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:

$$\begin{tikzcd} A \arrow[r] \arrow[d] & B \arrow[d] \\ C \arrow[r] & D \end{tikzcd}$$

(Imagine A standing for waking up,
B standing for taking a shower,
C standing for eating breakfast, and
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.

Topological Sort – Recursive Algorithm based on DFS

We can 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:

$$\begin{tikzcd}
& A \arrow[dl] \arrow[d] \arrow[dr] & \\
B & C & D
\end{tikzcd}$$

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:

$$\begin{tikzcd}
6 \arrow[r] \arrow[d] & 1 \\
2 \arrow[r] \arrow[d] & 5 \\
3 & 4 \arrow[l]
\end{tikzcd}$$

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:

```
6 --------> 1
|
|
v
2--------->5
|
|
v
3<---------4
```

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.

Topological Sort – Iterative Version

Just as there is always a vertex in a directed acyclic graph (DAG) that has no outgoing edges, there must ALSO be a vertex in a DAG that has no incoming edges. This vertex corresponds to one that is safe to put in the *front* of the topological sort, since it has no prerequisites.

Thus, the algorithm is as follows for a graph, G, with n vertices:

```
1. Initialize TOP to be an empty list

2. While TOP has fewer than n items:

        a. Find a vertex v that is not in TOP that has an in degree of 0.
        b. Add v to TOP.
        c. Remove all edges in G from v.
```

In implementing the algorithm, store separately, the in degrees of each vertex. Every time you remove an edge in step 2c, update the corresponding in degree count.

For a sparse graph, in order to implement step a, use a Priority Queue. Keep in mind that every time you change an in-degree of a vertex, you have to delete the item from the priority queue and re-insert it.

Let’s apply this algorithm to our class constraints:

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.

COP 3223 goes first, since it has no pre-requisites.
COP 3502 goes next, since it has no pre-requisites left.
COP 3330 goes next, since it has no pre-requisites left.
COT 3100 goes next, since it has no pre-requisites left.
COP 3503 goes next, since it has no pre-requisites left.
COT 3960 goes next, since it has no prerequisites left.
COP 3402 goes next, followed by
COT 4210.

Dijkstra's Algorithm

This algorithm finds the shortest path from a source vertex to all other vertices in a weighted directed graph without negative edge weights.

Here is the algorithm for a graph G with vertices $V = \{v_1, ... v_n\}$ and edge weights w_{ij} for an edge connecting vertex v_i with vertex v_j. Let the source be v_1.

Initialize a set $S = \emptyset$. This set will keep track of all vertices that we have already computed the shortest distance to from the source.

Initialize an array D of estimates of shortest distances. D[1] = 0, while D[i] = ∞, for all other i. (This says that our estimate from v_1 to v_1 is 0, and all of our other estimates from v_1 are infinity.)

While S != V do the following:

1) Find the vertex (not is S) that corresponds to the minimal estimate of shortest distances in array D. **Use a priority queue to speed up this step.**
2) Add this vertex, v_i into S.
3) Recompute all estimates based on edges emanating from v. In particular, for each edge from v, compute D[i]+w_{ij}. If this quantity is less than D[j], then set D[j] = D[i]+w_{ij}.

Essentially, what the algorithm is doing is this:

Imagine that you want to figure out the shortest route from the source to all other vertices. Since there are no negative edge weights, we know that the shortest edge from the source to another vertex must be a shortest path. (Any other path to the same vertex must go through another, but that edge would be more costly than the original edge based on how it was chosen.)

Now, for each iteration, we try to see if going through that new vertex can improve our distance estimates. We know that all shortest paths contain subpaths that are also shortest paths. (Try to convince yourself of this.) Thus, if a path is to be a shortest path, it must build off another shortest path. That's essentially what we are doing through each iteration, is building another shortest path. When we add in a vertex, we know the cost of the path from the source to that vertex. Adding that to an edge from that vertex to another, we get a new estimate for the weight of a path from the source to the new vertex.

This algorithm is greedy because we assume we have a shortest distance to a vertex before we ever examine all the edges that even lead into that vertex. In general, this works because we assume no negative edge weights. The formal proof is a bit drawn out, but the intuition behind it is as follows: If the shortest edge from the source to any vertex is weight w, then any other path to that vertex must go somewhere else, incurring a cost *greater than w. But, from that point, there's no way to get a path from that point with a smaller cost, because any edges added to the path must be non-negative.*

By the end, we will have determined all the shortest paths, since we have added a new vertex into our set for each iteration.

This algorithm is easiest to follow in a tabular format. The adjacency matrix of an example graph is included below. Let a be the source vertex.

	a	b	c	d	e
a	0	10	inf	inf	3
b	inf	0	8	2	inf
c	2	3	0	4	inf
d	5	inf	4	0	inf
e	inf	12	16	13	0

Here is the algorithm:

	Estimates	b	c	d	e
Add to Set					
a		10	inf	inf	3
e		10	19	16	3
b		10	18	12	3
d		10	16	12	3

We changed the estimates to c and d to 19 and 16 respectively since these were improvements on prior estimates, using the edges from e to c and e to d. But, we did NOT change the 10 because 3+12, (the edge length from e to b) gives us a path length of 15, which is more than the current estimate of 10. Using edges bc and bd, we improve the estimates to both c and d again. Finally using edge dc we improve the estimate to c.

Now, we will prove why the algorithm works. We will use proof by contradiction. After each iteration of the algorithm, we "declare" that we have found one more shortest path. We will assume that one of these that we have found is NOT a shortest path.

Let t be the first vertex that gets incorrectly placed in the set S. This means that there is a shorter path to t than the estimate produced when t is added into S. Since we have considered all edges from the set S into vertex t, it follows that if a shorter path exists, its last edge must emanate from a vertex outside of S to t. But, all the estimates to the edges outside of S are greater than the estimate to t. None of these will be improved by any edge emanating from a vertex in S (except t), since these have already been tried. Thus, it's impossible for ANY of these estimates to ever become better than the estimate to t, since there are no negative edge weights. With that in mind, since each edge leading to t is non-negative, going through any vertex not in S to t would not decrease the estimate of its distance. Thus, we have contradicted the fact that a shorter path to t could be found. Thus, when the algorithm terminates with all vertices in the set S, all estimates are correct.

Bellman- Ford Algorithm

This algorithm finds shortest distances from a source vertex for directed graphs with or without negative edge weights. This algorithm works very similar to Dijkstra's in that it uses the same idea of improving estimates (also known as edge relaxation), but it doesn't use the greedy strategy that Dijkstra's uses. (The greedy strategy that Dijkstra's uses is assuming that a particular distance is optimal without looking at the rest of the graph. This can be done in that algorithm because of the assumption of no negative edge weights.)

The basic idea of relaxation is as follows:

1) Maintain estimates for distances of each vertex.

2) Improve these estimates by considering particular edges. Namely, if your estimate to vertex b is 5, and edge bd has a weight of 3, but your current estimate to vertex d is greater than 8, improve it!

Using this idea, here is Bellman-Ford's algorithm:

1) Initialize all estimates to non-source vertices to infinity.
   Denote the estimate to vertex u as D[u], and the weight of an
   edge (u,v) as w(u,v).

2) Repeat the following |V| - 1 times:

```
        a) For each edge (u,v)
                if D[u]+w(u,v) < D[v]
                        D[v] = D[u] + w(u,v)
```

The cool thing is that it doesn't matter what order you go through each edge in the graph in the inner for loop in step 2. You just have to go through each edge exactly once.

This algorithm is useful when the graph is sparse, has negative edge weights and you only need distances from one vertex.

Let's trace through an example. Here is the adjacency matrix of a graph, using a as the source:

	a	b	c	d	e
a	0	6	inf	inf	7
b	inf	0	5	-4	8
c	inf	-2	0	inf	inf
d	2	inf	7	0	inf
e	inf	inf	-3	9	0

	Estimates		a	b	c	d	e
Edge							
none			0	inf	inf	inf	inf
ab, ae		0	6	inf	inf	7	
bc, bd, be, ec, ed		0	6	4	2	7	
all, with cb		0	2	4	2	7	
all, with bd		0	2	4	-2	7	

The proof for the correctness of this algorithm lies in the fact that after each ith iterartion each estimate is the shortest path using at most i edges. This is certainly true after the first iteration. Now, assume it is true for the ith iteration. Under this assumption, we must prove it is true for the i+1th iteration. Notice that either the shortest path using at most i+1 edges uses at most i edges OR, is a shortest path of i edges with one more edge tacked on. This is because all shortest paths contain shortest paths. BUT, the way the algorithm works, ALL shortest paths of length i are considered, along with all edges added to them. Thus, the algorithm MUST come up with the optimal path using at most i+1 edges.

Longest Path in a DAG

Since a DAG doesn't have cycles, there are well-defined longest paths in DAGs. We can use Bellman-Ford's algorithm to find the longest path in a DAG from a chosen source vertex. *Basically, just negate all the edge weights. Now, the negative of the minimum cost for a path in this adjusted graph is the same as the maximum cost path in the original graph!*

Another way to solve this problem would be to use memoization. Recursively, a solution looks like this, assuming that a path always existed:

```
int longpath(ArrayList[] graph, int source, int end) {

    if (memo[source] != -1) return memo[source];
    if (source == end) return 0;

    int best = Integer.MIN_VALUE;

    for (int i=0; i<graph[source].size(); i++) {

        int cur = graph[source].get(i).weight +
         longpath(graph, graph[source].get(i).vertex, end);

       best = Math.max(best, cur);
    }

    memo[source] = best;
    return best;
}
```

Each array list must be an array list of objects that have the instance variables weight and vertex that store those corresponding items for that connection from the source variable and the memo array has to be declared as a static class variable.

In essence, the code just says, "Try all paths that lead from source. The longest these paths could be is the sum of the edge from source to the next vertex, plus the longest path from that next vertex to our end vertex. Of all of these, take the longest!"