[SQUEAKING][RUSTLING][CLICKING]

ERIK DEMAINE: Today we're going to, in one lecture, cover an entire field, which is computational complexity. It's sort of-- it meets algorithms in an interesting way, which is, algorithms is mostly about showing how to solve problems well and showing that you can solve a problem well. And computational complexity is more about the lower bound side, proving that you can't prove-- you can't solve a problem very well, you can't find a good algorithm to solve it.

We've seen a little bit about lower bounds several lectures ago, proving search and sorting lower bounds in a bounded branching decision tree model. But these are much stronger notions of badness. This is not about n versus n log n or constant versus log n.

This is about polynomial versus exponential, which has been the sort of bread-and-butter model in this class. Polynomial is a good running time, and we're always striving for that. Exponential is usually pretty trivial to get. And so we're going to talk about some different-- they're called complexity classes that talk about this issue and different ways to prove hardness.

This is a pretty high-level lecture, so you're not going to be expected to be able to prove hardness. But you'll get a flavor of what it's like, and this will segue nicely into other follow-on classes, which is-- we're at pretty much the end of 006, so natural to talk about what other things you might study.

One result we'll prove today is that most problems actually have no algorithm, which is kind of shocking, and lots of other fun things. So let's get started with the notion of P. This is the set of all problems solvable in polynomial time. We talked about what polynomial time means a bunch last lecture. So just recall that polynomial time means polynomial in the problem size, which I'll denote as n here, the number of words in your input.

OK, so these are the problems that are efficiently solvable. P is the set of all of them. And for contrast, EXP is the set of all problems solvable in exponential time. It's the problems solvable in exponential time. Exponential here means something like 2 to the n to the constant.

That's one reasonable definition of exponential, so just the exponentiation of this-- of polynomial. So as you might expect, most-- every problem that we've talked about in this class so far can be solved in exponential time rather easily. And algorithms, in some sense, is about distinguishing these two, which problems are in P versus are in say EXP minus P.

So to formalize this a little bit, I'm going to draw a picture, which is a bit of a simplification of reality, but for the purposes of this class will suffice, and I think is a really helpful way to think about things, which is to have a big axis for-- a single axis for, how hard is your problem, what is the difficulty of solving your problem? And I want to be sure to leave-- so the easiest problems are over here. And each problem is a dot on this axis. Hardest problems are way down the line.

And I want to make sure to leave enough space for all the things that I care about. So P, I'm just going to call this segment up front. And then I'm going to have a bigger thing for exponential time. So this is just to say that P is nested inside EXP.

Every problem that can be solved in polynomial time can also be solved in exponential time because polynomial is less than or equal to exponential. These are just upper bounds. Being an EXP means you're somewhere from this line to the left. Being in P means you're somewhere from this line to the left, in terms of difficulty.

But formally, we would write P is contained in EXP as sets. In fact, they're also known to be different from each other. There are problems that can be solved in exponential time that cannot be solved in polynomial time. For example-- I'll put that here, sure.

For example, n by n chess is in exponential time, but not polynomial time. So what is the n by chess problem? This is, I give you an n by n chessboard, and I describe to you a position. Here's where all the white pieces are. Here's where all the black pieces are. You can have an arbitrary number of queens and bishops and pawns of each color, of course, up to n squared of them so they don't overlap each other.

And I want to know, does white win from this position? Let's say it's white to move. Can white win? And that problem can be solved in an exponential time by exploring the entire tree of all possible games. But it cannot-- but you can prove that it cannot be solved in polynomial time. So that's a nice example.

A more positive example, so to speak, is negative weight cycle detection. I guess it's literally negative, but it's morally positive. Negative weight cycle detection is the following problem. I give you a graph, a directed graph with weights, and I want to know, does it have a negative weight cycle, yes or no? And this problem is in?

AUDIENCE: P.

ERIK DEMAINE: P, because we saw a polynomial time algorithm for this. You run Bellman-Ford on an augmented graph. So this is an example of a problem we know how to solve. This whole class is full of examples that we know how to solve in polynomial time. But this is a nice, non-trivial and succinct one to phrase.

It's also an example of a decision problem. A lot of-- basically all the problems I'll talk about today are decision problems, like we talked about last class, meaning, the answer is just yes or no. Can white win from this position, yes or no? Is there a negative weight cycle, yes or no?

Tetris we can also formulate as a problem. This is a version of Tetris that we might call perfect information Tetris. Suppose I give you a Tetris board. It has some garbage left over from your past playing, or maybe it started that way. And I give you the sequence of n pieces that are going to come.

And I want to know, can I survive this sequence of n pieces? Can you place each of these pieces as they fall such that you never overflow the top of the board on an n by n board?

This problem can be solved in exponential time. But we don't know whether it can be solved in polynomial time. We will talk about that more in a moment. It's a problem that very likely is not in P, but we can't actually prove it yet.

All right, so there's one other class I want to define at this point. And we'll get to a fourth one also. But R is the class of all problems that can be solved in finite time. R stands for finite. R stands for recursive, actually. This is a notion by Church way back in the foundations of computing.

As we know, we write recursive algorithms to solve problems. In the beginning, that was the only way to do it. Now we have other ways with loops. But they're all effectively recursion in the end. So R is all the problems that can be solved in finite time on any computer. So very general, this should include everything we care about. And it's bigger than EXP, but includes problems that take doubly exponential time or whatever.

So I will draw a region for R. So everything-- it includes P. It includes EXP. And so we also have containment but not equal R. There's, of course, many classes in between. You could talk about problems that take double A exponential time, and that would have a thing in between here. Or there's also-- between P and EXP there's a lot of different things. We will talk about one of them.

But before we get to the finer side of things, let me talk in particular about R. So we have a nice example, we being computational complexity theory-- or I guess this is usually just called theoretical computer science-- has a problem. And if you're interested in this, you can take 6041, I think.

That doesn't sound right. That's a probability. It'll come to me. We have an explicit problem that is not in R. So this class has been all about problems that are in P. You have the number?

AUDIENCE: 6045.

ERIK DEMAINE: 6045, thank you. It's so close to this class. Or it's so close to 6046, which is the natural successor to this class. So in 6045 we talk about this.

So this class is all about problems that are in P, which is very easy. But in fact, there are problems way out here beyond R. And here is one such problem, which we won't prove here today. It takes a whole lecture to prove this.

Given a computer program, does it ever halt? Does it ever terminate? This would be a great thing if we knew how to solve. It's basically an infinite loop detector. If your problem doesn't halt, then it has an infinite loop of some sort. And you'd like to tell your user, hey, you have a bug in your program.

So this is one part of bug detection. And it's impossible. There is no algorithm that always-- that solves all inputs to this problem. Maybe given one program that, say, has 0 lines of code, it could solve that. It says, yeah, that one terminates.

And maybe you can detect simple kinds of infinite loops. So there's some inputs, some computer programs that you could detect. But there's no one algorithm that solves all inputs. This is kind of sad news. We call such problems uncomputable. This is just another word for being not in R.

OK, and next thing I'd like to do is prove to you that most decision problems are uncomputable, or sketcher proof. So remember, decision problems are problems where the answer is just yes or no. This is a very special kind of problem. And even those, almost all of them, cannot be solved. So halting is an example of a problem we want-- we can't solve.

This whole class, this 006, is about problems we can solve. But today I'm going to show you that, actually, those are in the minority. Most problems cannot be computed. This is very strange and also a little depressing. So we'll talk more about that in a moment. First let me argue why this is the case.

So I'm going to be a little informal about what exactly is a computer program and what exactly is a decision problem. But roughly, all I need to do, the only level of precision I need is just to count how many are there. What is a computer program? Well, it's usually a file. What's a file? It's like a string of characters.

What's a character? It's a string of bits. So a program is just, in the end, a string of bits, finite string of bits. We all understand that. Whatever language you define, in the end, every program is just a string of bits. And a string of bits we can translate into a number. So we can convert between strings of bits and numbers.

When I say number, I mean what's usually called a natural number or a non-negative integer. This is usually represented by bold board bold-- blackboard bold capital N. So this is just 0, 1, 2, and so on.

Now, what about decision problems? Decision problem is a specification of what we want to solve. So we can think of it as saying, for every input, is the answer yes or no? That's literally what a decision problem is. The only question is, what is an input?

And we've talked about inputs and the size of inputs. And there's lots of different ways to measure them. But in the end, we can think of an input as a string of bits also. It's just a file. So a decision problem is a function from inputs to yes or no. And inputs we're going to say, well, that's a string of bits, which we can associate with a number in N.

So here we can start to tie things together. So in other words, a program is a finite string of bits, and a problem is, in some sense, an infinite string of bits because there are infinitely many possible inputs. And for each of them, we specify yes or no. So this is basically an infinite string of bits.

So we can imagine 011010001110, infinitely. Just some-- for every string of bits, we can say, OK, if your input is the number 0, here's the answer-- no. If your input is the number 1, then the answer is yes. If your input is the number 2, your answer is yes, and so on down this line. Every infinite string of bits corresponds to exactly one decision problem, which specifies for every possible input integer, which corresponds to a string of bits, what is the answer, yes or no?

So this may seem subtle, or it may seem like not a big deal. This is a finite string of bits. This is an infinite string of bits. But mathematics has well studied this problem. And infinite strings of bits, there are very many of them, infinitely many. It's not surprising. There are also infinitely many integers.

So maybe it doesn't seem that deep. But there's a difference in infinitude. Programs and integers are countably infinite. And infinite strings of bits are what's called uncountable. I think the most intuitive way to see this is, an infinite string of bits, if I put a decimal or a binary point in front, this encodes a real number between 0 and 1. So this is roughly a real number in 01.

And when I'm writing approximately equal here, this really goes in both directions. Given a decision problem, I can define a string of bits, of course giving me the answer for all inputs. And I can convert that into a real number between 0 and 1. But also the other direction, if I take any real number, that is a corresponding decision problem. These are 1 to 1 bijection between them. And the bad news is, real numbers are uncountable, and natural numbers are countable, which means there's a lot more of these than there are these.

So one way you might phrase this is, informally, the number of natural numbers is way smaller than the number of real numbers. And so from that, we derive that most problems are unsolvable, because every program solves exactly one decision problem. We can also run a program, conceptually, on all possible inputs, and we will figure out what function at solving.

And if we don't allow random numbers in our program, which I'm not here, then every program solves exactly one decision problem. Possibly, it's even worse for us because multiple programs probably solve the same decision problem. They're just-- they add irrelevant lines of code or they don't do anything different. Or you run Bellman-Ford versus running Bellman-Ford five times, you'll get the same result.

And that's actually the bad direction for us. We'd like to know whether there is a program that solves every decision problem. And because there are only this many programs and this many decision problems, it just-- there aren't enough to go around. So most-- what's the phrasing? Not nearly enough programs for all problems, and so there's no assignment of programs to problems because there's just too many problems. More money, more problems, I guess.

So when I first saw this result, I was shocked and dismayed that-- why are we even doing computer science if most problems can't be solved? Luckily, it seems like most of the problems we care about can be solved. That's what this class is all about.

And in fact, even the problems that seem really, really hard for us to solve, like n by n chess, where we can prove it takes exponential time, there is an algorithm to solve chess. It's just really slow. And this is a statement about, most problems can even be solved in finite time no matter how much time you give them.

So it's not all bad. Luckily, most problems we care about are in R. I don't know why. This is sort of a mystery of life. But it's good news. Or it's why we keep persevering trying to solve problems with algorithms.

AUDIENCE: Is it because when we state problems, the statement tends to be small?

ERIK DEMAINE: Well, this-- so the question was, maybe it's just because these short statement problems are easy. But this is a pretty short statement, and it's hard. I think-- I don't have a great reason why. I wish I understood.

There's a general result that if you have any question about the program itself, then there's no algorithm to solve it. Basically, any non-trivial question about programs is hard, is not in R. And I guess if you took-- if you imagine taking a random statement of a problem, then maybe this will be in the middle of it with some probability. Maybe that's why most. But this is a very strong notion of most. There are so many more real numbers than natural numbers that-- I don't know.

I want to add one more class to this picture, which is NP. It nestles in between P and EXP. So we know that P is contained in or equal to NP. And NP is contained in or equal to EXP. We don't know whether there's a quality here or here. Probably not, but we can't prove it.

But what is this class? A couple of different ways to define it-- you might find one way or the other more intuitive. They're equivalent. So as long as you understand at least one of them, it's good. NP is just a class of decision problems. So I define P and EXP and R arbitrary. They can be problems with any kind of output. But NP only makes sense for decision problems.

And it's going to look almost like the definition of P-- problem solvable in polynomial time. We've just restricted to decision problems. But we're going to allow a strange kind of computer or algorithm, which I like to call a lucky algorithm. And this is going to relate to the notion of guessing that we talked about for the last four lectures in dynamic programming.

With dynamic programming, we said, oh, there are all these different choices I could make. What's the right choice? I don't know, so I'd like to make a guess. And what that meant in terms of a real algorithm is, we tried all of the possibilities, and then took the max or the OR or whatever over all those possibilities.

And so we were-- but what we were simulating is something that I call a lucky algorithm, which can make guesses and always makes the right guess. This is a computer that is impossible to buy. It would be great if you could buy a computer that's lucky. But we don't know how to build such a computer.

So what does this mean? So informally, it means your algorithm can make lucky guesses, and it always makes the right guess. And whereas in DP, we had to try all the options and spend time for all of them, the lucky algorithm only has to spend time on the lucky guess, on the correct guess.

More formally, this is called a non-deterministic model of computation. And this N is the-- the N in non-determinism is the N for NP. So this is non-deterministic polynomial time. So algorithm can make guesses. And then in the end, it should output yes or no.

Like say if you're exploring a maze, this algorithm could say, should I go left or go right? I'm going to guess whether to go left or go right. And let's say it guesses left. And so then it just goes left. And then it reaches another junction. It says, should I go left or right?

And it'll say, I'll guess, and it'll say, guess right this time. And in the end, if I get to some dead end maybe and I say no, or if I get to the destination I'm trying to get to, I say yes. So that's a non-deterministic algorithm.

And what does it mean to run that algorithm? What does it mean for the guesses to be lucky? Here's what it means. These guesses are guaranteed-- which way you end up going is guaranteed to lead you to a yes if there is one-- if possible.

So in my maze analogy, if my destination is reachable from my source, then I'm guaranteed, whenever I guessed left or right, I will choose a path that leads me to my destination. Whereas, if the destination is in some disconnected part of the maze and I can't get there, then I don't know what the guesses do. It doesn't really matter. Because no matter what I do, I'll end up in a dead end and say no.

That's the model. As long as you have an algorithm that always outputs yes or no in polynomial time-- because we're only talking about polynomial time, lucky algorithms-- if there's any way to get to a yes, then your machine will magically find it without having to spend any time to make these decisions. So it's a pretty magical computer, and it's not a computer that exists in real life. But it's a computer that's great to program on. It's very powerful. You could solve lots of things with it. Yeah.

AUDIENCE: If you had this magical computer, it can guess whether it's yes or no, why doesn't it just answer the question?

ERIK DEMAINE: Right. So what if we-- so a nice check is, does this make all problems trivial, all decision problems? Maybe I should say, well, I don't know whether the answer to the problem is yes or no, so I'll just guess yes or no. This is problematic because-- so I might say, it will guess A or B, and if I choose the A option, I will output yes, and if I choose the B option, I will output no. In this model, that algorithm will always output yes.

Because what it's saying is, if there's any way to get to a yes answer, I will do that way. And so such an algorithm that tries to cheat and just guess the whole answer to the problem will actually end up always saying yes, which means it doesn't solve a very interesting problem. It only solves the problem, which is represented by the bit vector 1111111, where all the answers are yes. But good check. Yeah.

AUDIENCE: Does there have to be a bound of a number of things it has to choose between when it [AUDIO OUT]

ERIK DEMAINE: Yes.

AUDIENCE: Does it have an exponential number of them?

ERIK DEMAINE: Exponential number of choices is OK. I usually like to think of it, as you can only guess one bit at a time. But we're allowed polynomial time, so you're actually allowed to guess polynomial number of bits. At that point, you can guess over an exponential size space, but not more than exponential.

So it's-- yeah, polynomial time let's say in the one-bit guessing model. What did I say? Makes guesses-- let's add binary here. Otherwise we get some other class, which I don't want.

OK, let's do an example, a real example of such an algorithm that's useful, which is Tetris. So I claim Tetris is in NP because there is a lucky algorithm and non-deterministic polynomial time algorithm that can solve the Tetris game. So again, you're given a board, you're given some sequence of pieces, and you want to know whether there's some way to place the pieces that lets you survive.

And so what I'm going to do is, for each piece, I'm going to guess how to place it. So for the first piece, I'm going to guess how far left or right do I move it. Then I let it fall one step. Maybe I rotate it. I choose a sequence of moves among left, right, down, rotate right, rotate left.

And all along the way, I check, is that move valid? If the move is invalid at any point, I just say, return no. And then if the piece gets nestled into a good spot, I continue to the next piece. I do the same thing, guess all the possible things I could do to that.

Again, I only need to guess one bit at a time. And I'll only need to do a polynomial number of guesses, like a linear number of guesses, for each piece about where it falls in, so maybe a quadratic number of guesses overall. And then at the end, if I survived-- oh, I also have to check if a line clears. Then I clear the line.

And if in the end I survive, I return yes. So this is a non-deterministic algorithm. So I would say, check the rules of the game. And if we survive, return yes. And if at any point we violate the rules-- for example, we go off the top of the board-- we return no.

So this is an algorithm that sometimes returns no and sometimes returns yes depending on what choices you make. And this model guarantees, if there's any way to get to a yes, it will find it. If I swapped these answers, if I returned yes when I violated the rules and returned no if I survived, this would be an uninteresting algorithm.

Because it's very easy to lose in Tetris. The hard part is to survive. If I say, is there any way to play the game in such a way that I violate the rules, then, of course, the answer is yes. You can just stack pieces and go off the top.

There's an asymmetry in this definition of yes versus no. It always finds yes answers if possible. It doesn't always find no answers if possible. So it's very important the way that I wrote these questions. It's important that I define Tetris as the problem of, can I survive? The problem of can I not survive, is it impossible to survive, that's a different question. That problem is not in NP, probably. OK, so slight subtlety there, yes versus no.

Let me give you the other definition of NP. So if this one's confusing, which-- although I prefer this definition. Most people do not. So this is confusing. Let's do the other definition. So another definition is that NP is a set of decision problems that can be checked in polynomial time.

This actually came up in the last lecture where we talked about subset sum. I said, here's a bunch of integers, here's a target integer, and I can prove to you that this integer can be represented as a sum of numbers from my subset of numbers from my set, because here they are. I gave you this plus this plus this equals the target sum. And so that is a solution, in some sense, that can be checked for a yes example.

If I can represent my number as a subset sum of a given set, it's easy for me to prove that to you. And you can check it just by adding up the numbers and checking that each number was in the set. Whereas no instances, I had an example of a target sum that could not be reached. And the only reason I knew that is because I had brute-forced the thing. And there's no succinct way to prove to you that that number can't be represented.

A similar thing with Tetris, what I would say is-- so this is version 1, version 2-- for Tetris is that, a certificate for a yes input of Tetris is a sequence of moves for the pieces. OK, if it's possible to survive in Tetris, I can prove it to you. I can just play the game and show you that I survived.

No answers, I don't know, it's hard to prove to you that I can't survive a given sequence of pieces. But yes answers are easy. I just show you, here's the sequence of button presses I'll do for this piece, then for this piece, then for this piece. Notice it's exactly the same thing that I guessed in the beginning of this algorithm. And then I did some other work to implement the rules.

And similarly, if I gave you a certificate, which is the things that I wanted to guess of how to play the game, I can check this certificate by just implementing the rules of Tetris and seeing whether I survived. And if you violate the rules at any point, you say no. And if you survive, you return yes. That's what's called a verification algorithm.

So let me formalize this notion. Given a problem input plus a certificate, like that one over there, there is a polynomial time-- so this is yet another definition. This is what I mean by this definition of NP-- verification algorithm that satisfies two properties. One is, for every yes input-- so every input where the answer is yes to the problem-- there exists a certificate such that the verifier says yes.

So this is saying, it's possible to prove to me that an answer is yes, because if you ever have an input that the answer happens to be yes, you can prove it to me by giving me a certificate. There's always some certificate that proves the answer's yes. Because the verifier, which runs in regular polynomial time-- this is a regular, old-fashioned, down-to-earth verification algorithm, polynomial time in our usual sense-- it will say yes. And furthermore, the yes answers from the verifier are actually meaningful, because if I ever give it a no input, it always says no, no matter what certificate I give it.

So this should really formalize what all this means. It's equivalent to the previous definition. This is saying that proofs exist for yes instances. And this is saying that proofs don't exist for no instances, meaning there are no false proofs. So if the verifier ever outputs yes, you know that the answer to your problem is yes.

But if it outputs no, you're not sure. Maybe you got the certificate wrong because we only know there's some certificate where the verifier will say yes. Or maybe it was a no input, and then it didn't matter what certificate you used.

But it's nice, because it says on, say, Tetris, if I give you the sequence of pieces, it's very easy to write down a verifier which just implements the rules of Tetris. And so then you can at least check whether a solution is valid in the yes case. In the no case, we don't have anything useful.

So NP is a structure, some additional structure about the yes inputs in your problem. And a lot of decision problems are in NP. A lot of the problems that we care about can be phrased as an NP problem. As long as it's a decision problem, usually, answering yes or no is provable, like subset sum, like Tetris. These are all problems where, if the answer is yes, I can give you a convincing proof why.

And it turns out a lot-- so a lot of problems fall into this NP setting. And so we have some tools for talking about problems being hard with respect to NP. Let me first talk a little bit about P. Does not equal NP, question mark. A lot of people conjecture that P does not equal NP. It's sort of a standard conjecture in theoretical computer science.

But we don't know how to prove whether P equals NP or does not equal NP. And so in this picture, I've drawn the hypothesis, which is that NP is a strictly bigger region than P is. But we don't actually know whether there are problems in this region. We don't know whether there are problems in this region between NP and EXP.

We conjecture there are problems here and there are problems here. There's definitely problems here or problems here, but we don't know which one. Because we know P does not equal EXP, but we don't know whether P equals NP, and we don't know whether P equals EXP. If you could prove that P does not equal NP, or disprove it, you would win $1 million, which not that much money these days. But you would be famous to for the rest of time if you could ever prove this.

Every year, there's usually a crackpot proof that doesn't work out. Some of them go to me. Please don't send them. And anyway, it's a very hard problem. It is sort of the core problem in theoretical computer science, how to prove P does not equal NP. But for the most part, we just assume it.

Now, what does this conjecture mean? It essentially means-- the way I like to say it is, you cannot engineer luck. Because NP problems are problems you can solve by lucky algorithms. P are problems you can solve by regular old algorithms. And so if P equalled NP, it means luck doesn't buy you anything, which seems weird. If I can magically make these super powerful guesses, then I can solve the problem that that's NP, that seems super powerful, way more powerful than regular algorithms, where we have to actually brute-force and try all the choices.

And so it seems pretty solid that P does not equal NP. That's my-- of course, we don't know how to prove it. Another phrasing is that it's harder to come up with proofs than it is to check them, from a mathematical perspective. This is equivalent to P does not equal NP. So that's why you should believe it.

Now, let's go over here. The next notion is NP-hardness. So in particular, I want to claim-- this is a theorem that exists in the literature-- that if P does not equal NP, then Tetris is not NP. So I said right here, Tetris is in EXP, but we don't know whether it's in NP.

But in fact, we conjecture it is not NP because we conjecture that P does not equal NP. If you could prove this conjecture-- and there's a lot of theorems that are conditioned assuming P does not equal NP-- then we get some nice results, like Tetris cannot be solved in polynomial time. It cannot figure out whether I can win a Tetris game in polynomial time in the input size.

Why? This is a consequence of another theorem, which is that Tetris is NP-hard. I'm going to define NP-hard informally first, and then I'll define it slightly more formally in a second. But this means, roughly, that Tetris is as hard as all problems in NP.

So let me draw this in the picture. So NP-hard is this part. Did I leave myself enough room? Maybe not. Well, we'll squeeze it in. There's another region here for EXP-hard. So your problem being in NP was a positive result. It says you're no more difficult than this line. You're either at this position or to the left.

Being in P was also a positive statement. It says you're here or to the left. Being in P is better than being in NP because this is a subset of that. NP-hard is a lower bound. It says, you are at this point, at this level of difficulty, or to the right. And so it goes from here off to infinity in difficulty. And EXP-hard says you're at least as hard as the right extent of the EXP set, or you're harder than that, in a sense that we will formalize in a moment.

And this place right here, as you might imagine, is kind of interesting. It's exactly where NP meets NP-hard. This thing is called NP-complete. You probably have heard about NP-completeness, a famous notion. And this is what it means. It is, the problems that are in NP-- so they have a lucky algorithm that solves them, they can be verified, there are certificates that can be verified-- and they are NP-hard.

So they're in NP, and they are the hardest among problems in NP. Now, they're not the hardest problem. There are actually many problems right here at this single level of difficulty called NP-complete. Among them is Tetris. There are many others, which I will list in a moment. So that is NP-completeness.

So because these problems are the hardest problems in NP, if there's any problems here in between-- in NP minus P, then these must be among them. And so if you assume that P does not equal NP, as most people do, then you know that all problems at this right-most extreme of NP, the hardest of the problems in NP, they must not be NP. And that's why I can say, if P does not equal NP, Tetris is not NP, and also, any NP complete problem is not NP.

OK, what does "as hard as" mean? This is our good friend reductions. We talked about reductions a lot in this class. Reductions are the easy way to use algorithms. You just take your problem and reduce it to a problem you already know how to solve. You take the input to some problem that you want to solve, and you convert it into an input to some other problem, like single source shortest paths or something like that that you already have an algorithm for solving.

So if you have an algorithm that solves problem B, you can convert that into a solution for B. And a reduction should also tell me how to-- given a solution to B, how to convert it back into a solution for A. And when I say solution here, I actually mean certificate from over there. So how-- so if I-- so if I have-- so reduction consists of these two pieces-- how to convert an input at A to an input for B, and given a solution to B, how to convert it to a solution to A.

Let me give you some examples of reductions you've already seen. You've seen a lot of them. If I have unweighted shortest paths on the left-- unweighted single source shortest paths-- I can reduce that to weighted shortest paths. How?

AUDIENCE: Set all the weights to 1.

ERIK DEMAINE: Set all the weights to 1. So here I'm given a graph without weights. If I set all the weights to 1, that turns it into an input for a weighted single source shortest paths. So if you didn't know how to solve this, you could solve it by converting it. If you've already written, say, a Dijkstra algorithm, you could apply it to solve unweighted single source shortest paths.

Now, we know a faster way to solve this, but it's only a log factor faster. And here we're talking about polynomial versus exponential. So this is a valid reduction. It's not the most interesting one from an algorithmic standpoint, but it is an algorithm.

Another one we've seen is, if you have integer weights in the left, you can convert that to unweighted on the right, positive integer weights, by subdividing each edge of weight W into W edges of no weight. So that's maybe a little bit less efficient. It depends what the sum of the weights are.

Another version that we've seen is longest path in a graph. We can-- weighted path we can reduce to shortest path in a graph, weighted by negating all the weights. We did this in some of the dynamic programming things. Like oh, longest path on a DAG? We can convert that into shortest path on a DAG just by negating all the weights.

So these are all examples of converting one problem to another. Usually, you convert from-- for algorithms, you convert from a problem you want to solve into a problem that you already know how to solve. But it turns out the same tool reductions can be used to prove negative results too. And in this case, we're going to reduce from a problem that we think cannot be solved and reduce it to the problem that we're interested in solving.

So let me write more precisely what this means. If you can find a reduction like this, it means that solving A is at least as easy as solving B. Because I could solve A, in particular, by converting it into B, solving B, and then converting it back to a solution to A. So in other words, if I can solve B, I can solve A, which I can phrase informally as, A is at least as easy as B.

And now using grammar, contrapositive whatever, this is the same thing as saying that B is at least as hard as A. And this is what I mean by at least as hard as. So this is my definition of at least as hard, in this notion of NP-hardness. So what NP-hard means is that I'm at least as hard as all problems in NP. So what that means is, every problem in NP can be reduced to Tetris, which is kind of funny.

But in particular, that means that if there's an algorithm for Tetris, there's an algorithm for all problems in NP. And so that's actually the contrapositive of this statement. So this is saying, if there's a polynomial-- if I take the contrapositive of this, this is saying, if there's a polynomial time algorithm for Tetris, then P equals NP, there's a polynomial time algorithm for every problem in NP. And the way we prove that is by reductions.

We take an arbitrary problem in NP, and we reduce it to Tetris. Luckily, that's not as hard as it sounds because it's already been done once. There is already a reduction from NP to-- from all problems in NP to singular problems out there, the NP-complete problems. There is some first NP-complete problem, which I guess is the Turing machine. It's basically simulating a lucky algorithm, so it's kind of a not very interesting problem.

But from that problem, if you can reduce it to any other problem, you know that problem is NP-hard as well. And so briefly, I want to show you some examples of that here. So I want to start out with a problem that I'm just going to assume is NP-complete. And it's called 3-partition.

One way to phrase it is, I give you a bunch of integers-- I think I have it written down over here, also the board. I give you n integers, and I'd like to divide them up into n over 3 groups of size 3, such that each group of size 3 has the same sum. And it's written there on the board. So you can also think of this as the following problem. I give you a bunch of rectangles that are a side length-- or a bunch of sticks, let's say, of varying lengths, and I want to group them up like on the right diagram, so in groups of 3, such that the total length of each group is exactly the same.

This is just a problem. And just believe for now that it is NP-complete. I won't prove that. But what I'd like to show you is a reduction from this problem to another problem-- solving jigsaw puzzles. So you might think jigsaw puzzles are really easy, and especially easy if I lose the projector. But in fact, if you have a jigsaw puzzle where some of the matches are ambiguous, if there's multiple pieces that could fit against a given tab or pocket, then I claim I can represent this 3-partition problem by building little sticks, like here.

So if I want to represent a stick of length ai, I'm just going to build an ai-- I didn't mention they're all integers, and they're polynomial-sized integers. I'm going to represent that by ai different pieces here. And the red tabs and pockets are designed to be unique global to the puzzle, like a regular jigsaw puzzle. Given this piece on the left and this tab on the right, there's a unique pocket, there's a piece with unique pocket that fits perfectly into that piece.

So this joining is forced. And also this joining is forced. But the blue tabs and pockets are different. They're all the same. They're all identical. And so if I build this frame using the red unique assignments, and I build these rectangles, if I want to pack these rectangles into this rectangle, that's exactly the 3-partition problem, with some details that I didn't fill in. But it turns out you'd be forced to group these into groups of size 3, something like this, with varying lengths.

OK, so that's an example of a reduction. If you believe the 3-partition is NP-hard, this proves to you that jigsaw puzzles are NP-hard, something you may not have known. Every time you solve a jigsaw puzzle, you can feel good about yourself now, especially if it has ambiguous mates.

Next is Tetris. So here is a reduction from the same 3-partition problem, which is one of my favorite problems, to Tetris. It starts out with this strange board. It has a bunch of columns here where I could put pieces. So I'm not allowed to put pieces in these dark regions.

They all have height T. T is the target sum that we want all of the numbers to-- all of the triples of numbers to add up to. And there's n over 3 of these slots where I can try to put pieces. And it's-- because of this thing over on the right, there's no way to clear lines in this game.

And now to represent a single number ai, I'm going to give you this sequence of pieces, which starts with an L piece. And then it has ai repetitions of this pattern, and then it ends with these two pieces. And so what ends up happening is that-- this is in the intended solution-- you first place an L at the bottom of one of these buckets, and then you repeat this pattern in this nice way. And it fills up the ai, roughly, height of this bucket. And then at the end, you have to put the I here.

And what this ends up guaranteeing is that all of these pieces go into a single bucket. You can check. It's tedious. But if you tried to put some of these pieces in one bucket and other pieces in a different bucket, you would lose some space, and then you would die. So if you want to survive, you have to put all these pieces into one bucket.

And so again, we're just stacking rectangles. We're putting a whole bunch of rectangles in one pocket and then a bunch of rectangles another pocket. We can switch back and forth however we want. But the only way to win, it turns out, is if you get all of those rectangles to add up to exactly the right height. Then you get a picture like this. If you don't get a picture like this, you can prove you end up dying.

Then I'll give you a bunch of Ls. Then I'll finally give you this T, which clears some lines. And then I'll give you-- the most satisfying Tetris game ever-- I'll give you a ton of I's, and you get Tetris, Tetris, Tetris, and you clear the entire board. And so if you can solve the 3-partition problem you can clear the board and win the game and be the best Tetris player ever. And if there is no solution to 3-partition, you're guaranteed to lose. And so this proves Tetris is NP-hard. Cool.

So what else do I want to say, briefly? I think that's the main idea. So another example-- so this spot is called EXP-completeness. And this includes problems such as n by n chess. So we know that chess requires exponential time because, in fact, it's among the hardest problems in exponential time. But most common are the-- that's somehow because of the two-player nature of the game. Most common are NP-complete problems.

And we have a bunch of example NP-complete problems I'll just briefly mention here. So we saw the subset sum problem, which we had a polynomial time algorithm for-- sorry, a pseudo polynomial time algorithm for last class-- in fact has no polynomial time algorithm, assuming P equals NP. So pseudo poly is the best you can hope for for subset sum.

There's a related notion called weakly NP-hardness, which I won't get into here. 3-partition is one we saw. We saw some reductions to other problems. So these are all NP complete. Longest common subsequence is another dynamic programming problem we saw with two sequences. I mentioned you could solve it for three or four or any constant number. But if I give you n sequences each of length n, that problem is NP-hard, NP-complete.

Longest simple path in a graph-- we know how to solve longest path. You just solve shortest path and negative weights. But longest simple path, where you don't repeat vertices, that's NP-complete. Relatedly, one of the most famous NP-complete problems is traveling salesman problem, finding the shortest path that visits all vertices in a given graph. So instead of just going from A to B, I want to visit all the vertices in the graph.

A lot of these problems I'm phrasing as optimization problems. But when I say NP-complete, I actually mean a decision version of the problem. For example, with this one, the decision question is, is the shortest path that visits all vertices in a graph less than or equal to a given value x. If you can solve this, then by binary search you can solve the overall weight.

3-coloring a graph is hard, even though 2-coloring a graph is polynomial. 3-coloring is NP-complete. Assigning three colors to the vertices so that no adjacent vertices have the same color, finding the largest clique in a given graph, which would be useful for analyzing social networks, whatever.

This is a fun one for me as a geometer. If you're in a three-dimensional world, which I am, and I want to find the shortest path from here to there that doesn't collide with any obstacles, like this desk and all the chairs and so on, in 3D, this problem-- if you can fly, so if you're a drone flying among all these obstacles, you want to find the shortest path from A to B, this is NP-complete.

It's quite surprising. In two dimensions, it's polynomial. You can reduce it to graph shortest paths. But in 3D, it's NP-hard. This is a formula problem that comes up a lot. Given a Boolean formula with AND, OR, or NOT, can you ever make it true, if it has some variables that are not assigned?

And some more fun examples are Minesweeper or Sudoku. Basically any paper and pencil puzzle you've ever played, there's probably a paper out there proving that it's NP-complete. And on the video game side, *Super Mario Brothers* is NP-hard. *Legend of Zelda* is NP-hard. *Pokemon* is NP-hard. These problems are actually all a little bit harder than NP, in a different class called P-space, which I won't go into.

But if you're interested in this stuff, there is a whole class devoted to it, which has online video lectures, so you can watch them whenever you want, called 6.892, that gives a bunch of especially fun examples of NP-hardness and other types of hardness proofs from a sort of algorithm perspective for lots of games and puzzles you might care about. And that's it.