[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: All right, why don't we get started? It's 2:35, time to begin. So everyone, welcome back-- good to see you all. Well, at least a few of you-- glad that you're here. So let's see. Why don't I get a layout, as I usually do, where we have been, where we're going today. I'll talk a little tiny bit about the homework, because I've gotten a couple of questions on that. And then we'll jump in.

So we have been talking about Turing machines, which is going to be our preferred model for the rest of the semester, since Turing machines are the model which we use to capture general purpose computing. And we looked at Turing machines in various variations on Turing machines-- multi-tapes, non-deterministic, and so on. It's a bit recapping the history of the subject, when people looked at a variety of different ways of formalizing the notion of algorithm.

And they showed that all of those different formalizations were equivalent to one another, which led to the Church-Turing thesis that all of these models-- each of these models really capture our intuitive idea of what we mean by a procedure or an algorithm for-- at least for addressing things like mathematical problems and precise problems of that kind. So we talked about the Church-Turing thesis. We also talked about a notation for encodings and Turing machines. We'll review that briefly.

So today we're going to give a bunch of examples of Turing machine algorithms for solving a variety of different problems. I should just say algorithms really-- nothing special about Turing machines. Turing machines are just going to be our formal counterpart. But from now on, we're going to use our Church-Turing thesis in a way to just talk about algorithms in general, because that's really our interest. Turing machines is just our way of reasoning about them mathematically, but we're really ultimately interested in understanding algorithms.

In terms of the format of today's class, I'm going to try a little experiment. We're going to have little breaks along the way, as well as the big coffee break in the middle. We're going to have a little mini breaks, because I sometimes feel that there really isn't enough time for people to be writing chat-- questions in the chat, because things just are racing on. And so this way we'll have a little break after the-- pretty much after each slide. Some of them are going to be a little longer than others. In case you have questions, you can pose them to me or to the TAs, and we'll try to get back to you on those.

So moving on then, so today, just as a quick review, Turing machines, as we set them up, they have-- on any input w, they have three possible outcomes. The Turing machine can halt and accept w, can halt and reject w, or it can go forever on w, which is rejecting by looping in our language. A Turing machine can recognize a language, the collection of old strings that it accepts.

And if the Turing machine always halts, we say it decides the language, and it's a decider, and so therefore, that language is a decided language, a Turing decider. Often we just say decidable, because we don't really have a notion of deciding in other models, so we often talk about Turing-recognizable or just decidable. Or we'll sometimes just say recognizable, when we understand that we're talking about Turing machines.

We briefly talked about encodings. When you write it inside brackets, some objects-- whatever they are-- could be strings, could be machines, could be graphs, could be polynomials-- we're representing them as strings, and maybe a collection of objects together represented as a string in order so that we can present that information as an input to a machine. And we talk about-- our languages our collections of strings.

And a notation for writing a Turing machine is going to be simply that English description put inside quotation marks to represent our informal way of talking about the formal object that we could produce, if we wanted to. But we're never going to ask you to do that. OK, so now let me see. Let's just take a quick break, as I promised. If there's any quick questions here, you can ask.

Let's see-- got a lot of questions here. All right, why don't we move on? And some of these things are good questions. Somebody asked, can you-- how do you tell if the machine is looping or if it's really just taking a very long time? You can't. That's what we're going to prove not in today's lecture, but going to prove that on Thursday. You just can't tell. So if that's in reference to problem 5, you're going to have to find some other way of doing-- solving that problem without knowing whether the machine is actually ever going to halt.

And somebody asked, how can we tell if an English description is possible for a Turing machine? Well, you can always write down an English description for any machine. It's-- might be very technical looking, but if you can write it down in a formal way as states and transitions, then you can simply write down an English description, which would just be, follow those states. OK, let's move on.

OK, this is going to be our first example of a algorithm that's going to answer a question about automata. And it's a very simple problem. It's a very simple problem to solve, because we want to start out easy. The name of the problem is the acceptance problem for deterministic finite automata, acceptance problem for DFAs. And I'm going to express it, as I always do, as a language.

So with this language here, I'm calling it a ADFA-- which stands for the acceptance problem for DFAs-- is the collection of pairs. B and w-- B is a DFA. w is considered to be some other string, which will be an input to B. We're going to be thinking of it as an input to B. I put the two of them in brackets to represent the pair of them as a single string. We're not going to make explicit what the form of the encoding is.

The only important thing is that the encoding should be something simple, but that the Turing machine can decode back into the DFA and this input string to that DFA. So anything reasonable is going to be a satisfactory encoding from us-- for us. So this is an encoding of the two of them into a string, and where B is a DFA, and B accepts w.

So now, if you want to test if something's a member of ADFA, then, first of all, you want to make sure that the string itself that you're getting really encodes a DFA and a string. So it has to be the right form. And once you know that, then you have the DFA, you have the string w, and you're then going to do the obvious thing, which would be to simulate B on w and see if it's actually accepting w.

So that's what the content of this slide is. I'm just going to write it down for you. So I'm going to give a Turing machine, which I'm going to call-- the name of that Turing machine is going to be D A DFA. To help you remember the function of this machine. This is a decider for the language below, the ADFA language. This is just a name, but-- so nothing fancy going on here, but just to help us remember, because I'm going to refer to some of these Turing machines later on.

So this is the decider for ADFA. And I'm going to describe that machine for you, and that machine decides the ADFA language. So what does that mean? So that machine-- I'm describing it now in English, as I promised. We're going to take an input string s, and first, it's going to check that s has the right form, as I mentioned-- has the form which is the encoding of a DFA and a w. If it's not of that form, the Turing machine should reject that input right away.

Now, I'm not going to go through the details of how that Turing machine is going to work, though I'll say a little bit more this time only just to give you a sense of how it actually might carry that out. If you don't believe that you can do it with a Turing machine, believe you could do it with your favorite programming language. That's good enough, because that's going to be-- that's equivalent to a Turing machine.

So first, you check that the input's of the right form. Then you're going to simulate the computation of B on w. And then, if B ends up in an accept state, then we know B is accepting the input, and we should accept, and we do. If B does not end up in an accept state at the end of w, then we should reject, because B is not accepting w. OK? That's my description of this machine.

Let's just turn to a little bit of details just to make sure we're comfortable with this. So here is our Turing machine-- D, the decider for ADFA. The input to that Turing machine, as I mentioned, is going to be B and w, provided it's of the right form. So that's what this string S is supposed to be of that form. And what does that mean? It's just an encoding of the machine w and the string-- the machine B and the string w.

Let me just write it down. Here is B written down in some just completely explicit way, where you're just listing the states of B, the alphabet of B, the transition function as a table, the start state, and the set of accepting states-- just writing it down explicitly, as you might do it if you would just want to describe that machine in a completely formal way, and then writing down with the string is-- whatever it is.

Once DADFA has that as its input, it can then go ahead and do that computation. And just to try to make it a little bit more explicit, I'm going to capture that here by saying, let's give that Turing machine an extra tape, because we already know that the multi-tape model is equivalent to the single tape model-- make our life perhaps a little easier.

And in the course of doing that simulation, what do we want to keep track of? Well, what is the current state of B, the DFA, as we're reading the symbols of w? And where in w are we at right now? So I'll call that k, which is the input head location on w. How many symbols of w have we read? OK? And that's all I'm going to say about what this Turing machine D looks like.

Oh, there's one more thing I do want to say for the-- that's coming up, because pretty much all of the Turing machines that we're going to talk about today and going forward are going to often want to check that their input is of the correct form. I don't want to repeat this every time, because that's going to be assumed. So my shorthand for that is to say my input is of the form I'm looking for, and that has built into it the check that the string-- the input string is of that form, and we're going to reject if it is not.

So all of our Turing machines are going to start out, on input, the string is of a certain form, and then go out and do something with it. OK? OK, so let me try to answer a few of the questions that I've gotten here, because I think this is important as a way of getting us all started.

Now, somebody asked me, can we use arguments of this form? Somebody asking, can we use-- can we give our description of a procedure, if I'm understanding this correctly, as using some other programming language? Well, typically, you just want to make sure you're understood. If you want to do that on a homework, I wouldn't advocate writing your algorithm in Java, because it's going to be hard to read.

But write it down in some pseudo programming language if you want, just to make sure it's clear that-- what you're doing. Probably English is going to be the easiest for you-- even though this person says-- feels it would be easier to do it in terms of a formal language. Well, whatever's easier-- as long as we can understand it.

This is a good question here I got to ask. What if B-- here's our B-- gets into a loop on w? Well, that's not going to happen. B is a DFA. DFAs-- they move from state to state every time they read a symbol of w. When they get to the end of w, it's the end of the story. There's no more computation to be done. So we know in exactly how many steps B is ever going to take on-- it's going to take the same number of steps as the length of the input. That's how many moves it gets to make. B as a DFA never loops. So that would be a problem if it did loop, but it doesn't. That input never does loop.

So have we verified that D is a decider? Well, I think I just did. From my standpoint, we've said enough to be sure that D is a decider. There's never any reason for D to be-- for that DADFA Turing machine to be getting into a loop. The input head location is referring to where we are on the string w? Yes. And somebody's asking me, is this the level of detail for the homework? Yes. That's all I want. It's all I'm looking for. OK?

Let us move on. I'm going to have to-- otherwise we'll never get anywhere. There are a lot of questions here. They're good questions. So why don't we go on? Some of these may get resolved as we're going to look at additional examples, because that's all of today is pretty much examples. Let's talk about the similar problem, the acceptance problem, but now for NFAs.

So now, actually, NFAs can loop, so we have to think about what that possibly could look at. But before we get ahead of ourselves-- so now the acceptance problems for NFAs looked very similar, except B is going to be an NFA. That's going to be a decidable language too. And now we have Turing machine A-- DANFA, the decider for ANFA that decides ANFA. OK?

So now, as promised, here's our new form for writing our Turing machine. On input B, w, we're assuming that B-- based on the context, sometimes you may want to say at this point, what B and w are. But from the context, we know what they are. They're going to be an NFA and an input w for that NFA. I do want to jump into the solution. First, before we actually look at the solution of this, we-- the Turing machine could simulate the NFA on input w.

And you have to be careful on that simulation that you don't end up looping, because don't forget, an NFA could have epsilon moves, and could be looping on those epsilon moves. And so that would be a problem, if you're not careful about how you do that simulation. Now, I think, if you were going to simulate an NFA, you would be-- wouldn't follow loop around loops forever.

And I think you can-- without getting-- because this is not the way I'm going to solve the problem-- you could find a way to avoid getting caught and getting stuck in loops for an NFA. So even though that looks like it could be a problem, in terms of looping forever, it turns out that it won't be a problem-- it wouldn't be a problem if you're careful. But I'm not going to solve it that way anyway, because I'm going to illustrate a different method for solving this problem.

And that is we have exhibited before a way of converting NFAs to DFAs. So my Turing machine is going to solve the ANFA problem by converting its input, which is an NFA, to an equivalent DFA. I'm calling the NFA B and the DFA that I got-- converting it into B prime. And what's nice about that is that, first of all, we already know how to do that conversion, because we essentially went over that in lecture a few lectures back, and it's spelled out in full detail in the textbook. So that is a conversion we know how to do-- we'll assume we know how to do. And we can implement that Turing-- that conversion on a Turing machine.

Then, once we have the equivalent DFA, what do we do with that? In the previous slide, we showed how to solve the problem for DFAs. So if we can convert the NFA to a DFA, and then we already know how to solve the problem for DFAs, then we're done. So that's how I'm going to say. We're going to convert the NFA to a DFA, and then I'm going to run that DADFA problem on the new machine that I produced.

So remember that the-- this machine here decides the ADFA problem. And now I'm going to accept, if that new machine-- if that previous Turing machine accepts the DADFA problem the machine accepts, and I'm going to basically do whatever it does. If it accepts, I'll accept. If it rejects, then I'll reject.

So I guess the thing that this illustrates is this idea of using a conversion construction inside a Turing machine, and then a previously constructed Turing machine basically as a subroutine. All this is perfectly legal, and it's the kind of thing we're going to be doing a lot of, and that you should be used to doing that-- get ready to be doing that on your homework too. And in fact, I'll give you another a little extra hint that problem 6 can be solved in this way.

OK. All right, so here, let's pause briefly. OK, got some interesting questions here coming up-- somebody asked me, do we need to be explicit about how we're going to simulate that NFA or the DFA? Because we don't know how many states it has. You do know how many states it has. Once it's presented to you on the input, you can see, oh, this is a machine that has 12 states-- because you're given the machine, and then you can do the simulation.

Let's see. And someone's asking me about limits on the simulation power of a Turing machine. That's a question that I'm going to postpone to later, because you're asking if a Turing machine can simulate itself. And we'll talk about things like that, but that's in a few weeks from now, so I'll hold off on that one. Decidable languages-- somebody's asking me a good question about closure properties of the class of decidable languages. Are they closed under intersection, union, and so on?

Yes. And the decidable languages are also closed under complement. That should be something you should think about. But yes, that is true. The recognizable languages, however, are closed under union and intersection, but they are not closed under complement. So that we will prove on Thursday's lecture.

So another question is, could we have just run B on w in this-- in solving this problem, instead of using reduced-- converting it to a new Turing machine-- the B prime? Well, yes, we could have just done that. OK. I think we better move on. I don't want to get too bogged down here-- got a lot of questions there, so sorry if I'm not getting to all of the questions. OK. Or you can write to the TAs too, obviously. I'm sure you are.

OK, so let's talk about a different problem-- emptiness problem for DFAs. I'm going to give you now a-- just a DFA-- B-- and no input. And I'm going to ask, is the language of that DFA the empty set, the empty language? You understand the problem, first of all? I'm just handing you in a DFA, and I want to know, does this DFA accept any strings at all, or is it just some dumb DFA-- it's just always very negative DFA, it just rejects everything-- and it has the empty language?

How do you tell? Not super hard, if you think about how you would write a program to do that or how you would do it yourself-- so that's a decidable language again. So we're going to give now a Turing machine decider for that language. That decider says, well, I'm giving that DFA-- I want to know if its language is empty. And the idea is just what you would think.

I'm going to see, is there a path from the start state of that DFA to the-- an accept state of the DFA? If there is such a path, then that DFA is going to have an input which goes along that path, and will be accepted. And so the language won't be empty. If there's no such path, then clearly, this DFA can never accept anything, and so this language will be empty.

OK. Now, there are many different path algorithms. I think it would be a little bit sparse of me just-- some of you know algorithms. Some of you don't know path checking algorithms. I would like you to just to-- if you were giving this kind of an answer on a homework, to give me some sense of how that algorithm is going to work. Don't be too high level about it.

So the one I have in mind is the breadth research, if you've heard of that. But it's very simple algorithm. What I'm going to use is kind of a marking procedure. So I'm going to start by coloring the-- here is this-- I should have indicated-- this is my DFAB. This is B over here. Should I try taking a chance of writing on this thing-- oops. This is B. Oh, great-- that didn't help.

So this is B here. And the way I'm going to test if it has a path-- if it accepts an input is by seeing if there's a path from the start state to any one of the accept states. And I'm going to start it by marking the start state, and then marking all states that I can get to from previously marked states, and keep doing that until I can't get to anything new. There's going to be a series of iterations here marking more and more states until there's nothing new to mark, and then I say, did I mark any accept states?

OK, so let's just see how I write that down in English. So I started marking the start state. I repeat until no new state is marked. I'm going to say, mark every state that has an incoming arrow from a previously marked state. Accept-- then, once I'm done with, that repeat loop-- accept if no accept state is marked, because that means-- don't forget, it's a little bit inverted from what you might think. I'm going to accept if there's no marked accept state, because that means there's no path to an accept state from the start state, which means the language is empty.

And EDFA is the DFAs that have empty language, so I should be accepting those. If there's no way to get to an accept state, no accept state is marked. And reject if some accept state earmark, because then the language is not empty. OK, so that's all I wanted to say about that.

Somebody's asking, can you just say something like breadth versus [AUDIO OUT]. The sketchier you are, the more chances that you're going to get caught by the grader, who's not going-- we have an army of people grading these problems, and just to stay on the-- be on the safe side of sketchiness and, say, don't cut too many corners, because you might miss something. Chances are it would be OK just to say breadth research, but I would prefer if-- you'd be safer if you said a little bit more than that. OK.

Oh, this is a good question here. Somebody asked, can we just run the DFA on all short-- on all strings? Well, first of all, one thing-- to say something bad would be, well, just feed in all possible strings into the DFA, and if any one of them-- if it accepts any one of them, then we know its language is not empty. Well, that's not a good algorithm, because that's going to potentially run forever, because there's lots of strings. There's infinitely many strings to feed in to that DFA. And so a Turing machine, if it's trying to be a decider, had better not do any infinite operations that are potentially going to go forever.

To be fair to the proposer here, the questioner here-- didn't ask that-- says, well, can we feed in all strings up to some bound, which would be the number of states of the machine in this case? And yes, that would work, but then you would have to argue that that's enough. And yes, it is enough, but it wouldn't-- would not be enough in answering the problem just to say, feed it in that number of them, and we're done. You would have to say why. OK.

Lot of questions here-- I'm going to move on. Let's see. Equivalence problem for DFAs-- now we're going to take things to the next level-- ask, are there two-- I'm going to give you two DFAs. And I want to know, do they describe the same language-- do they recognize the same language? OK? So how are we going to do that? So that's a decidable language. Here's the decider.

My input now is going to be two DFAs-- again, represented as a string, because that's what Turing machines deal with as their inputs. But they can unpack that string into two DFAs. And there are several different ways to do this problem, and I'm sure I'm going to get suggestions with other ways to go. One thing you could do is just to feed in strings up to a certain length.

Just like before, you can't feed in all possible strings and see if the machines ever behave differently on any of them, because that's an infinite operation, and we already decided we can't do that. Now, if you want to talk about this being a recognizer, instead of a decider, then you might be able to do something like that just to make sure your-- you have to be just careful. Let me not say more on that right now.

But certainly, for a decider, you can't go forever. You can't have infinite operations. So you would have to have a cut-off. So you can feed in all possible strings up to some length, say, into A and B, and see if there's any difference. Now, we actually had a problem on that in the problem set 1, which said, if two DFAs have unequal languages, then they're going to see a difference. Then there's going to be a string which acts differently on them, which is of length, at most, the product of the number of states of those two machines.

So you can either reference that problem-- that would be an adequate solution-- or reprove it or whatever. That would be fine. In fact, you can do even better than that, as the optional problem showed. You only have to go up to the sum of the number of states, not up to the product. But that's actually very difficult to show.

I'm not going to prove it that way. I'm going to prove it in an entirely different way, which doesn't require any analysis at all-- no proving something about balance. I'm going to take advantage of something we've already shown, which is I'm going to make a new finite automaton, a new DFAC built out of A and B, which accepts all the strings on which A and B disagree. And I'll show you how to-- that's easy to do.

So first of all, let's-- in terms of a picture, let's understand what this is. So here we have-- this is the language of A, this is the language of B written as a Venn diagram. And where are those places where they disagree? Well, they're either in A, but not in B, or in B, but not in A. I'm showing it to you here in terms of the blue region.

That actually has a name called the symmetric difference of these two sets. These are the-- all of the members which are in exactly one out of the two, but not both. If you can make a machine C that would accept all of the strings in the blue region, then what do we do with that machine? We test of its length language is empty, which is what we've already shown how to do-- because of the blue region is empty, that means that L of A equals L of B.

So I'm going to make a machine, a DFAC where the language of C is exactly that symmetric difference. It's all the strings in A intersected with the strings that are not in B-- so in A and not in B-- or in B and not-- then not an A-- take the union of those two parts. And how do we know we can make C? Well, we have those closure constructions, which we showed several lectures back. Those closure instructions can be implemented on a Turing machine.

So a Turing machine can build the DFAC, and then use that test from a few slides back, the emptiness-- the last slide-- the emptiness tester for DFAs on C to see whether its language is empty. And now, if C's language is empty, then we know we can accept, because that means the two-- that L of A equals L of B. Otherwise, we can reject. OK? So here's a note-- I'm going to ask you a check-in. You can also use that time to send me a few more questions, if you want.

OK, here's my check-in. OK, now, instead of testing equivalence of finite-- of DFAs, I want to test equivalence of regular expressions. So here are R1, R2. Regular expressions are called the EQ regular expressions problem-- equivalents of regular expressions. Can we now conclude that this problem is also decidable, either immediately from stuff we've already shown; or yes, but would take some significant extra work; or no, because intersection is not a regular operation?

So let's see if I can pull that out here-- launch the polling. Here we go. OK, I think we're just about done here. Five seconds more-- OK, ready, set, end. All right. Yes, the correct answer is A. We have basically already shown this fact, because-- why is that? Because we have shown that we can convert-- if you're given two regular expressions, and we want to test whether they're equivalent, they generate the same language, we can convert them both to find out automata.

We can convert them to NFAs, and then the NFAs to DFAs. And then we can apply what we've just showed about testing equivalence of DFAs. So yes, it really follows immediately from stuff we've already shown-- the conversion, number one, and then the testing of equivalence for DFAs. So there's not really any work to do here. And in fact, what I'm trying to illustrate with this check-in-- that, once you've shown how to do some kind of a test for one model, it-- going to apply for all of the equivalent models that we've shown to be equivalent, because the Turing machine can do the constructions which show the equivalence.

OK, so let's move on. Somebody didn't get the poll. Did most people not get the poll? Well, I think most of you have gotten it. Did you? I'm not seeing a lot of complaints. So if you didn't get the poll, something is wrong with your setup, and you're going to have to take the recorded check-ins that are going to launch right after this lecture's over. Sorry. But you should figure out what's wrong with the setup there, because I think most people are getting these.

OK, and with that, we're going to take a little break, and then we'll continue on afterward. Let me know how this is-- should I be speed a little speedier about the little mini breaks that we're taking, or is this good for you? Feedback is always-- I'm trying to make this as good as I can for all of you.

OK, I think there's a mixture of between people saying that these breaks are good. Someone says they're a little overlong, so I'll try to tighten them up a little. Some folks are saying what-- more people should be asking the TAs. I don't know how the TAs are doing, in terms of their-- I'll check with them afterward-- how well this is going for them, in terms of the questions.

But I think some amount of break is good so that we don't-- so there's time to be asking questions. Otherwise, what's the point of having a live lecture? So we will start promptly when this timer runs down. So be ready to go in 55 seconds from now.

Just to confirm, to show the decidability of the equivalence of two regular expressions, do we need to show that we can use a Turing machine to convert them to two DFAs first? If you want to test whether two regular expressions are equivalent, you can give any procedure for doing that deciding you want. I'm just offering one simple way to do it that takes advantage of stuff we've already shown.

But if you want to dig down and do some analysis of those regular expressions to show that they describe the same language, they generate the same language, be my guest. I think that's-- actually would be complicated to do that that way. OK, so we're just about ready to go here, so let's continue. And let me take this timer off here. All right.

Now, we are going to talk a little about context-free grammars. So we talked about decision problems for finite automata. Let's talk about some for grammars. OK, now I'm going to give us an analogous problem. I'm going to give you a-- I'm calling it the acceptance problem, but-- just for consistency with everything else, but it's really the generating problem.

So I'm giving you a grammar-- context-free grammar G and a string that's in a string. And I want to know, is it in the language of G? So does G generate w? I'm calling that the ACFG problem. So that's going to be a decidable again. These are getting slightly harder as we're moving along, so I'm giving you a grammar and a string, and I want to know, does the grammar generate that string?

Well, it's not totally trivial to solve that. One thing you might try doing is looking at all possible things, all possible derivations from that grammar and see if any one of them leads to w-- leads you to generate w. Well, you have to be careful, because as I've-- as we've shown in some of our examples, you actually could have-- because you're allowed to have variables that can get converted to empty string, you might have very long intermediate strings being generated from a, grammar which then ultimately give you a small string of terminals that get generated, a small string in the language that gets produced.

You certainly don't want to try every possible derivation, because there's infinitely many different derivations-- most of them generating, of course, things that are irrelevant to the string w. But you have to know how to cut things off, and it's not immediately obvious how you do that-- unless you have done the homework problems that I asked you to do, which I'm sure very many of you have not done-- the zero point X problems, because they're-- that's going to come in handy right now. And so I'll help you through that.

Remember-- you should remember, but you may not have looked at it-- something called Chomsky normal form, which is for context-free grammars, but only allows the rules to be of a special kind. And they have to look like this. They can be a variable that goes to two variables, on the right-hand side, or a variable that goes to a terminal. Those are the only kinds of rules that you're allowed.

This is a special case for the empty string, but let's ignore that for-- the start variable can also work to the empty string, if you want to have a-- the empty string in a language. But that's a special case. Let's ignore that. These are the only two kinds of rules that you can have in a Chomsky normal form grammar.

Once you have a Chomsky normal form grammar-- well, first of all, there's two things. First of all, you can always convert any context-free grammar into Chomsky normal form. So that's given in the textbook. You do the obvious thing. I'm not going to spend time on it. And you don't have to know it. It's just a little bit tedious, but it's straightforward and it's there, if you're curious.

But the second lemma's the thing that's important to us, which is that, if you have a grammar which is in Chomsky normal form and a string that's generated, every derivation of that string has exactly 2 times the length of the string minus 1 steps. If you think about it, this is a lemma-- very easy to prove. I think the homework problem asks you to prove that in the 0.2 or whatever it was in problem set 1-- or problem set 2-- I don't remember.

Rules of this kind allow you to make the string one longer, and rules of this kind allow you to produce a new terminal symbol. If the length w is n, you're going to have n minus 1 of these and n of. Those that's why you get 2n minus 1 steps. But the point is that I don't really care exactly how many. It's just that we have a bound. And once you have that bound, life is good from the point of view of giving a Turing machine which is going to decide this language-- because here's the Turing machine.

What it's going to do-- the first thing is it's going to convert G into Chomsky normal form. That we assume we know how to do. Now, we're going to try all derivations, but only those of length 2-- twice the length of the string minus 1, because if any derivation is going to generate w, it has to be this many steps, now that we know the grammar is in Chomsky normal form.

OK, so we have converted this problem of one that might be a very unboundedly lengthy problem and to one where it's going to terminate after some fixed number of steps. And so therefore, we can accept, if any of those generate w, and reject if not. OK? Before moving on, so this answers the problem-- shows that this language is decidable, the ACFG language. So that's something that-- make sure you understand.

We're basically trying old derivations of up to-- of a certain length, because that's all that's needed when the grammar is in that form. Now we're going to use that to prove a corollary, which is important for understanding how everything fits together. And that corollary states that every context-free language is a decidable language. Every context-free language is decidable. Now, why does that follow from this?

Well, suppose you have a context-free language. We know that language is generated by some context-free grammar G. That's what it means. So then you can take that grammar G and you can build it into a Turing machine. So there's going to be a term machine which is constructed with the knowledge of G.

And that Turing machine is going to take its w and run the ACFG algorithm with w combined with a G that's already built into it. So it's just going to stick that grammar G in front of w, and now we run the ACFG decider. It's going to say, does degenerate w? Well, that's going to be yes every time w is in A. And so this machine here now is going to end up accepting every string that's in A, because it's every string that G generates.

So that is, I think, what I wanted to say about this. Now, I feel that this corollary here throws a little bit of a curveball. And we can just pause for a moment here just to make sure you're understanding this. The tricky thing about this corollary is-- that I often get-- a question I often get where-- is when we start off with a context-free language, who gives-- how do we get G?

Because we need G to build a Turing machine M sub G. So we know A is a context-free language, but how do we know what G is? Well, the point is that we may not know what G is, but we know G exists, because that's a definition of A being a context-free language. It must have a context-free grammar, by definition. And so because G exists, I now know my Turing machine, M sub G, exists. And that's enough to know that A is decidable, because it has a decider that exists.

Now, if you're not going to tell me the grammar for G, I'm not going to tell you the Turing machine which decides A. But both of them exist. So if you tell me G, then I can tell you the Turing machine. So it's a subtle, tricky issue there. A certain little element maybe of-- sometimes people call it non-constructive, because we're just, in a sense, showing just that something is existing. It does the job for us, because it shows that A is a decidable language.

OK, so not so many questions here-- maybe the TAs are getting them. So let's move on. Here's another check-in. So now, can we conclude that A-- instead of ACFG, APDA is decidable? People are getting this one pretty fast. Another 10 seconds-- three seconds-- OK, going to shut it down-- end polling, share results.

Yeah. So this problem here is APDA is decidable. I was almost wondering whether or not to give this poll, because it has-- it's true for the same reason as poll number 1, because we know how to convert PDAs-- or we stated we can convert PDAs to CFGs. And that conversion has given in the book. We didn't do in lecture, but I just stated you have to know that fact, but not necessarily know how to do it. That's OK.

But the fact is true. The conversion is in the book, and it could be implemented on a Turing machine. So if you want to know whether a PDA's language is empty, you convert it to a context-free grammar and then use this procedure here to test whether it's a language-- oh, not empty-- I'm sorry. The acceptance problem-- I just want to know, does the PDA accept some input? I'm saying it wrong.

So I want to know, does the PDA accept some input? I convert that PDA to a grammar, and then I see if the grammar generates that input, because it's an equivalent grammar. So again, this is using the fact that grammars and PDAs are equivalent and convertible from one to another, just like regular expressions and DFAs from the previous poll. So you need to be comfortable with that, because we're going to not even talk about it anymore. We're just going to be treating those things-- going back and forth between them without sometimes even any comment. OK.

So let's move on. Emptiness problem for CFGs-- hopefully you're getting comfortable with the terminology I'm using. So now the emptiness problem for context-free grammars-- I'm just going to give you a grammar, and I want to know if its language is empty. OK, so remember, we did this for finite automata by testing a path. We don't really have paths here. You might think testing paths and pushed automata. That's not going to really work, because the stack is there.

So how are we going to do that test? Well, there's something sort of like testing a path, just working with the grammar itself, kind of working backwards from the terminals now. I'll illustrate that here. Here's a very simple grammar, and I want to know, does it generate any strings? Obviously, only care about strings of terminals, because those are the things that are in the language-- so does this grammar generate any strings of terminals?

So way we're going to answer that is by a marking procedure, but in a sense, we're going to start from the terminals and work backwards to see if we can get to the start variable. So first, we're going to mark all the terminal symbols, and then we're going to mark anything that goes to a string of completely marked symbols, either terminals or variables-- because anything that's marked, we know, can derive a string of terminals. That's what it means. Anything that's blue can derive a string of just terminals.

And so now, if you have a collection of those that are all marked, they together can generate some string of terminals together. And so then you can mark the associated variable. So T goes to a. So that may be oversimple, but we're going to mark T here everywhere in the grammar. So all these T's are going to get marked, because we know that T can generate a string of terminals.

Now let's take a look at R. R is going to a string of symbols that are all marked, and that means those symbols can, in turn, generate strings of terminals. So we're going to mark R2. We can't yet mark S, because we don't know yet whether S has some unmarked thing that it goes to. So we don't know yet whether S can generate a string of terminals. But R we know right now, so we're going to mark R.

But then now that gets us to mark this R, and so then we can go backwards and we can mark S. And we keep doing that until there's nothing new to mark. And here we've marked everything, so clearly there's nothing more you can mark. But you might stop before you've marked everything. And then you see whether you've marked the start variable or not. And if you have, you know the language is not empty.

OK, so let's just go through this in text. Mark all occurrences of terminals in G, then repeat until no new variables are marked. We mark all occurrences of variables A, if A goes to a string of symbols, and all of those symbols were already marked, because those are the things that already have been shown to generate a string of terminals. And so now we're going to reject if the start date-- start variable's marked-- because that means that the language is not empty-- and accept if it's not. OK?

I'll take a couple of quick questions here. OK, somebody asked whether-- if I understand-- are the regular languages also decidable? Well, remember, the regular languages are context-free languages, and the context-free languages are decidable, so yes, the regular languages are decidable. Some of those are going to be too long to answer, and they're trying to come up with alternative solutions. So I think we're going to move on.

All right. Just like we did for the finite automata, we talked about acceptance, we talked about emptiness, we talked about equivalence. So how about the equivalence problem for context-free grammars? I'm going to give you now two context-free grammars, and I'd like to know, are those two context-free grammars generating the same language?

OK, so how might you think about that? Well, one thing-- following some of the ideas that we've already had, you could try feeding strings into those grammars. You know how to test whether those individual grammars can generate those strings, so you can just try feeding strings in to G and to H, and seeing whether those grammars ever disagree or whether they generate some string. Find some string that say G generates, but H does not generate. And we can test that string by string. Unfortunately, we got a lot of strings to test. We would need to give some bound if we were going to use that procedure-- not clear what the bound is.

Another idea might be to use the closure construction that we had from before. So let's mull that over-- whether that might work. But in fact, let me give away the answer here. This language is not decidable. There's no algorithm out there-- no Turing machine, but therefore no algorithm-- which can take two grammars and test whether they generate the same language or not-- seems, at first, glance kind of surprising.

Such simple things as context-free grammars can nonetheless be so complicated that there's no procedure out there which can tell whether the two-- those two grammars generate the same language. We will show that next week. A related problem, which is related to your homework-- this that's due on Thursday-- is testing whether a grammar is ambiguous.

So given a grammar, I'd like to know, is that grammar an ambiguous grammar or not? Does it generate some string in the language of that grammar in two possibly different ways, two or more different ways? So is there some string that can be generated with two different parts. So the problem with testing whether grammar is ambiguous-- not decidable.

So I'm asking you to do something hard, when you have to produce that grammar which is unambiguous for that language. In general, testing whether a grammar is ambiguous or is not a decidable problem. Now, hopefully the grammar that you'll produce to show that-- that unambiguous grammar for that language that I'm asking you to produce is not going to require our graders to solve some decidable problem, but it'll be clear based the construction of that grammar why it's not ambiguous. So you'll have to hopefully say some explanation of your thinking there.

OK. And we will prove that the problem of testing ambiguity is not decidable. That's going to be a homework problem in problem set 3. OK. Last check-in here-- something that I alluded to, but didn't quite-- didn't want to give away. Why not use the same technique that we use to show equivalence of DFAs is decidable, to show that equivalence of context-free grammars is desirable? Obviously, something goes wrong because EQCFG is not decidable. Why doesn't that same technique just work?

Well, what's the answer? Got a real race here-- another 15 seconds-- this one gives you something to mull over. All right. Let's end it. OK, you're good to go. At least click something. OK, ending polling, sharing results-- OK, bunch of you have thought, well, CFGs are generators and DFAs are recognizers, and that's the issue.

Well, not really, because we could test equivalence of regular expressions-- those are generators. It's nothing to do with being a generator or not, because we can convert regular expressions to DFAs and then test equivalence for the DFAs, so that-- it's not really a matter of being generated. That's not the issue. The issue is that we can't follow the same construction that we did to show EQDFA is decidable, because the context-free languages are not closed under those operations we needed to make that symmetric difference machine-- if you remember how that worked. So they're not closed under implementation and not closed under intersection. We needed both in order to build that machine C, which accepted all the strings that are in the difference. OK? So let's continue on.

Let's move now to Turing machines. This is where stuff is really going to start to get interesting-- hope it's been interesting all along, but maybe even more interesting. So let's talk about the acceptance problem for Turing machines, ATM. This language is going to become an old friend, but we're just getting to know it.

So this is the problem. You're given a Turing machine now, and an input, and I want to know, does M accept that input? Does that Turing machine accept its input? OK, so that's going to not be a decidable problem either. So we've shifted gears from a bunch of decidable things to a bunch of undecidable things. So this is not a decidable language. We will prove that on Thursday. That's going to be the whole point of Thursday's lecture is proving the ATM is decidable. And that's going to be really a jumping off point for showing other problems are decidable. So that's going to be our first proof of decidability.

But we do know that ATM is recognizable, and that's worth understanding-- for multiple reasons, but for one thing, it's going to give us an example of a problem which we know is recognizable, but not decidable, as we'll prove undecidability. But it's also, I think, of historical importance, this algorithm for showing-- recognizable. So let's go through that algorithm. It's very simple, sort of doing the obvious thing.

The following Turing machine-- I'm going to call it U, for a reason that I'll make clear in a second. That's going to be a recognizer for ATM. So it takes as input an M and a w, and it's just going to simulate M on w, pretty much the way the algorithm the decider for ADFA work. But now the machine-- instead of being a DFA, it's a Turing machine, and the Turing machine might go forever. And so the simulation may not stop. And that's the key difference, which makes it from a decider into a recognizer.

So you're going to be simulating, just keeping track of the tape of M, the current state of M, where-- and proceeding to modify the tape as M modifies it. And then, if M enters an accept state, then you know M has accepted its input, so U also will enter an accept state. So the machine U enters the accept state if M observes during the simulation that M enters an accept state. Furthermore, if an M enters a reject state, then U also enters a reject state.

OK? Now I'm going to say something beyond that, which I want you to pay attention to, which is the kind of thing I do see sometimes people saying. We want U to reject if M never halts. That's also-- seems like what we want, because if it never halts, then M is rejecting by looping, so you should also reject. But I don't like that. I don't like that line here, step 4 of the Turing machine, because there's no way for a Turing machine to determine-- or at least as we-- no obvious way-- and in fact, there will not be any way, but certainly, at this point, no obvious way for M to even tell-- for U you to tell whether M is halting or not.

Well, certainly you can tell that it's to say it never halts. How can you make that determination? If I saw this on a solution, either on an exam or a homework, I would mark you off. This is no good. It's not illegal to Turing machine action. If M does not hold on w, then you should reject. That is correct. And you can make that reasoning external to the algorithm of U, but U is going to end up rejecting, because it never holds either.

It never actually knows that M is rejecting w in that case, if M is rejecting by looping. It's just blindly going along and doing the simulation of M on w-- and will end up halting-- rejecting by looping, if M is rejecting by looping. But that's something that you can argue if you need to make a proof or make some sort of reasoning about the machine, but it's not part of the algorithm of the machine.

OK. Now, what's, I think, from a historical standpoint, that's interesting about this machine U and why I'm calling it U is because this appeared-- this machine U appeared in Turing's original paper where he laid out Turing machines. He didn't call them Turing machines, by the way. He called them computing machines. People afterward called them Turing machines.

But Turing called this the universal computing machine. That's his language. Actually, I just looked at the paper yesterday just to refresh my memory of it. And he gives the description of the operation of U in gory detail, with all of the transitions and the states. He nails the whole thing down-- takes pages, and pages, and pages. So he it gives it there. So this is the original universal computing machine, universal Turing machine.

It's more than just an idle curiosity that this appeared in Turing's paper, because this actually turned out to be profoundly influential in computer science, because it really was the first example of a machine that operated based on a stored program. It really was a revolutionary idea. In those days, if you wanted to make a machine that did something different, you had to wire the-- rewire the machine.

But here's a machine that operated based on instructions. And instructions, in a sense, are no different than the data. So this is what's been-- come to be known as the von Neumann architecture, but von Neumann himself gave credit to Turing machine for having inspired him to think of this. And some people argue that it's really-- calling it the von Neumann-- bunch of people came up with this concept around the same time, maybe other people too.

There's Babbage and so on, and others who-- Ada Lovelace-- also who came up with notions of programming, but I think it's a little different than this, in concept. But anyway, this nevertheless played an important role in the history of the subject. So with that, I think we're out of time. I'm going to quickly review where we've been. So we just showed the decidability of various problems. These are all languages that we showed are decidable. We showed that ATM is Turing-recognizable, and I think that was all we had for today.

So I will stop right here. I will stick around and take a few questions, and our TAs can take a few questions, if you want, by chat. And then I also have my office hours, which will start in like five minutes or so, once I get everything set up on my end. OK, somebody wanted me to review this point here, which I'm happy to do.

This code here that I'm describing in English needs to be something that you can implement on a Turing machine. We're never going to go through the effort of building the transition functions, and the states, and so on, but we need to be sure that we could, if we had to. And how could you make a Turing machine do the test that M doesn't halt? That's something we don't know how to do.

I can see if M halts. During the simulation, I can see that M has entered the Q reject state or the Q accept state, so I can tell while I'm simulating that it has halted. But how would the machine, or how would you know that M is now-- someone says, well, do x if M never holds. Well, how do you know M-- how can you do the test that M is not halting?

There's no obvious way to do that. In fact, there is no way to do that. But as it stands right now, the proof would be on you to show how to implement that on a machine, and there's just no obvious way to do that. So I think that's why you should only put down things here which you're sure you could implement, even if it might take a long time. So you don't have to worry about how long it would take, but you have to put things down that you are sure you could at least implement in principle.

OK, so someone has asked me about the equivalence problem for context-free grammars being unsolvable. Why couldn't the machine be a human level system of logic so that the computer could logically deduce whether or not it was decidable? We're taking a turn into the subject of mathematical logic, which is supposed to formalize reasoning in a way.

In the end, what it really has come down to-- that there are certain grammars which are equivalent to one another, but there's not going to be any way to prove that they're equivalent in any reasonable system. Inequivalence you can always prove. You can exhibit a string that one-- you can show that this grammar's generating it, this grammar's not generating. This other one is not.

So inequivalence you can always prove, but equivalence-- there are going to be certain pairs of grammars which are going to be beyond the capability of any reasonable formal system to be able to prove that they're equivalent, because you can even convert-- make those grammars into something which talk about the system itself, ultimately. You're going to end up with a Russell paradox kind of thing. That's maybe going beyond more than you want to know, and I'm happy to talk about it offline.

But there's just no way to make a Turing machine which is going to implement human reasoning, and then get the right answer on all pairs of grammars-- just cannot be done. Goodbye. I'm going to shut down the meeting now.