[MUSIC PLAYING BY J.S. BACH]

PROFESSOR: Hi. You've seen that the job of a programmer is to design processes that accomplish particular goals, such as finding the square roots of numbers or other sorts of things you might want to do. We haven't introduced anything else yet. Of course, the way in which a programmer does this is by constructing spells, which are constructed out of procedures and expressions. And that these spells are somehow direct a process to accomplish the goal that was intended by the programmer. In order for the programmer to do this effectively, he has to understand the relationship between the particular things that he writes, these particular spells, and the behavior of the process that he's attempting to control.

So what we're doing this lecture is attempt to establish that connection in as clear a way as possible. What we will particularly do is understand how particular patterns of procedures and expressions cause particular patterns of execution, particular behaviors from the processes.

Let's get down to that. I'm going to start with a very simple program. This is a program to compute the sum of the squares of two numbers. And we'll define the sum of the squares of x and y to be the sum of the square of x-- I'm going to write it that way-- and the square of y where the square of x is the product of x and x. Now, supposing I were to say something to this, like, to the system after having defined these things, of the form, the sum of the squares of three and four, I am hoping that I will get out a 25. Because the square of three is nine, and the square of four is 16, and 25 is the sum of those.

But how does that happen? If we're going to understand processes and how we control them, then we have to have a mapping from the mechanisms of this procedure into the way in which these processes behave. What we're going to have is a formal, or semi-formal, mechanical model whereby you understand how a machine could, in fact, in principle, do this. Whether or not the actual machine really does what I'm about to tell you is completely irrelevant at this moment.

In fact, this is an engineering model in the same way that, electrical resistor, we write down a model v equals i r, it's approximately true. It's not really true. If I put up current through the resistor it goes boom. So the voltage is not always proportional to the current, but for some purposes the model is appropriate. In particular, the model we're going to describe right now, which I call the substitution model, is the simplest model that we have for understanding how procedures work and how processes work. How procedures yield processes. And that substitution model will be accurate for most of the things we'll be dealing with in the next few days. But eventually, it will become impossible to sustain the illusion that that's the way the machine works, and we'll go to other more specific and particular models that will show more detail.

OK, well, the first thing, of course, is we say, what are the things we have here? We have some cryptic symbols. And these cryptic symbols are made out of pieces. There are kinds of expressions. So let's write down here the kinds of expressions there are. And we have-- and so far I see things like numbers. I see things like symbols like that.

We have seen things before like lambda expressions, but they're not here. I'm going to leave them out. Lambda expressions, we'll worry about them later. Things like definitions. Things like conditionals. And finally, things like combinations.

These kinds of expressions are-- I'll worry about later-- these are special forms. There are particular rules for each of these. I'm going to tell you, however, the rules for doing a general case. How does one evaluate a combination? Because, in fact, over here, all I really have are combinations and some symbols and numbers. And the simple things like a number, well, it will evaluate to itself.

In the model I will have for you, the symbols will disappear. They won't be there at the time when you need them, when you need to get at them. So the only thing I really have to explain to you is, how do we evaluate combinations?

OK, let's see. So first I want to get the first slide. Here is the rule for evaluating an application. What we have is a rule that says, to evaluate a combination, there are two parts, three parts to the rule. The combination has several parts. It has operators and it has operands. The operator returns into a procedure. If we evaluate the operator, we will get a procedure. And you saw, for example, how I'll type at the machine and out came compound procedure something or other.

And the operands produce arguments. Once we've gotten the operator evaluated to get a procedure, and the argument is evaluated to get argument-- the operand's value to get arguments-- we apply the procedure to these arguments by copying the body of the procedure, which is the expression that the procedure is defined in terms of. What is it supposed to do? Substituting the argument supplied for the formal parameters of the procedure, the formal parameters being the names defined by the declaration of the procedure. Then we evaluate the resulting new body, the body resulting from copying the old body with the substitutions made.

It's a very simple rule, and we're going to do it very formally for a little while. Because for the next few lectures, what I want you to do is to say, if I don't understand something, if I don't understand something, be very mechanical and do this.

So let's see. Let's consider a particular evaluation, the one we were talking about before. The sum of the squares

of three and three. What does that mean? It says, take-- well, I could find out what's on the square-- it's some procedure, and I'm not going to worry about the representation, and I'm not going to write it on the blackboard for you. And I have that three represents some number, but if I have to repeat that number, I can't tell you the number. The number itself is some abstract thing. There's a numeral which represents it, which I'll call three, and I'll use that in my substitution.

And four is also a number. I'm going to substitute three for x and four for y in the body of this procedure that you see over here. Here's the body of the procedure. It corresponds to this combination, which is an addition.

So what that reduces to, as a reduction step, we call it, is the sum of the square of three and the square of four. Now, what's the next step I have to do here? I say, well, I have to evaluate this. According to my rule, which you just saw on that overhead or slide, what we had was that we have to evaluate the operands-- and here are the operands, here's one and here's the next operand-- and how we have to evaluate procedure. The order doesn't matter. And then we're going to apply the procedure, which is plus, and magically somehow that's going to produce the answer. I'm not to open up plus and look inside of it.

However, in order to evaluate the operand, let's pick some arbitrary order and do them. I'm going to go from right to left. Well, in order to evaluate this operand, I have to evaluate the parts of it by the same rule. And the parts are I have to find out what square is-- it's some procedure, which has a formal parameter x. And also, I have an operand which is four, which I have to substitute for x in the body of square.

So the next step is basically to say that this is the sum of the square of three and the product of four and four. Of course, I could open up asterisk if I liked-- the multiplication operation-- but I'm not going to do that. I'm going to consider that primitive. And, of course, at any level of detail, if you look inside this machine, you're going to find that there's multiple levels below that that you don't know about.

But one of the things we have to learn how to do is ignore details. The key to understanding complicated things is to know what not to look at and what not compute and what not to think. So we're going to stop this one here and say, oh, yes, this is the product of two things. We're going to do it now.

So this is nothing more than the sum of the square of three and 16. And now I have another thing I have to evaluate, but that square of three, well, it's the same thing. That's the sum of the product of three and three and 16, which is the sum of nine and 16, which is 25.

So now you see the basic method of doing substitutions. And I warn you that this is not a perfect description of what the computer does. But it's a good enough description for the problems that we're going to have in the next few lectures that you should think about this religiously. And this is how the machine works for now. Later we'll get

more detailed.

Now, of course, I made a specific choice of the order of evaluation here. There are other possibilities. If we go back to the telestrator here and look at the substitution rule, we see that I evaluated the operator to get the procedures, and I evaluated the operands to get the arguments first, before I do the application. It's entirely possible, and there are alternate rules called normal order evaluation whereby you can do the substitution of the expressions which are the operands for the formal parameters inside the body first. And you'll get also the same answer.

But right now, for concreteness, and because this is the way our machine really does it, I'm going to give you this rule, which has a particular order. But that order is to some extent arbitrary, too. In the long run, there are some reasons why you might pick one order or another, and we'll get to that later in the subject.

OK, well now the only other thing I have to tell you about just to understand what's going on is let's look at the rule for conditionals. Conditionals are very simple, and I'd like to examine this. A conditional is something that is if-- there's also cond, of course-- but I'm going to give names to the parts of the expression. There's a predicate, which is a thing that is either true or false. And there's a consequent, which is the thing you do if the predicate is true. And there's an alternative, which is the thing you do if the predicate is false.

It's important, by the way, to get names for, to get names for, the parts of things, or the parts of expressions. One of the things that every sorcerer will tell you is if you have the name of a spirit, you have power over it. So you have to learn these names so that we can discuss these things.

So here we have a predicate, a consequent, and an alternative. And, using such words, we see that an if expression, the problems you evaluate to the predicate expression, if that yields true, then you then go on to evaluate the consequent. Otherwise, you evaluate the alternative expression.

So I'd like to illustrate that now in the context of a particular little program. Going to write down a program which we're going to see many times. This is the sum of x and y done by what's called Peano arithmetic, which is all we're doing is incrementing and decrementing. And we're going to see this for a little bit. It's a very important program.

If x equals zero, then the result is y. Otherwise, this is the sum of the decrement of x and the increment of y. We're going to look at this a lot more in the future.

Let's look at the overhead. So here we have this procedure, and we're going to look at how we do the substitutions, the sequence of substitutions. Well, I'm going to try and add together three and four. Well, using the first rule that I showed you, we substitute three for x and four four y in the body of this procedure. The body of the

procedure is the thing that begins with if and finishes over here. So what we get is, of course, if three is zero, then the result is four. Otherwise, it's the sum of the decrement of three and the increment of four.

But I'm not going to worry about these yet because three is not zero. So the answer is not four. Therefore, this if reduces to an evaluation of the expression, the sum to the decrement of three and the increment of four. Continuing with my evaluation, the increment I presume to be primitive, and so I get a five there. OK, and then the decrement is also primitive, and I get a two. And so I change the problem into a simpler problem. Instead of adding three to four, I'm adding two to five. The reason why this is a simpler problem is because I'm counting down on x, and eventually, then, x will be zero.

So, so much for the substitution rule. In general, I'm not going to write down intermediate steps when using substitutions having to do with ifs, because they just expand things to become complicated. What we will be doing is saying, oh, yes, the sum of three and four results in the sum of two and five and reduces to the sum of two and five, which, in fact, reduces to the sum of one and six, which reduces to the sum of zero and seven over here, which reduces to a seven. That's what we're going to be seeing.

Are there any questions for the first segment yet? Yes?

STUDENT: You're using one plus and minus one plus. Are those primitive operations?

PROFESSOR: Yes. One of the things you're going to be seeing in this subject is I'm going to, without thinking about it, introduce more and more primitive operations. There's presumably some large library of primitive operations somewhere. But it doesn't matter that they're primitive-- there may be some manual that lists them all. If I tell you what they do, you say, oh, yes, I know what they do. So one of them is the decrementor-- minus one plus-- and the other operation is increment, which is one plus.

Thank you. That's the end of the first segment.

[MUSIC PLAYING BY J.S. BACH]

PROFESSOR: Now that we have a reasonably mechanical way of understanding how a program made out of procedures and expressions evolves a process, I'd like to develop some intuition about how particular programs evolve particular processes, what the shapes of programs have to be in order to get particular shaped processes. This is a question about, really, pre-visualizing.

That's a word from photography. I used to be interested in photography a lot, and one of the things you discover when you start trying to learn about photography is that you say, gee, I'd like to be a creative photographer. Now, I know the rules, I push buttons, and I adjust the aperture and things like that. But the key to being a creative

person, partly, is to be able to do analysis at some level. To say, how do I know what it is that I'm going to get on the film before I push the button. Can I imagine in my mind the resulting image very precisely and clearly as a consequence of the particular framing, of the aperture I choose, of the focus, and things like that?

That's part of the art of doing this sort of thing. And learning a lot of that involves things like test strips. You take very simple images that have varying degrees of density in them, for example, and examine what those look like on a piece of paper when you print them out. You find out what is the range of contrasts that you can actually see. And what, in a real scene, would correspond to the various levels and zones that you have of density in an image.

Well, today I want to look at some very particular test strips, and I suppose one of them I see here is up on the telestrator, so we should switch to that. There's a very important, very important pair of programs for understanding what's going on in the evolution of a process by the execution of a program. What we have here are two procedures that are almost identical. Almost no difference between them at all. It's a few characters that distinguish them. These are two ways of adding numbers together.

The first one, which you see here, the first one is the sum of two numbers-- just what we did before-- is, if the first one is zero, it's the answer of the second one. Otherwise, it's the sum of the decrement of the first and the increment of the second. And you may think of that as having two piles. And the way I'm adding these numbers together to make a third pile is by moving marbles from one to the other. Nothing more than that. And eventually, when I run out of one, then the other is the sum.

However, the second procedure here doesn't do it that way. It says if the first number is zero, then the answer is the second. Otherwise, it's the increment of the sum of the decrement of the first number and the second. So what this says is add together the decrement of the first number and the second-- a simpler problem, no doubt-- and then change that result to increment it.

And so this means that if you think about this in terms of piles, it means I'm holding in my hand the things to be added later. And then I'm going to add them in. As I slowly decrease one pile to zero, I've got what's left here, and then I'm going to add them back. Two different ways of adding. The nice thing about these two programs is that they're almost identical. The only thing is where I put the increment. A couple of characters moved around.

Now I want to understand the kind of behavior we're going to get from each of these programs. Just to get them firmly in your mind-- I usually don't want to be this careful-- but just to get them firmly in your mind, I'm going to write the programs again on the blackboard, and then I'm going to evolve a process. And you're going to see what happens. We're going to look at the shape of the process as a consequence of the program.

So the program we started with is this: the sum of x and y says if x is zero, then the result is y. Otherwise, it's the

sum of the decrement of x and the increment of y.

Now, supposing we wish to do this addition of three and four, the sum of three and four, well, what is that? It says that I have to substitute the arguments for the formal parameters in the body. I'm doing that in my mind. And I say, oh, yes, three is substituted for x, but three is not zero, so I'm going to go directly to this part and write down the simplified consequent here. Because I'm really interested in the behavior of addition.

Well, what is that? That therefore turns into the sum of two and five. In other words, I've reduced this problem to this problem. Then I reduce this problem to the sum of one and six, and then, going around again once, I get the sum of zero and seven. And that's one where x equals zero so the result is y, and so I write down here a seven. So this is the behavior of the process evolved by trying to add together three and four with this program.

For the other program, which is over here, I will define the sum of x and y. And what is it? If x is zero, then the result is y-- almost the same-- otherwise the increment of the sum of the decrement of x and y. No. I don't have my balancer in front of me.

OK, well, let's do it now. The sum of three and four. Well, this is actually a little more interesting. Of course, three is not zero as before, so that results in the increment of the sum of the decrement of x, which is two and four, which is the increment of the sum of one and-- whoops: the increment of the increment.

What I have to do now is compute what this means. I have to evaluate this. Or what that is, the result of substituting two and four for x and y here. But that is the increment of the sum of one and four, which is-- well, now I have to expand this. Ah, but that's the increment of the increment of the increment of the sum of zero and four.

Ah, but now I'm beginning to find things I can do. The increment of the increment of the increment of-- well, the sum of zero and four is four. The increment of four is five. So this is the increment of the increment of five, which is the increment of six, which is seven. Two different ways of computing sums.

Now, let's see. These processes have very different shapes. I want you to feel these shapes. It's the feeling for the shapes that matters. What's some things we can see about this? Well, somehow this is sort of straight. It goes this way-- straight. This right edge doesn't vary particularly in size.

Whereas this one, I see that this thing gets bigger and then it gets smaller. So I don't know what that means yet, but what are we seeing? We're seeing here that somehow these increments are expanding out and then contracting back. I'm building up a bunch of them to do later. I can't do them now. There's things to be deferred.

Well, let's see. I can imagine an abstract machine. There's some physical machine, perhaps, that could be built to

do it, which, in fact, executes these programs exactly as I tell you, substituting character strings in like this. Such a machine, the number of such steps is an approximation of the amount of time it takes. So this way is time. And the width of the thing is how much I have to remember in order to continue the process. And this much is space. And what we see here is a process that takes a time which is proportional to the argument x. Because if I made x larger by one, then I'd had an extra line.

So this is a process which is space-- sorry-- time. The time of this process is what we say order of x. That means it is proportional to x by some constant of proportionality, and I'm not particularly interested in what the constant is. The other thing we see here is that the amount of space this takes up is constant, it's proportional to one. So the space complexity of this is order of one.

We have a name for such a process. Such a process is called an iteration. And what matters here is not that some particular machine I designed here and talked to you about and called a substitution machine or whatever-- substitution model-- managed to do this in constant space. What really matters is this tells us a bound. Any machine could do this in constant space. This algorithm represented by this procedure is executable in constant space.

Now, of course, the model is ignoring some things, standard sorts of things. Like numbers that are bigger take up more space and so on. But that's a level of abstraction at which I'm cutting off. How do you represent numbers? I'm considering every number to be the same size. And numbers grow slowly for the amount of space they take up and their size.

Now, this algorithm is different in its complexity. As we can see here, this algorithm has a time complexity which is also proportional to the input argument x. That's because if I were to add one to three, if I made a larger problem, which is larger by one here, then I'd add a line at the top and I'd add a line at the bottom. And the fact that it's a constant amount, like this is twice as many lines as that, is not interesting at the level of detail I'm talking about right now.

So this is a time complexity order of the input argument x. And space complexity, well, this is more interesting. I happen to have some overhead, which you see over here, which is constant approximately. Constant overhead. But then I have something which increases and decreases and is proportional to the input argument x. The input argument x is three. That's why there are three deferred increments sitting around here. See? So the space complexity here is also order x.

And this kind of process, named for the kind of process, this is a recursion. A linear recursion, I will call it, because of the fact that it's proportional to the input argument in both time and space. This could have been a linear iteration.

So then what's the essence of this matter? This matter isn't so obvious. Maybe there are other models by which we can describe the differences between iterative and recursive processes. Because this is hard now. Remember, we have-- those are both recursive definitions. What we're seeing there are both recursive definitions, definitions that refer to the thing being defined in the definition. But they lead to different shape processes. There's nothing special about the fact that the definition is recursive that leads to a recursive process.

OK. Let's think of another model. I'm going to talk to you about bureaucracy. Bureaucracy is sort of interesting. Here we see on a slide an iteration. An iteration is sort of a fun kind of process.

Imagine that there's a fellow called GJS-- that stands for me-- and he's got a problem: he wants to add together three and four. This fella here wants to add together three and four. Well, the way he's going to do it-- he's lazy-- is he's going to find somebody else to help him do it. They way he finds someone else to-- he finds someone else to help him do it and says, well, give me the answer to three and four and return the result to me. He makes a little piece of paper and says, here, here's a piece of paper-- you go ahead and solve this problem and give the result back to me.

And this guy, of course, is lazy, too. He doesn't want to see this piece of paper again. He says, oh, yes, produce a new problem, which is the sum of two ad five, and return the result back to GJS. I don't want to see it again. This guy does not want to see this piece of paper. And then this fellow makes a new problem, which is the addition of the sum of one and six, and he give it to this fella and says, produce that answer and returned it to GJS. And that produces a problem, which is to add together zero and seven, and give the result to GJS. This fella finally just says, oh, yeah, the answer is seven, and sends it back to GJS. That's what an iteration is.

By contrast, a recursion is a slightly different kind of process. This one involves more bureaucracy. It keeps more people busy. It keeps more people employed. Perhaps it's better for that reason.

But here it is: I want the answer to the problem three and four. So I make a piece of paper that says, give the result back to me. Give it to this fella. This fellow says, oh, yes, I will remember that I have to add later, and I want to get the answer the problem two plus four, give that one to Harry, and have the results sent back to me-- I'm Joe. When the answer comes back from Harry, which is a six, I will then do the increment and give that seven back to GJS. So there are more pieces of paper outstanding in the recursive process than the iteration.

There's another way to think about what an iteration is and the difference between an iteration and a recursion. You see, the question is, how much stuff is under the table? If I were to stop-- supposing I were to kill this computer right now, OK? And at this point I lose the state of affairs, well, I could continue the computation from this point but everything I need to continue the computation is in the valuables that were defined in the procedure

that the programmer wrote for me. An iteration is a system that has all of its state in explicit variables.

Whereas the recursion is not quite the same. If I were to lose this pile of junk over here, and all I was left with was the sum of one and four, that's not enough information to continue the process of computing out the seven from the original problem of adding together three of four. Besides the information that's in the variables of the formal parameters of the program, there is also information under the table belonging to the computer, which is what things have been deferred for later.

And, of course, there's a physical analogy to this, which is in differential equations, for example, when we talk about something like drawing a circle. Try to draw a circle, you make that out of a differential equation which says the change in my state as a function of my current state. So if my current state corresponds to particular values of y and x, then I can compute from them a derivative which says how the state must change.

And, in fact, you can see this was a circle because if I happen to be, say, at this place over here, at one, zero, for example, on this graph, then it means that the derivative of y is x, which we see over here. That's one, so I'm going up. And the derivative of x is minus y, which means I'm going backwards. I'm actually doing nothing at this point, then I start going backwards as y increases.

So that's how you make a circle. And the interesting thing to see is a little program that will draw a circle by this method. Actually, this won't draw a circle because it's a forward oil or integrator and will eventually spiral out and all that. But it'll draw a circle for a while before it starts spiraling.

However, what we see here is two state variables, x and y. And there's an iteration that says, in order to circle, given an x and y, what I want is to circle with the next values of x and y being the old value of x decrement by y times dt where dt is the time step and the old value of y being implemented by x times dt, giving me the new values of x and y.

So now you have a feeling for at least two different kinds of processes that can be evolved by almost the same program. And with a little bit of perturbation analysis like this, how you change a program a little bit and see how the process changes, that's how we get some intuition. Pretty soon we're going to use that intuition to build big, hairy, complicated systems.

Thank you.

[MUSIC PLAYING BY J.S. BACH]

PROFESSOR: Well, you've just seen a simple perturbational analysis of some programs. I took a program that was very similar to another program and looked at them both and saw how they evolved processes. I want to

show you some variety by showing you some other processes and shapes they may have. Again, we're going to take very simple things, programs that you wouldn't want to ever write. They would be probably the worst way of computing some of the things we're going to compute. But I'm just going to show you these things for the purpose of feeling out how to program represents itself as the rule for the evolution of a process.

So let's consider a fun thing, the Fibonacci numbers. You probably know about the Fibonacci numbers. Somebody, I can't remember who, was interested in the growth of piles of rabbits. And for some reason or other, the piles of rabbits tend to grow exponentially, as we know.

And we have a nice model for this process, is that we start with two numbers, zero and one. And then every number after this is the sum of the two previous. So we have here a one. Then the sum of these two is two. The sum of those two is three. The sum of those two is five. The sum of those two is eight. The sum of those two is 13. This is 21. 34. 55. Et cetera.

If we start numbering these numbers, say this is the zeroth one, the first one, the second one, the third one, the fourth one, et cetera. This is the 10th one, the 10th Fibonacci number. These numbers grow very fast. Just like rabbits. Why rabbits grow this way I'm not going to hazard a guess.

Now, I'm going to try to write for you the very simplest program that computes Fibonacci numbers. What I want is a program that, given an n, will produce for me Fibonacci event. OK? I'll write it right here. I want the Fibonacci of n, which means the-- this is the n, and this is Fibonacci of n. And here's the story. If n is less than two, then the result is n. Because that's what these are. That's how you start it up. Otherwise, the result is the sum of Fib of n minus one and the Fibonacci number, n minus two.

So this is a very simple, direct specification of the description of Fibonacci numbers that I gave you when I introduced those numbers. It represents the recurrence relation in the simplest possible way.

Now, how do we use such a thing? Let's draw this process. Let's figure out what this does. Let's consider something very simple by computing Fibonacci of four. To compute Fibonacci of four, what do I do?

Well, it says I have-- it's not less than two. Therefore it's the sum of two things. Well, in order to compute that I have to compute, then, Fibonacci of three and Fibonacci of two. In order to compute Fibonacci of three, I have to compute Fibonacci of two and Fibonacci of one. In order to compute Fibonacci of two, I have to compute Fibonacci of one and Fibonacci of zero. In order to compute Fibonacci of one, well, the answer is one. That's from the base case of this recursion. And in order to compute Fibonacci of one, well, that answer is zero, from the same base. And here is a one. And Fibonacci of two is really the sum of Fibonacci of one. And Fib of zero, in order to compute that, I get a one, and here I've got a zero.

I've built a tree. Now, we can observe some things about this tree. We can see why this is an extremely bad way to compute Fibonacci numbers. Because in order to compute Fibonacci of four, I had to compute Fibonacci of two's sub-tree twice.

In fact, in order way to add one more, supposing I want to do Fibonacci of five, what I really have to do then is compute Fibonacci of four plus Fibonacci of three. But Fibonacci of three's sub-tree has already been built. This is a prescription for a process that's exponential in time. To add one, I have to multiply by something because I take a proportion of the existing thing and add it to itself to add one more step. So this is a thing whose time complexity is order of-- actually, it turns out to be Fibonacci-- of n.

There's a thing that grows exactly at Fibonacci numbers. It's a horrible thing. You wouldn't want to do it. The reason why the time has to grow that way is because we're presuming in the model-- the substitution model that I gave you, which I'm not doing formally here, I sort of now spit it out in a simple way-- but presuming that everything is done sequentially. That every one of these nodes in this tree has to be examined. And so since the number of nodes in this tree grows exponentially, because I add a proportion of the existing nodes to the nodes I already have to add one, then I know I've got an exponential explosion here.

Now, let's see if we can think of how much space this takes up. Well, it's not so bad. It depends on how much we have to remember in order to continue this thing running. Well, that's not so hard. It says, gee, in order to know where I am in this tree, I have to have a path back to the root. In other words, in order to-- let's consider the path I would have to execute this.

I'd say, oh, yes, I'm going to go down here. I don't care which direction I go. I have to do this. I have to then do this. I have to traverse this tree in a sort of funny way. I'm going to walk this nice little path. I come back to here. Well, I've got to remember where I'm going to be next. I've got to keep that in mind. So I have to know what I've done. I have to know what's left. In order to compute Fibonacci of four, at some point I'm going to have to be down here. And I have to remember that I have to go back and then go back to here to do an addition. And then go back to here to do an addition to something I haven't touched yet.

The amount of space that takes up is the path, the longest path. How long it is. And that grows as n. So the space-- because that's the length of the deepest line through the tree-- the space is order of n. It's a pretty bad process.

Now, one thing I want to see from this is a feeling of what's going on here. Why are there-- how is this program related to this process? Well, what are we seeing here? There really are only two sorts of things this program does. This program consists of two rules, if you will. One rule that says Fibonacci of n is this sum that you see

over here, which is a node that's shaped like this. It says that I break up something into two parts. Under some condition over here that n is greater than two, then the node breaks up into two parts. Less than two. No. Greater than two. Yes.

The other possibility is that I have a reduction that looks like this. And that's this case. If it's less than two, the answer is n itself.

So what we're seeing here is that the process that got built locally at every place is an instance of this rule. Here's one instance of the rule. Here is another instance of the rule.

And the reason why people think of programming as being hard, of course, is because you're writing down a general rule, which is going to be used for lots of instances, that a particular instance-- it's going to control each particular instance for you. You've got to write down something that's a general in terms of variables, and you have to think of all the things that could possibly fit in those variables, and all those have to lead to the process you want to work. Locally, you have to break up your process into things that can be represented in terms of these very specific local rules.

Well, let's see. Fibonaccis are, of course, not much fun. Yes, they are. You get something called the golden ratio, and we may even see a lot of that some time.

Well, let's talk about another thing. There's a famous game called the Towers of Hanoi, because I want to teach you how to think about these recursively.

The problem is this one: I have a bunch of disks, I have a bunch of spikes, and it's rumored that somewhere in the Orient there is a 64-high tower, and the job of various monks or something is to move these spikes in some complicated pattern so eventually-- these disks-- so eventually I moved all of the disks from one spike to the other. And if it's 64 high, and it's going to take two to the 64th moves, then it's a long time. They claim that the universe ends when this is done.

Well, let's see. The way in which you would construct a recursive process is by wishful thinking. You have to believe.

So, the idea. Supposing I want to move this pile from here to here, from spike one to spike two, well, that's not so hard. See, supposing somehow, by some magic-- because I've got a simpler problem-- I move a three-high pile to here-- I can only move one disk at a time, so identifying how I did it. But supposing I could do that, well, then I could just pick up this disk and move it here. And now I have a simple problem. I have to move a three-high tower to here, which is no problem. So by two moves of a three high tower plus one move of a single object, I can move the tower from here to here.

Now, whether or not-- this is not obvious in any deep way that this works. And why? Now, why is it the case that I can presume, maybe, that I can move the three-high tower? Well, the answer is because I'm always counting down, and eventually I get down to zero-high tower, and a zero-high tower requires no moves.

So let's write the algorithm for that. Very easy. I'm going to label these towers with numbers, but it doesn't matter what they're labelled with. And the problem is to move an n-high tower from a spike called From to a spike called To with a particular spike called Spare. That's what we're going to do.

Using the algorithm I informally described to you, move of a n-high tower from From to To with a Spare. Well, I've got two cases, and this is a case analysis, just like it is in all the other things we've done.

If n is zero, then-- I'm going to put out some answers-- Done, we'll say. I don't know what that means. Because we'll never use that answer for anything. We're going to do these moves. Else. I'm going to do a move. Move a tower of height less than n, the decrement of n height. Now, I'm going to move it to the Spare tower. The whole idea now is to move this from here to here, to the Spare tower-- so from From to Spare-- using To as a spare tower.

Later, somewhere later, I'm going to move that same n-high tower, after I've done this. Going to move that same n minus one-high tower from the Spare tower to the To tower using the From tower as my spare. So the Spare tower to the To tower using the From as the spare.

All I have to do now is when I've gotten it in this condition, between these two moves of a whole tower-- I've got it into that condition-- now I just have to move one disk. So I'm going to say that some things are printing a move and I don't care how it works. From the To.

Now, you see the reason why I'm bringing this up at this moment is this is an almost identical program to this one in some sense. It's not computing the same mathematical quantity, it's not exactly the same tree, but it's going to produce a tree. The general way of making these moves is going to lead to an exponential tree.

Well, let's do this four-high. I have my little crib sheet here otherwise I get confused. Well, what I'm going to put in is the question of move a tower of height four from one to spike two using spike three as a spare. That's all I'm really going to do. You know, let's just do it. I'm not going to worry about writing out the traits of this. You can do that yourself because it's very simple.

I'm going to move disk one to disk three. And how do I get to move disk one to disk three? How do I know that? Well, I suppose I have to look at the trace a little bit. What am I doing here? Well, and this is not-- n is not zero. So I'm going to look down here. This is going to require doing two moves. I'm only going to look at the first one. It's

going to require moving-- why do I have move tower? It makes it harder for me to move. I'm going to move a three-high tower from the from place, which is four, to the spare, which is two, using three as my-- no, using from--

STUDENT: [INAUDIBLE PHRASE].

PROFESSOR: Yes. I'm sorry.

From two-- from one to three using two as my spare. That's right. And then there's another move over here afterwards. So now I say, oh, yes, that requires me moving a two-high tower from one to two using three as a spare. And so, are the same, and that's going to require me moving and one-high tower from one to three using two as a spare. Well, and then there's lots of other things to be done.

So I move my one-high tower from one to three using two as a spare, which I didn't do anything with. Well, this thing just proceeds very simply. I move this from one to two. And I move this disk from three to two. And I don't really want to do it, but I move from one to three. Then I move two to one. Then I move two to three. Then one to three. One to two. Three to two. Three to one. This all got worked out beforehand, of course. Two to one. Three to two. One to three.

STUDENT: [INAUDIBLE PHRASE].

PROFESSOR: Oh, one to three. Excuse me. Thank you.

One to two. And then three to two. Whew.

Now what I'd like you to think about, you just saw a recursive algorithm for doing this, and it takes exponential time, of course. Now, I don't know if there's any algorithm that doesn't take exponential time-- it has to. As I'm doing one operation-- I can only move one thing at a time-- there's no algorithm that's not going to take exponential time.

But can you write an iterative algorithm rather than a recursive algorithm for doing this? One of the sort of little things I like to think about. Can you write one that, in fact, doesn't break this problem into two sub-problems the way I described, but rather proceeds a step at a time using a more local rule? That might be fun.

Thank you so much for the third segment.

Are there questions?

STUDENT: [INAUDIBLE] a way to reduce a tree or recursion problem, how do you save the immediate work you

have done in computing the Fibonacci number?

PROFESSOR: Oh, well, in fact, one of the ways to do is what you just said. You said, I save the intermediate work. OK? Well, let me tell you-- this, again, we'll see later-- but suppose it's the case that anytime I compute anything, any one of these Fibonacci numbers, I remember the table that takes only linear time to look up the answer. Then if I ever see it again, instead of doing the expansional tree, I look it up. I've just transformed my problem into a problem that's much simpler.

Now, of course, there are the way to do this, as well. That one's called memoization, and you'll see it sometime later in this term. But I suppose there's a very simple linear time, and, in fact, iterative model for computing Fibonaccis, and that's another thing you should sit down and work out. That's important. It's important to see how to do this. I want you to practice.