**PROFESSOR:** All right. Welcome to 6890, Algorithmic Lower Bounds, Fun with Hardness Proofs. I am your host, Erik Demaine. We have on my left Jayson Lynch and Sarah Eisenstat, your TAs.

First question is what is this class about? The tag line is hardness made easy. In general, we're interested in proving problems hard. Proving that there's no fast algorithms to solve problems under certain assumptions. And the goal is to give you a practical guide and give you lots of experience in how to prove problems hard, to make that an easy process.

There's a lot of technique involved, and the more experience you get improving hardness, it becomes relatively straightforward to take whatever problem you're interested in and prove it hard

So the three of us are pretty good at it, and our goal is to share with you that expertise so that everyone's good at it. This is not a complexity class. Because there are lots of computational complexity classes at MIT. So we're not going to talk about lots of beautiful deep mathematics about relations between complexity classes. We're just going to use a lot of those results wholesale without proving them. And you can take one of the many computational complexity classes to get that background.

I will tell you everything you need to know about complexity. So if you've taken a complexity class, there will be a small amount of repetition. Most of it will be contained in today's lecture. But very little repetition, but also no background required. I do expect you to have a background in algorithms, because we take a kind of algorithmic perspective. I call it an anti-algorithmic perspective, if you like.

And so, yeah. So that's what this class is. Why would you want to take this class? Why prove hardness? Well, the main reason is to show that you can't design algorithms in whatever model you're interested in. That encourages you either to change the problem, like to look for approximation algorithms, or fixed parameter algorithms. That's, of course, the topics of algorithms class.

Our goal is to prove when these things are not possible in this class. We're going to master a lot of techniques. You'll see a lot of key problems to reduce from to your problem. Get a lot of proof styles that are quite common in the literature, but unless you've done them and experienced them, it's hard to know what to look for. And one of the big ideas in this class is gadgets. And you may have seen gadgets before, but we're going to explore gadgets to their fullest. The idea of taking lots of small components and combining them together.

Even if you don't care about hardness, I think this class is a lot of fun for two reasons. Maybe three. One is that you see lots of cool connections between different problems that you might not think are related at first glance. Most of the problems in this class are equivalent to each other. And it's all about proving that. That's the goal.

We'll also study lots of fun problems, like Super Mario Brothers. We'll NP-complete today. Tetris, we'll do in a lecture or two. There are also serious problems, so if you don't like fun, don't worry. And in general, proving problems hard is really like solving a puzzle. And it's a lot of fun. And this is one of the rare areas where you can basically play with puzzles all day, and in the end have publishable papers.

So we're going to do that in particular through an open problem session, which is optional. If you want to solve open problems that no one knows the answer to, we will try to do it once a week. We'll be sending email with a call for times when that's ideal for everybody. We talked about background and requirements.

So first requirement of the class is to fill out the survey, which is circulating Does anyone not have a survey? Good. Everyone has one. So fill that out, so we know who you are. You should also join the mailing list. By filling out that survey, you should auto join the mailing list. But just in case, if you haven't, the mailing list is on the course website.

Yeah. Good. Another requirement is to attend lectures. You're doing good on that so far. And you'll have scribe-- I would guess at this scale, you'll scribe once, probably in a team. But we will figure that out as time goes on.

Again, they'll be email about signing up for scribing. Scribing is taking notes in the lecture so we have another form of notes. There will be something like five P sets every two to three weeks. The first one will go out Tuesday is the plan. And then the big part of the class is the project and presentation, so the final project, you could do it on almost anything related to the content of the class.

Typical projects are do something new theoretically, prove a problem hard, find some nice open problems, survey existing material, something that's not covered in class. You can code something. I think in this context, the most natural thing to do is to visualize cool proofs that we cover in class in a new way. You could contribute to Wikipedia, or you could make some art piece, like a sculpture, or a performance, or whatever you want related to hardness proofs.

That's your project possibilities. So what is in this class? I will look at what specific topics are covered. This is on the website. So a lot of the class will be about NP-completeness, which I will define in a bit. But we'll also look at even harder-- these are all notions of no polynomial time algorithm. We'll do even harder notions then NP, like PSPACE and X time, and so on.

In particular, we'll be studying those in the context of games and puzzles, and there's a whole theory called games, puzzles, and computation, the topic of this book, which we will talk about at length. Then, we will go to inapproximability. This is not necessarily in order, so I should say then. But we'll talk about when you cannot find good approximation algorithms, and what good means depends on what sort of problem you're looking at, and when you cannot find fixed parameter algorithms, which are fast algorithms when the optimal solutions happens to be small.

So that's all about not polynomial. Then, there's a small amount of work on understanding the polynomial aspect for small polynomials, distinguishing almost linear time from n squared time or n cubed time. That sort of thing.

Then there are other sorts of problems. We can think about less common types of problems, like where you want to count the number of solutions instead of just find one, or tell whether the solution is unique. There's some economic game theory stuff. This existential theory, the reals, comes up in some geometric settings. And if there's time, we'll talk a little bit about undecidability, although that's a pretty different world. That's where there's no algorithm given any finite time bound.

So that's in a nutshell what's the entire class is. Today, we're going to do a sort of crash course on computational complexity. Most of what you should need for the entire course, I think, will fit in about 40 minutes. Maybe 50 minutes. Something. And that will serve as a guideline. If anything's not clear, if I go too fast, feel free to stop, ask questions during class or after class.

And let's see. So we have some-- there's no real textbook for the class, but there are two recommended reading books. They are Garey and Johnson. *Computers and Intractability* is the title. Most people call it Garey and Johnson. This is an old book from pretty much early on in the world of NP completeness, but it's still a really good book. So that's good to check out.

And then there's my book with Bob Hearn. This is Bob Hearn's PhD thesis at MIT, *Games, Puzzles, and Computation.* This is available electronically for free online to all MIT people. So if you look it up in MIT libraries, it's actually linked right here.

So you're seeing that base. You can get an electronic copy. If you want to buy one, talk to me or order it. There's a couple other links on the website. There was some followup to the Garey and Johnson book by Johnson and some other cool websites.

All right. So I think that is the administrative part. Now we can start the fun part of the class. So let's do our crash course on complexity. So if you've seen complexity before, some of this will be review. We start with our favorite class of problems that can be solved in polynomial time.

Say on a RAM, I should specify model of computation and exactly what a problem is, but I'll be a little bit informal here. Polynomial time means n to some constant, where n is the size of the problem instance. And we will talk a little bit more about subtleties in defining n. But usually, polynomial time is pretty clear.

This is what we consider good algorithms. An example of something we consider bad is exponential time. So x is going to be all the problems that can be solved in exponential time. And exponential's a little less a uniquely defined, but I'll define it this way. I think this is the usual definition for x. . So 2 to a polynomial. Of course, it could be 3 if you prefer. Any constant will be the same here. This constant dwarfs any constant there.

So that is-- this is a huge class, any problem you can in exponential time. Most problems can be solved exponential time, though not all. Most problems we encounter, I should say. And then I'll define one more just for kicks. R is a recursively enumerable problems, or recursive problems. These are all the problems that can be solved in finite time. Always. And these are all worst case bounds.

So I'm going to have a running picture. This is my favorite picture to draw. We have on the x-axis a somewhat vague notion of computational difficulty. What you might call hardness colloquially. This is a bit informal, but it's still a useful way to think about things. So this initial chunk here is going to be P. I'm going to leave in some space to fill in some other things. And probably go to here. And this part is going to be EXP, and then everything to the left of this line are the problems that are solvable in finite time. Everything to the right of that line is not solvable by algorithms. That would be undecidable.

OK. So obviously, anything we saw in polynomial time can also be solved in exponential time. That's all that this is saying. Cool. So I have examples. I'll give you some examples to think about. For example, n by n Chess. So this is, I give you a Chess configuration on an n by n, and I want to know, let's say, white to move. Can white force a win? This turns out to be solvable in exponential time, and is not solvable in polynomial time. So that's a nice result. Something we will get to.

Another example is Tetris, suitably generalized. So we're thinking about problems where you have all the information. Usually in Tetris, you don't know all the pieces that are going to come. But supposing you knew the future, I give you the entire sequence of pieces that are going to come, sort of a Tetris puzzle. They used to publish these in *Nintendo Power Magazine.* And you want to know, can I survive from this board position? Can I survive this sequence of pieces

This is also in EXP. You can solve this in exponential time. That's a little more obvious. But we don't know whether it's in P. Probably it's not, and we'll see why in a moment. If you've taken an algorithms class, you know tons of examples of problems that are in P, like shortest paths or lots of good things.

Halting problem, you've probably heard of. Kind of a classic. Halting problem is, given an algorithm, does it terminate, or given some computer code, does it terminate? This is not recursive, meaning there's no algorithm to solve it in finite time in the worst case.

There's a more depressing result, which is that in fact, most problems-- let's say most decision problems are not in R. Most problems cannot be solved by an algorithm. If you haven't seen that, it's cool result. Basically, the proof is that the number of problems in the universe is about 2 to the N. And the number of algorithms is only about N. So if you know set theory, great. Otherwise, ignore this sentence.

So you could think of an algorithm as a number. It's like, you take this string and convert it to a giant number. So that's integers over here. You can think of problems as a mapping from inputs to Yes or No. Yes or No is the 2. Inputs is the N. This is the same as real numbers. This is the integers, and there are a lot more real numbers than integers. You probably heard of that.

So this means most problems have no algorithm if an algorithm can only solve one problem. That's the sad news of life. Luckily, most of the problems we tend to pose do have an algorithm, and it's more about P versus EXP that this class is about.

So let's go to more interesting things. I'm going to define a class NP, which is in between here. In between P and EXP. So as I said, decision problems are problems where the answer is Yes or No. There are lots of possible definitions of NP. I will cover two.

I would like there to be an algorithm to solve the problem in polynomial time, but not in a regular model of computation, But with something I call a lucky algorithm. Lucky algorithm comes to a decision, and it always makes the right one. It's just lucky. It doesn't have any reason to believe that's the right one. It just always makes the right choice given the choice between two options, let's say.

So this is, you make lucky guesses. You always guess the right one. It's a little bit biased in a way that I should say. So let me be a little more precise. This is called a non-deterministic model. And the N in NP is non-deterministic.

So the idea is that the algorithm makes a series of guesses. It could do it at the beginning, or could do it in the middle of the computation. And eventually, it outputs an answer. So it's going to say either Yes or No. And what we guarantee in this weird non-deterministic lucky model of computation is that you will be led to a Yes answer if it's possible. So it guesses-- this is asymmetric.

So what this means is say you run your magical lucky algorithm, and it outputs No. That means no matter what set of choices you made for each guess, you would always get to a No. If you get a Yes answer, you just know there's some set of guesses that lead to a Yes answer. So one is an existential quantifier, one's universal. So this is asymmetric. There is a notion of CoNP, which is exactly the reverse. CoNP, you let's say flip Yes with No. So CoNP, you prefer no answers if you can get them.

All right. So let me give another definition of NP. Another way to think of the same definition, really.

So you can also think of NP problems as problems that have solutions that are relatively succinct and can be checked in polynomial time. I guess really they need to be checkable efficiently. So what you can think of this as saying is, well, every time I make a guess, I'll write down whether I went left or went right in my maze, I guess.

And so that you could think of as a certificate. If you know what the right sequence of guesses are, of course, you can run the algorithm, because it's a polynomial time algorithm. Conversely, if I don't have the solution to the problem, yet I'm told that it exists, at the top of my algorithm, I could just guess what that solution is, and then check it.

So if I'm given such an algorithm, I can convert it into a lucky algorithm. If I'm given a lucky algorithm, I can convert it into one of these checking algorithms. OK. So let's do an example. Let's say Tetris is in NP. So if I give you a board, and I give you a sequence of pieces that are going to come. How would I prove to you that I can survive those sequence of pieces?

**AUDIENCE:** Say where they go?

**PROFESSOR:** Just say where they go. Say what sequence or moves I press, and at what times say. Just where should I drop each piece? So then all you need to do for this definition is check that that's about solution that you never have to push a piece up, for example, to get it into the right position.

Or you can think of the same thing as an algorithm that says, oh, OK. Every time I have to press left, or right, or wait a second, or push down, I'll just guess which one to do, and then do that. So these are the same algorithm, essentially. And that's why Tetris is in NP.

In general, let's see. Every problem that I can solve in polynomial time, of course, I can solve in non-deterministic polynomial time, so that's that containment. If I have an NP problem and I want to solve it in exponential time, well, I could just simulate all the possible guessing paths, because I run for only polynomial time. And so for each one, maybe I have two choices. I'll just try both. I'll do this sort of depth research. And yeah. I have exponential is exactly 2 to that polynomial, so I can afford to branch in both directions.

So I guess my guesses here are just binary. I could afford to branch in both directions, and eventually see whether any of them leads to a Yes. I could also figure out whether any of them leads to a No. But in general, NP is contained in EXP. Cool.

Still not too interesting. Where it gets interesting is when we start talking about hardness, which is next. I should mention big open question is whether there's any problem in here in NP minus P. This is the same as P equals NP open problem. Most sane people in the universe believe P does not equal NP.

What this means intuitively is that you can't engineer luck. Luck shouldn't exist in the real world. You can't just like, go one way or the other and always make the right choice. You could make a random choice. You could try both choices. But you shouldn't be able to always make the right choice for all problems. That seems insane. So if you believe that, you believe P does not equal NP and you believe there's some things in between those two lines.

But we don't know, unfortunately. So let me talk about hardness. So if I have some complexity class X, X could be NP, or EXP at this point. P we won't talk about. P hardness is a little weird. I'm not going to define this formally until a little bit later in today's class.

So problem is X-hard if it's sort of the hardest problem in the class X, if it's as hard as every other problem in X. Actually, this problem may not be in X, so I should say it's hard as every problem in X. So what that means in this picture is the following.

So every problem from here onward is EXP-hard. And every problem from here onward is NP-hard. Whoops. Here. We won't talk about P hardness. That's a notion in parallel computing. I mean, maybe we'll have time to talk about it, but it's not currently on the plan.

So this is the lower bound side, right? You're proving that you're at least as hard as the very hardest problem in NP, or you're at least as hard as the problem in EXP. The reason, the only reason, I know that Chess is not NP is because I know that Chess is actually X-hard. I know it's at least as hard as all problems solvable an exponential time. And there's a great theorem called time hierarchy theorem that tells you that P does not equal EXP in particular.

So we know there are some problems in EXP that are not in P. Some problems that require exponential time can't be done in polynomial time. And we know that Chess is as hard as all of them, so in particular, it also can't be solved in polynomial time.

Mind you, though, I haven't defined what as hard as a means yet, but I will get there. Another good term to know is X-completeness. This is just the and of two things, being X-hard and being in X. So in my picture, this dot right here is NP-complete. And this dot right here is-- got to write it this way-- is X-complete.

OK. I already mentioned that Chess is in EXP. So in fact, Chess is X-complete. It means there's an upper bound saying that you can solve it in exponential time. There's a lower bound saying that it's at least as hard as everything in EXP. And both are true, so you would know you're kind of right here. From the resolution of this picture, that's all you could hope to know about Chess.

So, good. One more class of problems good to know about, and it will come up a lot in games, is the notion of PSPACE. So so far, we've only thought about time. But usually, we measure algorithms in terms of time and space. How much memory does your algorithm use? And so PSPACE is going to be, I guess, say let's say problems.

As you might guess, this is problems solvable in polynomial space. In general, if you can solve a problem in polynomial space, you can solve it in exponential time, because there are only exponentially many states of your machine if you only have polynomial space.

So PSPACE fits here in between NP and EXP. And of course, there's PSPACE-hard and PSPACE-complete. If you don't remember anything from today except one blackboard, remember this blackboard. It's like the cheat sheet to everything we've defined. You just have to remember what all the letters mean.

But not too hard. So that's pretty much all the classes we'll be using. There are a few others. I'm sorry. I'll give you one example to go with our other examples. The problem we'll look at today is *Rush Hour.* This is a one player puzzle, where you're trying to move the cars, and they could only go vertically or horizontally.

This is in PSPACE, which is maybe not so obvious. And actually, it's PSPACE-complete. So this is from the diagram, PSPACE-complete is harder than NP-complete. Now of course, we don't actually know whether these two points are the same. It's kind of annoying. We don't know whether these two points are the same. We don't know whether these two points are the same. This whole thing could collapse.

We do know these two points are different. So somewhere here, or here, or here, we have a positive range. Most people believe all of these have problems in them, so none of these things are the same. It could be NP-complete is the same as PSPACE-complete. But again, most people believe these are different.

So in some sense, *Rush Hour,* which is here, is harder than Tetris, which is here. Cool. And if you believe that either one of these is non-empty, then we know that *Rush Hour* does not have a polynomial time algorithm. So to show that *Rush Hour* is not in P, you have a choice. You could prove either one of these as non-empty. For Tetris, you have to prove this one.

So you'd be less famous if you prove this one. Still pretty famous, but you wouldn't win the million dollar bounty that's on P versus NP.

All right. There are bigger classes. We talked about exponential time. You can, of course, talk about exponential space. In general, these interleave. You go polynomial time, polynomial space, exponential time, exponential space, doubly exponential time, doubly exponential space, and so on. That's the order in which they occur.

The only things we know is that polynomial time is different from exponential time is different from doubly exponential time, or any function of time, really. And we know that polynomial space is different from exponential space is different from doubly exponential space. But we don't know about the interrelation between time and space. That's one of the big questions. The other big question is non-determinism.

One fun fact you should know is that PSPACE equals NPSPACE. This is a useful fact. NPSPACE is non-deterministic polynomial space. So you take a lucky algorithm, and you don't guarantee how much time it takes, it will be a most exponential time. You only guarantee the amount of space it uses is, at most, polynomial.

This is a theorem called Savages Theorem, and it works for any space bound. In general, the space bound, I think, grows to the square of its original, if you want to convert non-deterministic to deterministic. This is useful for *Rush Hour,* because to play a *Rush Hour* game, in general, the number of moves you might have to make is exponential, so it's not obviously in NP, because NP would have to have a short polynomial-length solution that you can check in polynomial time.

But *Rush Hour,* you can solve in polynomial space if you're really lucky because you say, well, what move should I make? Well, I'll guess one, then I'll make that move. Guess another move. Make the move. And just maintaining the state of the board only takes polynomial space. So then if you solve it, you're happy. If there's no way to solve it, you will return No.

I guess you have a timer. After you've made exponentially many moves, if you still haven't solved the puzzle, you can return No. And in the lucky world, that means you will find a solution if there is one. Conveniently, lucky algorithms can be turned into regular algorithms when you're only worried about space bounds. And so that's how you prove *Rush Hour* is in PSPACE. So that's a good fact to know.

Cool. All right. There's one key thing we haven't defined yet, which is as hard as. So let's get to that. This is really the heart of the class. Let me go up there.

So this class is really all about one notion, and that notion is reductions. So if I have two problems A and B, then there's this notion of a reduction from A to B. This will be an algorithm. For us, almost all the time it will be a polynomial time algorithm, although you could put in a different adjective here than polynomial time. Most of the time, that is what we want.

And it's going to convert an instance of the A problem-- so instance just means input-- and we'll convert it into an instance of the B problem. And it's going to do so in a way such that the solution to A equals the solution to B. I mean, the solution of that instance of A is the same as the solution of the instance to B.

So this is-- think decision problems. The answer's either Yes or No. So we want to convert A into an equivalent instance of B, equivalent meaning that it has the same answer. Why do we care? Because let's suppose we had an algorithm to solve B. That would be this arrow. So let's say if we can solve B, then we can solve A by this diagram. Take an instance of A, convert it into an equivalence of B, solve B, and then that solution is equal to the solution to A, so we solved A.

This is as hard as. So what we say is B is as hard as A. That's a definition of this hardness. In general, depending on your definition of reduction, you'll get a different notion of as hard as, but we will stick primarily to polynomial time.

Reductions. This is what you might call a one call reduction. This is kind of a technicality. Also called a Karp style reduction because Karp gave a whole bunch of them in the '70s, '80s. So the idea is you only get to call your solution to be once. In general, you could imagine an algorithm that calls your solution to be many times. That would also be a notion of as hard as. For the problems we'll look, basically, these two notions don't seem very helpful, let's say.

So we'll stick to one call reductions because they seem sufficient for everything that we will cover in this class. Probably. Maybe in some very late lecture, we'll talk about multi-call reductions. But they're not so prominent. One call reductions are the bread and butter of hardness. So as you might imagine, this is how you prove a problem hard. Basically, all hardness proofs in the known universe are based on a reduction.

You start from a problem which you know is hard in whatever class you care about, and you reduce from that problem, the known hard problem, to your problem that you're not sure about. If you can do that, then you prove that your problem is as hard as the original problem. If you know that one is hard, than this one is hard.

Don't get this backwards. You will anyway, but try not to get it backwards. You're always reducing from the known hard problem to your problem. OK. So usually say, our proof is based on a reduction from-- pick your favorite problem. Never two. Easy to get wrong, because it's easy to make a sign error. But that's life.

So good. Anything else? Now if you've taken an algorithms class, you've seen lots of reductions. Reductions are a powerful tool in algorithms too. For example, some lame examples like, if you have an unweighted shortest path problem, you can reduce that to a weighted shortest path problem.

How do you do that? You set all the weights to 1. Yay. Why would you do that? Well, never mind. It kind of illustrates what's going on here. What we're showing is that A is a special case of B. Unweighted shortest paths is a special case of weighted shortest paths. In some sense, all reductions are that, but they're usually much less obvious than unweighted shortest paths to weighted shortest paths.

And the reason we can say B is as hard as A is because [? modular ?] this conversion algorithm A is a special case of B. So of course, if B has more cases than A, or maybe the same, but if it has at least as many cases as A, then of course, B is at at least as hard to solve as A in this formal sense.

OK. Cool. There are more interesting examples you've probably seen in problem sets, like if you want in the arbitrage problem, you want to find a path that has the minimum product of all the values, to convert products into a min. Some problem, you just take logs of all the values. So the reduction is compute logs and all the weights.

So you've probably seen lots of things like that. Some are more complicated than others. What we're going to do in this class is reduce instead of-- so for algorithms, you want to reduce to something you know how to solve. We're going to reduce from something we know we can't solve under certain assumptions.

So assuming P does not equal NP, if A is NP-hard and we reduce from A to B, then we know the B is NP-hard. That's a theorem. So in this situation, if we have a reduction, let's say if A reduces to B, I'll be explicit, and A-- better get this right. It would be pretty embarrassing if I got it wrong, but I probably will get it wrong at some point in this class. Hopefully not today. Then B is X-hard.

OK. That's actually kind of a trivial theorem if you read all the definitions. What did X-hard? It meant as hard as every problem in X. And as hard as meant there was a reduction. So that means-- get this right-- there's reduction from every problem in X to your problem. So if A is NP-hard there's a reduction from every problem to A, and we're saying now what if there's also a reduction from A to B, well then, we can just chain those are two reductions together, convert any problem in X to A, then convert it to B. And so we've shown every problem in X can be reduced to B, where B is X-hard. This is an easy theorem.

The converse would be if B can be solved in class X, and X contains polynomial time, then you could can also solve A in the same complexity class. So for example, if B is in PSPACE, then we learn that A is in PSPACE. Something like that. Cool. Any questions at this point?

**AUDIENCE:** Question.

**PROFESSOR:** Yeah?

**AUDIENCE:** You've talked a lot about thi NP thing. Why don't you ever talk about non-deterministic exponential orobelms or things like that.

**PROFESSOR:** OK. Yeah, I didn't define non-deterministic exponential problems, but that's a valid thing. I think it's usually written NEXP. And I believe there is a class of games that naturally fits into NEXP. Its mainly an issue of which classes commonly arise in problems that we care about. This is a pretty rare one, but I think there might be one instance where-- we might touch on this. Yeah, we definitely can.

It's the same open problem where EXP equals NEXP. I mean, it's another open problem. I shouldn't say it's the same. I think you could solve one without the other. Again, everyone believes you can't do it. If you believe you can't engineer luckiness, then of course, NEXP doesn't equal EXP. But that would fit right after EXP. If you get NEXP, and then you get EXPSPACE. According to the things we've written down and cared about, that would be the order of things. Of course there are classes in between these, but these are the ones that appear normally. Yeah?

**AUDIENCE:** Do we know how to use hardness assumptions, for example, P not equal to NP to say whether there exists problems that are not in P nor NP complete?

**PROFESSOR:** Yeah. That's another big open question. Let's say P does not equal NP. Are there problems here? Which is to say strictly in between being-- so they're still not in P to the right of this line, but they're also easier than NP-complete problems. There are a couple problems that are famously conjectured to live there, like factoring integers and graph isomorphism. But I'm not aware of any nice complexity class there.

You can, of course, say-- you can talk about a class of problems that are as hard as graph isomorphism in a certain sense, that if you believe-- if you have a graph isomorphism Oracle, what can you do? So there's a little bit of work trying to chart that space, but I'd say in general, it's been not super successful. Don't quote me on that, though. I also don't know the literature super well there.

So that's a big uncharted territory, let's say. Could be a big project. Probably lots of open problems there. Other questions? I think most people believe there are things there. One thing you could prove is there's a thing called exponential time hypothesis, which would say that [? sat ?] has no sub-exponential algorithm. Nothing to the [? little o of ?] N. If you believe that, then it's known there are some things in between. You can pick your favorite function, like N to Log [? Log ?] N.

That's bigger than polynomial, but smaller than exponential. And there are problems that are in that class and not [? NP. ?] So definitely with ETH, you can do it. Yeah.

**AUDIENCE:** So regarding that question, I think it has been proven that if P not equal NP, then there is stuff in there.

**PROFESSOR:** Oh, there is. Yeah.

**AUDIENCE:** Or you can kind of do a diagonalization.

**PROFESSOR:** Other questions? Yeah?

**AUDIENCE:** What's the decision problem for the factoring?

**PROFESSOR:** Decision problem for factoring. I think one version is I give you a number, and I give you a bit position, and I want to know whether there is a factor that has a 0 or 1 in that position. So if you can solve the problem, then by repeated calls, you could actually find a factor that's not 1.

I don't know if there's a better version than that. You have to be careful, of course, if you say, well, is the number prime? Then that's in polynomial time. That was a big result a bunch of years ago. So, yeah. Most of the time, it's easy to go from your optimization problem to a decision problem. But factoring is one where it's less clear. More questions?

That was a lot in a small amount of time. But that's all you should need from complexity theory. Yeah?

**AUDIENCE:** OK. I guess I have a question about the way you defined NP with the [INAUDIBLE] algorithms?

**PROFESSOR:** Yup.

**AUDIENCE:** So the definition that I've seen before is in terms of [INAUDIBLE].

**PROFESSOR:** Right.

**AUDIENCE:** And there, it specifically constrains every branch in the computation to run polynomial time, whereas you can imagine a lucky algorithm that avoids infinite looping just because it's lucky.

**PROFESSOR:** Oh. OK. Yeah. I was a little vague here. For the definition of NP , I said polynomial time in the worst case. Probably I need to say it's polynomial time no matter what branch you take. Not sure it matters. Because maybe if you know your algorithm's supposed to run in polynomial time, you could have a timer, and if it ever exceeds that time, just return no. So you might be able to convert the weaker notion of polynomial time to the stronger one.

But it won't matter too much. I mean, in all the algorithms we'll think about, every branch you could possibly think of even, the non-lucky ones, are polynomial time. So we will stick to that. We won't spend much time in general proving that problems are in NP or NPSPACE, only to check that this is the right class we're supposed to be working in. We'll spend most of our time proving the hardness cases, the lower bounds. Yeah?

**AUDIENCE:** So you mentioned that if you use different notions of a reduction, different limit space equally on the competition, you get different measures of as hard as.

**PROFESSOR:** Yes.

**AUDIENCE:** So as your classes get bigger, does it matter if you start using more polynomial time in your reduction? Does it actually change?

**PROFESSOR:** I'm sure it matters. But again, it's beyond the classes that are usually considered from an algorithmic standpoint. Bigger than these classes don't seem to matter much. And in these, maybe once I've seen like, a polynomial space reduction to prove X-hardness, so it's a slightly weaker notion than if you do a polynomial time reduction, but if you believe PSPACE does not equal X, but still a separation.

So sometimes, things like that happen. I think the most common is just switching to a space measure over time measure. I don't think it will matter in anything we see here. But it does arise in the literature. Definitely. I think PSPACE or log space are the two common ones. And of course, when you're doing P-completeness, it's another. Yeah?

**AUDIENCE:** You mentioned setting it [INAUDIBLE], and I'm curious how you could do that. Because for any instance of the problem, you can't say that I've used exponential time. It could just be that there's a bigger constant out front.

**PROFESSOR:** OK. Yeah, so you need to know what the polynomial is for it to have a timer. That's a small catch. Yeah. I don't see an easy way to avoid that. Yeah. There's some few subtle constructive issues there. If you don't know what the bound is, then things get more annoying. None of those things will happen in reality. So remember, this not a complexity class. We're looking at algorithms here.

This part is actually easy. It's all about the reductions. That's where the meat of this class will be. And for that, we need to get NP-hardcore.

**AUDIENCE:** [LAUGHING].

**PROFESSOR:** I think all of you soon will be NP-hardore, and we'll be able to prove really hard problems really hard. So in the spirit of NP-hardcoreness, we're going to take the classic of hardcore video games, *Super Mario Brothers*. And our first proof will be that *Mario Brothers* is NP-hard. This got lots of press, like here's Kotaku saying, "Science proves old video games were super hard."

**AUDIENCE:** [LAUGHING].

**PROFESSOR:** This is proving the obvious, I guess. But anyway. So let me tell you a little bit about how this proof goes, and then we will see it. All right. So we need a problem to reduce from. The problem is 3SAT. This is probably the most common problem to reduce from. We will spend a bunch of lectures on it. It won't be our very next class, but we will get to it pretty soon.

This is also known as 3-Satisfiability. If you're ever trying to prove NP-hardness and you don't know where to start from, the answer is 3SAT. Almost always, but not always.

**AUDIENCE:** [LAUGHING]

**PROFESSOR:** OK. Whatever. So what's the problem? You're given a 3CNF formula. This means something like x5, or I'll be nice and write English, or x3, or not x1. And another thing like that. x7, or not x2, or x5. Whatever. So some key words you should know. xI's. Those are called variables. xI or not xI. Those are called literals.

You have two choices. You could either have a positive variable or a negative variable. Then you always have three literals per clause. These things are called-- each row of this thing that I drew is called a clause.

The whole thing's called a formula. And the question is, can you make this formula true? So that's what you want to know. Do there exist xI's such that the formula is true? That's the decision problem. And it's NP-complete.

It is almost the first problem proved NP-complete. It's like the second problem that's NP-complete. The first one was without this particular style. It was just a bunch of ands and or's and nots in any combination. But this is also hard, and it's usually easier to start from this situation.

So what we want to do is reduce from 3SAT to *Super Mario Brothers.* OK. So let's do it. So I should mention our proof holds *Super Mario Brothers 1*, *Lost Levels,* and 3. *Super Mario Brothers 2* is another world. Also, *Super Mario Worlds*, I think *1* and *2*. But most the pictures will be *Super Mario Brothers Original.* So what we're going to do are build gadgets.

Gadgets are in this case little pieces of levels that we're going to join together to make the actual reduction. So it's usually-- it's not typical that you look at the instance of A, and just think really hard, and then just output an instance of B from nowhere. You just say, well, OK, instance of A has lots of little pieces. It's got variables, and literals, and clauses, and a formula. I'm just going to take each of those little pieces and convert them into a little piece in my output, and then just string them altogether in a usually fairly obvious way, though sometimes, there's some subtleties there. Almost all of our proofs will follow this structure. This is gadget structure, and this is the thing you're here to learn. And it's super cool.

Because once you know that you want to do 3SAT, you're like, OK, I need variables in closets. So here's the variable. Got Mario on the top. Mario has to decide should he go left or right? I mean, you could go back, but if you want to go through this gadget, you can either fall left or fall right. And all of these heights are so large that once you make that decision, you can't come back up because you have a limited jump height.

OK. Here's the clause gadget. So the idea of variables we're choosing, do I set x5 to be true or false? Let's say the left branch corresponds to true, the right branch corresponds to false. How that happens, we'll see in a moment. That's about how the gadgets fit together.

Clause gadget has two parts. On the one hand, you have these three entry points where your goal is to hit koopas and bounce them around. And then the other part is down here, is a bunch of bricks. And at the end of the level, we're going to set it up so Mario has to go through this part. And this will be traversable if and only if these bricks have been broken.

So how would you do that? This brick-- well, various things are in the way here, so what you need to do-- interesting. But we're going to change this gadget in a second. I already see some issues with it. But this was our original. Original proof didn't get published, and we ended up fixing it before we submitted it.

So the idea is what we want Mario to do is come down here, hit the koopa, and then knock the shell out here, and it will bounce and eventually break all these bricks. Right? Well, no, that would be *Super Mario Brothers 3*, where turtles actually break bricks. In *Super Mario Brothers 1*, they don't.

So we will use a different gadget with fire bars and question blocks with invincibility stars in them. OK. So same idea. There are three entrances down here. If you hit any one of them, then the star will just float around here forever. We tested it. It goes there for as long as the level can last.

Then later, if you come here, lots of testing involved, of course. Later if you come into here, and you can get the star, you have just enough time to run through all these fire. Bars if you don't, you will die. OK?

So that's the clause gadget. So if you visit in at least one of these three places-- you can visit all of them, it doesn't help to have three stars versus one in this case, because they don't stack or anything-- then and only then can you go through the top part.

This is what we might call a traversal, and this is-- you could call it setting the gadget to True. How does this all fit together? This is sort of the bigger issue. So we're going to take this three set instance. It's got variables and clauses. And let's ignore the negations for now. Am I going to ignore them? No, I'm not going to ignore them.

But on the one hand, we have variables. On the other hand, we have clauses. And we're going to connect each variable to the clause that contains it. So this is an actual set of four clauses here, and you can trace them all.

So the claim is that x in the positive form appears in the first clause and the second clause. You can see there's an x here and an x here. It appears in the negative one-- this is not x-- in clause three and four, because here's not x and not x. And so on.

So that's what all these connections in the middle are. In general, it's kind of a bipartite graph. You've got variables on the one side, clauses on the other side, and we happened to have coalesced things in these groups. What these edges are now going to be converted into are paths for Mario to follow.

So this is the variable gadget. It's this thing. We're just going to plug that in here. So the idea is you enter from here, and then you have two ways that you can go, either the true way or the false way. If you set the variable to True, you can then go and visit the corresponding clauses that contain it and get one of the stars.

OK. Now, for each variable, you only get to make one of those choices then you satisfy all the causes that contain that literal. And then when you're done, you can walk to the next variable. So there's actually two entrances to the variable. I guess that kind of matters. What we want in the variable-- so you came from the True setting, or you came from the False setting, you don't want to be able to run and jump over to the other side and the satisfy the previous variable both true and false. You only get to make one choice. Lots of things check here.

Then in the end, at the very end after you've set the last variable, you have to traverse all of these clauses through the fire bars. And that's going to be possible if and only if every one of the causes has a star in it. In other words, the variables satisfy all the clauses. In other words, the formula is true, because the clauses are combined with an and. OK? That is in a nutshell the proof. Once you've made this construction that the solution to 3SAT is equal to the solution to Mario in general, you want to prove on the one hand if the answer is yes to 3SAT, if there's a valid setting for the variables, then there is a solution to this Mario instance. You can actually solve the level.

The decision question here is, can I make it to the end of the level? There's a flag over there. And conversely, if there is an actual solution to this puzzle, you want to show that you can convert it into a valid setting for 3SAT that satisfies the formula. You need to check both. That gives you the equality here. Question?

**AUDIENCE:** Don't you need this graph to be planar?

**PROFESSOR:** Good question. This graph is not planar, and so there are these crossings. So there's one more gadget, the crossover gadget. And this is on the poster, so if you were analyzing it. There are many ways to do crossover, and this one is kind of overkill unless I tell you that *Super Mario Brothers* has tons of hacks and cheats that you can play, and run through walls, and crazy things.

But never mind that. The idea here is you are a big Mario. At the beginning of a level, there's a mushroom. And you better not lose it. Otherwise, you're in trouble. So on the one hand, I can go from left to right. It's a directional crossover. Or I can go from-- this is the bottom to the top.

But I can't go from bottom to right. I can't go from bottom to left. All these sorts of things. Why? OK. Let's do the positive case first. Say I'm from the bottom. I fall here. Can't go back. I can jump. If I'm big Mario, I can break through a couple bricks, and then I can escape.

OK? But I could run under here. For example, if you're good, you can crouch slide, and then jump, jump, jump, jump, jump. You get to here. But you cannot get through this. Or maybe I need to add one more wiggle. A little easier to see over here. Maybe I can get here and move over. But if you're in this position, you have no momentum you can gain, and so you can't crouch slide into there.

So if you're here coming from the bottom, you can't get out. Alternatively, if you come from the left-- it's so tempting to kill the goomba. But instead of killing him, you take damage, become small Mario, then you can traverse through here, because you don't need to crouch slide. You just jump. And there's another mushroom for you to restore big Mario and restore the invariant. And you better take it because otherwise, you can't get out through here.

So you're almost forced to go left to right or bottom to top. Now if you traverse both of these gadgets in both directions, then all bets are off. Then you can go from anywhere to anywhere. That's OK. Because what we're worried about in this reduction is whether you can reach certain things. If you can reach something, I don't care whether you reach it now or later. It's just you don't want to be able to reach unreachable things.

When you set the variable to True, I don't want to be able to visit this false vertex ever. And if you check all the crossovers, it's enough to build this kind of thing, which either you traverse left to right, or bottom to top, or you can reach both the left and the bottom, and then you can reach anything. And you never have to go right to left. You never have to go top to bottom. Because we always know the order in which we're traversing things and so on.

OK. I think I've waved my hands enough. There are details to check, but if you're interested, you can wait 'til this part of the class, where we'll cover the *Legend of Zelda*, *Pokemon*, *Metroid,* and I'm missing one. *Donkey Kong Country*. *Donkey Kong Country* is PSPACE-complete. That's a hard proof. No pun intended.

So that was *Super Mario Brothers*. Any more questions? You asked the right one. I would have gone to it otherwise. Yes?

**AUDIENCE:** Are you sure we'll never have to go say, left to right twice?

**PROFESSOR:** Yes. That's a good question. In this gadget, we are not able to go left to right twice, but that's OK. What's not really drawn here but should be is we're really taking an Euler tour of this star, so we're going to go sort of on the left path. There's actually two paths down here maybe. We're going to go down here, then we'll come back the other way.

Here, there are actual crossovers. When we come back, there are different crossover gadgets. Or you could say there's two crossover gadgets for each of these, one for going one direction, one for coming back. Yeah. Four crossovers for each intersection, for both directions and both guys.

**AUDIENCE:** I'm mostly willing to believe you, but I just have this nagging doubt that you can't actually arrange all these things and make them fit together.

**PROFESSOR:** OK. So there is a top level question which is, in general, it's the gadget assembly problem. If I have all these gadgets, can I actually put them together? And it's important, the output instance should have polynomial size. I probably should mention that here. It's important. Of polynomials. Oh, that's polynomial time algorithm. Good, good, good.

Yes. So this is in parentheses. Because it's a polynomial time algorithm, you will generate a polynomial size output, because our outputs have to be represented explicitly for reduction. So the main issue is, can you draw this in a grid of polynomial size? And the short answer to your question is, use standard graph drawing algorithms.

If I give you a planar graph, I can draw it with n vertices. I can draw it in a grid that order n by order n. I forget what the best bound is. Maybe 6n by 6n. Doesn't matter here. Now this graph is not planar. But if you just add a vertex for every intersection, and there's at most n squared intersections, then I will have a planar graph. Then I apply that algorithm. It draws everything into a grid.

I explode that grid by a factor of 100, whatever the largest size of this gadget is, plunk them in, and then I route the tunnels. So the only thing I haven't really filled in is how do you route the tunnels to make them traversable? Because if you go up, you've got to have enough stairs along the way. But it's an exercise for the reader.

There are definitely details there. And in some cases, they are subtle. I'll tell you the most annoying issue that can happen is a parity issue. Sometimes, these gadgets-- I mean, you could make them slightly wider or slightly taller. It doesn't matter. Sometimes they have to be even size or odd size. And then things don't always fit up well. And that's a pain to do. And I had a proof last month, where I had, I think, three separate parity issues in a row. I fixed one, and it's like, yes, I got the proof. And it was like, uh-oh, there's another parity problem. Then I fixed that one.

And then, uh-oh, there's another parity problem. And finally, the proof is hopefully correct. And we might go through that as an example. So there are definitely issues that can come up in gadget assembly, but this one I'm not worried about, let's say. Yeah?

**AUDIENCE:** Did the original *Super Mario Brothers* allow you to go left?

**PROFESSOR:** No. In the original *Super Mario Brothers,* you cannot scroll the screen left. Mario can go left. So this is all one screen. You always have to generalize something in your problem. And if you say the size of your screen, it's 320 by 240, or whatever in the original is constant, then you can solve Mario in polynomial time by dynamic programming. So that's not as interesting.

Mario 1. Of course, any other Mario, you can go left, except sometimes it forgets the status of your monsters. Again, if you have that, you can solve it in polynomial time by dynamic programming. So we're in the sort of, we'll say, as intended model, which is you can have a big level. 4K is already happening. Imagine the future, you have a giant screen, and you play a giant level.

**AUDIENCE:** [LAUGHING]

**PROFESSOR:** Other questions? Now, one question is is *Mario Brothers* an NP conjecture? No, I think by now we might have a proof that is PSPACE complete. But that's not published yet, or even written yet. So it's certainly not guaranteed, but we think so.

OK. Last proof is *Rush Hour*. Before I get to *Rush Hour*, I'm going to tell you about another source problem. So 3SAT is the most common problem for-- you might call them short puzzles. *Mario Brothers* is maybe a short puzzle. If you have a time limit in *Mario Brothers*, then you're guaranteed you're going to make at most, let's say, n moves, and then the game is over. You either dire or you finish. And 3-Satisfiability is a good representation of that.

For longer games, another model called constraint logic or constraint graphs is useful. So let me tell you this problem. It's in some sense simpler than 3SATs. On a graph, you have red edges and blue edges. Notice the blue edges are thicker. That means they're twice as heavy, so that you think of this as your machine.

Now a state of the machine is going to be an orientation of the graph. So every edge is just going to be oriented one way or the other. And the constraint-- in general, this is called a constrained graph. And the constraint you have to satisfy is that at every vertex, you have at least two units of flow pointed into the vertex.

So here there's actually three units of flow pointed in. There's the red single unit and the blue double unit. Blue's always 2, red is always 1. So if you get the Kindle edition of this book, don't read it on a black and white display. Or the PDF, whatever.

So what you're allowed to do in this game, the move you're allowed to do, is reverse an edge. As long as you always satisfy this invariant that at least two units are in, then you're OK. So I think here, we're going to reverse this one first. So now we have three units of flow pointed in. You have to check that the other vertice's satisfied, but this one's certainly OK.

Now because there are two red units, we can reverse the blue one. And we cannot reverse the red ones right now, because that would leave only one unit of flow inwards. There's always two. So in fact, this vertex that I've drawn is in some sense an AND gate. We call it an AND vertex, because it's not a regular gate.

Over here, we're going to think of-- it's a little bit asymmetric. We're going to think of these as the inputs to the gate, and this as the output of the gate. And if the inputs are pointing out, that's a false. If they're pointing in, that's a true. And the output is reverse, so if it's pointing in, that's a false. If it's pointing out, it's a true. So here's one state false, false. And so the AND of false and false is false.

Here's another state where both inputs are true, and then the output can be true. It doesn't have to be, though. So in this example, let's say we reverse this edge. So now, we have false and true. Still, we can't reverse this because false and true is false, and because there would only be one unit of incoming flow before we can reverse this.

If I could reverse it back, I could reverse the other one. Only once I reverse both of the inputs am I allowed to reverse the output. But I don't have. I could just let it sit there. It's not a gate in the sense that it doesn't compute the answer. But what you know is that if you have an output of true, you know that the inputs must be true. So it's kind of an AND. We call it a constraint logic AND.

Now here, this I claim is an OR. Now if you look at this in a graph, it's totally symmetric. It's just three blue edges. But if you think of these two as inputs and these two as outputs, it's an OR. And so what's the point? Well, if I reverse any of the edges, the incoming edges, the input edges I should say, then I can reverse the bottom edge. I don't have to. And I could also do both true. This could still be true, and so on.

**AUDIENCE:** So if it's symmetric, how do you constrain some of them to be inputs and others to be outputs?

**PROFESSOR:** The inputs and output distinction is only in your head. The way that I used it is I said, an input is true if it's pointing in, an output is true if it's pointing out. So that's asymmetric, and it's just a way of interpreting what's happening in the graph. the graph knows no difference.

This is useful, of course, because your output is the next vertex's input probably if you're building a bunch of ANDS and ORs. If you're building a regular circuit. So you want that definition to be asymmetric. It gets a little bit confusing, and we will spend a lot more time on constraint logic. This is more of a teaser. But that's just in the naming of things. Naming of true and false.

OK. So constraint logic, let's say, what's the decision problem? I give you such a graph. Notice every vertex is either an AND, two red and a blue, or an OR, three blues. And I want to know, can I reverse that edge? This is actually a crossover gadget if you're curious. In this world, crossover gadgets, you can just build using ANDs and ORs.

There's no NOTs in this world. NOTs don't even make sense because you can't force anything. But it turns out this problem is PSPACE complete. So even harder than 3SAT, assuming NP does not equal PSPACE. And the cool thing is once you have developed all that infrastructure, if you want to take a puzzle like *Rush Hour*-- so these are cars. Each car can slide in the direction of the car, no turns allowed, and your goal is to get some car out, or to move some car at all, let's say, I need to do two things.

I need to construct an AND date and construct an OR gate, and then I need to check that they fit together. So let me convince you this is an AND with this kind of orientation. And yeah. So this is going to feel a little backwards. Inward pointing means that the block is out, and outward pointing means the block is in. Ignore this picture for the moment.

What happens here is that if you want to push C into this gadget, imagine the dark gray blocks are rigid. They can't they can never move. That's going to be true. We'll see why in a moment. So then it's just about these inner guys moving.

If you want to move C in one unit, that will be possible if and only if A moves out and B moves out by one unit. A moves out by one unit, B moves out by one unit, then this block can slide here, this block can slide here. This block can slide one, this block can slide two, and then C can move in one.

On the other hand, this is an OR gate, OR vertex. If A moves out one or B moves out one, then either E can move down, or this guy can move down. Once either of those moves down, D can move all the way to the right or all the way to the left, and then C can move down one.

This is called a protected Or. Small, subtle detail. If A and B move out, then everything can fall apart. So basically, there's a proof in this book that says you don't need to worry about that. We can guarantee that we'll never have both A and B move out. Only one of them will happen. That's the protected part.

So that's basically the proof of PSPACE-completeness in two pictures. This is the cool thing about constraint logic you don't need a crossover gadget because those always happen for free. You just need to check that you can fit the gadgets together. And this is the intended tiling to fit them together. And you can check that as long the box only move at most one unit at any time, then these gray regions are solid all the way through. I think I need to add some more filler here. And therefore, the gray blocks can never move.

And so then, you can analyze each gadget one at a time and construct any constraint logic graph in this way. Questions? Yeah?

**AUDIENCE:** So for that graph orientation problem, I do not fully understand. So you're given a graph--

**PROFESSOR:** You're given a graph and an orientation, and you want to know, can I reverse this one edge? So in this world, it means you're given an initial placement of all the blocks, and you know whether each thing is in or out, because you know whether the edge is pointed to the left or pointed to the right. And you want to know, can I move this one block?

**AUDIENCE:** OK. And then for this graph, you satisfied the property that each vertex has at least two units [? of input. ?]

**PROFESSOR:** Yes, the input-oriented graph should already satisfy the invariant. And then at every move you do, you have to also satisfy the invariant.

**AUDIENCE:** Can you give a small example of a graph which satisfies that [INAUDIBLE]?

**PROFESSOR:** A small example. Like this one? This is not a single graph. But if I connect this edge to here, and this edge to here, that will satisfy the property. And it's one graph. I don't have a small example offhand but I think you could make. One

It's an interesting question, what's the smallest satisfied graph? Probably 10 vertices or something should suffice. I should put it on the problem set. But later. We'll talk about constraint logic more later. More questions?

So we did a lot. We proved Mario is NP-complete. I mean, some hand waving involved. We proved that *Rush Hour* is PSPACE-complete if you believe that constraint logic PSPACE-complete. In general, this whole class is about reductions, about taking a known hard problem and converting it into your problem. We will see a ton of them, and each kind of type of problem has a different flavor.

There's a whole range of different 3SAT proofs. This is not the only form of 3SAT. There are probably a dozen of them . But they all follow the same pattern, which is come up with a clause gadget, come up with a variable gadget, come up with ways to connect them together. And that's a very useful way of thinking about things for some problems.

And then for other problems, you might use things like constraint logic. Constraint logic is most relevant when it's designed around games and puzzles, but it comes up in other scenarios too. I think in graph labeling, there have been proofs of PSPACE-completeness using constraint logic.

Next class, we'll talk about a problem called three partition, which is really useful for problems that involve numbers and adding up numbers, where 3SAT isn't really useful but three partition turns out to be the right thing. So that will be next. And that's all for today.