[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: Greetings, everybody. Welcome to our last lecture of the term. We have survived a semester online in 18.404 and we are going to conclude our last topic today, which is interactive proof systems that we started last time. And with the big-- well, the big theorem of interactive proof systems is that IP equals PSPACE. And we're going to give the main idea for that in a slightly weaker theorem, as we'll see.

So why don't we jump in? So we have been doing interactive proofs. We gave an example of showing that the graph isomorphism problem, the complement of that is an IP, as I hope you remember. We had that interaction with the approver and a verifier. We're going to go through it quickly. Not that protocol, but just the setup. And then we're going to finish by showing that this number SAT problem is an IP and should conclude that coNP is a subset of IP.

All right, so let's go for it. Yes. So just remember, interactive proof systems, there are these two parties, the prover and the verifier. The prover has unlimited computational ability. I kind of model that as an army of students perhaps who can-- where we don't-- they can work all night. They can use computational resources. And the prover, however, we're not going to measure the computational power of the prover. That's unlimited. And so the prover can do things like find certificates. It can test whether things are satisfiable. It can factor numbers. We don't care. It can do whatever we'd like and there is no charge for the prover's computational demands.

OK. So the setup we had was the prover and the verifier. Both see the input. The exchange of polynomial number of messages. And then the verifier accepts or rejects. And we had this notion of the probability that the verifier ends up accepting when paired with a particular prover. And what we want is that for strings in a language, that probability should be high for some prover.

And for strings not in the language, that probability should be low no matter what the prover does. So there's nothing the prover can do. And the way it kind of suggests that at any prover. But whatever the prover's strategy cannot make the verifier accept with high probability. Just doesn't have enough information or it doesn't-- it's just not able to make the verifier accept with high probability.

You might think of the prover as trying to make the verifier accept. So the P tilde is a crooked prover. I don't think that went down very well with everybody. So I have it here. Another way of looking at it, maybe it looks a little bit more like NP here where IP is the collection of languages where there's a verifier, just like we had. You can think of NP as having a verifier which can check certificates. Here the prover is going to be like the certificate so that for strings in the language, there's a prover which can interact with the verifier and make it accept a high probability. And you're not in the language, there is no prover, which can interact with the verifier and make the verifier accept with even more than low probability.

What's important is this gap, just like with BPP, between acceptance or rejection. And that gap is there because we want to be able to use the amplification lemma. And if there was no gap, then you wouldn't be able to amplify and make the probability of acceptance extremely high when you want it to be in the language, when you're in the language, and extremely low when you're not in the language.

OK. So I hope that refreshes your memory as to how that works. We're going to walk ourselves through the-- well, through what we did last time. But let's set the stage for that. So the surprising theorem, as I mentioned, is that IP equals PSPACE. One direction of that is a fairly standard simulation. With PSPACE, you can basically work your way through the tree of possibilities for an interactive proof protocol. And you can calculate the probability that the verifier would end up accepting if you had the best possible prover that would try to make the verifier accept. And you can just do that calculation. It's in the book. You're not going to be responsible for knowing that, actually. We haven't covered it in lecture. But it's not very hard. A little technical, I suppose.

The other direction is the interesting one, and that's the direction we're going to be moving toward today. We won't quite get there, but the way it works is that to show that everything in PSPACE, which is kind of amazing, is contained with an IP. So everything in PSPACE can be done with an interactive proof system. And the way that is done is by using a PSPACE complete problem, TQBF, and showing that that problem itself is an IP. But we're not going to prove that. That would be sort of the next thing we would prove if we had a little bit more time. But we're going to be satisfied with just the somewhat weaker but very similar statement that coNP is contained in IP here.

Again, still very surprising, because you have to be able to show, for example, that a formula is not satisfiable with a prover. How can a prover convince a verifier that a formula is not satisfiable? Showing that it is satisfiable, you just give the certificate, which is the satisfying assignment. But how do you show something's not satisfiable? It's unexpected. And the proof of that is pretty much similar, slightly is one kind of technical point which we don't have to get into. So it's slightly easier but very much in the same spirit.

So remember this number set problem is you're given a formula and a number, and that number is supposed to be exactly the number of satisfying assignments of the formula. So in particular, a formula's unsatisfiable, then it would be paired with the number 0. And that's why the number set problem is coNP-hard, because you can easily reduce the unsatisfiability to number set. An unsatisfiability is coNP complete.

OK, so remember we introduced this notation last time. This is going to be critical for understanding this proof. So let's go through it once again. So if you have some formula, what I'd like to do is preset some of the variables of that formula. So that's going to be a formula on m variables x1 to xm. And I'd like to preset the first i variables to zeros or ones as I wish.

So I'm going to indicate that by phi with 0 means I'm setting x1 to 0 and the rest of the variables remain variables. And more generally, phi of i values a1 to ai, which to start off with are going to be just zeros and ones, just Boolean values. That's going to be the formula with those first x1 to set to a1 dot, dot, dot xi set to ai for those i constants, which were zeros and ones. I'm going to call those presets, because we're presetting some of the variables in the formula. And the rest of the variables we're going to leave as variables.

So we get a new formula on fewer variables by doing this pre-setting process. And we're going to get to do the same thing in terms of counting the number of satisfying assignments. So remember the notation number phi is the number of satisfying assignments. Number phi with a preset of 0 is the number of satisfying assignments when you've set x1 to 0. And no phi of a1 to ai is where you set the first i variables to those i values. And then you're going to look at the number of satisfying assignments with those presets in mind.

So there were two facts. I'm going to call them identities, because we're going to rely on those and we're going to actually extend those to the non Boolean case, as we'll see shortly. So these two identities say that, first of all, if I preset, I think understanding the first one is clear just by thinking about it in the case where i equals 0. So this is the case where the number of satisfying assignments altogether is the number of satisfying assignments when I've set x1 to 0 plus the number of satisfying assignments when I've set x1 to 1. And this just generalizes that when I look at having already preset the first i variables.

So if I preset the first i variables to these i values, the number of satisfying assignments I get there is the number of satisfying assignments I get with those presets plus the next variable being set either to 0 or to 1. And then you add those up. The same idea. And lastly, if I set all of the variables to values, so I have no variables left, and I look at the number of satisfying assignments consistent with that fully set variables, so there's no variables left, everything is set, everything is preset, that's just whether or not those values have satisfied the formula already or not. So this is going to be equal to 0 or 1, the number of consistent satisfying assignments with those m presets where m is a number of variables is just whether those m values satisfy the formula, in which case, I get 1, or they don't satisfy the formula, in which case, I get a 0.

Critical to understand these in the Boolean case, because we're going to generalize this to the non Boolean case, and it's going to be just more abstract. The formulas are going to look the same. We're going to have to kind of-- we're going to lose the intuition that those things correspond to satisfying assignments. Or counting the number of satisfying assignments.

All right. So let's have a quick check-in here. So we're just going to do an example to hope to nail this in, this idea. So here's a particular formula phi. And now remember, number phi is the number of satisfying assignments. So phi, the number of satisfying assignments where I've set x1 to 0 and so on. And here I'm really kind of giving you two options in each row for the value. Now you have to check all that are true. So it's really going to be at most one per row, presumably.

All right. Let's see if you're with me here. So the number of satisfying assignments for altogether, well, there are two ways of satisfying this formula. This is really like exclusive or. So either x1 is 1, x2 is 0, or x1 is 0 and x2 is 1. So one of the variables has to be true. The other one has to be false. And then you're going to end up satisfying both clauses, as you can easily see. So b is correct in the first line.

Now, if I'm going to already commit to saying the first variable is set to 0, now how many satisfying assignments can there be? Well, the second variable just has to be set to 1 in order to satisfy. So now there's going to be only one satisfying assignment consistent with setting the first variable to 0.

Now if I set both variables to 0, now how many satisfying assignments can there be consistent with that assignment? There can be 0, because in order to satisfy this formula, one of the variables has to be 0. The other one has to be 1. If I'm presenting them both to 0, there's not going to be any satisfying assignments, because 0, 0 not satisfy the formula. OK, apologies for messing up that check in on the last day. Oh well.

All right. Let's first go over the protocol we attempted for number SAT last week on Thursday. So we're given the input, the formula, and a k. And remember what we want to happen. We want the verifier to end up accepting with high probability when k is the correct value and with low probability when k is not the correct value. Now, this is going to be, as you may remember from last time, this is going to end up being a flawed protocol, because it's exponential. We're only allowed to have a polynomial size protocol.

But just looking ahead in this protocol, the verifier is going to end up accepting with probability 1 for an honest prover and with probability 0 no matter what the prover tries to do. So for any prover, the verifier cannot be made to accept. So this is kind of an extreme case where there's not going to end up being any probabilities. But it's an exponential protocol. So in that sense, it doesn't do what we need. So let's go through it, because it really sets us up for the polynomial protocol with the non Boolean values.

All right. So first the prover sends-- let's just look at it and not rush it. The prover sends the number of satisfying assignments according to the prover. The verifier checks that is equal to k. And I think it's best to understand this first with the case that the input is in the language. So k is correct and we have an honest prover. And then we'll understand what happens if k is not in the language. And we'll see that no matter what the prover tries to do, the verifier is going to end up not accepting.

And again, this is just a setup for the real protocol. So this is kind of a dopey protocol. You're going to think, what in the world, why am I doing this? It seems like I'm making something that's very simple complicated, but it's really just the framework that I'm putting together. Because, well, you'll see. All right. So the proof is going to send the claim for the number of satisfying assignments, which in the honest case is going to be the correct value. The verifier checks that it matches the input.

Now the verifier says, well, I want to be convinced that your claim is correct. So the prover is going to justify that claim by saying, well, the total number of satisfying assignments is whatever it is, 100 because the number when I have x1 set to 0 is 60. And the number when I have x1 set to 1 is 40. And that adds up to 100, which is what you would need to have happen. So the verifier checks that the sum is correct and then says, well, now how do I know those two values are right?

So then the prover unpacks it one level further. So breaks those two down by justifying that phi 0 was correct, that value 60 was correct, by saying, well, now if I set the next variable, x2 to 0 and 1, that's going to have to add up to phi 0. So maybe to get 60, I had 50 and 10. And to get 40 for number phi of one, I had 20 and 20. So these I have to add up. So each level justifies the preceding level. We're going to have that happen again.

Now, the prover says, well, I mean, I need to be convinced. I don't trust you. I need to be convinced that these values are correct. So level by level, the prover is going to be setting more and more of the variables in all the possible ways until it gets down to the very bottom where it's setting the variables in all possible ways. So exponentially many settings here. And the verifier now checks that the previous round was correct. So that's where we set only the first m minus 1, the very last variable hadn't yet been set. So checks all of those 2 to the n minus 1 possible settings in terms of the new settings that we got where we set those m minus 1 settings, but we extended it by 0 and by 1. Again, this is the same identity that we used from before.

And now that the prover has sent all of those possible values, the verifier needs to be sure that those are still correct. But the thing is that at this point, those are all zeros and ones because they all say whether that assignment satisfies the formula or doesn't satisfy the formula. So the verifier can check those directly. Checks each of those, whether just by plugging into the formula and seeing does it satisfy the formula or not.

So each one of these is a 0, 1 value, which is supposed to correspond to whether the formula was satisfied or not. Those all are correct and everything else along the way has been correct. The verifier is going to accept. Otherwise if at any point one of those checks failed, the verifier has already rejected or at this point it just rejects. So that is the protocol, the exponential protocol.

And I'm not sure if this is helpful to you or not, but I like to think of it sort of as a tree of possibilities. So these yellow values are what the prover is sending. So the prover first sends the number of satisfying assignments all together. The verifier in white is checking-- are doing these checks. So it checks that it equals k. And then the prover sends the next level. The verifier checks that the addition works out. Then the prover unpacks it further, assigns values to the first two variables, and the verifier checks that just the assignments, just a single variable are consistent with that and so on. And to assign all m variables and then it checks directly with the formula.

Now, what happens-- and here is the case. It's going to be important to understand in both here and in the non Boolean case. What happens if we had an incorrect value for the input? And what I want to show you is that the prover is going to-- I want to show you that the verifier is going to end up rejecting in this case with certainty. Later on it's just going to reject with high probability. But for this protocol, it's going to accept with certainty. And why is that?

Because first of all, if the prover, if k was wrong, so I'm indicating the wrong values in red. If k was wrong, so it did not equal the number of satisfying assignments, if the prover sends the correct value, the verifier is just going to say it doesn't match up. I reject right away. So what can the prover possibly do to prevent the verifier from accepting? You're going to see that there's nothing you can do. But later on, there's a chance that the prover can get lucky. But here there's nothing you can do. Let's try to humor me and see-- let the prover try to manage to keep the verifier going as long as possible.

So the prover in order to prevent the verifier from rejecting at the beginning would have to lie about the number of satisfying assignments. But then the prover is going to say, well, OK, you're claiming there's only 99 satisfying assignments. Prover doesn't know what the right real answer is. But we know it was 100, let's say. But let's say k was equal to 99. The prover claimed it's 99 now. And so the verifier says, OK, well, it's 99. Convince me of that.

So now the prover is going to have to say the number of satisfying assignments for 0 and the number of satisfying assignments for 1, they have to add up. At least one of those has to be wrong, because you can't have the two correct values adding up to the false value. So a lie here has to yield a lie in at least one of those two places. And then a lie there is going to have to yield a lie in one of those two places, just like each lie kind of forces more lies. As you know, you're trying to lie. The story gets more and more complicated in order to try to justify all this.

And so in the end, you're going to get an inequality. And the verifier is going to end up rejecting. Somewhere along the line, there's going to have to be an inequality, if not along the way then at the very end when the verifier does the check itself. Because one of those, you could trace that down, there's going to be lies and lies and lies and then there's going to be at the very bottom one of these values is going to be wrong. And when the verifier checks them all, it's going to find out that there is an inequality there. And so one of those checks is going to fail.

So I'm getting one question here. Why is this any better than just checking all possible assignments without a prover? It isn't. The only reason I'm doing this is to get us ready for the arithmetized protocol where we have non Boolean values coming in. So questions on this? I think it's important to understand this one. Don't ask the question why. The why is just going to be we are getting ourselves ready for something later, which you don't know yet. But I want you to understand it for what it is, even if it seems unnecessarily complicated.

OK, so let's keep going. So how are we going to fix that protocol so it's not exponential? So again, here is a picture of that exponential protocol. And we have that exponential blow up occurring because at every stage, each value is going to be justified in terms of two values at the next stage. So it's going to be exponentially many values after a while. So instead, we're going to try to justify each value here in terms of just a single value at the next stage.

But it's not going to be good enough just to pick either the 0 or the 1 at random. Because it might be each-- there might be just a single course of lies going through here. And the only way you would be to catch that would be to guess correctly at each stage which was the lie. And then you would catch it at the end. If you're just going to be randomly picking zeros and ones, you're not going to have a high probability of catching the prover when it's lying.

And so that's not going to be good enough. The input might be the wrong value and you might have a prover which just has one path of lies, and then your probability, you would still have a high probability of accepting in that case, even though the input was wrong. It's not what you want. When the input is wrong, you have to have only a tiny chance, a very small chance of accepting. So the way we're going to achieve that is by having these-- instead of picking a 0 or a 1 for these random values, we're going to have non Boolean assignments to the variables. And we have to make sense of that. And we've already seen an example of that. It's going to be very much the same.

All right. Are we all together here? So this is a place where we could try, if you have a question, we can try to answer that. Are we good? Let's keep moving.

OK, so how are we going to arithmetize Boolean formulas? It's, again, the same idea we had before. Simulating ands and ors with plus and times. So we had this from before, same exact picture. Actually it's even simpler, because now we're going to be using the true simulation of or instead of some kind of a special case simulation of or, which we had in the branching program case. So these faithfully do what and and or does when you plug in 0 for false and 1 for true.

So that means that we can take an entire formula and arithmetize it. The formula built out of ands and ors and negations. And you're going to get a polynomial that comes out. And that polynomial, what's going to be important for us is not going to be of extremely high degree. The actual degree is going to be at most the length of the formula in terms of the number of symbols it has. You can check that on your own. But for now you can just trust me. The degree of the polynomial, because it only goes up during the multiplications, but the degree doesn't become too big.

And we're going to be doing-- and I don't want this to be a confusing issue here. We're going to be doing-- but we have to be correct. I don't want to be cheating here. So all of the arithmetic is going to be done in a field. So we have to do plus and times mod some number, which turns out needs to be a prime number for reasons I'm not going to get into. But it doesn't really matter. It's just modular arithmetic. And that's one thing that enables us to pick random values in a natural way, because there's only finitely many values in the field. And so you're just going to pick one at random.

But here we want to be able to represent-- it's going to be more important for us to have a larger field, because we want to be able to represent the number of satisfying assignments which can be a number between 0 and 2 to the m. So we have to have a field which has at least 2 to the m elements in it so that we can in a sensible way write down those numbers. Let's not get caught up with that. But we can try to answer those questions offline if you want. But just think about it for this first pass. We're doing the arithmetic mod sum prime.

So now we have the same notion of presets as we had before. So if we have a formula and we preset some of the values but now those values may be non Boolean values. We may be plugging in values for the formula. Not just zeros and ones, but we might be plugging in sevens or 23's or whatever. And the formula is going to in order to have a value, a meaning to that, we're going to treat that formula as the polynomial from the arithmetization.

And just plug in those values into the polynomial and see what the polynomial does for you. So here we're going to be presetting some of the values of the formula like we did before. And now it's going to be the same thing. But now in the polynomial, we're going to be pre-assigning some of the values of the variables to these a's from the field. And the remaining variables are going to stay as unset.

Now we have to give an interpretation. So the new polynomial here. So I'm getting a question. Well, maybe I better take this. Let me hold off on that for now what the degree is. I'll answer the questions in a second.

So now remember from before, number phi was the number of satisfying assignments when I preset the first i values. It no longer makes sense to talk about satisfying assignments, because these values may no longer be Booleans. So I'm going to have to write this formally as I'm going to plug in those values, those i values, for the first i variables. And the remaining are variables which I have not set. I'm going to assign them to zeros and ones in all possible ways. Only to zeros and ones.

Because what I want to have, you might think, well, why aren't we assigning these to other values in the field? Well, because what I'm aiming at is that if I were to plug in zeros and ones at this point into the polynomial, I'm supposed to get exactly the same values as I had before, because I'm simulating and's and or's. So I'm just extending the definition, the evaluation into a new realm. But I shouldn't change the values on the old Boolean realm.

So I'm going to be adding up the unassigned, the unset variables in all possible Boolean ways. And the first i values could be non Boolean values. So you have to just accept this as an abstract notion. No longer has an interpretation as satisfying assignments.

So as I said, what's important is that if I happen to put Boolean values in now, then phi and number phi give the same values as they would have before. Because the polynomial acts identically to the formula on Boolean values. OK. So this is what I'm repeating what I said.

And there's another point that also you have to check, which is that the identities that we had earlier that connected up what happens when I set the first i values and I set the first i plus 1 values, those still hold. So if I set the first i values now to possibly some non Boolean assignment, that's what I get when I extend those values to one more variable being assigned. But I just need to assign that variable to 0 and to 1 and add those up because of the way I've defined things over here. So I've assigned those variables to zeros-- the unset variable to zeros and ones when I'm defining the number phi function.

And then lastly, when I assign everything now to possibly non Boolean values, that's going to be-- there's no longer anything to add up. So I'm going to get exactly the same as I got from before when I-- so assigning number phi of totally preset input, it's the same as phi with a totally preset input. Because in that case, there are no variables left to add up over. So there's just one. I just get one single. I sum it as just one element in it.

So I got a question here for earlier. What happens to the degrees of the polynomials? Well, the degree of number phi is going to be at most the degree of phi, because I'm just adding things up. And addition doesn't change degrees. As I preset values, the number of variables goes down, but the degree may not necessarily go down. So the question was I got are the new polynomials having lower degrees? Not necessarily. They have fewer variables but not a smaller degree.

So let's do this check. Let's see if that-- now again, this is I think I have messed up on this. Well, there's one of these that's-- I'll give it away in part. There's only one of them that was true anyway. So you can check the one that's true according to the way we've defined it. So this is a little bit of a trick question here, as I'll explain. But there's only one of them that faithfully does the arithmetization as I described on this page. And that's the one you should check.

So remember, over here this is the formula. This is the recipe for how I'm doing the arithmetization. This whole process here. So one of these lines, one of these, a, b, or c, corresponds to doing that. I'm going to close this down. So are we all in? Yeah.

So a is the correct answer. A does the arithmetization according to the recipe that I just described. Because if you look at x1 or x2, we can just check it in the very first part of the polynomial. x1 or x2. Well, it's x1 plus x2 minus the product x1 x2. So you can just see it right there. The others don't have that. And similarly for x1 bar and x2 bar. It becomes 1 minus x1, 1 minus x2, and then the product of those. So a is pretty straightforward as the arithmetization of phi.

Now, in fact, any of those would work. I don't want to confuse you here. But any of those would have worked, because they all agree on the Boolean assignment. And that's all that really matters. So if you have any-- all I care about is that they agree. The formula agrees with the polynomial and the Boolean cases, and these all happen to agree and zeros and ones. Doesn't matter though. I put those there just in case you tried it by just substitution of zeros and ones in. You might have picked the wrong thing.

OK. So let's take a break here, and then we will see about how to go about fixing the protocol after the break. All right. So also happy to take any questions. We haven't really done a whole lot. We basically, this has all been review of what we did last time. But let me start the timer. But feel free to ask questions.

I'll tell you where we're going. This whole proof really comes down to understanding one line, which is going to be in the second half. So I'm really kind of-- this is all big setup here to get you ready to be able to understand that one. I'll tell you when it's coming. So you won't have to worry that you'll miss it. But that line is not easy to understand. So I think it's important to get all of the framework and all of the context all set up for you so then you can understand that line and hopefully you see that line and understand it.

OK, so the important fact. So let's go back. You wanted to see the important fact. OK. So this is what I was saying before. If I plug in Boolean values into the arithmetization, I get the same exact thing as I would have if I applied the Boolean operations before I did the arithmetization. So plus and times in the arithmetization give a faithful simulation of and and or according to these little formulas. That's all I'm saying with this. And so if I plug in Boolean values for the a's I get exactly the same as I would have gotten before I did the arithmetization. Because the arithmetization is a faithful simulation. Not sure how else to say it. Let's see.

What does the or rule now-- why does the or rule now contain the minus ab term while the previous instance of arithmetization didn't? Remember in the case of branching programs, we didn't need the minus ab term over here. And that was because we could argue that it was a disjoint or in the case of the branching programs. I don't want to get confusing by trying to explain why that was. But in that earlier case, we never took an or of two ones.

It was an of 0, 0 or possibly 0, 1 or possibly 1, 0. So therefore we never had to deal with a case when we had an or of a 1, 1. And here we can have that. So we have to subtract off that ab term, because otherwise we'd have-- if we just had a plus b, then the 1, 1 case, we would end up with a 2. And that would not be a faithful simulation of the or operation, because 1 or 1 should be just 1, not 2.

So this is a good question here. Do all the numbers need to be zeros and ones? I'm not sure how negation would work with larger numbers. The negation, you just blindly follow it. Even though we're going to be plugging in non Boolean values, it's going to be 1 minus 7. So you're going to get minus 6. You have to do that mod P, mod Q, whatever that value you get. But you can no longer think about it as negation in the former sense.

Now it just becomes a formal thing. You're just plugging along doing what the polynomial says. Numbers are coming out. You think this is just nonsense. But the thing is it's going to have a meaning that's going to be useful to us. That's what this protocol is going to show. So you can't think about it as negation anymore. It's just negation becomes 1 minus x in the arithmetized world and you just have to live with that. Let's see.

Another question here. If all the phi are equivalent for Boolean inputs in the check in, so this is back into this check in here, so if all of the-- yeah. So the question is if they're all equivalent in the Boolean case, why is only a correct? Because I defined P sub phi in a particular way. And so this was the value you got if you follow the way I define P sub phi. The others would work, they just weren't the way I defined it.

Any other questions here? We should probably move on. Can arithmetization be used in other contexts? Offhand, I don't know. There are these two cases where arithmetization works. Whether there are other cases too, I'm actually not sure. OK, so let's move on. So our timer is up. The candle has burned down.

OK. So this was-- OK, here we go. This is the real protocol. So I'm going to present it to you the way I did before. Let's think about it with the case first where the input is in the language and we have an honest prover. So we start off the same way. The prover sends phi, sends number phi. Which in the old sense was the number of satisfying assignments.

It actually still is, because since we're not presetting anything, there's no non Booleans in the picture yet. So this is going to be the same value as before. The verifier checks that k equals number phi. So that's why we have to have a big enough field there, so that we can represent numbers up to the number of potential number of satisfying assignments. But that's a side note. But anyway, this is exactly what we did before. No change. The number of satisfying assignments if you like.

Now, let's just see. Let's remember. And this is one of those cases where not having a big blackboard hampers us. So I'm just going to remind you what we did last time. But I'm going to change this. So remember before P sent-- and unpacked at one level. Sent the number of satisfying assignments said number phi of 0 and number phi of 1. And then we did that check to justify the previous value, which the verifier doesn't necessarily trust.

OK. Fasten your seatbelts, everybody. This is the whole proof in the next line. But it's a doozy. All right. P is going to send phi of z as a polynomial in z. It's going to send just a single object. But that object is an entire polynomial. And the way it's going to send that is by sending the coefficients of that polynomial. So let's digest that statement.

So first of all, let's understand the value of doing that. So if I can send the entire polynomial phi sub z represented as a polynomial, I can plug in 0 and 1 into that polynomial and allow the verifier to do the check that it needs to do to demonstrate that number phi is correct. So it's going to check that number phi is number phi of 0 plus number phi of 1. But instead of getting those values directly from the prover, it's going to take that polynomial it got and evaluate that polynomial at 0 and 1.

And just to remember, let's go back and remember how we defined-- defined number phi to make sure that we understand what it means to have a polynomial here. So remember, here we're just taking the very first value. But you are OK with putting a constant 0 or 1 and then adding up over all possible extensions, all possible Boolean extensions to that. And maybe it's OK to put in a non Boolean value here, like 7. And then you take the remaining variables and assign them zeros and ones in all possibilities and add it up.

Now I'm going to do something even a little wilder. I'm going to put in a variable for a1. Some symbolic, if you want, symbolic value. So I'm going to put in a value z for a1. So now I plug in z for a1 here. And a2 through am are going to be zeros and ones in all possible ways. So I just get a polynomial in z. The other variables get assigned and added up over the various Boolean assignments. And now I get some polynomial. So I get some expression in z. That's just going to be a single variable polynomial. Whose degree is it going to be? At most the degree of number phi. So degree is not going to be too big. So it sends the coefficients so the degree of that is not too big. So there are not too many coefficients to send.

So the coefficients are in terms of the xi's. No. I'm not sure what the mean-- the coefficients are not in terms-- the xi's are gone at this point. The xi's, we've added up the xi's being assigned to zeros and ones in all possible ways. So there are no other variables left. There's only z. So I'm going to do the same protocol in a more pictorial way in a minute. So you're going to see this whole thing twice. But try to get it. You'll have two chances to get this. Try to get it. Try hard each time.

So I've got send phi of z as a polynomial in z. Now, that's going to be enough for me to figure out what number phi of 0 and number phi of 1 is, because I plug it in for 0 and 1 for z. But now I can figure out what number phi of 2 is also, because I can plug 2 in for z or number phi of 7. I plug 7 in for z.

So let's stop here and see are there other questions. So is the size of number phi-- I don't understand. This question about the size of no phi. Is it 2 to the m? No, it's not 2 to the m, because the degree of that polynomial, number phi of z, I mean, it's a very large expression if you want to initially-- yes, it's going to be an exponentially big sum. But the prover adds it all up for you, and you're just going to have at most a small number of coefficients, because the polynomial is only of a certain degree. And a polynomial in one variable of degree d has at most d or d plus 1 coefficients to worry about. So it's not that many coefficients as an expression.

So shouldn't the summation take 2 to the m time? I'm not caring about the prover's time. The prover has a lot of work to do. But the prover sends phi of z. So yes, the prover has an exponential job. I don't care. The verifier needs to be able to check it in polynomial time. And that checking is going to, well, we'll have to see. How does the verifier know that that polynomial is right? That's a question maybe you should be asking.

Yeah. I'm getting lots of questions about how much time the prover needs to take. Yeah, the prover is going to have to spend exponential time to figure out that polynomial. That's all right. We don't care about the prover's time. Yeah. So the summation here is going to be adding up polynomials. That is correct.

I'm happy to spend time, because really here this is the whole proof. You have to understand. Well, we have to understand why this works. But we kind of understand half of it, because knowing that polynomial is enough to-- if you could certify that that was the correct polynomial for number phi of z, then we can use that polynomial to confirm the previous value, what number phi was, because you just plug in zeros and ones for z, and you add it up.

But now how are we going to justify that the polynomial is correct? Because this looks like even a worse job. Now we have a whole bunch of coefficients and have to make sure all of those coefficients are right. And so instead of just two values, now we have d values where d is the degree of that polynomial, which could be at most the length of the formula.

So here is the next idea. So the prover needs to show that phi of z is correct. The way it's going to do that, so even before we do that, so phi of z is going to be some polynomial. Now, the prover may be lying, may be sending the wrong polynomial. How does the prover convince the verifier that the polynomial is the right polynomial?

Well, that seems like a tough job. So what it's going to do is remember that the-- so there is a correct polynomial that you would get by plugging in to this expression for the correct value. So there's some correct polynomial. The prover may be sending some incorrect polynomial. So now we have the correct polynomial and the possibly incorrect polynomial. And the point is those two can only agree in a small number of places by that fact we proved a couple of lectures back regarding polynomials. So two different polynomials can agree only rarely.

So what we're going to do, the way the prover is going to justify that this polynomial was the correct one, is by evaluating it at a random place and then demonstrating that that value you get is a correct value. If the polynomial was the wrong polynomial, then evaluating it at a random place is probably going to disagree with the correct polynomial at that place, because they can only agree rarely.

So the prover is going to demonstrate that by evaluating that polynomial at a random place, that value you get is going to be the correct value, and it's going to continue to do that in the way, using the same protocol, as we'll see. So that's where we're going. So in order to show that phi of z is correct, the verifier now gets to pick a random value in the field. And the prover is going to show that evaluating that polynomial at r1 is correct. Remember this looks a lot like what we had from before where we were showing that number phi of 0 is correct and number phi of 1 is correct. Now we're trying to show that number phi of r1, this random value from the field is correct.

So the way we're going to do that is now by unpacking it one level down. And we're going to be using that identity, because this value here is going to be equal to number phi of r1 comma 0 plus number phi of r1 comma 1. But we don't want to send both of those. So we're going to send them combined into a polynomial of number phi of r1 of z as a polynomial in z. This is a new polynomial in z.

So now if you understood the previous line, then hopefully this one won't be too hard to swallow. Because now we're going to check the identity, but here by evaluating the polynomial again but one level at the next level. So this is perhaps a good place to take questions, because this is the-- this is really what I spent all the time setting things up for so that you would be ready to get this thing hopefully without-- and hopefully be able to appreciate it and understand it. So I'm not getting questions. Let's move on a little further.

So now again, the prover had sent this polynomial in stage two. Now the verifier needs to be sure that that polynomial is correct. So it's going to evaluate that new polynomial at a random location. So by picking a random value r2 in the field. And now we need to show that this value is correct, because if that polynomial had been the wrong polynomial, it disagreed with the correct polynomial almost everywhere.

And by picking a random place, it's probably not going to be the right value and so on. Until we get to the end where we have almost all of the values have been picked, and so we have one last value to select a 0 and 1. This corresponds to the n-th. It would be great if I could put both pictures on your screen, but I can't. So this very much corresponds to what happened in the exponential protocol but just along sort of this arithmetization single path.

So it checks that the previous value is correct in terms of expanding it with 0 and 1. But again, the 0 and 1 comes from evaluating the polynomial. And now the verifier needs to be convinced that that polynomial was right. So it picks a random value, but now it doesn't rely on the prover anymore. It's going to see whether that assignment that it gets by evaluating the polynomial with that random value rn plugged in is the same as what I get by evaluating the polynomial for the formula itself that the verifier can do directly. Because this is now a polynomial now just plugging into the formula and using the arithmetization to get a value out. So this was the last line of the identity. We had those two identities. So this is the second identity. And we had to check that this is correct.

So I'm going to show this to you in a picture. Not sure it'll help if you're confused. But why don't we take some questions on this? So as I said, I'm going to give you two chances to understand this. Because I know it's tough. Especially with the constraints of Zoom, this is a particularly challenging idea to explain.

OK, so let's see. So the benefit of this approach is that the prover only sends one item for each depth level instead of multiple items. That's right. But that one item is the polynomial. So that captures all of the values for the entire field. But taking advantage of the arithmetization, that one polynomial has a lot of information in it. And what's nice is that you can check that polynomial by just evaluating it at one random place. You can check that that polynomial is correct.

So I'm getting another question here. Where does this come from here? V checks that this here. So this where does this-- so you have to look-- to understand where this is coming from, you have to-- we're at the n-th round now. So you have to look back like at round two. V has to check that phi of r1, which comes from the end of the first round. So this checks that this phi of r1 is correct because that was how we justified the polynomial with just a single variable. The very first polynomial was correct. A little hard to say. But this comes from the previous round, this guy here. So this is the polynomial for the current round. This is the value from the previous round. All right. More questions.

So why doesn't this run in exponential time? Another question I'm getting. Doesn't V need to check twice at each layer? Yes. The verifier needs to check-- gets two values, but those two values come from the one polynomial. So there's no blow up anymore. Those two values. Maybe you'll see it in the picture that I'm going to show. So maybe just hold that question. Maybe this will become clearer in the diagram.

So another question. Does this work because the polynomial kind of encodes all the possible values together? I think that's sort of true. It sort of mixes them all together into one object. Then you have to check that one object, which can be done with this sort of random probing of it.

So this is another good question that we'll see explained in the next slide. So similarly in attempt one, the prover can keep lying by picking polynomials by continuing to pick polynomials, by lying about the polynomials. But eventually it's going to get caught, because this value is going to be the wrong value. If the polynomial in the previous stage and the m minus-- if a polynomial that the prover sent in the m stage is the wrong polynomial, then you evaluate it, you're going to get the wrong value probably. And so then that wrong value is not going to match the correct value, which is you can read off yourself by reading the formula.

I think we need to move on to the next slide. All right. So same proof, version two, but looks different. Again, the input is that. Here is what the prover sends. Here is what the verifier sends. I'm going to sort of whimsically design this as a telephone chat where they're sending each other messages through messaging. So the prover sends the number phi to start off with. And then off on the side, these are the checks that the verifier is going to be doing.

So here in our first round of the chat, the prover is going to send phi of z. Remember this is just a polynomial in not too many coefficients. So it's a polynomial in one variable. The degree is small. So there are not too many coefficients here. So this is just pretending this is what it might look like. So from that polynomial, the verifier can plug in 0 and 1 and see that that adds up.

Now the verifier, to check that this polynomial is correct, it picks a random value to evaluate this polynomial on. And so now it's going to have to check that this is correct. So this is nothing to check. You're just writing this down in anticipation of the next check. Now, the prover to justify that this value is right, that this polynomial is right, so we evaluate-- the prover in order to check that this value is right is going to send the polynomial for the next level.

Now, we can from that, we can plug in 0 and 1 for z. See if that adds up. And now to be sure that this polynomial is right, we evaluate it at a random place, calculate that value, and then have to see that this value is correct. So now we expand to one level further. We take a polynomial for the next variable. And we see that adds up. OK, I'm not sure whether this is helping or not.

But we keep doing that until we get to the very last round with a prover sending a polynomial. Make sure that this adds up correctly. And the verifier to see that this polynomial is right picks a random value and evaluates it and now checks that this agrees with the formula. Because we've now assigned all of the variables. And then we can check this number phi directly in terms of the phi, because they have to agree. And so the verifier would accept if everything checks out.

Let's see what happens. So this answer will answer some questions. Why don't I walk through what happens if the input was wrong. And we'll see how the verifier is likely to catch the prover but not guaranteed to catch the prover in this case. So if k was correct, the verifier will accept with the honest prover. But if k was wrong, so I'm going to, again, indicate the wrong values in red. I want to show you that the verifier is almost certainly going to accept but not guaranteed. So did I say that wrong? So if k is wrong, the verifier is going to probably reject, but it's not guaranteed to reject.

So first of all, if the prover does not lie, does not send the wrong value for number phi, The verifier is certainly going to reject, because it's not going to get any quality there. So the prover has to lie. Say if k was 99 but the real value was 100, the prover if it says 100, the verifier's going to reject immediately. So the prover's going to say, well, let's see what the prover can do to make the verifier hopefully accept from the prover's standpoint.

So the prover is going to send 99. Well, the verifier says, OK, 99, fine. Convince me. So the prover-- now one of these two is going to be wrong. Because the two correct values can't add up to the wrong value. So one of these is wrong. So that means the prover had to send the wrong polynomial. Because the correct polynomial would evaluate the correct answers here. So the prover had to send the wrong polynomial. So now when we evaluate it at a random place, chances are this is going to be the wrong-- this is not going to be the same value that the correct polynomial would have given you.

The prover could get lucky. The verifier might have just happened to pick a place where the correct polynomial and the incorrect polynomial agree. In that place, the prover will think, huh, I'm saved. Now I can act like the honest prover from this point on and the verifier will never catch me. It's sort of a little bit analogous to the situation maybe-- I'm trying to see if you really studied the whole course. So I'm giving you an exam by picking sort of random places there. But maybe you just studied a few facts from the course.

You might get lucky. I might happen to ask just about those facts. And then you give the appearance of having studied everything, but you really didn't. So here the prover might send the wrong polynomial, but the verifier just queries that at the place where it happens to agree with the correct polynomial, and the prover just gets lucky. And the verifier is going to accept, in that case. But there are very few of those. So that's why the prover is almost certainly to be caught if it tries to lie. But not guaranteed.

So just tracing this down. If this was a lie, then one of those two has to be a lie. So therefore, the next polynomial has to be a lie. And so we continue. So then the next value is almost certainly going to be a lie. Not guaranteed. And so then one of those two values has to be a lie. At least one has to be a lie. Therefore, the polynomial has to be a lie and so on until-- unless the prover got lucky along the way somewhere, which is very unlikely, even though it has a several opportunities.

We've arranged it so that the chance of getting lucky is tiny at each stage. So even though he has a few chances, there's still going to be a tiny chance that you're going to get lucky somewhere. And so this is wrong, then chances are that's wrong. And so therefore, this is going to be a disagreement. And the verifier at that point when it doesn't agree is going to reject. Unless the prover got lucky somewhere along the way, which is unlikely.

So I don't know if you had-- so that's all I was going to say about this proof. I don't know if you had any questions on it, but let's just see. OK. So do we have any questions I can answer? How the prover gets-- how does a prover get number-- how does the prover get number phi of z?

So you have to-- why is number phi of z have no other variables? You have to go back and look at the definition of number phi of a. Because you add up over all the other variables. So now instead of a, we're plugging a variable for that. But you're still adding up over the other variables. So this is a function in just one variable, because it-- the original thing was a polynomial. This is also going to be polynomial.

I think we're starting to run low on time. So this is our very last check in for the semester here. So of course there's one natural question to ask you all. And for our very last check in, as we're in our last couple of minutes of the course or at least the lectures, does P equal NP? What do you think? Will maybe PB equal NPB solved by a deep learning algorithm? Or maybe we'll never prove it. Give me your best guess. We're kind of running out of time. So let's not think too hard here. Another five seconds.

All right. Ending polling. I'll share that with you. Oh, I did share. So what did we get? D here. We will prove it in somewhere between 20 and 100 years from now. That seems to be the majority opinion. I don't know. I hope it'll be sooner than that, because I'd like to see the answer. But we don't know. Yeah, if you can prove P different from NP, I'll give you an A+. You won't have to take the final. But you better be sure you're right.

All right. So that is our quick review. We finished number set in IP and therefore that coNP is a subset of IP. If you're interested in further pursuit of this material, I got a couple of questions on that. These are some courses you may want to look at. I know I checked with Ryan Williams. He's planning to teach Advanced Complexity fall 2021. So that's going to be the most natural follow-on subject to this one. There's the crypto classes also are kind of make use of some of the same ideas. And there's, of course, also randomness computation that Ronitt Rubinfeld teaches. If I didn't check with her. I'm not sure the next time she's going to be teaching that.

And good luck on the final and best wishes. And I'm going to have office hours. So if you have any questions, happy to answer those. But otherwise, see you all. Good luck. Thank you for the comments. Yeah, I enjoyed having you all as students. It was a fun time. A lot of work, but it was a fun time.

I've always been intrigued by the P versus NP problem, and I proved a kind of a-- I proved the exponential complexity of computing the parity function in a certain weak model of computation. So parity function is obviously very trivial function. But for the parity function, if you can't count, whatever that means, but there is a model you can kind of set up where you can't count. Then parity requires exponential complexity. And surprisingly, not easy to prove. But that's probably the theorem that I'm most known for. Anyway. But that would be a topic for another day.

Another question. Why not include Myhill-Nerode theorem. I don't know. That's a theorem about finite automata and all of those ways of characterizing the regular languages. That seems kind of a technical theorem. I don't see much point in covering it.

And another question that some of my colleagues ask me is why don't I have Rice's theorem, which sort of provides a kind of a machine for proving undecidability. And I don't know. I think that you can use Rice's theorem without understanding how to prove undecidability. It's like checking off a box. Checking some boxes and then you conclude something's undecided. I'd rather have somebody understand it rather than be able to use some powerful tool.

Can we understand that proof about the parity function that I just alluded to? It's super hard. With the knowledge from this class, I think you can. That theorem relies on a certain technique which we didn't cover called the probabilistic method, which is a kind of an amazing method. Not hard to explain, but basically you show that something exists by showing that the probability that a random object has the property you're looking for is more than 0. And so therefore, the thing that you're looking for that has that property has to exist. There are lots of examples of that these days. But it's kind of an amazing method. So we use that method.

Do I think quantum computing can solve useful problems beyond the capability of computers? I have no idea whether one can really build a quantum computer. It seems to be always 20 years off at least to doing one that factors. And I've been literally I remember people 20 years ago saying it's 20 years off. So I don't think it's converging. I'm skeptical that they'll ever build a quantum computer that can factor. I'll go out on a limb and say that. But that's controversial. And whether it can solve other useful problems, I'm not sure what other useful problems are there. Well, I guess they're simulating quantum systems. So maybe that might be possible.

All right. I think I'm going to end this now. But thank you, everybody. Take care. Bye bye.