[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: OK, everybody. Let's begin. Welcome back. Good to see you all here on Zoom. So we're going to pick up with what we had been discussing last week, which was an introduction to NP-completeness. So we're following on our description of time complexity. We started talking about the time complexity classes, the class P, the nondeterministic classes, the class NP, P versus NP problem, and then leading to this discussion of NP-completeness. And today we're going to prove the big theorem in the field, which really kind of got things going back in the early 1970s by Cook and Levin that there was actually an NP-complete problem, that SAT in particular is an NP-complete problem. And then we'll also talk about 3SAT, which is a useful tool.

Just to remember, we had this notion of NP-completeness. A language is NP-complete if it's in NP. And everything else in NP is polynomial time reducible to it. And if an NP-complete problem turns out to have a polynomial time solution, then every NP-problem has a polynomial time solution. And that's part of the importance of NP-completeness, because since we consider it unlikely that P equals NP, and that there probably are some problems that are an NP, but are not solvable in polynomial time. That would imply that an NP-complete problem would have that property. And so proving a problem being NP-complete is evidence, very strong evidence, that it doesn't have a polynomial time solution. And so therefore it's called intractable. It's a very difficult problem.

So the way we are going to typically show problems NP-complete is by reducing a known-- a previously known NP-complete problem to that problem. Often it's 3SAT, as we've seen in several examples already, or it could be some other example. So let's just survey briefly the things that we've already-- languages that we've already seen, which are NP-complete. So we have the languages SAT, which has a direct reduction from every NP problem. And so we're going to show that today, that every NP language is polynomial time reducible to SAT, which is, in turn, reducible to 3SAT.

And we showed previously that 3SAT is reducible to CLIQUE and HAMPATH. And in recitation, if you went to that, they showed that the SUBSET-SUM problem and the undirected HAMPATH problems are also reducible from previously shown-- well, either from 3SAT, or in the case of the undirected HAMPATH problem, it's reducible from the HAMPATH problem. And the conclusion, once we have these two blue reducibilities shown, then we know that all of these problems are NP-complete.

So the class NP basically breaks down into the NP-complete problems, P problems, problems that are in P, and then there might be problems in between as well. So there are some problems that are not known to be in either category. And in fact, there's an old theorem which shows that if P is different from NP, then there are actually problems that are in the intermediate state. And of course, it's possible that P equals NP, and then everything collapses down to be the same with the tiny exception of the sigma star and empty set languages, which can never be complete.

OK. So that's our quick review. Here is a check-in that I'm going to use to get us ready for the big proof that we're going to be spending most of the lecture on about showing SAT is NP-complete, but just to define a little notation, which you might have-- maybe you've seen already. But I'm going to do it in the form of a check-in.

So I'm sure you've all seen the big sum notation using a big sigma to represent a sum over some set of possibilities. Just as there is the big sigma notation, we have-- you can have other operations that apply to a set of elements. So in this case, we're going to be seeing the big end AND and the big OR operation-- it's going to-- notation, which is going to allow us to talk about taking the AND of many things or the OR of many things, because we're going to be building these Boolean formulas. And so ANDs and ORs are going to be the operations that we're going to be focusing on.

And so just as an example just to make sure we're all understanding this notation, if you have two strings x and y written out in terms of their individual symbols. So they're both of length n. So x is x1 to xn. Y is y1 to yn. And now I write the following expression. The big AND for i ranging between 1 and n of xi equal to yi. If that big AND is true, what does it tell us about x and y? And I'm just going to offer you two possibilities, that either x and y agree on some symbol or are equal, namely that they agree on every symbol.

And so let's just pull that up as a quick poll to get us going here. I just want to make sure you understand the notation because we're going to be using that a lot in describing the polynomial time reduction from languages in NP to the SAT language. OK. I think most of you have got the idea. So let's finish this up quick. So another 15 seconds, please, just to give you a chance to participate. OK. We're going to close this poll. Last chance, last call.

All right. So yes, the big AND-- as most of you can see, the big AND says that x1 equals y1, and x2 equals y2, and x3 equals y3 so they're all-- every symbol in x is equal to the corresponding symbol in y. If we had a big OR instead of a big AND, then A would be the correct answer, because then just there would be some place where they agree instead of every place where they agree.

So let's then start to launch into this big theorem. The proof itself is a mass of details with one underlying idea. And in fact, it's an idea we've seen before. But let's-- before we get ahead of ourselves, let's just understand what we're trying to do. So we want to show that this language SAT is NP-complete. We remember what SAT is. It's Boolean formulas that are satisfiable.

So we already showed that the SAT prob-- so being NP-complete means it has these two features. It's in NP, and everything in NP is reducible to it. So first of all, SAT is in NP, as we've already seen. The witness that shows the formula is satisfiable is simply the satisfying assignment that evaluates to true. So now we're going to pick some language in NP, A, and show that A is polynomial time reducible to SAT.

So this is going to apply to any language A in NP. So let A be some language in NP. It's decided by some nondeterministic Turing machine M in time n to the k. That's what it means to be an NP. I'm going to be ignoring the constant factors when we could carry that throughout the proof. It just would make the description a little more cumbersome and wouldn't change any of the ideas. So let's just say it's M runs in time n to the k, and recognizes or decides this language in A. So it's a nondeterministic machine.

So we've got to give a polynomial time reduction from A to SAT. That's what I have to demonstrate to you. So what does that reduction look like? It's going to map strings, which may or may not be in A, to formulas, which may or may not be satisfied. That's what the reduction has got to do. So saying that out more formally is it's going to take some string w and map it to a formula we'll call phi sub M, w, where M is the machine that decides A and w is the input.

So f is going to map w to this formula, where the string is in the language exactly when the formula is satisfied. And the formula is going to depend upon M and w. That's why it just has-- it's written in this way. So we've seen that kind of thing before. So basically, my job-- we have this language A that's in NP. And now we have some string w, which might be in A, or maybe not is in A. And I have to quickly in polynomial time now produce a formula which is going to be satisfiable exactly when the string w is in A.

So how is that formula going to work? How can I produce such a formula? Which I mean, of course, I don't know whether w is in A or not, because I'm just the polynomial time reduction. And A is an NP language. So polynomial time probably is not enough to solve whether w is in A. So I've got to do that mapping without knowing the answer.

And the idea is that-- and this is where we've seen things like this before-- the formula we're going to construct simulates the machine on w. So in some way it's going to do that simulation. The trick is figuring out how to-- what that means. But the interpretation of that formula is that in a sense describes, says, in my informal language, that M accepts w.

Very much like-- there's a lot of parallel here between this construction and, for example, the construction, the PCP construction, where we made an instant-- given a machine and an input, we made a set of those dominoes, where finding a match forced you to simulate the machine. Here finding a satisfying assignment is going to force you to simulate the machine. So the satisfying assignment is going to be a computation history for M on w.

And I'm just going to write that computation history in a particular way. So it's going to have a somewhat different encoding, which is going to help us to visualize what's going on better. So that's the approach. That's the basic idea of what we're trying to-- what we're going to do. So I'm happy to take a minute if you have any questions about this part. But otherwise, I'll just move on to start doing the actual construction of what the formula looks like. How is it going to do that? How is that formula going to work? OK. So no questions. Hopefully, we are all together.

So first of all, let me describe what my computation history is going to look like. And the situation is a little bit different than what we had before because when we were talking about the Post correspondence problem we had a deterministic machine. And now our machine is nondeterministic. So we're going to call the object-- instead of an accepting computation or a computation history, we're just going to-- we're going to call it a tableau, or sometimes an accepting tableau if you want to emphasize the accepting nature of it. But generally, we're just going to call it a tableau.

So a tableau is really an accepting computation history for the machine on an accepting branch of its nondeterministic computation. So if M accepts w, it's got to have some accepting branch, and the tableau is going to be the sequence of configurations that the machine goes through on that accepting branch. If there are several accepting branches, there may be several tableaus. There will be several tableaus. So there's going to be one tableau for each accepting branch of the machine's computation on w.

If the machine does not accept w, there won't be any tableaus. And so the whole point is that we're going to make our formula represent the statement that there is a tableau. And satisfying that formula is going to correspond to filling out the symbols in the tableau to make it a tableau.

So here is a tableau. So a tableau is just, again, an accepting computation history on some branch, some accepting branch of the machine's computation. The rows are the-- instead of writing the computation history out linearly, we're going to represent it in a table form where each configuration is going to be on a separate row. Now, the dimensions of that table are going to be n to the k by n to the k, because the machine runs for n to the k steps. So there's going to-- if there's an accepting branch, it's going to accept within that number of steps. And we'll have enough rows here to write down all of the configurations that the machine goes through one after the next, row by row, each one having a configuration in it.

And then at the bottom, there'll be an accept. Minor detail, if the machine accepts earlier, we'll just say the machine stays in the-- once it enters an accept, the machine does not change from that point on. So the rule of the machine is nothing changes. And it just remains in the same configuration from that point on.

So important to understand what we're-- if you're not following what I mean by a tableau, you're doomed for this lecture. So it makes sense for you to ask a question to understand what we mean by tableau. So just a few more elements here. So this is going to be the start configuration for M on w. This would be an accepting configuration here down at the very last row. And you might imagine-- I think I've filled out some hypothetical first step of the machine after the start where maybe the machine was-- when it's in-- remember how we encoded our configurations.

So this is the machine is in state q0, looking at the w, the first symbol of the input, w1. And maybe when it's in q0 looking at that first symbol, it moves to state q7, and goes right, and then changes that w1 to an A. And so now here is the head shown moved one position to the right in the new state q7. I mean, of course, that depends on what the machine is designed to do, what the transition function is. But this possibly is what happens.

So does the-- OK, so good. Does the tableau trace all steps of all branches? No. The tableau corresponds to one accepting branch. Each different accepting branch is going to have its own tableau. So there might be several different ways of filling out that second row even. Of course, the first row is going to be-- in any tableau, it's got to be the same. Once I know w-- here I've written down-- maybe I should unpack this for you. It's in the start state here, then here are the first-- here are n symbols of w, w's of length n. So there's w1, w2, up to wn. And then I'm padding out the rest with blanks. So I should have said that, too. But OK.

So I want to make my table n to the k by n to the k because that's going to be enough to represent the entire-- all of these configurations of M running for most n to the k steps, because the machine, even if it sends its head moving to the right as fast as it possibly can, it's never going to have-- it's never going to go outside this box if it only runs for n to the k steps. So this is going to be big enough to represent the entire computation of M. So it means running for n to the k time. And the input w is of length n.

All right. What is k? So k is the running time of the machine. So we assumed from the previous slide that M runs in time n to the k. OK.

So a good question here. How can the tableau be a square table if we have a low number of computation histories, but a lot of tape? Well, the low number computation history, that's a problem. You're not saying that well. You know, each computation history is in a different table. So this, I'm representing a single computation history here. There may be-- there are going to be many configurations. So I can't have a small number of configurations using a lot of tape, because I can only use one more cell, one additional tape cell each time I-- each additional step of the machine.

Yeah. So there's really no difference between this. If you want to think of this as a computation history, that's fine. This is really just the standard terminology that's used when you're proving this particular theorem. Typically people talk about it as a tableau, but it's really just a computation history.

The q state-- I mean, the question about what are these states' q, this is the way we represent configurations. So this means the machine is in a state. They're not all going down the diagonal. The states are going to zigzag here through this picture here, depending upon how the head of the machine moves. So you have to go back and review how we represent the configurations of the machine. Remember the configuration is a snapshot of the machine at a given point.

How do we know that M runs in polynomial time? We're assuming M runs in polynomial. We started off with a language that's in NP. So it's a nondeterministic polynomial time machine, and we're picking one branch that accepts and writing down all the sequence of configurations the machine goes through. Let's move on. And maybe ask more questions as they come to you, and I'll pick out some to answer if that's going to be helpful to others.

So we're going to now construct this formula to say that M accepts w. Again, that was what we-- that's our goal. And it says that a tableau for M on w exists. And basically what that means is we want to say that it starts right, it ends right, and everything in between is right, and then we're going to need some other stuff to talk about how we're going to be encoding those symbols using Boolean variables. So those are going to be the four parts. Here's the start. It starts right. Here it ends right. Here it moves right. And here it talks about the encoding of the symbols into Boolean variables. So those are the four parts of this formula that I'm going to describe.

I hope you got your question answered as to why the total number of columns is n to the k, because it's just big enough to fit the entire-- all configurations that the machine is running for at most n to the k steps, which is what we're assuming.

OK. So now just getting back to that, I'm going to describe these different components now on separate slides. Let me start out with this component phi cell, which is sort of the most fundamental one because it talks about how we're going to be encoding those symbols of the tableau into the Boolean variables. So again, here's kind of the picture to have in mind of this tableau, this n to the k by n to the k table, representing some accepting branch of the machine's computation if there is one.

And so now let me draw one of the cells here. I'm going to magnify it. So this is the i, j cell here. And I'm going to-- there are going to be a collection of Boolean variables associated with each one of the cells. So each one of the cells is going to have a bunch of variables all to itself. And those are going to be basically indicator variables. They're going to indicate which symbol that cell gets to have in it.

So again, picture here-- in this tableau, we don't know how it's going to get filled out. But however it gets filled out, each one of these cells gets some symbol. And that symbol could either be a tape alphabet symbol, or a symbol representing a state. That's the way we do our configurations. So it could be a tape alphabet symbol here showing the magnification. Maybe it's the tape alphabet symbol, which represents the blank symbol. Or maybe it's represents some state.

Now, how am I going to encode that with variables? So this-- let me-- this is the collection of variables that's going to apply to the entire formula phi sub M, w. Each cell is-- so each cell i, j is going to have a set of variables, one for each possible symbol sigma that's in the configuration alphabet, namely a tape symbol or a state symbol.

So I'm going to have-- well, maybe this will become clear as I'm writing it. So if I turn the variable x i, j, sigma equal to true, that's just a way of saying that cell i, j contains a sigma. So if I have x i, j, a, that means the symbol contains an a. So I'm going to illustrate that now for you. So imagine you have lights representing all the different x i, j's for the different sigmas. Didn't say that well. So we're in cell i, j. So these are all x i, j variables. They're all x i, j sigma variables for the different sigmas that can go in this cell. So all different possible sigmas. So this is gamma union Q.

So now if I have an A here in that cell, so then the variable x i, j, a is true. And just helping you visualize that, that's going to correspond to turning the light on. There's going to be a light associated with each one of these variables. And it's going to be turned on when that variable is true.

Similarly, if I have the blank symbol is the thing that goes in that cell, then that variable gets turned on. I hope you can see it. Maybe it's a little bit small on your screen. The x i, j blank, a variable is true. And similarly, if it's q7 here, the x i, j q7 variable is true. So that's the way we're going to be encoding the contents of these cells using these indicator variables.

And now we have to start making some Boolean logic to make sure that those variables reasonably represent the cell con-- the contents of these cells. So for example, what would be the first thing that comes to your mind? Well, we better not have two lights going on in any one of the cells, because then we have two symbols in that cell. And that's not allowed. Each cell is going-- we want each cell to have exactly one symbol. And that corresponds to each cell having exactly one of its lights turned on, or equivalently, each cell should have exactly one of the variables be true.

That's the very first part of the formula is just going to say that. So let me show you what that looks like. So here we're talking about this phi cell. It says that there's exactly one light on per cell, or in other words, exactly one of the x i, j sigmas is true for each i, j.

So this is how I'm going to actually express that using my Boolean formula. I'm sort of color coding the different parts of the formula, which I'm writing out to you here in English. So first, I want to say this. So in every cell, there's at least one light that's on, and there's at most one light that's on. So here's the green part. This is going to say at least one light is on. So I'm going to say that by taking all of the var-- all of the symbols that can appear in that cell, and taking an OR over all of those different associated variables. So it's either got the first symbol on, or the second symbol is there, or dot dot dot, or the last symbol is there. One of those has got to be there.

I'm going to write this using my big OR notation. So for sigma appearing in this set of possibilities, one of those variables has got to be on at least. That's what this big OR tells you. Now we want to make sure that there's at most one that's on. So that there are not-- there's at least one on, but there are not two that are on. So I'm going to have an additional part of the formula here, which says-- and I hope you can read this. It's a little small.

If I have two different symbols, sigma and tau, that are configuration-- possible configuration symbols, where sigma and tau are not equal. So that's I'm reading it out to you if it's too small for your screen. Then I'm going to say it's not possible. So I have the negation of x i, j sigma, and x i, j tau. So saying it another way, it's not the case that that cell contains both sigma and tau for any two symbols sigma and tau as long as they're different.

And we want to take these two formulas and add them together. And this tells me in the cell i, j, there's exactly one of the variables is true. Exactly one of the lights is on. And that's going to represent which symbol goes into that cell. And then I want to take the AND over all possible cells to make sure that I'm going to now apply that everywhere. And so now I do an AND for i and j ranging between 1 and n to the k to apply this logic throughout the picture.

Yeah. Sigma unions-- asking sigma union Q contains the input, output, yes. This is not a sigma. This is a gamma. Gamma is the tape alphabet. This is any symbol, including an input symbol, from sigma is going to be in gamma. So this is any symbol can appear on the tape. And this expression here, that is the expression phi sub cell.

So here's a little check-in for you. But maybe before we jump into the check-in, let's just make sure-- it may be better to take some questions, and then we can ask the check-in for you. Do you understand? I mean, if you're not getting this, you should try to figure out how to get it, because this is really just the foundation. It only gets more-- it is not a very complicated proof once you sort of get the idea of what's going on, but if you're not getting this part, you won't be able to get the rest.

OK. I have no idea what the-- what that means, but I'll read it out to you guys. This looks like a one-hot tensor encoding, same from common-- same form commonly used in ML. OK. It's just an indicate-- I would just call them indicator variables.

Why is it sigma union Q for the big AND here? Sigma union Q? You mean gamma union Q?

[GASPING]

This is wrong. OK. Now I understand why everybody is upset. This should be a gamma, not a sigma. There's a boo-boo. Sorry about that. I don't know if I can fix that without breaking the whole slide, so I'm not going to even try. This symbol here should be gamma, not sigma. It's a typo. Thanks for catching that.

Yes. So the question is-- OK. So phi sub cell is just trying to make sure that the encoding represents setting a bunch of symbols into the tableau. Not-- so each cell is going to have one symbol exactly, not two, not zero. So that's what phi sub-- if you've satisfied, if you've set the variables to satisfy phi sub cell, then there's going to be one symbol on in each-- one symbol in each of those cells.

Now, another question, this is not a CNF. No, this is not a CNF. That's the second half of-- that's going to be-- I hope we don't run out of time. But I have a way of converting general SAT formulas to CNFs and preserving satisfiability. So we're going to do that reduction afterward.

Here's a little check to see if you understand at some level what's going on. How many variables does this formula actually have? Is order n? Order n square? n to the k? Remember, k is the running time of the machine. Or n to the 2k? What do you think? So I mean, for how many variables. I mean that in all of phi sub M. How many variables do we have all together in this formula if that's what the question is.

And here are the variables. So describing them here-- x i, j, sigma. OK. I'm going to close this. So pick something. All right. Ending polling, 1, 2, 3. OK. Yeah.

OK. That's a good question. So first of all, the correct answer is, in fact, D. It's order n to the 2k. Now, I'm getting some questions about, what about the size of gamma and Q? Well, those are going to be fixed. They depend only on the machine, but they don't depend on n. So thinking about it functionally in terms of n, that's going to be a constant multiplier. And so it's going to be absorbed within the big O. So that's why we have-- these are constant relative to n. These are fixed.

There are not n to the k possible symbols. There's a fixed number of symbols. It depends only on the machine. So we're looking at a particular machine, and what happens when you look at large inputs.

So why is D and not C? Well, don't forget-- how big is this table? This is n to the k by n to the k. So there are n to the 2k cells, n to the k quantity squared, or n to the 2k cells here. And so there's a collection of variables for each cell, some fixed number of variables for each cell. So that's why its order n to the 2k. Good. So let's move on. I think we're actually-- hold on.

So we have one more slide, and I think then we have a break after. So now let's next talk about constructing two more pieces of the phi sub M, w formula. So we already got phi sub cell done. Let's look at phi sub start and phi sub accept. And phi start is going to tell us that the start configuration has exactly these symbols. And phi accept tells us that the bottom configuration contains an accepting state somewhere.

So how are we going to write that down? Well, first of all, I'm going to write these down just cell by cell. So first of all, phi start is going to say the cell 1, 1 contains a q0. I mean, I know what the start configuration should be, because thinking of me, or think of us as the reduction, the reduction is given M. It's given w. So it knows what the start state is. It knows what the symbols of w are. So it knows what that start configuration is. It's just q0 followed by the n symbols of w followed by blanks.

So it wants the very first cell in the left-hand corner here to be a q0, the start state of the machine, which it knows. So it's going to say x sub 1, 1, q0, that has to be turned on. So it's going to be AND of a bunch of variables here. And in order to satisfy phi sub start, all of those variables have to be set to true. So that means we have to have a q0 in that cell.

So now we're going to do the next cell here. The 1, 2, the next cell of the start configuration. So phi start is going to have x 1, 2. So that's the next place. It contains w1. And so on. x 1, 3 contains w2, all the way up to wn, and then there's going to be a bunch of additional parts which say that we have blanks in the rest, just spelling out exactly all of the symbols in that top row. Because that's what the phi start formula, or sub formula of the overall formula we're making looks like.

Now let's take a look at phi accept. Phi accept, because I'm just looking for q accept to appear somewhere in that bottom row, I'm going to do that in terms of an OR. So here is-- the variables now, notice, have n to the k because it's the last row in the table. So row n to the k here.

And then I'm going to vary j from 1 to n to the k. So j, the column number, is going to range from 1 to n to the k. And I'm looking for that to accept. So x n to the k j, where j's vary, and q accept. One of those has to be true. One of those has to be turned on. And so that's why it's a big OR. And that's my phi accept piece.

And now we'll take a little break. And feel free to ask me some more questions. Let me just start our clock. Go grab yourself some coffee, or ask me some questions. I'm happy to answer them. Why don't we check that q accept only appears once? Is it possible for q accept to appear twice? That's a great question.

So that would definitely be a broken configuration if that happened, because a configuration can have-- must have exactly one state symbol appearing. The way we're going to enforce that is with the phi move part of the formula, which we haven't seen yet. So phi move is going to guarantee that the machine is acting correctly, so that all of the rows of the tableau are all legal configurations, and they all legally follow from the previous. So really, in a sense, the hard-- heavy lifting is coming in phi move, but it's really not that bad.

Somebody says, out of curiosity, how close is this intuition proof to the actual proof? This is the actual proof. There's no-- I'm not hiding anything. I mean, we're being a little loose here, but you can turn this-- we're not cutting any corners here. This is exactly how the proof goes. And so not-- you're getting the real deal here.

Somebody wanted to see the previous slide, so here we are. Whoops. Is there something you want to ask? So phi cell says there's exactly one symbol per cell. And the variables are set in a way in terms of thinking of them as indicator variables. There's exactly one variable set to true in each cell. So there's exactly one symbol per cell. That's what phi cell tells you. If you don't have that, then you have a mess. So you've got to start with that. And then with the-- other things are going to be additional conditions, which when satisfied are going to enforce the rest of the properties that we want.

Why would the proof fail if we replace n to the k with 2 to the n to the k? So OK. I presume where does-- we'll just use the polynomial running time of the machine of M, the nondeterministic machine. I mean, if you had an enormous tab-- we have to show ultimately that this reduction is a polynomial time reduction. And that's going to depend on how big the tableau is, because that's going to tell us how big the formula we're producing is, phi sub M, w. If phi sub M, w is exponentially big, we don't have a prayer of being able to output that formula in polynomial time.

If there were less than n to the k steps, do we repeat the last configuration? Yeah, that's what I said. If the machine ends early, the last configuration just stays there. So we're going to modify the definition of the machine slightly so it just stays. Yeah. OK.

Yeah. Let me not take the other-- there's a bunch of other questions, some of them a little on the technical side. Let me-- maybe I'll try to address them as they come along if it turns out to work to do that. So the break is over. Why don't we-- is it possible that the encoding configuration will not fit in n to the k? So the question is, is there a possibility that the encoding of the configuration won't fit in n to the k? If the machine runs for n to the k steps, the configuration has to fit within n to the k, because it can't use anything more within n to the k steps. So think about it. But no. The answer is the configurations-- if the machine runs in time n to the k, that whole tableau is big enough to write down the entire computation history.

All right, so let's continue. Phi sub move, this is, in a sense, the part which is going to tell us that we started right, we ended right. Phi cell says every cell contains one symbol. And now we have to say that the whole interior is correct. How are we going to do that? So these are the parts we've already done.

And the way I'm going to describe that is in terms of these kind of little windows I'm calling neighborhoods. So imagine here we have a 2 by 3 rectangle, which I'm going to call a 2 by 3 neighborhood. And what I'm going to argue, but I'm not going to prove here. I'm just going to really state it, but it's really just a sort of more or less obvious fact. But the proof-- the book has the formal proof, that if any-- every one of these here is legitimate, is legal according to the rules of the machine, if every single-- imagine you have these-- oops. Let me put myself back on here so you can see me.

If you have here every 2 by 3 window, you can take this as a window, and you slide that over the entire picture of the tableau. And everything here looks OK as far as the running of the machine. So I'll say what that means in a second. But if everything looks locally fine everywhere, then the whole tableau has to be a valid tableau in terms of the rules of M.

Maybe it's easier if I describe what I mean by these being legal. So these neighborhoods, these 2 by 3 neighborhoods are legal if they're consistent with M's transition function. So I'm going to describe rather than-- I mean, to do this formally, I would have to go through a process that we went-- like what we did when we went through the construction for the Post correspondence problem, and say if the machine moves left, this thing happens. If it moves right, that's-- I think that it's not really necessary. You can kind of get the idea very clearly by doing it at a little bit of a higher level.

So let's look at what I mean by a legal neighborhood. So a legal neighborhood is a setting of the values, the six values of this 2 by 3 neighborhood, in a way which doesn't violate M's rules. So for example, if M, when it's in state q7 reading of b, goes into state q3 and moves left, then this would be a legal neighborhood, because it shows the head moving left, the b becoming a c. So reading a b, I should also say that it converts that b to a c. And we just head left into state q3. So this would be a legal neighborhood if that's the way-- so being legal depends upon the transition function of the machine. So given the transition function, that's going to tell you which are the legal neighborhoods.

So another legal neighborhood-- this would always be a legal neighborhood-- is that if nothing changes. So that means the head of the machine was somewhere else. And so whatever was on the tape in this step is going to be the same stuff in those places one step later.

Here's another possible legal neighborhood. If the head suddenly appears on one of the cells either in the left or the right, that would correspond to the machine moving its head from somewhere off the neighborhood into the neighborhood in that step. So this could be a legal neighborhood, provided the machine actually does move its head left into a state q5 at some point under some conditions.

And here is another kind of a weird legal neighborhood. If you have a, b, c, and then the a changes to a d, that could also be a legal neighborhood if the machine transition function allows an a to get converted to a d when there is some machine-- when there is some state reading that a, and that state also moves its head left. So it doesn't move into this picture.

So those are examples of legal neighborhoods. Let me show you some illegal neighborhoods. Just I'm doing this-- this is kind of a proof by example now. This is perhaps the most intuitive part. But I claim that this is easy to turn this into something airtight and formal. So this would be clearly illegal. If you have a piece of the tape in the previous step where it's a, b, c, and then suddenly the b changes to a d. The symbol on the tape changes out of nowhere without having a head nearby to a different-- to something else. That could never happen. So that would be illegal.

Another thing that would be illegal is if a state appears from nowhere. That could never happen. Or if it just disappears. That could never happen. And here's another-- here's an interesting one. If a state becomes two states. Don't forget the machine is nondeterministic. So the machine in principle could move its head left on one branch and move its head right on a different branch, but those would have to be in different tableaus. They can't be in the same tableau, because that doesn't correspond to any of the threads of the computation, those with multiple threads.

And I say this because if you think about my claim, which is going to put down over here, that if every 2 by 3 neighborhood is legal, then the tableau overall corresponds to a computation history. This illustrates why it's not enough to have a 2 by 2 neighborhood, where you really need the 2 by 3. Because if this was a 2 by 2 neighborhood, if you just look at these four-- this leftmost 2 by 2, that could be a legal neighborhood if it was a 2 by 2, if the rules of the machine allowed for that. And the right four box-- right four cells could also be a legal neighborhood. So you could have something that looks OK from the perspective of 2 by 2 neighborhoods, but globally, in terms of the overall tableau, it's completely nonsensical because it has multiple hits.

But if you have a 2 by 3 neighborhood, it's big enough to prevent this situation from occurring. And then you can check the details. And I think it's very plausible that it guarantees that the overall tableau is legitimate if all of the 2 by 3 neighborhoods are legal. And so that's what we're going to turn into a Boolean expression.

We're going to say for each cell that the set-- for each neighborhood-- so here's a neighborhood at the i, j location. I'm calling this position here sort of the home location for that neighborhood. For each neighborhood, it has to be set to one of the legal possibilities. And there's, again, only a fixed number of those because there's a fixed number of possible symbols that can appear in those cells. So this is that fixed number to the sixth power at most.

And I'm going to say that the cell in the upper left, which would be this one, is in r. And this one here is an s. And this one is a t. And this one is a v, if you just trace down what the indices are telling you. It says that that piece of the tape, that piece of the tableau here is set according to one of the possible legal settings. And we're just going to OR over all of those possible fixed number of legal settings. And then I take an AND over all possible tape cells, over all possible neighborhoods. And so that's going to be my phi move. And that's it.

OK. Let's see. Can I explain again the third example of illegal? So this one over here, I presume, I'm being asked about. Well, if the machine is in a state q7 reading a c, the head has to move either left or right. So at the next configuration, there's got to be a state symbol appearing either in this cell or in this cell. And here the tape-- the head has basically just vanished with nowhere to-- it's gone. That could not happen according to the rules of the machine the way we talk about Turing machines. So that's not possible. So this would be an illegal neighborhood. You want to prevent any of the bad stuff from happening anywhere in here. So only good stuff can be happening locally. And that guarantees the overall picture is OK.

Do we have to check that the head doesn't leave the tableau from the left most to right right? Yeah, there are some little details here like that. So the question is, do I have to make sure that the-- yeah, you probably need to mark. I think the book probably does this correctly. You may have to mark the left and right ends to make sure that-- I mean, the right end is not a problem, because the machine can never go off the right end.

And if you design the machine so that it never moves its head off the left end either, which you can do, then you wouldn't have to worry about that possibility. But otherwise, you would have to put some sort of delimiter here to enforce the head not moving off the left end. So there are some details like that, too.

There will be two heads in the same row. No, this can-- I don't know what you-- somebody says, there will be two heads in the same row. Please elaborate, because this is designed not to allow two heads in the same row.

Could I go over the OR for legal again? OK. The OR, the big OR here, what I have in mind is I take-- there's going to be-- first of all, I look at the machine and I look at the transition function. And based on that, I write down the list of all the legal 2 by 3 neighborhoods. So all the settings which correspond to legal 2 by 3 neighborhoods. There's going to be some fixed number of those. 100. There's 100 possible legal neighborhoods of which I've written down here four. But maybe there's some number, say 100.

So now there's going to be an OR over those 100 different possibilities. It's either going to be this legal neighborhood, or some other legal neighborhood, or some other legal neighborhood. And for each one of those legal neighborhoods, I'm going to say, well, the variables are set according to that legal neighborhood, or the variables are set according to the next legal neighborhood, or the variables are set according to the next legal neighborhood, and do that 100 times.

One of those has got to end up-- I mean it's an OR, so one of them has to work. Otherwise, the formula fails. And it will be false, because you're going to AND that over all of the neighborhoods in the picture.

Is it possible to have a head on the far left of the configuration and one on the far right? You mean a head over here and a head over there? I mean, how'd the head get there? It can't happen. You know, the head has to come from a head above it. If you're going to be worrying about the details of the boundaries here, all that's fixable. So let's not lose sight of the main idea. I mean, if you understand the main idea, you can fix the little details. So I want to make sure you understand the main idea of what's happening.

So let's finish up this proof. So in summary, we gave a reduction from 8 to SAT. This is what we needed. It was in those four pieces. And you really just need to argue that that formula we're building is not too big. And it's going to be basically the size of the tableau if you look at what we constructed. The number of variables is roughly the size of the tableau. And the amount of logic that we're putting into the formula is also going to be a fixed amount of logic independent of n for each of the variables in that tableau.

And somebody asked me about the size-- how big the indices are. The indices for the x i value, the i and j values, technically, they're going to be numbers between 1 and n to the k. So you're going to have to write those down. And so that's going to be a slight additional logarithmic cost to write those things down. It's not really that interesting a point. And so the overall f is going to be computable in polynomial time because the output is not very big. And it's also not complicated to write the output down. So that's the end of the proof.

I can take a couple of questions. Why can't we just check that the whole-- this is a good question. Why can't we just check that the whole row is legal? You can check that a row actually is a configuration. But to check that the row follows from the previous row, ultimately, the operation of a Turing machine is a local thing. I mean, the way it moves from one configuration to the next depends locally on how where the head is. And so really, that's just another way of-- the way I'm saying it is just really checking the whole configuration, but just doing it locally. I don't know if that's satisfying to you.

Why don't I move on because I just want to make sure we have enough time to get to the very last part, which is a little bit-- I'm afraid, a little technical. So we're going to kind of shift gears now and talk about reducing SAT to 3SAT. And let's see how it goes. I don't always have the most success with presenting this little piece, because it's slightly a technical argument. But if you don't get it, don't worry. Just you have to accept that it's true. But I'd like to show it to you just to make the whole presentation complete in that sense.

So I'm going to give a reduction that maps general formulas to 3CNF formulas. So that's how we map SAT to 3SAT. If you remember 3SAT is satisfiability but for three CNFs. So a conjunctive normal form in the form of those clauses, which are ANDed together. And each clause is an OR of a bunch of literals, which are variables or negated variables.

So I want to convert phi to phi prime, which is 3CNF formula, but preserve the satisfiability. And phi prime is not going to be logically equivalent to phi, because I could do that, too. I can convert any formula to a logically equivalent CNF formula. Maybe not even a 3CNF, but yeah, you won't be able to get a 3CNF, but you can get a CNF. But it might be exponentially larger. And that's not good enough. I have to do the reduction in polynomial time. So I can't generate a much larger formula that's exponentially larger.

And so I'm going to do that by adding additional variables. So it won't be logically equivalent because the new formula is going to have additional variables in it. I'm going to kind of do it by example. And we'll see how that goes. So here's phi, which is not in 3CNF. It's not even in CNF, because it's got ORs of ANDs appearing, which is not allowed to happen in a CNF.

So how are we going to convert that into a 3CNF formula preserving the satisfiability? And just working it through with this example, I hope to at least give you some idea of how you do the conversion in general. So first of all, I'm going to represent this formula as a tree using its natural tree structure. So you understand. So a AND B becomes a AND b written as a tree. And then I OR that with c. So I get the tree structure here in sort of the natural way.

And I'm going to label all of these intermediate nodes, which are associated now with operations. And I'm assuming also that the formula is fully parenthesized so that each operation I'm only thinking about is applying just the two-- it's a binary operation. And let's ignore negations for the minute, because negations, you can always push those through down to the leaves. But it's just going to make it too complicated. So negations turn out not to be a problem.

So there's only going to be negations at the level of the inputs, not at the negation operations in the middle. So we have this tree structure here. And now I'm going to use these two logical facts. And I don't know if-- you've probably all seen ANDs and ORs, I hope. Otherwise, it's going to be really tough. But there's also other logical operators, such as the implication operator, where you have A implies B sort of as a logical operation. And so this requires that if A is true, then B is true. However, if A is false, B can be anything. And similarly if B is true, A can be anything.

The only thing that's prohibited is that if A is true and B is false. That's the only thing that would be invalid. And so if you think about it, that's going to be equivalent to saying that either A is false or B is true. One of those has to be. And that's going to be logically equivalent to saying that A implies B.

Another logical equivalence may be more familiar to you. It's just simply De Morgan's law, which says that if you have the not of A AND B, that's equivalent to saying the not of A or the not of B. I'm going to make use of both of these.

Now, here, I want to-- I ran out of room on this slide. So I'm going to take myself out of the picture here for a minute. I had no place else to put this. So here we have-- if you're going to think of the AND in terms of its truth table, so here's a and b in terms of a and b. So 1 and 1 is 1, but all other settings of a and b yield 0 for the AND.

And I'm going to represent those-- if you imagine a and b is going to be called c. I'm going to represent this information with four small formulas, which taken together, you AND them together, are going to force c to have the correct behavior associated with a AND b. So if a AND b are both 1, then c is 1. If a is 0 and b is 1, then it forces c to be 0. And similarly, every other setting besides a AND b being true, for C to be false, which is what you want when you have AND.

So I'm going to write this expression here down with z1 being in the place of c by just taking those four expressions and ANDing them together. So this is exactly those same four expressions written out linearly. Now I want to do the same thing for z2, but now that's written in terms of an OR. So there's a slightly different truth table here up in this corner.

So now if either one is 1, we get a 1 result. And so now if a AND b are true, you get c is true. However, if a is true and b is false, that still implies c is true. So I'm going to write down those rules for specifying how z2 must be set. And each one of these things is going to get converted into clauses, three clauses with three literals, using these rules over here. So I'm going to do that for each zi.

And lastly, to make sure the whole thing is satisfied, which means there's an output of one here, I'm going to have one clause associated, which says that z4, the output, is 1. Now, I can convert all of those when I have a AND b implies c. That's logically equivalent to not a OR not b OR c. And the way you can see that is really by repeated application of these rules here.

We're running a little low on time. So maybe you'll just have to check this offline. But quickly, a AND b implies c using the first equivalence is the not of this part, OR C. And then I can use De Morgan to convert that not of an AND to an OR of the nots. And then I can remove the parentheses because OR is associative. And so I get a clause, which is what I need. So each one of these guys is really equivalent to a clause. And so I just get a bunch of clauses.

And actually, technically, this needs to be three, a copy of three things here. It should be z4 OR z4 OR z4, which is a lot. So check-in-- [INAUDIBLE] I realize my check-in is broken, because I only realized that last point just now as I was talking. So the actual value that you get in terms of-- oh, no, the number of clauses is correct. No, I take it back. This is fine.

So if you understood what I was saying, hopefully, you can see how big the formula phi prime is in terms of the number of operations in phi. So let's see how many people get that. I acknowledge this may be a little on the technical side. OK. I'm going to close it, close it down. Please enter your value. OK.

Yeah, the correct answer is 4k plus 1, because each one of these operations is going to end up being a row in this picture. Each operation is going to have a variable associated to it. It's going to become a row in this picture. And so then each row is going to have four clauses, which define what you need-- set what you need in order to force that variable to have the right value corresponding to that operation. And so then you need-- and you need one extra clause here for saying that this whole thing evaluates to true.

So that's all I wanted to do today. We proved those two main theorems. And now we know that there are NP-complete problems. And all of the other problems that we can get from-- by reductions from these problems are also going to be NP-complete as long as they're in NP. So that's it. Feel free to put some questions into the chat, or move on to whatever else you're going to be doing next.

So a good question here is, why is phi prime not logically equivalent with this construction? It can't be logically equivalent. Logically equivalent means that it gives you exactly the same function. If you set the variables in the same way, you get the same result coming out. Well, phi prime has more variables than phi does. So it wouldn't even make sense to talk about logical equivalence because they're two functions on different numbers of variables. So in that sense, it doesn't really make sense.

What you could say is that for every setting of the overlapping variables, so the variables that appear in both phi and phi prime-- so those are the original variables of phi-- there's going to exist some setting of new variables, which is going to make the-- there's going to be-- there will exist some setting of the new variables which will make the two formulas agree, but that's not the definition of logical equivalence.

So why-- going back to the proof, the satisfiability proof and the legal neighborhoods, could I go over why the number of legal neighborhoods is polynomial. The number of legal neighborhoods is not only polynomial. It's constant. It depends only on the machine. It does not depend on M. Because each cell can have at most some fixed number of-- can have the number of tape symbols plus the number of state symbols. That depends on the machine only.

So now we have six tape cells for the six cells in a 2 by 3 neighborhood. So you're going to have that number to the sixth power. But still, it's a constant to the sixth power. It's still a constant. It doesn't depend on M. So it's not a question of even being polynomial. It's a constant value. It's a constant multiplier if you want to think about it in terms of the size of the formula that's going to result.

Don't forget we're trying to make a formula which is-- the reduction has to be polynomial. It's a polynomial time reduction. So that means that as n increases, the time to calculate the reduction increases as a polynomial. But we're fixing M. So M does not change. So therefore anything that depends on M only is just going to be a constant impact on the formula. It's not going to be-- it doesn't depend on M.

OK, everybody. Bye bye. See you.