[SQUEAKING]

[RUSTLING]

[CLICKING]

PROFESSOR: So we've been talking about P and NP and time complexity classes. And today we're going to shift gear. We're going to talk about space complexity or memory complexity, as space complexity is what complexity theorists usually refer to it as.

And time and space are the two basic most basic measures of complexity that we consider. And so today we're going to look at the second of those two, the space complexity. A lot of this is going to be by analogy with what we did for time complexity.

We're going to define complexity classes. We'll talk about polynomial space and nondeterministic polynomial space, see how those classes connect up with the time complexity classes that we've already defined. And we'll do some examples that will be setting us up for our further discussion about space complexity next week.

So we're going to talk about, first of all, what it means for a Turing machine to run in a certain amount of space. And that's simply going to be counting the number of cells that the Turing machine scans over on its tape during the course of its computation. It might be reading that cell. It might be writing on that cell.

But the total number of cells that it actually visits-- of course, visiting the same cell multiple times only counts once because space can be reused. But we're going to count the number of cells that the Turing machine visits during the course of its computation, and then define the space utilization by analogy with what we did for time. So we'll say a Turing machine runs in a certain amount of space-- f of n, we'll say.

First of all, it has to always hold, so all of the machines are deciders. And it uses at most that much tape-- it visits that number of cells-- on all inputs of length n. So just like we said for time complexity, the machine has to run within t of n time on all inputs of length n, here it's going to have to use at most f of n cells on all inputs of length n in order for it to be running in space f of n.

A tape cell is simply a little square of the tape where you can write a symbol-- answering a question, that good question that came in from the chat. So I'm not sure we I have the diagram for that. And each of the little squares on the tape are going to be the tape cells.

Generally, we're going to be sticking to one-tape Turing machines. But I'll make a brief remark about multi-tape Turing machines shortly. Better? Tape cells, sorry, on all inputs of length n. So that's for deterministic Turing machines.

For nondeterministic Turing machines, we will say that it also runs in a certain amount of space. So for a nondeterministic machine, it has to use at most that many tape cells on each branch of its computation separately.

You don't add up the total number of cells used across all of the branches, just like we don't add up the total amount of time the machine uses across all of its branches. For the machine to be running in, say, space n squared, it has to be using at most n squared cells or order n squared cells on each one of its nondeterministic branches separately. There might be exponentially many branches, but that's OK. But on each branch, it's going to be using at most n squared or order n squared cells.

Importantly, though, that still the machine has to be a decider. It's not enough to be looping forever and using a small amount of space. It could do that, but that's not going to count toward the machine contributing to its space complexity of that language.

So for the machine to be running in a certain amount of space, we say that the machine holds on all of its branches. And each one of its branches uses at most that much space. I can see lots of typos here. Thank you. Why did I mess this all up today? So nondeterministic-- good, thank you.

All right, so we're going to define the space complexity classes analogous to this time complexity classes. So these are languages that you can do with machines that run within that space bound. So space f of n-- you can think of space n squared-- is all of the languages that a deterministic one-tape Turing machine can do within-- can decide within-- by using at most n squared tape cells, order n squared tape cells. Similarly, the nondeterministic space complexity class are all of the languages that a nondeterministic one-tape Turing machine can decide, running within that amount of space.

And lastly, we have polynomial space. So that's the union over all polynomial space bounds of the space complexity class, and nondeterministic polynomial space-- the same for all of the nondeterministic polynomial space classes. So I think I do have a check-in on this which talks about multi-tape Turing machines.

So we can define space complexity for multi-tape Turing machines, just as we did for one-tape Turing machines, and then define the associated space complexity classes, and then define the class P space. But that would be for multi-tape Turing machines.

Now for time, remember that the class P that you would get for multi-tape Turing machines is exactly the same as the class P that we got for one-tape Turing machines. That was part of the nice quality of the class P. It's robust in that sense, and natural.

So how about for PSPACE? What do you think? Do we get the same class? No? Maybe? Or yes, because we can convert a multi-tape Turing machine to a single-tape Turing machine, by only squaring the amount of space? That was what happened with time, as you remember.

Or maybe we can do even better. Converting a multi-tape Turing machine to single-tape only increases it by less-- by, say, a constant factor. Here, remember, this is how we're defining space complexity for multi-tape Turing machines. We're taking the sum of all the cells used on all of the tapes.

All right, so let's launch that poll and see what you think. Hopefully, this is not too hard. Yeah, I think most of you have got the idea, though. Some of you are-- I worry sometimes about some of the answers that I get. I don't know if you're serious, or you're really badly confused. But anyway, let's wrap this up-- another 10 seconds or so. Last call. I'm going to end it.

Yeah, I mean, I think answer B is a reasonable answer. In fact, answer C is the correct answer. If you just look at the same simulation from multi-tape to single-tape and how much space overhead that simulation introduces, it's only linear. You're basically just taking all of the tapes of the multi-tape machine and writing them down next to one another, obviously ignoring all the infinitely many blanks.

We're just taking the active portion of the tapes, writing them down next to each other. So the total amount used is just going to be the sum on the single-tape of what was used on each of the individual multi-tapes in the original machine. So there is just a linear cost overhead by converting from multi-tape to single-tape when you're looking at space, the amount of memory that's used.

For time, remember there was some additional overhead because we had to be updating where the virtual heads were. And that cost extra time to move our single head around to do that. But for space, the amount of time that's introduced is irrelevant.

We're only looking at the amount of memory. And so that's a-- the overhead on that is very low. I do worry about the folks who are answering A, for example, for this question. You should be rethinking what's really going on here.

Now let us move on here from that to our next slide and compare the time and space complexity classes. Time and space complexity-- how do they relate to one another? And so first of all, we're going to point out-- let's start out-- here, t of n is going to be representing some bound, either on the time or the amount of space. And generally, at least up until this point, and mostly going forward, though there's going to be one variation on that a little later, but we're going to be focusing on bounds which are at least big enough to either read the input or at least hold the input. So that's why we refer to t of n being at least n.

So now if we look at the time complexity class t of n-- think of-- t of n typically would be, say, n squared, maybe. And the things that you can do in n squared time I claim you can also do in n squared space. And basically, it's just using the very same machine.

Suppose you have a machine that runs in n squared time. How could it possibly use, say, n squared-- n cubed-- space, if it's running only in n squared time? Even if it tries to use as much tape as it possibly could, as many tape cells as it possibly could, and sending its head cruising out into the blank portion of the tape, chewing up as much-- as many tape cells as it possibly can-- in n squared time, it's only going to be able to use n squared space.

So the very same machine that runs in t of n time is also going to run in t of n space. So this containment here follows really without doing any work at all. So just restating that here, a Turing machine that runs in t of n steps cannot use more than t of n tape cells. So right now, we're focusing on-- we could prove some analogous statements about nondeterministic complexity, but let's focus here on the deterministic complexity.

Now let's look at going the other direction. Suppose we have a Turing machine that uses t of n space now. Does that immediately imply it's using only t of n time? And that's not so clear and, in fact, probably not true, because space appears to be much more powerful than time. And within a certain amount of space, you can run for much longer than that same amount of time.

So how long could you run? So what you can show is that if you're running within a certain amount of space-- t of n space, let's say-- n squared space, for example. The amount of time you could use is going to be exponential in n squared, a 2 to the order n squared.

Sometimes we also write that as the union of c to the n squared by pulling down that constant here. Well, it's also just to understand what we mean by order t of n up in the exponent. It means the union over c to the t of n for all c. Either of these are just completely equivalent, so whichever one you're more comfortable with.

But why is this going to be true? Why does a Turing machine that runs in, say, n squared space, use at most 2 to the order n squared time? And that's because if you look at how many possible configurations the machine can have, remember that a configuration is essentially the contents of the tape.

There's also the head position and the state. But the dominant aspect of a configuration is the tape. And so how many different tape contents can you have? Well, it's going to be exponential in the length of that tape because each cell can have some fixed number of symbols in it.

If a machine repeats a configuration, it's going to go forever, which we're forbidding in these machines, because they're all going to be deciders. So they can only run for an amount of time which is bounded by the number of configurations that the machine can have. And so the machine can have-- if it's running in t of n space, then the amount of time that it could be running is going to be at most some constant to the t of n or 2 to the order of t of n, saying the same thing, unless it's going to repeat a configuration and end up looping.

So these are the two fundamental connections between time and space. Time is contained within the same amount of space. Space is contained within that amount of time exponentiated. So one corollary of that is that the class P is contained within PSPACE. Similarly, NP is going to be contained within NPSPACE for the very same reason.

Is this understandable? This is a good place or a moment to add one more line to tell you about. But leading into the next slide, we're-- so if you understand the definitions of what we've done so far, all of-- this is a fairly straightforward theorem, and the corollary is immediate. So anything that you can do in n squared time you can do in n squared in space. And so anything you can do in polynomial time you can also do in polynomial space.

Yeah, c-- somebody's asking me, what is the c? c is essentially going to be the size of the tape alphabet because that's going to govern how many different configurations you have. There's a slight extra factor for the tape-- the head location and also the state. But the main thing is going to be the number of tape symbols and the length of the tape. But what's going to come next is we're going to prove something more powerful than this corollary that P is contained in PSPACE, because not only is P contained in PSPACE, but NP is also contained in PSPACE. And for that, we're going to have to do more work.

So somebody's asking me about the number of states. The number of states is going to be-- is fixed depending on the machine only. So it doesn't depend upon n. So it could at most affect the number of configurations by a constant factor. And those constant factors are going to be absorbed within the definitions of these complexity classes because that's how we define them to be ignoring the constant factors.

Why don't we just take a-- this may be a good place to pause for a second and see if there's any questions, because I think for some of you, this may be straightforward. But I think it's less common to be measuring-- thinking about the amount of memory as a complexity measure. So this is perhaps a little less familiar.

Some of you have seen measuring time in other classes, but measuring the amount of space that the algorithm uses probably is a little less familiar. And maybe it's worth spending a moment or two answering questions about that. So I'm not sure I understand the question that just came in, but I'll read it out there.

Is it possible that a Turing machine can loop forever? Absolutely. But a Turing machine that loops forever does not count as one that runs within the space bound. To run within the space bound, the machine must halt on every input. It has to be a decider. We're only considering deciders here.

So is it possible a Turing machine could loop forever? Yes. Isn't the Turing machine we're talking about a member of space, and thus a decider? Not totally sure I understand the question. But if a Turing machine is not halting on all inputs, it's not a decider. That's our definition.

Are we good? We're not getting very many questions here. So I'm assuming you're all with me or so lost you don't even know what to ask, which is not good. So be bold. If you're confused, throw a question out there, because I don't want to race through this lecture, since it's maybe a little less familiar to some of you.

So let's move on. As promised, I'm going to show you now that NP, not only P, as happens immediately, but NP is contained as a subset of a PSPACE. So that is-- I did get a question. I moved on before I answered this question. Can I explain part two of the proof again? Part two? OK, let's just do it.

If something runs in a certain amount of space, you have to just think about how many different configurations the machine can have within that amount of space. Remember the configurations that we defined way back at LBAs? So the number of configurations of the machine can have depends on how much space it's allocated.

Like the LBAs-- they had a fixed number of configurations, and we gave a calculation for that, which is basically an exponential in the amount of space. That's how many configurations the machine can have. So if the machine is not looping, if it's a decider, it can never repeat a configuration. And that's going to tell us how long the machine can possibly run for. It's important to understand. I'm not sure if I knew how to say that in any way that different from what I said before.

OK, getting back now to proving that NP is a subset of PSPACE. So now we're going to have to do something that's in a way different from what we did on the previous slide, because now it's not going to be enough to work with the same machine. Before when we were converting, we're showing that a certain amount of time, time class, is contained within a space class by virtue of the very same machine-- by just showing that if it's running within a certain amount of time, then it has to be running within that same amount of space. Or in terms of the space, it was in the-- given a certain amount of space, it has to be running that same machine within a certain amount of time.

Here we're going to-- mixing non-determinism and determinism. So we're going to have to take a machine that's an NP type machine, a nondeterministic polynomial time machine, and convert it into a deterministic machine that doesn't use a whole lot of space. So there's a difference in the character of this theorem because we have to introduce a new machine.

And the way we're going to prove that, I'm going to take advantage of some of the things we've already shown to prove this. One could also prove it a little bit more directly. And maybe it's worth making sure you understand both proofs.

So the first thing I'm going to observe is that SAT, our NP-complete language, the satisfiability language, itself is a member of PSPACE. And the reason for that is when you're given a formula, and now you want to test if that formula is satisfiable, one way to do it, the most obvious way to do it, is try all assignments one by one and see if any of them satisfy the formula. Now that's going to take a lot of time.

But how much space does it use? I have in mind reusing the space every time we try the next assignment. Think of going through all of the assignments the way an odometer would work-- just trying every possible assignment, but reusing the space where you're going to write that assignment down, incrementing it like a number written in binary, if you wish.

Going through all the possible assignments, every time you get in the next assignment, you plug it into the formula and see if the formula's satisfied. If it is, then you can accept immediately. If not, you go on to the next assignment. And only when you've gone through all the assignments in that way, and none of them have satisfied the formula, then you can reject.

So how much space does that use? That doesn't use a whole lot of space because you're reusing the space to write down one assignment after the next. It's only going to be using an amount of space which is big enough to hold an assignment, which is basically linear, because it's the size of the number of variables of the formula. So that's going to be a linear amount of space to solve the satisfiability problem. And so the satisfiability problem is certainly in PSPACE-- step one.

Step two is we're going to take advantage of what we know about reducibility. So if A is polynomial time reducible to B, we've already commented-- we didn't say this exactly in this way. But it's still going to follow that anything you can do in a certain amount of time, you can also do in that amount of space, because it's the very same machine-- can't use any more space than the amount of time it was allocated.

So if A is polynomial time reducible to B, it's also going to be reducible in polynomial space. A polynomial space machine could do the reduction. So that means if A is polynomial time reducible to B, and B is in polynomial space, then A is also in polynomial space.

But we know, because satisfiability is NP-complete, that every language of NP is reducible to SAT. So put SAT in place of B. Every NP language is polynomial time-reducible to SAT. And we now know that SAT is in PSPACE. So therefore, every language in NP is in PSPACE because they're all polynomial time reducible to SAT.

So just by using some of the technology we've developed, namely the notion of completeness shows some of its power, that if you want to conclude something about an entire class, an entire complexity class, if you have a complete problem for that complexity class, often it's enough just to work with the complete problem. And then everything else by virtue of the reducibility, is going to inherit the same property. Doesn't work in all cases, but in many of the cases, as long as the reducibility can be computed by the type of procedure you're working with, then you can-- then it follows.

You could also prove this more directly. I think it's in some ways a little clumsy or a little bit less elegant. But you can say, well, let me just take my-- take a language that's in NP. It has a nondeterministic polynomial time algorithm. And then give a deterministic polynomial space algorithm-- simulates that NP algorithm just by going through all the different branches, but making sure that going through all those different branches, you're reusing the space and not using new space every time you're going through a different branch.

And you can arrange things if you're just a little bit careful to do it that way. So you could give a direct simulation in polynomial space of any NP Turing machine. So I mean, that's also completely satisfactory. But I think this is a little more elegant.

This furthermore is going to allow us to conclude some additional languages are in PSPACE. Let's define a class we have not yet seen, though maybe you've seen this-- I think we've talked about this notion of co before. I think we talked about co-Turing-recognizable. Those are the class of languages whose complements are Turing-recognizable, and the same for co and P.

This is the class of languages whose complements are in NP. If you take the complement of every language that's in NP, and now you got all the languages that are in this class coNP. It's complement of NP. So for example, the complement of the HAMPATH problem. So all the graphs which don't have Hamiltonian paths from SAT, so the non-Hamiltonian graph-- the non-Hamiltonian path problem-- that's a coNP problem.

Or here's a language we haven't-- I'm not going to define as in terms of its complement-- the tautology problem. These are the formulas where all assignments satisfy the formula. All assignments make the formula true. So a tautology is a statement that's always true, no matter how you plug in the variables.

So the tautology language is in coNP because its complement, which is the non-tautologies-- those are the formulas for which there's some assignment which makes it false. So that's going to be clearly an NP language. So tautology is a coNP language.

Now one thing that we get immediately from the theorem as a corollary-- really should write this as a corollary-- is that coNP is also a subset of PSPACE And the reason for that is-- and this is something that-- it's, again, easy, but make sure you understand it-- is that PSPACE itself is closed under complement because it is defined in terms of deterministic machines. And deterministic machines-- you can always flip the answer and get a machine of the same type which will decide the complementary language. So for deterministic machines-- deterministic deciders, I should say-- you can always flip the answer.

Now, so here, we have anything that's in PSPACE. It has a deterministic polynomial time-- polynomial space-- machine. And so its complementary language is also going to be in PSPACE. So PSPACE and coPSPACE are equal. And so that's why coNP is going to be in PSPACE-- going to be a subset of PSPACE.

I hope that's not getting mixed up by all of the different alphabet soup here. But here is maybe a picture maybe that'll be helpful of how the world looks for the time and space complexity classes so far. So we have P is a subset of NP. It's also a subset of coNP, again, for the same reason that P and coP are equal. We never even really talk about coP because it's the same as P.

But NP and coNP-- those are two classes where we don't know whether they're equal or not, because an NP machine-- you can't necessarily complement the behavior of an NP machine and end up with an NP machine. So a question here-- how do we know that coNP is a complete class of problems? I didn't say that there's anything about completeness.

And coNP is just a collection of languages. I'm not saying it's any-- any particular feature about it. In fact, it does have a complete problem, just like NP has a complete problem. The complements-- and I'm not going to prove this right here, though it's pretty straightforward. Complements of all the NP-complete languages are going to be coNP-complete languages.

I will answer some of the questions about possible alternate worlds. This is how we believe the world looks like, with each one of these regions being separated from one another, including this little corner of the world here, NP, and intersect coNP, which is not-- there might be languages in here which are not in P. And we actually believe there are such languages. But again, all of this is conjectural.

And even whether P and PSPACE are the same or different is an open question. We don't even know the answer to that, which is perhaps even more shocking that we don't know how to solve P and-- prove P different from NP, that we don't know how to prove P different from PSPACE, which seems to be a much bigger class. It would be incredible that anything you can do with a polynomial amount of space you can also do with a polynomial amount of time. But don't know how to prove that. They're different.

And in fact, so this is how the world could look. Everything could collapse down. P could equal PSPACE. And then all of these classes would be the same. And I should also mention-- I don't have this as another diagram here, but just to answer-- there's other possibilities.

For example, P could equal NP without being equal to PSPACE. And then you'd have a different-looking Venn diagram here, where there'd be just two classes. P, NP and coNP would all be the same. PSPACE would be different. That's possible. At least we have no idea.

A lot of these things can collapse in various ways. And you just have to make sure that-- there are some collapses that obviously could not occur, like P-- if P equals NP, it's also going to equal coNP. So you can't get-- there are obviously some crazy collapses which could not happen-- that P collapsing-- P and NP being the same, but different from coNP. That can't happen. But avoiding some obvious contradictory situations, everything else is possible.

So somebody said-- so here's a question. Let me just answer a few of these. Did we use the completeness of coNP to show that coNP is a subset of coPSPACE? No, we didn't do it that way. We showed that coNP-- well, let's see. Did we? Is that fair?

Well, I suppose. NP of subset of PSPACE immediately implies, because you're complementing both sides, that coNP is a subset of coPSPACE. So you don't have to deal with the complete problems on the other side. That's too complicated to get into here. You don't need to talk about coNP-complete problems. Though, again, those are very simple to get from NP-complete problems. Let's see what else is here.

Are there NP-complete problems that are in coNP? So the answer to that is, no, not as far as-- well, I mean, there would be-- if there was an NP-complete problem in coNP, then all of NP would be in coNP, and they would be equal. So we suspect the NP-complete problems are not in coNP, but don't know how to prove that.

So why is tautology in coNP? So here is-- tautology sits in this class, here. The reason is that its complementary language is in NP. The complement of tautology are the languages where there is some assignment which makes the formula false. So with an NP machine, you can just guess that assignment and check that it makes the formula false. So the complement of tautology is an NP language. And so tautology is a coNP language.

So somebody's asking about PSPACE and NPSPACE. And how do those relate to one another? So that's looking ahead to what we're going to be doing next week, but I'll give you a preview.

An old, but at the time surprising, theorem was that PSPACE and NPSPACE actually are equal. So they had this analogy where time breaks down. So polynomial space and non-deterministic polynomial space do turn out to be equal.

The most obvious way of proving-- of trying to simulate an an NPSPACE machine would be give you an exponential deterministic space algorithm. So we'll go through that. But there is an algorithm which collapses non-deterministic polynomial space down to deterministic polynomial space, which, again, at the time was kind of surprising.

Last question I'll take-- is there some equivalent concept to the idea of a certificate for coNP? Yes, there is a notion of a certificate. But now it's going to be a certificate that you're not in the language instead of a certificate that you're in the language, and then, again, work for the very same reason that we have certificates for NP languages, where you had certificate for membership. For coNP, you have certificate for non-membership. There's no other certificate for membership in coNP. So let's move on.

So now we're going to introduce-- we're going to look at some important examples. These are examples that we're going to-- I'm going to give you two examples, first, one called TQBF. And then we're going to have a second example. Both of those we're going to-- one of them is going to be an example of a problem in PSPACE.

Then the other one is going to be an example of a problem in NPSPACE. And these are going to be important languages for us. So they're not just going to serve as examples for today, but they're going to be useful languages for us later on. So just keep that in mind as we're going through it.

So to understand TQBF, you have to understand what are called quantified Boolean formulas or QBFs. So those are Boolean formulas, just like the ones we've been seeing-- we've been talking about-- with Boolean variables and the ands/ors and negated variables. But now you're going to add qualifiers, the "exists" quantifiers and "for all" quantifiers.

If you haven't seen quantifiers, you need to go back and review those. I think that we already introduced-- talked about them briefly earlier in the term. But this is part of the basic math that you need to know. Maybe you'll-- not comfortable with them, you'll pick it up somewhat during the course of today's and the next few lectures, but anyway.

So if you have a Boolean formula-- I'll give you some examples-- that has "exists" and "for all" quantifiers, the requirement for it to be a QBF is that all of the variables have to be within the scope of one of the quantifiers. So all of the variables of the formula have to be quantified by one of the quantifiers. And we're going to assume the quantifiers are in front-- are leading quantifiers in front of the rest of the expression.

So because all of the variables have been quantified, then a quantified Boolean formula is going to be either true or false following the meaning of the quantifiers. And again, some of this may become clearer as we do some examples. So here are some examples coming.

So here is one-- here is a QBF. So all of the variables, which are just x and y-- they both appear in front of-- next to some quantifier. So that's going to be-- that's a requirement if we have a QBF. And so this says, for all x, there exists a y. This expression holds.

So we need to unpack that and understand what it means. It says, for every x-- for every way of assigning a Boolean value to x, so x is going to be either true or false, there exists a way of assigning a Boolean value for y to make this true-- to make the rest of the expression hold true. And we'll go through that.

But let's contrast that with the second example, where I invert the order of the quantifiers, because that's going to be important for the meaning of the formula. So if I say for every x, there is a y, which makes the rest of it true, that says, well, no matter how I set x, there's going to be way to set y to make this true. So that says, well, if I set x to true, [AUDIO OUT] got to be some way to set y to make the remaining expression hold. So if I set x to true, what should I set y to be?

Well, if I set x to be true, maybe I could set y to be true. Well, then this clause is satisfied, but this clause won't be satisfied. So setting y to be true is not-- it won't work. But for every x, I only need to show there exists some y. So if I take x to be true, I can set y to be false.

And now this one is-- this one holds, and this one holds, and the formula holds. But I have to make sure that that's going to be the case for both settings of x because I'm saying for all x. So if I set x now to false, because I already showed that it works for x equal to true, if I set x equal to false, if I set now y to be true, this is going to hold.

So this expression is true, because it is the case that for every way to set x, there is a way to set y. So this part holds. Let's compare that with this case.

Is there some way to set y such that no matter how I set x, this is going to hold? And that's not going to be true. No matter what you pick for y, there is going to be some way to set x to make this false. So does there exist a y such that every x makes this true? No.

If you try x equal to true, it's not going to work. If you try x equal to false, it's not going to work. So this second phi 2 expression-- quantified QBF-- is false. We're going to be playing with these a lot. So it's important to understand how this quantification works.

So TQBF is the problem of testing whether one of these QBFs is true. Or phrased as a language, it's the collection of true QBFs. And that's where we get the acronym TQBF-- not acronym, the abbreviation TQBF for the True Quantified Boolean Formulas.

So going back to that example, phi 1 is a true quantified Boolean formula, and phi 2 is not a true quantified Boolean formula. So that's why phi 1 is in the language. Phi 2 is not in the language. Now our computational problem is to test whether quantified Boolean formulas are true or not. And then we can do it in polynomial space.

Oh, there's a Check-in first. I claim that SAT is a special case of TQBF. Why is that? How can we think of SAT as a special case? If I give you a SAT formula, how can I see that as also a TQBF problem?

If you want to test if that formula is true, what would you say? Remove all the quantifiers or add some quantifiers? And what kind of quantifiers, maybe? How is SAT, just testing a formula is satisfiable, a special case of this-- what I claim is a more general problem of solving these TQBF problems? Closing down. Last call.

Yes, indeed. Satisfiability-- so C is correct. When you're talking about a satisfiability problem, you're saying, is there a satisfying assignment? Another way of writing that down is take that Boolean formula and put "exists" in front of all the variables.

Does there exist a way to set x1 and x2 and x3 and x4 to make the formula true-- to make that formula hold? So SAT is a special case by aiding exist qualifiers of a TQBF problem. So C is correct.

So why is this problem in PSPACE, as I claimed? And for that, we're going to give a simple recursive algorithm. In any quantified Boolean formula-- now, if you want to test if it's true or not, we're going to basically strip off the leading quantifiers. So if it's an exist quantifier, we'll remove it and plug in true and false associated to its variable, and then solve those problems recursively. So this is just going to be a recursive procedure for solving TQBF problems, operating by stripping off the quantifiers in front and getting smaller and smaller formulas. But now we're going to be plugging in values, true and false, instead of relying on the quantifier to give us the meaning of the formula.

So first of all, if there are no quantifiers, then there are no variables, because all variables have to-- bound within quantifiers. And in that case, that quantified Boolean formula has to simply be the statement true or the statement false. And so you're going to output accordingly because that's all it can be if you have no variables.

If the formula starts with an exists quantifier, what you're going to do-- so here, psi is the remainder of the formula after you strip off that exist quantifier. So you're going to evaluate psi now, but take that variable that was bound by the exists and just plug in true and false respectively. So you're going to get two, now, new problems, and run them and evaluate them using the same procedure recursively, but now with x plugged in for-- true plugged in for x, and also then with false plugged in for x-- and get the answers for those two cases.

And if either one of them ended up accepting, then you're going to accept, because there exists a value of 4x which makes the whole thing true, because you just recursively showed that there was such a value, either true or false, the thing has accepted. And if both of them fail, then you're going to reject. And the very same idea if you have a for all quantifier.

You're going to evaluate the remainder of the formula, again, with x equal to true and false, so as two subproblems. But now you're going to require them both to accept because that's the meaning of for all-- that both assignments to x have to make the formula true. So you're going to evaluate them recursively and accept if both of them are true, as determined by your recursive-- your recursion.

So how much space does this use? I'm not going to go through this in great detail. But each recursive level uses just a constant amount of space. So every time you do a recursion, you have to remember that that value-- that assignment to that variable. You want to think of recursion as being implemented on a stack.

So you're just going to pop-- push on the stack that value of that variable, which is that true or false. So basically it's 1 bit of memory that you're going to require every time you're going down the recursion. You just have to remember what-- which case you're working on, whether x equal to true or x equal to false. And so each recursive level just involves constant space.

And the depth of the recursion-- how much might you have to remember? Well, it's going to be at most 1 for every quantifier, because that's-- you're stripping them off as you're going down the recursion. So that's going to be at most the length of the formula. That's at most the number of quantifiers you can have.

And so the total amount of space used by this is going to be just n, order n. So this problem is solved in NSPACE, and so that's why it's in PSPACE. I think that's all I wanted to say about this.

If we regard the tape in a Turing machine as memory in a modern computer, what does the finite control correspond to? The finite control corresponds to just a finite additional memory. The tape is an unlimited amount of memory.

Or if we're putting bounds, the amount of tape is going to be, say, n squared memory, where n is the length of m-- n is the length of the input. So, yeah, they're both memories, but the finite control is-- it doesn't grow with n. So that's going to be just some constant amount of memory.

What would be the time complexity of this algorithm? Time complexity would be bad. It's going to be exponential. So you'd have to just double-check that. But it's going to be something like 2 to the number of variables that you have-- 2 the number of quantifiers, plus some small overhead for evaluating the formula multiple time. But it's going to be exponential. What else can I answer for you?

So someone is asking, going back again to coNP, and how do we know there exists a problem in coNP that's coNP-complete? We didn't define even what that means. But coNP-complete means-- we're going to start seeing other examples of completeness for different complexity classes.

In particular, one thing that's going to happen on Tuesday is we'll see a problem that's complete for PSPACE. In fact, it's going to be TQBF. So looking ahead is going to be a PSPACE-complete problem.

But we even have to have the notion of what we mean by complete for these other classes. And in the case of coNP, a problem is coNP-complete if it's in coNP, and every other coNP problem is polynomial time-reducible to it, so just exactly the same as we had for NP, just plugging in coNP instead. And you just have to work through the logic, but it's pretty straightforward.

The complement of any NP-complete problem is going to be a coNP-complete problem using that definition. So I don't want to go through that, those simple steps. But you just can go and verify that offline-- that that's going to be true. And I think we're going to probably talk about that later in the semester, too.

So another question-- how does the TQBF algorithm-- ah, that is a good question here. Why is the TQBF algorithm that I just described in PSPACE? Doesn't the thing-- every time I'm doing a recursion, doesn't things branch out so that I end up using exponential space?

Critical thing which I actually don't think I mentioned, which I think is important to observe, is that when you're doing those two recursive calls, when you set x equal to true and set x equal to false, after you've determined the answer for when you set x equal to true, now you reuse that space, that very same space, to test what happens when you have x equal to false. So that's the power of space, which makes it different from time, is that it can be reused. So after you've got the answer for when you have x equal to true, now you free up that space.

That's no longer needed anymore. You just remember the answer. And now you see what happens when you have x equal to false, using that same space. So there's no exponential blow-up. That's an important point. I'm glad you gave me the chance to say it.

So somebody's asking about defining time of a non-deterministic Turing machine to the maximum time of each branch. Well, that's sort of what we have done. Maybe I don't understand your question. But you'll have to ask it afterward because I don't want to be delaying any more than we have. So we're going to go back and move on, here.

Second example-- and this one is a kind of a fun example, but it's also going to be an important one for us. It's called the Ladder Problem. So you may have seen something called a word ladder. But in general, a ladder is a sequence of strings which are all of the same length, but where consecutive strings differ in a single symbol.

So for example, if you a word ladder for English, it's going to be a ladder where all the words are-- all of the strings are English words. So here's an example. I thought I fixed that. OK, here is a word ladder for English. And maybe you've seen these.

Suppose I want to try to get from "work" to "play." But all of the intermediate strings should be English words with four letters that differ from their previous one in only a single letter. And I want to somehow change the word "work" to the word "play." So I don't know if you know.

So for example, I can change "work" to "pork." So here's just one letter difference, which looks like it's an improvement, because now I have the-- I'm in agreement on the play. But sometimes, you might change it. You might have a good change, and then you have to undo it later, which I think actually happens here.

So "pork" then is "port." But then we gave up that progress. We made "port" to "sort" to "suit" to "slot." You've got to understand what I'm doing here. Each case, I'm just changing a single letter.

But all of these words-- all of these have to be legitimate English words of length 4. "Plot," "ploy," and then "play"-- so that's what a word ladder in English would be. Of course, you can do it in different languages.

And now I'm going to talk about it abstractly, where instead of having a natural human language as being the test for a word-- for being-- a string being legitimate-- I'm going to define any old language. Let's say A is going to be some language, some set of strings. And those are going to be the legal strings that can be in the ladder. So ladder A is a ladder of strings that are all members of A.

And now the ladder DFA problem is A going to be the language of some DFA. So I'm giving you B and then a start string and an end string. So this is like work and play. U and v are like work and play, so where B is a DFA. And its language has a ladder that goes from u to v, and here are the intermediate strings.

All right, so I'm going to show you that this ladder DFA problem is an NPSPACE. This not super hard because basically-- well, let's just actually look at the slide here. The way it's going to work is it's nondeterministically going to guess that sequence from u to v.

So if I'm trying to get from work to play, imagine those-- I'm going to be using this in place of the language of my finite automaton, just because this is easier to talk about. But imagine these are being strings that are accepted by that DFA. So now I'm trying to get from my string u to the string v.

And I want to test. Can I get there by changing one letter at a time, but staying at strings that are accepted by the DFA? I'm just going to guess that sequence nondeterministically. But I have to make sure-- careful of two things.

I don't want to guess the sequence all in advance because that sequence might be exponentially long. You have to calculate how long it could possibly be. But you might change to one symbol, then change it to a different symbol, then change it back to that original symbol.

So the only bound that you can write down is the number of possible strings that you can have of that length. So it might be exponential. You don't want to write down that whole thing because that's going to be exceeding your space bound. But you don't need to. You're just going to guess them one at a time, forgetting about the previous one. So just keep guessing the next one in the sequence and only remembering that one and seeing if you ever get to the string-- your target string.

But then when you do that, you have to make sure that you don't end up going forever, because that's not allowed in your NPSPACE algorithm. So you're going to have to keep a counter to make sure that if you go beyond that bound, which is going to be the maximum number of strings you could possibly have, then you're going to just shut that branch of the nondeterminism off. You're going to just reject on that branch.

So here is-- I'm going to write-- say this here. Here is my nondeterministic polynomial space procedure. I'm given my language-- my DFA B and my start and end strings. I let y equal to start string. Write down the length of my strings that I'm going to have to keep in mind all the way through.

And then I'm going to just repeat the following t times, where t is the maximum length this can be, which is the size of the alphabet of these things to the mth power, where m is the length of those strings. And I'm just going tc nondeterministically change one symbol at a time, making sure that I'm staying in the language, so rejecting immediately if that change introduced a string outside the language, and accepting if that string that I get by changing that single symbol is now my target. And if I've gone through my bound, and I haven't managed to reach that target, then I'm just going to reject.

And we just have to observe that this algorithm doesn't use too much space. So if you imagine what we need here, here's my input, u and v, which is a blanked n, and the total amount of space. I just have to remember the current y and also my counter t, my counter up to t.

So each of those can be written down essentially in space. So the total amount is going to be order n space. So that shows that this ladder DFA problem is actually in nondeterministic space n-- nondeterministic linear space. And what we're going to show next is that this language is actually solvable in deterministic space. And this is perhaps a bit of a surprise.

So what's the size of the input? The size of the input is going to be what it takes to write down the DFA and the two strings, u and v. So here, yeah, I mean, I should have also included as part of the input the description of B itself. But that's going to be more in my favor because-- this is slightly incorrect, because B itself has to appear as part of the input, so apologies for that. But still, the amount of space used is going to be order n, because these are going to be actually less than n.

So we don't run out of time for the lecture, we can save additional questions for afterward. I'll stick around for a few minutes. But I just really have one more slide here, and that is proving this theorem that ladder can be done deterministically in polynomial space. And that's going to be important as a kind of preview of what we're going to be doing on Tuesday.

If this goes a little fast, I'll go over it again on Tuesday. So let's just see how it goes. So I'm going to show the same ladder DFA problem is solvable deterministically in polynomial space. But this time, it's going to be n squared space instead of nondeterministically in n space. So there's going to be some cost, but it's only going to be a squaring.

So remember what the problem is. I'm giving you that DFA, and I've given you two strings in the language of that DFA. And I want to know, can I get from the first string to the second string by changing one symbol at a time, but always making sure that the strings along the way are accepted by that DFA?

So I'm going to introduce notation saying, can I get from string u to v by a ladder? But now I'm limiting how many steps I can take. So I'm writing u to v, but doing it only within B intermediate strings, B steps. So is there a ladder from u to v of length at most B? That's what it means to write this notation down.

So I'm going to give you a recursive procedure to solve the bounded ladder problem, where it's just like before, but now I'm going to say, not only does there a ladder from u to v, but there's a ladder of length at most B. And that's going to allow me to solve the ladder problem recursively by shrinking the size of B. So how is this going to work?

Here is going to be the idea. So here's my u and my v. And the procedure is going to work by instead of nondeterministically guessing the steps that take me from work to play, because I don't have nondeterminism anymore, I have to operate deterministically.

What I'm going to do is work-- instead of going from-- looking at the very first thing that follows from u, I'm going to jump right to the middle and try every possible middle string. I have no clue even what that middle string should look like, so I'm going to try all possibilities in sequence.

Once I have one of those possibilities, I'm going to recursively try to solve the problem by splitting that. But I'm now going to divide that B value in half. So here is the maximum value we can have. This is the t from the previous slide which is the maximum length.

Here, I'm going to try every possible intermediate. Let's start off with A, all A's. And now I can cut the problem in half. Can I get from work to all A's and all A's to play? Well, very first thing I should check is making sure that all A's in fact is a string in the language.

And if we're thinking of the language as-- imagining it's like English-- all A's is not a legitimate word. So you try the next one, AAB. And this is how it's going to work, but now you're going to be-- instead of using English, you're going to feed it into the finite automaton, just one after the next, trying every possibility until-- like a clock, like an odometer, just trying them all, until eventually you find a string that's in the language.

I'm representing that by an English word "able." Maybe that's the first word that you would have found. And then once you find that, you're going to-- can I get from "work" to "able" and "able" to "play" recursively, reusing the space again, but now where the bound is cut in half? So that's the whole algorithm.

So just going through it quickly, and I will do this again, here is my DFA going from u to v within B steps. First of all-- oh, this is bad. t should not be 1. This should be B. If B is 1-- let me quickly fix that. So these t's should be B's. My apologies.

So if t is 1-- if B is 1, then they have to-- then I'm only allowed a ladder of length 1. Now I just check intermediate directly. Do u and v differ in just-- in one place? If yes, then I accept, else, I reject.

If it's greater than 1, now I'm going to do this procedure that I described. I'm going to try for each possible w in the middle, I'm going to try that w, test whether I can get from u to w in half the number of steps and from w to v in half the number of steps, and accept if they both accept. And if trying all possible w's, none of them work, then I know that there's no way to get from u to v in B steps. And so then I reject.

And then to do the original problem, which was not the bounded ladder problem, I do the bounded ladder problem where I put in t, which is the maximum possible length that it could be to get from work to play or to get from u to v. So the space analysis-- well, I'm kind of out of time, here. So we're going to go through this again next time.

So let me skip that analysis. I'll review this next time. I have a very quick Check-in. I just want to get here. Find an English word ladder that connects the word "must" to the word "vote." You can think about that.

I mean, it's not that hard to come up with such a word letter, so I encourage you to think about that-- also, to think about voting, which is also important. That's coming up. Another 5 seconds here. I'm going to end this, so make sure you get your credit for the Check-in.

So we're at the end of the hour-- end of the 80 minutes, anyway. So this is what we did today. And looks like I ran over by a minute, so my apologies. But I'll stick around here if any of you have any further questions. But otherwise, lecture is over. See you guys.

Do we know anything about ladder for other kinds of languages? I don't know. Interesting question, whether you can say some nice things about the ladder, the ladder problem, in other cases. I don't know.

Why is t here, this value of t? Sigma to the m, the maximum length of a word ladder. First of all, we have the m. Maybe I should have written this down. m is the length of the words.

Sigma is the alphabet of the words. So the number of possible different words is sigma to the m. These are all possible words that there could possibly be. So there's no reason in the word ladder ever to repeat the word because you can just find a shorter word ladder that still does the job of connecting a start and the end, because you can just cut out that middle part, the repeated part. So in that case, the longest possible word ladder is going to be the total number of possible words that you can have, which is going to be sigma-- the size of sigma to the m.

Explain again why coNP is a subset of PSPACE. Well, maybe I'll say it this way. Why is it every coNP language also in PSPACE? Well, take the complement of your coNP language. That's an NP language.

An NP language is in PSPACE because we proved that. That's what we proved. But if a language is in PSPACE, its complement is also in PSPACE, because for a deterministic procedure, you can just flip the answer of the machine. So if B, language B, is in coNP, its complement, B complement, is in NP, which is in PSPACE. So B complement is in PSPACE. So now PSPACE-- you can invert the answer. And now B is also in PSPACE. I hope that helps.

Somebody's giving me the answer to get from "must" to "vote." But I've seen that answer. There are online tools that will answer word ladders. So you just plug in the start and the finish, and it'll give you the word ladder.

And then the one that this person has sent me is the one you get from that tool, so I suspect he didn't find it himself. Before lecture, I actually solved that on my own, besides the one that-- I know the one that the tool will give you. So that tool gives one in, I think, five steps.

And I found one on my own in six steps. It's not that hard. Yeah, "must," "most," "lost," "lose," "rose," "rote," and "vote." So they made that seven steps. Anyway, for short words, you can solve these generally pretty quickly on your own. What else? What else can I do for you?

Do we need to worry about coming back to a previously visited word on the construction on this page? No, we don't have to worry about coming back to a previously visited word. All you need to worry about is making sure that you bound how long are you going to go for. And that's where the previously visited issue comes in.

If the word ladder that you found repeats some word, well, then there would have been a shorter word ladder that would have also worked. But it still shows that it's possible to get from the start word to the finish word if you have a repeated one in between. So that doesn't matter.

We don't have to worry about that. If you did, then it would be problem. So I think I will-- 4:05. I think I'm going to head out. See you all, guys. And I'm going to join my TAs in a meeting shortly. So bye-bye. Thank you for being here.