PROFESSOR: Computational complexity is basically about how hard is a problem, right? There are some problems that are really easy and some problems that are really hard. And Eric drew this line on the board where you have really easy on the left and really hard on the right. What's the hardest problem that we can possibly have?

AUDIENCE: It was chess, right? Or something like that?

PROFESSOR: No, there's something beyond chess.

AUDIENCE: A halting problem?

PROFESSOR: Halting problem.

AUDIENCE: Yeah.

PROFESSOR: Yep. Halting problem is somewhere here. So what's the best solution for the halting problem? Running time.

AUDIENCE: It is exponential, or something?

PROFESSOR: You'd wish.

AUDIENCE: Oh really? It's not even exponential?

AUDIENCE: I think there's like no solution.

PROFESSOR: There is no solution for the halting problem. No matter how much time, computers, and money you have, you cannot solve the halting problem. It's undesignable. So this is the worst kind of problem you can come up with.

AUDIENCE: What exactly is the halting problem?

PROFESSOR: So halting problem is given a program in any representation that makes sense to you, machine language, CE, assembly, parsing tree, whatever you want, decide if it terminates or not. Sounds really simple, right? Turns out there's a proof that says it's impossible to solve. So you can't say will terminate or will not terminate.

AUDIENCE: Well, but for some programs, it's like while true, but there's no break.

PROFESSOR: So Turing Machine. Yeah. If you have a machine where you're not allowed loops, then it's easy to know what's going to happen. So you need to have a Turing machine, something that allows loops. So some instances are easy, right? But for the general case of you have to write a program that gets another program and outputs this bit, we don't know how to do that. But we've proven that nobody will know how to do that, so we're OK with it. So this means undecidable. We know that nobody will ever be able to solve this. So if you're given the halting problem on an interview, it's safe to laugh in the guy's face and tell them, why don't you show me how to do it?

So what is better than undecidable? There are these problems here that are outright impossible. What's the next best thing?

AUDIENCE: Chess? I know chess is in there somewhere.

PROFESSOR: Chess is still here. There's something harder than that.

AUDIENCE: Is it R, or something like that?

PROFESSOR: Yup. So this is the boundary for R and this whole thing is R. What R? R is-- so if these are undecidable, R means decidable, right? So you can write an algorithm that will compute the answer and do that in a finite amount of time. Will the world end before the algorithm ends? Maybe, who knows. But at least there's a finite amount of time, so the problem can be solved somehow. That means R.

AUDIENCE: R is like, real, like it's not.

PROFESSOR: Real, OK.

AUDIENCE: So like numbers, like, you know, real numbers--

PROFESSOR: So you know it actually comes from recursive, because some old people decided that recursive means it terminates at some point. This is really old school stuff, 1940. So R actually stands for recursive. Just know that R means decidable. Weird abbreviation, but oh well.

AUDIENCE: I'll go with real.

PROFESSOR: OK. What's next? So some problems are in R. All the ones that we care about are in r. So what's better than r? R means it's going to terminate at some point, there is a running time for the problem. What's the better than that? Exponential, OK. What does exponential mean?

AUDIENCE: Exponential?

PROFESSOR: Yup. So the running time looks like what?

AUDIENCE: 2 to the 10, or something constant to the n? 2 to the n of the C?

PROFESSOR: For some constant C, right? So it's not just 2 to the n. The moment you have a 2 to the n, we're like, OK, if you're already in this bad of a situation, I don't care if this is a polynomial. I don't care if it's an n or n to the 100, you're already pretty much screwed, you're not going to solve this before the world ends. So that's what exponential means. Going to solve it, but not before the world ends for most practical problems. What's better than exponential?

AUDIENCE: NP.

PROFESSOR: OK, and before we talk about NP, let's talk about the easy case, the problems that we've talked about all the time in the semester.

AUDIENCE: May I ask you a stupid question? What does NP stand for?

PROFESSOR: It's not a stupid question. It has a really hard to answer. So the one-sentence answer is NP stands for non deterministic binomial. And we're going to go over that at some point in the next few minutes.

OK, so what problems have we talked about? All the terms, except for one problem, for the knapsack problem, the entire term we talked about some sort of problems. Polynomial. These are the reasonably easy ones. So polynomial problems are those where t of n is n to the C, so it's a binomial.

The running time is a binomial in terms of the input sites. Why does this matter? If you have a polynomial algorithm, than if you double the input, then the running time is going to be 2 to the C times n to the C.

So every time you double the input, the running time will increase by a constant factor. And you know what that is. If your problem is exponential, then if you double the input, then the running time increases quadratically. It's not a linear increase.

Here it's a linear increase. The factor isn't 2, but it's still a linear increase. Make sense? So that's why we like these problems, at least we like them more than anything else. So polynomial problems are the nice and easy ones that we've talked about so far. What does NP stand for? Give me a practical definition.

AUDIENCE: You can check it [INAUDIBLE].

PROFESSOR: OK, so NP means that we can write the verifier. I'm going to say exactly what you said with different words. That takes a solution to the problem.

And of course, it has to think the input to the problem and tells us is it correct or is it not correct. And this very fire is NP. So the very fire runs in polynomial time. Yes. So if the verifier runs in polynomial time, what's the solution size and what's the input size?

AUDIENCE: I think the-- I mean, the solution said it's like 01, right?

PROFESSOR: If we have decision problems. An algorithm that solves an NP problem. If you run it against the verifier, also has to convince you that its answer is correct. So the problems that we talked about in class were decidability problems where you have to say, is it possible to do this? Yes or no.

When you have a verifier with you, then you have to output a string that convinces the verifier that your answer is correct. So like if you have a problem that says-- if you have three sets, for example-- that says, given a logical expression, can you assign variable values such that the expression is true? If I say yes, what does that mean? That means nothing.

How do I convince you that my answer is correct? I will have to give you the assignment, right? And then you can verify in polynomial time if my answer is right or not.

So the input to the verifier, this solution, is not necessarily the decision. It's not just the bit, yes or no. So if the verifier runs in polynomial time, what's true about the problem input and what's true about the problem output which becomes an input to the verifier? They have to be polynomial in size. Otherwise, my algorithm wouldn't be able to consume them fast enough.

So actually, polynomial means that the running time is polynomial in the input size. So the input size is automatically all set. But the solution has to be polynomial in the input size as well.

And if you're thinking decision problems, you can be more rigorous and call this a proof. So if this proof is binomial in the input size, then if I'm really, really lucky, I can take the input to a problem, and I can flip coins and use the corn results as bits, and put them together, and create the proof. If I'm really, really lucky, I'll get the right proof.

Chances of that happening almost zero, but if I'm really, really lucky, conceptually I can get a proof. This is what a non-deterministic polynomial means. It means that if you have a magical machine where you can flip a coin, and it will always be lucky, then you can output the proof in polynomial time. So far so good?

So this definition of a machine that can flip random bits is really useful, because we can use it to build a common sense proof that P is not NP. There is no real rigorous math proof that P's not NP. That's worth a million dollars. There is no proof that P equals NP. If you have proof that P equals NP, then that's worth a lot more. And I will use that in the common sense proof that I'm going to show you now.

So we don't know if P is NP-- get it right-- if P is NP or if P is not NP. This is an open problem, and there's at least $1 million worth of prize money for it.

Let's talk about an algorithm where if you would know-- so if you would be able to solve NP problems in polynomial time, you would be able to make a lot of money. A million dollars is nothing compared to what you could make if you could do that. So I'm going to use that to convince you that nobody knows how to solve NP problems in P time.

Suppose you want to do something really bad. Suppose you want to impersonate a bank. So you guys probably imagine that banks nowadays don't transfer money to each other by sending trucks with gold bars. They do that every once in awhile, but most of the time the transfers happen very quickly over the internet. So banks have a secure connection, and they say, yo, give me a billion dollars. I'll give it back to you in a few days. Sure. Done deal.

Now imagine what you could do if you could impersonate one of those banks. Sweet happy life in some faraway place, right? What would it take to impersonate another bank? It turns out that the encryption algorithm that use to secure this links is an algorithm called RSA. Do you guys remember that? Heard about it?

So in RSA, each party-- so each bank has a secret key that consists of two numbers-- P and Q-- big prime numbers. And say each of these are N-bit prime numbers. And then they have a probably key. So they have a number that announce that is N equals P times Q. So this is announced to the entire world. This is public. We went through this [INAUDIBLE] set, right? So this rings a bell.

They have to announce this so that the other people can encrypt messages for them. If you want someone else to be able to send you a message, you have to tell them your n. Now if you know P and Q, you can decrypt messages. So you can pretend you're the bank.

Now let's set up a problem in this way. Given a verifier that does long division-- so we have a verifier that the verifiers input is N, and some guess P, P guess, right? So the verifier will compute N modulo P. And if N modulo P is 0, then it will say happy. Actually, it will say, you're rich as hell. If it's not 0, then well, tough luck, try again.

So if you could make this verifier happy, you could get P. So given public information, you could get the private information that would allow you to impersonate the bank. This is called a factoring problem by the way. So P and N are usually 1024 bits. So the chances of doing that with a coin flip, no. The chances of guessing it are one in two to the 1024-- it's basically worse than having to pick a random atom in the universe and choosing the right one. So not going to happen.

But if you had an algorithm that takes a verifier and produces an input that makes it happy, then you would solve this problem, and you would impersonate the bank, and you would become really rich, and then the whole world's economic system would be in a terrible situation.

AUDIENCE: P and Q are primes, right? So couldn't you just find all the primes?

PROFESSOR: Well, so look at it from this perspective, the economic system still functions. It's not like everyone's money disappeared somewhere. Therefore, nobody was able to pull this off.

AUDIENCE: Isn't there a quantum algorithm to do it or that's something different?

PROFESSOR: So this assumes Turing machines. We're assuming regular Turing machines. A quantum computer breaks that abstraction. For practical purposes, if you had a quantum computer that can manipulate this many bits. Then you would break RSA, and the world would go to hell. So hopefully by the time quantum computers get this powerful, we'll invent new security algorithms that replace them.

But the fact that eCommerce works now and that banks work means that nobody is able to solve NP problems in polynomial time. So by the way, factoring is not the hardest type of NP problem. Factoring is somewhere here. OK. Any questions about this part?

AUDIENCE: So factoring is not an NP problem?

PROFESSOR: Factoring is NP, but it's not the most difficult type of NP problem. We'll get to those right now. So this is a common sense proof not the math proof that NP is not P-- that at least nobody knows how to solve NP problems in polynomial time.

So there are these NP problems that if you think about it, the solution is polynomial in size. We can write a verifier in polynomial time. So why can't we write the solver in polynomial time? If you're a very high picture guy like managerial type, then you might think, yeah why, can't you guys just figure this out? Like come up with an algorithm. Isn't that what you guys do?

Well, so theory people have been thinking about this for 40 years or more. So what do you do when you think about something and you can't come up with a solution?

AUDIENCE: Say it's impossible?

PROFESSOR: You try to say that it's impossible. So you try to prove that P is not NP. But if you fail to do that too, what do you do next?

AUDIENCE: Offer a million dollars to someone who can do it.

PROFESSOR: That too. That might help. That might help. Well, you complain that it's really, really hard, right? You want to go back to your boss or you want to go back to the world, and say, we thought about this. It is true that with thought about it for four years, but it's a really, really hard problem.

Well, so for undecidable problems-- so for the whole thing problems, they are able to convince the world that nobody can solve this, so we're all good. So we couldn't come up with a solution, because nobody can. Here, we can't prove that P is not NP yet.

So instead the next best thing that theory people could come up with is-- they saw, what are the hardest kind of NP problems? Let's look at them and let's see if we can solve them in some way. And they found some problems here that are the hardest NP problems. And they called them NP-complete problems. And we'll see why complete in a bit.

So these are the hardest NP problems that you can have. Factoring is not one of them. Factoring is not with the cool kids, even though it would make you rich. There are some problems that are harder than that. OK so for these NP-complete problems, turns out there are a few hundred of them that would have practical implications. So you wouldn't be able to get rich right away, but still you'd be able to solved the important problems that would save companies a lot of money. So you think, if there's a solution someone would come up with it.

It also turns out that they're all interrelated. So if you can solve one problem, you can solve all of them. And you do that through reductions, which we'll go over in a bit. But the bottom line is there are hundreds of NP-complete problems. And if you solve one, then you solved everything.

For now theory people are trying to say, look there are all these problems. If anyone would solve any of them, all of them would go away. Nobody was able to solve any of them, so they must be really, really hard. This is the best they could come up with. This is NP complete.

AUDIENCE: This is same lecture that you-- any problem that's already [INAUDIBLE] transform it into a different problem?

PROFESSOR: Yeah. So if you have the solution for one of them, you could transform all the other ones into that problem just like we do with graph transformations, solve it, use the solution for it to solve all the other ones. So if anyone could solve any of those hundreds of problems, we'd have the solution to all of them in an instant. The fact that we don't have a solution to them means that nobody was able to solve any of them.

AUDIENCE: Like there are some problems like the chess problem that can be solved exponential time.

PROFESSOR: Those are different. Chess is here.

AUDIENCE: So we can solve chess though, right? It's just exponential time. Right?

PROFESSOR: As far as I know, because it's exponential time, there's no algorithm that can actually look at the entire configuration space, and tell if you can win or not. So solving chess, what's chess? Chess has given a board. Can I win or not from this board? This would be very useful in playing chess, because you start with the initial board, and then for every move you can make, you look at the resulting board, and you say can I win or not from that board.

If you're in a board configuration in which you can win, you don't want to go to a configuration where you won't be able to win anymore. So if you keep doing this you eventually win. So it would be really nice to have this algorithm. Yeah, we don't have it. We don't have it and chess is unsolved. Wait.

AUDIENCE: So only things that are below P-- but there are some algorithms that run in exponential time. They just take longer.

PROFESSOR: So the problem is what's the input size? Exponential in what? If it's going to be-- if the running time is 2 to the n, and it's 10, then that's fine. But for chess its exponential in the board configuration. And it turns out that there's so many board configurations that we don't have computers fast enough to solve the whole thing.

So deep blue goes to a level-- to a depth I think of somewhere around 15 and can barely competes with humans, but it's not guaranteed to win all the time. So we don't have something that has completely solved chess.

AUDIENCE: So we can only solve things below P then.

PROFESSOR: We can only solve things below P fast, reasonably fast. So what we care about in this class is how do our algorithm scale. So do we have this or not? Does your algorithm scale or not? The reason we don't like algorithms that are after this is that for problem sizes that are really small like 8, 10, or something small, they're going to work. But the moment the problem gets bigger and bigger, you're going to run into a dead end.

AUDIENCE: Well, can you use DP on chess?

PROFESSOR: You can use DP until you get to an exponential algorithm. You might get to an exponential algorithm where the exponent is better. So far all the smart algorithms for chess have reduced the c here, but they haven't been able to get away from 2 to the n part. If you make c smaller, sure it's going to run faster, and you can explore a bigger part of the board. But you're still not going to be able to run through the whole thing before the world ends.

That's a good thing to think about. Will we solve chess before the world ends with the current algorithms and machines? No.

AUDIENCE: I guess. Remember the Rubik's cube? Wasn't that bordering exponential or something? I think Eric said that there is order one solution where the constant factor is really high. So that's different. raises I think Tetris is here, but the Rubik's cube--

AUDIENCE: It's in the P, isn't it? Yeah.

PROFESSOR: There's a solution that's order one with a high constant [INAUDIBLE].

AUDIENCE: But our brains can play chess.

PROFESSOR: Sorry? Our brains--

AUDIENCE: We can play chess.

PROFESSOR: Well, we can play reasonably well, but if someone could play optimally, they'd probably beat you. Yeah.

AUDIENCE: Can someone play optimally? [INAUDIBLE]

AUDIENCE: What about machine learning?

PROFESSOR: Machine learning problems are usually here. That's why they don't scale, and that's why we've seen a lot of interesting machine learning, but so far I can't talk to my computer right. They can't make a lot of progress, because they have these really hard problems that they're working on.

AUDIENCE: There's Siri, right. You can talk to Siri.

PROFESSOR: Yeah OK. Sure. I think I've seen on the Internet that it says some awful things sometimes. So I wouldn't consider that solved.

So what does NP mean to us like practical terms? Here's how I see it, you're on an exam or in an interview, you start out with a problem. Suppose all the possible problems are a graph, because we work with graphs. You have your starting problem. This Is the problem that you're trying to solve now. Then you have a few destination nodes. Say you have merge sort. We know how to solve. If you can reduce your problem to merge sort, then you're in a good position. You're writing your solution. You're out of the room.

If you can reduce your problem to a graph search algorithm, so to BFS or DFS, you're done. If you can reduce it to shortest path, you're happy. You're done. If you can reduce it to a dynamic programming problem that we understand, you're done. If you can somehow use hashing or use divide and conquer with an algorithm that we studied, you're also happy.

So we have all these destination points. And what you're trying to do is you're trying to get from here all the way to here via reductions. So suppose you see three reductions that you could possibly do. You don't have time to work on all three of them, so let's say you choose the middle one, because this one seems like it's the easiest one. So I'm going to guess that this is the easiest one I can tackle.

So now I'm going to work on this. I'm going to put the original problem aside. And I'm going to say if I can solve this, then I'm happy. Now say this turns into three more reductions. And this one looks promising. it's a hard problem. You have to do a few reductions. It doesn't work right away. But say I take this one which looks promising, and I'm able to reduce it to a happy problem. Reduction, reduction, reduction. I write this up. I'm done. Exam solved or interview question solved.

Now the problem is what if instead of going on this path which is reasonably happy, what if I go on a path that looks like this? All the reductions that I can see from there go really, really far away from the problems that I know how to solve. Well you're going to try some reductions, and eventually you're going to run out of time. And this other interviewer is going to say, yeah, do you have any questions about us? Or the exam people will be like, hey, can we have the exam back? We want to go home now. So this is a bad outcome.

Up until now, all you had was good destination points. So this is where you want to reach. Now we have these NP-complete problems that we know are hard. I've convinced you hopefully that their hard. So if you know some NP-complete problems, you also have some landmines. You have some places where if you got there via reductions, not a good path. You want to backtrack and think of something else.

So for example if this was an NP-complete problem like if this reduction then was, oh yeah, solve [INAUDIBLE] in polynomial time. Let's see. I don't know how to do that. The world tried really hard. Maybe I shouldn't try this avenue on the exam. So if you know that this is NP-complete. You stop right there. You don't waste anymore time. So this isn't time wasted.

If you understand NP-complete, and you read the CLRS chapter on NP-complete problems. Then you also have some sad faces here. You have some land mines and you know not to go there. So now your space that you're looking through is a lot more bonded. Hopefully, you're going to have better chances of finding a happy path, because not only you have some destinations, but you have some places that you know you shouldn't go to.

So this is the point of NP-complete. You're probably really busy at the end of the term as we all are, so I'm guessing you're not going to have time to read CLRS beyond what we really ask you to read now. But after you're done, do yourself a favor. For the sake of your future interviews and general happy programming life, read the NP-complete chapter in CLRS. Read the problem statements and understand them. Don't worry about the proofs. Don't worry about the reductions that say their NP-complete.

Read the statements. Believe CLRS that it's NP-complete. And remember that whenever you solve a problem, you want to stay away from those. So those will be your landmines. Those will be your sad spots.

Guaranteed, after you do that, your interviews will go a lot better, because you'll be faster at solving problems. And that's what we really care about. So that's why NP-complete is relevant in real life. It gives us these data points. Makes sense?

All right, I want to talk about one NP complete problem that's really important. And that is SAT. So SAT means satisfiability. So given the logical expression, find some values for the variables that make it true. Let me grab an example from here.

So in SAT your expressions come out in a really nice form, so parsing isn't a big deal. The way they come out is there a bunch of terms separated by AND. So your expressions look like this-- and, and, and-- say we have four terms. These look good.

So you have some terms, say n terms, and they're all separated by ANDs. This is a term. In a term you have variables separated by ORs. Nothing else. Just either the variable or a variable with a NOT in front of it. So say x1 OR NOT x2, NOT x1 OR x4, NOT x3 or x4, and X2 OR X4. So this is called the Conjuntive Normal Form, CNF.

The reason this is really nice to work with is-- OK, you just have some variables here. You know that these are all joined by AND, so you're going to have to make all these terms true. And then you have ORs inside. So you know that for every term, at least one of the things inside the term has to be true. And that's it. This is the problem. This is SAT.

You have n terms. And then the number of variables that you have inside is k. So you have at most k variables in a term. This k looks like a constant factor. But it's so important that in fact it shows up in the name of the problem. The problem is called k-SAT.

And there are two values of k for which the problem is important. There's 2-SAT, and there's 3-SAT. 2-SAT is polynomial. We have a solution that runs in order and time in fact. Really good.

AUDIENCE: But you're saying that to make-- you're actually rearranging things or you're just--

PROFESSOR: So I have to come up with values for x1, x2, x3, and four, so that the whole thing is true. And the expression is already arranged in this nice form for you.

AUDIENCE: It's just-- it seems like order and, because you're just evaluating to block, right?

PROFESSOR: Well, no, you have to come up with the values.

AUDIENCE: Oh, I know. I know. But like--

PROFESSOR: OK and this seems really easy, right? Well, wait for it. Wait for it. 2-SAT is polynomial. 3-SAT is NP-complete. So if you solve it, you're in turn for a Turing reward, or you can become really, really rich. So n-SAT can be reduced to 3-SAT. So for anything bigger than three, n-SAT can be reduced to 3-SAT using reasonable reduction. So the input size won't explode too much.

So why is this important. This basically shows us that 3-SAT is NP-complete. And I can make a three-minute proof that 3-SAT is NP-complete. Let's take that verifier guy that we have there. A verifier wants the input, which already have and the proof. Let's write the verifier as a circuit that takes the input and the proof as bits. Those are input to the circuit. The circuit is going to evaluate to true or false.

Now that we have a circuit, it's easy to turn that into a logical expression. Turn AND gate and OR gate into-- and ANDs and ORs, and then reduce N-SAT to 3-SAT and get one of these. If you can solve 3-SAT, then you can compute values for the proof bits. Bam. That's it.

So I take the verifier and I turn it into a circuit. The circuit is all logic gates. And then it has some inputs, the original problem input and the proof. These are represented as a series of bits, and they're all inputs in the circuit.

If the algorithm is polynomial time, then the circuit also has to be polynomial in the input size. Proof already has to be polynomial in the input size, so we're all good. We're all in polynomial world.

Now we take this circuit, and we turn it into a logical expression. Sorry not logical, logic, also known as a Boolean expression. So the input bits become Boolean variables, and then we express gates as Boolean functions-- AND, OR, NOT, whatever we want.

And this is going to give us some expression. That expression is going to map to N-SAT for some value of n. So then we're going to take this expression, and we're going to reduce it to 3-SAT. So actually we're going to reduce the expression to 3-CNF and then run 3-SAT on it. The expression will be reduced to 3-CNF. So to something that looks like this, except you have three variables inside everything.

Once it's in 3-CNF, you can run 3-SAT on it. If it is satisfiable, then 3-SAT has to give us some values here for the variables. Some of the variables are the problem input-- Sorry the problem input gets hard coded in the circuit. Sorry. Not thinking here-- hard coded in circuit, because we don't want it to tell us what the input is. And then the proof is encoded as a series of bits, which are inputs in the circuit.

OK. So problem hard coded, proofs are inputs in the circuit. We've taken the circuits, turned it into an input for 3-SAT. If 3-SAT says there is a valid variable assignment, then that assignment tells me what the bits are in the proof. So that means I can take the proof, feed it to the verifier. The verifier will say it's happy, so we solved the problem. And this is true for any NP-complete problem. NP-complete means it has to have a verifier that runs in polynomial time. So I have to be able to follow this process.

So for the factoring problem, for example, I would take this modulo algorithm, I would express it as a circuit. The bits of P would be the input. I take that circuit and I turn into a 3-CNF formula, run it on 3-SAT. The variable values for which the circuit happy are the bits of P that make up. So they're the bits of P. They make up a factor of n. So if I can run 3-SAT fast enough to get an answer, then I have P and I have factor of n. And I've become really, really rich, and bye guys, I'm gone.

AUDIENCE: For each of those expressions, you said you make that circuit n. You've got all these inputs, which are x1, x2, everything like that. Could you just go through every possible combination?

PROFESSOR: That's exponential, right? That's two to the n, where n is number of variables. And the number of variables can be proportional to the number of terms.

AUDIENCE: Yeah. That would be fine with it.

PROFESSOR: So that means you have an exponential running time. It's not polynomial, which is reassuring, because otherwise it would mean that the whole world has been spinning their wheels around nothing all this time, right? So it's reassuring that you can't find the solution in two minutes to this problem. In fact, so far I've been trying to convince you that you should not attempt to solve this problem in real time, so not on exam time, not on [INAUDIBLE] time, nor anywhere else where time actually matters. You guys bored? Maybe.

Well, so there's this proof that's not too hard to follow that says that if you solve 3-SAT you have solved any NP-complete problem by turning the verifier into a logical expression. That's what it comes down to. Now 2-SAT is polynomial. And if you guys want, we can go through the solution. If not, we can skip that. But this tiny difference here, how many variables you're allowed inside, which looks like it might be a constant factor in an algorithm actually makes the difference between a very easy problem, a problem that's in P, and the hardest problem in NP.

So 2-SAT is order n, so it's somewhere here. 3-SAT is all the way here. Small difference how many variables you're allowed, two or three. Makes all the difference in the world.

AUDIENCE: There are logic reductions you can make, right?

PROFESSOR: There are optimizations that you can make. And there are some people who do research on how to solve 3-SAT in a reasonably fast time. Because if you solve 3-SAT, any problem can be reduced to 3-SAT.

All right, we have a feeling that we're not going to be able to solve this in polynomial time, because a lot of people tried, but some researchers are hoping that well maybe you can solve 3-SAT in order of 2 to the n to the power of 0.00001, which exponential, right? But you can solve many problems with this.

Active research, so far I don't think they've come very far. Actually, I shouldn't say that, because if you apply the brute force algorithm, then you die at about 20 variables. State of the art 3-SAT solving algorithms, I think it can handle 1,000 to 10,000 variables in a reasonable time, like a few hours. There progress.

And that is all by doing the reductions that you said. You see some optimizations that you can make in the expression. You draw some inferences, and you explore the possible configurations, taking us part way. So there's a lot of active research going in that area too. So far, they didn't get here.

AUDIENCE: That's almost solving NP-complete problem then, right?

PROFESSOR: Well, yeah, the thing is in the end we care about practical solutions, right? So they're hoping to get to a practical solution.

AUDIENCE: Well, a few hours is pretty practical. So that's for some problems. Obviously they didn't solve factoring, right? If they would have solved 1,000-bit factoring in a few hours, then we would have noticed.

AUDIENCE: Maybe. Seriously, OK.

PROFESSOR: We would have noticed. So far so good? Any questions on this stuff? So I want to reemphasize in two minutes, the reduction part of this. So the reason why we can solve, once we solve one NP-complete problem, we solve all the other ones is reductions.

So we've done a lot of graph transformations in this class. We had three recitations and two or three exam problems on that, and I think we had some homework problems on it too-- a lot of graph transformations. Graph transformations are just reductions. You take some problem that looks hard, and you do some magic, and you build a graph that represents that problem, and you run shortest path on it. Then you take the output for the shortest path algorithm, and you turn it into a solution to the initial problem.

This means that we've reduced those problems to shortest path. Well, you don't have to reduce to shortest path. If you have a problem that you know how to solve, you can take any other problem and reduce to it, and you've solved that other problem. So basically, here, if you have your starting problem, shortest path is just one smiley face. As long as you arrive to any smiley face, you're in good shape. Of course, you have to be careful that while you're doing you're reduction, your problem size doesn't explode.

So for example, if you take this problem-- if you take 3-SAT-- if you take a 3-CNF expression, and you reduce it to the graph that has 3 to the n vertices, and then you try to run [INAUDIBLE] on it, it was the running time, 3 to the n. Now I've got a new polynomial, right?

So when you do the reductions, you have to be careful about what happens to your input size. And that's about it. This is complexity theory. It's basically one big excuse for why we can't solve some problems that are hard.