PROFESSOR: So we're going to talk about state machines, which is a topic that you're going to see in many further courses, because state machines model step by step processes. And of course, if you think about a computation, normally the way you think about it is that it's doing instructions, step by step, one after another, until it finally reaches termination. Likewise, various digital circuits move through stages or states until they produce a final answer. So state machines come up in at least those circumstances and in many others.

And the general model of state machine involves the idea that you can give it input and it responds to them, but we don't really need that for our purposes. So let's look at our example of a state machine. Here's maybe a particular simple one.

This is a description of a state machine that counts to 99. So the circles are indicating what its states are, and we've named them from 0 through 99. And then there's a final state called overflow, and that funny arc is an indication that if you're in overflow mode and you make another step by following the arc, you get back to overflow mode. But if you're in 0, you can make a step to 1. And if you're in 1, you can make a step to 2, and so on.

So starting off at the start state, which is generally indicated by that double mark. So to indicate where to start. Then the description of this machine, consistent in complete description is a set of states which are pictured as 0 through 99 plus overflow, a set of transitions which are indicated by the arrows, which is how one state can move to another state.

And the transitions can be summarised by saying that there of the form of i to i plus 1 for i between 0 and 99. And then there's a 99 transition to overflow, and once you're in overflow, you stay in overflow. So the picture at the top is saying exactly the same thing as we've said here in mathematical notation, explicitly describing what the transitions are.

So this is a machine that if you really build something to behave this way, it wouldn't be much use, because once it's overflowed, it's dead, because it stays there. Real machine to be useful would have a reset transition, which took overflow back to zero. But this illustrates the basic idea.

So let's look at a fun example from a Die Hard movie. I've forgotten which one it was. But there was one with Bruce Willis and Samuel Jackson playing a detective and a friend that he meets who helps him deal with a bad man, as is the case in all these movies. This time, the bad man's name is Simon.

And what Simon says to them as they stand behind the fountain in the park shown on the previous slide is that on the foundation, there should be two jugs, do you see them? A five gallon and a three gallon. Fill one of these jugs with exactly four gallons of water and place it on the scale, and the timer will stop. The timer and the scale are not shown in that picture, but there's a scale and a timer nearby. You must be precise, one ounce more or less will result in detonation. If you're still alive in five minutes, we'll speak.

OK. So let's think about formalizing this as a state machine to see what's going on. So but first of all, to understand the specification informally. What there is a three gallon jug and a five gallon jug that's capable of holding water, and an unlimited supply of water that you can get from the fountain. And the basic moves that you can make--

So with this set up, the kind of moves that you can make would be, say you fill up the three gallon jug with water, and then you could pour the three gallon jug into the five gallon jugs. And the three gallon jug was empty and the five gallon jug you knew had exactly three gallons in it. And then you can do other things like empty a jug and fill a jug and empty them into each other.

So let's model this as a state machine. And the first decision we need to make is what's the state going to be. Well, the state-- the obvious model for the state is the amount of water in each of the jugs. So b is the amount in the big jug and l is the amount the little jug. And what we know about b and l is that they're going to be some amount between 0 and 5 for b, and 0 and 3 for l.

We're going to quickly realize that we need them to be integers, but off hand we can allow them to be real numbers. Because after all, you could just pour some arbitrary amount of water into the big jug, any amount that it'll hold between 0 and 5. Although, that'll be dangerous, because as soon as you do that, you're going to lose track of exactly how much is in there and you won't know when you have four gallons or not.

So let's formalize the possible moves that we can have. So first of all, the start state is 0,0, because we start with both jugs empty. And then what are the possible transitions of how b and l moves? Well, let's see. The fill the little jug move amounts to saying that if you have an amount of b in big and l in little, then you can make a transition called fill the little jug into b, and big is still unchanged, and 3 in little for l less than 3.

I'm going to forbid the vacuous move where the little jug is already full and you try to fill it. That doesn't count, so l has to be less than 3, you can make it 3 by filling the little jug. Similarly, you could fill the big jug if b is less than 5. Then you can turn it into 5 by filling it.

And then you can empty the little jug, which is easy. If you go from b, l you go to b, 0. And likewise, you can empty the big jug. Those are the easy moves. A slightly more complicated move is pour the big jug into the little jug. Well if there's no overflow, what that means is that there's l in the little jug and b in the big job. And after you've poured the big jug into the little jug, there's b plus l in the little one, and nothing in the big one.

But let's be careful here about what exactly-- we're doing math, we're not sort of-- we're trying to get away from the metaphor. So what is no overflow means? It simply means that b plus l will fit. b plus l is less than or equal to 3.

All right. What's the other case of pouring the big jug into the little jug? Well that's when b plus l won't fit, in which case, you pour into the little jug. It's got l, so you pour in 3 minus l to fill it up. And then what's left in b is b minus the 3 minus little l that you poured. So that the other wise case, when there is overflow. And similarly, there are moves for pouring the little jug into the big jug.

So that then is a formal specification of the Die Hard machine and the moves that we're going to allow. Now, you could allow other moves like randomly pouring out a little water, or randomly filling up a little water. But if you did that, again, you lose track of how much water is in the jug. So these are the only safe moves. And they're the only ones we're going to model.

All right. So let's go back to Simon's challenge. He wants to disarm the bomb by getting exactly four gallons of water in the jug and measure it on the scale, or things will blow up. And how do you do it? Well, why don't you take a moment to think about it before I proceed to the next set of slides or before you let me proceed. But just to understand the rules again, watch the work here's how.

We're going to start off with both jugs empty. So we start off in state 0,0, and the first move is going to be to fill the big jug, which takes us to state 5, 0. Where the big jug has 5 and the little jug is still empty.

Then we're going to pour from the big into the little. So now, the little jug has 3. We're filling up the little one. That leaves two in the big jug.

Now we're going to empty the little one, we still have 2 left in the big one. And now we're going to pour from the big one into the little one, so the little one has 2 gallons and the big one is empty.

Now, we fill the big jug, and there's still 2 gallons in the little one and 5 gallons of the big one. Now we pour off from the five gallon jug until the one gallon jug is full, that's removing the 1 gallon that the 3 gallon jug still has capacity for. We reduced to full 2 gallons in the little jug, and four gallons, our target in the big jug. So we've accomplished it. And we're done. So the bomb doesn't go off.

All right. So the point of this exercise is really just to practice how the moves work and what the states are, but the questions I want to raise is suppose that we want to have a Die Hard once and for all scenario, in which we're tired of the remakes of these movies. And we proposed that in the next movie, that Simon, if he's still around, offers an alternative challenge, where instead of a three gallon and a five gallon jug, there's a three gallon jug and a nine gallon jug.

And now the question is, can you get four gallons into the big jug by pouring back and forth with rules like these, or can't you? And can you prove it? Well my guess is that you probably can figure out what's going on, because what's happening is if you start off with nothing in either jug, and you do these moves of filling up a jug and pouring one jug into another, you'll discover that the amount of water in any jug is always divisible by 3. This is a preserved invariant.

Any state that you can get to, starting off from 0,0, 3 will divide the number of gallons in each jug. We could state it this way. There's a property of states, property of b and l, which is the state, which is that 3 divides b-- that vertical line is a shorthand for the symbol divides. So three divides b, or b is a multiple of 3. 3 divides l. Synonym, l is a multiple of 3. And the claim is that that's always going to be true.

So in case that's not obvious, you might not have all the rules in your head. Let's just take a look at one of the more complicated rules. This was the rule where you're pouring the big jug into the little jug up until the little jug is full. And that transition is that if you're in state b, l, you move to b minus 3 minus l, and 3.

And if you look at this now, clearly if 3 divides both b and l, both components of the state you're at, then in the new state, well 3 obviously divides the contents of the little jug, which is 3. But three also divides the contents of the big jug, which is a multiple of 3, namely b minus 3, which is a multiple of 3 minus w, which is a multiple of 3.

When you take a linear combination of multiples of 3, you get a multiple of 3. And you look at all the other transitions, and they check equally well. If you're in a state b, l, and you move to a new state b prime, l prime, if 3 divides b and l, then 3 divides b prime and l prime. So this is what's called a preserved invariant.

And of course the corollary is that in the Die Hard once and for all game with the 3 gallon jug and the 9 gallon jug, you can't get to any state of the form 4, x, because 4 is not divisible by 3, and therefore Bruce is going to die.

Now what we've illustrated here is an argument that's known as Floyd's Invariant Principle, and it's really nothing but induction reformulated for state machines. The statement of what is invariant principle is that we're going to define a preserved invariant as a property of states. And a preserved invariants means it has the property that if you're in a state that has property p, and it's possible to make a single transition to state r, then r will also have property p. This is just like the induction step. We have to prove that p n implies p of n plus 1.

So if you have a preserved invariant, then if the property holds at the start state, then it's obvious that the property will hold for all of the states that you can possibly get to. That p of r will hold for all reachable states. And you can prove this by induction on the number of states, but I think it's clear that if you have a property that you begin with, and it doesn't change making a step, it's never going to change. And that's all that Floyd's invariant principle states.

So in particular, since the property p holds in all reachable states, if there is any final state which the machine reaches, then p is going to hold in that state. And what we saw was-- we're using the word preserved invariant to distinguish the definition here from another usage that's co-opted the word invariant to mean a property that's true in all states.

And while it's nice to know that some property is true in all states, the way you prove that is by having a preserved invariance. You want to distinguish the two. Technically if you look at this, the predicate that's always false is a preserved invariant. Because of the condition, as usual the way implication works. If the antecedent is false, then the entire implication is true. So if you're always false, then it's always the case that if false held in a state, which it never does, then it has to hold in any state you can get to, so that implication is true.

So just remember that preserved invariants that are constantly false exist, they are good preserved in variance. But they're not what other people would call an invariant. We use preserved invariance to prove that a property is true in all states. It's a methodology.

So let's do one more example to wrap this up. Suppose I have a robot on a grid, the integer grid, and we can think then of the coordinates of the integer as a pair of-- the coordinates of the robot as the coordinates of the square that it's in, a pair of non-negative integers. Now the way that this robot can move is we can make a diagonal move in one step. So it could move one step to the northeast or southeast or northwest or southwest and that's it.

And the question I want to ask is, suppose you start the robot off at the origin, at 0,0. Is there some way that it can wander around, following its moves, and get to a next state where it's moved 1 square over? That is, it gets to the square 0,1.

The answer to that is settled again by a preserved invariant. I don't know whether it's obvious to you yet, but it will be in a moment. I'm claiming you can't get to the square 0,1, and the reason is that there's are preserved invariant of that set of robot moves, namely the sum of the coordinates is even is an invariant. If the sum of the coordinates is even, it stays even.

And why is that? Well, any move adds plus or minus 1 to the coordinates of both x and y. Maybe x and y both go up by 1, in which case, the sum of x and y increases by 2. So if it was even, it stays even, or they both go down by 1, or maybe one goes up and the other goes down, in which case, the sum doesn't change in every case. If x plus y was even, it stays even. As a matter of fact, if it was odd, it stays odd. Moves actually preserve the parity of x plus y. But the invariant is that x plus y is even.

Now, what else is the case. Well 0,0-- 0 plus 0 is 0, which is even. And so we're in Floyd invariant principal case, where all the positions you can get to from the origin 0,0, which has an even sum, have to have an even sum. And since 1 plus 0 is odd, you can't get to that square, 1,0. It's not reachable.

So the parity invariant of the diagonally moving robot will set us up for an analysis of the 15 puzzle game. That's this logo that we've had on every slide in 6042 so far with 6 blank, 4, 2, on the diagonal. This is a game where there are these little numbered tiles that slide into the blank square, and the question is how to permute-- how to get from one permutations of the numbers to another.

It turns out that the analysis of that game depends on a parity invariant reminiscent of a slightly more sophisticated than the diagonally moving robot.

Let's look at one more example of using the invariant to verify a little algorithm that actually will be quite important as the course progresses, and that is fast exponentiation. So in this set up, a is a real number and b is a non-negative integer. I want to compute the b power of a.

Let's say b was 128, and I want to compute the 128th power of some real number a. Well, I can multiply a by itself 127 times, that would work fine. But you think about it, suppose I square a and then I square it again, and I square it again, then in about eight squarings, instead of 99 multiplications, I'm going to get to 8 of the 128th. Now if b is not a power of two, you adjust it slightly, and using that idea, you can compute the bth power of a much more rapidly than if you just naively multiplied out b minus 1 times.

So let's look at the pseudocode for this algorithm. Here, XYZ in our temporary registers y and z, hold-- y, z, and r all hold integers. And x holds this real number a. And you can read the code if you wish but in fact, I'm going to immediately jump to the slightly more abstract and easier to understand version of it as a state machine.

So what matters about this fast exponentiation algorithm as a state machine is that first of all, there are three states to real numbers, and a non-negative integer. And the start state is going to be that the number a is in the first register, or in the first location, first coordinate of the states. 1 is the real number in the second coordinate, and b the target exponent, is the non-negative integer in the third component.

The transitions are going to be as follows. Here's a simple one. If I have an amount x in the first location, y in the second location, z in the third, then if z is positive and even, then I'm going to square x, leave y alone, and divide z by 2. And that's the new state that I get. On the other hand, if z is odd and positive, then I'm going to square x, multiply y by x, and again take the quotient of z, divide z by 2.

OK. Why does this state machine do fast exponentiation, why is it correct? And the insight is that there's a preserved invariant of this machine. And the preserved invariant is that y times x to the z is always a to the b. SO let's see how to verify that, that yx to the z is equal to a to the b.

Let's just look at maybe the slightly more interesting of the two transition rules, which is when z is positive and odd, the xyz state moves to a new state, indicated in green. Where the new value of x is x squared, the new value of y is xy, and the new value of z is z minus 1 over 2. Remember, you went to the quotient of z divided by 2, and when z is odd, that means z minus-- it's literally z minus 1 over 2.

Well, do the new values satisfy the invariant if I plug-in the green values of x squared for y and xy for x-- I'm sorry, x squared for x, xy for y, and z minus 1 over 2 for z? Well let's see what happens. If I take the new value of y, which is xy, and I multiply it by the new value of x, which is x squared, raised to the new value of z, which is z minus 1 over 2.

Let's do a little algebraic simplification of that. Well, the x squared to the z minus 1 over 2 becomes x to the z minus 1. And I'm just carrying over the xy. And then that simplifies to simply y times x times x to the z minus 1, or yz to the z, which we assumed was equal to a to the b, so sure enough, the new values of x, y, and z satisfy the invariant. It's a preserved invariant, and an even simpler argument applies to the other transition, when z is positive and even. So we verified that this is a preserved invariant.

Now at the start, remember, we start off with the real number a in register x, the real number b in z, and the real number 1 in y, which is the accumulator. And 1 times a to b is equal to a to b. So this is a-- this preserved invariant is true of the start state.

That means by Floyd's Invariant Principle, that it is true at this the final state, if and when the thing stops. Well, when does this machine stop? As long as z is positive, it can keep moving. So it gets stuck when z is 0?

What happens if it ever gets to z is 0? If it gets stuck, then the invariant says that yx to the 0 has to equal a to the b. But of course, yx to the 0 is nothing but y. And what we conclude is, that sure enough, this machine leaves the desired exponential value in the register y, which is where we get the answer. And that's why this algorithm is correct.

Now another aspect of what's going on here is proving that the algorithm does terminate. So let me just say a word that Floyd distinguished sort of these two aspects of program correctness that typically come up. One is showing that if you get an answer, it's correct, and that's what we just did. If this machine stops, if it ever gets to the case where z is 0, then y has the right answer. But we haven't proved that it stops.

So we've shown that it's partially correct like a partial function. It might not be defined everywhere, we haven't shown that yet, but when it is defined, if gives the right answer.

The other aspect of correctness is termination, showing in effect, that the function is total, that the program always does stop. Well in this case, there's an easy way to see why it always stops. Because at every transition, z is being halved or more. z is a non-negative integer valued variable.

And since we're halving it, or making it even smaller than half of it at every step, it means that since it starts with the value z, it can't get smaller, more than log to the base 2 of b times, because by then, it would have hit 0. And so we can be sure that this machine makes it most log to the base 2 of b transitions. And then it has to get stuck at the only place it can get stuck, which is when z equals 0.

And there is a picture of my friend, an early colleague, Bob Floyd, whom I met at the very beginning of my career at Carnegie Mellon University. We worked together for about one year before he went off to Stanford. And you can read much more about his life in a warm and detailed eulogy written by his best friend, Don Knuth. Floyd won the Turing Award for his major contributions, both to program correctness and to programming language parsing.