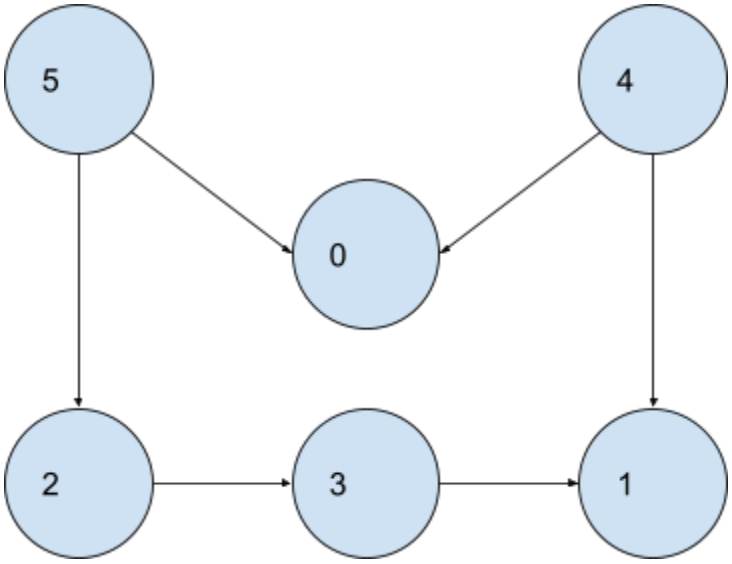
MEDIUM

Kahn's Algorithm | Topological Sort Algorithm

Intuition



Definition : Linear Ordering of vertices such that if there is an edge between U and V before V in that ordering.

DAG - Directed Acyclic Graph , directed graph that doesn't have any cycle

Eg.

5 -> 0
4 -> 0
5 -> 2
2 -> 3
3 -> 1
4 -> 1

5 appears before 0
4 appears before 0
5 appears before 2
2 appears before 3
3 appears before 1
4 appears before 1

It gives us two orders possible

Order 1 : 5 4 2 3 1 0

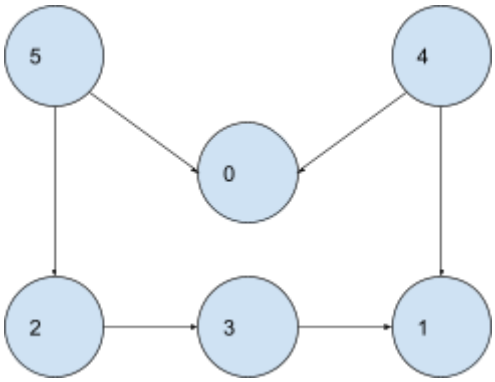
Order 2 : 4 5 2 3 1 0

Why only in DAG ?

If we take an undirected graph then 1-2 then 2-1 is also there which is not possible hence only directed graph.

Now if we have a directed graph with a cycle then also a condition arises.
Eg. path 1-2-3-1
1 before 2
2 before 3
3 before 1 [not possible]

Eg.



Visited = [1 1 1 1 1 0]

Adjacency List :

0 - [ ]
1 - [ ]
2 - [3]
3 - [1]
4 - [0, 1]
5 - [0, 2]

Stack = [0, 1, 3, 2, 4, 5]

DFS(0)
DFS(1)

DFS(2) - DFS(3) - DFS(1) : VISITED GO BACK
DFS(3) : VISITED
DFS(4) - DFS(0) : VISITED DFS(1) : VISITED RETURN
DFS(5) - DFS(0) : VISITED DFS(2) : VISITED RETURN

Traverse to pop the stack :

Returns // 5 4 2 3 1 0

Approach

topologicalsort( ) :

- Declare :
  - Visited array having n elements and all initialized with 0
  - Creating an initially empty stack
  - Initially empty result vector
- Traversing through all of the component nodes
  - Check if the node is unvisited :
    - Call for the dfs as dfs(node,adj,visited,stack)
- Traverse for all the elements of stack :
  - Insert the top of stack into result vector
  - Pop the stack
- Return the result vector

DFS( ) :

- Mark the source node as visited
- Traverse through the adjacent nodes of the source node :
  - If the given node is unvisited :
    - Make a dfs call as dfs(adjacentnode, adj, visited, stack)
- When DFS call is completed insert the node into the stack

Function Code

```
void dfs(int node,vector<int> adj[],vector<int> &visited,stack<int> &s)
      {
          // mark the source node as visited
          visited[node] = 1;

          // run through the adjacent elements of the node
          for(int i: adj[node])
          {
```

```
            // checking if the adjacent node is unvisited
            if(!visited[i])
            {
                // calling for the dfs of the adjacent node
                dfs(i,adj,visited,s);
            }
        }
        // pushing the given node to the stack
        s.push(node);

    }
    //Function to return list containing vertices in Topological order.
    vector<int> topoSort(int n, vector<int> adj[])
    {
        // Declare
        // visited vector having all elements as 0
        vector<int> visited(n,0);
        // creating a stack of elements
        stack<int> s;
        // creating a vector that will contain the topological order
        vector<int> topological_order;

        // traversing through all of the component nodes
        for(int i=0;i<n;i++)
        {
            // checking if the node is unvisited
            if(!visited[i])
            {
                // calling for the dfs of the unvisited node
                dfs(i,adj,visited,s);
            }
        }
        // traversing through the stack of the nodes
        while(!s.empty())
        {
            // extracting the top of the node
            int node = s.top();
            // popping the stack
            s.pop();
            // inserting the stack top element into the topological sort
vector
            topological_order.push_back(node);
        }
```

```
        // return the topological order
        return topological_order;

    }
```

Time Complexity

O(V+E)