[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: Hi, everybody. Can you hear me? Yes. Good. Welcome back to the course.

And so we are now at lecture 10, and let's see. What have we been doing? We've been talking about undecidability. So we introduced, last time, reducibilities and mapping reducibilities for proving various problems are undecidable. And this time, we're going to-- and today, we're going to introduce a more sophisticated method for proving undesirability using reducibilities.

And that's called the computation history method. And that's pretty much what's used in all of the more serious cases where people have proved problems undecidable. It's pretty much a widespread method. Especially it's not an active area of research these days, but at times when people were proving problems undecidable, this was frequently a method that was used. So we'll go over that today, and we'll prove a few examples of problems undecidable using that method.

All right. So why don't we get started. So first of all, most importantly, you should remember how we prove problems are undecidable in the first place. And that is by showing some other undecidable problem that we already know is undecidable is reducible to the problem of interest.

So typically, you might be reducing ATM, which is a problem that we started with and showed to be undecidable by the diagonalization method. Or it might be some other problem that we've subsequently shown to be undecidable, and you take that problem, and you reduce it to the problem you're trying to show undecidable. OK?

Whoops. Let me fix that. OK. So this is the thing to keep in mind. To prove a language is undecidable, show that ATM or some other known undecidable language is reducible to the problem B that you have at hand that you're trying to show undecidable.

All right. So we are now going to-- first, I'm going to start off by remembering Hilbert's tenth problem, which we discussed briefly a few lectures back, as you may remember. So that's a problem where you were given, say, some polynomial over several variables, like 3x squared y minus 15z plus 2xz equals 5. And you want to solve that problem. You want to find a solution that solves that equation, but you require that the solution involves only integers. So those are so-called diophantine problems because the requirement is that the solution be in integers.

And Hilbert's problem was to ask for an algorithm. Not the language that he used, but doesn't matter. Essentially, he asked for an algorithm to test whether a polynomial has a solution in integers, whether the polynomial equation has a solution in integers. And as we now know, that problem was posed back in 1900, it took 71 years to find the solution, but the answer is no. There is no-- there is no such algorithm.

We talked about this when we were discussing the Church-Turing thesis. And the method showing that there is no algorithm is by a reduction from ATM to that problem. So one can use exactly the method that we're using today to prove that problem is undecidable about polynomials.

The only thing is is that the reduction, that single reduction, would take the entire semester. And I know that's in fact correct because when I was a graduate student, there was a woman in the math department at Berkeley, where I studied, and the whole course was to go through the proof of Hilbert's tenth problem, of the solution to Hilbert's tenth problem, the proof of the undecidability of the solution, of the undecidability of testing whether a polynomial has an integral solution.

So the thing is is that involves some fairly hairy number theory in order to come up with the right polynomials to basically simulate the Turing machine. So that's kind of getting ahead of ourselves. That's what we're going to be doing today. But we're going to-- instead of looking at that problem because we don't have a whole semester to spend on it, we're going to look at a toy problem instead where there's no number theory, but it still has the same basic underlying idea that was employed in the solution to Hilbert's tenth problem.

So that toy problem is called the post correspondence problem. And it's going to have some other utility for us. Well, the main reason we're studying it is just to illustrate the method. So and the method is going to be, as I have been saying, is this computation history method. OK, so why don't we-- we'll first talk about this post correspondence problem because that's very easy to understand, and then we'll spend a little time introducing that method.

All right. So the post correspondence problem is as follows. You're given a bunch of pairs of strings. So here is one pair, t1 and b1.

And I'm writing them as dominoes. I'm calling them dominoes because I'm writing one of the strings on top of the other string. In fact, I'm using t and b for top and bottom here. So this is t1 is the top string, b1 is the bottom string in the first domino. And then in the second domino, we have two other strings, a top and a bottom and so on.

So we have these here, this collection of dominoes, as sort of the legal dominoes. And what the goal is, to find a sequence of those dominoes. So you want to pick from these dominoes here. You're allowed to repeat dominoes.

So you want to pick a sequence of those dominoes, line them up next to each other, and so that the string that you get by reading along the top, all of the ts together, is going to be exactly the same as the string you get by reading along all the bottoms. I'll give you an example. So here, to write it down a little bit more formally, so what I'm calling a match is a sequence of these dominoes, so it's ti1, ti2. Well, it's i1 is the first domino, i2 is the second domino and so on. And then what you're going to have is all the top strings concatenated together is exactly the same as what you get by concatenating together all the bottom strings.

So here's an example, and I think it'll become clear if you didn't understand it so far, but it'll be completely clear from the example, I hope. So here is a bunch of dominoes, four. And what I want to know is, is there some selection of these dominoes-- again, you can repeat dominoes.

Otherwise, it would be not an interesting problem. So you can repeat these as many times as you like. And I want to pick these dominoes and line them up so that what you get by reading along the top is the same as what you get by reading along the bottom.

So first of all, if you just stare at this and you're trying to make a match, you'll see that there's only one possible way to start this. For example, if you started with the third domino here, ba over aa, it would fail immediately because the b and the a are different. And if you use the second domino as your starting point, that would fail to match as well because aa would have to be-- the top string would start with aa, the bottom string would start with ab, and they could never be equal.

So by looking at this, you can see that the only possible choice that you have is to start with the first domino. So I'm going to start to build a domino for you just so you can see it. So here's the match so you can see the process. So the match starts with the very first domino.

And the way I'm going to write it is I'm going to take the dominoes and kind of skew them so that you can see how the top and bottom strings are lining up. But they're the same dominoes, they've just been-- I changed the shape. So this is the first domino, ab over aba.

Now, the second domino in my match, it's got to start with an a as the leftmost symbol on the top string. So that would rule out this one, for example. But there might be several choices, and so it's not exactly obvious which one you should take.

What I'm going to suggest is we take this one here. So we take aa over aba. You see it. aa over aba.

It's just written-- I'm just skewing it so that I can line up the a over a, the b over b, the a over a and so on. So I'm starting to build this concatenation of the top being equal to the concatenation of the bottom. Following me so far, I hope?

So now, what's next? So I need to have something where the top string is going to start with a ba. Fact, there's only one choice for that, so it's clearly going to have to be this domino is going to be next. So it's going to put a ba up here, but then it's going to force an aa down below because that's what this domino does. So this ba matches with that.

Now we have the aa to deal with. So we're going to reuse this domino because that's the only one that starts with an aa. So we're going to reuse that one. Now we have aa and aba.

Now we need something that starts with aba. I see two choices. Could be this one because this is consistent with-- at least it captures the ab. We could try putting this one over here, but a better choice would be to use this one because if we use this one, we finish the match. And therefore, we have found what we're looking for. And this collection of dominoes here has a match.

Now, it's not always obvious. Some collections of dominoes may not have a match. And so the computational problem is, is there a match?

And the theorem that we're going to prove today is that this problem is undecidable. Simple as it is, simple little combinatorial problem of trying to put together different tiles like that to form a match, there is no algorithm for solving that problem. And I think this is kind of interesting because it's the first example of a problem that we have where the underlying question does not really seem to have anything to do with computation.

You might argue that ATM being undecidable is perhaps a little unsurprising after the fact because it's a problem about Turing machines, so you can imagine Turing machines are going to have a tough time answering. But here's a problem just about strings. And we're going to show it's undecidable.

So just to formulate this as a language, I'm going to call the PCP language, is the collection of these PCP problems, PCP instances, which are just themselves, each one is a collection of dominoes. So it's the language of all collections of these dominoes where there is a match. So it's all of the PCP problems where you can solve them with a match. And we're going to prove that it's undecidable by showing that ATM is reducible to that PCP language.

Now, before we do that, I'm going to have to-- I'm going to explain to you what the computation history method is in the first pla-- that we're going to be using. So before we do that, here's our first check-in for the day. Just to make sure you're following me about what this PCP problem is, I am going to give you a question to test, is there a match with these three dominoes? So I want you to think about that.

I mean, the main thing I want to make sure you understand is what a match is in the first place. This is not a super hard problem to tell whether or not there's a match. OK. Five seconds. All ready? OK. Closing it out.

OK. So for those of you who said there is a match, I want you to exhibit the match because in fact, there is no such-- there is no match. And I mean, you could get-- by fiddling around with it, I think if you try to find a match, you'll pretty quickly see that you're going to get-- I think what happens is you kind of get into-- if you try to build a match with this particular setup, you're just going to get stuck sort of repeating yourself. And one point here that you may have missed is that a match has to be a finite sequence.

So if you were thinking about-- there is a kind of an infinite match that you can build with this set of dominoes, but that's not allowed. We're only allowing finite matches. And so that means that you might change your answer as a result, but too late.

So and one way to see that you can't have a match in this problem is that you're going to have to start-- neither of these two are possible starting points, so this one is going to be clearly the only chance that you're going to have as a match, the first domino here. But then once you put this one down, the bottom one, the bottom string, the bottom bs that you're going to get, the bottom string is going to be longer than the top string. And then the other two are going to have the same length.

So you're never going to have the top be the same length as the bottom. The bottom is always going to be longer no matter how many more dominoes you lay out. So there's no chance of there being a match here.

OK. So now we're going to work our way up to introducing this computation history method. And first, let me define something for a Turing machine that I'm going to call a configuration. Configuration is just a snapshot of the Turing machine in the middle of its computation. So if you're running the Turing machine and you just stop it at some moment, it's going to be in a certain state, the head is going to be on a certain position, and the tape is going to have a certain contents. And with that information, you can then continue the computation of the Turing machine.

That's a full set of the information that you need about the Turing machine that tells you everything about its computation at that moment in time. The state, head position, and tape contents. And so that, we're calling that information together, state, head position, tape contents, we're calling that a configuration. Fairly basic notion. I mean, if your program-- if you have something-- if you have the states of all the variables and where the current execution of the program is at a moment in time, same idea.

OK. So in terms of a picture here, here is the Turing machine. Imagine it's in state q3. The head position is in the sixth position on the tape, so p would be 6 because it's in the sixth position, and the tape contents is going to be this bunch of as followed by this bunch of bs. So I would write it down like this. State's in q3, head position number 6, this is the contents of the tape.

Now, what we're going to often do for convenience in using this notion in proofs is that we're going to want to represent a configuration as a string in a particular way that's going to be-- that's going to-- this concept is going to come up kind of again and again during the course. And so it's going to be handy to have a particular way of writing down configurations for doing proofs about them.

And so the way we're going to write them down is-- I think it's just maybe-- I can just put it right up here like this. We're going to write down the symbols of the tape. But what we're going to do is stick in the middle of those symbols the state of the machine immediately to the left of the position that the machine is currently reading.

So you can imagine the head here, which is coming. Imagine the state is kind of where the head is. It's pointing at the symbol immediately to its right.

So this is just another way of writing down a configuration. Here's the tape contents, which I'm sort of writing formally here. I'm breaking the tape contents into two parts, which I'm calling t1 and t2. This is the t1 part, the t2 part.

And I'm putting the state in between t1 and t2 where I have in mind that the machine's head position is right at the beginning of t2. But maybe this picture just says it all and it's clear enough. So just keep in mind that when we're going to be writing down configurations of machines, we're typically going to write it down this way.

OK, so I think this is a good moment to take a-- to take a moment for questions. Oh. I see there's a lot of questions already. So one question is, how does the encoding differentiate between the string and the state?

If I'm understanding you correctly, I'm going to be assuming that the symbols that represent states and the symbols that appear on the tape are distinct from one another. So you can always tell just like usually the way we write things down when you have a state symbol or whether you have a tape symbol. And so in a configuration, you're going to have a bunch of tape symbols and a single state symbol represent in the position where the head is.

Somebody says-- 6 here is just this is 1, 2, 3, 4, 5, 6. We're in head p-- the head is in position six. That's all I had in mind for 6, and that's why that 6 is appearing over there because that's the position of the head. Good. All right. So why don't we continue.

Now, we're all ready to start defining computation histories, which themselves are not very difficult concept. Computation history for a Turing machine M on an input w is just the sequence of configurations you go through. When you start the machine at the beginning with M on the tape-- with w on the tape, I'm sorry, and the head at the beginning in position one and the state in 20 or whatever the start state is of the machine. So that's going to be the starting configuration here.

And these are the sequence of configuration that machines go through-- that the machine goes through step by step. Every step of the machine is getting represented here until it ends up at an accept. That's what we mean by a computation history, or sometimes we're going to emphasize that by calling it an accepting computation history. Represent meaning that the machine has accepted its input, and these are the sequence of configurations that the machine goes through. That's what I mean by a computation history.

You know, I'm sure in-- I mean, I think the terminology changes over the years, but there's a notion similar to that for any kind of-- if you're running a program and you just want to keep track of how the variables are changing as you're in this particular run of the machine, and you're writing them all down, it's like a log of the history of the settings of the internal states of the machine and variables and so on. So this is just all of the configurations the machine goes through on the way to accepting its input. If the machine does not accept its input, there is no computation history. OK? So there is no accepting computation history, at least, to emphasize that point. Because that can only occur obviously if the machine has accepted its input.

OK. So just as we had with configurations, we're going to want to be able to write down computation histories as strings. And it'll be convenient to have a particular format for doing that as well. And it's kind of the obvious thing.

We're just going to take the sequence of configurations in the computation history and write them down as a string where each configuration is going to be the string for that configuration. I'll show you kind of a little example in a second. But just to focus on the concept here, we're just going to write down the string for this configuration, the starting configuration and then the next configuration and so on until you get to the accepting configuration and separate each of those strings by some pound signed letter.

OK? So here is a computation history for M on w where w is the string w1, w2, dot, dot, wn. So here is the-- here is the first configuration. So this part here is the computation history encoded as a string.

I'm giving you this extra side information just to make it clearer what's going on there. So this is the first configuration, this is the second configuration, and so on. And I'm trying to even give you a little bit more detail of how it might look by kind of making the example a little bit more concrete.

So let's say that the Turing machine, when it's in the starting state q0, looking at the very first symbol of the input tape in this particular input, say w1, it goes from q0 to q7 and writes an a on the tape and moves its head to the right. So that's why the second configuration reflects that. See here, it went from q0 to q7. Now the head is in the second position, so that's why that state symbol is moved over one. And the very first position now has become an a because the machine has changed whatever was on that input tape there in the first position to an a.

And then the next step, it goes from q7 reading a w2, which is where the head is now, looking at the w2, and goes to q8, writing a c, and again moves right. OK? So that's why I drew these in the way I did. And so you go through a sequence of those, you get to q accept, and that's the encoded form of the computation history. All right?

I think that's-- is that all I wanted to say? Yeah. So you can feel free to ask another question if you want. I'm just going to relaunch this. Yeah. Good. So if we're all together on configurations, computation histories, and how we're going to be writing them down as strings, that's what we've done so far. No questions, so I'm going to move on.

All right. So let me define a new automaton that we're going to mainly use as just to provide an example for us today. I'm going to call this a linearly bounded automaton. And all it is is a Turing machine where the Turing machine is going to be restricted in where it can-- the tape is not going to be infinite anymore. The tape is just going to be big enough to hold the input.

So the machine no longer has the ability to move into the portion of the tape to the right of the input because there is no tape out there. It just has the tape sitting here that contains the input, which the tape itself can vary in size. However big the input is, that's how big the tape is. So the tape adjusts to the length of the input.

But once you've started the machine with some particular input, that's as big as the tape is. There's no more. The reason why it's called linearly bounded is because the amount of memory is a linear function of the size of the input because you can effectively get somewhat more memory by enlarging the tape alphabet, but that's going to be fixed for any given machine, so that's where the linearly comes from, if that's helpful. But if you don't get that, it's sort of a side remark.

But what's important to me is that you understand what I mean by a Linearly Bounded Automaton, or an LBA. It's just like a Turing machine, but that portion of the tape that originally had blanks is just not there. As the machine tries to move its head off the right end of the input, it just sticks there just as if it tried to move its head off the left end of the input. Doesn't go anywhere.

So now, we're going to ask the same kinds of questions about LBAs that we ask for other automa. So the acceptance problem. If I give you an input and some particular LBA and I want to know, does the LBA accept that input? Well, and now the question is, is that the decidable or not?

So at first glance, you might think, well, an LBA is like a Turing machine, and the ATM problem is undecidable, so that might be a good first guess. And also, if you try to simulate them, if you try to figure out how you would go about simulating the machine, if given b and w, if you actually tried to simulate the machine to get the answer, so you run b on w, well, of course, if you run it for a while, and it eventually halts, either accepting or rejecting, then you know the answer and you're finished. But this machine might get into a loop. You know, nothing to prevent the machine from looping on that finite amount of-- on that limited amount of tape that it has. And then you might be in trouble.

But in fact, that's not the case because when you start out with a limited amount of tape, if you run the machine for a long time and it's not halting, it's going and going and going, inevitably, it's going to have to repeat, get into exactly the same configuration that it did before because there's only a limited number of configurations that the machine has. And once it repeats a configuration, it's going to be repeating that configuration forever, and it's going to be in a loop.

So this problem, in fact, is decidable because the idea is if b on w runs for a very long time and an amount that you can calculate, then you know it's got to be cycling. More than just looping. It's got to be repeating itself. And so therefore, once it starts repeating itself, it's going to be going forever.

So and here is the actual calculation, which is something I'm sure you could do on your own, but just to spell it out. So if you have an input of length n that you're providing to b, so if w is of length n, the LBA can only go for this number of different-- it can only have this number of different configurations. The number of states times the number of head positions, which is n, the number of head positions on the tape, times the number of different tape contents. If the tape was only one long, this is the-- this is the size of the tape alphabet. So if the tape were two long, the tape had two cells on it, the number of possible tape contents would be the square of the alphabet.

And if the tape is going to be n symbols long, it's going to be the tape alphabet size to the nth power. So therefore, if a Turing machine runs for longer, it's got to repeat some configuration, and it'll never hold. So the decider is going to be hopefully clear at this point.

You're given b and w, so this is the decider for a LBA. It's going to run b on w for this number of steps. If it's accepted by then, then you accept, and if it hasn't, if it's rejected or it's still running, then you can reject. And you know, if it's still running at this point, it's never going to accept. All right. Any questions on this?

OK, let's move on. All right. So now, but what's different now, or what's perhaps interesting now is that even though the acceptance problem for LBAs is decidable, the emptiness problem is undecidable. And that's where the computation history method is going to come in. So this is where we're going to be starting to do something new in terms of a proof technique.

So we're going to reduce ATM to ELBA, the emptiness problem for LBAs. So given an LBA, does it accept anything or not? And this is going to use the computation history method.

So this will be a chance to illustrate that method for the first time. So the setup initially is just like before. We're going to assume we have a Turing machine that decides this ELBA problem that of interest. And we're going to use that to construct Turing machine-deciding ATM for our contradiction.

OK, so here is S, supposedly deciding ATM, and what is it going to do-- and this is going to be the tricky part. S is going to use M and w to design an LBA. That LBA is going to be built with the knowledge of M and w. In fact, it's going to have M and w built into it. So that's why I'm calling that LBA B of Mw because it depends on Mw, as I'll describe.

And what that LBA does, it takes its input, let's call it the x, the input to the LBA, and examines that input to see if that input is an accepting computation history for M on w. That'll be just looking for computation histories for M on w. And that's the only thing it's going to ever accept. If you feed it something which is not a computation history, that sequence of configurations for M on w leading to an accept, if you feed it something else, the LBA is just going to reject it. It only accepts the accepting computation history for M on w.

And now, if you can build such a thing, then you can use that machine in your emptiness tester to see if it accepts any strings at all. Because the only thing it could possibly be accepting is an accepting computation history. You don't even know if there exists one because you don't know if M accepts w.

So you're going to build this LBA, which is looking for accepting computation histories, and then see if its language is empty or not. If its language is not empty, the only thing it could be accepting is this computation history, so you know M accepts w. Whereas if the language is empty, you know that there is no computation history for M on w, and so M does not accept w. So that's the whole idea.

The trick is, how do you build this LBA? So first, you're going to make this LBA which tests its input to see if it's an accepting computation history for M on w. And you have to build this LBA without even knowing whether there is a computation history.

It's not like you can take that string and just build a string into the LBA because you don't even know if that string exists. But what you can do is you can make the LBA follow the rules of M. It knows w, and it knows M, so it can take its input, and using w and the rules of M, see if that input is a computation history for M on w. And that's what it's going to do.

So here is this machine I'm going to construct. I'll give it to you written down and with a little bit more details of its procedure. But just to kind of illustrate it, so here is a proposed input to B of Mw. This would be the x here. This would be an x that would be accepted. But anything else that would be provided would not be accepted.

So this is the sequence of configurations written down as a string. That would be the x that I'm providing to the B of Mw. And it's supposed to be checking this to make sure it's legit. So on input x, the way it's going to-- the way it's going to proceed, it's going to first check to see whether x begins in the right way because this LBA, it knows M and it knows w.

So the very first thing is it takes a look at the first part of the string up to the pound sign. If there's no pound sign, if you're going to just feed junk in, it's going to be easily identifiable as junk. So if you're going to feed something-- so it's going to-- the LBA is going to take everything up to the first pound sign and just confirm that that thing is the first configuration, the starting configuration of M on w, which means it has to start with the start state, then here is w. So just kind of check that.

Then it's going to check that each one of these guys follows legally from the previous one according to the rules of M. It's going to first check that this one is correct, and then this one leads to that one, according to the rules of M, then this one leads to that one, according to the rules of M. All of that-- the knowledge of M and w is built in so it can do that. And then it just follows along, checking this computation.

It's easy to check that a computation is correct. That's all that's really going on here. It's going to check that the computation is correct until it gets to the end and then makes sure that the last configuration that it's been given as input is an accepting configuration, that there's an accept state in it. And if everything that passes it accepts, otherwise, it rejects. OK?

And the point is that this is an LBA. You don't-- oh, wait. Just kind of jumped ahead of myself. You don't need any additional information. So how does it actually do this? So how do you actually do this on the tape?

So I claimed that the LBA doesn't need any extra space to do this-- to do this check. It's just going to be zigzagging back and forth here on the input, checking that the corresponding symbols match up except around the head position where it gets updated correctly. And it may need to mark-- it will need to mark on the tape, it's allowed to write on the tape, just to make sure it keeps track of where it is.

But this is a very simple-- I mean, I'm not trying to-- if you're not following it, I'm not trying to alarm you, but I'm just trying to help you understand that this is not a complicated procedure here to do this check that the actual computation that's written down of the Turing machine is valid. But we can deal with the question here. Oh, good. Now we got some questions. Oops.

You can be thinking about-- while I'm thinking about this, you can think about that check-in over there. So here's a good question. Going from each configuration to the next configuration, is it a unique next step? Well, I'm assuming that the Turing machine we're starting with is deterministic, so there should be only one way to go. So the answer to that question is yes.

This is going to be a unique string here. There's really going to be only one computation history. Not that it really matters in a sense, the answer to that question, as long as that accepting computation history corresponds to the machine actually accepting.

Oh boy, we're all over-- we're all over the place here. OK, are you ready to end this poll? All right. Well, this is where we are.

Everybody who said they'd rather be in 6.046, I know who you are. You're all going to be expelled. No, I'm joking. I don't know who you are.

So yeah. Good. So we are here at the break, and I made this poll on purpose at this moment so we can spend a little time. Let me just start our clock going, and I can try to help you, those of you who answered C, that you're baffled. I can try to help you understand what's going on.

Ah. So this is a good-- this is a good question here. OK, well, OK. Several good questions here. So somebody asks, why don't we just test all possible strings for B?

Remember, if we're trying to test emptiness for B's language, that's the ELBA problem. Now, why don't we just try all possible strings? And that would work if we had enough time, but there's infinitely many strings, and so that's not going to be good if we're trying to be a decider. So that's why we don't just try all strings.

But here's this other question here. This is a-- OK. So the question is, how do we find the input x? So this is an important question because we don't find the input x. There's no-- the input x, at least the accepted input x, would be the accepting computation history for M on w. We're never going to find an-- we're never going to find an x.

We're building this LBA. We're never going to run that LBA. We're designing that LBA not for the purposes of running it. We're building that LBA only for the purpose of using the LBA language emptiness tester.

We're going to build that LBA and feed it into the Turing machine R, which is going to tell us whether that LBA's language is empty. We, ourselves, is never going to run that machine. We're never going to come up with an x. We're just having-- we are designing a computation checker.

And then the emptiness tester for that that we assume to exist for ELBA is going to say, yes, there is some computation which this machine accepts, or no, there is no computation which this machine accepts, and that's going to be a computation for M on w. And so that's going to tell us whether or not M accepts w. So we're never going to actually be finding an x. We're never going to be running that LBA. We're just feeding, using that LBA as an input to R.

Does the computation history method always use LBAs? No, as you will see right after the break. Because the LBAs is just kind of an easy place to get started, but we're going to use the computation history method and computation histories next on the post correspondence problem.

Yes. The computation histories and the input x are always going to be finite. So that was an answer to a question about whether these histories or the inputs are going to be finite or not. So those strings are always going to be finite, and the computation history is going to be finite. So we are at the end of our five minutes. Let me just see if there was another question I wanted to answer.

Let's move on. OK, now coming back to the undecidability of this post correspondence problem. So remember the post correspon-- this problem, you're given those dominoes. You want to know if there's a match. Here is a little mini version of the diagram, if that helps you remember it.

And we're going to prove that this language is undecidable. And so it's undecidable to test whether you have a match, given a set of dominoes, whether a match is possible. And we're going to use-- we're going to reduce ATM to this language using the computation history method. So how in the world are we going to do that?

First of all, there's a little detail here I want to mention. Just don't focus on this, but I'm going to assume the matches always start with the very first domino on the list, the very first domino in the collection. So there's going to be, like, a starting domino. Just going to make my proof a little simpler to do it that way, and then you can fix that assumption later.

And if we have time at the end, I'll do that or maybe after the lecture is over. If there are questions, I can show you how to fix that assumption. But for now, to simplify the proof, we're going to assume that there was a starting domino, that the matches always have to start with that particular domino. If you didn't follow that point, don't worry. You can just ignore it. You'll see where it comes up later.

So now we're going to reduce ATM to the PCP. So assuming we have a machine that decides the PCP, we're going to use that to make a machine that decides the ATM as before. So here is the Turing machine S, which is going to decide ATM.

And the way we're going to do it is like this. We're given M and w. We want to know, as always, does M accept w?

What we're going to do is we're going to build an instance of the PCP problem. So we're going to build a collection of dominoes which are going to-- and that collection is going to be built knowing M and w. So that's going to affect the dominoes we're going to create. So this collection of dominoes is going to be called P of Mw. Depends on an M and w.

And finding a match for this set of ws is going to force you to simulate M on w because the match is going to correspond to a computation history for M on w. So we're going to use-- once we do that, once we build the set of dominoes where a match corresponds to a computation history, we're going to use R to determine whether or not there is a match. Or in other words, whether or not there is a computation history, which is whether or not M accepts w. So if there is a match, we're going to accept because we know M accepts w. And if there is no match, we're going to reject because we know M does not accept w.

OK. This is the plan. I haven't told you how to do this yet. I mean, I'm worried about the significant number of you who are feeling confused by the method. I mean, you guys should be texting me and the TAs to try to at least get a sense of how this is working. I mean, we're going to be going-- we're going to be doing the method again. It's just going to be a little more complicated because we have to also deal with these dominoes and all that stuff, and that's why I presented it to you the first time in the setting of the LBAs, which were, in a sense, the method comes out perhaps more simply.

So a match is going to correspond to an accepting computation history. Sometimes if I don't use the word accepting, I'm just being a little sloppy. Computation history and accepting computation history, for us right now, they're the same. I mean, later on, we may actually talk about rejecting computation histories, but let's not get ourselves confused. Computation histories always have to end with the machine accepting.

OK. Somebody's asking, what does it mean for match to correspond to a computation history? You'll see. That's going to be on my next slide.

Oh. So this is a good question. Is my step two trying to use R to determine whether M accepts w? Well, yes, but I'm doing that by testing whether these dominoes have a match. Because R decides PCP. I can only use R to test whether things have a match.

Someone asks, should step two be to use R to determine whether M accepts w? Well, in effect, that's what it's doing, but it's doing indirectly through testing whether this PCP-- whether these dominoes have a match. Because those dominoes are going to force you to simulate M on w. OK, I think I'm repeating myself here, so let's avoid getting into a loop on the lecture and see how we actually build P of Mw.

So now you understand what-- you have to have the plan in your mind. We are given M and w. We're trying to make a set of dominoes where a match is going to be a computation history for M on w. So the string you're going to get in that match, the top string and the bottom string, which are going to be-- have to match, they're going to be computation histories-- they're going to be a computation history of M on w. So I want to figure out how to make my dominoes force that.

OK. So my starting domino, as I told you, there's going to be a special starting domino in my collection, which is going to require the match to start with that domino. It's going to be this one here. It's going to have these two strings in it. And you can see already, it's starting to look like a computation history. The dominoes are going to have that feature.

So it's the pair of strings, and I've written it kind of to help you see what's kind of going on. It's a pound sign on the top and the starting configuration for M on w on the bottom. And what I'm going to do for you here is at the bottom of the slide, I'm going to take the dominoes I've written down so far and try to be building a match, and you'll see how that match is forcing a simulation of the machine.

So let's take this as an illustration. Let's assume the input to M, so I'm running M on w now. I'm trying to see, does M accept w?

w is a string 223. So the start configuration for M on w is q0, that's the starting state for M, and then w following it is 223. That's what is going to appear on the tape of the Turing machine to start off. So this is the start configuration for M on w, assuming that I had this particular input string to w.

And so given the dominoes that I've given you so far, just one, this is how the match is going to start. Now, there's going to be more dominoes coming. So for every possible tape symbol and state, don't get confused by the language here, this is the important thing.

If in the Turing machine-- and I'll just read this in English to you. If the Turing machine, when it's in state q, and the head is reading an a, it moves to state R, writes a b at that point, and its head moves to the right. I'm focusing on rightward moving Turing machine steps right now. But for every possible state and tape symbol and looking at what happens, I'm going to have a domino that's going to capture this information, and it's going to be this domino here.

q a on top, b r on the bottom. So just a string. q a on the top, b r on the bottom. So let's see why? Well, that's sort of arcane-looking. Why is that a good thing to-- why is that a useful domino to put in here?

Well, if you take a look, let's assume my Turing machine, when it's in state q0, which is the starting state, and the head is reading a 2, which it will happen to be reading because the string w starts with a 2, if it goes into state q7 and writes a 4 and then moves right, I'm going to have-- in this domino, I'm going to have q02 on top and 4q7 on the bottom because 4 is what I-- right here, that's the b, and the new state that I'm going into is q7. So that's going to be the domino that I'm going to get. And here it is. Here's that domino appearing in the match.

So it's going to match up with q02, which don't get the-- the top string has to equal the bottom string. So and this is going to be the only choice that I have for extending the match. So q02 is going to be on the top when I put that there, but that's going to force 4Q7 to appear at the bottom because that's what's going to be the bottom string corresponding to the q02 on the top. OK?

So if you're looking down here, this is the beginning of the second configuration of the machine. That's what I want to be happening. I want to be-- that match should look like a computation history. So this is going to be-- all possible right moves are going to be handled in this way, and it's going to be a similar process for the left moves, but let me not-- I'll leave that to your imagination.

Now, how do I continue on from there? What does the rest of this configuration look like? Well, that should just be a copying over of the remaining symbols that were on the tape because they don't change. Things only change around the head. The rest of those symbols, which is the 2 and the 3, those should just get copied over here.

So I'm going to have two additional-- I'm going to have additional symbols in my-- dominoes in my collection. So for every tape symbol, I'm going to have aa be a domino. So for every tape symbol a, I'm going to have aa be a domino in my collection. And so that says I can have-- so there's going to be a 2 2 domino. So I can match up this 2 over here, but that forces me to put a 2 down over there.

There's also going to be a 3 3 domino, which is going to match up with that 3, but it's going to force me to put a 3 down over there. That's the only way I can extend this match that I built so far. I have no choice. Those are the only dominoes that I'm going to be given. And so doing, I'm forcing you to basically simulate the machine.

Now, what I want to have happen next is a pound sign to appear here, and that will conclude my second configuration. So there's going to be a pound sign pound sign domino as well because there's a pound sign here. It's going to get matched with the pound sign up top. Forces a pound sign to come down on the bottom.

And if you look at the way we are right now, we're exactly like where we were at the beginning when we had just this first domino appearing. But now we're one configuration later. So if you understood that, and I admit that it's-- there's just one idea here.

And once you get the idea, it's all trivial. It's all very simple. But there's just you have to get that idea. I'm trying to figure out how to get that idea into your head. Once you get the idea, you can write all this stuff yourself.

So following this description of how the transition function of the machine works and copying over the symbols from the tape to the next configuration, I'm going to be able to get configuration after configuration going until I get to a point when there's an accept. And now, from the machine's perspective, we're done. The machine has accepted its input. This is our computation history.

But is it a match? Well, up until-- this bottom thing is the computation history, but it's not a match because the top doesn't equal the bottom. There's still extra stuff at the bottom, which the top, it doesn't have.

So what I'm going to need to do now is add some additional kind of pseudo steps of the machine where I'm going to allow the top to catch up to the bottom. And the way I'm going to be thinking about that, and this is not real for a Turing machine as much as the Turing machine is real anyway, but you're going to imagine I'm going to add a new kind of move to the machine which is going to allow the head to eat the symbols off the tape like Pac-Man. OK? And the way I'm going to get that effect, on the top, if I have any tape symbol next to a qaccept on either side on the top, what I'm going to get you to write on the bottom is just qaccept with that tas-- with that tape symbol gone.

And by repeating that move after move, the actual tape is going to be shrinking. The symbols on the tape are going to be going down one by one, move by move, until there's nothing left except just the qaccept all by itself. OK?

So I have this sort of Pac-Man idea. I'll be eating the symbols on the tape. And then finally, I get to a point when there's just the qaccept alone on the tape. There's no symbols left to consume.

And then I'm going to add one last domino here, which is qaccept pound pound matching with just pound, which just conveniently finishes off the match. And so the match is completed. This is actually a little detailed here.

I hesitate even to bring it up because I think it's the kind of thing where if you understand everything up to this point, you could fill it in yourself. But just for completeness' sake, you have to deal with the situation when the head of the Turing machine might move into the blank portion of the tape, which is not taken into account here. And the way we get that effect is by having another domino here, which allows me to add blank symbols at the bottom as needed. That's just a detail. I'm more worried that you understand the underlying concept.

So here is going to be a check-in. I'm trying to remember-- OK. But so OK. So what else can we conclude from this information?

So we know-- at this point, we know that PCP is undecidable. That's what we just finished proving. What else do we know, if anything? Or do we even know that? Let's see. My picture is a little bit covering the check-in, but it's still readable.

So I mean, this is not an easy concept to get the idea. But once you get it, you'll see it's not that bad. OK. Another 15 seconds.

So this portion-- this question is not really relying so much on the computation history method. It's just relying on the fact about PCP being undecidable. OK. Everybody almost done? Five seconds. All right.

So I can see that there is some level of confusion. So the reason why B is correct is that we know PCP is undecidable, but we also know that it's recognizable because you could try all possible ways of combining dominoes and one after the next, except if you ever find a match. So that might go forever. But if there is a match of a possible, you'll find it eventually.

So PCP is a recognizable language. Undecidable. And so therefore, we know its complement has to be unrecognizable because that's something we've shown before. If a language and its complement are both recognizable, then it's decidable, but we know PCP is not decidable, so both sides can't be recognizable.

OK. So let me prove to you one last theorem that's going to be useful for your homework involving the computation history method. So you'll have one last chance to get the way we're going to use this, though this one is going to be a little bit more similar in spirit to the LBA version than to the PCP version, which has this extra kind of complication about coding the computations of the machine into dominoes. Here, we're going to operate with another automaton which where it's going to be more straightforward.

But anyway, OK. Getting ahead of myself. So if you remember, for context-free grammars, we had the ECFG problem, the emptiness problem. We showed that was decidable. So testing whether a context-free grammar's language is empty is decidable.

However, testing whether a context-free grammar's language is everything, whether it's equal to sigma star, that turns out to be-- that turns out to be undecidable. OK? So emptiness testing for context-free grammars, decidable. Sigma star testing, undecidable.

OK. So we'll show that ATM is reducible to old PDA via the computation history method. And so assume we have same patterns. Assume we have a decider for all PDA and make a decider for old t-- decider for ATM.

Here's the ATM decider. And now, similar to what we did for the LBA case, but with a twist. We're going to make a pushdown automaton that's going to check its input to see whether it's an accepting computation history of M on w. Just like the LBA did, if you think back to how that worked.

Remember, the LBA tested its input to see whether it's an accepted computation history. Accepted if it did, and then we test the LBA's language for emptiness to see if there are any computation histories. So we're doing the same kind of thing, except now, the PDA is going to test its input to see whether it's an accepting computation history, and if it is, it's going to reject.

It's going to do the reverse of what the LBA did. And that's going to turn out to be necessary. But for the moment, let's just go with it, and maybe we'll see it in the proof, why the proof needs it to be this way.

So this pushdown automaton is going to accept its input if it's not in accepting computation history for M on w. Otherwise, it will accept. So you can think of this PDA as accepting all junk. It just doesn't accept the good stuff, the accepting computation history. OK? It's accepting all the things which fail to be an accepting computation history. OK?

And then once we have that, we test whether R's language is everything, whether BMw's language is everything using R. Because if that PDA's language is everything, then we know there could not be an accepting computation history because that's the one thing that gets not accepted. That's the one thing that gets rejected. So if it's accepting everything, there is not going to be an accepting computation history. So if there is no-- if this is equal to sigma star, then there couldn't be an accepting computation history, and so we're going to accept.

OK. So how is this going to work? So what's different now about this case from the LBA case is remember, the LBA got the computation history on its input, and it used its ability to write on the tape to check that each configuration followed the next one. We don't have the ability to write on the tape in a pushdown automaton.

And by the way, I hope you're all comfortable with my using PDAs instead of grammars because we can interchange one to the other. Should have mentioned that when I did it. But so the PDA is going to be using its stack to compare each configuration with the next one, OK?

So the way that's going to happen is it's going to non-deterministically take one of these-- so the very first thing it does is, as before, it checks to make sure that the beginning of the input is the start configuration. But then once that's-- that's the easy part. But once we get going with that, we push each configuration-- well, first of all, we non-deterministically choose which configuration might be the one that fails where one fails to go to the next one. Because those are the ones we're trying to accept, when there's a failure.

So we're now determined to simply look for the place that there's a failure. We push that onto the stack, and then we pop it off the stack and compare it with the next configuration. So that's how we're using the stack instead of being able to mark on the input.

Now there's a-- so I'm kind of going to illustrate that here. So as we're going to read this thing here, I'm going to put it onto the stack. So this thing moves over to here, and this input here got put onto the stack.

q0, w1, w1, it's q0, w1, w2. You see that. That first configuration is now sitting on the stack. Now, as we're going to pop it off, we're going to match it with the second configuration.

Now, if you're following me, you'll realize that there's a difficulty here because it's coming out in reverse. It comes out in the reverse order that we put it in, and that's not what we needed to do. So what we're going to do here, and here's a little sort of a twist, we're going to change the way we're writing down computation histories by making them-- by writing them-- reversing the even-numbered ones. So this one here is going to be written down in reverse.

And that's perfectly OK. We can write down computation histories in any way we want to meet our needs. So we're going to reverse-- we're going to reverse the alternate ones. And now, when we push one, it's going to come off in the right order to compare with the next one. And so that's how the procedure works. OK?

We're running a little short on time. You need to be able-- let's see. So let me just go to a quick recap.

The computation history method is useful for showing the undecidability of problems when you're testing for the existence of an object. Each of those four cases that we showed today involved in testing whether something exists. So is there an integer solution to the polynomial? Is there some string in the language?

Is there string that's not in the language? Or is there a match? So that's a typical case when this computation history method comes up.

And so as a quick review, these are the things we showed today. OK, so why don't we-- we're just out of time. And so I'm going to let you go, but I will stick around for another 5 minutes or 10 minutes. Happy to answer any questions if you have them, but that's officially the end of the lecture.

OK. So let me try to get to-- there's a lot of questions in the chat. Don't forget, write to the TAs too. So if M does not accept w, what will happen to the computation history? So if M does not accept w, then there is no computation-- there is no accepting computation history. That's the only kind of competition histories we're considering.

So if M does not accept w, there is no accepting computation history. There's no computation history. So I don't know what it means, what will happen to it. It just doesn't exist. So I hope that's helpful.

Don't we need to be able to add states to the end of the tape? Hm. I don't know what that means. Add states to the end of the tape. I don't under-- you'll have to repeat that one, sorry, or explain that better.

Ah. So this is a go-- this is a very good question here. Where does the previous proof fail when trying to reduce ATM to ECFG? It's got to fail because ECFG is decidable. So that's a very good question. Maybe we can just go back to that last slide here.

Because I was running a little short on time, I didn't really focus on why we have to accept the non-computation history strings. Why are we accepting all the junk strings and only rejecting the strings which are the computation histories? So we're looking for strings that fail.

A string could fail because it starts wrong. It doesn't have the start configuration. Or it maybe doesn't end with the accepting configuration. Or maybe one configuration doesn't lead properly to the next one. Any of those cases are a failure, and are going to be a reason to accept.

The reason why we can do that is because we're taking advantage of the pushdowns non-determinism. We don't know where the failure might occur, so we're going to non-deterministically guess where that failure occurs. If we were going to flip this around and say let's accept only the good ones and reject everything else, we would have to test that each configuration led to the next one. So we'd have to check them all.

So if something fails, it only has to fail in one place, and then we can accept. If it's a good computation history, we have to-- it has to be good everywhere. And so the pushdown automaton, really, if you're going to get down to the nitty gritty, the pushdown automaton can check that this configuration leads to that configuration pushing on the stack and then popping it. But now, by the time you get to here, you want to check that this second configuration leads to the third one. You would need to push the second configuration onto the stack, but at this point, you're already after the second configuration, so we cannot back up and push the second one on the stack.

So the pushdown automaton is not able to check in the positive sense that you have a computation history. It's only able to-- and accept them. It's only to check in the negative sense that you don't have a computation history and accept all the ones that fail. And that's just because it only has to fail in one place.

I hope that helps. So that's why we couldn't flip this around and make this proof work for the emptiness problem and only have to work for the everything problem, the all problem. All right. OK, so bye-bye everybody, and I will see you on Tuesday.