[SQUEAKING] [RUSTLING] [CLICKING]

PROFESSOR: All righty, why don't we get started. So welcome back. Nice to see you all. And what have we been doing in theory of computation? We have been talking about Turing machines and about the power of Turing machines. We started at the beginning by showing a bunch of decidability theorems that exhibit the power of Turing machines to calculate properties of finite automata, context free grammars, and so on in some cases.

And last lecture, we talked about the limitations of the power of Turing machines by proving undecidability theorems. So we showed that this language A TM, the acceptance problem for Turing machines itself is an undecidable problem. That was the first of many undecidable problems that we're going to encounter.

And though we proved the undecidability of A TM using the diagonalization method as hopefully you remember, we're going to introduce a new method which we kind of basically previewed last time called the reducibility method, which is the way other problems are typically shown to be undecidable.

And so we're going to stick with that for this lecture and also next lecture. We're going to be talking about undecidability. And I think there's going to be a few additional discussions after that. But this is one of the important themes of the course is to understand that threshold between decidability and undecidability, or the limitations of computation, OK.

So today, as I mentioned, we're going to talk about the reducibility method for proving problems undecidable and also for proving problems non-turing recognizable, Turing unrecognizable. We're going to introduce this notion of a reducibility in general. And we'll also talk about a very specific kind of reducibility called the mapping reducibility.

So today as promised, we're going to talk about using reducibilities to prove problems are undecidable, or unrecognizable. So that's going to be our general method, oops, make myself smaller, thank you. I always forget. Thank you for the reminder.

So using reducibilities to prove problems are undecidable, or unrecognizable, and the basic way that works is we're going to leverage another some problem we already know is undecidable say, or unrecognizable to prove other problems are unrecognizable. So we did a quick example of that last time. We're going to go over that example again just to set the stage. And then we're going to talk about that in greater detail.

So as you recall from last time, we had this problem HALT TM, which is the problem of testing for a given Turing machine and an input to that Turing machine, whether the Turing machine halts, either accepting or rejecting, but just whether it halts. Which is a somewhat different problem, closely related obviously, but somewhat different than the A TM problem, which is just testing whether the Turing machine accepts. We already showed that A TM is undecidable.

Now, conceivably, HALT TM might be decidable. You know, it's not exactly the same problem. But we're going to show that HALT TM is likewise undecidable. We did this last time, but I'm just going over it again.

I'm going to likewise show that HALT TM is undecidable. We could go back to the diagonalization method and do it from scratch. But generally, that's not what's done. Generally what people do is they use a reducibility from a known undecidable problem.

And so what we're going to show is a proof by contradiction which says that if HALT TM were decidable, then A TM would also be decidable. And we know it isn't. And that's by virtue of what we call a reducibility from A TM to HALT TM. And I'll explain with the terminology. And we'll have a chance to play with the concept all lecture long. So we're going to see all sorts of different variations.

So as I said, we'll assume HALT TM is decidable and use that to show that A TM is decidable, which we know is not true. So quickly going through it because we did it already once before, we're going to assume that HALT TM is decidable. Let's say Turing machine R is the decider.

And now we're going to show that ATM is decidable by constructing a Turing machine S, which uses R to decide A TM. That's going to be our contradiction. So here is the machine S.

You have to keep in mind what the goal of S is. We're going to design S to solve A TM, which we know is not decidable. So don't get confused by that. We're aiming for a contradiction.

So we're going to use S as typically-- well, there might be other variations. But for now, S is going to be used to decide A TM. So we can try to figure out, how can we decide A TM.

And the way we're going to do it is use our HALT TM tester that we assumed to have. And we'll first take our M and w, where we're trying to determine, does M accept w? And we'll first test whether M halts on w. If it doesn't, we're done.

Because it couldn't be accepting w, M couldn't be accepting w if it's not even just halting on w. So if R reports doesn't hold, we can reject right off. But even if R says it does halt, we're still in good shape because now we can run M on w until completion because R has promised us that it's going to halt.

R is stated. And R is assumed to be correct. R is stated that M halts on w, so now we don't have to worry about getting into a loop, which we're not allowed to do since we're making a decider. We're trying to decide A TM here. But now we can run them on w to completion. We can find out what M does on w. And then we can act accordingly.

So we're using the HALT TM decider to decide A TM. That's the name of the game here, OK? And that's a contradiction. And so therefore our assumption that HALT TM was decidable had to be false. So it's undecidable. OK?

Important to understand this because this is sort of the prototype for all of the other undecidability proofs that we're going to do going forward. OK, so we can just take a few seconds here. If there's something that you're not getting about this, it's a good time to ask. Not seeing many messages here, or any, so why don't we go on?

But if you ask, I can get to it next slide too, all right. Here we go. So here's the concept of reducibility. And I know, I've taught this course, many times. I know where the bumpy places are in terms of people struggling with material. The concept of reducibility is a bit tricky.

So don't feel bad if you don't get it right away. You know, so that's why I'm going to try to go slowly in this lecture to make sure we're all together on understanding how reducibility works. OK, so the concept of reducibility is that we say one problem is reducible to another, say A reducible to B. It means that you can use B to solve A. That's what it means for A to be reducible to B.

OK, so I'm going to give a bunch of sort of informal examples of that, or easy examples of that. And then we'll start to use it for real. So example 1, this is sort of really outside material from the course. But I think it's something you can appreciate.

You know, everybody knows you can measure the area of a rectangle by measuring the lengths of the two sides, measuring the length and width of the rectangle. So in other words, if you had the problem of determining the area, you could reduce that problem to the problem of measuring the length and width of the rectangle.

So here, we're taking one problem and reducing it to another problem. You know, it's conceivable that measuring the length and width is easier than it would be to measure the area directly by somehow covering the space with tiles, is one way of measuring it. But it tells you, you don't have to do that.

The problem of measuring the area is easier than covering with tiles. You can just measure the length and width and you're done. So reducibility is a way of making problems easier by translating them into some easier problem.

So here's another example that we've already seen. We didn't call it a reducibility. But if you remember back a couple of weeks ago, we were talking about the languages A NFA, and A DFA, the acceptance problems for NFAs and DFAs. And we gave a way of solving the A DFA problem.

As you remember, the Turing machine simulated the finite automaton. And we solved the NFA problem not by doing it directly but by converting the NFA to a DFA and then using the solution for A DFA. In effect, what we were doing was we were reducing the A NFA problem to the A DFA problem.

So let's do another example. Here's a problem. Here's an example that you again, probably didn't think about it this way. But from your homework, we had this pusher problem, the problem of determining whether a pushdown automaton ever pushes on its stack for any input. I know a bunch of you were struggling with that problem working on it, hopefully solving it in one way or another. So there's one way to solve it is in effect by reducing the pusher problem to the E CFG the emptiness for CFG which is the equivalent to the emptiness for PDAs.

I mean, this is the solution I had in mind, which is a particularly simple and short solution. Of course, not the only solution. You can take your pushdown automaton where you're trying to determine if it ever pushes. And you can take the states that are about to make a push. And instead of making them make a push, you make them accept states. And you get rid of the original accept states. So now you've converted this automaton to one that accepts every time the original push on automaton pushes.

And it accepts, and then has to move to the end of the input, of course. So it goes into an accept state and moves to the end of the input. So every time the original machine was about to push, the new machine that you're just creating here is going to go into an accept state at the end of the input.

Now to test whether the original machine ever uses stack, it's enough to test whether the new machine ever accepts an S string. OK, so that's a way. I don't want to overcomplicate this right here and get you thinking about the homework again. But this is a way of reducing one problem to another problem.

And if you don't quite get this one, just focus on the other two examples. I don't want to spend time on the homework set 2 right now. So we can address that all separately if you want.

It's also the solution that's written up in the solution set that's posted on the home page by the way. OK, so getting back to let's see, thinking about reducibility. What I have in mind, again, this is sort of rephrasing it, but I'm trying to hammer it in that if A is reducible to B, then solving B gives a solution to A.

Because that's what happens in each of these examples. Now, how are we going to use that? We're going to use that in the following two ways. One is to observe that, if A is reducible to B and B is an easy problem, then A must also be easy because we have a way of converting A problems into B problems. We have a way of solving A using B. So B is easy. Then now you can solve A too easily.

Because you can solve A using B, which is easy. Maybe that's clearest up here in example 1, where measuring the area might seem at first glance hard. You could have to walk out over the whole area. But it's not hard because you only have to measure the length and the width.

So the fact that B is easy tells you that A is easy. But actually, this is not the way we're going to be using it most typically. We're going to be most typically using it in the second version, which is a little bit more convoluted.

But this is the way you're going to have to get used to this. So if A is reducible to B, and you know A is hard, undecidable, unrecognizable, whatever the form of hard you care about, if you know A is hard, and A is reducible to B, then that tells you B also has to be hard. Why?

Because if B were easy, then A would be easy. But we're assuming that A is hard. So B also has to be hard. OK, so I'm inverting the logic here. But this is logically equivalent.

So you have to mull that over a bit. So why don't you think about that. And let me just take a few questions on the chat and don't forget the TAs are there too. So they're happy to answer your questions. Don't make them sit there lonely, all right.

So somebody is asking, is it possible that A is reducible to B and that B is also reducible to A? So that's a good question. That can certainly happen. In that case in a certain sense, A and B are going to be equivalent.

So solving 1 is going to be just as easy or hard as solving the other one, OK? So they're going to be equivalent from the perspective of the difficulty of solving them.

So somebody is asking-- and this is a perennial confusion-- so in the previous slide here, I think I'll just flip back to it here. So which direction are we doing? Are we reducing A TM to HALT TM or HALT TM to A TM?

The way it's written on the slide is what I have in mind. Here we're reducing A TM to HALT TM because we're using HALT TM to solve A TM. And that's reducing A TM to HALT TM. Just like here, measuring the area is reducible to measuring the lengths of the sides, we're using measuring the length of the sides to solve the area.

So we're reducing the area to the lengths, the area to the length of the sides. But I know you're going to have to play with it, digest it, get used to it. All right, OK, so let's continue.

OK, so as I said, this latter one because the focus on this course is mainly on the limitations of computation. So we're going to be looking at ways of showing problems or difficulty. It could be difficult in principle, like undecidable. Or it could be difficult in terms of complexity, which is what we're going to focus on in the second half.

But in both cases, we're going to be using the concept of reducibility. So reducibility is going to be a theme. You've got to get comfortable with reducibility, OK. So we're going to be focusing more on the notion that if you reduce A to B, and you know A is hard, that tells you B is also hard.

OK, so I'm going to say that a few times during the course of today's lecture to try to help you get it. All right, here's a check in. A little bit sort of off to the side. But I thought it was a fun check in more.

The question is, some people say biology is reducible to physics. Well, maybe everything is reducible to physics since physics tells you about the laws of the universe. And biology is part of the universe. So my question to you is, do you think?

And there's no right answer here. Do you think, in your opinion, is biology reducible to physics? Maybe yes, or maybe there are some things like consciousness which cannot be reduced to physics. Or maybe we don't know. So I'm curious to know your thoughts.

But it does kind of use in a sense the notion of reducible in the spirit of what I have in mind here. In the sense that if you could fully understand physics, would that allow you to fully understand biology? OK, here we are. We're almost-- kind of interesting, though not too unexpected I suppose.

So we are, I think, just about done. 5 seconds, pick anything if you want to get credit for this and you haven't selected yet. Ready to go, ending polling. Here are the results.

And as I say, there's no right answer here. But if I had been in the class, I would have picked B. But I'm not surprised, especially in an MIT crowd that A is the winner, all right.

Let's continue. OK, so now we're going to use reducibility again. This is going to be yet another example like the HALT TM example, but a little bit harder. And we're going to be doing this. You know, next lecture, we're going to be doing more reducibilities but much harder.

So we really got to get really comfortable, all right. I want to show E TMs. So E TM is the emptiness problem for Turing machines. Is this language empty? I'm just going to give you a machine. I want to know, is this language empty or not? Does it accept something or is this language empty?

That's going to be undecidable, no surprise, proof by contradiction. And we're going to show that A TM is reducible to E TM. So these things often take a very similar form. And I'm going to try to use the same form. So if you're feeling shaky on it, at least you'll get the form of the way the solution goes and that will help you maybe plug-in to solve problems, OK.

So proof by contradiction, assume that E TM is decidable, opposite of what we've been trying to show. And then show that A TM is decidable, which you know is false. So we'll say we have a decider for A TM, R, using the same letters on purpose here just to try to get the pattern for you, so R deciding E TM.

Construct S deciding A TM, OK. So now, let's think about it together for a minute before I just put it up there. So S, I'm trying to make a decider for A TM, using my decider for the emptiness problem, OK?

So we have R, which can tell us whether M's language is empty. So why don't we just, I don't know, stick M into that emptiness tester and see what it says? I'm not saying this is the solution, but this is how one might think about coming up with the solution.

So are you with me? We're going to take M, we have an emptiness tester. Let's take M and plug it into R, see what R says. R is going to come back and tell us whether M's language is empty or not.

Now, one of those answers will make us happy. Why? Suppose R tells us that M's language is empty. Why is that good? With that, we're done.

Because S is trying to figure out, we're trying to figure out exactly, somebody told me the answer which is correct. Because now we can reject. If M's language is empty, it's clearly not accepting w because it's not accepting anything.

So if R says M's language is empty, then we're good. The only problem is we also say M's language is not empty. And then what do we know? Well, not much, not much that's useful for testing whether M accepts w. We just know M accepts something. But that something may or may not be w. OK, so what do we do?

Well, the problem is that M is possibly accepting all sorts of strings besides w, which are kind of mucking up the works. They're interfering with the solution that we like. We'd like to be able to use R on M to tell us whether M is accepting w. But M is accepting other things. And that's making the picture complicated.

So what I propose we do, why don't we modify M so that it never accepts anything besides w? The very first thing M does in the modified form is it looks at its input and sees whether it's different from w. If it's different from w, it immediately rejects. Now we take that modified machine, and we feed it into the emptiness tester.

Now the emptiness tester is going to give us the information we're looking for because if the emptiness tester says the modified machines language is empty, well, we know that M is not accepting w because we haven't changed how M behaves when it's given w. But if R says M's language is not empty, well, then it must be that M is accepting w. Because we've already filtered out all of the other possibilities when we've modified the machine.

So let me repeat that on the slide and write it down a little bit more formally. So what I'm going to do is I'm going to transform M to a new Turing machine. I'm going to call it M sub w to emphasize the fact that this new machine depends on W. It's going to actually have w built into as part of the rules of the machine.

So for a different w, we're going to end up with a different machine here. So this is a machine whose structure is going to depend on knowing w. And that machine is going to be very much like the original machine M, except that when it gets an input, let's say it's called X now, that machine is going to compare X with w and reject if it's not equal.

Otherwise, if X is equal to w, it's going to run M on w as before. So it's not going to change the behavior when the input is w. It's only going to change the behavior when the input is something different than w, and then it's going to reject, all right. So I'm going to look at two aspects of this.

First, let's understand the language of this new machine. And then we'll also talk about how we go about doing this transformation. So first of all, just for emphasis, so Mw works just like M. It has all the rules of M in it, except some extra rules.

It always at the very first step, it tests whether X is equal to w or not. And if it's not equal to, it rejects. Not equal to, reject.

So the language of that new machine is either going to be just the string w when M accepts w because everything else is filtered out. Or the empty set if M rejects something. So it's important that you understand the behavior, at least, of this new machine. It's just like M except filtering out all of the inputs which are not w. Those are going to be automatically rejected.

So it's also important that S be able to make this transformation. But I claim that you'll have to accept this if you don't totally see it. But the transformation is simply taking M and adding some new rules, some new transitions and states so that the very first thing that M sub w does is it just has a sequence of moves where it's checking that the input string is equal to w or not.

And if it's not equal to w, it just rejects. So it's easy to modify M. You could easily write a program which would modify the states and transitions of M to make it do that test at the beginning.

So I'm not going to elaborate on those kinds of things in the future. But just for the very first time, I just want to make sure you understand that we're not doing anything fishy here. This is a completely legitimate thing for S to be able to do. So S can modify M to this new machine Mw, which filters that new machine, filters out all strings except for w and rejects them.

So S takes that new machine, and what is it going to do with it? Is it ever going to run that machine? No. This machine is built not for running. This machine is built for feeding into R. Because as you remember, feeding M into R had the problem that M might accept things besides w. And that confuses the result that we get from R in the sense that it's not useful.

But if we feed Mw into R, now we're good because if the information about whether Mw's language is empty from over here tells us whether or not M accepts w. If Mw's language is not empty, then M accepts w. If M's language is empty, M rejects w.

OK, so starting to get some good questions here. Let me just finish the description of S. So somebody is asking here. So this is an excellent question. How do we know that Mw halts on w, or whatever? We don't. Mw may not halt on w. We don't care.

We're never going to run Mw on anything. We're going to take Mw as a machine, and we're going to feed it into R as a description. We are going to take the description of Mw and feed it into R. Then it's R's problem. But R has been assumed to answer emptiness testing.

So we just took the original machine, modified it so that the only possible thing it could accept is w. And now feed it into the emptiness tester to see whether its language is empty or not. Now if its language is not empty, it has to be accepting w because it's built not to accept anything else.

So we don't care whether Mw might end up looping. We're never going to run Mw. I acknowledge, it's a leap for many of you. So you're going to have to mull it over. So we're going to use R to test whether Mw's language is empty.

If yes, that means that M rejects w. So then we're going to reject, if we know that Mw's language is empty, that must have been that M rejected w. So now as an A TM decider, which is what S is, S is supposed to reject, which is what we have here in the description.

And if no, that means the language is not empty. So M accepts w, and so therefore we're accepting. So there's a little bit of a twist here also. OK, so let's take some more. I'm expecting some questions here.

So somebody is asking, how do you determine if the language is decidable? I mean, that's what we're doing. You can show a language is decidable by exhibiting a Turing machine which decides it. And you can show a language is not decidable, which is what we're doing here by proving that it's not possible for a Turing machine to decide it.

You know, we did that first with A TM. We got that contradiction by diagonalization. And here we're doing a reducibility to show as the method of proof, all right. Let's continue.

So now we're going to talk about a special kind. So far we talked about reducibility. We didn't define it in a precise way because there are several different ways to get at the notion of reducibility precisely. And I'm going to introduce a one version, which is a little bit more restrictive. I mean, somewhat more restrictive and a little bit different way of looking at it than we have been doing so far.

But there were going to be some benefits to looking at this particular kind of reducibility, which we're calling mapping reducibility. It's going to have several benefits for us immediately and down the road. But this is also a little technical so don't get scared off. It might look complicated at first. But we'll try to unpack it for you, OK.

So first of all, we have to first of all talk about the notion of a computable function. So generally when we've had Turing machines, they're doing yes, no. They're doing accept, reject kinds of things. So it's like a function, just sort of a computing, sort of a binary function.

For here, we're going to want to talk about Turing machines that are computing a function which converts one string to another string. So it's mapping from strings to strings. And it could be like the function which reverses the string, for example. That's one possible function you could be having here. But there's of course zillions of possible functions here.

And we're going to talk about functions that you can compute with the Turing machine. And that basically means you provide the input to the function as input to the Turing machine. And the output of the function, the value of the function comes out as the output of the Turing machine, which let's just say it leaves that value on its tape when it halts. It halts with the value of the function on the tape.

But just, we're thinking about algorithms here. Come up with your favorite method of thinking about algorithms. It has an input and an output. And the algorithm just computes the function by taking as input w, and the output is f of w. It doesn't have to be a Turing machine. Just any algorithm that can compute something is good enough. They are all equivalent.

And now we're going to use this notion of a function that you can compute to define a kind of reducibility called mapping reducibility. I'm going to say that A is mapping reducible to B, written with this less than or equal to sub M symbol. And you're going to see that a lot. It's on the homework also by the way.

If there is some computable function as I just described, where whenever w is in A, f of w is in B. And the way to think about it is with this picture. So A and B are languages, written like, here's A and here's B.

And now there are strings. So w might be in A. It might be out of A. And you think of you're trying to solve A. You're trying to decide membership in A. So you want to test whether w is in A or not.

A mapping reducible is a function which maps things from this space over to that space in a way that strings that are in A get mapped to strings that are in B. So if you start out with w in A, f of w is in B. And if w is not in A, then f of w is not in B. Pictorially it's a simple idea. We'll have to make sure we understand why this fits with our concept of reducibility. But we'll do that.

But anyway, let's first understand just what we're doing here. We're just coming up with a function that can do this kind of a mapping. It sort of translates problems which inputs which may or may not be in A into other strings which may or may not be in B. But sort of maintaining the same membership property.

So if you start out with something in A, when you apply f, you're going to end up with something in B. And conversely, if you're not in A, then you won't be in B, OK? Somebody is asking, so just a couple of questions here.

Not necessarily 1-to-1? No, so the function doesn't have to be 1-to-1. There could be multiple things that map to the same point. And is there any restriction on the alphabet? No.

So before we actually get into the example, let me try to give you a sense about why we call this a reducibility. And the reason is, suppose we have such an f which can do the mapping as I described. And we also have a way of deciding membership in B. So B is decidable. So that's going to tell us right away that A is decidable.

Because if you have some input, and you want to know, is it an A or not, you can now apply f and test whether f of w is in B. So the test of w is in A, you're going to instead test whether f of w is in B. And we're assuming that shows that A is reducible to B. So if you could solve the B problem, that also gives you a way to solve the A problem. So again, we're going to say this several times in several different ways. So if you didn't quite get it yet, don't panic. So here is going to be an example.

Sort of building on what we just showed last time in the previous slide, A TM, we're going to show how A TM is actually mapping reducible to the complement of E TM. And the complement is necessary here. The computable function that we're going to give, which is basically, the computable function is going to translate problems about A TM to problems about E TM because we're mapping reducing A TM to the complement of E TM.

So what we're doing here is with mapping M and w to the machine Mw. Kind of in a way, we're boiling out the essence, boiling down to the essence of the proof that we gave in the previous slide. This is really the core of the proof.

This translation of Mw where you want to know, is M accepting w to a new machine Mw where you're testing whether Mw's language is empty. And so remember Mw from before. It's the machine that filters out all the non-w's and rejects them.

And the reason why this reduction function works is that Mw is in A TM, if and only if M sub w is in the complement of A TM. So M accepts w exactly when Mw's language is not empty, OK? So M accepts w if and only if the language of Mw is not empty.

So you have to mull this over a bit to realize it's-- I know this can be a little tricky. But I think what we're going to do here, I think we're at the time for the break. So oh, no, there's one more slide. I apologize. So let's talk about this and then we're going to have our coffee break.

So these properties are really going to be getting at what makes mapping reducibility fit with our understanding of what a reducibility should be. So if A is mapping reducible to B, and B is decidable, then so is A. So that fits with what we want. Because if A is reducible to B, and B is easy, then A is easy. So here easy means decidable.

And here is the proof. Let's take a Turing machine that decides B and construct a Turing machine that decides A as claimed, S operates like this. It takes its input, computes f of that input, tests whether f of w is in B using the R machine that we were assuming. We have R deciding B. And if R halts then output the same result.

So if R accepts, we're going to accept. If R halts and rejects, we'll reject. And of course we're going to be similarly running R. So if R is not going to be halting, we are not going to end up halting either. OK?

So the corollary is, and this is the way we're going to be using it, if A is reducible to B and A is undecidable, then so is B. So this is as I mentioned, the focus for us is going to be on undecidability. And you may want to think about A is like the A TM problem which we know is undecidable. We're going to show the A TM is mapping reducible to some other problem to show that other problem is undecidable.

And the important thing about mapping reducibility is that it also applies to recognizes. So if A is mapping reducible to B, and B is Turing recognizable, then so is A. So if you're reducing A to a recognizable problem, then A is also recognizable, same proof. Because you can just map your w to f of w and feed it into the recognizer. That's going to give you a recognizer for the original language.

And the corollary is that if A is mapping reducible to B, and A is unrecognizable, then so is B. So this is again that inverted logic. So now I think we're-- oops, I meant to put this picture up earlier. OK, so here's a check in. It will be more of a check in for me to see how well you're following me.

So these are some properties-- so I'll give you a minute here to think about this-- some properties of mapping reducibility. Suppose A is mapping reducible to B, what can we conclude? Does that mean that we can flip it around? If A is mapping reducible to B, does that mean that B is mapping reducible to A?

What about this one? If A is mapping reducible to B, is the complement of A mapping reducible to the complement of B, or maybe neither? So you can check all that apply, multiple choice, 5 seconds. Sorry to pressure you, but we have to move on here. Pick anything. If you don't know, OK, 1, 2, 3, the end, OK.

Well, the majority is correct. In fact, it's only B. Now, A really is not in the spirit of reducibility. Because as suggested even by the inequality sign there, A being reducible to B is really a rather different thing than B being reducible to A.

So that's something. We're not going to prove that right here. But that's something that you could think about. But part B, I think, if you just look here at the definition of mapping reducibility, it maps strings in to in and out to out.

Well, that's just going to be if you exchange in and out as you do when you're flipping compliments on both sides, it's still by the same f going to still work as a mapping reduction, OK? So now we're at our coffee break. So we're going to take five minutes here, and I'll be happy to take questions here. Don't forget the TAs. They're here to. OK.

OK, so this is a fair question here. You know, so we had this notion of a general reduction and a mapping reduction. They are not the same. So any time you have a mapping reduction, it's going to be an example of a general reduction, but not the other way around.

So if you go back and look at the reduction that we offered for HALT TM where we showed A TM is reducible to HALT TM where we started, it's actually not a mapping reduction because we're doing something more complicated than translating an A TM problem to a HALT TM. We're kind of using the HALT TM decider in a more complicated way. And there are cases where that's actually necessary. So we're not going to discuss that here. But it's actually kind of an interesting homework problem perhaps, or some kind of a problem to think about.

So Turing machines for f's, it's not really a decider, but it has to be-- well, I guess it does have to be a decider in a way. It's always halts. The Turing machines for f has to always halt. It always has to have an output.

So f for the computing the function always has to halt. So someone is asking me, can I explain the statement that if error halts, then output the same result? I just mean that in that previous slide, or two slides back, if R accepts, halts and accepts, then we're going to halt and accept. And if R halts and rejects, then we halt and reject. So I don't know if this is a good idea, but we can just pull that back here.

So that was the statement here. If our halts and output the same result, I just mean that S is going to do the same. You know, we're translating an A problem to a B problem, and then answering the B problem. And we're going to give the same value, the same answer there. So whatever R says, we're going to say too, if that's helpful, all right.

Boy, this is a good question here. If A is reducible to B, why can't we just get B reducible to A by inverting the function? That's a great question. I like that question.

The reason is because the function that's mapping onto B doesn't have to be onto, subjective I guess. So if it was onto, so if it covered all of B, then I think then you would get an invertible function. And you would get the reduction going the other way as well.

But though, I'm not sure what happens when you have. It doesn't matter if you have collisions. It turns out that's not going to matter. But anyway, let's not get it too complicated here. But the problem with inverting it is that it's not necessarily onto the whole range of B.

So we're kind of out of time here. Is A reducible to A compliment? Let me just handle that. No, not necessarily. It's reducible. A is reducible to A complement, but not mapping reducible to A compliment.

But A is general reducible to A compliment. So actually we'll talk. I have a slide on that. So let us move on, OK?

Mapping I think, it's actually this line, mapping versus general reducibility. So we're going to contrast it to a bit. So mapping reducibility, which is what we've just been talking about has this picture, which is, I think, a very useful picture to remember. And because I like to think of a mapping reduction as a problem translator.

Your problem is sort of in the A domain. And the mapping reduction allows you to translate that problem into the B domain. OK, and then if you have a way of solving it in the B domain, combining that with the reduction, you get a solution to the problem in the A domain. So that's why if A is mapping reducible to B, and B is solvable, then A is also solvable.

So mapping reduction is a special kind of reducibility, as opposed to the general notion in general reducibility where we started. And it's particularly useful to prove Turing unrecognizability. So when you want to prove Turing unrecognizability, as we'll see, general reducibility is not fine enough in a way, it doesn't sort of differentiate things as well as mapping reducibility does. And for that reason, it's not always going to be useful to prove Turing unrecognizability. It's better for proving undecidability.

So what we're calling reducibility, or general reducibility is where we just use a solvent for B to solve A in sort of a most general possible way. So I'm writing that as a picture here. If you want to solve A, you're going to use the B solver as a subroutine to solve A.

That's the way we did the HALT TM reduction at the beginning. But we didn't necessarily translate an A TM problem to a HALT TM problem. It's slightly different. So you can go back and look at that.

So I find that people struggle more with the mapping reducibility concept. And that the general reducibility is what people naturally gravitate towards. And so in some sense, it's conceptually simpler. And it's useful to proving undecidability. But you really have to be comfortable with both. And especially in the complexity part, we're going to be focusing on mapping reducibility.

So one noteworthy difference here as sort of foreshadowed by the person who made this question, which is a good question, is that A is reducible using a general reduction to A compliment, which kind of makes sense. I mean, if I can test whether things are in A compliment, well, I can test whether things are in A. You just invert the answer.

But A may not be mapping reducible to A compliment because there is a very special kind of reduction. And you have to just translate things in the language to things in the language, and things out of the language to things out of the language. And they don't necessarily allow you to do that inversion.

So for example, A TM complement is not mapping reducible to A TM. Because as we pointed out, anything that's reducible to a recognizable language is going to be recognizable. Anything mapping reducible to a recognizable language is going to be recognizable. But we know that A TM complement is not recognizable. We showed that before. So it couldn't be mapping reducible to A TM. It's coming a little fast I realize. You're going to have to digest it.

So here's the last check in for today. OK, we showed that if A is mapping reducible to B, and B is Turing recognizable, then so is A. And so let's just say that again carefully. If A is mapping reducible to B, and B is Turing recognizable, then so is A. And here are the emphasis on Turing recognizable as opposed to decidable.

Is the same true if we use general reducibility instead of mapping reducibility? So you got it? So we're saying A is mapping reducible to B using this picture over here. And we're going to assume that B is Turing recognizable, so that we have a machine which halts an accepts when you're inside B, and you know, is going to reject possibly a looping when you're not inside B.

Now, that allows you to get a recognizer for A if you have a mapping reduction. Does it always work to give you a recognizer if you have just a general reduction? If you just have now, assuming you have a B solver, and you're going to build an A solver out of that. OK, so mull that over while I'm setting this thing up.

Well, the right answer is winning, but not by much. I suppose I shouldn't be laughing about it. But I knew that this is going to be challenging. So I think it's the kind of thing you're going to have to work at. So let's see we're almost done here, 5 seconds to go.

Better answer that, I can see a few of you, either you've left the room, or you're-- OK, 2 seconds, 1, 2, 3. Somebody hasn't answered it. There we go. So the correct answer is B. It's not the same.

The reason is that in general, I mean, the picture is right here. Let's see, how do I explain this? So we know that a language is going to be-- OK, if we're using general disability and A is just reducible to B,

we know that a language is always reducible to its complement in using general reducibility.

So if this were true, then we would have here so if this were true, when a language is reducible to its complement, if the complement were recognizable, the language would also be recognizable. That clearly is not going to be the case because you know, A TM complement is reducible to A TM using general reducibility. But A TM complement is not recognizable even though A TM is recognizable. So we kind of have a proof that this has to be no. But as you can see, I'm even getting myself confused.

So you have to stare at it. So let me see, we can try to take a couple of questions see if I can clear up people's confusion. So why again, is A reducible to its complement in the general sense?

So I'm saying, if you have a solver, if you have a decider for A compliment, it gives you a solver for A. You just ask the solver, is the string in the language or not? And now you just give the opposite answer if you want to solve the complimentary problem. So the A complement is general reducible to A. You just invert the answer for whatever the solver is doing for A. But you can't just do that inversion when you have a mapping reduction. It's a much more kind of specific translation that's allowed.

I mean, the fundamental difference between general reducibility and mapping reducibility, I'm trying to bring it out here. It's just a difference in the nature of the way things are used. Mapping reducibility is a special kind of general reducibility. So to answer the question about what's the fundamental difference, one is using the problem as a subroutine and the one is using it as a transformation.

Anyway, I think we're going to have to move on here. And I am going to have couple of examples which may help. And then, there are office hours too after the lecture. OK, oh, yeah, so I wanted to again to help you. I'm putting these down as sort of templates for how do you use reducibility.

I'm not saying you should just apply things blindly. But I think it's sometimes good just to see the pattern and then to understand how the pattern works. So once you just start to understand the pattern of how things are used.

So to show a language is undecidable, to prove a language B is undecidable, show undecidable languages reducible to B. Using just a general reduction is going to be good enough. And the template for that is, assume we have R deciding B, which you then can use as a subroutine when you make a Turing machine S deciding A. And that's going to be your contradiction. If A was originally shown to be known to be undecidable.

But now to prove something unrecognizable, this kind of reduction it's not as restrictive enough because this kind of reduction allows for complementation which is not going to be satisfactory when you're trying to prove Turing unrecognizable. So you're going to have to prohibit the complementation. And that's really one of the effects of the mapping reducibility if that sort of is getting at the essence of it.

So you're going to show an unTuring of a recognizable A is mapping reducible to B. Often you start out with the complement of A TM, which is a language we know is Turing unrecognizable as we showed before,

OK, here the template is you give the reduction function f, that computable function, OK. So here are going to be two examples, one showing that E TM is Turing unrecognizable. We showed it was undecidable before.

Now we're going to show it's even in a sense worse. It's not even recognizable. And the way we'll do that is to reduce a known unrecognizable language to E TM in the emptiness language.

So here is the picture that we have when we're doing mapping reductions. We're going to map strings that are in the complement of A TM, so strings that are outside of A TM if you wish to strings where the language is empty to machines where the language is empty. And here are strings describing machines which are where the language is empty. And here we're going to take A TM problems and map them to machines where the language is not empty.

And the thing that's going to do the trick is going to be that same reduction function that we saw earlier. We're going to take that machine w from before, the machine that filters out all the non-w's. And we're going to take Mw, which is an A TM compliment problem. So if M rejects w that it's in the complement of A TM, and that's supposed to map to a string, a machine which is where the language is empty, OK. So if Mw is in the complement of A TM, so M rejects w, then Mw's language is going to be empty, which is what you want to have happen.

Let me move on to my last. I mean, this example is in a way kind of similar to the one we did before. And I really want to get to the last example here. OK, so we'll have to just talk through this rather than having it build.

Let's take EQ TM. That's the equivalence problem for Turing machines. Do they recognize the same language? So this is a language of a new kind for us. This is a language where neither it nor its complement are going to be recognizable, Turing recognizable.

So the way we get that is, the way we show problems are not recognizable is mapping reduce a non-recognizable language to typically the complement of A TM. So we're going to mapping reduce the complement of A TM to both EQ TM and to the complement of EQ TM to show that both of those are not recognizable.

And here we're going to introduce a new machine that we're going to be building inside the reduction function. And that's going to be a machine I'm going to call Tw. And Tw is a machine that always behaves the way M behaves on w for every input. So if M accepts w, T is going to accept everything. If M rejects w, T is going to reject everything. So it copies the behavior of M on w onto all inputs.

And the way I describe that machine Tw is it ignores its input. Whatever the input is, it just simulates M on w. You could easily give an M and w, you can build the machine Tw. It just always runs M on w, no matter what input it gets.

And so now we're going to give a function which maps A TM problems which have the form Mw. So it's an A TM compliment problem. So this want to test if M accepts w or not. So that's an A TM compliment-type problem.

And I want to map that to an EQ TM problem with the form-- you know, EQ TM problems repairs the machine now, and where going to be testing equivalence. So I'm just trying to give you the form of the output of the reduction function f. And specifically, what it's going to look like is when we have f is processing on Mw, it's going to produce two machines. One of them is going to be Tw which always behaves the way M behaves on w but expanded to all inputs, and then a machine I'm going to call T reject, which just is designed to reject everything.

Now, just walk through the logic with me. If M rejects w, Tw rejects everything. And so we'll be equivalent to the machine T reject. That's what we want. If M rejects w, so we're in the language A TM compliment, then these two machines that I produce for you are going to be in the EQ TM line. That's what I want to have happen for a reduction from A TM compliment to EQ TM.

Similarly, to do part 2, I'm going to make here a different f, maybe I should call it f prime, all right, f1 and f2 for the two different parts. So these are two different f's. I'm going to make f here. Instead of generating Tw and T reject, I'm going to have Tw and T accept which is a Turing machine that always accepts its input.

Now, if M rejects w, it's in A TM compliment, then Tw is going to reject everything. And it's going to be different from its companion here, T accept. And so it won't be in EQ TM compliment. But if M accepts w, then Tw is going to accept everything. And it's going to be equivalent to T accept. And you will be equivalent.

So the here is where we're taking A TM compliment and mapping it to the compliment, the EQ TM. Too many compliments here I realize. Compliments are confusing. But anyway, why don't you mull this over.

And just to summarize, OK, we're out of time here. But why do we use reducing when we talk about reductions? It's because when we reduce A to B, we kind of bring A's difficulty down to B's difficulty, that's where the reducing comes from.

Or we bring B's difficulty up to A's difficulty, because it's really A's difficulty relative to B that we're talking about when we're reducing A to B. So that's why the term reducing seems a little out of place when we're proving things undecidable, or unrecognizable. But that's where it's coming from.

Anyway, quick review, we introduced the reducibility method. We defined mapping reducibility as a special kind of reducibility. We showed E TM as undecidable and unrecognizable. And that EQ TM is both, it and it's complement are unrecognizable.

So we're out of time. I will shut this down. But I'll take a few questions here actually. I'll stick around for a few questions. And then I'll move to the other chat room for office hours, OK?

So question, go over the case for the complement of EQ TM. So I will do that. So that's in this slide here. OK, so this is proof part 2 for the person who asked me to go over it. But I think it's helpful for those of you who might be a little bit shaky on this. I want to mapping reduce the complement of A TM to the complement of EQ TM.

By the way, I don't know if this is going to be helpful. But as we pointed out in the check in a while back, that's completely equivalent to having a mapping reduction from A TM to EQ TM. You can complement both sides, and you get an equivalent statement. Maybe let's stick with the compliments here though. I hope that doesn't make it too confusing, OK.

We're trying to show the complement of A TM is mapping reducible to the complement of EQ TM. What does that mean? So that means when M rejects w, so you're in the compliment, we want the two Turing machines to be inequivalent. No, yeah, so we're in the compliment of EQ TM. So in other words, when we're in the complement of A TM, we want the result of the f to be in the complement of EQ TM.

So in other words, when M rejects w, the two machines should be inequivalent. Right? When M accepts w, the two machines should be equivalent. Because when we're not in this language, so we're in A TM, we want to be not in that language. So we should be in EQ TM.

So when M accepts w, we should be equivalent. When M rejects w, we should be inequivalent. That's what we want. Let's go down here.

So if M accepts w, we want them to-- so when M accepts w, we want them to be equivalent. So if M accepts w, Tw accepts everything. And it's equivalent to T accept. When M rejects w, Tw rejects everything. And so it's not equivalent to the machine that accepts everything. So just go through the logic yourself. You'll see why it's working. All right, so, bye, bye, everyone.