PROFESSOR: Today, we are going to do computational complexity. This is rather different from every other thing we've seen in this class. This class is basically about polynomial time algorithms and problems where we can solve your problem in polynomial time. And today, it's about when you can't do that. Sometimes, we can prove you can't do that. Sometimes, we're pretty sure you can't do that. But it's all about negative results when your problems are really complex.

And there's a lot of fun topics, here. This is the topic of entire classes, like 6045. We're just going to get a 1 hour flavor of it. So think of it as a high level intro. But we're going to prove real theorems and do real things and you'll get a sense of how all this works.

So I'm going to start out with three complexity classes-- P, EXP, and R. How many people know what P is? And it is? Polynomial time. More precisely, it's the set of all problems you can solve in polynomial time. This is what the class is all about. Almost every problem we have seen in this class-- there's one exception-- is in P. Does anyone know the exception? It's a good puzzle for you. Not NP.

What's next? EXP. How many people know what EXP is? Or you can guess. Any guesses? Exponential. These are all the problems you can solve in exponential time.

If you want to be formal about it, in this case, exponential means 2 to the n to some constant. So not just 2 the n, but also 2 to the n squared, 2 to the n cubed. Those are all considered-- exponential and a polynomial is considered in the class EXP.

Now, basically, almost every problem you can dream of you can solve in EXP. Exponential time is so much time. And this class has always been about taking

things that are obviously in EXP and showing that they're actually in P. So if you want to draw a picture, you could say, OK, here's all the problems we can solve in polynomial time.

Here's all the problems we can solve in exponential time. And there are problems out here. These are different classes. And we want to sort of bring things into here as much as possible.

I actually want to draw this picture in a different way, which is as a horizontal line. So an axis. I'm going to call this computational difficulty. You could call it computational complexity, but that's a bit of a loaded term that actually has formal meaning. Difficulty is nice and vague. So I can draw an abstract picture. This is not a true diagram, but it's a very good guideline of what's going on.

So we have-- I'm going to draw-- I believe-- three notches. No, eventually four, so let me give myself some room. We have over here, the easy problems are P. Then, we have these problems, which are EXP. We're going to fill in something in the middle. And then this is something called R. So you've got P is everything, here. EXP is all the way out to here, in some abstract view.

The next thing is R. How many people know what R is? This one, I had to look up. It's not usually given a name. No one. Teaching staff? You guys know it? These are all problems solvable in finite time. R stands for finite. R stands for recursive. Recursive used to mean something completely different, back in the '30s, when people were thinking about what's computable, what's not computable.

These are, basically, solvable problems, computable problems. Finite time is a reasonable requirement, I think, for all algorithms. And that's R. Now, I've drawn this arrow to keep going because there are problems out here. It's kind of discouraging, but there are problems that are unsolvable. In fact, most problems are unsolvable. We're going to prove that. It's actually really easy to prove. Kind of depressing, but true.

Let me start with some examples before we get to that proof. So I'm writing

examples of some things we've seen. So here's an example of a problem we've seen. Negative-weight cycle detection.

I give you a graph-- a weighted graph. I want to know does it have any negative-weight cycles? What classes is this problem in? P. We know how to solve this in polynomial time-- in VE time-- using Bellman-Ford. VE time-- well, that finds negative-weight cycles reachable from s. But, I guess, if you add a source that can reach anywhere-- zero weight-- then that'll tell you overall that it's in P.

It's also in EXP, of course. Everything in P is also in EXP. Because if you can solve it in polynomial time, you can solve it in exponential time. This is at most exponential time. At most polynomial.

Here's a problem we haven't seen. But it's pretty cool. N by n Chess. So this is the problem I give you. So we're in an by n board, and I give you a whole bunch of pieces on the board, and I want to know does White win from here? I say it's White to move or Black to move, and who's going to win form this position?

This problem, can be solved in exponential time. You can sort of play out all possible strategies and see who wins. And it's not in P. There's no polynomial time algorithm to play generalized Chess. This sort of captures why Chess-- even at eight by eight Chess-- is hard-- because there's no general way to do it. So there's no special way to do it, probably.

Computational complexity is all about order of growth. So we can't analyze eight by eight Chess, but we can analyze n by n Chess. And that gives us a flavor of why 8 by 8 is so difficult. Go is also in EXP, but not in P-- lots of games are in this category, lot's of complicated games, let's say.

And so this is a first example of a problem that we know we cannot solve in polynomial time. Bad news. I also talked about Tetris a little bit. Unlike the Tetris training, which we saw, this is sort of realistic Tetris-- all the rules of Tetris. The only catch is that I tell you all the pieces that are going to come in advance. Because, otherwise, it's some random process and it's kind of hard to think about what's the

best strategy.

But if I tell you what's going to come-- say it's a pseudo-random generator and you know how it works. You know all the pieces that will come. I want to know can I survive from a given initial board mess and for a given sequence of pieces.

This can also be solved in exponential time. Just try all the possibilities. We don't know whether it's in P. We're pretty sure it's not in P. And by the end of today's lecture, you'll understand why we think it's not in P. But it's going to be somewhere in between here. Tetris is actually right here. But I haven't defined what right here is yet.

And then the next one is halting problem. So halting problem is particularly cool, as we'll see-- or interesting. It's the problem of given a computer program-- Python, whatever, it doesn't really matter what language. They're all the same in a theoretical sense-- does it ever halt?

Does it ever stop running, return a result, whatever? This would be really handy-- you're writing some code, and you've run it for 5 hours, and you don't know is that because there's a bug and you've got an infinite loop? Or is it just because it's really slow?

So you'd like to give it to some program-- checking program-- that says will this run forever or will it terminate. That's the halting problem. And this problem is not in R. There is no correct algorithm for solving this problem. There's no way to tell, given an arbitrary program, whether it will halt.

Now, in some situations-- take the empty program-- I can tell that it halts. Or I take some special simple class of programs, I can tell whether they halt or determine that they don't halt. But there's no algorithm that solves it for all programs, in finite time. In infinite time, I can solve it. Just run it. Run the program. Given finite time, there's no way to solve this.

And so this is a little bit beyond what we can prove today. It's not that hard to prove, but it takes half an hour or something. I want to get to other things. But if you take

6045, they'll prove this. What I want to show you instead is an easier result-- that almost every problem is not in R.

I need one term, though, which is decision problems. All of these problems, I set it up in a way that the answer is binary-- yes or no. Is there a negative-weight cycle? Yes or no? Does White win from this position in Chess? Can you survive in Tetris? And does this program halt?

For various reasons-- basically convenience-- the whole field of computational complexity focuses on decision problems. And, in fact-- so decision problems are ones where the answer is yes or no. That's all. Why? Essentially because it doesn't matter. If you take a problem you care about, you can convert it into a decision problem. We can see examples of that later. Decision problems are basically as hard as optimization problems or whatever.

But let's focus on decision problems. The answer is yes or no. Claim that most of them are uncomputable. And we can prove this pretty easily if you know a bit of set theory, I guess.

On the one hand, I have problems I want to solve. These are decision problems. And on the other hand, I have algorithms, or computer programs to solve them. I'm going to think of computer programs because more precise algorithms can be a little bit nebulous for thinking about pseudocode-- what's valid, what's invalid.

But computer programs are very clear. I give you some code. You throw it into Python. Either it works or it doesn't. And it does something. Runs for a while. How can I think about the space of all possible programs? Well, programs are things you type into a computer in ASCII, whatever. In the end, you can think of it as just as a binary string. Somehow it gets encoded in binary. Everything is reduced to binary in the end, on a computer.

So this is a binary string. Now, you can also think of a binary string as representing a number, in binary. So you can also think of a program, then, as a natural number-- some number between 0 and infinity. And an integer. So usually we represent this

as math bold N. That's just 0, 1, 2, 3. You can think of every program is ultimately reducing to an integer. It's a big integer, but, hey. It's an integer. So that's the space of all programs.

Now, I want to think about the space of all decision problems. So how can I define a decision problem? Well, the natural way to think of a decision problem is as a function that maps inputs to yes or no. Function from inputs to yes or no. Or you can think of that as 1 and 0.

So what's an input? Well, an input is a binary string. So an input is a number-- a natural number. Input is a binary string, which we can think of as being in N. So we've got a function from N to 0,1.

So another way to represent one of these functions is as a table. I could just write down all the answers. So I've got, well, the input could be 0-- the number 0. And then, maybe it's a 0. Input could be could be 1 and then, maybe, output is 0. Then, the input could be 2, 3, 4, 5, 1, 0, 1, 1, whatever. So I could write the table of all answers. This is another way to write down such a function.

What we have, here, is an infinite string of bits. Each of them could be 0 or 1. It would be a different problem. But they all exist. Any infinite string of bits represents a decision problem. They're the same thing. So a decision problem is an infinite string of bits. A program is a finite string of bits. These are different things.

One way to see that they're different is put a decimal point, here. Now, this infinite string of bits is a number-- a real number-- between 0 and 1. It's written in binary. You may not be used to binary point. This dot is not a decimal point. It's a binary point. But, hey. Any real number can be expressed by an infinite string of bits in this way-- any real number between 0 and 1.

So a decision problem is basically something in R, the set of all real numbers, whereas a program is something in N, the set of all integers. And the thing is, the number of real numbers is much, much bigger than the number of integers. In a formal sense, we call this one uncountably infinite, and this one is countably infinite.

I'm not going to prove that here, today. You may have seen that proof. It's pretty simple.

And that's bad news. That means that there are way more problems than there are programs to solve them. So this means almost every problem that we could conceive of is unsolvable by every program.

And this is pretty depressing the first time I saw it. That's why we put it at the end of the class. I think you get all existential. I mean the thing is every program only solves one problem. It takes some input, and it's either going to output yes or no. And if it's wrong on any of the inputs, then it's wrong. So it's going to give an answer. Say it's a deterministic algorithm. No random numbers or things. Then, there's just not enough programs to go around if each program only solves one problem.

This is the end of the proof. Any questions about that? Kind of weird. Because yet somehow, most of the problems that we think about are computable. I don't know why that is. But mathematically, most problems that you could think of are uncomputable.

Question?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. It's something like, the way that we describe problems is usually almost algorithmic, anyway. And so, usually, most problems we think of are in EXP. And so they're definitely computable. There's some metatheorem about how we think about problems, not just programs.

So that's all I'm going to say about R. So out here, we have halting problem and, actually, most problems. You can think of this as an infinite line and then there's just this small portion which are things you can solve. But we care about this portion because that's the interesting stuff. That's what algorithms are about. Out here kind of nothing happens.

So I want to talk about this notch, which is NP. I imagine you've heard about NP. It's pretty cool, but also kind of confusing. But it's actually very closely related to something we've seen with dynamic programming, which is guessing.

So I'm going to give you a couple of definitions of NP-- not formal definition, but high level definitions. So just like P, EXP, and R, it's a set of decision problems. And it's going to look very similar to P. NP does not stand for not a polynomial. It stands for nondeterministic polynomial. We'll get to nondeterministic in a moment.

The first line is the same. It's all decision problems you can solve in polynomial time. That sounds like P. But then, there's this extra line, which is via a "lucky" algorithm.

Let me tell you-- at a high level what a lucky algorithm does is it can make guesses. But unlike the way that we've been making guesses with dynamic programming-- with dynamic programming we had to guess something. We tried all the possibilities. A lucky algorithm just needs to try one possibility because it's really lucky. It always guesses the right choice. It's like magic. This is not a realistic model of computation, but it is a model of computation called nondeterministic model.

And it's going to sound crazy because it is crazy, but nonetheless it's actually really useful-- even though you could never really build this on a real computer. The nondeterministic model is not a model of real computation. It is a model of theoretical hypothetical computation. It gets at the root-- at the core of what is possible to solve. You'll see why, in a little bit.

So in this model, an algorithm-- it can compute stuff, but, in particular, it makes guesses. So should I do this or should I do this? And it just says-- It doesn't flip a coin. It's not random. It just thinks-- it just makes a guess. Well, I don't know. Let's go this way. And then it comes another fork in the road. It's like, well, I don't know. I'll go this way. That's the guessing. You give it a list of choices and somehow a choice is determined, by magic-- nondeterministic magic.

And then the fun part is-- I should say, at the end the algorithm either says yes or no. It gives you an output. The guesses are guaranteed-- this is the magic part-- to

lead to a yes answer, if possible.

So if you imagine the space of executions of this program, you start here, and you make some guess and you don't know which way to go. In dynamic programming, we try all of them. But this algorithm doesn't try all of them. It's like a branching universe model of the universe.

So you make some choice, and then you make some other choice, and then you make some other choice. All of these are guesses. And some of these things will lead to yes. Some of these things will lead to no. And in this magical model, if there's any yes out there, you will follow a path to a yes. If all of the answers are no, then, of course, it doesn't matter what choices you make. You will output no. But if there's ever a yes, magically these guesses find it.

This is the sense of lucky. If you're trying to find a yes-- that's your goal in life-- then this corresponds to luck. And NP is the class of all problems solvable in polynomial time by a really lucky algorithm. Crazy. I know.

Let's talk about Tetris. Tetris, I claim, is in NP. And we know how to solve it in exponential time. Just try all the options. But, in fact, I don't need to try all the options. It would be enough just use this nondeterministic magic. I could say, well, should I drop the piece here, here, here, here, here, or here. And should it be rotated like this, or like this, or like this, or like this? I don't know. So I guess. And I just place that piece. I make another guess where to place the next piece. Then I make another guess where to place the next piece.

I implement the rules of Tetris, which is if there's a full line it clears. I figure out where these things fall. I can even think about, should I rotate at the last second. If I don't know, I'll guess.

Any choice you have to make in playing Tetris, you can just guess. There's only polynomially many guesses you need to make. So it's still polynomial time. That's important. It's not like we can do anything. But we can make a polynomial number these magic guesses. And then at the end, I determine did I die-- or rather, did I

survive.

It's important, actually. It only works one way. Did I survive? Yes or no? And that's easy to compute. I just see did I ever go above the top row. So what this model says is if there is any way to survive-- if there is any way to get a yes answer, then, my guesses will find it, magically, in this model. Therefore, Tetris is in NP.

If I had instead said, did I die, then, what this algorithm would tell me is there any way to die-- which, the answer's probably yes, unless you're given a really trivial input. So it's important you set up the yes versus no, correctly. But the Tetris decision problem "can I survive," is in NP. The decision problem "can I die," should not be in NP. But we don't know.

Another way to think about NP. And you might find this intuitive because we've been doing lots of guessing. It's just a little crazy. There's another way that's more intuitive to many people. So if this doesn't make sense, don't worry, yet. This is another way to phrase it.

Another way to think about NP-- which turns out to be equivalent-- is that don't think so much about algorithms for solving a problem, just think about algorithms for checking the solution to a problem. It's usually a lot easier to check your work than it is to solve a problem in the first place. And NP is all about that issue. So think of decision problems and think about if you have a solution-- so let's say in Tetris, the solution is yes. In fact, I need to say this, probably.

The more formal version is whenever the answer is yes, you can prove it. And you can check that proof in polynomial time. This is the more formal-- this a little bit high level. What does check mean? Here's what check means.

Whenever an answer is "yes," you can write down a proof that the answer is yes. And someone can come along and check that proof in polynomial time and be convinced that the answer is yes. What does convinced mean?

It's not that hard. Think of it is a two player game. There's me trying to play Tetris, and there's you trying to be convinced that I'm really good at Tetris. It seems a little

one sided, but-- it's a asymmetric game. So you want to prove Tetris is-- I want to show Tetris is in NP. Imagine I'm this magical creature. Actually, it's kind of funny. It reminds me of a story.

On the front of my office door, you may have seen there's an email I received, maybe 15 years ago-- oh no, I guess it can't be that long ago. Must've been about 7 years ago when we proved that Tetris is NP-complete. And the email says, "Dear Sir,"-- or whatever-- "I am NP-complete." We don't what NP-complete means, yet, but it's a meaningless statement. So it doesn't matter that you don't know what it means. It might get funnier throughout the lecture today.

And he's like, I can solve Tetris. I'm really good at playing Tetris. I'm really good at playing Minesweeper-- all these games that are thought to be intractable. He gave me his records and so on. It's like how can I apply my talent. So I will translate what he meant to say was, "I am lucky." And this is probably not true, but he thought that he was lucky. He wanted to convince me he was lucky.

So how could we do it? Well, I could give him a really hard Tetris problem. And say, can you survive these pieces? And he says, "yes, I can survive. " And how does he prove to me that he can survive? Well, he just plays it. He shows me what to do.

So proof is sequence of moves that you make. It's really easy to convince someone that you can survive a given level of Tetris. You just show what the sequence of moves are.

And then I, as a mere mortal polynomial time algorithm can check that that sequence works. I just have to implement the rules of Tetris. So in Tetris, the rules are easy to implement. Its the knowing what thing to do is hard.

But in NP, knowing which way to go is easy. In this version, you don't even talk about how to find the solution. It's just a matter of can you write down a solution that can be checked. Can prove it. This is not in polynomial time. You get arbitrarily much time to prove it. But then, the check has to happen in polynomial time. Kind of clear? That's Tetris.

And every problem that you can solve in polynomial time you can also, of course, check it. Because if you could solve it in polynomial time, you could just solve it and then see did you get the same answer that I did. So P is inside NP. But the big question is does p equal NP. And most people think no. P does not equal NP-- most sane people.

So this is a big problem. It's one of the famous Millennium Prize problems. So in particular, if you solved it, you would get $1 million, and fame, and probably other fortune. You could do TV spots. I think that's how people mostly make their money. You could do a lot. You would become the most famous computer scientist in the world if you prove this.

So a lot of people have tried. Every year, there's an attempt to prove either what everyone believes or, most often, people try to prove the reverse-- that they are equal. I don't know why. They should bet the other way.

So what does P does not equal NP mean? It means that there are problems, here, that are in NP but not in P. Think about what this means. This is saying P are the problems that we can actually solve on a legitimate computer. NP are problems that we can solve in this magical fairy computer where all of our dreams are granted. You say, oh, I don't know which way to go. It doesn't matter because the machine magically tells you which way to go. If you're goal is to get to a yes.

So NP is a really powerful model of computation. It's an insane model of computation. No one in their right mind would consider it legitimate. So obviously, it's more powerful than P, except we don't know how to prove it. Very annoying.

Other phrasings of P does not equal NP is-- these are my phrasings, I them up-- you can't engineer luck. You can believe in luck, if you want. But it's not something that we can build out of a regular computer. That's the meaning of this statement. And so I think most people believe that.

Another phrasing would be that solving problems is harder than checking solutions. A more formal version is that generating solutions or proofs of solutions can be

harder than checking them.

Another phrasing is it's harder to generate a proof of a theorem than it is to check the proof of a theorem. We all know checking the proof of a theorem should be easy if you write it precisely. Just make sure each step follows from the previous ones. Done.

But proving a theorem, that's hard. You need inspiration. You need some clever idea. That's guessing. Inspiration equals luck equals guessing, in this model. And that's hard. The only way we know is to try all the proofs. See which of them work.

So what the heck? What could we possibly say? This is all kind of weird. This would be the end of the lecture if you say, OK, well we don't know. That's it. But thankfully-- I kind of need this board. I also want this one, but I guess I'll go over here.

Fortunately, this is not the end of the story. And we can say a lot about things like Tetris. See I drew Tetris not just in this regime. We're pretty sure Tetris is between NP and P. That it's in NP minus P.

So let me write that down. Tetris is in NP minus P. We don't know that because we don't know-- this could be the empty set. What we do know is that if there's anything in NP minus P-- if they are different, then-- if there's anything in NP minus P, then Tetris is one of those things.

That's why I drew Tetris out there. It is, in a certain sense, the hardest problem in NP. Tetris. Why Tetris? Well, it's not just Tetris. There are a lot of problems right at that little notch. But this is pretty interesting because, while we can't figure this out, most people believe this is true. And so as long as you believe in that-- as long as you have faith-- then you can prove that Tetris is in NP minus P.

And so it's hard. It's not in P, in this case. In particular, not in P. That's kind of cool. How in the world do we prove something like this? It's actually not that hard. I mean it took us several months, but that's just months, whereas this thing has been around since, I guess, the '70s. P versus NP.

Why is this true? Because Tetris is NP-hard. What does NP-hard mean? This means as hard as every problem in NP.

I can't say harder than because it's non-strict. So it's at least as hard as every problem in NP. And that's why I drew it at the far right. It's sort of the hardest extreme of NP. Among everything in NP you can possibly imagine, Tetris is as hard as all of them. And therefore, if there's anything that's harder than P, then Tetris is going to be harder than P because it's as far to the right as possible.

Either P equals NP, in which case the picture is like this. Here's P. Here's NP. Tetris is still at the right extreme, here. But it's less interesting because it's still in P. Or the picture looks like this, and NP is strictly bigger than P. And then, because Tetris is at the right extreme, it's outside of P. So we prove this in order to establish this claim.

Just to get some terminology, what is this NP-complete business? Tetris is NP-complete, which means two things. One is that it's NP-hard. And the other is that it's in NP. So if you think of the intersection, NP intersect NP-hard, that's NP-complete. Let me draw on the picture here what this means. So I'm going to draw it on the top.

This is NP-hard. Everything from here to the right is NP-hard. NP-hard means it's at least as hard as everything in NP. That means it might be at this line or it might be to the right. But in the case of Tetris, we know that it's in NP. We proved that a couple of times. And so we know that Tetris is also in this range. And so if it's in this range and in this range, it's got to be right here.

Completeness is nice. If you prove something is something complete-- prove a problem is some complexity class complete-- then you know sort of exactly where it falls on this line. NP-complete means right here. EXP-complete means right here.

Turns out Chess is EXP-complete. EXP-hard is anything from here over. EXP is anything from here, over this way. Chess is right at that borderline. It is the hardest problem in EXP. And that's actually the only way we know to prove that it's not NP. It's is pretty easy to show that EXP is bigger than P. And Chess is the farthest to the right in EXP-- of any problem in EXP-- and so, therefore, it's not in P.

So whereas this one-- these two, we're not sure are they equal. This line we know is different from this one. We don't know about these two, though. Does NP equal EXP? Not as famous. You won't get a million dollars, but still a very big, open question.

What else do I wanna say? Tetris, Chess, EXP-hard. So these lines, here-- this is NP-complete And this is EXP-complete.

So the last thing I want to talk about is reductions. Reductions-- so how do you prove something like this? What is as hard as even mean? I haven't defined that. But it's not hard to define. In fact, it's a concept we've seen already.

Reductions are actually a way to design algorithms that we've been using implicitly a lot. You may have even heard this term. A bunch of recitations have used the word reduction for graph reduction. You have some problem, you convert it into a graph problem, then you just call the graph algorithm. You're done. That's reduction.

In general, you have some problem, A, that you want to solve. And you convert it into some other problem, B, that you already know how to solve. It's a great tool because, in this class, you learn tons of algorithms for solving tons of problems.

Now, someone gives you, in your job or whatever, or you think about some problem that you don't know how to solve, the first thing you should do is-- can I convert it into something I know how to solve because then you're done. Now it may not be the best way to solve it, but at least it's a way to solve it. Probably in polynomial time because we think of B as things you can solve in polynomial time. Great.

So just convert problem A, which you want to solve, into some problem B that you know how to solve. That's reduction. Let me give you some examples that we've already seen, just to fit this into your mental map of the class. It's kind of a funny one but it's a very simple one.

So how do you solve unweighted shortest paths? In general? Easy one. Give you a graph with no weights on the edges and I want to the shortest path from s to t.

AUDIENCE: BFS

PROFESSOR: BFS. Linear time, right? Well, that's if you're smart or if you feel like implementing BFS. Suppose someone gave you Djikstra. Said, here, look, I've got Djikstra code. You don't have to do anything. There's Djisktra code right there. But Djikstra solves weighted shortest path. I don't have any weights. What do I do? Set the weights to 1.

It's very easy, but this is a reduction-- a simple example of reduction. Not the smartest of reductions, but it's a reduction. So I can convert unweighted shortest paths into weighted shortest paths by adding weights of 1. Done. Adding weights of 0 would not work. But weights of 1. OK. Weights of 2 also works. Pick your favorite number, but as long as you're consistent about it. That's a reduction.

Here's some more interesting ones. On the problems set-- problem set six-- there was this RenBook problem, "I Can Haz Moar Frendz?" That was the name of the problem. And the goal was to solve-- to find paths that minimize the product of weights. But what we've covered in class is how to solve a problem when it's the sum of weights. How do you do it? In one word, or less? Logs. Just take logs. That converts products into sums.

Now you start to get the flavor. This is a problem that you could take Djikstra or Bellman-Ford, and change all the relaxation steps and change it to work directly with products. That would work, but it's more work. You have to prove that that's still correct. It's annoying to think about. And it's annoying to program. It's not modular, blah, blah, blah.

Whereas if you just do this reduction, you can use exactly the code that you had before, at the end. So that's nice. This is why reductions are really the most common algorithm design technique because you don't want to implement an algorithm for every single problem you have. It would be nice if you could reuse some of those algorithms that you had before. Reductions let you do that.

Another one, which was on the quiz in the true-false-- quiz two-- was converting

longest path into shortest path. We didn't phrase it as a reduction. It was just can you solve longest path using Bellman-Ford. And the answer is yes. You just negate all the weights. And that converts a longest path problem into a shortest path problem. Easy.

Also on the quiz-- maybe I don't need to write all of these down because they're a little bit weird problems. We made them up. There was the-- what was the duck tour called? Bird tours? Bird tours? Aviation tours? Whatever. You want to visit a bunch of sites in some specified order. The point in that problem is you could reduce it to a single shortest paths query. And so if you already have shortest path code, you don't have to think much. You just do the graph application. Done.

Then there's the leaky tank problem, which is also a graph reduction problem. You could represent all these extra weird things that were happening in your car by just changing the graph a little bit. And it's a very powerful technique. In this class, we see it mostly in graph reductions. But it could apply all over the place.

And while this is a powerful technique for coming up with new algorithms, it's also a powerful technique for proving things like Tetris is NP-hard. So what we proved is that a problem called 3-Partition can be reduced to Tetris.

What's 3-Partition? 3-Partition is I give you n numbers. I want to know can I divide them into triples, each of the same sum. So I have n numbers. Divide them into n over 3 groups of 3, such that the sum of each of the 3s is equal. Sounds like an easy enough problem. But it's an NP-complete problem. And people knew that since one of the first papers. I guess that was late '70s, early '80s, by Karp.

So Karp already proved this is standing on the shoulders of giants. Karp proved 3-Partition is NP-complete, so I don't need to think about that. All I need to focus on is showing that Tetris is harder than 3-Partition. This is what I mean by harder.

Harder means-- so when I can reduce A to B, we say the A-- B is at least as hard as A. Why's that? Because I can solve A by solving B. I just apply this reduction and then solve B. So if I had some good way to solve B, it would turn into a good way to

solve A.

Now 3-Partition-- which is A, here-- we're pretty sure there's no good algorithm for solving this. Pretty sure it's not in P. And so Tetris better not be P either because if Tetris were in P, then we could just take our 3-Partition, reduce it to Tetris, and then 3-Partition would be in P.

In fact, all of the NP-complete problems, you can reduce to each other. And so to show that something is at that little position, NP-complete, all you need to do is find some known NP-complete problem and reduce it to your problem. So reductions are super useful for getting positive results for making new algorithms, but also for proving negative results-- showing that one problem is harder than another. And if you already believe this is hard, then you should believe this is hard.

I think that's all I really have time for. I'll give you a couple more NP-complete problems. Kind of fun. Traveling salesman problem, you may have heard of. Let's say you have a graph. And you want to find out the shortest path that visits all the vertices, not just one vertex. That's NP-complete.

We solved longest common subsequence for two strings, but if I give you n strings that you need to find the longest common subsequence of, that's NP-complete. Minesweeper, Sudoku, most puzzles that are interesting are NP-complete. SAT. SAT is a-- I give you a Boolean formula like x or y AND NOT x-- something like that. I want to know is there some setting of the variables that makes this thing come out true? Is it possible to make this true? That's NP-complete complete. This was actually the first problem that was shown NP-complete.

There's this issue, right? If I'm going to show everything's NP-complete by reduction, how the heck do I get started? What's the first problem? And this is the first problem. You could sort of prove it by definition, almost, of NP, here. But I won't do that.

Three coloring a graph. Shortest paths. This is fun. Shortest paths in a graph is hard. But in the real world, we live in a three dimensional, geometric environment.

What if I want to find the shortest path from this point, where I am, to that point, over on the ceiling or something. And I can fly. That's NP-complete. It's kind of weird. Shortest paths in a two dimensional environment is polynomial. It's a good thing that we are on ground because, then, we can model things by two dimensions. We can model things by graphs. But in 3D, shortest paths is NP-complete.

So all these things where a problem-- knapsack, that's another one. We've already covered knapsack. We saw a pseudo-polynomial algorithm. Turns out, you can't do better than pseudo-polynomial unless P equals NP because knapsack is NP-complete. So there you go. Computational complexity in 50 minutes.