PROFESSOR: Good morning everyone. Morning. Let's get started. So the second of two lectures on numerics. Last time we had this motivating question of finding the millionth digit of the square root of 2, or the square root of quantities that end up becoming irrational. And we talked about high-precision arithmetic, and we use Newton's method to compute the square roots.

You saw a demo of computing square roots, but there's a few things missing. We don't quite know how to do division, which is required for the Newton's method, and we didn't really talk at all about algorithmic complexity beyond talking about the complexity of multiplication. So multiplication is a primitive that at this point we know how to do in a couple of different ways, including the naive order n squared algorithm and the Karatsuba algorithm, which is something like n raised to 1.58.

But how many times is multiplication called when you compute square roots? In fact, multiplication is called when you call the division operator when you compute square roots. So there's really two levels off a computation going on here and we need to open this up, and look at in detail, and figure out what our overall algorithmic complexity is. So that's really the meat of today's lecture. Getting to the point where we know what we have with respect to asymptotic complexity of computing the square root of a number.

So let me start with a review of what we covered last time. We decided that we wanted the millionth digit of square root of 2. And the way we're going to do this is by working with integers and computing the floor, since we needed to be an integer, of 2 times 10 raised to 2d, where d is the number of digits of precision. N over there.

So we'll take a look at an example or two here as to how this works with integers.

But what we do is compute essentially the floor of some quantity a, the square root of some quantity a, via Newton's method. And the way Newton's method works is you go through an iteration. You start with x0 being one, which is your initial guess, and compute xi plus 1 equals xi plus a divided by xi over 2. And as you can see, this requires division, because we're computing a divided by xi. That's the outer Newton iteration.

And I said a couple of things that's said you are going to have a quadratic rate of convergence. The precision with respect to the number of digits is going to increase by a factor of 2 every iteration. And so if you started out with one digit of precision, you go to two, then four, eight, et cetera. And so that's a geometric progression. And that means that we're going to have a logarithmic number of iterations, which is nice. And we were all happy about that, and you believed me. I gave you an example and it looked pretty good, but didn't really prove anything about the rate of convergence.

What I'd like to do now is take a look at this particular iterative computation, where we're computing xi plus 1 given xi , and argue that this, in fact, has a quadratic rate of convergence. So you can think of this as doing an error analysis of Newton's method.

And let's say that xn equals square root of a 1 plus epsilon n, where epsilon may be positive or negative. So we have an error associated with xn in the n-th iteration with respect to what we want, which is the square root of a. And it's off by something. It may be a large quantity in the beginning. We want to show convergence, so obviously we want epsilon n, as n becomes large, do tend to 0. How fast does this approach 0? That's the question.

And so if you take this equation and plug this into that, and say, what is xn plus 1? xn plus 1 would be square root of a times 1 plus epsilon n plus a divided by square root of a 1 plus epsilon n divided by 2. Just plugging it in, the value of xn. And then some a couple of steps of algebraic simplification, you can pull out the square root of a here, then you have 1 plus epsilon n, 1 divided by 1 plus epsilon n over here.

The whole thing divided by 2.

And if you keep going-- there's one step that I'm skipping here in terms of simplification, but let me write this last result out. Which is xn plus 1 is square root of a times 1 plus epsilon n squared divided by 2 times 1 plus epsilon n down at the bottom.

So what do we have here in terms of the overall observation for epsilon n plus 1, which is the error in the n plus 1-th iteration given that you have an epsilon n error in the n-th iteration? You have a relationship like so where epsilon n plus 1 is related to epsilon n whole square. And this part here, as n becomes large, epsilon n is going to go to 0 assuming a decent initial guess.

And so you can say that this is essentially 1, which means you have this quadratic rate of convergence where the error, which is a small quantity, is getting squared at every iteration. And so if you have something like a 0.01 error at the beginning for epsilon n, epsilon n squared is going to be 0.0001. So that's where you get the quadratic rate of convergence. So it really comes from this relationship, the relationship epsilon n squared to epsilon n plus 1, Any questions about this?

Great. So if you have the quadratic rate of convergence, if you want to go to d digits of precision like I have here, you can argue that you need to log d iterations. So that's kind of nice, you have a logarithmic number of iterations. I'm going to get back to that. There's one little subtlety that is associated with asymptotic analysis that goes beyond simply the number of iterations that you have and the digits of precision. But so far so good.

We're happy with this logarithmic number of iterations. And if we can now compute the complexity of the division, then obviously we need an algorithm for that. But if you have an algorithm and we figure out what the complexity of the division algorithm is, then we have complexity for the square root of 2 or square root of a using Newton's method.

So just justified what I said last time with respect to quadratic rate of convergence.

And then we talked about multiplication last time. I want to revisit that. You have multiplication algorithms, and we want to be able to multiply d digit numbers. And the naive algorithm. And you could imagine doing divide and conquer. So you take x1, x0; y1, y0 where x1 is the most significant half of x. You're trying to multiply x times y. And same thing for y1 and y0. So each of these will have d by 2, digits of precision. And if you implement the naive algorithm that looks like tn equals 4 tn by 2 plus theta n, you end up with theta n squared complexity out so you have to do four multiplications corresponding to x1 times Y1 x1 times y0, et cetera.

And at each level in the recursive tree, you're breaking things down by a factor of 2 respect to the digits of precision that you need to multiply on as you're going down the tree. And this is the four multiplications. You get your theta n squared complexity. This gentleman by the name of Karatsuba recognized that you could play a few mathematical tricks, which I won't go over again, but reduce to three multiplications. And you do a few more additions, but given that the additions have theta n complexity, the recurrence relationship turns into tn equals 3t of n over 2 plus theta n. And this ends up having 1.58 dot dot dot complexity.

No reason to stop with breaking things up into two parts. You could imagine generalizing Karatsuba and people have done this. Two different researchers, Toom and Cook, generalized Karatsuba for the case where k is greater than or equal to 2, where you're breaking it into k parts. So the Toom-Cook 2 algorithm is basically Karatsuba, but you have Toom 3, Toom 4, and so on. And I'm not going to give you a lot of details on this. We don't expect you to work on this, at least in 6006.

But just to give you a sense of what happens, the Toom 3 method, or the Toom-Cook 3 method, breaks and number up into three parts. So each of these would have d by 3 digits of precision. So this is what you're starting out with. You're starting out with a d digit number. But the very first level of recursion, you're going to break things up into three xi numbers that are d by 3 digits long. Same thing for y. And if you did a naive multiplication of this, how many multiplications do I need? If I just forget about any mathematical tricks, if I just tried to multiply these things out, how many d by 3 by d by 3 multiplications do I need?

AUDIENCE: Nine.

PROFESSOR: Nine. So if you can beat nine using mathematical tricks, you have a better divide and conquer algorithm. And it turns out that Toom 3 plays some arithmetic games and ends up with a recurrence relationship that looks like this. Where you reduce the nine multiplications down to five. So that's a win. And that ends up being theta of n raised to what? Someone? Someone loudly. Log--

AUDIENCE: Base 3.

PROFESSOR: Log with a base 3 of 5. Another irrational number. And this ends up being n raised to 1.465. So you won. If you use Toom 3, assuming the constants worked out-- and Victor can say a little bit more about that, because we're having a little trouble justifying this particular problem set question that we want to give you, given the constant factors involved. So the issue really here is this is correct. It's n raised to 1.46. That's n raised to 1.5. And then the naive algorithm is n square.

But how big does n have to be in order for the n raised to 1.58 algorithm to beat the n square algorithm, and for the n raised to 1.46 algorithm to beat the n raised to 1.58 algorithm, et cetera. And it turns out n needs to be really, really large if you implement these in Python. So if you're having a little trouble here, giving you this pristine problem set that you can go off and learn about multiplication, and also appreciate asymptotic complexity. So that's a bit of a catch-22.

Anyway, for the purposes of theory, this is great. It turns people have done even better. Multiplication is just this obviously incredibly important primitive that you would need for doing any reasonable computation. And so people have worked on using things like fast Fourier transforms and other techniques improve the complexity of multiplication. And best scheme until a few years ago was this scheme called Schonhage-Strassen scheme, which is almost linear in complexity. It's n log n log log n time. And this uses the fast Fourier transform, FFT.

And you can play with all of these things. You can play with Karatsuba the naive algorithm, Toom 3, et cetera in the gmpy package in Python. And you can see as to what the value of n needs to be in order for one of these algorithms to beat the other. This is not something that you're going to do specifically in the problem set, but I say that as an aside. These algorithms are implemented, and they're used in real life. Eric?

ERIC: It may be worth mentioning that Python itself for long integers uses Karatsuba.

PROFESSOR: Yeah, so Python uses-- beyond a certain n, you are going to have decisions that are made within the package. And Python shifts to Karatsuba after n becomes large. But if n is small, then it's going to run the naive algorithm. Now if you write your own multiplication, you can do whatever you want. You can have your own adaptive scheme, assuming you have many of these algorithms implemented, or you're calling them using the gmpy package.

So lastly, this looked pretty good for a while. And from a theoretical standpoint there was a breakthrough. Guy by the name of Furer came up with this algorithm that is n log n-- and let me write this carefully-- 2 raised big O-- that's an upper bound-- of log star n. That makes sense? No. I'll have to explain it.

OK, so what does this mean? This part is clear. This is like sorting. It doesn't need to really use sorting, but that's n log n. And then you have this 2 raised to big O log star n. I need to define what log star n is. And log star n is what's called the iterative algorithm-- logarithm, rather. I guess it's an iterative algorithm, but it computes logs. And the iterative logarithm is the number of times log needs to be applied to get a result that is less than or equal to 1.

So this thing really cuts you down to size really fast. So it doesn't matter. You could be a 10 raised to 24, or 2 raised to 50, let's say, if you were doing binary logs. And in the very first iteration you go down to 50, right? And then you take a log of 50 and you go down to about 7 or something. And then you take the log of 7. And if you're talking about base 2, like we were, you're down to less than 3. And so four or five iterations, you're down to less than or equal to 1. And that's what log star n computes. It's not the logarithm as much as the number of times so you have to apply log to get the result that's less than or equal to 1.

So you have these giant numbers, and it's only like five, six, eight times do you apply log and you're down to one. So for all practical purposes, you can think of-- and this is upper bound-- you can think of this, even though this is 2 raised to something, it's 2 raised to a pretty small number. 2 raised to 10, that would be 1,000. And so from an asymptotic complexity standpoint, this is the winner. From a practical standpoint, Schonhage-Strassen is really what you probably want to use when n becomes very large, to the billions and so on and so forth.

And as of now, to the best of my knowledge this hasn't been implemented in the gmpy package. So if you actually want to use gmpy, this is where you stop.

So that's multiplication. So we have a bunch of different ways that you could do multiplication. What I'd like to do is give you a sense of assuming a given complexity of multiplication, how long would division take? So we are 1 and 1/2 lectures in, and I haven't really told you how we're going to do division, which is what we have to do when we compute a divided by xi, which is the basic integration in the Newton method. So let's get to that.

So finally high-precision division. So we want a high-precision rep off a divided by b. And we're going to compute a high-precision rep off 1 divided by b first. And what we mean by that is that we'll compute r divided by b floor where r is a really large value. And more importantly, it's easy to divide by r in a particular base. So for example, r equals 2 raised to k, when we use base 2, you can easily divide through a shift operator.

So if I give you r divided by b and I give you this long computer word that's in base 2, which typically could have millions of digits in its representation, I can shift that by the appropriate amount to a given r divided by b. I can get 1 over b by shifting that quantity. So it feels like, hey wait a minute. Why are we dividing by r?

Well remember that you want 1 over b. And if you're computing r divided by b floor, and you actually want 1 over b, which then you could use to multiply by a so you can run your Newton iteration, then you want to divide by r. And that division is essentially going to be something that shifts things to the right. So the most significant bits move to the right, and you get a smaller number. That make sense?

So we all know how to divide by using shifting assuming the bases work out right. And if you had a representation that was decimal, suddenly you could certainly divide by 10 raised to k. That's easy. You've done this many times. But you just changed the decimal point when you're working with decimal arithmetic. When you divide 72 by 100 and you get 0.72. And that's a very similar notion here. It doesn't really matter what base you're talking about.

So that's the setup. That's how are we going to try and tackle this division problem. But we still have this problem of computing r divided by b. So how are we going to compute r divided by b? And we want this to be a large number of digits of precision. So we're going to use Newton's method again. You've got some non-linearity here with respect to 1 over x. And we're gonna use Newton's method again. And we'll have to hope that this works out, that we can get Newton's method, it'll converge, and it'll require operations that we know how to do. And all of this is going to work out really well.

I'm going to set up a function, f of x equals 1 divided by x minus b divided by r. So what this means is that this function has a 0 at x equals r divided by b. So if I try and find the 0 of this function, and I start out with a decent initial guess, I'm going to end up with r divided by b. And if I'm working with integers, effectively that's the floor that I have for r divided by b. And then I do my shift and I end up with 1 over b.

So someone who remembers differentiation, if you're gonna apply Newton's method, tell me what the derivative of f of x is. Somebody's stretching at the back, but I don't think that was an answer. Someone at the back? Too easy a question? For the cushion.

AUDIENCE: 1 over negative x squared.

PROFESSOR: 1 over negative x squared. Who's that? All right. You can come pick this up.

Whatever. Cut the monotony here. Just veered to the left. I think next time I'm going to weight them or something. Let's just do frisbees next time. Let's just do frisbees next time. It makes it easy. Forget cushions. No? Frisbees or cushions? How many want frisbees? How many want cushions? Frisbees win.

So you got derivative of x is minus 1 divided by x squared. And then if you go off and apply Newton's method-- and I'm not going to go through the symbolic equations here associated with Newton's method-- but that's basically the same as we did before. You are computing a tangent, and the new value of xi plus 1 given the value of xi is the x-intercept. And we needed the derivative to compute that.

But bottom line, you have xi plus 1 equals xi minus f of xi divided by f prime of xi. So that's the Newton iteration. And it's worth plugging in the various values here. 1 divided by xi minus b divided by r. That's f of x on top divided by minus 1 divided by xi square. So that's the derivative over here. So all I'm doing is plugging things in. But you want to visualize this because this is really what we need to compute. And we have xi plus 1 equals xi plus xi square times 1 over xi minus b divided by r. And finally I get 2xi minus b xi square divided by r. That is key. This is pretty important.

So let's us look all the way to the left, which is xi plus 1, all the way to the right, 2 times xi. That doesn't scare us, 2 times something. Especially base 2, pretty easy. That's a multiply. Multiplies don't scare us because we know how to do multiplies anyway. This is a simple multiply. And then I got a square here. Square. Not a square root. Squares don't scare us because that's a multiply, just multiplying the same number to itself. And this doesn't scare us because we know that we've chosen r to be an easy division.

So all of the operations here are either easy, or they require a multiply. So remember I'm going to put a picture up towards the end here that tells you the overall structure for computing square root of a or square root of 2. But we've just sort of sold out to Newton, if you will. Because we said that we're going to use Newton's method to compute essentially, iteratively, square root of a. And within the Newton method, the first iteration, if you will, of the Newton method, we had to compute a reciprocal. We had to compute 1 over xi. And in order to compute 1 over xi, we're going to apply Newton's method again like I showed over here and over there.

And so that division is going to require iteration. But the iteration at the second level is one of multiplication. You're gonna repeatedly apply multiplication because you're going to go xi plus 1 based on xi using multiplication and some easy operations. And then you go xi plus 2, xi plus 3, and so on and so forth. That make sense? I'll try and put this up to give you the complete picture once we're done talking about the division algorithm and its complexity.

But before I do that, I just want to give you a sense of the convergence of this scheme. Again, I want to give you an example first, and then I'll argue about the convergence. You have to run this iteratively. You've got to make i to get to the point where it's large enough that you have your digits of precision. And just as an example, let's say we want r divided by b equals 2 raised to 16 divided by 5. So this is a fairly straightforward example. But when you get up to integers, it turns out it's evocative.

So r was selected to be 2 raised to k to make for easy division. And what I really want is that. And I want to see how I get to that using Newton's method. And our initial guess, let's say we try 2 raised to 16 divided by 4, because we know how to divide by a power of two. And so that's 2 raised to 14. And that's our initial guess. So think of that as being x0. That is x0. And that 16384. x1 is going to be 2 times 16384, which is exactly that, minus 5 times 16384 whole square.

So now you're starting to square a fairly big number. And obviously if you'd started with an even bigger r, this would be a large number. You go 65536 equals-- and this is 12288. So you really have one digit of precision there. But the next time around, you get 2 times 12288 minus 5 times 12288 square divided by 65536. And this division is easy. It's a shift. You get to 13056. And I won't write this whole thing out, but if you take that, the next thing you'll get is 13107.

So as you can see, there's rapid convergence here. And you can actually do a very similar analysis to the epsilon analysis-- and I'll put it in the notes, but I won't do it here-- that I did for the square root iteration to show that you have a quadratic the rate of convergence when you apply Newton's method to division as well.

So you can prove that using the symbolic analysis than we did very similar to the epsilon n relationship to epsilon n plus 1. I'd suggest that it's a difference equation here so that analysis is not exactly the same. But you can run through that, and you can read that in the notes.

So we're in business. Finally things are looking up with respect to being able to actually implement this in practice. I want to talk about complexity. And I promise that there was a subtlety associated with the complexity of division in relation to multiplication, but let me just go over and write down what I just told you with respect to the number of iterations that division requires.

So division, quadratic convergence. So number of digits doubles at each step. Good news. So d digits of precision, log d iterations. Now let's say that we have a particular algorithm for multiplication that I'm just going to say, since we have so many different algorithms, I'm going to say multiplication in theta n raised to alpha time, where alpha is greater than or equal to 1. I just want to be general about it.

And so assuming that I have a multiplication algorithm, that can run in theta n raised to alpha, where clearly you know alpha can be 1.46 for Toom 3, et cetera. And it's not quite that for Schonhage-Strassen, but I just want to be working with one particular complexity. So I'll parameterize it in this fashion. And everything I say is going to be true for Schonhage-Strassen and Furer as well.

But first, easy question. What is the complexity of division using the analysis that I've put on the board so far? n digit numbers it's going to be? I wanna hear from you. How many hard multipliers do I have? Log of?

AUDIENCE: n.

PROFESSOR: Log of n, right? It wasn't a hard question. So division would be theta log n times n raised to alpha. Everybody buy that? No? Ask a question if you're confused. Maybe

I should say everybody buy that? How many people agree with that? Big O? How many people agree with that? Yeah, that's right. Big O. I'm hedging my bets here. I'm just saying big O. I could say big O of n cubed and you should all agree with me. Or big O of whatever. You had a question?

AUDIENCE: What's the longest [INAUDIBLE] number of [INAUDIBLE] we need to get a certain level of [INAUDIBLE]?

PROFESSOR: That's right. So if you want d digits of precision, then according to this argument-- and I think you guys are a little doubtful here because I kept talking about subtleties, and in fact there's a subtlety here, which I want to get to-- but this big O thing is perfectly correct. But to answer your question, yes. Let's assume that it's n digits of precision. That's what we assume whether it's n or d. You can plug in the appropriate symbol here.

And we're saying that, look, every iteration is bounded by n raised to alpha complexity for the multiply. And I'm going to do a logarithmic number of iterations. So I end up getting log n times n raised to alpha. So that is correct, in fact. Big O is correct.

So now it comes to the interesting question, which is can you do a better analysis? So this sort of hearkens back to three weeks ago, maybe you've forgotten. Maybe you've blanked it out of your memory, but I thought I described to you build max-heap. And we had this straightforward analysis of build max-heap that was n log n complexity. And then we looked at it a little more carefully, and we started adding things up much more carefully. We turned into bank accountants. And then we decided that it was theta n complexity. People remember that? Right?

So I want you to turn into bank accountants again, and then tell me first, there's a nice observation that you can make here that we haven't made yet with respect to the size of these numbers. We know what we want to eventually, but there's a nice observation we can make it with respect to the size of these numbers. And then we want to exploit that observation to do a better analysis of the theta complexity of division.

So who wants to tell me what the observation is. This is definitely worth a cushion. What's the observation? I want to end up with d digits of precision. If I give you another hint, I'm gonna give it away. Someone tell me.

This is a dynamic process, OK? So what do I start with? What do I start with? If I want to compute something and you want to use Newton's method, what do you start with? Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: You start with one digit of precision. That's fantastic. I don't know if you already have a cushion or not, but here's the second one. So you start with a small number of digits of precision. And then you end up with a large million, whatever, number, which is your d. So what does that mean?

So now somebody take that and run with it. Somebody take that and run with it. You already have a cushion. Like many? You guys, usual suspects. So someone take that and run with it. What can I do now? What does it mean if I start with a small number of digits of precision? My initial guess was one, right? I mean, that had one digit of precision. And then the number of digits doubles with each step. So is there any reason why I'm doing, if I had d digits of precision, eventually that I'll have to do d digit multiplies in each iteration? Any reason why? Yeah.

AUDIENCE: You don't have to, because [INAUDIBLE] multiplies are going to be trivial. And [INAUDIBLE] then you're going to eventually approach the d to the alpha iteration.

PROFESSOR: That's exactly right. Exactly right. That's worth a cushion. But now I want you or someone else, tell me what the iteration looks like. So this is the key observation. The key observation is that if I want d digits of precision, I'm going to start with maybe one digit of precision. So this is d of p, or dig of p, not to be confused. I start with 1, 2, 4, and I end up with d. And our claim was that this was log d iterations, right? So the initial multiplies are easy. Initially you're doing constant work if you have really small numbers associated with these multiplies. It's only towards the end that you end up doing a lot more work, right?

[illegible]

So someone tell me if I have n raised to alpha, and if I say I want to write an equation. And I don't want to use theta here. I'm going to use constants because I want to add up constants, and it's a little iffy then you add up thetas. You need to be looking at constants. Now I can imagine that for this iteration, the very first one, that I have something like c times 1 raised to alpha, because it's just a single digit of precision. OK And the next one, I'm using the same algorithm. This is c times 2 raised to alpha, c times 4 raised to alpha.

And then out here I'm going to have c times d by 4 raised to alpha plus c times d by 2 raised to alpha plus finally c times d raised to alpha. And someone give me a bound. Who wants to give me a bound on this? Who wants to give me a bound on this? Less than or equal to. Let's just make it less than. What? Someone? Just plug in a value of alpha. And remember your convergent geometric series and things like that. What is that? Someone? Yeah.

AUDIENCE: Just some constant times d to the alpha?

PROFESSOR: That's exactly right. Just some constant times d to the alpha. And in fact, you can say, it's 2c d to the alpha. Keep a question for you aside. So that' sit. That's the little careful analysis that we had to do, which basically without changing your code, really, suddenly gave you a better complexity. Isn't that fun? That's always fun. You had this neat algorithm to begin with. And bottom line is you're just computing things a little more accurately, than essentially saying that you had to do all of this work with large number of digits of precision at every iteration. The number of digits actually increases.

So what does this mean? I guess ultimately, the complexity of division is now what? It's the same as the complexity of multiplication, right? So regardless of whether we did a Newton iteration or not, the complexity of division. You are doing a logarithmic number of iterations, but since eventually all of the work is going to get done at the end here. Most of the work is getting done at the end when you have these long numbers. That's basically the essence of the argument.

So let me finish up and talk about the complexity of computing square roots. And as you can imagine, even though you have two nested Newton iterations here, you can make basically the same argument. So let's recall what we're doing in terms of computing square roots. We want to compute square root of a. And we said, well we don't quite know how to do this. We're going to end up doing 10 raised to 2d times a, and we're going to run Newton's method on it. So you've got one level of Newton's method. And the iteration here with respect to Newton's method is something like xi plus 1 equals xi plus a divided by xi.

Now every time you do that for a particular xi, you're going to end up having to call a division. So you're going to call a division here, and then you're going to call a division here. For each iteration you have to call a division. And what we're saying is, well we're going to end up having to call for each of these division methods we're going to call Newton's method. And what that is something like 2xi minus b xi square divided by r. And that's going to be a bunch of multiplications.

And what we argued up until this point was that the complexity of the division, even though we had a bunch of iterations here, a logarithmic number of iterations, the complexity of the division was the same as the complexity of the multiplication because the numbers started out small and grew big. All right? Everybody buy that?

I'm going to use exactly the same argument for this level of iteration as well. And again, when you start out with the digits of precision corresponding to square root of 2, you're going to start out guessing 1.5, which is your initial guess for the square root of 2, and it's going to be a small number of digits of precision. And eventually you'll get to a million digits. So using essentially the same equation summing, you can argue that the complexity of computing square roots is the complexity of division, which of course is the complexity of multiplication.

And that's the story. So obviously the code would be a little more complicated than a multiplication code, because you have all this control structure outside of it. It's really two nested loops. The multiply is getting called a bunch of times to do the divide, and the divide is getting called a bunch of times to compute the square root.

But ultimately, because the numbers are growing and you start out with small numbers, most of the work is done when you get to the millions of digits of precision. And you basically have theta n raised to alpha complexity for computing square roots. If you have n raised to alpha multiply, and you want n digits of precision. All right? See you next time. Stick around for questions.