PROFESSOR: All right. So today we finish our game characterization. We have this giant table. We've done two players, zero players, one player, of course. That's the oldest.

Today we're going to do team imperfect information games, which when you have a polynomial bound on the number of moves, you get NEXPTIME-- nondeterministic exponential time. This is the exponential version of NP. And for games with unbounded number of moves-- normally, I would say exponentially many moves, but, in fact, this will be really unbounded, could be close to infinitely many moves-- we get Undecidability, so no algorithm, whatsoever, to solve it.

We don't know any natural games in these classes. But I have a couple of question marks here. Bridge is an example of a game with imperfect information and teams in one hand. And it has bounded play. Once you play your cards-- all of the cards are down-- you're done. So potentially, it falls in this class. Although, I don't know if Bridge is sophisticated enough or-- it depends on your notion of the bidding rules and what can be communicated there.

On the other hand, for the unbounded game, one candidate problem is called Rengo Kriegspiel. Kriegspiel is blind chess. Rengo Kriegspiel is blind Go with teams. So there are two black players, two white players. And you make moves without knowing what the other moves of any of the other players are, usually by writing on a piece of paper and giving it to some referee who tells you whether that was valid.

This is a game people actually play. And I'm told-- I know Bob Hearn has played it. He likes playing Go. But it's open, whether that problem is Undecidable. Potentially, yes. That seems more likely.

And then finally, one other cell that we didn't do, which I'll get to after those two, is

this box, which sounds pretty boring because it's just polynomial time things-- polynomial bounded zero player games. It's just like Game of Life running for a polynomial amount of time. You can generate-- I mean, you can solve those problems in polynomial time.

But there's a stronger sense in which all polynomial time problems can be represented, say, by Game of Life. And this, actually, is relevant to parallel computing. And it essentially says, there's no good way to parallelize Game of Life, among other problems.

So I mentioned that way back in lecture one. And a lot of people said, oh, what's this P-completeness business? And so today we will cover that, as well.

But let's start with NEXPTIME and Undecidability. Let's start with a bounded case. So in the bounded case, there's this cool paper by Peterson and Reif, introducing a problem variation of QSAT. Remember, QSAT, also known as QBF. So this is DQBF.

And it's a funny problem. First, I'll phrase it as a formula problem. And then I'll talk about it as a game.

So normally, in QSAT QBF, we have N alternating quantifiers. And that's PSACE complete. And every quantifier can depend on all of the previous values.

But now we're going to have two sort of in parallel existential quantifiers. So we have two universal-- I should mention the capital letters. That's a bunch of variables. So there's like N bits here, N bits here that has to work for all choices of those. Why do I separate them out? Because over here, I'm going to separate them out.

And Y1-- the choice of Y1 is only allowed to depend on X1. And the choice of Y2 is only allowed to depend on X2. And then we're going to have some CNF formula on the X1, X2, Y1, Y2.

So we can think of that as this is one player. It could be two players. But it doesn't really matter. So make it one-- simpler.

It's going to be a three player game. Black makes these choices. And then white player 1 chooses the Y1s. And the private information aspect of this game is that the white player one can only see the X1 variables-- choices made by black.

Maybe it's more intuitive to think of this as a four player game. There are two black players. And the white 1 player can see the values chosen by the black 1 player but not the values chosen by the black 2 player.

And then, over here, we have white player two, who can only see the variables chosen-- X2. Question?

AUDIENCE: Does the black team have full information?

PROFESSOR: Black team has full information. I mean, the black team goes first. So it doesn't, obviously, see Y1 and Y2. But they can see each other. Yeah. So X2 can see X1. Yeah.

So if I don't write a parenthetical dependency, then it can depend on everything. Yeah. Really, this is one player. So they get to choose everything in the X [INAUDIBLE]. OK?

So then the question is given this-- I mean, this quantified formula's equivalent to saying, can white force a win, where white wins if that formula is satisfied? And white collectively needs to make these choices. But it's acting as a team with impartial information. So it's not acting as a single player anymore, unlike when we had full information. Then three players was the same as two. OK?

So let's first observe that this is in NEXPTIME. And then, furthermore, the claim, which I won't prove, is that this is NEXPTIME complete. So the idea is there are only exponentially many choices for these variables. So then I can guess-- both player 1 and player 2 can guess what-- white 1 and white 2 can guess what the possibilities are for all possible choices of X1 and X2.

So they're making exponentially many guesses. For every state of X1, X2, guess what the right strategy is. And then just combine-- then check whether everyone

plays properly. So then white can decide whether they can win. That's cool.

This can be reduced to a bounded team, private information version of constraint logic-- TPCL. Let me define that. We're going to do this with three players And a plane or a graph. I think just and/or graph is enough.

But it's a team. There are going to be, again, two white players, one black player. And it's private information, meaning that each edge is marked as visible to white or visible to black.

Now, the figures-- it's a little hard to draw. So the figures-- the color is just indicating whether that player is allowed to flip that edge. Each player can only flip-- each white player can only flip white edges. Black player can only flip black edges, just like before.

Well, now it's white team can only flip-- either of the white players can only flip a white edge. So some edges are flippable by two players. But then, in addition, some edges-- some white edges are going to be marked as invisible to black. And some black edges are going to be marked as invisible to white. And furthermore, you can say invisible to this white player or this white player. OK?

So it's a little bit more general. And the rules of the game are pretty simple. You can make moves just like before. But you must be able to know that that move is legal.

So unlike Rengo Kriegspiel, you're not allowed to make impossible moves. You must know, based on the information that you can see, that the movie is valid. There are probably other variations that work. But this is a clean one to work with.

So if you can see all of the edges, or you see that there's enough incoming white to an edge, then you can reverse one of the ones you have control over. So this problem-- I mean, so a funny thing about DQBF is that there's sort of only one round to the game-- I mean, black plays, white plays, white plays, and then you see who won-- whereas most games we think of as having many rounds.

And in particular, bounded TPCL has many rounds. They play in round robin.

You've got black, white 1, white 2, black, white 1, white two. And you play in that sequence, over and over. Each turn, someone's just reversing a single edge, whereas over here, you're choosing a ton of variables.

Turns out this problem is also in NEXPTIME, essentially because if there is a winning strategy for white, over here, that winning strategy is deterministic. And it's just a function of the visible state. And there are at most N different edges that you can see. So the number of possible states is exponential.

So at the beginning, you just guess your strategy. That's exponentially many bits to guess. So in NEXPTIME, you can do that. And then you just play deterministically, with that strategy.

So even though the game may run for a very long time, and you do many, many rounds, still, the information you need to guess was only exponential. So you can do this in NEXPTIME.

So it's a funny thing here. In this particular world, unlike all of the other cells of that matrix, one round versus many rounds is the same. You get NEXPTIME completeness in both cases. OK.

So let me tell you how we reduce from DQBF to bounded TPCL. So many acronyms. So the idea's pretty simple. Again, we want black to play first. So there's a bunch of black variables, here, which are represented by this. These two figures are almost the same. Just these edges are blacked down here and white down here. That's going to correspond to the variable setting.

In this context, we only want variables to be set once. You don't get to reset them because it's a bounded game. As in all of the bounded constraint logic games, you can only flip each edge once. Question?

AUDIENCE: It's kind of a question about your earlier point, but could you clarify the difference between why having one round of one of the games is equivalent to having all-around [INAUDIBLE] game?

PROFESSOR: I don't have an intuitive reason why-- only that this game is in NEXPTIME and this game is NEXPTIME complete. I mean, they both are NEXPTIME complete. So in that sense, they're equivalent. Yeah.

So we're going to simulate this one round game with multiple rounds, over here. But yeah. It's a little bit odd that alternation doesn't help. It doesn't give you more power. All of the other game settings that it does. So it's just a weird thing about team. Yeah.

AUDIENCE: So are you [INAUDIBLE] kind of rely on the fact that white doesn't learn anything from black's moves when black moves on an invisible edge? Sorry. It seems like in some board configurations, if black moves on an invisible edge, you can actually learn something based on what moves were allowed, and not allowed, for black.

PROFESSOR: Maybe I should say that passing is allowed in this game. Otherwise, the fact that you moved does tell you something. But if you're always allowed to pass, you really have no idea whether something happened, except by what was visible. Yeah. So that's-- make explicit, passing is allowed. Yeah, these are a bit subtle. OK.

So the intent is, first, we let black choose variables, either by flipping true edge, flipping the false edge. We want to set it up so you can't do-- well, you can't do both because of this vertex constraint. So that's cool.

As soon as that happens, probably, white will flip this edge and black will flip this edge. That's a split, activating-- well, not yet activating the formula part, but enabling black to activate the chosen edge, saying that this variable has been chosen. And so black will sort of output that.

Then in here, so there's n such variables. And B1 and B2 are both in here. And we take the and of all of those chosen edges. And that's this chosen wire.

So this-- black is motivated to do this. Black wants to set their variable some way. And they could, I guess, set them back and change the variables. But there won't be any motivation to do that. The goal for black is to flip this edge, which is connected by a long path from the AND of all of the chosen wires.

So basically, B has to choose all of the variables, get this AND to be true, and then just sit there, flipping these edges, until this edge is flipped. So that's a threat line, once B has set their variables. So B is going to race to do that. White is going to have just enough time to satisfy the formula and flip this edge if the formula was true, obviously.

So what happens next-- white will basically sit there and twiddle their thumbs, flipping some useless edge, until all of the black variables are set. They could actually do something else. But there's going to be enough time to wait. So it's better to wait.

Then the white variables can depend on the black variables that it can see. Obviously, pretty much all of the edges in here are visible to only one of the white players. If this is a B1 variable, then this is only visible the white 1. White 1 can see that it was chosen but can't see which of these edges got flipped and can't see which of these edges got flipped or these ones. OK? So that's representing the visibility.

And so then white will set their variables, the same kind of construction. And once chosen is activated for both of them, we take the AND here, and we trigger the unlock paths. It's going to unlock the white 1, unlock the white 2, unlock all of the black variables. And the unlock part is what lets you trigger the formula activation.

So if true was selected here, and this was flipped, once the unlock is flipped, then this guy can activate and start inputting all of the true variables into the formula. And then white will sit there and evaluate the formula. It has just enough time, if it's satisfied, to flip and be done.

So that's how we represent dependency QBF. I mean, it's pretty natural. It's just some fun stuff to get these to happen in essentially two rounds. The visibility constraints of the partial-- the privateness of visible information makes this part really easy to implement. So no big surprises. Questions? OK.

Let's go to the unbounded case, which is the much more exciting one. So I mean,

Undecidability is weird and unusual. It's weird in this context because this game has finite resources. All of our games-- in particular, in constraint logic-- you have some graph of size n.

And Undecidability is about, essentially, simulating a Turing machine for arbitrary amounts of time and arbitrary amounts of space. It's bigger than 2 to the n. It's bigger than 2 to 2 to the n. It's bigger than 2 tower. And it's bigger than any-- I guess, it's any computable function of n is how far you have to go. So a little bit less than busy beaver.

That's a lot of space. So in game graph, we can't represent the state of the machine that we're trying to simulate. So I should say-- we haven't done Undecidability at all, but the canonical problem to start from is a halting problem.

You're given a Turing machine. And here, I'd like to say algorithm, but we need some model of the machine growing to arbitrarily large sizes. And this Turing machine is really ideal for this.

Given the Turing machine, does it ever halt? So remember, a Turing machine, you have some tape. Let's say, initially, all of the tape squares are empty. You have some pointer into the tape, starts at some canonical position. And you can read and write squares in the tape. And you can move left and right. And that's it.

The feeling, as this corresponds-- this includes all possible computation. And so deciding whether this machine will ever finish-- so there's an instruction that just says, halt-- or whether to just keep going on forever seems-- is provably impossible. There is no algorithm to decide that, essentially because it's an algorithm to study algorithms. But I won't prove that here.

I'm going to reduce from this problem. And so I need to be able to simulate a Turing machine of arbitrary size and time. Cool.

So while most of the time I don't show you the source problem, I think, here, it's instructive to see-- well, I guess I'm not proving this as undecidable. But it's

instructive to see how we turn this into a game, with bounded resources.

So before I get to the constrain logic version, I'm going to talk about something called a team computation game-- this is from our book, *Games, Puzzles, Computation*-- that will allow us to simulate the halting problem, but using finite resources. This is where the action is. Going from here to a constrain logic is not that hard and not super interesting. But this part is quite interesting. OK.

So in team computation game, there's going to be three players-- one black player, two white players. And the instance to the game-- sort of the board set up, whatever-- is going to be-- here, I could use any algorithm or Turing machine. It's going to be machine. And it has some space bound which are called k. k is going to be, essentially, a constant. But it's part of the input.

And the idea-- if this is an algorithm, it starts with blank memory. If it's a Turing machine, it starts with a blank tape. And then we're going to define black moves and white moves. So remember, there's one black player. And what the black player does, or what the black player's forced to do, is run the algorithm or the machine for k time steps. So k obviously is an input. So black is forced to do that.

Now, that algorithm may return an answer. And the answer is always black wins or white wins. Or it may not do anything, in which case it wants to keep running. In the next black move, it's going to run for another k steps. And in the next black moves, it runs for another k steps, and so on. Think of k as a constant.

So if there's output from this algorithm or machine, that will determine the winner. OK? And otherwise, black actually gets to make a choice. This was completely forced, what black had to do here. You could think of it as part of black's move. But it's really-- the game mechanics make this happen.

And in this case, black can set two variables-- X1 and X2-- to be anything in the set A, B. OK? Black gets to choose two bits if the machine doesn't say that the game is over. OK? We'll see what those bits do, in a moment.

So that was black's move. Now, what about white? Again, there are two players--

white 1 and white 2. And white i is basically blind. White i can only see Xi.

These Xis are messages to the white players. And that's all that white can see. White can't see anything else about the machine. Good that it knows what the machine is, knows what it's doing, probably knows k. But that's because that's all part of the instance. But you don't know anything else about the state of the machine.

And white can also do one other thing. There's one memory cell in the machine, called M1. And there's another one called M2. And player white i can set Mi. And so that will communicate, back into the machine, what white i did. And that is exactly the move of white i. White i move is just set Mi.

And I'm not specifying how big that is. It's not just a single bit. It could be some stuff. In particular, it's going to be one cell in the Turing machine we want to simulate, essentially. But OK.

So now the question is, does white have a forced win? So this is the problem I want to prove undecidable by a reduction from halting problem. OK.

So is the problem clear? We have a finite setup, which is this space k situation, that the entire state of the game is, in some sense, encoded by that-- the state of that algorithm, it would seem. And black is just telling-- in each turn, white 1 says A or B, white 2, it says A or B. White 1 and 2 then respond with some symbol. And then the game continues. So a pretty simple setup. But amazingly, this can simulate any Turing machine.

So let's do it. So this algorithm is going to be a function. Given the Turing machine, we're going to construct the algorithm, naturally.

So we basically need a constant space algorithm to check that the white players produce-- so the white players are outputting a strain of symbols. White 1 is outputting M1 and then outputting something else to M1, and so on. That strain of symbols, for both white players, should essentially form a valid computation history, an execution trace of the Turing machine we want to simulate-- the given Turing

machine. It's a little bit more complicated than that. But that's the idea.

And we'd like it to end in a halt state. So there are lots of canonical ways to write down the state of the Turing machine. Basically, you write down the tape and what instruction it's on. And that should be about it.

And we're going to put a special symbol-- hash mark-- in between each of these states. So this is the state of the machine at time 0. It's going to be a blank tape.

And you only have to write as many symbols as you need. So in the beginning, don't write any symbols. As you start writing stuff to tape, this will get longer. And yeah.

And then the goal is to get it to a halt state. Each of these states should be the uniquely determined state from the previous one, which is, if I do one instruction, what happens? OK?

Now, this is problematic. So our goal-- we're trying to design this algorithm to confirm that these states are being generated correctly. That's really hard to do because we can't compute the next state from the previous one because we can't even store the previous state in space k. We have constant space. Very soon, the state is going to get huge.

So you think of this as kind of a streaming [INAUDIBLE] We're just seeing these characters go by. And we can't remember the entire previous states. So the fact-- to check that the next state is almost the same as the previous one is very hard.

And that's where we're going to use this A, B thing. This is not literally true. The sequence of characters-- M1, say-- will not produce a valid state.

Essentially, what we want player white 1 to do is to maintain two pointers into this state-- A and B. And when we-- I'm identifying with black here. When we specify X1 equals A, we would like to know what character's being pointed to by pointer A. And then we'd like to advance A to the next character. OK?

When we request B, with X1 equals B, we want to know what character is at position

B, and then advance to the next character. So there's one strain that we're trying to look at. And each white player has to maintain an A pointer and a B pointer. And as black, we can make independent requests to each of the two.

It's like you have two disks with two heads on the disk. And they both have the exact same data. Or they're supposed to. And I can say, give me the next character at this pointer from this disk.

And I can give you-- ask for what is-- simultaneously, I get one character from one of the two pointers on the other disk. And then it advances by one. OK? Seems almost the same. But that's what will make this undecidable. OK.

So here's the trick, how we're going to build this algorithm. I mean, the real trick is that we have nondeterminism. We're asking whether white wins. That means-- or whether white has a winning strategy. For white to have a winning strategy, that must mean it can win or the white team can win, no matter how black plays.

And black is just non-- so basically, black is nondeterministically choosing between the A and the B pointer requests. And so what this is saying is, no matter what sequence of As and Bs you get, white must win.

And winning is going to happen because we will build this algorithm to only output yes when it gets to a halt state, basically. The algorithm must not complain, no matter how we choose the A and B sequence. So let me get to what we're going to do.

So the algorithm maintains-- let's call this white 1. I would like to know when-- let me be specific-- when white 1's A state equals white 2's B state. But in fact, we'll do this for all pairs. It'll be four different things. A, B here. A, B here.

But let's say I want to know when white 1's A state is in exactly the same place as white 2's B state. So we have-- in some sense, there are two of the strains, one for white 1, one for white 2. And I want to know when I have just to read through a state by white 1 and I've also read through the exact same state from white 2.

So that will happen when-- first of all, both white players should have just returned a hash mark. And then-- hash tag, if you prefer. And then also, while we were running through this tape, I see whether they happen to always return the same result.

So this will basically happen when-- so whenever both A and-- so if we just ask for A, A, and white 1 returns a hash mark and white 2 returns a hash mark, then we're going to start paying attention to every single character we see. If-- assuming we keep playing A, A, is black. And this has to work for all possible assigned choices of A, B. So in particular, it will be an execution where we play A, A throughout this entire state.

And as we're doing that, we just check, character by character, are they returning exactly the same result, and have they, so far-- have they always, since the last hash mark? OK? And if they did, that's a constant space algorithm. I just compare character by character. Remember the AND of all of those comparisons.

If they were all the same, and I reach a hash mark, that means I know that both the W1 strain and the W2 strain just output the same state. So that means the A and the A pointers are in the same place at this moment. Cool.

That's not quite where I want them. But at least I can detect that. OK? I can detect identical states.

Remember, I can't even store-- I can't even count A and B because these could get to really huge numbers. That only buys me an exponential when I write things in binary. So I can't afford to write down the A and the B pointers. It's what I'd like to do. But I can check, oh, did they just output the same state? So far so good?

Now, what I'd really like them to be is one step out of sync because what I do have a streaming algorithm for is if I have the previous data of the Turing machine and the proposed next date of this Turing machine, and I compare them letter by letter, I can just check, are all of the tape squares identical except for the one that the pointer's on? And that got changed by a simple rule. And then all the others should be identical.

So it's very easy, in a streaming sense. If I'm giving the previous state and the next state, I'd be happy. So my goal is to get W1 state to be one step out of sync, one step behind W2 state. OK.

So how can I make that happen? Again, so I mean, black isn't really making-- well, black is making all possible choices among A and B, at all times. So in particular, we can consider the situation in which black plays, let's say A for the W1 player, and plays B on the other player.

W2's A state does not change because we're advancing B. And then W1's state is advancing. And if I keep doing that, and then, at some point, I get a hash mark, which I can obviously detect in constant space, then I advance my finite state machine, and say, OK, cool, that means W1 is one step ahead-- one state ahead of W2.

And now I'm going to compare. So when that happens, so then out of sync by 1. And now what I'd like to happen next is X1 and X2 moves. Then now I'm going to run my streaming algorithm to check that this was a valid transition.

So from whatever W2 is outputting to whatever W1 is outputting because W1 is ahead, that should be a valid state. So basically, I had-- I ran for some time. Who knows what happened. Then I got this thing which is identical to this thing.

I detect that. And I see a hash mark. So I say, OK, they must be in the same state.

Now I'm going to skip over whatever state was written here. And then I'm going to check that every thing I get here is a valid transition for whenever I get here. So this should go to that. This should go to that.

And I don't need any memory to do this. I'm just looking at this square, knowing the state of this machine, looking at this square, making sure that was a valid transition. Usually, it's just that this is equal to that, except at the one place where the tape head is, which is written in that little symbol.

And in that spot, I do whatever the Turing machine does to that symbol. If the writes

to it, it might change. Maybe it moves the pointer left or right. But in constant space, I just need to know what's happening here. And maybe it's neighboring squares. I can figure out what should be written here. And if white 1 ever does the wrong thing, if it's not a valid transition, then I will say black wins as this algorithm.

So for white to win, it can't make any mistakes. White has to conform to this idea that basically, first, it-- well, it basically-- as it's playing, it must be computing whatever the Turing machine that's part of the game, essentially. It's embedded in this algorithm, the Turing machine.

And so we have to-- it has to be simulating the Turing machine, maintaining these pointers, A and B, and running through. It has to do that because it has to win, no matter what black does. And it could be that black does this really weird sequence of As and Bs in order to verify the one place where you lied about one of these being an invalid transition. It will eventually check for some choices of the A and B moves. It will check that it was a valid transition, from here to here. And therefore, all transitions must be valid.

So it's a pretty crazy use of essentially the universal quantifier that's given to black. White has to win, no matter what black's moves are. So you might ask, where is the Turing machine state. And it's essentially in the heads of the white players, which are not represented directly by the game.

The game is just, you know, what is the state of this algorithm? And it's, in some sense, informed by the history of the plays, the sequence of moves that have been made. That's, in some sense, where you can also see this long sequence of moves. I mean, each of these got output at some point.

But initially, you could sort of think of, all of the execution was in the players' heads. Then they're just maintaining these A and B pointers and playing. And there must be such an execution in order for white to win. So white will win, if, and only if, there's a-- the machine halts, basically, because this algorithm only output yes-- output that white wins when the machine halts.

And it will report that black wins, if you ever cheat. So you have to not cheat. And you have to get to a halting state.

It's pretty crazy. And this probably won't make sense the first or second time. But it's clear to me, again, every time I learn. It's like, oh, yeah, that's really cool. Hopefully, it will become clear to you. But if there are any questions--

AUDIENCE: Why is it important that the algorithm is only one space, in the first place, though?

PROFESSOR: I mean, it's not technically-- the overall algorithm is not technically constant space because it needs to include the entire Turing machine as part of it, to know how to run. But I mentioned that the checking algorithm is constant space. And the point is, really, that the size of this space key algorithm has to be only a function of the given Turing machine. It can't depend on the execution time or the execution space of the Turing machine. But that's all I want to emphasize.

So effectively, I mean, if you think of the Turing machine as being pretty small, but it's running for some huge amount of time, like busy beaver time, relative to that, we're thinking, basically, constant space. And in fact, we can get away with constant space. That's for the checking part. That's my point. If you view the Turing machine execution as an oracle.

So when I check that this goes to this-- in fact, I have to run the Turing machine on this tape head input, to see what it would output. But the Turing machine is inside the algorithm. So I can do that and [INAUDIBLE] some time with the lookup table.

So the point is, really, you need, effectively, a streaming algorithm to do this because these states get to be really, really huge, because we want this reduction to be polynomial time. And it is. Basically, copy the Turing machine over, write some constant amount of code to do this checking, and that's all you need.

I guess we don't need it to be polynomial time. But it is. So it's kind of nice. But we need that it's finite time. And there's no finite way to give an upper bound on the running time of the Turing machine, other than running it. So polynomial time is good. OK.

Let me briefly mention a more formula-oriented game. This is not terribly exciting. I'll just write it down quickly.

Basically, you can convert this problem-- team computation game where black is forced to run a particular machine-- you can convert machines into formulas. So it ends up looking something like this. It might be possible to simplify.

This is one thing that definitely works. OK. So again, white 1 can only see one variable-- X1-- that is set by black. And white 2 can only see one variable-- X2-- that's set by black.

Black can set a bunch of variables in one round. And there's some condition that it has to satisfy. This will basically force you to simulate the Turing machine and not break any rules.

And if black ever satisfies something, which is when the Turing machine's, hey, black wins. Then black wins. And then you just turn that into a formula.

And then black sets some other stuff. And so this is more in the style of the dependency QBF. Again, you have this kind of dependency. But now there are multiple rounds. And now there's no bound to the number of moves you make. We can reset variables many times. And this is undecidable by essentially simulating this machine, turning that machine into a formula.

So once we have that, we can turn it into a constraint logic version. This is the team private but not bounded constraint logic. Pretty much the same setup. Black has these variables, X and X prime. Y1 has these variables, Y1. Y2 has these variables, Y2.

They're fed into this formula. And there are various things to-- that when those conditions that I wrote down happen, you trigger the appropriate target edge. And then black wins, up there, or white wins, down there.

So not too exciting to go through the details of that. But the result is team private

information constraint logic is undecidable for three players. And you can use a crossover. And you can make this planar graphs if you want. Any more questions about Undecidability?

AUDIENCE: Are you ever going to give some examples of undecidable-- like real games?

PROFESSOR: No. There are no known good examples of games. These are the games, I'm afraid. Yeah. Sorry.

So I would like to switch gears and go to the extreme opposite. Instead of Undecidability, when there's no algorithm, let's go back to good old-fashioned polynomial time algorithms. I think you remember what P is.

But let me tell you a stronger notion of P, which comes up in parallel algorithms. There are many different models of parallel computing. So it's a little bit-- you know, it's hard to write down one good model of computation for parallel computing. But fortunately, if we're proving lower bounds, it doesn't matter whether our model is good. It really matters that our model is super powerful. If we can prove something is impossible in a super powerful model, that means it's impossible in weaker models too.

So I'm going to give you a super powerful model called NC. So NC is Nick's class, named after Nick Pippinger, who did a lot of parallel algorithms. And there are a few equivalent definitions. Let me give you two of them.

I would like my problem to be solvable. With P, we wanted things to be solvable in poly time. So a natural improvement is that I want to run in poly log time for parallel algorithms, given a huge number of processors, but not incredibly huge-- just polynomial number of processors. OK?

If I was given an exponential number of processors, I could solve SAT and all sorts of crazy things. So shouldn't go too extreme. But let's say polynomial-- not problems but processors.

Polynomial number of processors is reasonable, maybe a little overkill, but OK. With

polynomial number of processors, I'd like to get poly log time. That's a pretty natural, good situation.

I let these processors communicate however they want, super fast. You can think of it as a circuit. If you're thinking about the p-set, you'll be thinking about small weft circuits. But here, it's just good old-fashioned depth.

We want poly log depth. That will be the running time if every gate of the circuit can run in parallel. You still have to run in sequence, along the depth lines. But if there are a polynomial number of gates-- OK, that should be, of size.

Polynomial number of gates and poly log depth, then in particular, this gives you an algorithm. Each processor just waits for the inputs to be ready and then writes out the output to the next gate that it-- or to whatever gates it has that it's connected to by a wire. And you get an algorithm that runs in poly log time, polynomial number of processors. So whichever you think of is more intuitive. OK.

So I have a small example here. Suppose you want to sort n numbers in the comparison model. You can do that. And there are good algorithms to do it. I will give you a not so great algorithm. But at least it works in NC.

Namely, with n squared processors, I'm going to compare, in parallel, all pairs of items. I'm going to compare ai to aj for all i and j. Now I know which items are less than which.

And now, for each item i, I basically just want to compute how many items are smaller than it. If there are items smaller than it-- let's say all of the items are distinct-- then that item should go to the I-th position in the array.

So it's going to compute this matrix, i versus j. Each one says ai is smaller or greater than aj. And now, for a given-- what do I say, a given row-- I want to compute how many less than signs are in that row. This is basically computing a sum where I view less than as 1 and greater than as 0.

And you can compute a sum of n items in log n time, just by building a binary tree.

I'm going to add these two guys in parallel with these two guys, and so on. And then I compute my sum.

And I can do each of these rows in parallel. So in log n time, I can compute all of these sums. Then I know where every item needs to go and put it there.

So this is a little overkill, that I use a ton of processors. But still polynomial. So in n square processors, I can easily, in log n time, sort n numbers.

There are more efficient algorithms. But the point is when you have a par-- when there is a parallel algorithm, in this powerful model, you can do it without too much effort. But what we're going to show, here, is something called P-hardness, which, assuming not all problems can be solved in this-- not all polynomial time algorithms can be solved in this model, these problems can't. Just like NP hardness.

Note, in particular, NC is contained in P. If I had such an algorithm, I could simulate this algorithm. Because I have parallelism only polynomial, I could simulate the whole algorithm in polynomial time, still. So this is stronger than being in P.

And P hardness means that all problems in NC can be reduced in the usual Karp-style reduction, in P reduction. But now the reduction must be a parallel time reduction. So it must be an NC algorithm to your problem.

So P-hard means you're as hard as all problems in P. And reductions must be parallel algorithms. They must in NC.

And so this implies that you're not in NC, assuming NC does not equal P, which is another standard complexity theoretic assumption. Makes sense-- not all problems should be parallelizable.

So in particular, P-hard problems would be such problems. And P-complete, of course, means you're in P and P-hard. OK.

AUDIENCE: Shouldn't it be that all problems in P can be reduced by an NC algorithm?

PROFESSOR: Yes. Thank you. All problems in P. Yeah. Cool.

So all problems, presumably including the ones that are hard to parallelize-- not just the NC ones-- can be reduced to your problem. And so that means your problem is probably in P minus NC. Good.

So let me give you some base P-complete problems, starting with a very natural one-- general machine simulation, sequential machine. OK?

So given a sequential algorithm, such as a Turing machine, run it for t steps. Now, this is a little bit subtle. Let me write this down. And then I'm going to modify it.

Does the algorithm, say, halt within t steps? Or does it keep running for longer than t? So this is a version of the halting problem.

I mean, we-- same thing with a nondeterministic Turing machine. Running for polynomial time was our prototype for NP problems. We've done this kind of problem a lot.

We did a similar thing with W1. It was a run for k steps. And k was a parameter.

I want this problem to be in P. But if I phrase it this way, it would be X time complete because t could represent an exponentially large value. So what I want is for this to be encoded in unary, which usually, we're not supposed to do.

But here, I'm going to explicitly say, look, t is bounded by a polynomial in the size of the machine-- or equivalently, it's written in unary. And now I can do this in polynomial time. Because I have t steps to do it, I'll just run the machine t steps. OK?

So this is in P. And if you believe anything is not parallelizable, then this shouldn't be because this lets you simulate all things. So pretty easy to reduce all problems in P. If they're in P, that means there's an algorithm to solve them, sequential algorithm.

So you plug it in here. It's going to run for only polynomial time. So you can set t to that polynomial. And boom. You know, the algorithm halting is it's answering, yes.

Or you could change this to returning s or whatever. OK? So that's actually how you

prove this problem is P-hard, or actually P-complete.

AUDIENCE: Is there any problem with symmetry between yea and no answers, here?

PROFESSOR: Here, yes and no are the same. Yeah. Unlike NP. NP had an asymmetry between yes and no. So I could also change this to return no. That would also be as hard. Yeah.

But of course, this is not a very useful problem. It's just a starting point. And it's a very natural P-complete problem. More useful for reductions are these circuit value problems, CVP.

This is going to be like circuit set. So with circuit set, we were given an acyclic Boolean circuit which had one output. Or maybe we're given a particular output we care about. And we wanted to know whether there was some input setting that made the output true. Here, we are given the input setting.

So I mean, the circuit actually has the inputs written down, like 0, 1, 1. And then let's say we take an AND, and then we take an OR, something like that. And we want to know, what is this thing at the output?

Of course, this can be solved in polynomial time. You compute this AND, and you get a 0. This was a 1. So you compute this OR, and you get 1. So the answer is 1. OK?

And you can do that in linear time. But to do it parallel is really hard because it's really hard to predict what these inputs are going to be. So you can't do anything on this gate until the inputs are ready.

So presumably, this circuit will not have poly log depth. Otherwise, it isn't in NC. But if you have a really large debt circuit, simulating-- running it in parallel is just as hard as this problem. So you can reduce a general machine simulation to a circuit value. And the standard way of converting machines into circuits, that's the electrical engineering problem. OK.

What was the question? Is the output true? OK.

Now, there are many special cases of this that are also hard. For example, NAM CVP. OK.

We know how to build computers out of NAND. We know how to build computers out of NOR. Cool. So that's easy.

More interesting is monotone CVP. So in monotone CVP, you just have AND and OR gates. You don't have negation or an AND or an OR or an XOR. Those sorts of things.

This is also hard, essentially. By dual-rail logic, representing the true and the false as separate wires and manipulating them, you can get this to be hard. Also kind of similar to constraint logic. OK? But I won't prove it here.

A little stronger is alternating monotone CVP. This alternates AND OR down any path. So if you take a path from an input bit to the output, it alternates AND OR, AND OR, AND OR. And furthermore, we can assume that it starts and ends with OR.

It's nice to nail this down. It's not hard to prove. But I will prove it in a moment, once I tell you a general enough version that I want to analyze. This will just let us worry about fewer gadgets.

So up here, we had to think about all Boolean gates. Now we just have to think about AND and OR. And we can always assume the output of an AND is the input of an OR, and vice versa, and that we only need to connect inputs to OR gates and not like this picture, and we only have to get the output from an OR gate. So it reduces the number of cases we need to worry about. Yeah?

AUDIENCE: So is this with bounded fanin?

PROFESSOR: Good question. Basically, bounded fanin doesn't matter with this model. So usually, when NC is defined, it's bounded fanin and fanout. And there's another class, called AC, which is when you have unbounded fanouts and fanin, let's say.

But that only hurts us by a log factor. So it matters if you're worried about something called AC0 and NC0, when this is 0 and you want constant height.

AUDIENCE: It matters for alternation, as well.

PROFESSOR: Please wait, I guess. It won't matter, is the answer.

AUDIENCE: Is this the only setting of starting and ending that works?

PROFESSOR: Sorry?

AUDIENCE: Starting and ending with both AND or starting with AND?

PROFESSOR: Oh, yeah. I think you could also make this an AND, although I haven't checked that. I don't think it's critical that this is OR. But you could probably do one of them in OR, one of them in AND-- all of those combinations.

But this is the one I've seen proved. We'd need to check the other cases. But I think it's not hard.

This is called AM2CVP. Great acronyms. Every gate has fanin2 and fanout2. So in particular, bounded but also nice and uniform.

Every single-- not less than or equal to 2. This is exactly 2. OK? We'll prove that.

And even stronger, synchronous AM2CVP. I prefer AVP, personally. So this is SAM2CVP.

Synchronous means that all of the inputs of the gate are available at exactly the same time. If you imagine running all of the gates of depth i at time i, then you can do that. At time i, all gates of depth i will be ready. They'll have all of their inputs.

In other words, a gate of depth i has one of its inputs at depth i, minus 1. But in the synchronous circuit, both of the inputs will have depth i, minus 1. And so at time i, you'll actually be ready to do it. This is useful, in particular, for things like constraint logic and other constructions, as well.

One more. Planar CVP. I don't know how many of these things you can import into the planar case. Probably many of them.

But just like planar circuit set, out of NAND and NOR, for example, you can build crossovers. And so planar circuits are also hard. But actually, not all of them because planar monotone CVP is in NC.

So you can't take the monotone AND OR- ness with planarity, which is kind of annoying because that problem can be done in parallel. But planar with an AND-- then you've got negation. And you can build a crossover. OK.

Let me prove some of these things. This was to give you a flavor for what's hard. Still, all of these problems up are P-complete.

And we're going to do it in a series of reductions. So first, let's deal with starting and ending with OR. So this is the part you'd switch. If you want to start or end with an AND, it should be no problem.

So this is an input bit. Either 0 or 1, they're going to be drawn with squares. Oh, I should mention, by the way, there's a textbook devoted to P completeness.

It's a short textbook. But it's a nice read. And it's online. So I linked to it from the website now. So if you want to learn a little more, check out that book. These are figures from that book.

So if I have some input being sent to some gates, I'm just going to put an OR there. And let's say i OR that bit with itself. So now everything starts with an OR. Done.

If I want to make things end with an OR and they end with an AND, I'll just put an OR gate. I don't know why here we use only one input, here we put two inputs. But I could have just put one input up there. I'm allowing, for now, one input OR gate. So I'm going to fix that later. Question?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Coming soon. So this is just to start and end with ORs. Next thing I'd like to do is

make things alternating. So if I have two gates-- actually, before I do this, I need to do this later. It's important to get the sequence right.

First, I want to make sure the fanout is less than or equal to 2, in this way. This is a standard trick. If I have large fanout, I just build a binary tree. And you can put ORs or ANDs here. There's only one input. So it doesn't do anything to it.

But now I have fanout, at most, 2. Fanin is whatever. Did we deal with large fanin? Not yet.

So I'm going to do that first. This will make lots of ORs next to each other. But then whenever I have two ORs or two ANDs next to each other, I just add the opposite gate, in between. OK.

So now it's alternating fanout, at most, 2. So we've gotten this property in a weak sense, with less than or equal to 2. We're going to fix the 1s later. We haven't dealt with this yet.

I assumed-- sorry-- that I'm starting with monotones. So I just have ANDs and ORs. No indication. I'm reducing from monotone CVP to these other problems. I've got alternation, starting and ending with or and fanout, at most, 2, at this point.

Next reduction is to make fanin exactly 2. So yeah. I guess I didn't write it. But we can use the same trick on the other side. If you have fanin larger than 2, then you can just take ANDs and ANDs and ANDs. So you can also get fanin, at most, 2.

Now if I have fanin 1, which I've used all over the place, now I'm going to get rid of it in two different ways. If I have an OR gate with one input that happens to be an AND or original bit, I'll just add in another input of 0. That won't affect things.

And if I have an AND bit, I would like to put a 1 here. But I'm not allowed to feed a number-- an input-- directly to an AND gate because I want to preserve the property that I start and end with ORs. So this is one thing that would be-- you'd have to tweak a little bit if you want to start and end with ANDs. But you can do, I think, a similar trick.

I want to build the 1 bit using an OR gate. So I take two one bits, OR them together. Now I have a 1 bit fed into here. And then that will just pass through. OK?

So now I have fanin, exactly 2. No 1s. You can imagine this is useful when you're actually building gadgets because then you don't have to do all of the work.

AUDIENCE: [INAUDIBLE]

PROFESSOR: I haven't dealt with fanout 2, yet. At this point, just fanin 2. So variables-- some of them still only have fanout 1. But they will have at most, 2.

So now we're going to do fanout, exactly 2. So what I'm going to do is take my circuit, make two copies of it. There's the primed copy and the original copy. OK?

So now I just need to hook things how I want them. The inputs, which are not drawn here-- if I have an input out degree 1, then I'll merge them into one copy. Then I'll have out degree 2. If they already have out degree 2, just leave them as two copies. OK? So now all of the inputs have out degree 2, or fanout 2.

If I have an AND gate with out degree 2, again, I don't touch it. I'll leave them as two copies. If I have an AND gate with one-- a fanout of one-- then here are the two copies of it. I basically want to add an extra output that gets thrown away.

So because I have to end with an OR, I'm going to put it OR here. And then that's an output. It's not the output that we're asking the question about, where I want to know, does the output become true? That's a specific output.

So for this reduction I need to allow there to be multiple outputs in my circuit, but only one of them of interest. OK? And the rest is connected as before. So now these guys have fanout 2.

If I have an OR gate with one output, this is a little trickier because I want to end with ORs, I guess. We're going to combine these two bits. And I have to alternate. So next thing is I have to go to an AND.

This also has to have two outputs. So I'm going to put one of them here and one of them here. These guys need two inputs because I don't want to [INAUDIBLE] with the two input conditions. So I add another bit.

It could be 0, 1. It doesn't matter. Feed it to both of them. Now this has out degree 2. These have in degree 2. And this is an output. So output is-- you could think of there as being two of them. Yeah?

AUDIENCE: Can you chain the outputs together, so that there's a 1 output?

PROFESSOR: Oh, combine the different outputs together?

AUDIENCE: If you have fanout, exactly 2, and fanin, exactly 2, doesn't that mean that you're preserving the number of inputs, So that you can't reduce those extra outputs to 1?

PROFESSOR: Yes. Good. So I think if we tried to combine all of the outputs together, we'd end up with a similar problem. And we'd basically have to do this construction again, and then produce more outputs. Good. Yeah.

So we violate fanin 2, obviously, at the inputs. But you're right. After the input level, the number of lines should be preserved. OK. So never mind.

We do need multiple outputs. And exactly one of them is marked as the one we care about, whether an output's true. Is that the end? Yes.

At this point, we've proved AM2CVP is P-complete. OK. So one more. Reduction is for the synchronous case. This is cool.

Let me, maybe, first say what we're doing. So for the synchronous reduction, we're going to make n over 2 copies of the circuit, where n is the number of gates. At a high level, that's what we're doing. And basically, the i-th copy will feed into the i plus 1st copy.

Well, not quite, because we want to alternate ANDs and ORs. So in the i-th copy-- the i-th copy of the circuit will be the depth levels 2i and 2i plus 1, in the final circuit that I'm producing, where these are ANDs and inputs. And this level is going to be

ORs. OK?

So basically, I want to take an AND level and an OR level from one copy, then an AND level and an OR level from the next copy, and so on. The inputs are a little more subtle. But in particular, I'm at least going to make the copies of the inputs. And I'm going to change them later.

And so the outputs of the ORs from one copy will go to the inputs of the ANDs in the next copy, and vice versa. So the outputs from the ANDs stay within the same copy. And then the outputs from here go to the next copy.

Basically, this will force synchronization, in a sense. Now, the one tricky part is the inputs. I want the i-th copy to be triggered at time 2i, exactly.

And so for that, I can't just have the inputs. Maybe some of the gates in there take an input from-- sort of straight. I need to delay that input from coming and still preserve fanin and fanout 2 and alternation.

So to do that, I'm going to use this gadget. So I just have the same bit written twice. And then I OR them with themselves and then AND then with themselves and OR them. OK?

It seems innocent enough. That latter obviously will not change any of the bits. So just duplicating those bits at every time step. It's all about timing here.

And now here's the fun thing. If this gate needs this bit, obviously, I can just take it out, here. That will still preserve fanout 2, here. And if I'm taking one bit from here, I'm going to have to throw away another bit. And magically-- this is very cool-- if I throw away a bit here, it doesn't matter what that bit value is. This will still be X2.

You can think of the two cases-- either x2 is 0, in which case this is 0, and then it doesn't matter what this bit is. It will output 0. OK? Or this bit could be a 1. Then I go here. And I mean, it actually doesn't matter that it's preserved. But it is. I think it's kind of cool.

If this was a 1, then this will output whatever that bit is. But because, again, this will

be a 1 because it gets it from here, then the OR will turn it back into a 1. So either way, the bit is restored.

So you could actually reuse this gadget a few times if you wanted. But we can also be lazy and just make many copies of this gadget because we have polynomial size. So basically, we can get a 1 bit and also destroy an output, basically, with this kind of gadget, and get the inputs to be triggered exactly at the right time.

So the i-th copy-- all of the gates will trigger at time 2i. And then all of the AND gates will trigger at 2i. And then all of the OR gates will trigger at time 2i, plus 1.

And they'll feed into the next copy. All of those will trigger. And so on. So it's a little bit redundant. But it works.

And then the output is going to be the output of the last copy, n over second copy, I guess you'd call it. OK.

So you can, of course, convert this into bounded deterministic constraint logic, where each edge only flips once. Once you have the synchronous version of CVP, it's very easy. I didn't draw a synchronous picture here because it's tedious to draw these pictures.

But if you just set these-- whichever the 1 bits are-- as your initially active edges-- those are the ones that just flipped, just reversed, then things will just propagate and everything will be timed exactly right, whenever you arrive at a gate. Both of the inputs have just activated. And then the output will activate.

So just like before. But now each edge only has to reverse once. And you'll get the results at the end. So deciding whether the last edge flips is the same as whether the output was a 1 in the circuit.

Open problem, I guess, would be to use bounded deterministic constrain logic to prove interesting games P-complete or interesting problems P-complete. That hasn't been done. But I have, for fun, an example of the P completeness-- P hardness reduction.

So suppose you want to find an independence set. Independence set is hard. Suppose I want to find a maximal independence set, an independence set where I can't add anymore vertices.

So here's a way to do that. I start with nothing. I do a for loop. Let's say the vertices are numbered. It's going to be important.

And if I can add that vertex, add it. If V is not adjacent to S, add it. OK.

I mean, this whole world is weird because it's all about polynomial time problems. Here's a polynomial time algorithm. It produces what I would call the lexically first maximal independence set because it adds 1, if it can. And then, subject to that, it adds 2 if it can, and so on. I will always add one, I guess.

So with this labeling of vertices, it finds the lexically smallest one. Suppose you want to find the lexically smallest maximal independence set. finding a maximal independence set can be done in parallel. But finding the lexically smallest one cannot. This is P-complete.

And the reduction is super cute. Reduction from NOR CVP. Suppose I give you a NOR circuit, and I want to know whether the output becomes true. What I'm going to do is build a special vertex, called 0, which is connected to all of the 0 inputs.

So the circuit has 0 inputs and 1 inputs. I'm going to make the inputs into nodes. OK? And so there are some 1 nodes. Those are just going to be nodes. They connect to whatever the gates are. Then these things are combined with various gates, and so on.

I'm just going to replace a gate with a node. I'm going to replace an input with a node. And I'm going to add a special node, 0, which is numbered first, that connects to all of the 0 inputs.

And then I'm also going to order the vertices in a topological sort. So this will have the earliest number. Then these guys. And so on.

So any topological sort. Lemma topological sorting can be done in parallel. There's an NC algorithm for topological sorting. We need that here because this reduction has to be in NC.

But once you do that, I claim that V will be in S if, and only if, that gate outputs a 1. Proof by induction. So initially, this is going to be put in the independent set, which means these guys won't, but these guys will because they're lexically first-- or they're first in the topological order-- and then induction.

There's only one type of gate here. It's a NOR. So basically, this guy will be circled if, and only if, neither of the inputs are circles. So that's a NOR.

And so then the decision question is there's some last thing corresponding to the output. It's going to be circled if, and only if, the circuit outputs 1. And that's it. It's kind of a beautiful reduction.

And I just, for fun, I have a list of some more P-complete problems without proofs. Game of Life, obviously. When you're given a time bound that is written in unary, and I want to know, is this cell alive at time t, where t is written in unary, by the proof we did already, that's P-complete. And just, it's with the unary version. Also, one-dimensional cellular automata. Same kind of thing.

Generalize geography. Remember that PSPACE complete problem. But if you have an acyclic directed graph, and you want to solve geography on an acyclic directed graph-- so it's a two player game-- that can be done in polynomial time. But it's P-complete.

If I have a point set in the plane, and I compute the convex hull, remove those points, compute the convex hull, remove those points, it's called an onion peeling. And given a point, I want to know, is it on the k-th layer for some value k, that's P-complete. Basically, you have to sequentially compute the convex hulls.

Another good canonical one is multilist ranking. Suppose I give you k, say, sorted lists of numbers, and I want to know, given an item, what is its rank in the union of the lists? So items can appear in multiple lists. And I want to know, is it the kth item,

in sorted order, in the union? So basically, I want to sort the union. That's P-complete.

If I want to compute A mod B1, mod B2, mod B3, mod B4, n times-- so the repeated mod-- I want to know whether that equals 0, that's P-complete. If I wanted to-- linear programming, that's polynomial time. This is P-complete, even when the coefficients are 0 and 1. So this is called strongly P-complete. You don't need large numbers, whereas something like max flow is weakly P-complete and can be, in the analogy to full P test-- fully polynomial time approximation scheme-- there is a fully RNC approximation scheme. R means there's randomization. I think it's open, whether you can get rid of that.

There's some fun open problems, like deciding whether two numbers are relatively prime. It's conjectured to be P-complete. But we don't know. Computing A or the B mod C is conjectured to be P-complete.

Maximum matching is conjectured to be-- with large edge weights, it's conjectured to be P-complete. But there is a pseudo RNC algorithm. So if the weights are small, and you allow randomization, then it's parallelizable.

Another open problem is graph isomorphism with bounded degree, which is polynomial time. I didn't know that. But bounded degree graph isomorphism is easy in a serial setting. But a conjecture is it's hard in a parallel setting.

So lots of things out there. I don't see too many papers about P completeness these days. But it's a fun thing. Once you decide your problem is in P, next thing, next level, you can find hardness anywhere, it turns out.

Next thing is to prove P completeness for your problem, make sure there's no parallel algorithm, or find an NC algorithm. If you do the upper bound side, once you have NC, you want to get the smallest depth possible. That's another story. Cool. That's it for today.