**Topological Sorting**

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



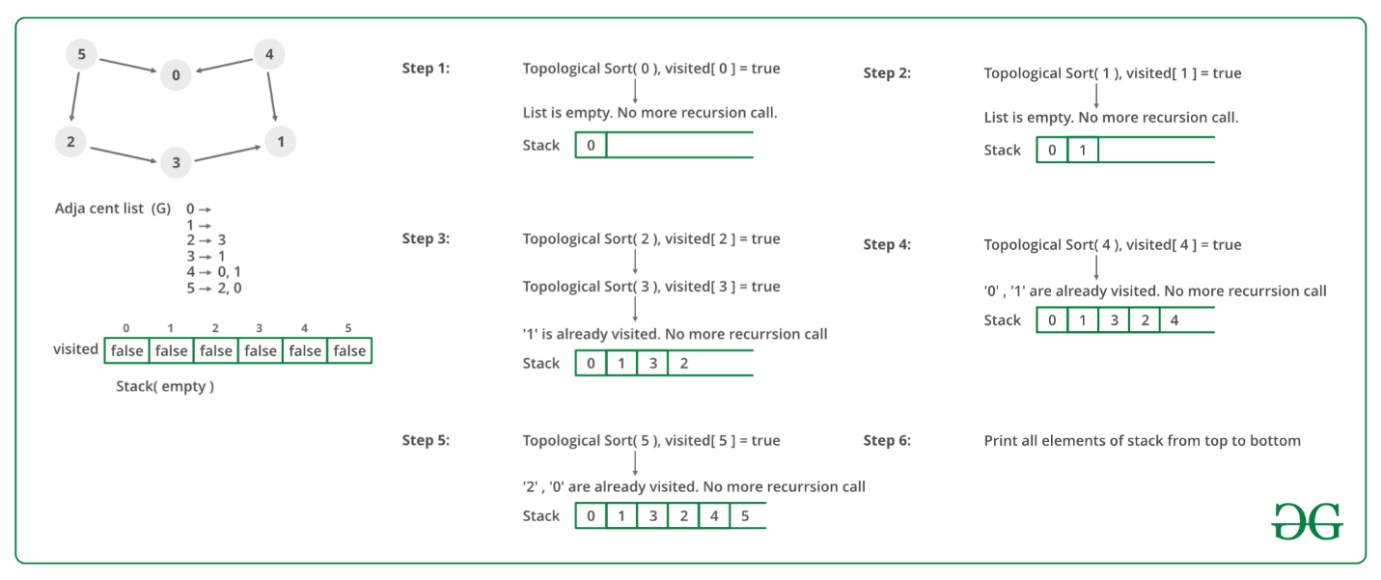
***Topological Sorting vs Depth First Traversal (DFS)***:

In [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting.

**Algorithm to find Topological Sorting:**

We recommend to first see the implementation of [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/). We can modify [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)to find Topological Sorting of a graph. In [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.

Below image is an illustration of the above approach:



**Complexity Analysis:**

* **Time Complexity:** O(V+E).   
  The above algorithm is simply DFS with an extra stack. So time complexity is the same as DFS which is.
* **Auxiliary space:** O(V).   
  The extra space is needed for the stack.

**Note:** Here, we can also use vector instead of the stack. If the vector is used then print the elements in reverse order to get the topological sorting.

**Applications:**   
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers [[2](http://en.wikipedia.org/wiki/Topological_sorting)].

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/topologicalsort/TopologicalSort.java>

**All Topological Sorts of a Directed Acyclic Graph**

Topological sorting for **D**irected **A**cyclic **G**raph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.  
Given a DAG, print all topological sorts of the graph.

For example, consider the below graph.



All topological sorts of the given graph are:  
4 5 0 2 3 1   
4 5 2 0 3 1   
4 5 2 3 0 1   
4 5 2 3 1 0   
5 2 3 4 0 1   
5 2 3 4 1 0   
5 2 4 0 3 1   
5 2 4 3 0 1   
5 2 4 3 1 0   
5 4 0 2 3 1   
5 4 2 0 3 1   
5 4 2 3 0 1   
5 4 2 3 1 0

In a Directed acyclic graph many a times we can have vertices which are unrelated to each other because of which we can order them in many ways. These various topological sorting is important in many cases, for example if some relative weight is also available between the vertices, which is to minimize then we need to take care of relative ordering as well as their relative weight, which creates the need of checking through all possible topological ordering.   
We can go through all possible ordering via backtracking , the algorithm step are as follows : 

1. Initialize all vertices as unvisited.
2. Now choose vertex which is unvisited and has zero indegree and decrease indegree of all those vertices by 1 (corresponding to removing edges) now add this vertex to result and call the recursive function again and backtrack.
3. After returning from function reset values of visited, result and indegree for enumeration of other possibilities.

Below is implementation of above steps.

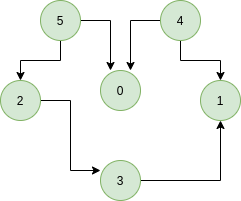
<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/topologicalsort/AllTopologicalSortOfDirectedAcyclicGraph.java>

**Kahn’s algorithm for Topological Sorting**

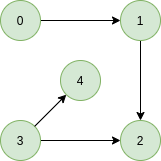
Topological sorting for **D**irected **A**cyclic **G**raph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.  
For example, a topological sorting of the following graph is “5 4 2 3 1 0?. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 0 3 1″. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).



Let’s look at few examples with proper explanation,   
**Example:**



***Output:****5 4 2 3 1 0****Explanation:****The topological sorting of a DAG is done in a order such that for every directed edge uv, vertex u comes before v in the ordering. 5 has no incoming edge. 4 has no incoming edge, 2 and 0 have incoming edge from 4 and 5 and 1 is placed at last.****Input:***

**

***Output:****0 3 4 1 2****Explanation:****0 and 3 have no incoming edge, 4 and 1 has incoming edge from 0 and 3. 2 is placed at last.*

A [DFS based solution to find a topological sort](https://www.geeksforgeeks.org/topological-sorting/) has already been discussed.  
**Solution:**In this article we will see another way to find the linear ordering of vertices in a directed acyclic graph (DAG). The approach is based on the below fact:  
**A DAG G has at least one vertex with in-degree 0 and one vertex with out-degree 0**.   
**Proof:** There’s a simple proof to the above fact is that a DAG does not contain a cycle which means that all paths will be of finite length. Now let S be the longest path from u(source) to v(destination). Since S is the longest path there can be no incoming edge to u and no outgoing edge from v, if this situation had occurred then S would not have been the longest path   
=> indegree(u) = 0 and outdegree(v) = 0  
**Algorithm:** Steps involved in finding the topological ordering of a DAG:   
**Step-1:** Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.  
**Step-2:**Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)  
**Step-3:** Remove a vertex from the queue (Dequeue operation) and then. 

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighbouring nodes.
3. If in-degree of a neighbouring nodes is reduced to zero, then add it to the queue.

**Step 4:** Repeat Step 3 until the queue is empty.  
**Step 5:**If count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph.  
**How to find in-degree of each node?**   
There are 2 ways to calculate in-degree of every vertex: 

1. Take an in-degree array which will keep track of   
   Traverse the array of edges and simply increase the counter of the destination node by 1.

for each node in Nodes

indegree[node] = 0;

for each edge(src, dest) in Edges

indegree[dest]++

1. Time Complexity: O(V+E)
2. Traverse the list for every node and then increment the in-degree of all the nodes connected to it by 1.

for each node in Nodes

If (list[node].size()!=0) then

for each dest in list

indegree[dest]++;

1. Time Complexity: The outer for loop will be executed V number of times and the inner for loop will be executed E number of times, Thus overall time complexity is O(V+E).  
   The overall time complexity of the algorithm is O(V+E)

**Longest path between any pair of vertices**

We are given a map of cities connected with each other via cable lines such that there is no cycle between any two cities. We need to find the maximum length of cable between any two cities for given city map.

Input : n = 6

1 2 3 // Cable length from 1 to 2 (or 2 to 1) is 3

2 3 4

2 6 2

6 4 6

6 5 5

Output: maximum length of cable = 12

**Method 1 (Simple DFS)**   
We create undirected graph for given city map and do [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) from every city to find maximum length of cable. While traversing, we look for total cable length to reach the current city and if it’s adjacent city is not visited then call [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) for it but if all adjacent cities are visited for current node, then update the value of max\_length if previous value of max\_length is less than current value of total cable length.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/topologicalsort/LongestPathBetweenAnyPairOfVertices.java>

**Longest Path in a Directed Acyclic Graph**

Given a Weighted **D**irected **A**cyclic **G**raph (DAG) and a source vertex s in it, find the longest distances from s to all other vertices in the given graph.  
The longest path problem for a general graph is not as easy as the shortest path problem because the longest path problem doesn’t have [optimal substructure property](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/). In fact, [the Longest Path problem is NP-Hard for a general graph](http://en.wikipedia.org/wiki/Longest_path_problem). However, the longest path problem has a linear time solution for directed acyclic graphs. The idea is similar to [linear time solution for shortest path in a directed acyclic graph.](https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/), we use [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/).   
We initialize distances to all vertices as minus infinite and distance to source as 0, then we find a [topological sorting](https://www.geeksforgeeks.org/topological-sorting/) of the graph. Topological Sorting of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a) ). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.  
Following figure shows step by step process of finding longest paths.

LongestPath

Following is complete algorithm for finding longest distances.   
**1)** Initialize dist[] = {NINF, NINF, ….} and dist[s] = 0 where s is the source vertex. Here NINF means negative infinite.   
**2)** Create a topological order of all vertices.   
**3)** Do following for every vertex u in topological order.   
………..Do following for every adjacent vertex v of u   
………………if (dist[v] < dist[u] + weight(u, v))   
………………………dist[v] = dist[u] + weight(u, v)

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/topologicalsort/LongestPathInDirectedAcyclicGraph.java>