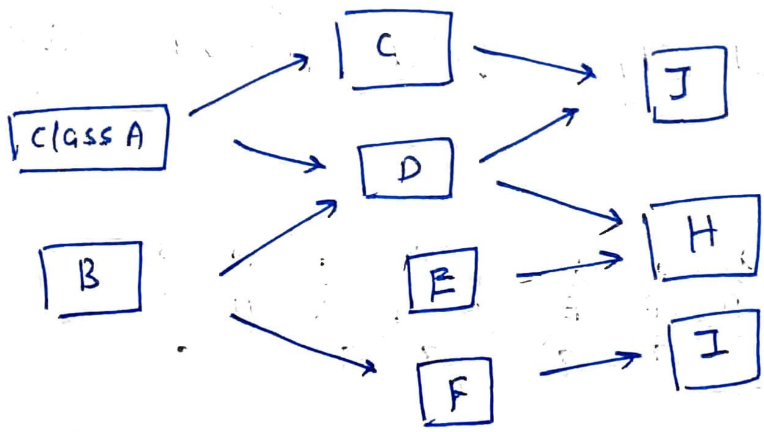
Topological Sort

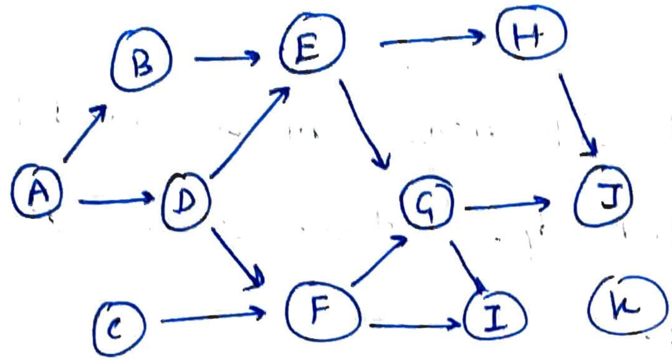
→ Many real world situations can be modelled as a graph with directed edges where some events must occur before others.

eg.
- → School class prerequisites
- → Program dependencies
- → Event scheduling
- → Assembly instructions etc.

→ suppose a university student wants to class H, then we must take A, B, D and E as prerequisites. In this sense there is an ordering on the nodes of the graph.



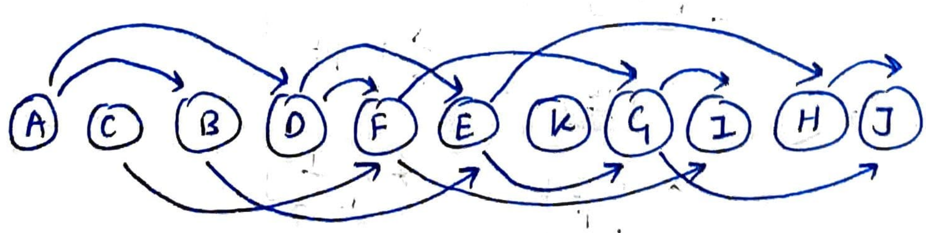
→ Another example where an ordering of nodes of the graph matters is for program build dependencies. A program cannot be built unless it's dependencies are built first.



Let us say we want to build J, then what should be the order in which we built dependencies.

→ Thus, a topological ordering is an ordering of the nodes in a directed graph where for each directed edge from node A to node B, node A appears before node B in the ordering.

→ If we put nodes in a straight line, then all the directed edges point towards right



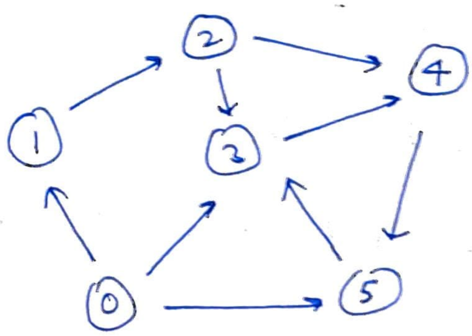
All edges point rightward; thus, if we start building our application in this order, there will be no problem.

→ Topological sort algorithm can find a topological ordering in O(V+E) time.

Topological orderings are not unique.

→ Directed acyclic graphs (DAG)

Not every graph can have a topological ordering. A graph which contains a cycle cannot have a valid ordering.



Here, 3 → 4 → 5 is a directed cycle, thus topological ordering is not possible.

→ The only graphs that have valid topological ordering are Directed Acyclic Graphs (DAG). These are graphs with directed edges and no cycles.

(Q) How do we verify that the graph does not contain a directed cycle?

→ One method is to use Tarjan's strongly connected component algorithm which can be used to find these cycles.

→ By definition, all rooted trees have a topological ordering since they do not contain any cycles.

→ For trees, we just start picking the leaf nodes iteratively until root node is leaf node. This procedure gives the topological ordering.

Topological sort Algorithm

(i) Pick an unvisited node.

(ii) Beginning with selected node, do a DFS exploring only unvisited nodes.

(iii) On the recursive callback of DFS, add the current node to the topological ordering in reverse order.

Topsort pseudocode

Assumption : graph is stored as adjacency list.

```
function topsort (graph):

    N = graph. numberOfNodes():
    V = [false, false, .... ]   # length N
    ordering = [0, .... 0]   # length N
    i = N-1   # Index for ordering array.

    for (at = 0; at < N; at++):
        if v[at] == false :
            visitedNodes = []
            dfs (at, v, visitedNodes, graph)
            for nodeID in visitedNodes :
                ordering [i] = nodeID
                i = i-1

    return ordering.

function dfs (at, v, visitedNodes, graph):
    v[at] = true.
    edges = graph. getEdgesOutFromNode (at)
    for edge in edges :
        if v[edge.to] == false
            dfs (edge.to, v, visitedNodes, graph)
    visitedNodes. add(at).
```

Topological sort: kahn's Algorithm

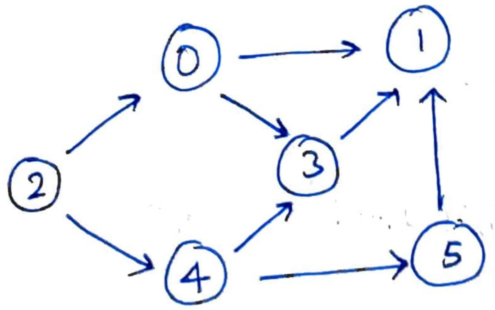
↳ An intuitive topological sort algorithm.

kahn's algorithm

↳ The intuition behind kahn's algorithm is to repeatedly remove nodes without any dependencies from the graph and add them to topological ordering.

→ As nodes without dependencies (and their outgoing edges) are removed from the graph, new nodes without dependencies should become free.

→ We repeat removing nodes without dependencies from the graph until all nodes are processed, or a cycle is discovered.



select the node with no dependencies and remove it from graph. Repeat.

2 0 4 3 5 1.

```
# 'g' is a directed acyclic graph represented as an adjacency list.

function FindTopologicalOrdering(g):

    n = g.size()
    in_degree = [0,0, .... 0]      # size n.
    for( i=0 ; i<n ; i++) :
        for (to in g[i]):
            in_degree[to] = in_degree[to] + 1


# 'q' always contains the set of nodes with no incoming edges
                                    # empty integer queue data structure
    q = ....
    for( i = 0 ; i < n ; i++):
        if ( in_degree[i] == 0):
            q.enqueue (i)


    index = 0
    order = [0,0, .... 0]    # size n
    while (! q.isEmpty()):
        at = q.dequeue ()
        order [index ++] = at
        for ( to in g[at]):
            in_degree [to] = indegree[to] - 1
            if in_degree [to] == 0 :
                q.enqueue (to)
        if index != n
            return null        # graph contains a cycle.
    return order.
```