[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: Hi, everybody. Glad to have you all back for our next to last installment of theory of computation. Today, we are going to embark on the very last big topic for the semester, and that is, in some ways, going to be following on what we started a couple of lectures back when we looked at probabilistic Turing machines and probabilistic computation and its associated class BPP. Now what we're going to discuss is, in some sense, a probabilistic version of NP. And that's going to be a complexity class called IP, which stands for Interactive Proof systems. And so we're going to present that model and look at a couple of examples.

I would just like to say at the beginning that this model is a very important one. It really has been the starting point for a great deal of research in complexity theory. So we're just really going to be touching on it, but there's a lot more that people have pursued with this model.

And it's also a connection into the cryptography field, which also makes use of the interactive proof system model. In fact, some of the genesis of that model comes out of cryptography where you're having multiple parties either communicating or, in some ways, interacting to achieve certain goals of communication or signing or passwords or what have you. So this is both an applied area and also one that has a lot of very interesting theory associated to it.

So with that, why don't we-- we're going to jump in and start out by making myself smaller and just do an introduction. I'm going to introduce the model or the concept of an interactive proof with an example, and that example involves the graph isomorphism problem. That's the problem of testing whether two graphs are isomorphic.

What do we mean by two graphs being isomorphic is that they're really just the same graph with one of them perhaps being relabeled or permuted so that they may look superficially different, they may appear with a different sequence of labels, or the nodes are appearing in a different order, but except for that, it's really just the same graph. So I'm kind of illustrating that here if you can see those two graphs here, which look different from each other. Both on 8 nodes. They are, in fact, the same graph as I can illustrate by a little animation which will convert this one into that one.

So the two graphs, these graphs being the same, we call that isomorphic. So these are graphs G and H, and they're really the same graph. So we called them isomorphic graphs, and we have an associated computational problem called ISO, which is given a pair of graphs, we'd like to know, are they isomorphic or not? So ISO is the collection of pairs of graphs which are isomorphic.

And it's easy to see that this problem is an NP problem because all you need to do in order to see or to give a certificate that the two graphs are isomorphic to each other is tell you-- it's just to say which nodes in the one graph correspond to which other nodes in the other graph. And then you all you'll need to check is that the edge relationships are consistent with that mapping or that isomorphism, as it's called. So it's easy to see that the ISO problem is an NP. And if you're not getting that, make sure you understand because the whole first part of the lecture will be lost if you don't understand this ISO problem.

Now, the question of whether you can test two graphs being isomorphic in polynomial time is not clear. And in fact, that's an unsolved problem to this day. And it's a problem that has generated an enormous literature. There are hundreds of papers on the graph isomorphism problem, as it's called, to try to resolve-- to try to see if one can find a polynomial time algorithm. And in fact, it was a very big result just in the last 10 years where there was subexponential algorithm given, so that was more faster than the brute force search approach but didn't get it all the way down to polynomial.

Now, why is there so much attention just to this one particular NP problem? It's because it's not known whether the graph isomorphism problem is NP complete. ISO is not known to be an NP complete problem, and that puts it into a very, very small class of problems in NP which are not known to be either NP or NP complete. It's kind of a curiosity that for NP problems, almost all of them have ended up being in one side or the other. And in fact, it's a-- in fact, I think it's the only problem that just involves graphs that's not known to be either in P or an NP.

So I got a question here. What would be in between exponential and polynomial? For example, I don't remember what the bound is, but it's something in the range of n to the log n, a time complexity for the graph isomorphism. I may be getting that wrong. I don't remember exactly what the bound is. But that's significantly better than 2 to the n or some exponential amount of time, but it's more than n to any constant, so it's more than any polynomial time.

So another question of the same sort is whether the complementary problem is an NP or whether ISO is in coNP, or let's talk about it in terms of the complement, whether the complement of ISO, which I'll refer to as the non-ISO problem, whether that's known to be an NP. So that's also not known. In other words, if I give you two graphs and I ask you to show that they're not isomorphic, suppose they aren't isomorphic and you go through the effort of determining that by a brute force search and now you want to prove that they're not isomorphic, well, it's not known to be an NP either. So there's no known short certificate of two graphs not being isomorphic. We don't know how to do that either.

But there's something that's very interesting, nevertheless, and it has to do with the ability for one party to prove to another that graphs are either isomorphic or not isomorphic. So if you're just having it like a prover, we haven't really been necessarily formulating that this way so much in this class, but this is a completely equivalent way of formulating the notion of NP whether you have a polynomial time verifier and a prover who can produce certificates. Say it's a powerful prover. So if you have a problem that's in NP, a prover can convince a polynomial time verifier that strings are in the language if in fact they are. So in the case of the ISO problem, a prover can convince a polynomial time verifier that graphs are isomorphic just by exhibiting the isomorphism.

Now, for the non-isomorphism case, we don't know that that problem is in NP, but it's still possible for a prover to convince a verifier that graphs are not isomorphic if you change the rules of the game slightly. So even though the non-ISO problem is not known to be an NP, a prover can still convince a polynomial time verifier that graphs are not isomorphic, assuming they are, in fact, not isomorphic, provided the prover and the verifier can interact with one another. So the verifier can ask questions of the prover, and the verifier gets to be probabilistic. So that's in this-- that's in a sense in which I mean that this notion is a kind of a probabilistic version of NP.

OK. So let me show you how that's done. So before we jump in to the method for a prover to show a verifier that graphs are not isomorphic, let's try to get a little clearer on the model. So I'm going to first show it to you informally, and then we'll look at it formally.

OK. So in interactive proofs, there are two parties. And I'm going to think about them as one of them is going to be the professor. So the professor is going to play the role of the verifier, in a sense, but it's like the one who checks. And the professor, being kind of old and tired and teaching too long maybe, can only operate in probabilistic polynomial time. So the professor, if he wants to tell whether two graphs are isomorphic or not, probabilistic polynomial time doesn't seem to be enough to tell whether two graphs are isomorphic or not because it seems to be a more than polynomial problem.

However, the professor has help. It has an army of graduate students, and the graduate students, they're not limited in the same way the professor is. The graduate students are young. They are energetic. They can stay up all night.

They know how to code. So the graduate students have unlimited computational ability. So then we're going to think of the graduate students playing the role of the prover because they're not limited in their capabilities, we'll assume.

The professor, on the other hand, is limited. So the professor wants to know if the two graphs are isomorphic, let's say, whatever they are. Can't do it by himself, so he's going to ask his students to figure out the answer and report back.

Now, there's only one problem. The professor knows that students-- well, in the old days, they'd like to party. I guess these days, they like to play computer games a lot. And so they're not really that eager to spend all their time figuring out whether graphs are isomorphic.

So he's worried that the students will just come up with some answer and figure that he won't be able to tell the difference. So the professor does not trust the students. It's not enough for the professor to give the problem to the students and just take any answer that they're going to give. The professor wants to be convinced.

So now, how could the students convince the professor of the answer, that they've really done the work and figured out whether the graphs are isomorphic or not? Well, if the graphs are isomorphic, if it turns out that the graphs were isomorphic and the students figure that out, then life is good because what are they going to do to convince the professor?

They're going to hand over the isomorphism and show, yeah, I mean, they are. Those graphs really are isomorphic, and here's how the correspondence works. Professor can check, oh, yeah. Now I'm convinced.

But suppose the graphs were not isomorphic. What are we going to do then? The students have figured out graphs are not isom-- the professor wants to be convinced. Oh, no. What are we going to do?

Well, in fact, we're going to engage-- the professor and the students are going to engage in the following protocol. Dialogue. What's going to happen is-- now, you have to make sure your-- this is critical to understand this little part of the story here because it's really going to set the pattern for everything in today's and tomorr-- in today's lecture and the next lecture. So we're going to engage in the following interaction between the students and the professor, which is going to enable the students to convince the professor that the two graphs really are not isomorphic. So how is that going to work? This is a beautiful little thing, by the way.

So the professor is going to take the two graphs and pick one of them at random. Has these two graphs, G and H. Let's say they really are not isomorphic. The professor doesn't know that for sure. That's what the students claim. The professor really wants to be convinced that the students are right.

So the professor's going to pick one of the two at random. Randomly permute that choice, the one that he picked, and hand it over to the students. Say, OK, here is one of those two graphs randomly scrambled. Then I'm going to ask the students, which one did I pick?

Now, if the graphs were really not isomorphic, the students can check whether that randomly scrambled graph is isomorphic to either G or to H. It's going to be isomorphic to one or the other. And then the students can figure it out, and they say, oh you picked G. Or no, you picked H, as the case may be. The students can figure that out.

But if the graphs were isomorphic, then that scrambled version of G or H could equally well have come from either of them. And the students would have no way of knowing which one the professor picked. So there's nothing they could do which would be better than guessing. So if we do that a bunch of times, the professor picks at random, sometimes go secretly of course, picks either G or picks H and the students get it right every time, either the students are really doing the work and the graphs are really not isomorphic or the students are just incredibly lucky. They're managing to guess right, let's say, 100 times.

So how would the stu-- the professor randomly and secretly picks G or H, uses its probablism. Flips a coin. Just a two-sided coin.

Says, OK, sometimes I'm going to do G, sometimes I'm going to do H. Just completely at random picks one or the other. Then with some more randomness, finds a random permutation of the one that he picked and then sends that over to the students and say, which one did it come from?

So I'm not sure-- OK, so let's pause here. Let's make sure we all understand this because this is really important. So I'm getting a question here.

How do we-- I'm not sure what your question is. OK, so let me just say the professor is going to play the role of the verifier. The graduate students play the role of the prover that's coming, but I really want to understand this protocol here.

OK. So how is the professor picking the graphs again? OK, I don't-- picking the graphs at random. You have just two graphs. They're part of the input. Both the students and the professor can see the graphs, and the professor's just picking one of them at random using a coin. So I'm not sure I understand the question there.

Could P and V engage in a protocol where the secret here is on the prover side instead? The question of revealing the isomorphism-- there is no iso-- I'm not sure I understand this question either. Maybe we'll make this clear-- for this little illustration, the professor doesn't know. The graphs could be isomorphic or they could be not isomorphic. And so the professor wants to be convinced either way, whatever the students-- whatever answer the students come up with.

We're going to shift this into a problem about deciding a language next. But right now, I'm just trying to give a sense of how the model works. I want to move from this informal model, and now I'm going to formalize that in terms of model which will be deciding a language. OK?

So the interactive proof system model, we have two interacting parties, a verifier, which is probabilistic polynomial time, played by the professor in the previous slide, and the prover, which is unlimited computational power, played by the students in the previous slide. Both of them get to see the input, which in the previous case, well, it could be, for example, the pair of graphs. The exchange of number of polynomial-size messages. So the whole exchange, including the verifier's own computation, is going to be polynomial. The only thing that's not included within the computational cost is the prover's work, which is unlimited.

After that, the verifier-- after the interaction, the verifier will accept or reject. And we're going to define the probability that the verifier, together with a particular prover, ends up accepting as you look over the different possible coin tosses of the verifier, which could lead to different behavior on the part of the verifier and therefore, different behavior on the part of the prover. So over all the different possibilities for the verifier's computation, we're going to look at the probability that the verifier with this particular prover ends up accepting.

And I've written it this way. It says the probability of the verifier interacting with the prover accepts the input. It's just simply that. And so we're going to work through an example. We're going to work through the previous example more precisely in a second.

The class IP for Interactive Proofs stands for-- it's a class of languages such that for some verifier and a prover, for strings in the language, the prover makes the verifier accept with high probability. And here is the interesting part. For strings not in the language, the prover makes it accept with low probability, but there's no prover which can make it accept with high probability. So there's no way to cheat.

If you think about it in the case of the graph non-isomorphism, if the graphs were really isomorphic and the students were trying to, in a devious way, prove through that protocol that they're not isomorphic, they would fail because there's nothing they can do. If the graphs were isomorphic, then when the verifier, or the professor, picks one or the other at random and scrambles it, the students would have no way of telling which one the professor did. So no matter what kind of scheme they try to come up with, they're going to be out of luck.

So it's no ma-- for any strategy, for strings that are not in the language, for any prover-- calling that P with a tilde to stand for a devious or crooked prover. For any possibly crooked prover, even that would be working with the verifier is still going to end up accepting with low probability. So strings in the language, there's going to be an honest prover who just follows the protocol in the correct way, which makes the verifier accept with high probability. For strings not in the language, every prover is going to fail to make it accept with high probability.

OK. So I mean, the way I like to think about it is that P tilde is a possibly crooked prover which is trying to make the verifier accept when it shouldn't because the string is not in the language. It's like you can think of this in the case of satisfiability. A crooked prover might try to convince the verifier that the formula's satisfiable when it isn't by somehow trying to produce a satisfying assignment, but that's going to be impossible. There's nothing any strategy can possibly work when the formula is not satisfiable if that's what the verifier is going to check. It's going to be looking for that satisfying assignment. OK?

And by the way, we're not going to prove this, but it's really going to be proved in the same way. You can make that one third error that occurs here, something very tiny, by the same kind of repetition argument. OK?

So let's see. So why can't the prover in the first case be crooked? The prover in the first case could be crooked, but that's not going to serve the purposes.

What we want to show-- think about it like we think about NP. For strings in the language, there exists a certificate. There is a proof that you're in the language. So if somebody is going to not produce the proof, that's irrelevant.

The question is, if you look at the best possible case, the best possible prover who's going to be able-- we're asking, does there exist a way to convince the verifier that the string is in the language? So it doesn't matter that there might be some other silly way that doesn't work. We just were looking at the best possible way.

So the best possible way when you're in the language is going to end up with the verifier having high probability. When you're not in the language, the best possible way is still going to end up with low probability. When I talk about best possible, I'm trying to maximize the probability that the verifier is going to end up accepting.

Let's continue. Not sure I was as clear as I would like, but maybe again we're going to stick with that example because this is a very helpful example to try to understand the setup. And so we're going to-- I'm going to revisit that previous example about non-isomorphism but now in the context of this thinking about it as a language.

So we're going to take this non-isomorphism-- yeah. We're going to take the non-isomorphism problem and show that it's an IP. So there's going to be a verifier together with a prover, which are going to make the verifier accept with high probability for strings in the language, namely graphs not being isomorphic, and nothing that's going to be no way to make the verifier accept with high probability for strings out of the language. Therefore, that's when the graphs are isomorphic.

OK. So the protocol is just we're going to repeat the following thing twice. You know, I said in the previous case do it 100 times just to help us to think about it, but actually, twice is going to be enough to get the bound we need. So the verifier is going to operate like this, in terms of this is the verifier's first communicating, sending messages to the prover. It's going to randomly choose G or H, just like what the professor did last time, randomly permute the result to get a new graph, K, which was going to be-- which is isomorphic either to G or H depending upon the choice the verifier made, and then send that graph K.

Now, the prover's turn is going to respond by-- the prover's going to compare K with both of the original graphs. It's got to be isomorphic to one or the other. And it's going to report back which one. Just going to say, well, you picked G. No. Or you picked H.

Because the prover, with its unlimited capabilities, can determine that. And then V accepts if the prover was right both times. And if the prover was ever not right, the verifier says, oh, something's fishy here. Because we know that the prover has unlimited capability, so could get it right if this was an honest prover. And so if it's not getting it right, then the verifier is going to reject.

So if the graphs are not isomorphic, the prover can tell which one it picked randomly. So therefore, if the graphs are not isomorphic, the verifier with that honest prover will accept with probability 1 because that honest prover is always going to get the right answer, which is at least 2/3, is the bound we need. We don't care about the space used, in answer to a question.

If we were not in the language, so G and H are not isomorphic, then there's nothing any crooked prover could possibly do because it gets a graph. Can't tell. There's no way to tell whether it came from G or came from H. So that crooked prover would have o-- the best thing it could do is guess. So a 50% chance of answering correctly each time and only a 25% chance for doing it twice.

And that's why I did it twice, in order to get that error to be small. So it's only a 25% chance of the prover getting lucky, so that would be an error case if the prover, just by chance, picked the right answer twice, even though the graphs were isomorphic. So therefore, for the isomorphic case, the verifier interacting with any prover is going to accept that input with, at most, one quarter, 25% of the time, which is less than a third. So that's just to achieve that bound. OK?

So let's answer some questions first, and then I'll try to-- I'll ask you. You understand this? So I think it's worth trying to understand this model of this interactive proof system. It's a little slippery, I realize, but if you just hold onto your intuition of the prover trying to convince-- a powerful prover trying to convince a limited verifier of some string being in a language. You want the prover to be able to succeed when the string is in the language but fail when the string is not in the language.

Yes. We are going to-- somebody's asking if the prover is identifying G or H by brute force. Yes. The prover is going to use its unlimited capabilities to determine, given K, whether it came from G or H.

The computational cost of the prover is irrelevant for this. It's just like when we think about a certificate for satisfiability. We don't talk about the cost of finding that certificate for NP. For IP, again, we don't talk about the cost of the prover running.

So somebody is asking, does the crooked prover answer just randomly, or can the crooked prover have a strategy? The crooked prover can have a strategy. We're assuming the crooked prover is devious. But it's still going to fail.

OK. Let's do the check-in. Suppose we change the model so that the prover can watch the verifier picking its random choices. So the verifier cannot act in secret anymore, but the prover can watch the verifier. Now, let's suppose we had the same protocol that I just described.

What language do we end up with? Is it the same language, different language, and what is that language? So going to hopefully-- it'll give me some sense of how well you're following me by how well this goes. Yeah, someone's asking about how this connects up, for example, with NP. So we're going to look at that also in a second.

OK, so this is reassuring that most of you, I think, are on the right track, at least for this check-in. Do we assume P uses this access to guess right? What access? P is not really guessing.

The P is actually-- I don't think a P is non-deterministic or anything like that. P is actually trying to get the right answer and using its computational ability to do that if it's possible. It may not be possible. Then there's nothing you can do.

OK, so let's end this. Are you all in? Two seconds left. Please vote. Vote now or never. OK, ending.

Yeah, so C is the correct answer here. If the prover can watch what the verifier is doing, the prover can see what graph the verifier picked right from the beginning. And so the prover, without having to do any work, can say-- prover looks over the verifier's shoulder and says, oh, you pick G. And now you're randomly permuting it, but I don't care about that. I know you pick G, so the prover is going to respond back a G.

Even if the graphs were isomorphic, the prover is going to be able to get the right answer. Kind of interestingly, you can make a-- you can change the protocol somewhat to make it that even if the prover has access to the verifier's randomness, you can still achieve this, but not with the same protocol. So that's a separate question.

OK, so let's move on here. Don't want to get too bogged down. OK, here's another check-in.

OK, so you have to tell me, which of the following statements are true? As far as you know. You'll have to think a little bit how these relate to-- how NP and IP or BPP and IP relate to one another.

OK, how are we doing on this? So we're going to have to close this pretty soon too. Do the best you can.

Interesting. OK. Closing up shop. Last vote. OK, 1, 2, 3. There's one more person out there who hasn't voted who voted last time. Oh well.

All right. In fact, they're all true. Let's see. Why is NP contained with IP, contained in IP? Well, many of you have seen this already, so let's just quickly go through it.

If we just had a deterministic V, maybe it's just-- is that going to be enough if deterministic V-- I think it's just going to be equivalent, but actually, just to be doubly sure, the deterministic V and the prover just sends a message to the verifier and then checks it. That's the way we normally think about a certificate for NP. I don't think it's going to change anything, but should double-check that, if the verifier can still ask questions. But I think as long as the verifier is deterministic, you're going to get exactly NP here.

And now, how about BPP? Well, there you don't even need a prover because the verifier is already probabilistic. So verifier can ignore the prover.

And this one is a little tricky, IP contained in PSPACE, because we haven't covered that. So there's no way for you to know that unless you happened to read ahead in the book. But it's, in fact, true.

In some ways, it's a little bit like the proof that NP is contained in PSPACE. IP is sort of an enhanced version of NP. And there's just basically a piece-based brute force algorithm that goes through the entire tree of possibilities of the verifier and verifier with exchanges with the prover and can determine that the verifier is either going to accept for some prover or is going to end up rejecting for every prover. So we're not going to prove this statement, but something good for you to know anyway, just the fact.

But we're going to do-- the surprising thing, in reference to part C, is that the containment also goes the other way. This is the amazing-- is an amazing result, that everything in PSPACE you can do within IP. So this is-- IP actually turns out to be incredibly powerful. Gives you everything in PSPACE.

You get IP equals PSPACE. So that says that any problem that you can solve in PSPACE, like any of the-- a game, for example. If you can imagine formulating checkers or chess as a PSPACE problem, which depending upon some details of the rules you can do because you have to generalize it to an n by n board, but OK. Let's not quibble.

Then we don't know which side has a forced win in chess, and even if somebody goes to the effort of going through the game tree and determines that, let's say, white has a forced win, there's no way for them to-- there's no short certificate. We don't know that that problem is not an NP. But by going through an interactive proof, an all-powerful prover could still convince somebody that white had a forced-- convince somebody in polynomial time that a white has a forced win, let's say, in chess. Again, a little stretching things because this is-- you really need to talk about this as an n by n, not an 8 by 8, but I think the spirit is fair.

So OK. So let's continue. So we're not going to quite prove that PSPACE is contained in IP. We're going to prove a somewhat weaker statement, but very similar and historically came first, that coNP is contained in IP.

So not only is NP contained in IP, but we're going to prove that coNP is contained in IP. And this actually has most of the idea for the PSPACE being contained in IP. And itself, it's just an amazing proof. A little easier.

OK. This was done, if I'm remembering-- somebody's asking me, how old is this? It's something in the, I think, late '90s, but I'm not-- I don't remember. Maybe early '90s. I think it's late '90s when this was shown, so it's been a while now.

OK. So yeah. So in terms of the relationship with cryptography, there were two parallel threads that both independently came up with the notion of an interactive proof system. I was a little bit personally involved with this in a way as well, but mainly that there was one group in cryptography working on this, and there was another group who was actually coming out of the graph isomorphism world, working on it.

And they came up with two separate models, one involving the private randomness and one involving the public randomness. And it was turned out that they were actually equivalent. And it's an interesting story, but unfortunately, we don't have time for it. So why don't we move on.

And I'm going to start showing you how the proof that coNP is contained in IP goes. And what we're going to do is work with a problem that's almost like coNP complete, but going to be-- well, it's going to be this #SAT problem. We'll see the connection with coNP in a second. So coNP, so it's supposed to be exactly k satisfying assignments. Phi comma k is a set of pairs where the formula phi has exactly k satisfying assignment.

So really, this is a problem of counting how many satisfying assignments you have in a formula. So for NP, you have at least one. But I want to know exactly how many. So the #SAT problem is the pair's formula and the count.

And so if we define the count, #phi is the number of satisfying assignments of a phi. Then in another way of writing this #SAT problem is the pair's phi k where k is the number of satisfying assignments of phi. So we're going to be using this notation #phi a lot, so just make sure you got that notation. This is the number of satisfying assignments of that formula. OK?

And here's a definition I probably should have given you earlier in the term, but better late than never. So the notion that a language is NP hard, it's like NP complete except without being necess-- without necessarily being in NP. So this is just the reduction part.

A language is NP hard or coNP hard or PSPACE hard or any of those other classes that we've looked at if every problem in the class is reducible to that language. But you don't know whether that language is in the class. So we just call it NP hard instead of NP complete. So you could say the language is NP complete if it's hard and it's in NP.

OK, and so we're going to show that this #SAT problem is coNP hard. So everything in coNP is polynomial time reducible to #SAT. That's easy because what we're going to do is take a coNP complete problem, which is the unsatisfiability problem, the complement of satisfiability, and show that reduces to the #SAT problem.

And that's easy because a formula is unsatisfiable exactly when it has zero satisfying assignments. So if you can tell how many satisfying assignments something has exactly, or you can answer the question, does a formula have exactly 1,000 satisfying assignments, if you can do that in general, then you can solve coNP. You can solve the unsatisfiability problem by asking if it's zero satisfying assignments, and that allows you to solve anything in coNP. OK. So we're going to just work with this one problem, the #SAT problem, and show that that problem's in IP. OK?

Let's take a quick break. OK. Feel free to send me-- let me see if I can catch up with some of the questions that have been cropping up here. So if the prover knows the random choices of the verifier, can flip the answer to make the verifier reject?

Not sure what that-- you mean in the context just of the graph isomorphism problem or something in general? I'm not sure I-- you'll have to explain. So I will respond with a question mark.

What else can I answer for you guys? So I've got a question. If IP equals PSPACE, does that mean that ISO or non-ISO might be N, might be PSPACE complete? But no. That's not known. So we're about out of time.

OK. Let's continue here. OK, so this is where we're kind of going to start to get into the meat of things. And if you didn't quite understand everything up till now, maybe just try to keep your intuition about how does a powerful party convince a probabilistic polynomial time party of the number of satisfying assignments? An exact number. Not at least, but you want to know exactly the number of satisfying assignments.

So it could be zero, for example. How do you convince a-- how do you convince someone that there were zero assignments? And you can have an interaction which does that, and that's not obvious at all how you're going to do that.

All right. So OK. So we're going to have to introduce some notation, which I hope that it doesn't cause heartburn here.

So let's say, again, here is the language we're working with, #SAT. And we have a phi that has m variables, x1 to xm. Now, here's the notation. I'm going to-- if I write phi with a-- phi of 0, that just means the formula that I get by plugging in 0 for x1 and leaving all the rest of the variables alone.

OK, so I substitute 0 for x1 where 0 means false and 1 means true as usual. And but it's still going to be some other formula but just with that substitution. If I write phi 01, that means I've preset the first two variables to 0 and 1.

If I write phi with a bunch of preset values, I'm just setting the first i variables, x1 to xi, to some values and leaving the other variables as unset. So I'm calling the ones that I'm nailing in there, as I'm already saying, these are the presets. So this is just converting some formulas into other formulas that have somewhat fewer variables. All right?

Now, let's recall that number notation and number sign notation, #phi is the number of satisfying assignments. Now, if I say #phi of 0, that's the number of satisfying assignments when I've preset x1 to 0. Similarly, if I preset the first i variables to some values and then I take-- I want to take how many satisfying assignments subject to those presets, I write it this way.

So I'm going to use this notation a lot. You have to understand this notation. Ask if you don't un-- if you don't get it.

So another way of writing it-- I don't know if this is helpful, but another way of writing #phi of a1 to ai, remember, we have m variables altogether, that means I take the variables which I have not yet preset, and I allow them to range of all possible 0s and 1s, and I add up the formula's values for all of those. So there's a 1 every time I satisfy and a 0 every time I don't satisfy. So I'm adding up all the satisfying assignments subject to these i presets. OK?

So here are two critical facts about this number sign notation. First of all, if I preset the first i values to something, now I can, in addition, set the next variable either to 0 or to 1, and I get this relationship, which is just simply a generalization of the fact that the total number of satisfying assignments of the formula is equal to the number of satisfying assignments when x1 is 0 plus the number of satisfying assignments when x1 is 1. They together have to add up to the total number because x1 is going to be either 0 or 1. So that's fact number one.

Fact number two is that if I preset everything, all of the variables, so there are no variables left, then the number of satisfying assignments subject to that preset of everything is just whether or not I've satisfied the formula, which is the value of the formula on those presets. OK? Both two simple facts, but it's going to be critical in the protocol I'm about to describe.

Questions on this? I think I actually do have a question for you. So let's just see. What do you think?

It's just to check your understanding. OK. Got about 80% getting this. I'm not sure that's good. But all right. Almost done? Closing. OK.

OK, so yes. A is the correct answer. If there are 9 satisfying assignments all together and there are 6 satisfying assignments with the first variable is set to 0, then there's only 3 satisfying assignments with the first variable set to 1 because 9 has got to be equal to 6 plus 3. That's actually this fact number one.

It's not going to be 15. This is not true either. So it's just A. Good. OK.

OK, so let's try to-- with that knowledge, let's try to see how we can put #SAT in IP. So this is not going to quite work, but it's really going to set us up to do this-- to finish this next time. So you might immediately see where this is going wrong, but you'll have to put up with it because the setup is what's important.

OK. So understand, now, here's the setup. We have the input is a formula and a number where that number is supposed to be the number of satisfying assignments. It could be wrong, in which case, we're not in the language. But if it's right, you're in the language. So the prover is supposed to convince the verifier that it's correct if it is correct. And it's not going to-- it's going to fail no matter what it tries to do if it's not correct.

So this says the prover is going to send, first of all-- so the prover is going to send a claim about the number of satisfying assignments. Going to send-- when I say this value here, this is what the prover-- if it's honest, it's going to send the right value. Of course, the verifier does not know if the prover is honest, but I'm describing how the honest prover is going to operate. And we'll have to understand what happens if the prover tries to cheat.

So the prover is going to send-- the honest prover is going to send the number of satisfying assignments altogether, and the verifier just makes sure that that matches up with the input. If it doesn't match up with the input, the verifier is just going to-- the verifier is going to not be convinced that the input is in the language. So it's going to just reject at that point.

OK. Then now the verifier says, OK. That was very good that you sent me this. How do I know that's right?

So what the prover is going to do to try to convince the verifier that this value was correct is unravel that by one level by say, well, there were 9 satisfying assignments altogether. 6 them were when x1 is 0, and 3 of them were when x1 is 1. What does the verifier have to check? That these add up correctly.

When I preset x1 to 0 and to 1, it had better add up to the total number of satisfying assignments. If that works out, the verifier's happy. It's still being-- it's still consistent with being convinced that this k was the right value.

So the next step is, well, the verifier says, well, how do I know those two values are correct? The prover says, OK. Well, I want to unravel them one level further then the number of satisfying assignments when the next variable is set to both possibilities for each of the possibilities of the first variable.

Now, if you're understanding me about what the prover is sending, you should start to be getting a little nervous because something is-- I mean, this is going to be correct, but it's going to start-- it looks like it's starting to blow up in terms of the amount of work that's involved, and that's actually a problem. But let's bear with that for the moment. Let's just worry about correctness, not about complexity for the moment.

So the prover's going to now send the number of satisfying assignments for each of those four possible ways of presetting the first two variables, and the verifier is going to check that that was consistent with the information the prover sent in the previous round by, again, checking this identity here. So then the prover's going to continue doing that until it's done that through m rounds, where m is the number of variables. So at this point, the prover's going to send all possible ways of presetting all of the variables. So now there's 2 to the m possibilities here.

Again, this is hopelessly not allowed, but OK, ignoring that. The prover's got to use this at the nth round to check what happens at the previous round, so that's when they were m minus 1 values sent because each one has one more-- you're extending the presets by 1. So we're using this to check that the previous round values were correct. So it's looking for-- the m minus 1 presets have to add up correctly in terms of the presets of m values for each of those ways of doing those m minus 1 presets.

And so now, the prover has sent all of those 2 to the m counts, which are, by the way, 1s and 0s because at this point, we have preset all of the values of the variables. And so there's only one possible assignment at most that there can be. And now the prover is done. The verifier is going to check by itself that these values make sense, that these values are correct. So it's going to do that by looking back at the formula.

So far, up until this point, the verifier has not been looking at the formula. It's just been checking the internal consistency of the prover's messages with each other. But now at the end, the verifier is going to take these values that the prover sent for each of the 2 to the m presets and see if it matches up with what the formula would do. Remember, that was the other-- sort of the base case of the fact number two from the slide or two ago. Make sure that these agree.

OK, and now the verifier says, well, OK. If everything has checked out and all of these are in agreement, then the verifier is going to be convinced that phi had k satisfying assignments. But if anywhere along the way one of these checks fails, the prover is not-- the verifier is not going to be convinced, and it's going to reject.

So in a sense, this is kind of dopey. I mean, I'm just kind of giving you a complicated way of just counting up, one by one, each of the satisfying assignments of the formula and seeing if that matches k. But nevertheless, this way of looking at it is going to help us to understand the way to fix this. So bear with me for another minute on this one.

So another way of looking at this, which I think is particularly useful, is to think of what happens-- well, OK. We'll get there in a second. I want to look at what happens if k was wrong, but before I do that, let's look at the-- I'm going to give a kind of a graphical view of the information that the prover sends and the verifier's actions in this protocol.

So the values that the prover's sending are going to be in yellow. So and the information that the verifier has or checks is going to be in white. So the verifier has the k, the input value, which is supposed to be the number of satisfying assignments, and the prover sends some value, and the verifier checks that this value, which is supposed to be the number of satisfying assignments, corresponds with k. So that's one of the checks it does.

Then the prover is going to send-- going to take, to justify this value, it sends the number of satisfying assignments when you have x1 set to 0 or set to 1. The verifier adds those up to give you-- and it's supposed to equal the total number of satisfying assignments. And so this is-- if you understood this protocol, this is just-- I'm writing it out in a sort of a simplified way perhaps. OK. And so keeps checking that these things add up correctly until you get down to setting all m values in all 2 to the m possible ways, and now the verifier is going to then check to make sure that that equals what the formula would say.

OK. So now, what happens if k was the wrong value? It did not agree with the number of satisfying assignments. And what happens now? Could the prover-- what happens if the prover tries to make the verifier accept anyway?

So the only thing the prover can do at the very first step would be to lie about-- if the prover sends the-- if k is wrong and the prover sends the correct value for the total count, the verifier's going to reject. So I'm trying to see, could the prover try to make the verifier accept? What happens?

So the prover has to lie here, and I'm going to indicate that by saying the prover is sending in the wrong value for the total count. Well, if the prover's going to lie here, then just like if you have a child who tells a lie, and then you start-- as the parent, you start asking questions to try to see if the story is consistent, one lie is going to lead to another lie. And that's what happens here.

In order to justify this lie, the prover is going to have to lie in one, or perhaps both, but at least one of these two values because you can't have the two correct values adding up to the incorrect value. So you have to think about what's going on here. So this is a lie that's going to force a lie at one side or the other one level down, which is then going to force a lie to propagate down. And so there's-- a lie at every stage is going to force a lie at least in one place or another to propagate all the way down to the bottom. And then at the bottom, the verifier will see that the check doesn't work as when it tries to connect it up with the formula itself, and the verifier will reject.

So it's just a way of looking at this. If the for-- if the value-- if the input was not in the language. So but the problem is that, as I said, this is exponential. So how are we going to fix that? So just looking ahead to what we're going to do on Tuesday-- OK, let's see if there's any questions here first of all.

OK. I got a question. Should this be-- should this be a minus? I purposely made this bracket not include the very last 0. Yeah, there's a total of m 0s here altogether, but I left out the last 0. That's why I said m minus 1. Maybe it would have been better to say m.

OK, so I've got another interesting question here. Why can't we reject right away if k is wrong? Well, the verifier is probabilistic polynomial time. How does the verifier know if k is wrong?

So I mean-- or right. So what we're trying to do is something like NP where we have a certificate, but now we have this kind of interactive certificate in the form of this prover. Maybe that's another way to look at it.

Where if you're in the language, there should be some way for the prover to make you accept. But if you're not in the language, there should be no way for the prover to make you accept. So the verifier just can't reject right away because there's no way to tell. How does the verifier know? It's going to start rejecting things when it shouldn't if it's just going to be rejecting willy-nilly here.

OK. How does the verifier need to determine if the prover is internally consistent instead of just asking-- so why does the verifier need to determine if the prover is internally consistent instead of just asking the questions in step n plus 1? Yeah, so maybe that's-- because it looks like all of the work is happening at the very end. But I'm really presenting this to you as a preparation for what we're going to do on Tuesday.

So it's important to think about the connection from each step to the next. Each step is going to be justified by what happens at the next step until we get to the very end. So you'd have to just understand it for what it is. Don't try to make it more efficient. I realize this is kind of dumb.

Good point. We're not using the probablism here. And moreover, we're not really even using the interaction here. The prover is doing all the sending. The verifier is just accepting at the end. Yeah. We're not using the power, and we're getting a weaker result.

So let's move on before we run out of time here. So how are we going to fix this? So the problem is this blowing up. To justify each stage, each value we're needing to present two values which add up to it. And that's leading to a blowup.

Now, it would be nice if we can do something where each value was supported by just a single value at the next level. So here's an idea. In order to understand to see that this total count is correct, why don't we just pick at random either 0 or 1 and only follow that one down?

Well, the problem with doing that is because the sequence of lies could be just a single path through this tree. And the chances you're going to find that path down to a contradiction at the bottom is very low if you're just doing it at random. So just randomly picking 0s and 1s as the one you're going to justify, used to justify the previous value, is not going to be good enough.

But this is what we're going to do. However, the values that we're going to pick for these random inputs are not going to be Boolean values. We're going to pick non-Boolean assignments to the variables.

Which again, just as with the branching program case, didn't make any sense on the surface of it. We're going to have to make it make sense. And we'll have to see how to do that in Tuesday's lecture. So that's kind of the setup.

OK. Yeah, so in a similar question. Why is this any different from just non-deterministically guessing the assignments? It's because of this. We're really setting the stage.

OK. So what we did today was we introduced the model and defined the complexity class. We did show this one in its full glory. We showed that non-ISO is an IP. Really worth understanding this protocol here, making sure you're comfortable with that and also the model itself.

And so for Tuesday's lecture, we're going to finish this up. Well, we started showing that #SAT is an IP which is what we need to do to prove coNP is an IP. And we'll finish that next time, which will be our last time.

OK. So that's it for today. I'll stick around for questions. So a good question here.

Why can't V just reject if some of the checks are incorrect? Yes. As soon as there's a check that fails, V can just reject at that stage. I'm just trying to argue that at some point along the way, if the input is not in the language, there's going to be a check that fails. I mean, I said reject at the end, but yeah. I mean, you could have rejected at any point along the way.

OK. Someone's asking for what role did I play? So I did-- my own personal role in this was twofold. First of all, I came up with the idea of-- not the idea. I came up with the name interactive proof.

I remember when Silvio Micali was explaining this to me in my apartment many, many years ago. He had kind of a little bit complicated-- and I don't even remember what the protocol was for. It was not for something simple. It was something involving prime numbers.

And I said, oh. That's a kind of an interactive proof. And it stuck from that point on. So that was one thing.

But the other thing, in terms of more mathematically, my role was-- so Shafi Goldwasser and I proved the equivalence of the two models, the public coin and the private coin version. So that was my role in this back when this was all first coming out. Proved it on an airplane on the way to a conference somewhere.

Anyway, so I think we're going to-- unless there's any other questions, I think we'll head out. Take care, everybody. See you on-- see you on Tuesday. Bye-bye.