[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: Welcome, everyone. Welcome back to theory of computation. And just to recap where we are, we have been looking at time complexity and space complexity. And we just finished proving what are called the hierarchy theorems, which, in a nutshell, basically say that, if you allow the computational model to have a little bit more resource, a little bit more time, a little bit more space, then you can do more things with certain conditions.

So we proved that last time. It was a proof, basically, by a diagonalization. I don't know if you recognized the diagonalization there, but when you're encoding a machine by an input and then basically running all possible different machines, that's essentially a diagonalization. So today, we're going to build on that work to give an example of what we call a natural intractable problem. We'll say a bit more about what that means.

And then, we're going to talk about something which is a different topic, but nevertheless related, having to do with oracles and methods which may or may not work to solve the P versus NP problem, which, of course, is a big open problem in the field. OK. So the time and space hierarchy theorems-- because we're going to be using those today-- they say that if you give a little bit more space here-- so for space constructible functions, functions that you can actually compute within the amount of space that they specify, you can show that the things that you can do in that much space is probably larger than what you can do in less space. And you can prove a similar slightly weaker fact about the time complexity classes.

So what that means is that these classes form a hierarchy. So as you add more time, or let's say, in this case, space, from n squared, to n cubed, to n to the 4th, you get larger and larger classes, which I'm kind illustrating here by putting a dot there, which shows that there's something that we know that's new in those classes as you go up these different bounds. And this is going to be true for space complexity and it's also going to be true for time complexity.

And one of the corollaries that we pointed out last time is that, PSPACE is a-- properly includes non-deterministic log space, NL. So NL is a proper subset of PSPACE. So there's stuff in PSPACE that is not in NL. And remember this notation here, this means proper subset.

One of the things that-- a follow-on corollary that we didn't mention last time, but that's something that you should know, is that the TQBF problem, our PSPACE based complete problem, is an example of a problem that's in PSPACE, obviously, but we know it's also not in NL. And in order to get that conclusion, you have to look, again, at the proof that TQBF is PSPACE complete, and observe that the reductions that we gave in that proof can be carried out not only in polynomial time, but they can be carried out in log space.

And therefore, if TQBF turned out to go down to NL, then because everything in PSPACE is log space reducible to TQBF, that would bring all of PSPACE down to NL. But that we just proved is not the case. So therefore, TQBF could not be in NL. OK, and we're going to be using that kind of reasoning again in this lecture.

So just a quick check-in. These are a few, more or less easy, maybe more or less tricky, follow-ons that you can conclude from the time and space hierarchy theorems plus some of the other things we've proven along the way. And so just as a check of your understanding, maybe these a little bit on the tricky side, so you have to read them carefully.

Which of these are known to be true based on the material that we've presented? And this is also just material that's the facts that we know to be true in complexity theory. So let me launch that poll. And just check off the ones that we can prove. Hmm. OK.

I'm going to close it down. So please answer quickly if you're going to. OK, 1, 2, 3, end. OK. Well, the two leading candidates are correct. And the two that are the laggards here are, in fact, the ones that are not true.

So A and D are not true, based on what we know. And B and C are true. So let's understand, first of all, A, we know it's false because 2 to the n plus 1 is just 2 times 2 to the n. And so these two bounds differ only by a constant factor.

And so in fact, they're the same complexity class. And so you don't get proper containment for A. So that one we absolutely know is false.

D, well, if we could prove that, then we would have solved the famous problem, because we don't know whether even P equals PSPACE. So if P equals PSPACE, then certainly PSPACE would equal NP, which is in between the two. And so we don't know how to prove PSPACE is different from NP, that's based on the current state of knowledge of the field. So this would not be something that we know to be true based on what things that we've said.

Now, B follows directly from the time hierarchy theorem, because 2 to the 2n is the square of 2 to the n. And that is, asymptotically, a significantly larger bound. And so you can prove that time 2 the n is properly contains time 2 the n.

C is a little trickier because you need to remember Savitch's theorem. Savitch's theorem applies to space. But you also need to remember that what you can do in time, in non-deterministic time n squared, you can also do in non-deterministic space n squared, which, then, in turn, you can do in deterministic space n to the 4th, which is properly contained within space n to the 5th. So you can prove that PSPACE properly contains non-deterministic time n squared.

OK, just a bunch of containments there. A and C are perhaps, in a sense, it may be the most tricky of this group. OK. So let's move on.

So we're going to introduce, today, two new classes. And actually, I want to go back to here. What are we going to be trying to accomplish in today's lecture?

So we're going to be looking at provable intractability. So a problem being intractable for us means it's outside of P. So we can't solve it in polynomial time. For our perspective, we're going to call that an intractable problem.

Now, this problem over here, that's sitting in time 2 the n, but not in smaller classes, so this is an intractable problem. That's outside of P. But this example of a language, if you remember how the time hierarchy theorem or the space hierarchy theorem was proved, basically, this language itself is not an interesting language for other than the purpose that it serves, to be in that class and not in a lower class. But it's not a language that anyone would care about. And it's not even a language that is easy to describe.

It's just the language that some Turing machine decides, where that Turing machine is especially designed to have the property that its language is at a particular complexity level. But otherwise, there's no nice description of that language. It's not like a to the n, b to the n, or some equivalence of 2 dfa's or something like that.

So I would say that that language is, in a sense, it serves its purpose, but it's not a natural language that you really care about. So one one of the goals of today's lecture is to give an example of a natural language, a naturally-occurring language, in a sense, that's easy to describe, where you can prove that that language is intractable, is actually outside of P. So that's a bit of motivation where we're going.

So along the way, we're going to introduce these exponential complexity classes, exponential time and exponential space, which are exponentially bigger than polynomial time and polynomial space classes. So it's 2 tc the n to the k in both cases. 2 to a polynomial.

And the first five of these classes, L through PSPACE we've already seen, and exponential time and exponential space extend the containments that we've already seen. So you have to double check that you understand why PSPACE is a subset of exponential time. But that's because that, as we showed, going from space to time, you can do that with an exponential increase. That's the cost of the simulation.

And going from time to space, you don't need any increase at all. Anything that you can do in a certain amount of time, you can do in that much space. So anything you can do in a certain amount of space, you can also do in exponentially more amount of time. OK, so those were simple theorems that we proved right at the very beginning.

Now, the hierarchy theorems allow us to conclude some separations among these classes. So we already looked at this one, NL versus PSPACE. And we saw that because NL is, by Savitch's theorem, in deterministic log squared space, which is properly contained in polynomial space, you get a separation between those two classes, provably. And for similar reasons, polynomial space to exponential space, you're going to get a separation from the space hierarchy theorem. And polynomial time to exponential time, you get a provable separation by virtue of the hierarchy theorem.

Now we're going to define complete problems for these two classes, exponential time and exponential space. So we have exponential time complete. It's going to be analogous to what we showed before, which is that it's a member of exponential time. And every problem in exponential time is reducible to it, let's say, in polynomial time, though it's not going to really turn out to be matter. It could be in log space.

Some simple method of doing the reduction is going to be good enough. Let's say polynomial time is the typical definition. And the same thing for exponential space complete. We'll say it's exponential space complete, if it's an exponential space. And anything else in exponential space is polynomial time reducible to it.

OK. But the important thing is that if something is exponential time complete, you know it's outside of P, for the same reasons we've now seen several times. Namely, that if an exponential time complete problem ended up being in P, then because everything else in exponential time is reducible to the complete problem, they would also be in P. And so exponential time and P would be equal.

But we just said they're not equal because of the hierarchy theorem. So the logic is the hierarchy theorem separates the class, and then the complete problem inherits the difficulty of the larger class. So the complete problem cannot be any lower than the other problems in the class, because they're all reducible to it.

So the same thing is going to be true for an exponential space complete problem. Can't be even in PSPACE because exponential space and PSPACE are different. And if it's not in PSPACE, it's not going to be in P.

And so in both cases, if you have a problem that's complete for exponential space or exponential time, we know that those problems are intractable. And our strategy, then, for giving a natural intractable problem is to show it's complete for one of these classes. And it's actually going to turn out to be an exponential space complete problem that we're going to give as our example. OK, so that is the plan.

I think it's a good time to-- let's just take a few questions here to make sure we're all on the same page as what we're doing. So let me just read. I got a couple of questions already in here.

So this is a little bit of a side comment that somebody-- that's an interesting question. Basically, is it possible that we may not be able to prove, solve the P versus NP problem, that it's not a problem that one can answer from the basic axioms of mathematics, if I'm interpreting the question correctly. There are certain problems in mathematics-- and I think I, perhaps, I mentioned earlier in the term, the problem of whether there is a set whose size is in between the integers and the real numbers.

We know the real numbers are larger in size than the integers. That was our first example of a diagonalization. And is there a problem of size strictly in between the two? Bigger than the integers, smaller than the real numbers. So that's a problem that was posed a long time ago. It was one of Hilbert's problems. And was eventually shown to be unanswerable using the basic axioms of mathematics.

So the question is, maybe P versus NP is in the same category. Could be. That could be true of any unsolved problems in mathematics.

But at least our experience has shown that the kinds of problems that, at least, have been shown to be unsolvable from mathematical axioms tend to involve infinities and very large things, things that are very far from our intuitions. And something as down to earth as P versus NP, at least, it would be very surprising to me if that turned out to be unanswerable using our mathematical axioms. But, who knows?

Oh, this is another good question. Do the time and space hierarchy theorems have non-deterministic variants? Yes, they do. They're much harder to prove, however, and we're not going to cover that. But you can also prove that non-deterministic time, n cubed properly includes non-deterministic time n squared. You're not going to be responsible for that. Don't worry.

If you try to actually prove that, you'll see the diagonalization doesn't directly work. And so you have to do something fancier. People are asking about which reduction method to use. Again, the kinds of reductions that we encounter are always very simple. So we're just going to be working with very weak notions of reductions. Not interesting yet, generally, to consider powerful kinds of reductions like polynomial exponential time reductions or things like that. So it's just not something that people really think about much.

I mean, I can talk about it at length offline. But let's just assume that our reduction strength is something very low. Log space is going to be good enough to do all of the reductions in this class. OK, so let's move on, then.

So here is the problem that we're going to spend the next 20 minutes or so proving to be exponential space complete. I have got to do a little introduction first. So this is not the problem, but this is related to the problem.

So the problem of testing if two regular expressions are equivalent. Write down to regular expressions, do they generate the same language? So that problem actually turns out to be in PSPACE. So it's not going to be exponential space complete. It's actually in PSPACE.

I don't think we're going to have-- I thought about presenting it in the lecture. It's not that hard to show. But it just took too much time and doesn't really introduce new methods. It's a good exercise, actually, using Savitch's theorem. But maybe we'll do it in recitation, or if the lecture miraculously ends earlier, I'll do it at the end. But I don't think we'll have time.

But that's a setup for the intractable problem that we're going to talk about, which is very related. Now, OK, before we get to that, so if I have a regular expression, I'm going to enhance our regular expression in one simple way, by allowing exponents or exponentiation.

And that means if I have a regular expression R, I can write R to the k to mean R concatenated with itself k times. We've been sort of informally using that all the way along anyway, like when we talk about 0 to the k, 1 to the k. So if we're going to formally allow that when we write down regular expressions, in some cases, that might allow the regular expression to be much smaller, especially if we're writing down k in binary.

Because I can write R to the million with just a few symbols if I have exponentiation. But if I don't have exponentiation, then I have to write R concatenated with R out a million times, and I get a much, much longer, an exponentially longer expression if I don't have that exponent as a way of describing regular expressions. And that's going to make a big difference.

So now, the equivalence problem for regular expressions with exponentiation-- that's what that little up arrow means, what it signifies-- now I'm giving you two regular expressions. But they're allowed to have the exponentiation operation in addition to the standard regular operations. So now, testing whether two of these regular expressions that have exponentiation, that problem turns out to be exponential space complete.

So here's the equivalence problem for regular expressions with exponentiation. That's an exponential space complete problem. And as we pointed out, that means this problem is provably intractable. So there's just no way, in general, to solve that problem in polynomial time. That's proven, that's known.

So we're going to go through the reduction. I think it's going to be our last reduction of the term, of proving problems complete for some class. But each one of those has their own kind of thing that makes it special.

So first of all, we have to show that it's in exponential space. That's really going to rely on this other fact that we didn't prove. So I'm going to go over that very quickly. But the interesting part is doing the reduction. So if I have something in exponential space that I can show that I can reduce it to the equivalence problem for regular expressions with exponentiation.

OK, so quickly arguing part one that we're in exponential space, basically, what you do is you take your two regular expressions that you want to test to see if they're equivalent, but now they have exponentiation. And as a first step, you get rid of the exponentiation. You just expand things out by repeating the parts that have the exponents.

And of course, as I said, that's going to make the expression themselves exponentially bigger. But now, you run the PSPACE algorithm on those two exponentially larger expressions. So the input that the PSPACE algorithm is now exponential in the original input size, but it's PSPACE in that enlarged input. So that's going to give you an exponential space algorithm in the original input size, because you expanded it to become exponentially bigger, and then you run the PSPACE algorithm on that expanded problem. So that gives you an exponential space algorithm for this problem.

But now, what we're going to do-- the interesting part is the reduction. So given some language and exponential space, say, decided by some Turing machine in that amount of space, 2 to the n to the k, we're going to give a reduction that maps a to this equivalence problem. Got it? That is the plan.

So let's make sure we're all together on the plan before we go ahead and carry out that plan. We just sort of set things up here, in a sense, for what we're going to be doing. So feel free to ask a question on just the plan.

It's going to get technical. Because, as doing these reductions always is, there's a simulation involved, and you have to kind of describe that simulation in its own way. So now, we're going to be simulating, in a certain sense, M on w, the decider for this exponential space, problem A, we're going to take M on w and we're going to somehow have to express the fact that M accepts w using this equivalence problem for regular expressions with exponentiation.

So no questions? Why don't we move on? I have three slides on this, but they're kind of dense, I'm sorry to say.

So here is the plan as usual. We're going to map A with a polynomial time reduction to the equivalence problem for regular expressions with exponentiation. So that means we're going to have to take an input, which may or may not be in A, and produce two regular expressions with exponentiation, which are going to be equivalent when w is in A. Or when M accepts w.

So it's going to be, as these things always are, these are going to be in terms of the computation history for M under w. But in this case, it's going to turn out to be convenient to work with the rejecting computation history for M on w. So remember, now we have a Turing machine M. It's a decider, so that means it always holds-- for the strings in the language, it ends up at a Q accept state, for things not in the language, it ends up at a Q reject state.

So a rejecting computation history is the sequence of configurations the machine goes through from the start configuration until it ends up at a configuration with a reject state, a rejecting configuration. And we're going to make a regular expression that describes all strings except for that one. It's going to avoid describing a rejecting computation history for M on w. Otherwise, it's going to describe all possible strings.

Now, if M does not reject w, so there is no rejecting computation history-- namely, M accepts w, by the way. So if M accepts w, does not reject w, it does not have a rejecting computation history, what is R1 describing? Well, it's describing, in that case, everything, because there is no rejecting computation history.

So it's describing every other string besides. So that means it's describing all strings, if there is no rejecting computation history in the case that M accepts w. So what does that suggest we should use for R2?

R2 is going to be the regular expression that just generates all strings. So we'll be testing whether R1 generates all strings or not, which is the same as saying does M accept w or not. So R2 is going to be-- I would like to say sigma star, but sigma is really the input to M, and gamma is the tape alphabet for M. So we have a lot of Greek letters to play with, so we're going to use delta for the alphabet that we write the computation histories in.

If you want to get reminded what that delta is, a computation history can have a tape alphabet symbol for M, it can have a state symbol for M, or it can have a delimiter pound-- hashtag. So it's either a capital delta alphabet is a tape alphabet symbol, or state, something representing a state symbol, or a hashtag. That's just delta. So don't get-- I always feel bad if somebody gets confused by something that's supposed to be very simple. Don't get confused by delta star. This is just all possible strings over the alphabet delta.

OK, so what does R1-- so my job is to do R1. R2, I already told you. R1 now has to describe all those strings except for the rejecting computation history. So everything that fails to be a rejecting computation history-- so it fails either because it started wrong, or it ended wrong, or it's wrong somewhere in the middle. And by wrong I mean, it fails to correctly describe the way the machine operates if it's ending up rejecting w.

All right. So I'm going to describe all those possible strings by breaking it down into those three categories. Starts wrong, ends wrong, or somewhere computes wrong along the way. OK.

So rejecting computation history looks something like this. Here's the start configuration as we usually envision it. It's a start state looking at the first symbol of the input, and there's the rest of the input. So let me just write this out. This is a rejecting computation history now. So the first configuration, the second one, and so on and so on, until we end up at a rejecting computation-- rejecting configuration.

Now, for convenience, I'm going to insist that all of these configurations are the same length. It's going to make my life easier in doing the proof. But why can I do that? Well, I'm just going to take them-- you know, because usually you think of the configurations, they start small because they're just basically of length n, but this is using exponential space, they're getting longer and longer. Let's just pair them all out with blanks so that they're all the same size.

So as I've indicated over here, we're adding in a bunch of blanks. It's going to be a lot of blanks here, to make sure they all have length 2 to the n to the k, which is the maximum size of a configuration when you have that much space.

I'm going to construct-- so basically, that's my job. I'm going to construct R1 so that it generates all those strings. I wrote a little box around that thing I'm trying to-- that's my to do. It's going to help me in the coming slides because they're a little bit dense.

When I'm going to draw this sort of reddish, pinkish box around something, that means that I'm going to try to describe all strings except for that one. I want to avoid describing that one, because that's the rejecting computation history, but I want to describe everything else. That's my wish.

So here's a check in before we move forward. But we can also-- maybe we should just take some questions, even before we launch the check in. How are we doing here?

So, is our one describing-- well, R1 is a regular expression. Over here, we're talking about a-- this is just an ordinary computation history, but it ends with a reject. That's all. A rejecting computation history is just one that's a little different at the end. The machine just ended up rejecting instead of accepting. Otherwise everything has to be spelled out in accordance with the rules of the machine and the start configuration.

Yeah, we were assuming one rejecting state. Yeah, that's the way we actually define Turing machines in the first place. But, who's arguing. Yeah, there's one reject state. We're all deterministic, correct.

Why do we need the padding? Because I want to make these all the same size, all of these configurations. That's going to help me later in terms of describing the invalid configurations, the ones that are not legal configurations, legal rejecting configurations. So just simply a matter of convenience, but just accept it for now. I just want all of those configurations to be the same length in my rejecting computation history. Otherwise I'm not going to-- I'm just coding that rejecting computation history in this particular way.

So people are asking about the details of bad start. That's yet to come. I have two more slides on this. So I'll tell you about how we're going to do those. So R bad-start-- that's a good question-- is R bad-start all-- these are all the strings that don't start this way. We'll see it in a second. But R bad-start are all the things that don't start with the-- they start bad. They're not starting with the start configuration. They're starting with some other junk.

Do we need only one rejecting computation history? What about the other ones? This is a deterministic machine, so there's only going to be-- if I prescribe the lengths as I've done, there's going to be only one rejecting computation history. Because it's deterministic, everything is going to be forced from the beginning.

Should R1 be the not of those three? No. R1 is describing all of the strings except, except this one string. So I'm capturing all the different possible ways a string could fail to be the string. It could start wrong. Could be wrong along the middle somewhere. So I have to union them together. Because I'm describing-- as I always believe, negations are the most confusing thing to everybody, including me.

So we're describing all the things that are not this string. We're trying to stay away from that one. We want to describe everything else.

All right, I think I'd better move on here. We've got a lot of questions. Talk to the TAs.

All right, check in. How big is this rejecting computation history anyway? Interesting. There's a lesson here. I got a big burst of answers right at the very beginning. All wrong.

But then the bright-- the people who took a little bit more time to think started getting the right answer, which is-- let's look. We've got a close election here folks, so now I have to report. Hope we don't have to do a recount.

OK, come on guys. Answer up. 10 seconds. This is not super hard. Stop the count. Yeah, I think we'd better stop at this, we're on the edge. OK, 3 seconds. End polling. Share results.

The correct answer is, in fact, c. Why is that? Because each configuration is 2 to the n to the k. So that's how much space the machine has, exponential space. But the amount of time, which is each one-- the number of configurations is going to be the amount of time that's used. It's going to be exponentially more even than that. So it's going to be 2 to the 2 to the n of the k, is how many steps the machine can run. And that's going to be how long the computation history could be. So it's a very long thing.

And when you think about it, the regular expression we are generating, how big is that? The regular expression-- again, a lot of people playing off my comments here. Were the votes legal or not? OK. Let's focus here.

So this is doubly exponentially large. How big is the regular expression that we're generating? Well that has to be produced in polynomial time, so it's only polynomially big. So we have this little teensy weensy, relatively speaking, regular expression, which is only n to the k. It's having to describe all strings except for this particular string, which is 2 to the 2 to the n to the k. So in a sense, this string that is related to that regular expression is doubly exponentially larger than that. And that kind of presents some of the challenge in doing the reduction, in constructing that regular expression.

So let's move on and start doing-- this is the hard stuff. Here is the bad start, which is challenging enough. Even this little piece is going to be a little bit challenging to describe.

Just rewriting from the previous slide. So we're trying to make R1, which is generating all the strings except the rejecting computation history for M on w. It's in those three parts. Right now I'm describing the bad start piece. So that's going to describe all strings that don't start with this C1.

So let me write that out here. This is going to generate all strings that don't start with C start or C1, which is as specified. Looks like this. So any string that doesn't start with these symbols, doesn't start exactly like this, should be described by bad start, that regular expression.

So that, in itself, is going to be further subdivided. And the reason for that is not that hard to understand. I'm going to-- bad start is going to accomplish its goal by saying, well, anything that doesn't start this way either doesn't start with a q0, or doesn't or doesn't have a w1 in the next place, or doesn't have a w2 in the next place. Or somewhere along the way, it has a wrong symbol. Each one of these guys is going to be about one of those symbols being wrong in some particular place.

So I'm going to show you what those look like. So right now, I'm going to focus my attention on describing all strings except for this one. All strings that start with something except for this one. So just remember, delta is the alphabet for the competition histories. And some notation here, delta sub epsilon, we've seen this before, is you're going to add in epsilon as an allowed thing for delta. So it's all the symbols, or epsilon, now thought of as a set here.

And furthermore, it's going to be convenient to talk about all of the symbols in delta, except for some symbol. So like at the very beginning, q0. I want to talk about all of the symbols except for q0 symbol. Because that's what I'm going to be using to start off R bad-start. It's going to be anything except for q0.

So let's just see how that looks. So here is S0, the very first part of our bad start. It's going to say-- I'm trying to color the active ingredient here in the pink color. So delta, with q0 removed, followed by anything. So this little regular expression here describes all strings that don't start with a q0, as I'm indicating over here. All strings that don't start with a q0 is what as S0 describes. You have to understand that, because it's just going to build up from there.

So what do we want to say for S1? What's going to be all strings that don't have w1 in the second place? So I'm going to write that over here. S1 is anything in the first place-- I mean, if the first place was wrong, S0 took care of it. So I'm just going to keep my life simple. All I want to do is describe all of the places where the second symbol is wrong. Namely, it's not w1. So anything in the first place, something besides w1 in the next place, and then anything at all afterward. Those are all strings that don't have-- [AUDIO CUTS]

So I'll write it over here like that. Now S2 similarly is going to d since I have exponentiation, let's use that for convenience. Delta delta, or just delta squared. So anything in the first two places, then not w2, and then the next place, and then anything. So that's going to capture this part.

So this is what these S's do, and you can sort of get the idea. So dot, dot, dot. This Sn is going to describe everything except for wn in that location, which is going to be the n plus first location, actually. And now I have to continue on doing that for the blanks. So now, if you think with me, let's just take a look how that could go.

The next symbol, which is skipping over the n plus 1 that I've already taken care of, I want to say it's not a blank symbol in this very first location after the input. So again, I'm describing these non-- these strings which are not the start configuration. It could fail because there's not a blank where there's supposed to be a blank.

Suppose I do that for each one of these guys. That would work. But. But what? Think.

This is actually not going to be a good solution for us. Because there are exponentially many blanks over here. This is a hugely long configuration. And so there's exponentially many blanks. If I do it this way, I'm going to end up with an exponentially large regular expression. And that's not doable in polynomial time.

So I have a more complicated way of getting the same effect. Which is-- I don't really expect you to fully parse through this right now, in real time in lecture, but let me try to help you. What I'm going to do is skip over these first initial n plus 1 places, and then a variable number of places, which is indicated by the next piece here. And the way that works is-- these are all strings of length n plus 1 through the end of the configuration.

And to understand that, it's almost a little too technical to even try, but let's see. If I put delta to the 7, that's all strings of length 7. But if I put delta sub epsilon to the 7, if you think about what that means, that's all strings of length between 0 and 7. Because I can either have it as epsilon as my variable or a symbol from delta.

And so that's what I'm doing over here. I'm getting a variable length space, spacer of deltas, that are going to then end up at a certain location-- I'm going to say at that place. Then I have a non-blank. Because all I need to do is describe the strings that fail to have a blank somewhere in this range. So we've got to sort have a variable spacer out to that spot, where that missing blank might be. So that's what this describes. If you didn't get that, don't worry. That is a technical point and you can try to think about it offline.

And then at the very end, I'm going to describe what happens. Describe the strings that fail to have a hashtag in that location. It's how I describe all strings that don't start right. That's a lot of work, just to do that little piece. Fortunately, the next two pieces are easier, surprisingly.

You can jump in with a question, but maybe I should move, push on.

So now I'm going to describe the bad move and bad reject pieces. And bad reject generates all strings that don't contain the q reject symbol. So that's going to certainly describe all of the strings that don't end correctly. And that's just simply the delta with the q reject symbol removed, and then any string of those.

That's all strings that don't have q reject. So that's going to describe all strings that don't end with a q reject, plus some other junk strings along the way. But that's all that's never a problem, to put in other strings that you might be capturing in some other part of the regular expression that you know are bad strings. You just want to make sure you don't put in that one uniquely good string, which is the rejecting computation history, good string.

And lastly, we're going to use the notion of the neighborhoods. You might think this is the hardest part, but in fact not that hard. So these are all of the strings that have somewhere along the way a violation according to M's rules. You want to describe all of those as well. I'm going to do that in terms of the neighborhoods. But the neighborhoods are going to be stretched out. We don't have a tableau anymore, so they're not so easily visualizable, but it's the same idea, the neighborhood.

So this is abc and def. But now it's an illegal neighborhood. def does not follow from abc. If all the neighborhoods are legal, then the whole computation is a legitimate computation, provided it starts and ends correctly. So if it's not a legitimate computation, there's got to be an illegal neighborhood somewhere. And I'm going to just describe all strings that have an illegal neighborhood. And the interesting part is that you have to describe-- you have to place that separator between abc and def.

So this is another place where we're going to critically use the exponentiation, and the fact that all of the configurations are the same length. That's what we're using there. We know exactly how far apart the bottom of the 2 by 3 neighborhood is from the top of the 2 by 3 neighborhood.

So we're going to take a union over all illegal 2 by 3 neighborhoods. Neighborhood settings, I should say. And there's only a fixed number of those, for the same reason that we had in the Cook-Levin theorem. There's a fixed number of those, depending upon the machine.

And now we're going to have, say, we're going to start with anything. Here's the top of the neighborhood. Here is the separator that separates the top from the bottom in the two consecutive configurations, here's Ci going C i plus 1 inside my computation history. And then after that separator, I put in the second part of the neighborhood, which is the def. You have to really be comfortable with the way we've been presenting these other reductions up till now, to really get this.

Anyway, I think we're at the break. So we can just take questions during the break, if you have any. And I will, otherwise, see you in five minutes.

In my description back here-- let me just take this off. For bad reject, it looks like I'm doing kind of overkill, and maybe doing something wrong here. I'm describing all strings that don't have a reject anywhere. But as long as I don't describe the legitimate rejecting computation history, I do describe all strings that don't end correctly, I'm good. I could go through more effort to make sure that I'm only describing the very last configuration here as not having the reject. But that would just be more work, and I don't need to do that work. So maybe it would be good just to understand why this is sufficient, what I've described here, and it's not going to cause me any problems.

I'm getting a note from one of my TAs, Thomas, saying that the notion "bad" perhaps is confusing, because bad sounds like rejecting. Yes. I mean bad in the sense of not describing a legal computation history. If you can think of another name, I'm happy to switch that for future years. Too late for now. But, yeah. I don't mean that rejecting, I mean that it's-- well, I don't know what the right term is. Illegal? Or-- I'm not sure what a good--

How are the neighborhoods defined here? What is the tableau here? I think you do need to think about it after lecture. But the tableau, you can think of the tableau now here just written out linearly. There are all of the rows now, instead of nicely organized into a table. They just appear consecutively, because I'm just trying to describe-- I need to do it to describe a string, whether my regular expression doesn't really make sense to think about. I mean you can fold it up into a tableau, if you like. And then abc and def will line up. But here, if you think about them written consecutively, this is exactly how far apart they end up being.

Are there only polynomially many illegal neighborhoods? That's why I kind of corrected myself. It's not illegal neighborhoods that we're talking-- because the number of neighborhoods in this picture is vast. But the number of neighborhood settings, the way to set these values to abc, def. I mean these are symbols that can appear in a configuration of the machine. There's only a fixed number of symbols that can appear here, that depend upon the definition of the machine.

So it's not only polynomial. There's a constant number of these things, that only depends on the machine. So you have to think about what's going on. There's a lot-- this is a lot on the slide.

Bad history for reject. It's a bad history for rejecting, somebody's suggesting. Yeah, it's a bad history. Fake news. Maybe we should be fake. Fake would be a good term. No, that's not so good. I don't know.

Yeah, 2 by 3. The reason 2 by 3, is the right-- Somebody's asking why 2 by 3. 2 by 3 is exactly the size you need to say that, if all the 2 by 3 neighborhoods are correct everywhere in the computation history, then the whole history is going to be consistent with the rules of M. It's going to be a legal representation of a computation of M.

So if the string, which is allegedly a computation history, has a bad neighborhood somewhere, bad 2 by 3 neighborhood somewhere, then-- well if it's not a legal computation history, it's got to have a bad neighborhood, 2 by 3 neighborhood somewhere.

OK, let's move on. Because I think we're out of time here. Our timer is up.

We're going to shift gears now anyway. So if you got a little lost in the previous proof, we're going to talk about something different. And in some ways, a little bit, I think a little lighter, a little less technical. And that's about oracles.

What are oracles? Oracles are a simple thing. But they are a useful concept for a number of reasons. Especially because they're going to tell us something interesting about methods, which may or may not be useful for proving the P versus NP question, when someday somebody hopefully does that.

What is an oracle? An oracle is free information you're going to give to a Turing machine, which might affect the difficulty of solving problems. And the way we're going to represent that free information is, we're going to allow the Turing machine to test membership in some specified language, without charging for the work involved.

I'm going to allow you have any language at all, some language A. And say a Turing machine with an oracle for A is written this way, M with a superscript A. It's a machine that has a black box that can answer questions. Is some string, which the machine can choose, in A or not? And it gets that answer in one step, effectively for free.

So you can imagine, depending upon the language that you're providing to the machine, that may or may not be useful. For example, suppose I give you an oracle for the SAT language. That can be very useful. It could be very useful for deciding SAT, for example. Because now you don't have to go through a brute force search to solve SAT. You just ask the oracle. And the oracle is going to say, yes it's satisfiable, or no it's not satisfiable.

But you can use that to solve other languages too, quickly. Because anything that you can do in NP, you can reduce to SAT. So you can convert it to a SAT question, which you can then ship up to the oracle, and the oracle is going to tell you the answer.

The word "oracle" already sort of conveys something magical. We're not really going to be concerned with the operation of the oracle, so don't ask me how does the oracle work, or what does it correspond to in reality. It doesn't. It's just a mathematical device which provides this free information to the Turing machine, which enables it to compute certain things.

It turns out to be a useful concept. It's used in cryptography, where you might imagine the oracle could provide the factors to some number, or the password to some system or something. Free information. And then what can you do with that? So this is a notion that comes up in other places.

If we have an oracle, we can think of all of the things that you can compute in polynomial time relative to that oracle. So that's what we-- the terminology that people usually use is sometimes called relativism, or computation relative to having this extra information. So P with an A oracle is all of the language that you can decide in polynomial time if you have an oracle for A. Let's see.

Yeah. Somebody's asking me, is it really free or does it cost one unit? Even just setting up the oracle and writing down the question to the oracle is going to take you some number of steps. So you're not going to be able do an infinite number of oracle calls in zero time. So charging one step or zero steps, not going to make a difference. Because you still have to formulate the question.

As I pointed out, P with a SAT oracle-- so all the things you do in polynomial time with a SAT oracle includes NP. Does it perhaps include other stuff? Or does it equal NP? Would have been a good check-in question, but I'm not going to ask that.

In fact, it seems like it contains other things too. Because co-NP is also contained within P, given a SAT oracle. Because the SAT oracle answer is both yes or no, depending upon the answer. So if the formula is unsatisfiable, the oracle is going to say no, it's not in the language. And now you can do the complement of the SAT problem as well. The unsatisfiability problem. So you can do all of co-NP in the same way.

You can also define NP relative to some oracle. So all the things you can do with a non-deterministic Turing machine, where all of the branches have separately access. And they can ask multiple questions, by the way, of the oracle. Independently.

Let's do another, a little bit of a more challenging example. The MIN-FORMULA language, which I hope you remember from your homework. So those are all of the formulas that do not have a shorter equivalent formula. They are minimal. You cannot make a smaller formula that's equivalent that gives you the same Boolean function.

So you showed, for example, that that language is in P space, as I recall. And there was some other-- you had another two problems about that language.

The complement of the MIN-FORMULA problem is in NP with a SAT oracle. So mull that over for a second and then we'll see why.

Here's an algorithm, in NP with a SAT oracle algorithm. So in other words, now I want to kind of implement that strategy, which I argued in the homework problem was not legal. But now that I have the SAT oracle, it's going to make it possible where before it was not possible. So let's just understand what I mean by that.

If I'm trying to do the non minimal formulas, namely the formulas that do have a shorter equivalent formula. I'm going to guess that shorter formula, called psi. The challenge before was testing whether that shorter formula was actually equivalent to phi. Because that's not obviously doable in polynomial time.

But the equivalence problem for formulas is a co-NP problem. Or if you like to think about it the other way, any formula in equivalence is an NP problem, because you just have to-- the witness is the assignment on which they disagree.

So two formulas are equivalent if they never disagree. And so that's a co-NP problem.

A SAT oracle can solve a co-NP problem. Namely, the equivalence of the two formulas, the input formula and the one that you now deterministically guessed. And if it turns out that they are equivalent, a smaller formula is equivalent to the input formula, you know the input formula is not minimal. And so you can accept.

And if it gets the wrong formula, it turns out not to be equivalent, then you reject on that branch of the non-determinism, just like we did before. And if the formula really was minimal, none of the branches is going to find a shorter equivalent formula. So that's why this problem here is in NP with a SAT oracle.

So now we're going to try to investigate this on my-- we're getting near the end of the lecture. We're going to look at problems like, well, suppose I compare P with a SAT oracle and NP with a SAT oracle. Could those be the same? Well, there's reasons to believe those are not the same.

But could there be any A where P with A oracle is the same as NP with an A oracle? It seems like no, but actually that's wrong. There is a language, there are languages for which NP with that oracle and P with that oracle are exactly the same. And that actually is an interest-- it's not just a curiosity, it actually has relevance to strategies for solving P versus NP. Hopefully I'll be able to have time to get to.

Can we think of an oracle like a hash table? I think hashing is somehow different in spirit. I understand there's some similarity there, but I don't see the-- hashing is a way of finding sort of a short name for objects, which has a variety of different purposes why you might want to do that. So I don't really think it's the same. Let's see, an oracle question, OK, let's see.

How do we use SAT oracle to solve whether two formulas are equivalent? OK, this is getting back to this point. How can we use a SAT oracle to solve whether two formulas are equivalent? Well, we can use a SAT oracle to solve any NP problem, because it's reducible to SAT. In other words-- P with a SAT oracle contains all of NP, so you have to make sure you understand that part.

If you have the clique problem, you can reduce. If I give you a clique problem, which is an NP problem, and I want to use the oracle to test if the formula-- if the graph has a clique, I reduce that problem to a SAT problem using the Cook-Levin theorem. And knowing that a clique of a certain size is going to correspond to having a formula which is satisfiable, now I can ask the oracle. And if I can do NP problems, I can do co-NP problems, because P is a deterministic class. Even though it has an oracle, it's still deterministic. It can invert the answer. Something that non-deterministic machines cannot necessarily do. So I don't know, maybe that's-- let's move on.

So there's an oracle where P to the A equals NP to the A, which kind of seems kind of amazing at some level. Because here's an oracle where the non-determinism-- if I give you that oracle, non-determinism doesn't help.

And it's actually a language we've seen. It's TQBF. Why is that? Well, here's the whole proof.

If I have NP with a TQBF oracle. Let's just check each of these steps. I claim I can do that with a non-deterministic polynomial space machine, which no longer has an oracle. The reason is that, if I have polynomial space, I can answer questions about TQBF without needing an oracle. I have enough space just to answer the question directly myself.

And I use my non-determinism here to simulate the non-determinism of the NP machine. So every time the NP machine branches non-deterministically, so do I. Every time one of those branches asks the oracle a TQBF question, I just do my polynomial space algorithm to solve that question myself.

But now NPSPACE equals PSPACE by Savitch's theorem. And because TQBF is PSPACE complete, for the very same reason that a SAT oracle allows me to do every NP problem, a TQBF problem allows me to do every PSPACE problem. And so I get NP contained within P for a TQBF oracle. And of course, you get the containment the other way immediately. So they're equal.

What does that have to do with-- somebody said-- well, I'll just-- I don't want to run out of time. So I'll take any questions at the end.

What does this got to do with the P versus NP problem? OK, so this is a very interesting connection. Remember, we just showed through a combination of today's lecture and yesterday's lecture, and I guess Thursday's lecture, that this problem, this equivalence problem, is not in PSPACE, and therefore it's not in P, and therefore it's intractable. That's what we just did. We showed it's complete for a class, which is provably outside of P, provably bigger than P.

That's the kind of thing we would like to be able to do to separate P and NP. We would like to show that some other problem is not in P. Some other problem is intractable. Namely, SAT. If we could do SAT, then we're good. We've solved P and NP. So we already have an example of being able to do that.

Could we use the same method? Which is something people did try to do many years ago, to show that SAT is not in P. So what is that method really? The guts of that method really comes from the hierarchy there. That's where you were actually proving problems that are hard. You're getting this problem with through the hierarchy construction that's provably outside of PSPACE. And outside of P.

That's a diagonalization. And if you look carefully at what's going on there-- so we're going to say, no, we're not going to be able to solve SAT, show SAT's outside of P in the same way. And the reason is, suppose we could. Well the hierarchy theorems are proved by diagonalization.

What I mean by that is that in the hierarchy theorem, there's a machine D, which is simulating some other machine, M. To remember what's going on there, remember that we made a machine which is going to make its language different from the language of every machine that's running with less space or with less time. That's how D was defined. It wants to make sure its language cannot be done in less space. So it makes sure that its language is different. It simulates the machines that use less space, and does something different from what they do.

Well, that simulation-- if we had an oracle, if we're trying to show that if we provide both a simulator and the machine being simulated with the same oracle, the simulation still works. Every time the machine you're simulating asks a question, the simulator has the same oracle so it can also ask the same question, and can still do the simulation.

So in other words, if you could prove P different from NP using basically a simulation, which is what a diagonalization is, then you would be able to prove that P is different from NP for every oracle. So if you can prove P different from NP by a diagonalization, that would also immediately prove that P is different from NP for every oracle. Because the argument is transparent to the oracle. If you just put the oracle down, everything still works.

But-- here is the big but, it can't be that-- we know that P A is-- we know this is false. We just exhibit an oracle for which they're equal. It's not the case that P is different from NP for every oracle. Sometimes they're equal, for some oracles.

So something that's just basically a very straightforward diagonalization, something that's at its core is a diagonalization, is not going to be enough to solve P and NP. Because otherwise it would prove that they're different for every oracle. And sometimes they're not different, for some oracles.

That's an important insight for what kind of a method will not be adequate to prove P different from NP. And this comes up all the time. People who propose hypothetical solutions that they're trying to show P different from NP. One of the very first things people ask is, well, would that argument still work if you put an oracle there. Often it does, which points out there was a flaw.

Anyway, last check in. So this is just a little test of your knowledge about oracles. Why don't we-- in our remaining minute here. Let's say 30 seconds. And then we'll do a wrap on this, and I'll point out which ones are right. Oh boy, we're all over the place on this one. You're liking them all. Well, I guess the ones that are false are lagging slightly.

OK, let's conclude. Did I give you enough time there? Share results.

So, in fact-- Yeah, so having an oracle for the complement is the same as having an oracle. So this is certainly true. NP SAT equal coNP SAT, we have no reason to believe that would be true, and we don't know it to be true. So B is not a good choice and that's the laggard here. MIN-FORMULA, well, is in PSPACE, and anything in PSPACE is reducible to TQBF, so this is certainly true. And same thing for NP with TQBF and coNP with TQBF. Once you have TQBF, you're going to get all of PSPACE. And as we pointed out, this is going to be equal as well.

So why don't we end here. And I think that's my last slide. Oh no, there's my summary here. This is what we've done. And I will send you all off on your way.

How does the interaction between the Turing machine and the oracle look? Yeah, I didn't define exactly how the machine interacts with an oracle. You can imagine having a separate tape where it writes the oracle question and then goes into a special query state. You can formalize it however you like. They're all going to be-- any reasonable way of formalizing it is going to come up with the same notion in the end.

It does show that P with a TQBF oracle equals PSPACE. Yes, that is correct. Good point.

Why do we need the oracle to be TQBF? Wouldn't SAT also work because it could solve any NP problem? So you're asking, does P with a SAT oracle equal NP with a SAT oracle? Not known. And believed not to be true, but we don't have a compelling reason for that. No one has any idea how to do that.

Because, for example, we showed the complement of MIN-FORMULA is in NP with a SAT oracle. But no one knows how to do-- because there's sort of two levels of non-determinism there. There's guessing the smaller formula, and then guessing again to check the equivalence. And they really can't be combined, because one of them is sort of an exist type guessing, the other one is kind of a for all type guessing. No one knows how to do that in polynomial time with a SAT oracle. Generally believed that they're not the same.

In the check-in, why was B false? B is the same question. Does NP with a SAT oracle equal coNP with a SAT oracle? I'm not saying it's false, it's just not known to be true. It doesn't follow from anything that we've shown so far.

And I think that would be something that-- well, I guess it doesn't immediately imply any famous open problem. I wouldn't necessarily expect you to know that it's an unsolved problem, but it is.

Could we have oracles for undecidable language? Absolutely. Would it be helpful? Well, if you're trying to solve an undecidable problem, it would be helpful. But people do study that. In fact, the original concept of oracles was presented, was derived in the computability theory.

And a side note, you can talk about reducibility. No, I don't want to even go there. Too complicated.

What is not known to be true? What is not known to be true is that NP with a SAT oracle equals coNP with a SAT oracle, or equals P with a SAT oracle. Nothing is known except the obvious relations among those. Those are all unknown, and just not known to be true or false.

Is NP with a SAT oracle equal to NP? Probably not. NP with a SAT oracle, for one thing, contains coNP. Because it's even more powerful. We pointed out that P with a SAT oracle contains coNP. And so NP with a SAT oracle is going to be at least as good. And so it's going to contain coNP. And so, probably not going to be equal to NP unless shockingly unexpected things happen in our complexity world.