[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: So, welcome, everybody, to the Fall 2020 online Introduction to the Theory of Computing 18.404/6.840. My name is Mike Sipser. I'm going to be your instructor for the semester in this class. So let me just tell you what the course is about.

Basically, it's going to be in two halves. We're going to be talking about what are the capabilities and limitations of computers-- of computer algorithms, really, computation. And the two parts of the course are more or less divided in half. The first half of the course is going to talk about a subject called computability theory, which it really asks what you can compute with an algorithm in principle. That's-- was an active area of research in the earlier part of the 20th century.

It's pretty much closed off as a research subject these days, mainly because they answered all of their big questions. And so a mathematical field really only stays vital when it has problems to solve, and they really solved all of their interesting problems-- for the most part, not 100%. But for the most part, it sort of finished off in the 1950s-- just to say a little bit more about what we're going to talk about there.

When you're interested to know what kinds of problems you can solve with an algorithm-- there are problems that you might want to solve that you just can't solve. For example, given a specification for a computer problem you want to solve, whatever that specification might be-- say your algorithm actually is a sorting algorithm, for example-- and you want to write down that specification and have an automatic verifier that's going to check whether a program meets the specification. Well, that's just in principle impossible. You cannot make a verifier which is going to answer, in all cases, whether or not a program meets a certain specification. So with things like that, we will prove this semester.

Questions about mathematical truth-- if you're given a mathematical statement, is it true or is it false? It'd be great if you can write a computer program that would answer that problem. Well, it would not be great if you were a mathematician, because that would put us all out of business. But you can imagine that might be a nice thing to have, but you can't. I mean, there is no algorithm which can answer that question.

Well, along the way, we're going to introduce models of computation, like finite automata, which we'll see today, Turing machines, and some other models that we'll see along the way. The second half of the course, which is going to be after the midterm, we're going to shift gears and talk about complexity theory, which is instead of looking at what's computable in principle, you're going to look at what's computable in practice, so things that you can solve in a reasonable amount of time.

And, for example, I'm sure many of you are aware of the factoring problem, which has connections to the RSA cryptosystem, cryptography, and asks whether you can factor big numbers quickly. That's a problem we don't know the answer to. We just don't know how to factor big numbers quickly. But it's possible that there are algorithms out there that we haven't discovered yet that can do so.

It's connected with this very famous problem in the intersection of computer science and mathematics called the P versus NP problem, which many of you may have heard of. We'll talk about that. We'll spend a lot of time on that this term. And along the way, we'll talk about different measures of complexity, of computation, time and space, time and memory, theoretical memory, electrical space. That's going to be a big part of the course in the complexity theory part-- introduce other models of computation, such as probabilistic and interactive computation.

Talk about the expectations of the course. First of all, prerequisites. There are a bunch of prerequisites listed, 6.042, 18.062 , or maybe some other subject as well. The real thing is that this is a math class. This is a class where-- and it's not a beginning math class, this is a moderate-to-advanced math class. And I'm expecting people to have had some prior experience, of a substantial nature, with mathematical theorems and proofs.

We'll start off slow, but we're going to ramp up pretty fast. So if you haven't really got the idea or gotten comfortable with doing proofs, coming up with proofs to mathematical statements, that's going to be a concern. I would just be monitoring yourself and seeing how you're doing. Because the homeworks and the exams are going to count on your being able to produce proofs, and so you're going to be struggling if that's going to be a real-- something that you haven't had experience with.

And let me talk a little bit about the role of theory in computer science. This is a theory class, as you know. So before we jump into the material, I just thought it would be worth it for you to give you at least my perspective on the role of theoretical computer science within the field. So I've been in computer science for a long time.

I go back-- I'm sure I'm getting to be a dinosaur here-- but I go back to the days when you had punch cards. That's what we did when I was an undergraduate. And, obviously, things are very different now. And you can argue that computer science as a discipline has matured, and sort of the basic stuff has all been solved. Well, I would say there's a certain truth to that, but there's a certain way in which I would say that's not true. I think we're still at the very beginning, at least in certain respects, of computer science as a discipline.

For one thing, there are a lot of things that we do, a lot of things relating to computation, that we just don't know the answer to-- very fundamental things. Let's take as an example, how does the brain work? Obviously, the brain computes in a certain fashion. And we've made good progress, you can argue, with machine learning and all of those things that have very-- very powerful and doing very cool things.

But I would also say that at some deeper level, the methods that we have so far don't allow us to understand creativity. We're not close to being able to create a computer program that can do mathematics or that can do many of the creative kinds of things that human beings can do. I think machine learning, powerful as it is, is really successful only for a very narrow set of tasks.

And so I think there's probably something deeper and more fundamental going on that we're missing. That would be my hunch. Now, whether something like theoretical computer science is going to give you an answer there-- or this kind of theory, or some kind of theory-- I think some kind of theory has at least a decent shot at playing a role in helping us to understand computation in a deeper way. And the fact that we can't understand something as basic as, can you factor a big number quickly or not? You can't really say you understand computation until you can answer questions like that.

So I would argue that we have a really very primitive understanding of computation at this stage and that there is a lot that has yet to be discovered, not just on the technological side, but just on the very fundamental theoretical side that has a real shot at playing a role in affecting the practice of how we use computers. And so I think for that reason-- again, I'm not sure what kind of theory is going to be the most useful, but the theory we're going to cover in this course is a particularly elegant theory, and it has already paid off in many applications and in terms of our understanding of computation.

And I think, at least as a starting point, it's a good subject to learn. Certainly, I enjoy it, and I've spent a good chunk of my career doing that. So let's move on then and begin with the subject material. So we're going to talk about models of computation, as I mentioned. We want to try to understand computers, and we want to understand what computers can do.

But computers in the real world are pretty complicated objects, and they're really not nice to talk about mathematically. So we're going to talk about abstract models of computers that are much simpler but really capture-- just like models in general-- capture the important aspects of the thing we're trying to understand. And so we're going to look at several different kinds of models that vary in their capabilities and the way they approximate the real computers that we deal with every day.

And for starters, we're going to look at a very simple model called the finite automaton. And that's going to represent-- you can think of it as representing a computer that has a very small amount of memory and a very limited and small amount of memory. And we're going to look at the capabilities of those kinds of machines.

And what's nice about them is that you can understand them very well. And so more powerful models that we're going to look at later are going to be harder to understand in as deep a way. But for these, we can develop a very comprehensive theory. And so that's what we're going to do for the next lecture and a half.

So I'm starting off with an example. I'm presenting a finite automaton as a diagram-- we call it a state diagram. It has these circles and lines and labels on the lines and also on these circles. So what's going on here? So this is a finite automaton. I'm giving it the name M1.

And it has-- these circles are called states. So in this case, there were three states, q1, q2, and q3. Those are the labels there. There are arrows connecting states with each other. So these we'll call transitions. And they're going to tell you how to compute with this device.

And there's going to be a specially-designated starting state, which has an arrow coming in from nowhere. And there are other specially-designated states called accepting states, and that's going to have to do with how the machine computes. But those are the ones that have these double circles.

And so talking about the way it computes, the idea is pretty simple. The input is going to be some finite string of 0's and 1's, in this case. We might have other types of symbols that are allowed for other automata, but the example that I have here, it's going to be 0's and 1's. And the way you compute with the thing is you first put your finger-- which I can't do on Zoom, so I'll use the pointer-- you put your pointer on the starting state, the one that has the arrow coming in from nowhere.

First, you put your pointer there. And then are you start reading symbols from the input, one after the next. So let's take an example here, 01101. So you start reading those symbols, and you follow those transitions. So you go 0-- and you go back to the same state. Then you go-- the next symbol is a 1, so you go over to this state, from q1 to q2. Now you have another one that comes in. So now you're starting at q2, you have another one, so you follow its associated transition.

So if you notice, every state has an outgoing transition for 1 and another outgoing transition for 0. So there's always somewhere to go every time you read symbols from the input. So now you're at q2. You read that next, that third symbol, which is a 1. That's going to take you over to q3. And now you have a 0, which loops you back to where you were, and another 1, which loops you back to where you were.

And because you ended up at an accept, you say we accept that string. So that's going to be the output of this finite automaton. For each string, it's either going to accept it or reject it. So it's just a binary decision that is going to be made. It's sort of like a 1 or a 0 output, but we're calling it accept or reject.

So this one here, because it ended up at the accepting state, is accepted. But if you look at the second example, 00101, so you're going to have 0, 0, 1, 0, 1. Now we ended up at q2. That's not an accepting state. So therefore, we say we reject this input. OK? Very simple.

And now, for example, one of the questions you might want to ask, given one of these things, is, well, which are exactly those strings that the machine accepts? And a little bit of thought will help you understand that the only strings which are going to take you over to q3 are those strings that have a 11 appearing somewhere along the way, two consecutive 1's, and you will end up at the accepting state. I encourage you to think about that for a minute if not immediately obvious. But those are the strings that are going to be accepted by this machine.

And we call that collection of strings the language of the machine. So that set A of those strings that have a 11, for this particular machine, is the language of M1. We also say that M1 recognizes that language, recognizes A. And in terms of notation, we write that A is L of M1. A is the language of M1. So the language of a machine is exactly the set of strings that machine accepts. OK?

So one of the first things we're going to want to be able to do is take a machine and understand what its language is, what's the set of strings that that machine accepts. Another thing we might want to do is, given a language, build a machine which recognizes that language. And then understanding, what are the class of languages? Can you get any language from some machine, or are there going to be some languages that you can do and other languages that you cannot do? So those are the kinds of questions we're going to be asking about these finite automata. What kinds of things can those machines do, and what can they not do?

OK. Here's our next check-in. So wake up, everybody who's not paying attention. A check-in is coming. So we have more questions, though I can't keep-- are these three statements equivalent? What three statements?

AUDIENCE: At the bottom of the slide.

MICHAEL SIPSER: Oh, oh, oh, oh, oh, yes. Those three are equivalent. A is the language-- yeah, those mean the same thing. Not only are they equivalent, but they're just different ways of saying the same thing. That M1 recognizes the language is the same as saying that's the language of the machine and that A equals that L of M. That's all the same way of saying they all-- six of one, half a dozen of the other. It's two ways of saying the same thing.

OK, so let's pop up our poll and get that started. Whoops. Still showing the old one-- oh, here we go. Move it to the next question. OK. OK, so you understand the question here? Where do we end up after we read 101? What state are we in?

Do we end up in state q1, q2, or q3? OK? Go fast. This is a-- OK, so I think we got pretty much converged here. I think almost everybody got it right. The answer is indeed that you ended up in state q2. Because you go 1, 0, 1, and that's where you ended up, in state q2. So is this string accepted? No, because you didn't end up at an accept state. So this machine rejects 101.

OK, let's keep going. So now-- yeah. OK, so now we gave it this informal idea of a finite automaton. We're going to have to try to get a formal definition now, which is going to be a more mathematical way of saying the same thing that I just said. And the reason for having a formal definition is, for one thing, it allows us to be very precise. Then we'll know exactly what we mean by a finite automaton, and it should answer any questions about what counts and what doesn't count.

It also is a way of providing notation. So it'll help us describe finite automata. And sometimes there might be an automaton where the picture is just too big, so you might want to be able to describe it in some mathematical terminology rather than by giving a picture. Or maybe you're going to be asked to give a family of automata, where there is going to be a parameter, N, associated with the class of languages you're trying to describe with the automaton. And then it'll be more helpful to describe it in this formal notation rather than as a kind of a picture, because it might be infinitely many pictures that are being needed. So maybe examples of that will come up now.

So a finite automaton, we call it a 5-tuple. Don't be put off by that. A 5-tuple is just a list of five things. So a finite automaton, in our definition, is going to have five components. It's going to have Q, which is going to be a finite set of states, so it's going to be a finite set, which we'll designate as the states of the automaton. Sigma is the alphabet symbols of the automaton, another finite set.

Delta is the transition function. That tells us how the automaton moves from state to state. Those describes how those transition arrows-- those arrows which connected the states with each other-- it describes them in a mathematical way instead in terms of a picture. And the way I'm doing that is with a function. So delta is a function which takes two things.

So I'm hoping you've seen this notation before. I'll help you through it once, but this is the kind of thing I would expect you to have seen already. So we have Q cross sigma. So I'm going to give delta a state and an alphabet symbol. So Q is states, sigma is alphabet symbols. So you're going to get a state and an alphabet symbol, and it's going to give you back a state.

So describing it kind of a little bit more detail, delta, if you give it state q and symbol a equals r, that means q, when you read an a, you go to r. So that's the way this picture gets translated into a mathematical function, which describes those transitions. And then now q0 is going to be the starting state. That's the one with the arrow coming in from nowhere. And F is the set of accepting states. So there's only going to be one starting state, but there might be several different-- or possibly even 0-- accepting states. That's all legal when we have a finite automaton.

And so in terms of using the notation-- going back to the machine that we just had from the previous slide, which I've given you here again-- let me show you how I would describe this using this notation that comes out of the definition. So here is M1 again. It's this 5-tuple where Q now is the set-- q1, q2, q3-- that's the set of states. The input alphabet is 0, 1. It might vary in other automata. And f is the set q3, which has only the element q3, because this has just one accept state, q3.

So I hope that's helpful. Oh, of course, I forgot the transition function, which here I'm describing as a table. So the transition function says if you have a state and an input alphabet, you can look up in the table where you're supposed to go under the transition function according to the state and the alphabet symbol that you're given. So, for example, if we were in state q2 here getting a 0, then q2 goes back to q1 so that q2 on 0 is q1. But q2 on 1 here is q3. OK? So that's how that table captures this picture. OK? And it's just a function. It's a way of representing a function, a finite function, in terms of this table here.

So I realize, for some of you, this may be slow. We will ramp up in speed, but I'm trying to get us all together in terms of the language of the course here at the beginning. OK, so now let's talk about some more the computation, so strings and languages. A string is just a finite sequence of symbols from the alphabet.

This class is not going to talk about infinite strings. All of our strings are going to be finite. There's other mathematical theories of automata and so on that talk about infinite inputs and infinite strings. We're not going to talk about that. Maybe rarely, we'll make it very clear, we'll talk about an infinite string, but that's going to be an exception.

And a language is a set of strings. That's the traditional way that people in this subject refer to a set of strings. They call it a language-- really because the subject had its roots in linguistics, actually. And they were talking about-- they're trying to understand languages, human languages. So this is just a historical fact, and that's the terminology that's stuck.

OK, so two special string-- a special string and a special language. The empty string is the string of length 0. This is a totally legitimate string that you are going to run into now and then. And there's the empty language, which is the set with no strings. These are not the same. They're not even of the same type of object. So don't confuse them with one another. I mean, you can have a set, a language, which has just one element, which is the empty string. That is not the empty set. That is a set-- that is not the empty language. That is a language that has one element in it, namely, the empty string. So those are separate things.

OK, so here's a little bit of a mouthful here on the slide, defining what it means for an automaton to accept its input-- accepts its input string w. And we can define that formally. And it's a little technical looking, it's really not that bad. So if you have your input string w, which you can write as a sequence of symbols in the alphabet-- w1, w2, dot dot dot, wn, so like 01001. I'm just writing it out symbol by symbol here.

So what does it mean for the machine to accept that input? So that means that there's a sequence of states in the machine, sequence of states of members of Q. So a sequence from Q, these are the states of the machine that satisfy these three properties down here. First of all-- and I'm thinking about the sequence that the machine goes through as it's processing the input w.

So when does it accept w? If that sequence has the feature that it starts at the start state, each state legally follows the previous state according to the transition function. So that says the i-th member of the sequence is obtained by looking at the previous one-- the i minus first member of that sequence, the i minus first state in that sequence-- and then looking at what happens when you take the i-th input symbol. So as you look at the previous state and the next input symbol, you should get the next state. That's all that this is saying. And this should happen for each one of these guys.

And lastly, for this to be accepted, the very last member here, where we ended up at the end of the input-- so you only care about this at the end of the input-- you have to be in an accepting state. So you can mathematically capture this notion of going along this path. And that's what-- I'm just trying to illustrate that we could describe all this very formally-- I'm not saying that's the best way to think about it all the time-- but that it can be done. And I think that's something worth appreciating.

OK. So now in terms of, again, getting back-- we've said this once already, but in terms of the languages that the machine recognizes, it's the collection of strings that the machine accepts. Every machine accepts-- it might accept many strings, but it always recognizes one particular language, even if the machine accepts no strings-- then it recognizes the empty language. So a machine always recognizes one language, but it may have many, many strings that it's accepting. And we call that language the language of the machine. And we say that M recognizes that language. These three things mean the same thing. OK?

And now important definition-- I try to reserve the most important things or the highlighted things to be in this light blue color, if you can see that. We say a language is a regular language if there's some finite automaton that recognizes it. OK? So there are going to be some languages that have associated to them finite automata that actually solve those languages, that recognize those languages.

But there might be other languages-- and we'll see examples-- where you just can't solve them. You can't recognize them with a finite automaton. Those languages will not be regular languages. The regular ones are the ones that you can do with a finite automaton. That's the traditional terminology.

OK, so let's continue. Let's go on from there. So let's do a couple of examples. Here, again, is that same-- getting to be an old friend, that automaton M1. Remember, its language here is the set of strings that have the substring 11. That is that language A. Now, what do we know about A from the previous slide? Think with me. Don't just listen.

A is a regular language now, because it's recognized by some automaton. So whenever you find an automaton for a language, a finite automaton for language, we know that that language is a regular language. So let's look at a couple of more examples. So if you take the language-- let's call this one B, which is the strings that have an even number of 1's in them. So like the string 1101, would that be in B? No, because it has an odd number of 1's.

So the string 1111 has four 1's in it. That's an even number, so that string would be in B. The 0's don't matter for this language. So strings that have an even number of 1's, that's a regular language. And the way you would know that is you would have to make a finite automaton that recognizes that language. And I would encourage you to go and make that automaton. You can do it with two states. It's a very simple automaton.

But if you haven't had practice with these, I encourage you to do that. And actually, there are lots of examples that I ask you to solve at the end of chapter 1 in the book, and you definitely should spend some time playing with it if you have not yet seen finite automata before. You need to get comfortable with these and be able to make them. So we're going to start making some of them, but we're going to be talking about it at a sort of a more abstract level in a minute.

Basically, the reason why you can solve this problem, you can make a finite automaton which recognizes the language B, is because that finite automaton is going to keep track of the parity of the number of 1's it's seen before. This has two states, one of them remembering that it's seen an odd number of 1's so far, the other one remembering it's seen an even number of 1's before.

And that's going to be typical for these automata, finite automata. There's going to be several different possibilities that you may have to keep track of as you're reading the input, and there's going to be a state associated with each one of those possibilities. So if you're designing an automaton, you have to think about-- as you're processing the input-- what things you have to keep track of. And you're going to make a state for each one of those possibilities. OK? So you need to get comfortable with that.

Let's look at another example, the language C where the inputs have an equal number of 0's and 1's. That turns out to be not a regular language. So, in other words, what that means is there's no way to recognize that language with a finite automaton. You just can't do it. That's beyond the capabilities of finite automata. And that's a statement we will prove later.

OK. And our goal over the next lecture or so is to understand the regular languages, which you can do in a very comprehensive way. So we're going to start to do that now. So first, we're going to introduce this concept of regular expressions-- which, again, these are things you may have run into in one way or another before.

So we're going to introduce something called the regular operations. Now, I'm sure you're familiar with the arithmetical operations, like plus and times. Those apply to numbers. The operations we're going to talk about are operations that apply to languages. So they're going to take, let's say, two languages, you apply an operation, you're going to get back another language.

Like the union operation, for example, that's one you probably have seen before. The union of two languages here is a collection of strings that are in either one or the other. But there are other operations, which you may not have seen before, that we're going to look at-- the concatenation operation, for example. So that says you're going to take a string from the first language and another string from the second language and stick them together. And it's called concatenating them.

And you do that in all possible ways, and you're going to get the concatenation language from these two languages that you're starting with, A and B. The symbol we use for concatenation is this little circle. But often, we don't. We just suppress that and we write the two languages next to one another with the little circle implied. So this also means concatenation over here, just like this does.

And the last of the regular operations is the so-called star operation, which is a unary operation. It applies to just a single language. And so what you do is now you're going to take-- to get a member of the star language, you're going to take a bunch of strings in the original language, A, you stick them together. Any number of members of A, you stick them together, and that becomes an element of the star language. And we'll do an example in a second if you didn't get that. But one important element is that when you have the star language, you can also allow it to stick zero elements together, and then you get the empty string. So that's always a member of the star language, the empty string.

OK, so let's look at some examples. Let's say A is the language-- these are two strings here-- good, bad. And B is the language boy, girl. Now, if we take the union of those two, we get good, bad, boy, girl. That's kind of what you'd expect. And now let's take a look at the concatenation. Now, if you concatenate the A and the B language, you're going to get all possible ways of having an A string followed by all possible ways of having a B string. So you can get goodboy, goodgirl, badboy, badgirl.

Now, looking at the star, well, that applies to just one language. So let's say it's the good, bad language from A. And so the A star that you get from that is all possible ways of sticking together the strings from A. So using no strings, you always get the empty string. That's always guaranteed to be a member of A. And then just taking one element of A, you get good, or another element, bad. But now two elements of A, you get goodgood or goodbad, and so on. Or three elements of A, goodgoodgood, goodgoodbad.

And so, in fact, A star is going to be an infinite language if A itself contains any non-empty member. So if A is the empty language or if A contains just the language empty string, then A star will be not an infinite language. It'll just be the language empty string. But otherwise, it'll be an infinite language. I'm not even sure-- OK. I'm not-- [LAUGHS]

I'm ignoring the chat here. I'm hoping people are getting-- are you guys are getting your questions answered by our TAs? How are we doing, Thomas?

AUDIENCE: One question is, are the slides going to be posted?

MICHAEL SIPSER: Are the slides going to be posted? Well, the whole lecture is going to be recorded. Is it helpful to have the slides separately? I can post the slides. Sure. Remind me if I don't, but I'll try to do that. Yes, it is helpful. I will do that. Yeah. Yeah, I will post the slides. Just, Thomas, it's your job to remind me.

AUDIENCE: OK.

MICHAEL SIPSER: All right, good. So we talked about the regular operations. Let's talk about the regular expressions. So regular expressions are-- just like you have the arithmetical operations, then you can get arithmetical expressions, like 1 plus 3 times 7. So now we're going to make expressions out of these operations.

First of all, you have, the more atomic things, the building blocks of the expressions, which are going to be like elements of sigma, elements of the alphabet or the sigma itself as an alphabet symbol, or the empty language or the empty string. These are going to be the building blocks for the regular expressions. We'll do an example in a second. And then you combine those basic elements using the regular operations of union, concatenation, and star.

So these are the atomic expressions, these are the composite expressions. So, for example, if you look at the expression 0 union 1 star-- so we can also write that as sigma star. Because if sigma is 0 and 1, then sigma star is the same thing as 0 union 1-- sigma is the same as 0 union 1. And that just gives all possible strings over sigma. So this is something you're going to see frequently. Sigma star means this is the language of all strings over the alphabet we're working with at that moment.

Now, if you take sigma star 1, you just concatenate 1 onto all of the elements of sigma star, and that's going to give you all strings that end with a 1. Technically, you might imagine writing this with braces around the 1, but generally, we don't do that. We just-- single element sets, single element strings, we write without the braces, because it's clear enough without them, and it gets messy with them.

So sigma star 1 is all strings that end with 1. Or, for example, you take sigma star 11 sigma star, that is all strings that contain 11. And we already saw that language once before. That's the language of that other machine that we presented one or two slides back. OK? Right.

Yeah, but in terms of readings-- by the way, sorry, I don't know if it's helpful to you for me to do these interjections-- but the readings are listed also on the homework. So if you look at the posted homework 1, it tells you which chapters you should be reading now. And also, if you look at the course schedule, which is also on the home page, it has the whole course plan and which readings are for which dates. So it's all there for you.

And so our goal here-- this is not an accident that sigma star 11 sigma star happens to be the same language as we saw before from the language of that finite automaton M1. In fact, that's a general phenomenon. Anything you can do with a regular expression, you can also do with a finite automaton and vice versa. They are equivalent in power with respect to the class of languages they describe. And we'll prove that. OK?

So if you step back for a second and just let yourself appreciate this, it's kind of an amazing thing. Because finite automata, with the states and transitions, and the regular expressions, with these operations of union, concatenation, and star, they look totally different from one another. They look like they have nothing to do with one another. But, in fact, they both describe exactly the regular languages, the same class of languages. And so it's kind of a cool fact that you can prove, that these two very different looking systems actually are equivalent to one another.

Can we get empty string from empty set? Yeah. There are a bunch of exotic cases, by the way. So empty language star is the language which has just the empty string. If you don't get that, chew on that one. But that is true. OK, let's move on.

OK, let's talk about closure properties now. We're going to start doing something that has a little bit more meat to it, in terms of we're going to have our first theorem of the course coming here. And this is not a baby theorem. This is actually-- there's going to be some meat to this. And you're going to have to not totally-- this is not a toy. We're proving something that has real substance.

And the statement of this theorem says that the regular languages are closed, that really, the class of regular languages are closed under union, closed under the union operation. So what do I mean by that? So when you say a collection of objects is closed under some operation, that means applying that operation to those objects leaves you in the same class of objects.

Like the positive integers, the natural numbers, that's closed under addition. Because when you add two positive integers, you get back a positive integer. But they're not closed under subtraction. Because 2 minus 4, you get something which is not a positive integer. So closed means you leave yourself in the collection.

And the fact is that if you look at all the regular languages-- these are the languages that the finite automata can recognize-- they are closed under the union operation. So if you start off with two regular languages and you apply the union, you get back another regular language. And that's what the statement of this theorem is. I hope that's clear enough in the way I've written it. If A1 and A2 are regular, then A1 union A2 is also regular. That's what the statement of this is. And it's just simply that-- that's proving that the class of regular language is closed under union. So we're going to prove that.

So how do you prove such a thing? So the way we're going to prove that is you start off with what we're assuming. So our hypothesis is that we have two regular languages. And we have to prove our conclusion, that the union is also regular. Now, the hypothesis that they're regular, you have to unpack that and understand, what does that get you?

And them being regular means that there are finite automata that recognize those languages. So let's give those two finite automata names. So M1 and M2 are the two final automata that recognize those two languages, A1 and A2. That's what it means, that they're regular, that these automata exist. So let's have those two automata, M1 and M2, using the components as we've described, the respective state sets, input alphabet, transition functions, the two starting states and the two collections of accepting states.

Here I'm assuming that they're over the same alphabet. You could have automata which operate over different alphabets. It's not interesting to do that. It doesn't add anything. The proof would be exactly the same. So let's just not overcomplicate our lives and focus on the more interesting case, so assuming that the two input alphabets are going to be the same.

And from these two automata, we have to show that this language here, the union, is also a regular language. And we're going to do that by constructing the automaton which recognizes the union. That's really the only thing that we can do. So we're going to build an automaton out of M1 and M2 which recognizes the union language A1 union A2.

And the task of M is that it should accept its input if either M1 or M2 accept. And now what I'd like you to think about doing that, how in the world are we going to come up with this finite automaton M? And the way we do that is to think about, how would you do that union language? If I ask you-- I give you two automata, M1 and M2, and I say, here's an input, w. Is w in the union language?

That's the job that M is supposed to solve. And I suggest you try to figure out how you would solve it first. I mean, this is a good strategy for solving a lot of the problems in this course. Put yourself in the place of the machine you're trying to build. And so if you want to try to figure out how to do that, a natural thing is, well, you take w, you feed it into M1, and then you feed it into M2. And if M1 accepts it, great, then you know it's in the union. And if not, you try it out in M2 and see if M2 accepts it.

Now, you have to be a little careful, because you want to have a strategy that you can also implement in a finite automaton. And a finite automaton only gets one shot at looking at the input. You can't sort of rewind the input. You feed it first into M1 and then you feed it into M2 and operate in a sequential way like that. That's not going to be allowed in the way finite automata work.

So you're going to have to take it to the next level, be a little bit more clever. And instead of feeding it first into M1 and then and then into M2, you feed them into both in parallel. So you take M1 and M2, and you run them both in parallel on the input w, keeping track of which state each of those two automata are in. And then at the end, you see if either one of those machines is in an accepting state, and then you accept.

So that's the strategy we're going to employ in building the finite automaton M out of M1 and M2. So in terms of a picture, here's M1 and M2. Here is the automaton we're trying to build. We don't know how it's going to look like yet. And yeah, so kind of getting ahead of myself, but here is a strategy, as I just described, for M.

M is going to keep track of which state that M1 is in and which state M2 is in at any given moment. As we're reading the symbols of w, we're going to feed that into M1 and also into M2. And so the possibilities we have to keep track of in M are all the pairs of states that are in M1 and M2, because you're going to really be tracking M1 and M2 simultaneously.

So you have to remember which state M1 is in and also which state M2 is in. And so that really corresponds to what pair of states to remember, one from M1 and one from M2, and that's why I've indicated it like that. So M1 is in state q, M2 is in state r at some given point in time. And that's going to correspond to M being in the pair q comma r. That's just the label of this particular state of m that we're going to apply here. OK?

And then M is going to accept if either M1 and M2 is an accepting state. So it's going to be if either q or r is an accepting state, we're going to make this into an accepting state too. OK? Whoops. There we go. So let's describe this formally instead of by a picture, because we can do it both ways. And sometimes it's better to do it one way and sometimes the other way.

So now if we take-- the components of M now are the pairs of states from M1 and M2. Again, I'm writing this out literally, explicitly here, but you should make sure you're comfortable with this cross product notation. So this is the collection of pairs of states, q1 and q2, where q1 is in the state of the first machine, q2 is the state of the second machine.

The start state is you start at the two start states of the two machines. So this is q1, q2-- probably I should have not reused the Q notation. I should have called these r's-- now that I'm looking at that. But, anyway, I hope you're not confused by reusing this. q1 and q2 here are the specific start states of the two machines. These are just two other states, representative states of those machines.

Now, the transition function for the new machine is going to be built out of the transition functions from the previous machines. So when I have a pair, q, r, and I have the symbol a, where do we go? Which new pair do we get? Well, we just update the state from M1 and update the state from M2 according to their respective transition functions, and that's what's shown over here.

Now let's take a look at the accepting states for M. The natural thing to do is look at the set of pairs of states, where we have a pair of states-- a pair of accepting states, one from the first machine and one from the second machine. But if you're thinking with me, you realize that this is not the right thing.

What is DFAs? Did I would call them DFA somewhere? Oh, somebody else is probably doing that in the chat. The DFA-- careful what notation you're using. We haven't introduced DFAs yet. We'll do that next on Thursday. But these are DFAs. These are just finite automata, Deterministic Finite Automata. That's why the D.

Anyway, so this is actually not right, because if you think about what this is saying, it says that both components have to be accepting. And you want either one to be accepting. So this is not good. This would be the wrong way of defining it. That actually gives the intersection language. And really, kind of along the way, it's proving closure under intersection, which we don't care about but might be useful to have in our back pocket sometime in the future.

In order to get closure under a union, we have to write it this slightly more complicated looking way, which says the pair, what you want to have is either the first state is an accepting state and then any state for the second element, or any state for the first element and an accepting state for the second element. That's what it means to have the union, to be doing the union. OK?

So let's do-- oh, here's a quick check-in. So let's do another poll here. We thought we were done with these. Again-- oh, here we go. So it was too complicated to write it out in the polls, so I actually put it up on the slide for you. So all I'm asking is that if M1 has k1 states and M2 has k2 states, how many states does M have? Is it the sum, the sum of the squares, or the product?

OK, you have to think about the states of M, what do they look like? And come on, guys. All right, ending the poll, sharing results. Yes, indeed, it is-- most of you got it correct. It is C, the product. Because when you look at the number of pairs of states from M1 and M2, you need all possible pairs. And so it's the number of states in M1 times the number of states in M2. So make sure you understand that and think about that so that you're following and get this.

All right, so let's move on here. So we have another five minutes or so. Let's start thinking about closure under concatenation. So if we have two regular languages, so is the concatenation language. We're going to try to prove that. We won't finish, but we'll at least get our creative juices going about it. So we're going to do the same scheme here. We're going to take two machines for A1 and A2 and build a machine for the concatenation language out of those two.

So here are the two machines for A1 and A2 written down. And now here is the concatenation language. And I'm going to propose to you a strategy-- which is not going to work, but it still is going to be a good intuition to have. So what I'm going to do is I'm going to make a copy of-- OK, let's understand what M is supposed to do first. So M should accept its input. So think about this. M is doing the concatenation language.

So it's given a string. And it has to answer, is it in the concatenation language A1A2 or not? So it should accept it if there's some way to divide w into two pieces where M1 accepts the first piece and M2 accepts the second piece. So here would be the picture. OK? And now we have to try to make a machine which is going to solve this intuition. So how would you do that yourself? I'm giving you w. And you can simulate M1 and M2.

So the natural thing is you're going to start out by simulating M1 for a while and then shift into simulating M2 for a while, because that's what's supposed to be happening as you're processing the input. So I'm going to suggest that in terms of the diagram like this. So we have here M1 and M2 copied here. And what I propose doing is connecting M1 to M2 so that when M1 has accepted its input, we're going to jump to M2, because that's perhaps the first part of w. And now we're going to have M2 process the second part of w.

So the way I'm going to implement that is by declassifying the start state of M2, having transition symbols from the accepting states of M1 to M2, and then removing these guys here as accepting states. And we would have to figure out what sort of labels to apply here. But, actually, this reasoning doesn't work. It's tempting, but flawed. Because-- what goes wrong?

What happens is that-- it might be that when M1 has accepted an initial part of w and then it wants M2 to accept the rest, it might fail because M2 doesn't accept the rest. And what you might have been better off doing is waiting longer in M1, because there might have been some other later place to split w, which is still valid. Splitting w in the first place where you have M1 accepting an initial part may not be the optimal place to split w. You might want to wait later, and then you'll have a better chance of accepting w.

So I'm not sure if you quite follow that. But, in fact, it doesn't work. The question is where to split w, and it's challenging, because how do you know where to split w? Because it depends upon what-- it depends on y, and you haven't seen y yet. So when you try to think about it that way, it looks hopeless. But, in fact, it's still true. And we'll see how to do that on Thursday.

So just to recap what we did today, we did our introductory stuff, we defined finite automata, regular languages. We defined the regular operations and expressions. We showed that the regular languages are closed under union. We started closure under intersection, to be continued.