Exercise 1: Cycle Detection in Directed Graphs

Implement a function to determine whether a directed graph contains at least one cycle and, if a cycle exists, print the vertices forming the cycle. Your solution must use a modified depth-first search (DFS) that keeps track of the recursion stack.

A directed graph contains a cycle if there exists a path from some vertex v back to itself. During DFS traversal, encountering an edge that points to a vertex currently in the recursion stack indicates a *back edge*, which confirms the presence of a cycle.

Maintain three states for each vertex:

- **UNVISITED (0)**: the vertex has not been explored,
- **VISITING (1)**: the vertex is part of the current DFS recursion stack,
- **VISITED (2)**: the vertex and all its descendants have been fully processed.

If DFS reaches a vertex in the VISITING state, a cycle has been found. To reconstruct the cycle, maintain a `parent[]` array. When a back edge $u \to v$ is discovered, follow the parent pointers from u until you return to v, thereby producing the sequence of vertices that form the cycle.

Use an adjacency list to represent the directed graph and a `state[]` array to track the state of each vertex. Your function should return a boolean value indicating whether a cycle exists, and if so, print the cycle.

Time Complexity:

$O(V + E)$, where V is the number of vertices and E is the number of edges.

Sample Test Case

Input: A directed graph with 4 vertices and edges: $0 \to 1$, $1 \to 2$, $2 \to 3$, $3 \to 1$
Output:
Cycle detected: true
Cycle: $1 \to 2 \to 3 \to 1$

Exercise 2: Bipartiteness Testing for Undirected Graphs

Determine whether a given undirected graph is bipartite. A graph is bipartite if its vertices can be partitioned into two disjoint sets such that every edge connects vertices from different sets.
A graph is bipartite if and only if it contains no odd-length cycles. The algorithmic approach involves attempting to 2-color the graph using BFS. If a valid 2-coloring exists, the graph is bipartite; otherwise, it is not.

Algorithm Steps

1. Initialize a `color[]` array with -1 for all vertices (indicating uncolored).
2. For each uncolored vertex (to handle disconnected components):
   (a) Start BFS from that vertex and assign it color 0.
   (b) For each neighbor:
      - If uncolored, assign the opposite color (1 if current is 0, or 0 if current is 1).

- If a neighbor already has the same color as the current vertex, the graph is not bipartite.

(c) Continue BFS until the component is fully processed.

3. If all components can be successfully 2-colored, the graph is bipartite.

Use BFS for the coloring process and handle disconnected graphs by iterating over all vertices. Use an integer array `color[]` with values $\{-1, 0, 1\}$, and return or print a clear statement about bipartiteness. As an optional extension, if the graph is bipartite, output the two vertex sets.

$$O(V + E), \text{ as each vertex and edge is processed at most once.}$$

Sample Test Case

Input: An undirected graph with 4 vertices and edges: $0-1$, $1-2$, $2-3$, $3-0$

Output: `The graph is bipartite`

Explanation: Sets $\{0, 2\}$ and $\{1, 3\}$ form a valid bipartition.

Exercise 3: Topological Sorting Using Indegrees

Implement Kahn's algorithm for topological sorting of a directed acyclic graph (DAG) using a queue and indegree computation. If the graph contains a cycle, report that no valid topological order exists.
A topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge $u \to v$, vertex u appears before v in the ordering. Topological sorting is only possible for DAGs.
The indegree of a vertex is the number of incoming edges. A vertex with indegree 0 has no prerequisites and can be processed immediately.

Algorithm Overview (Kahn's Algorithm)

1. Compute the indegree for all vertices by iterating through all edges.
2. Initialize a queue and enqueue all vertices with indegree 0.
3. While the queue is not empty:
   (a) Dequeue a vertex v and append it to the topological order.
   (b) For each outgoing edge $v \to w$:
      - Decrement the indegree of w.
      - If the indegree of w becomes 0, enqueue w.
4. After processing, if the number of vertices in the topological order equals the total number of vertices, output the order. Otherwise, report that the graph contains a cycle and no topological order exists.

Use an adjacency list representation and maintain an `indegree[]` array. Use a queue (FIFO structure) for processing vertices, output the topological order as a sequence of vertices, and detect and report cycles appropriately.

$$O(V + E), \text{ as we process each vertex and edge exactly once.}$$

Sample Test Case

Input: A directed graph with 6 vertices and edges: $5 \to 2$, $5 \to 0$, $4 \to 0$, $4 \to 1$, $2 \to 3$, $3 \to 1$

Output: `Topological order: 4 5 2 0 3 1`
(or another valid ordering)