[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: Hi, folks. Welcome back. So we will continue our discussion that we had-- that we've been doing for the past few lectures. We first talked about time complexity. And then we shifted gears to talk about space complexity. So we had a couple of lectures on PSPACE, kind of culminating in proving that there were languages which are PSPACE complete, namely this TQBF language, which is where we started.

And then we also proved that there are problems involving games, such as the generalized geography game, where determining which side has a winning strategy is PSPACE complete. At the end of the lecture last time, we moved to a different regime of space, namely from polynomial space down to log space. And we introduced the classes L and NL.

And so I'm going to begin today's lecture by reviewing some of the material on N and NL, which I think came a little too quickly last time. And then we have two important theorems we're going to cover today. One is about proving that there are complete languages for NL that has a bearing on the L versus NL question-- log space, deterministic log space, versus nondeterministic log space. That's yet another unsolved problem in our field. And so there is a notion of a complete problem for NL.

And then we're going to prove a theorem that was, in its day, very surprising to people. I remember when it came out in the 1980s, that NL, in fact, is closed under complementation, that the NL class and coNL class both collapse to 1, that they're equal, which is not the way we believe the situation to be for NP. But until someone has a proof that they're different, strange things can happen.

OK. So with that, I will move myself into the corner. And let's do our review. So we are-- in order to talk about space complexity classes that are smaller than n, we had to introduce a new model, which was the two-tape model, where there was a read-only tape that had the input on it, which you normally would think of something very large-- the whole internet, or something so big that you can't read it into your local memory.

And then you have a work tape, which is your local memory. And the way we're going to think about it in the context of log space is that that work tape is logarithmic in the size of the input. And that is enough to have small counters or pointers into the input, because a reference location of the input is just-- you only need log n bits to do that.

So we gave a couple of examples of the L and NL-- of L and NL languages, so this language of ww reverse, as you may remember. So here is an input in ww reverse. It's a string follow-- it's a palindromic string, which is of even length.

And so you can make a machine in log space here that can test whether its input is of that form. And the work tape is only-- all it needs is a pointer into the-- or a couple of pointers that refer to the corresponding places of the input that you're looking at at the moment. So you maybe start out looking at the two outside a's and then the b symbols that are next to that.

And you can write down on your tape where you're looking currently. And so that's going to be enough for you to- - you may have to zigzag, of course, back and forth a lot in order to do that test. But that's completely fine, using the model. We're not going to be measuring time. We're only going to be focusing on how much space we're using.

Another example that we gave is the PATH language, where you're given a graph, and a start node, and a target node. And you want to know, is there a path in this directed graph that goes from s to t? And that's the language that's also-- that language is in NL-- in fact, not known to be in L.

So the way that would look, shifting to an input in the PATH language, you would have a graph represented, say, by a sequence of edges, and a start, and a target. And the work tape would keep track of the current node. So the nondeterministic machine would guess a path that takes you from s to t, node by node. And the work tape would keep track of the current node.

OK, so I hope you have this-- develop a little bit of an intuition for these classes, L and NL. We're going to be spending the entire lecture today talking about that. OK. So as I mentioned, the L and NL problem is an unsolved problem. And it's very much analogous to the P versus NP problem, except, as I mentioned, as we'll show, that NL and its complement end up being the same, which is not something that seems to be the case for NP, though we don't know.

Can we think of this as a multi-head Turing machine? I'm getting a question about that, which is, I think, you can. In fact, that's an alternative way that people look at it. You can think of it as having multiple-- you know, a head basically needs log space to store the location-- to store the location of where that head would be.

So if you imagine having several different heads on the input tape, you can think of a log space machine as being sort of a Turing machine that has multiple heads on the input table. It's equivalent. Good question. So let's move on, then.

OK. So one of the things we proved last time was that anything that you can do in L, you can also do in polynomial time. And I'll answer some of these chat questions in a minute. But the-- so the class L is a subset of P. This is easy to prove. But I think it's nevertheless important to see why it's true, because it sort of sets some definitions of things that we're going to use later.

So in particular, we really need to know the notion of a configuration of a log space machine on an input. So because the input does not change, we don't really consider the input to be part of the configuration. The only thing that's-- the thing that's relevant in the configuration is the dynamic part of the machine-- the state, the head locations, and the work tape contents.

So we're defining that the configuration for the machine on a particular input, w, is those four things-- state, the two head locations, and the tape contents. And the important thing to keep in mind for this theorem is that we have only a polynomial number of different configurations if you just do the calculation. The main part is the number of different tape contents as you can have, which is exponential in the log n. And that's a polynomial.

And so therefore, any machine that runs in log space, provided it always holds, and we always assume our machines always hold, they can only run for a polynomial number of steps, because that's as many different configurations as they have. If they ran for, say, an exponential number of steps, they would have to be repeating configurations. And then they would be looping.

OK, so there we go. OK, so let me just get back-- so somebody asked me a question. Which is harder? P versus NP or L versus NL? Completely no idea. It's a-- I guess there was a common line of thinking that if you're going to-- that it's good to try to think about-- if you're trying to separate classes, you might as well take classes that are as far apart as one another.

Like, if you're trying to prove-- if you're comparing P different from NP and P different from PSPACE, maybe P different from PSPACE might be easier, because P and PSPACE seem to be even further apart than P and NP. Nobody knows.

And I suspect that there's something fundamental about computation that we just don't understand. And then once somebody makes a breakthrough and solves one of those problems, a lot of them are going to get solved in short order. But again, it's purely speculation.

OK, d. What is d here? d would be the size of the tape alphabet. OK, so this is the number of different tape contents we have. Good. All right. So let's continue on. Another thing we mentioned kind of quickly in passing, but still an important fact, is that Savitch's theorem works down to the level of log space-- same exact proof.

So that means that nondeterministic log space is contained in deterministic log squared space, because that's what Savitch's theorem does for you. It converts nondeterministic machines to deterministic machines at the cost of a squaring in the amount of space you need.

And so I'm not going to go through this in detail. But the same picture that I copied off an earlier slide with a simple modification is that instead of-- that I'm right down-- the size of the configuration is going to be now log n, because that's how big the configurations are when you have a nondeterministic log space machine.

And so simulating that-- so this would be what the tableau would look like for an NL machine. And then you can simulate that in the same way by trying all possible intermediates and then splitting it, doing the top half and then the bottom half. We're using the space, of course, recursively.

The amount of space you're going to need is going to be enough to store, for one level of the recursion, one configuration. And that's order log space. And then the number of levels of recursion is going to be another factor of log n, because that's log to the running time, which is going to be exponential in log n, which is polynomial.

So the total amount of space that you would need would be log squared space. Again, this is sort of saying the proof of Savitch's theorem, just over again. So if it's coming too fast for you, just review the proof of Savitch's theorem and observe that it works, even if the amount of space that the machine-- that the nondeterministic machine starts off with is log space. All right.

And last thing I was going to-- last thing in the category of a review is our theorem that not only is all of L within P-- and that's kind of trivial, kind of immediate. You don't even have to change the machine. If you have a log space machine for some language, the very same machine is a polynomial time machine for that language, because it can only be running for a polynomial amount of time.

But now, we have a nondeterministic machine for some language. We're going to have to change it to become a deterministic machine that runs in polynomial time. And so we're going to give a deterministic polynomial time simulation of a nondeterministic log space machine.

And we kind of did this last time, but a little quickly. So now, if we have some nondeterministic log space machine, so an M, which decides the language A in log space, we're going to show how to simulate that machine with a deterministic polynomial time machine.

And the key idea, which is going to come up in a later theorem, so good to understand it not only here, but to understand it for the next theorem that's coming, is the notion of a configuration graph. I was sort of thinking about calling it a computation graph. But now, on further reflection, I think configuration graph maybe is the more suggestive term. So let's stick with that.

So a configuration graph for a machine on an input is just a set of all configurations that the machine has, all the possible different configurations the machine can have, with edges connecting configurations that correspond to legal moves of the machine. So here is some configuration. This is a snapshot of the machine at a moment in time.

Here is some other configuration, another snapshot of the machine at a moment n time. And you're going to draw an edge between ci and cj if cj could follow in one step from ci. And you could tell by just looking at the configurations whether that could be possible. Obviously, the head has to be one place over in cj from where it was in ci. And it has to be updated according to the rules of the machine.

So you can tell whether-- so you could fill out this graph. You could write down all the possible configurations. And you can put the edges down. Now, the point is that when we have a log space machine, we don't have too many possible configurations. There's only a polynomial number. So the size of this whole graph is polynomial.

So our polynomial time simulation is going to write down that entire configuration graph of the log space machine on its input. There [INAUDIBLE] many configurations. There can be only polynomially many. So you can write down all those configurations as the nodes and then go look at each pair of nodes, whether this configuration could lead to that configuration.

According to-- it's a nondeterministic machine. So a configuration could go to several different locations-- there could be several different ways to go. But those are just several different outgoing edges from a particular node in this graph representing the configuration, which might have several different legal successors in the nondeterministic computation.

OK. Now, the important thing here is that M accepts its input, w, exactly when there is a path [INAUDIBLE] configuration graph that takes you from the start configuration to the accept configuration. And as I mentioned, let's assume, as we've been doing, that the machine-- I should have put it here, but I didn't. But the machine, when it is about to accept, it erases its work tape and moves both of its heads to the home position at the left end of the tape.

So there's just one accepting configuration you have to worry about. It just makes life simpler. So there's going to be a start configuration, a single accept configuration in this configuration graph. And now there's going to be a path, indicated here, that connects the start configuration to the accept configuration if and only if M accepts w, because that path is the sequence of configurations that the machine would go through if you launched it on w.

It would start at the start. And there might be several different ways to go. But if there is one of them that leads you to an accept, that's going to correspond to a branch of the computation that it's accepting. OK, so that tells us what the polynomial time algorithm is.

On input w, you construct that configuration graph for M on w, G sub Mw. And you test whether there's a path from c start to c accept using any polynomial time depth-first search or breadth-first search algorithm for testing whether there's a connection path in a graph.

And if there is such a path, you accept, because that means the machine M accepted. And if there was no path, you reject, because then M must have not accepted. And therefore, you have a polynomial time simulation of your nondeterministic log space machine M, OK? How are we doing?

OK, so that tells us that NL is contained within P. And also, L is contained within NL, as before. So we have kind of this hierarchy of classes. Now, you can even talk about, not only is L different from NL, even is L different from P? Is it possible that anything that you can do in polynomial time, you can do [INAUDIBLE] space? Don't know. Open question.

OK, getting a very good question here just now. Why is this construction taking log space? It doesn't. This construction takes polynomial time. This algorithm here is not a polynomial-- this is not a log space algorithm that I'm giving you. I'm giving you a polynomial time algorithm for simulating a nondeterministic log space machine.

Now, later on-- I don't want to confuse the issue right now. For this particular slide, all I need to do is construct that graph in polynomial time. It's a polynomial size graph. I can't store that whole graph in a log space memory. OK, so question here-- we can see that listing out the nodes and edges would be polynomial time. But how do we actually provide structure to this graph representation? I don't even know what that means. So if you can clarify that for me, then maybe I can try to answer it.

But a graph is just a list of nodes and a list of edges. After that, we know what the graph is. I mean, you may like a picture. But the machine doesn't need a picture. Our definition of-- we just represent these things as strings, in the end. So please clarify if you want me to-- I'll answer it the next at the next pause.

I'm grateful for all the questions, because I'm sure, any question that any of you have, another 20 of you also have. So questions are good. Don't be bashful. And also ask the TAs if I become overloaded. OK, now we're going to shift gears into some new material.

All right. We're going to talk about the notion that-- sort of thinking about, analogous to the P versus NP problem, where there were these NP-complete problems, now we have the L versus NL problem. There are going to be an NL-complete problems that kind of capture the essence of NL the way NP-complete problems capture the essence of NP, in a sense, in that all of the problems in NP are reducible to them. So they're kind of like the hardest NP problems.

Here, we're going to have exactly analogous situations for NL, where we're going to show problems where all other NL problems are reducible to them. So if you can kind of solve one of them, like solve one of these NL-complete problems in log space deterministically, then you solve all of NL problems in log space deterministically.

So it's a very similar-looking definition to what we had before. It's NL complete if it's in NL. And then all other languages in NL should be reducible. But now, we have a new notion here, with an L instead of a P. Now, before, when we talked about NP completeness, we had polynomial time reducibility.

That's not going to work anymore, because if you remember, NL is a subset of P. So all NL languages are polynomial-- are languages in P. And if we're talking about-- if we use polynomial time reducibility, all languages in P are reducible to each other. We need to have a notion of reducibility which is kind of weaker than the class.

And so polynomial time reducibility just would not work here, because everything would become NL complete, because everything is reducible to each other. So we need to have a weaker notion. We're going to use log space reducibility, which we have to define. So here, for that, we're going to have to talk about the notion of a function that you can compute in log space.

And it's a little tricky here. Just like when we talked about language recognition in log space, where we had the work tape had to be smaller than the input tape, because the inputs can be large. The work area is small. Now the output also could be large relative to the work area.

So we're going to have a three-tape model, where there's the input is going to be a read-only, the output is a write-only-- it's like a printer. It's something you can only write on, but you can't read back, because otherwise, you could cheat by using the output as a kind of storage. And then you have your storage area, which is your read-write work tape.

OK, so this-- we'll call this-- the traditional name of this is a log space transducer. So it converts inputs to outputs, but uses only log space for its working memory. OK, so the input tape stores n bit-- n symbols. The work tape stores log n symbols, order log n symbols. And then we have the output tape. You may want to think about how big-- there's going to be a check-in coming to kind of ask you, how big could the output be? But we'll save that for the end if you-- you can mull that over if you want to think ahead.

OK. So we think of a log space transducer as computing a function, which is just a mapping from the input to the output that the transducer provides for you. A transducer is a deterministic machine, by the way. So you take the transducer. You give it w. And you turn it on. And then it halts with f of w on its output tape. That's what it means to be computing the function f, OK?

And we'll say that A is log space reducible to B, using the l subscript symbol on the less than or equal to sign, if it's mapping reducible to B, but by a reduction function that's computable in log space, just the same way we define polynomial time reducibility. But there, we insisted that the reduction function was computable in polynomial time.

OK. Just quickly, I got a question again. Why log space here? Because polynomial time would be too powerful for doing reductions internal to P. Every language in P is reducible to every other language in P. And so everything in NL would be reducible to everything else in L with a polynomial time reducibility. And so that would not be an interesting notion.

We have to use a weaker notion than that, a weaker kind of reduction, using a weaker model, so that you don't get-- otherwise, the reduction function would be able to answer whether-- would be able to solve the problem A itself if we had a polynomial time reduction, and we're mapping things from NL to other problems in NL. The reduction would solve the problem. And that's not what you want.

The reduction should be constrained only to be able to do simple transformations on the problem, not to solve the problem. Anyway, you have to look at that. This is an issue that's come up before when we talked about, what's the right notion of reduction to use for PSPACE completeness? Same exact discussion.

OK. Now, there is an issue here, though, that we have to be careful of. When we have A being log space reducible to B, and B in L, then what you want-- if A is log space reducible to B and B in L, then you want A to be in L. That's the same pattern we've always had for reductions.

If A is reducible to B, and B is easy, then A is easy. So here, the notion of easy is being an L. Now, if you remember the proof of that we had from before, which I'll just put out for you, is that to show a log space solver for A, you take an input, w.

And now, if A is reducible to B, you compute the reduction. And then you run the decider for B. So if we're assuming A is reducible to B, and B is in L, so B has a log space decider, you take your w, which you want to know, is it in A? You map it over to a B problem using the reduction function. And then you solve it using the decider for B. And you give the same answer.

Now, this actually doesn't work anymore, or it doesn't work in an obvious way, because, if you're following me-- I hope most of you are-- there is a problem here, which-- because we're trying to give a log space algorithm for A.

And that algorithm is going to be computing this reduction function, mapping w to f of w. f of w might itself be very large, as this picture suggests here. You may not be able to store f of w in the log space memory for the machine that's deciding A.

So this is an obstacle that we need to solve in order to prove this theorem, which we need, because that's the whole justification for doing these reducibilities-- should be a familiar-looking kind of line to what we've seen before.

So we don't have space to store f of w. What do we do? And I'll also mention that this is going to be relevant to one of your homework problems. So what do we do? We don't have space to store the intermediate result that we need in order to solve the problem. We started with w. Now we'd need to test if f of w is in B. That can run in log space. But just simply getting your hands on f of w-- what do you do about that?

So what we're going to do is the following, is the decider-- the decider for B, which needs f of w, because it's deciding if f of w is in B-- it doesn't need all of f of w sitting there in front of it all at once. If you think about how the Turing machine operates on its input, it only looks at one symbol at a time.

It starts out reading the leftmost symbol of f of w, then maybe it moves its head right and moves to the second symbol of f of w, then the third symbol of f of w. Maybe it gets up to the 10th symbol of f of w. Maybe it moves his head back and goes to the ninth symbol and the eighth symbol. But the Turing machine's head, which is deciding B, only looks at one symbol of f of w at a time.

So instead of writing down all of f of w, the idea is that we are going to compute the individual symbols of f of w that we need only at the moment we need them. So if the decider for B is reading the 10th symbol of f of w, we fire up the transducer on w.

And as it's writing out its output, which we don't have space to store anymore, we throw away all of the output values until we get to the 10th one. And then we say, ah, the 10th one is whatever, is a c, whatever the value is. Now we feed that into the decider for B. We can now simulate that decider for one more step.

Now the decider says, all right, now I need the 11th symbol of f of w. OK, now we can run that machine for one more place. But if it needs-- but we don't even have to do it. I think the better way to think about it is, every time that decider for B needs another symbol, we start the transducer over again and just keep-- throw away everything except for that one symbol output that we need.

So every time we do another step of simulating B, we're going to have to rerun the transducer from the beginning, just to recompute that, or compute maybe for the first time, or recompute it if we need it subsequently. This is going to be slow, but we don't care, to recompute that symbol that the simulator-- that the decider for B requires, OK?

So I'm saying that over here. Recompute the symbols of f of w as needed. OK, so let me-- let's take a couple of questions. And then we're going to move to a check-in. So somebody's asking, why did we have to introduce transducer for log space reducibility when we didn't do it for polynomial time reducibility?

We could have for polynomial time reducibility. But we didn't need to, because we could just all do it on the same tape. The problem is, for log space, the tape is-- the work tape is too small to hold the input on the output. So we can't-- since we're only working-- we have a log n bound that we have to work within. We need to separate those functions from the work functions, the input function and the output function.

So if we have more than the amount of resource we have, either time or space was at least n, then we could just lump them all together and have that one tape do multiple functions. And somebody's asked me here, yeah, this is mapping reducibility, this m. This is from the notion we saw before.

OK. Does f of w lie on the input tape of B? Well, yes. So we are-- good question. So f of w-- you know, because what are we doing? We're trying to find a decider for A here, using the decider for B and the mapping from A to B, the reduction from A to B.

So the decider for B expects to find its input on an input tape. That input is going to be f of w. But we have to get the effect of that without actually writing down that input tape, because we don't have enough room to write down the input tape for the decider-- for the B decider, because that could be very large.

And we only have-- we have no place to put the f of w. So think about what's going on here. We're making a log space machine whose input is w, has to compute f of w as an intermediate value, to feed it into the B decider. That is not going to be possible to hold onto that whole f of w at one-- altogether, because it's too big.

But that doesn't matter. We don't need it. We only needed one symbol at a time, which we can recompute. OK, so let's see. So somebody says, can we just ensure that the output tape is order n so we don't need to use more tape than the input?

Order n is still going to be too big. Where are you going to put that output? Even if it's just order n-- first of all, the answer is no, we can't, because there are going to be reductions which are bigger than that. But the other question is, can we just ensure that the output is order n?

You can't put the output on the input tape. The input tape is read-only. The output tape is write-only. So there's no place to-- even if the output is just as big as the input, it doesn't help you. If the output is only log n, OK, then we could do it. But that's not going to be interesting for us. You're going to need, for these large space reductions, big outputs, as we'll see in a minute.

What's the running time for this log space reduction? It's all going to be polynomial. It's all going to be a log space algorithm. So it's all going to be polynomial. Is there any NP completeness reduction which can be done in log space?

All of the NP-- all typical NP completeness reductions, those polynomial time reductions, they all can be done in log space, because they are-- reductions tend to be very simple transformations. And log space is going to be enough to do all of them. OK. I can't answer the second part of that. That's too complicated. And I think we should move on.

So let's look at the first check-in here. So if we have a long space transducer that computes f, and if you feed it inputs of length n, how big can the outputs be, actually? So why don't you think about that and give me an answer? I'll give you a minute to answer this question.

Oh, this is a tough one. Let me just say up front, there are-- I struggle with this lecture, because some-- especially the stuff in the second half, it's kind of hard. I wouldn't say it's technical. But conceptually, I think some of the material is a little harder, maybe in part because people are not used to thinking about memory complexity or space complexity, even though I don't see why-- I mean, I think it's an important resource to be considering. But I think it's less common. And I think there's some discomfort with that.

OK, so we're just about done here. Five more seconds, please. All right, about to wrap. Wrap the check-in. 1, 2, 3. All right. So yes, the correct answer is c. As I mentioned, we're going to want to have outputs that are larger than log n. And there's no reason why they wouldn't be able to be larger than log n, according to the definition that I gave you.

There's no bound on the output. We're only measuring the running space of this algorithm in terms of its work tape. The input and output tapes don't count. So they can be more than log n. They can be more than n. Polynomial is the right answer. Why? Because a log space transducer, if you just ignore the output, is just an ordinary log space machine. And it can only run for a polynomial number of steps without it end up going into a loop. The same argument that we gave for that before applies here as well.

So if it's going to exceed a polynomial number of steps, it's never going to hold. And so that's going to be-- not allow it-- it's got to halt with the output on the output tape. And so it'll be disqualified as a log space transducer if it doesn't halt. So it can't be anything longer than polynomial. It's a good thing to think about, to understand. OK, so let's continue.

So we're going to show that the PATH problem is NL complete. Now, we defined NL completeness. And we've seen the PATH problem before. And we're now going to show that PATH occupies a very special position for NL, namely that it's an NL-complete problem.

So if you can solve the PATH problem deterministically in log space, you have gotten a big result. No one knows how to do that. And it would collapse all of NL down to log space if you could do PATH in log space deterministically.

OK, so let's see why that is. So first of all, the two components of being complete are being in the language and the reduction part. So in the language, we've shown already. Now, we want to show that for any other language in NL, it's going to be log space reducible to PATH.

In a certain sense, this may not feel so surprising, thinking back to our proof that NL is a subset of P, because we managed to convert any NL machine, the running of any NL machine, to a PATH problem that the polynomial time machine then solved.

And so it's really the same idea that says that PATH really captures any NL machine. The computation of any NL machine really can be seen as a PATH problem, where the nodes are the configurations of the machine. So let's just see how-- let me just try to go through that if that wasn't super clear, which I'm not sure it was.

So suppose we have a machine decided by-- a language decided by a nondeterministic-- an NL machine, a nondeterministic machine in log space. Again, I should have put this before. But we're going to modify M to erase its work tape and move its head to the left end on accepting. So it has a unique accepting configuration.

Now I'm going to give it the log space reduction that maps our language A, which is in NL, to the PATH language. So thinking about what that means, I'm going to take an input, w, which may or may not be in A, and produce for you a graph with a start and target node, start and target notes, where w is going to be in the language if and only if G has a path from s to t.

And what do you think that graph is going to be? That's going to be the configuration graph for the machine that decides A, OK? So that is how it's going to look. So maybe here's a picture. Right. So f of w, where w, again, is your problem about membership in A, is going to become a problem about membership in PATH. And it's just going to be the configuration graph for M on w.

Now, what's left is to show that we can do this conversion with a log space transducer. So it's a log space computable reduction. So let's just try to go through that quickly-- conceptually, not super hard. So here's our transducer. Let's just think about what it needs to do.

It needs to take an input, w, and convert that f of w to this thing here-- computation graph of M on w-- the configuration graph M on w, the start and accept configuration. So that's going to look like this down here. That's what we want to eventually appear on the output tape.

So the way we're going to achieve that-- we only have a small log space, order log space work tape. And the way we're going to be able to produce this output is-- the configuration graph is just a series of edges, which are-- say, you can go from this configuration to that configuration in one step.

So what we're going to do is, on our work tape, we're going to go through all possible pairs of configurations, again, just in some like odometer order, just by looking at all possible strings, really, of length order log n that are big enough to represent two configurations. Every once in a while, it's going to be actually a pair of configurations. At that point, we look at those two configurations, look at M, and see, can this configuration go to that configuration?

If yes, you print it out on the output tape. If no, you just move on to the next pair of configurations. And then, at the end, you write down on the start and accept configurations. So I've indicated that here. Here is the transducer. It says, on input w, for all pairs of configurations, that-- now, this is getting written down on the work tape-- you output those pairs which are legal moves for M. And then finally, you output the start and the accept. That's it.

So let's just see. Let me take any questions here. Why do we need special accept state for M? Well, we want to have-- I think you mean accepting configuration. I just want to have a-- I don't want to have a multiplicity of different possible accepting configurations, because then it's not really a PATH problem. Then it becomes a question of, can I get from the start to one of those nodes representing accepting configurations? That's a little messy. I could fix it. But the simplest fix is just to make there be a single accepting configuration.

Well, why do I output start and accept at the end of the output tape? That's the way I write down my PATH problem. It's a graph, followed by a start node and a target node. So I have to follow that form. I'm not sure what you're asking. You want me to put that first? I'm not sure what the-- or why at all? Because it has to be a-- here it is. Here's the output I'm looking for.

OK. Do the three-- do the read-write work tape here store pointers to configuration or some sort of counter? No, they store the actual configuration. The configuration for M is-- just think about what it is. It's a log space size object. It's a tape for M. It's a location of its heads and its state.

So you could kind of write down that stuff right over here, on the left side of this-- this left slot. And on the right slot, you're going to write another configuration for M on w. And you're going to just put the edges in accordingly. OK, so somebody-- did that help? Somebody, again, is asking, why is the configuration only log space? It's just a tape. It's a log space tape. That's the main thing in the configuration of the tape.

On the read-write work tape, do we only write two configurations at once? Yeah. We're just writing down a candidate edge that we're going to output onto the output tape. So that's why we have two configurations. I want to know, can I get from this configuration to that configuration? If yes, I print it out, print out that pair. That's an edge in my configuration graph, which is what I'm supposed to be outputting here.

Can there be multiple-- OK, why don't we move on? Again, direct questions to our TAs, who would be more than happy to help you. And we will-- let me just quickly give-- we're running a little tight here time-wise. But let's just see. Here's an example of showing some other problem is NL complete. You have a homework problem on that. So I thought I wanted to give you an example. Maybe we can just defer this to the recitation. So maybe we'll try to do this a little quickly to save us on time.

But the 2SAT problem, which is just like the 3SAT problem, except with two literals per clause-- curiously, the complement of that problem, so the unsatisfiable formulas, that form an NL-complete language. And so first of all, you have to show it's in NL. We're not going to do that. It's a nice exercise. It's not totally trivial to do. But you might want to try that.

We're going to show that PATH is reducible to the complement of 2SAT. We've got to give a reduction that converts graphs to formulas, where there is a PATH, now, when the formula is unsatisfied. And what's going to happen is the PATH is going to correspond to a sequence of implications in the formula, which yields a contradiction and forces it to be unsatisfied. Again, this is going to come a little fast. And then maybe we can discuss it over the break, which is next.

So every node in G is going to have associated variable in the formula. So there's a variable for every one of the nodes. For every edge, there's going to be a clause of implication connecting those two associated nodes. So if there's an edge from u to v, then there's going to be an implication in the formula that says, if xu is true, then xv is true.

And note that that's equivalent to the more conventional way [INAUDIBLE] xu complement or xv. These are logically equivalent. So I'm not cheating you here in terms of being a 2SAT problem. They really just look like this. And lastly, I'm going to put two additional clauses. It's [INAUDIBLE] x for the start variable-- from the start node, s-- here, s. I want to force that one to be true.

So it's x-- since I want to have exactly two per clause, that's xs or xs. So that forces x-- that variable true. And lastly, if t is true, that's going to force the-- if xt is true, that's going to force xs to be false.

So now, if there's actually a path in the graph that goes from s to t, there's going to be a sequence of implications, starting now with s being true, forcing other things being true, including forcing t to be true, which then forces s to be false. And that's our contradiction, which shows that the formula cannot be satisfied.

So now, you have to prove that this works. As I said, for the forward direction, if there is a path, you follow the implications to get a contradiction. For the reverse-- let me not spend time here. I'll leave this to you to think about offline. But if there is no path, there is a way of assigning the variables to true and false to make a satisfying assignment to the formula. So that gives the other direction, OK?

And you can show it's computable in log space. That's very simple, because a very simple transformation there, OK? So I think we're going to move on to the break. And I'm happy to take questions at this point about this.

Does the configuration, going back, include the input? No. The configuration does not-- as I said, the configuration for M on w is the state, the head positions, and the work tape contents, not the input tape, because then you would be-- it's not there for a reason. The input is huge.

But you don't need the input there, because the input is going to be constant for everybody. Everybody can look at that input, which is a fixed, sort of external thing. Somebody's asking me, are there NP-complete problems in-- there are definitely NP-complete [INAUDIBLE]. I don't know-- there are some problems in number theory where it's-- like factoring, where we don't know the status, somewhere between P and NP, formulated as a language, of course.

But there are problems in solving certain kinds of equations, low-degree equations, that I don't remember now if [INAUDIBLE] actually known to be NP complete. Now, you asked about NL complete [INAUDIBLE]. I don't know if there are NL-complete number theory problems.

Oh, good question. Somebody's asking me, does NL also have an alternative definition using certificates or witnesses? Yeah. Yes, sort of. For NL, you can make a certificate, which is, again, polynomial size certificate. But it has to be-- you're only allowed to read it with a one-way head. So it's like a one-way certificate.

So it has to be-- you can only process it in a certain way. That's a nice exercise, actually, itself. But anyway, let us-- we are now done. And we're going to move back. We're going to continue. So everybody return. This is what's next on the agenda, proving that NL equals coNL. This is a hard proof.

I'm going to try to break it down as much as I can. And let's hope you get-- I hope you get it. I'll try to be as helpful as I can. OK. But if you're finding it tough, you won't be alone. So first of all, we're going to show-- the way we're going to solve this is by showing that the complement of PATH is solvable in NL, because the complement of PATH is-- just as PATH is complete for NL, the complement is complete for coNL.

And so by doing that problem in NL, we're going to reduce all of-- all of coNL will be reducible to problems in NL. And so therefore, we'll be in NL. coNL will be then inside NL. And then NL is going to be equal to coNL. If that sequence of logical connections, is not clear. Don't worry.

The point is that we want [INAUDIBLE] go back and figure out why that's enough later. But what this means is we want to give a nondeterministic machine, which will accept when there is no path from s to t. OK? And please don't say, why don't we just take the machine for PATH and flip the answer? You can't do that with a nondeterministic machine.

So you better-- if you're thinking that that's allowed, go back and review nondeterminism. So you want to make a nondeterministic machine, which is going to accept when there's no path. So some branch is going to make a sequence of guesses. And it has to be sure that there's no path. And then it's going to be-- and then it can accept when there's no path.

Now, if you can find a way of like making a separator, something that cuts the graph in half and separates s from t, then you would be good. The only problem is there's no obvious way of doing that, because those kind of separators, even if they were [INAUDIBLE] probably too big to write down in log space. So I'm going to give you a completely different way of doing it.

And I'm going to make-- this is a little different presentation than what's in the book. I think hopefully, this is a little longer, and therefore, a little clearer. We'll see. So first of all, I'm going to define a notion of a nondeterministic machine computing a function.

And that's a simple idea. What you want is, on the different branches-- so you have some function, f, which has, for every w, there's an output, f of w. And the nondeterministic machine can operate that on all of its branches, it's allowed to either give f of w or say reject, meaning punt, or say, I don't know. So every branch has to give the right answer. So all the branches that give an answer have to agree, because there's only one right answer. All the branches that give an answer have to give the right answer, or they can say, I don't know.

The only thing is you have to also say that at least one of the branches actually gives an answer. So somebody cannot reject. Somebody cannot say, I don't know. So at least one of the branches gives an answer and-- gives the answer. And all the other branches can either give the answer, or they can say-- they can just reject. But there's no notion of accepting. There's just a notion of this nondeterministic machine, on some branches, giving the output value, and other branches just punting and saying reject. Maybe reject is the wrong word. I could just say punt.

All right. So we're going to be talking about functions that you can compute with nondeterministic machines, with NL machines in particular. All right? So we're going to look at this path function now. Now, this is not exactly the same as the PATH language. This is a function here, written with lowercase.

So given a graph, s and t, I'm going to say yes if there is a path and no if there's no path. And this is a function now, which is going to output yes or no, not a language. This is a function. It's very closely related. I understand. So if you can solve the function, you can do the language. But what we're going to give is a NL machine, a nondeterministic machine, which is going to compute this function. And therefore, you can use that to do the PATH language.

Two important things for us is, if G is some graph, well, here's the starting node, s. R is all of the nodes that you can reach from s. This is some collection of nodes. And c, which stands for count, is the number of reachable nodes. So I've written that down here more-- if you like it more formally. R is the number-- is the collection of nodes for which there's a path from s to the node. And c is the size of R. So you have to understand these two, because we're going to be playing with this for the next three slides.

OK. Now, first of all, this is kind of a little bit of an exercise theorem. But it's still going to be a useful fact that we're going to end up needing later. But it's also a little bit of just to test your understanding. Suppose there's some NL machine which computes this path function.

So on the different branches of the nondeterminism, given a graph, G, s, and t, there are going to be some branches which may output yes, or some branches that may output no. And other branches might say, I don't know. But the machine always has to give the right answer if it's going to give any answer. So all branches either have to say yes, or all branches-- all branches have to say yes or punt, or all branches have to say no or punt, because one of those answers is going to be the right answer.

So suppose I have a way of computing path by an NL machine. Then can I also compute the-- can I make some other NL machine which computes the count, the number of nodes reachable? So if I can test if a node is reachable, can I figure out how many nodes are reachable? This is supposed to be easy. This is kind of a little bit of a practice.

So if I can figure out if nodes are reachable, yes or no, then I can say, figure out how many nodes are reachable. You just go through them one by one, testing if they're reachable, and count the ones that are. That's all I have in mind. So start with a counter that's set to 0 initially. And go through each of the nodes of G one by one.

And I use my NL machine that computes path. That's what I mean by this part. So I test it. If the NL machine says yes, there is a path, then I increase the counter. And if it says there's no path, then I just continue without increasing the counter.

Now, when I'm running my NL machine to compute this function, that NL machine might punt, might reject sometimes on some branches. That's OK. I'm also allowed. I'm also an NL machine. I'm computing a value. And I also might punt on some branches. So at the end, I'm going to output that count, OK?

So what I'm going to prove next is the converse of this. And that's-- and that's the magical hard part, that if I can compute the count, then I can do the test of whether individual nodes are connected, have a path from s. OK, so let's just see.

Somebody is asking if nondeterministic machines-- so like M is not allowed to loop? No. If a machine, if any one of these machines, like an NL machine, loops, it's going to be going forever. That's not allowed. So no looping. I'm not sure why that's relevant, but no looping.

But what I'm more worried is that you understand this theorem here. I think we have a check-in coming. Let's see. OK. This might be helpful. So consider the statement that PATH complement is NL. That's what we're trying to prove, and also that some NL machine can compute the path function.

These are going to be related facts. Which one can we prove from the other easily? I mean, they're both going to be true. So in some sense, it's trivial. But I want to know, which one can we prove kind of immediately without doing much work? That I can solve this PATH problem in NL, the complement of the PATH problem in NL, or that I can compute the path function in NL? So what do you think?

OK, almost done here? Yeah. Ending. You guys didn't do well. That's OK. Actually, the right answer is c. Most of you got that if I can solve the path function, so the yes-no value, I can use that now to solve both PATH and PATH complement. That seems more clear cut.

But suppose I can solve the PATH complement problem in NL. And I also know I can solve the PATH problem in NL. That, we've already shown. So knowing both of those, if I'm given a G, s, and t, what I can do is nondeterministically pick which of those two directions. You know, I pick-- I'm going to guess, well, it's in PATH, or it's in the complement of PATH. So there are two different nondeterministic ways to go.

One of those is going to always end up rejecting. And so that's going to end up punting. The other direction is going to sometimes end up accepting and sometimes punting. And based upon whether which side ends up-- one or the other is going to have some accept-- is going to be accepting. And so the one that's accepting is going to tell me whether to answer yes or no.

So actually, both directions, both implications follow pretty easily. OK. Anyway, let's try to show-- this is the hard part. And we have five minutes. Let's see how far we can get. So this theorem works by magic. So it kind of blew everybody's mind when it first came out. So let's just see. It's really not that hard. But it's sort of-- it's kind of twisted.

So suppose some machine can compute c, the count, the number reachable from s. I'm going to use that to solve path, the path function, to test, yes, I can output yes if there is a path or no, there is no path, for each node t. So if I know how many nodes are reachable, then I can solve now for individual nodes, which is strange that you can do that.

Now, I'm not telling you how to compute c. That's for later, which I probably won't get to. But just pretend we can somehow figure out what the count is of the number of reachable nodes, OK? So here is my nondeterministic algorithm for computing path.

First, I'm going to compute c, or let's say c is given. And now, maybe the best thing to do is to try to give you the idea up front. What we're going to do, since we're a little short on time, what we're going to do is, suppose I tell you, there are, in this graph, 100 nodes reachable from s. So c is 100. There's 100 reachable nodes.

Now I want to know-- I say, well, I don't really-- that's all very nice. But I'd like to know this particular node, t. Is that reachable from s? Now, I'm a nondeterministic machine. Now, if t was reachable, then I'd be fine, because nondeterministically, I don't even care about the 100. I take, nondeterministically, on some branch, starting from s, I'm going to hit t. And that branch is going to say yes. The other branches, maybe they'll punt. But some branch is going to get the right answer.

The problem is, suppose t is not reachable. Then you want some branch to say no. And how could that branch ever say no, unless it's sure that t is not reachable? And how can one branch be sure? The idea is this. Suppose I know that there are 100 reachable nodes. What I'm going to do nondeterministically is I'm going to guess those 100 nodes, one by one. You can't store them all, because it could be-- 100 could be a big number.

I'm going to guess them one by one. I'm going to guess them. And every time I guess a node, I'm going to prove it's reachable by guessing the path that shows it's reachable. So I'm going to guess 100 nodes, prove that they're reachable, and then see, was t of those reachable nodes?

If it was, well, then I would have found it, and I would know to say yes. But if t was not one of the 100 reachable nodes, and I know there's only 100-- so if t is not one of those nodes-- in other words, if I found them all, and t wasn't one of them, then I know it's not reachable.

And that's how, using the count, I can be sure that certain nodes are not reachable, because I just find all the ones that are, prove that they are, check that the count agrees with what I was given, and then say no, t is not reachable, if it's not one of those nodes that I've found to be reachable, which adds up to my given count. That's the whole idea.

Of course, how do you get the count? Oddly enough, it's kind of the same idea repeated over and over again. But I guess we'll have to do that next time. So let's just write this down. And we'll kind of use it as the beginning of Thursday's lecture. So we're going to go through each node u, one by one. Now we're going to guess, for each node, whether there's a path to it or not. So I'm going to call it either p or n.

Again, this is now-- think about my 100 nodes. I'm going to be guessing all 100 nodes. I'm going to nondeterministically pick a path from that node that I guess is reachable. So if I guess a node, there is a path. I'm going to confirm there's a path by nondeterministically picking it. If I don't find that path, I just reject punt on that branch.

If that path that I found actually led me to t, so u, that node that I'm working on, is currently t, then I know to accept. But otherwise, I'm just going to count the number of nodes that I find are reachable. If I've guessed that u is not reachable, I'm just going to skip it.

At the end, I see whether the number of nodes that I have determined are reachable agrees with my original count, c. So does k equal c or not? If it doesn't equal, they're not equal, then I didn't find all the reachable nodes. I didn't guess right. And so I punt. I say, well, bad branch of the nondeterminism. I just give up.

But some branch of the nondeterminism is going to guess all of the correct nodes which are reachable. And then, if t hadn't been found already to be one of them, at this point, I know t is not reachable. And so I can output no. OK? So that's the whole thing.

What is m? m is the-- yeah, good question. m is the number of nodes of the graph. I should have said that. So you don't want to go-- you don't want to get into a loop. So you better cut off your picking of a path to some cutoff value. So you're going to cut it off at m, which is the number of nodes, which is going to be long enough. Actually, we're going to play with that in a bit later, but-- OK, let's just see.

How do I know I did not visit the same node twice when counting? Because I'm just going to go through all of the nodes in some order. Pick any order. The nodes appear in some order in the representation of the graph on the input. So any old order-- I'm just going to go through the nodes in order. Therefore, I'm never going to see the same node twice.

What does step 4 mean? So step 4 is-- step 4 means, for each node, I'm guessing that that node either has a path to it from s or does not have a path to it from s. So kind of thinking about it, the original-- we're out of time.

So why don't I-- I'm happy to discuss this in the office hours. I'm just going to skip over the rest of the slides here and review. We have a missing check-in. Let's just-- I want to make sure everybody's got all their check-ins here. So why don't we just-- if we know NL is equal to coNL, we also-- we showed 2SAT complement is NL complete. It also then follows that 2SAT itself is NL complete, because NL equals coNL.

So I'm going to give you the answer to this just, because I want you all to finish this poll. Still, some of you are getting it wrong. OK. So please answer it quick. And then we're going to end. Are we all done? Get your participation points here. Three seconds. OK, ending.

OK, doesn't matter. So here, we ran over. Sorry about that. Quick review. This is what we didn't quite finish. This is part 5. But we'll finish that next time. OK, when showing PATH is NL complete, we also need to list the nodes for constructing the graph. The slides only mention-- yeah, I kind of skipped that. But yeah, you can just write down all the nodes.

Again, but that's also going to just take log space, as you observed. Yeah, technically, when you're writing down a graph, you write down a list of the nodes, and you write down a list of the edges. I kind of skipped writing down the nodes. But yeah, it's the same-- doesn't matter. So I'm going to say goodbye to you all. Thank you.