PROFESSOR: I apologize to those of you watching on OpenCourseWare. I forgot to turn on my microphone. And you missed some incredibly brilliant things. But such is life. Let me go back to where I was, which was we we're looking at this code to find the cube root of a perfect cube. We saw this last week, and indeed, you also saw it in recitation. I'm not going to belabor it.

I do want to ask you the question, for what values will this program terminate? That is to say the only input is to x. For what values of x is this program guaranteed to always stop? Anybody want to volunteer an answer to that? Ask a simpler question. Let's assume that x is a number. In fact, let's assume it's an integer. Will it terminate for all positive integers, all positive values of x? Yeah. All negative values? As far as I can tell. How about 0? Yeah. So in fact, it terminates for all values of x.

How do I know that? And that's a key question. I know that because I've used, and I mean by used as a mental tool, something in my head, the notion of a decrementing function. And every time I write a loop, I think about one of these, because that explains to me why the loop is guaranteed to terminate. We'll go over here where we have a bigger board and look at the properties that a decrementing function needs to have.

One, it will map some set of program variables to an integer. Two, when the loop is entered for the first time or when I encountered the test of the loop for the first time, its value is non-negative. Three, when its value gets to be less than or equal to 0, the loop terminates. And finally four, it's decreased each time through the loop.

So what we see is if it starts to be a positive value or non-negative, and it's decreased every time I execute the body of a loop, that eventually, it's got to reach

0 or something less than 0. And when it does, the loop stops. If such a function exists, then the loop is guaranteed to always terminate.

Now, of course, one can count up to a value instead of down. But there's always a trick we can use of subtracting to make it the same. So what's the decrementing function for this loop? How did I know it will always terminate? Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Answer equals answer plus 1. I don't think so. Does that satisfy all of these properties? Remember, a function is going to map some set of program variables to an integer. So what are the interesting program variables here? Well, there are only two, ans and x. Right? At least, that's all I can see.

So what would be an interesting function? Somebody? Surely, there's someone who can figure this out. Yes? Or no, you're just scratching your head. You fooled me. I can't see because of the light, but I'm sure there must be dozens of hands up if I could only see them. Actually, I don't see any hands up.

This is not so hard, guys. It's the absolute value of x minus answer cubed. So what does this value start at? Let's pick a value. Suppose that x is equal to 8. What is the initial value of the decrementing function? Wow. Come on. Let's be a little cooperative, please. Yes?

AUDIENCE: So it's 8, answer is 0 and absolute value of x is 8.

PROFESSOR: So it's 8 minus 0, which is equal to 8. So it satisfies conditions one and conditions two. What happens to this value every time through the loop? Does x change? Does answer change? And how does it change? It's increasing. What does answer start at? It starts at 0, and it increases. So I know that answer cubed will always be positive. Right? So I know that every time through the loop, it will be 8. The first time through, it'll be 8 minus 1 cubed, which is 7. And then the next time through, it'll be 8 minus 2 cubed, which is 0. And then, I exit the loop.

And it's that kind of reasoning that I used to convince myself that this loop terminates. And every time I write a loop, and I hope every time you write a loop, you will think about what's the reason the loop is going to terminate. And you will do it by thinking about what the decrementing function is. People get me on that? And whoever finally answered a question surely deserves to be fed. I obviously have to bring better candy to encourage better responses.

Now, let's go back and look at the program itself. Now that we know it stops, and you can take my word for it that it actually computes the correct answer, let's think about what kind of algorithm this is. What's the method? This is an example of guess and check. And it's a particular kind called exhaustive enumeration.

Each time through the loop, I'm taking a guess at to what the value is, and I'm checking whether it's true. But I'm enumerating the guesses in a very systematic way. They're not just random guesses. I'm enumerating all the possible answers. If I get through the entire space of answers, or possible answers, and I don't find a solution, then I know that it doesn't exist, and it's not a perfect cube. So that's why it's called exhaustive enumeration because I'm exhausting the space of possible answers. Does that makes sense to everyone?

So let's try it. Let's try it with say a very large value of x, because that's always an issue, of course, when we do exhaustive enumeration. So I'm going to enter this value. Is that a perfect cube? Who thinks it is? Who can tell me what it is? What's the cube root of that? Well, this is a question I did not expect you to answer. But it's 1,251. That'll be in the first quiz, so make a note of it.

Notice how quickly the computer did this. It found the cube root of quite a large number very quickly. And so while one might initially think that exhaustive enumeration is a silly technique because takes a lot of guesses, for an awful lot of problems, we can actually just write a pretty stupid program that's solves it by exhaustive enumeration. We typically refer to such programs as brute force. And brute force is often exactly the right way to solve a problem.

Why does it work? Because computers are really fast. How fast are computers? Well, today, a good computer can execute in a single processor in the order of 100 million instructions a second. How fast is that? And now, we're going to see if Mitchell has answered the question I asked in the way in the class. How many instructions can a computer execute between the time I say something and the time the people in the back row hear it?

Mitch thinks it's 400 million instructions. I think that's about right. It's hundreds of millions at any rate. It's kind of amazing between the time I say something and the time you hear it, hundreds of millions of instructions. It's mind boggling how fast that is. And that's why we can often use these kind of solutions.

Next lecture, actually, even a little bit later in this lecture, I hope to get to an example of why that doesn't really get the job done, at least not always. Before I do that, I want to look at one more programming construct. And that's a variant on the while loop. So if we think about what the while loop we were just looking at did or does, as the decrementing function told us, it's looking at all the possible values of answer ranging from 0 to the absolute value of x. And at each step testing and doing something, we can abstract this process using something called a for loop.

So let's look at this code. It's essentially exactly the same algorithm. I got bored of typing ans times ans times ans. So I used a Python notation for exponentiation, which you'll see is star, star. Now, be easier to read if I get rid of that. But other than that, the interesting thing I did was replace the while loop by a for. So you'll see this line of code there, for ans in range 0 to abs of x plus 1.

What that says is range is a built-in function of Python that generates, in this case, a sequence of integers, something called a tuple, which we'll be looking at in a lecture or so. But for now, it's pretty simple to think about what you get is if you look at the expression range of x to y, that gives me a sequence of values, x, x plus 1 up to y minus 1. Notice not up to y, but up to y minus 1. So it gives me a sequence of length y-- well, assuming that's 0. Right? It doesn't have to be 0. It can be anything. It can be another value as well. 0 in my example.

And then, the for loop executes it on this value and the next iteration on this value, and finally, at the very end on that value. So it executes it one iteration of the loop on each value in this sequence of values generated by the range statement. And normally, it does all of them.

However, you'll see I've added something called a break here, a command in Python. And what break says is exit the loop. So it exits it prematurely without executing all of the values generated by range. You can nest loops just like you nest if statements. And if you do that break, always executes-- always exits rather the innermost loop.

So what this does is it's generates a set of values to test. It checks whether or not it's got the answer. If it does, it terminates the loop. And eventually, you exit the loop. And then, it just checks as before whether or not it found a correct answer and does the right thing.

So you'll find, particularly when you're iterating over integers, but later we'll see when you're iterating over a lot of other kinds of things, for loops are a very convenient shorthand. There's nothing you can't do with a while loop. You don't need for loops. But they do make life easy. And over the semester, I think you'll end up writing a lot more for loops than you will while loops. Any questions about this? If not, I'm going to move right along. So this is the moment. I'm going to move right along.

So we've now got a program that does something really silly, really. It finds cube roots of perfect cubes. Well, that's not typically useful. Right? You've even got these $0.50 four function calculators that find square roots. And they don't insist that you only give it perfect squares. So now, let's think about how we would take this kind of program, and indeed, this kind of method of writing programs and use it to find-- for now, we'll look at the square root of any number, of any floating point number.

Well, the first question we need to ask is what do I mean? That's kind of a subtle question. What does it mean to find the square root of a number? What does it mean, for example, to find the square root of 2? Well, we know that that was an endless series of digits before we can find the square root of 2. Right? It does not have a nice answer.

So we can't just say we have to find something that if we multiply it by itself, it will equal 2. Because we can't find such a thing. So we've got to think about a different notion of what we mean. Furthermore, even for some numbers which there is a square root, it might be a million decimal places long, and consequently, really hard to find. So we need to think about a different kind of concept here. And it's the concept of an approximation, finding an answer that is good enough.

So what should we do here? How do we think about this? Typically, what we do when we think about an approximation is we define how good an approximation we're willing to accept. So for example, we might want to say, I want to find a square root that lies within epsilon of the true square root. So find a y such that y times y is equal to what? What does it mean? How would I express it within epsilon of the perfect answer? I don't want to say it's equal to x, because that may be impossible or too time consuming to find.

So really, what I mean is x plus or minus epsilon. So that's what I'm asking. Find one that's close enough. And that's what the next piece of code I want to show you does. Excuse me. So I'm starting, just giving it a value for x, so I don't have to keep typing one in. Let's say it's 25. I'm going to take epsilon to be 0.01. So I want it within that distance of the true answer. I'm going to keep track of the number of guesses here, not because we need it to actually compute the answer, but because I want to then discuss how many iterations of the loop we're doing.

We're going to start by setting my first guess at 0.0. Again, this is going to be exhaustive enumeration. Then, I'm going to essentially encode this as a test of my while loop while the absolute value of answer squared minus x is greater than or equal to epsilon, and answer is less than equal to x. So it's now a more complicated test. I've got a Boolean value. Two things have to be true to execute the body of the loop. I'm going to increment answer by a tiny amount, increment the number of guesses just so I can keep track of it. Maybe I'm going to comment this out for the first go around just so we don't see too many print statements and keep doing it.

And then when I'm done I'm going to see whether or not what I found is indeed a square root or close enough. So if we think about why this loop terminates, why am I guaranteed that this loop will terminate? What's my decrementing function here? Somebody? What's the decrementing function? What am I guaranteed to reduce each time through, and when I get through, I'm done? Yeah?

AUDIENCE: [INAUDIBLE] answer squared minus x1 times 1.

PROFESSOR: No. Close, sort of. But I appreciate you're trying. That's worth something just for the effort. Somebody else. Remember, if we look at the properties it has to have, it's going to guarantee me that when it gets to the right value, I exit the loop, which suggests it's going to certainly be part of the test of the while.

Just look at this piece over here at the end. Answer starts at 0. I keep incrementing it. Eventually, answer minus x will hit a value, right? Eventually, I'll get to the point that this condition must be true-- must be false rather. And then, I exit the loop. So this piece is not really the key. It's this piece that guarantees me I'm going to get out eventually. This piece can get me out sooner. It's kind of an optimization, if you will.

So I'm just going to go until I find the answer. Let's see what happens when I run it. It tells me that 4.99, et cetera is close to the square root of 25. So there are some things to note about this. First, it didn't find 5, 25 happens to be a perfect square, yet I didn't find it. Is that OK? Yeah. Because that wasn't what I said. What I said is find a y that has these properties over here. And I did. I didn't say find the y that gets closest to the square root of x. I said find one that has these properties.

Effectively, what this is is a specification of the problem. And I've now written a piece of code that meets the specification. It does what I set out to do, and that's good enough. Now, let's turn this print statement back on. It took almost 1/2 million guesses to get there. But it was still blindingly fast. So once again, exhaustive enumeration seems to be OK.

Suppose, however, I choose a bigger number. Now, first, let's choose something that doesn't have a good answer. Let's see what it does for that. All right. Pretty good. Also pretty fast. Not too many guesses. But now, let's try this one. Well, it's going to wait. It's going to get done, but it's going to take a little bit longer than maybe we want it to take. Why is it taking so long? There it is. It found an answer, which I think is good.

But as you can see, it took quite a few guesses to get there. So why? Well, let me first ask this question. Can we look at the code and anticipate how many guesses it's going to have to take? We're going back to this issue of computational complexity, but here, not of the problem but of the solution. So this is algorithmic analysis. We're analyzing the algorithm, this exhaustive enumeration algorithm, and trying to figure out how long it's likely to take to run. Well, what does the running time of this algorithm depend upon? Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: It depends on the actual square root. Yes. But in particular, the distance of the actual square root from the starting point. So that's one factor it depends on. But that's not the only factor. What else does it depend on? Oh, do we have an injury? We had a dropped pass and a deflection there. All right. Yes?

AUDIENCE: It depends on the level of accuracy, so how you define epsilon.

PROFESSOR: It depends upon the value of epsilon. Absolutely. How long it takes to run.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Someone with a concern for safety. it depends upon the actual value of epsilon, because if epsilon is small, we may have to take more steps to get a precise enough answer. And it depends upon one more thing. Yeah?

AUDIENCE: [INAUDIBLE] the increment that [INAUDIBLE]?

PROFESSOR: The increment. Exactly. Because the number of times we go through the loop is going to be related to how big a step we take each time through. No applause? Thank you. So it depends upon all of these things. And here, since we're trying to find a pretty [? big ?] square root and a sort of precise answer, but we're taking tiny steps, it's going to take a long time to execute. So we could make it faster.

For example, suppose I change the step size to this. Plus equal by says increment the value by whatever the right side is. So I'm going to increment it by 1. Wow, it was really fast. But it didn't work. It failed. That's not so good. So I can't just do that. And of course, it's not surprising, because I ended up jumping all over the answer. I could make epsilon smaller, but that seems like cheating.

So really, what I need to do is find a better algorithm, a better way to attack the problem. Fortunately, I don't have to invent it. Some people a lot smarter than I am figured out a long time ago a good method for solving this kind of problem. And they're doing it using something called bisection search.

As we look at this particular implementation of it, we're going to use two algorithmic techniques that you'll use over and over again because they're generally useful. So the first one related to bisection search is we'll cut the search space in half each iteration. So with my brute force algorithm, we're trimming the search base only a little bit each step.

So if we think about it, what it looks like, we had a space of values to search for the answer. And I started here. And each step, I just trimmed off a tiny, tiny little bit, 0.001, leaving me a lot of space to search. And that's why it took so long. When I do bisection search, the basic idea is each step I want to cut the search space in half. Get rid of half of the search space each time.

So one way I could do it is I start with a guess say in the middle. Just pick some guess that's in the middle of my search space. And now I say is it too high or too low? I can easily answer that question. I square it. See is my result bigger than the actual square root or smaller than the actual square root? That tells me whether my guess is too big or too small. Once I know that, I know which side of the guess the right answer is on.

So if I knew that my guess was too big, then I know there's no point in looking over here for my next guess. So I can get rid of this whole half in one step. Now, what should my next guess be? Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: My next guess should be half way through there. Exactly. And now, let's say this time my answer is too small. Then I know I can get rid of this. So now, I'm very quickly pruning my search space. If I think about that, how many times am I likely to have to prune it? It's much faster, right? As we'll see later, it's basically log base 2.

If I have some number of values to look at-- and by the way, how many values do I have in my search space to start with? What determines it? Clearly, the first value and the last value, but also, how small I'm dividing it up. Right? So I have to think about that too. What is the precision with which I do this. Am I looking at every one millionth of a number or every 0.01? That will tell me how big it is.

Once I know how big my search space is, I know that if I search it linearly looking at every value, my worst cases, I look at everything until I get to the end. Well, my best case is I get lucky, and my first guess is right. But if I use bisection search, my worst case is going to be log number of values in that space. Because each time, I throw half of them away. We'll see that in more detail later on.

Let's go back and look at the code now. So it starts as before with a value for epsilon. But now, I'm going to take a lower bound, here, and an upper bound on my search space. I'm going to say my initial guess will be the upper bound plus the lower bound over 2 halfway through my search space. And then, I'm just going to work my way through it until I get to the answer or don't find it.

So should we look at what's going on here? Let's try this. Well, let's first make sure it works for a small value. Never test your program first on something big. Always try your program in something small first. Let's try it on that. Got an answer. Notice that it's different from the answer we got last time we looked for the square root of 25. But that's OK, because it's still meets the specification. It's still within epsilon of the actual square root. And I didn't have to say that I wanted it below or the square root or above, just said within epsilon. Sure enough, different algorithm, different answer, but equally good, but a lot faster.

Now let's try it for the big value. Wow, that was a lot faster, wasn't it? It got me an answer. Probably, not exactly the same answer as before but pretty close. But it did it in only 26 guesses. Pretty cool. And in fact, we'll see over and over again that bisection search is a really good technique for finding quick answers.

And again, why is it 26? Well, we had some number of guesses to start with. After 1, it was half as big, then a quarter is big, and eventually, log 2 of the size. But what was the size? Was it the number 12345? No. We already sort of talked about that. What was it? Let's look at the code, and let's think about what was the size of our initial search space. It's a little bit tricky to think about this, right?

Now, we have to think a little bit harder about when we exit the loop, because in fundamentally, that's telling me the size of the search space. So what determined the size of the search space? Well, we talked about the upper and the lower bound. But what's telling me roughly speaking how many divisions I have? It's epsilon. It's not 0.01 because when I square it, it has to be smaller than 0.01.

But that tells me roughly how many I have. And so it's going to be roughly 12345 divided by 0.01 squared, which turns out to be 26.897 more or less. So we could predict it. And son of a gun, when we ran it, we actually matched the prediction. That's the great thing about algorithmic analysis. We can actually get accurate guesses as to how long a program is likely to take to run.

This is an important thing because sometimes we do that and we say, oh, it's going to take a year. I better not even try. I better find a smarter algorithm. Or we do it and say, well, it's going to take almost no time at all. I'm not going to waste my time looking for a smarter algorithm. I'm going to live with the one I've got. It's important, and again, as I said, it's a topic we're going to get back to.

Of course, whether it's 26, 27, or even 50 doesn't really matter. What matters is it's not a billion. Right? Because we don't really care small differences. Whether it takes 25 or it takes 50 will be an imperceptible difference. It's whether it's a huge difference that matters. And that's really the kind of things we're going after is orders of magnitude.

Now, I have a question about this program. I've been obsessing about whether it's fast enough. And we've shown it is. But does it work? Kind of more important. It's always possible to write a really fast program that gives you the wrong answer. The problem is to write a fast program that give you the right answer.

Does this program always work? Well, it worked for 25. It worked for 12345. Is that good enough? Probably not. We might want to try it in some other values. I'll ask a simpler question. Does it always work on positive values? All right. I'll give you a hint. No. It does not. I'm not going to, however, tell you why it doesn't, because I want you to think about it. And I want you to tell me why it doesn't in the next lecture. But because I'm not a complete sadist, I'll give you a hint.

When we use bisection search, or for that matter, any search method, we are depending upon the fact that the answer lies somewhere in the region we're searching. If indeed the answer is out here or out here, then it doesn't matter how carefully I search this region. I'm not going to find the answer. And so this program doesn't work on some potential values of x because the actual square root of x will not lie in the region that the program is searching. I leave it to you to think about what such values are. And we can talk about that on the next lecture.

Suppose I want to use this program to find the cube root. Suppose it worked, and I want it to use it to find a cube root. What would I have to change? How would I change it, so it found cube roots instead of square roots? Well, I can take it up. I could use cut and paste, and paste it into my editor and get a new program.

And how would I change that new program to make it do cube roots? Not very hard. Think only two places have to get changed. That's for the simplicity, say cube roots of positive numbers. I think you said the right thing. All I have to do is change that two to a three and that two to a three, and I'm done. And I should probably change the message to say cube root. Pretty easy.

On the other hand, suppose I also want it to find the fourth root, and the fifth root, and the sixth root, however many roots. Well, I'm going to get pretty tired of cutting, and pasting, and building a whole bunch of things. So really, what I want to do is find a way to write the code that will find the nth root of a number for any n. To do that, I'm going to introduce a new programming concept. And that concept is the function. And that will be the main topic of Thursday's lecture.