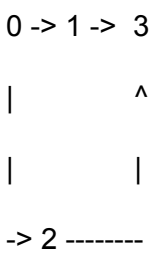
Algorithm for Exercise 1

1. **Input**:
    - A directed graph `graph`.
    - Two vertices: `source` and `target`.
2. **Initialize**:
    - Let `V` be the number of vertices in `graph`.
    - Create a `visited` array of size `V`, initialized to `false`.
    - Create an empty `queue` (FIFO) to manage BFS traversal.
    - Mark `source` as visited and enqueue it.
3. **BFS Traversal**:
    - While the `queue` is not empty:
        1. Dequeue the front vertex `v` from the `queue`.
        2. If `v` equals `target`, return `true` (a path exists).
        3. For each neighbor `w` of `v`:
            - If `w` is not visited:
                - Mark `w` as visited.
                - Enqueue `w` into the `queue`.
4. **Termination**:
    - If the `queue` is empty and the `target` has not been reached, return `false` (no path exists).

Example Walkthrough

Graph:



Input:

- `source = 0`, `target = 3`.

Execution:

1. Initialize:
    - `visited = [false, false, false, false]`.
    - `queue = []`.
    - Mark `0` as visited, enqueue it: `queue = [0]`.
2. BFS Traversal:
    - Dequeue `0`: Mark neighbors `1` and `2` as visited, enqueue them: `queue = [1, 2]`.
    - Dequeue `1`: Mark neighbor `3` as visited, enqueue it: `queue = [2, 3]`.
    - Dequeue `2`: No unvisited neighbors.
    - Dequeue `3`: Target found. Return `true`.

Algorithm for `hasCycle` Function

1. **Input**:
    - A directed graph `graph`.
2. **Initialize**:
    - Let `V` be the number of vertices in the graph.
    - Create a `visited` array of size `V`, initialized to `false`, to track whether a vertex has been visited.
    - Create an `onStack` array of size `V`, initialized to `false`, to track whether a vertex is part of the current DFS path.
3. **Outer Loop**:
    - For each vertex `v` in the graph:
        - If `v` is not visited:
            - Perform DFS starting from `v`.
4. **DFS Traversal**:
    - Use the reverse postorder (`DepthFirstOrder`) to traverse the vertices in a specific order.
    - For each vertex in reverse postorder:
        - Mark the vertex as visited (`visited[vertex] = true`).
        - Mark the vertex as part of the current DFS stack (`onStack[vertex] = true`).
5. **Cycle Detection**:
    - For each neighbor `adj` of the current vertex:
        - If `adj` is not visited:
            - Mark `adj` as visited and continue the DFS.
        - If `adj` is already in the current DFS stack (`onStack[adj] = true`):
            - A cycle is detected. Return `true`.
6. **Backtracking**:
    - After all neighbors of a vertex are explored, mark it as no longer part of the current DFS stack (`onStack[vertex] = false`).
7. **Termination**:
    - If no cycles are detected after exploring all vertices, return `false`.

Example Walkthrough

Graph:

Vertices: `0 -> 1 -> 2 -> 3 -> 4 -> 1` (cycle: `4 -> 1`).

Execution:

1. Initialize:
    - `visited = [false, false, false, false, false]`.
    - `onStack = [false, false, false, false, false]`.
2. Process Vertex `0`:
    - Mark `0` as visited and on the stack.
    - Process neighbor `1`.
3. Process Vertex `1`:
    - Mark `1` as visited and on the stack.
    - Process neighbor `2`.
4. Process Vertex `2`:
    - Mark `2` as visited and on the stack.
    - Process neighbor `3`.
5. Process Vertex `3`:
    - Mark `3` as visited and on the stack.
    - Process neighbor `4`.
6. Process Vertex `4`:
    - Mark `4` as visited and on the stack.
    - Process neighbor `1`.
    - Neighbor `1` is already on the stack. A cycle is detected. Return `true`.