PROFESSOR: We're going to start a brand new, exciting topic, dynamic programming.

AUDIENCE: Yes!

PROFESSOR: Yeah! So exciting. Actually, I am really excited because dynamic programming is my favorite thing in the world, in algorithms. And it's going to be the next four lectures, it's so exciting. It has lots of different facets. It's a very general, powerful design technique. We don't talk a lot about algorithm design in this class, but dynamic programming is one that's so important. And also takes a little while to settle in. We like to injected it into you now, in 006.

So in general, our motivation is designing new algorithms and dynamic programming, also called DP, is a great way-- or a very general, powerful way to do this.

It's especially good, and intended for, optimization problems, things like shortest paths. You want to find the best way to do something. Shortest path is you want to find the shortest path, the minimum-length path. You want to minimize, maximize something, that's an optimization problem, and typically good algorithms to solve them involve dynamic programming. It's a bit of a broad statement.

You can also think of dynamic programming as a kind of exhaustive search. Which is usually a bad thing to do because it leads to exponential time. But if you do it in a clever way, via dynamic programming, you typically get polynomial time.

So one perspective is that dynamic programming is approximately careful brute force. It's kind of a funny combination. A bit of an oxymoron. But we take the idea of brute force, which is, try all possibilities and you do it carefully and you get it to

polynomial time. There are a lot of problems where essentially the only known polynomial time algorithm is via dynamic programming. It doesn't always work, there's some problems where we don't think there are polynomial time algorithms, but when it's possible DP is a nice, sort of, general approach to it.

And we're going to be talking a lot about dynamic programming. There's a lot of different ways to think about it. We'll look at a few today. We're going to warm up today with some fairly easy problems that we already know how to solve, namely computing Fibonacci numbers. It's pretty easy. And computing shortest paths. And then in the next three lectures we're going to get to more interesting examples where it's pretty surprising that you can even solve the problem in polynomial time.

Probably the first burning question on your mind, though, is why is it called dynamic programming? What does that even mean? And I used to have this spiel about, well, you know, programming refers to the-- I think it's the British notion of the word, where it's about optimization. Optimization in American English is something like programming in British English, where you want to set up the program-- the schedule for your trains or something, where programming comes from originally.

But I looked up the actual history of, why is it called dynamic programming. Dynamic programming was invented by a guy named Richard Bellman. You may have heard of Bellman in the Bellman-Ford algorithm. So this is actually the precursor to Bellman-Ford. And we're going to see Bellman-Ford come up naturally in this setting.

So here's a quote about him. It says, Bellman explained that he invented the name dynamic programming to hide the fact that he was doing mathematical research. He was working at this place called Rand, and under a secretary of defense who had a pathological fear and hatred for the term research. So he settled on the term dynamic programming because it would be difficult to give a pejorative meaning to it. And because it was something not even a congressman could object to. Basically, it sounded cool.

So that's the origin of the name dynamic programming. So why is the called that?

Who knows. I mean, now you know. But it's not-- it's a weird term. Just take it for what it is. It may make some kind of sense, but--

All right. So we are going to start with this example of how to compute Fibonacci numbers. And maybe before we actually start I'm going to give you a sneak peak of what you can think of dynamic programming as. And this equation, so to speak, is going to change throughout today's lecture. In the end we'll settle on a sort of more accurate perspective.

The basic idea of dynamic programming is to take a problem, split it into subproblems, solve those subproblems, and reuse the solutions to your subproblems. It's like a lesson in recycling. So we'll see that in Fibonacci numbers.

So you remember Fibonacci numbers, right? The number of rabbits you have on day n, if they reproduce. We've mentioned them before, we're talking about AVL trees, I think. So this is the usual-- you can think of it as a recursive definition or recurrence on Fibonacci numbers. It's the definition of what the nth Fibonacci number is.

So let's suppose our goal-- an algorithmic problem is, compute the nth Fibonacci number. And I'm going to assume here that that fits in a word. And so basic arithmetic, addition, whatever's constant time per operation. So how do we do it? You all know how to do it. Anyways-- but I'm going to give you the dynamic programming perspective on things. So this will seem kind of obvious, but it is-- we're going to apply exactly the same principles that we will apply over and over in dynamic programming. But here it's in a very familiar setting.

So we're going to start with the naive recursive algorithm. And that is, if you want to compute the nth Fibonacci number, you check whether you're in the base case. I'm going to write it this way. So f is just our return value. You'll see why I write it this way in a moment. Then you return f. In the base case it's 1, otherwise you recursively call Fibonacci of n minus 1. You recursively call Fibonacci of n minus 2. Add them together, return that. This is a correct algorithm.

Is it a good algorithm? No. It's very bad. Exponential time. How do we know it's exponential time, other than from experience? Well, we can write the running time as recurrence. T of n represents the time to compute the nth Fibonacci number. How can I write the recurrence? You're gonna throwback to the early lectures, divide and conquer. I hear whispers. Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. T of n minus 1 plus t of n minus 2 plus constant. I don't know how many you have by now. So to create the nth Fibonacci number we have to compute the n minus first Fibonacci number, and the n minus second Fibonacci number. That's these two recursions. And then we take constant time otherwise. We do constant number of additions, comparisons. Return all these operations-- take constant time. So that's a recurrence.

How do we solve this recurrence? Well one way is to see this is the Fibonacci recurrence. So it's the same thing. There's this plus whatever. But in particular, this is at least the nth Fibonacci number. And if you know Fibonacci stuff, that's about the golden ratio to the nth power. Which is bad. We had a similar recurrence in AVL trees.

And so another way to solve it-- it's just good review-- say, oh well, that's at least 2 times t of n minus 2. Because it's going to be monotone. The bigger n is, the more work you have to do. Because to do the nth thing you have to do the n minus first thing. So we could just reduce t of n minus 1 to t of n minus 2. That will give us a lower bound.

And now these two terms-- now this is sort of an easy thing. You see that you're multiplying by 2 each time. You're subtracting 2 from n each time. How many times can I subtract 2 from n? N/2 times, before I get down to a constant. And so this is equal to 2 to the n over 2-- I mean, times some constant, which is what you get in the base case. So I guess I should say theta. This thing is theta that. OK. So it's at least that big. And the right constant is phi. And the base of the exponent.

OK. So that's a bad algorithm. We all know it's a bad algorithm. But I'm going to give you a general approach for making bad algorithms like this good. And that general approach is called memoization. We'll go over here. And this is a technique of dynamic programming.

So I'm going to call this the memoized dynamic programming algorithm. So did I settle on using memo in the notes? Yeah. The idea is simple.

Whenever we compute a Fibonacci number we put it in a dictionary. And then when we need to compute the nth Fibonacci number we check, is it already in the dictionary? Did we already solve this problem? If so, return that answer. Otherwise, computer it. You'll see the transformation is very simple.

OK. These two lines are identical to these two lines. So you can see how the transformation works in general. You could do this with any recursive algorithm. The memoization transformation on that algorithm-- which is, we initially make an empty dictionary called memo. And before we actually do the computation we say, well, check whether this version of the Fibonacci problem, computing f of n, is already in our dictionary.

So if that key is already in the dictionary, we return the corresponding value in the dictionary. And then once we've computed the nth Fibonacci number, if we bothered to do this, if this didn't apply, then we store it in the memo table. So we say well, if you ever need to compute f of n again, here it is. And then we return that value.

So this is a general procedure. It can apply to any recursive algorithm with no side effects I guess, technically. And it turns out, this makes the algorithm efficient. Now there's a lot of ways to see why it's efficient. In general, maybe it's helpful to think about the recursion tree.

So if you want to compute fn in the old algorithm, we compute fn minus 1 and fn minus two completely separately. To compute fn minus 1 we compute fn minus 2 and fn minus 3. To compute fn minus 2 we compute fn minus 3 and fn minus 4. And so on. And you can see why that's exponential in n. Because we're only

decrementing n by one or two each time.

But then you observe, hey, these fn minus 3's are the same. I should really only have to compute them once. And that's what we're doing here. The first time you call fn minus 3, you do work. But once it's done and you go over to this other recursive call, this will just get cut off. There's no tree here. Here we might have some recursive calling. Here we won't, because it's already in the memo table. In fact, this already happens with fn minus 2. This whole tree disappears because fn minus 2 has already been done.

OK. So it's clear why it improves things. So in fact you can argue that this call will be free because you already did the work in here. But I want to give you a very particular way of thinking about why this is efficient, which is following.

So you could write down a recurrence for the running time here. But in some sense recurrences aren't quite the right way of thinking about this because recursion is kind of a rare thing. If you're calling Fibonacci of some value, k, you're only going to make recursive calls the first time you call Fibonacci of k. Because henceforth, you've put it in the memo table you will not recurse.

So you can think of there being two versions of calling Fibonacci of k. There's the first time, which is the non-memoized version that does recursion-- does some work. And then every time henceforth you're doing memoized calls of Fibonacci of k, and those cost constant time. So the memoized calls cost constant time. So we can think of them as basically free.

That's when you call Fibonacci of n minus 2, because that's a memoized call, you really don't pay anything for it. I mean, you're already paying constant time to do addition and whatever. So you don't have to worry about the time. There's no recursion here. And then what we care about is that the number of non-memorized calls, which is the first time you call Fibonacci of k, is n. No theta is even necessary.

We are going to call Fibonacci of 1. At some point we're going to call Fibonacci of 2 at some point, and the original call is Fibonacci of n. All of those things will be called

at some point. That's pretty easy to see. But in particular, certainly at most this, we never call Fibonacci of n plus 1 to compute Fibonacci of n. So it's at most n calls. Indeed it will be exactly n calls that are not memoized. Those ones we have to pay for.

How much do we have to pay? Well, if you don't count the recursion-- which is what this recurrence does-- if you ignore recursion then the total amount of work done here is constant. So I will say the non-recursive work per call is constant. And therefore I claim that the running time is constant-- I'm sorry, is linear. Constant would be pretty amazing.

This is actually not the best algorithm-- as an aside. The best algorithm for computing the nth Fibonacci number uses log n arithmetic operations. So you can do better, but if you want to see that you should take 6046.

OK. We're just going to get to linear today, which is a lot better than exponential. So why linear? Because there's n non-memoize calls, and each of them cost constant. So it's the product of those two numbers. This is an important idea. And it's so important I'm going to write it down again in a slightly more general framework.

In general, in dynamic programming-- I didn't say why it's called memoization. The idea is you have this memo pad where you write down all your scratch work. That's this memo dictionary. And to memoize is to write down on your memo pad. I didn't make it up. Another crazy term. It means remember. And then you remember all the solutions that you've done. And then you reuse those solutions.

Now these solutions are not really a solution to the problem that I care about. The problem I care about is computing the nth Fibonacci number. To get there I had to compute other Fibonacci numbers. Why? Because I had a recursive formulation.

This is not always the way to solve a problem. But usually when you're solving something you can split it into parts, into subproblems, we call them. They're not always of the same flavor as your original goal problem, but there's some kind of related parts.

And this is the big challenge in designing a dynamic program, is to figure out what are the subproblems. Let's say, the first thing I want to know about a dynamic program, is what are the subproblems. Somehow they are designed to help solve your actual problem.

And the idea of memoization is, once you solve a subproblem, write down the answer. If you ever need to solve that same problem again you reuse the answer. So that is the core idea. And so in this sense dynamic programming is essentially recursion plus memoization.

And so in this case these are the subproblems. Fibonacci of 1 through Fibonacci of n. The one we care about is Fibonacci of n. But to get there we solve these other subproblems. In all cases, if this is the situation-- so for any dynamic program, the running time is going to be equal to the number of different subproblems you might have to solve, or that you do solve, times the amount of time you spend per subproblem.

OK. In this situation we had n subproblems. And for each of them we spent constant time. And when I measure the time per subproblem which, in the Fibonacci case I claim is constant, I ignore recursive calls. That's the key. We don't have to solve recurrences with dynamic programming. Yay. No recurrences necessary. OK. Don't count recursions.

Obviously, don't count memoized recursions. The reason is, I only need to count them once. After the first time I do it, it's free. So I count how many different subproblems do I need to do? These are they going to be the expensive recursions where I do work, I do some amount of work, but I don't count the recursions because otherwise I'd be double counting. I only want to count each subproblem once, and then this will solve it.

So a simple idea. In general, dynamic programming is a super simple idea. It's nothing fancy. It's basically just memoization. There is one extra trick we're going to pull out, but that's the idea.

All right. Let me tell you another perspective. This is the one maybe most commonly taught. Is to think of-- but I'm not a particular fan of it. I really like memoization. I think it's a simple idea. And as long as you remember this formula here, it's really easy to work with. But some people like to think of it this way. And so you can pick whichever way you find most intuitive.

Instead of thinking of a recursive algorithm, which in some sense starts at the top of what you want to solve and works its way down, you could do the reverse. You could start at the bottom and work your way up. And this is probably how you normally think about computing Fibonacci numbers or how you learned it before. I'm going to write it in a slightly funny way.

The point I want to make is that the transformation I'm doing from the naive recursive algorithm, to the memoized algorithm, to the bottom-up algorithm is completely automated. I'm not thinking, I'm just doing. OK. It's easy. This code is exactly the same as this code and as that code, except I replaced n by k. Just because I needed a couple of different n values here. Or I want to iterate over n values.

And then there's this stuff around that code which is just formulaic. A little bit of thought goes into this for loop, but that's it. OK. This does exactly the same thing as the memoized algorithm. Maybe it takes a little bit of thinking to realize, if you unroll all the recursion that's happening here and just write it out sequentially, this is exactly what's happening.

This code does exactly the same additions, exactly the same computations as this. The only difference is how you get there. Here we're using a loop, here we're using recursion. But the same things happen in the same order. It's really no difference between the code. This code's probably going to be more efficient practice because you don't make function calls so much.

In fact I made a little mistake here. This is not a function call, it's just a lookup into a table. Here I'm using a hash table to be simple, but of course you could use an

array. But they're both constant time with good hashing. All right. So is it clear what this is doing? I think so. I think I made a little typo.

So we have to compute-- oh, another typo. We have to compute f1 up to fn, which in python is that. And we compute it exactly how we used to. Except now, instead of recursing, I know that when I'm computing the k Fibonacci number-- man. So many typos.

AUDIENCE: [LAUGHTER]

PROFESSOR: You guys are laughing. When I compute the kth Fibonacci number I know that I've already computed the previous two. Why? Because I'm doing them in increasing order. Nothing fancy.

Then I can just do this and the solutions will just be waiting there. If they work, I'd get a key error. So I'd know that there's a bug. But in fact, I won't get a key error. I will have always computed these things already. Then I store it in my table. Then I iterate. Eventually I've solved all the subproblems, f1 through fn. And the one I cared about was the nth one.

OK. So straightforward. I do this because I don't really want to have to go through this transformation for every single problem we do. I'm doing it in Fibonacci because it's super easy to write the code out explicitly. But you can do it for all of the dynamic programs that we cover in the next four lectures.

OK. I'm going to give you now the general case. This was the special Fibonacci version. In general, the bottom-up does exactly the same computation as the memoized version. And what we're doing is actually a topological sort of the subproblem dependency DAG.

So in this case, the dependency DAG is very simple. In order to compute-- I'll do it backwards. In order to compute fn, I need to know fn minus 1 and fn minus 2. If I know those I can compute fn. Then there's fn minus 3, which is necessary to compute this one, and that one, and so on. So you see what this DAG looks like.

Now, I've drawn it conveniently so all the edges go left to right. So this is a topological order from left to right. And so I just need to do f1, f2, up to fn in order. Usually it's totally obvious what order to solve the subproblems in. But in general, what you should have in mind is that we are doing a topological sort. Here we just did it in our heads because it's so easy. And usually it's so easy. It's just a for loop. Nothing fancy.

All right. I'm missing an arrow. All right. Let's do something a little more interesting, shall we? All right. One thing you can do from this bottom-up perspective is you can save space. Storage space in the algorithm. We don't usually worry about space in this class, but it matters in reality.

So here we're building a table size, n, but in fact we really only need to remember the last two values. So you could just store the last two values, and each time you make a new one delete the oldest. so by thinking a little bit here you realize you only need constant space. Still linear time, but constant space. And that's often the case. From the bottom-up perspective you see what you really need to store, what you need to keep track of.

All right. I guess another nice thing about this perspective is, the running time is totally obvious. This is clearly constant time. So this is clearly linear time. Whereas, in this memoized algorithm you have to think about, when's it going to be memoized, when is it not? I still like this perspective because, with this rule, just multiply a number of subproblems by time per subproblem, you get the answer. But it's a little less obvious than code like this. So choose however you like to think about it.

All right. We move onto shortest paths. So I'm again, as usual, thinking about single-source shortest paths. So we want to compute the shortest pathway from s to v for all v. OK. I'd like to write this initially as a naive recursive algorithm, which I can then memoize, which I can then bottom-upify. I just made that up.

So how could I write this as a naive recursive algorithm? It's not so obvious. But first I'm going to tell you how, just as an oracle tells you, here's what you should do. But

then we're going to think about-- go back, step back. Actually, it's up to you. I could tell you the answer and then we could figure out how we got there, or we could just figure out the answer. Preferences? Figure it out. All right. Good. No divine inspiration allowed.

So let me give you a tool. The tool is guessing. This may sound silly, but it's a very powerful tool. The general idea is, suppose you don't know something but you'd like to know it. So what's the answer to this question? I don't know. Man, I really want a cushion. How am I going to answer the question? Guess. OK?

AUDIENCE: [LAUGHTER]

PROFESSOR: It's a tried and tested method for solving any problem. I'm kind of belaboring the point here. The algorithmic concept is, don't just try any guess. Try them all. OK?

AUDIENCE: [LAUGHTER]

PROFESSOR: Also pretty simple. I said dynamic programming was simple. OK. Try all guesses. This is central to the dynamic programming. I know it sounds obvious, but if I want to fix my equation here, dynamic programming is roughly recursion plus memoization. This should really be, plus guessing. Memoization, which is obvious, guessing which is obvious, are the central concepts to dynamic programming. I'm trying to make it sound easy because usually people have trouble with dynamic programming. It is easy.

Try all the guesses. That's something a computer can do great. This is the brute force part. OK. But we're going to do it carefully. Not that carefully. I mean, we're just trying all the guesses. Take the best one. That's kind of important that we can choose one to be called best. That's why dynamic programming is good for optimization problems. You want to maximize something, minimize something, you try them all and then you can forget about all of them and just reduce it down to one thing which is the best one, or a best one.

OK. So now I want you to try to apply this principle to shortest paths. Now I'm going to draw a picture which may help. We have the source, s, we have some vertex, v.

We'd like to find the shortest-- a shortest path from s to v.

Suppose I want to know what this shortest path is. Suppose this was it. You have an idea already? Yeah.

AUDIENCE: What you could do is you could look at everywhere you can go from s. [INAUDIBLE] shortest path of each of those notes.

PROFESSOR: Good. So I can look at all the places I could go from s, and then look at the shortest paths from there to v. So we could call this s prime. So here's the idea. There's some hypothetical shortest path. I don't know where it goes first, so I will guess where it goes first. I know the first edge must be one of the outgoing edges from s. I don't know which one. Try them all. Very simple idea. Then from each of those, if somehow I can compute the shortest path from there to v, just do that and take the best choice for what that first edge was. So this would be the guess first edge approach.

It's a very good idea. Not quite the one I wanted because unfortunately that changes s. And so this would work, it would just be slightly less efficient if I'm solving single-source shortest paths. So I'm going to tweak that idea slightly by guessing the last edge instead of the first edge. They're really equivalent. If I was doing this I'd essentially be solving a single-target shortest paths, which we talked about before.

So I'm going to draw the same picture. I want to get to v. I'm going to guess the last edge, call it uv. I know it's one of the incoming edges to v-- unless s equals v, then there's a special case. As long as this path has length of at least 1, there's some last edge. What is it? I don't know. Guess. Guess all the possible incoming edges to v, and then recursively compute the shortest path from s to u. And then add on the edge v.

OK. So what is this shortest path? It's delta of s comma u, which looks the same. It's another subproblem that I want to solve. There's v subproblems here I care about. . So that's good. I take that. I add on the weight of the edge uv. And that should

hopefully give me delta of s comma v.

Well, if I was lucky and I guessed the right choice of u. In reality, I'm not lucky. So I have to minimize over all edges uv. So this is the-- we're minimizing over the choice of u. V is already given here. So I take the minimum over all edges of the shortest path from s to u, plus the weight of the edge uv. That should give me the shortest path because this gave me the shortest path from s to u. Then I added on the edge I need to get there.

And wherever the shortest path is, it uses some last edge, uv. There's got to be some choice of u that is the right one. That's the good guess that we're hoping for. We don't know what the good guess is so we just try them all. But whatever it is, this will be the weight of that path. It's going to take the best path from s to u because sub paths are shortest paths are shortest paths. Optimal substructure. So this part will be delta of su. This part is obviously w of uv. So this will give the right answer. Hopefully. OK. It's certainly going to-- I mean, this is the analog of the naive recursive algorithm for Fibonacci. So it's not going to be efficient if I-- I mean, this is an algorithm, right? You could say-- this is a recursive call. We're going to treat this as recursive call instead of just a definition. Then this is a recursive algorithm. How good or bad is this recursive algorithm?

AUDIENCE: Terrible.

PROFESSOR: Terrible. Very good. Very bad, I should say. It's definitely going to be exponential without memoization. But we know. We know how to make algorithms better. We memoize. OK. So I think you know how to write this as a memoized algorithm.

To define the function delta of sv, you first check, is s comma v in the memo table? If so return that value. Otherwise, do this computation where this is a recursive call and then stored it in the memo table. OK. I don't think I need to write that down. It's just like the memoized code over there. Just there's now two arguments instead of one. In fact, s isn't changing. So I only need to store with v instead of s comma v.

Is that a good algorithm? I claim memoization makes everything faster. Is that a fast

algorithm? Not so obvious, I guess. Yes? How many people think, yes, that's a good algorithm?

AUDIENCE: Better.

PROFESSOR: Better. Definitely better. Can't be worse. How many people think it's a bad algorithm still? OK. So three for yes, zero for no. How many people aren't sure? Including the yes votes? Good.

All right. It's not so tricky. Let me draw you a graph. Something like that. So we wanted to commit delta of s comma v. Let me give these guys names, a and b. So we compute delta of s comma v. To compute that we need to know delta of s comma a and delta of s comma v. All right? Those are the two ways-- sorry, actually we just need one. Only one incoming edge to v. So its delta of s comma a. Sorry-- I should have put a base case here too. Delta of s comma s equals 0.

OK. Delta of s comma a plus the edge. OK. There is some shortest path to a. To compute the shortest path to a we look at all the incoming edges to a. There's only one. So delta of s comma b. Now I want to compute the shortest paths from b. Well, there's two ways to get to b. One of them is delta of s comma b-- sorry, s comma s. Came from s. The other way is delta of s comma v. Do you see a problem? Yeah. Delta of s comma v is what we were trying to figure out.

Now you might say, oh, it's OK because we're going to memoize our answer to delta s comma v and then we can reuse it here. Except, we haven't finished computing delta of s comma v. We can only put it in the memo table once we're done. So when this call happens the memo table has not been set. And we're going to do the same thing over and over and over again. This is an infinite algorithm. Oops. Not so hot. So it's going to be infinite time on graphs with cycles.

OK. For DAGs, for acyclic graphs, it actually runs in v plus e time. This is the good case. In this situation we can use this formula. The time is equal to the number of subproblems times the time per subproblem. So I guess we have to think about that a little bit. Where's my code? Here's my code. Number of subproblems is v. There's

v different subproblems that I'm using here. I'm always reusing subproblems of the form delta s comma something. The something could be any of the v vertices.

How much time do I spend per subproblem? That's a little tricky. It's the number of incoming edges to v. So time for a sub problem delta of sv is the indegree of v. The number of incoming edges to v. So this depends on v. So I can't just take a straightforward product here. What this is really saying is, you should sum up over all sub problems of the time per sub problem.

So total time is the sum over all v and v, the indegree of v. And we know this is number of edges. It's really-- so indegree plus 1, indegree plus 1. So this is v plus v. OK. Handshaking again.

OK. Now we already knew an algorithm for shortest paths and DAGs. And it ran a v plus e time. So it's another way to do the same thing. If you think about it long enough, this algorithm memoized, is essentially doing a depth first search to do a topological sort to run one round of Bellman-Ford. So we had topological sort plus one round of Bellman-Ford. This is kind of it all rolled into one. This should look kind of like the Bellman Ford relaxation step, or shortest paths relaxation step. It is. This min is really doing the same thing. So it's really the same algorithm. But we come at it from a different perspective.

OK. But I claim I can use this same approach to solve shortest paths in general graphs, even when they have cycles. How am I going to do that? DAGs seem fine-- oh, what was the lesson learned here? Lesson learned is that subproblem dependencies should be acyclic. Otherwise, we get an infinite algorithm. For memoization to work this is what you need. It's all you need.

OK. We've almost seen this already. Because I said that, to do a bottom up algorithm you do a topological sort of this subproblem dependency DAG. I already said it should be acyclic. OK. We just forgot. I didn't tell you yet. So for that to work it better be acyclic. For DP to work, for memoization to work, it better be acyclic. If you're acyclic then this is the running time. So that's all general.

OK. So somehow I need to take a cyclic graph and make it acyclic. We've actually done this already in recitation. So if I have a graph-- let's take a very simple cyclic graph. OK. One thing I could do is explode it into multiple layers. We did this on quiz two in various forms. It's like the only cool thing you can do with shortest paths, I feel like. If you want to make a shortest path problem harder, require that you reduce your graph to k copies of the graph.

I'm going to do it in a particular way here-- which I think you've seen in recitation-- which is to think of this axis as time, or however you want, and make all of the edges go from each layer to the next layer. This should be a familiar technique. So the idea is, every time I follow an edge I go down to the next layer. This makes any graph acyclic. Done. What in the world does this mean? What is it doing? What does it mean? Double rainbow. All right.

AUDIENCE: [LAUGHTER]

PROFESSOR: So-- I don't know how I've gone so long in the semester without referring to double rainbow. It used to be my favorite. All right.

So here's what it means. Delta sub k of sv. I'm going to define this first-- this is a new kind of subproblem-- which is, what is the shortest-- what is the weight of the shortest s to v path that uses, at most, k edges.

So I want it to be shortest in terms of total weight, but I also want it to use few edges total. So this is going to be 0. In some sense, if you look at-- so here's s and I'm always going to make s this. And then this is going to be v in the zero situation. This is going to be v in the one situation, v-- so if I look at this v, I look at the shortest path from s to v, that is delta sub 0 of sv. So maybe I'll call this v sub 0, v sub 1, v sub 2.

OK. Shortest path from here to here is, there's no way to get there on 0 edges. Shortest path from here to here, that is the best way to get there with, at most, one edge. Shortest path from here to here-- well, if I add some vertical edges too, I guess, cheating a little bit. Then this is the best way to get from s to v using at most

two edges. And then you get a recurrence which is the min over all last edges. So I'm just copying that recurrence, but realizing that the s to u part uses one fewer edge. And then I use the edge uv.

OK. That's our new recurrence. By adding this k parameter I've made this recurrence on subproblems acyclic. Unfortunately, I've increased the number of subproblems. The number of subproblems now is v squared. Technically, v times v minus 1. Because I really-- actually, v squared. Sorry. I start at 0. And what I care about, my goal, is delta sub v minus 1 of sv. Because by Bellman-Ford analysis I know that I only care about simple paths, paths of length at most v minus 1. I'm assuming here no negative weight cycles. I should've said that earlier.

If you assume that, then this is what I care about. So k ranges from 0 to v minus 1. So there are v choices for k. There are v choices for v. So the number of subproblems is v squared. How much time do I spend per subproblem? Well, the same as before. The indegree-- where did I write it? Up here-- the indegree of that problem. So what I'm really doing is summing over all v of the indegree. And then I multiply it by v. So the running time, total running time is ve. Sound familiar? This is Bellman-Ford's algorithm again. And this is actually where Bellman-Ford algorithm came from is this view on dynamic programming. So we're seeing yet another way to do Bellman-Ford. It may seem familiar. But in the next three lectures we're going to see a whole bunch of problems that can succumb to the same approach. And that's super cool.