[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: Welcome back. I hope you had a good Thanksgiving and all refreshed and ready to think about some theory of computation. We're in the homestretch now.

We have this lecture and two more to go. And so today, I have for you, a completion of the theorem we started before the break, where we introduced probabilistic computation and we talked about the class BPP, as I hope you remember, and we looked, in particular, at these problems involving branching programs, where we started the proof that the problem of equivalence of two read-once branching programs can be solved in this class BPP.

So what I'm going to do is spend the first 15 minutes or so just reviewing where we were, because we started this, it feels like a long time ago now. And I just want to make sure that you're all on the same page and we're all remembering what we were doing. And then, I will finish off the proof.

And along with doing that, we're going to introduce an important method. Well, we started that. We looked at the method of arithmetization last time. So we'll review that.

We're going to use that again in the work that we're going to start on Thursday on interactive proof systems. So this is a kind of, in some ways, both an interesting theorem in its own right, and a warm up for what we're going to be doing in the last topic of the semester. OK.

So let's just remember what we were doing. So we introduced probabilistic Turing machines. So those are these machines that have-- a kind of non-deterministic machine, but there's a different rule for acceptance. And these are also non-deterministic machines which can either make one choice, just to have a deterministic move at a step, or they can make two choices. And when the machine makes two choices, we actually think of there being a probability there, where the machine is tossing a coin to decide which branch to go on.

So with that, there is a tree of possible branches. And the probability of some particular branch is going to be 1 over 2 to the number of coin tosses on that branch. And so we then use that to define the probability that the machine accepts, which is the sum over all of the probabilities of the accepting branches. And the probability that it rejects is 1 minus the probability that it accepts.

So thinking about it-- this captures the idea that if you just run the machine on a random set of inputs from the coin tosses, what the probability that you're going to end up with the machine accepting. That's the probability of acceptance, defined in that way. Now, if we're thinking about the machine deciding some particular language, it's supposed to accept the strings in the language and reject the strings which are not in the language. But because of the probabilistic nature of the machine, it might get the wrong answer on some branches.

And so we say that a machine decides a language with a certain error probability, means that the probability of getting the wrong answer is going to be, at most, that error probability epsilon over all of the possible inputs to the machine. So if we say that the machine is error probability 1/3 that means that it gets the right answer for every string with probability at least 2/3. OK, so that led us to the definition of this complexity class BPP, which I don't even remember if I told you what it stands for.

It's bounded probabilistic polynomial time. That's what BPP stands for. The "bounded" means is bounded away from 1/2 because we don't want to allow the machine to have probability 1/2, because then bad things happen. The machine can just toss a coin when it decides to make an answer, and not really give us any information.

Then, we also went over the amplification lemma. We did not give the proof, but we went over the statement of the theorem. The proof is really just a calculation that you can drive down that error probability to something extremely tiny just by basically repeating the machine and taking the majority vote of what it does on several different runs. If you run the machine 100 times and you see if it's mostly accepting, then you want to accept.

And the chances that the machine was really biased toward rejecting, even though you're in your sample see mostly acceptance, is extremely small. And you can calculate that, but you can make that very tiny. So small that, for all practical purposes, it's really giving you the right answer. But it's not deterministic. So it's not quite 100% guaranteed.

And the way I like to think about BPP in terms of the computation tree of the machine, so that when it's accepting, most of the branches are accepting, weighted by their probability, of course. So the there are many accepting branches when you're accepting, and many rejecting branches when you're rejecting. So just another way of saying the same thing.

Now, we're going to jump right in with a check-in. And this is a little bit more, not exactly the material of the course, but a little bit more on the philosophical side. But let's just see how you do with it.

When you're actually running a probabilistic machine, you imagine the machine, as we're kind of informally describing it, is tossing coins. Every time it has a non-deterministic-- every time it has a choice. So it choice tosses a coin to decide which way to go.

Of course, a real computer does not have a coin to toss, presumably. Well, maybe you might actually build some hardware into the machine that lets it access randomness in some sense. Maybe it uses some quantum mechanical effect to get some random value or maybe it uses the timer. I'm not exactly sure. You can imagine having a bunch of ways of implementing that.

A typical way that people implement randomness in an algorithm is to use a pseudo random number generator, which is a procedure that might give you some kind of a value that looks random, but may not actually be random. It's, for example, giving you the digits of pi. If you want binary, expressing pi as a binary number, then you might calculate the different successive digits of pi and use that as for your random number generator.

Of course, that's a deterministic procedure, so it's not really random. But often, people do use those kinds of things when they're simulating random machines. So what do you think about doing that? Could we use a pseudo random generator as the source of randomness for our randomized algorithm? Yes, or no, or what do you think?

So let's launch a poll on that, so I can see what your opinion about using pseudo random number generators instead of true randomness for our algorithms. I'll give you a few seconds, a minute to weigh in on that.

OK. We're going to close this down. Everybody's participated who wants to? 1, 2, 3. OK.

Yeah, I think probably the best answer is A. Let's take a look. There were a couple of answers here that, really, that don't make-- that aren't as good. I would say, B, well, usually people think of pseudo random generators as pretty fast procedures. They're not that interesting, otherwise.

So I wouldn't say that B is a good choice because they're usually pretty quick to implement. C is a worse choice, even, because Turing machines can do anything that any other algorithm can do. So, certainly, if there is such a thing as a pseudo random number generator, and there is, then you could implement it on the Turing machine.

D is kind of an interesting answer because you're saying, well, that would imply that P equals BPP if you could actually simulate randomness with a deterministic procedure. But in fact, the reason I would not choose D is because it's perfectly conceivable that P does equal BPP. We don't know that P is different from BPP, so it's conceivable that they're equal.

And in fact, I think if you polled most complexity theorists, most people in my field would believe that P does equal BPP just for this very reason, that if you had sufficiently good pseudo random number generators, you could actually eliminate the probabalism in these probabilistic computations. You could just run them on the pseudo random number generator.

And in fact, there is some theory around that that has been developed. But at the present time, we do not know how to prove that there were pseudo random number generators. And it has some, actually, there's actually, in some line of this research, has some connection with the P versus NP problem, but we don't know how to prove that there are sufficiently good pseudo random number generators that would allow you to run them on a probabilistic algorithm and have a guaranteed behavior which is as good as running truly random numbers into the probabilistic algorithm.

And so the answer that I would pick would be A, that you could use it, sure. You might get the right answer, but it's not guaranteed. We just don't know how to do the analysis for the pseudo random number generators.

And if you had ones that were good enough, they would show P is equal to BPP. But that might be, in fact, the correct-- that might actually be true. OK, so let's continue on.

And remember, now, branching programs. We had these kind of networks of nodes and edges. And there was a procedure, we'll see a couple of examples again, some of the ones that we had from before where you have branching programs that look like this.

And you have a bunch of query nodes. You look at the settings of the variables to decide whether to go down to 0 edge or a 1 edge. And eventually, you're going to end up at an output node, and that's going to be the output of the branching program. And in such a way, these branching programs defined Boolean functions, from the settings of the input variables to a 0 or 1 output.

Now, you might have two branching programs and wonder whether they're computing the same Boolean function or not. And testing that is a coNP complete problem, as you're asked to show on your homework. Now, if the branching program, however, has a restriction, namely, that it's not allowed to ask to query the same variable more than once on a path, then with that restriction, we call it a read-once branching program. And then, the situation for testing equivalence seems to be different.

In fact, we can give a BPP algorithm for testing the equivalence of read-once branching programs, even though such a thing is unlikely to be the case for general branching programs because of the coNP completeness. OK, so I hope you're comfortable and with me on all of that reasoning. OK. All right.

So the idea for proving this is, what we're going to do is we want to take the two branching programs and run them on a randomly-selected input. But as we observed last time, if you just run them on a randomly-selected Boolean input where we assign the variable 0s and 1s, then that doesn't give you the right answer with high probability, because the two branching programs might be different, computing different Boolean functions, but they differ only on a single input setting.

And then, just picking them at random, you're not going to have a very high probability of finding that one place of difference. So instead, what we're going to do is define a way to run these branching programs on non-Boolean inputs, where the variables are set to values other than 0 and 1-- 2, 3, 7, 22-- and make sense of that.

And then argue that, by running the two branching programs on a randomly-selected non-Boolean input, that that's very high probability of giving you the right answer. So somehow, by expanding the domain of possibilities, you're going to better your chance of getting the right answer very significantly. OK. So even though these two branching programs might agree on almost all of the Boolean inputs, we're going to show that by doing this arithmetization-- so this is the method-- if they're really not equivalent, they're going to differ almost all of the time on the expanded domain.

OK, and then we have the proof. That's where today's work is going to be. OK. So why don't we just stop and make sure we're all together on this? I can take any questions.

I'll also review how the arithmetization goes. But I'll do that next. So, are we all OK on this? Good. So let's move on.

So in order to move toward understanding what it means to run the branching programs on these non-Boolean values, we're going to have to get a somewhat different perspective on the computation of a branching program. So the standard perspective is that you take your setting, your assignment to the input, which is 0,1,1 for x1, x2, and x3, and use that to follow an execution path through the machine. So we know x1 is 0, x2 is 1, x3 is 1, the output is 1, as I've indicated in yellow.

This other perspective says, well, we're going to operate by labeling the nodes and edges of the machine. And that's going to have a very direct correspondence with the execution path perspective. So we're going to label all of the nodes and edges on the path with a 1, as indicated in yellow, and all of the nodes and edges that are not on the path, all the other nodes and edges are going to be labeled 0.

So by following the 1s, it's like the breadcrumbs in*Hansel and Gretel.* This is the path you need to follow to get through the machine. The 0s are the places where you don't go.

OK, so the output label, here, the output is going to be the label of the one output node, whatever you're labeling that. Because if it's a 1, that means the path went to the 1, and if it's a 0, that means the path didn't go to the 1, it went to the 0. So just by looking at this value, you can see what the output of the machine is.

All right. So let's describe a different way of defining this labeling without just looking at the path. Well, it's going to capture exactly the same thing. So we're going to say, if you've already labeled a node, I'll tell you how to label the two edges that emanate from that node.

I'm going to label the one edge a and the query variable. Why is that? Well, a is going to be either a 0 or 1. And it's going to tell us whether or not the path entered that node.

So if it's a 1, it entered that node, if 0, it didn't enter that node. The only way it's going to go down this branch here is if it did enter the xi node. If it didn't enter the xi node, there's no way it can go down this branch. So we're going to and that value.

So the only way you can go down this branch is if it went through that node-- so that's the a, the value of a-- and the xi is a 1. That's why we say a and xi. So you really have to understand this little expression here. If you don't understand that, you're toast.

OK, so you better understand this so we can move forward. I'm happy to take a question. These are the simplest questions are sometimes the most valuable. If you don't understand why I'm labeling it this way, shoot me a chat.

OK. Now, the other branch, I'm going to say, well, I'm only going to go down this edge if, well, a is true, so I did go through that node. And xi is false. So this is going to be a and the complement of xi.

All right. So that's how I'm going to label these. This is another way, giving these expressions for labeling these edges based on the label of that node. And similarly, in order to complete the picture, I've got to tell you how to label the nodes based on the edges that are coming into it.

So if I know that I have a1, a2, and a3, which tell me the status in terms of the path of whether the path went through any of these edges, well, I know that it's going to go to that node if the or of these values-- if the path went through here, or it went through there, or went through there, then it's going to go to that node. That's why the or is the right thing to say.

So this gives me another way of constructing the labeling over here without even talking about paths. As I describe it, I argue that it's going to give the same result. All right.

So there's a question. Can we quickly say again why we can't do that on Boolean? I'm not sure I understand the question. So send it to me again.

Right now, everything is Boolean. We haven't done, arithmetically, anything yet. And the reason why we can't just live in the Boolean world is that, just by taking Boolean values of Boolean assignments here, we don't have a high enough probability of catching a difference between the two machines. All right. So let's continue.

All right. So now, I'm going to talk about how we're going to extend this to the non-Boolean case, using the arithmetization method. So first of all, arithmetization is a simulation of and and or with plus and times, such that if I think about true as a 1 and false as a 0, this is going to give me a faithful simulation. It's going to do the right thing. It's going to compute exactly the same values that we expect.

So like a and b, well, times works just like and. It does for 1 and 0 as true and false, times exactly works like and. And negation is 1 minus. Or is going to be the sum minus the product.

And then, these just give you the right values, a or b. If you just calculate it out by plugging in 1s or 0s, you get the right answer just by using this arithmetic. So now, what we're going to do, instead of using the Boolean labeling, we'll just use the arithmetical labeling. But it's going to compute exactly the same thing because the arithmetic simulates the Boolean.

So we always go through the start node. So there's no question about labeling the very start node with a 1. But now, I'm going to give expressions just like the Boolean expressions, but now they're going to use plus and times instead of ands and ors. So let's just see. Remember what we did from before.

We had a and xi for this edge. I'm going to replace that. What is and? We just look up here in our table, in our translation table. And becomes times. So we're going to replace that with a times xi. And it's going to work exactly the same.

But the difference is that this makes sense even when we have non-Boolean values. Times and plus are defined for non-Boolean values, whereas ands and or are not. So what goes down on this edge?

Well, this was a and the complement of xi, as you remember. So that's going to become a times 1 minus xi. And then similarly, we had or over here. And here's a little bit of a trick, but that's going to be important for the analysis that we're going to do.

Instead of using the recipe for or in terms of plus and times, we're going to have something a little simpler. It's just going to be the sum. And the reason why that works-- good to understand-- is that because of the acyclic nature of the branching programs, at most, one of these edges can have a path through it. So this is a kind of very special or, sometimes called the disjoint or.

You're not allowed to have more than one of the values be 1, because that never happens when you have an acyclical graph. You can never have the path coming down this way, and then, again, coming down that way. Then it would be entering that node twice. Have to be a cycle.

So it's going to be good enough for us, and necessary for us to represent this or as a sum. OK. So I think that's all I wanted to say on this slide.

So somebody is asking, is it possible for some of these values to be negative? Yes. As it stands right now, some of these values can be negative. I haven't put any restriction on what the values are going to be. So the input could be a negative number. And then, you're going to just get negative stuff happening.

In fact, there's subtractions going on here. So even with positive numbers-- I think we did an example last time. I think I'm going to do that example again of exclusive or, where you get negative numbers coming up. That doesn't matter.

But actually, what we're going to end up doing is doing these calculations modulo some prime number q. OK. I'm going to pick some prime like 17, and do all the calculations mod 17. And the reason for doing it that way is really because we're going to be picking random assignments to the variables as our input. And it makes the most sense to do that when you have a finite set of possibilities to pick them up.

So we're not going to pick like a random integer. There's infinitely many possibilities. And yeah, you could set up a distribution there, but that's very complicated. That actually might work. I'm not sure. I haven't actually gone through that analysis.

But the typical way people do this is by looking at what's called a finite field. So I'll talk about that in a second. Why is there at most one 1 among a1, a2, and a3? The 1s-- I'll say once again-- but the 1s correspond to the path.

So this is a 1 if the path went this way. Just think about it. The path cannot go through a1 and can, at the same time, go through a2, because that means the path went through this node. Then, how is it going to get over to a2? It's going to go through that node twice.

In an acyclic graph, you cannot have a path going through it's the same node more than once. So you're going to have to think about that. OK, let's move on.

So now we're going to talk about the same-- we're going to look at that non-Boolean labeling applied to an example. So here is a very simple branching program that actually computes the exclusive or function in the Boolean world. So this is the labeling that I just developed for you, the arithmetical labeling. And we always label the start node with 1, because the path always goes through there.

And now, let's look at this before we jump ahead, let's look at this edge here. Remember what it is. We have to apply this rule here. This is the one edge coming out of a node that already is labeled.

So it's that label on that node times the xi. Because if you're thinking about it, that, that's the and, captures the and. So it's just a times xi. So it's x1, in this case.

So x1 is a 2 in our input. So it's going to be 1, which is the a, times the x1, which is 2. So this edge gets labeled 2. Now this-- well, OK, let's look at this edge now. I think that's next.

This is the 1 minus xi, it's the 1 minus x1. So it's 1 minus 2. So you're going to end up with a minus 1. 1 minus times minus 2. 1 times 1 minus 2, which is minus 1, so it's a minus 1.

Now we're going to label these two nodes using the other rule. So this gets a 2 because that's a sum of the incoming edges. This gets a minus 1, because that's the sum of its incoming edges. And now we're going to look at-- which order did I do this in? OK, I'm going to do this edge now.

0 edge, which is going to be 2 times 1 minus its variable. So it's 1 minus x2. X2 is a 3. So it's 1 minus 3 to minus 2. So 2 times minus 2 is minus 4. OK, I don't want to rush through this. No point in just blabbering on. I'm just trying to work this example so you get the idea.

OK, so could you fill this next one out yourself? So this is the one outgoing edge from this x2 node. So x2 is labeled with 2 here. So there's a is 2.

This is the one edge going out, so it's one on this side here. So it's 2 times the x2. x2 is a 3, so it's 2 times 3. So this should be 6. Oops. Right here-- 6.

I'm going to do the same thing. So x2 is labeled minus 1. So minus 1 times 1 minus. So you get a 2 here, you get a minus 3 here, just following, again, the same process. And now, what is the label on this 0 node here?

So again, you take the sum of its incoming labels. So there was a 2 and a 6. So that's an 8. And this is a a minus 3 and a minus 4 coming in. So it's a minus 7.

What's the output? The output is minus 7, because that's the label on the 1 node. OK, output is minus 7. OK. So this is going to be our way of defining the output of a branching program when it has a non-Boolean setting on its inputs. If you had the Boolean setting on its inputs and you followed this procedure, what would you get out?

You would get the same answer that you would get by following the path because the arithmetical simulation is a faithful simulation of the ands and ors. And the ands and ors capture exactly when-- what the path does. So this is a strict extension of the operation of the branching program into a new realm, these non-Boolean values.

On the old realm, it behaves just as it did originally. And that's critical to understand that. Yeah, somebody's asking what the final value of the 0s, they'd also be the same. Sure.

In the Boolean case, if we follow this in the Boolean cases, all of the labels would be exactly what they were. They would just be the 0s and 1s that we had from before. So does this exactly mimic x or if you take this all mod 2? I don't know. I'd have to think about that.

I don't think that that's essential. In this case, it might behave correctly if you take the answer mod 2 for XOR. But the XOR is going to be very special. And I'm not sure. That might happen to be true. I'd have to think about it for a second, but I'm not sure that's relevant.

OK, so this is a good question. If I picked a different branching program that also implements XOR-- or-- so it would be an equivalent branching program-- would it behave the same way on the non-Boolean values? And the answer to that is yes and no. You understand the question? It's a very good question.

And actually, we're going to prove this in the analysis. It's going to be easy to prove because I gave you both possibilities, yes and no. But let me tell you what I mean by that.

So you understand the question. Suppose I had a different branching program. I'm not sure you can come up with a different branching program, but let's say you could. You have a different branching program, yeah, sure you can come-- you can do the variables in a different order, for example. So suppose you come up with a different branching program that gives you XOR.

And now you plugged in 2 and 3. Would I always get the same value out? Yes, if that other branching program was read-once. No, not necessarily if it's not read-once. And that's why read-once is critical.

As we will prove, for two read-once branching programs, if they behave the same way on the Boolean values, they behave the same way even on the non-Boolean values. That's not necessarily true if the branching programs are not read-once.

OK, we will see that. We will prove that and use that. That's going to be important.

OK, so here is the algorithm sketch, which is kind of a little bit even sort of suggested by that very good question. So what we're going to do is we want to take the two branching programs that we're trying to test if they're equivalent. We're going to pick a random non-Boolean input assignment-- so set the values here chosen from the field, but we'll get there. A random value for x1, random value for x2, and so on.

These could be numbers like 17, and 25, and 23, and so on. And then, using this process, run the branching programs to evaluate them on that non-Boolean assignment. If they disagree, then we're going to know that the two branching programs were not equivalent, even on the non-Boolean case. Even on the Boolean case. Did I say that wrong?

So if they disagree, even on the non-Boolean case, they have to be an equivalent even in the Boolean case. Not obvious. But if they agree, then it's not a guarantee that they're equivalent, but it's going to be very strong evidence that they're equivalent. OK, so that's where the probabilistic nature is going to come in. So we're going to prove that.

So first, we have to develop an algebraic fact. And that involves polynomials. This is a simple thing that I think many of you have run into already, perhaps even in high school. I'm not going to prove the algebraic facts, but I'm going to state them. And actually, the proofs are not hard. They're in the textbook.

So suppose we have a polynomial of degree d here. It happens to have p. So there's a bunch of constants. These a's are constants. x is the variable of the polynomial. And I presume you've seen polynomials written out like this.

And so if you assign x to some value, some constant value z, and the polynomial evaluates to 0, we often call that a root of the polynomial. So these are the places where the polynomial evaluates to 0 that I've of shown over here. Those are the roots.

It's not hard to show that a low-degree polynomial cannot have lots of roots. Basically, if the polynomial has degree at most d, it can have at most d roots, as long as the polynomial itself is not the everywhere 0 polynomial, because obviously, then everything is a root. Oops, typo. Thank you. Should be d minus 2 over there. Pretend that's a 2.

All right, so if we have a low-degree polynomial-- let's not get ahead of ourselves. If we have a low-degree polynomial, a polynomial of degree at most d, it has at most d roots. And that's a simple proof.

The basic idea is every time you have a-- if you have a root of the polynomial, so if setting x equal to 5 gives you a root of the polynomial, it's a 0 of the polynomial, then x-- you can easily to see that x minus 5 is a factor of the polynomial. You can divide by x minus 5, and you get a new polynomial of degree one less. And you can just, which is going by induction, has one fewer root. So you can just divide out by the roots, basically. It's very straightforward.

And other very important thing-- if I have two polynomials that are both low-degree, they cannot agree on very many places. That follows from what I just proved above. Because what I'll do is I take those two polynomials, and they look at the difference, which is also a low-degree polynomial. Every time there's an agreement between those two original polynomials, there's a zero of the difference polynomial. And because that difference polynomial cannot have too many zeroes, the two original polynomials cannot have too many agreements.

So the corollary is that if x and y are both polynomials of degree at most d, and they're not the same polynomial, because then the difference would be the 0 polynomial, then the number of places where they're equal is at most d. So the proof is just letting p be the difference of p1 and p2-- very standard kind of a trick. Now the above is going to hold for any field.

A field is just a set with plus and times with the familiar properties of distributive law and so on and identities and all the stuff that you would expect plus and times to have in the normal world. And so we're going to talk about the finite field that has only q elements-- that has exactly q elements, where q is a prime number.

So it turns out that-- and I'm not going to prove all this, but it's pretty simple stuff-- that if you just take the numbers from 1 to q minus 1-- from 0 to q minus 1, and use plus n times mod q, that has all the right properties to be a field. So just think of it-- just modular arithmetic, mod some prime q. And then we can in a natural way pick a random assignment to a variable from the field because it's just choosing from among q possibilities.

Yeah, so getting a question here. The coefficients of the polynomial and the assignment to the variables-- they're all going to come from this field. So everything's going to be operating in this field. Don't let that throw you off. Just your ordinary intuition about the way arithmetic works is going to be just fine.

But this is important here from the perspective of thinking about this probabilistically. So I'm going to rethink about this polynomial lemma, which says there are not too many roots. In terms of the probability of picking an element of the field, what are the chances that it happens to be a root?

So if you have a low-degree polynomial, and you pick a random value in the field, what's the probability that you've got a root? Well, it's just the number of roots divided by the size of the field. So if you have this really big field, and you have this low-degree polynomial, it's going to be pretty unlikely that you're going to end up picking one of the zeroes, one of the roots, just at random. That's all that this is saying. So there's at most d roots out of the q possibilities.

And the last thing I'm going to introduce here is the multivariable version of this which is called, perhaps somewhat unfairly, but it's called the Schwartz-Zippel lemma, though in various forms, it had been known prior to their work. In some cases, the literature actually goes back a long ways. But anyway, this is called the Schwartz-Zippel lemma. Doesn't really matter, except to the people whose credit is being denied. But that's not one of us.

So anyway, the Schwartz-Zippel lemma says that if you have now a polynomial in several variables, which is not the 0 polynomial, where each variable has low degree-- so if I say if it has degree at most d in each xi. So each variable is going to have at most an exponent of d appearing in that polynomial.

And now if we pick random values to assign to all of those n variables from the field, the probability that we ended up with a root, that we ended up with a 0, is something you can bound. So it's m times d, so the number of variables times this maximum degree, divided by the size of the field. And this is going to come up later for us.

And this is another fairly simple proof, a little bit more sophisticated than the one that we had above. And in fact, it uses that one as a lemma to prove this theorem. So we're going to-- not going to prove any of that, but I refer you to the book if you're curious.

Yeah, so a couple of good questions here. What happens if these values are bigger than q, for example? Then it's not telling you anything. If d is bigger than q, m is bigger than q, or the product is bigger than q, then you learn nothing from this lemma-- from this theorem. So typically in applications, you're going to pick a large-- you're going to have the flexibility.

You get to choose q to be something that you want. So we're going to pick the field to be big enough so that the m and d are going to be relatively small. In fact, d is going to end up being 1, as we will see. And m is the number of variables. So we're going to pick q, which is going to be substantially larger than the number of variables.

And how is the degree defined in multivariable polynomials? If the polynomial has xy squared plus 3x to the 5th y squared z, you just pull out each variable separately. And you look at the maximum degree of that variable. So the x in that case had had a degree 5 appearance.

The y had a degree 2 appearance. So you take the maximum over all of the variables of the degree of that variable. And that's going to be the bound on the degree of the polynomial. So in fact, in our case, d is going to be 1. So all of the variables-- there's not going to be any exponents on anything. Everything is going to be exponent 1.

Is q related to the number we choose for the mod? Yeah, q is the number we're choosing for the mod. We're doing everything mod q. So all the arithmetic is going to operate in mod q, and that's the size of the field that we're going to pick.

So I think we're here at the break. Happy to take some more questions, but why don't we just start that off. And I will see you in five minutes. But in the meantime, happy to shoot me questions.

So what happens if we use Boolean assignments in the XOR example? Would that work to be able to check agreement? It would. So it's hard to make an argument based on just a single example. I think the better thing would be to look at two branching programs that just differ in a single place.

So I can even suggest two. You can make a branching program that always outputs true. It doesn't even read its variables. Or if it reads them, they always go to the same place. And it ends up always at the q output. So you imagine a branching program that always outputs 1, no matter what the assignments to the variables are. And you can easily make such a thing.

And then you make another branching program that computes the or function. So it reads every variable. And it's going to be 1 if any one of those variables is set to 1. So the only time the or function is 0 is if everything is set to 0.

But now if you're trying to randomly check whether the always one function is equal to the or function-- of course, without knowing in advance what they are, because that's cheating. You're just given these two functions, and you want to know-- these two branching programs. And you want to know, are they computing the same thing or not?

And by this procedure of randomly sampling, you're always going to get these branching programs both to say 1, unless you just happen to pick the random assignment of everything set to 0. And that's very unlikely that you're going to pick that. If you imagine you have-- your branching program has 100 variables in it. It's only 2 to the minus 100 chance that you're going to set them all to 0 randomly.

And so you're extremely unlikely to find that one place of difference if there's only a single place. If there's lots of places of difference, then it's not so bad. But if the number, the fraction of differences, is low, you're going to have to do a lot of samples in order to find that-- possibly exponentially many samples. And then you won't run in polynomial time.

So let me just see if there's other questions here. Why can we accept-- going back to the Boolean labeling side, why can we accept that b1 equals b2 if b1 and b2 agree on-- only on just one input assignment? No, we didn't say that.

All right, I'll go back there. Boolean assignment-- is this-- I'm not sure which one you mean. Is this the one you mean? I don't know-- Boolean labeling, so it must be it. Why do we non-Boolean labeling?

No, I see what you're saying. You're saying about this here. We're just going to pick one random assignment. And if they agree on that one random case, then we will say accept, because you might think, well, we should take a whole bunch of samples. That's a good question.

But in fact, we're going to arrange the probability such that if the two things are not equivalent, then it's going to be-- the values will be different almost everywhere, or a large number of places. So just picking one and having them agree is going to be strong enough evidence that you're still going to accept. And you'll have still a low probability of getting an error. You'd have to see the analysis.

Are we assuming the roots of the polynomials are integers? No, we're operating over a field here. Even talking about integers doesn't totally make sense. But it doesn't really matter. We're not assuming that. Oh, I should have taken this away.

The bound still works. The bound still works even if we have non-integers. I'm not sure if I'm being helpful here. Why don't we just move on? But we're not assuming that these are integers because the bound doesn't matter. If it says there is at most five roots, including the reals, there's still going to be five roots, including the integers. All right, so let's continue. Good, all right.

So now everybody's back, I hope? Let's talk about moving forward here. Where we're going with this is we want to analyze the algorithm, which picks a random non-Boolean input and evaluates the two branching programs. And in order to do that, we're going to look a little bit more carefully at what happens when we arithmetize the branching program, and we run it on these non-Boolean values.

And so what I'm going to do is take this branching program. Let's say this is the same XOR, exclusive or, branching program. But instead of labeling it as we did before by setting x1 to 2 and x2 to 3, I'm going to leave x1 and x2 variables and just do a symbolic execution. So I'm going to label these things, just leaving x1 and x2 as variables.

So let's just see what we get if we do that. So remember, we assigned this to be 1. Now this edge here is-- here's the rule. It's a, which is 1, times x1. So this should be-- without knowing what the value of x1 is, leaving it as a variable, we're just going to put down x1 over here. Over here, what goes over there? Well, it's 1 times 1 minus x1-- just 1 minus x1.

Now we're going to add things up, as we did before. And now what happens, for example-- I think I have this edge coming next. Let's look at this edge, the one edge coming out. The label now is x1 on this node.

This is the one edge coming out, so you multiply by the value of x2. We're leaving it as a variable, so we're just going to multiply it by x2. And so we're just going to get x1 times x2-- x1x2 on this edge.

And now what happens on this edge? So this x1 times-- think with me-- times 1 minus x2. And similarly over here, we have 1 minus x1 on this node. So I think on the one edge coming out, it's 1 minus x1, now times x2, because that's this rule again. 1 minus x, 1 times x2. And this is going to be 1 minus x1 times 1 minus x2.

Now we're going to add this up for the 0 node. So we have this value, 1 minus x1, 1 minus x2, plus x1 x2. And on this note here, we're going to add these two values up. So 1 minus x1 times x2 plus x1 times 1 minus x2. And that's the output, now expressed symbolically.

Now you could plug things in, and you're going to get the same value out as you did before. But let's leave it as a polynomial for now because that's going to help us and analyze this. So now, notice the form of this polynomial is something special.

What happens is it's going to look like a bunch of products of xi's and 1 minus xi's added up. So it's a sum of products of that form. So each row here is a product of xi's and 1 minus xi's. And then those rows are all added together. I claim that's going to be the form of this polynomial. You see this already has that form.

And the reason for that is every time you go to a node, you're just adding things up. So that's just going to be like adding up more rows. And every time you go down to through an edge, you're multiplying what you have so far either by an xi or a 1 minus xi. So you're just accumulating products of xi's and 1 minus xi's, and you're just adding them up. So this is what that polynomial is going to look like.

Now let's look a little bit more carefully at the form of this. So for one thing, could we have higher powers here, like x2 cubed? Could that happen? And when I say it's products of the xi's and 1 minus xi's, maybe there's going to be some xi's that appear several times in the product.

Well, that cannot happen. Why? It's a read once branching program. So every time you multiply by an xi or a 1 minus xi, you're never going to do that again, because doing that would imply you're querying that variable more than once. So this can't happen. So I cross that out. This just appeared. It's off to the side, here. But yeah, I'm crossing that out. That does not happen.

Another thing that is part of-- that's worthy-- that's going to be helpful to notice about this polynomial-- and by the way, maybe I'm being confusing here. This I'm supposed to be representing as a generic form of the polynomial. This is not some particularly-- yeah, I should have said this at the beginning. This is not some particular polynomial that came from anything. I'm just trying to describe what the general form of the polynomial looks like, just as an illustration.

So this polynomial's 1 minus x1 times x2 times 1 minus x3 times x4 and so on, and adding up a bunch of rows like this. I'm just saying this is what the polynomial will look like for maybe some branching program. So every branching program is either going to have some polynomial that looks sort of like this.

And what I'm also going to say-- for convenience, now, I want to say that each row is going to have every single variable appear either as an xi or as a 1 minus xi. So in order to get that, I need to make a further minor assumption about the branching program-- that it's a read exactly once. Currently when I say "read once," it can avoid reading some variables on some branches, because it's like a read at most once. But now I want to say that every variable gets read exactly one time on every branch. And what that's going to mean is that every row is going to contain every variable, either as an xi or as a 1 minus xi.

We can eliminate that extra assumption easily. And I'm going to leave that as an exercise to you. It's not very hard to do. So I think if you follow me, you can see-- and you play with it for a minute or two, you'll see that it doesn't really matter. But I think just for the first time through this, let's assume that every row has every variable-- so important to understand.

So this is the output polynomial of this branching program. So let's look furthermore at this polynomial and understand the rows. Let's take one row out of this polynomial to understand what it represents.

So one row here-- it's a product of a bunch of things, product of a bunch of variables, either variables or 1 minus the variables. Let's think about this in the Boolean setting, first of all. So in the Boolean setting, each of these variables are going to be 0's and 1's. And the 1 minus the variables are also going to be 0's and 1's. So it's going to be a product of 0's and 1's.

If there's a 0 that appears in that product, that product is going to be a 0, because 0 times anything is a 0. So the only way that product cannot be 0 is if all of those values are 1's in the Boolean case. So that means that x1-- well, let's look at the second row. So x1 had to be a 1. x2 had to be a 1. x3 had to be a 1. x4 had to be a 0 in order to continue the product of 1's, and so on.

So in fact, there's only a single Boolean assignment to these variables which make that row 1. Every other assignment to those variables makes that row 0. Saying that another way, each of these rows corresponds to one of the rows of the truth table for the Boolean function, where the truth table is true, gives a true value, gives a 1 value for the function on that row.

So I hope you're all familiar with the notion of a truth table of a Boolean function. You just write down the Boolean function, every possible assignment to the Boolean function, and you write down 1 or 0 or true or false for what the value of that function is. It's just a tabular representation of the Boolean function. It's called a truth table. This thing here gives you all of the 1's-- all of the rows that are 1 in that Boolean function. That's what this polynomial gives you.

So I think we're at a pause point for this slide. It's deathly silent on the chat. So I have a feeling that that went down rough for you. It's important to understand the form of this polynomial here. It corresponds to the truth table of the Boolean function. So each one of these rows is only going to be-- again, thinking Boolean now, each one of these rows is only going to be 1 on an assignment which makes the function 1-- one of the assignments that makes the function 1.

And somebody says, and similarly for the expression for the 0 node. Yeah, the 0 node, which I'm not focusing on, but, yeah, the 0 node would be all of the false rows of the truth table. But the 1 node, the polynomial for the 1 node, are all of the true rows-- correspond to all of the true rows in the function of the branching-- the function that branching program computes.

Let me just tell you where we're going. Is it possible to have two rows that are the same? If you think about how the rows are being produced, no. You can't have two rows that are going to be the same, because for one thing, you have to think about what this looks like in the Boolean case.

If you have two rows are the same, that means this thing is going to-- would have an output which is non-Boolean because you're going to end up with a 2 coming out that way by adding those rows together. That can never happen. And if you just look at the way it's constructed, you're never going to-- because every time you have a branching, one way is an xi.

The other one is 1 minus xi. So every time you're branching there, every time there's a node, they're different. So you're never going to have two rows that are going to be the same.

But let me tell you the importance of connecting up this polynomial with the truth table, because that tells us that if the two functions of the two branching programs that we started off with agree in their Boolean values, then the two polynomials are going to be the same. Because if the two branching programs have the same Boolean function, so they're equivalent, then the truth tables will be the same. And therefore, these polynomials will be the same.

And therefore, they will behave the same way on all non-Boolean values, because they're the same polynomial. So I'm getting ahead of myself, but that's what we're going to argue. That's why it's important to understand the connection with the truth table, because it builds on the-- understanding something about how this thing behaves in the Boolean case is going to give us information about how it behaves in the non-Boolean case.

But let's continue here, then. Yeah, this is essentially the last slide, but we're going to spend some time on this one. So here's the algorithm. We are going to take our two branching programs. The variables are x1 to xm. First of all, we're going to find a prime which is at least 3 times m, the number of variables.

m is not a very big number. It's just the number of variables. So finding a prime that's bigger than that is straightforward. We're not talking about huge primes here. We're talking about very modest-sized primes. Even trial and error is going to be good enough.

Now that's going to be the size of the field. It's going to be a field of size q. And now we're going to pick a non-Boolean assignment to the variables. We're going to pick a non-Boolean assignment to the variables and evaluate the two branching programs on that non-Boolean assignment using the arithmetization. If they agree, then we'll accept. If they don't agree, then we're going to reject. Now we have to argue that this works.

So we're going to first of all arithmetize these two branching programs, and we're going to get these two polynomials. They each have the form that-- as I described, so a bunch of rows that correspond to the truth tables of those two respective branching programs. First claim-- that if the branching programs were equivalent, so they compute the same Boolean function, then the two polynomials agree everywhere.

So then the two branching programs are going to get the same value on every non-Boolean case as well as on the Boolean cases. So they agree in the Boolean. That means they always agree, even on the non-Boolean. And I kind of argued that already.

The other point is that if the two branching programs are not equivalent, so they differ at some Boolean value, now picking a random value for the polynomial evaluation, you're going to have only a 1/3 chance that they're going to agree, so a small chance. All right, so now let's prove these two facts.

The first one I already kind of argued. If the two branching programs agree on all the Boolean values, then their functions have the same truth table. So then the polynomials are identical because the polynomials correspond to the truth table. And so therefore, they always agree, even on the non-Boolean values. So that means that the probability that if you evaluate the two polynomials on a random place, whether they'll be equal, that's a certainty, because in fact, in this case, p1 and p2 are the same polynomial.

Now for two, if the branching programs differ somewhere, even in one place, well you know the polynomials could not be the same. They have to be different polynomials because the polynomials include the behavior in the Boolean case as well as all the rest of the field. So the polynomials have to be the same.

And now we're going to apply the Schwartz-Zippel lemma. We have two different polynomials. They can only agree in a relatively small number of places. So that says that, from the Schwartz-Zippel theorem, then the probability that p1 and p2 agree at this random location is at most this value that we had from before, the degree times the number of variables divided by the size of the field.

The degree is 1. And the field is at least 3 times the number of variables in size. So that means you get this inequality here. And so therefore, the probability is a 1/3-- at most 1/3. And that's good enough.

This is the probability that you get the wrong answer is going to be at most 1/3. So you're going to get the right answer with at least 2/3 probability. So even just doing a single sample point is going to be enough to give it a PPP algorithm.

What I have are a couple of check-ins here now for you. Whoops, somehow this-- going to take me out of here. All right, so this is a little hard, but let's see how you do on it.

Suppose the branching program is-- well, maybe I a little bit discussed this already, but that's OK. The branching programs were not read once. The polynomials might have exponents bigger than 1. So where would the proof fail?

Would they fail at the point where b1 and equivalent to b2 implies that they agree on all Boolean inputs? So that's the first step here. Or was it that agreeing on all Boolean inputs implies that the polynomials are the same?

Or would it be that having the two polynomials being equal implies that they always agree? So those are the three steps in the proof of part one. So let's see. What do you think there?

I'm getting a couple of questions about picking the prime number. The prime number here is-- this is a very small prime number. You could even represent that prime number in unary within the amount of time we have, because don't forget, this is a prime number whose magnitude is at most the number of variables.

So you can write that prime number in unary. And finding the prime and testing primality, testing whether the number is prime, is something that you can do even with a brute-force algorithm, and it would be good enough. You don't have to do anything fancy about testing primality in this case.

So why does it have to be prime? You need it to prime in order for it to be a field. So this is just the algebra part. If you did not have a prime number, then some of the field properties don't work. And you may no longer get the fact that the polynomial has a small number of roots. So that's all I can say about that.

Is the polynomial like a hash function for the branching program? Are they equal if they are the same, but sometimes the value is also equal if the programs are different? That's an interesting idea. Is the polynomial acting like a hash function?

I think there is something to what you're saying, but I think it's actually in the other direction. It's related to a hash function, but it's actually acting more like an error-correcting code. Let's save that for later. It's a very good point. It's a very good question. Maybe we can talk about it after if you remind me.

OK, let's end this poll here. Should C be-- if having these two agree implies-- well, I mean, the question is, should we change p1 and p2 always agree to b1 and b2 always agree? Well, b1 and b2 are behaving exactly the way p1 and p2 behave, so I'm not sure it really matters. Too much time on this chat, on this poll here. Let's end this poll-- oh, sharing results.

Yeah, so the correct answer is B, that agreeing on all Boolean inputs implies that they are equal. The other two follow immediately. They're still true. But if it's not read once, even though they agree on all the Boolean inputs, they won't necessarily agree as polynomials.

For one thing, if you just take the two polynomials x1 squared and x1, they agree on all the Boolean inputs, but they're not the same. They agree in the Boolean world, because 0 squared is 0, and 1 squared is 1. But they're not the same polynomial.

Let's move on. Actually, I have another check-in on the same slide here. And this is actually answering a question that I got in the chat. If p1 and p2 were-- how big are these polynomials? These look like they could be big. If they're exponentially large, would that be a problem for the time complexity?

So pick A or B here. We're running out of time here. So why don't I not say too much and just let you go with it. I'm going to close this down. All in? Well, oh, my god, by one point. This is like Georgia here-- do a recount.

In fact, B is correct by a hair. They are not polynomial in size. The truth tables can be very large. As we did with the branching program for the exclusive or case, you don't actually write down the polynomials to evaluate them. You can evaluate them as you're going along.

The polynomials are huge. But you don't have to write down the polynomials to evaluate them. That's only part of the proof. The algorithm doesn't have to deal with the polynomials itself, so maybe good to think about this.

And somebody says, did I invent this proof? No, I did not invent this proof. It's a wonderful proof, but it's not mine. I would love to have been-- get to take the credit for it. So why don't we wrap this up?

Is there some way to simplify the polynomials as we're going along so that we don't end up with them being too big and so that we can then just look at the polynomials? Not that I know of. I think the polynomials are really going to be big.

And so there's not going to be any way to view it just as-- if you could, it would be fantastic, because that would give you a deterministic algorithm. I think the only way that people know how to do this in terms of random inputs to a polynomial, which is too big to write down. If you could write it down and just analyze the polynomials, you'd have a huge, huge result there. Oh, I'm glad people like this proof. That's good.

How many actual quantities are there in the formula? I'm not sure what that means. What formula? I mean, the polynomial is huge. The number of different polynomials-- well, I guess I don't understand the question.

What motivates the idea of arithmetization? What would make somebody think of this? I'm not sure, actually. But we're going to use it even in a more remarkable way in the last two lectures of the course, so stay tuned.

I mean, this is sort of clever, but seems very specialized. But the next part, where we go to the next application, we're going to use this to analyze satisfiability, which is a much more general kind of a situation. And that, I think, is especially remarkable.

For the polynomials p1 and p2, it's only a polynomial number-- I don't think so, because-- well, it depends on the size of the field. The size of-- no, it's going to be something like m to the mth power, right? So those are the number of possible inputs.

Each field element has 3m possibilities, roughly. And there are m field elements, so it's m to the 3m different possible inputs that you're picking at random. So anyway, I'm going to shut this down and move over to the office hours Zoom. Feel free to join me there. Otherwise, I will see you all on Thursday. Take care.