PROFESSOR: OK so we're going to test it. Here's testOne. It just says for name and range 10, I'm going to build some nodes, put in a few edges, and then I'm going to print the graph. And all I really want to show you here, is that if we run it for digraph or we run it for graph, we'll get something different.

Yes, I'm happy to save the source. Oh, and now the syntax. It was valid last time I looked, what have we done wrong here? I edit something badly? Sort of looks valid to me, doesn't it? We'll retype it. Yes, we'll save. Nope. Well, it's one of those days, isn't it?

All right this is a good debugging exercise for us. Let's think about how we go and find this. I'm sure you've all had this sort of problem. Well the first thing to do is I think I'll just comment out all of this. Let's see if I still get an error. I do. All right, well that suggests that, that wasn't the problem, so I can put it back.

Now it shows a problem all the way down there. So let's see what's going on here. Or maybe, I'll just skip this, but doubtless I'll get in trouble if I do. So let's see. I must have just made a sloppy editing error somewhere this morning in commenting things out for the lecture. Well, I think what we're going to do, for the moment, is move on and hope it goes away. Now that seems silly.

Sorry about this everybody. People should look with me, and someone may see it more quickly than I can. In fact, I'm hoping someone sees it more quickly than I can. We've now got a microphone, and the embarrassment of a code with syntax error, [UNINTELLIGIBLE].

MALE SPEAKER: You've got a sign. For some reason it wasn't on the schedule, so--

PROFESSOR: Well just because we've been teaching at 10 o'clock.

MALE SPEAKER: Yeah, I know. Yeah. [UNINTELLIGIBLE].

PROFESSOR: February. There's no reason to suspect that we would teach at 10 o'clock today.

MALE SPEAKER: I actually looked. I double checked.

PROFESSOR: Well this is embarrassing, folks. And I wish one of you would bail me out by telling me where my syntax error is. Well random looks OK, right? Node looks OK. And it gets more complicated.

MALE SPEAKER: Hey professor, can you turn on the transmitter?

PROFESSOR: No.

[LAUGHTER]

PROFESSOR: OK, I'll turn on the transmitter. But I'm really focused on a different problem right now. Guys help! Where are my TAs? Why does it keep doing that to me? Maybe there's something just funny going on here. Pardon?

AUDIENCE: Restart IDLE.

PROFESSOR: Restarting IDLE? You think maybe that's the issue? We could try that. Ha! So it looks like IDLE was just in some ugly state. Let's hope. Yes, all right, so I didn't have a bug. It was just IDLE had a bug. All right, phew. But we did squander ten minutes. Oh well.

So we have the graph, and you can see when we look at a graph, we have a node from 1 to 2, an edge from 1 to 2, from 1 to 1, et cetera. And that's the digraph. When we look at the graph, we'll see that in fact we have a lot more nodes, because everything goes in both directions. But that's what we expected-- nothing very interesting. All I want you to do is notice the difference here between graphs and digraphs.

Now getting to the whole point, once we have this mechanism set up to think about graphs, we can now think about interesting problems and formulate them as graph problems. And I want to list a few of the interesting problems, and then we'll look at how to solve some of them. So probably the most common graph problem that people solve, is called the shortest path problem.

We talked about this briefly last time. The notion here is for some pair of nodes, n1 and n2, we want to find the shortest sequence of edges that connects those two nodes. All right? So that's very straightforward. Then there is the shortest weighted path, where instead of trying to find the shortest sequence of edges, we want to find the smallest total weight.

So it may be that we traverse a few extra edges, but since they have a shorter weights, we end up getting a shorter path. So we might indirect to do the shortest path. This is probably the more common problem. So for example, that's the problem that Google Maps solves, when you ask it to give you driving directions.

And you'll notice when you use something like Google Maps or MapQuest, you can tell it to minimize the time, in which case maybe it will route you on a freeway, where you can drive at 80 miles an hour, even though you drive a few extra miles. Or you can tell it to minimize the distance in which case it may take you on these crummy little surface roads where you have to drive slowly, but you'll cover fewer miles and use less gas. So you get to tell it which set of weights you care about, and then it finds you the shortest path, given those weights. We'll come back to this since we're going to look at some code to implement it.

Another slightly more complicated problem to understand is finding cliques. So to find a clique, we're looking to find a set of nodes, such that there exists a path connecting each node in the set. So you can think of this as similar to, say a social clique-- who your friends are. It's a group of nodes or group of people that somehow can get to each other. It's not saying you can't get outside the clique. But it is guaranteeing that from any member of the clique, you can reach any other member of the clique. And so well, we'll look at some examples of where finding a clique is useful.

And the final kind of problem I want to mention is the minimum cut problem-- often abbreviated mincut. So the problem here, is given a graph, and given two sets of nodes, you want to find the minimum number of edges such that if those edges are removed, the two sets are disconnected. i.e. you can't get from a member of one set to a member of the other set.

This is often a question that gets asked. For example, imagine that you were the government of Syria and you want to ensure that nobody can post a video on YouTube. You would take the set of nodes in Syria, and you would take the set of nodes probably outside Syria, and ask what's the minimum number of communication links you'd have to cut to ensure that you can't get from a node in Syria to a node outside Syria.

People who do things like plan power lines worry about that. They want to say what's the minimum number of links such that if they're cut, you can't get any electricity from this power plant to say, this city. And they'll typically design their network with redundancy in it, so that the mincut is not too small. And so people frequently are worried about mincut problems, and trying to see what that is.

All right, now let's look at a couple of examples, in slightly more detail. So what we see here is a pictorial representation of a weighted graph generated by the Centers for Disease Control, CDC, in Atlanta in 2003 when they were studying an outbreak of tuberculosis in the United States-- a virulent and bad infectious disease. Each node, and you can see these little dots are the nodes, represents a person. And each node is labeled by a color, indicating whether the person has active tuberculosis, has tested positive for exposure, but doesn't have the disease, or tested negative for exposure, or not been tested.

So you'll remember when we looked last time at class node, and asked why did I bother creating a class for something so simple, it was because I said well maybe we would add extra properties to a node. So now in some sense these colors would be easy to add. So I could add to class node-- well I could attribute color, and call it red or blue or green-- or more likely an attribute saying TB state, which would indicate active not active, et cetera. The edges, which you can see here, represent connections among pairs of people.

What I didn't bother, you can't see on these pictures, is the edges are actually weighted. And the weights there are about how closely people are connected. And there are really only two weights I think they used: close, someone who say lives in your house or works in the same office, or casual, a neighbor you might have encountered, but you wouldn't expect to necessarily see them every day. So I've taken a fairly complicated set of information and represented it as a graph.

Now what are some of the interesting graph theoretic questions we might proceed to ask about this? So an important question they typically ask when diseases break out unexpectedly is, is there an index patient? The index patient is the patient who brought the disease into the community-- so somebody who visited some country, picked up tuberculosis, flew back to their neighborhood in the US and started spreading it.

How would we formulate that as a graph question? Again, quite simply. We would say does there exist a node such that node has TB, or maybe not even that, no, let's simplify it. You might say, "or tested positive" because maybe you can communicate it without having it-- has TB and is connected to every node with TB. Now this doesn't guarantee that the patient is an index patient. But if there is no such patient, no such node, then you know that there's not a single source of this disease in the community.

How would we change the graph to model it in a more detailed way, and remember this is all about modeling, so that we could ask a question or more precisely? Well we'd have to change to a more complex coloring scheme, if you will, in which we'd include the date of when somebody acquired the disease, or tested positive. And then we could ask those kinds of questions in a little bit more detail. But again once we've built the model, we can then go and ask a lot of interesting questions.

By the way the answer to this question, for this graph, is almost. There is an index patient that's connected to every node in the graph, except for the nodes in this black circle. They are not connected to any index patient. So the CDC actually did that analysis, and they reached that conclusion that there didn't seem to be. And then later, it came to light, in fact, that this particular graph is missing an edge.

Somebody had moved from neighborhood A to neighborhood B, and they had not kept track of that. And if they had, they would have discovered there was a link that's missing-- an edge that's missing from this graph-- which in fact would've connected everybody to the index patient. It was an interesting question. They only found that, because they were puzzled about this tiny little black circle out here, and started investigating all the people in the black circle, and discovered that one of them had moved from one place to another.

What's another question you might ask once you've built this model? Well suppose this is the current state of the world, and I want to reduce the spread of the disease, by vaccinating uninfected people so that they don't contract tuberculosis. But I have a minimum, it's expensive to do this, I only have so much vaccine. Who should get it? What's the graph theory problem that I would solve to address the question of what's the best way to allocate my limited supply a vaccine?

Exactly. I, by the way, have much better candy now. So I think that's where the minimum cut came from. Well, all right. It's better for eating. It's just worse for throwing. That's easier to throw. All right.

It's the minimum cut problem. I take the people who are already infected, view them as one set of nodes. I take the people who are not infected, and view them as another set of nodes, find the edges that I need to cut to separate them, and then vaccinated somebody on one side of the edge, so that they don't contract the disease. So again a nice, easily formalized, problem. All right, so that's an example.

Let's look at another example. Let's think about the shortest path problem here. And we'll do that by thinking about social networks. So I suspect that at least a few of you have used Facebook, and you have friends-- some of you more, than others. I see people laughing. This is someone who probably has two friends, and is said. I don't know, or 1,000 friends and is happy. Who knows-- I don't want to know please.

And I'm not going to tell you how many friends I have either.

But you might ask the question, suppose you wanted to reach Donald Trump -- erstwhile Republican, vice presidential candidate, or presidential candidate. Say is there a connection from you to Donald Trump? Do you have a friend, who has a friend, who has a friend, who is a friend with Donald Trump? Or for Barack Obama, or anyone else you'd ask. Well what's the shortest path? How many friends do you have to go through? This is what's called the six degrees of separation problem.

In the 1990s, the playwright John Guare published a play called *Six Degrees of Separation,* under the slightly dubious premise, that everybody in the world was connected to everybody else in the world with at most six hops. If you took all the people you knew, all the people they knew, et cetera, you could reach any person in the world in six phone calls-- say any person who has a phone. I don't know whether that's true, but this is the whole notion of a social network.

So if we wanted to look at that in Facebook, we could either assume that the friend relation is symmetric-- if I'm your friend, you're my friend, which it is. Or you could imagine a different model, in which it's asymmetric. If it's symmetric you have a graph. If it's asymmetric you have a directed graph. And then you just ask the question. What's the shortest path from you to whomever you care about? And you get that. You could imagine that Facebook already knows the answer to that question, but just won't tell you. But they'll sell it to somebody who has enough money.

All right, So how does Facebook solve this problem? They have a very simple piece of code, which we'll now look at which solves the shortest path. So let's go back. So here's a recursive implementation of shortest path. Comment this out while I'm in the neighborhood. It takes the graph, a start node and end node to print, and this extra argument call visited. We'll see why that's gets used.

And we'll think about the algorithm. This particular algorithm is what's called a depth first search algorithm. It's a recursive depth first search. We've seen these before, often abbreviated DFS. So if you think about having a graph of a bunch of nodes connected to one another, just for fun we'll say it does something like this.

What depth first search does is it starts at the source node for the shortest path, let's called it this one, it first visits one child, then visits all the children of those children. This one has no children. Visits this child, picks one of its children, visits all of its children-- let's say it had another one here-- and goes on until it's done. And then it back tracks, comes back and takes the next child. Then we have to be a little bit careful about the circle.

So to summarize it, first thing we have to say is the recursion ends, when start equals end. That is to say I've called it and I've asked is there a path from A to A, and the answer is yes, there is. I'm already there. Now you could argue, and in some formulations the answer is not necessarily, you'd say there's only a path if there's an edge from A to A. But I've chosen to make the simpler assertion that if you want to get to A, and you're already there, you're done. Kind of seems reasonable.

So then the recursive part, starts by choosing one child of the node you're currently at. And it keeps doing that until either it reaches a node with no children, or it reaches the node you're trying to get to, or, and here's an important part, it reaches a node it's already seen. And that's what visited is about. Because I want to make sure that when I explore this graph, I don't go from here to here to here to here to here to here ad nauseum, because I'm stuck in what's called a cycle. You have to avoid the cycles.

Once it's got to a node that has no children, if that's not the node it's trying to get to, it back tracks and takes the next child of the node it was at. And in that way, it systematically explores all possible paths, and along the way, it chooses the best one. So we can look at the code here.

I've just commented out something we'll look at later just as we try and instrument it to see how fast it's working. I've got a debugging statement just to say whether I'm going to print what I've been asked to do, in case it's not working. And then the real work starts. I get the original path is just the node we're starting at, if start is end, I stop. If I get to here, or say shortest equals none, I haven't found any paths yet. So there is no shortest path.

And then for node in the children of start, if I haven't already visited the node-- this is to avoid cycles-- I create a visited list that contains whatever used to contain plus the node. Notice that I'm creating a new list here, rather than mutating the old list. And that's because when I unravel my recursion, and back track to where I was, I don't want to have think I visited something I haven't visited, right? If I had only one list, and I mutated each time, then as I go down the recursion and back up the recursion, I'm always dealing with the same list. By getting a new list, I'm ensuring that I don't have that problem.

Then I say the new path is whatever the shortest path is, from the node I'm currently at to the desired end node. And I use the current set of visited nodes to indicate where I've already been at this part of the recursion. If the new path is none, well didn't find one, I continue. Otherwise, I found a path, and now I just want to check is it better, or worse, or the same, as the previous shortest path. And then I'm done. Very straightforward. The only really tricky part was making sure that I kept track of visited properly, and didn't get stuck in cycles.

OK let's run it. So here's testTwo -- builds the same kind of graph we've built before. And then it tries to find the shortest path. And I'm going to do it for the same input, essentially, the same at edge operations, but once when it's a graph and once when it's a digraph.

So you'll notice that it found two different answers. When it was a graph, it could get from 0 to 4 in essentially one hop. But when it was a digraph, it took longer. It had to go from 0 to two to 3 to 4. And that's not surprising, because the graph has more edges. And in fact, what we saw is that in the graph there was an edge from 4 to 0, but there was no such edge in the directed graph. So again you'll get, unsurprisingly, different answers-- but very straightforwardly.

Now let's try it on a bigger problem. So I've called this big test. And what this does, is rather than my sitting there and typing a bunch of at edge commands, it just generates edges at random. So I tell it whether I want it to be a graph or digraph, and then I give it the number of nodes I want, and the number of edges. And it just generates, at random, a graph in this case with 25 nodes and 200 edges. So let's see what happens here.

So it's printed out the graph, and now we're waiting a little bit. It will eventually finish, there. I can get from 0 to 4. It turns out there's a short path for this random graph from 0 to 14 to 4. It's not so surprising that there's a short path. Why is it not surprising that there's a pretty short path? It had a lot of edges, right? I had 200 edges in my graph. So things are pretty densely connected.

Why did it take so long? Well remember what it's doing is exploring all the possible paths from 0 to 4, in this case. This is very much like what we saw when we looked at the knapsack problem, right? Where, there when we looked at the recursive implementation, we saw that well all right, generating all possibilities, there were an exponential number of possibilities there in the number of items. Here, depending upon the number of nodes and the number of edges, it's also large, and in fact, exponential. We could explore a lot of different paths, but let's see what's going on when we explore those.

So what I'm going to do now, is go back to our small example. We'll run testTwo That was the small one we looked at. But I'm going to set to print onto true, and if you remember what that code did is they told us what each recursive call was, what the start node was and what the end node was. So it found the same shortest path. That's a good thing, 0 to 4. But how did it do that?

Well it first got called with the question of starting at 0 find me a path to 4. It visited the first child of 0, which was 1. It said, all right see if you can find a path from 1 to 4. It then backtracked and sort of asked the same question, can I get from 2 to 4? From 0 to 4? And then it said well I can get from 0 to 2, let me try 2 to 4, 3 to 4, 4 to 4, that's good. Get to 5 to 4, and then it tried to find 4 to 4 again. Here it tried to find 2 to 4 again. So what you can see, is as I do that backtracking, I'm solving the same problem multiple times. Why am I doing that? Because there may be multiple ways to get to the same node.

So if, for example, I looked at this graph, what we would see is I would try and let's say I want to get to here, just for the sake of argument, I'd first say can I get to here from here. I'd try this, then I'd solve here to here. And I'd do that. I'd also go from here to here to here, and then for the second time, I'd try and solve the problem here to here. Now here since it's only one connection, it's a short thing. But you can see if I have multiple ways to get to the same intermediate node, each time I get there I'm going to solve a problem I have already solved-- how to get from that intermediate node to the final destination.

So I'm doing work I've already done before. This is obviously troublesome. Nobody likes to solve a problem they've already solved before. So what do you think the solution is? How would you solve this sort of thing yourself? What would you do? Well what you'd-- yeah, thank you. This guy is hungry. Go ahead.

AUDIENCE: Some way of storing information that you've already looked at.

PROFESSOR: Exactly. What you try and do, is remember what you did before, and just look it up. This is a very common technique. It's called memoization. We use this to solve a lot of problems where you remember what the answer was, and rather than recalculating it, you just look it up. And that can, of course, be much faster. So it's a fancy way to say we're going to use a table look-up.

This concept of memoization is at the heart of a very important programming technique called dynamic programming. In the algorithms class that's taught in this room immediately following this class, they have spent at least four lectures on the topic of dynamic programming. But since you guys are much smarter than those guys taking that class, we're going to do it in about 20 minutes, in today and a little bit in the next lecture.

All right, so let's look at an example. We'll look at a solution. So I've taken the recursive implementation we had before, and rewritten it just a little bit, to call dp, dynamic programming shortest path. And the most important thing to notice is I've given yet another argument to the function, and that's the memo, which is initially an empty dictionary. The rest of the algorithm proceeds as before, except what happens here is when I want to get from a path, the first question I ask is I say new path is equal to the memo of node to end.

So when I get to one of these interior nodes, and I want to say what's the shortest path from here to here, the first question I ask is do I already know the answer? Is it already in my memo? If so, I just look it up, and I'm done. I found it, and I proceed as before. If it's not in the memo, well this look up will fail, and I'll enter the except clause, and I'll make a call again. So this is a very conventional way of using try, except as a control structure. Failing to find in the memo is not an error, it just means I haven't yet stored it away. And as I go, I'll build up the memo, and then I'm done.

So it's very simple. So I should ask the question. Does anyone need me to explain this again, or does it makes sense what we're doing here with a memo? OK, I'm assuming it makes sense. Let's test it.

And we'll first do a very simple test. We're just going to use the same little graph we used before. And I'm going to run shortest path, and dp_shortest path, and at least confirm that for one search I get the same answer. It's just fire testing it to make sure that it's not a complete disaster. And we do. We get 0234, 0234. So at least for one thing, it's the same thing.

Let's see about performance, because that's really what we got interested in. So we'll go back to our big test. And let's go back and for both of these, I'm going to uncomment, tracking this global variable, just keeping track of the number of calls, and we'll see whether we get a substantially different amount of recursion, in one versus the other. So it's built some random graph again. This is the non-dynamic programming one, which as we recall, takes a bit longer.

I probably should have said-- all right, so it's pretty big difference. They found the same path, 0, 2,3, 4. But you'll notice the straightforward depth first search took over 800,000 recursive calls, whereas the dynamic programming one took only an order of 2000-- a huge difference. If I ran it again, I might see a slightly smaller difference. I might even see a considerably larger difference. I've run this on some examples where the recursive search depth first took a million, and got through the dynamic programming in 50, 60. But what you can see is there's a huge improvement in going from one to the other.

Dynamic programming was invented in the 1950s by someone named Richard Bellman. Many a student has wasted a lot of time trying to understand why it's called dynamic programming. And you or I could invent lots of theories. Relatively recently, I found out why it was called dynamic programming, and this is a quote from Bellman. "It was an attempt to hide what I was doing from government sponsors. The fact that I was really doing mathematics was something not even a congressman could object to."

So he was doing this thing that was pretty evil, which was mathematics, which is what he thought this was-- the math of dynamic programming. And he just didn't want to admit it. So he made up a name out of nothing, and it fooled the government, and he got to do it.

Now why do I teach you dynamic programming? And we're going to talk a little bit more about it, the next lecture. It's because it is one of the most important algorithms that we know today. It's used over and over again to provide practical, efficient solutions to optimization problems that on their surface appear intractable. They appear exponential. It says there is no good way to solve it. In fact, if it has certain kinds of properties, it will always be amenable to solutions by dynamic programming, which will most of the time-- and I'll come back to the most of the time-- end up taking an exponential problem, and solving it really quickly, as we did here. I could have made this graph enormous, and dynamic programming would have given us a very fast solution to it.

So when can we use dynamic programming? Not all the time. We can use it on problems that exhibit two properties. The problem must have optimal substructure. What this means is that you can find a globally optimal solution by combining locally optimal solutions. So we can again see that with our graph problem, that we can combine the solutions from nodes at a distance from the root node to get the solution of getting there from the root node. If I know I can get from A to B, and I can find the optimal solution from B to C, then I can use that to find the optimal solution from A to C. So it has optimal substructure.

The other thing it has to have is overlapping sub-problems. And that's the thing I emphasized earlier-- that finding the optimal solution involves finding optimal solution to the same sub-problem multiple times. Otherwise, we could build this memo, but we'd never successfully look up anything in it. And so the algorithm would give us the right answer, but we'd get no speedup,. So it's this property that we need to know that we'll get the correct answer-- that when we combine the local solutions, we'll get the right global solution. It's this property that gives us an indication of how much of a speedup we can expect to achieve. How many problems will we not have to solve, because we can look up the solution?

We'll come back to this. And we'll see how it applies to another problem that you've already looked at say the knapsack problem, to give us a fast solution to that, so that if you want a answer, go back to a previous problem set, and take the full database of classes, you'll be able to solve it quickly using dynamic programming.

OK, see you next time.