[SQUEAKING] [RUSTLING] [CLICKING]

PROFESSOR: OK, hello, everybody. We'll get started. So just recapping what we did in the last lecture on Tuesday, it was the second part of a two-lecture sequence on the hierarchy theorems for time and space and using the hierarchy theorems to show that there is a problem, which is intractable, that's provably outside of polynomial time. And that was this equivalence problem for regular expressions with exponentiation.

And then we had a short discussion about oracles and the possibility that similar methods might be used to show that satisfiability is outside of P, which would then of course solve the P versus NP problem and argue that it seems unlikely this kind of a meta theorem-- not a well-defined notion-- but it seems unlikely that the methods that were used for proving the intractability of equivalence of regular expressions with explanation could be used to solve P versus NP, at least the diagonalization method in a pure form, whatever that means, that's not going to be enough.

So today, we're going to shift gears again, begin a somewhat different topic, which is really going to be our, again, a few lectures on probabilistic computation, which is going to round out the semester for us. And we're going to start by talking about a different model of computation which allows for probabilism in the measuring the amount of probabilism we're measuring the probabilities allowing for probabilism in the computation defined an associated complexity class-- this class BPP-- and then start the discussion of an example about something called branching programs.

OK, so with that in mind-- so we're going to start off by defining the notion of a Probabilistic Turing Machine or PTM. The Probabilistic Turing Machine is a lot like the way we have thought about not deterministic Turing machines in that it's a kind of a machine that can have multiple choices, multiple ways to go in its computation. So there's not just going to be a fixed deterministic path of its computation, but there's going to be a tree of possibilities.

And for our purposes, we're going to limit the branching in that tree to be either a step of the computation where there's no branching, where it's a deterministic step, as shown over here. So every step of the way leads uniquely to the next step. Or there might be some steps which have a choice. And we're only going to allow for these purposes to keep life simple, having only a choice among two possibilities. And we'll associate to that the notion of a probability that each choice will have a 50-50 chance of getting taken.

And this kind of corresponds with the way some of us or some of you think about non-determinism, which is not exactly right up until this point, is that the machine is kind of taking a random branch. Really, we don't think about it randomly until now. Now we're going to think about the machine as actually picking a random choice among all the different branches that it could make and picking that choice uniformly by flipping a coin every time it has an option of which way to go.

Now you could define-- I'm getting a question here-- a machine that has several different ways to-- more than two ways to go. And then you would need to have a three-way coin, a four-way coin, and so on. And you could define it all that way as well. But it doesn't end up giving you anything different or anything interesting or new for the kinds of things we're going to be discussing. And it's just going to be simpler to keep the discussion limited to the case where the machine can only have two possibilities if it's going to be having a choice at all or just one possibility when there's no choice.

OK, so now, we're going to have to talk about the probability of the machine taking some branch of its computation. So you imagine here, here is the same computation tree that we've seen before in the case of ordinary non-deterministic machines, where you have M on w. There could be several different ways to go. And there might be some particular branch.

But now, we want to talk about the probability that the machine actually ends up picking that branch. And it's going to be when we talk about the machine having a choice of ways to go, we're going to associate that with a coin flip. So we're going to call that a coin flip step when the machine has a possibility of ways to go.

And so on a particular branch, the probability of that branch occurring is going to be 1 over 2 to the number of coin flip states on that branch. and the reason for-- I mean, this is kind of the definition that makes sense in that if you imagine looking at the computation tree here-- and here is the branch that we're focusing on of interest-- every time there's a coin flip on that branch, there's a 50-50 chance of taking a different branch or staying on that branch.

So the more coin flips there are on some particular branch, the less likely that branch would be the one that the machine actually ends up taking. And so it's going to be 1 over 2 to the number of coin flips. And that's the way we're defining it.

Now once we have that notion, we can also talk about the probability that the machine ends up accepting because as before, each of these branches is going to end up at an accept state or reject state, thinking about this only in terms of deciders. And the probability of the machine accepting here is just going to be the sum over all probabilities of the branches that end up accepting. So just add up all of those probabilities of a branch leading to an accept, and we'll call that the probability that the machine accepts its input.

And the probability that the machine rejects is going to be 1 minus the probability that it accepts because the machine, on every branch, is either going to do one or the other, OK? Now if you're thinking about a particular language that the machine is trying to decide, this probabilistic machine now is trying to decide, on each input, some of the branches of the machine may give the correct answer. They're going to accept when the input is in the language. Other branches may give the wrong answer. They may reject when the input is in the language and vise versa. So there's going to be a possibility of error now in the machine in any particular branch. It might actually give the wrong answer.

And what we're going to say is bound that error over all possible inputs, and we'll say that the machine for any epsilon greater than or equal to 0, we will say that the machine decides the language with error probability epsilon if that's the worst that can possibly happen if, for every input, the machine gives the wrong answer with probability at most epsilon.

Equivalently, if you like to spell it out a little bit more, a little differently, for strings that are in the language, the probability that the machine rejects that input is going to be at most epsilon. And for strings in the language the probability for strings not in the language, the probability that accepts is at most epsilon. So again, this is the machine doing the thing that it's not supposed to be doing. For things in the language, it should be rejecting very rarely. For things not in the language, it should be accepting very rarely. And that's what this bound is doing for you, OK?

So let's just see. So we'll talk about-- so I'm getting some questions here about-- so let me just look at these. One second here. Yeah, so probability 0-- so there's a possibility that the machine might have a probability 0, say, of accepting. That means there are no branches that end up accepting or probability 0 of rejecting. There were no rejecting branches.

I think we're going to talk in a minute about the connection between this and the standard notion of NP. So just hold off on that for a second. Also, what about the possibility that the machine is being a decider or running in a certain amount of time? So we will look at time-bounded machines in a second on the next slide or two, talking about machines that run in polynomial time. So that means all branches have to halt within some polynomial number of steps.

So that's where we're going. But for the time being, we're the only looking at deciders, where the machine has to hold on every branch. But some branches might run for a long time. But for now, we're not going to be thinking about machines that have branches that run forever. All of our machines are deciders, so they hold on every branch. See, is there anything else here? No. So why don't I move on?

So let's define now the class BPP using this notion of a probabilistic Turing machine, which is now going to be running in polynomial time. So BPP is going to be another one of this complexity classes, a collection of languages, like P, and NP, and PSPACE, and so on. But it's going to be now associated with the capabilities of these probabilistic machines, the kinds of languages that they can do.

So we'll say, the class BPP is the set of languages, A, that there's some probabilistic polynomial time Turing machines, so all branches have to hold within-- into the k for some k. So some polynomial time probabilistic Turing machine decides A with error possibility, at most, 1/3. So in other words, when it's accepting for strings in the language, the machine has to reject with, at most, 1/3. So it's saying it equivalently. For strings in the language, it has to accept with 2/3 probability.

And for strings not in the language, it has to reject with 2/3 probability, at least, in both cases. OK. Somehow, I ended up with-- I didn't check my animation here. But OK, that's fine. So there is a-- now if you look at the 1/3 here in the definition, it seems strange to define a complexity class in terms of some arbitrary content like 1/3. Why didn't we use 1/4 or 1/10 in the definition of BPP and say the machine has to get have an error with at most 1/10 or 1/100? Well, it doesn't matter.

And that's the point of this next statement called the amplification lemma, which says that you can always-- if you have a machine that's running in a certain polynomial time that's running with a certain error, which is, at most, 1/2. If you have an error 1/2, it's not interesting because the machine could just flip a coin for every input, and it could get the right answer with probability of 1/2. So probability 1/2 is not interesting. You have to have probability strictly less than 1/2 for the machine to actually be doing something that's meaningful about that language.

So if you have a probabilistic Turing machine that has error, let's say, epsilon 1, which is, at most, 1/2-- which is less than 1/2, then you can convert that to any error probability you want for some other polynomial time probabilistic Turing machine. So you can make that error, which maybe starts out as 1/3, and you can drive that error down to 1 over a googol. And seriously, you can really make the error extremely, extremely small using a very simple procedure.

And that's simply this-- so if you're starting out with a machine that has an error possibility of 1/3, say, so that means 2/3 of the time it's going to get the right answer, and, at most, 1/3 of the time-- the least 2/3 of the time, the right answer, at most, 1/3 of the time, the incorrect answer, whether that's accepting or rejecting. And now you want to get that answer down to be something much-- that error down to something much smaller.

The idea is, you're going to take that machine, and you're going to run it multiple times with independent runs, if you want to think about it more formally speaking. But it's intuitive. You're just going to run the machine, tossing your coins. Instead of just running it once, you're going to run the machine 100 times or a million times. But you can do it at just, say, it's a constant factor.

And even 1,000 times is going to be enough to increase your confidence in the result tremendously because if you run the machine 1,000 times, and 600 of those times, the machine accepts, and 400 of the times, the machine rejects, it's very powerful evidence that this machine is biased toward accepting, that it's accepting most of the time. So if it had an error probability, at most, 1/3, the probability that you're seeing it accept 600 times when, really, 2/3 of the time, it's rejecting overall, is extremely unlikely.

And you can calculate that-- which we're not going to bother to do, but it's a routine probability calculation-- to show that the probability that if you run it a whole bunch of times, and you see the majority coming up, which is not the right answer, the probability of that is extremely small. So I'm not saying that very clearly. But the method here is you're going to take your original machine, which has error probability 1/3 or whatever it is-- maybe has error probability 49%.

And you run it for a large number of times. And then you take the majority vote. And you're sampling the outcomes of this machine. And if you take enough samples, it's overwhelmingly likely-- since you're just doing them uniformly-- you're taking those samples uniformly, it's overwhelmingly likely that you're going to be seeing the predominant one come up more often.

And exactly what that right value is, we're not going to bother to calculate. But that's something that, I will refer you to the textbook, or this is the kind of thing that comes up in any elementary probability book and sort of very intuitive. So I don't want to spend the time and do that calculation, which is not all that interesting.

OK, so just one quick question here that I'm getting-- what happens if you bound-- if the error is greater than a half? I don't think that because we're bounding the error, so we're not saying the error actually is one, like, 60% on everything because then if you knew the error was 60% guaranteed, you can always just flip your answer around and get your error to be 40%. But I'm saying the error's at most whatever epsilon is. And so if you're saying the error is, at most, 60%, it doesn't tell you anything. OK.

Another question is, does the amplification lemma also justify that the choice of model with binary branching choices is equivalent to any other? Perhaps, you could say that because you can change those. If you had three-way branching, you can simulate that with two-way branching to any accuracy that you want. You're not going to get it down to zero, but you're going to get it very close. So maybe it's the amplification lemma. Maybe it's, yeah, sort of all related. OK, let's move on.

So the way that it's helpful to think about this class, let's contrast it with the other model of non-deterministic computation that we have is non-determinism, is NP. So non-deterministic, the model of non-deterministic polynomial time computation was NP, the other class. And so the way-- I think one way to look at or think about non-determinism in the case of NP is, for strings in the language for your NP Turing machine, there's at least one accepting branch. So I'm indicating the accepting ones in green and the non-accepting one, the rejecting branches in red.

So you could have almost all of the branches be rejecting branches for strings in the language as long as there is at least one accepting branch. That's just the way non-determinism works. The accepting branch overrules all of the others. It's only when you're not in the language that all of the branches turn out to have to be rejecting. That's when the rejecting has no accepting branch to overrule it.

But the situation for BPP is a little different-- is different. There, it's kind of the majority rules. So in the case for strings in the language, you need to have a large-- or the overwhelming majority of the branches have to be accepting. And for strings not in the language, the overwhelming majority have to be rejected.

What you're not going to allow in the case of BPP is kind of an in-between state where it's sort of 50-50 or very, very close to 50-50. Those kinds of machines don't qualify as designing a language in BPP. They always have to lean one way or lean the other way for every input. Otherwise, you won't be able to do the amplification. So you need to have some bias away from half in accepting or rejecting.

So let me-- so I was going to ask a check-in, I think, at this point. Yes, let's. So just thinking about BPP, I hope I was clear. So if there's questions about that, I think I've somehow didn't-- I'm not sure I described it totally well here. So I'm going to ask a few questions about BPP. But if you have any questions for me first, go ahead.

OK, why don't we just run the check-in? Let me launch this, and then I can answer questions as you're asking. Did I start that? Yeah. OK, so you have to check all of these that you think are true. Can you think of non-deterministic Turing machines as try all branches at once and get the right answer? And BP gets only one branch? No. I would say a little differently. I would say, I would think of non-determinism as you can still just try one branch, but you always guess the right one. So there's some sort of magical power that allows you always to guess the right answer if there's a right guess If you're in the language.

In the case of BPP, you're going to be picking a random branch no matter what. And you know that the random branch is likely to give the right answer but not guaranteed. And the amplification lemma tells you you can arrange things so that it's extremely likely that the random branch is going to give you the right answer. OK, let's see. How we doing on our check-in here? Got a lot of support for all candidates. And I'll give you another-- a little bit of time here because there's a bunch of questions. It's almost like four check-ins at once. But we have two more real check-ins coming later.

OK, so why don't we come and let's give it another 10 seconds, and then I'm going to stop. Closing down-- 1, 2, 3, close.

OK, so we've got a lot of support here. And in fact that's good because all of them are true. Some of them are easier to see than others. So first of all, c is very easy to see because that's going to be a machine that has the correct answer all of the time. So that's error probability zero on both accepting and rejecting.

This is a little harder. D is a little bit harder to see that it's in PSPACE. But you could calculate for every branch what its probability is. And you can just go through all the branches and add up all those probabilities in a PSPACE machine. So you have to think about it a little bit. But d is not too hard to see either. Closure under complement-- if you just take your BPP machine and you flip the answer on every branch, that's typically doesn't work in ordinary non-determinism, but it does work here because it's going to change a bias toward accepting into a bias toward rejecting and vice versa. So BPP is closed under complement.

And closure under a union-- it kind of follows from the amplification lemma. As long as you can make the probability extremely small, then you can just run the two different machines. And even though the they each may make a mistake cumulatively, the total, the probability that each one of them-- that either of them will make a mistake is still small. And so you can just run the two machines and take the or of the responses that they get, and it's still very likely to give you the right answer for the union.

OK, let's continue. So what I'm going to do now for the rest of the lecture is-- and it's actually going to spill over into the lecture after Thanksgiving because this is going to introduce an important method for us-- is to look at an example of a problem that's in BPP. I love to teach things by using examples, and so this is a very good example because it has a lot of meat to it. And it's a very interesting example. In general, proving things in BPP, which are not trivially there because they're already in P, they tend to be somewhat more involved than some of the other algorithms we've seen. So there are no simple examples of problems in BPP which are not already in P.

So this is one example that we're going to go through of a problem in BPP that's not known to be in P. Of course, things could collapse down, but as far as we know, this language is not in P. So let's see what the language is.

It has to do with these things called branching programs. The branching program is a structure that looks like this. So let's understand what the pieces are. First of all, it's a directed graph. And we're not going to allow-- there are no cycles allowed in this graph. It's a directed acyclic graph-- so no loops allowed. And the nodes are in two categories. There are query nodes, which are labeled with a variable letter, and output nodes, which are labeled either 0 or 1.

And lastly, one of the query nodes is going to be-- or one of the nodes is going to be designated as a start, OK? And so what you do is, the way-- this is a model of computation. And the way we actually use a branching program is we have some assignment to the variables. That's going to be the input. So you take all of the variables. There are three variables in this case-- x1, x2, and x3. You give them some truth assignment.

So let's say, zero's a 1. And so x1 is 0. x1 is 1, or whatever. And once you have the truth assignment, you start at the start variable, and you look at its label. And you see what value the input has assigned to that variable. So if x1 is assigned a 1, you're going to follow down the 1 branch. If it assigned a 0, you go down the 0 branch. And then when you get down to the next node, that's another variable that you're going to have to query depending upon what the input assignment is.

And you're just going to continue that process. Because there are no cycles, you're going to end up at one of the output nodes because all of the variable nodes-- all the query nodes have two outgoing edges, one labeled 0 and one labeled 1. So you're going to eventually end up at an output node, and that's going to be the output of the branching program. So let's do a quick example.

So if x1 is 1, x2 is 0, and x3 is 1-- so again, we start at the start variable-- the start node that has the indicated with the arrow coming in from nowhere. So you're going to start at the nose labeled x1. So you have to look and see, what is x1 in the input? It's a 1. So you're going to follow down the 1 branch.

Now you see the next node. Oh, that's an x3. We see, what's x3 in the input? x3 is a 1. So go down to 1 branch again. Now we have an x2 node. Take a look at the input. x2 is a 0. We follow the 0 branch. Now, you're at an output branch, an output node. So that's a 0, and that's the output of this computation.

So writing it this way and thinking about it as a Boolean function which maps strings of zeros and ones, we have f of 101 representing those that assignment. That equals 0. That was the output. And that's the output of this computation, OK? So important to understand. We're going to spend a lot of time talking about branching programs-- so critical to understand this model. I think it's fairly simple. But if you didn't get it, please ask. We can easily correct up any misunderstanding at this point.

It's not exactly the same as a DFA. DFAs, for one thing, can take inputs of any length. This has inputs of some particular length, where the branching program has some fixed number of variables. This one has three. So this only takes inputs of length 3. So there's maybe some connection to thinking of these as states and so on, but it's a different model.

So now, we'll say that two branching programs-- OK, let me just ask one more question. Not all nodes need to be used, right? Yeah, I mean, there's no requirement that all nodes need to be used. And there even could be inaccessible nodes. I'm not preventing that. That could be OK.

So on the particular branch, certainly, you're not going to-- when you're executing this branching program on an input, obviously, certainly, you're going to have a path that's going to only use some part of the tree, a part of the graph. But there might be some paths that can never occur. So if you went down x equal to 1 here, and then x3 was 0, now, you're re-reading x1, so you wouldn't go down this branch unless you-- I think all of the branches in this particular branching program could get used. But I didn't check that, so maybe I'm wrong.

OK, so let's continue. Two branching programs may or may not compute the same function. We'll say they're equivalent if they do. Now two branching programs can be equivalent even though they superficially look different from one another. And we're interested in the computational problem of, given two of these branching programs, do they compute the same function?

In other words, do they always give the same answer on the setting of the input? So we'll define the associated language. Equivalence problem for branching programs says that you're given two of these branching programs, and they're equivalent to be in the language. We're going to sometimes write equivalents using the mathematical notation of the three-lined equals sign, equivalence sign.

OK, that means a computer saying-- they always give the same answer. Now that problem turns out to be coNP-complete, as I've asked you to show on your homework, I believe. This is not a super hard reduction. And coNP complete, by the way, is the complement of an NP-complete problem. Or equivalently, it's a problem to which all coNP problems are polynomial time-reducible, and it's in coNP.

So this is coNP-complete, and that's for you to show. But that has an important significance for us right now because if-- looking at the question of whether this problem is in BPP, the fact that it's coNP-complete suggests that the answer is no because if a coNP-complete or NP-complete problem more in BPP because everything else in NP or coNP is reducible to that problem, then all of those NP or coNP problems would be in BPP for exactly the same reason that we've seen before. And that's not known to be the case and not believed to be the case. So we don't expect that coNP-complete problem is going to be in BPP. That would be an amazing and surprising result.

So because I hope I made it clear in my previous discussion that the BPP, from a practical standpoint, is very close to being like P because you can make the error probability of the machine so incredibly low that it's a comparable-- if you run the machine and the error probability is like 1 over googol, then it's sort of even greater than the probability that some alpha particle came in and flipped the value of some internal memory cell in your computation.

So if you have an extremely low error probability, it's pretty good from a practical standpoint. So it would be amazing if NP problems were solvable in BPP. So this is not the language we're going to use as our example. We're going to look at a related, restricted version of this problem about equivalence for branching programs. And that, I'm going to introduce right now. OK, any questions here? I don't see any questions. I'm fading out.

OK, so let's move on. So we're going to talk about branching programs that are what are called read-once, read-once branching programs. And those are simply branching programs that are not allowed to reread an input that they've previously read. So for example, is this branching program a read-once branching program? No. This branching program is not a read-once branching program because you can find a path that's going to cause you to read the same variable more than once.

So it's not going to be a read-once. So over here, it's not read-once because there's two occurrences of an x1 on the same branch. Now you might ask, why would anybody want to do that because you've already read the value of x1? Well, in the case of this particular branching program, there might be a value because you could have got to this x1 branch by going this way or that way.

But that's a separate question. If we restrict our attention to read-once branching programs, then the problem of testing equivalence becomes very different in character. And in fact, we're going to give a probabilistic algorithm-- a BPP algorithm to solve that problem. So the equivalence problem for read-once branching programs, which are not allowed to reread variables on any path, that's interestingly going to be solvable with a probabilistic polynomial time algorithm with a small error probability.

So I'm going to run a check-in now. But let's make sure we're all together on this. So I've got a good question here. Can every Boolean function be described by a branching program? Yes. That's an easy exercise. But you can make-- branching programs are-- they may be a large, but you can describe any Boolean function with some branching program. That's not hard to show. Other question-- are we all together on understanding what read-once means, and branching programs, and all that stuff? This is a good time to ask if you're not.

OK, so let's do the check-in. So as I pointed out, we will show that the equivalence for read-once branching programs is solvable in BPP. Can we use that to solve the general case for branching programs by converting general branching programs to read-once branching programs and then running the read-once test? So what do you think? OK, I'm seeing a lot of correct answers here. So let's wrap this one up quickly. So another 10 seconds, please. OK? Are we all ready? 1, 2, 3, closing.

All right, yes. Most of you have answered correctly. Well, answer A is not a very good answer because we already commented on the previous slide that we don't know how to do the general case in BPP, so it would be kind of surprising if, right here, I'm saying, yes, we could do it by using the restricted case. So I think a better answer would be to one of the no's, but as I did comment, you can always convert-- you can always do any Boolean function with a-- well, maybe I didn't say it for read-once branching programs.

But even read-once branching programs can compute any Boolean function. So the conversion is possible but, in general, will not be polynomial time. And if you imagine even trying to do the conversion over here, you could convert this branching program to read-once, but you'd have to basically separate the two. You know, instead of rereading the x1, you could remember that x1 value.

But then you would not be-- you couldn't converge over here. You'd have to keep those two threads of the-- those two branches of the computation apart-- those two paths apart from one another. And already, the branching program would start to increase in size by doing that. And so, in general, converting is possible, but it requires a big increase in the size. And then it will not allow a polynomial time algorithm anymore, even in the probabilistic case.

OK, so now, let's start to look at the possibility of showing that this equivalence problem is solvable in BPP. And it's going to take us in a kind of a strange direction, but let's try to get our intuition going first by doing something which seems like the most obvious approach. So here, so we're going to give an algorithm now, which is going to be an attempt. This is not going to work, but nevertheless, it's going to have the germ of the right idea or not the germ but the beginning of the right way to think about it.

So here are the two read-once branching programs, B,1 and B,2. And I want to see, do they compute the same function or not? So one thing you might try is just running them on a bunch of randomly selected assignments or inputs. So you can just take two random input assignments. Just take x1, flip a coin to say it's 1 or 0. Do the same for x2 and so on. Then you get some input assignment. You run the two branching programs on that assignment.

And maybe that doesn't give-- even if they agree, it doesn't give you a lot of confidence that you get the right answer, that they're really equivalent. So you do it 100 times, whatever-- some number of times. And of course, if they ever disagree on one of those assignments, then you know they're not equivalent, and you can immediately reject.

But what I'd like to say is if they agree on those 100 tries or those 100 assignments there, then they are, at least I haven't found a place where they disagree, so I'm going to say that they're equivalent. Is that a reasonable thing to do? Well, it might be. It depends on k. So the critical thing is, what value of k should you pick which is going to be big enough to allow us to draw the conclusion that if you run it for k times, and you never see a difference, then you can conclude, with good confidence, that the two branching programs are equivalent because you tried to look for a difference, and you never found one.

Well, the thing is that k is going to have to be pretty big. So looking at it this way, if the two branching programs were equivalent, then, certainly, they're always going to give the same value. So the probability that the machine accepts is going to be one. And that's good because we want for-- this is a case when we're in the language. We want the probability of acceptance to be high. And here, the probability of acceptance is actually 1. So it's always going to accept when the two branching programs were equivalent.

But what happens when the branching programs are not equivalent? Now, we want the probability of rejection to be high. The probably of acceptance should be very low, right? So if they're not equivalent, what we want-- the probability that the machine rejects is going to be high if they're not equivalent because that's what the correct answer is. Well, the only way the machine is going to reject-- if it finds a place where the two branching programs disagree.

But those two branching programs, even though not equivalent, might disagree rarely. They might only disagree on one input assignment out of the 2 to the n possibilities. So these two in-equivalent branching programs might agree almost everywhere, just except at one place. And then that's enough for them not to be equivalent. But the problem is that if you're just going to do random sampling, the likelihood of finding that one exceptional place where the two disagree is very low. You're going to have to do an enormous number of samples before you're likely to find that point of difference.

And so in order to be confident that you're going to find that difference, if there is one, you're going to have to do exponentially many samples. And you don't have time to do that with a polynomial time algorithm. You're just going to have to flip too many coins. You have to run too many different samples, different assignments through these two machines. And because they're different, but they're almost the same. So we're going to need to find a different method.

And the idea is we're going to run these two branching programs in some crazy way. Instead of running them on 0's and 1's that we've been doing it so far, we're going to feed in values for the variables which are non-Boolean. They are going to be-- we're going to set x1 to 2, x3 to 7, x4 to 15. Of course, that doesn't seem to make any sense. But it's nevertheless going to turn out to be a useful thing to do, and it's going to give us some insight into the equivalence or in-equivalence of these branching programs.

OK, so let's just-- I think-- are we at the end of? Yeah, we're at the break here. So I'm getting some questions coming in, which is great. I will answer those questions. But why don't I start off our break and then-- OK.

OK, so there's a question about whether this machine runs deterministically or not. So which machine are we talking about? So the branching programs themselves, they run deterministically. You give them an assignment to the input variables that's going to determine a path through each branching program, which is eventually going to output a 0 or a 1. And you want to know, do those two branching programs always give the same value no matter what the input was? But the branching programs themselves were deterministic.

Now the machine that's trying to make the determination of whether those two branching programs are equivalent, that machine that we're going to be arguing is going to be a probabilistic machine. So it's a kind of non-deterministic machine that's going to have different possible ways to go depending upon the outcome of its coin tosses. So you can think of as non-determinism in the ordinary sense, that it has a tree of possibilities.

But now the way we're thinking about acceptance is different. Instead of accepting, if there's just one accept branch, the machine, for it to accept, has to have a majority of the branches be accepting. And so there's some similarities but some differences with the usual way we think of non-determinism.

So what's the motivation behind introducing this type of Turing machine? Well, I mean, I guess there are two motivations. Probabilistic algorithms, sometimes called Monte Carlo algorithms, turn out to be useful in practice for a variety of things. And so that led to people to think about them in the context of complexity. They're related in some ways to quantum computers, which are also probabilistic in a somewhat different way. But they also have a very nice formulation in complexity theory.

So complexity theorists like to think about probabilistic computation because, I mean, you can do interesting things with probabilistic machines. And the complexity classes associated are also interesting. So as you'll see, it leads us in an interesting direction to consider how to solve this problem, this read-once branching program problem equivalence with a probabilistic machine. It's just an interesting algorithm that we're going to come up with.

So in our proof attempt, where did we use the probabilistic nature for BPP? Because we're running the two branching programs on a random input. I mean, so you have your two branching programs. You pick a random input to run those two branching programs, and you see what they do. That's why it's probabilistic. When you think about random behavior of the machine, that's a probabilistic machine.

So each branch of the machine is going to be like the way we normally think about non-determinism. Somebody is asking whether we think of the complexity of the machine in terms of all of the branches of the machine or each branch separately. We always think about, for non-deterministic machines, each branch separately. I'm not totally sure. I understand the question there. So are all the inputs built-in, and we randomly choose one through coin flips?

Not sure I understand that question either. We're given as input the two branching programs. And then we flip coins using our non-determinism-- you can think about it equivalent in terms of coin flips-- to choose the values of the variables. So now, we have a set of variable inputs to the values of the variables. And we use that as input to the branching programs to see whether they-- to see what answers they give, and, in particular, whether they give the same answer on that randomly-chosen input. Let's move on.

All right, so now moving us toward the actual BPP algorithm for read-once branching program equivalence testing, we have to think about a different way to-- we need an alternate way of thinking about the computation of a branching program. It's going to look very similar, but it's going to lead us in a direction that's going to allow us to talk about these non-Boolean inputs that I referred to. Just kind of where we're going-- we're going to be simulating branching programs with polynomials, if that helps you as an overarching plan. But we'll get there a little slowly.

So OK, here's a read-once branching program. We're not going to use the read-once feature just yet, but that'll come later. But anyway, here's a branching program. Oh, here's my branch and my-- I crashed here. I'll start that again.

OK, so we take an input, whatever it is, and thinking about the computation of the branching-- so we're not thinking about the algorithm, right now. We're just thinking about branching programs for the minute. We're going to get back to the algorithm later.

So the branching program follows a path, as I indicated, when you have a particular input. If x1 is 0, x2 is 1, x3 is 1, so the output is going to be 1 in this case. OK? So the way I want to think about this a little differently is I want to label all of the nodes and all of the edges with a value that tells me whether or not this yellow path went through that node or edge. It's going to be just doing the same, but you may think this is no difference at all.

But I want to label all of the things on the yellow path, I'm going to label them with a 1. And all of the things that are not on the yellow path, I'm going to label with a 0. So I'm trying to keep those labels apart from the original branching program, which are written in white. These labels are written in yellow.

But these labels have to do with the execution of the branching program on an input. So once I have an input, that's going to determine a 1 or a 0 label for every node and edge. Now if we want to look at the output from this branching program after we have that labeling, we only have to look at the label of the one output node because if that one has a 1 on it, that means that the path went through that 1. And so therefore, the output is 1.

So I'm going to give you another way of assigning that. Instead of just coming finding the path first and then coming up with the labeling afterward, I'm going to give you a different way of coming up with that labeling, kind of building it up inductively, starting at the start node and building up that labeling. You'll see what I mean by my example.

So if I have a label on this node, so I already know whether or not the path went through that node, label 1 means the path went through it. Label 0 means the path does not go through it. That's going to tell me how to label the two outgoing edges. So if I've already labeled this with a, where a is a 0 or a 1, then what expression should I use? Under what circumstances will I label-- what's the right label for this one outgoing edge here?

Well, if a is 0, that means we know the path did not go through this node. So there's no way it could go through that a edge. Similarly, if xi is a 0, that means, even if we did go through that node, the path would go through the other outgoing edge and not through this one. So that tells us that the Boolean expression which describes the label of this node in the execution is going to be the and of the value on the node and the query variable of that node.

Now think about, what's the right way to label the other edge, the execution value of the other edge? Again, you have to have-- go through this node, so a has to be 1. But now, you want xi to be 0 in order to go through that edge. So that means it's going to be a and the complement of xi. OK? So this is going to just tell me this. I'm writing a formula for how we labeling these edges based on the label of the parent node.

Similarly, if I have a bunch of edges where I already know the values, the labels, the execution labels there, let's say, so I have a1, a2, and a3, what is the right label to put on this node? Well, if any one of those is a 1, that means the path went through that edge. And so therefore, it's going to go through that node. So that tells us that the label to put on that node is the or of the labels on the incoming edges. OK? Questions on this?

So now this is setting the stage for starting to think about this more toward polynomials instead of using a Boolean algebra. So I'm getting a question-- how do we know what the execution path is, which nodes to label? We're going to be labeling all of the nodes. So we start off with labeling the-- did I say that here? We start up-- I didn't say it, but I should have. We labeled the start node with 1 because the path always goes through the start node. So without even talking about a path, we just label the start node 1.

Well, maybe we'll do an example of this also. But now, once we label this start-- this node 1, we have an expression that tells us how to label the two outgoing edges, this edge and that edge. And I'm doing it without knowing the values of the variables. I'm just making an expression, which is going to describe what those labels would be once you tell me what the input assignment is. OK so I'm just sort of-- it's almost like a symbolic execution here. I'm just writing down the different expressions for how to calculate what these things should be.

Let me-- maybe this will become clearer as we continue. So now this is the big idea of this proof. We're going to use something called arithmetization. We're going to convert from thinking about things in the Boolean world to thinking about things in the arithmetical world, where we have arithmetic over integers, let's say, for now. So instead of ands and ors, we're going to be talking about pluses and times. And the way we're going to make the bridge is by showing how to simulate the and and or operations with the plus and times operations.

So assuming 1 means true and 0 means false, if you have the expression a and b as a Boolean expression, we can represent that as a times b using arithmetic because it computes exactly the same value when we have the Boolean representation of true and false being 1 and 0. So 1 and 1 is 1. And 1 times 1 is 1. And anything else-- 1 and 0, 0 and 1, 0 and 0-- if you applied the times operator, you're going to get the same value. So times is very much like and in this sense.

OK, we're going to write it as just ab, usually, without the time symbol. So if we have a complement, how would we simulate that with arithmetic? Well, again, here we're just flipping one and 0 in using the complement operation that's going to be the same as subtracting the value from one that also flips it from between 1 and 0.

How about or-- if you have a or b? Well, it's slightly more complicated because you use a plus b, but you have to subtract off the product because what you want is this simulation should give you exactly the same value. So if you have 1 or 1, you want that to be a 1 answer. You don't want it to be a 2. So you have to subtract off the product.

And the goal is to have a faithful simulation of the and or by using plus and times. So you get exactly the same answers out when you put in Boolean values here. OK, so just to say where we're going, what this is going to-- superficially, we haven't really done anything. But what this is going to enable us to do is plug in values which are not Boolean because it doesn't make sense to talk about-- it makes sense to talk about 1 and 0, but it doesn't make sense to talk about 2 and 3. But it does make sense to talk about 2 times 3. And that's going to be useful.

OK, so let's just see. Remember that inductive labeling procedure that I described before, where I gave the execution labels on the edges depending upon the label of the parent node and which node, which variable is being queried. So if I know that this value is an a, but now the-- OK, so I'm just going to write this down using arithmetic instead of using Boolean operations.

So before, we have-- this was a and xi, If you remember from the previous slide. Now what are we going to use instead because we're going to use this conversion here? Instead of and, we're going to use multiplication. That's just a times xi. What about on this side? Here was and a and the complement of xi. Now the complement of xi is 1 minus xi arithmetically. So this becomes a times 1 minus xi, OK?

Similarly, here, we did the or to get the label on the node from the labels of its incoming edges. Now, we're going do something a little strange because we have a formula here for or. But for technical reasons that will come up later, this is not a convenient representation for us. What I'm going to use instead of this one-- I'm just going to simply say, let's take the sum.

Why is that good enough? In this case, this is still going to be a faithful representation and give the right answer all of the time. And that's because for our branching programs, read-once or otherwise, read-once is not coming in yet, for our branching programs, they're acyclic. So they can never enter a node on two different paths.

There's, at most, one way to come into a node on a path through the-- on an execution path through the branching program. If it comes in through this edge, there's no way for this edge to also have a path because that means you have to go out and come back and have a cycle in the branching program, which is disallowed. So at most, one of these edges can have the path go through it.

So at most, one of these a's can be a 1. The others are going to be 0. And therefore, just taking the sum is going to give us a value of either 0 or 1, but it's never going to give a value higher. And so you don't have to subtract off these product terms. A little bit complicated here. If you didn't totally get that, don't worry for now. We're more concerned that you get the big picture of what's going on.

OK, so I think we're almost-- let me just see how far we are. Yeah. So I'm just going to work through an example. And I think that'll bring us-- let's just see, Do we have any questions here? Not seeing any. That means you're either all totally understanding, or you're totally lost. I never can tell. So feel free to ask a question, even if you're confused. I'll do my best.

OK, maybe this example might help. So now, what we're going to do is, using this arithmetical view of the way a branching program's computation is executed when we're running an input through it, this is going to allow us now to give a meaning to running the branching program on non-Boolean inputs. So maybe this example will illustrate that.

So let's just take this particular branching program here, OK? This branching program it's just on two variables, x1 and x2, and it actually computes a familiar function. This is the exclusive or function if you look at it for a minute. You'll see that this is going to give you x1 exclusive or x2. So it's going to be 1 if either of the x1 or x2 are 1, but it's going to be 0 if they're both 1. That's what this branching program computes. Now, but let's take a look at running this branching program instead of on the usual Boolean values, let's run it on x1 equal to 2 and x2 equal to 3.

Now a common confusion might be that you're looking-- when you do the x1 query, you're looking for another outgoing edge, which is labeled 2. No, that's not what I'm doing. What I'm doing here is, I'm somehow, through this execution, by assigning these other values, I'm blending together the computation of x1 equal to 0 and x1 equal to 1 together. I don't know if that makes any sense. But let's look through the example. So first of all, these are the labeling rules that I had from the previous slide when I used plus and times instead of and and or, OK?

Now, I'm going to show you how to use that to label the nodes and edges of this graph based on this input. And that will determine an output would be the value on the 1 node. OK, so we always start out by labeling the start with 1. That's just a rule. And OK, sorry. Let's think about it together before I blurt out the answer. What's going to be the label on this edge? So this is one of the outgoing edges from a node that already has a label. So that's going to be this case, here.

And what we do is if we take the label of that node, and since it's a 1 edge that's outgoing, we multiply that label by the value of that variable of the assignment to that variable. So x1 is 2. So we take-- the a here is 1. x1 is assigned to 2. So it's going to be 1 times 2 is going to be the execution value we put on this edge. So it's going to be 2. What's going to be the value we put on the other edge, the 0 outgoing edge from x1? So I want you to think about that for a second.

So now, we're going to use this expression. It's a times 1 minus xi. And so xi, again, is 2. So 1 minus xi is 1 minus 2. That's how compliment-- OK, well, let's save that.

So it's 1 minus 2. So that's minus 1 times the label 1 here. So you get minus 1 is the label on this edge. Now keep in mind that if I had plugged in-- and this is very important-- if I had plugged in Boolean values here, I would be getting out the same Boolean values that you would get just by following through the path. The things on the path would be 1. The things off the path would be 0.

But what's happening here is that there's still a meaning when the inputs are not Boolean. So let's continue here. How about-- what's going to be the value on this node? So think with me. I think it'll help you. So now, we're using this rule here. We add up all the values on the incoming edges. There's only one incoming edge, which is value 2. So that means this guy is going to get a 2. And similar on this one. This guy is going to get a minus 1.

Now, let's take a look at this edge. So this is the 0 outgoing edge from a node labeled 2 with label x2. So this is the 0 outgoing edge. The label is 2, so it's going to be 2 times 1 minus the x2 value. x2 is 3. So 1 minus 3 is minus 2. It's going to be 2 times minus 2, which is minus 4. So similarly, you can get the value here, the value on the one outgoing edge is going to be 2 times the x2 value, which is 3, so that's going to be 6. And these two here-- so now, we have a minus 1. And the outgoing is a 0 edge, so it's 1 minus 3. And here, it's going to be 1 times minus 3. No. 1 times 3. I'm sorry. 1 times 3. The answer's minus 3.

So now what's the label on the 0 output edge? So you have to aid up the two incoming edges here. So we have this edge here was a 2. This edge coming in here is a 6, so it's going to be 2 plus 6. It's 8. What about this edge-- this node here? This is an important node because this is going to be the output. So it has minus 3 coming in and a minus 4 coming in, so you add those together. You get a minus 7.

I mean, you may wonder, what-- [LAUGHS] what in the world is going on here? Is this a lot of mumbo jumbo? But we're going to make sense of all this-- not today. We're going to have to argue why this is-- what the meaning that we're going to get out of this is going to be. But the point is that this is going to lead to a new algorithm for testing. This is, again, getting back to what we were doing.

This is the equivalence problem for read-once branching programs. So now, what the new algorithm is going to do is going to pick a random non-Boolean assignment. So it's going to randomly assign values to the x's and to some non-Boolean values. Instead of zeros and ones, we're going to plug in random integer values. We'll make that clear next time what the domain is going to be.

And then once we have that non-Boolean assignment, we're going to value B1 and B2. And if they disagree out there in that extended domain, then we have to show that they're not equivalent, and then we'll reject. And we'll also show that if they were equivalent, then even when we evaluate them, we have to show that if they're not equivalent, that they're very likely to have a difference in the non-Boolean domain. And so if they agree, it gives you evidence that the two are really equivalent. So the completeness proof will come after Thanksgiving. So with that, I'm going to wish you all a nice break. Oh, we have a check-in here. Sorry.

Oh, yeah, this is a good one. I don't know if you're following me, but if I plug in 1 for x1 and y for x2-- do the inputs in the assignment need to be distinct? No. It could be the same value. I could be 2 and 2 here. That's perfectly valid.

But here, I'm going to plug in 1 for x1, and I'm going to plug in a variable for x2-- y. And I'm going to do the whole calculation that I just did. And now, what's going to be the output? And I mean, this looks like a pain to figure out. You could do it. It looks like a pain. But let me give you a big hint. Remember that this thing is supposed to be calculating. The original branching program calculates the exclusive or function. And that means when I plug in a Boolean value, I should get the exclusive or value coming out. So if I already know that x1 is 1, which of these is consistent with getting a value that the exclusive or function would compute?

So let me launch a poll on that. So we're out of time. So let's just let this run for another 10 seconds. OK, I'm going to close this. Ready? Yes, indeed, a is the right answer because that's one-- if I know that one variable is 1, then the exclusive or is going to be the complement of the other variable, which is 1 minus y. So that's what you would get if you calculated this.

OK, so this is what we did today. And feel free to ask questions. So we're going to spend a good chunk-- I'll review this, what we've done so far, but then we're going to carry it forward and spend a good chunk of Tuesday's lecture after the Thanksgiving break proving that this procedure that I just described worked and works. And it's an interesting but somewhat-- it's not such an easy proof. So we're going to spend-- try to do it slowly and clearly. But this notion of arithmetization is going to be-- it's an important notion in complexity. And so we'll see it again coming up in another proof afterward about interactive proof systems.

OK, so please ask questions. So the output is the value of the output 1 state, yes. There was a question I got. Other questions? Somebody saying, minus 7 is not the XOR of 2 and 2. [LAUGHS] What is the XOR of 2 and 3?

So by the way, I should say, we kind of ran a little short on time, I'm not saying that we discovered some fundamental new truth about XOR here because that would be bizarre. It really depends on the arbitrary decision that we made to say true is 1 and false is 0. We could have come up with a different representation for true and false. And then you would get a different value for XOR coming out of that-- from the arithmetization that I just described. But for this particular way of representing true and false, that's how XOR and this particular branching program, that's how XOR evaluates.

The remainder of the proof-- so somebody is asking, which is true, the fundamental theorem of algebra, which talks about polynomials and the number of routes that you can have-- that's going to be critical. So that is the fundamental theorem of algebra. That's where we're going. Good question. Somebody is complaining that we're not taking the digit binary representation of 2 and 3 and taking the bit by bit XOR. Binary representation is not a part of this. We're thinking of these as two elements of a finite field, which we'll talk about later. The binary representation is not entering into this discussion.

So I'll talk about why just doing the sum is enough. I think that was-- so why is it-- I mean, here it is. Why is just doing the sum when I'm looking at how to describe the value of this node based upon the values of all the incoming nodes. And remember, the starting point of this is that we have to faithfully represent the Boolean logic with the arithmetic. And then we're going to use that and extend it to non-Boolean values. But as a starting point, we have to faithfully represent the Boolean values.

Now the Boolean values, on the incoming edges, at most, one of them can be a 1 because the 1's correspond to the edges of the execution path. And you can't make an execution path that's going to have two branches-- that's going to go through a node twice because then you have a loop. And we don't have-- there's no cycles allowed. OK, so I think we're-- it's at 4:00. I want to say farewell to you all. Have a great week. And I'll see you when you get back.