PROFESSOR: The greatest common divisor of two numbers is easy to compute. And that's a factor will play a crucial role in the number three we're going to develop, and the properties of some of the modern codes that are based on number theory. The efficient way to compute the GCD of two numbers is based on a classical algorithm known as Euclidean algorithm, which is several thousand years old. And let's describe how it works now.

So the Euclidean algorithm is based on the following lemma, which we'll call the remainder lemma, and it says that if a and b are two integers, then the greatest common divisor of a and b is the same as the greatest common divisor of b, and the remainder of a divided by b-- providing, of course, b is not 0, because otherwise you can't divide by b.

OK how do you make sense out of this? Why is this true? Well, it's actually a very easy proof. Remember that by the so-called division algorithm-- or it's really a theorem-- if you divide a by b and we're doing integer division, what that means is you find a quotient of a divided by b in the quotient, and a remainder. And the quotient has the property that q times b plus the remainder is equal to a. The remainder is always going to be smaller than a. It will be the range from 0 up to, but not including, a.

OK, if you look at this simple expression, what becomes apparent is that if you've got a divisor of two out of three of these terms, then it's going to divide the third term. So for example, if you have a divisor of b and r, then the sum of those two things is also going to have the same divisor, which means that a will have that divisor. If something divides both a and b, then it divides r. And if it divides b and r, it divides a. And that means that a and b and b and r have exactly the same divisors. They not only have the same greatest common divisor, all their divisors are the same. So obviously, the greatest one is the same. And that proves this key remainder lemma.

Well, the remainder lemma now gives us a very lovely way to compute the GCD. And here's an example. Suppose I want to compute the GCD of 899 and 493. A is 899, b is 493. Well, so I want this GCD, 899 of 493. Well, according to the remainder lemma, if I divide 899 by 493, I get a quotient of 1, and a remainder of 406. So that means that 899 and 493 have the same GCD as 493 and 406. That is the original number b, and the new remainder 406.

But now, I can divide 493 by 406. I get a quotient of zero and a remainder of 87. So 406 and

87 have the same GCD. Dividing 406 by 87, I get that 87 and 58 have the same GCD. Dividing 87 by 58, I get that 58 and 29 have the same GCD. And now I win, because look, when I divide 58 by 29, I get a remainder of 0. And the GCD of anything and 0 is that thing. So the GCD of 29 and 0 is 0. I guess the only exception is the GCD of 0 and 0, which is not defined. But if it's not 0, then the GCD of x and 0 is x.

And there it is. So I've just found that the GCD of 899 and 493 is 29. And this is a quite fast algorithm, because I keep dividing the numbers that I have by each other, and it gets small fast. We'll be more precise about that in a minute.

OK, it's a good exercise in state machine thinking and practice in program verification to reformulate the Euclidean algorithm, or formulate it explicitly as a state machine. It's a very simple kind of state machine. The states of this Euclidean algorithm state machine will be pairs of non-negative integers. So the states are n cross n, the Cartesian product of the non-negative integers, with itself. The start state is going to be the pair a, b, whose GCD I want to compute.

And the transitions are simply repeatedly applying the remainder lemma. Namely, if I'm in state x, y, where you think of x as and y as the GCD that I'm trying to compute, I simply convert x and y to y, and the remainder of x divided by y. And I keep doing that as long as y is not 0.

OK, very simple state machine-- really, just one transition rule. Well, according to the lemma, since I'm replacing the GCD of x and y by the GCD of y and the remainder of x divided by y, the GCD is actually staying constant. This transition preserves the GCD that's left in the pair of registers, x and y. So what we can say is that since the GCD of x and y doesn't change from one step to another, we can say that the GCD of x and y at any point is equal to its original value, which is the GCD of a and b.

So in other words, this equation, GCD of x and y in the current state is equal to GCD of a and b, the GCD of a and b that we started with, is a preserved invariant of the state. So p of a state xy, the property that GCD of x and y is the original GCD is a preserved invariant of the state machine. Moreover, p of start is trivially true, because at the start, x and y are a equals b. So p of x and y is just saying the GCD of a and b is equal to GCD of a and b.

Cool. So I've got that this property is true at the start, and it's preserved by the transitions. So the invariance principle tells me that if the program stops, I'm going to have the GCD of x and

y when it terminates is equal to the actual GCD that I want. And that enables us to prove partial correctness. The claim is that if this program terminates-- we haven't determined that it does yet-- but at termination, if any, I claim that x is left in-- that the GCD of a and b is left in register x. The value of x at the end is going to be the GCD of and and b.

Well, why is that? Well, look-- at termination, what we know is that y is 0. That's the only way that this procedure stops, because otherwise, the transition rule is applicable. So that means that when y equals 0 at termination, what we have is that since y is 0, GCD of x and y is equal to the GCD of x and 0. And that's equal to x, assuming, again, that x is positive, or not 0.z

So x is the GCD of x and y. And by the invariant, the GCD of x and y is equal to the GCD of a and b. So I've prove this little fact. This procedure correctly computes the GCD of a and b, leaving the answer in register x, if it terminates. Well, of course it terminates, and it terminates fast. So let's see why.

Notice that at each transition, we're going to replace x by y, and y by the remainder of x divided by y. Let's just assume for simplicity that of the [? pairings, ?] y that x is the bigger one. So there's two cases of why these numbers are getting small fast. The first case is suppose that y is less than x over 2, or less than or equal to x over 2. Well, since at this step, you're going to replace x by y, it means that you're replacing x by something that's less than half x. So x gets halved at this step.

What about if y is big? Well, if y is bigger than x over 2, then the remainder of x divided by y is simply x minus y. And it's going to be less than x over 2. But that's going to be the value of y after the next step. So y is going to be halved either at this step or the next step when it's replaced by the remainder of x and y. And the net result is that y it gets cut in half, or even smaller, at every other step, which means that this procedure can't continue for more than twice the log to the base 2 of the original value of y, which is b number of steps, because that's how many halves you can do before you start hitting 0.

So we've just shown that this procedure holds in logarithmic number of steps, which is the same as saying that it's about the length of b in binary, and even fewer steps than the length of b in decimal. The GCD algorithm is really very efficient.