PROFESSOR: All right, let's get started. We return today to graph search. Last time we saw breadth-first search, today we're going to do depth-first search. It's a simple algorithm, but you can do lots of cool things with it. And that's what I'll spend most of today on, in particular, telling whether your graph has a cycle, and something called topological sort.

As usual, basically in all graph algorithms in this class, the input, the way the graph is specified is as an adjacency list, or I guess adjacency list plural. So you have a bunch of lists, each one says for each vertex, what are the vertices I'm connected to? What are the vertices I can get to in one step via an edge? So that's our input and our goal, in general, with graph search is to explore the graph. In particular, the kind of exploration we're going to be doing today is to visit all the vertices, in some order, and visit each vertex only once.

So the way we did breadth-first search, breadth-first search was really good. It explored things layer by layer, and that was nice because it gave us shortest paths, it gave us the fastest way to get to everywhere, from a particular source, vertex s. But if you can't get from s to your vertex, than the shortest way to get there is infinity, there's no way to get there.

And BFS is good for detecting that, it can tell you which vertices are unreachable from s. DFS can do that as well, but it's often used to explore the whole graph, not just the part reachable from s, and so we're going to see how to do that today. This trick could be used for be BFS or for DFS, but we're going to do it here for DFS, because that's more common, let's say.

So DFS. So depth-first search is kind of like how you solve a maze. Like, the other weekend I was at the big corn maze in central Massachusetts, and it's easy to get lost in there, in particular, because I didn't bring any bread crumbs. The proper way to solve a maze, if you're in there and all you can do is see which way to go next and then walk a little bit to the next junction, and then you have to keep making decisions.

Unless you have a really good memory, which I do not, teaching staff can attest to that, then an easy way to do it is to leave bread crumbs behind, say, this is the last way I went from this node, so that when I reach a deadend, I have to turn around and backtrack. I reach a breadcrumb that say, oh, last time you went this way, next time you should go this way, and in particular, keep track at each node, which of the edges have I visited, which ones are still left to visit. And this can be done very easily on a computer using recursion.

So high-level description is we're going to just recursively explore the graph, backtracking as necessary, kind of like how you solve a maze. In fact, when I was seven years old, one of the first computer programs I wrote was for solving a maze. I didn't know it was depth-first search at the time, but now I know. It was so much harder doing algorithms when I didn't know what they were.

Anyway, I'm going to write some code for depth-first search, it is super simple code, the simplest graph algorithm. It's four lines.

That's it. I'm going to write a little bit of code after this, but this is basic depth-first search. This will visit all the vertices reachable from a given source, vertex s. So we're given the adjacency list. I don't know why I put v here, you could erase it, it's not necessary. And all we do is, we have our vertex b, sorry, we have our vertex s. We look at all of the outgoing edges from s. For each one, we'll call it v, we check, have I visited this vertex already? A place where we need to be careful is to not repeat vertices. We need to do this in BFS as well.

So, the way we're going to do that is by setting the parent of a node, we'll see what that actually means later. But for now, it's just, are you in the parent structure or not? This is initially, we've seen s, so we give it a parent of nothing, but it exists in this dictionary. If the vertex b that we're looking at is not in our dictionary, we haven't seen it yet, we mark it as seen by setting its parent to s, and then we recursively visit it.

That's it. Super simple, just recurse. Sort of the magical part is the preventing yourself from repeating. As you explore the graph, if you reach something you've already seen before you just skip it again. So you only visit every vertex once, at most once. This will not visit the entire graph, it will only visit the vertices reachable from s.

The next part of the code I'd like to give you is for visiting all the vertices, and in the textbook this is called the DFS, whereas this is just called DFS visit, that's sort of the recursive part, and this is sort of a top level algorithm. Here we are going to use the set of vertices, b, and here we're just going to iterate over the s's. So it looks almost the same, but what we're iterating over is different. Here we're iterating over the outgoing edges from s, here were iterating over the choices of s.

So the idea here is we don't really know where to start our search. If it's a disconnected graph or not a strongly connected graph, we might have to start our search multiple times. This DFS algorithm is finding all the possible places you might start the search and trying them all. So it's like, OK, let's try the first vertex. If that hasn't been visited, which initially nothing's been visited, then visit it, recursively, everything reachable from s. Then you go on to the second vertex.

Now, you may have already visited it, then you skip it. Third vertex, maybe you visited it already. Third, fourth vertex, keep going, until you find some vertex you haven't visited at all. And then you recursively visit everything reachable from it, and you repeat. This will find all the different clusters, all the different strongly connected components of your graph. Most of the work is being done by this recursion, but then there's this top level, just to make sure that all the vertices get visited.

Let's do a little example, so this is super clear, and then it will also let me do something called edge classification. Once we see every edge in the graph gets visited by DFS in one way or another, and it's really helpful to think about the different ways they can be visited. So here's a graph. I think its a similar to one from last class.

It's not strongly connected, I don't think, so you can't get from these vertices to c. You can get from c to everywhere, it looks like, but not strongly connected. And we're going to run DFS, and I think, basically in alphabetical order is how we're imagining-- these vertices have to be ordered somehow, we don't really care how, but for sake of example I care.

So we're going to start with a, that's the first vertex in here. We're going to recursively visit everything reachable from a, so we enter here with s equals a. So I'll mark this s1, to be the first value of s at this level. So we consider-- I'm going to check the order here-- first edge we look at, there's two outgoing edges, let's say we look at this one first. We look at b, b has not been visited yet, has no parent pointer. This one has a parent pointer of 0. B we're going to give a parent pointer of a, that's here. Then we recursively visit everything for b.

So we look at all the outgoing edges from b, there's only one. So we visit this edge. for b to e. e has not been visited, so we set as parent pointer to b, an now we recursively visit e. e has only one outgoing edge, so we look at it, over here to d. d has not been visited, so we set a parent pointer to e, and we look at all the outgoing edges from d. d has one outgoing edge, which is to b. b has already been visited, so we skip that one, nothing to do. That's the else case of this if, so we do nothing in the else case, we just go to the next edge. But there's no next edge for d, so we're done.

So this algorithm returns to the next level up. Next level up was e, we were iterating over the outgoing edges from e. But there was only one, so we're done, so e finishes. Then we backtrack to b, which is always going back along the parent pointer, but it's also just in the recursion. We know where to go back to. We were going over the outgoing edges from b, there's only one, we're done. So we go back to a. We only looked at one outgoing edge from a. There's another outgoing edge, which is this one, but we've already visited d, so we skip over that one, too, so we're done recursively visiting everything reachable from a.

Now we go back to this loop, the outer loop. So we did a, next we look at b, we say, oh b has been visited, we don't need to do anything from there. Then we go to c, c hasn't been visited so we're going to loop from c, and so this is our second choice of s in this recursion, or in this outer loop. And so we look at the outgoing edges from s2, let me match the order in the notes. Let's say first we go to f. f has not been visited, so we set its parent pointer to c.

Then we look at all the outgoing edges from f. There's one outgoing edge from f, it goes to f. I guess I shouldn't really bold this, sorry. I'll say what the bold edges mean in a moment. This is just a regular edge. We follow the edge from f to f. We see, oh, f has already been visited, it already has a parent pointer, so there's no point going down there. We're done with f, that's the only outgoing edge. We go back to c, there's one other outgoing edge, but it leads to a vertex we've already visited, namely e, and so we're done with visiting everything reachable from c. We didn't visit everything reachable from c, because some of it was already visited from a.

Then we go back to the outer loop, say, OK, what about d? D has been visited, what about e? E's been visited, what about f? F's been visited. So we're visiting these vertices again, but should only be twice in total, and in the end we visit all the vertices, and, in a certain sense, all the edges as well.

Let's talk about running time. What do you think the running time of this algorithm is? Anyone? Time to wake up.

AUDIENCE: Upper bound?

PROFESSOR: Upper bound, sure.

AUDIENCE: V?

PROFESSOR: V?

AUDIENCE: [INAUDIBLE].

PROFESSOR: V is a little bit optimistic, plus e, good, collaborative effort. It's linear time, just like BFS. This is what we call linear time, because this is the size of the input. It's theta V plus E for the whole thing. The size of the input was v plus e. We needed v slots in an array, plus we needed e items in these linked lists, one for each edge. We have to traverse that whole structure.

The reason it's order v plus e is-- first, as you were saying, you're visiting every vertex once in this outer loop, so not worrying about the recursion in DFS alone, so that's order b. Then have to worry about this recursion, but we know that whenever we call DFS visit on a vertex, that it did not have a parent before.

Right before we called DFS visit, we set its parent for the first time. Right before we called DFS visit on v here, we set as parent for the first time, because it wasn't set before. So DFS visit, and I'm going to just write of v, meaning the last argument here. It's called once, at most once, per vertex b. But it does not take constant time. This takes constant time per vertex, plus a recursive call. This thing, this takes constant time, but there's a for loop here. We have to pay for however many outgoing edges there are from b, that's the part you're missing.

And we pay length of adjacency of v for that vertex. So the total in addition to this v is going to be the order, sum overall vertices, v in capital V, of length of the adjacency, list for v, which is E. This is the handshaking lemma from last time. It's twice e for undirected graphs, it's e for directed graphs. I've drawn directed graphs here, it's a little more interesting.

OK, so it's linear time, just like the BFS, so you could say, who cares, but DFS offers a lot of different properties than BFS. They each have their niche. BFS is great for shortest paths. You want to know the fastest way to solve the Rubik's cube, BFS will find it. You want to find the fastest way to solve the Rubik's cube, DFS will not find it. It's not following shortest paths here. Going from a to d, we use the path of length 3, that's the bold edges. We could have gone directly from a to d, so it's a different kind of search, but sort of the inverse. But it's extremely useful, in particular, in the way that it classifies edges.

So let me talk about edge classification. You can check every edge in this graph gets visited. In a directed graph every edge gets visited once, in an undirected graph, every edge gets visited twice, once from each side. And when you visit that edge, there's sort of different categories of what could happen to it. Maybe the edge led to something unvisited, when you went there. We call those tree edges. That's what the parent pointers are specifying and all the bold edges here are called three edges. This is when we visit a new vertex via that edge.

So we look at the other side of the edge, we discover a new vertex. Those are what we call tree edges, it turns out they form a tree, a directed tree. That's a lemma you can prove. You can see it here. We just have a path, actually a forest would be more accurate. We have a path abed, and we have an edge cf, but, in general, it's a forest. So for example, if there was another thing coming from e here, let's modify my graph, we would, at some point, visit that edge and say, oh, here's a new way to go, and now that bold structure forms an actual tree. These are called tree edges, you can call them forest edges if you feel like it.

There are other edges in there, the nonbold edges, and the textbook distinguishes three types, three types? Three types, so many types. They are forward edges, backward edges, and cross edges. Some of these are more useful to distinguish than others, but it doesn't hurt to have them all. So, for example, this edge I'm going to call a forward edge, just write f, that's unambiguous, because it goes, in some sense, forward along the tree. It goes from the root of this tree to a descendant. There is a path in the tree from a to d, so we call it a forward edge.

By contrast, this edge I'm going to call a backward edge, because it goes from a node in the tree to an ancestor in the trees. If you think of parents, I can go from d to its parent to its parent, and that's where the edge goes, so that's a backward edge-- double check I got these not reversed, yeah, that's right. Forward edge because I could go from d to its parent to its parent to its parent and the edge went the other way, that's a forward edge. So forward edge goes from a node to a descendant in the tree. Backward edge goes from a node to an ancestor in the tree. And when I say, tree, I mean forest. And then all the other edges are cross edges.

So I guess, here, this is a cross edge. In this case, it goes from one tree to another, doesn't have to go between different trees. For example, let's say I'm visiting d, then I go back to e, I visit g, or there could be this edge. If this edge existed, it would be a cross edge, because g and d are not ancestor related, neither one is an ancestor of the other, they are siblings actually. So there's, in general, there's going to be some subtree over here, some subtree over here, and this is a cross edge between two different subtrees. This cross edge is between two, sort of, non ancestor related, I think is the shortest way to write this, subtrees or nodes.

A little puzzle for you, well, I guess the first question is, how do you compute this structure? How do you compute which edges are which? This is not hard, although I haven't written it in the code here. You can check the textbook for one way to do it. The parent structure tells you which edges are tree edges. So that part we have done.

Every parent pointer corresponds to the reverse of a tree edge, so at the same time you could mark that edge a tree edge, and you'd know which edges are tree edges and which edges are nontree edges. If you want to know which are forward, which are backward, which are cross edges, the key thing you need to know is, well, in particular, for backward edges, one way to compute them is to mark which nodes you are currently exploring.

So when we do a DFS visit on a node, we could say at the beginning here, basically, we're starting to visit s, say, start s, and then at the end of this for loop, we write, we're finished with s. And you could mark that in the s structure. You could say s dot in process is true up here, s dot in process equals false down here. Keep track of which nodes are currently in the recursion stack, just by marking them and unmarking them at the beginning and the end. Then we'll know, if we follow an edge and it's an edge to somebody who's already in the stack, then it's a backward edge, because that's-- everyone in the stack is an ancestor from our current node.

Detecting forward edges, it's a little trickier. Forward edges versus cross edges, any suggestions on an easy way to do that? I don't think I know an easy way to do that.

It can be done. The way the textbook does it is a little bit more sophisticated, in that when they start visiting a vertex, they record the time that it got visited. What's time? You could think of it as the clock on your computer, another way to do it is, every time you do a step in this algorithm, you increment a counter.

So every time anything happens, you increment a counter, and then you store the value of that counter here for s, that would be the start time for s, you store the finish time for s down here, and then this gives you, this tells you when a node was visited, and you can use that to compute when an edge is a forward edge and otherwise it's a cross edge. It's not terribly exciting, though, so I'm not going to detail that. You can look at the textbook if you're interested.

But here's a fun puzzle. In an undirected graph, which of these edges can exist? We can have a vote, do some democratic mathematics. How many people think tree edges exist in undirected graphs? You, OK. Sarini does. That's a good sign.

How many people think forward edges exist in an undirected graph? A couple. How many people think backward edges exist in an undirected graph? Couple. How many people think cross edges exist in undirected graph? More people, OK. I think voting worked.

They all exist, no, that's not true. This one can exist and this one can exist. I actually wrote the wrong ones in my notes, so it's good to trick you, no, it's I made a mistake. It's very easy to get these mixed up and you can think about why this is true, maybe I'll draw some pictures to clarify.

This is something, you remember the-- there was BFS diagram, I talked a little bit about this last class. Tree edges better exist, those are the things you use to visit new vertices. So that always happens, undirected or otherwise. Forward edges, though, forward edge of would be, OK, I visited this, then I visited this. Those were tree edges. Then I backtrack and I follow an edge like this. This would be a forward edge. And in a directed graph that can happen. In an undirected graph, it can also happen, right? Oh, no, it can't, it can't. OK. So confusing. undirected graph, if you look like this, you start-- let's say this is s. You start here, and suppose we follow this edge. We get to here, then we follow this edge, we get to here. Then we will follow this edge in the other direction, and that's guaranteed to finish before we get back to s.

So, in order to be a forward edge, this one has to be visited after this one, from s, but in this scenario, if you follow this one first, you'll eventually get to this vertex and then you will come back, and then that will be classified as a backward edge in an undirected graph. So you can never have forward edges in an undirected graph. But I have a backward edge here, that would suggest I can have backward edges here, and no cross edges. Well, democracy did not work, I was swayed by the popular vote. So I claim, apparently, cross edges do not exist.

Let's try to draw this. So a cross edge typical scenario would be either here, you follow this edge, you backtrack, you follow another edge, and then you discover there's was an edge back to some other subtree that you've already visited. That can happen in an undirected graph. For the same reason, if I follow this one first, and this edge exists undirected, then I will go down that way. So it will be actually tree edge, not a cross edge. OK, phew. That means my notes were correct. I was surprised, because they were copied from the textbook, uncorrect my correction. Good.

So what? Why do I care about these edge classifications? I claim they're super handy for two problems, cycle detection, which is pretty intuitive problem. Does my graph have any cycles? In the directed case, this is particularly interesting. I want to know, does a graph have any directed cycles? And another problem called topological sort, which we will get to. So let's start with cycle detection. This is actually a warmup for topological sort. So does my graph have any cycles? G has a cycle, I claim. This happens, if and only if, G has a back edge, or let's say, a depth-first search of that graph has a back edge. So it doesn't matter where I start from or how this algorithm-- I run this top level DFS algorithm, explore the whole graph, because I want to know in the whole graph is there a cycle? I claim, if there's a back edge, then there's a cycle.

So it all comes down to back edges. This will work for both directed and undirected graphs. Detecting cycles is pretty easy in undirected graphs. It's a little more subtle with directed graphs, because you have to worry about the edge directions. So let's prove this. We haven't done a serious proof in a while, so this is still a pretty easy one, let's think about it. What do you think is the easier direction to prove here, left or right? To more democracy. How many people think left is easy? A couple. How many people think right is easy? A whole bunch more. I disagree with you. I guess it depends what you consider easy.

Let me show you how easy left is. Left is, I have a back edge, I want to claim there's a cycle. What is the back edge look like? Well, it's an edge to an ancestor in the tree. If this node is a descendant of this node and this node is an ancestor of this node, that's saying there are tree edges, there's a path, a tree path, that connects one to the other. So these are tree edges, because this is supposed to be an ancestor, and this is supposed to be a descendant. And that's the definition of a back edge. Do you see a cycle? I see a cycle. This is a cycle, directed cycle. So if there's a back edge, by definition, it makes a cycle. Now, it's harder to say if I have 10 back edges, how many cycles are there? Could be many. But if there's a back edge, there's definitely at least one cycle.

The other direction is also not too hard, but I would hesitate to call it easy. Any suggestions if, I know there is a cycle, how do I prove that there's a back edge somewhere? Think about that, let me draw a cycle. There's a length k cycle. Where do you think, which of these edges do you think is going to be a back edge? Let's hope it's one of these edges. Sorry?

AUDIENCE: Vk to v zero.

PROFESSOR: Vk to v zero. That's a good idea, maybe this is a back edge. Of course, this is symmetric, why that edge? I labeled it in a suggestive way, but I need to say something before I know actually which edge is going to be the back edge.

AUDIENCE: You have to say you start to v zero?

PROFESSOR: Start at v zero. If I started a search of v zero, that looks good, because the search is kind of going to go in this direction. vk will maybe be the last thing to be visited, that's not actually true. Could be there's an edge directly from v zero to vk, but intuitively vk will kind of later, and then when this edge gets visited, this will be an ancestor and it will be a back edge. Of course, we may not start a search here, so calling it the start of the search is not quite right, a little different.

AUDIENCE: First vertex that gets hit [INAUDIBLE].

PROFESSOR: First vertex that gets hit, good. I'm going to start the numbering , v zero, let's assume v 0 is the first vertex in the cycle, visited by the depth-first search. Together, if you want some pillows if you like them, especially convenient that they're in front. So right, if it's not v zero, say v3 was the first one visited. We will just change the labeling, so that's v zero, that's v1, that's v, and so on. So set this labeling, so that v0 first one, first vertex that gets visited. Then, I claim that-- let me just write the claim first. This edge vkv0 will be a back edge. We'll just say, is back edge. And I would say this is not obvious, be a little careful.

We have to somehow exploit the depth-first nature of DFS, the fact that it goes deep-- it goes as deep as it can before backtracking. If you think about it, we're starting, at this point we are starting a search relative to this cycle. No one has been visited, except v zero just got visited, has a parent pointer off somewhere else.

What do we do next? Well, we visit all the outgoing edges from v zero, there might be many of them. it could be edge from v zero to v1, it could an edge from v zero to v3, it could be an edge from v zero to something else. We don't know which one's going to happen first. But the one thing I can claim is that v1 will be visited before we finish visiting v zero. From v zero, we might go somewhere else, we might go somewhere else that might eventually lead to v1 by some other route, but in particular, we look at that edge from v zero to v1. And so, at some point, we're searching, we're visiting all the things reachable from v zero, that includes v1, and that will happen, we will touch v1 for the first time, because it hasn't been touched yet. We will visit it before we finish visiting v zero.

The same goes actually for all of v i's, because they're all reachable from v zero. You can prove this by induction. You'll have to visit v1 before you finish visiting v zero. You'll have to visit v2 before you finish visiting v1, although you might actually visit v2 before v1. You would definitely finish, you'll finished v2 before you finish v1, and so on. So vi will be visited before you finish vi minus 1, but in particular, what we care about is that vk is visited before we finish v zero. And it will be entirely visited. We will finish visiting vk before we finish visiting v zero. We will start decay vk after we start to v zero, because v zero is first. So the order is going to look like, start v zero, at some point we will start vk. Then we'll finish vk, then we'll finish v zero.

This is something the textbook likes to call, and I like to call, balanced parentheses. You can think of it as, we start v zero, then we start vk, then we finish vk, then we finish v zero. And these match up and their balanced. Depth-first search always looks like that, because once you start a vertex, you keep chugging until you visited all the things reachable from it. Then you finish it. You won't finish v zero before you finish vk, because it's part of the recursion. You can't return at a higher level before you return at the lower levels.

So we've just argued that the order is like this, because v zero was first, so vk starts after v zero, and also we're going to finish vk before we finish v zero, because it's reachable, and hasn't been visited before. So, in here, we consider vkv zero. When we consider that edge, it will be a back edge. Why? Because v zero is currently on the recursion stack, and so you will have marked v zero as currently in process. So when you look at that edge, you see it's a back edge, it's an edge to your ancestor. That's the proof. Any questions about that? It's pretty easy once you set up the starting point, which is look at the first time you visit the cycle, than just think about how you walk around the cycle.

There's lots of ways you might walk around the cycle, but it's guaranteed you'll visit vk at some point, then you'll look at the edge. v0 is still in the stack, so it's a back edge. And so this proves that having a cycle is equivalent to having a back edge. This gives you an easy linear time algorithm to tell, does my graph have a cycle? And if it does, it's actually easy to find one, because we find a back edge, just follow the tree edges, and you get your cycle. So if someone gives you a graph and say, hey, I think this is acyclic, you can very quickly say, no, it's not, here's a cycle, or say, yeah, I agree, no back edges, I only have tree, forward, and cross edges.

OK, that was application 1. Application 2 is topological sort, which we're going to think about in the setting of a problem called job scheduling. So job scheduling, we are given a directed acyclic graph. I want to order the vertices so that all edges point from lower order to high order.

Directed acyclic graph is called a DAG, you should know that from 042. And maybe I'll draw one for kicks.

Now, I've drawn the graph so all the edges go left to right, so you can see that there's no cycles here, but generally you'd run DFS and you'd detect there's no cycles. And now, imagine these vertices represent things you need to do. The textbook has a funny example where you're getting dressed, so you have these constraints that say, well, I've got to put my socks on before put my shoes on. And then I've got to put my underwear on before I put my pants on, and all these kinds of things.

You would code that as a directed acyclic graph. You hope there's no cycles, because then you can't get dressed. And there's some things, like, well, I could put my glasses on whenever, although actually I should put my glasses on before I do anything else, otherwise there's problems. I don't know, you could put your watch on at any time, unless you need to know what time is. So there's some disconnected parts, whatever. There's some unrelated things, like, I don't care the order between my shirt and my pants or whatever, some things aren't constrained.

What you'd like to do is choose an actual order to do things. Say you're a sequential being, you can only do one thing at a time, so I want to compute a total order. First I'll do g, then I'll do a, then I can do h, because I've done both of the predecessors. Then I can't do be, because I haven't done d, so maybe I'll do d first, and then b, and than e, then c, then f, then i. That would be a valid order, because all edges point from an earlier number to a later number. So that's the goal. And these are real job scheduling problems that come up, you'll see more applications in your problem set.

How do we do this? Well, at this point we have two algorithms, and I pretty much revealed it is DFS. DFS will do this. It's a topological sort, is what this algorithm is usually called. Topological sort because you're given a graph, which you could think of as a topology. You want to sort it, in a certain sense. It's not like sorting numbers, it's sorting vertices in a graph, so, hence, topological sort.

That's the name of the algorithm. And it's run DFS, and output the reverse of the finishing times of vertices. so this is another application where you really want to visit all the vertices in the graph, so we use this top level DFS, so everybody gets visited. And there are these finishing times, so every time I finish a vertex, I could add it to a list. Say OK, that one was finished next, than this one is finished, than this one's finished. I take that order and I reverse it. That will be a topological order. Why? Who knows. Let's prove it.

We've actually done pretty much the hard work, which is to say-- we're assuming our graph has no cycles, so that tells us by this cycle detection that there are no back edges. Back edges are kind of the annoying part. Now they don't exist here. So all the edges are tree edges, forward edges, and cross edges, and we use that to prove the theorem. So we want to prove that all the edges point from an earlier number to a later number.

So what that means is for an edge, uv, we want to show that v finishes before u. That's the reverse, because what we're taking is the reverse of the finishing order. So edge uv, I want to make sure v finishes first, so that u will be ordered first. Well, there are two cases. Case 1 is that u starts before v. Case 2 is that he v before u. At some point they start, because we visit the whole graph. This top loop guarantees that. So consider what order we visit them first, at the beginning, and then we'll think about how they finish.

Well, this case is kind of something we've seen before. We visit u, we have not yet visited v, but v is reachable from u, so maybe via this edge, or maybe via some other path, we will eventually visit v in the recursion for u. So before u finishes, we will visit v, visit v before u finishes. That sentence is just like this sentence, so same kind of argument. We won't go into detail, because we already did that several times. So that means we'll visit v, we will completely visit v, we will finish v before we finish u and that's what we wanted to prove. So in that case is good.

The other cases is that v starts before u. Here, you might get slightly worried. So we have an edge, uv, still, same direction. But now we start at v, u has not yet been visited. Well, now we worry that we visit u. If we visit u, we're going to finish u before we finish v, but we want it to be the other way around. Why can't that happen?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Because there's a back edge somewhere here. In particular, the graph would have to be cyclic. This is a cycle, so this can't happen, a contradiction. So v will finish before we visit u at all. So v will still finish first, because we don't even touch u, because there's no cycles. So that's actually the proof that topological sort gives you a valid job schedule, and it's kind of-- there are even more things you can do with DFS. We'll see some in recitations, more in the textbook. But simple algorithm, can do a lot of nifty things with it, very fast, linear time.