[SQUEAKING][RUSTLING][CLICKING]

MICHAEL SIPSER: So we are-- welcome back, everybody. And we are going to continue our discussion of computability theory, Turing machines, and how to prove things undecidable, which is what we've been doing. So we talked about this more advanced method of proving things undecidable last lecture, called the computation history method, which comes up in all sorts of proofs of undecidability, usually more complex ones.

Such as, for example, the proof of Hilbert's 10th problem that I mentioned, whether you want to-- if you want to test whether a polynomial has a solution in integers. It's a reduction from ATM, just like we've been doing all along. All of those proofs are pretty much reductional from something undecidable. This reduction from ATM is as good a starting point as any.

And it uses the computation history method. So what they end up doing is, given a Turing machine and an input, you construct a polynomial that has several variables. And where in order to get an integer root, an integer solution of that polynomial, one of the variables is going to have to be assigned to some kind of encoding of a computation history of the Turing machine of M on w. One of those variables is going to be a computation history-- an integer which represents the computation history for m on W.

And the other variables are there to help you kind of decode that so that the polynomial can actually check and make a solution. It becomes a solution if that actually is a legitimate computation history of m on W. So it really uses the very same method that we've been using all along, but it's pretty hairy to construct that polynomial and do the check in the way that you need to do.

So for the Post Correspondence problem which we introduced last time, doing the check is relatively simple. You know that the match is the computation history, and following the rules of the match, it's fairly simple to construct that Post Correspondence problem instance. We talked about linearly bounded automata. Of course, we defined configurations and computation histories along the way and proved certain problem-- other problems are undecidable as well using the same method.

OK, so today, we're going to shift gears. We're going to-- in our last lecture on the computability section of the course, we're going to talk about something called the recursion theorem, which basically gives Turing machines the ability to refer to themselves. Turing machines in any program, to do self-reference so that you can actually get at the code of the Turing machine or the code of the program that you're writing. Even if that's not a built-in primitive of the programming language or the operating system that you're working with, it still gives you that access.

And also, we're going to-- if we have time at the end, I'm going to talk a little bit about mathematical logic, which is sort of a nice application of the recursion theorem. And it's a beautiful subject on its own. And it's something that I can give a brief introduction to.

OK, so today's topic is about self-reference, self-reproducing machines, and the broader topic called the recursion theorem. So let me introduce it with what I would call the self-reproduction paradox. And that is, suppose you have a factory, like a Tesla effect or a car manufacturing factory. See, there's a picture of the factory, and it's producing cars.

All right? So we have a factory that makes cars. And what can we say about the relative complexity of the cars compared with the factory, in some informal sense? So I would argue that you would be reasonable to say that the complexity of the factory is going to have to be greater than the complexity of the cars that it makes.

Because not only does the factory have to know how to make the cars, so it has to have all the instructions and whatever things that go into a car, it has to be included in at least some kind of-- it has to be, in some sense, represented in the factory. But the factory also has to have other stuff-- the robots, and the other manufacturing items, tools, and so on-- for making the cars. So the factory has to have all the complexity of a car incorporated plus other things as well. And for that reason, one could imagine that the factory's complexity is more than the car's complexity.

But now, suppose you want to have a factory that makes factories-- so imagine here's the picture-- or in general, a machine that makes copies of itself. Well, that seems, at first glance, to be impossible. Because not only does the factory obviously have to have all of the instructions for what a factory is like, but it needs to have all of the extra things that it would need to do the manufacturing.

And so for that reason, it seems like it's not possible to have a machine make copies of itself. I mean, you would run into the very same problem if I asked you to produce a program in your favorite language that prints out itself-- an exact copy of the same code. You can always write a program which is going to print out some string, like Hello, world. That's easy because you just put Hello, world into some kind of a variable or some sort of a table into the program and say print that table.

But if you want the program to print out a copy of itself, you can't take the whole program and stick that into a table because the program is going to have to be bigger than the table. And so, you're going to end up with something impossible happening. Because the program-- an entire copy of the program can't fit inside the program. You just get the program inside itself, inside itself, inside itself, forever. And so, you end up with an infinite program that way.

So if you just kind of naively approach the problem for how to make a program which is going to print out a copy of itself, it's not so easy to do. But hopefully, after today's lecture, you will see that it is possible and in fact, how to do it. And not only that is an idle bit of curiosity, but there are actually applications for why you might want to do that, mainly within mathematics and in computer science theory. But there's even a kind of a real-world application, if you will, in a way too. So we'll get to that at the end.

So it seems, as I'm saying, impossible to have a self-reproducing machine. But we know that in the world, there are things that make copies of themselves-- living things. So it seems like a paradox. Cells can make copies exactly of themselves. All living things can make copies of themselves.

So how do they manage to get around this paradox? Well, in fact, it is no paradox because it is possible to make a machine that self-reproduces, that makes copies of itself. And this has been known for many years. Probably, it goes back to Von Neumann who wrote a famous paper on self-reproducing machines. OK, so self-reproducing machines are, in fact, possible.

So let me give you an example of how you would make a self-reproducing Turing machine. What do we mean by that? I mean a machine-- I'm going to call it SELF-- which ignores its input. So on any input, you turn it on, the machine grinds around for a while, and halts with a description of itself on the tape-- with the description of SELF, its own code, sitting on the tape. So very much like producing a program which would print out its own code, that's really what we're doing.

So for that, we're going to first need a little lemma, which is a very simple lemma, but it looks worse than it is. So let me just read it out to you, and then I'll explain what its saying. Because what it's saying is extremely simple.

So there's a computable function, I'm going to call it q, that maps strings to strings, which will take any string, w, and produce from w a Turing machine which will print w. OK? That's all it does.

So as you know, if I give you a string, w, you could produce a Turing machine which would have w represented in the states and transitions of the machine. So that if you turn the machine on, the machine will output w. If I want you to give me a Turing machine that prints the string 1, 1, 0, 1 on the tape, you could do that-- I hope.

And no matter what that string was, instead of 1, 1, 0, 0, or whatever, it's 20 0's and then five 1's, you could do that too. And in fact, there's a simple procedure that takes a string and maps that onto a Turing machine which prints out that string. So that's a computable function, which basically takes a string and converts it to something that evaluates to that string.

And I'm calling it q. I don't know if this is helpful to you or not. It's kind of like it converts the string w to w in quotes. So q stands for quotes, in a way. So if that's helpful, then good.

But anyway, Pw is a Turing machine. When you turn it on, it just prints out w and halts. And I can find Pw from w-- straightforward proof.

So now, I'm going to tell you, assuming that we have that computable function, q, I'm going to tell you how to make this machine SELF. And it's not complicated.

The Turing machine SELF is going to have two parts I'll call A and B. Here's a schematic for the machine. So here's SELF. It's in two parts A and B.

And what I mean by A and B, it's like two separate teams. SELF is going to start out running A, and when A is finished, it's going to pass control to B. And then B is going to finish the job. And when it's done, you're going to have the description SELF sitting here on the tape. All right?

So what's left is to give you the code for A and for B. So A is going to be something super simple. A is going to be that machine which prints out B-- prints out a description of B. The one that I described up here.

So remember, Pw is the machine which prints out w. And P sub the description of B is simply going to machine that has this string, PB, stored in its states and transitions. You turn on that machine, and it prints out B, and then it's done. So this is a very simple-- A is very simple.

So here, PB is a part of A. And when it's done, B appears on the tape. So that's at the point when A has finished its work. Now it's going to pass control to B.

So we're not obviously done yet, because what we want is A and B both to be on the tape, not just B. Because SELF is a combination of A and B together. So I have to tell you how B works to finish the job.

So you might think, as a first, given what we did for getting B on the tape, that we'll get A on the tape in the same way, by putting a copy of A inside B. So a copy of B is inside A, and a copy of A is inside B.

And at some conceptual level, that seems like that might do the job. But it's really not-- that is not a solution. Because the fact that I can put B inside A kind of forbids me from also putting A inside P, because that's going to be the same kind of circular reasoning of just putting a machine inside itself. You just can't do that because you're going to end up with an infinitely big machine that way.

In fact, if I'm putting B inside A-- a copy of B in terms of its description inside A, A is really going to be much bigger than B. Because it has all of B with inside it, plus other stuff-- all the states and transitions for printing that B out. So I can't have the A be much bigger than B and then B also much bigger than A.

So this is no good. We're going to have to do something else. So how is B now going to get ahold of A? And the trick for doing that, without having a copy of A inside B-- which doesn't work. That's not going to be a good solution. Instead, the way that B is going to get A, it's going to figure out what A is. It's going to figure out what A is.

And how is it going to figure out what A is? Because if you remember, B can now look can look at the tape. It sees some string there which happens to be a description of itself, but it doesn't care. It sees some string on the tape. A is the machine that prints out that string.

A is q of this string. So B is simply going to compute q of whatever it sees on the tape. That is A. OK? So I don't know if you can read. That's kind of small here. It's going to compute A from B sitting on the tape. So here is the instructions for B.

It's going to compute q of the tape contents, which happens to be the description of B. But that's irrelevant to B. B just sees some string on the tape. It computes q of that, and that is A. Because A is the machine which prints out that string.

Then it's going to combine A with B, doing whatever slight interfacing that needs to happen-- I'm not going to get into those details-- to convert those two pieces into one machine, which is the machine SELF. And then it's going to print out with SELF on the tape, as I'm going to indicate over here.

OK? So that's how a Turing machine can print out a copy of its own description and leave it on the tape. And what's nice about this is nothing specific about Turing machines. This is a general procedure that allows any programming language to print out a copy of its own code. You can even carry this out in English, as I'm going to do in the next slide.

OK, so here's a good question. There are many possible Turing machines that can print out B. That's right. How do I know how to get the particular one? What I have in mind, that's a little bit of a subtle question, but it's a good question.

I have in mind the particular Turing machine that prints out B, which is the one that q produces. Remember, so we have to refer back to this lemma. This lemma produces a particular machine that prints out B from B.

And that's the one that I'm going to use for A, and that's the one that B is going to be able to obtain by running the q algorithm to figure out what A is. So you have to make sure you're being consistent then. That's a little bit of a detail, but it's a good question.

And why doesn't this create a circular argument too? Well, so that was another question I'm seeing here on the tape. Well, there's no longer anything-- see, B does not have to have A stored within it. It figures out A.

So in a sense, you're going to write the code for B first. B is just a simple program. Here it is. There's nothing circular about it. It says B is a simple-- the code for B is, take a look at the tape, compute q of that, combine the result with whatever was sitting on the tape from before, and print it out. I mean, that's a piece of code which you can just write.

This will become more clear, hopefully, in our next slide where we talk about the English implementation. But just, I don't want to rush to that. So you could figure out B without even knowing what A is. B stands alone.

But then, because B is just a piece of code that runs q based on what it sees on the tape. A, now you need to know what B is in order to obtain A. Because A has the code for B embedded within A as a string.

So first you produce B, then you can figure out-- then you can obtain A. There's nothing circular in this argument. I don't know if that's helpful to you, but you may need to mull it over. Or maybe it'll be helpful from the next slide. So let's go there now.

So as I'm saying, you can implement this in any language. In particular, you can implement it in English. So let's just shift gears. Let's talk about writing down English instructions. And then I'll show what happens if you carry out those instructions.

So let's start simple. How about the sentence, Write "Hello World." So an obedient person reading those instructions would then write "Hello World" on their paper or wherever. OK, fine, hopefully you get the idea.

So now, what I'd like-- here's another sentence, another instruction. Write this sentence. And the obedient reader would then, OK, Write this sentence. This is the kind of thing I'm looking for. Here is a sentence which instructs the reader to make an exact copy.

But I don't like this one, even though it does, in a sense, what I'm looking for. Because it kind of cheats. When it has this, this refers to the entire sentence itself. It's using kind of implicitly here a construct which a Turing machine does not have.

The Turing machine does not have a way of accessing its own code. And in fact, really what the point of this whole theorem that we're going to present is, is that you don't need to have this self-reference as a primitive. You can get that effectively using the procedure that I'm describing, which will give you the same effect. So you don't need it as a primitive. You can design some software basically, which will give you the same effect.

So let me show you how to do that in English. So let's look at a slightly more-- whoops, cheating, so Turing machines don't have this self-reference primitive. So let's look at another sentence here. Write the following twice, the second time in quotes, and then, "Hello World."

So what do we get if we follow that instruction? Well, you get "Hello World" and then "Hello World" again, now in quotes. OK? Hopefully not too bad.

But now, I'm one step away for doing the implementation of the self-reproducing algorithm in English. Write the following twice, the second time in quotes, and then in quotes, the same text. Now following those instructions, you get, well, "Write the following twice, the second time in quotes," so that comes out here.

And the second time, you put it in quotes, just like you did with the "Hello World." But that's exactly here, the output is exactly what the input was. And even though there is a part of the sentence here which refers to a different part, so here the first part is referring to the second part, never do I have to have the sentence referring to its entirety. There's no part of the sentence here that's pointing at the entire sentence.

And so, this here manages to get the effect that I'm looking for, where the output is equal to the code, but avoids having the self-reference. One thing is referring to something else, but not referring back to itself. So let me just see here.

So here, I'm going to have a check-in on this in a minute. So I'm going to try to contrast what's happening here with what's happening in that Turing machine SELF that I had from the previous slide. So why don't you mull this over, and I'm going to give you a check-in to see-- this is a little bit of a challenging check-in. But let's see if you can figure your way through it.

And basically it says, so really what we're doing here is called the recursion theorem, as you'll see. We'll actually present the recursion theorem formally on the next slide. But here, in both of these cases, we kind of have a template part and an action part. In both cases, there are two parts to the instructions, the template and the action.

OK? So I'm going to leave it to you to try to imagine which of those is which, in each of these two examples. And then, I'm going to ask you to pick. In the Turing machine, which is the action and which is the template, and in the sentence, which is the action and which is the template.

The action is the part where there's some interesting sort of instructional stuff happening that you have to carry out. The template is really basically just text or just a string. So let me pull up that poll. See what you think. Because I'm asking you now to indicate where is the action part in both of those cases.

What is the upper phrase and lower phrase? I mean, of this sentence here. So write the following twice. This is the upper phrase. And the part in quotations is the lower phrase-- sorry.

OK, almost done here? 5 seconds-- a few of you have not answered yet. Answer it. One second to go. OK, here we go, ending polling.

So the majority here is correct. I would say, in the English sentence here, the action part is the first part. That's where you actually have been directed to do something. The second half of the sentence, the lower part of the sentence, is just a template written.

This is just some string here. There's no action really being directed. It happens to be the same as the top, but this could have been just Hello World. This could have been anything.

And then the upper part acts on that. So the upper part is the action part. So it's the upper phrase that's the relevant part.

Now, in the Turing machine, in a sense, it's the other way around. The first part is really just the template. The second part, B, is where you're doing some actual work on the template.

You're taking that, basically text, which could be anything. A could be anything. And you're looking at that template and reconstructing what A was from that string that appears on the tape. So B is actually the one that's doing the work. So it's B and the upper phrase with part c is correct.

So let us continue then. Oh, I want to mention here problem 6 on the Pset. So your job really is to implement this in-- if you have a programming language that you like-- it could be Python or whatever your favorite Java, whatever you like-- you can implement this. If you don't know any programming languages, then just make up some sort of pseudo programming language and implement it there.

Let me point out that getting the quoting right is a bit of a pain because you have to kind of escape the quotes and so on. I'm not going to be fussy about that. So you can still get full credit even if you don't get the quoting part quite correct. Do your best.

I think it's an interesting problem to try to solve. And if you struggle with it for a while, it's slippery. It's the kind of thing you can easily spend a couple of hours on this problem. Because it's a bit tricky to manage to make a program which prints itself out, which is what the task of problem is on the Pset.

But don't fuss about too much on the quoting if that's the only thing that's hanging you up. Try to get the main structure of it, which is fairly simple, actually. And if you can't get the quoting part to work, I'll ask the graders not to penalize you for that part.

Let's look at the sort of the more formal version of what we've just done and really kind of also taking it to the next level. Because being able to print out a copy of yourself is kind of a curiosity in a way. But the recursion theorem says, not only can you make a machine which prints out a copy of itself, but you can actually make a machine which can obtain a copy of itself and then do some computation on that copy of itself, on its own description, which actually turns out to be a useful thing to do. Once you have access to your own description, as you'll see from some examples, then you can do perhaps some interesting things with that.

So basically, what the recursion theorem is, is a kind of a compiler. It allows you to have a new primitive when you're writing Turing machines, which is, compute your own description. And the recursion theorem will implement that for you. OK?

So the technical form of the recursion theorem is going to look a little bit counterintuitive, perhaps. Let me put it out there. If you struggle a little bit with the slide, don't sweat it. The main thing to remember, and we'll see from examples, is that you can compute your own description in a Turing machine. And that's going to be allowed as code.

So the way we're going to do this is, what the recursion theorem does for you is, it says you can write a piece of code here. Let's call it a piece of Turing machine code-- algorithmic code-- called T. And T is going to get transformed into a new machine, R. And R is going to get provided a copy of the program itself, which is just a description of R, for free.

But otherwise, it's going to act exactly like T. So R is going to act exactly like T, except R is going to have provided a copy of R. And that's what the theorem shows you how to implement.

So let me just see. So for any machine, T, there is a machine R which, for any w, which is going to be the input for R. R, on w, operates the same as R on the input with w where it's given R. So R is going to be getting access to R without-- it's going to be obtaining R by calculating it. Maybe it'll be clear from the proof. I always struggled with how to explain this clearly.

So now, the proof of this is going to be very much like the proof from two slides back, except it's going to be three parts. This is the part, T, that you're going to provide. And the T is going to be the Turing machine code that says, get your own description. And you don't have to worry about how that happens.

The compiler is going to add on the A and B parts, which is going to get the whole description of the whole thing, which is R, and feed it into T as if T had it as an input. So T is going to be allowed to get its own description now and operate on-- now does its thing on the input w.

So the way it's going to work, so T is given. A is going to be, as before, the description of B now with T. So when A is done, it's going to produce BT sitting on the tape next to the w.

B is going to now figure out what A was from the BT sitting on the tape. And then, it's going to combine that to get ABT, which is R. And after that, it passes control to T with w and now R sitting on the tape.

And now, T is going to have its own description. But don't forget, now T has been modified to be R. So it's not that T is going to get T on the tape. For this to make sense, this is going to be now the new machine R. And R appears on the tape. And now, the code that you provided, T, is now going to get to operate on that. OK?

If you didn't get that, I'm not so worried. The main thing is that you can use compute your own description when you're describing Turing machines. That's what this thing is telling you. I think it'll be-- oh, there's a check-in here.

Yeah, so I don't know. Let's do this one kind of quickly here. Can we use the recursion theorem to design a machine T which, instead of producing its own description, accepts only its own description as an input? So the language of this machine is going to be simply the one string, the description of T. So can we make a machine, T, which does this?

Now I'm looking at this check-in. This T here is confusing with that T. It's not the same T. That's bad. I should call it M. Design a machine, M, where L of M is the description of M.

And can we use the recursion theorem to do that? What I would ask you to do is think about it in this context. You can use compute your own description when you're writing the code for this machine. If you could do that, could you make a machine which just accepts strings which happen to be their own description? This is supposed to be easy. But I think it ended up being a little bit more complicated than I wanted.

Launch polling, make your best guess. I think you all kind of see-- I was kind of leading you, leading you along the path here. Yes so, I think you're pretty much all are getting it. Maybe a few of you are unsure. But anyway, let's just wrap this one up quickly so that we can move on. I think you're pretty much-- so, 5 seconds, I'm going to end it.

So the correct answer is Yes, as I was hinting at. So I think maybe this example would have been better after the next example. And then we're going to have a break. So here is a new proof that ATM is undecidable but now using the recursion theorem. And this is going to give you a nice example of how we use the recursion theorem in action.

So remember, we spent half a lecture or more with a proof by diagonalization. As our first example of an undecidable problem ATM, we subsequently showed other things undecidable by reduction. But for the very first example, we used that diagonalization. Now I'm going to give you a new proof.

So proof by contradiction, assume we have a Turing machine, H, that decides ATM. It starts the same way that the diagonalization proof went. But now I'm going to make a new machine called R. So this is going to be different from the earlier proof.

R says, on input w, I get my own description. Use the recursion theorem. That's the way these things always start.

Now, I'm going to use H. Now that I know my own description, I'm going to feed-- I can ask H. I can feed R, w-- w was the input here. I can feed R, w into H to determine whether R accepts w. That's what H does. Solving ATM, H will tell this machine whether R accepts w.

R is the machine we're writing, however. That is the machine that's currently running. So R now uses H, and it knows what it's supposed to do. H is going to say, well, you're going to accept w or you're not going to accept w. That's what H is assumed to be able to do.

But then what is R going to do after that? R is going to do the opposite of what H says it's going to do. So if H says, R accepts w, then R is going to reject w. If H says R doesn't accept w-- it rejects it by looping or halting, doesn't matter-- H just says it rejects, then it's just going to accept.

So whatever H says, R is going to show that H is wrong. So that's a contradiction. It says that H cannot be deciding ATM.

So if you step back and think about what this is here, that's that whole diagonalization proof in one line. Basically, we've done that proof in a different way, though there is some similarity here. I don't want to say that we've totally reinvented things and done that totally differently.

But it's kind of, in some ways, sort of gets at the essence of the diagonalization in a certain sense. But anyway, it gives you kind of a new, very short proof that ATM is undecidable. I think that's kind of a cool thing.

So why don't we take our little coffee break here, and you can feel free to ask questions during the break. I will start my timer going. And we'll be back continuing lecture in five minutes.

OK, so questions-- so we're getting some questions on when I said, we don't have to worry about the quotes when we're solving problem 6 on the Pset. Somebody says, can we just say print A, B, C, instead of print quote, "A, B, C," and it will print A, B, C. Yes, you can-- don't worry about the quotes.

I think it's kind of-- you'll see a challenge if you want to try to get the quotes to work. But it's also kind of a pain. So yeah, you can just kind of ignore the quotes, and I'll ask the graders to give that full marks there. So any reasonable interpretation will-- as long as you get the rest of the concept right-- will be fine.

Let's see, somebody is asking, In the recursion theorem, why doesn't T get the description of T instead of the description of R? Because the machine that's running is R, back in the previous slide, two slides ago. So if R got T, it would not be getting its own description. It would be getting the description of some other machine. So you need to think about what we actually need to have happen here in the proof of the recursion theorem. But it needs to have R, not T.

Let's just see if I can-- why does R do the opposite of what H says? Why does R do the opposite of what H says? Well, first of all, I'm the one who gets to design R. So here is the code arc.

We're assuming we have H. I'm going to design R to do anything I want to satisfy the proof. And R here is designed to do the opposite of H. So I'm asking R-- I'm programming R to find out what H predicts it will do and then do the opposite.

Maybe the situation is sort of like this. Suppose somebody says, I have a crystal ball, which is going to be like the role of H. And you say, oh, really? That's kind of cool. I don't believe you, but it still sounds interesting.

And the person says, yeah, I can see the future. I know what you're going to do in five minutes. And in fact, I can see that in five minutes, you're going to say, Hello. And you can say, well, you can think to yourself, well, this person is nuts. I'm not going to do that. I'm just not going to say Hello.

And then the genie there with the crystal ball waits five minutes and you don't say Hello. And then you've proven that the crystal ball doesn't work. It's very similar.

I'm not going to explain how we can do the combining in SELF. I just want to explain at a high level. That's just going to be messy. Because I said, we were somehow going to combine in SELF. Let me leave that as a conceptual level.

OK, how does this idea work for Turing machines? I don't really understand that question. How does this idea work for Turing machines are decidable? You'll have to resend that because I don't understand the question.

Can I explain, use your own description? So when you write code that says, get your own description, after that code executes, the Turing machine appears on the tape, like magic, the description of its own code. Say, sitting next to the input to the machine, because the machine may have an separate input from that. So the machine just magically gets its own code. And the proof of the recursion theorem implements that, so it's not magic after all.

But I still don't understand the question. So for problem 6, is it enough to attach code? If you're going to attach code for problem 6, and that's good enough. Or if you can explain, it's also good, like usual.

Do we worry about tabs and new lines? No. OK, we're going to have to defer the rest of the questions. Don't forget we have a zillion questions here which I didn't get to.

OK, last one here-- why does programming R do the opposite of H? Is that a contradiction? Well, H is predicting that R accepts, but R doesn't accept, so H is wrong. We're a little short on time. Let me skip this one. You can look at this on your own.

I mean, this is proving sort of the cool fact that-- I'll just say it at a high level. If you have a program transformation, so if I have some method of transforming one program to another program, but it's done by algorithm. So an algorithmic way that transforms one program to another program. There's always going to be some program whose behavior is unchanged by the transformation.

That's called the fixed point theorem. So there some program whose behavior doesn't change no matter how you try two transform programs-- easy proof using the recursion theorem. You can look at the slide offline separately if you like to see how that goes. It's pretty simple.

Here's another exercise of the recursion theorem. So if I have a-- let's say a Turing machine is minimal if its description is the shortest among all Turing machines which behave the way it does, which are equivalent to it.

When I was an undergraduate, I took a programming class. And some of us sort of enjoyed writing short programs to carry out the exercises. Probably these days, that's forbidden because it just encourages bad programming style.

But anyway, so you kind of won if you found the shortest solution for a given programming exercise. It was Heap Sort, I remember, was one of the ones that we had to do. So this is sort of similar. You might imagine if people try to find the shortest possible universal Turing machine.

So short is, in our sense is, in terms of whatever encoding method we have in mind, a machine is minimal if there is no shorter program which is equivalent using our encoding system, whatever it is. So M is minimal if anything that has a shorter description has a different language.

OK, so let's look at the collection of all descriptions of minimal Turing machines. And I want to prove that that language is unrecognizable. I'm going to do it using the recursion theorem. And it's kind of a cool exercise. And you can actually use it to prove something more powerful. But let's focus on this theorem for now.

So assume we have-- and it's also in the little nice exercise about enumerators. I don't know how comfortable you ever got with enumerators. But I'm trying to prove this language here is not recognizable. And so remember, enumerators, you can enumerate exactly all the recognizable languages.

So I'm going to assume I have an enumerator for this language, which just prints out all of the minimal Turing machines. So I have some enumerator. It's a program that prints out the descriptions of all of the minimal-- the shortest possible Turing machines.

And now I'm going to get a contradiction. So we're going to build this Turing machine, R, which gets its own description. And then, it's going to start the enumerator until-- so looking at the strings that the enumerator produces. So this enumerator is producing these minimal Turing machines, one after the next-- chunk, chunk, chunk, chunk, chunk. All these minimal Turing machines are coming out.

And you keep looking at those until you find one of them which is bigger than yourself. And how do you know how big you are yourself? From the description you're given by the recursion theorem. So you keep on printing out these Turing machines until you find one that's bigger than yourself.

And then, what do you do with that? You simulate that. So now, so what? Well, the point is that you're going to be smaller than that machine that you're simulating. Because you waited to find a machine that the enumerator produced which is bigger than you. So you're going to be simulating that machine that's bigger than you.

And so, you're going to be doing exactly what that machine does on every input, because you're always going to be simulating that same machine on every input. And so, you're going to be equivalent to that bigger machine. But that bigger machine is supposed to be minimal because E is producing it. But here, you are exhibiting a machine that's smaller than that-- that allegedly minimal machine couldn't be minimal. That's the contradiction.

So the language of R, this machine I just produced, equals the language of B. Because R ends up simulating B. But R is smaller than B because R waited until it found a bigger machine that's bigger than it. So B couldn't be minimal, but B was one of the machines that the enumerator produced. That's a contradiction.

So let me do a check-in on this. I expect this is going to cause some of you heartburn, but let's do the best you can. Suppose I have this collection of minimal Turing machines and I take some infinite subset of that. So now I'm not demanding that I have all of the minimal Turing machines. I just have infinitely many minimal Turing machines.

Is it possible that subset-- whatever it is-- could be Turing-recognizable? Now, think about that. Now, you can have languages which are not Turing-recognizable, that have infinite Turing-recognizable subsets. Could that be for this language?

And maybe I'll give you a-- it will be helpful to you to understand and perhaps apply the methodology that I gave you in this proof here. And that might be helpful to you. But I can see this is not-- this one is a bit of a struggle. OK, let's end it. Two seconds, just pick something.

So the majority has the correct answer here. So in fact, this proof would still work if the enumerator was enumerating an infinite subset of minimal Turing machines. Because all you need is to wait until one that's bigger than you appears. And all that R needs to do is wait until one that's bigger than R appears, which will certainly happen eventually if the subset is infinite. And then R simulates that bigger machine and acts the same way, thereby proving it could not be minimal.

So it's exactly the same proof shows that the answer here is No. And it's a kind of a curiosity. It's not necessarily that easy to construct languages which not only are they not recognizable, but they have no recognizable subsets-- infinite subsets. Obviously, a finite subset is going to be recognizable because it would be decidable as well.

So anyway, let's move on. Some other applications-- so first, a real-world application-- somebody is asking for an example of a language of a recognizable subset of a non-recognizable language. So starting out with something which is not recognizable, and can we come up with a quick--

So here is, I don't know if this will help you. So the question is, can I give an example of a non-recognizable language that has a recognizable subset? So I didn't prepare this, but let me see if this helps you.

So let's take ATM complement. We already showed that ATM complement is not recognizable. So these are the sets of pairs, M and w, where M does not accept w.

So if I focus only on those M's, which are finite automata, which are a subclass of Turing machines, then I can get the answer for those M's. So for the infinitely many cases where the Turing machine never writes on its tape, it even becomes decidable. So I don't know if that's helpful, but you can definitely find cases where there are undecidable languages that have decidable subsets, unrecognizable languages that have recognizable infinite subsets. But there's one example where it's not true, in the previous slide.

OK, a lot of questions are coming up here. Yes, I'm seeing some other proposals here. If you take ATM complement and you union with just any old strings of 1's, just one star, assuming that one star-- just strings of 1's are never going to code for a Turing machine-- that's still going to be unrecognizable. But then you can just throw away all of those just strings of 1's, and you're going to get an infinite subset. Oh, wait a minute. It's still unrecognizable. That's no good.

Oh, no, yeah, I threw away the wrong stuff. You throw away the descriptions of the Turing machines, and you just have the one-star strings left. And so, that becomes decidable, even regular. Anyway, I'm not sure I'm helping you.

Let's move on to other applications. So this is kind of a curious application that actually is in the real world, where a machine might want to get a copy of itself and then do something perhaps even nefarious with a copy of its own code. And that would be a computer virus.

Computer viruses make copies of themselves and then propagate them through the internet, or whatever media you have, to infect other computers. And I'm sure we all know computer viruses. Well, they need to get copies of themselves in order to do the infection.

How do they do that? Many of them operate in a way where in either in a language or in a system, where they can make reference to their own code, either by looking at the machine code, or whatever direct access to their own code. There are languages and systems which allow for that.

But I'm not an expert on computer viruses. I would be shocked if they're not some other viruses that get access to their own code by using something in the same-- using basically the method of the recursion theorem. I haven't done a systematic study. But I'm sure, if you can't access to your own code directly using some operating system mechanism, some primitive, the only other way is basically doing the method that we just described.

OK, so another application is in a branch of mathematics called mathematical logic. Where, I imagine that many of you have heard of the work of Godel from the earlier part of the 20th century, where they show that you can come up-- it's possible to demonstrate that there are true mathematical statements but which cannot be proven to be true. So proof-- there might be something like maybe even questions of interest to us, like P versus NP question, which we will, at some point, look at in a few weeks-- actually, yeah, maybe two or three weeks from now.

There are many unsolved mathematical problems. And people wonder, maybe there's just no way to prove them one way or the other. So in the 1930s, when Godel did his work, he shocked the mathematical community by showing that proof does not work for everything. There may be things that are true that you cannot prove.

In Hilbert, in particular, from Hilbert's 10th problem, he was dismayed by this result. Because he had earlier believed that anything that was true you could prove. So anyway, let's just see how-- I'm going to sketch how we actually go about doing that. Because we now have a kind of enough technique to at least give you an idea of how do you demonstrate that there are true but unprovable statements and actually even exhibit one.

OK, mathematical logic is the mathematical study of mathematics itself. Godel's first incompleteness theorem, as we described, is that in any reasonable formal system of mathematical provability, there are going to be some true statements that are not provable. And in order to sort of get the sketch of the proof-- I shouldn't say proof here. We're going to proof sketch. We're going to basically use two properties of formal proof systems.

One is that kind of obvious property that you would expect all probability systems to have, is that you can only prove true things. So if something is being proven, it's going to be true. You can't prove anything false. If you can prove false things, your system of provability is bad.

And the other thing is that proofs are checkable by machine. So if you write down-- do your system in a formal way and you have this formal notion of proofs, which underlies all mathematical reasoning, by the way. This is completely well accepted. Both of these properties are accepted by mathematicians. Then in principle, you can convert any mathematical proof into a form that you can check it by computer. It might become much longer. But in principle, you can put the proof into a form where a computer could check the proof.

And the way we're going to frame that in the way we've been talking about things is that the language of all pairs of proof, comma statement being proved-- so where pi is a proof of the statement phi-- that's a decidable thing. So you can check, by machine, whether pi is a valid proof of phi. So your proof checker can say, Yes, it's valid, No, not valid. And that's something you can do by algorithm. So those are the two assumptions that we're going to make about our system of proofs. And that's all we're going to need.

Now, the first conclusion, which is, I think, a good sort of a little bit of exercise on the kind of thinking we've been doing in this course. Number 2, checkability implies that the set of provable statements is recognizable. Why?

Suppose I give you a statement that has a proof-- a provable statement. I'm not saying it's a true statement, necessarily. That's going to be perhaps a larger class of statements. But the statement that do have proofs, that's a recognizable language.

Because, if I give you a statement, your recognizer is going to take that statement and start looking through all possible proofs. It's going to look at string after string as a candidate proof, one after the next. Some strings, of course-- most strings are going to be junk. But every once in a while, a proof is going to come out. It's going to be a string which is a valid proof of something. And then you're going to check, oh, let me just see if that's a valid proof.

And if it does prove the statement that I have in mind, and if it is, then I accept that statement. And it goes through statement by-- the input is a mathematical statement. And that's going to be accepted if the machine, by looking through all possible proofs, finds one and then it accepts that statement. So the collection of all statements that have a proof, that's recognizable.

So similarly, if you take statements of the form M and w is in the complement of ATM. So M doesn't accept w. If you take all statements of that form where M doesn't accept w, or M, w is in the complement of ATM, some of those statements may have proofs in your system. Some of them may not have proofs.

If all of them had proofs, if you can prove every statement of this form when it happens to be true-- obviously, you can't prove the ones that are not true. But if you can prove all the true statements of this kind, then ATM complement would be Turing-recognizable. Because you can go through, just for the same reason as above. But we know that that's false. So there must be some statement of this kind which is true but does not have a proof. Because otherwise, the ATM complement would be recognizable.

So we've actually done the first half of Godel's incompleteness theorem. The second half, which we're going to unfortunately not have time to finish, but let me just give you the outline, is that we can use the recursion theorem to give it specific-- see, what we showed here is that there is some statement of this form which is true but unprovable. It doesn't exhibit a particular one. Now the recursion theorem allows you to give a particular one.

And the one, it basically implements Godel's so-called Godel statement or Godel sense that says, "This statement is unprovable." And you can formalize that precisely. And then that statement becomes true but unprovable.

Let's just say why that is. Because if the statement were false, suppose the statement were false, then, well, then it would be provable. Because the truth says it's not provable. But if the statement were provable, then it had to be true, which would mean it would be unprovable.

So the only viable outcome here is that the statement is unprovable. And which is, it's therefore that it's true that it's unprovable. So that this is a true statement, but then it has no proof.

And let me not go through it because again, we're unfortunately running short on time. But you can implement this using the recursion theorem to make a particular machine, R, where you cannot prove that R does not accept, say, some string 0. So you can find a particular R using the recursion theorem, where it's impossible to prove that R doesn't accept 0. Even though, by construction, R does not accept 0.

So it's a little bit slippery there because you have to understand what we mean by proof within the formal system versus our external form of reasoning, but taking us a little bit far afield. So for those of you who care, I hope this little digression was interesting. As I mentioned at the beginning, for those of you who don't care, you don't have to worry about it. It's not going to be on the midterm or on the final exam. You're not going to be responsible for this last five minutes or so of the lecture. But I thought it's kind of an interesting application of the recursion theorem to a problem outside of computer science in mathematical logic.

OK, so here is the entire reasoning here again. I invite you to look at that on the slide that I posted if you're curious. So anyway, a quick review of today is, we went through self-reference and the recursion theorem. We gave a few applications. And we did a sketch of Godel's first incompleteness theorem in mathematical logic.

OK, so that's all I'm going to have for you today. We're out of time. And I will take any questions. So getting back to the MIN Turing machine example, somebody is asking, how do I know there's a Turing machine that's longer than the machine, R, that I'm building? Well, there are infinitely many machines in MIN TM or in the infinite subset of MIN TM. So eventually, one of them is going to have to be longer than the machine that I'm constructing. Because that's a machine of some very specific size, and so eventually there's going to have to be one that's bigger that shows up.

So this may be a similar question. Now, another similar question is about, how big is R? And does the size of R in that previous thing here-- I don't know if we want to go through this. But, OK, let's quickly look at it. This machine, R, has a fixed, predetermined size. Its size does not depend on B.

It depends on E because it's going to be simulating E. But E might be producing very, very long strings. Eventually, it will. So E is fixed. And then, R's size is fixed. So eventually, R will find a machine that's bigger than it is.

But let's look at-- right, so this is a good question. I was wondering if I would get questions of this kind. This is getting back to the question in logic here at the end. Yes, because I said this statement here is unprovable. But in a sense, I proved it. Because how do I know it's true?

And I gave you an argument for it. And so, you have to differentiate between the reasoning that we're providing and the formal system that we're reasoning about. And the formal system that we're reasoning about is not capable of proving this. But we're outside that formal system so that we can reason about the formal system.

I know it's a little bit perverse seeming. And mathematical logic is a little tricky because it has to deal with those kinds of issues. But this is arguing within any particular formal system of probability, this is going to be true. But that's kind of an approximation to our own thought process. So it's slippery, I agree.

So, a good question here, would Godel's theorem still hold informal systems where we don't require that proofs of statements are decidable? So I'm not saying that the proofs are decidable. That you can-- proofs are checkable. So you can test whether a proof is a proof.

If you can't test whether a proof is a proof, I don't know of any people who have studied that situation. So that's a little bit of a trickier case. I'm not sure what to say about that.

Can I give an example of two equivalent Turing machines where one has a shorter description than the other? How do we define the length of the description? Well, we never really precisely defined our encoding system. But whatever encoding system you want to use, is going to represent Turing machines as strings. And those strings are going to have a length. And so, it doesn't really matter which encoding system you're going to use because the statement isn't going to be true in any of them.

We could go through the exercise of defining a particular encoding system. It's going to be pretty tedious to do that. But you can just imagine writing down the states, the transition function, et cetera, et cetera, as this big long string. And that's going to be our encoding system.

And then there are going to be some long machines. Some machines will have long representations and other machines that have short representations. And there's going to be some machine where you can introduce a bunch of useless states that are never accessed. So you can expand-- you can kind of add junk to the description of the machine, which is not going to do anything. But it's just going to make the description of the machine unnecessarily long compared to what with some other description, which is going to compute the same thing but will be much shorter.

So you can certainly find examples of pairs of machines that do the same thing where one is much longer than the other. So I will then close the session here. And I will be very shortly on the office hours link and see some of you there.