PROFESSOR: Good morning, everyone. Let's get started on lecture number two of four lecture sequences of shortest paths. So, last time, we talked about a general structure for a shortest path algorithm.

Today, we'll actually look at a concrete algorithm that's due to Dijkstra. Before we get to that, I want to do a little bit of a review of the concepts that we covered in the lecture last week. In particular, we talked about this notion of relaxation, which is a fundamental operation in all shortest path algorithms.

And I want to go over that again. We look at a couple of special cases today, with respect to algorithms for shortest paths. We look at a Directed Acyclic Graph. Then your graph has no cycles in it.

Regardless of whether you have negative edges or not, there's a straightforward algorithm that we look at to find shortest paths and DAGs. And then, we'll focus in on the case where there are no negative edges. And talk about Dijkstra's algorithm.

So, to start with the review, here's, really, a trivial example of a graph that we want to compute the shortest paths on. And the numbers that are inside these vertices are our priority values. So, think of d of v as the length of the current shortest path from the source, s, to v.

And, given the source, s, the length to the source is 0. So d of s is 0. It starts at 0 and ends at 0. And other ones, I initialized to infinity.

And through this process that we call relaxation, we can generally reduce these d values, that are the lengths of the current shortest paths, down to what we call the delta values. Which is the length of our shortest path. It may be unique. It may not

be unique.

But you have to get the minimum value. And then, all of the vertices have convergent values of d that converge to delta. Then, your algorithm is done.

And one last thing that is important to reconstruct the path is the notion of a predecessor and pi v is the predecessor of v in the shortest path from s to v. And you can follow this predecessor chain to reconstruct the shortest path, once you've converged, and all of the values are down to the delta s comma v. So, in this trivial example, you start with d of s being 0, d of a and d of b being infinity.

Let's put on it a few weights here. And what you do is potentially relax the edges that go out of s. And this notion of relaxation, that I'll write out formally in a minute-- we looked at it last time, is a process of following this edge, and updating the d of a value.

And this infinity becomes 1 because you say, well, if I start here with 0 and I add 1 to it, I get 1 here. Similarly, this infinity becomes 3. And, at this point, you've relaxed the edges that go out of s two these other two nodes, a and b.

You're not quite done yet. At this point, you could imagine that, at least in this example, you found the shortest path to the vertex a. But it is, in fact, a path of length, 2, to vertex b.

Right now, we think that the the current shortest path to b, after the first step of relaxing the edges from s, happens to be 3. But if you go like so, then you end up with the 2. And, at this point, you're done.

Now we have to prove that any particular algorithm we put up is going to converge to the delta values, and the algorithm to terminate. And then, we have to worry about the asymptotic complexity of the algorithm. But that's the general overall flow.

And we look at, as I said, two algorithms today. Both special cases. One for DAGs and one for non-negative edges. And we'll go through, and maybe not do a formal proof, but suddenly give you a strong intuition as to why these algorithms work. Any

questions about this material? OK.

So, what I want to do is give you a sense for why this relaxation step is useful. But also, importantly, safe, or correct. And recall that our basic relaxation operation, which we did over here, as we updated the infinity value to 1 and the 3 value to 2, et cetera, looks like this.

It says, if d of v is greater than d of u plus w u,v. Then, I'm going to update d of v to be d of u plus w u, v. You found a better way of reaching vertex v. A shorter way.

And this way happens to be going through the vertex, u. So you update not only the priority value, but also the predecessor relationship. All right? That's the relaxation step.

Now, I want to be able to show that relaxation is safe. What do I mean by that? Well, I want to make sure that I never relax an edge and somehow do something wrong, which gets me a value that's less than delta s v.

I want to be able to converge from the top. I want to be able to start with these infinity values because I don't have a path to this particular vertex, and continually reduce the values of the priorities. And then get down to delta, the correct values, and don't go, I don't want to go any further. All right?

Because, if I get below, then you're talking about, essentially, you may be able to get back up, but that is not the kind of algorithm that we want. At least, algorithms we look at here. And that is dangerous. So we want relaxation to be safe.

And we can fairly easily prove a simple lemma, using induction, that says that the relaxation operation-- and it doesn't matter what sequence you relax things. This is a fairly powerful lemma that says that if you have an algorithm that uses relaxation, and that's the only way of updating these d values, then it's safe.

You're not going to get a wrong, shortest path value. Either at the end of the algorithm or at any time during the running, or the execution, of this algorithm. OK? So the relaxation operation algorithm maintains the invariant that d of v is greater

than or equal to delta s, v for all vertices. OK?

So that's a powerful lemma. It's a fairly straightforward lemma to prove. But it's an important lemma. It tells us that we can create the generic structure of the shortest path algorithm that I talked about last week.

It says, pick an edge. Relax it. Pick another edge. Relax it. And hopefully everything will work out and you'll get your delta values.

And what this lemma says is, you'll never get something in the middle that is less than your shortest path value. And if you keep running over for long enough time, depending on the particular heuristic that you use for selecting the edges, your algorithm will eventually terminate. And, hopefully, it'll run in polynomial time.

So, how do we prove this? I'm going to do about half of it, then try and get you to finish it. So it's by induction on the number of steps, in the sense that we are going to essentially assume that d of u is greater than or equal to delta s,u.

And we're going to do this relaxation operation. So it's like a base case is that this is correct. And now we want to show that the relaxation operation doesn't make d of v incorrect. So, that's the inductive hypothesis.

Now, we can say by the triangle inequality that I talked about late in last week's lecture, you have delta s, v less than or equal to delta s comma u plus delta u comma v. And what is that? Well, that just says, if I have something like this, that I have s. Let's call this u and v.

This is not an edge between s and v. It's a path. It could be a single edge. But we think of this as a path between s and v. This is a path between s and u. This is a path between u and v.

And, in particular, if there's a way of getting from s to u and u to v that happens to be shorter then the best way of getting from s to v, well, that's a contradiction. OK? Because this is the shortest way of getting from s to v. And it has no constraints over the number of edges that it incorporates.

And so, by definition, the shortest way of getting from s to v is either some direct way. Maybe there's a single edge. Or it may go through this vertex, u. All right? So that's the triangle inequality.

Notice that, what I have here, is something where going from s, to a, to b is actually shorter than going from s to b. But these are single edges we're talking about. These are weights we're talking about.

And there's no contradiction here because all this says is that, what I want to see here is delta s comma b is going to be 2. OK? Initially, I may be starting out with infinity and 3 for the d values. But the delta value, which is the shortest way of getting to b, happens to go through a.

And so, if you use that, then the triangle inequality makes sense. So don't get confused when you see pictures like this, where the weights don't obey the triangle inequality. The triangle inequality has to do with the shortest paths, not the single edge ways. OK?

So, that's half the proof here. What I've done is assumed that d of u is correct. And I've used the triangle inequality. And I've just written this down. Now, someone do the last step, or the second to last step, of this proof. Anybody?

What can I say now, given that what I have here. Look at these two values. What can I say about these values? How can I prove what I want to prove, which is, basically, delta of s comma v should be less than or equal to d of v? OK. That's what I want to show.

I've just written another way here. How do I do that? Anyone? What can I substitute for-- there's a less than operator, which means that I can replace things over here. Yeah.

AUDIENCE: If you, like, you have a [INAUDIBLE]?

PROFESSOR: Ah. Excellent. So the first thing is, I could put d of u over here, right? Less than or

equal to d of u. And the reason I can do that is because d of u is greater then delta s comma u.

So that's cool, right? Sorry, delta. Thank you. Delta s comma u. Thank you. And so, that's what I got here. What else? Yeah?

AUDIENCE: You replace delta u, v with w u, v.

PROFESSOR: I can replace delta u, v with w, u, v. Exactly right. Exactly right. Great. That deserves a cushion. I think you already have one. Yep. Oh, man. I should have not-- so you get that because I messed up.

Seems like you need to get-- whoa. Hey. OK. You get one because I hit you on the head. All right. And this time, I'll just save. I'm running out of cushions here. But I've got some in my office. All right.

So that's it. That's the proof. OK? Fairly straightforward. You get to the point where you want to apply the triangle inequality. You simply look at each of these terms and, by induction hypothesis, you could put d,u here.

And, I just talked about the weights, and so on, and so forth. And you know that w u,v, which is a direct way, a single edge way, of getting to a node, has to be greater than the shortest path. Like here, this 3 value is a direct way of getting from s to b.

And, in this case, it's greater than the shortest path, which is of length 2. But it can never be smaller than the shortest path. And so, once we have that here, we can essentially say, we know that delta s, v is less than or equal to d u plus d v.

Which implies, of course, that this is simply-- once we are done with the relaxation step-- that equals d v. This part here equals d v. OK? That's how that works. So that's good news.

We have a relaxation algorithm that is safe. We can now arbitrarily, and we'll do this for all of algorithms we look at, really. At least in 006, for shortest paths. Which applies some sequence of relaxations. And, depending on the special case of the

problem, we're going to apply these things in different ways to get the most efficient algorithm. All right?

So, we can now do algorithms. Let's look at DAGs first. So, DAG stands for Directed Acyclic Graphs. So that means we can't have cycles. So we can't have negative cycles.

So that's why this is an interesting special case. It makes things a little bit easier for us because we don't have to worry about negative cycles. We're actually going to look at DAGs that have negative edges in them. All right?

So, we're allowed to have negative edges in these DAGs. But we don't have negative cycles. And, as I said last time, it's not the negative edges that cause a problem. If you only go through at negative edge once, you can just subtract that value. And it's cool.

It's only when you get into a situation where you're going through a negative edge, a negative cycle. And you can just iterate through them to get to minus infinity. And you have an indeterminate shortest path value.

So the way this is going to work-- if you ever have a DAG, by the way, the first thing you want to try-- and this is certainly true in your problem set-- when there's a question, try to topologically sort it. OK? It's a fine hammer to use, when you have a DAG.

And it's not an exception here. To do shortest paths, we're going to topologically sort the DAG. And the path from u to v implies that u is before v in the ordering.

And, once you do that, you have this linear. And I'll show you an example. You have this linear ordering. And we're just going to go through, in order, from left to right, relaxing these edges. And we're going to get our shortest paths for all the vertices.

So, the second and last step is, one pass, left to right, over the vertices, in topologically sorted order. And we're going to relax each edge that leaves the particular vertex we are trying to process right now. And so, we know topological

sorting is order v plus e, includes depth-first search.

And this pass over the vertices, you're touching each vertex. And you're touching every edge a constant number of times. In this case, once. So this is our first special case shortest path algorithm. And that takes order v plus e time.

All right? Why does this work? And just one little interesting aspect of this, which is related to a DAG. And the relationship between the DAG and the particular starting vertex that we're going to be looking at.

So, this is an example. Suppose I have a DAG like this. And I marked this vertex as s. And I want to find the shortest path from s to these other nodes that are a and b.

Well, they don't exist, right? So, in this case, I'm going to have a shortest path to a being infinity and shortest path to b being infinity. And this is a trivial example.

So, this algorithm is general. It doesn't say anything about what the starting vertex is. Right? It should work for any choice of starting vertex.

The nice thing is that you can do the topological sort. And then you can commit to what the starting vertex is. And you can go off, and you can say, from this starting vertex, I'm going to go and compute the shortest paths to the other vertices that I can actually reach. OK?

So let's say that you had a DAG that looks like this. All right, once you've topologically sorted it, you can always draw a DAG in linear form. That's a nice thing. I'm going to put edge weights down in a minute.

All right. So that's my DAG. Let's see. 5, 3, 2, 6, 7, 4, 2, minus 1, minus 2. So that's my DAG. And I've drawn it in topologically sorted form. And I go left to right.

Now, let's say that, at this point, I get to step two. And I want to find shortest paths. Now, I have to say, what is my source?

And, if I just happen to have this as my source, well, there's nothing to do here. There's no edges that go out of this. And so that means that everything to the left of

me is infinity. OK?

So the first thing that you do is, you say, find the source that corresponds to the starting vertex. And let's say, this is the starting vertex, in this case. Which I'll mark in bold. So that's my starting vertex.

I'll take a nontrivial case. And everything to the left is going to get marked with infinity. And now, I've got to do some work on relaxation.

And I'm not going to get the shortest path instantly for a particular vertex, once I get to it, because there may be better ways of getting there. And especially if I have negative edges. And that's certainly possible, that a longer length path is going to be the shortest path.

But what I'll do is take s. And I'm going to relax edges that emanate from s. And so, step one, all of these are going to be infinity to start with. So everything is infinity. The ones to the left stay infinity. The ones to the right are going to be reachable.

And you're going to update those values. And so, when you go like so, this becomes 2. This becomes 6. As I follow that. And I'm done with this vertex, s. And this is what I have. 2 and 6.

So the next step is to get to this vertex. Let's call that the vertex a. And I'm going relax the edges going out of a. And, when I go out of a, I get 2 plus 7 is 9, which is greater than 6. So there's no reason to update that.

2 plus 4 is less than infinity. And so, that's 6. 2 plus 2 gives me 4 here. And so on and so forth.

So then, now I'm done with vertex a. If this vertex is b, then I have a value of 6 for this. And 6 minus 1 is less than 6. So this becomes 5. And 5 minus 2-- well, that's the next step after that. I haven't put-- this is a 1.

And so 6 plus 1 is 7. But that's greater than 4. So we don't have to anything there. So the final values that I end up getting are 3 for this one. So this is the final value.

5 is the final value here. 6 is the final value here. 2 is the final value here. And that one is 0. And this stays infinity. OK? So fairly straightforward.

Do a topological sort. Find the starting point. And then run all the way to the right. Interestingly, this is actually a really simple example of dynamic programming, which we'll talk about in gory detail, later in November.

But what I have here is the simplest special case of a graph that has an order of v e [INAUDIBLE] shortest path algorithm. And the reason for that is we don't have cycles. All right? Any questions about this? People buy this? It works? OK.

So, we've got one algorithm under our belt. And we look at, really, a more interesting case because most graphs are going to have cycles in them. But we will stay with the special case of no negative edges, now. All right?

So Dijkstra's algorithm doesn't work for negative edges. So it's different. This algorithm is not subsumed by Dijkstra. That's important to understand. So Dijkstra's algorithm works for graphs with cycles. But all of the edge ways have to be either 0 or positive.

This algorithm works for DAGs that can have negative edges. But you can't have cycles. So both of these algorithms have their place under the sun.

So, let's take a look at Dijkstra's algorithm. Actually, I guess I have a demo. So, the one demo we have in 6006.

[INAUDIBLE] Dijkstra is a very straightforward algorithm. It's not trivial to prove its correctness. But from a standpoint of coding, from a standpoint of understanding the flow, it's a very straightforward algorithm.

One of the reasons why that's the case is because it's a greedy algorithm. It does things incrementally, maximizing the benefit, as you will. And intuitively builds the shortest paths. And it goes vertex by vertex.

So here's a demo of Dijkstra, which, the reason I want to show you this, is because

it will give you some intuition as to why Dijkstra works. Now, some points of note. I can't tilt this more than about this much because then these balls will fall off. So, cameraman, can you get this? All right? For posterity.

So I got an undirected graph here, right? And each of these things are nodes. The balls are the nodes of the vertices. And I've drawn the picture over there. And G stands for green. And Y stands for yellow, et cetera. So, this graph is essentially what I have up there.

And I've put strings connecting these balls, associated with the weights that you see up there. So, if I got this right, the string that's connecting the green ball to the yellow ball up on top is 19 centimeters. And so on and so forth for these other ones. All right?

So, that's Dijkstra. And what do you think I have to do to compute shortest paths, mechanically speaking? What do you think I have to do? Yeah, someone.

AUDIENCE: Pick up the green ball and just--

PROFESSOR: Pick up the ball and lift it up. That's right. Good. It's worth a cushion. All right, so, let's all this works. So, first, let me show you by those values that I have there.

If the green ball is the starting vertex, then the shortest path to the purple vertex, p, is 7. And that's the closest node to G. And then, the next closest node is the blue one, which is b, which is 12. 7 plus 5. And so on and so forth.

And so, if this all works, and I haven't tried this out, because this is a one use demo. Once I pull this up, the strings get so tangled up, it doesn't work anymore. All right? So that's why I had to do all of this, lug these over.

Otherwise, it'd be-- so this is not a computer reversible kind of thing. So, if you want to code Dijkstra up. OK, so if I just lift it up, and if I do that, and if I tilt it in the right direction. Yeah. I want to that.

So you can see that this is a little bit of fudging going on here, with respect to getting this right. But you see green is up on top. And what is the next one you see?

AUDIENCE: Purple.

PROFESSOR: Purple. That's good. What's the next one you see?

AUDIENCE: Blue.

PROFESSOR: Blue. That's good. Y, and then R. And strings that are taught, that have tension in them, are the predecessor vertices, OK? That's the pie. All right?

So, again, I computed the shortest paths, right? Mechanically. And, if I could have a way of measuring the tension on the strings, I have my pie, my predecessor relationship, as well. All right?

Now, let's see if this works. This works, right? So, if the second thing doesn't work, don't hold it against me. But, let's say if I take R, and I lift it up like that. Yikes. So, R, followed by Y, followed by B, followed by P, followed by G.

Hey. Come on. All right? This works. Thank you. Thank you. All right. So there's actually a reason why I did that demo.

There's a greedy algorithm here. And, I guess, greedy is gravity. Right? Gravity is greedy. So, obviously, the reason why those balls are hanging is because they have weight. And they have gravity.

And you can imagine that you could now-- people in physics. I don't know anybody majoring in physics. Anyone double majoring in physics or something here? All right.

So, you know your Newton's laws of mechanics. And you know about gravity, and all of that. So you can imagine that you said, you know, the heck with all this priority queue stuff in the problem set.

In the algorithm that we're going to be talking about for Dijkstra, I'm going to do a kinetic simulation of shortest paths in order to get the actual values of these shortest paths. OK? Now, that would be cool. But it'd be horribly slow.

And so, the Dijkstra algorithm we're going to be talking about is going to just

compute the steady state, corresponding to the closest vertex that is closest to G. All right? So Dijkstra, the algorithm, the intuition behind it, is that it's going to greedily construct shortest paths. And it's going to be starting with G, which is your source vertex.

And then, the first thing that it's going to process, and find the shortest path to is going to be the purple vertex. And then the blue. And then the yellow. And then the red. All right? So it actually mimics, to some extent, this demo. All right?

So, let's take a look at the pseudocode for Dijkstra. So, g is your graph. w are the weights. Small s is the starting vertex. We're going to initialize g and s, which means we just mark s a starting vertex.

And we're going to also have this capital S, that I'll use these little bars to differentiate from small s. So this is a set. Capital S is a set. And we're going to initialize that to null.

And there's another set called Q, which is initialized to the entire set of vertices. And all this means is that, initially, we haven't done any processing. And we don't know the shortest paths to any vertex because this set of vertices is null.

And Q is the set of vertices that need to be processed. And, as we start processing vertices from Q, we're going to move them to capital S. And they're going to contain the set of vertices that we know the shortest paths to already.

And that's the invariant in this algorithm. s is going to contain the set of vertices that we know the shortest paths to. And so, Dijkstra is a little while loop that says, while they're vertices that need to be processed, then I'm going to take u. And I'm going to extract-min from Q.

And this is going to delete u from Q. And this initialization-- and this is a small s here-- is going to set d of s to be 0. That's all this initialization does. Because that's all we know.

We have a starting vertex. And we know that the shortest path to the starting vertex,

from the starting vertex, is 0. So, all that means is that, all of the other ones have infinity values. So, at this very first step, it makes sense that extract-min Q is going to pull the starting vertex, small s, out. And is going to assign it to this u value.

And we're going to set s to be-- capital S-- to be capital S union u. And then, all we have to do is relax the edges from the vertex that we just added. So, for each vertex, v belonging to adjacent s, so that you can reach from u. We relax u, v, w. All right? That's it. That's Dijkstra.

It's a greedy algorithm. It's iterative. And the reason it's greedy is because of this step here. It's just picking the min priority from the un-processed vertices, Q. And, essentially, claiming that this min value is something that you already computed the shortest paths for.

So, when you're putting something into S, you're saying, I'm done. I know the shortest path to this particular vertex. And I need to now process it, in the sense that I have to relax the edges that are coming out of this vertex.

And update the priority values because relax is going to go change the d values, as we know, corresponding to the vertex, v. It might change the value. It might not. But there's a possibility that it would.

And you're going to do this for all of the edges that are emanating out of the vertex, u. And so you may be changing a bunch of different priority values. So the next time around, you will get a different minimum priority vertex.

For two reasons. One is that you've extracted out the minimum priority vertex. You've deleted it from Q. And the second reason is that these priority values change as you go through the loop. All right?

And so, in our demo, essentially what happened was, the first time, the process of lifting the green vertex, corresponding to choosing it as a starting vertex. And the first thing that was closest to it, which had the taught string hanging from it, has the min priority value. And you pull that out. And then so on and so forth, as you go down.

And I'm not going to go through and prove this. But it's certainly something that is worth reading. It's half of page proof, maybe a page in CLRS.

And you should read the proof for Dijkstra, the formal proof for Dijkstra. Which just, essentially, does all the accounting and gets things right. And uses the lemma that we have, with respect to the relaxation operation being safe.

OK? Any questions about Dijkstra? Or about the pseudocode, in particular? I guess you guys are going to code this at some point. Yeah?

AUDIENCE: How are the vertices comparable? In what way?

PROFESSOR: Oh, so that's a good question. And I should have made that clearer. So, Q is a priority queue. And the priorities of the vertices are the d values, OK?

s being null is clear, I hope. That's clear. And then Q being the set of vertices are clear, as well. Now, Q is a priority queue, OK?

And we'll talk about how we'll implement this priority queue, and the complexity of Dijkstra, before we're done here. But, as an ADT, as an Abstract Data Type, think of Q as being a priority queue. And there's priorities associated with each vertex that's in Q. And these priorities change. And they're the d values. All right?

So the priorities. So, initially, d of s-- small s-- is 0. And all of the other ones are infinity. So it's clear that, the very first time, you're going to set u to be small s, which is a starting vertex. And then you relax the edges coming out of s, potentially change some of these other infinity values of the vertices that you can reach from s to be less than infinity.

And you're going to, essentially, change the values of the priority queue. And go around. And then select the min value the next time. And so on and so forth. OK? Thanks for the question. Any other questions? OK.

So, let's just go through a couple of steps in an example. I'm not going to go through the whole thing. But you'll see an execution of Dijkstra in the nodes. I think

it's worth spending just a couple of minutes going through the first few steps of a Dijkstra execution.

Just so how this priority queue works is clear, let's take a look at a directed graph that has five vertices. So that's 7. So let's start with a being the starting vertex. And so d of a is 0. And d of b through e are all infinity.

Your s is null to begin with. And Q has all of the five vertices in it. So extract-min is going to select a. That's the only one that is a 0. Because you've got 0, infinity, infinity, infinity, infinity.

And so, you select that, and you set s to be a. And once you set s to be a, you relax the edges coming out of a. And there's two of them. So you end up with 0, 10, 3, infinity, infinity. And the next extract-min is going to select 3. And you're going to set s to be a comma c.

And so you're, essentially, doing kind of a breadth-first search. But you're being greedy. It's a mixed breadth-first depth-first search.

You do a breadth-first search when you're given a particular vertex, and you look at all of the vertices that you can reach from that vertex. And then you say, I'm a greedy algorithm. I'm going to pick the vertex in this frontier that I've just created, that is the shortest distance away from me, that has the lowest priority value.

And, in this case, it would be c because this other one is 10. And this is shorter. Right? So that's why we pick c over here. And one last one.

Once you process c, you're going to end up processing this edge going out here. This edge going out there. This edge going out this way. And you're going to end up with 0, 7, 3, 11, 5.

And you've processed a bunch of edges coming out of c. And, at this point, 0 is gone and 3 is gone. I'm just writing the values here, just so you know what they are.

But these are out of the picture because, in s, those values should never change.

Dijkstra essentially guarantees. And that's the proof of correctness that takes a bit of doing, is that this value is never going to reduce anymore.

The pre-value is never going to reduce. And it's been put into s. But what's remaining now is 5. And that corresponds to the e vertex.

So s becomes a, c, e. The 5 gets stuck in there. And so on and so forth. All right? So, that's Dijkstra. And now, let's start complexity.

So, it we have the code for Dijkstra on the left, we have an ADT associated with the priority queue. And now, we're back to talking like we did early on in the term, where we compared linked lists, and arrays, and heaps, and trees. And said, for a particular set of operations, which one is going to be the best? OK?

So, if you analyze Dijkstra, and you look at the pseudocode first, and you say, what are the operations that I'm performing? I got an operation here, corresponding to theta v inserts into the priority queue. And that's inserting things into Q. I got theta v extract-min operations.

I'm only going to delete a vertex once, process of vertex once. And that's why I have theta v extract operations. And I have theta e, what decrease key or update key operations because when I do, I relax here. I'm decreasing the key.

It's in particular, it's not an update key. It happens to be a decrease key, which is not a big deal. We don't need to get into that. But you are reducing the d value. So it's a decrease key operation.

And, again, it's theta e because, in a directed graph, you're only going to process each edge that's coming out of the vertex once. Since you're processing each vertex once, and you're looking at all of the outgoing edges from that vertex. OK? So that's what you can get looking at the pseudocode.

And now, you're a data structure designer. And you have some choices here, with respect to actually implementing the priority queue. And let's look at the complexity of Dijkstra for arrays.

So, suppose I ended up using an array structure for the priority queue. But then, what do I have? I have, if I look at this, my extract-min, what is the complexity of extract-min in an array?

AUDIENCE: Theta v.

PROFESSOR: Theta v. And what's the complexity of a decrease key in an array? I just go access that element. And I change it. State of one, right?

So I have theta v for extract-min. I'll just call it ex-min. Theta one for decrease key. And if I go do the multiplication, I get theta v times v plus e times 1, or a constant, which is theta v squared.

Because I know that e is order v squared. Right? If I have a simple graph, it may be a complete graph, but-- we talked about this last time. e is, at most, v squared. So I can just call this theta v squared. All right?

So we have a theta v squared Dijkstra implementation that uses an array structure. But do we want to use an array structure? What data structure should we use? Yeah?

AUDIENCE: Heap.

PROFESSOR: You can use it a min-heap. Exactly right. So, if you use a binary min-heap, then my extract-min is finding the min is a constant because you just pick it up from the top. But we know that, if you want to update the heap, and delete it, then it's going to take that theta log v.

And decrease key is the same thing. Theta log v. So that's worse than array. And if I go do the multiplication again, I get v log v plus e log v. OK?

And this is not quite the complexity that I put up, as some of you may remember, last time. This is not the optimum complexity of Dijkstra. Or an optimal complexity of Dijkstra. You can actually take this out by using a data structure that we won't talk about in 006.

But you can read about it. It's not 6006 level material. You're not responsible for this in 006. But it's got a Fibonacci heap. And you might learn about it in 6046.

The Fibonacci heap is an amortized data structure that has theta log v for extract-min. And theta one amortized time for decrease key. And what's nice about it is that, once you do that, you end up with theta v log v plus e time.

And that's the complexity I put up way back, I guess, last Thursday. So that's to show you, with respect to two special cases, we have the DAGs, which are linear time, essentially. And Dijkstra, with amortized, and their proper data structure, also, essentially, linear time. Right?

Next time, we'll look at the general case where we have potentially negative cycles. And we end up with algorithms that have greater complexity. See you next time.