Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.

◆ Introduction to Topological Sorting

Topological sorting is an ordering of vertices in a **Directed Acyclic Graph (DAG)** such that for every directed edge (u,v), **vertex u appears before vertex v in the ordering**. This is useful in scheduling tasks, dependency resolution, and ordering steps in a workflow.

◆ Concepts Used in the Program

1. **Graph Representation**
   - The graph is represented using an **adjacency list** implemented with a vector<vector<int>> adj.
   - Each vertex stores a list of vertices it directs edges to.
2. **Depth-First Search (DFS) Approach for Topological Sorting**
   - The function topologicalSortUtil uses **DFS** to visit nodes and push them onto a stack **after all their neighbors have been visited**.
   - The **stack ensures that dependencies are resolved** first before processing a node.
3. **Stack-based Ordering**
   - Once DFS traversal is complete, elements are popped from the stack to obtain the **topological ordering**.
4. **Graph Construction**
   - The function addEdge(v, w) adds a **directed edge** from v → w to the adjacency list.

◆ DFS-Based Topological Sorting

1. **Mark the current node as visited**.
2. **Recursively visit all its adjacent nodes**.
3. **Push the node onto the stack after visiting all its neighbors**.

The above program is based on **DFS (Depth-First Search) for Topological Sorting**.

◆ Why is it DFS-Based?

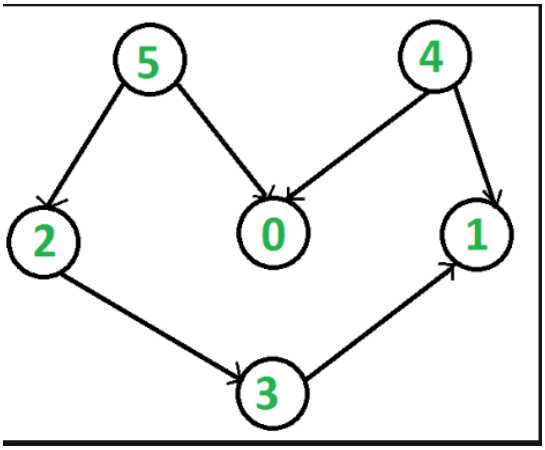
1. **Recursive Traversal**
   - The function topologicalSortUtil(int v, vector<bool>& visited, stack<int>& Stack) is a **DFS-based recursive function** that explores each node and its adjacent nodes **before pushing it onto the stack**.
2. **Stack-Based Ordering**
   - Once all adjacent nodes of a vertex v are visited, v is pushed onto the stack, ensuring that **dependencies are processed first**.
3. **Reverse Order Output**
   - The final **topological order** is obtained by popping elements from the stack, which maintains the correct dependency order.

Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.

◆ Difference from Source-Removal Algorithm (Kahn's Algorithm)

- **DFS-Based Approach (Current Program)**
  - Uses **recursion and a stack** to process dependencies.
  - Time Complexity: **O(V + E)**
  - Works well in **recursive problems and graph traversals**.
- **Source-Removal (Kahn's Algorithm)**
  - Uses **in-degree counting** to find nodes with no incoming edges.
  - Uses a **queue (BFS-based)** instead of recursion.
  - Also runs in **O(V + E)** but is better suited for **iterative approaches**.

📝 Dry Run (Step-by-Step Execution)



Given Graph with 6 vertices (0 to 5)
The edges are:
5 → 2
5 → 0
4 → 0
4 → 1
2 → 3
3 → 1
This forms the **Directed Acyclic Graph (DAG)**.

Graph Representation (Adjacency List)
0 → []
1 → []
2 → [3]
3 → [1]
4 → [0,1]
5 → [2,0]

Initialization
visited = {false, false, false, false, false, false}
Stack = {} (empty)

Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.

Step-by-Step Execution of topologicalSort()

We iterate over all vertices from i = 0 to i = 5, calling topologicalSortUtil() for unvisited nodes.

Iteration 1: i = 0
Already visited, skip.

Iteration 2: i = 1
Already visited, skip.

Iteration 3: i = 2
Not visited → Call topologicalSortUtil(2)
Inside topologicalSortUtil(2)
visited = {false, false, true, false, false, false}
neighbor = 3 (not visited) → Call topologicalSortUtil(3)
Inside topologicalSortUtil(3)
visited = {false, false, true, true, false, false}
neighbor = 1 (not visited) → Call topologicalSortUtil(1)
Inside topologicalSortUtil(1)
visited = {false, true, true, true, false, false}
No outgoing edges → Push 1 to Stack
Stack = {1} → Return to topologicalSortUtil(3)
Back to topologicalSortUtil(3)
Push 3 to Stack
Stack = {3, 1} → Return to topologicalSortUtil(2)
Back to topologicalSortUtil(2)
Push 2 to Stack
Stack = {2, 3, 1} → Return to topologicalSort()

Iteration 4: i = 3
Already visited, skip.

Iteration 5: i = 4
Not visited → Call topologicalSortUtil(4)
Inside topologicalSortUtil(4)
visited = {false, true, true, true, true, false}
neighbor = 0 (not visited) → Call topologicalSortUtil(0)
Inside topologicalSortUtil(0)
visited = {true, true, true, true, true, false}
No outgoing edges → Push 0 to Stack
Stack = {0, 2, 3, 1} → Return to topologicalSortUtil(4)
Back to topologicalSortUtil(4)
neighbor = 1 (already visited) → Skip
Push 4 to Stack
Stack = {4, 0, 2, 3, 1} → Return to topologicalSort()

Iteration 6: i = 5
Not visited → Call topologicalSortUtil(5)
Inside topologicalSortUtil(5)
visited = {true, true, true, true, true, true}
neighbor = 2 (already visited) → Skip
neighbor = 0 (already visited) → Skip
Push 5 to Stack
Stack = {5, 4, 0, 2, 3, 1} → Return to topologicalSort()

Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.

Final Output

Thus, the Topological Sort of the given graph is:
5 → 4 → 0 → 2→ 3 → 1

📌 **Time Complexity Analysis**

- **DFS Traversal:** O(V+E), where V is the number of vertices and E is the number of edges.
- **Pushing into stack:** O(V).
- **Overall Complexity: O(V+E)**.

✅ **Key Takeaways**

✅ **DFS-based Topological Sorting** works by **visiting a node's dependencies first** before the node itself.

✅ **Stack-based ordering** ensures correct dependency resolution.

✅ **Works only for DAGs**, as cycles make topological sorting impossible.

✅ **Time Complexity: O(V+E)O(V + E)O(V+E)**, making it efficient for large graphs.

Source code:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

class Graph {
   int V;
   vector<vector<int>> adj;
   void topologicalSortUtil(int v, vector<bool>& visited, stack<int>& Stack);

public:
   Graph(int V) : V(V), adj(V) {} // Constructor using member initializer list
   void addEdge(int v, int w) { adj[v].push_back(w); }
   void topologicalSort();
};

void Graph::topologicalSortUtil(int v, vector<bool>& visited, stack<int>& Stack) {
   visited[v] = true;
   for (int neighbor : adj[v]) {
      if (!visited[neighbor]) {
         topologicalSortUtil(neighbor, visited, Stack);
      }
   }
   Stack.push(v);
}

void Graph::topologicalSort() {
   stack<int> Stack;
   vector<bool> visited(V, false);
   for (int i = 0; i < V; i++) {
      if (!visited[i]) {
         topologicalSortUtil(i, visited, Stack);
      }
   }
   cout << "Topological sorting: ";
   while (!Stack.empty()) {
      cout << Stack.top() << " ";
```

```
        Stack.pop();
    }
    cout << endl;
}

int main() {
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of the given graph:\n";
    g.topologicalSort();
    return 0;
}
```

Output:

Following is a Topological Sort of the given graph:
Topological sorting: 5 4 2 3 1 0

=== Code Execution Successful ===