[SQUEAKING] [RUSTLING] [CLICKING]

MICHAEL SIPSER: OK, folks. Here we are again. Welcome back for another episode of theory of computation. This is lecture number 3.

I'm going to review what we've been doing. We've been looking at finite automata and regular languages. Those are the languages that finite automata can recognize.

And we talked about nondeterminism. So we had non-deterministic finite automata and deterministic finite automata. We showed that they're equivalent. We looked at the closure properties over the regular operations union, concatenation, and star, and showed that the regular language is really-- the class of regular languages is closed under those regular operations. And we used the constructions that we developed in the proof of those closure properties to show that the-- we can give a way to convert regular expressions to finite automata.

So that is-- was partway toward our goal of showing that these regular expressions and finite automata are equivalent with respect to the class of languages they describe, namely, the regular languages. So regular expressions of finite automata are interchangeable from the perspective of what kinds of languages you can do with them. So we're going to finish that off today.

So let's take a look at what our next topics we're going to be covering. We're going to reverse the construction we gave last time which allowed us to convert regular expressions to finite automata. Now we're going to go backwards. We're going to show how to convert finite automata back to regular expressions. And that-- those two constructions together show us that the regular expressions and finite automata can be interconverted from one another, and they're therefore equivalent with respect to the kinds of things they can do in language recognition or generation.

Then, we're going to prove that-- we're going to look at how you prove certain languages are not regular, they're beyond the capabilities of finite automata. And finally, at the end, we're going to introduce a new model of computation which is more powerful than the finite automata and regular expressions, namely, the context-free grammars. Those can do other kinds of languages that the simpler finite automata regular expressions models can't do.

And I would also just like to note that a lot of what we're doing is a warm-up toward the more powerful models of computation that we're going to be looking at later-- well, in a week or so-- which are more general purpose computation. But along the way, introducing these models of finite automata in context-free languages is interesting and helpful because many of those-- a number-- those models turn out to be useful in a number of applications, whether it's from linguistics to programming languages. And a variety of different parts of computer science and other fields as well use those notions. So they're useful notions beyond just in this course.

So I just want to-- a couple of administrative things to touch on. We are going to have additional check-ins today, as I mentioned to you. We're going to start counting participation-- not correctness, just participation-- in the live check-ins.

So with that, let us move on to today's material. As I mentioned, we're going to be showing how to convert finite automata to regular expressions. And that's going to complete our equivalence of finite automata and regular expressions.

So just to recap what we did last time, we showed that if you have a regular expression and it describes some language, then that language is regular. So in other words, we have a way of-- we gave a way of converting regular expressions to finite automata, as kind of shown in this diagram. That's what we did last time.

Now we're going to go the other way. We're going to show how to convert-- oh, and just a reminder in case you're just getting yourself-- your memory to work, maybe it'll help you just to remember that we actually did an example of that conversion. We looked at this regular expression, a union ab star. And we actually worked through the process of converting that.

Oops. I need to make myself smaller so you can see all that.

We went through the process of converting a union ab star as an example of that-- of-- made a mis-- [LAUGHS] OK. Well, we went through the process of actually doing that conversion. And now we're going to show how to do it the other way around. So we're going to invert that and go backwards the other way.

So today's theorem is to show that if a is regular, namely, it's the language of some finite automaton, then you can convert it to a regular expression which will describe that same language. So basically, we're going to give a conversion from finite automata to regular expressions.

But before we do that, we're going to have to introduce a new concept. So we're not going to be able to dive right into that conversion. We're going to have to do-- introduce a new model first, which is going to facilitate that conversion. And that new model is called-- it's yet another kind of finite automaton called a Generalized Nondeterministic Finite Automaton, or a Generalized NFA, or just simply a GNFA.

So this is yet another variant of the finite automaton model. And conceptually, it's very simple. It's similar to the NFAs. I'll give you-- here's a picture of a GNFA named G, G1. Very similar to the NFAs. But if you look at it for a second, you'll see that the transitions have more complicated labels. For the NFAs, we're only allowing just single symbols, or the empty string, to appear on the labels. Now I'm actually allowing you to put full regular expressions on the labels for the automaton.

Now, we have to understand how a GNFA processes its input. And the way it works is not complicated to understand. When you're getting an input string feeding-- when a GNFA is processing an input string, it starts at the start state, just like you would imagine. But now, to go along a transition, instead of reading just a single symbol, or the empty string, as in the case for the nondeterministic machine, it actually gets to read a whole string at one step, kind of, at one bite. It can read an entire string and go along that transition arrow, provided that chunk of the input that it read is in the regular expression that that transition has as its label.

So for example, this-- you can go from q1 to q2 in one step in this GNFA by reading a, a, b, b off the input. So it reads all of those four symbols all at once. It just swoops them up and then moves from q1 to q2 in one step. And then, when it's in q2 it can read aab and move to q3. And q3 happens, there's nowhere to go.

So this is going to be a nondeterministic machine. There might be several different ways of processing the input. And if any one of them gets to an accepting state at the end of the input, we say the GNFA accepts. So it's similar to nondeterministic-- to NFAs in the way the acceptance criterion works.

So you could do an example. But hopefully the concept of how this works is reasonably-- you can at least buy it, that it processes the input in chunks at a time. And those chunks have to be described by the regular expressions on the transition arrows, as it moves along those transitions.

So what we're going to do now is to convert not DFAs to regular expressions, we're going to convert GNFAs to regular expression. That's even harder, because GNFAs are-- allow you to do all sorts of other things besides just ordinary DFAs. So that's a harder job. Why am I making my life harder? Well, you'll see in a minute that it's going to actually turn out to be helpful to be working with a more powerful model in the way this construction is going to work.

Now, before I dive in and do the construction from GNFAs to regular expressions, I'm going to make a simplifying assumption about the GNFAs. I'm going to put them in a special form that's going to make it easier to do the conversion. And that simpler form is, first of all, I'm going to assume the GNFA has just a single accepting state. And that accepting state is not allowed to be the start state. So it has to have just a single accepting state.

I've already violated that convenient assumption in this GNFA, because I have here two accepting states. That's not what I want. I want to have just one.

Well, the thing is, it's easy to obtain just one, just to modify the machine so that I have just one by adding a new accepting state which is branched to from the former accepting states by empty transitions. So I can always jump from q2 to q4 at any time without even reading any input, just going along this empty transition. And then I declassify the former accepting states as accepting.

And now I have here just a single accepting state. And because it's going to be a new state that I added, it won't be the start state. And I have accomplished that one aspect of my assumption about the form of the GNFA.

But there's another thing that I want to do, too. I want to assume-- as you will see, which is going to be convenient in my construction-- that we will have transition arrows going from every state to every other state. In fact, I want transition arrows going from every state even back to themselves. I want there to be-- all possible transition arrows should be present, with two exceptions. For the start state, there should be only transition arrows exiting the start state. And for the accepting state-- there's just one now-- there should be only transition arrows coming into the start state.

So it's kind of what you would imagine as being reasonable. For the other states, which are not accepting or starting, there should be transition arrows leaving and coming from everywhere else. But for the start states, just leaving. And from the accept state, just coming in. And you could easily modify the machine to achieve that.

Let's just see how to do that in one example. So from-- notice that from q3 to q2 there is no transition right now. And that's not good. That's not what I want. I want there to be a transition from q3 to q2. Well, I'll just add that transition in.

But I'm going to label it with the empty language regular expression. So that means, yeah, the transition is there, but you never can take it. So it doesn't change the language that the machine is going to be recognizing. But it fulfills my assumption, my convenient assumption, that we have all of these transition arrows being present in the machine.

So anyway, I hope you will buy it. It's not going to be-- if you don't quite get this, don't worry. It's not totally critical that you're following all these little adjustments and modifications to the GNFA. But it will be helpful to understand what GNFAs themselves-- how they work. So as I mentioned, we can easily modify GNFA to have the special form that we're assuming here.

So now we're going to jump in and start doing the conversion. So we're going to have a lemma, which is like a theorem that really is just of local interest here. It's not a general interest theorem. It's going to be relevant just to GNFA, which are really just defined to help us do this conversion. They really don't have any other independent value.

So every-- you want to show that every GNFA has an equivalent regular expression R. That's really my goal. And the way we're going to prove that is by induction. It's going to be by induction on the number of states of the GNFA.

Now, you really should be familiar with induction as one of the expectations for being in this course. But in case you're a little shaky on it, don't worry. I'm going to unpack it as a procedure. It's really just recursion. You know, induction is just-- a proof that uses induction is really just a proof that calls itself. It's just a proof that-- it's a recursive proof. That's all it is.

So if you're comfortable with recursion, you'll be comfortable with induction. But anyway, I'm going to describe this as a procedure. So if you're a little shaky on induction, don't worry.

So the basis is-- so first I'm going to handle the case where the GNFA has just two states. Now, remember, I'm assuming now my GNFAs are in the special form. So you can't even have a GNFA with one state, because it has t have a start state and it has to have an accept state, and they have to not be the same. So the smallest possible GNFA to worry about is a two-state GNFA.

Now, if we have a-- if we happen to have a two-state GNFA, it turns out to be very easy to find the equivalent regular expression. Why? Because that two-state GNFA can only look like this. It can have a start state, it can have an accept state, and it can only have a transition going from the start to the accept because no other transitions are allowed. It only has outgoing from the start, only incoming from the-- to the accept. And so there's only one transition. And it has a label with a regular expression R.

So what do you think the equivalent regular expression is for this GNFA? It's just simply the one that's labeling that transition, because that tells us when I can go from the start to the accept. And there's nothing else the machine can do. It just makes one step, which is to accept its input if it's described by that regular expression. So therefore, the equivalent regular expression that we're looking for is simply the label on that single transition.

So two-stage GNFAs are easy. But what if-- what happens if you have more states? Then you're going to actually have to do some work.

So we call that the induction step. That's when we have more than two states. And what that-- the way the induction works is we're going to assume we already know how to do it for k minus 1 states. And we're going to use that knowledge to show how to do it for k states.

So in other words, we already know how to do it for two states. I'm going to use that fact to show how to do it for three states, and then use the fact that I can do it for three states to show how to do it for four states, and so on, and so on. And the idea for how to do that is actually pretty easy to grasp.

What we're going to do is, if we have a k state GNFA that we want to convert, we're going to change that k state GNFA to a k minus 1 state GNFA and then use our assumption that we already know how to do the k minus 1 state GNFA. So in terms of a picture, I'm going to take a k state-- to prove that I can always convert k state GNFAs to regular expressions, I'm going to show how to convert the k state one into an equivalent k minus 1 state GNFA. And then, if you just like to think of this procedurally, the k minus 1 gets converted to a k minus 2, gets converted to a k minus 3, and so on, and so on, until I get down to two, which then I know how to do directly.

So the whole name of the game here is figuring out how to convert a GNFA that has k states into another one that has one fewer state that does the same language. So you have to hold that in your head. I mean, I wish I had more blackboard space here, but it's very limited here. So you have to remember what we're going to be doing on the next slide, because that's going to finish the job for us. As long as I can show in general how to convert a K, state GNFA to a GNFA that has one fewer state but it still does the same language, I'm good, because then I can keep iterating that till I get down to two.

So here is-- this is the guts of the argument. So I have my k state machine. Here's my start state. Here's my accept state. Here's my k minus 1 state, that machine that I'm going to be building for you. It's actually going to be-- look almost exactly the same. I'm just going to remove one state from the bigger machine.

So I'm going to pick any state which is not the start state or the accept state. Here it is pictured here. I mean, all of the states of the k state machine are going to appear in the k minus 1 state machine except for one state that I'm going to rip out. That's the state x.

It's now here as a ghost. It's been removed. It's not there anymore. But I'm just helping you to remember that it used to be there by showing this shadow. But it's a-- I have taken my original machine that had k states and basically just ripped out a state. And now I have one fewer state.

So the good news is that I now have a machine with k minus 1 states. That's what I want. But the bad news is that it doesn't do the same language anymore. I broke the machine by rip-- if you're just going to rip out a state, who knows what the new machine is going to do. It's going to be probably not the same as what the original machine did.

So what I need to do, then, is repair the damage. I've got to fix the damage that I caused by removing x. And whatever role x was playing in the original machine, I've got to make sure that the new machine that I have, which doesn't have x anymore, can still do the same things that the original machine did.

And so the way I'm going to do that is look at all of the paths that could go through x and make sure that they are still present even though I don't have x anymore. And the way I'm going to do that is, I'm going to take-- consider a part of a path that might use x. So it starts-- let's pick two states, qi and qj, in the machine that had k states.

Let me just see here-- I don't know if this-- OK. We have-- if we have-- we'll pick two states, qi and qj, in the original machine. Now, qi might have the possibility of going to state x. And then x might have a self loop. And then it might go to qj.

The new machine doesn't have an x anymore. The way I'm going to fix that is by replacing the label that goes directly from i to j with a new label that adds in all of the things I lost when I removed x. That's the whole idea here.

So here is qi to qj, but there's no x anymore. How could I get from qi to qj? What were the inputs that could have brought us from qi to qj via x? Well, they would have been an input that read a string described by r1. I might have self-looked at x a few times, so I might have read several strings that are described by r2. And then I would have read a string that was described by r3. And now I'm at qj.

So the new label that I'm going to place over here is going to be the strings that I get from reading r1-- reading a string that's described by r1, then multiple copies of strings-- multiple strings that are possibly describing r2, which is the same as r2 star. Oh, and then multiples-- and then a string that could be described by r3. So that is a new addition to the transition that takes me from qi to qj.

Of course, I need to include the things that would have taken me from qi to qj in the first place. So I'm also unioning in r4, which is the direct route from qi to qj that did not transit through x. So by making that new regular expression on the qi to qj transition, I have compensated for the loss of x for paths that go from qi to x and then out to qj.

Now, what I need to do is to do that same thing for every pair qi and qj that are in the original machine. And so if I do that for every possible pair, I'll be modifying all of the transitions in the new machine in a way that compensates for the loss of x. And now the new machine has been repaired from the damage that I caused by removing x. And it does the same language.

It's the kind of thing you need to think a little bit about. I understand. But at least hopefully, the spirit of what I just described to you comes through, that we're going to convert this k-- machine with k states to one with k minus 1 states by removing a state and repairing the damage. And now it does the same language. And then I can remove another state and do the same thing over and over again until I get down to two states.

So that's the idea. And that really completes the proof. That shows that I can convert every GNFA to a regular expression. And that really is the end of the story for this. And thus I claim that DFAs, now, and regular expressions are equivalent.

So let me-- going to give you a little check-in here on this, really just to see, high-level, if you're following what's going on. So just take a look. So we just showed how to convert GNFAs to regular expression. But we really wanted to convert DFAs to regular expressions. So how do we go from GNFA-- converting GNFAs to converting DFAs? Because they're not the same, obviously. Right?

So how do we finish that? So there are three choices here. First, we have to show how to convert DFAs to GNFAs, maybe? Or show how to convert GNFAs to DFAs? Or maybe we're already done? So maybe I better launch that poll while you're reading that.

And there you go. Hopefully you can-- all right. Why don't I end this? It's a little worrisome, because I would say we have a plurality who got the right answer, but not a majority. So let us share the results.

I think-- so I sense not all of you are with me. But you're going to have to-- either that or you're playing-- you're reading your email while we're talking. I'm not sure.

But whatever it is, you need to think a little bit about what's going on here, because the reason why we are done is because DFAs are a kind of GNFAs. They're just-- they have a very simple kind of regular expression on each transition. They just have the regular expression which is just a single symbol.

So all DFAs are automatically GNFAs. So if I can convert GNFAs, I can certainly convert DFAs, because GNFAs include the DFAs. I'm done. It really was-- number C was the correct answer.

So good thing we're not [LAUGHS] counting correctness here. So participation is good enough. But I do think you need to think about what's going on and making sure that you're following along. So anyway, that's a-- we'll carry on here. But it makes me a little concerned.

So let us now move on. So we're going to talk a little bit about non-regular languages.

So somebody's asking, don't we have to still make the DFAs into the special type? Yes, we do have to make them to the special type. But we already showed how to make GNFAs into the special type.

And DFA-- that is going to apply to DFAs as well. They'll become GNFAs. You can add the extra starts-- add a new start state, add a new accept state, add in all the transitions with-- which you didn't have before with the empty language label, and you'll have a GNFA from a DFA.

But that applies to GNFAs as-- in general. So it's nothing special about DFAs there. Anyway, I think you need to chew on that. And hopefully you're-- you'll be following going forward.

Anyway, let us look now at non-- proving non-regularity. So we're finished with our goal of showing that regular languages-- that the regular languages can either come from DFAs or from regular expressions. Those are the same in terms of-- from the perspective of our course, they're interchangeable.

So now, as we mentioned, there are going to be some languages which are not regular, which can't be done by DFAs. They're actually-- DFAs are actually pretty weak as a computational model. And so there's all sorts of very simple things that they cannot do-- though there are some fairly complicated things that they can do, surprisingly enough.

But anyway, there are some simple things they can't do. And so we have to develop a method for showing that a language is not regular. And that's going to be useful for your homework and in general for just understanding the power of DFAs.

So how do we show a language is not regular? So remember, if you want to show a language is regular, basically what you need to do is give a DFA. Or you can use the closure properties. That's another way of showing a language is regular. But underneath that, it's basically constructing DFAs.

To show a language is not regular you have to give a proof. Generally it's not a construction, it's a proof that there is no DFA or that whatever-- that it's just going to be impossible to make a DFA. And we have to develop a method. What is that proof method?

Now, there is a tempting-- you know, I've taught this course many times, and there's a tempting approach that many people have. It's not only going to apply for finite automata, but for other things too. And believe me, it's not only people in this class, it's for people out there in the-- who are trying to think about computation in general-- which is to say, well, I have some language. I'm trying to figure out if it's regular or not. And so I thought really hard how to make a DFA, and I couldn't find one. Therefore, it's not regular.

That's not a proof. Just because you couldn't find a DFA doesn't mean there is no DFA. You need to prove that the language is not regular using some method.

So I'm going to give you an example where that kind of approach can lead you wrong. And that is-- I'll give two examples of languages where you might try to prove they're regular or not, and you could be in trouble if you just follow that kind of informal approach. So if you take the language B, where these are strings-- well, let's assume our alphabet is zeros and ones. B is the language of all strings that have an equal number of zeros and ones. So you want to know, if I have 1,000 zeros, I need to have 1,000 ones.

So basically, the way you test that, you'd have to count up the number of zeros, count up the number of ones, and see if those two counts are the same. And that's going to be really tough to make a DFA do, because how are you going to remember such-- that really big number of zeros that-- the DFA might have 50 states. But you might need to count up to 100 or a million to figure out-- to count up how many zeros you've seen.

And it seems really hard to be able to do that kind of a count when you only have 50 states. So whatever number of states you have, it seems hard to count when you have a finite automaton. So the intuition is, it's not regular because a finite automaton can't count. Which, in this case, you can convert that intuition into a real proof. I would say it's not a real proof yet, but it can be made into a real proof.

But compare that case with another language, which I'll call C, which, instead of looking at its input to see whether it has an equal number of zeros and ones, I'm going to look at the input and look at the substrings of 01s and 10s-- those two substrings-- and count the number of occurrences of 01 as a substring and the number of occurrences of 10 as a substring.

Just to make sure you're understanding, let's look at some example-- two examples. So the string 0101 is not going to be in C, because if you count up the number of 01s and the number of 10s, not the same. So I'm even going to help you here, if you can see that. The number of 01s is two. But there's only a single occurrence of 10. So those are-- those two counts are different. And so that's why this input string is not in C.

Compare that with the string 0110. Now, if you count up the number of 01 and 10 substrings, you're going to get the same value, because here we have a single 01 and a single 10. And so now the two counts of those number of substrings are the same. And so that's where you're in C.

Now my question is, is C a regular language? Well, it looks like it shouldn't be regular for the same reason that B isn't regular-- because you have to count up two quantities and compare them. OK?

So now, so if we-- so that's our intuition, that you just can't do it for the-- with a finite automaton, because you have to do the same kind of counting that you would have had to do for language B. But here you'll be-- you would be wrong, because C, in fact, is regular. It has a much simpler description than the one I gave over here at the beginning.

The very same language, C, can be described in a much, much simpler way. I'm not going to tell you what it is. You can mull that over. You can try some examples to figure it out. But it has a much simpler description. It's not a totally trivial description. There is some content there. But there is-- it is the kind of thing that a finite automaton can do. It wouldn't do the counting this way.

So the moral is-- the punch line is that sometimes the intuition works, but it can also be wrong. And so the moral of the story is, you need to give a proof when you're doing things like that.

So what we're going to do next, in the second half of the lecture, is to give a method for proving languages are not regular. And again, you're going to need to use that on your homework. So I hope you get it.

But first of all-- did I-- never stopped sharing that poll. Forgive me. And so with that, I think we'll take our little requested break. And-- for five minutes. And we'll be back in five minutes. So, break time.

We are done here. And proving languages not regular. The way we're going to prove languages are not regular is by introducing a method called the pumping lemma.

And the overarching plan at the pumping lemma, without getting into the specifics of it, is to say-- show that-- that lemma says all regular languages have a certain property, which we will describe. And so to show a language is not regular you simply show the language doesn't have that property, because all regular languages have to have that property. And so by showing a language fails to have the property, it could not be regular. That's the plan.

Now, the property itself is a little complicated to describe, but not too bad. I'll try to unpack it for you. But first, let's look at the statement of the lemma, which says that whenever you have a regular language-- let's call it A. So for every regular language A there's always a special value called the pump-- a number. p, we'll call it-- called the pumping length. It's a special number.

And it's-- and that length tells you that whenever a string is in that language and it's longer than that length, then something special happens. You can take that string and you can modify it, and you still stay in the language. So anything that's longer than that special length can be modified in a certain way, and you still stay in the language.

So let's look at the actual statement of the lemma. So there is a number p such that if s is a string in the language and it's longer than p, or at least of length p, then you can take s and you can cut it up into three pieces-- x, y, and z-- so that's just breaking s into three pieces-- where you can take that middle piece, repeat it as many times as you like, and you still stay in the language.

That's the-- what the pumping lemma is saying. And there's a bunch of other conditions here too. But the spirit of the pumping lemma says, whenever you have a regular language there's some cutoff such that all strings longer than that cutoff can be what we call pumped. You can take that string, you can find a section somewhere in the middle of that string or somewhere-- you cut it up in three pieces, you take that center piece, and you can repeat it. You can pump it up.

And by repeating that string and repeating that piece, the string gets longer and longer. But you still stay in the language. That's the special property that all regular languages have.

So in an informal way-- and we'll do-- I'll try to help you get the feeling for this. Informally, it says that if you have a regular language, then every long string-- so a long is by-- informal way of saying bigger than this value p. Every long string in the language can be pumped. And this result still stays in the language.

And by "pumped" means I can cut the string into three pieces and repeat that middle piece as many times as I want. That's what I mean by pumping a string.

So we'll do some examples in a second. But first we're going to see how to prove this. And hopefully, that'll give you some feeling, also, for why it's true.

So-- and actually, maybe before I actually jump into the proof, let me-- let's look at these three conditions here just to understand it a little bit more thoroughly. So condition one kind of says what I just was telling you. I can break s into three pieces-- x, y, z-- such that if I take x y to the i z-- so that's repeating y as many times as I want. So here's y to the i defined, if that's helpful to you-- it's just y-- i copies of y. So I can take x y to the i z, and I remain in the language for every value of i-- even i equals 0, which means we're just removing y, which is sometimes actually a useful thing to do. But let's not get ahead of ourselves.

So if-- you know, I can cut s-- I'm guaranteed to be able to cut s up into x, y, z so that the string xyyy is still in the language, or xyyyyy-- it's still in the language. That's going to be guaranteed for every regular language. That's a feature that's going to be true.

And furthermore-- and this is going to be turning out to be-- it's not really part of the core idea of the pumping lemma, but it actually turns out to be very helpful in applying the pumping lemma. You can always cut it up in such a way that the first two pieces are not longer than that value p. So this-- it restricts on the ways you can cut the thing up. And that actually turns out to be very helpful.

But let's first just look at the proof of this, giving a little bit the high-level picture. So my job is to show, if I have a string in my language-- let's say it's a-- think of it as a long string, really long. So its length is more than p. But I think intuitively, it's just a very long string. And I'm going to feed that string into the machine and watch what happens.

Something special happens when I feed the string and I look at how the machine proceeds on that string, because s is so long that as I wander around inside the machine I have to end up coming back to the same place more than once. It's like if you have a small park and you go for a long walk. You're going to end up coming back to where you've-- what you've already seen. You just can't keep on seeing new stuff when you have a more small area of space to explore.

So we're guaranteed that M is going to end up repeating some state when it's reading s because s is so long. So in terms-- pictorially, if you imagine here this wiggly line is describing the path that M follows when it's reading s, it ends up coming back to that state qj more than once. So it comes back here, cycles around, comes back again before it ends up accepting.

We know it ends up accepting because we're assuming we have a string that's in the language. So we picked s in the language. So it has to be accepted by M. But the important thing is that it repeats a state.

Now, how does that tell me I can cut s up into those three pieces? Well, I'm going to get those three pieces here. First of all, let's observe that here is processing-- as processing s. Here is the-- written right on top of the string, that state repetition occurring, qj, more than once.

And now, if I look inside the machine, the part of s that took me to qj I'm going to call x. The part that took me from qj back to itself I'm going to call y. And the part that took qj to the accept state I'm going to call z. And I'm going to mark those off in s. And that gives me the way to cut s up into three pieces.

Now, if you're appreciating what's going on inside the machine, you will see why M will also accept the string xyyz-- because every time-- once you're at qj, if you go around once, you come back to qj. And then if you go again, you'll come back to qj. And as many times as you keep seeing that y, you're just going to keep coming back to qj.

So it doesn't matter how many y's you have. You're going to still-- if you follow it by z, which is what you will do-- you'll end up accepting this string. And that's really the proof.

I mean, you have to do a little bit more here just to understand-- I should have mentioned why I want to forbid y being the empty string, because if y's the empty string it's not interesting. It doesn't change-- repeating it doesn't actually change anything. So I have to make sure it's not empty. But anyway, that's a detail here. If you look at the string xyyz, that's still going to be accepted. So that's the proof of the pumping lemma.

So let's have a little check-in related to that. This is not going to be-- again, not super hard. But more just a curiosity.

So the pumping lemma depends on the fact that if M has p states and it runs for more than p steps, then it's going to enter some state twice. So you may have seen that before. It actually has a name which some of you may have seen.

So let's see how to just get a poll here. And I hope not too many of you are going to pick C, as it's-- some of you are. [LAUGHS] Oh well.

Yes, I think this one most of you are-- you've seen this before. This is-- I think you pretty much all got it. This is what's known as the Pigeonhole Principle. So here, sharing the results, obviously I was having a little fun with this. I'm sure some of you were having fun back at me. That's OK.

So let's continue on. Let's see how to use the pumping lemma to prove a language is not regular. So I put the pumping lemma up here just so you can remember the statement of it.

So let's take the language D, which is the language 0 to the k 1 to the k for any k. So that's some number of zeros followed by an equal number of ones. We're going to prove that language is not regular by using the pumping lemma.

And this is going to be just an ironclad proof. It's not going to say, well, I couldn't think of how to-- I couldn't think of how to find it a finite automaton. This is going to be-- this is going to really be a proof.

So we want to show that D is not regular. And we're going to give-- these things always go as a proof by contradiction. So proof by contradiction-- hopefully as a reminder to you, the way that works is you're going to assume the opposite of what you're trying to prove. And then from that, something crazy is going to happen, something you know is obviously false or wrong.

And so therefore your assumption, which is the opposite of what you were trying to prove, had to be wrong. And so therefore, the thing you're trying to prove has to be right. That's the essence of what's called proof by contradiction.

So first of all, we're going to assume, to get our contradiction, that D is regular, which is what we're trying to show is not the case. Now, if D is regular, then we can apply the pumping lemma up above here, which gives us that pumping length p, which says that any string longer than p can be pumped and you stay in the language. That's what the pumping lemma tells you.

So let's pick the string s, which is the string 0 to the p 1 to the p. Here's sort of a picture of s off on the side here. So a bunch of zeros followed by an equal number of ones.

And that string is in D because D is strings of that form. And it's longer than p. Obviously, it's of length 2p. So the pumping lemma tells us there's a way to cut it up satisfying those three conditions.

So how in the world could we possibly cut s up? Well, remember the three conditions. And especially condition 3 is going to come in handy here.

Say that you can cut s up into three pieces-- x, y, and z-- where the first two pieces lie in the first p symbols of s at most p long. So x and y together are not very big. They don't extend beyond the first half of x-- first half of s. And in particular, they're all zeros. x and y are going to be all zeros. z is going to perhaps have some zeros and will have the rest of the ones-- will have the ones.

Now, the pumping lemma says that if you cut it up that way, you can repeat y as many times as you like and you stay in the language. But that's obviously false, because if you repeat y-- which now has only zeros-- you're going to have too many zeros. And so the resulting string is no longer going to be of the form 0 to the k 1 to the k. It's going to be lots of zeros followed by not so many ones.

That's not in the language. And that violates what the pumping lemma tells you is supposed to happen. And that's a contradiction. So therefore, our assumption that D is regular is false. And so we conclude that D is not regular. So that's a fairly simple one.

I thought I would do another couple of examples, because you have this on your homework and I thought it might be helpful. So here's the second one-- slightly harder, but not too much. Let's take the language F, which is-- looks like the string's ww. These are strings that-- two copies of the same string. For any string that might be in sigma star, so for any string at all, I'm going to have two copies of that string. And so F is those strings which can be-- which are just two copies of the same string.

We're going to show that F is not regular. These things always go the same way. It's the same pattern. You prove by contradiction.

So you assume for contradiction that-- oh, D. That's bad. That was copied from my other slide. That's wrong. Let's see if I can actually make this work here. Good.

Assume for contradiction that F is regular. The pumping lemma gives F as above. And so now we need to choose a string s that's in F to do the pumping and show that the pumping lemma is going to fail. You're going to pump and you're going to get something which is not in the language, which is-- shows that the pump-- something has gone wrong.

But which s to choose? And sometimes that's where the creativity in applying the pumping lemma comes in, because you have to figure out which is the right string you're going to pump on. So you might try the string-- well, 0 to the p 0 to the p. That's certainly in F. It's two copies of the same string.

Here it is. I've written lots of zeros followed by the same number of zeros. The problem is, if you use that string, it actually is a string that you can pump. You can break that string up into three pieces.

And then, if you let y be the string 00-- actually, you have to be a little careful. The string just 0 doesn't work, because there's an evenness-oddness phenomenon going here. So you might want to just think about that.

But if you let y be the string 00, then if you have the string xy-- x any number of y's-- it's still just going to be a bunch of zeros. And you're going to be able to see that that string is still in the language. So you haven't learned anything.

If the pumping lemma works and you're satisfying the pumping lemma, you haven't learned anything. So what you need to find is some other string. That was a bad choice for s. Find a different string.

So here's a different choice, 0 to the p 1 0 to the p 1. So that's two copies of the same string. And you're going to show it can't be-- we're going to show it can't be pumped. So here's a picture of that string here. So zeros followed by 1, zeros followed by 1.

And now it's a very similar to the first argument. If you cut it into three pieces in such a way that it satisfies the conditions, the first two pieces are going to be residing only among the zeros. And so therefore, when you repeat a y you're no longer going to have two copies of the same string. And so it won't be in the language. So therefore, you've got a contradiction and F is not regular.

So you have to play with the pumping lemma a little bit. If you haven't seen that before it's going to be-- it takes a little getting used to. But you have a few homework questions that need to be solved using the pumping lemma.

So now, let's look at-- lastly, there is another method that can come in, which is combining closure properties with the pumping lemma. So closure properties sometimes help you. So let's look at the language B, which is actually, we saw earlier in the lecture, where we have an equal number of zeros and ones.

Now, we could prove that directly, using the pumping lemma, as not being regular. But it's actually even easier. What we're going to prove-- that-- we're going to prove that it's not regular in a different way. First we're going to assume for contradiction, as we often do, that it is regular. And now we're going to use something-- we're going to use some other knowledge. We're not going to use the pumping lemma here because we're going to take advantage of an earlier case where we used the pumping lemma.

And so now we know that the string-- the language 0 star 1 star is a regular language, because it's described by a regular expression. If you take the B, which is the equal numbers of zeros and ones, and you intersect it with 0 star 1 star, that's going to be a regular language if B was regular, using closure under intersection. But this language B intersect 0 star 1 star is the language of equal numbers of zeros and ones where the zeros come first. And that's the language D that we showed two slides back, that we already know can't be regular.

So that intersection cannot be regular. And so it violates the closure property. And again, we get a contradiction. So that's a different way of sometimes making a shortcut to prove a language is not regular.

So we have-- in our last 10 minutes or so, we're going to shift gears totally, in an entirely different way, and consider a new model of computation which is more powerful, that can actually do things that we can't do with finite automata. And these are called context-free grammars. So this is really just an introduction. We're going to spend all of next lecture looking at context-free grammars and their associated languages. But let's just do-- get a preview.

So a context-free grammar looks like this. You have a bunch of these-- what we call substitution rules, or rules, sometimes, which just look like a symbol, arrow, a string of symbols. That's what a context-free grammar looks like at a high level.

Let's define some terms. So a rule, as I just described, is going to be-- look-- it's going to be a symbol, which we're going to call a variable. And that's going to have an arrow to a string of other-- possibly, other variables and symbols called terminals.

So a variable is a symbol that appears on the left-hand side of a rule. Anything that appears on the left-hand side is going to be considered to be a variable. So S and R are both variables.

Now, other symbols that appear in the grammar which don't appear in the left-hand side-- those are going to be called terminals. So here, 0 and 1 are terminals. Now, you may think that empty string should also be a terminal . But that's not a symbol. Empty string is a string. It's just a string of length 0. So I'm not considering empty string to be a terminal.

So-- and then there's going to be a special variable which is going to be considered the starting variable, just like we had a starting state. And that's typically going to be written as the top-left symbol. So this symbol s, here, is going to be the starting symbol.

And grammars can be used to define languages and to-- well, to generate strings and to define languages. So first of all, let's see how a grammar, using this as an illustration, can generate strings. Actually, just to emphasize this terminology here, in this particular example we had three rules. The two variables were R and S. The two terminals were 0 and 1. And the start variable was this top left-hand symbol, as I mentioned-- the S.

So grammars generate strings. The way they do is you follow a certain procedure, which is really pretty simple. You write down, first of all, the start variable. And I'll do an example in a second. You write down the start variable.

And then you take a look what you've written down. And if it has any variables in it, you can apply one of the corresponding right-hand sides of a rule as a substitution for that variable. And so-- like, for example, if you have an S in the thing you've written down, you can substitute for that S a 0S1. Or you could substitute for that S an R. Or if you have an R, you can substitute for the S an empty string.

So you're just going to keep on doing that substitutions over and over again until there are no variables left, so there's nothing left to substitute. Only terminals remain. At that point, you have generated a string in the language. So the language, then, is the collection of all generated strings.

Let's do an example. Here's an example of G1 generating some string. So as I mentioned, first of all, you're going to write down the start variable. And I'm just going to illustrate this in two parallel tracks here. On the left side I'm going to show you the tree of substitutions. And on the right side I'm going to show you the resulting string that you get by applying those substitutions.

So over here I'm going to substitute for S the string 0S1. So on the right-hand side I just have 0S1, because that's what I substituted for S. But you'll see it's not going to-- it's going to look a little different in a second. Here, I'm going to-- again I still have a variable. So I'm going to substitute for S 0S1.

Now I have the string-- resulting string 00S11, because I've substituted 0S1 for the previous S, but the 0 and 1 stick around from before. They don't go anywhere. So I have, at this point, 00S11.

Now I'm going to take a different choice. I'm going to substitute for S-- I could have gone either way. This would have something-- almost like non-determinism here, because you have a choice. I'm going to substitute for S-- instead of 0S1 I'm going to substitute R, because that's also legitimate in terms of the rules. And so now I'm going to have 00R11.

And now R-- there's no choices here. R can only be substituted for by an empty string. So I get to R becomes just empty string. And in terms of the string generated, empty string doesn't add anything. It just really is-- it's a nothing. So I get the string 0011. And this is a string just of terminal symbols. And so that is a string in the language of the grammar G1.

And if you think about it, G1's language is that language that we saw before, which I think we called D-- 0 to the k 1 to the k for k greater than or equal to 0. So this is an example of a language that a context-free grammar can do but a finite automaton cannot do.

So that is our little introduction to-- oops. There's one more check-in here. Oh, yeah. So I'm asking you to actually look at-- let me get myself out of this picture so you don't see me blocking things. And we will do one last check-in. Make sure you're staying around for the whole thing.

Now there could be several of these strings that are in the language. You have to click them all-- all of the ones that you have found that are in the language of this grammar that can be generated by grammar G2, you have to click those. I'll give you a little bit more time on this one to see which ones G2 can generate.

I'll give you a hint. It's more than one, but not all. So I see you're making some progress here. Interesting.

So please-- we're going to wrap this up very quickly. You can-- somebody's telling me you can't unclick. Thank you. Good to know. Still, things are coming in here.

So let's not-- we're running toward the end of the hour here. I don't want to go over. So I'm going to end it in five seconds. Click away. And don't forget, we're not going to charge you if you get it wrong.

Sharing results. I don't know why it has an orange one there, because there are several correct answers here. So it's A, B, and D are correct. You can get any of those. It's really sort of two copies of the language we had before next to one another. And so the only thing you cannot get is 1010. So I encourage you to think about that.

And I will come to our last side of today, which is just a quick review. I can put myself back. So we showed how to convert DFAs to regular expressions. And the summary is that DFAs, NFAs, GNFAs, even, and regular expressions are all equivalent in the class of languages they can describe.

The second thing we did was a method for proving languages not regular by using the pumping lemma or closure properties. And lastly, we introduced context-free grammars. And we're going to see more about those on Thursday.

So with that, I think we're out of time. And thank you for the notes of appreciation. And I will-- I think we're going to end here. And see you on Thursday, if not before.