

## TOPIC 5: GREEDY

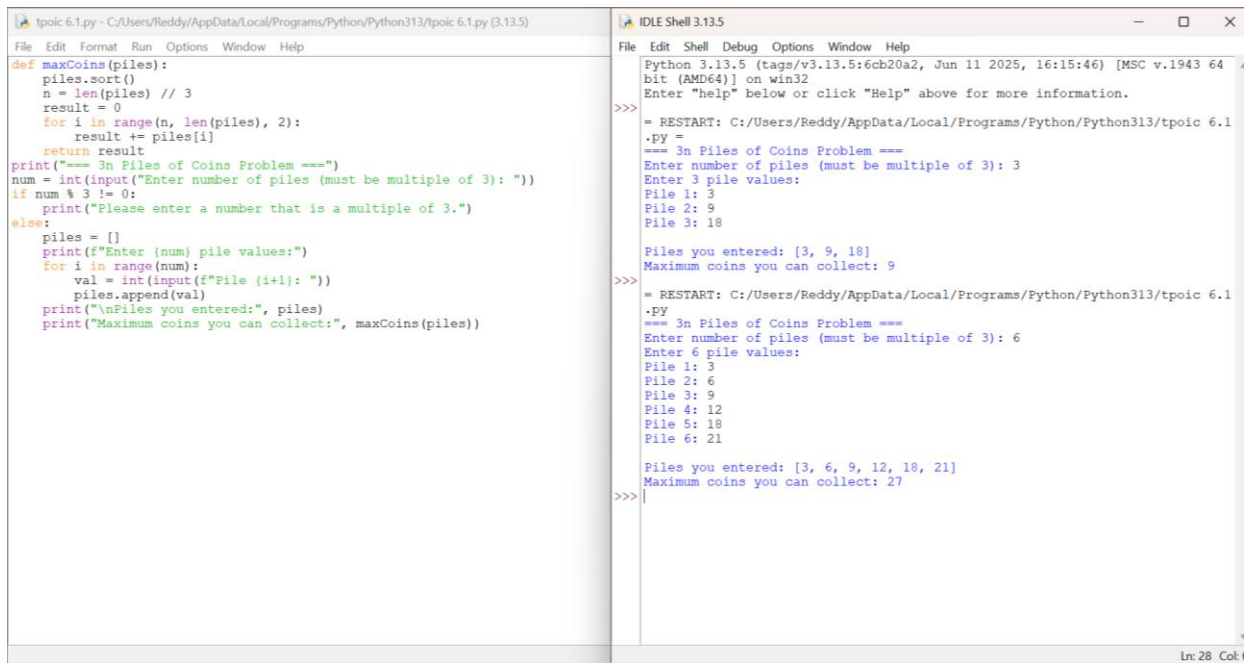
### 1.Maximum of Coins

**Aim:** To find 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: -**



The screenshot displays a Python IDE with two windows. The left window shows the source code for a program named 'tpoic 6.1.py'. The code defines a function 'maxCoins(piles)' that sorts the piles and sums every third element. It then prompts the user for the number of piles (must be a multiple of 3) and the values of the piles. The right window shows the IDLE Shell output, which includes the program's execution for two test cases. In the first case, 3 piles with values [3, 9, 18] are entered, resulting in a maximum of 9 coins. In the second case, 6 piles with values [3, 6, 9, 12, 18, 21] are entered, resulting in a maximum of 27 coins.

```
def maxCoins(piles):
    piles.sort()
    n = len(piles) // 3
    result = 0
    for i in range(n, len(piles), 2):
        result += piles[i]
    return result
print("=== 3n Piles of Coins Problem ===")
num = int(input("Enter number of piles (must be multiple of 3): "))
if num % 3 != 0:
    print("Please enter a number that is a multiple of 3.")
else:
    piles = []
    print(f"Enter {num} pile values:")
    for i in range(num):
        val = int(input(f"Pile {i+1}: "))
        piles.append(val)
    print("\nPiles you entered:", piles)
    print("Maximum coins you can collect:", maxCoins(piles))
```

```
>>>
= RESTART: C:/Users/Reddy/AppData/Local/Programs/Python/Python313/tpoic 6.1
.py =
=== 3n Piles of Coins Problem ===
Enter number of piles (must be multiple of 3): 3
Enter 3 pile values:
Pile 1: 3
Pile 2: 9
Pile 3: 18

Piles you entered: [3, 9, 18]
Maximum coins you can collect: 9
>>>
= RESTART: C:/Users/Reddy/AppData/Local/Programs/Python/Python313/tpoic 6.1
.py =
=== 3n Piles of Coins Problem ===
Enter number of piles (must be multiple of 3): 6
Enter 6 pile values:
Pile 1: 3
Pile 2: 6
Pile 3: 9
Pile 4: 12
Pile 5: 18
Pile 6: 21

Piles you entered: [3, 6, 9, 12, 18, 21]
Maximum coins you can collect: 27
>>>
```

### 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: -

```

topic 6.2.py - C:/Users/Reddy/AppData/Local/Programs/Python/Python313/topic 6.2...
File Edit Format Run Options Window Help
def minPatches(coins, target):
    coins.sort() # Sort coins in ascending order
    added_coins = 0
    reachable = 1
    i = 0

    while reachable <= target:
        if i < len(coins) and coins[i] <= reachable:
            reachable += coins[i]
            i += 1
        else:
            reachable += reachable
            added_coins += 1

    return added_coins

# --- Main Program ---
print("=== Minimum Coins to Cover Range [1, target] ===")

# User input
coins = list(map(int, input("Enter coin values separated by spaces: ").split))
target = int(input("Enter target value: "))

# Output result
print("Minimum coins needed to be added:", minPatches(coins, target))

IDLE Shell 3.13.5
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64
bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.

>>>
= RESTART: C:/Users/Reddy/AppData/Local/Programs/Python/Python313/topic 6.
2.py =
=== Minimum Coins to Cover Range [1, target] ===
Enter coin values separated by spaces: 3 4 6 8 9
Enter target value: 2
Minimum coins needed to be added: 2
>>>

```

### 3. Maximum Working time

#### Aim:

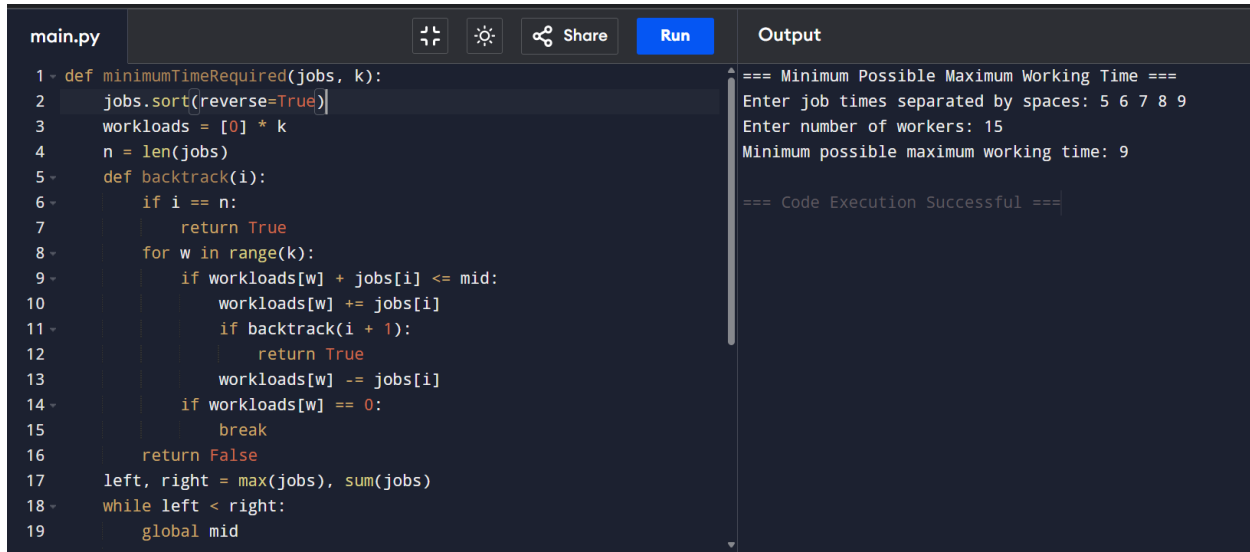
To write a Python program that assigns n jobs among k each job is assigned to exactly one worker.

#### 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  [Icons] [Share] [Run] Output
1 - def minimumTimeRequired(jobs, k):
2     jobs.sort(reverse=True)
3     workloads = [0] * k
4     n = len(jobs)
5     def backtrack(i):
6         if i == n:
7             return True
8         for w in range(k):
9             if workloads[w] + jobs[i] <= mid:
10                workloads[w] += jobs[i]
11                if backtrack(i + 1):
12                    return True
13                workloads[w] -= jobs[i]
14            if workloads[w] == 0:
15                break
16        return False
17    left, right = max(jobs), sum(jobs)
18    while left < right:
19        global mid
```

```
=== 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

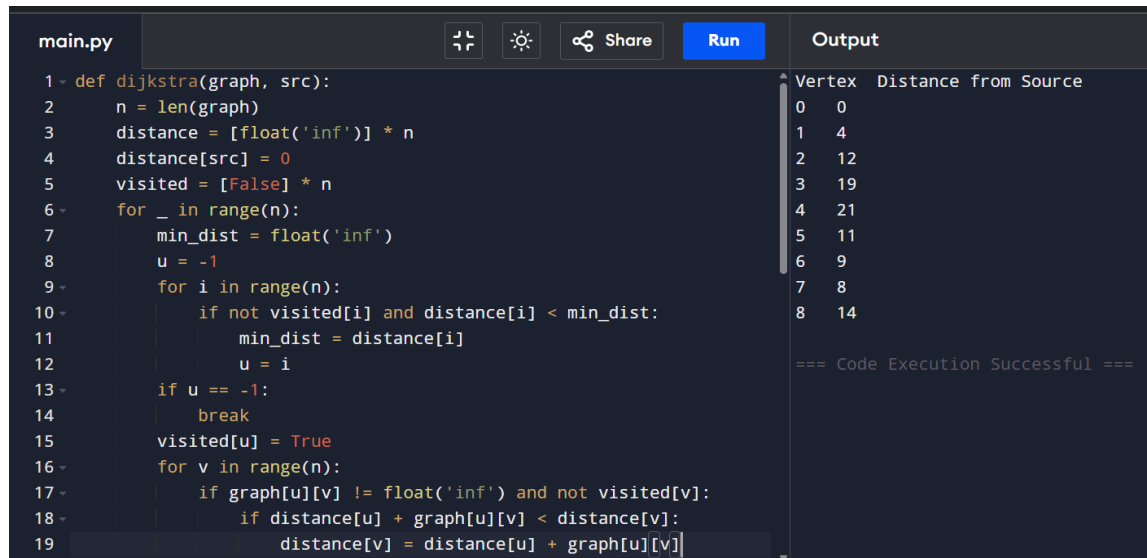
main.py	Output
<pre>1 from bisect import bisect_right 2 def jobScheduling(startTime, endTime, profit): 3     jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1]) 4     n = len(jobs) 5     dp = [0] * n 6     end_times = [job[1] for job in jobs] 7     for i in range(n): 8         incl_profit = jobs[i][2] 9         idx = bisect_right(end_times, jobs[i][0]) - 1 10        if idx != -1: 11            incl_profit += dp[idx] 12        dp[i] = max(incl_profit, dp[i-1] if i &gt; 0 else 0) 13    return dp[-1] 14 startTime = [1, 2, 3, 3] 15 endTime = [3, 4, 5, 6] 16 profit = [50, 10, 40, 70] 17 print("Maximum Profit:", jobScheduling(startTime, endTime, profit))</pre>	<pre>Maximum Profit: 120 === Code Execution Successful ===</pre>

## 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



The screenshot shows a code editor with a file named 'main.py'. The code implements Dijkstra's algorithm. The output panel on the right displays the results of the algorithm, showing the shortest distance from the source vertex to each other vertex in the graph.

```
1 def dijkstra(graph, src):
2     n = len(graph)
3     distance = [float('inf')] * n
4     distance[src] = 0
5     visited = [False] * n
6     for _ in range(n):
7         min_dist = float('inf')
8         u = -1
9         for i in range(n):
10            if not visited[i] and distance[i] < min_dist:
11                min_dist = distance[i]
12                u = i
13        if u == -1:
14            break
15        visited[u] = True
16        for v in range(n):
17            if graph[u][v] != float('inf') and not visited[v]:
18                if distance[u] + graph[u][v] < distance[v]:
19                    distance[v] = distance[u] + graph[u][v]
```

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 ===

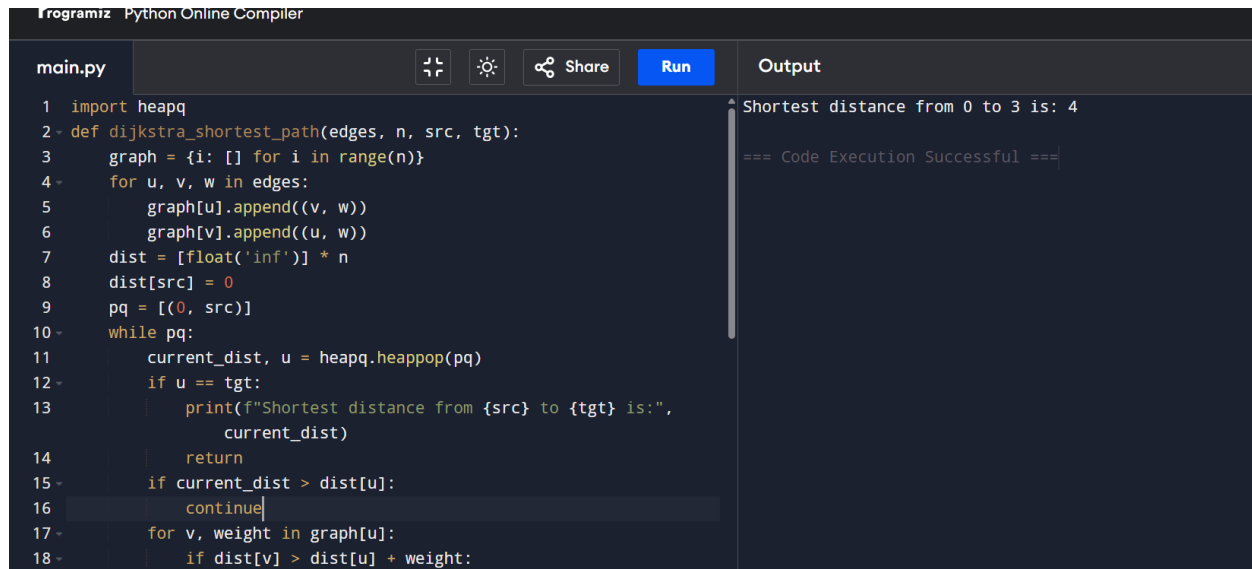
## 5.Shortest Path (vertex to target)

**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



The screenshot shows a web-based Python compiler interface. The top bar includes the text 'Programiz Python Online Compiler' and several icons: a window icon, a sun icon, a share icon, and a 'Run' button. Below the bar, the code editor displays a file named 'main.py' containing the following Python code:

```
1 import heapq
2 def dijkstra_shortest_path(edges, n, src, tgt):
3     graph = {i: [] for i in range(n)}
4     for u, v, w in edges:
5         graph[u].append((v, w))
6         graph[v].append((u, w))
7     dist = [float('inf')] * n
8     dist[src] = 0
9     pq = [(0, src)]
10    while pq:
11        current_dist, u = heapq.heappop(pq)
12        if u == tgt:
13            print(f"Shortest distance from {src} to {tgt} is:",
14                  current_dist)
15            return
16        if current_dist > dist[u]:
17            continue
18        for v, weight in graph[u]:
19            if dist[v] > dist[u] + weight:
```

The output panel on the right shows the result of the code execution:

```
Shortest distance from 0 to 3 is: 4
=== Code Execution Successful ===
```

## 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	Output
<pre>1 import heapq 2 def dijkstra(graph, source, target): 3     distances = {vertex: float('inf') for vertex in graph} 4     distances[source] = 0 5     pq = [(0, source)] 6     while pq: 7         current_dist, current_vertex = heapq.heappop(pq) 8         if current_vertex == target: 9             return distances[target] 10        if current_dist &gt; distances[current_vertex]: 11            continue 12        for neighbor, weight in graph[current_vertex]: 13            distance = current_dist + weight 14            if distance &lt; distances[neighbor]: 15                distances[neighbor] = distance 16                heapq.heappush(pq, (distance, neighbor)) 17    return float('inf') 18 graph = {} 19 n = int(input("Enter number of vertices: "))</pre>	<pre>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 ===</pre>

## 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	Output
<pre>1 import heapq 2 class Node: 3     def __init__(self, char, freq): 4         self.char = char 5         self.freq = freq 6         self.left = self.right = None 7     def __lt__(self, other): 8         return self.freq &lt; other.freq 9 def huffman_codes(characters, frequencies): 10     heap = [Node(characters[i], frequencies[i]) for i in range(len     (characters))] 11     heapq.heapify(heap) 12     while len(heap) &gt; 1: 13         left = heapq.heappop(heap) 14         right = heapq.heappop(heap) 15         merged = Node(None, left.freq + right.freq) 16         merged.left = left 17         merged.right = right 18         heapq.heappush(heap, merged)</pre>	<pre>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 ===</pre>

## 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	Output
<pre>1 import heapq 2 class Node: 3     def __init__(self, char, freq): 4         self.char = char 5         self.freq = freq 6         self.left = None 7         self.right = None 8     def __lt__(self, other): 9         return self.freq &lt; other.freq 10 def build_huffman_tree(chars, freqs): 11     heap = [Node(chars[i], freqs[i]) for i in range(len(chars))] 12     heapq.heapify(heap) 13     while len(heap) &gt; 1: 14         left = heapq.heappop(heap) 15         right = heapq.heappop(heap) 16         merged = Node(None, left.freq + right.freq) 17         merged.left = left 18         merged.right = right 19         heapq.heappush(heap, merged)</pre>	<pre>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 ===</pre>

## 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	Output
<pre>1 def greedy_load(weights, max_capacity): 2     weights.sort(reverse=True) 3     total = 0 4     for w in weights: 5         if total + w &lt;= max_capacity: 6             total += w 7     return total 8 n = int(input("Enter number of items: ")) 9 weights = list(map(int, input("Enter item weights: ").split())) 10 max_capacity = int(input("Enter max capacity: ")) 11 print("Maximum weight loaded:", greedy_load(weights, max_capacity))</pre>	<pre>Enter number of items: 4 Enter item weights: 23 45 67 89 Enter max capacity: 165 Maximum weight loaded: 156  === Code Execution Successful ===</pre>

## 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	Output
<pre>1 def min_containers(weights, max_capacity): 2     containers = 1 3     current = 0 4     for w in weights: 5         if current + w &lt;= max_capacity: 6             current += w 7         else: 8             containers += 1 9             current = w 10    return containers 11 n = int(input("Enter number of items: ")) 12 weights = list(map(int, input("Enter item weights: ").split())) 13 max_capacity = int(input("Enter container capacity: ")) 14 print("Minimum containers required:", min_containers(weights,     max_capacity))</pre>	<pre>Enter number of items: 5 Enter item weights: 3 43 65 98 10 Enter container capacity: 95 Minimum containers required: 4  === Code Execution Successful ===</pre>

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

**Aim:** To find the MST and 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  [Icons] [Share] [Run]  Output
1 def find(parent, i):
2     if parent[i] != i:
3         parent[i] = find(parent, parent[i])
4     return parent[i]
5 def union(parent, rank, x, y):
6     rootX = find(parent, x)
7     rootY = find(parent, y)
8     if rootX != rootY:
9         if rank[rootX] < rank[rootY]:
10            parent[rootX] = rootY
11        elif rank[rootX] > rank[rootY]:
12            parent[rootY] = rootX
13        else:
14            parent[rootY] = rootX
15            rank[rootX] += 1
16 def kruskal(n, edges):
17     edges.sort(key=lambda x: x[2])
18     parent = [i for i in range(n)]
19     rank = [0] * n

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  [Icons] [Share] [Run]  Output
1 class DisjointSet:
2     def __init__(self, n):
3         self.parent = [i for i in range(n)]
4         self.rank = [0] * n
5     def find(self, x):
6         if self.parent[x] != x:
7             self.parent[x] = self.find(self.parent[x])
8         return self.parent[x]
9     def union(self, x, y):
10        root_x = self.find(x)
11        root_y = self.find(y)
12        if root_x != root_y:
13            if self.rank[root_x] > self.rank[root_y]:
14                self.parent[root_y] = root_x
15            elif self.rank[root_x] < self.rank[root_y]:
16                self.parent[root_x] = root_y
17            else:
18                self.parent[root_y] = root_x
19                self.rank[root_x] += 1

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 ===
```