extremely sorry, forgot to attach the description of the codes with the file.

1(a).

The code performs a Depth-First Search (DFS) traversal on the graph to detect cycles. It initializes a `color` dictionary to track visited vertices during the DFS traversal. If a vertex is encountered that is already in the "grey" state, indicating it has been visited in the current traversal path, it means there is a cycle in the graph.

The `dfs()` function initiates the DFS traversal, while `dfs_visit()` performs the actual traversal, marking vertices as visited and exploring adjacent vertices recursively.

Finally, the code checks if a cycle has been detected (`has_cycle` flag). If so, it writes "IMPOSSIBLE" to the output file. Otherwise, it writes the path obtained from the DFS traversal.

A cycle indicates that all the courses cannot be completed in any certain order.

1(b).

The code constructs an adjacency list (`adjlis`) representing the prerequisites of each course and initializes an array `in_degree` to keep track of the in-degree of each course.

Next, it iterates over the input to populate the adjacency list and update the in-degree array based on the prerequisites.

It then defines a function `bfs_courseorder` to perform a Breadth-First Search (BFS) traversal to find the order in which courses can be taken, considering the prerequisites. It initializes a deque `Q` and an empty list `order`. The BFS traversal starts with courses having an in-degree of 0, **meaning they have no prerequisites**. During the traversal, it updates the in-degree of adjacent courses and adds them to the queue if their in-degree becomes 0.

After obtaining the course order, it checks if all courses have been included in the order. If not, it writes "IMPOSSIBLE" to the output file, **indicating that it's not possible to schedule all courses due to cyclic dependencies**. Otherwise, it writes the course order to the output file.

2.

A function `topologicalSort` to perform a topological sort on the given directed graph. It initializes an array `in_degree` to store the in-degree of each vertex. The function utilizes a priority queue (`heapq`) to select vertices with in-degree 0, indicating they can be added to the topological ordering. It iterates through the graph, updating in-degrees and adding vertices to the topological order as necessary.

If the count of visited vertices does not match the total number of vertices, indicating there's a cycle in the graph, the function returns "IMPOSSIBLE". Otherwise, it returns the topological order as a string.

3.

The code constructs two adjacency lists: `adjlis` representing the original graph and `adjlisTr` representing the transpose of the graph.

The script then defines a function `dfs_finishtime(graph)` to perform a Depth-First Search (DFS) traversal to determine the finish time of each vertex in the original graph. It initializes a `color` dictionary to track visited vertices and returns a list `path` containing vertices ordered by their finish time.

Another function `dfs(graph, finish)` is defined to perform DFS traversal on the transposed graph, using the finish times obtained earlier. It writes each strongly connected component (SCC) to the output file, separating them by newlines.

The main logic of the script involves performing DFS on the original graph to obtain the finish times of vertices, then reversing this list to get the correct order. Finally, DFS is performed on the transposed graph using the finish times to identify and output SCCs.