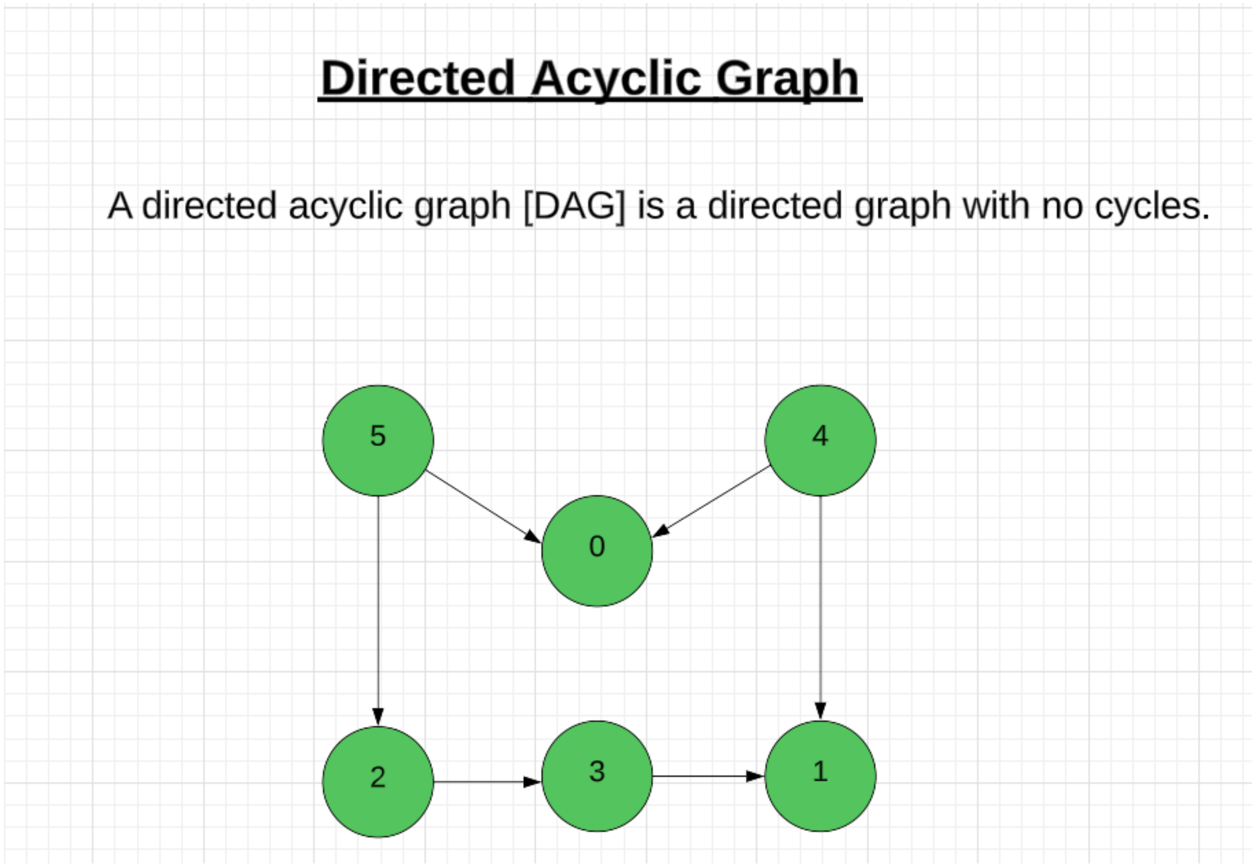
Topological Sorting Algorithm

What is Topological Sorting ?

1. Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge from ‘u’ to ‘v’, vertex u comes before v in that ordering.
2. A topological sort can only be performed on a directed acyclic graph (DAG), a Directed Acyclic Graph is a directed graph with no cycles.

Example:


Directed Acyclic Graph
A directed acyclic graph [DAG] is a directed graph with no cycles.
5
4
0
2
3
1

There can be more than one topological sorting for a given graph.

For the above graph,

1. One possible topological sorting can be: 5, 4, 2, 3, 1, 0.

a. **Explanation:** *In topological sorting is a linear ordering of vertices such that if there is an directed edge between “u” to “v” (”u” → “v”), “u” comes before “v” in that ordering.*

    i. According to edge **5 -> 0,** node 5 must appear before node 0 in the ordering.

    ii. According to edge **4 -> 0,** node 4 must appear before node 0 in the ordering.

    iii. According to edge **5 -> 2,** node 5 must appear before node 2 in the ordering.

    iv. According to edge **2 -> 3,** node 2 must appear before node 3 in the ordering.

    v. According to edge **3 -> 1,** node 3 must appear before node 1 in the ordering.

    vi. According to edge **4 -> 1,** node 4 must appear before node 1 in the ordering.

2. Another possible topological sorting can be: 4, 5, 2, 3, 1, 0.

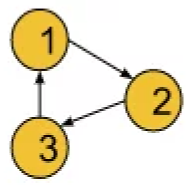
Now, let’s understand Why topological sort only exists in DAG:

1. **Case 1 (If the edges are undirected)**:

    a. If there is an undirected edge between node u and v, it signifies that there is an edge from node u to v(u -> v) as well as there is an edge from node v to u(v -> u). But according to the definition of topological sorting, it is practically impossible to write such ordering where ***u appears before v*** and ***v appears before u*** simultaneously. So, it is only possible for directed edges.

2. **Case 2(If the directed graph contains a cycle):**

    a. The following directed graph contains a cycle:

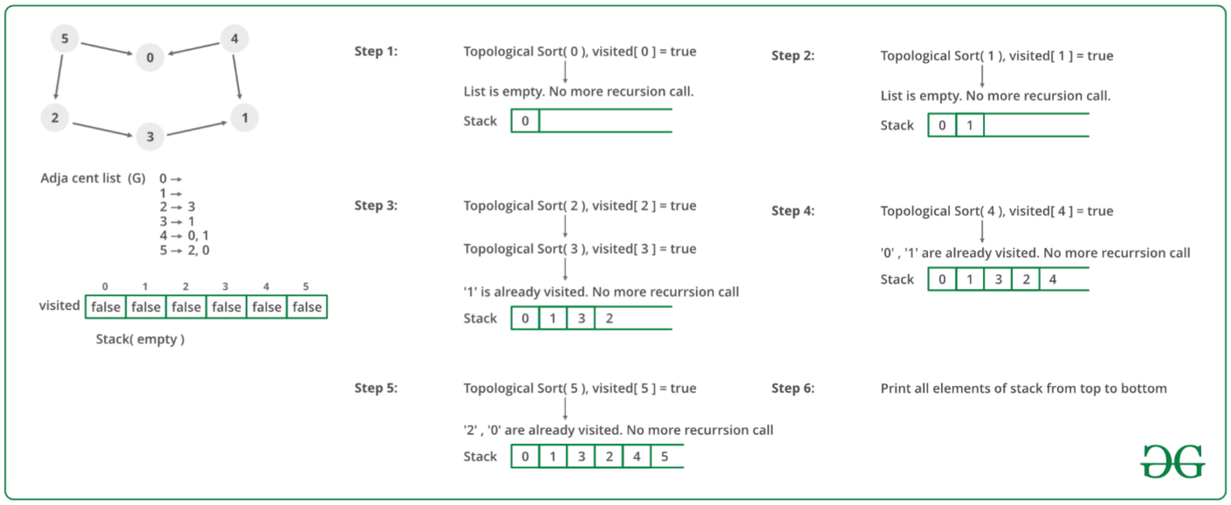


    b. If we try to get topological sorting of this cyclic graph, for edge 1->2, node 1 must appear before 2, for edge 2->3, node 2 must appear before 3, and for edge 3->1, node 3 must appear before 1 in the linear ordering. But such ordering is not possible as there exists a cyclic dependency in the graph. Thereby, topological sorting is only possible for a directed acyclic graph.

Algorithm for Topological Sorting Using Depth First Search (DFS):

We can modify **DFS** to find the Topological Sorting of a graph.

1. **Traverse all components of the graph:**
    a. This step is important to make sure we visit all vertices in the graph. If the graph is disconnected, we need to start DFS from each unvisited vertex.
2. **Initialize visited array and stack:**
    a. The visited array keeps track of which vertices have already been visited, while the stack is used to stores the vertices the topological order. Both data structures are initialized before starting the DFS traversal.
3. **DFS traversal:**
    a. For each unvisited vertex, we perform a DFS traversal to visit all its adjacent vertices. During the traversal, we mark the current vertex as visited, then recursively call DFS on each of its unvisited neighbors.
4. **Push the current vertex onto the stack:**
    a. After all the adjacent vertices of a particular node have been visited, we push the current vertex onto the stack. This is because we only add a vertex to the stack once all its neighbors have been visited, ensuring that the vertices are in topological order in the stack.
5. **Return the stack:**
    a. After all the vertices have been visited, the stack contains the vertices in topological order.



Adja cent list (G) 0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0
visited false false false false false false
Stack(empty)
Step 1: Topological Sort(0), visited[0] = true
List is empty. No more recursion call.
Stack 0
Step 2: Topological Sort(1), visited[1] = true
List is empty. No more recursion call.
Stack 0 1
Step 3: Topological Sort(2), visited[2] = true
Topological Sort(3), visited[3] = true
'1' is already visited. No more recurrsion call
Stack 0 1 3 2
Step 4: Topological Sort(4), visited[4] = true
'0' , '1' are already visited. No more recurrsion call
Stack 0 1 3 2 4
Step 5: Topological Sort(5), visited[5] = true
'2' , '0' are already visited. No more recurrsion call
Stack 0 1 3 2 4 5
Step 6: Print all elements of stack from top to bottom

Topological Sorting Using Breadth First Search (BFS) (AKA Kahn’s Algorithm):

1. Topological sorting using BFS is commonly known as Kahn's algorithm, named after its inventor, Arthur B. Kahn. It is based on the observation that if a vertex has no incoming edges, then it can be included in the topological sorting as the first vertex.
    a. In a directed acyclic graph (DAG), if a vertex has no incoming edges, it means that there are no edges that point to that vertex. This implies that the vertex does not depend on any other vertex in the graph, and can therefore be included as the first vertex in any valid topological sorting of the DAG.
    b. To see why this is true, consider the definition of a topological sorting: it is a linear ordering of the vertices such that for every directed edge (u, v), vertex u comes before vertex v in the ordering. If a vertex has no incoming edges, then there is no vertex that must come before it in the ordering, and so it can be placed at the beginning of the ordering.
    c. Kahn's algorithm takes advantage of this observation by starting with the vertices that have no incoming edges, adding them to the topological order, and then iteratively removing them along with their outgoing edges from the graph. By continuing this process until all vertices have been included in the order, we obtain a valid topological sorting of the DAG.

The algorithm works as follows:

1. Initialize a list to store the topological order and a queue to store vertices with no incoming edges.
2. Compute the in-degree (i.e., number of incoming edges) of each vertex in the graph and add all vertices with in-degree 0 to the queue.
3. While the queue is not empty, remove a vertex from the queue and add it to the topological order list.
4. For each of the vertex's neighbors, decrement their in-degree by 1. If a neighbor's in-degree becomes 0, add it to the queue.
5. Repeat steps 3-4 until the queue is empty.
6. If the topological order list contains all vertices in the graph, return it as the topological order. Otherwise, the graph contains a cycle and is not a DAG.