##Task-01

The code uses a depth-first search (DFS) algorithm to determine the order of courses based on prerequisites. It initializes a graph and tracks visited nodes and nodes in progress. The `dfs` function checks for cycles in the graph and updates the course order. The `course_order_dfs` function constructs the graph from prerequisites and performs DFS for each course. The script reads input from a file, computes the course order using DFS, and writes the result to an output file. If a cycle exists in the prerequisites, it returns "IMPOSSIBLE"; otherwise, it provides the course order.

##Task-02

This code implements a course order determination using breadth-first search (BFS). It constructs a graph and maintains the indegree of each node based on prerequisites. The algorithm initializes a queue with nodes having zero indegree, iterates through the queue, decrements the indegree of connected nodes, and adds nodes with zero indegree to the queue. The resulting order represents the sequence of courses. It reads input from a file, processes the course order using BFS, and writes the output to another file. If the resulting course order does not cover all courses, it returns "IMPOSSIBLE" due to a cycle in prerequisites.

##Task-03

This modified version aims to find the lexicographically smallest course order using a topological sort with BFS. It initializes the queue with nodes having zero indegree, and in each iteration, it sorts the queue lexicographically before picking the node. This ensures that the smallest node among those with zero indegree is chosen first. It then updates the indegrees accordingly and continues until the course order is determined. If the resulting course order doesn't include all courses, it returns "IMPOSSIBLE." The code reads input from a file, processes the course order, and writes the output to another file.

##Task-04

This code identifies strongly connected components in a directed graph using Kosaraju's algorithm. It consists of two depth-first search (DFS) functions - `dfs1` and `dfs2`.

1. `dfs1` performs a DFS traversal on the original graph, storing nodes in a stack based on finishing times.
2. The function `graph_func` initializes a stack by running `dfs1` on the entire graph, then reverses the graph to find the transposed graph.
3. It performs another DFS on the transposed graph (`dfs2`) using the nodes popped from the stack, thereby finding strongly connected components.

4. The script reads input from a file, constructs the graph, identifies strongly connected components using the `graph_func`, and writes the components to an output file, each line representing a strongly connected component.