Q3)

The fundamental concept behind this algorithm is to utilize a two-pass Depth-First Search (DFS) approach to identify the strongly connected components (SCCs) of a given directed graph. The first pass is conducted on both the original graph (G) and its reverse (G_{rev}) to establish the topological ordering of vertices based on their finish times. Subsequently, these finish times are utilized to identify the SCCs. By traversing the original graph in the order of decreasing finish times obtained from the first pass, we explore the SCCs in the reverse graph, ensuring their identification.

The creation of the "Meta Graph" involves connecting these SCCs with edges by comparing the vertices in the original graph. This ensures that relationships between vertices within the SCCs are accurately represented. The process leverages the understanding that vertices within SCCs are mutually reachable, a property guaranteed during the second pass of DFS on the reverse graph.

The algorithm exhibits a time complexity of (O(m + n)), where (m) represents the number of edges and (n) denotes the number of components. The initial pass to establish topological ordering and conduct DFS takes (O(m + n)) time, scanning every unvisited vertex and launching recursive DFS calls. Similarly, the computation of the reverse graph occurs during pre-processing. Building the meta graph and adding edges between components are also (O(m + n)) operations, ensuring the overall efficiency of the algorithm.

Q4)

The `nodesAndClearanceLevel` method recursively selects nodes with the minimum cost until all nodes of a given clearance level are connected. Here's why the algorithm is correct:
1. Initialization: The algorithm initializes the `linkedEdge` map to store already linked nodes and sets the `isFirstConnection` flag to true initially.
2. Base Case: If `isFirstConnection` is true, the algorithm connects two nodes with the minimum weight edge and updates the `linkedEdge` map and `totalCost`. It then recursively calls `nodesAndClearanceLevel` to select nodes of the next clearance level.
3. Recursion: During recursion, the algorithm checks for nodes that can be connected to already linked nodes (from the `linkedEdge` map) with the minimum cost. It updates `linkedEdge`, `totalCost`, and removes selected nodes from the `aConnection` map.
4. Termination: The recursion stops when all nodes of a clearance level are connected or if no more nodes can be selected due to insufficient connections or higher costs.
5. Completeness: The algorithm guarantees completeness because it ensures that all nodes of a given clearance level are connected before proceeding to the next level. It exhaustively explores all possible connections and selects the minimum cost option at each step.

6. Optimality: Since the algorithm selects edges based on their weight in ascending order, it ensures that the selected connections collectively minimize the total cost of connecting all computers with different clearance levels.

Time Complexity Argument:
1. Input Reading: Reading the number of computers and connections takes constant time, O(1).
2. Input Processing: Storing node details and edge details in maps and lists takes linear time, O(n + m).
3. Sorting: Sorting edges based on weight takes O(m log m) time due to the use of Collections.sort.
4. Recursion: The `nodesAndClearanceLevel` method iterates through all edges once and recursively explores possible connections, making a maximum of m calls. Each call may remove a node from the `aConnection` map, resulting in at most n iterations. Thus, the time complexity of the recursion is O(m * n).

Overall, the time complexity of the algorithm is dominated by the sorting step and the recursion, making it O(m log m + m * n).