Graphs have many applications outside of networks.

⇒ Scheduling courses to satisfy a track in major (e.g. systems)

Courses with pre-reqs: 200, 201 ⇆ 101, 150, 190
302 ⇆ 200, 201

node ~ course

edge \( (u,v) \) if \( u \) is a pre-req for \( v \)

![Graph diagram]

Ordering of courses to take?

\[ \text{exs': } 101, 201, 202, 315, 431, 701 \]  \[ \text{More than one} \]
\[ 150, 201, 202, 311, 455, 701 \]  \[ \text{correct solution!} \]

"What must be true about graph?"
- Directed
- No cycles

Topological sort - in a directed acyclic graph (DAG), an ordering of vertices s.t. each edge is directed towards later vertex.
Not obvious for all graphs:

How to find ordering?

Hint: indegree of vertex $v$: number of edges $(uv)$ (incoming edges to $v$).

ex: indegree of $A = 0$, $B = 0$, $C = 5$, etc.

"Which node(s) should be output first?"

$\rightarrow$ Those with indegree = 0

Which node(s) """" second?"

Hint: Look back at course graph and consider deleting nodes/edges.

$\rightarrow$ Those that have indegree = 0 after deleting the first set of output nodes and their outgoing edges.

$\text{1st: these}$
Note: Since graph is acyclic, there will always be some node with indegree = 0.

If (by assumption) all nodes had indegree > 0:

\[ \text{if this edge connects to } a \text{ or } b \text{, there will be a cycle.} \]

**Top-Sort-Print (T):**

1. Until all vertices printed:
2. Look for a non-printed vertex, \( u \), with indegree = 0
3. Print \( u \)
4. ("Delete" \( u \)): Decrement indegrees of nodes adjacent to \( u \).

**Ex.:**

```
(101) → (201) → (202)
(150) → (190)
```

Print: \( (101, 150, 190, 201, 202) \ldots \)

**Run Time:** \( O(|V|, |E|) = O(|V|^2) \)
How can we make this faster?

Notice in every step, we update indegrees and look for node with indegree = 0.

When a node has indegree = 0, store it to be next node to be deleted. Store in queue.

TopSort (G) // G=(V, E). Prints nodes of G in top-sort order
1. For all \( v \in V \)
2. \( \text{if (v.indegree == 0)} \)
3. \( Q.enqueue(v) \)
4. while \( Q.empty() \) \( \rightarrow O(1) \)
5. \( u = Q.dequeue() \)
6. \( \text{print}(u) \)
7. \( \text{// decrement indegrees of neighbors for all neighbors, } v, \text{ of } u \) \( \rightarrow O(|E|) \)
8. \( v.indegree-- \)
9. \( \text{if (v.indegree == 0)} \)
10. \( Q.enqueue(v) \).

\[ \begin{align*}
  A & \rightarrow D \\
  C & \rightarrow B \\
  G & \rightarrow C \\
  D & \rightarrow G \\
  E & \rightarrow G
\end{align*} \]

Printed: A B E F G D C
Run-Time:
Steps 1-3: \( O(1V_1) \)
Steps 4-10: \( O(1V_1) + (\text{not } *) O(1E) \)

Consider each vertex linear \# times, each edge once: \( O(1V_1 + 1E) \)

Turns out, can use top-sort to find shortest paths in DAG's more efficiently than Dijkstra's and Bellman-Ford.

Even for DAGs with negative edge weights.

First consider DAGs with positive edge weights.

How does top-sort ordering help?
Consider nodes in top-sort order.

**DAG vs. non-DAG:**

For DAGs: a node earlier in the top-sort order is guaranteed to have shorter distance from s.

For non-DAGs, cycles prevent a similar guarantee (e.g. \( s \rightarrow a \) dist = 2).

How does this help over Dijkstra's?

(Recall: Dijkstra's uses minHeap to find next node to process (unprocessed node with min distance).

⇒ Put nodes in top-sort order (in a queue) and process nodes in this order!

**Ques:** Does this work for graphs with negative edge weights?
Yes' Recall problem with neg. edge weights and Dijkstra's was that a future (unprocessed) node could possibly improve distance to an already processed node:

![Diagram]

a processed before b but a.dist can be improved to 0 via b.

If nodes are processed in top-sort order, there are no paths from future nodes to previous nodes.

Top-sort order: 

![Diagram]
TopSort - SP $(G, s)$ // $G$ is DAG : $(V, E)$, $s \in V$

// Finds shortest paths $u$ from $s$ to all other nodes:

1. $Q = \text{TopSort}(V, E)$ //store vertices in top sort order
2. for all $v \in V$, $v$ dist = $\infty$, $v$ pred = null. (no more 'processed
3. $s$. dist = 0
4. while (! $Q$. empty)
5.   $u = Q$. dequeue() (6) $\rightarrow$ (5)
6.     for each edge $(u, v)$
7.       update $(u, v)$ -> if ($v$. dist > $u$. dist + $w_{u,v}$)
8.         $v$. dist = $u$. dist + $w_{u,v}$
9.         $v$. pred = $u$

ex : TopSort - SP $(G, s)$

\[\begin{array}{c}
\text{Top Sort order} \\
(3) \rightarrow (6) \rightarrow (2) \rightarrow (1) \rightarrow (5) \rightarrow (4) \rightarrow (1) \rightarrow (7) \rightarrow (1) \rightarrow (5) \\
\end{array}\]

\[\begin{array}{c}
\text{Q:} \\
\{r, s, t, x, y\} \\
\end{array}\]

r is "thrown" away (can't get to it):
Run Time: Step 1 (Top-Sort): $O(1V_1 + 1E_1)$
Steps 2-3: $O(1V_1)$
Steps 4-7: $O(1V_1) + (\text{not +}) = O(1E)$

Total: $O(2(1V_1 + 1E_1)) = O(1V_1 + 1E_1)$ vs. $O(1V_1 \log 1V_1 + 1E_1 \log 1V_1)$ for Dijkstra’s, $O(1V_1 \cdot 1E_1)$ for BF.

<Can SKIP the rest>

Correctness:
When $u$ is dequeued, $u$'s dist is correct distance to $u$:
1. Only edges $(v,u)$ where $v$ comes before $u$ in the toposort order can improve the distance to $u$.
2. Algorithm finds optimal distance to all such nodes before setting distance for $u$.
   
   Distance to $u$ must be optimal.

Why processed not needed:

Cyclic

u most recently deleted:

Acyclic (nodes deleted in toposort order)

u most recently deleted

edge $(u,v)$ exists where $v$'s dist < $u$'s dist so don't have $\text{dist} > u$'s dist
update $v$. 