Topological Sort

The goal of a topological sort is given a list of items with dependencies, (ie. item 5 must be completed before item 3, etc.) to produce an ordering of the items that satisfies the given constraints. In order for the problem to be solvable, there can not be a cyclic set of constraints. (We can't have that item 5 must be completed before item 3, item 3 must be completed before item 7, and item 7 must be completed before item 5, since that would be an impossible set of constraints to satisfy.)

We can model a situation like this using a directed acyclic graph. Given a set of items and constraints, we create the corresponding graph as follows:

1) Each item corresponds to a vertex in the graph.
2) For each constraint where item a must finish before item b, place a directed edge in the graph starting from the vertex for item a to the vertex for item b.

This graph is directed because each edge specifically starts from one vertex and goes to another. Given the fact that the constraints must be acyclic, the resulting graph will be as well.

Here is a simple situation:

A → B (Imagine A standing for waking up,
| | B standing for taking a shower,
V V C standing for eating breakfast, and
C → D D leaving for work.)

Here a topological sort would label A with 1, B and C with 2 and 3, and D with 4.

Let's consider the following subset of CS classes and a list of prerequisites:

CS classes: COP 3223, COP 3502, COP 3330, COT 3100, COP 3503, CDA 3103, COT 3960 (Foundation Exam), COP 3402, and COT 4210.

Here are a set of prerequisites:

COP 3223 must be taken before COP 3330 and COP 3502.

COP 3330 must be taken before COP 3503.

COP 3502 must be taken before COT 3960, COP 3503, CDA 3103.

COT 3100 must be taken before COT 3960.

COT 3960 must be taken before COP 3402 and COT 4210.

COP 3503 must be taken before COT 4210.

A goal of a topological sort then is to find an ordering of these classes that you can take.

Iterative Algorithm

This algorithm will build one valid topological sort given the appropriate constraints, if at least one exists, and otherwise can detect when no topological sort exists.

Notice that we can't perform an event if there are direct "pre-requisite" events that haven't yet been completed. Thus, the very first event must be one that has no pre-requisite events. In terms of the structure of a graph, the in-degree of a vertex is equal to the number of pre-requisite events, for the purposes of this problem. The in degree of a vertex is the number of directed edges coming into that vertex. So, the idea for the algorithm is as follows:

1) Calculate the in degree of each vertex, looping through each edge (Run time = O(E) where E is the # of edges in the graph). Store these in an array.

2) Maintain a queue of all vertices with current in degree 0. Thus, create a queue, and add in it all of the vertices that already have in degree 0 via the array created in step 1. (Run time = O(V), where V is the # of vertices in the graph.)

3) Create a third array (res) in which the topological sort will be stored (indexed 0 to V-1).

4) for (int i=0; i<V; i++) // Run time = O(E), since in step C, each edge
// in the graph is potentially considered.

a) If the queue is empty, return that no top sort exists.

b) Dequeue the next item, t, from the queue and store it as the item in index i of the result.

c) For each edge leaving vertex t, subtract 1 from the in degree of where the edge goes. After you subtract 1 from its in degree, if its in degree is 0, add it to the queue.

5) Return the array storing the top sort.

Total Run Time = O(V+E)

Recursive Algorithm - Edit to DFS

We can also utilize a DFS in our algorithm to determine a topological sort. In essence, the idea is as follows:

When you do a DFS on a directed acyclic graph, eventually you will reach a node with no outgoing edges. Why?

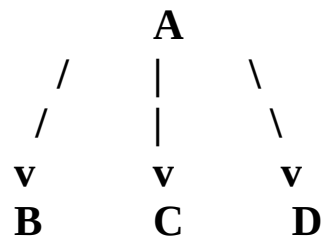
Because if this situation never occurred, you eventually hit a cycle since the number of nodes is finite.

This node that you reach in a DFS with no outgoing edges is "safe" to place at the end of the topological sort.

One could simply then rerun the whole DFS on the same graph n times over, where n is the original number of vertices in the graph. (The running time of this would potentially be $O(V^2)$. Can you determine why?)

But, we can do better than that. Rather, what we find is that as we run through our DFS, when we exhaust all of the edges to explore from a particular vertex, if we have added each of the vertices "below" it into our topological sort, it is safe then to add this one in.

Consider the following situation:



When we explore from A in a DFS, if we have completed marking B, C, and D, as well as everything we can explore from those vertices and added all of those vertices (in backwards order) into our topological sort, it will then be safe to add A into our topological sort. Thus, it is safe to add a vertex into our topological sort when we no longer have any more edges to explore from it.

So, basically, all you have to do is run a DFS, with the added component that at the end of the recursive function, go ahead and add the node the DFS was called with to the end of the topological sort. (An example was done in class.)
Here a step-by-step description of the algorithm:

1) Do a DFS on an arbitrary node.
2) The "last" node you reach before having to backtrack, (ie. a node that has no adjacent nodes) can safely be put as the last item in the topological sort. In the case above, we can safely put our shoes on last, since there is no item that requires shoes to be put on before it.
3) Continue with your DFS, placing each "dead end" node into the topological sort in backwards order.
4) Repeat step 1, picking an unvisited node, if one exists.

Here is some pseudocode for the algorithm:

```
top_sort(Adjacency Matrix adj, Array ts) {

    n = adj.last
    k = n  // assume k is global
    for i=1 to n
       visit[i] = false
    for i=1 to n
       if (!visit[i])
           top_sort_recurs(adj, i, ts)
}

top_sort_recurs(Adjacency Matrix adj, Vertex start, Array ts){
    visit[start] = true
    trav = adj[start] // trav is pointing to a linked list
    while (trav != null) {
        v = trav.ver
        if (!visit[v])
            top_sort_recurs(adj, v, ts);
        trav = trav.next
    }
    ts[k] = start, k=k-1 }
```

Another Topological Sort Example

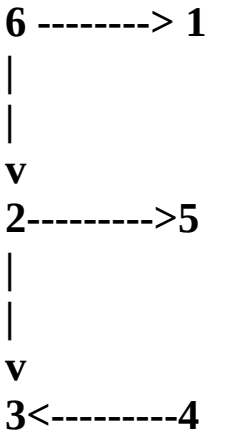
Consider the following items of clothing to wear:

Shirt	Slacks	Shoes	Socks	Belt	Undergarments
1	2	3	4	5	6

There isn't exactly one order to put these items on, but we must adhere to certain restrictions

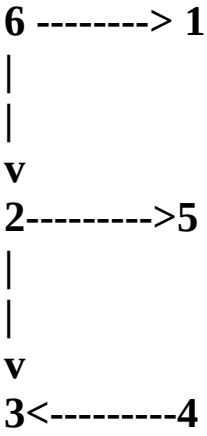
Socks must be put on before Shoes
Undergarments must be put on before Slacks and Shirt
Slacks must be put on before Belt
Slacks must be put on before Shoes

Using this information we can create this directed acyclic (dag) graph:



This is created by placing a directed edge from each item that must come before another item to that other item.

Let's trace through the pseudocode with the example graph:



When we call top_sort_recurs(adj, 1, ts), we can't get anywhere at all, and will simply end up placing 1 at the end of our topological sort.

Next, we will call top_sort_recurse(adj, 2, ts). We will then proceed to make that same call on vertex 3. Here we get stuck, and will then place vertex 3 at the end of the remaining slots:

				3	1
1	2	3	4	5	6

Now, we backtrack to vertex 2, and see that we have another node adjacent to it, node 5. Here we get stuck and place that right before node 3:

			5	3	1
1	2	3	4	5	6

Finally, we get back to node 2 with no more adjacent nodes, so it goes in our list:

		2	5	3	1
1	2	3	4	5	6

Node 4 is the next unvisited node, so we call top_sort_recurse(adj, 4, ts). There's nothing it's adjacent to that we haven't seen, so it gets placed in index 2. Naturally, it follows that 6 will get placed in index 1.

6	4	2	5	3	1
1	2	3	4	5	6

So, here's one way we can dress ourselves:

1. Undergarments
2. Socks
3. Slacks
4. Belt
5. Shoes
6. Shirt

Okay, so it's a bit odd, but it'll work!!! Notice that we could get different answers if we simply renumbered the nodes in a different order which would affect the order in which we visit nodes from the non-recursive function.