.
Step-3: Iterate through the sorted item weights, placing each item in the current container until it is full or all items are placed.
Step-4: If the current container is full, increment the number of containers and reset the current container capacity.
Program:



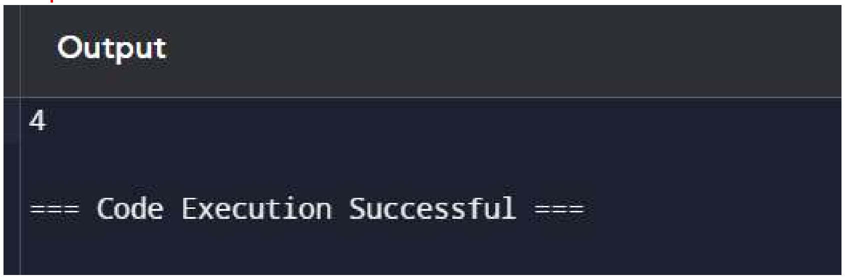
```python
def min_containers(weights, max_capacity):
    weights.sort(reverse=True)
    num_containers = 0
    curr_capacity = 0
    for weight in weights:
        if curr_capacity + weight <= max_capacity:
            curr_capacity += weight
        else:
            num_containers += 1
            curr_capacity = weight
    return num_containers + 1
n = 7
weights = [5, 10, 15, 20, 25, 30, 35]
max_capacity = 50
print(min_containers(weights, max_capacity))
```

Input:



```
n = 7
weights = [5, 10, 15, 20, 25, 30, 35]
max_capacity = 50
```

Output:



```
4

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

Time Complexity: T(n)= O(n log n)

Result:The program returns the minimum number of containers required to load all items.

80. Given a graph represented by an edge list, implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) and its total weight.

Test Case 1:

Input:

n = 4

m = 5

edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

Output:

Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

Total weight of MST: 19

Test Case 2:

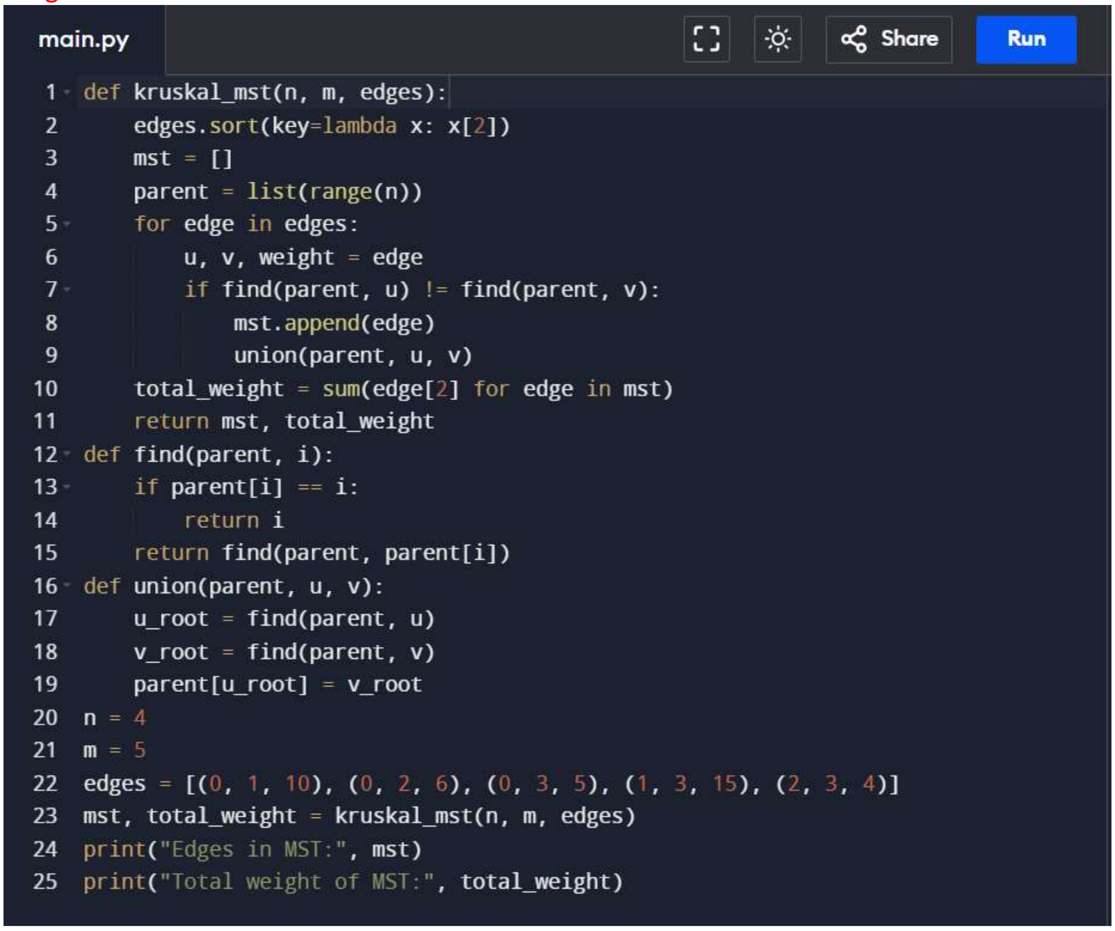
Input:

n = 5

m = 7

edges = [(0, 1, 2), (0, 3, 6), (1, 2, 3), (1, 3, 8), (1, 4, 5), (2, 4, 7), (3, 4, 9)]

Output:

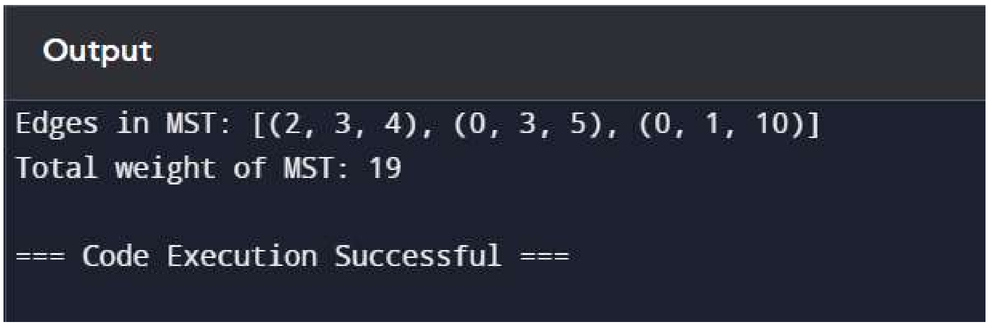
Edges in MST: [(0, 1, 2), (1, 2, 3), (1, 4, 5), (0, 3, 6)]
Total weight of MST: 16
Aim: implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) and its total weight.
Algorithm:
Step-1: Sort the edges in non-decreasing order of their weights.
Step-2: Initialize an empty Minimum Spanning Tree (MST) and a disjoint set.
Step-3: Iterate through the sorted edges and add them to the MST if they do not form a cycle.
Step-4:Return the MST and its total weight.
Program:


```python
def kruskal_mst(n, m, edges):
    edges.sort(key=lambda x: x[2])
    mst = []
    parent = list(range(n))
    for edge in edges:
        u, v, weight = edge
        if find(parent, u) != find(parent, v):
            mst.append(edge)
            union(parent, u, v)
    total_weight = sum(edge[2] for edge in mst)
    return mst, total_weight
def find(parent, i):
    if parent[i] == i:
        return i
    return find(parent, parent[i])
def union(parent, u, v):
    u_root = find(parent, u)
    v_root = find(parent, v)
    parent[u_root] = v_root
n = 4
m = 5
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
mst, total_weight = kruskal_mst(n, m, edges)
print("Edges in MST:", mst)
print("Total weight of MST:", total_weight)
```

Input:

```
n = 4
m = 5
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
```

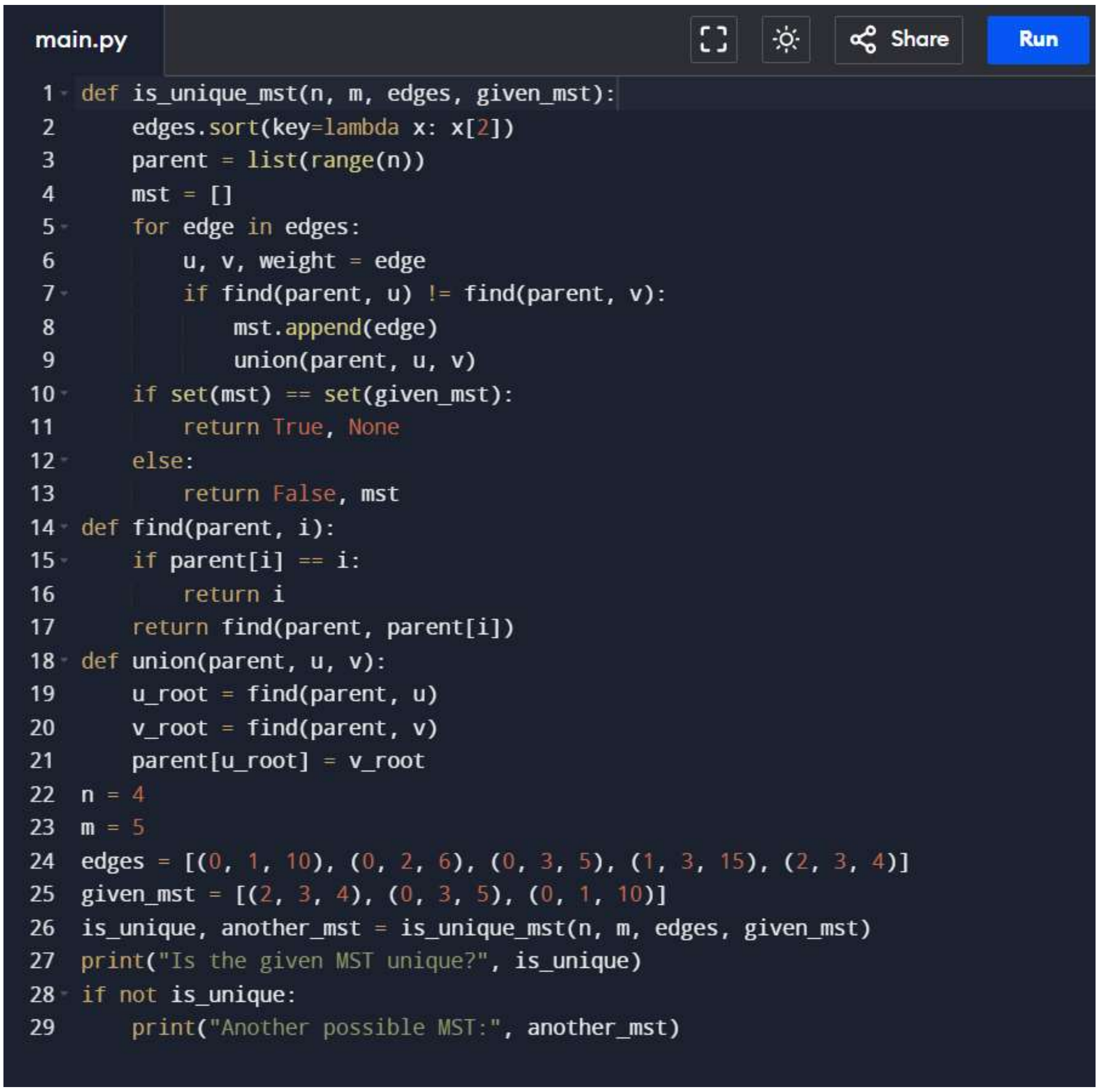
Output:



```
Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
Total weight of MST: 19

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

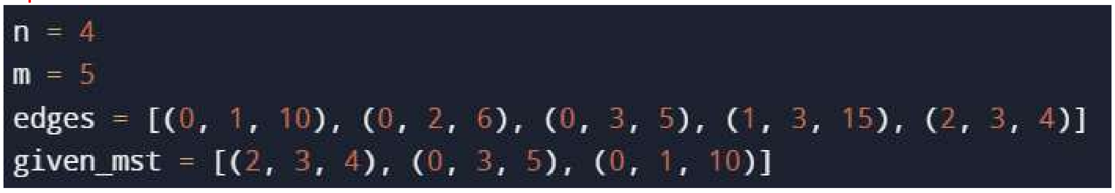
Time Complexity: The time complexity of Kruskal's Algorithm is O(E log E) or O(E log V), where E is the number of edges and V is the number of vertices. This is because we sort the edges in non-decreasing order of their weights, which takes O(E log E) time, and then iterate through the sorted edges, which takes O(E) time.
Result: The program outputs the Minimum Spanning Tree (MST) and its total weight for the given test cases.

81. Given a graph with weights and a potential Minimum Spanning Tree (MST), verify if the given MST is unique. If it is not unique, provide another possible MST.
Test Case 1:
Input:
n = 4
m = 5
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
given_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
Output: Is the given MST unique? True
Test Case 2:
Input:
n = 5
m = 6
edges = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (2, 3, 2), (3, 4, 3), (4, 2, 3)]
given_mst = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (3, 4, 3)]
Output: Is the given MST unique? False
Another possible MST: [(0, 1, 1), (0, 2, 1), (2, 3, 2), (3, 4, 3)]
Total weight of MST: 7
Aim: implementation to verify if the given MST is unique and provide another possible MST if it is not unique.
Algorithm:
Step-1: Sort the edges in non-decreasing order of their weights.
Step-2: Construct a Minimum Spanning Tree (MST) using Kruskal's Algorithm.
Step-3: Compare the constructed MST with the given MST.
Step-4: If they are not the same, return False and another possible MST.
Program:



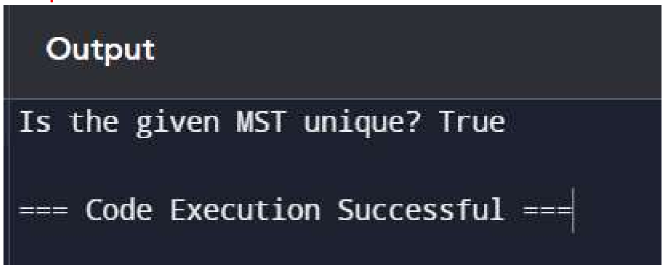
```python
def is_unique_mst(n, m, edges, given_mst):
    edges.sort(key=lambda x: x[2])
    parent = list(range(n))
    mst = []
    for edge in edges:
        u, v, weight = edge
        if find(parent, u) != find(parent, v):
            mst.append(edge)
            union(parent, u, v)
    if set(mst) == set(given_mst):
        return True, None
    else:
        return False, mst
def find(parent, i):
    if parent[i] == i:
        return i
    return find(parent, parent[i])
def union(parent, u, v):
    u_root = find(parent, u)
    v_root = find(parent, v)
    parent[u_root] = v_root
n = 4
m = 5
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
given_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
is_unique, another_mst = is_unique_mst(n, m, edges, given_mst)
print("Is the given MST unique?", is_unique)
if not is_unique:
    print("Another possible MST:", another_mst)
```

Input:



```python
n = 4
m = 5
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
given_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
```

Output:



```
Is the given MST unique? True

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

Time Complexity: The time complexity of this implementation is O(E log E) or O(E log V), where E is the number of edges and V is the number of vertices. This is because we sort the edges in non-decreasing order of their weights, which takes O(E log E) time, and then construct the MST using Kruskal's Algorithm, which takes O(E log V) time.

Result:The program outputs whether the given MST is unique and provides another possible MST if it is not unique.