[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: OK, so let's get started on today's lecture. Last lecture I left you off with the promise of bigger and better algorithms to do what we've been trying to do, which is to approximate square roots and things like that. So today will be the introduction of our last algorithm for a bit before we'll start talking about more Python syntax. But today we're going to introduce the bisection search algorithm.

OK, but before we get into that, let's try to remember where we left off last time. So last time we talked about floating point numbers and then we talked about approximation algorithms. So the reason why we talked about floating point numbers is because we wanted to come up with an algorithm that was better than guess and check. Guess and check was really limiting. We were basically limited to some exhaustive number of potential solutions.

But we didn't just want to have an exhaustive set to look through for a solution. We wanted to be able to actually come up with an approximation to solve our problems. And so we talked about floating point numbers because we said, well, instead of having, for example, integer increments when we searched for square roots of values, let's try to have smaller increments. And so if we have smaller increments than an integer, well, we were starting to look at incrementing by 0.1 or 0.25 or 0.001, whatever we want.

And so then since we started talking about these floating point numbers, it was important to understand what happens behind the scenes. And we saw that these floating point numbers can't actually be represented in memory directly exactly. There's always, for the majority of the numbers, there's going to be some sort of rounding that happens when that number is stored in memory. And the rounding is very small. It's something like 10 to the negative 10. Or sorry, 2 to the negative 32, which is approximately 10 to the negative 10, which seems small, but we saw even with just a loop that added 0.1 to itself 10 times, we were already getting very surprising results.

So the approximation method introduced the idea of, yes, we can get an approximation for the square root of a number, but we can't check for equality. We can't say I'm going to come up with this approximation such that this approximation squared or whatever problem we're trying to solve is exactly equal to the number we're looking for. So we had to have a little wiggle room. And that wiggle room came in the form of an epsilon. So we were approximating a solution by basically saying, does this solution come within plus or minus epsilon of my desired value.

So we came up with a nice algorithm, the approximation algorithm, and we tested on a bunch of different values. We were incrementing a small increment a little bit at a time. And for the problem where we're trying to find approximate the square root of some value x, we were saying, well, I'm going to keep making these small incremental changes to my guess until I come within plus or minus epsilon of my actual value. The guess squared was within plus or minus epsilon of my x.

And this was the nice slide that was kind of the big bang of last lecture where we said we have to be careful about the way we write these approximation algorithms, because we might overshoot our epsilon. So if this is our guess and this is a guess squared, the blue arrow increments normally, whatever increment we choose. But then it's possible that at some point the guess squared comes just short of the epsilon, the x minus epsilon. And with the following increment, the guess squared becomes just past x plus epsilon.

And so the code that we ended up writing, which it made sense right when we wrote it, actually ended up giving us an infinite loop, because it never stopped. We never were within that plus minus epsilon, and so we would just keep making guesses from there on out. So we ended up getting an infinite loop for our program.

The solution was to take a little bit of code from guess and check and said let's add an additional little sanity check stopping condition. And so everything except for this box was the approximation algorithm. And we added the thing that I've boxed here as our sanity check that we grabbed from the guess and check algorithm that basically said if we've made a guess that is just past the reasonable number, a reasonable guess, we know that all the guesses from here on out will also be unreasonable. And so there's no need to keep searching, and that condition will cause us to stop our infinite loop, our potential infinite loop.

So this guess squared is less than or equal to x. Basically says stop when we go past the last reasonable guess. And that condition plus the regular condition from an approximation algorithm, which says, I want my guess square to be plus or minus epsilon of the actual x, those two conditions together made up my algorithm. And that's the algorithm. It's just this loop right here, this while loop with this increment. So it looks really, really simple.

And so what we ended up having is these two conditions. So I want to be within epsilon and I want to still be making reasonable guesses to be, sorry, I want to be, sorry, outside of the bounds of epsilon and still be making reasonable guesses. That's the condition that causes me to keep making more guesses. And when either one of these becomes false, I'm going to stop making guesses. And that's what the if else down here says. It says one of these conditions became false, either this one. I'm making unreasonable guesses now. Or I've come within plus or minus epsilon. So which one is it?

So here I'm making unreasonable guesses. So I've exited the loop because I've gone too far. In which case, I print I fail to find the square root. And otherwise, I've exited because I am now within plus or minus epsilon. So let me just run the code to remind ourselves what it looked like. So here we're trying to find 54,321 was this troublesome value being within plus or minus 0.01. Our guess squared to be plus minus 0.01 of 54,321. Our increment seemed really small, 0.0001.

But when we ran it, it took a couple of seconds, and we made about 2.3 million guesses. And the code says we failed to find the square root. And we're also reporting what the last guess was and what the last guess squared was as well.

So what's the solution to this? The solution was, well, we can make our epsilon bigger. So if we made our epsilon be 1, so if we want it to be within plus or minus 1 of 54,321, yeah, that code works. It didn't fail. It made still about 2.3 million guesses, and it came up with this estimate. So as soon as we came with an epsilon, that boundary we stopped the program. It didn't try to do better. It didn't try to get closer to x.

The other solution, if we were unhappy with the fact that we failed, was to make our steps smaller. But what's the problem if we make our steps smaller? Do you guys remember when I run the program? Yeah, it takes longer. And can you approximate how much longer it'll take?

I decreased my step size by 10. So every one step I made last run, I'm now going to take 10 steps. So I'm waiting basically, what, 15, 20 seconds here if the last run took two seconds to run. And now I've also doubled the number of guesses. Or sorry, not doubled. I'm making 10 times the number of guesses. 23 million as opposed to 2.3 million. But the code didn't fail. It found something that's pretty close to the square root of 54,321.

So that's where we left off. And I don't know about you, but I don't want to wait 20 seconds to figure out what the square root of 54,000 is. That seems like an unreasonably long amount of time to come up with an approximation. And we don't wait that long when we do it on the computer or when we do it on the calculator. And so that leads me to the bisection search algorithm. It's got to be a better way for us to solve certain types of problems much faster, but only certain types of problems.

So to motivate the bisection search, before we even look at code, I just want to give you a bit of motivation with a few different examples. The first one is I'm going to give you guys a chance to win some money. So suppose I put a $100 bill at one page in this book. This is actually the last edition, not the edition we're using this year. But I don't have this year's edition, unfortunately, in my office.

So this book is 448 pages long. And I've put some money in this book. And if you can guess where the money is in eight or fewer guesses, I will give you the money. And if you fail, you get an F. Not really. Is this a game anyone would want to play? No. That's what I thought. And in fact, your chances of winning are about 1 in 56. And yeah, I don't want to play that game either.

But now let's say I give you some additional information. With each guess you make, I will tell you whether you are correct, too low, or too high. So I give you some additional information. Is this a game that now you would want to play? Would anyone like to play the game with me? You want to play the game? OK. All right, so you're up.

So I'm going to write down your guesses, because you only have eight. You remember that there's only eight guesses. So what's your first guess? There's 448 pages. You pick 1 to 448. What's your first guess?

STUDENT: 224.

ANA BELL: 224. All right. Smack in the middle, right there. All right, 224. Don't look.

STUDENT: I have no idea what I'm doing.

[LAUGHTER]

ANA BELL: No money. But now I give you extra information. The guess is too high.

STUDENT: My guess was too high?

ANA BELL: Yes, your guess was too high. So not 224.

STUDENT: 112.

ANA BELL: 112. So you want to go here. 112. All right. That's two guesses now. Nope. The guess is too high as well. Still too high. 56? OK. So here somewhere. OK. All right. 56. The guess is too high. Still too high.

STUDENT: [INAUDIBLE]

ANA BELL: 28? All right, 28. I'm going to start writing up here. 28. You have 1, 2, 3. You're at four guesses now. 28. It is-- no. Wait, now I have to remember where I put it. It's too high. It's too high. Sorry. [LAUGHS] Still too high.

STUDENT: 14.

ANA BELL: 14. OK, 14. Right there. OK, now 14. Now it's too low. Now that I remembered-- now that I remember where I actually put it, it's too low.

STUDENT: Turns out it's on page like 400.

ANA BELL: Between 14 and 28. So now you know it's-- yeah. 21? OK. So right there. 21. OK, let's see. Guys, I'm shaking.

[CHEERING]

It's not 100, but there is a 1 and a 0 in it, so there you go. All right. That was awesome. So yes, I'm really glad you played. And actually, you only took seven guesses to get it. So I could have probably rigged it a little bit better. Because your chances of winning this game are about 1 in 3. And you did a really good job.

So what was your thought process, basically? And I think once you did a couple of them, anyone who maybe didn't think about this way kind of figured it out. You were basically guessing the halfway point each time I told you too high or too low.

And so bisection search is a method that you can use to solve problems where there is some sort of order to the thing you're trying to search. So let's say we know our interval. In this case in the book, we knew that we had page 1 to 448. So we had this low point, page one, and this high point, 448. And we know that our answer lies within this interval. And it can be integers, like in this book, or it can be fractional pieces as well. And the idea is you're just guessing the midpoint between this interval. It's as good as guess as anything.

But based on the answer that I give you, because now I give you extra information if it's too high or too low, you can basically eliminate half of the search space. So with the guess 244, what ended up happening is you eliminated this entire half of the book. So it's like I take this original book, 448 pages, get the midpoint, rip it in half, throw away these upper pages. And now you think of it like having this skinnier book. And this is now the book you're searching through.

And then you're repeating the process all over again. The low endpoint is still page one, right because I have no information about how low I need to go. But my high endpoint becomes the guess that I had just made, 224. And now I make another guess midpoint in this skinnier book.

And so this should trigger something in your brain about computation and things that we've learned. As soon as we're saying I'm repeating this process of now just doing the guess again with this smaller version of a book where basically we have a loop. That's something that you should be thinking about.

And this loop is going to be something that just repeats the same process over and over again. Once I've eliminated this upper half of the book, now I'm looking through this lower half, midpoint. Based on the answer I give you, you can rip the book again in half where you are now and now you're searching the skinnier version of the book.

So we're basically cutting the number of things we need to search for in half every time we make a guess, which is really powerful. With guess and check or with approximation method, if we're going to do guess and check on this book, we'd be asking is it page one, is it page two, is it page three, is it page four. And that's tedious. We're not eliminating half the book with each guess. We're just eliminating one page with each guess.

So this idea of logarithmic growth, which is what happens when you eliminate the search space in half with each stage, is logarithmic growth. And we'll come back to this idea again towards the last few weeks of lecture when we talk about comparing algorithms in this class. And what does it mean for one algorithm to be more efficient than another algorithm? What does it mean to run faster? So that's just something I mentioned.

When we do guess and check where we ask one page at a time, that's linear growth. Because if I give you now a book that's double the size and just by bad luck I put the money away at the end, if I put the money at the end in this book that's twice as big, then you're going to have to ask me twice as many questions until you get to the answer.

But with logarithmic growth, if I still put the money in this book that's double the size somewhere, all you need to do is make one extra guess, not 400 extra guesses to figure out which interval it's in. You take from this double book, make your first guess, and all of a sudden we are at this book again.

So let's do another analogy just so we get kind of the sense of where this is going. So suppose that we don't just need to work with numbers. We can also work with anything that has an ordering property to it. So suppose when you came in, I asked you to sit alphabetically. Front left is last name A, back right is last name Z. And then I'm looking for a particular person.

For me, the bisection search algorithm could be basically ask the person in the middle of the class what is their last name. If they say what it is and depending on what they say, I can basically dismiss half of the people. If their last name starts with a letter later than the one I'm looking for in the alphabet, I dismiss the upper half and vice versa.

And then I have this only half of the people to search through. And I keep repeating this process until I have only one person left and either that person is the one I'm looking for, in which case I've decreased by half the size of the class with each guess, and I have one person left to ask. It's the one I'm looking for, or that person that just isn't here.

So let's try to apply the same idea of bisection search to programming. And specifically, let's do the problem we've been trying to solve kind of as a common thread throughout these algorithms figuring out the square root of a number. Not exact. Actually, we're still going to be looking only for an approximation to the square root of a number.

So the idea here is that our interval is if we're trying to find the square root of x is going to be between 0 and x. So basically I can just reuse this number line here. And I have my interval for the square root is 0 and x. So like that. So with approximation method, we would start at 0 and painstakingly make our way little by little. But with bisection search, we're making our initial guess to be the halfway point. Again, we're working with numbers, so the ordering properties is very intuitive.

We ask at this halfway point what is-- with this guess at the halfway point, what is the guess squared? So if the guess squared is up here, so g squared is up here, then I know this guess is too big. So I know I do not need to make any further guesses up here. So that's this case here.

If I know this guess is too big, then my interval now becomes this is going to be the low still, but now this is going to be my high point. And this new interval I'm looking through. But if you think about it, it's the exact same problem I started with when my interval was larger. I still have an interval with a low and a high. I'm still going to make a guess halfway. This new g here. And I'm going to ask again, is this new guess squared less than or greater than x?

Let's say this case is less than. So if the new guess, new g, is less than-- the new g squared is less than x, then I know-- this is new g. Then I know that anything lower than this is definitely not going to be closer to the answer I'm looking for. So now I'm eliminating this half of the search space. And then I keep making the same guesses. Next g, g, latest g.

This is like when you guys name your files, remember, and you've got new file, final file, latest file, version two, all that stuff. That's basically what I did. So anyway, I have this latest g here, which is my new midpoint. And I keep making these guesses and asking the question whether this guess squared is greater than or equal or less than x. And I adjust my boundaries accordingly.

So at each stage, the number of values I have to search through are just half of what I had to search through last guess. So the bisection search takes advantage of two properties. And you can only use it when you have these properties in hand. There's some sort of ordering to the thing you're searching. So you know last names are alphabetical. You know you have this range of values. And you have some sort of feedback. The feedback tells you whether the guess that you made was too low or too high or exact or approximate, whatever you want.

So think about this for a second and answer the question. So you're guessing a four digit PIN code on a phone or whatever. And the feedback the phone tells you is whether the guess is correct or not. Can you use bisection search in this situation to quickly and correctly guess the code? No. Why is that? What are we missing?

STUDENT: It doesn't tell you if it's too big or too small.

ANA BELL: It doesn't tell you if it's too big or too small. Yeah. So I mean, you could use bisection search and you could choose which have to look through. But then basically, you just have to search through all the values anyway in worst case. And then you might as well have just gone from 0000, 0001, 0002, and just have gone incrementally upward.

So how about this extreme guessing game? So you have a friend and you like to play this extreme guessing game where you want to guess a number exactly. So your friend has a decimal number in mind. So it can be the decimal point like any real number from 0 to 10, let's say, including 0, including 10, to any precision in mind.

So the feedback your friend gives you when you play this extreme guessing game is whether your guess was correct, too low, or too high. In this case, can you use bisection search to quickly and correctly guess the number.

STUDENT: You could, but the number might be really long and then it would take a long time.

ANA BELL: Yeah. So I included this word exactly here. If I didn't include that, I think the answer could be yes, because you could play the game to around or approximation to two decimal places or something like that. But I guess if your friend wants to flex with pi in your extreme guessing game, then bisection search wouldn't work. Because if you're trying to find that number exactly, then you'll never get to it. So yeah, you're using bisection search, but it's going to basically be an infinite algorithm. It won't terminate.

So this is the same slide I had at the beginning of lecture, just to remind ourselves what the code looks like when we use the approximation algorithm. Nothing new here. So we had the thing that basically did the work was this while loop. While we were still farther away from epsilon and we were still making reasonable guesses, increment our guess by 0.0001.

Now let's write the code for the approximating the square root of a number but with bisection search. So we're going to follow the same kind of procedure we did here. And we're actually going to write it together on the slides. And I'm going to explain the thought process that goes behind each step.

So the first thing we're doing is we're initializing some stuff up here. So the thing we want to find the square root of, why not do the same number. That gave us trouble last time. And we still want to be within some plus or minus epsilon, again, because we do not want to be comparing floats.

And this num guesses is going to keep track of how many guesses we've made. Basically when we played the guessing game, how many guesses did you do to get to the money. Good. And then at the bottom here, we're going to print out the number of guesses and what the guess was that brought us close to the plus or minus epsilon.

So the first thing we do is we notice there was a repetition. And the while loop here is exactly the same as the while loop for the approximation method. While we're still farther than epsilon away, while our guess squared is plus or minus epsilon away from x, so absolute value of guess squared minus x is greater or equal to epsilon. I guess this could just be greater than details. Let's keep making guesses.

Now, the guesses, we're not incrementing anything. This isn't the approximation method. We need to make the guesses in a smart way. So we're going to initialize some stuff for our algorithm to work, like our original endpoints. And then we're going to do some stuff inside the loop, whatever is repeated, whatever we noted when we were talking about the algorithm. What did we note that gets repeated every time?

Let's talk about the initializations. We need to initialize our two endpoints. For the bisection search to work, we need to know what our endpoints are. So the low is going to be 0. So if we're trying to find the square root of x, we might as well make our low 0. And let's make our high point x. Our high point can be 2x, it could be 3x, whatever we want, but that's too big. We using algebra that definitely it won't be that big. So we can just make our high point x.

And then we just kick off this algorithm with our initial guess. It's going to be the midpoint of low and high. So high plus low divided by 2. So that brings us to just before the while loop here, right here.

And now we're going to repeat some stuff while we're still farther than epsilon away from our answer. So the thing that we're repeating is going to be checking if we are too low or too high. We have a guess in hand now, this midpoint here, and now with this guess in hand, that kind of kicked off our algorithm, we're going to say is this guess too low or too high? That's what the algorithm needs.

So that's an if else, a little conditional here. If the guest squared is less than x, then the guess is too low. So if this guess squared brings us to somewhere here, then we know this guess is too low. What do I do in this case? What does the algorithm say to do? Yes.

STUDENT: [INAUDIBLE]

ANA BELL: Other way around. Yeah. So this is too low. So I definitely don't want anything lower than here. Exactly. So we're going to set our low endpoint. If the guess is too low, let's set our low endpoint to be whatever guess we just made. Because we know this is too low. Anything lower than this is definitely too low. So I don't care about these.

Else, we don't need an el if, because we know the else is the other way around. Else our guess was too high. So if the next time around we make a guess here, something like that, then we know we're too high. And then we neec to set our high endpoint to be the guess. Is everyone OK with that so far? OK.

What remains? So I've changed one of my boundaries, either my low or my high boundary to be whatever guess I just made. What is the next step? What does this algorithm do or this loop do as is? It finished doing whatever is inside and it goes back and uses the guess and check whether the guess squared minus x is greater or equal to epsilon.

Have I changed my guess inside this loop yet? No. So that's the last step that remains. Make the guess be the new midpoint using either the changed high or the changed low. So each time through my loop, I'm either changing my low to be the guess or changing my high to be the guess. So I'm making one of those two changes. After I've made that change, I need to find the new midpoint. So if I changed my low, now I need to make my new guess.

And with this new guess, then I'm happy for the while loop to check it again. Take that guess squared. See how far away it is from x. And then it does the changing of the boundary all over again. And that's it. There's no other lines of code in here.

So in some sense, there's a little bit of trust with this loop that it does the right thing. But if you kind of do a little bit of iteration in your brain or through the Python tutor, you'll see that it actually does it correctly. So we can just use that same number line.

And let's look at the square root. Let's find the approximation to the square root of 36. The epsilon, I made it one just because I don't want to do. So many steps in the Python tutor. But you can imagine if it's smaller, it'll just give us a better approximation.

So we're initializing the x, the thing we want to find the square root of, and epsilon, the low and the high. 0 and 36 in this particular case. Stepping through, the first guess is half of 36 and 0. So 18. So here's my guess is 18. And now we kick off our while loop by saying what is 18 squared? It's pretty big. Definitely bigger than 36. So I'm going to go inside this else because my guess is too high. So my high becomes this. And this is still the low. I know nothing about the low end at this point.

So then my guess becomes the high plus low, 0 plus 18 divided by 2. So that's going to be 9. So you can see my guess has updated to 9. And now I find the guess squared. What is 9 squared? Is it still farther than plus or minus 36 plus minus 1? Yes. In fact, it's still way too big. So now my high, since I know 9 is still way too big for my guess, my high becomes 9. Like that.

And then I make a new guess based on 0 and 9 and the halfway point between there. So 4 and 1/2. So there it is updated. And using this guess, square it and see whether it's less than 36 or greater than 36. It's less than 36. So now this 4.5 becomes my low endpoint. Now I have some information about the low endpoint like that. So I know my final answer is within this little interval right here.

And then I'm just going to go quicker, because now we're dealing with some fractions. My low point becomes 4.5 and now I get the midpoint between 4 and 1/2 and 9 and that's 6.75. And then we keep doing the same process over and over again. Hopefully you get the idea now where we keep changing this.

While the guess squared is still 36, outside of the boundary of 36 plus or minus 1. So if it's less than 35 or greater than 37, keep making guesses. So we're going to go till probably 6 0 something. There. I think that's the end. Yep. So the guess being 6.0469 brings us to a guess squared within plus or minus 1. Yes, question?

STUDENT: What about if your guess was [INAUDIBLE], look through the library?

ANA BELL: If the guess was? If my guess was correct, then we would break immediately. Because this is false. Yeah, we don't even enter the while loop.

OK, so let's run the code. So this is the bisection search code that I just ran through the Python tutor. We looked on the slides. But running with 54,321. So just to recap, the number of guesses we did with the approximation method was 23 million. To give us an answer that said the square root of 23-- the square root of 54,000 is about 233.

And now we run it with our bisection search. And I didn't even have to wait. That took less than a second compared to 20 seconds that we had to wait for. And it didn't fail. It gave us very similar answer. It's this 233.068 is close to the square root of 54,000. And we did 30 guesses. Dramatic pause.

23 million for the approximation method, 20 seconds later, versus 30 guesses less than a second later. So it's not like we went from 23 million to 5 million guesses. We went from the order of millions to just tens, which is really, really cool. That's very impressive. And that's what logarithmic growth means. That's the power of logarithmic growth and recognizing that we can apply bisection search to these problems.

So with approximation method, again, we're decreasing our search space by 0.0001 with each guess. But with the bisection search, we're decreasing our search space by half with each guess. So if we had however many things to search for, in the book we had 400 pages to search through, with our first guess, we now only have 200 pages to search through. With the second guess, we only have 100 pages to search through. With the next guess, we only have 50 pages to search through.

And the idea of bisection search just that it's logarithmic comes from the fact that we have to ask ourselves, how many guesses do we make until we have only, for example, one page left to search through for the money? Or how many guesses do we have to make till we are within epsilon? There's only that one-- we reach the one value that gives us within epsilon. And so this many guesses means that we've divided our search space by 2 to the power of k many times. And that's when we've converged on the answer.

And so to converge on the answer means you've divided your search space by 2 k times. So n divided by 2 to the power of k equals 1. You have reached your one answer. The money is at this page. The student is sitting there. Or we have come within 0.01 of the actual answer.

And so when this is true, n is equal to 2 to the k, and what we want is to solve this problem in terms of n. So k is equal to log of n. And that's where the logarithmic growth comes from for this particular problem.

So in terms of loops, yes, it took us k times through the while loop to figure out the answer. But in terms of the size of our search space, it took us log of n times to get to our answer.

So let's look at a couple of nuances of the code we just wrote. So if we try to run the code for values between 0 and 1, what actually happens? So if we run it with, for example, what's the square root of 0.5. It's running. It's still running. I'm pretty sure it should have given us an answer by now, so let's just stop it. We've entered an infinite loop.

So in that case, let's see what actually it's printing out. So when you've entered an infinite loop, it's time to put some print statements. Best place to put print statements is within the loop itself and just print out some values for things. So here I have this print statement where we print out what-- oops. Let me get that out of the way. What the low value is.

So we've got low equals-- and actually, I don't need to convert this to string. It should just be low. And oops. And then the high value and then the guess itself. Oops. Like that. So if we run it, that's what we get. And it looks like it's just repeating, repeating over and over again.

So what happens when I'm looking for a square root of a value between 0 and 1? So this is my 0 to x. But if x is between 0 and 1, the square root of x, It's bigger than x itself. So the square root of 0.5 is bigger than 0.5. It's not smaller than 0.5. So what this program is doing is it's making its initial guess. High plus low divided by 2. So 0.

If my initial guess is 0 to x, it's making an initial guess there. And then at some point, it just gets stuck in this loop. Because the low becomes 0.5 after our first guess. The high becomes 0.5 as well. And the halfway point between 0.5 and 0.5 is just 0.5. So now it's just reassigning the new guess to itself over and over again.

So we need to make a fix to that, and I'm going to have you guys make the fix to that. So you don't need to account for both cases. But change the endpoints for this particular problem such that it works with values of x between 0 and 1. So if we're trying to find the square root of a decimal number between 0 and 1, what are the endpoints that you want to choose for the code to now work? And the code is exactly the same as before. So all you need to do is choose different endpoints. Yes.

STUDENT: Sorry, I just don't understand why it got stuck, like how the high and low [INAUDIBLE].

ANA BELL: OK, we can run it with the Python tutor. And so if this is 0.5. So basically we've made our guess like that. And then we're changing our guesses. And so you can see that it's actually changing the low and the high. And it originally did the right thing. The first few guesses, it's making the changes appropriately.

But then the floating point errors come into play where at some point, this 0.4999 and this low that it keeps dividing is just going to become 0.5. And 0.5 is a power of 2, remember, as floating points are. And in this particular case, once it reaches the 0.5, then floating point errors don't come into play anymore, because that 0.5 can just be represented exactly. So I'm going to have to probably hit Next for quite a few more times. But you can kind of see where that's getting that 0.5 from. Does that help?

STUDENT: Yeah. So the floating point error just [INAUDIBLE] get the [INAUDIBLE].

ANA BELL: That and also the fact that we didn't really account-- this code doesn't actually work correctly with these values. So it enters an infinite loop because of the floating point error towards the end. And that causes us to see just 0.5, 0.5, 0.5. But if we were doing it to infinite precision, you would start to see numbers that approach 0.5 but never quite get there. But I think our code-- the reason we saw 0.5 here is because it already ran 100 times, 200 times. And so now we're just seeing the tail end of it.

So here is the code for fixing that. So what do you guys think the low endpoint should be and the high endpoint should be if we wanted this to work with values between 0 and 1? So if this is our-- this is our x. And we know x is less than 1, greater than 0. The square root of x is going to be somewhere up here. And we know the maximum place it will be is 1.

And what's a minimum place that the square root of x could be for values within this range? I heard x. So this is the minimum value for the square root of x. And this is the maximum value for the square root of x. So all we need to do is say the low is equal to x and the high is equal to 1. And then I think this code should work. Yeah.

And so I did just that down here. So here is the code with actually allowing for the user to give us any value, not just between 0 and 1 or greater than 1. So all I did here to make the code work and be robust is add an if else right at the beginning. So I allow the user to give me whatever x they'd like.

But then I do a little check here that says if the x is greater or equal to 1, then my low and high end points become 0 to x, because I know the square root is going to be within that boundary. But then otherwise if the user gave me a value that's less than 1, and I guess I should do greater than 0 just in case the user gives me negative numbers, then I would choose the boundary for the low to be x and the high to be 1. So just a very simple if else here. And otherwise the rest of the code works just the same.

Yeah, so this is exactly what we just saw in the slides. An if and an else where I choose the endpoints accordingly. Any questions about this code? Does it make sense? Yeah.

STUDENT: I make the low equal to 0, and it still gives me the same answer for square [INAUDIBLE] of 0.5. Why is that?

ANA BELL: Oh, if you make the low equal to 0 here? I think that's fine. Because then that means you're making your low lower than it needs to be. And so your first guess is basically the halfway point, x itself. And then I think it just fixes it.

STUDENT: So it just goes through one extra guess?

ANA BELL: It goes through one extra guess. Exactly. And that's, again, the power of bisection search. For values greater than 1, if we made our high boundary be 2x, it would just make one extra guess to bring us to x and then below and so on and so on. So one extra guess is nothing to the computer.

So a couple observations with bisections for bisection search. So it takes a significantly less amount of time to solve problems using bisection search than it does using the approximation method. And it gives us an approximation to, in this case, the square root of a number that was just as good as the approximation method itself.

When we did the book example, and that's the second point here. It might be easier to illustrate. When we did the book example, the very first guess eliminated more number of pages than later guesses. Our first guess eliminated 200 pages right off the bat. Our second guess only eliminated 100 pages. Our third only 50. And at some point, you can imagine that we're only eliminating something like four pages. And then we're eliminating only two pages at a time the more and more guesses you make.

So it feels more dramatic at first, but then it kind of dies down. But that's just logarithmic growth. It feels dramatic at first, but then as you get closer and closer to the actual approximation, the actual answer, you're not taking as big of steps or you're not making such dramatic cuts to the book.

And so the bisection search algorithm is really awesome, but again, there are some limitations to when you can use it. You have to have your search space have endpoints. That search space needs to be ordered alphabetically, in order by numeric or whatever. And you have to be able to get the feedback. Is this guess too low or too high? If you don't have those, then you can't use bisection search for this.

I'm going to give you a couple of moments to work on this code by yourself. So this is you writing the bisection search algorithm to find the cube root of positive cubed. So don't worry about negatives or whatever. Just assume the user gives you a positive cube. I'm initializing the values for you here. So the cube is 27.

I want you to be within plus or minus 0.01. So your guess squared should be within plus or minus 0.01 of 27. Start with a low of 0 and a high of cube. And write the rest of the algorithm. Don't copy and paste what we did for square. Try to write it all by yourself all over again. It'll, A, give you practice coding, B, make sure that you understand the actual steps of the algorithm.

You don't need to write it top to bottom. You can write the inside of the while loop first, whatever feels comfortable for you. But as long as you try to write it all by yourself to try to make this coding second nature, I'm all for that. So I'll give you a couple of moments to do that, and then we can write it together. But basically, it's going to be almost the same as what we've been seeing on the slides.

All right, does anyone have a start for me? What do you want to start with? Do you want to do a while loop or a for loop? Let's ask that. A while loop. OK. Let's do while. And what's the condition going to be for the approximation? Yep. Oh, I needed to define a guess. Perfect. What should my guess be? Yes. High plus low over 2. So I have my initial guess.

And then what is happening with my loop? I want to keep doing things as long as-- guess to the third minus cube. Yep, absolute value of guess. Yep. OK. Exactly. We want it to be larger, larger or equal, whatever you'd like, epsilon. So while I'm still too far away.

STUDENT: [INAUDIBLE] write not equal [INAUDIBLE]?

ANA BELL: No, because then we're comparing floats. We want to be farther. Because if it's not equal to, you only stop when it becomes exactly 0.01 away. So we can draw. It's easier if we draw. This is our x and this is epsilon. And our guess cubed if it's equal to, that means g cubed is exactly here, I guess, or exactly here.

STUDENT: Oh, so we [INAUDIBLE].

ANA BELL: Yes, you want to be out of bounds to still be making guesses. Yep. What's our process for making a new guess using bisection search? So we have a guess. And now what do we need to do? We need to decide whether it's too low or too high. That's what the bisection search says.

So guess or guess cubed is too low or too high. Exactly. If the guess cubed-- yep, larger than cube. Then our guess is too high. So I can even make a note for myself here. Guess too high. So if it's too high. I know anything bigger than it I don't want. So I need to set my high endpoint or my low endpoint. Yeah, my high endpoint becomes my guess. I'm resetting my high to be the guess, because I know that guess is too big anyway. Else opposite. My low endpoint is my guess.

Am I done? Nope. OK. What do I need to do? I need to redefine my guess. If I don't redefine my guess, my code has an infinite loop. So my guess is exactly as before. High plus low divided by 2. And then at the end, same indentation level as the while loop. We can just print our guess. Because I know I'm going to break as soon as I become within or equal to epsilon. Yay, that's what we were expecting.

And it's fine that it's 3.000 something. I wouldn't expect it to be exactly 3, even though we as humans know it is 3, because the algorithm says to stop as soon as we came within epsilon. Yes, we can find a better answer if we keep going, but that's not what we asked the code to do. We asked the code to stop as soon as we came within plus or minus epsilon of this.

STUDENT: Matter whether the high goes in the if or the low?

ANA BELL: It does not matter if you put the high in the if or the low. I mean, as long as you're consistent. If it's greater than, you have to reassign the high. If this is less than, you reassign the low.

OK, so we're going to look at one more algorithm to figure out an approximation to the square root of a number. Just to show you that there is something else, yet another thing. And this particular algorithm only works to find roots of a polynomial. So this is a Newton-Raphson algorithm.

And basically we don't need to prove this, but basically they showed that if you have a polynomial of this form, so ax squared plus bx plus c or ax to the power of 4 plus bx cubed plus cx plus d, something like that, if you have a polynomial like that, then you can start with a guess, any guess you'd like.

And you can come up with a better approximation to the square root by saying a new guess. So the new better approximation for the guess is whatever your current guess is minus that polynomial evaluated at the guess. So replace x with your guess divided by the derivative of that polynomial evaluated at the guess. So get the derivative and replace x with your guess.

This should sound familiar, because lecture two, we actually implemented just this part. Remember when we were learning about expressions and combining them together? I mentioned this algorithm and I said, we're not going to be writing the whole algorithm today, but we are going to be implementing the part that makes a new better guess for the square root of a number. Well, today we're actually going to take that line, put a wrapper around it, the wrapper being a little loop that makes successive guesses, better and better guesses using guesses that we have just made to get us close to the approximation for a square root.

So let's start with this. So the idea here for finding the square root of a number is to realize that if we want to find the square root of, let's say, 24, that's essentially us applying this algorithm to the polynomial that says that's x squared minus 24. Because if x squared minus 24 equals 0, then basically x squared is equal to 24.

And to solve for x means that we are looking for the square root of 24. So we can try to apply this Newton-Raphson method to find an approximation to the square root of a number by simply solving using their method to solve applied to this polynomial, x squared minus whatever value you want to find the square root of.

So just to give you a little intuition for how this works is so we have an initial guess. Let's say it's this x1 right here. And you take f of x1. That brings you up here. You find the derivative over here and you follow the tangent line to the x-axis for the next guess. And you repeat the process. Evaluate this guess to get f of that guess. This is the tangent line. Follow it down to the x-axis for a better guess. And you keep doing this until you get as close as you'd like to the square root here.

So just for completeness sake since I did link it, this is what it looks like. That's your initial guess. That's your f. There's your tangent line. That gives you the next guess. Evaluate that. Get your tangent line. Get your next guess. Evaluate that. Get the tangent line. There's your next guess. And it basically works for any polynomial. But we are applying it to just finding the square root of a number, so our polynomial is pretty simple.

So if we want to find the square root of k, the polynomial we're interested in here is x squared minus k. The derivative, I think have you guys done derivatives yet in math? OK, good. The derivative of x squared minus k is just 2x. And then we can initialize our guess to be whatever we'd like. And then all we need to do for a better guess than the one we currently have is to take our current guess minus that guess plugged into the polynomial of interest. So g squared minus k divided by the derivative with the guess plugged in two times g.

And if we repeat this many, many, many times, this will eventually get us to a nice approximation for the square root of the number. And this is the code. It's even simpler than the bisection search code.

So let's say we want to be within plus or minus 0.01 of 24 with our guess. We can start with any guess we'd like, but I guess a reasonable guess is to just take that k, the thing you want to find the square root of, divide by 2. Once again, we can keep track of how many guesses we do. And surprise, the while loop condition for while we keep making guesses is exactly the same as what we've seen before. In approximation method and in bisection search method. As long as we're outside this plus or minus epsilon boundary, keep making guesses, because I'm not happy with my guess.

So here while the absolute value of guess squared minus k, k being the thing we want to find the square root of, is bigger than epsilon. So if we're farther away in both ends, we keep track of how many guesses we've done and make our new guess. So this is what's different than bisection or approximation. The guess is done by the Newton-Raphson method. And this line right here is what we wrote in lecture two or three. Our new guess is our old guess minus the guess evaluated at x. So guess squared minus k divided by the derivative evaluated at guess 2 times guess.

And that's it. The loop takes care of the rest. And it'll keep making new guesses until it comes within plus or minus epsilon. So that's our function. That's f of guess and that's f prime of guess. So let's run it. Here it is.

So we made four guesses to find the square root of 24 is about 4.9, which just pretty good. We came within 0.01. And if we try 5, 4, 3, 2, 1, our favorite number so far in this class, we only did 10 guesses. And it gave us just as good an approximation as bisection search and that ridiculously long approximation method. Yes?

STUDENT: Why is the guess [INAUDIBLE]?

ANA BELL: Why is the guess k over 2? It can be anything you want. We just started with something reasonable that's a function of k. Yeah, it can be 100. It can be whatever you'd want to do. Because the algorithm will work no matter what.

So that's awesome. There's less guesses. But this is a pretty limiting algorithm. You can only use it to find square roots of a particular value. We can't use it, apply this algorithm to finding the person in the middle of the room or something like that. It's really specific to this particular problem.

So a little wrap up before we go on to just introducing the next lecture is we talked about iteration. That was kind of the big thing that we added after conditionals. So finding a way to repeat certain lines of code to do something useful for us. And we looked at guess and check methods. Now, I guess I'm putting all the methods we saw under guess and check, because they're kind of all guess and check. We're guessing a value and we're checking whether that value is exact or within some epsilon of what we want to be.

And all these guess and check methods have the same kind of three things associated with them. There's some sort of loop. There's something that you need to do over and over again. We need some way to generate the guesses. And that's where things are different between the different algorithms. And then we need some way to check that the guess is right or within some epsilon or something like that. And then a way for us to continue making guesses. So we saw exhaustive enumeration, the original guess and check method where we basically had integers or some set values that we wanted to check. It was exhaustive, so we knew exactly how many values we would have to iterate over.

Approximation algorithms allowed us to have smaller increments and we were able to search for approximations to square roots or cube roots or whatever problem we were trying to solve. Bisection search we saw was an improvement over approximation methods, but only for problems that had an ordering property and for problems that you could figure out whether your guesses were too high or too low. If you can't have those, then you can't apply bisection search. So you're stuck with an approximation algorithm or something else.

And then this Newton-Raphson was kind of the last thing we saw. It's very specific algorithm for finding square roots of values, but still valuable in kind of showcasing this looping construct, checking for something, and then making a new guess. This is basically a summary of what I just said also. So we don't need to go over it. But are there any questions about these three algorithms? Do they make sense? Hopefully the coding practice kind of helped a little bit during the lectures. Any questions? No? OK.

So in the last five or so minutes, I want to introduce the motivation for the next topic we're going to talk about. So far we've basically learned how to write a bunch of code. We learned expressions, we learned variables, we learned conditionals, we learned loops, conditionals and loops as a way to add control flow to our program. And we had this nice little toolbox of things to use to write algorithms.

So it is true we have all that you need to know to write interesting algorithms. We wrote these interesting algorithms. But we actually haven't taught you about some important concepts in programming. And these concepts actually exist in all of the modern programming languages. And these ideas are decomposition and abstraction. So I'll just motivate these ideas today. We're not going to look at any code. But I'll show you some simpler version of decomposition abstraction that you've already been kind of doing. And then next lecture, we'll see how we can actually implement these ideas in code.

So the idea of decomposition is that you take a large program and you try to divide it into smaller parts. Each one of these parts will be self-contained. So they won't really interfere with each other, as in the code from one part is not going to influence the code in another part. But they can talk to each other. They can send values to each other back and forth.

So if you take one large spaghetti code program and you try to divide it into these nice self-contained parts, you can have each one of these parts solve a different part, a different portion of your large program. And in the end, they can kind of come together to solve the larger program. That's the idea of decomposition. And the idea of abstraction is once you write these self-contained parts one time and you've done the work, you've done the thought process, you've thought about how to write them in an efficient way, nobody else needs to know exactly how you implemented them.

You want to abstract away all the details that went into figuring out how to solve that problem into just some text or some interface that allows you to say, hey, I solved this problem. All you need to do is give me this input and this input and this input and my code will solve your problem and give you this output back.

Kind of like if you're working in a group project, every one of you goes and does your own part. I don't care if you use the internet or the library to solve your part. All I care is that we all come back together and we put our results together. And so that's the idea of abstraction. There's some unnecessary details that might be in some code. I don't care about those details, how you solved your problem. I just care that you solved the problem. So tell me how I can interact with you.

So this is sort of very low level, I guess, in some ways that we've already been employing the ideas of decomposition and abstraction. So decomposition is the idea that you can write smaller pieces of code that are kind of self-contained. So if I gave you this, I kind of talked about spaghetti code, this is kind of like a simpler version of spaghetti code.

If I gave you this line of code, it's a little bit messy. I've got some value here that I know is going to be important, especially if I define it to some large number of decimal places. I've got these two values here that I'm copying over, basically. This is not great coding style. It's not great coding practice.

But I can kind of take these values and save them and/or decompose them into things that are reusable. So I've got pi here, which is interesting to me. I can save it in a variable. I've got r here, 2.2. I'm saving it as a variable named r that I know I'm going to use in many places. So instead of copying and pasting 2.2 here and here, I might make a mistake if I type it out, I just use the variable.

And so I've decomposed this little bit of spaghetti code into these nice modular pieces. I've got pi as a module, r as a module, and then I'm just putting them together to achieve this common goal, which is to find the area. And we're going to see this on a larger scale using these things called functions next lecture.

Now, the idea of abstraction, again, we've already been kind of doing this. Hopefully you guys have been doing this through comments in your code. So if you spend some time on your problem set when it's first released and you write a whole chunk of code and you do a really good job at it and you did it in a really cool way, come a week later, you forgot some details that you've done right and you didn't comment your code. That could lead you into big trouble, because now you have to figure out what the code is doing.

If you had just written a little bit of comment at the beginning of the code for an interesting way or, hey, I used the bisection search algorithm here or so and so, that would actually suppress a lot of the details from your code. But you would still be able to remember what the code is doing. And so the idea of suppressing details is done through comments. And we're going to see in the next lecture how we can suppress details for larger chunks of code as well. So that's the idea of abstraction here.

So the big idea that we're going to look through in the next lecture is to stop writing large chunks of code where we copy and paste things that do the same thing over and over again, because that could lead to errors being introduced. You change it in one place. You forget to change it in another place. We're going to see how we can write these little modules called functions that you write only once, you debug only once, and then you can use them over and over and over again in your code with different inputs to give you different outputs.

So the idea here is we want to create code that's easy to modify, easy to maintain, and easy to understand. So if you come back to it a week from now or a year from now, you'll still be able to know what that code is doing. So that's the motivation for next lecture. We'll start with a real life example, and then we'll dive right into functions.