The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

ERIC DEMAINE: All right, today we do NP completeness, an entire field in one lecture. Should be fun. I actually taught an entire class about this topic last semester, but now we're going to do it in 80 minutes. And we're going to look at lots of different problems, from Super Mario Brothers to jigsaw puzzles, and show that they're NP -complete.

This is a fun area. As Srini mentioned last class, it's all about reductions. It's all about converting one problem into another, which is a fun kind of puzzle in itself. It's an algorithmic challenge. And we're going to do it a lot.

But first I'm going to remind you of some of the things you learned from 006, and tell you what we need to do in order to prove all of these relations, what exactly we need to show for each of those arrows, and why it's interesting.

So this is generally around the P versus NP problem. So remember, P is all the problems we know how to solve in polynomial time. Well not just the ones we know how to solve, but also the ones that can be solved, which is pretty much-- which is the topic of 6.006, and 6.046 up till now. But for now, in the next few lectures, we'll be talking about problems that are probably not polynomially solvable, and what to do about them.

Polynomial, as you now, is like n to the some constant. Polynomial good exponential bad. What is n? I guess n is the size of the problem, which we'll have to be a little bit careful about today. And then NP is not problem solvable not in polynomial time, but it's problem solvable in nondeterministic polynomial time.

And in this case we need to focus on a particular type of problem, which is decision problems. Decision just means that the answer is either yes or no. So it's a single bit answer. We will see why we need to restrict to that kind of problem in a moment.

So this is problems you can solve in polynomial time. Same notion of polynomials, same notion of n, but in a totally unrealistic model of computation. Which is a nondeterministic model. In a nondeterministic model, what you can do is say instead of computing something from

something you know, you could make a guess. So you can guess one out of polynomially many options in constant time.

So normally a constant time operation, in regular models, like you add two numbers, or you do an if, that sort of thing. Here we can make a guess. I give the computer polynomially many options I'm interested in. Computer's going to give me one of them. It's going to give me a good guess. Guess is guaranteed to be good. And good means here that I want to get to a yes answer if I can. So the formal statement is, if any guess would lead to a yes answer, then we get such a guess. OK, this is weird. And it's asymmetric. It's biased towards yes. And this is why we can only think about decision problems, yes or no. You could bias towards no. You get something else called coNP. But we'll focus here just on NP.

So the idea is I'd really like to find a guess that leads to a yes answer. And the machine magically gives me one if there is one. Which means if I end up saying no, that means there was absolutely no path that would lead to a yes. So when you get a no, you get a lot of information. When you get a yes, you get some information. But hey, you were lucky. Hard to complain. So in 006, I often call this the lucky model of computation. That's the informal version. But nondeterminism is what's really going on here.

So maybe it's useful to get an example. So here's a problem we'll-- this is sort of the granddaddy of all NP-complete problems. We'll get to completeness in a moment. 3SAT-- SAT stands for satisfiability. So in 3SAT, the input to the problem looks something like this. I'm just going to give an example. And in case you've forgotten your weird logic notation, this is an and. These are ORs. And I'm using this for negation, not.

So in other words, I'm given a formula which is and of ORs. And each or clause only has three things in it. These things are called literals. And a literal is either a variable x sub i, or it's the negation of a variable, not x sub i.

So this is a typical example. You could have no negations. You could here have one negation, two negations, any number of negations per clause. These groups of three-- these or of three things, three literals, are called clauses. And they're all ANDed together. And my goal is, this should be a decision question, so I have a yes or no question. And that question is, can you set the variables-- So they're x1 to true or false? So each variable I get to choose a true or false designation such that the formula comes out true.

I use T and F for true and false. So I want to set these variables such that every clause comes out true, because they're ANDed together. So I have to satisfy this clause in one of three ways. Maybe I satisfy it all three ways. Doesn't matter, as long as at least one of these should be true, and at least one of these should be true, and at least one of each clause should be true.

So that's the 3SAT problem. This is a hard problem. We don't know a polynomial time algorithm. There probably isn't one. But there is a polynomial time nondeterministic algorithm. So this problem is in NP because if I have lucky guesses, it's kind of designed to solve this kind of problem. What I'm going to do is guess whether x1 is true or false.

So I have two choices. And I'm going to ask my machine to make the right choice, whether it should be true or false. Then I'll guess x2. Each of these guess operations takes constant time. So I do it for every variable. And then I'm going to check whether I happen to satisfy the formula. And if it comes out true, then I'll return yes. And if it comes out false, I'll return no.

And because NP is biased towards yes answers, it always finds a yes answer if you can. If there's some way to satisfy the formula, then I will get it. If there's some way to make the formula come out true, then this algorithm will return yes. If there's no way to satisfy it, then this nondeterministic algorithm will return no. That's just the definition of how nondeterministic machines work. It's a little weird. But you can see from this kind of prototype of a nondeterministic algorithm, you can actually always arrange for your guessing to be at the beginning. And then you do some regular polynomial time checking or deterministic checking.

So when you rewrite your algorithm like this with guesses up front and then checking, you can also think of it as a verification algorithm. So you can say, your friend claims that this 3SAT formula is satisfiable, meaning there's a way to set the variable so that it comes out true. So this is called a satisfying assignment. Satisfying just means make true. And you're like, no, I don't believe you. And your friend says no, no, no, really, it's true. And here's how I can prove it. You set x1 to false. You set x2 to true. You set x3-- basically they give you the guesses.

And then you don't have to be convinced that those are the right guesses, you can check that it's the right guess. You can compute this formula in linear time, see what the outcome is. If someone tells you what the xi's are, you can very quickly see whether that was a satisfying assignment. So you could call this a solution, and then there's a polynomial time verification algorithm that checks that solutions are valid.

But, you can only do that for yes answers. Your friend says no, this is not satisfiable, they have no way of proving it to you. I mean, other than checking all the assignments separately, which would take exponential time, there's no easy way to confirm that the answer to this problem is no. But there is an easy way to check that the answer is yes, namely I give you the satisfying assignment.

So this definition of NP is what I'll stick to. It's this sort of-- I like guessing because it's like dynamic programming. With dynamic programming we also guess, and guessing actually originally comes from this world, nondeterminism. In dynamic programming, we don't allow this kind of model. And so we have to check the guesses separately. And so we spend lots of time. Here, magically, you always get the right guess in only constant time.

So this is a much more powerful model. Of course there's no computers that work like this, sadly, or I guess more interestingly. So this is more about confirming that your problem is not totally impossible. At least you can check the answers in polynomial time. So that's one thing.

So this is an equivalent definition of NP because you can take a nondeterministic algorithm and put the guessing up top. You can call the results of those guesses a certificate that an answer is yes. And then you have a regular old deterministic polynomial time algorithm that, given that certificate, will verify that it actually proves that the answer is yes. It's just that certificate has to be polynomial size. You can't guess something of exponential size. You can only guess something of polynomial size in this model.

So seems a little weird. But we'll see why this is useful in a little bit. So let me go to NP completeness. So if I have a problem X, it's NP-complete if X is in NP and X is NP-hard. But I haven't told you what NP-hard is. Maybe you remember from 006, but let me remind you.

So, I need to define reduce. So maybe I'll do that as well, then we can talk about all these.

OK, a lot of definitions. But the idea of NP hardness is very simple. If problem X is NP-hard, it means that it's at least as hard as-- sorry, that is a Y-- it's at least as hard as all problems in NP. Intuitively, X means it's at least as hard as everything in NP. Whereas being in NP is a positive statement. That says it's not too hard, at least there's a polynomial time verification algorithm. So being in NP is good news. It says you're no harder than NP. NP-hard says you're at least as hard as everything in NP. And so NP-complete is a nice answer because this says you're exactly as hard as everything in NP-- no harder, no easier.

If you draw, in this vague sense, computational difficulty on one axis-- which is not really accurate, but I like to do it anyway-- and you have P is all of these easy problems down here. And NP is some larger set like this. NP-hard is from here over. And this point right here is NP-complete.

Being in NP means you're left of this line, or on the line. And being NP-hard means you're right of this line, or on the line. NP-complete means you're right there. So that's a very definitive sense of hardness.

Now there is this slight catch, which is we don't know whether P equals NP. So maybe this is the same as this, but probably not. Unless you believe in luck, basically, unless you imagine that a computer could engineer luck and always guess the right things without spending a lot of time, then P does not equal NP. And in that world, what we get is that if you have an NP-complete problem, or actually any NP-hard problem, you know it cannot be NP.

So if you have that X is NP-hard, then you know that X is not in P unless all of NP is in P. So unless P equals NP. And most reasonable people do not believe this. And so instead they have to believe this, that your problem is not polynomially solvable.

So why is this true? Because if your problem is NP-hard, it is at least as hard as every problem in NP. And if you believe that there is some problem in NP-- we don't necessarily know which one-- but if there is any problem out there in NP that is not in P, then X has to be at least as hard as it. So it also requires nonpolynomial time, something larger than polynomial time.

What does at least as hard mean though? We're going to define it in terms of reductions. Reduction from one problem to another is just a polynomial time algorithm, regular deterministic polynomial time, that converts an input to the problem A into an equivalent input to problem B. Equivalent means that it has the same yes or no answer. And we'll just be thinking about decision problems today.

So why would I care about a reduction? Because what it tells me is that if I know how to solve problem B, then I also know how to solve problem A. If I have a, say, a polynomial time algorithm for solving B and I want one for A, I just take my A input. I convert it into the equivalent B input. Then I run my algorithm for B, and then it gives me the answer to the A problem because the answers are the same.

So if you have a reduction like this and if say, B, has a polynomial time algorithm, then so does

A, because you can just convert A into B, and then solve B. Also this works for nondeterministic algorithms. Not too important.

So what this tells us is that in a certain sense-- get this right-- well this is saying, if I can solve B, that I can solve A. So this is saying that B is at least as hard as A. I think I got that right, a little tricky.

So if we want to prove the problem is NP hard, what we do is show that every problem in NP can be reduced to the problem of X. So now we can go back and say well, if we believe that there is some problem Y, that is in NP minus P, if there's something out here that is not in P, then we can take that problem Y, and by this definition, we can reduce it to X, because everything in NP reduces to X. And so then I can solve my problem Y, which is in NP minus P, by converting it to X and solving X. So that means X better not have a polynomial time algorithm, because if it did, Y would also have a polynomial time algorithm. And then in general, P would equal NP, because every problem in NP can be converted to X. So if X has a polynomial time algorithm, then every problem Y does. Question?

AUDIENCE: For the second if statement, why can't you say that if A is in NP, B is in NP?

ERIC DEMAINE: So you're asked us about the reverse question. If is A in NP, can we conclude that B is in NP? And the answer is no. Because this reduction only lets us convert from A to B. It doesn't let us do anything for converting from B to A. So if we know how to solve A and we also know how to convert A into B, it doesn't tell us anything. It could be B is a much harder problem than A, in that situation. That's, I think, as good as I can do for that. Other questions?

All right. It is really tricky to get these directions right. So let me give you a handy guide on how to not make a mistake. So maybe over here.

What we care about, from an algorithmic perspective, is proving the problems are NP-complete. Because if we prove NP-completeness-- I mean, really we care about NP-hardness, but we might as well do NP-completeness. Most of the problems that we'll see that are NP-hard are also NP-complete.

So when we prove this, we prove that there is basically no polynomial time algorithm for that problem. So that's good to know, because then we can just give up searching for a polynomial time algorithm. So all the problems we've seen so far have polynomial time algorithms, except a couple in your problem sets, which were actually NP-complete. And the best you could have

done was exponential, unless P equals NP. So here's how you can prove this kind of lower bound to say look, I don't need to look for algorithms any more because my problem is just too hard. It's as hard as everything in NP.

So this is just a summary of those definitions. The first thing you do is prove that X is in NP. The second thing you do is prove that X is NP-hard. And to do that, you reduce from some known NP-complete problem-- or I guess NP-hard, but we'll use NP-complete-- to your problem X. Maybe I'll give this a name Y.

OK, so to prove that X is in NP, you do something like what we did over here, which is to give a nondeterministic algorithm. Or you can think of it as defining what the certificate is and then giving a polynomial time verification algorithm.

So sort of two approaches. You can give a nondeterministic polynomial time algorithm, or you give a certificate and a verifier. There's no right or wrong certificate. I mean, a certificate, you can define however you want, as long as the verifier can actually check it and when it says yes, then the answer to the problem was yes. So it's really the same thing. Just want to say there's some certificate that a verifier could actually check. So that's proving that your problem is in NP. It's sort of an algorithmic thing.

The second part is all about reductions. Now the definition says that I should reduce every problem in NP to my problem X. That's tedious, because there are a lot of problems in the world. So I don't want to do it for every problem in NP. I'd like to just do it for one.

Now if I reduce sorting to my problem, that's not very interesting. It says my problem is at least as hard as sorting. But I already know how to solve sorting. But if I start from an NP-complete problem, then I know, by the definition, that every problem in NP can be reduced to that problem. And if I show how to reduce the NP-complete problem to me, then I know that I'm NP-complete too. Because if I have any problem Z in NP, by the definition of NP-complete of Y I can reduce that to Y. And then if I can build a reduction from Y to X, then I get this reduction. And so that means I can convert any problem in NP to my problem X, which means X is NP-hard. That's the definition.

So all this is to say the first time you prove a problem is NP-complete in the world-- this happened in the '70s by Cook. Basically he proved that 3SAT is NP-complete. That was annoying, because he had to start from any problem in NP, and he had to show that you could reduce any problem in NP to 3SAT.

But now that that hard work is done, our life is much easier. And in this class all you need to think about is picking your favorite NP-complete problem. 3SAT's a good choice for almost anything, but we'll see a bunch of other problems today from here. And then reduce from that known problem to your problem that you're trying to prove is NP-hard. If you can do that, you know your problem is NP-hard. So we only need one reduction for each hardness result, which is nice.

And this picture is a collection of reductions. We're going to start from 3SAT. I'm not going to prove that it's NP-complete, although I'll give you a hint as to why that's true. We're going to reduce it to Super Mario Brothers. We're going to reduce it to three dimensional matching. We're going to reduce three dimensional matching to subsets sum, to partition, to rectangle packing, to jig saw puzzles. And we're going to do all those reductions, hopefully. And that's proving NP-hardness of all those problems. They're also all in NP.

So 30 second intuition why 3SAT is NP-hard. Well, if you have any problem in NP, that means there is one of these nondeterministic polynomial time algorithms, or there is some verifier given a polynomial size certificate. So that verifier is just some algorithm. And software and hardware are basically the same thing, right? So you can convert that algorithm into a circuit that implements the algorithm. And if I have a circuit with like ANDs and ORs and NOTs, I can convert that into a Boolean formula with ANDs, ORs, and NOTs. Circuits and formulas are about the same.

And if I have a formula-- fun fact, although this is a little less obvious-- you can convert it into this form, an AND of triple ORs. And once you've done that, that formula is equivalent to the original algorithm. And the inputs to that verification algorithm, the certificate, are represented by these variables, the xi's. And so deciding whether there's some way to set the xi's to make the formula true is the same thing as saying is there some certificate where the verifier says yes, which is the same thing as saying that the problem has answer yes. So given an NP algorithm, one of these nondeterministic funny algorithms, we can convert it into a formula satisfaction problem. And that's how you prove 3SAT is NP-complete. But to do that can take many lectures, so I'm not going to do the details. The main annoying part is being formal about what exactly an algorithm is, which we don't do in this class. If you're interested, take 6.045, which is some people are actually in the overlap this semester.

Cool. Let's do some reductions. This is where things get fun. So we're going to start with

reducing 3SAT to Super Mario Brothers. So how many people have played Super Mario Brothers? Easy one. I hope if you haven't played, you've seen it, because we're going to rely very much on Super Mario Brothers physics, which I hope is fairly intuitive. But if you haven't played, you should, obviously. And we're going to reduce 3SAT to Super Mario Brothers.

Now this is a theorem by a bunch of people, one MIT grad student, myself, and a couple other collaborators not at MIT. And of course this result holds for all versions of Super Mario Brothers so far released, I think. The proofs are a little bit different for each one, especially Mario 2, which is its own universe. What I'm going to talk about the original Super Mario Brothers, NES classic which I grew up with.

Now the real Super Mario Brothers is on a 320 by 240 screen. It's a little bit small. Once you go right, you can't go back left, except in the maze levels anyway. So I need to generalize a little bit. Because if you assume that the screen size of Super Mario Brothers is constant, in fact you can dynamic program your way through and find the optimal solution in polynomial time.

So I need to generalize a little bit to arbitrary board size, arbitrary screen size. So in fact, my entire level will be in one screen, no scrolling. Never mind this is a side scrolling adventure. And so that's my generalized problem. And I claim this is NP-hard. If I give you a level and I ask you, can you get to the end of this level? That problem is NP-hard. Also no time limit. The time limit would be OK, but you have to generalize it. Instead of 300 seconds or whatever, it has to be an arbitrary value.

So how are we going to do this? We're going to reduce from 3SAT to Super Mario Brothers. So that means I'm given-- I don't get to choose. I'm given one of these formulas. And I have to convert it into an equivalent Super Mario Brother instance. So I have to convert it into a level, a hypothetical level of Super Mario Brothers. Given a formula, I have to build a level that implements that formula.

So here's what it's going to look like. I'm going to start out somewhere. Here's my drawing of Mario. Mario-- or you could play Luigi. It doesn't matter. First thing it's going to do is enter a little black box called a variable. This is supposed to represent, let's call it x1. And so it's some black box. I'm going to tell you what it is in a moment. And it has two outputs. There's the true output and the false output. And the idea is that Mario has to choose whether to set x1 to true or false. Let me show you that gadget.

So here's the-- whoops, upside down-- here is the variable gadget. So here's Mario. Could enter from this way or that way. We'll need a couple of entrances in a moment. And then falls down. Once Mario is down here, if you check the jump height, you cannot get back up to here. So this is like a one way.

Once you're down here, you have a choice. Should I fall to the left or fall to the right? And if you make these falls large enough, once you fall, you can't unfall. So once you make a choice of whether I leave on the true exit or the false exit, that's a permanent choice. So you can't undo it, unless you can come back to here. But we'll set up so that never happens.

I mean, if you're trying to solve the level, you don't know which way to go. You have to guess. Can I go fall to the left or fall to the right, or do something. So the existence of a play through, this level, is the same as saying there is a choice for the x1 variable.

Now we have to do this for lots of variables. So there's x2 variable, x3 variable, and so on. Each one has a true exit and a false exit. So the actual level will have n instances of this if we have n variables.

Now, what do I do once Mario decides that this is a true thing? What I'm going to do is have-- this is called a gadget by the way. In general, most NP-hardness proofs use these things called gadgets, which is just saying, we take various features of the input, and we convert them into corresponding features on the output. So here I'm taking each variable, x1, x2, x3, and so on, and building this little gadget for each of those variables.

Now the other main thing you have in 3SAT are the clauses. We have triples of variables or their negations. They have to come together and be satisfied. One of them has to be true.

So down here I'm going to have some clause gadgets, which I will show you in a moment. OK, and I think I'll switch colors. This is about to get messy.

So the idea is that some of the clauses have x1 in them. The true version of x1, not x1 bar. So for those clauses, I want to connect. I'm going to dip into the clause briefly. So from this wire going to dip into the clause here. And then I'm going to go to the next clause that has x1. Maybe it's this one, and the next one, and so on. All the clauses that have x1 in it, I dip into. The other ones I don't. And then once I'm done, I'm going to come back and feed into x2.

Next, I look at this false wire for x1. So all the clauses that have x1 bar in them, I'm going to

connect. So I don't know which ones they are. Maybe this one, or this one, something. And then I come here.

And so the idea is that Mario makes a choice whether x1 is true or false. If x1 is true, Mario is going to visit all of the clauses that have x1 true in them. And then it's going to go to the x2 choice. Then it's going to choose whether x2 is true or false, and repeat. Or Mario decides x1 should be false. That will satisfy all the clauses that have x1 bar in them. And then again, we feed back into x2.

So this is why we have two inputs into the x2 gadget. One of them is when the previous variable was true. The other is when the previous variable was false. The choice of x2 doesn't depend on the choice of x1. So they feed into the same thing. And you have to make your choice. So far, so good.

Now the question is, what's happening in these clauses. And then there's one other aspect, which is after you've set all of the variables, at the very end, after this last variable xn, at the very end, what we're going to do is come and go through all the clauses. And then this is the flag. This is where you win the level. Sorry, I drew it backwards. But the goal is for Martin to start here and get to here. In order to do that, you have to be able to traverse through these clauses.

So what do the clauses look like? This is a little bit more elaborate. So here we are. This is a clause gadget. So there are three ways to dip into the clause. It's actually upside down relative to that picture, but that's not a problem.

So if Mario comes here, then he can hit the question mark from below. And inside this question mark is an invincibility star. And the invincibility star will come up here and just bounce around forever. We checked. The star will just stay there for as long as you let it sit.

Unfortunately, all of these are solid blocks, so Mario can't actually get up to here to get the star. But as long as Mario can visit this question mark or this question mark or this question mark, then there will be at least one star up here. So the idea is that each of these represents one of the literals that's in the clause. And if we choose-- so let's look at this first clause, x1 or x3 or x6 bar. So if we choose x1 to be true, then we'll follow the path and we'll be able to hit the star. Or if we choose x3 to be true, then we'll come in here and hit this star. Or if we choose x6 to be false, then that path will lead to here and we'll be able to hit this question mark and get the star up here. So as long as we satisfy the clause, there will be at least one

star. Won't help if you have multiple stars.

Then the final traversal part-- so that was this first clause. And now we're traversing through. Actually in this picture, it's left to right. Just turn your head. And so now Mario is going to have to traverse this gadget from left to right on this top part. And if Mario comes in here and you can barely jump over that. If there's a star, you can collect the star and then run through all of these flaming bars of death. If there's no star, you can't. You'll die if you try to traverse.

So in order to be able to traverse all these clauses, they must all be true. And them all being true is the same is their AND being true. So you will be able to survive through all these clauses if and only if this formula has a satisfying assignment. The satisfying assignment would be given to you by the level play. The choices that Mario makes in this gadget will tell you whether each variable should be true or false.

So to elaborate just a little bit more in general, when you have a reduction like this, to prove that it actually works, you need to check two things. You need to check that if there is a way to satisfy this formula, then there is a way to play this level. And then conversely you need to show that if there's a way to play this level, then the formula has a satisfying assignment.

So for that latter part, in order to convert a level play into a satisfying assignment, you just check which way Mario falls in each of these gadgets, left or right. That tells you the variable assignment. And because of the way the clauses work, you'll only be able to finish the level if there was at least one star here. And stars run out after some time. So you can barely make it through all the flaming bars of death. Then you get to the next clause. You need another star for each one.

Conversely, if there is a satisfying assignment, you can actually play through the level, you just make these choices according to what the satisfying assignment is. So either way it's equivalent. We always get a yes or no answer here whenever we get a corresponding yes or no answer to the 3SAT process.

You also need to check that this reduction is polynomial size. It can be computed in polynomial time. So there's an issue. Given this thing, you have to lay this out in a grid and draw all these wires.

And there's one problem here, which is, these wires cross each other. And that's a little awkward, because these wires are basically just long tunnels for Mario to walk through. But

what does it mean to have a crossing wire? Really, if Mario's coming this way, I don't want them to be able to go up here. He has to go straight. Otherwise this reduction won't work.

So I need what's called a crossover gadget. And everywhere here I have a crossing, I have a crossover. And this gadget has to guarantee that I can go through one way or the other way, but there's no leakage from one path to the other path.

Actually, if I first traverse through here, and then I traverse through here, it's OK if I leak back. Because once I visit a wire, it's kind of done. But I can't have leakage if only one of them is traversed.

So this is the last gadget, the most complicated of them all. So this took a while to construct, as you might imagine. So this is what we call a unidirectional crossover. You can either go from left to right or from bottom to top, but you cannot go from bottom to right or bottom to left or left to bottom, that kind of thing.

So I'm told that Mario is only going to enter from here to here, because all of these wires, I can make one way wires. I only have to think about going in a particular direction. I can have falls to force Mario to only go one way along these wires.

And so let me show you the valid traversals. Maybe the simplest one is from here. So let's say Mario comes in here, falls. So I can't backtrack, can jump up here. And then if Mario's big, he can break this block, break this block. But if he's big-- there should be a couple more zig zags here. Let's try to run. You can crouch slide through here. But then you'll sort of lose your momentum, and you won't be able to go through all these traversals as big Mario.

So you can break these blocks and then get up to the top and leave. Or, if big Mario comes from over this way, you can first take a damage, become small Mario. Then you can fit through these wiggly blocks. But you cannot break blocks anymore as small Mario. So once you've committed to going small, you have to stay small, until you get to here. And then there's a mushroom in this block. So you can get big again, and then you can break this block and leave. But once you're big, you can't backtrack because big Mario can't fit through these tiny tubes. See it clear, right?

So slight detail, which is at the beginning, we need to make Mario big-- so there's a little mushroom. I think they have three spots-- at the beginning. And also at the end, there has to be something like this that checks that you actually have a mushroom. So the only time you're

allowed to take damage is briefly in this gadget you take damage. If you tried to backtrack, you would get stuck. There's a long fall here. And then you have to get the mushroom so you can escape again. So at the end there's like a mushroom check. Make sure you have it.

So most of the time Mario is big. And just in these little crossovers you have to make these decisions. This would make a giant level, but it is polynomial size, probably quadratic or something. Therefore Super Mario Brothers is NP-hard.

So if you want more fun examples like this, you should check out 6.890, the class I taught last semester, which has online video lectures, soon to be on OpenCourseWare. So you can play with that.

Any questions about Mario? All right, I hope you all play.

So the next topic is a problem you probably haven't heard about, three dimensional matching. This is a kind of a graph theory problem. We're going to call it 3DM for short. And you've seen matching problems based on flow. Matching problems are usually about pairs of things. You're pairing them up, which you might call two dimensional matching. That can be solved in polynomial time. But if you change two to three and you're tripling things up, then suddenly the problem becomes NP-complete. So it's a useful starting point, similar to 3SAT.

So you're given a set X of elements, a set Y of elements, a set Z of elements. None of them are shared. But more importantly, you are given a bunch of triples. These are the allowable triples. So we'll call the set of allowable triples T. And so we're looking at the cross product. This is the set of all triples X, Y, and Z, or X is in X, Y is in Y, and Z is in Z. But not all triples are allowed. Only some subset of triples is allowed. And your goal is to choose among those subsets-- sorry, among those triples a subset of the triples. So we're trying to choose a subset S of T such that every element-- so the things in X, Y, and Z are called elements. So I'm just taking somebody in the union XYZ. It should be in exactly one triple s in big S.

This is a little weird, but you can think of this problem as you have an alien race with three genders-- male, female, neuter I guess. Those are the X, Y, and Z's. There's an equal number of each. And every triple reports to you whether that is a compatible matching. Who knows what they're doing, all three of them?

So you're told up front-- you take a survey. There's only n cubed different triples. For each of them they say, yeah, I'd do that. So you were given that subset. And now your goal is to

permanently triple up these guys. And everybody wants to be in exactly one triple. So it's a monogamous race, imagine. So everybody wants to be put in one triple, but only one triple. And the question is, is this possible? This is three dimensional matching. Certainly not always going to be possible, but sometimes it is. If it is, you want to answer yes. If it's not possible, you want to answer no.

This problem is NP-complete. Why is it in NP? Because I can basically guess which elements of T are in S. There's only at most n cubed of them. So for each one, it is guess yes or no, is that element of T in S? And then I check whether this coverage constraint holds.

So it's very easy to prove this is in NP. The challenge is to prove that it's NP-hard. And we're going to do that, again, by reducing from 3SAT.

So we're going to make a reduction from 3SAT to three dimensional matching. Direction is important. Always reduce from the thing you know is hard and reduce to the thing you don't know is hard.

So again, we're given a formula. And we want to convert that formula into an equivalent three dimensional matching input. So the formula has variables and clauses. For each variable, we're going to build a gadget that looks like this. And for each clause we're going to build a gadget.

So here's what they look like. If we have a variable x1, we're going to convert that into this picture. Stay monochromatic for now. Looks pretty crazy at the moment, but it's not so crazy.

This is not supposed to be obvious. You have to think for a while. It's a puzzle to figure out this kind of thing. But I call this thing a variable gadget because locally-- so there's basically a wheel in the center here. And then there's these extra dots for every pair of dots, consecutive pairs of dots in a wheel. And what I've drawn is the set of triples that are allowed. There's tons of other triples which are forbidden. The triples that are in T are the ones that I draw as little triangles.

And two color them because there are exactly two ways to solve this gadget locally. Now these dots are going to be connected to other gadgets. But these dots only exist in this gadget, which means they've got to be covered. They've got to be covered exactly once.

So either you choose the blue triangles, or you choose the red triangles. Each of them will exactly cover each of these guys once. You cannot mix and match red and blue, because you

either get overlap if you choose two guys that share a point, or you'd miss one. If I choose like this blue and this red, then I can't cover this point because both of these would overlap those two. And over here you have to choose [INAUDIBLE] triples. They can't overlap at all. And everybody has to get covered.

So just given those constraints, locally you can see you have to choose red or blue. Guess what? One of them is true, the other one is false. Let's say that red is true and blue is false. In general, when you're trying to build a variable gadget, you build something that has exactly two solutions, one representing true, one representing false.

Now how big do I make this wheel? Big enough. You could make it as big as the number of clauses. I'm going to make it into two and x1. So wheel-- and this number is the number of occurrences of x1 in the formula. So this is the number of clauses that contain either xi or xi bar. That's in xi.

I'm going to double that. Because what I get over here is basically xi being true for those guys. Actually, yeah, that's actually right. It looks backwards. And false for these guys. One way or the other, we'll figure it out.

So in order for xi to appear in, say, five different clauses, I want five of the true things and five of the false things. And so I need to double in order to get-- potentially I have twice as many as I actually need, but this way I'm guaranteed to have false or true, whichever I need. In reality I have some true occurrences. I have some false occurrences, some x1's, some x1 bars. This will guarantee that I have enough of these free points to connect into my clause gadgets.

How do I do a clause gadget? It's actually really easy. So these would be pretty boring by themselves. So a clause always looks like this. Maybe there's some negations. Yeah, let's do something like that. I'm going to convert it into a very simple picture. It's going to be xi dot, and xj bar dot, and xk dot. And then-- well maybe I'll stick to these colors.

Again, these two points only appear in this clause gadget. These dots are actually these dots. So there's one of these pictures for x1. There's another one for x2, x3. And so xi has one of these wheels. I want this dot to be one of these dots of the wheel. And then I want this dot to be one of the dots in the xj wheel with the false setting, one of the red dots. I want this one to be xk true setting in the xk wheel.

So these things are all connected together in a complicated pattern. But the point is that within

this gadget, I only have three allowed triples. And these points only appear in this gadget, which means they have to be covered in this gadget. They can be covered by this triple or this triple or this triple. But once you choose one, you can't choose the others.

What this means is if I set x1 to be true, it leaves behind these points marked true. If I choose the red things, then it's the blue points that are left behind. Leaving points behind in this case is going to be good, because this clause, in order to satisfy this clause, in order to choose one of these three triples, at least one of these must be left behind by the wheel. If all of these are covered by their wheels, then there's no way. I can't choose any of these guys. But if at least one of these is left behind by the wheel, then I can choose the corresponding triple and cover these points. So I'll be able to cover these points if and only if at least one of these is true.

And that's a clause. That's what a clause is supposed to do in 3SAT. If at least one of these is true, then the clause is satisfied. I need all the clauses to be satisfied because I need to cover of these points for all the instances of these clauses. And that's how it works.

Now, slight catch. If you do this, not all the points will be covered, even so. Maybe all of these are true. And so they're all left behind. And I can only cover one of them with the clause.

It's a little messy. You need another gadget, which is called garbage collection. I don't want to spend too much time on it. But you have two dots. And then you have every single xi-- these dots, all true and false dots. And you're going to have this triple, and this triple, and this triple, and this triple, and so on. It looks an awful lot like a clause. But this is like a clause that's connected to everybody in the entire universe. And you repeat this the appropriate number of times, which is something like sum of nx minus the number of clauses.

OK, why? Because if you look at a wheel, it has size 2 times nx for a variable x. And half of the points will be left uncovered. So that means nx of them will be uncovered.

Then the clause, if everything works out correctly, the clause will cover exactly one of those points. So for each clause we cover one of the points. That means this difference is exactly how many points are left uncovered. And so we make this gadget exactly that many times. And it's free to cover anybody. So whatever is left over, this garbage collector will clean up. And if we use exactly the right number of them, this garbage collector won't run out of things to collect. So this makes the proof messy. But I want to move on to somewhat simpler proofs and for other problems. Yeah?

AUDIENCE: Real quick, what about the t or f points that we didn't cover because we didn't actually need that many?

ERIC DEMAINE: Right. So this also includes the points that weren't even connected to clauses. I think this is the right number no matter what, because this is counting the total number of uncovered guys, whether they're connected to clauses or not. Each clause will, in a satisfied situation, it will cover exactly one of those points. The ones that are connected to the clauses won't be covered at all, but that will still be in this difference. So yeah, it's good to check that. The first time I wrote this down I forgot about those points and got it wrong. But I think this is right, hopefully.

I did not come up with this proof. Garey and Johnson I think-- or no. This is-- I forgot. Yeah, this is a Garey and Johnson proof. There's a cool book from the late '70s by Garey and Johnson, does a lot of NP-completeness, if you're curious.

All right, so hopefully you believe three dimensional matching is hard. Now I'm going to use it to prove that some very different types of problems are hard. This is a kind of graph theory problem. You'll see more graph theory problems in recitation. This one, I can erase 3SAT and Mario.

So in the world, most NP-hardness proofs are reductions from 3SAT, or some variation of 3SAT. In some sense, you can think of three dimensional matching as kind of like a version of 3SAT, but it's a little bit more stringent. And that stringency helps us to do other reductions.

So here's another problem where we'll reduce from three dimensional matching. It's called subset sum. So you're given n integers, a1 up to an. And you're given a target sum, also an integer. Call it t. What you'd like to know is, is there a subset of the integers that adds up to that target. Can you choose a sum of the integers so that-- I'll write it the sum of S. But what this means is the sum over the ai's that are in S of the value ai. I want that to equal t. So this is the definition. This is the constraint.

So I give you a bunch of numbers. Do any subset of them add up to t? That's all this is asking.

This problem is NP-hard. It's NP-complete, in fact, when you can guess which integers should go in the subset, and then add them up to see if you got it right.

It is NP-hard, but it's something special we call weakly NP-hard. And why don't I come back to the definition of that in a moment? Let me first show you the proof. It's actually really easy now

that we have this three dimensional matching problem. It's pretty cool.

So these numbers are going to be huge. What we're going to say is, let's view-- so again, we're given a three dimensional matching instance. Get the directions, right? We're given a set of triples. We want to solve this problem by reducing it to a subset sum. So we get to construct integers that represent triples. That's what we're going to do.

So here we go. We get to choose a number. So I'm going to think of them in a particular base, b, which is going to be 1 plus the max of the mxi's. So again, this is the number of occurrences of variable xi in a true or false form. So I take the maximum occurrence of any variable, add 1. That's my base. It just has to be large enough.

And this is basically the entire reduction, is one line. If I have three triples-- if I have a triple xi, xj, xk, I'm going to convert that into a number that looks like this where the one positions are-- I don't really know the order, but they are i, j, and k. Everything else is zero. And this is in base b, not base 2. It's a little weird. All my digits are 0 or 1, but I'm in base b. And three of the digits are 1. And the rest are zero.

Why? Because of my target sum. Target sum is going to be 1111111111. So this number, in algebra, you're write this as b to the i plus b to the j plus b to the k. This you would write as the sum of b to the i for all i.

Do you see why this works? It's actually really simple. For this instance, my goal is to choose a subset of these numbers that add up to this number. How could that possibly happen? Well, I've got to choose-- every time I choose one of the numbers, those three digits get set to 1 in my sum. If I ever have a collision, if I add two 1s together, I'm going to get a 2. That's not good, because once I get a 2, I'll never be able to get back to a 1, because my base is really big.

This base is designed so that the total-- this is the total number of colliding 1s. So we set it one larger than that, which means you'll never get a carry when you're adding up in this base. That's why I set the base to be something large, not base 2. Base 2 might work, but this is much safer.

So what that means is for each of these 1s in the target sum, I've got to find a triple that has those 1s. And those triples can't overlap. So that means choosing a set of numbers that add up to this is exactly the same as choosing a set of triples that covers all of the elements. Done,

super easy once you have the right problem. OK, good.

Now why do I call this weekly NP-hard? Because these numbers are giant. If I have n elements in X, Y, Z over there-- I guess here they're called xi, yk, zk. Sorry, maybe I should've called them that here. Doesn't matter.

If I have n of those elements in X union Y union Z, the number of digits here is n. So the number of digits in order n. This is fine from an NP-completeness standpoint. This is polynomial size. The number of digits in my numbers is a polynomial. And this base is also pretty small. So if you wrote it out in binary, it would also be polynomial. So just lost a log factor.

But the size of the numbers, the actual values of the numbers, is exponential. With weak NP-hardness, that's allowed. With strong NP-hardness, that's forbidden. In strong NP-hardness, you want the values of the numbers to be polynomial. So in this case, the number of bits is small, but the actual values are giant, because you have to exponentiate.

It would be cool. And this problem is only weakly NP-hard. Maybe you actually know a pseudo-polynomial time algorithm for this. It's basically a knapsack. If these numbers have polynomial value, then you can basically, in your subproblems in dynamic programming, you can write down the number t and just solve it for all values of t. And it's easy to solve it in polynomial time, polynomial in the integer values.

So we call that pseudo-polynomial, because it's not really polynomial. It's not polynomial in the number of digits that you have to write down the number. It's Polynomial in the values. Weak NP-hardness goes together with pseudo-polynomial. That's kind of a matching result. Say look, pseudo-polynomial is the best you can do. You can't hope for a polynomial because if you let the numbers get huge, then the problem is NP-complete. But if you force the numbers to be small, this problem is easy. So subset sum is a little funny in that sense. Cool.

Let me tell you about another problem, partition. So partition is pretty much the same set up. I'm given n integers. Let's say they're positive. And I want to know, is there a subset-- I'm not given a target sum t. Target sum is basically forced. What I would like is the sum of all the values in S to equal the sum of all the values not in S. That's A minus S, which in other words is going to be the sum of all values in A divided by 2.

So this is called partition because you're taking a set, you're splitting it into two halves of equal

sum. Every element has to go in one of the two halves. And they're called S and A minus S, like cuts in the flow stuff. And you want those two halves to have exactly the same sum, which means they will be the sum divided by 2. So that better be even, otherwise it's not going to be possible. So again, you want to decide whether this is possible or impossible, yes or no.

I claim this problem is also weakly NP-complete, and we can reduce from subset sum to partition. This is a little interesting because partition is actually a special case of subset sum. It is the case where t equals this. Subset sum, you're trying to solve it no matter what t is. t is a given input. So there's more instances over here. Some of them, some of these instances are the case where t equals the sum over 2. Those are partition instances. So this is like a subset of the possible inputs as over there, which means this problem is easier than this one-- no harder anyway.

In other words, I can reduce partition to subset sum. I just compute this value and set that to t, and then leave the a's alone. That will reduce partition to subset sum.

But that's not the direction I want. I want to reduce from subset sum, a problem I can prove is NP-complete, to partition, because I want to prove that partition is NP-complete.

So in this case, there's an easy reduction in both directions. This direction is a little harder. So reduction from subset sum. So I'm given a bunch of ai's. I'm not going to touch them. And I'm given a target sum t. And I basically want to make that target sum into this half.

To do that, I'm going to add two numbers to my set. So I'm going to let sigma be the sum of the given a's. And then I'm going to add-- so I'm given a1 through an. I'm going to add an plus 1, is going to be sigma plus t. And I'm going to add an plus 2 to be 2 sigma minus t.

Why? So these are two basically huge numbers. Because sigma is bigger than-- I mean, it's the sum of all the numbers, so it's bigger than all of them. And so imagine for a moment that I put these two in the same side of the partition. I put them both in S or I put them both out of S. Their sum by themselves is 3 sigma. The t's cancel. Whereas all the other items, their sum is sigma. So I'm host. If I have 3 sigma on one side and sigma on the other, I'm not going to make them equal.

So in fact, these two elements have to be on opposite sides. So there's a side that has sigma plus t. There's a side has 2 sigma minus t. And then there's all the other n items, and some of them are going to go to this side, some of them are going to go to this side. Their total value is

sigma.

Right now this is close to sigma. This is close to 2 sigma. So they have to kind of meet in the middle. In fact, what you'll have to do is add sigma minus t over here and add t over here.

Think about it for a second. If I add sigma minus t, this comes out to 2 sigma. If I add t to this, this comes out to 2 sigma. That would be good because they're equal. And notice that this is sigma minus t. This is t. Their sum is sigma.

So in fact, it has to be like this. You add something over here, and sigma minus something over here for all the other ai's. And the something has to be t in order for these two values to equalize. So in order to solve this slightly larger partition problem, you have to actually solve the subset sum problem because you have to construct a subset that adds up to t. t was an arbitrary given value. So this is pretty nifty. We're adding some values so that the new target sum is the 50/50 split when we're given some values that have an arbitrary target sum. So partition is weakly NP-complete.

Let me go to rectangle packing. So rectangle packing-- I'm going to draw a picture. I give you a bunch of rectangles of varying sizes. And I give you a target rectangle. Let's call it T. These are the Ri's. I want to put these rectangles into this picture without any overlaps. Each of these rectangles here corresponds to one of the rectangles over here.

So I'll tell you that the sum of the areas of these rectangles is equal to the area of T. And the question is, can you pack those rectangles into T without any overlaps, and therefore without any gaps, because the areas are exactly the same.

I claim this problem is weakly NP-hard-- I guess NP-complete by reduction from partition. This will be super easy if you followed what the definition of partition is. We're given some integers ai. And we're going to take each of them and convert them into a, let's say, 1 by 3ai rectangle. Three is to avoid some rotation we'll see.

And then we're also given the targets. Oh no, target sum is given. Target sum is the sum over 2. But anyway, we're going to build our target rectangle to be-- it's actually going to be really big. It's going to be 2 by 3 times t. So this is that thing. So this is 3/2 sum of the a's. OK, that's about it.

In order to pack these rectangles into here, because each of them is at least three long, you cannot pack them vertically. They have to be horizontal. So in fact what your packing will look

like is they'll be the top half and the bottom half. And the top half, the total length of those rectangles has to add up to 3/2 sum of A. Everything was scaled up by 3, so that's 1/2 of A on the top and the bottom. That's a partition. In order to pack the rectangles into here, you have to solve the partition problem, and vice versa. Easy.

OK, let me show you one more thing, jigsaw puzzles. This is not the jigsaw puzzles you grew up on, somewhat more generalized. So a piece is going to look something like this. I drew them intentionally different. So on each, you have a unit square. Some of the sides can be flat. Some of them can be tabs. Some of them can be pockets. Each tab and pocket has a shape. And they're not in a perfect matching with each other. So there could be seven of these tabs and seven of these pockets, all the same shape. This is what you might call ambiguous jigsaw puzzles. Plus, there is no image on the piece, so this is like hardcore jigsaw puzzles. This is NP-complete.

And what I'd like to do is to simulate a rectangle with a bunch of jigsaw pieces. So it would look something like this. If I have a 1 buy something rectangle, I'm going to simulate it with that same something, little jigsaw pieces. And I'm going to make these shapes only match each other. And so for every rectangle, they're going to have a different shape. This one will be squares. At that point I ran out of shapes I can easily draw, but you get the idea. Each rectangle has a different shape. And so these have to match to each other. You can't mix the tiles, which means you have to build this rectangle. You have to build this rectangle. And then if the jigsaw problem is, can you fit these into a given rectangle, then you get rectangle packing.

But this is not a valid reduction. You can't reduce from partition. Why? Because these numbers are huge. Remember, the values of the numbers in my partition instance are exponential. So if I have a value ai and it's exponential in my problem size, and I tried to make ai have little tiles, that means a number of jigsaw pieces will be exponential in n. That's not good. That's not allowed. This is why weak NP-hardness is annoying.

So instead, we need a strong NP-hard problem. This is a problem that's NP-hard even when the numbers are polynomial in value, not just in size. And it's called 4-partition. 4-partition, you're given n integers, as usual. Say set is A. And you want to split those integers into n over 4 quadruples of the same sum. So this would be the sum of A divided by n over four. That's your target sum.

So before we had to split into two parts that had the same sum. That was partition. Now we have to split into n over 4 parts. Each part will have exactly four numbers, four integers. And they should all have the same sum.

This problem is hard even when the integers have polynomial value. So the values are at most some polynomial in n. I won't prove it here, but it's in my lecture notes if you're curious. It's like this proof, but harder. You end up, instead of having n digit numbers, you have five digit numbers. Each digit only has a polynomial in n different values. So the total value of the numbers is only polynomial. It's like n to the fifth or something.

Good news is that this reduction I just gave you is also a reduction from 4-partition because it's the same set up. Again, I'm given integers. Each integer I'm going to represent by that many tiles. Now the number of tiles is only polynomial, so this is a valid reduction. And again, if I have to pack all of these tiles into a rectangular board, that's exactly the same as packing these integers.

Well, I guess I should do rectangle packing again. So this is a proof rectangle packing was weakly NP-hard. But in fact it's strongly NP-hard. You just change these dimensions. You say well, I need whatever, n over 4 different parts, each of size the sum over n over 4. You need some scale factor here. Three doesn't work. Use n or something-- n and n. That will prove that rectangle packing is actually strongly NP-hard because we're reducing for 4-partition instead of partition. And then you can reduce rectangle packing to jigsaw puzzles because you have strong hardness over here.

Over here we don't have numbers. We just have these pieces. So whenever you convert from a number problem to a non-number problem, if you're representing the numbers in unary, which is what's going on here, you need strong NP-hardness for it to work. Weak NP-hardness isn't enough. Then we get jigsaw puzzles, which we know and love, are NP-complete. That's it.