##Task-1

This Python code implements Dijkstra's algorithm to find the shortest distances from a given source node in a graph. It starts by importing the `heapq` module for heap operations. The `dijkstra` function initializes distances with infinity for all nodes except the starting node, which is set to 0. It then uses a priority queue (implemented as a heap) to iteratively explore nodes and update their distances.

The code reads input from a file `Input1.txt`, where the first line contains two integers `N` and `M` representing the number of nodes and edges, respectively. The subsequent `M` lines contain three integers `u`, `v`, and `w`, denoting an edge from node `u` to node `v` with weight `w`. The last line in the file contains the source node.

After computing the shortest distances using Dijkstra's algorithm, the code writes the resulting distances to `Output1.txt`. If a node is unreachable (distance remains `float('inf')`), it writes `-1` for that distance in the output file.

##Task-2

This code extends the previous Dijkstra's algorithm implementation to find the shortest meeting time and meeting node between two given nodes `S` and `T` in a graph. It follows a similar structure to the previous code but introduces additional logic at the end.

After reading input from `Input2.txt`, the code computes the shortest distances from nodes `S` and `T` to all other nodes using the `dijkstra` function.

Then, it iterates through all nodes and calculates the sum of distances from `S` to the current node and from `T` to the same node. It identifies the node that minimizes this sum, indicating the potential meeting point.

Finally, it writes the result to `Output2.txt`. If no meeting point is found (i.e., `min_time` remains infinite), it writes "Impossible" to the output file. Otherwise, it writes the minimum time required to meet (`min_time`) and the corresponding meeting node.

##Task-3

This code processes a series of friendships among a group of individuals. It reads input from `Input3.txt`, where the first line contains two integers `N` and `K`, denoting the number of individuals and the number of friendships, respectively.

Initially, it initializes a `parent` array and a `size` array. The `parent` array keeps track of the parent node in a group, while the `size` array maintains the size of each group.

It employs the concepts of union-find (disjoint-set union) to track connections between individuals. The `find` function implements path compression to find the root of a group, and the `union` function merges two groups by linking their roots.

For each friendship in the subsequent lines of the input file, it merges the groups of the two individuals involved in the friendship using the `union` function. It then writes the size of the resulting friend circle to `Output3.txt`.

##Task-4

This script reads input from `Input4.txt`, where the first line contains two integers `N` and `M`, denoting the number of nodes and edges, respectively. It initializes an empty list `edges` to store edges and a `parent` array to track the parent node in each disjoint set.

The script implements the union-find (disjoint-set union) algorithm to find the minimum spanning tree (MST) of a graph represented by the edges provided in the input file. The `find` function implements path compression, and the `union` function merges two disjoint sets based on their roots.

It reads each edge (represented by `u`, `v`, and `w` denoting two nodes and the edge weight) from the input file and adds them to the `edges` list. Then, it sorts the edges based on their weights in ascending order.

Next, the script iterates through the sorted edges. For each edge, if the nodes `u` and `v` belong to different sets (i.e., `find(u) != find(v)`), it merges these sets using the `union` function and adds the weight of the edge to the `total_cost`. This ensures that it constructs the minimum spanning tree by considering edges in ascending order of weight without creating cycles.

Finally, it writes the total cost of the minimum spanning tree to `Output4.txt`. This algorithm efficiently determines the minimum cost required to connect all nodes in the graph without forming any cycles, giving the minimum spanning tree cost.