[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: OK, so let's get started. Today's lecture, we're going to do a little bit of a recap of the last lecture. We had begun talking about binary numbers. And then, we're going to dive into our second algorithm of the class, the approximation method algorithm.

So let's remember the motivation we had for even talking about binary numbers and how numbers are represented in a computer in the first place, and the motivation was this piece of code. So it's very simple, we have an initial x is 0, and then we have a loop that just adds 0.1 to itself 10 times. And then, we printed whether that sum equals 1.

And what we saw was that it was false. Printing x equivalent to 1 was false. So then we printed what the actual value of x was after adding 0.1 to itself 10 times, and we saw that that summation was actually 0.9999999. And, of course, to Python, 0.99999 is not equal to 1. So that's why we had printed false for x equivalent to 1, that expression. And so this is our motivation, why in the world does this happen in programming and Python, and something like this could really screw us up if we're not even able to compare floating point numbers.

So last lecture, we ended with this piece of code. It was a way for us to get the binary representation of a number in base 10. So given some number, we followed a really simple recipe, a really simple algorithm to convert that number into base 2.

The stuff that's in boxes, let's not worry about it for now but let's look at just this part right here. So the stuff that's in between the two boxes, and this is the part that does most of the work for us, or all of the work even for us. It basically creates a string, initially empty, and the idea was that we were going to prepend either 0 or 1 to that string, depending on whether the number we had was odd or even.

So for a number like 19, if we wanted to convert 19 base 10 into base 2, what the algorithm was doing is over here in the loop, it says while this num, whatever it is, initially, 19 is still greater than 0, let's get the remainder when we divide the number by 2. So that's what this num%2 is doing, it's either getting a 0 or a 1. So the remainder when we divide 19 by 2 is 1. And we're going to prepend, so we're casting this one integer to a string and prepending it to this result, which is initially empty.

So that's what this line is doing, result equals this thing here. And then, we're going to take our number and integer divided by 2. So we're going to take the number 19 and divide it by 2. So that's going to be 9.5, but we're only interested in the integer portion of it, so 9.

And then, the loop does the check again, is 9 still greater than 0? It is, so then, we're going to say, what's the remainder when we divide 9 by 2? It's another 1, so we're going to prepend it, that remainder to the string that we're building up.

And again, we're going to divide this number by 2, so now we have 4.5 when we only grab the integer portion of it, 4. And again, we ask what is the remainder when we divide 4 by 2. it's a 0, so we prepend the 0 to this binary string we're building up. Again, we divide it by 2, it's a 2. The remainder when we divide 2 by 2 is 0, it's an even number. Again, we divide 2 by 2, it's a 1, and the remainder we get when we divide 1 by 2 is a 1.

And so this is the string that we had eventually systematically, iteratively built up with this loop here. And num after we divide this is going to be 0, and then we break out of the loop. So the binary representation of 19 was 10011 base 2. We just kept it as a string.

The parts that are in red boxes is us dealing with a negative number. So if the user wanted to convert negative 19 to a binary representation, this if else up here, says is the number less than 0, if yes, let's set a negative flag to true, and let's just assume the user gave us a positive number. So we convert that negative 19 to the absolute value of itself, positive 19.

This code goes through as if the user had given us a positive number. And then, at the end, we would get the same number as before, except that we're going to prepend a negative sign. So the binary representation of negative 19 is just negative the same thing.

OK, so that was where we ended up, we talked about these integers, but now what about fractions? Integers seems really easy. There's a really easy simple procedure algorithm recipe for us to follow to get the binary representation. But what about these fractions? Oh yeah, sorry.

AUDIENCE: So how does the negative, everything's going to be 0 or 1, so how does it read the negative?

ANA BELL: Oh, it doesn't read it, we just pretend like we were given a positive number. And then we just do the same process over and over again.

AUDIENCE: OK, so the computer doesn't know it's negative?

ANA BELL: I mean, for the purposes of the algorithm, it doesn't need to know because the number will come out the same. We just flag it as being a negative number. And then at the end, we say, hey, we were actually given a negative number. So let's just pop this negative sign right in front of it.

AUDIENCE: OK. And then when we talk about the powers that 2 is to, are we going from left to right descending powers or ascending powers?

ANA BELL: We are actually doing ascending when we're building up the string because we're going right to left.

AUDIENCE: So right to left is ascending?

ANA BELL: Yeah, exactly. So this is 2 to the 0, and this is 2 to the 4, yeah. So in terms of fractions, if we're thinking about what it means to talk about a fraction in human-readable base 10, so numbers 0 through 9, when we have 0.abc we're basically saying that's a divided by 10, plus b divided by 100, plus c divided by 1,000, and so on.

And in base 2, we're going to have the same sort of thing going on. If we're talking about a base 2 representation of a number 0.abc, where now a, b, or c is just 0 or 1, instead of 0 through 9, it's going to be the same thing. So we would have a divided by 2, plus b divided by 4, plus c divided by 8, and so on. So now we're dealing with powers of 2 instead of powers of 10 because our base is now 2 instead of 10.

So that means the binary representation of a decimal fraction basically means can we find some sort of combination of these values, 0.5 times a 01, plus 0.25 times a 01, plus 0.125 times 01, and so on and so on. So these are all the powers of 2.

So I'll give you the recipe for how we can actually find the representation of a fraction. And this is not something that we expect you to come up, just like the recipe for this is not something we expect you to come up with. But given the recipe, you should be able to intuitively figure out what is the code that actually performs this action that does this recipe.

So the idea to convert a decimal fraction in base 10 to a binary fraction in base 2 is as follows. So let's look at the decimal number 3 divided by 8 just as an example. So that's 0.375, but we know it's 3 over 8. In base 10, so using numbers 0 through 9, we end up saying it's 3 over 10, plus 7 over 100, plus 5 over 1,000, that's just base 10.

But we need to come up with a way to convert this into base 2, and so the trick here is to basically say what is the biggest multiple of 2 that I can multiply my number, my decimal number with such that I end up getting a whole number, an integer? That's sort of the trick to this whole thing, can I multiply my 0.375 or whatever fraction I'm interested in changing to base 2 by some power of 2 big enough such that I'm going to get a whole number out of the multiplication? And it has to be a power of 2 because we're converting it to binary, 0s and 1s.

So in this simple example, 0.375 is 3 divided by 8. So that means that the smallest power of 2 I can multiply 3 over 8 by to give me a whole number is 8, that's 2 to the power of 3. So if I multiply 0.375 by 8, 3 over 8, times 8 gives me 3 in base 10.

And now, this whole number I know how to convert to binary. I have a recipe. We've done it on the board here. We have the code on the previous slide. So all we have to do now is convert the number 3 to binary, which is just 11, base 2. But this 11 is a representation of the number 3. So in order to get back to 0.375, I need to divide my 3 by 2 to the power of 3. So I need to divide my 11 by 2 to the power of 3.

And in binary, dividing by some power of 2 just means shifting the decimal point over, just like in base 10, dividing by 10 means shifting the decimal point over. So if number 3 is 11, and I multiplied by 2 to the 3 to get this value, to divide by 2 to the 3, I just need to move the decimal point from just after the 11, over 1, 2, and then add another 0. So the representation of 0.375 becomes 0.011, I just shifted this decimal point over by 3 because now we're dealing in base 2.

OK, so that's the system. That's the recipe for getting this binary representation out of a decimal number. But there's a problem, this is all relying on the fact that I can find this magical power of 2, that if it's big enough, I can find such a power of 2 that when I multiply it with my decimal number, I get a whole number out of it.

And that's not always the case, sometimes that power of 2 is going to be really, really big, or it might not even exist. And so if it's really big, or if it doesn't exist, that's where we run into problems as we're going to see in a little bit. So this is all relying on the fact that I can find this power of 2.

So here's the code to actually do this recipe that I had on the previous slide, finding the power of 2, doing the conversion, and then shifting the decimal point over. So I'm going to do a quick overview of the pieces, and then we can run the Python Tutor just to show you exactly step by step what's going on.

So let's say I want to do 0.625 and convert that to a power of 2. So I've got my x initialized up there. This bit here, so this big box here, is the part that finds this magical power of 2 for me. It's just a loop that keeps incrementing the p, the power, such that 2 to the power of p multiplied by x, this %1 just gives me the decimal bit out of that multiplication is 0.

So I'm going to keep multiplying 2 to some power of p by x as long as I still have a decimal piece to my number. As soon as this %1 becomes 0 that means that the number I end up with is some number .0. There is no more decimal part to it.

At that point, I break out of the loop and I found my power p. This is going to be the integer, so I'm multiplying x by that special power, by 2 to the power of that special power. And now, I have this number, so on the previous slide, it's the number 3 in base 10.

And then, this box here is exactly the same as two slides ago, it's this procedure here. It's taking my number, whatever it may be, and getting the binary representation of it. And after that, we need to figure out how many spaces to move the decimal point backward. So what is the power of p we multiplied that number by, and now we need to work our way backward and say, well, that dot is here let me move the dot back p steps.

And that's what this is doing, so it's iterating through p minus however long this thing is, to pad the front with 0s. Because sometimes this is going to be a really small number, so I need to add some leading 0s before I put my decimal point. And then I put my decimal point and that's all this line is doing, and then I print my result.

So Python Tutor. All right, so step through. So this is 0.625 just like in the slides. P is initially 0, so now this loop is just incrementing p one by one to find the point where I have a remainder of 0.

So here, I'm actually also printing the remainder. So here we still have a non-0 remainder. So it's 0.625 as a remainder, 0.25 as a remainder, 0.5 as a remainder. And then at some point, I had multiplied it by 2 to the power of 3 because p is 3, and I had a 0 remainder. So now I've broken out of that loop and I know num is equal to 5. I multiplied by 2 to the power of 3, times 0.625 to give me 5.

So now I need to convert num, which is 5, using this process we did here, into binary. That's what this code is doing, and this is exactly this process we had here. So I'm creating this result string and then prepending a 0 or 1 whether the number is divisible by 2 or not.

So the number 5 in binary is 101. So that means I have 101. as my binary representation of 5. And now the code is going to go through this loop, which means it's going to move the decimal point to the left three slots because I have to multiply by 2 to the power of 3 to get the 5.

So you can see it's going to go loop through three slots. So here it is, it made the 0.101. And then sorry, this bit, which I skipped over, applies the dot, so it puts the dot in front of it. And then, the last step is to just print the representation. So the binary representation of 0.625 is 0.101.

So here is the code, and we can run it. So 0.5, the representation is 0.1, 0.625, which is what we had just done, the representation is 0.101. And we can play around with a bunch of these values. But then when we do something like 0.1, what is the representation of 0.1 going to be? Because now we can use this code to get the representation of whatever decimal we'd like.

0.1 was this troublesome decimal, so let's see exactly what happened. Well, it had to do a whole lot of divisions, it had to test a whole bunch of powers of 2 before it actually got to a whole number. In fact, about 50 of them. And we know that because there's about 50 of these 0s and 1s here. So it was approximately 2 to the power of 50 that it had to multiply 0.1 by before it got to a whole number.

So what that means for us is a number that's kind of a linear combination of powers of 2 is really easy and fast to compute, something like this one here, 1 times 2 to the negative 3 is 0.001. But something like 0.1, which isn't as easy to see what the linear combination of all these powers of 2 are, is not so easy to compute. And in fact, we had to use our program to figure out exactly what it is. And for us, it was about 50 of these digits long, which was pretty long. And some of these numbers could be even longer, potentially infinite.

So the point here is that everything in computer memory is represented in terms of bits, 0s, and 1s. The reason we went through this whole computation is because there are some numbers that are just going to be way too big to fit inside the computer hardware, inside these representations. So for integers, it's straightforward to deal with. We had a really fast way to compute the base 2 representation, but for fractions, it's a lot harder, and those numbers can be really, really big.

So now how are these numbers actually represented inside computer memory? So they're actually being represented in two pieces. One piece is a significant digit, and the other piece is the power of 2. So if we had the representation 1, 1 inside computer memory, basically the significant digit is 1, and the power of 2 is 1. So that means we're going to take this one dot and give it the power of 2. So we're going to add a 0 after it. So this is the binary 2 representation because we basically just move the dot from here to here, and then the number 1, 0 in base 2 is 2.0. That's what we have on the first line.

1, negative 1, that representation means I'm going to take the significant digit 1, and the power of 2 is negative 1, so I'm going to take this decimal point and move it to the left 1. So this is going to be 0.1, that's this, number 0.1, which is 0.5. This is base 2, this is base 10.

And just to bring the point home, 125 is going to be 125 as a significant digit, and 2 to the negative 2, how is this going to work? Well, we're going to take the 125 and convert it to a power of 2. So it's what is this, I'm not going to remember what it is, 1111101, this is what 125 is in base 2.

But the exponent here tells me it's -2. So instead of putting the dot here. I'm going to move it one, two over. So this is the actual number I'm representing in memory. And now I can just convert the two pieces separately. So this is going to be 31.25.

OK, so this is how computers actually represent numbers inside memory. And we call this the object type, which is decimal or real number float because this decimal point kind of floats around.

AUDIENCE: Is it 5 or 31.25 per number in base 10?

ANA BELL: Base 10 is 31.25, and 125 is how it's represented inside memory. So it's a base 10 sort of thing. And then what is the power of 2.

So there's a couple conversions being done here. We're representing the 125 is base 10, and how much we need to move the decimal point, but first we need to make the conversion of 125 to binary, which is this long thing here, not counting this decimal point. The negative 2 tells us we need to move the decimal point over, and then we have the actual number we're trying to store. And the reason we're doing this is because we're mostly just storing numbers as whole numbers inside the memory. We're not storing fractions.

AUDIENCE: Yeah, I'm just a little confused because you went to all that trouble to convert the decimal to-- oh, that was for fractions.

ANA BELL: That was for fractions. Exactly, yeah. OK, so in the end, we did all that because we're trying to figure out the error, why do we get this error inside our programs? Well, in the end, it's because computers have a finite number of bits to store data. Most modern computers maybe have 32, maybe 64 bits to represent significant digits. So if we have 32 slots in order to put these significant digits, if our number base 2 representation was 50 digits long, then we're going to truncate at 32. We can't store those extra bits.

And so a number like 0.1 ends up actually being an approximation in base 2 inside computer memory. We're not able to store that number exactly perfectly. So it becomes an approximation. And the approximation actually ends up being at the 32nd bit, that either will be 0 or 1 depending on how we decide to truncate.

And so the error is actually 2 to the negative 32, which is on the order of 2 times 10 to the negative 10, which seems pretty small. It's a very small error but we just saw that that error accumulates really, really quickly. So while 0.1 has an error at the 2 to the negative 32 slot, if we take that error and we just kind of accumulate it over 10 increments, as we had this loop that went through 10 times, we see that that error ends up becoming a big problem. We see that it actually at the negative 16th slot or something like that, it starts to round to the wrong thing. And so we see things like this, we expect it to be 1, but it's not 1.

OK, so the moral of the story is we don't want to use equivalents, the equivalent operator, the == operator, when we're comparing floats. Because of errors like this, the errors can accumulate and then we start getting the wrong answer, and then your programs end up not doing what you expect them to do.

So we always want to test whether some float is within some epsilon of another float, and so that brings us to an approximation method. Last lecture, we saw the guess-and-check method as a really simple algorithm for solving problems. We have a set number of solutions that we can check, and then we check each one by one, and then at some point, we either find the solution, or we've checked all that we can check and we haven't found the solution.

It's usually an integer what were the things that we're checking but as long as you have some finite set of values you can check for a solution through, guess-and-check is totally applicable. But the problem is it's a little bit limiting, it doesn't give us an actual approximation to the square root. If you remember the code we wrote last time, it didn't actually say I'm approximating the square root of 5 to be 1.4 something or whatever it is, or 2 point something, it was just able to tell me the square root of a perfect square or that the number you gave me is not a perfect square and so it's a really limiting algorithm.

But the approximation method, the one we're going to see today actually is going to be able to give us an approximation to the square root of any number. So it's better than guess-and-check because we don't just want the correct answer or nothing. It's not an all-or-nothing kind of situation, it's that we can approximate the answer to some degree.

So we're going to use guess-and-check when the exact answer that we want might not be accessible. We need some way to find an answer that's just good enough. And approximation methods will not always, and not usually, actually, most of the time will not give us the right answer. They'll usually give us an approximation that's good enough. And approximation methods, they came about because of the exhaustive enumeration limitation. We're not able to test all the possible values to find the exact square root of a number because those values are all infinite.

So floating points come into play here. The whole thing we've been talking about at the beginning of this lecture and last time, floating points come into play here because they're very important to this method. Now that we're comparing floats we're going to have to be careful about how we actually do the comparison.

So how can we approximate the square root? Well, instead of looking at just whole numbers and saying whether we found the root or not, what we're going to do is have smaller increments. So no longer are we doing just integer guess-and-check, we can do 0.1 0.2, 0.3, 0.4, and so on, until we get to a guess that's close enough to x. So we say that 2.1 or whatever is good enough to the square root of 5.

What does it mean to be good enough? Suppose we wanted to find this approximation to the square root, the guess-and-check was not able to do this for us but the approximation method can. So what we're asking for can we find a root such that that root, times that root, times itself is equal to x?

And we're going to do this such that we have a good enough approximation. So that means that root that we're going to find, minus x is going to be less than some epsilon, or the absolute value of that subtraction is going to be less than epsilon. So where we did incremental step-by-step, we're going to go through as long as we are within or until we are within some epsilon of x.

So the algorithm will be as follows, we're going to start with a guess that we know is too small. So for the square root of a number, let's start with 0. And then we're going to increment it by a really small value. With guess-and-check, we incremented it by integers, with this particular method we can increment it by 0.5s or 0.1s or 0.0001, whatever we'd like.

That new increment gives us a new guess. We're going to check whether this new guess is now close enough to x If it is, we're good, and if it's not, we're just going to keep incrementing the guess until we get close enough to the actual answer.

So we have two parameters we actually need to set in the approximation algorithm. The first is an epsilon, so this is down here, how close do we want to be to the final answer, what's the leeway we're going to allow? And second, is the increment, so how much do we want to change our guess by? The way the algorithm performs depends on the values we choose for these. Obviously, if our guess is smaller, if we decrease the increment, we're going to get a much more accurate approximation.

If we increase the epsilon, how close we want to be to x, our program is going to be faster because we're going to enter that plus minus epsilon boundary faster, but it's going to be less accurate because at some point, we're going to enter the boundary and I'm going to say good enough, I'm not going to get any closer to x because there's no need to, I'm already within epsilon. So here, the good enough guess was to the square root of 5 was 1 point something, but on the previous slide, when we had a smaller epsilon, the good enough guess was 2 point something.

So the approximation algorithm is like guess-and-check except that we have some small increment, we change by a small amount, and we stop when we're close enough. So we're going to check that the absolute value of this solution minus the actual answer is within epsilon.

So here's some code where we can implement what finding the square root of a number with approximation method. We have some stuff here that we're initializing, so this is the thing we want to find the square root of. This is how close we want to be to the final answer, and this is our increment. Num_guesses is just to keep track of how many actual guesses we're doing, and we're going to start with a guess that we know is too small, 0.

This is the loop that does all of the work for us. So the way we would say it in English, it says basically while our guess is not within epsilon keep making new guesses. So while what does it mean for the guess to not be within plus or minus epsilon? Well, the absolute value of our guess squared minus x is greater or equal to epsilon. So while we're still too far away, let's make a new guess.

So we increment our guess by the increment value. So originally, we were 0, then we're 0.001, then we're going to be 0.0002, and so on. This num_guesses again, is just for us to keep track of how many times we've actually gone through this loop. And at the end, we can print how many guesses we've done.

OK, so here's the code, and 36, so we can run it. What do we see? Here's our approximation to the square root of 36. Now, we know it's 6 and, of course, if we kept going we could have found probably exactly 6, but notice this approximation algorithm stops as soon as you enter that plus minus epsilon boundary. Yes?

AUDIENCE: Do for loops always increase in integer amounts?

ANA BELL: Do for loops always increase in integer amounts? Yes, the step has to be an integer, positive or negative, yeah. So exactly, a for loop would not have worked here.

So here we stop this algorithm as soon as we entered that plus minus boundary of epsilon. And so 5.9991 is close enough to the square root of 6, and that's what we're reporting.

The number of guesses here was about 59,992. and that makes sense because our increment is 0.0001, and we went all the way up to 5.99. So with each time through the loop, we incremented by 0.0001. So that's just this times 10,000. That makes sense.

So let's try it with a couple other values. So here it is with 24, 4.89. Again, we're seeing these floating point errors pop into play whenever we see this weird like 0.00000 and then some small amount at the end, that's these floating point errors just given the numbers we're working with adding up. Here's the square root of 2, 1.41. Again, floating point error. But this time on the other side, 0.9999.

12345, run it. It took a second, there was a little pause, and then it gave us the answer just because it has to loop through about, what is this, 1 million times, so did that loop 1 million times to get us the answer? And then we can try one more, 54321, this should take about five times as long because 12,345 took about one second, this one should take about five seconds but it's not.

I'm pretty sure I was talking for more than five seconds and this program is not ending. So something's gone wrong. I'm going to stop it. Remember, you can stop it by clicking the shell, hitting Control-C, or the little square here in the corner.

So what went wrong? Oh yes, my question is, will this loop always terminate? And 54321 was an example of the loop not terminating. So what happened? We did all this. Let's try to debug what exactly happened because clearly what we have in code right now is not really giving us much information. So let's add some print statements.

The print statements I'm adding is just in here. So everything else that's not boxed is the same as on the previous slide. The only thing I'm adding new is this if statement here. So every 100,000 guesses, so every time I've gone through this loop 100,000 times, I'm going to print what the current guess is and what the guess squared minus x is, so how far away I am from x, the epsilon. Yeah, not the epsilon but how far away I am from x.

So let's run that code. It's down here. I added a little bit of extra thing, which is just it's not printing the whole time, it's just going to pause for me just to talk about what's going on. So here I have the code run, has run, so my first 100,000 times through the loop I have my guess being about 10. And how far I am from x is about 54,000. So I want to be 0.01 away from x because that's what my epsilon is. And so here, I'm 54,000 away from x. So clearly, that's too much.

Let's continue, so then we make more guesses, and then here, when my guess is 100, I am about 44,000 away from x from 54 so looking good. Let's continue. So with 120, I'm 39,000 away from x.

With 200, I'm 14,000 away from x, so it's looking much better. I'm getting closer and closer to getting that difference being 0 or 0.01, continue. With 210, I'm 10,000 away from x, and then I'm almost 6,000 away from x, and then I'm 1,000 away from x. And then, from 230, as my guess, which brought me 1,400 away from x, the next time, I have 240. The next printout I have brings me to 3,000 away from x. So I was 1,000 but now I'm 3,000.

And then from there on things break down really quickly because I just get now farther and farther away from x. So here I am continuing the program for a little bit, and then I just keep making guesses because I was never within that epsilon. So here's 500, and now I'm almost 200,000 away from x. And so now you see what's happening, this program is just going to keep getting further and further away from where I need to be.

So let's visualize what exactly happened. This is our x, 54,321, and this is our epsilon, let's say it's 0.01, obviously not to scale. Blue is going to be representing one guess. So here's a guess, and then we have the guess squared, a green. So let's just for visualization purposes, let's say this is our guess, and this is our guess squared. OK, we're far away from x, we're definitely outside the epsilon boundary. We make another guess by incrementing it a little bit. This is the guess squared.

We make another guess by incrementing it a little bit because we're still far away from that plus minus epsilon, this is our guess squared. We make another guess, this is our guess squared, we're pretty darn close to that plus minus epsilon boundary. We want to be within that plus minus epsilon. So one more guess should make it right. This is our next guess, but now the guess squared is on the other side.

This is the big reveal, you guys. So what happened? What happens now? The program says keep guessing because we're not within epsilon. So it's going to make another guess, guess squared, and it's just going to keep guessing. And then our guess squared is just going to keep getting bigger and bigger.

So we basically overshot the epsilon, we've overshot our little plus-minus boundary that we were interested in being within. We didn't account for that when we wrote the loop, all we wanted to do was be within epsilon and our program would end.

So let's fix that. One addition will fix that, and it's something that we had been doing in guess-and-check anyway. In guess-and-check we would say something like if we've passed the reasonable guess, when we know that guess squared from here on out is definitely too big, just stop. Stop guessing. Just stop.

And so we can have that same thing here as just another ending condition. So everything in this code is the same as before, except for this red box. We're adding another stopping condition that basically says keep guessing while we're still guessing something reasonable. But when we get something that's not reasonable, which is when the guess squared is greater than x, we're way past it, stop guessing as well.

So whichever one of these conditions, either this one or this one being within epsilon is true, we break out of the loop. And then, we have an if-else, kind of the same sort of thing we've been doing so far in the guess-and-check, why did we break the loop? Did we break it because we were within epsilon? That is the else clause here. If we did, then we say this is close to the square root of x. But if we broke it because we've passed a reasonable number of guesses, then we know we failed to find the square root because we overshot the mark or whatever.

So here is the code with 54321, but now we have that extra condition here, guess squared less than x. So we see that we've done some number of guesses, 2,300,000, and the message we get is we failed to find the square root, makes sense because we knew we would fail, and we're also reporting what the last guess was, and the last guess squared just in case the user wants to use that information for anything.

What are some solutions to fix this? If we don't want to fail, what can we do? Well, I gave you a hint right here. We can decrement our increment, or we can decrease our increment. If instead of adding 0.0001 every time through the loop, let's add 0.00001, so let's make it guess 10 times as many guesses. We're going to have to wait a little bit, maybe about 10 seconds, but the program will end, it's taking this long obviously because it's making all of these extra guesses. For every one guess we had with the program that failed we're now making 10 guesses because we decreased our increment by 10.

OK, so it ended, and we see exactly that idea in the number of guesses. So here we had 2.3 million guesses when our increment was 0.0001, but when our increment was 0.00001, four zeros, we had 23 million guesses. So obviously we had 10 times as many guesses, which made our program be 10 times as slow. But now, we didn't fail because we were able to go within that epsilon. So we found that 233.06864, which is pretty close to what we had before, is within 0.01 of epsilon.

So with approximation methods, it's possible to overshoot the epsilon. We have to be a little bit more careful now about what our end condition is. Yes, we can check that we are within epsilon, but we have to also use a little bit of common sense, maybe algebra, something like that, to figure out is there a way we can overshoot the epsilon, and how else can we stop the program without it running into an infinite loop because that would be bad.

So I think I already went over this, what are some observations about running it? Yes, it reported failure so we reset the increment down to 10 times smaller than what it was before. The program was slower because we had more values to check through.

So the big idea here is we want to be careful when comparing floats. If we were using something like == sign, that would have been a complete disaster. That we might have never been within epsilon or something like that.

Yeah, so what are some lessons we learned in approximation, so we can't use == sign to check for exit conditions? We always have to check whether we are within plus or minus some epsilon of the actual answer. We have to be careful that the exit condition being plus or minus within some epsilon doesn't jump over our exit test as we just saw. In that case, we add some extra conditions.

And then we saw that we actually have a trade-off, we can have a program that does terminate and reports a correct answer, it doesn't say we failed, but it does report a correct answer but that could be a program that's a lot slower. It's a lot slower because we had to decrease our increment.

Alternatively, we could have increased our epsilon boundary, our plus minus epsilon that we allowed to be within could have been bigger, but then we would give up on some accuracy as well. So there's always this trade-off of speed versus accuracy to get the program to actually give you an answer or to do what you'd like. And depending on the application, you might want accuracy versus speed or vice versa.

So this approximation algorithm is really slow. To get an answer for the square root of 54,321 we had to decrease our increment to something like 0.00001. And we ran it, and that program took maybe 10 seconds to run on my computer because we started from 0 and we were just painfully incrementing that increment one at a time, even though we knew sort of from what the number actually was, 54,000, that the square root of it could not really be that low. But that's just the algorithm we had. We had to start from something, 0 just in case the user gave it other numbers which didn't make sense to start higher than that.

And so the approximation algorithm as you saw, can be really slow. The question I have is, is there a faster way that still gets good answers? And the answer, of course, is yes. And we're going to see this algorithm in the next lecture.

So a quick summary, we saw floating points. We did a lot of calculations with binary numbers. You don't need to know how to do those calculations. But again, given a recipe or an algorithm, can you take that and put it into code? Floating point numbers introduced a little bit of challenge for us in comparing them because of the way they're stored in memory.

We can't represent some of these numbers exactly in memory. So that's a problem. Because they're not represented exactly in memory, we might magnify some errors, as we saw with that loop, and the approximation methods use floats. Unfortunately or fortunately, they need to use floats because we need to have a small increment, and we have to be mindful of these issues when using them.