ANA BELL: All right everyone, let's get started. So good afternoon. So this is the 3rd lecture of 6.0001 and 600. As always, please download slides and code to follow along.

So a quick recap of what we did last time. Last time, we talked about strings as a new object type, as sequences of characters. And then we introduced two new concepts that allowed us to write slightly more complicated programs.

So we introduced branching, with these keywords, if, elif, else. And branching allowed us to write programs that, us, as programmers, could introduce decisions into our programs. And then we introduced two different kinds of loops, while loops and for loops. And those also added a little bit of complexity to our programs.

Today, we're going to talk a little bit more about strings. So we're going to see a couple of more operations that you can do on strings and string objects. And then we're going to talk about three different algorithms, a guess and check algorithm, an approximate solution algorithm, and a bisection method algorithm.

So let's dive right in. We'll talk a little bit about strings, first. So strings, we thought of them as sequences of characters, case sensitive, as we saw in programs we wrote last lecture. And strings are objects. And we can do all of these operations on string objects, like test if they're equal, less than, greater than, and so on.

It turns out, we can do more than just concatenate two strings together or do these little tests on them. So we're going to start introducing the idea of a function or a procedure. And we're going to see more about functions and how you can write your own functions next lecture.

But for today, you can think of a function as sort of a procedure that does something for you. Someone already wrote this. So the first one we're going to look at is a pretty popular function.

And when applied on a string, this function, called len, will tell you the length of a string. So that's going to tell you how many characters are in the string. And characters are going to be

letters, digits, special characters, spaces, and so on. So it's just going to count how many characters are in a string.

So if I have the string s is equal to "abc"-- remember a string is in quotation marks-- then, if I do this, if I write this expression, len s, here, since it's an expression, it has a value. So it evaluates to a certain value. And by definition, it's going to tell me what the length of the string, which is 3 characters long.

Another thing that we can do on strings is, since they're a sequence of characters, I might want to get what character is at a certain position. So we do this using this fancy word called indexing. But pretty much what indexing into a string means is you're going to tell Python, I want to know the character, at this certain position or at this certain index, inside my sting.

So once again, let's use this string, s is equal to "abc." And let's index into it. So in computer science, we start from 0, counting by convention. Notice, we had a problem set 0 in this class. Python is no different.

So in Python, you start indexing at position 0. Or you start indexing at 0. So the first character, in your string, we say is at position 0 or at index 0. The next character in the string is at index 1. And the next character in the string is at index 2.

In Python, it turns out, you can also use negative numbers to index. And if you index into the string with negative 1, for example, that means that you want the last character in the string. So the last character in your string is always going to be at position negative 1, the second-to-last character is at negative 2, third-to-last character is at negative 3, and so on and so on.

So the way you index into a string is with these square brackets, here. And this is the notation. So if I want the character at position 0 or at index 0, I say s, which is the string I want to index into. And then, inside the square brackets, I say what index I want.

So s at index 0 is going to be the value "a." s at index 1 is going to be the value "b," and so on and so on. And we can also do negative indexing, as well.

I added this in here. If you do try to index into a string beyond the limits of the string-- and we can even try this out, just to show you that it's not the end of the world if we do that. If we have s is equal to "abc," and we have s at position 20, for example, obviously, my string is only length 3, so what's at position 20?

I get an error. I call this angry text, here, in Python. But really, the most relevant thing to note is these last couple of lines here. This tells you what line is problematic. So s at position 20 has an issue. And this last line here tells me what actual error I have. So it's an index error, which means I'm trying to index too far into the string, because it only has three characters.

So it's nice to be able to get a single character out of my string. But sometimes, I might want to get a substring. So I want to start at the first character and go halfway into the string, or I want to take a few characters in between, or I want to skip every other letter or something like that in my string.

So if I want to do this slightly more complicated interaction with strings, we call that slicing, slicing into a string. And this notation here should seem a little bit familiar, because we saw it last lecture when we did it with range.

We had a start, stop, and a step. The notation was a little bit different, because, in range, we had open-close parentheses and commas in between. But except for that, this sort of works the same.

The start is the index, starting from 0, from where you want to slice into this string. The stop is the stop index. So you're going to go up until stop minus 1 and take that index. And then the step is how many letters you wish to take.

So this is the full notation here. But sometimes, you can not give it a third sort of number in here. So if you only give it two numbers, then, to Python, that represents just the start and the stop. And by default, step is going to be 1.

And there's a lot of other things you can do with strings. You can omit numbers and just leave colons in Python. By definition, the way that whoever wrote slicing had decided, if you omit numbers, then it's going to be equivalent to these things here.

So we slice using square brackets, just like indexing. Except now, we can give it two numbers. So with this string, s, if we slice into the string s, we start from index 3 and go up until index 6. So if we have abcdefgh, this is position 0, 1, 2, 3, 4, 5, 6, 7.

And you just count. So s, starting from 3 and going till 6, is going to start here, 3. So it's going to come up with-- sorry d. And then we're going to take e. And then we're going to take f. And since we're going until stop minus 1, we're not going to take g. Because this is position 6, and

we're going until 6 minus 1.

The next one here, 3, 6, 2 is going every other one. So we start at 3, and then we skip every other one, so we go d but not e, and then f, and then stop.

If you do s and then nothing inside except colons, notice that you're going to have s, and then nothing, and then colon, nothing, colon, nothing. So nothing for start, nothing for stop, nothing for step. And that's just going to value it to the string, itself. It's the same as 0 to the length s going every step.

This one might actually be useful. It reverses the string automatically for you. So with this one little line here, you can get the inverse of your string. And that's equivalent to that. So the minus 1 represents starting from the end and going back every letter. And then this one's a little bit more complicated but also not too bad.

So as we're doing these string slices, again, if you're unsure what something does, just type it into Spider. And you might be surprised. You might not be. But it's a good way to check yourself, to make sure you're understanding what's happening.

One thing I want to mention, and it's good to keep this in the back of your mind. We're going to come back to this as we start talking about slightly more complicated object types. But strings are immutable. So just keep this word in the back of your mind as we go through this class.

And what I mean by this is that an actual string object, once it's created, cannot be modified. This might not mean anything right now. But let me just draw a little something. Let's say I have this string, s is equal to hello.

Remember, in the first lecture, we drew a diagram sort of like this. This is my memory. I have this object "hello." And this object, "hello" is bound to this variable s. So now I can access the object "hello" using this variable s.

Now you might think, well, since I could index into a string, I might be able to just say something like, s at position 0 is equal to y. And that will just change the little h into a y, and I'll have a new object.

Well strings are immutable, which means, in Python, you're not actually allowed to do this. And it gives you an error if you do try to do that. If you want the variable s to point to the string, Y-E-L-L-O, you could just say s is equal to Y-E-L-L-O.

Or you could do string operations like this. And this takes the y and it concatenates it to the string s, all of the elements starting from position 1, which is e, l, l, o. So this makes Y-E-L-L-O.

Now internally, what happens when I write this line is Python says, OK, I'm going to break my bond with this original object "hello." I'm going to bind my string variable s to the new object "yello." and this other, old object still is in memory somewhere. But it's an entirely different object that I've created here.

Again, it might not mean anything right now, but just keep this in the back of your mind, strings are immutable.

So the next thing I want to talk about is a little bit of recap on for loops. And we're going to see how we can apply for loops, very easily, to write very nice, readable code when dealing with strings.

So remember that for loops had a loop variable. My loop variable being this var, here, in this particular case. It can be anything you want. And this variable, in this particular case, iterates over this sequence of numbers, 0, 1, 2, 3, 4.

So the very first time through the loop, var has a value of 0. It does the expressions in the loop. As soon as they're done, var takes the value 1. It does all the expressions in the loop. And then var takes the value 2, and it does that all the way up until 0, 1, 2. And the last time it goes around is with var is equal to 3.

And remember, we said that we can customize our range in order to start from a custom value to end at a different value and to skip certain numbers.

So, so far, we've only been using for loops over a sequence of numbers. But actually, for loops are a lot more powerful than that. You can use them to iterate over any sequence of values not just numbers but also strings.

So here are two pieces of code, this one and this one here. These two pieces of code both do the exact same thing. To me, possibly to you, this one looks a lot more readable than this one, just at a first glance.

If I were to read this one, just using the keywords and variables here, it would sound like broken English. But you could decipher what I'm trying to say. For a char in a string s, if the char is equal to "i" or a char is equal to "u," print "There is an i or a u."

It sounds weird, but you could probably tell what I was trying to do here. Whereas up here, it's a little more complicated to tell what I'm doing. You have to sort of think about it a little bit.

For some index in this range of numbers, 0 through the length of the string s, if s, at position index, is an "i" or s at position index is a "u" print, "There is an i or a u." Both of these codes just go through the string s. And if it encounters a letter that's an i or a u, it's just going to print out this string here.

But this bottom one is a lot more pythonic. It's an actual word created by the Python community. And it just looks pretty, right? You can tell what this code's supposed to do. Whereas this one is a little bit harder to decipher.

So that's sort of an illustration of a for loop over a sequence of characters. So char is going to be a loop variable, still. And the loop variable, instead of iterating over a set of numbers, it's going to iterate over every character in s, directly. And char is going to be a character. It's going to be a letter.

So here's a more complicated example. I wrote this code a couple of years ago. And it was my attempt at creating robot cheerleaders , because I needed some motivation. And then I googled, last night, "robot cheerleaders," and was not disappointed. Created this GIF. It looks pretty cool. And it looks like they kind of stole my idea. But that's fine.

So let's look at what this code's supposed to do. I'm going to run it. I'm going to run it, and then we'll go through it. All right, it prints out, "I will cheer for you! Enter a word."

You know what, I like robots, so I'll put in "ROBOTS." How enthusiastic am I about robots? Let's say 6. So what this is going to print is-- it's a cheerleader, right? "Give me an r, r." "Give me an o, o." "Give me a b, b," and so on and so on.

"What does that spell? ROBOTS." And it's going to print it 6 times, because I'm 6 out of 10 enthusiastic about robots. So that's pretty much what that code's supposed to do. And you can write it using what we've learned so far.

Now let's go through it a little bit. And I'm going to show you just how easy it is to convert this code using a for loop over characters. Right now, what it does is it asks the user for input, so a word and a number.

And then it does this thing, here, right? First, it uses a while loop. And second, it uses indexing. And what tips you off that it's using indexing is it's using the square bracket, here, into the word.

And obviously, it's using a while loop. And it has to first create a counter, initialize it. And then, down here, it's going to increment it inside the while loop. If you remember, that's sort of what we need to do for while loops.

So it's going to start at 0, and it's just basically going to go through index i is equal to 0, 1, 2, 3 4, which is going to go all the way to the end of the word, whatever the user typed in, in this case "ROBOTS." It's going to get the character at that position. word at position i is going to be a character.

This line here is just for the cheerleading to make sense. It's just to take care of letters that make sense to use an, right? So give me a b, give me an b. So give me an b does not make sense, right? So that's just taking care of that.

And I'm using this in keyword to check whether the character-- so the character, r, for example, in robots-- is inside an letters. And an letters I've defined up here, which is these are all the letters that make sense to put an an before the letter. So give me an r for example, here, on the right.

And so if it makes sense to use an before the letter, use that, and otherwise use just an a. And after I'm done, I say, "What does that spell?" And then it's just a for loop that goes times many times and prints out the word and the exclamation mark.

So this code might have been a little bit more intuitive if I rewrote it or if I'd originally written it with a for loop. So this part here, the while loop and indexing and creating my original counter, we can get rid of that.

And we can replace it with this, for char in word. I'm originally using char, so I can use char as my loop variable again. And simply, I'm just going to iterate over the word, itself.

So now, instead of having this mess here, I have a one-liner that says, for every character in my word, do all this stuff here. So that remains the same. And then I don't even need to increment a counter variable, because I'm not using while loops anymore. I'm just using a for loop.

So the code becomes-- delete that-- for char in word. And then delete that. And that does the exact same thing. And it's a lot more readable.

So this was our toolbox at the beginning of this course. We are two and half, I guess, lectures in. These are the things we've added to it. We know integer, floats, Booleans. We know a bit of string manipulation, math operations. We added, recently, these conditionals and branching to write slightly more interesting programs.

And now we have loops, for and while loops to add interesting and more complicated programs. So with these, the second part of this lecture is going to be looking at three different algorithms. That's the sort of computer science part of this class, Introduction to Computer Science and Programming using Python.

Don't let the word algorithm scare you. They're not that complicated. You just have to sort of think a little bit about them. And you'll be able to get them.

So we're going to look at three algorithms, all in the context of solving one problem, which is finding the cube root. The first algorithm is guess and check, then we're going to look at an approximation algorithm, and then a bisection search.

So the first is the guess and check method. You might have done this, in math, in high school. The guess and check method is also sometimes called exhaustive enumeration. And you'll see why.

So given a problem, let's say, find the cube root of a number, let's say you can guess a starting value for a solution. The guess and check method works if you're able to check if your solution is correct.

So if your guess is originally 0, you can say, is 0 cubed equal to the cube of whatever I'm trying to find the cube root of? So if I'm trying to find the cube root of 8, is 0 cubed equal to 8? No. So the solution is not correct.

If it's not correct, guess another value. Do it systematically until you find a solution or you've guessed all the possible values, you've exhausted all of your search space.

So here's a very simple guess and check code that finds the cube root of a number. So I'm trying to find the cube root of 8. So my cube is 8. I'm going to have a for loop that says, I'm going to start from 0. And I'm going to go all the way up to--

So I'm going to start from 0 and go all the way up to 8. For every one of these numbers, I'm going to say, is my guess to the power of 3 equal to the cube 8? And if it is, I'm going to print out this message.

Pretty simple, however, this code is not very user friendly, right? If the user wants to find the cube root of 9, they're not going to get any output, because we never print anything in the case of the guess not being a perfect cube. or the cube not being a perfect cube.

So we can modify the code a little bit to add two extra features. The first is we're going to be able to deal with negative cubes, which is kind of cool.

And the second is we're going to tell the user, if the cube is not a perfect cube, hey, this cube is not a perfect cube. So we're not going to silently just fail, because then the user has some sort of feedback on their input.

So let's step through this code. We have, first of all, a for loop just like before. And we're going to go through 0 to 8 in this case. We're using the absolute value, because we might want to find the cube root of negative numbers.

First thing we're doing is doing this check here. Instead of guessing whether the guess to the power of 3 is equal to the cube, we're going to check if it's greater or equal to, and we're going to do that for the following reason.

So if we're trying to find the cube root of 8, for example, versus a cube root of 9-- this is 8 and this is 9-- what is this code going to do? It's going to first guess 0. 0 cubed is not greater or equal to 8. 1 cubed is not greater or equal to 8.

2 cubed is greater or equal to 8, so here, once we've guessed 2, we're going to break. Because we found a number that works. And there's no need to keep looking. Once we've found the cubed root of this number 8, there's no need to keep searching the remainder, 3, 4, 5, 6, 7, 8.

Sort of the same idea when we're trying to find the cube root of 9. We're going to start with 0. 0 to the power of 3 is less than 9. 1 to the power of 3 is less 9. 2 to the power of 3 is less than 9.

When we get to 3 to the power of 3, that's going to be greater than 9. So this code tells us, once we've picked a number that's beyond the reasonable number of our cubed root, of our

cube, the cubed root of our cube, then we should stop.

Because, again, it doesn't make sense to keep searching. Because if 3 to the power of 3 is already greater than 9, 4 to the power of 3 is also going to be greater than 9 and so on. So once we break here, we either have guess being 2 or guess being 3 depending on what cube we're trying to find.

And if the guess to the power or 3 is not equal to the cube, then, obviously, the cube was not a perfect cube. So that's this case here, if we were looking at at the cube root of 9. And otherwise, this part here just looks at whether we should make it a positive or a negative cube. So if our original cube was less than 0, then, obviously, the cube root of a negative number is going to be a negative number, and, otherwise, it's just our guess.

So that's the guess and check method, slightly more feature-rich program for guessing the cube root. But that only tells us the cube root of perfect cubes and doesn't really give us anything else, any more information.

So sometimes, you might want to say, well, I don't care that 9 is not a perfect cube, just give me a close enough answer. So that's where approximate solutions come in. So this is where we're OK with having a good enough solution.

So in order to do that, we're going to start with a guess and then increment that guess by some small value. Start from 0 and start incrementing by 0.001 and just go upwards from there. And at some point, you might find a good enough solution.

In this program, we're going to keep guessing as long as we're not close enough. And close enough is going to be given by this epsilon value in the program. So as long as the guess cubed minus the cube-- so how far away are we from the actual answer-- is greater than some epsilon, keep guessing, because the solution is not good enough.

But once this is less than epsilon, then we've reached a good enough solution. So two things to note with approximate solutions. So you can get more accurate answers if your step size is really, really small. If you're incrementing by 0.0001, you're going to get a really good approximate solution, but your program will be a lot slower.

Same sort of idea with epsilon, you can change epsilon. If you change epsilon to be a bigger epsilon, you're sacrificing accuracy, but you're going to reach a solution a lot faster.

So here's the code for the approximate solution of a cube root. It might look intimidating, but, look, almost half this code is just initializing variables. So we're initializing, this is the cube we want to find the cube root of. We pick an epsilon of this. We start with a guess of 0. We start with an increment of 0.0001. And just for fun, let's keep track of the number of guesses that it takes us to get to the answer.

This is similar to the guess and check from before. It's not similar. Well this part is similar to the guess and check from before. So we're going to take the guess to the power of 3 minus the cube, right? So that's how far away are we from the actual answer?

And we're going to say, if that's not good enough-- so if we're still greater than or equal to the epsilon-- then keep guessing. So we're going to be stuck in this loop, where we keep guessing values, until we've reached a guess that's good enough, so until we're less than epsilon.

And way we keep guessing is just with this line, right here, which says, increment my guess by increment, and increment being this really small value. That make sense?

So I'm going to keep incrementing my guess by that small value. Before I go on, I'm going to run the code. And we're going to discover a small issue with it.

So with 27, we're going to run it. Perfect, it took me 300 guesses. But 2.99999 is close to the cube root of 27. We can find the cube root of this guy here. And it took me 20,000 guesses, but I figured out that 200.99999, so 201, is close to the cube root of that large number.

I should have done this. This is going to be a giveaway, you guys. Sorry. Then we're going to have-- let's say I want to try cube of 10,000. So 10,000 is not a perfect cube. So we can run the code. And with 8,120,601 I had already gotten an answer. But with 10,000, I'm not getting an answer yet, right?

So I'm thinking that there might be something wrong. So I'm going to stop my code. So I just hit Control C, because I feel like I've entered an infinite loop. And, in fact, I have. So what ended up happening is this problem here.

So I'm going to draw something. According to the code, I'm going to start from 0, and I'm going to increment my guesses, like that. With every little increment, I'm going to make a new guess. I'm going to take that guess to the power of 3. I'm going to subtract the cube, and I'm going to figure out if I'm less than epsilon.

This is the epsilon that I want to be in, this little bit here. So with every new guess, I might be, maybe-- so this is where I want to be, within this little boundary here. With every new guess, I might be here.

With the next guess, over here, I might be here. When I make another guess, I might be here. So I'm getting close to being within epsilon. But maybe with my next guess, I'm going to hop over my epsilon and have made too big of a guess.

So just because of the way the numbers were chosen in this example, just to illustrate this, using an increment of 0.01, a with finding the cube of 10,000 and epsilon of 0.1, it turns out that, as I'm doing all these calculations, I'm going to skip over this perfect sort of epsilon difference.

So first, I'm going to be too small. And then I'm going to be too large. And once I've become too large or too far away from epsilon, the guesses I continue to make are just going to be even farther away from epsilon. And I'm not going to get to my answer.

And that's why I've reached an infinite loop in this code. All I'm doing in this code is checking whether my guess cube minus cube is less than epsilon. The only thing I need to do here is sort of add this little clause, here, that says, oh, by the way, also check that I'm less than cube.

Because this is just like we did in the very first program, when I'm checking 0, 1, 2, 3, 4, 5, 6, 7, 8, when I'm trying to find the cube root of 8. Once I've reached 8, I'm going to stop. And it's the same thing here.

So I just added this little clause that says, well, while I'm greater than or equal to epsilon and I'm still less than the actual cube, just keep searching. But once I've reached the cube, then stop searching.

And with 10,000, you can see that I failed to actually find-- so that's what this part, here, does. It tells me I've failed to find the cube root with those particular parameters.

The last thing we're going to look at is bisection search. And to illustrate this, I'm going to need one volunteer. And you're going to play a game with me in front of the whole class. And there will be a prize. There go the hands. In the blue shirt, right there. Cool.

So the prize is going to be, once again, this. I promise I don't have millions of these, Google glasses. I also don't work for Google. I just happened to get a couple.

So the game is this. I'm going to ask you to pick a number, a whole number, between 0 and 100. And I'm going to try to guess it. And you need to make it hard for me. And you need to make it so hard for me that I cannot guess it within 10 guesses.

And if you can do that, if I cannot guess it within 10 guesses, you get this. And I'm going to draw out what I do as we go along. So do you have your number? Yes?

AUDIENCE: Yeah.

ANA BELL: Perfect. Let me erase that. Actually, I should've probably kept that, because I'll still use it. There's the numbers, 0 to 100. Is your number 50?

AUDIENCE: No.

ANA BELL: 50 Was my guess. So I've made one guess. Is your number higher or lower than 50?

AUDIENCE: Higher.

ANA BELL: Higher. Is your number-- my next guess is going to be 75. And the reason I'm guessing 75 is because-- what's your name?

AUDIENCE: Sophie.

ANA BELL: What's that?

AUDIENCE: Sophie.

ANA BELL: Sophie. Sophie said, 50 was too low. So I immediately know that it cannot be any less than 50. So I've already eliminated half of the numbers. So my next guess is 75. Is your number 75? Is your number lower or higher?

AUDIENCE: Higher.

ANA BELL: Since it's higher, I'm eliminating this half here. Is your number-- so between 75 and 100. Oh, boy, you're putting me on the spot. What's that?

AUDIENCE: 87.

ANA BELL: 87, thank you. 87?

AUDIENCE: No.

ANA BELL: Higher or lower?

AUDIENCE: Lower.

ANA BELL: Lower. So since it's lower, I'm eliminating that half. Is your number 81? Higher or lower?

AUDIENCE: Lower.

ANA BELL: So she said, lower, so I'm eliminating that half. Is your number 78? Oh, boy, that's really hard. 78, OK. Higher or lower?

AUDIENCE: Lower.

ANA BELL: Is your number 76?

AUDIENCE: Yeah.

ANA BELL: Yay. All right, perfect, 76 was the number. So how many guesses have I made? One, two, three, four, five, six-- I made six guesses. So I did get it under 10. But you know what? The game was rigged. So you get the prize anyway, just because I rigged the game. Here you go. Pass it down.

AUDIENCE: Thank you.

ANA BELL: Thank you. So notice, in bisection search, what I did was I eliminated half the search space with every guess. I said, well, she said it's higher or lower, so I definitely cannot be in the other search space, right? If I was doing approximate solution or, in this case, guess and check, I would be asking Sophie, is your number 0, 1, 2, 3, 4, and so on?

So with guess and check, it would have taken me 76 guesses to get to the number, whereas, with this bisection search, that I just did, it only took me 6. Isn't that cool?

So that means that the larger the space actually is, that I need to search, the better it is to use bisection search, this bisection search method. So that's basically what I'm illustrating here. So we have our original search space. We're going to choose a guess halfway, eliminate half of the guesses. Then we're going look in the remaining interval, eliminate half the guesses, and so on and so on.

So then this is the code for bisection search. Also looks intimidating, but it's not so bad. So we're initializing a bunch of stuff up here. The most important couple of things we're initializing are, first of all, this high and this low boundaries.

So with the guessing game, the low boundary was 0, and the high boundary was 100. When we're looking at the cube root, the low boundary is going to be 0, and the high boundary is going to be just my cube, because a guess to the power of 3 cannot be any greater than cube.

And then, I'm just going to do the same procedure that I did with the guessing game, which is I'm going to make my guess, be halfway in between. So with this guessing game, I had to sort of choose, if there were four numbers in between, should I go higher or lower?

Well, when we're doing by bisection search, here, we don't care about that. We're just going to do floating point division, because we want decimal numbers. So I have a low boundary and a high boundary. And I figured out my halfway point.

Then I have this while loop here. A while loop is similar to the approximation method, where, as long as I don't have a guest that's good enough-- so this, depicted by this greater or equal to epsilon-- as long as my guess is not good enough, I'm going to keep guessing. That's what this while loop is saying.

So if the guess to the power of 3 minus cube is not good enough, keep guessing. And the way I keep guessing is this part, here, says, my guess was too low. So if my guess was too low, set the low boundary to be the guess. Because I don't care about all of the other numbers that are much lower than me.

So set the low to be the guess. That's what that line is doing. And otherwise, my guess was too high. That's what this else is doing. So set the high to be the guess, because I don't care about numbers any higher than my guess.

Once I have these new boundaries, I make another guess, again, halfway in between the new boundary points. So essentially, I'm just halving my interval with every single guess. And that's what the while loop is doing. And then I print out the remaining part.

So notice the search space originally being N, we're halving it with each guess. So the first guess divides it by 2, the second guess divides it by 4, and so on. So by the time we get to the k-th guess, N/2k, the k-th guess, let's say that's the actual answer we're interested in. There's only one value in that little interval. And that's the answer we want.

So 2 to the k is equal to N. And then how many guesses did we make? k is equal to log base 2 of N. So when we are playing the guessing game of 100, my end was 100. Log base 2 of 100 is 6.-something, I think.

So in fact, I could have said, if I don't guess it within seven guesses, you would have won as well. So that's why the game was rigged. So the guess, notice, it converges on the order of log base N instead of just linearly in terms of N. So that's why it's so powerful.

One last thing I want to mention is the code I showed only works for positive cubes. And that's because of the following. So I have this 0 and 1. Let's say I'm trying to find the cube root of 0.5.

When I first set my initial boundaries, my low is this one, and my high is this one. But what's the cube root of 0.5? Is it within this boundary or is it outside this boundary?

AUDIENCE: Outside the boundary.

ANA BELL: I heard, outside. It's like 0.7 something. So it's out here. So with this particular code, I'm going to be halving my interval in between those numbers, but I'll never get to an answer. Because the actual cube root of 0.5, or numbers less than 1, is going to be outside that boundary.

So there's a small change you can make to the program, which will fix that. And that's in the code. I didn't put it in, but it's a very small change, a small if statement. So that's it. All right, thank you.

[APPLAUSE]