PROFESSOR: [INAUDIBLE] audio. We're going to have a series of lectures on shortest paths. And one of the big differences between this module and the previous one, at least from a mathematical standpoint, is that we're going to be looking at graphs that have weights on their edges.

So when Eric talked about depth first search and breadth first search in the last couple of lectures, we had directed graphs and undirected graphs. But we didn't really have attributes on the edges. In particular, you have a much richer class of problems and applications if you allow for weights on graph edges.

And these weights can be integers. They could be real numbers, irrationals. They could be negative, what have you. And for different classes of graphs and different restrictions on weights, there's really a set of shortest path algorithms, that we'll look at in the next few lectures, which are kind of optimized for a particular application.

So we won't do specific algorithms today. But we'll set up the problem. We'll talk about the general approach that most shortest path algorithms take to solve a particular instance of a problem. And then we'll close with talking about a particular property that's pretty important.

That's called the optimum or optimal substructure property. That is a technique that most shortest path algorithms, or actually all shortest path algorithms use to get efficient complexity. So asymptotic complexity is important, obviously. And we're always looking for the best algorithm with the best asymptotic complexity. And optimal substructure is a hammer that we're going to use to get that.

So the canonical motivation, of course, for shortest paths is-- now, if you want to steal, or I guess borrow, a cannon from Caltech and bring it over to MIT, then you

want the fastest way of getting here with your illegal goods. And you want to find the shortest way or the fastest way of getting from one location to another.

So Google Maps go from point A to point B. That's a classic application of the shortest path problem. In this case, you could imagine that distance would be something that would be a very simple metric that you could use for the weights on the edges.

So for this entire module, we're going to be looking at a graph G V, E, W. And you know what V and E are. They're the vertices and the edges.

And W is a weight function that maps edges to weights. And so we're adding that in here. And so W would be E to R. So the set of real numbers.

We're going to be looking at two different algorithms in subsequent lectures. And you'll implement one of them in your problem set. The simpler algorithm, which we'll look at first, is called Dijkstra, after Edsger Dijkstra, who did some similar work in concurrent programming, won the Turing Award.

But on the side invented this cool algorithm-- or at least gets credit for it-- called Dijkstra's algorithm that assumes non-negative weight edges. So I should really say non-negative. So read that as non-negative. And that has a complexity of order V log V plus E. All right.

So this is practically linear time. And typically, you're going to be dominated in many cases by E. In general, if you talk about a simple graph, what's the asymptotic relationship between E and V? Can you relate E to V? Can you give me a bound?

AUDIENCE: V squared.

PROFESSOR: Sorry? V squared, thanks. That's good.

So you can think of E as being order V square. And you can certainly have-- that's worth recursion. So now, you can kind of imagine a complete graph. And a complete graph is something that has an edge between each pair of vertices. And that's where you'll get E being k w squared.

So when you say simple graph, you're saying you have at most one edge between any pair of vertices. A multigraph is something that could have multiple edges between pairs of vertices. We won't really be talking about multigraphs in this sequence of lectures. But something to think about or keep in the back of your mind as we go through these algorithms.

And so the dominating factor here, and in many cases really, is E. And Dijkstra is a nice algorithm, because it's linear in the number of edges. So that's Dijkstra.

And that's the first of the algorithms that we'll look at next time. But we'll see the general structure of Dijkstra today. And then there's the Bellman-Ford algorithm that works on positive and negative edges, weight edges. And this has a complexity order V E.

So you could imagine a particular implementation of Bellman-Ford running in order V cubed time. Because E could be V square. And you've got this additional E factor. So it's order V cubed versus order V log V. So when you have a chance, use Dijkstra. When you're stuck, you'd want to do Bellman-Ford, because you have these negative weight edges.

And one of the challenges in negative weight edges, and I'll say a little bit more as we go along, is that you end up having to have to find cycles that are of a negative weight, because they kind of throw off your shortest path algorithm if you were just assuming that shortest path lengths are only going to decrease. But when you have negative weights, you might take a step and the overall weight might decrease.

So it's kind of a longer path in terms of the number of edges. But the weight is smaller. And that kind of makes the algorithm more complicated. And it has to do more work.

So that's really why there's a difference between these two complexities. And I guarantee you, you'll understand this much better after we're done with the lectures on Dijkstra and the lectures on Bellman-Ford.

So that's the set up for the problem. That's what we're going to be looking at. Let's look at a couple more definitions beyond what I have here with respect to just the notation. And you can think of path p as a sequence of vertices-- V0, V1, et cetera, to Vk.

And this is the path if Vi, Vi plus 1 belongs to E for 0 less than or equal to i less than or equal to k. So a path is a sequence of edges. And each of those edges has to be in the graph, has to be in the set of edges E.

And W of p, which is the weight of the path, we know that by the weight of edges, those are easy. They're given by the W function. The weight of the path is simply the summation of the weights of the edges. All right.

So fairly obvious definitions. But obviously, we have to get these right in order to actually solve the problem correctly. And the shortest path problem is, as you can imagine, something that tries to find a path p that has minimum weight. So in general, you have some set up for the problem. But it comes down to find p with--

And there are many, many possible paths. You have to understand that there are potentially an exponential number of paths in the graphs that we would consider. And here's a real simple example where you would have an exponential number of paths. And we'll come back to this example later in the lecture.

But let's assume that all the directions go this way. And it's a directed graph. Well, you could have the path that goes all the way here. But you could have the path that goes on top and all the way this way. You have basically two choices on getting to this vertex.

Then you've got, given the two ways you have of getting to this vertex. You've got four ways of getting here. And then, you have eight ways of getting there. So on and so forth. So there's an exponential number of paths potentially.

The other thing that's interesting here, which is important in terms of this complexity is, what's interesting about what you see here with respect to the complexity and what you see here. Anybody want to point that out?

So I have this complexity here and order VE out there. What's an interesting observation if you look at this board here and the two complexities? Anybody? Yeah, back there.

AUDIENCE: It's not a function of weight.

PROFESSOR: It's not a function of weight. Great. That's definitely worth recursion. And I'll let you throw this one. All the way back there, all right? Right.

All right, good. Good, good. That was good. That was better than what I could do. No, not really. But I would've been right in his hands right there.

Anyway, so that's a great observation, actually. And I should have pointed it out right up front. But I'm glad I got it out of you. W doesn't exist in the complexity. This is pretty important. W could be a large number. I mean, it could be 2 raised to 64.

The fact is that there's only E square different values possible for a weight, right. I mean, roughly speaking. If you have a complete graph, it's a simple graph, there's order E square possible weights.

But the range of the weights could be exponential. I could have an edge weight of 0.0001 and a different edge weight of 10 raised to 98. There's nothing that's stopping me from doing that or putting a specification like that.

But the nice thing about Dijkstra, and Bellman-Ford, and virtually all of the algorithms that are useful in practice is that they don't depend on the dynamic range of the weights. And so keep that in mind as you think of shortest path algorithms. And we'll talk a little bit about this in section tomorrow, or the TAs will, as to why breadth first search and depth first search aren't directly applicable to the shortest path problem. And the hint really is the dynamic range of the weights. So keep that in mind.

So a couple things why this is an interesting algorithm, or interesting problem to solve, and harder than the problems we've looked at so far like sorting and search,

is that you have an exponential number of paths. And then the dynamic range of the weights can be very, very large. And it's not linear by any means. All right.

So these algorithms are going to have to have some smarts. And the optimal substructure property that we'll look at towards the end of today's lecture will give you a sense of how these algorithms actually work in basically linear time. Or VE, you could think of that as being cubic time in terms of the vertices.

So keep that in mind. Let's talk a little bit more about weighted graphs. I want a little bit more notation.

And what I have is V0 using path p to Vk. So I'm going to write that to say that there's a particular path of V0 to Vk. Sometimes I'm searching for the path p with a minimum weight. And that's how I'm going to represent that.

V0, which is a single vertex path, is the path from V0 to V0. So it's really a 0 length path. And it has weight 0.

So that's one condition. The other condition that we need to look at, which is the other case, is what if there isn't a path? So I want to put those two things together, the two extremes, and of course all of the cases in between, in this definition of the shortest pathway.

And so I'm going to talk about the shortest path value of the weight of the shortest path between u and v as delta, u, v. And my goal is to find delta. It's also to find the path.

It doesn't help you very much if you know that there's a way of getting from here to Lexington within 14 miles if you don't know what that path is, right. So that's one aspect of it, which is you want to get the weight. But you want to get the path as well. And these algorithms will do that for you.

And in particular, what we want is delta u, v to be the minimum over all the paths W p, such that p is in fact the path from u to v. And this is the case where if there exists any such path, and the last thing is you want this to be infinity, the weight to be

infinity otherwise. So if you're only talking about roads going from here to Tokyo, should have length infinity. A little matter of the Pacific Ocean in between.

So that's the set up in terms of the numbers that we want to see. If you're starting from a particular point, you can think of the shortest path length from your source as being a 0. Initially, everything is infinity because you haven't found any paths yet. And what you're going to do is try and reduce these infinities down for all of the vertices that are reachable from the source vertex.

And it's quite possible that you may be given a graph where a particular vertices, or in your set V, that can't be reached from the particular source that you started with. And for those vertices, you're going to have your delta u, v. If v is unreachable from you, it will stay at infinity.

So let's look at an example. Let's take-- it's going to be an iterative process here of finding the shortest paths. And so let's take an example that corresponds to a fairly complex graph, or at least a nontrivial one, where that's my source vertex. And I've labeled these other vertices A through F. And I have a bunch of edges. 5.

I got one more here. So that's what's given to me. And I want to find delta S plugged in for u. And A, B, D, et cetera plugged in for V for this graph.

And let's just do this manually if you will. And just trying to do some kind of breadth first search. And we do know breadth first search. We know depth first search. You can imagine trying to use those notions to try and find the shortest paths here.

So now we have to prove afterwards when we are done that these are, in fact, the shortest paths. And that's the hard part of it. But we can certainly try and fill in some numbers associated with paths that we do know about.

So I'm going to say that the numbers that are inside each of these vertices-- d of u is the current weight. And so initially, I'm going to start with D of S being 0, because that's a source. And all of these other ones are going to have-- I'm not going to write this down-- but they're going to have infinity for their D of Vs.

So D of A is infinity. Do of B is infinity, et cetera. And what I want to do is decrease this D number to the point where I'm confident that all of the D numbers that are inside these vertices, these are the current weights, or end up being the delta numbers. So my algorithm is done when my d numbers shrink down. And I got the delta values, the correct delta values.

But if I wanted to do this, sort of a seat of the pants approach, just go off and try and iteratively reduce these numbers, you say, well, this one was infinity. But clearly, if I start from S and I follow the edges in S, I'm going to be able to mark this as a one. And similarly here, I'm going to be able to mark this as a 2.

Now, I could arbitrarily pick this one here and this A vertex, and then start looking at the edges that emanate from the A vertex. And I could go off and mark this as 6, for example. And if I start from here, I'd mark this as 3.

Now, is it in fact true that 6 equals delta S comma C equals 6? No. What is in fact-- is there a better way of getting to C? And what is the weight of that? What vertex do I have to go through?

I mean, one way is to go from S to B to D to C, right? And that would give me 5 right? So that's 5. Can I do better than 5? Not in this graph. OK

So it's not the case that the shortest length path gave you the smallest weight. I mean, that was one example of that. And I can go on an and bore you with filling in all of these numbers. But you can do that on your own. And it's really not particularly edifying to do that.

But you get a sense of what you need to be able to do in order to converge on the delta. And it might take some doing. Because you have to somehow enumerate in an implicit way-- you can't do it in an explicit way, because then there'd be an exponential number of paths. But you'd have to implicitly enumerate all the different ways that you can possibly get to a vertex and discover the shortest path through that process, all right. And so we have to be able to do that in these shortest path algorithms.

And this is a simple graph that has positive weights, non-negative weights with edges. It gets more complicated when you have negative weights. But before I get to that, there's one other thing that I want to talk about here with respect to discovering the actual path.

So what we did here was we had delta u, v that corresponded to the weight of the shortest path. But if you want the path itself, we need to have a way of finding the sequence of vertices that corresponds to the minimum weight path. And in particular, we're going to have to define what we call the predecessor relationship.

And so what I have is d of V is the value inside the circle, which is the current weight. And as d is something you're very interested in, eventually you want it to go to delta. The other thing that you're very interested in-- and this is really a fairly straightforward data structure corresponding to just the d number and this predecessor number. And pi of V is the predecessor vertex on the best path to V. And you said, pi of S equals NIL.

And then you can think of this as this is eventually what we want, and this gets modified as well. So right now, when you're working and trying to find the path, you have some particular path that happens to be the current best path. And that's a sequence of vertices that you can get by following the predecessors.

So once you're at a particular vertex E, you say all right, right now I can look at pi of E. And if that points me to C, then that's good. I'm going to look at pi of C. And that might point me to A, and so on and so forth.

In this particular instance, pi of E is going to, when you're finally done, is going to point to A. And pi of A is going to point to S, all right, because that's the path that is the best path is this one. Like so and like that.

And so those are the two data structures you need to keep in mind that you need to iterate on, this predecessor relationship and the current distance. And then this ends up being delta. You're done.

And at that point, your predecessor relationship is correct. So that's the set up. The

last complication I want to talk about here is negative weights. And it's, I think, appropriate to talk about it when we have Bellman-Ford up here. Which is really the general algorithm.

So let's talk about-- so the first question is why. Why do these things exist, other than making our lives more difficult? So give me an example. What is the motivation for a graph with negative weights?

I mean, I really would like to know. The best motivation is definitely worth recursion. Then I can use it next time. Yeah, go ahead.

AUDIENCE: I'm just thinking like if your goal, if your goal [INAUDIBLE]. And some of them cost too much. Some of them get you money, and you want to know what-- you're trying to find [INAUDIBLE].

PROFESSOR: Sure. Yeah, I mean, I think that's a good motivation. I think driving, when you think about distances and so on, there's no notion of a negative distance, at least physically. But you can imagine that you could have a case where you're getting paid to drive or something, or it costs you to drive, and that would be one. Yeah, go ahead.

AUDIENCE: It sounds like Monopoly. So the vertices are supposed to be [INAUDIBLE].

PROFESSOR: Oh, if you land on something, you have to pay rent. Or sometimes you land on something and you actually get money.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Takes you forward, backwards, right. Yeah go ahead.

AUDIENCE: [INAUDIBLE].

PROFESSOR: So that is such an interesting notion. Sometimes you may want to go. And maybe in this case, you're saying it's better to take your distance metric and go further away in order to get the best way of getting there, or something like that.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Right. Sure. That'd be good. Right. Victor, you had your hand up.

AUDIENCE: Yeah. I'm going to give [INAUDIBLE] on the highway, you can't [INAUDIBLE] distances [INAUDIBLE] negative. Well, if a government uses [INAUDIBLE] police to regulate traffic, then you might have a negative distance. Because obviously, you could go a certain way minus the [INAUDIBLE].

PROFESSOR: Right. Yeah, that's a good example. One of the things that we have to think about is-- and this is something that might come up, by the way, in a problem set or a quiz-- which is, is there a way of shifting these weights to make them all positive? So the examples we've talked about, not clear to me that in the particular settings that we talked about that you can somehow create the base case to be 0 rather than being negative.

So it may not be possible in a particular scenario. But if you can do that-- and the reason I bring this up is if you can do that, suddenly instead of using an order V, E algorithm, if you can prove correctness of the final solution is exactly what you'd have gotten for the initial problem certification, you've gone from an order V, E algorithm to an order V log V algorithm. So that's a wonderful thing to do.

So keep that in mind. Try and get rid of negative weight edges if you can without changing the problem certification. I saw a hand back there.

AUDIENCE: Oh, no. I thought you were just asking the question, if we could do that? So I was just gettin ready to answer.

PROFESSOR: OK, yeah, so that's something to keep in mind. One example that I think has come up here, which came up, I think, the last time I lectured was imagine that you're driving and there are all these advertisements. And you get paid to drive on a freeway. So the reverse toll.

I mean, it's a reverse toll, because you get to go there and you have to see all these ads. And then I guess you drive pretty fast through those ads. But you have to go

through. And so you get paid to go through those particular roads.

And then what about social networks? I mean, there's liking people and disliking people. I mean, that sounds pretty-- that's negative and positive. One could imagine that social networks would have positive weights and negative weights.

I'm surprised one of you-- I mean, I don't have an account on Facebook. But presumably, you guys do. So think of what's the-- yeah, that's right. Well, I'm not sure how this works. But you guys figure it out.

So why? Reverse tolls, social networks. Lots of things. Even if you're not convinced by the motivation, I will spend a whole lecture talking about Bellman-Ford. So that's just so that's clear.

So the issue with the negative weight cycles is something that is worth spending a minute on. And I talked about the fact that you had an exponential number of paths. And that causes a bit of a problem, even in the case where you have positive weights. And I will revisit that example.

But here's an even worse problem that corresponds to negative cycles. So eventually, you want to terminate. The faster you terminate, and if you can talk about asymptotic complexity, obviously that means that you've terminated within a worst case bound of time.

And if that's exponential, that's bad. You'd want it to be small. But what if you didn't even terminate?

So suppose you have something like this where you have a graph that has negative weights on some of the edges. But others are positive. So this one has a minus 6. I think I got those right.

So 2 for minus 6 over here, 3, 2, 1, and minus 2. So one thing that you notice from this graph is that you got this annoying cycle here. That's a negative weight cycle. And that's why I've picked this particular example.

Minus 6 plus 2 is minus 4. Minus 4 plus 3 is minus 1. So if you had something where

you can't depend on the fact that the D's are going to keep reducing. And that eventually, they'll stop reducing.

Well, that's true. Eventually, they'll stop reducing. Because they're lower bounded by 0 when you have positive weight edges or non-negative weight edges. But if you have a graph with a negative cycle-- and I mean, this is a recipe for an infinite loop, right-- in your program, potentially a bug. But maybe not even a bug, not a bug in implementation, but a bug in the algorithm. Because this determination condition isn't set properly.

So you can imagine that you would get to B and the first time-- whoops, I'm missing a weight here. So you get to B. And you say, well, I'm done. Delta of SB is 4.

But that's not true. Because you could get to B. And then you could get back to B with the weight of 3. And then you could do it again with the weight of 2, and so on and so forth. So that's a problem.

So what would you like an algorithm to do? What would you like the Bellman-Ford to do here? It's not the case that all of the delta values, that is the shortest path values, are undefined for this graph. Some of them are well defined.

This one, you can't ever get back to it. So clearly, delta S, S is 0. Everybody buy that? What about this one?

It's 2, right. Delta S, A is 2. And everybody buys that, because there's just no way. You can't, you don't touch a negative weight cycle.

You, in fact, don't touch a negative weight edge. But more importantly, you don't touch a negative weight cycle in order to get to A. And there's no way of touching that.

On the other hand, anything that's in here you could run many times. And you could end up with whatever weight you wanted. There'd be a minus infinity weight.

So what you want an algorithm that handles in particular negative cycles, which are

the hard part here. Negative weights aren't the hard part if you can't run through these edges more than once. It's actually the negative cycles that are hard.

And the negative cycles are going to make shortest path lengths indeterminate, but not necessarily for every node in the graph, like this example shows. So what you want your Bellman-Ford algorithm to do, or your shortest path algorithm that handles negative cycles to do, is to finish in reasonable amounts of time. Order V, E will take and give you the delta numbers for all of the vertices that actually have finite numbers and then mark all of these other vertices as being indeterminate, or essentially minus infinity. OK

So that's your termination condition. It's different from the termination condition if you simply had non-negative edge weights. All right. So remember, it's cycles that cause a problem, not just the edges. And you have to do something about the cycles. But they may not affect the entire part of the computation.

So if you don't know that you have a cycle or not, then you end up with having to use Bellman-Ford. And so that also tells you something which is interesting, which is Bellman-Ford has to detect negative cycles. If Bellman-Ford couldn't detect negative cycles, then how could it possibly be a correct algorithm for the arbitrary case?

So Dijkstra doesn't have to do that. And that's why Dijkstra is simpler. All right.

So let me talk about the general structure of shortest path algorithms. And the 2 important notions that I want to talk about here are the notion of relaxation, which we sort of did already when we ran through this example. But I need to formalize that. And then we'll go back and revisit this exponential graph example.

So the general structural of shortest path algorithms are as follows. We're going to initialize for all u belonging to the vertex set, we set d v to be infinity. And we set the predecessor to be NIL.

And then we'll set d of S to be 0. We're talking about a single source, here. We'll set that to be 0. And what we're going to do is essentially repeat.

Select some edge u comma v. And I'm not specifying how. This is going to result in a different algorithm depending on the specifics of how.

But the important notion is that we're going to relax edge u, v. And what the notion of relaxation is is that you're going to look at it. And you'll say, well, if d of v is greater than d of u plus w u, v, then I've discovered a better way of getting to v then I currently know.

So d of v would currently be infinity, which means I haven't found a way of getting to v yet. But I know that d of u, for example, is a finite number. And I do know that this edge exists from u to v, which means that I can update the value of d of v. And that's what we call relaxation of the edge u, v.

And so what you do here is if the if is true, then you set d, v to be d, u plus w u, v. And you'll also update the predecessor relationship, because the current best predecessor for v is going to be u. So that's the notion of relaxation.

And I kind of ran out of room here. But you keep doing this. This repeat. So what is the repeat? Well, the repeat is until all edges have d of v less than or equal to d of u plus w u, v.

And the assumption here is that you have no negative cycles. We need a different structure. The notion of relaxation is still going to be relevant. But don't think of this structure as being the structure that Bellman-Ford uses, or algorithms that can handle negative cycles use.

So hopefully, you got the notion of relaxation, which is from a pictorial standpoint, it's simply something that we did when we looked at updating the value of 6 to 5, for example. So we said through this process, if I relax this particular edge and d was already set up-- let's say d, the vertex here had 3. And this was originally 6. And I look at it and I say, D of C is 6.

On other hand, 6 is greater than d of the vertex D, which happens to be 3 plus 2. And since 5 is less than 6, I can relax this edge and update the value of 6 to 5. And

then I update the predecessor relationship to have a pi of C to be D. That's the notion of relaxation. Fundamental notion. Going to use it in every algorithm that we talk about.

When do you stop? Well, when you don't have negative cycles, there's a fairly clean termination condition, which says that you can't relax any of the edges any more. OK You get to the point where you have values that are associated with each of these vertices inside. And it doesn't matter what edge you pick, you can't improve them.

So this termination condition, it could involve an order E check. So we're not talking complexity here yet in terms of being efficient.

But you can imagine when I say until all edges cannot be relaxed, that you'd have to look at all the edges. And if any one of them can be relaxed, it's possible that another one can now be relaxed. So you've got to keep going until you get to the point where none of the edges can be relaxed.

So this is a brute force algorithm. And it'll work. It'll just be slow. It'll work for known negative cycles. And if you just kind of randomly select these edges and just keep going, I'll give you an example where it works pretty badly in a minute. But this is an algorithm.

So I guess I lied when I said we weren't going to give you an algorithm. It is an algorithm. It's just an algorithm that you never want to implement. You do want to implement the relaxation condition. But not this random way of selecting edges and having this termination condition that, in of itself, is an order E check.

And one of the reasons why you don't want to implement this algorithm is coming up shortly in our exponential graph example. But let me make sure that people aren't bored. Any questions about the general structure, relaxation, anything? Are we good? OK.

So you guys, I walk away from lecture thinking I've given this spectacular lecture and everybody understands. And then Victor tells me when he shows up in section

in the morning, he says did people understand graphs? And everyone says no. Or did people understand x? And people say no.

So at least tomorrow, tell Victor that you understood. Whether you did or not. So then I feel better.

AUDIENCE: That's going to make my life real easy.

PROFESSOR: Yeah, right. So good. Well, you probably like hearing stuff from Victor better than me anyway. That's the secret here, right?

All right, so one of the reasons why you don't want to implement this algorithm is precisely this example that I put up. And this is a really neat example that I like a lot, because it points out two different things. It points out that exponential number of paths, an exponential number of paths in a graph, could cause a problem with this algorithm.

The other thing that it points out is that we got issues with the weights of edges. One of the nice observations one of you made earlier on is that we had these neat algorithms that did not depend on the dynamic range of the weights.

So let's just say that I in fact had an exponential range for the weights. I know 4 isn't exponential, but at some level, you could imagine that it's exponentially related to 1 or 2. And the point here is that if I created a graph that looked like this, where I have V4, V5, V6, V7, V8, and it had this structure, then I'm going to end up having something like 2 raised to n over 2 weight if I have n vertices in this graph. Or at least the dynamic range of these weights is going to be 2 raised to n divided by 2. Everybody buy that?

So think of this graph as being a fragment of this large graph, which where n could be 100 and the weights could be 2 raised to 50. And 2 raised to 50 isn't a number that we can't handle on a computer, right? It's still less than 64 bits, right? So it's a pretty reasonable example.

And we talked about multiple precision arithmetic, infinite precision arithmetic. So we

can handle arbitrary numbers of an arbitrary position. So there's nothing that's stopping us from putting square root of 2 and all sorts of things. We won't do imaginary numbers. But you could imagine putting numbers with a high dynamic range as edges in a particular graph and expect the Dijkstra, assuming that all of the edges are non-negative, that Dijkstra should be able to run on it.

So what happens with this example? Well, with this example, here's what happens. Let's say that I ran this algorithm. And initially, I just followed this chain here. And I get-- this starts with a 0. And this is a 4, because I get there with 4. This one is 8. And this is 10. And this is 12, 13, 14.

And that's the initial pass. That's the selection. What ends up happening is that you could now relax at this-- you see 14. And let's say you relax this edge. You see that 12 and 14, you've turned that into 13.

And then when you relax this edge, this turns into 12. So you go through that process. Now, this one stays 12. But now you relax this edge.

And so this 12 becomes 10. And then when this changes, you need to-- if you relax this edge first, then this 13 is going to become 11. It doesn't really matter. This becomes, I guess, 11.

And-- is that right? Yup. This is 11 and that's 11 as well. It might start out being 12 if you relax this edge and that edge. So you might go to 12 to 11, and so on and so forth.

So for a pathological ordering, I won't belabor the point. But you see that you're going 14, 13, 12, 11 with a bad ordering that corresponds to the selection of the edges. And so if the overall weight here and overall weight here, when you start out with, is going to be order 2 raised to n over 2. OK And you could be, in this particular graph, relaxing edges an exponential number of times in order to finish.

And so the number of times you relax an edge could be of the order of the weights that you start out with. And that makes this algorithm an exponential time algorithm. So clearly, we have to do better than that when it comes to Dijkstra or Bellman-

Ford.

So how are we going to do better than that? Yeah, question back there.

AUDIENCE: Is it an issue that we're starting at the [INAUDIBLE]?

PROFESSOR: You're exactly right. There's an issue with the ordering that we've chosen. But what you have to show is that for any graph, the particular ordering that you choose will result in V log V plus E and so on and so forth. So you're exactly right.

I mean, it's an issue with the ordering we've chosen. This is a pathological ordering. It's just meaning to say that we have to be careful about how we select. If you selected wrong, you've got problems.

And so the purpose of next week is going to be how do we select these edges properly. And so I leave you with this notion of, very simple notion of, optimal substructure using two very simple terms that you can prove in literally a line of text. And the first one says as subpaths of a shortest path are shortest paths.

And all that means is if I had V0, and I went to V1, and I went to V2, and these are paths here. So this could be p01, p02, p03. And so there are many vertices potentially between V0 and V1. And if you tell me that V0 through V3, the concatenation of p01, p02, and, sorry, p03 are a shortest path. If this is an SP, shortest path, then that implies that each of these are shortest paths as well.

And that makes sense, because if in fact there was a better way of getting from V0 to V1 that was better than p01, why would you ever put p01 in here? You would use that better way. So very simple. That's what's called the optimum substructure property.

And this notion of the triangle inequality is also related to that. And that simply says that if I have something like this, that I have V0, V1, and V2, then when I look at the delta value of V0, V1, and I compare that with the delta values of V0, V2, and V2, V1, then this has got to be smaller than or equal to this plus that.

And that again makes sense. Because if this plus this was smaller than that, well remember I'm talking about paths here, not edges. And the better way of getting to V1 would be to follow-- go through V2 rather than following this path up on top.

Amazingly, these two notions are going to be enough to take this algorithm and turn it into essentially a linear time algorithm. And we'll do that next time.