PROFESSOR: Another basic proof technique is called Proof By Cases, which we prove something by breaking it up into pieces that are easy to prove but that together cover all possibilities.

Let's look at an explicit, simple example from computer science. Here's a Java logical expression. The way to decipher this is that the double vertical bar means "or" in Java. And the double ampersand means "and" in Java. So this is a conditional test, an IF test, that is the guard on a bunch code to be executed if this test comes out to be true.

Let's read the test. If x is greater than 0, or x is less than or equal to 0, and y is greater than 100, go ahead and do the code that's in there indicated by the vertical dots.

We're going to assume here that x and y are variables that are declared to be of type floating point, [? or a ?] real number, or integers for that matter. OK. Now what I claim is this code can be improved if it's rewritten in the following way. Namely, if x is greater than 0 or y is greater than 100.

So the claim is that these two hunks of codie, if I just replace this test, which has three components that require an extra step to evaluate some cases, by this code the programs are going to behave exactly the same way. And therefore it's just more efficient and easier to understand. One is one step faster if I replace this longer expression by this shorter expression.

Now how do I argue that these two pieces of code are going to behave in exactly the same way, or come up with the same final output? They won't behave exactly the same because one will be faster than the other. But they're going to yield the same results. OK, let's consider how these two behave in two cases.

The first case will be that the number x really is positive, that it's greater than 0. What happens then? Well, the first test above in the "or" comes out to be true. And that means that the whole "or" expression is true. Because when you have a true "or" anything at all, it comes out to be true. And you go ahead and execute code that follows.

Likewise, the second expression starts with x greater than 0 "or." So it comes out to be true. So, in this case, if x is greater than 0, both conditional expressions will allow the code that follows them to be executed. Because they both evaluate to true. OK.

The next case is that x is less or equal to 0. Let's see what happens then. Well, in the top expression, since x is less than or equal to 0, that first expression, x greater than 0, if evaluated, returns false. And same thing in the second expression. The initial test x greater than 0 returns false.

Now one of the things that the ways "or" works is that if you have an "or" of a bunch of things, if the first thing is false, you ignore it and just proceed to the other things to see how they come out. So what that means is that in both of these expressions, since the first test in the sequence of things that are being "ored" together came out to be false, I can just ignore them.

The code is going to behave as though, after the false was detected, it's just going to behave in the same way that the rest of the test says to behave. Well, in the top case, the expression to be check now is that x is less than or equal to 0, and y is greater than 100.

But what do we know? Well, x less than or equal to 0 in this case. So this test comes out to be true. And we have something of the form true "and" something or other. That means that the net outcome of this expression, it depends entirely on the something or other. That is, it depends entirely on whether y is greater than 0. Because the x is less than or equal to 0.

And so this expression can be simplified. It's going to behave exactly according to whether or not y is greater than 100. So look what I've just done. I've argued that in this case, both of these tests cards act like y is the test y greater than 100. Which is, they behave the same in this case as well.

So what I just figured out was that, in both cases, these two expressions yield the same result. And the only possible cases are that x is greater than 0, or x is less than or equal to 0. So in all cases, they're the same, and we're done. That's why it's safe to replace the upper complicated expression by the lower, less complicated expression.

So, in general as I said, reasoning by cases breaks complicated problems into easier sub problems. Which is what we just saw there. [? It ?] wouldn't be clear how to prove that these two things were equivalent, but I chose those cases and it made each case easy to figure out that the two things with the same.

Now, the truth is that there are some philosophers who worry about reasoning by cases for kind of subtle reasons. They're called intuitionists. And here's what bothers them. Let me illustrate it.

There's a million dollar Clay Institute question. One of the dozen or so questions that are considered to be the major open problems in various disciplines of mathematics. And one of the disciplines of mathematics is complexity theory in computer science, computational complexity theory. This question is known as the p equals np question.

And we're actually going to talk about it a fair amount in just the coming few lectures. But for now, it doesn't matter what it means. Well, I'll tell you what it means. p stands for polynomial time, and np stands for nondeterministic polynomial time. I'm not going to define not deterministic polynomial time, but it would be momentous if those two things were equal.

And everyone expects that they're not equal, but no one knows how to prove that. So the million dollar question is, is p equal to np, yes or no? And you get a million dollars for settling this question. Now, I claim that in fact the answer to this question is on my lecture table. And I will show it to you in class tomorrow.