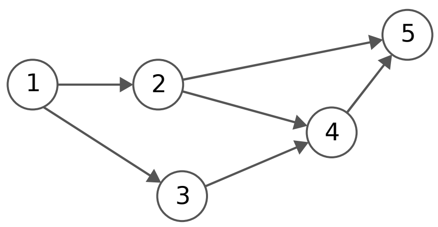
Topological Sort

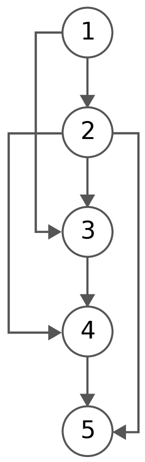
The **topological sort** algorithm takes a directed graph and returns an array of the nodes where each node appears *before* all the nodes it points to.

> The ordering of the nodes in the array is called a *topological ordering*.

Here's an example:



Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.

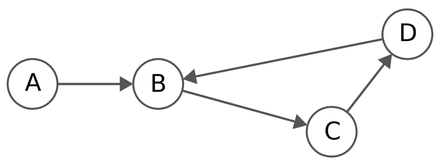


So [1, 2, 3, 4, 5] would be a topological ordering of the graph.

> Can a graph have more than one valid topological ordering? Yep! In the example above, [1, 3, 2, 4, 5] works too.

Cyclic Graphs

Look at this directed graph with a cycle:

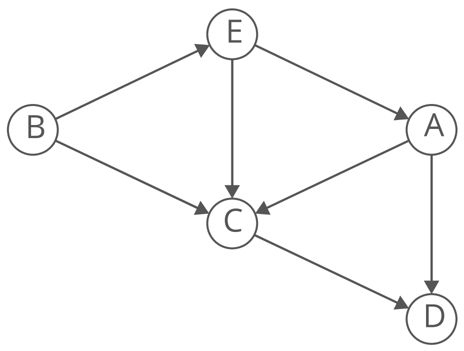


The cycle creates an impossible set of constraints—B has to be before *and* after D in the ordering.

As a rule, **cyclic graphs don't have valid topological orderings.**

The Algorithm

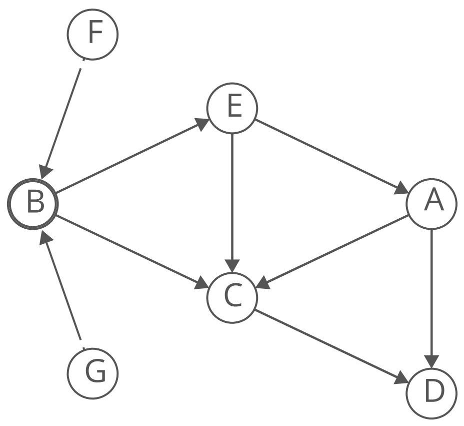
How can we produce a topological ordering for this directed graph?



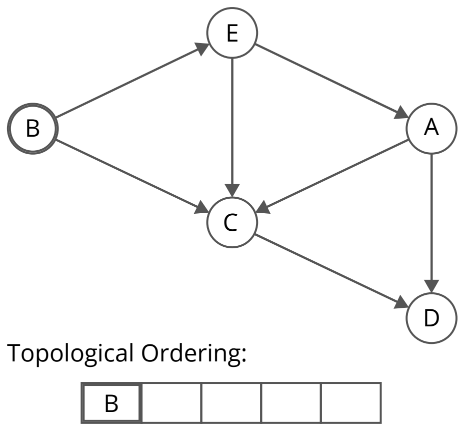
Well, let's focus on the *first* node in the topological ordering. That node can't have any incoming directed edges; it must have an indegree ↴ of zero.

Why?

Because if it had incoming directed edges, then the nodes pointing to it would have to come first.

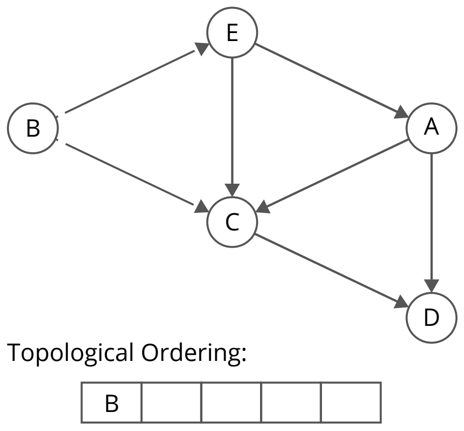


So, we'll find a node with an indegree of zero and add it to the topological ordering.



That covers the first node in our topological ordering. What about the next one?

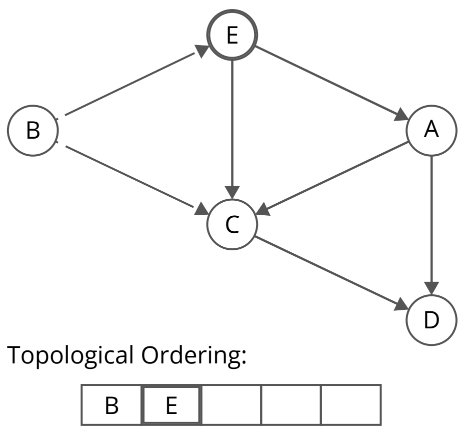
Once a node is added to the topological ordering, we can take the node, and its outgoing edges, out of the graph.



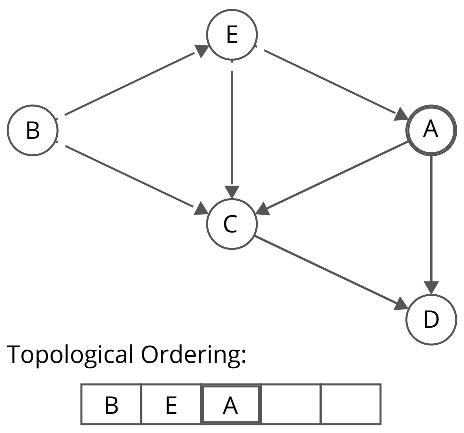
Then, we can repeat our earlier approach: look for any node with an indegree of zero and add it to the ordering.

> This is a common algorithm design pattern:
>
> 1. Figure out how to get the *first* thing.
> 2. Remove the first thing from the problem.
> 3. Repeat.

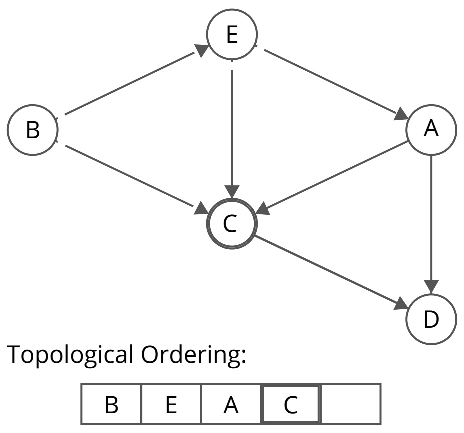
Here's what this looks like on our graph. We'll grab a node with an indegree of 0, add it to our topological ordering and remove it from the graph:



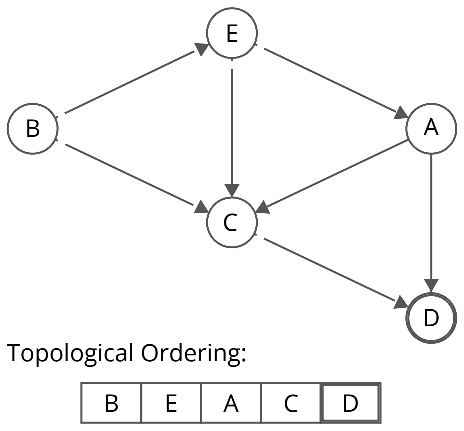
`and repeat`



`and repeat`



`until we're`



out of nodes.

> Note: this isn't the *only* way to produce a topological ordering.

Implementation

We'll use the strategy we outlined above:

1. Identify a node with no incoming edges.
2. Add that node to the ordering.
3. Remove it from the graph.
4. Repeat.

We'll keep looping until there aren't any more nodes with indegree zero. This could happen for two reasons:

- There are no nodes left. We've taken all of them out of the graph and added them to the topological ordering.
- There are some nodes left, but they all have incoming edges. This means the graph has a cycle, and no topological ordering exists.

One small tweak. Instead of actually *removing* the nodes from the graph (and destroying our input!), we'll use a hash map to track each node's indegree. When we add a node to the topological ordering, we'll decrement the indegree of that node's neighbors, representing that those nodes have one fewer incoming edges.

Let's code it up!

```
def topological_sort(digraph):
    # digraph is a dictionary:
    #   key: a node
    # value: a set of adjacent neighboring nodes
    # construct a dictionary mapping nodes to their
    # indegrees
    indegrees = {node : 0 for node in digraph}
    for node in digraph:
        for neighbor in digraph[node]:
            indegrees[neighbor] += 1

    # track nodes with no incoming edges
    nodes_with_no_incoming_edges = []
    for node in digraph:
        if indegrees[node] == 0:
            nodes_with_no_incoming_edges.append(node)
```

```python
# initially, no nodes in our ordering
topological_ordering = []

# as long as there are nodes with no incoming edges
# that can be added to the ordering
while len(nodes_with_no_incoming_edges) > 0:

    # add one of those nodes to the ordering
    node = nodes_with_no_incoming_edges.pop()
    topological_ordering.append(node)

    # decrement the indegree of that node's neighbors
    for neighbor in digraph[node]:
        indegrees[neighbor] -= 1
        if indegrees[neighbor] == 0:
            nodes_with_no_incoming_edges.append(neighbor)
# we've run out of nodes with no incoming edges
# did we add all the nodes or find a cycle?
if len(topological_ordering) == len(digraph):
    return topological_ordering  # got them all
else:
    raise Exception("Graph has a cycle! No topological ordering exists.")
```

Time and Space Complexity

Breaking the algorithm into chunks, we:

- *Determine the indegree for each node.* This is $O(M)$ time (where M is the number of edges), since this involves looking at each directed edge in the graph once.
- *Find nodes with no incoming edges.* This is a simple loop through all the nodes with some number of constant-time appends. $O(N)$ time (where N is the number of nodes).
- *Add nodes until we run out of nodes with no incoming edges.* This loop could run once for every node—$O(N)$ times. In the body, we:
    - Do two constant-time operations on an array to add a node to the topological ordering.
    - Decrement the indegree for each neighbor of the node we added. Over the entire algorithm, we'll end up doing exactly one decrement for each edge, making this step $O(M)$ time *overall*.
- *Check if we included all nodes or found a cycle.* This is a fast $O(1)$ comparison.

All together, the time complexity is $O(M+N)$.

That's the fastest time we can expect, since we'll have to look at all the nodes and edges at least once.

What about space complexity? Here are the data structures we created:

- indegrees—this has one entry for each node in the graph, so it's $O(N)$ space.
- nodesWithNoIncomingEdges—in a graph with no edges, this would start out containing every node, so it's $O(N)$ space in the worst case.
- topologicalOrdering—in a graph with no cycles, this will eventually have every node. $O(N)$ space.

All in all, we have three structures and they're all $O(N)$ space. **Overall space complexity: $O(N)$.**

This is the best space complexity we can expect, since we *must* allocate a return array which costs $O(N)$ space itself.

Uses

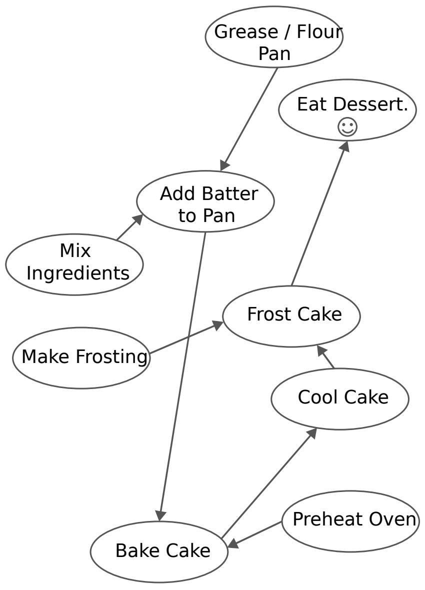
The most common use for topological sort is ordering steps of a process where some the steps depend on each other.

As an example, when making chocolate bundt cake,

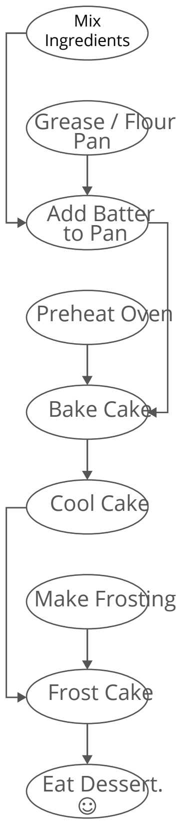
- The ingredients have to be mixed before going in the bundt pan.
- The bundt pan has to be greased and floured before the batter can be poured in.
- The oven has to be preheated before the cake can bake.
- The cake has to be baked before it cools.
- The cake has to cool before it can be iced.

> While we've chosen a fun example here, the same logic applies to any set of tasks with dependencies, like building components in a large software project, performing data analysis in Map-Reduce job, or bringing up hardware components at boot time. (For example, the mother board has to initialize the hard drive before the BIOS tries to load the bootloader from disk.)

This process can be represented as a directed graph. Each step is a node, and we'll add directed edges between nodes to represent that one step has to be done before another.



Once we have our dependencies represented using a directed graph, we can use topological sort to provide a valid ordering to tackle all the steps.


Mix Ingredients
Grease / Flour Pan
Add Batter to Pan
Preheat Oven
Bake Cake
Cool Cake
Make Frosting
Frost Cake
Eat Dessert. ☺