1 Depth-first search in directed graphs

In directed graphs, dfs classifies the edges of the graph into four types:

- Tree edges: these are the edges of the depth-first search tree, the pathways whereby the recursion proceeds. That is, (u, v) is a tree edge if `explore`(v) was called from `explore`(u).
- Forward edges: these go from a vertex to a descendant (other than child) in the depth-first search tree. You can tell such an edge (v, w) because `pre[v]` $<$ `pre[w]`.
- Back edges: these go from a vertex to an ancestor in the depth-first search tree. You can tell such an edge (v, w) because, at the time it is traversed, `pre[v]` $>$ `pre[w]`, and `post[w]` is undefined.

  item Cross edges: these go from "right to left," from a newly discovered node to a node that lies in a part of the tree whose processing has been concluded. You can tell such an edge (v, w), because, at the time it is traversed, `pre[v]` $>$ `pre[w]`, and `post[w]` is defined.

2 Directed acyclic graphs

A *cycle* in a directed graph is a path $(v_0, v_1, \ldots, v_n)$ such that (v_n, v_0) is also an edge. A directed graph is *acyclic*, or a *dag*, if it has no cycles.

Claim: A directed graph is acyclic if and only if depth-first search on it discovers no backedges.

Proof: If (u, v) is a backedge, then (u, v) together with the path from v to u in the depth-first search tree form a cycle.

Conversely, suppose that the graph has a cycle, and consider the vertex v on the cycle assigned the largest `pre[v]` number. Then the edge (v, w) leaving this vertex in the cycle *must* be a back edge, since it goes from a higher `pre[v]` number to a higher `pre[w]` number.

It is therefore very easy to use depth-first search to see if a graph is acyclic: Just check that no backedges are discovered. But we often want more information about a dag: We may want to *topologically sort it.* This means to order the nodes of the graph from left to right so that all edges go from left to right. (*Note:* This is possible if and only if the graph must be acyclic. Can you prove it? One direction is easy; and the topological sorting algorithm described next provides a proof of the other.) This is interesting when the nodes of the dag are tasks that

must be scheduled, and an edge from u to v says that task u must be completed before v can be started. The problem of topological sorting asks: in what order should the tasks be scheduled so that all the precedence constraints are satisfied.

To topologically sort a dag, we simply do a depth-first search, and then arrange the nodes of the dag in decreasing `post[v]`. That this simple method correctly topologically sorts the dag is a consequence of the following simple property of depth-first search:

For each edge (u, v) *of* G, `post[u]` $<$ `post[v]` *if and only if* (u, v) *is a back edge.*

Proof: If `post[u]` $<$ `post[v]` then v is visited before u (otherwise the existence of edge (u, v) ensures that v must be pushed onto the stack before u can be popped, resulting in `post[u]` $>$ `post[v]` —s contradiction). Furthermore, since v cannot be popped before u, it must still be on the stack when u is pushed on to it. It follows that v is on the path from the root to u in the depth first search tree, and therefore (u, v) is a back edge.

The other direction is trivial: If (u, v) is a back edge then u is a decsendant of v on the tree, and therefore `post[u]` $<$ `post[v]`.

This property proves that our topological sorting method correct. Because take any edge (u, v) of the dag; since this is a dag, it is not a back edge; hence `post[u]` $>$ `post[v]`. Therefore, our method will list u before v, as it should. We conclude that, using depth-first search we can determine in linear time whether a directed graph is acyclic, and, if it is, to topologically sort its nodes, *also in linear time.*

Dags are an important subclass of directed graphs, useful for modeling hierarchies and causality. Dags are more general than rooted trees and more specialized than directed graphs. It is easy to see that every dag has a *sink* (a node with no outgoing edges). Here is why: Suppose that a dag has no sink; pick any node v_1 in this dag. Since it is not a sink, there is an outgoing edge, say (v_1, v_2). Consider now v_2, it has an outgoing edge (v_2, v_3). And so on. Since the nodes of the dag are finite, this cannot go on forever, nodes must somehow repeat –and we have discovered a cycle! Symmetrically, every dag also has a *source*, a node with no incoming edges. (But the existence of a source and a sink does not of course guarantee the graph is a dag, it's easy to come up with a counterexample.)

The existence of a source suggests another algorithm for outputting the nodes of a dag in topological order:

```
Find a source, output it, and delete it from the graph.  Repeat until the graph
is empty.
```

Can you see why this correctly topologically sorts any dag? What will happen if we run it on a graph that has cycles? How would you implement it in linear time?