PROFESSOR: OK. Who's going to lecture? Wow. Nice. Does lecture make sense? One, two, three-- sort of. OK. Then I'm going to have an easy job. So any pointed questions? Any specific pinpoints?

We talked about DFS, but DFS and BFS are sort of related, so I'm happy to take both.

AUDIENCE: The classification of edges?

PROFESSOR: OK. We're going to talk about that for sure. Anything else?

AUDIENCE: I thought it was funny that in undirected graphs, you could have a backward-- or something like that.

PROFESSOR: You don't have to give a radiance. I will derive it. OK So what edges do we have in directed versus undirected. OK. What else? Cool. So I'm going to go through the concepts in DFS really quickly. And we're going to focus on this. Because it seems like this is where the issues are. I want this.

So what's a graph?

AUDIENCE: Interconnected notes.

PROFESSOR: OK. Fancy names. Come on. We had this in the last workshop.

AUDIENCE: A set of edges and vertices.

PROFESSOR: All right. How do I draw my vertices?

AUDIENCE: Dots. Circles.

PROFESSOR: Dots. How do I draw my edges?

AUDIENCE: Dots. Lines. Oops.

PROFESSOR: Looks like a graph? What kind of graph? Directed. Because there are arrows and not straight up lines. Right? How do we store graphs in Python? Fancy name first and then implementation.

AUDIENCE: [INAUDIBLE] list.

PROFESSOR: OK. What's an adjacency list in Python?

AUDIENCE: It just shows what notes are adjacent to other notes.

PROFESSOR: OK. So what data structure do we use? What elementary python data structure do we use?

AUDIENCE: A dictionary?

PROFESSOR: All right. So an adjacency list is a dictionary that keys our--

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. So the keys are the vertices. So on and so forth. And what are the values?

AUDIENCE: The nodes? Adjacent nodes?

PROFESSOR: OK. A list of something. So after our presentation, is the object the one that has edges and the simplified one that has vertices. So suppose I want to go for the simple one. Where vertices do I have in A's list?

AUDIENCE: B and G. Right?

PROFESSOR: B and G. Sounds good. Right. What vertices do I have in B's list?

AUDIENCE: C.

PROFESSOR: Why do I not have A?

AUDIENCE: Because it's an arrow, so it can't get to A.

PROFESSOR: Yeah. So the nodes that I have in the list are the nodes that are reachable from that node. I have an edge from B to C, so C is reachable from B. I do not have an edge from B to A because my edges are directed. So A is not there. OK. Let's write a bit of pseudo- So although it's Python, let's say our graph is represented by a class G. Let's say in this class I have a dictionary called al, which has what we want.

If I want to get all the vertices in the graph, I want to write the method that returns all the vertices. What should it look like? Just to make sure that we're all getting the data structure.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. So the keys are in the vertices. Now, given the vertex, I want its neighbors. How do I write that?

AUDIENCE: You mean only the vertices that it can go to?

PROFESSOR: Yeah.

AUDIENCE: Yeah. Dictionary and that vertex.

PROFESSOR: So return dictionary--

AUDIENCE: al bracket vertex v.

PROFESSOR: Cool. OK. So this makes sense for everyone, right? Let's start writing DFS because no one said they don't understand DFS. So let's say we have a DFS procedure. And it gets a graph. And it has to do DFS. What do you need for DFS? Starting points would be good. Just like for DFS, right?

So how do we do DFS?

AUDIENCE: Start with one node and then we keep on following a path down by picking the first vertex. So like for A, we'd start with A. Then we'd pick the first vertex in its list. And then we pick the first vertex in that one's list, first vertex in that one's list, until we run out.

PROFESSOR: OK. So basically you start today. And for every neighbor, you recursively do DFS on that neighbor. Right? So since this is recursive, let's write it in a recursive manner, nice and easy. So say DFS calls DFS visit of g and s. And let's say DFS visit takes the graph, takes the node that we're visiting, and how are we implementing this?

AUDIENCE: Just return [INAUDIBLE]. No. It marks as visited.

PROFESSOR: OK. Marking something as visited is important to lead this. So what's the main thing that you do in DFS? Iterate over the neighbors and recurse. Right? So for n in g neighbors of v called DFS visit of g and n. So this is the basic idea. It has some holes that we need to fill because otherwise it doesn't quite work. This is going to recurse forever, which is not very good if you have a time limit that your code needs to obey.

So how do I make it not recurse forever?

AUDIENCE: Every time you visit, you're about to visit a node, check if you've already visited it.

PROFESSOR: All right. So we should keep track of the nodes that we've already visited and not visit them again. Right. So you start with A, go to B, go to C, you'd better not go to A again because you've already been there. So let's implement that this way. Let's make a new object called DFS result that is sort of that drawer that you have where you stick everything in where you don't have a good place. So everything I do not have a good place to put it, you just stick it in a drawer. And then you can close the drawing that be like, it's clean. Everything's clean.

So let's say DFS result is that drawer. Everything that they need from now on, we're just going to stick in there. So I have an object DFS result. And we're going to figure out what we put in it. But at the beginning of DFS, I'm going to create that. So create the new object. And then I'm going to pass it onto to DFS visit.

So now we have a drawer where we can put everything we want. So let's keep track of the visited information here. How would I keep track of what nodes I visited?

AUDIENCE: In a list?

PROFESSOR: In a list. So if I do a list, it turns out that in DFS, and VFS, for that matter, you check if you visited a node pretty often. So it would be kind of slow. So let's have a dictionary. What are we going to put in that dictionary?

AUDIENCE: The node [INAUDIBLE] and listed [INAUDIBLE].

PROFESSOR: OK. So we're going to have a visited dictionary. And then the keys are going to be nodes. And let's say we're only going to put the nodes that we visit in it so we don't have to initialize it. And the values are going to be true. Because if the node is in the dictionary, it means we visited it. So for all the nodes that are in there, visited is going to be true. Sorry.

OK. So where do I fill this in? Let's use it.

AUDIENCE: Four n and g neighbors, set it to true. But before that--

PROFESSOR: So here?

AUDIENCE: R not. Um.

PROFESSOR: OK. So let's say this. Let's say these are lines 1, 2, 3. Line 1.5 or 2.5?

AUDIENCE: So at n 1.5, check if v is in the dictionary.

PROFESSOR: Here? OK. If v in r dot visited, then--

AUDIENCE: Return. Then 2.5, set r dot n to true. You would only do it n times--

PROFESSOR: Wait. So here you're saying say r dot visited of n equals true?

AUDIENCE: Oh, no. Change the v in v visited return, change that into a loop. Put that after the loop. Put that into the loop. And it's n visited to the [INAUDIBLE].

PROFESSOR: n in r dot visited-- we probably want not n, right?

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. R dot visited is true and then the DFS. So will this achieve the thing that I wanted at the beginning. You start at A, you go to B, you go to C, and then you do not go to A? Not quite.

AUDIENCE: We haven't had our starting yet. Yeah. We could add V at the beginning and check at the end. Because that's only the visited, right?

PROFESSOR: So I think I like Christian's suggestion. r dot visited of v is true. And then we can remove it from here.

AUDIENCE: It has to be [INAUDIBLE] because the other case is going to cover for if you visited again. But the first time it's being called, it's not checking--

PROFESSOR: Well you mean the first time here?

AUDIENCE: The first call that's being made, and the recursive call, it's going to be as if it's not into the visited node.

PROFESSOR: Yeah. But this is at the beginning. Right? So this calls DFS visit to the starting node. Do we care? So I think this should work. It's not the only way to write it, but it's reasonably clean. It's easy to reason about, and it works. Does it make sense to everyone?

OK. We're going to change things a little bit. So instead of using visited, we're going to keep track for every node of its parent. So the node that we visited from. So we're not going to use visited anymore. Instead we're going to use parent of v is going to be some other node.

So what I want is for this graph, I said I'm going to go from A to B, and then from B to C. So I want the parent of B to be A, and the parent of C to be B. Make sense for everyone? So what's the parent of A?

AUDIENCE: None.

PROFESSOR: None's a good value, right? There's no parent. OK. So how would I modify this code to write that?

AUDIENCE: Stick a line in 2.75 and establish 3b as the parent of n.

PROFESSOR: So r dot parent?

AUDIENCE: Yeah, of n is v.

PROFESSOR: Of n is v. Good. And I don't have visited anymore, so this line is going to blow up. So what do I do instead?

AUDIENCE: Oh. It doesn't have a parent?

PROFESSOR: Yep. So if it doesn't have a parent, we didn't visit it. So the parents works just like visited, except the values are not going to be all true. They're going to be something a bit more useful.

AUDIENCE: We can't test for none because the first one has a value of none.

PROFESSOR: But they're not testing for none, we're saying is it in the dictionary or not?

AUDIENCE: Oh. OK.

PROFESSOR: That's a good point. You could be testing for none. That wouldn't work. So you have to write the check like this. And we're going to erase this because otherwise, it's going to throw an exception. And what else do we need? There's one missing parent.

AUDIENCE: Inverse form?

PROFESSOR: OK. Where do I say that?

AUDIENCE: In DFS.

PROFESSOR: OK. One, two, three, which line? 1.5, 2.5, 3.5?

AUDIENCE: 2.5.

PROFESSOR: 2.5. All right, so here. What do I write? Yes?

AUDIENCE: Parent of s is none.

PROFESSOR: OK. So r dot parent s is none. So now this works. Right? Any questions so far? Nope. Basic DFS works. Everyone's happy with it? Let's try to track it for this graph here. And you're going to be in a better position than me because I'm not sure I can see the graph all the way from here. Almost.

So let's have the parents or the parent's dictionary here. And here, let's write the call structure as it will happen. So where do I start? DFS of A, right? DFS of A. And this is going to call DFS visit A.

What are A's neighbors?

AUDIENCE: b and g.

PROFESSOR: Cool. Is b in parents? What's in parents, at this point?

AUDIENCE: Just a. Oh.

PROFESSOR: A is none. Good. Excellent. So B is not in parents, so what am I going to do? And it's the parents in visited. Right? What's B's parents?

AUDIENCE: A.

PROFESSOR: OK. I'm going to call DFS visit of B. These children. C. Is C parents? No? What happens? Feel free to take over. When you get it, feel free to start talking, and I can just write.

AUDIENCE: Of C.

PROFESSOR: OK. What else?

PROFESSOR: C's parent is B. So I'm in B. The only child is C. C is not in parents. C gets in parent. C's parent is B. Call DFS visit of C, what are C's children?

AUDIENCE: [INAUDIBLE] and d.

PROFESSOR: So we're wondering about the order, right? Let's assume that in the adjacency list, all the nodes are in alphabetical order. So I'm going to have A, D, and F. OK. Is A in parents?

AUDIENCE: Yes.

PROFESSOR: Is D in parents?

AUDIENCE: No.

PROFESSOR: So what happens?

AUDIENCE: Parents visit D. D's parent is C.

PROFESSOR: D's neighbors?

AUDIENCE: E and F.

PROFESSOR: E is in the-- no.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Cool. Does E have any children?

AUDIENCE: No.

PROFESSOR: So I'm done. I get out of it, right? Is F in parents?

AUDIENCE: No.

PROFESSOR: F's parent is? Cool. F doesn't have any children. Right? So I get out of this DFS visit. I'm done with F. So I get out of this visit. And I get back to this one. I'm done with D, and I'm at F. Is F in parents? OK.

So what kind of edge is C to F?

AUDIENCE: Forward.

PROFESSOR: OK. So forward maybe tree. Let's start with the easy ones. So now we've gone forward with AB, BC, CD, DE, and DF. What kind of edges are these ones? Tree edges. Does anyone know why they're called tree edges? So if you look at the parent pointers that they have there, they're going to end up defining a tree.

And the tree is, the tree shows the order in which DFS looked at the notes. So let me erase this and draw the tree for DFS that we had so far. So we started at A, then we went to B. Then we went to C, then we went to D, E, F. So this is all part of the DFS tree.You can see that for all these nodes, their parent pointers point the right way.

So if you have parent pointers for every node, this is going to give you a tree. So these are tree edges because they're part of this tree. So now we have this edge from C to F. And this is what kind of edge? Forward edge. Why is it a forward edge?

AUDIENCE: Somewhere up higher in the tree is trying to get somewhere lower on the tree. It's going forward in time.

PROFESSOR: So it's going from a node in the tree to a node's child in the tree. Sorry, not child. Descendant. So F is strictly under C. So yeah. It takes us forward in time. It's a shortcut in the tree. So that's why it's a forward edge.

So we now have two types of edges so far. Tree edges and forward edges.

AUDIENCE: So tree edges [INAUDIBLE]?

PROFESSOR: Yep. So the tree edges are the ones that DFS follows. All of these are tree edges. Yes.

AUDIENCE: So like an edge from B to C, why can't that be a forward edge? Because technically it's not a shortcut, right? But it's saying it's pointing to the children, but it only has one child.

PROFESSOR: Because that's the edge that DFS took. So the edges that we actually follow in DFS have a special name, and they're tree edges. So first off, the most important edges are the edges that make up your DFS tree. And that's why those are tree edges. We don't even look at the other types. If you have an edge and DFS followed it, it's a tree edge. Done.

So these are all tree edges. We don't care about the other definitions. So don't be ashamed to ask questions about this because this is hard to understand. It's an issue we're trying to clarify here. So the point, why we're doing that example, is to figure this out. So please do ask your questions. Yes?

AUDIENCE: So is it edges that are followed at any point during the search process? I guess they're edges that never get touched. Right?

PROFESSOR: Yep. So when I'm at C, and I look at this edge, F is already in parents. So this edge is not going to be followed by DFS. There are some other types of edges that will not be followed. And we'll get to that by the time DFS completes. Yes.

AUDIENCE: You mean that like G to C is not--

PROFESSOR: You're ruining my example, man. I'm supposed to ask you that later on. So let's let go through it and see what kind of edge it is. You're right, but let's pretend we don't know that. And we'll see as we follow the DFS. OK. So yep. That one's a forward edge. We'll deal with it later. We're here.

We decided that C to F is a forward edge. F is already in parents. So we're going to return from DFS visit C. We're in DFS visit V. We're done with C. Do we have anything else to do here? So we go up to A. We're done with B. And we look at G. A to G. Is G in parents? No.

So we're going to call it. So A, B. DFS visit G. What are G's neighbors?

AUDIENCE: C.

PROFESSOR: OK. So is C in parents? Actually I was wrong. Sorry. So C is in parents, so we're not going to visit it. So let's see how the tree looks like at this point. So it looks like this. Is C a child of G?

So I'm wondering about this edge that we chose not to follow. We followed A to G so we know this is actually the edge. So from this edge, G to C, I'm wondering what kind of edge it is. So C is not G's child, right? So another forward edge. Sorry. I got confused earlier because they look the same in a drawing.

So it's not a forward edge, then it's a cross edge. There's one more edge that we haven't talked about because it was too early in the search. And that edge is from C to A. So if you remember there, we were in DFS visit C. And the first thing we did was we looked at A, we said, hey it's already in parents, so we're not going to visit it. So DFS did not follow the edge C to A.

That edge looks like this. What kind of edge is it? Back edge.

AUDIENCE: Wait. So it has to be direct descendant in order to be a forward edge? Because they are related through A, right? C and G?

PROFESSOR: So you have to look and which direction does it go? Does it go down the tree or up the tree?

AUDIENCE: It's going down if you consider G to C. From G to C, so it's a cross edge.

PROFESSOR: Yeah. But is C G's descendant?

AUDIENCE: No. But they are related.

PROFESSOR: They're related, but they're just related because they're in the tree.

AUDIENCE: OK. So it's only directly descendants?

PROFESSOR: Yep. So DFS puts together a tree. A forward edge is a shortcut in that tree. It lets you go forward in the VFS. A backward edge points to a parent in the tree, so it lets you go back in time. A cross edge takes you from one point to another point. And it's not a forward edge, not a backward edge. So it takes you in a whole different world.

So C, A, G, has-- A, G is the sub-tree, and then this guy's this other sub-tree, and they're different little worlds. And this edge goes from one to the other. So the algorithm to decide what kind of an edge it is, let's put it together. Which edge do we look at, which type do we think about first? Tree edge. So DFS followed the edge. It's a tree edge.

So we have tree edges. What do we do next? So say we have an edge from u to v. If DFS follows it, it's a tree edge. If not, then what question do I ask myself?

AUDIENCE: It could be one of three edges then, back, forward, or cross.

PROFESSOR: OK. So let's try to write a simple algorithm so that if I ask you this on a quiz, you can decide which is which.

AUDIENCE: So I think that you look forward if u is the parent of v.

PROFESSOR: OK. So if u is a parent of v-- well, let's say ascendant. It's not just the direct parent, but this is the right intuition. Then what kind of edge is it?

AUDIENCE: Then it's the forward edge.

PROFESSOR: So the edge is from u to v. If u is a parent of v, forward edge. Good. Otherwise?

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. Then?

AUDIENCE: Then backward edge. Else cross edge.

PROFESSOR: Does this make sense? So these are the four types of edges. DFS takes the edge, it's a forward edge. If not, we draw the DFS tree. And we see does the edge go forward in the tree, forward edge. Does it go backward, backward edge. Does it grow between completely unrelated nodes, cross edge.

OK. Now let me ask another question. Let's suppose we do this DFS thing again, but let's suppose we do it on an undirected graph. What types of edges do I have? Anyone remember off the top of their heads? Don't think so, right? So let's do it and find out.

So let me see where do I get some room. Here. Let's write the tree that's we're going to go. So let's do a VFS of this quickly and write the tree. So we start at A, then what? Then? Then? D. E.

AUDIENCE: And go back to D. It could go from C to A.

PROFESSOR: OK. So we have C to A. What kind of edge is this?

AUDIENCE: Backward edge.

PROFESSOR: OK. So we still have tree edges for sure and backward edges.

AUDIENCE: How do you get from C to A?

PROFESSOR: This was supposed to happen way before. So we went A, B, C, and then when we were at C, oh. Sorry. My bad. So we went from A to B, B to C, and then A was the first thing in C's adjacency list. So we saw this back edge. So C to D, D to E, D to F.

AUDIENCE: E to F? Or F to C?

PROFESSOR: And then F to C. So what kind of edge is this?

AUDIENCE: Backward.

PROFESSOR: So none of these guys. And then we're all the way back at A, right? A to-- oh. Sorry. No we're not done. So C to D, D to E, D to F, then we're back at C. And C to G.

AUDIENCE: There's supposed to be a line between C and F?

PROFESSOR: Oh, yeah. Sorry. I guess I can't copy. One, two, three, four, five, six, seven, eight, nine. One, two, three, four, five, six, seven, eight, nine. Now it's right. Thank you. OK. So what am I missing? Edge from C to G. What kind of edge?

AUDIENCE: C and G?

PROFESSOR: OK. So we have tree edges and we have backward edges. What do we not have? Forward edges and cross edges, right?

AUDIENCE: You need one from A to G. A and G are related.

PROFESSOR: What edge is this? Cool. Thanks.

AUDIENCE: You never add those, though, right? You just see that A was in the parent's list, and be like, OK.

PROFESSOR: Yeah. So for all the edges that are not tree edges, DFS doesn't actually follow them. We just care about them because other algorithms care about them. They let you compute fancy things on the graph. OK. So are we good with the types? So why can't I have a forward edge? So there are two types of edges I can't have, cross edges and forward edges. Why can't I have a forward edge? On an undirected graph?

So in order to have a forward edge, I would have to go C, D, F. And then not follow this edge. I mean not see this edge here. Right? If this edge would be undirected, I would say, hey, it goes from F to C. It's a background edge.

So I would have seen this edge when I would have been in C. So I'm comparing this tree with this tree, and trying to figure out why they're different. How does a forward edge look like? You have a node u, then you have some more tree stuff. And from here, you got to v. And then when you're at v, you didn't see the edge. Right? Because if you would have seen it, it would have been a background edge.

So you can only see the edge from u to v later on. So this is tree edges, a bunch of tree edges, and this is a forward edge. In an undirected graph, this never happens because when you're at v, you're going to see the edge. And you're going to mark it as a backward edge. So forward edges can ever happen. Yes? No? Is everyone happy?

So cross edges. Why can't they happen? A cross edge can never happen because in order for a cross edge to happen, I would have to go A, B, C, visit C's children, then go up and go somewhere else. And then see this edge. But, hey, when I was at C, why didn't I see this edge? Why did I only see it later? If it's an undirected graph, I would see this edge here and it would be a forward edge. Sorry. A tree edge, because that would take it. Yes?

AUDIENCE: So is it just when you're-- is it just because it's the way you traverse in DFS or would it apply generally to VFS as well?

PROFESSOR: VFS doesn't have forward and backward. Yeah. VFS is completely different. So these are all DFS terms, purely DFS. OK. So no forward edges, no cross edges. And if you forget which ones you can and can't have, now you know how to reason about it quickly and remember. Yes? OK.

So this is DFS. Are we all happy with DFS? Let's talk about topological sorting then. Because it's really useful. It's one of the few algorithms that is really useful and that you might have to write yourself later. So suppose these are classes. So let's get back to this oriented thing and suppose these are classes. And the edges show prerequisites. So A is a prerequisite of B. Like say A is 601 and B is 6006. And you have to take 601 before you take 6006. Otherwise you will cry during programming assignments.

So what we want to do is these are all the classes you need to graduate. We need to come up with an order in which you can take them so that when you take a class, you took all the prerequisites. So you don't cry while you're taking that class. How do we do that? And let's use this graph as an example.

AUDIENCE: Use a directed graph that's acyclic?

PROFESSOR: OK. Is this graph acyclic?

AUDIENCE: No.

PROFESSOR: OK. So what then?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So if the graph has a cycle, you can't solve the problem. It has no solution.

AUDIENCE: It's like those cardboard boxes that [INAUDIBLE]. It's really annoying.

PROFESSOR: I don't know the cardboard box problem.

AUDIENCE: Yeah. You know those cardboard boxes with the four flaps when you close them, and they close like-- Yeah. Those are so annoying. That's just what this reminds me of. That's all. Continue.

PROFESSOR: There's a cycle there, so you can't just put them in an order. Right? You have to violate the repeating variance somehow. And you do that by twisting the edges. So if we had classes A, B, C, you cannot take them in any order. Right? If you take A first, you would need C. If you take B first, you would need A. If you take C first, you would need B. So you can't start with any of them.

So if you have a cycle in a graph, it's not a dependency graph. You can't compute dependencies. So topological sorts only works on cyclic graphs. So acyclic graph. And what else? So a dependency graph is a special kind of graph. And I'm looking for three fancy words. You already have two. So we have acyclic and we have graph. Directed.

So you have to have a directed, so that you know which class goes first. Needs to be acyclic, otherwise there's no solution. Needs to be a graph, because that's what we're talking about. So the shortening for this is a DAG.

OK. How do you compute an order? So say we remove that edge. How do we compute an order?

AUDIENCE: You run DFS and then print out the reverse.

PROFESSOR: And then print out the reverse?

AUDIENCE: Of your final output.

PROFESSOR: Which is?

AUDIENCE: Your final output? You want me to do it on a graph?

PROFESSOR: You can tell me how to change the codes, too. So I'm not sure what-- you said print the reverse of something, but what's that thing?

AUDIENCE: So I go down a path, right, until you've hit all the nodes, then basically you start from the last node you visited and print out the reverse of that. Go back to the first node.

PROFESSOR: OK. So you're saying I go A, B, C, D, E, and then I print E, D, C, B, A?

AUDIENCE: No. Because that wouldn't hit all the nodes.

PROFESSOR: OK. He's on the right track, by the way, so that's why we're--

AUDIENCE: You go A, B, C, D, E, F, G, and you print G, F, E, D, C, B, A.

PROFESSOR: OK so we print them in the reverse. So you're printing them in the order in which you're done with them after DFS? No. You're printing them in the reverse order of the order which you visited them? So you visited A, B, C, D, E, F, G, so you're saying print G, F, E, D, C, B, A.

AUDIENCE: Yeah. Because in lecture wasn't it flipped, so if an arrow points from A to B then A depends on B. And we're looking at the other way around. In lecture, based on the graph you gave, if the arrow goes from A to B, then A depends on B.

PROFESSOR: If the arrow goes from A to B, then what?

AUDIENCE: If you're saying A is a prerequisite for B and G, then we just sweep from left to right and say, I have to do this. And then those two.

PROFESSOR: No. They have to--

AUDIENCE: It's the order that-- because when you're doing DFS, you recurse. It's the order that they finished the recurse.

PROFESSOR: Yeah. You have the right answer. I'm just trying to build out the annotation for it. I'm pretty sure this is the right way. I can look at this, but I can't promise you that this is the right way because I coded it and it works. So no. This is the right way. Yes. This is the right way to represent them. So forward edge means B depends on A.

OK. So let's figure out how we do this. Let's do this in pseudo-code and then build the intuition for it. So you said reverse of the finishing times. So let's build a list that has all the nodes in the order of their finishing times. So let's build a list finished that is empty at first. Then I'm going to add all the nodes in the list as I'm done with them. And then I'll reverse the list.

Where do I add nodes to the list?

AUDIENCE: In the check. Line 2.75.

PROFESSOR: Not quite. So line 4. So I'm adding them in the order in which I'm done with them. So when I'm about to leave a node completely, I'll add it to that list. Which node? What's the name?

AUDIENCE: V.

PROFESSOR: OK. And then where can I reverse the list?

AUDIENCE: Line 4 of the first--

PROFESSOR: OK. How would I do this? r dot finished dot reverse. Like this?

AUDIENCE: I think that's OK in Python.

PROFESSOR: I think so, too. So this will give me a topological sort. Let's figure out why this works intuitively. Yes? So while we're building the topological sort, while we're building the inverse of the final list. So the first thing that we put in the list is the last class we're going to take. So as I go forward in this graph, my VFS is going to go A, B, C, D, E. There's nothing after E. And it's done with E.

So this means there is no class that depends on E. Otherwise, DFS would keep recursing. So E is the last class I take. If I take this last, there's definitely no dependencies that I'm violating. All right? So it's safe to start with E.

Now I'm out of E, I'm back to D. I go from D to F, I print that. Let's not worry about this for a little bit. And let's go back to D. When DFS comes back, it's going to print D. So I know that by the time DFS is out, I printed all the classes that depend on D. Right?

So when I'm at D, whenever I have forward edges that I haven't visited yet, I will call DFS visit on them. DFS visit returns before I can get out of D. So all the edges that depend on D have been printed. Sorry. All the nodes that depend on D have been printed. So when I get out of D, I know that all the nodes that depend on D have been printed, so it's safe to print D. This is the intuition behind topological sort.

So you can build sort of an induction proof based on this. So whenever I'm here, I assume that all the nodes that I have forward edges to are somewhere in my results. So I can include my nodes. So this means that whenever you put a node here, all the nodes that have forward edges to this node have already been output. So this means that no dependency relationships are going to be violated.

OK. Let's keep building this and see the result. So D, E, F, we get out of B. We get out of C. What do we do here? Print C, right? What classes depend on C? D, E, and F. I had a tree edge to D. I had DFS visit here, so I know that all the classes that depend on C because they depend on D have been output. And then I have a forward edge on F.

So I didn't recurse from C to F, but I know that has been covered somewhere. Right? Forward edge means that I've already seen it in DFS, and that I've already returned from it. So it has already been printed. So I'm going to write this.

Now I get out of C. I get out of B. I go into G. I get out of B, and I go into A. And I go out of A, and I print it. OK. So I have tree edges that I can handle. So these are all tree edges. And the reason that topological sorting works on tree edges is that I call DFS visit on the tree edge, and I know it returns by the time I return. So I know that whatever's underneath that tree has already been printed.

I have forward edges that will just take me forward in the DFS. So I know that by the time I return from a node, I've already returned from all the nodes that I have forward edges to because they're lower in the tree. Right? Forward edges work like this. So by the time I'm out of C, I've definitely printed F. Now I have this cross edge from G to C.

A cross edge means that there is no direct relationship here, but I've already visited C. If there's no direct relationship, it means that for sure I'm done visiting C and I've returned so that I can get to G. So there's some common parent between C and G. I'm already done with C, and I've returned to that parent. And then I went to G.

So all the nodes that are pointed to my cross edges have also been printed in topological sort. OK. Now what about back edges? What if I had a back edge? What if I had this back edge between C and A? What happens then?

If I had a back edge, then that would break topological sort. Right? Because this is saying that hey, you should print A before you print C. But I know that I'm going to come out of C way before I have a chance to come out of A. So if I have a back edge, topological sort doesn't work. When do I have a back edge? When I have a cycle.

So this is why I don't care about back edges. Back edges [INAUDIBLE] cycles. So back edge means I have a path forward. And I have a path backward. That does a cycle. OK. There's an awful lot of silence here. Does everything makes sense or?

OK. Any questions? Nope? Everyone's happy? So you don't need to reason about this formally. Just remember the intuition that the reason we're printing them in this order is the first thing you print is the last class you're going to take. Because for sure there are no dependencies left on it, otherwise DFS would keep recursing. And then there's that recursive structure that makes this work. OK. Cool.