[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: Hi, everybody. I hope you can hear me. Give me a thumbs up if you can. Good. I see them coming through. So we're going to get started.

So getting back to what we've been doing. We've been talking about space complexity. Measures how much memory the algorithm requires or various problems require. And we defined the space complexity classes, space f of n and non-deterministic space f of n, the polynomial space and non-deterministic space classes, and gave some examples and so on. And today we're going to pick up where we left off last time.

One of the examples, which is going to be an important one for us, concerns this latter DFA problem. So I'm going to go over that again, give a little bit more emphasis to the space analysis, which I got some questions about last time. And then we are going to move on from there and prove Savitch's theorem and then talk about a complete problems for PSPACE and show that this problem TQBF, which we introduced last time, is actually a PSPACE complete problem. But all in due course.

A little bit of review. So we defined what we mean by a Turing machine to run in a certain amount of space. That means it uses at most f of n. If it's running in space f of n, uses at most f of n cells, tape cells, on every input of length n. And similarly, a non-deterministic Turing machine does the same. But in addition to that, the non-deterministic machine has to halt on every branch of its computation. And each branch of its computation has to use at most that bounded amount of tape cells. So we're going to be talking about non-deterministic space computation today as well. It's going to be relevant to us.

So we defined the classes, as I mentioned, and the polynomial and the PSPACE and non-deterministic PSPACE classes. And this is how we believe they relate to one another. The classes coNP and NP as well. And of course, as I mentioned last time, there are some very major unsolved problems in this area. So everything could conceivably collapse down to P, which would, of course, be very surprising. But we don't know how to prove otherwise.

And the big theorem that we're going to prove today is that polynomial space and non-deterministic polynomial space actually do collapse down to each other. And being the same class. So in contrast with the situation that we believe to be the case for time complexity where we believe converting non-deterministic to deterministic gives an exponential increase for space complexity, it only gives a squaring increase, as we'll see. So any questions on any of this? We will just march into a little review of this latter problem.

So reviewing some of the notation. And let me emphasize that. So the big theorem we're going to be proving today is that PSPACE and NPSPACE are equal. And also we're going to be talking about PSPACE completeness. But both of those involve proving theorems. In the first case, Savitch's theorem that converting non-deterministic to deterministic spaces is squaring. And in the second case, proving that TQBF is PSPACE complete. Both of those theorems can be thought of as generalizations of this theorem here that the latter DFA problem can be done in the deterministic polynomial space or n squared space.

So it really pays to try to understand how the proof of this theorem works. Because in a sense, this theorem is a more concrete version of what we're going to be seeing in those other two theorems in a somewhat more abstract form. So I like understanding things in a more concrete way first. So that's why this is a good example to start out with. But really in the end of the day, it's the same proof just repeated for those three theorems. So this is really three for the price of one. Three theorems, one proof here. So you're going to be seeing the same proof repeated three times but in different levels of abstraction.

So let's review again. I know some of you got it, but maybe some of you didn't. And let's just try to be clear on the algorithm to solve the ladder DFA problem. So if you remember, first of all, let me just jump on ahead. The ladder problem is-- a ladder, first of all, is a sequence of strings that change one symbol at a time that perhaps connect, go from one string to another. So you're going to go from work to play, changing one symbol at a time. So we gave an example of this or you can easily come up with an example of doing this.

But the computational problem is can you do it? Can you get from this string to that string and stay within a certain language? So it might be the language of English words or it might be the language of all strings that some specific DFA recognizes. So it might be all of the string. These might all be strings that some DFA accepts or might be English words or some other rule. And so that's what we mean by trying to test if there's a ladder.

And so the ladder problem is, well, I don't think I wrote down the ladder problem itself. But the bounded ladder problem is basically the same idea. You're given the DFA. You're given the strings u and v. And now this is the bounded version of the problem where I'm going to give you a limit on the number of steps you can take. So I'm illustrating that here. So you're going to be given a b. And you want to say, can I get from this string to that string within b steps? And we had a notation for writing that. Going from u to v, if there's a ladder that connects u to v within at most b steps.

And so the bounded ladder problem, which I'm introducing because I'm going to be aiming toward a recursive algorithm to solve this problem is can I get from u to v by a ladder, changing one symbol at a time, where each string along the way is accepted by b, and I'm only allowed b steps. Little b steps. So that is the computational problem that I'm going to be solving with the algorithm that I'm going to describe.

So the algorithm I'm going to call BL for bounded ladder problem. And here is the input. And the algorithm is, first of all, going to look to see if b equals 1. If I'm just trying to get from u to v in a single step. In that case, it's a very simple problem, because you want to test, obviously, that u and v are accepted by the automaton. And they just have to differ in one place. And then you have a very simple one step ladder that takes u to v. So for the case b equals 1, it's very simple.

For larger values of b, we're going to solve the problem recursively in terms of smaller values of b. So for b greater than 1, we're going to recursively test. If you're trying to solve the problem can I get from u to v, instead we're going to try each possible halfway through. We don't know that it's halfway through. So we're just going to try each possible string. And we're going to test can we get from u, the initial string, to that new string, that w, in half the number of steps. And can I get to the final string v in half the number of steps? If I can do that, then I can get from u to v the total number of steps b.

So I'm just going to try to do this one w at a time for every possible w. This is going to be very expensive in terms of time, but we're not worried about time right now. We're trying to cut down on the amount of space that we're using. And this is going to be a big savings in space. Let's not worry about the division, b over 2 here. All of the divisions, we're going to be seeing this several times going forward in the lecture, we'll think of them rounding up. But I'm not going to make the notation look cumbersome by writing that every time.

OK. So here we go. Here is some candidate w string, which is halfway through. Recursively test. Can I get from the starting string to that w and from w to that ending string? If I can, if I find such a w, then I accept. And if I try all possible w and I never manage to find a way to make both the top and the bottom work, then I know I cannot get from the starting string to the ending string within b steps. And so I reject.

And now I'm going to solve the original unbounded ladder problem by simply putting the biggest possible bound into the bounded ladder problem. And that's this value t, which gives the very trivial bound of the total number of possible strings that I can write down within my length m that I'm working with. So this if sigma is the alphabet of these strings, it's just sigma to the m. That's all possible strings. Of course, that's going to be a maximum size on the ladder.

So now how much space does this take? And I think this is where people got a little bit lost in the lecture last time. So I'm going to try to animate this. I don't know if that's going to help or not. But in the end of the day, you're just going to have to think through how do you account for the cost of this recursion.

But the main thing, to start off, you have to make sure you understand the algorithm. And if you get from here to there, we're going to try all possible midpoints and then solve the upper part and the lower part recursively, reusing the space. That's the way we're going to get a saving. By solving this problem, reusing the space that we use to solve this problem.

So I'm going to try to show this to you on actually how the space gets used on the Turing machine. You can kind of think of here's the input and then after that is going to be the stack for the recursion. If you're not that familiar with how to implement recursion, it doesn't really matter. But you can just think about what the algorithm needs to keep track of. And so as it's trying every possible w, so just in order like an odometer, just trying every possible string, eventually maybe it finds a string that's in the language that hears an English word, one of the first English words of length 4 that you might run into.

And so now it makes sense, actually, to do the recursion. So that's all. Every time you're going to have to have a register or a location on the tape where you're going to be writing down those different w's. So let's say it's over here. And we're just going to go through. I hope that's not too small for you to see. That's really where that action is happening. And finally, maybe you got to the string w. Now you're going to try to do the recursion. So here as you're doing the recursion on the top half again, you're going to be cutting-- you're going to be finding a new w for the intermediate point just solving this upper problem where we're testing if I can get from work to able. Later I'm going to have to deal with getting from able to play. Good.

So here, again, we're going to be fixing able, fixing the first w. We're going to try every possible way of getting from the start string to that middle string. So we're going to try every possible thing here. Eventually maybe we find some other string in the language. We get down to the string book. And that's all going to get stored. You can't forget the string able. But now we're going to use some more space to store those candidates. So that's a second version of w deeper in the recursion. So here we're going to be triangle to possible strings here.

Again, eventually we get to some string book. And if that succeeds in getting us from work to able via book, now we're going to jump down to do the bottom half, to see if I can get from able to play as a separate problem, which gets solved in the same space. So now here we're going to try all these possibilities getting from able to play. Maybe call is the right intermediate string there. And so now we're going to erase the book and now we're going to solve the lower sub problem in the same location. I hope this is helpful. This was a lot of work making these animations.

So the point of all this is every time we go down the level of the recursion, there's another register whose size is big enough to hold one of the strings, is needed. And that register gets reused times throughout as we're going through this recursion. So anyway, I hope that's helpful. Anyway.

So each level of the recursion adds another order in to record the w. And so you have to do-- how many levels do we get. Well, the depth of the recursion is going to be how many times we end up having to divide this picture in half until we get down to 1. And so the height of this when we start off is going to be basically an exponential in m. m is roughly the size of n.

So when you take the log of that, you're going to get-- you're going to pull down the exponential. So it's going to be order m or, which is, again, roughly the same size as the input. m is like half the input because the whole input is u and v. m is just the size of u. And so each level requires order n, and the depth is going to be order n deep. The log of the initial height of this ladder. And so the total space used is going to be order n squared.

So why don't we just take a minute? I'm happy to spend a little time going through this, either the algorithm's correctness, understanding recursion, or understanding the space analysis. If there's any question that you feel you can ask that would be clarifying for you, jump in. I'll set aside a few minutes just to answer questions here.

So I've got a question here. In step five, why do we reject if all fail instead of just one fails? Well, here, so remember what we're trying to do. We're trying to say can I get from u to v in b steps? The way I'm going to be doing that is trying every possible intermediate string w. If I find some w which does not work, that doesn't mean that there's not some other w which might work.

All I need is one w for which I can get from u to w in half the number of steps and w to v in half the number of steps. So I'm going to try every possible w. If any one of them is good, then I can accept. If any one of them succeeds where I can get from u to w in half the steps and to w to v in half the steps, then I know I can get from u to v in the full number of steps. So I only need to find one. If one particular one doesn't work, I'll just go on to the next one.

OK. This is a good question. Do we have to save the word book? So once we succeed in getting from work to able, let's say via book, do we need to save that word book anywhere? No. All we need to remember is that we've succeeded in getting from work to able. We don't need to remember book anymore. We just remember that we've succeeded. And that is by virtue of where we are in the algorithm. If we have succeeded, then we move on to the second recursion, second call, recursive call.

So we found some way to do it. So we found some intermediate point which succeeds for this one. So we move on to that one. We don't have to keep any of that work anymore. All you have to do is remember, yes, I can get from work to able in half the number of steps. Now all that's left is to get from able to play in half the number of steps. It doesn't matter how I got to able in the first place. So we don't have to remember that. That was a good question though.

So I think I understand this. Before we replace the value for book with call with the work involved to find call, yeah, we have to check that we can get from work to book and book to able. So we keep onto book while we're working on the upper half. And only when we've finally succeeded in getting from work to able, let's say via book, then you can throw a book away. But while you're working on the upper half, you try book, you try different strings of length four until one of them works.

I'm not sure. Somebody is asking me about breadth first search and depth for search. I'm not sure I see it. I'm not sure that's going to be a helpful way of thinking about this. So I'm not going to answer that right now. But you can ask that offline later if you want.

Why is the recursion depth log t instead of log m? Well, how high is this thing? Initially it's t high. But every time we're doing a level, we're calling the recursion, we're cutting t in half. I'm solving this in general for b, but we starting off with b equal to t. t is the maximum size. So initially this is going to be t, and then it's going to be t over 2, then t over 4. So it's going to be log t levels before we get down to 1.

Yeah. So somebody is asking, can we think of this as a memory stack? Yes, this is like-- that's the way your typical implementation of recursion is kind of with a stack, where you push when you make a call and you pop when you return from the call.

Is it possible that v can appear during BL procedure on t? Is it possible that v can appear? I'm not sure what that means. It can reappear. So I'm starting with u to v. Is it possible that v might be one of these intermediate strings? Yeah. You're going to try every possible intermediate stream blindly. Including v is one of them.

If you can reach v more quickly, well, great. I guess I have not dealt with the issue of what happens if you get to a-- technically it's going to work out because I'm allowing the difference to be in at most one place. So even if you get there early, you're allowed to not change anything, and that still is a legal step in the ladder.

Yeah. I don't see how to do this from a bottom up perspective. Somebody's is asking is there a bottom up version of this. I don't think so. No, I don't think so.

All right. Why don't we move on? So now we're going to see this proof again. But this time we're going to be proving that you can convert any NFA to a DFA with only a squaring increase. So really, well, let me just put that up there. So this is going to be Savitch's theorem, that among other things proves that PSPACE equals NPSPACE.

So it says that you can convert a non-deterministic machine to a deterministic machine only squaring the amount of space. So you're comfortable with this notation here. Anything that you can do in f of n space non-deterministically you can do in f squared of n based deterministically. And we're going to accomplish that by converting an NTM to a deterministic TM but only squaring the space used. So n is going to convert it to an m.

And now this proof is going to look very similar to the proof in the previous slide. It's the same proof. And the fact from the previous slide about ladder really is implied by this, because we had an easy algorithm to show that the latter problem is solvable in non-deterministic, in NPSPACE. So ladder problem was easily shown to be in here. If you remember, you just basically guess the steps of the ladder.

So non-deterministically, you can easily check, can I get from the start to the end? But Savitch's theorem tells us that anything you can do non-deterministically in polynomial space you can do deterministically in polynomial space. So what we showed in the previous slide follows from this slide, but this slide is really just a generalization of the same proof. Maybe I've said it too many times now.

So we're going to introduce a notation very similar to the notation we had last time. But now we're going to be talking about simulating this non-deterministic machine with a deterministic machine. And we're going to take two configurations of this non-deterministic machine, ci and cj, and say can I get from ci to cj in at most b steps? I'm going to have a notation. Very similar to the notation for the ladder where I can get from this word to that word in at most b steps by a ladder.

Here can I get from this word to that-- can I get from this configuration to that configuration with at most b steps of the Turing machine's operation? So these are two configurations now of n. So can n go from this configuration ci to that other configuration cj but only taking b steps along the way? That's now the computational problem that I'm going to solve for you with an algorithm. And it's going to be a recursion exactly like the previous one.

So n gets its input the two configurations ci and cj and the bound b and want to check can I get from i to j within b? So now the picture is a little different but very similar. So instead of a ladder appearing here, it's really something that's basically a tableau for the machine n where I have an initial configuration and an ending configuration.

This would happen to be the starting point for the whole procedure if you're testing whether n accepts w. But we would be solving this in general for any-- so that case, so I have the start configuration of n on w and the accepting configuration or an accepting configuration. But in general, what I will be solving is starting with any configuration ci and going to any configuration cj. So I want to test can I get from ci to cj within at most b steps.

So first of all, if b is 1, you can just check that directly. And now, again, we're operating deterministically to simulate the non-deterministic machine. So this is the deterministic machine m. So m can easily, if it's given two configurations of n and says, can I get from the first one to the second one in one step? Well, that's an easy check. You just lay out those two configurations, look at the n's transition function, and say is this a legal move for n? Yes or no? And you accept or reject accordingly.

Now, if b is larger than 1, you're going to try all possible intermediate configurations, calling them c mid. This was like the w from the previous theorem. This is all possible strings c-- all possible configurations c mid. And a configuration is just going to be a-- so far so-- OK. This is all possible. Looked like my PowerPoint crashed, but it seems OK.

This is all possible configurations, which is just a string with a string of tape symbols with a state symbol appearing somewhere. That's all it is. Going to try all possible configurations as candidate middle configurations. And say can I get from the upper one to this candidate middle one and from that middle one to the lower one within half the number of steps each time. And solving that problem recursively.

So I got a question here about the possibility of looping forever. First of all, if n is going to be looping, I don't have to worry about it, because I'm starting off, I only need to simulate machines that are deciders. Because I'm trying to show that any language in here has to be accepted, has to be decided by some non-deterministic machine. So I'm not going to worry about machines that are looping. If they're looping, m may misbehave in some way, but that's not going to be a problem for me. So let's keep life simple. Think about the deciders only.

So we're going to recursively test here. So that means I'm going to try every possible middle, see if I can get from the start to the middle and the middle to the end. If both of them work after I test them recursively, then I'm going to accept. If not, I'm going to continue. And I reject if I try them all and none of them have worked. Then I know there's no way for n to get from this configuration to that configuration in b steps.

And the overall picture, I test whether n accepts w, as I mentioned, by starting with ci is the start configuration and cj is the accept configuration. And now how big is t? Because I need to calculate a bound on how deep the recursions are going to be. So t here is going to be the total number of possible configurations. If this is the whole thing, it never repeats a configuration. So this is going to be a bound on how many steps then can be taken. And that's simply we calculated this before. It's the number of states times the number of head positions times the number of tape contents. And this is really going to be the dominant consideration anyway.

And so now each recursive level, and maybe I should have emphasized this at the beginning, how wide is this picture? It's big enough to store a configuration. A configuration is essentially a tape contents. So that's going to be f of n wide. So each recursive level stores one configuration. Now the w costs f of n space to write down.

And the number of levels is going to be the log of the initial height, which is this. So this is going to be the dominating part of it. So the log of this is going to be, again, order f of n. So each one takes f of n space to write down. The depth of the recursion is going to be order f of n. So the total is going to be order f squared of n, and that's how much space this uses. And that's the proof of Savitch's theorem.

So yeah, so this is a good point there. Somebody asks can there be multiple accepting configurations. I should have made the-- I forgot to say this and I was just realizing it as I was explaining it. One of the things I should have-- you can enforce that there is just a single accepting configuration. This is kind of a detail, so don't worry about it if you don't want to. But you can make sure that there's a single accepting configuration by telling the machine when it accepts, it should erase its tape and move its head into the left most cell in the accept configuration. So there's just going to be a single accepting configuration to worry about instead of having to try multiple possibilities, which you could do in this algorithm, but it would just be annoying to have to write that down that way. So we often assume there's just going to be a single accepting configuration for these machines.

How do you know f of n? So that's actually a little bit of a delicate issue. I mean, if you could compute f of n, the bound, which is, for example, if it's going to be a polynomial bound, you can just compute f of n. It's very easy to compute n squared or n cubed. And so you can just compute that and then use that as the size of the registers you're going to be writing down. If you want to prove this in general for f of n, it's a little bit technical to have to deal with it.

And I'm going to have to refer you to the book on that one. The book tells you how solve this for a general f of n. You basically have to try every possible value from one until it works. And I'm afraid that's going to derail us trying to decipher that. So let's not worry about that aspect of it. But you can handle general f of n here. You don't need to put any conditions on f of n.

Can we go over this term here? So we've seen this term once before when we talked about LBAs and seeing that LBAs always-- we can solve the ALBA problem. This is simply the number of different configurations the machine can have. Because the configuration is a state. It's a head position and a contents of the tape. And this is the number of each of those that you can possibly have. Number of states. A head position, the size of the tape of f of n is f of n. So this is that many different head positions. And this is the number-- if d is the size of the tape alphabet, this is the number of tape contents that you can have.

How is seeing if n accepts w with this algorithm convert a non-deterministic Turing machine to some deterministic Turing machine? Well, n is the non-deterministic Turing. So n is we're converting non-deterministic Turing machine n to deterministic Turing machine m. So m is a deterministic simulator of n. That's what this whole m is. So if we can do this for any n, then we've proved our theorem.

Why don't we defer. I think I got through most of the questions here. If there's other things, we can save them for the coffee break, which is coming soon. I think we have one more slide before that.

So I'm going to define PSPACE completeness I think. Yeah. And then I think after that we have the break. So PSPACE completeness is defined and very much inspired similarly to NP completeness. So a problem is PSPACE complete if it's in NPSPACE and every other member of PSPACE is reducible to it in polynomial time. And we'll say a bit about why we choose polynomial time reducibility here.

So here's kind of a picture of how PSPACE-- how complete problems relate to their complexity classes. So you kind of think of a complete problem for a class. It's kind of the hardest problem in that class because you can convert. You can reduce any other problem in that class to the complete problem. So here are the NP complete problems. Sort of the hardest for NP. You have the PSPACE complete problems, kind of the hardest for PSPACE.

If an NP complete problem goes into P, that pulls down all of NP to P. If any PSPACE complete problem goes into P, it pulls down all the PSPACE into P by following the chain of reductions, because any PSPACE problem is reducible to the complete problem. Which in turn if it's in P, then everything goes into P. So if you have a PSPACE complete problem which is in P, then all the PSPACE goes into P.

So why do we use polynomial time reducibility instead of, say, polynomial space reducibility when we define this notion? It's kind of a very reasonable question. But if you think about it, using polynomial space reducibility would be a terrible idea. And we've seen this phenomenon happen before. Every two problems in PSPACE are going to be PSPACE reducible to one another. We haven't even defined that notion yet, but you can imagine what it would be. Because a PSPACE reduction can solve the problem for a problem in PSPACE. And then it can direct its answer anywhere that it likes.

So in general, when we think about reductions, the reduction should be not capable of solving the problems in the class. Because if they could, then every two problems in the class would be reducible to one another. And then all problems in the class would be complete, because everything in the class would be reducible to any one of the other problems. So it would not be an interesting notion. What you want to have happen is that the reductions should be weaker than the power of the class.

And if you look at the reductions that we've defined so far, they're actually very simple. The only thing is they have to make sure that they can make the output big enough. But actually constructing the output, they're very simple transformations. In fact, even polynomial time is more than you typically need. There's even much more limited classes that are capable of doing the reductions, as we'll see. So having powerful reductions is really not in the spirit of what reductions are all about. You want very, very simple transformations to be the reductions. Anyway, I hope that's helpful. So what we're going to be aiming for in the second part of the lecture is showing that TQBF is PSPACE complete.

And let me-- here is the check in. So this is our first check in, coming a little late in the lecture. Suppose we have proven that, as we will, that TQBF is PSPACE complete. What can we conclude if TQBF is actually not necessarily in P, only goes to NP? And this is relevant to a question that's coming in from the chat, but I'll answer that later.

So suppose TQBF ends up being an NP and not in P. What can we conclude? Remember, if TQBF is in P, then PSPACE equals P. Suppose it goes to NP. What happens then? There may be several correct answers here. Check all that apply. All right, so we're near the end of the poll. So let me give you another 10 seconds and then we're going to shut it down. OK, are we all in? Closing it down.

Here are the results. So yes, so first of all, the most reasonable solution, most reasonable answer is b, which I think most of you have gotten. That if a PSPACE complete problem goes down to NP, well, NP is capable of simulating the polynomial time reduction. And so any other problem in PSPACE would then also be in NP. And PSPACE would equal NP. But note if PSPACE equals NP, they're also NP equals coNP, because PSPACE itself is closed under complementation. So that was kind of a little bit extra fact that you could conclude from this as well.

So let's move on then to our coffee break, and we'll pick up the proof that TQBF is PSPACE complete after that.

So was d true or not? D was P equal NP. No, we cannot conclude that P equals NP from PSPACE equal to NP. So if TQBF is in NP, it doesn't tell us anything. For all we know, P equals NP. But from the stuff that we know so far, we cannot conclude that P equals NP.

Oh, and yeah, so you can conclude-- oh, I'm sorry. B and d are both correct here. Let me just shut this thing off. So b and d are correct. So if a PSPACE complete problem goes to NP, then NP equals PSPACE, N equals coNP. So the correct answer, b and d. Sorry. I got myself confused. But c is not something you can conclude or a.

So somebody is asking me a fair question. I say the reduction method should be weaker than the class. But for example, even in the case of PSPACE, PSPACE might be equal to P. And then it wouldn't be weaker than the class if we use polynomial time reductions. But I think maybe I should say apparently weaker. As far as we know, it's weaker. But we believe it to be weaker. It's true if P equals PSPACE, then every problem in P is going to be PSPACE complete. It's just going to be a weird world if P equals PSPACE. Same thing for NP and NP.

So I'm getting a number of questions also about other possible reducibilities that are even weaker than polynomial time reducibility. So we're going to see very soon weaker reduce-- complexity classes within P. So first of all, PSPACE seems to be bigger than P. We're also going to look at log space. But that's going to be actually in Thursday's lecture. These are classes that seem to be inside. Well, they're inside P. We believe they're properly inside P. But we'll see that later.

Let me just see here. So we're almost out of time. Let me put our timer back. In fact, our timer is showing us out of time. So why don't we get going? Let me move this back to-- OK. Continuing.

TQBF is PSPACE complete. So first of all, let's remember TQBF. These are all of the quantified Boolean formulas that are true. So TQBF stands for True Quantified Boolean Formulas. And remember, we saw these examples from the previous lecture that these are two quantified-- these are two QBFs. The first one is true. The second one is false. And it's going to be interesting to think about. Here they're exactly the same except for the order of the quantifiers. And so what's really going on here? I think it's good to understand these expressions. They come up everywhere in mathematics, these quantifiers.

In the upper one, when we say for every x, there exists a y, that y can depend on the choice of x. You choose to make a different x. You're allowed to pick a different y. But the lower expression says there's a universal y. There's one particular y that works for every x. So in a sense, the lower statement is a stronger statement. Whatever you have in the quantifier free part. So the lower one implies the upper one. Happens that the lower one in this case is false and the upper one is true. But in general, when you have this change of quantifiers like this, the lower one would imply the upper one. Anyway, that's sort of a side remark.

So let's get back to the proof that TQBF is PSPACE complete. That's what our goal is. All right. Now, as I mentioned, this is the same proof. You're going to be seeing it for the third time today. But there's a certain amount of-- it's sort of the context changes in each case. So now we want to show that TQBF is PSPACE complete. So it's one of these hardest problems now but for PSPACE, where satisfiability was the hardest problem for NP.

So we want to show that every language in PSPACE is reducible to TQBF. And so we're going to give polynomial time reductions that map some particular problem a, which can be done in space n to the k. It's a problem solvable in PSPACE. We're going to show how f maps a to TQBF. We have to construct the f. So f is going to be a mapping that map strings which may or may not be in a. So strings w, which may or may not be in a, to these formulas, these quantified formulas.

So w is going to get mapped to some formula phi sub mw. It had exactly the same even symbols we used when the proof of the Cook-Levin theorem about SAT being NP complete. This is a very similar proof. But you'll see that we have to do something more in order to make it work in this case. So w is in a if and only if this formula is going to be in TQBF. In other words, if and only if this formula is true.

So this formula is going to kind of express the fact that m accepts w, which means that w is in a, because m is the machine, is the PSPACE machine for a. So this formula says m accepts w, and it achieves that by building in a simulation of m on w. It kind of describes a simulation for m on w which ends up accepting. And if m does not accept w, that description is going to inevitably be false.

So let's just see what that's going to look like. So we're going to use the same idea that we used for the Cook-Levin theorem that SAT is NP complete. This notion of a tableau. So if you remember, it was basically a table which was just simply a way of writing down a computation history for m on w. So the rows are the steps of the machine. The top row is the start configuration. The bottom row is, let's say, some particular accept configuration such as I just described where the machine clears its tapes and moves its head to the left end. So there's only one accepting configuration we have to worry about. And each of the rows here is a configuration of m.

Because m runs in space into the k, the tableau kind of similarly to what I described before has width n to the k. So now we're talking about polynomial time machines. So the f, which is the bound, the space bound, is going to be some polynomial n to the k. So the width of this tableau, the size of these configurations are going to be n to the k. How high is this tableau going to be? Well, that's going to be limited by the possible running time of the machine, which is similar to what we saw before. It's going to be exponential in the space bound. So it's going to be d to the n to the k, where d is essentially the tape alphabet of the machine.

So are we all together on this? This is very similar to the proof of SAT is NP complete. The key difference there was m was non-deterministic, which might be something to think about later. But let's not focus on that right now. This m is deterministic. But the important difference was the shape of the tableau. The size of the tableau was very different. In the case of SAT is NP complete, we started off with a polynomial time non-deterministic machine. So it only could run for a polynomial number of steps. Here is a polynomial space machine, which can run for an exponential number of steps. That's going to be an important difference here. So let's see why.

The reduction has to construct this formula phi sub of mw, which basically says that this tableau exists. Now, we already saw how to do that when we proved the Cook-Levin theorem that SAT is NP complete. Remember, we hac all of those. We had variables for each cell that told us what the contents of that cell was. And then we had a lot of logic here. We had a bunch of logic that said that all those neighborhoods were correct, which basically says that the tableau corresponds to a correct computation of the machine.

So why don't we just do the same thing? Why don't we just build our formula using exactly the same process that we used to build the formula when we had SAT being NP complete? Something goes wrong. We can't quite do that. The problem is that if you remember the formula that we built before was really about as big as the tableau is. Because it had some logic for each one of the cells. It had a set of some of the variables for each one of the cells and it had some logic for each of those neighborhoods, basically. So it says that each of the cells does the right thing.

So it was a pretty big formula, but it was still polynomial. The problem is that tableau is now n to the k by d to the n to the k. That's an exponentially big object. So if your formula is going to be as big as the tableau, there's no way you can hope to produce that formula in polynomial time. And that's the problem. The formula is going to be too big. Remember, we're trying to get a polynomial time reduction from this language a. So we have an input to a, a string that might be an a, which is simulating the machine. And the size of the tableau relative to w is going to be something enormous. And so the formula is as big as the tableau. There's no way to produce that formula in polynomial time. So this is not going to work.

Let's try again. So now we have here-- now, so remember this notation from ci to cj in b steps. So we're going to give a general way of constructing formulas which express this fact that I can get from configuration i of m to configuration j of m in b steps. Whatever that b is. b is going to be some bound. And I want to know can I get from this configuration to that configuration. And I want to write that down as a formula, which is going to express that fact. And it'll be either true or false.

And I'm going to give you a recursive construction for this formula. So I'm going to build that formula for a value b out of formulas for smaller values of b. So this is going to be a way of constructing that formula in terms of other formulas that I'm going to build. And there's going to be a basis for the recursion when b equals 1. So that's the big picture. So let's see how does this formula look. So let's not worry about the case for b equal 1 right now. This is the case for larger values of b.

So the fact that I can get from ci to cj within b steps. I'm going to write this down in this way. And let's try to unpack that and see what it's saying. Without worrying about how are we going to carry this out, let's just try to understand at a high level of semantics of this thing what it's trying to say to you. It's going to say, well, I can get from ci to cj in b steps. So m can get from ci to cj in b steps.

If there is some other configuration c mid, some other configuration, I'm calling it c mid, very much inspired by the previous proof of Savitch's theorem, where there was c mid was that intermediate configuration. So now instead of trying them all, I'm saying does there exist one where I can get from ci to c mid in half the number of steps and from c mid to cj in half the number of steps?

So if I can build these two formulas, then I can combine them with this sort of extra stuff out here, and them together, and put an exist quantifier that says, does there exist some configuration, some way to find a configuration such that it works as these formulas require? If I can do that, then I'm going to be good. Because then I can-- well, good. At least I'll be good in terms of making something that is going to work.

So first of all, let's understand what I mean by writing down does there exist c mid. It's really if you're thinking back to the way we did the Cook-Levin theorem, we represented these configurations by variables which were indicator variables for each of the cells. And we're going to do exactly the same thing. So we're going to have a bunch of Boolean variables which are going to represent some configuration. So more formally, or in more detail, what this does, there exists a c mid, really is an assignment to all of those variables that represent the contents of the cells of that configuration.

OK, so now let's see how to-- how will the recursion work? So to get this value here, I'm going to do the recursion further. So does there exist a c mid? And now for getting from ci to this c mid, is there some other c mid? This is like another value of w from the previous slide where I'm getting, again, I'm cutting the number of steps in half again. So going from b over 2 steps to b over 4 steps. And I'm going to do the same thing over here.

So I'm just unraveling the construction of this formula in terms of building-- by building it up recursively. And then I'm just going to keep doing that until I get down to the case where b equals 1. And if I'm now trying to make a formula that's going to be, say I can get from ci to cj in just a single step. So this is really talking about a tableau of height 1 or height 2. Then I can just directly write that down the way I-- now the tableau is not very big. So now I can write that down using the neighborhoods and so on that I did in the Cook-Levin theorem proof. And this is how you put it all together.

And now if you want to talk about the getting from does m accept w. So I initially say, can I get from the start configuration to the accept configuration in t steps, which is the maximum running time of the machine? So again. And if you followed me, what happened in Savitch's theorem, it's the same values.

Now, the thing is we have to understand how big this formula is. And if you think it through, there's a problem. Because what's going on here? I'm expressing this formula in terms of formulas where the size of b is cut in half. But now there are two of them. So it's two formulas on half the value of b. That's not going to be a recursion that's going to work in our favor. So let's just see what happens.

So each recursive level doubles the number of formulas. Here we have two formulas. Here we're going to have four formulas and so on. So the number of formulas doubles each time. So the length of the thing that we're writing down is doubling in size each time we go down the recursion. That's going to be OK if we don't go too many levels. But unfortunately, we are going quite a few levels, because the number of levels is going to be log o this initial exponential size thing. So it's going to be n to the k levels. And so if you're doubling something n to the k times, you're going to end up with an exponentially sized formula. And again, we failed.

OK, so I have a check in on this. But maybe we should spend a couple of minutes just trying to understand what's going on here. Because the next slide is really my last slide, and it's going to fix this problem. But let's make sure we understand what the problem is before we try to fix it.

Why can we no longer write over each layer of the recursion as we did in ladder? Oh, that's kind of a cool question. What does it even mean to write over the different-- well, so that's kind of an interesting question here. So in a sense, that's going to be the solution. We're going to reuse things in a certain way. But I want you to understand that this method itself does not work, because this recursion here where I'm writing the formula, I'm building the formula for b out of formulas for smaller values of b. If I do it that way, I'm going to end up-- if I do it as it's described in this procedure here, I'm going to end up with an exponentially big formula. And that's not good enough.

So if you cut the formula in half each time, even though you have two formulas, won't the length be the same? I'm not cutting the formula in half. I'm cutting the value of b in half. So you have to say b is initially an exponentially big value. So we're going to end up with an exponentially big formula. So it's not really cutting the formula in half. Cutting the b in half. b starts out big. I mean, b is initially this value here, d to the n to the k.

I'm worried. Not too many questions here. I have a feeling that's probably not a good sign. Well, I mean, if you're hopelessly confused, maybe I can't fix it quickly. So anyway, why don't we move on and see how to repair this, how to fix this problem. And that is going to be by a trick. In fact, I know the people who were involved with coming up with this. This was actually, this proof was done originally at MIT many years ago in the 1970s. And the folks who were involved with it called it the abbreviation trick. So that's what we're going to do on the next slide.

Oh no, there's a check in first. Why shouldn't we be surprised that this construction fails? A, well, we can't-- just the notion of defining a quantified Boolean formula by using recursion is just not allowed. So you can't define formulas that way. Doesn't use the for all quantifier anywhere. Or because we know that TQBF is not in P. You can see we do-- what do you think? Why should we not be surprised?

I guess I could have put a d in there. Not surprised because you don't know what's going on. That's another reason not to be surprised. But anyway, hopefully you have some glimmer of what's happening here. And why don't we just-- almost finished. So I'm going to shut this down in a second. Last call. OK, ending.

All right. So in fact, the right answer is b. I mean, one should be suspicious that if there's no for alls appearing in this construction anywhere. So really what we're doing is we're constructing a formula that has only exist qualifiers. So it's a satisfiability problem. So really what we just did was we constructed-- we did in a more complicated way the Cook-Levin construction, because we end up with a SAT formula only with exist quantifiers. And so really try one and try two were the same. So it's not surprising that you end up with an exponentially big formula as a result. I don't know.

A lot of you answered we know that TQBF is in P. It's not in P. We don't know that. I don't know where you-- what's happening with you guys? But no. Maybe that was a protest vote. But anyway, we don't know that. And what has that got to do with anything anyway? So anyway, let's see. We solve this in our remaining few minutes here.

So here is the solution. Remember, this part where we're saying we're trying to find c mid, does there exist a c mid such that I can get from ci to c mid in half the number of steps and c mid to cj in half the number of steps. I'm expressing one formula in terms of two formulas. That's where the blow up is occurring. Because these two are then in terms are going to each-- so these two are going to become 4, become 8, and that's not good.

Can I express this fact in terms of just one formula? And this kind of a little bit in the spirit of your suggestions. Can we kind of reuse things in a way? And that's what we're actually going to kind do. So here's another way of saying the same thing but with just a single formula. And it uses a for all. And the idea behind it is that an and is kind of like a for all. Or a for all is kind of like an and. Just like in exist as kind of like an or. So when you're saying does something exist, is it this thing or that thing or that thing or that thing? And when you're saying for all, it has this thing and that thing and that thing and that thing. Ands and for alls are very much related.

And so we're going to convert this and into a for all. We're going to say for all configuration cg and ch that are either set to ci c mid or to c mid cj, the formula cg ch-- I can get from cg to ch in half the number of steps. So you have to think about what this is meaning here. And I also want to make sure that you don't feel I'm cheating you, because well, first of all, so now we have just a single formula. We're going to go down to the case b equals 1, as we did before. You have to make sure that saying this restricted for all is not a cheat. When you have for all x is in s, like we have over here, is equivalent to saying for all x if x happens to be in s, then the other thing follows. And this implication can be expressed using ands and ors.

And as before, the initial starting point is going to be going from c start to c accept in t steps. And so the analysis that we get is that because there's no longer a blow up, each recursive level just adds this stuff here in front. The exist c mid and this for all part. So that's going to be adding order n to the formula instead of multiplying because we have two formulas. And now the total number of levels is order n to the k, as before. So the size is going to be n to the k times n to the k order n to the 2k.

I actually I had a brief check in here, which I'd like you to do just in our remaining few seconds. Does this construction depend on m being deterministic? So let me just launch that. I want you to guys to get your check in credit here. But in fact, you have to think this through. That formula says that the tableau, you get a tableau. Is that going to matter depending upon whether it's deterministic or non-deterministic? It's actually, well, it's kind of running 50/50 here.

Why don't you just pick something? Because I'd like to just close this out and just get to our last slide. So if you don't follow, don't worry. But it's actually an interesting point that the fact that this-- so I'm going to end this. All in?

So in fact, the right answer is it still would work if it's non-deterministic. And this would give you an alternative way of proving Savitch's theorem. So really this all comes down to this proof, which implies Savitch's theorem and then in turn implies the ladder DFA problem. So anyway, that's side note, not critical for understanding, really. You can take those as all separate, the results, and that's good enough.

All right, so coming back. Whoops. Coming back. This is what we did. And so each recursive level, the size of the QBF is not the same. Somebody's asking is it the same at each recursive level. No, we had to add in-- let's just see. Each recursion. This is recursively built here, but now we have to add this part in front and this part here in front.

So the quantifier which is quantified over a bunch of variables representing the configuration, that gets added on at each level. So it's not just it stays the same. There's stuff that's get added in. But what's important is that it doesn't blow up exponentially. The stuff gets added in every time but not multiplied.

OK. So we're done here. So anybody, you can all take off. I think many of you already have. Bye bye. Thank you.