PROFESSOR ERIC GRIMSON: Last time, we ended up, we sort of did this tag team thing, Professor Guttag did the first half, I did the second half of the lecture, and the second half of the lecture, we started talking about complexity. Efficiency. Orders of growth. And that's what we're going to spend today on, is talking about that topic. I'm going to use it to build over the next couple of lectures.

I want to remind you that we were talking at a fairly high level about complexity. We're going to get down into the weeds in a second here. But the things we were trying to stress were that it's an important design decision, when you are coming up with a piece of code, as to what kind of efficiency your code has.

And the second thing that we talked about is this idea that we want you to in fact learn how to relate a choice you make about a piece of code to what the efficiency is going to be. So in fact, over the next thirty or forty minutes, we're going to show you a set of examples of sort of canonical algorithms, and the different classes of complexity.

Because one of the things that you want to do as a good designer is to basically map a new problem into a known domain. You want to take a new problem and say, what does this most look like? What is the class of algorithm that's-- that probably applies to this, and how do I pull something out of that, if you like, a briefcase of possible algorithms to solve?

All right, having said that, let's do some examples. I'm going to show you a sequence of algorithms, they're mostly simple algorithms, that's OK. But I want you to take away from this how we reason about the complexity of these algorithms. And I'll remind you, we said we're going to mostly talk about time. We're going to be counting the number of basic steps it takes to solve the problem.

So here's the first example I want to do. I'm going to write a function to compute integer power exponents. a to the b where b is an integer. And I'm going to do it only using multiplication and addition and some simple tests. All right? And yeah, I know it comes built in, that's OK, what we want to do is use it as an example to look at it.

So I'm going to build something that's going to do iterative exponentiation. OK? And in fact, if you look at the code up here, and it's on your handout, the very first one, x 1, right here-- if I could ask you to look at it-- is a piece of code to do it. And I'm less interested in the code than how we're going to analyze it, but let's look at it for a second.

All right, you can see that this little piece of code, it's got a loop in there, and what's it doing? It's basically cycling through the loop, multiplying by a each time. So first time through the loop, the answer is 1. Second time it-- sorry, as it enters the loop, at the time it enter-- exits, the answer is a. Next time through the loop it goes to a squared. Next time through the loop it goes to a cubed. And it's just gathering together the multiplications while counting down the exponent. And you can see it when we get down to the end test here, we're going to pop out of there and we're going to return the answer.

I could run it, it'll do the right thing. What I want to think about though, is, how much time does this take? How many steps does it take for this function to run? Well, you can kind of look at it, right? The key part of that is that WHILE loop.

And what are the steps I want to count? They're inside that loop-- I've got the wrong glasses so I'm going to have to squint-- and we've got one test which is a comparison, we've got another test which is a multiplication-- sorry, not a test, we've got another step which is a multiplication-- and another step that is a subtraction. So each time through the loop, I'm doing three steps. Three basic operations.

How many times do I go through the loop? Somebody help me out. Hand up? Sorry. b times. You're right. Because I keep counting down each time around-- mostly I've got to unload this candy, which is driving me nuts, so-- thank you. b times. So I've got to go 3 b steps. All right, I've got to go through the loop b times, I've got three steps each time, and then when I pop out of the loop, I've got two more steps. All right, I've got the initiation of answer and the return of it. So I take 2 plus 3 b steps to go through this loop. OK. So if b is 300, it takes 902 steps. b is 3000, it takes 9002 steps. b is 30,000 you get the point, it takes 90,002 steps.

OK. So the point here is, first of all, I can count these things, but the second thing you can see is, as the size of the problems get larger, that additive constant, that 2, really doesn't matter. All right? The difference between 90,000 steps and 90,002 steps, who cares about the 2, right? So, and typically, we're not going to worry about those additive constants.

The second one is, this multiplicative constant here is 3, in some sense also isn't all that crucial. Does it really matter to you whether your code is going to take 300 years or 900 years to run? Problem is, how big is that number? So we're going to typically also not worry about the multiplicative constants. This factor here.

What we really want to worry about is, as the size of the problem gets larger, how does this thing grow? How does the cost go up? And so what we're going to primarily talk about as a consequence is the rate of growth as the size of the problem grows. If it was, how much bigger does this get as I make the problem bigger? And what that really says is, that we're going to use this using something we're going to just call asymptotic notation-- I love spelling this word-- meaning, as in the limit as the size of the problem gets bigger, how do I characterize this growth? All

right?

You'll find out, if you go on to some of the other classes in course 6, there are a lot of different ways that you can measure this. The most common one, and the one we're going to use, is what's often called big Oh notation. This isn't big Oh as in, oh my God I'm shocked the markets are collapsing, This is called big Oh because we use the Greek letter, capital letter, omicron to represent it.

And the way we're going to do this, or what this represents, let me write this carefully for you, big Oh notation is basically going to be an upper limit to the growth of a function as the input grow-- as the input gets large.

Now we're going to see a bunch of examples, and I know those are words, let me give you an example. I would write f of x is in big Oh of n squared. And what does it say? It says that function, f of x, is bounded above, there's an upper limit on it, that this grows no faster than quadratic in n, n squared.

OK. And first of all, you say, wait a minute, x and n? Well, one of the things we're going to see is x is the input to this particular problem, n is a measure of the size of x. And we're going to talk about how we come up with that. n measures the size of x.

OK. In this example I'd use b. All right, as b get-- b is the thing that's changing as I go along here, but it could be things like, how many elements are there in a list if the input is a list, could be how many digits are there in a string if the input's a string, it could be the size of the integer as we go along. All right.? And what we want to do then, is we want to basically come up with, how do we characterize the growth-- God bless you-- of this problem in terms of this quadra-- sorry, terms of this exponential growth

Now, one last piece of math. I could cheat. I said I just want an upper bound. I could get a really big upper bound, this thing grows exponentially. That doesn't help me much. Usually what I want to talk about is what's the smallest size class in which this function grows? With all of that, what that says, is that this we would write is order b. That algorithm is linear. You can see it. I've said the product was is 2 plus 3 b. As I make b really large, how does this thing grow? It grows as b. The 3 doesn't matter, it's just a constant, it's growing linearly. Another way of saying it is, if I, for example, increase the size of the input by 10, the amount of time increases by 10. And that's a sign that it's linear.

OK. So there's one quick example. Let's look at another example. If you look at x 2, this one right here in your handout. OK. This is another way of doing exponentiation, but this one's a recursive function. All right?

So again, let's look at it. What does it say to do? Well, it's basically saying a similar thing. It says, if I am in the base case, if b is equal to 1, the answer is just a. I could have used if b is equal to 0, the answer is 1, that would

have also worked.

Otherwise, what do I say? I say, ah, I'm in a nice recursive way, a to the b is the same as a times a to the b minus 1. And I've just reduced that problem to a simpler version of the same problem. OK, and you can see that this thing ought to unwrap, it's going to keep extending out those multiplications until gets down to the base case, going to collapse them all together.

OK. Now I want to know what's the order of growth here? What's the complexity of this? Well, gee. It looks like it's pretty straightforward, right? I've got one test there, and then I've just got one thing to do here, which has got a subtraction and a multiplication.

Oh, but how do I know how long it takes to do x 2? All right, we were counting basic steps. We don't know how long it takes to do x 2. So I'm going to show you a little trick for figuring that out. And in particular, I'm going to cheat slightly, I'm going to use a little bit of abusive mathematics, but I'm going to show you a trick to figure it out.

In the case of a recursive exponentiator, I'm going to do the following trick. I'm going to let t of b be the number of steps it takes to solve the problem of size b. OK, and I can figure this out. I've got one test, I've got a subtraction, I've got a multiplication, that's three steps, plus whatever number of steps it takes to solve a problem of size b minus 1.

All right, this is what's called a recurrence relation, there are actually cool ways to solve them. We can kind of eyeball it. In particular, how would I write an expression for t of b minus 1? Well the same way. This is 3 plus 3 plus t of b minus 2. Right? I'm using exactly the same form to reduce this. You know, you can see what's going to happen. If I reduce that, it would be 3 plus t of b minus 3, so in general, this is 3 k plus t of b minus k. OK. I'm just expanding it out.

When am I done? How do I stop this? Any suggestions? Don't you hate it when professors ask questions? Yeah. Actually, I think I want b minus k equal to 1. Right? When this gets down to t of 1, I'm in the base case. So I'm done when b minus k equals 1, or k equals b minus 1. Right, that gets me down to the base case, I'm solving a problem with size 1, and in that case, I've got two more operations to do, so I plug this all back in, I-- t of b is I'm going to put k for b minus 1 I get 3 b minus 1 plus t of 1, so t of 1 is 2, so this is 3 b minus 1 plus 2, or 3 b minus 1.

OK. A whole lot of work to basically say, again, order b is linear. But that's also nice, it lets you see how the recursive thing is simply unwrapping but the complexity in terms of the amount of time it takes is going to be the same. I owe you a candy. Thank you.

OK. At this point, if we stop, you'll think all algorithms are linear. This is really boring. But they're not. OK? So let me show you another way I could do exponentiation. Taking an advantage of a trick. I want to solve a to the b.

Here's another way I could do that. OK. If b is even, then a to the b is the same as a squared all to the b over 2. All right, just move the 2's around. It's the same thing. You're saying, OK, so what? Well gee, notice. This is a primitive operation. That's a primitive operation. But in one step, I've reduced this problem in half. I didn't just make it one smaller, I made it a half smaller. That's a nice deal.

OK. But I'm not always going to have b as even. If b is odd, what do I do? Well, go back to what I did before. Multiply a by a to the b minus 1. You know, that's nice, right? Because if b was odd, then b minus one is even, which means on the next step, I can cut the problem in half again.

OK? All right. x 3, as you can see right here, does exactly that. OK? You can take a quick look at it, even with the wrong glasses on, it says if a-- sorry, b is equal to 1, I'm just going to return a. Otherwise there's that funky little test. I'll do the remainder multiplied by 2, because these are integers, that gives me back an integer, I just check to see if it's equal to b, that tells me whether it's even or odd. And in the even case, I'd square, divide by half, call this again: in the odd case, I go b minus 1 and then multiply by a.

I'll let you chase it through, it does work. What I want to look at is, what's the order of growth here?

This is a little different, right? It's going to take a little bit more work, so let's see if we can do it. In the b even case, again I'm going to let t of b be the number of steps I want to go through. And we can kind of eyeball this thing, right? If b is even, I've got a test to see if b is equal to 1, and then I've got to do the remainder, the multiplication, and the test, I'm up to four. And then in the even case, I've got to do a square and the divide. So I've got six steps, plus whatever it takes to solve the problem size b over 2, right? Because that's the recursive call. b as odd, well I can go through the same kind of thing. I've got the same first four steps, I've got a check to see is it 1, I got a check to see if it's even, and then in the odd case, I've got to subtract 1 from b, that's a fifth step, I've got to go off and solve the recursive problem, and then I'm going to do one more multiplication, so it's 6 plus, in this case, t of b minus 1. Because it's now solving a one-smaller problem.

On the next step though, this, we get substituted by that. Right, on the next step, I'm back in the even case, it's going to take six more steps, plus t of b minus 1. Oops, sorry about that, over 2. Because b minus 1 is now even. Don't sweat the details here, I just want you to see the reason it goes through it.

What I now have, though, is a nice thing. It says, in either case, in general, t of b-- and this is where I'm going to abuse notation a little bit-- but I can basically bound it by t, 12 steps plus t of b over 2. And the abuse is, you know, it's not quite right, it depends upon whether it's all ready, but you can see in either case, after 12 steps, 2 runs through this and down to a problem size b over 2.

Why's that nice? Well, that then says after another 12 steps, we're down to a problem with size t of b over 4. And if I pull it out one more level, it's 12 plus 12 plus t of b over 8, which in general is going to be, after k steps, 12 k because I'll have 12 of those to add up, plus t of b over 2 to the k.

When am I done? When do I get down to the base case? Somebody help me out. What am I looking for? Yeah. You're jumping slightly ahead of me, but basically, I'm done when this is equal to 1, right? Because I get down to the base case, so I'm done when b u is over 2 to the k is equal to 1, and you're absolutely right, that's when k is log base 2 of b.

You're sitting a long ways back, I have no idea if I'll make it this far or not. Thank you.

OK. There's some constants in there, but this is order log b. Logarithmic. This matters. This matters a lot. And I'm going to show you an example in a second, just to drive this home, but notice the characteristics. In the first two cases, the problem reduced by 1 at each step. Whether it was recursive or iterative. That's a sign that it's probably linear. This case, I reduced the size of the problem in half. It's a good sign that this is logarithmic, and I'm going to come back in a second to why logs are a great thing.

Let me show you one more class, though, about-- sorry, let me show you two more classes of algorithms. Let's look at the next one g-- and there's a bug in your handout, it should be g of n and m, I apologize for that, I changed it partway through and didn't catch it.

OK. Order of growth here. Anybody want to volunteer a guess? Other than the TAs, who know? OK. Let's think it through. I've got two loops. All right? We already saw with one of the loops, you know, it looked like it might be linear, depending on what's inside of it, but let's think about this. I got two loops with g. What's g do? I've got an initialization of x, and then I say, for i in the range, so that's basically from 0 up to n minus 1, what do I do? Well, inside of there, I've got another loop, for j in the range from 0 up to m minus 1.

What's the complexity of that inner loop? Sorry? OK. You're doing the whole thing for me. What's the complexity just of this inner loop here? Just this piece. How many times do I go through that loop? m. Right? I'm going to get back to your answer in a second, because you're heading in the right direction. The inner loop, this part here, I do m times. There's one step inside of it. Right? How many times do I go through that loop? Ah, n times, because for each value of i, I'm going to do that m thing, so that is, close to what you said, right? The order complexity here, if I actually write it, would be-- sorry, order n times m, and if m was equal to n, that would be order n squared, and this is quadratic. And that's a different behavior.

OK. What am I doing? Building up examples of algorithms. Again, I want you to start seeing how to map the characteristics of the code-- the characteristics of the algorithm, let's not call it the code-- to the complexity. I'm

going to come back to that in a second with that, but I need to do one more example, and I've got to use my high-tech really expensive props. Right. So here's the fourth or fifth, whatever we're up to, I guess fifth example.

This is an example of a problem called Towers of Hanoi. Anybody heard about this problem? A few tentative hands.

OK. Here's the story as I am told it. There's a temple in the middle of Hanoi. In that temple, there are three very large diamond-encrusted posts, and on those posts are sixty-four disks, all of a different size. And they're, you know, covered with jewels and all sorts of other really neat stuff. There are a set of priests in that temple, and their task is to move the entire stack of sixty-four disks from one post to a second post. When they do this, you know, the universe ends or they solve the financial crisis in Washington or something like that actually good happens, right? Boy, none of you have 401k's, you're not even wincing at that thing.

All right. The rules, though, are, they can only move one disk at a time, and they can never cover up a smaller disk with a larger disk. OK. Otherwise you'd just move the whole darn stack, OK? So we want to solve that problem. We want to write a piece of code that helps these guys out, so I'm going to show you an example. Let's see if we can figure out how to do this.

So, we'll start with the easy one. Moving a disk of size 1. OK, that's not so bad. Moving a stack of size 2, if I want to go there, I need to put this one temporarily over here so I can move the bottom one before I move it over. Moving a stack of size 3, again, if I want to go over there, I need to make sure I can put the spare one over here before I move the bottom one, I can't cover up any of the smaller ones with the larger one, but I can get it there. Stack of size 4, again I'm going there, so I'm going to do this initially, no I'm not, I'm going to start again. I'm going to go there initially, so I can move this over here, so I can get the base part of that over there, I want to put that one there before I put this over here, finally I get to the point where I can move the bottom one over, now I've got to be really careful to make sure that I don't cover up the bottom one in the wrong way before I get to the stage where I wish they were posts and there you go. All right? [APPLAUSE] I mean, I can make money at Harvard Square doing this stuff, right?

All right, you ready to do five? Got the solution? Not so easy to see. All right, but this is actually a great one of those educational moments. This is a great example to think recursively. If I wanted to think about this problem recursively-- what do I mean by thinking recursively? How do I reduce this to a smaller-size problem in the same instant?

And so, if I do that, this now becomes really easy. If I want to move this stack here, I'm going to take a stack of size n minus 1, move it to the spare spot, now I can move the base disk over, and then I'm going to move that stack of size n minus 1 to there. That's literally what I did, OK?

So there's the code. Called towers. I'm just going to have you-- let you take a look at it. I'm giving it an argument, which is the size of the stack, and then just labels for the three posts. A from, a to, and a spare. And in fact, if we look at this-- let me just pop it over to the other side-- OK, I can move a tower, I'll say of size 2, from, to, and spare, and that was what I did. And if I want to move towers, let's say, size 5, from, to, and spare, there are the instructions for how to move it. We ain't going to do sixty-four. OK.

All right. So it's fun, and I got a little bit of applause out of it, which is always nice for me, but I also showed you how to think about it recursively. Once you hear that description, it's easy to write the code, in fact. This is a place where the recursive version of it is much easier to think about than the iterative one.

But what I really want to talk about is, what's the order of growth here? What's the complexity of this algorithm? And again, I'm going to do it with a little bit of abusive notation, and it's a little more complicated, but we can kind of look at. All right?

Given the code up there, if I want to move a tower of size n, what do I have to do? I've got to test to see if I'm in the base case, and if I'm not, then I need to move a tower of size n minus 1, I need to move a tower of size 1, and I need to move a second-- sorry about that-- a second tower of size n minus 1.

OK. t of 1 I can also reduce. In the case of a tower of size 1, basically there are two things to do, right? I've got to do the test, and then I just do the move. So the general formula is that.

Now. You might look at that and say, well that's just a lot like what we had over here. Right? We had some additive constant plus a simpler version of the same problem reduced in size by 1. But that two matters. So let's look at it. How do I rea-- replace the expression FOR t of n minus 1? Substitute it in again. t of n minus 1 is 3 plus 2 t of n minus 2. So this is 3, plus 2 times 3, plus 4 t minus 2. OK. And if I substitute it again, I get 3 plus 2 times 3 plus 4 times 3 plus 8 t n minus 3.

This is going by a little fast. I'm just substituting in. I'm going to skip some steps. But basically if I do this, I end up with 3 times 1 plus 2 plus 4 to 2 to the k minus 1 for all of those terms, plus 2-- I want to do this right, 2 to the k, sorry-- t of n minus k.

OK. Don't sweat the details, I'm just expanding it out. What I want you to see is, because I've got two versions of that problem. The next time down I've got four versions. Next time down I've got eight versions. And in fact, if I substitute, I can solve for this, I'm done when this is equal to 1. If you substitute it all in, you get basically order 2 to the n.

Exponential. That's a problem. Now, it's also the case that this is fundamentally what class this algorithm falls into,

it is going to take exponential amount of time. But it grows pretty rapidly, as n goes up, and I'm going to show you an example in a second. Again, what I want you to see is, notice the characteristic of that. That this recursive call had two sub-problems of a smaller size, not one. And that makes a big difference.

So just to show you how big a difference it makes, let's run a couple of numbers. Let's suppose n is 1000, and we're running at nanosecond speed. We have seen log, linear, quadratic, and exponential. So, again, there could be constants in here, but just to give you a sense of this. If I'm running at nanosecond speed, n, the size of the problem, whatever it is, is 1000, and I've got a log algorithm, it takes 10 nanoseconds to complete. If you blink, you miss it. If I'm running a linear algorithm, it'll take one microsecond to complete. If I'm running a quadratic algorithm, it'll take one millisecond to complete. And if I'm running an exponential algorithm, any guesses? I hope Washington doesn't take this long to fix my 401k plan. All right? 10 to the 284 years. As Emeril would say, pow! That's a some spicy whatever.

All right. Bad jokes aside, what's the point? You see, these classes have really different performance. Now this is a little misleading. These are all really fast, so just to give you another set of examples, I'm not going to do the-- If I had a problem where the log one took ten milliseconds, then the linear one would take a second, the quadratic one would take 16 minutes. So you can see, even the quadratic ones can blow up in a hurry.

And this goes back to the point I tried to make last time. Yes, the computers are really fast. But the problems can grow much faster than you can get a performance boost out of the computer. And you really, wherever possible, want to avoid that exponential algorithm, because that's really deadly. Yes.

All right. The question is, is there a point where it'll quit. Yeah, when the power goes out, or-- so let me not answer it quite so facetiously. We'd be mostly talking about time.

In fact, if I ran one of these things, it would just keep crunching away. It will probably quit at some point because of space issues, unless I'm writing an algorithm that is using no additional space. Right. Those things are going to stack up, and eventually it's going to run out of space. And that's more likely to happen, but, you know. The algorithm doesn't know that it's going to take this long to compute, it's just busy crunching away, trying to see if it can make it happen. OK. Good question, thank you.

All right. I want to do one more extended example here., because we've got another piece to do, but I want to capture this, because it's important, so let me again try and say it the following way. I want you to recognize classes of algorithms and match what you see in the performance of the algorithm to the complexity of that algorithm. All right?

Linear algorithms tend to be things where, at one pass-through, you reduce the problem by a constant amount, by

one. If you reduce it by two, it's going to be the same thing. Where you go from problem of size n to a problem of size n minus 1.

A log algorithm typically is one where you cut the size of the problem down by some multiplicative factor. You reduce it in half. You reduce it in third. All right?

Quadratic algorithms tend to have this-- I was about to say additive, wrong term-- but doubly-nested, triply-nested things are likely to be quadratic or cubic algorithms, all right, because you know-- let me not confuse things-- double-loop quadratic algorithm, because you're doing one set of things and you're doing it some other number of times, and that's a typical signal that that's what you have there.

OK. And then the exponentials, as you saw is when typically I reduce the problem of one size into two or more sub-problems of a smaller size. And you can imagine this gets complex and there's lots of interesting things to do to look to the real form, but those are the things that you should see.

Now. Two other things, before we do this last example. One is, I'll remind you, what we're interested in is asymptotic growth. How does this thing grow as I make the problem size big?

And I'll also remind you, and we're going to see this in the next example, we talked about looking at the worst case behavior. In these cases there's no best case worst case, it's just doing one computation. We're going to see an example of that in a second. What we really want to worry about, what's the worst case that happens.

And the third thing I want you to keep in mind is, remember these are orders of growth. It is certainly possible, for example, that a quadratic algorithm could run faster than a linear algorithm. It depends on what the input is, it depends on, you know, what the particular cases are. So it is not the case that, on every input, a linear algorithm is always going to be better than a quadratic algorithm. It is just in general that's going to hold true, and that's what I want you to see.

OK. I want to do one last example. I'm going to take a little bit more time on it, because it's going to both reinforce these ideas, but it's also going to show us how we have to think about what's a primitive step., and in a particular, how do data structures interact with this analysis? Here I've just been running integers, it's pretty simple, but if I have a data structure, I'm going to have to worry about that a little bit more.

So let's look at that. And the example I want to look at is, suppose I want to search a list that I know is sorted, to see if an element's in the list. OK? So the example I'm going to do, I'm going to search a sorted list. All right. If you flip to the second side of your handout, you'll see that I have a piece of code there, that does this-- let me, ah, I didn't want to do that, let me back up slightly-- this is the algorithm called search. And let's take a look at it. OK?

Basic idea, before I even look at the code, is pretty simple. If I've got a list that is sorted, in let's call it, just in increasing order, and I haven't said what's in the list, could be numbers, could be other things, for now, we're going to just assume they're integers. The easy thing to do would be the following: start at the front end of the list, check the first element. If it's the thing I'm looking for, I'm done. It's there. If not, move on to the next element. And keep doing that. But if, at any point, I get to a place in the list where the thing I'm looking for is smaller than the element in the list, I know everything else in the rest of the list has to be bigger than that, I don't have to bother looking anymore. It says the element's not there. I can just stop.

OK. So that's what this piece of code does here. Right.? I'm going to set up a variable to say, what's the answer I want to return, is it there or not. Initially it's got that funny value none. I'm going to set up an index, which is going to tell me where to look, starting at the first part of the list, right? And then, when I got-- I'm also going to count how many comparisons I do, just so I can see how much work I do here, and then notice what it does. It says while the index is smaller than the size of the list, I'm not at the end of the list, and I don't have an answer yet, check. So I'm going to check to see if-- really can't read that thing, let me do it this way-- right, I'm going to increase the number of compares, and I'm going to check to say, is the thing I'm looking for at the i'th spot in the list? Right, so s of i saying, given the list, look at the i'th element, is it the same thing?

If it is, OK. Set the answer to true. Which means, next time through the loop, that's going to pop out and return an answer. If it's not, then check to see, is it smaller than that element in the current spot of the list? And if that's true, it says again, everything else in the list has to be bigger than this, thing can't possibly be in the list, I'm taking advantage of the ordering, I can set the answer to false, change i to go to the next one, and next time through the loop, I'm going to pop out and print it out. OK?

Right. Order of growth here. What do you think? Even with these glasses on, I can see no hands up, any suggestions? Somebody help me out. What do you think the order of growth is here? I've got a list, walk you through it an element at a time, do I look at each element of the list more than once? Don't think so, right? So, what does this suggest? Sorry?

Constant. Ooh, constant says, no matter what the length of the list is, I'm going to take the same amount of time. And I don't think that's true, right? If I have a list ten times longer, it's going to take me more time, so-- not a bad guess, I'm still reward you, thank you.

Somebody else. Yeah. Linear. Why? You're right, by the way, but why? Yeah. All right, so the answer was it's linear, which is absolutely right. Although for a reason we're going to come back in a second. Oh, thank you, I hope your friends help you out with that, thank you.

Right? You can see that this ought to be linear, because what am I doing? I'm walking down the list. So one of the

things I didn't say, it's sort of implicit here, is what is the thing I measuring the size of the problem in? What's the size of the list? And if I'm walking down the list, this is probably order of the length of the list s, because I'm looking at each element once.

Now you might say, wait a minute. Thing's ordered, if I stop part way through and I throw away half the list, doesn't that help me? And the answer is yes, but it doesn't change the complexity. Because what did we say? We're measuring the worst case. The worst case here is, the things not in the list, in which case I've got to go all the way through the list to get to the end.

OK. Now, having said that, and I've actually got a subtlety I'm going to come back to in a second, there ought to be a better way to do this. OK? And here's the better way to think about.

I'll just draw out sort of a funny representation of a list. These are sort of the cells, if you like, in memory that are holding the elements of the list. What we've been saying is, I start here and look. If it's there, I'm done. If not, I go there. If it's there, I'm done, if not, I keep walking down, and I only stop when I get to a place where the element I'm looking for is smaller than the value in the list., in which case I know the rest of this is too big and I can stop. But I still have to go through the list.

There's a better way to think about this, and in fact Professor Guttag has already hinted at this in the last couple of lectures. The better way to think about this is, suppose, rather than starting at the beginning, I just grabbed some spot at random, like this one. And I look at that value. If it's the value I'm looking for, boy, I ought to go to Vegas, I'm really lucky. And I'm done, right?

If not, what could I do? Well, I could look at the value here, and compare it to the value I'm trying to find, and say the following; if the value I'm looking for is bigger than this value, where do I need to look? Just here. All right? Can't possibly be there, because I know this thing is over.

On the other hand, if the value I'm looking for here-- sorry, the value I'm looking for is smaller than the value I see here, I just need to look here. All right?

Having done that, I could do the same thing, so I suppose I take this branch, I can pick a spot like, say, this one, and look there. Because there, I'm done, if not, I'm either looking here or there. And I keep cutting the problem down.

OK. Now, having said that, where should I pick to look in this list? I'm sorry? Halfway. Why? You're right, but why? Yeah. So the answer, in case you didn't hear it, was, again, if I'm a gambling person, I could start like a way down here. All right? If I'm gambling, I'm saying, gee, if I'm really lucky, it'll be only on this side, and I've got a little bit of

work to do, but if I'm unlucky, I'm scrawed, the past pluperfect of screwed, OK., or a Boston fish. I'll look at the rest of that big chunk of the list, and that's a pain. So halfway is the right thing to do, because at each step, I'm guaranteed to throw away at least half the list. Right? And that's nice.

OK. What would you guess the order of growth here is? Yeah. Why? Good. Exactly. Right? Again, if you didn't hear it, the answer was it's log. Because I'm cutting down the problem in half at each time. You're right, but there's something we have to do to add to that, and that's the last thing I want to pick up on. OK.

Let's look at the code-- actually, let's test this out first before we do it. So I've added, as Professor Guttag did-- ah, should have said it this way, let's write the code for it first, sorry about that-- OK, I'm going to write a little thing called b search. I'm going to call it down here with search, which is simply going to call it, and then print an answer out.

In binary search-- ah, there's that wonderful phrase, this is called a version of binary search, just like you saw bin-- or bi-section methods, when we were doing numerical things-- in binary search, I need to keep track of the starting point and the ending point of the list I'm looking at. Initially, it's the beginning and the end of it. And when I do this test, what I want to do, is say I'm going to pick the middle spot, and depending on the test, if I know it's in the upper half, I'm going to set my start at the mid point and the end stays the same, if it's in the front half I'm going to keep the front the same and I'm going to change the endpoint. And you can see that in this code here.

Right? What does it say to do? It says, well I'm going to print out first and last, just so you can see it, and then I say, gee, if last minus first is less than 2, that is, if there's no more than two elements left in the list, then I can just check those two elements, and return the answer. Otherwise, we find the midpoint, and notice what it does. First, it's pointing to the beginning of the list, which initially might be down here at 0 but after a while, might be part way through. And to that, I simply add a halfway point, and then I check. If it's at that point, I'm done, if not, if it's greater than the value I'm looking for, I either take one half or the other.

OK. You can see that thing is cutting down the problem in half each time, which is good, but there's one more thing I need to deal with. So let's step through this with a little more care. And I keep saying, before we do it, let's just actually try it out. So I'm going to go over here, and I'm going to type test search-- I can type-- and if you look at your handout, it's just a sequence of tests that I'm going to do.

OK. So initially, I'm going to set up the list to be the first million integers. Yeah, it's kind of simple, but it gives me an ordered list of these things, And let's run it. OK. So I'm first going to look for something that's not in the list, I'm going to see, is minus 1 in this list, so it's going to be at the far end, and if I do that in the basic case, bam. Done. All right? The basic, that primary search, because it looks at the first element, says it's smaller than everything else, I'm done.

If I look in the binary case, takes a little longer. Notice the printout here. The printout is simply telling me, what are the ranges of the search. And you can see it wrapping its way down, cutting in half at each time until it gets there, but it takes a while to find. All right. Let's search to see though now if a million is in this list, or 10 million, whichever way I did this, it must be a million, right?

In the basic case, oh, took a little while. Right, in the binary case, bam. In fact, it took the same number of steps as it did in the other case, because each time I'm cutting it down by a half. OK. That's nice.

Now, let's do the following; if you look right here, I'm going to set this now to-- I'm going to change my range to 10 million, I'm going to first say, gee, is a million in there, using the basic search. It is. Now, I'm going to say, is 10 million in this, using the basic search. We may test your hypothesis, about how long does it take, if I time this really well, I ought to be able to end when it finds it, which should be right about now. That was pure luck. But notice how much longer it took. On the other hand, watch what happens with binary. Is the partway one there? Yeah. Is the last one there? Wow. I think it took one more step. Man, that's exactly what logs should do, right? I make the problem ten times bigger, it takes one more step to do it. Whereas in the linear case, I make it ten times bigger, it takes ten times longer to run. OK.

So I keep saying I've got one thing hanging, it's the last thing I want to do, but I wanted you see how much of a difference this makes. But let's look a little more carefully at the code for binary search-- for search 1. What's the complexity of search 1? Well, you might say it's constant, right? It's only got two things to do, except what it really says is, that the complexity of search 1 is the same as the complexity of b search, because that's the call it's doing. So let's look at b search. All right? We've got the code for b search up there.

First step, constant, right? Nothing to do. Second step, hm. That also looks constant, you think? Oh but wait a minute. I'm accessing s. I'm accessing a list. How long does it take for me to get the nth element of a list? That might not be a primitive step. And in fact, it depends on how I store a list.

So, for example, in this case, I had lists that I knew were made out of integers. As a consequence, I have a list of ints. I might know, for example, that it takes four memory chunks to represent one int, just for example. And to find the i'th element, I'm simply going to take the starting point, that point at the beginning of memory where the list is, plus 4 times i, that would tell me how many units over to go, and that's the memory location I want to look for the i'th element of the list.

And remember, we said we're going to assume a random access model, which says, as long as I know the location, it takes a constant amount of time to get to that point. So if the-- if I knew the lists were made of just integers, it'd be really easy to figure it out. Another way of saying it is, this takes constant amount of time to figure

out where to look, it takes constant amount of time to get there, so in fact I could treat indexing into a list as being a basic operation.

But we know lists can be composed of anything. Could be ints, could be floats, could be a combination of things, some ints, some floats, some lists, some strings, some lists of lists, whatever. And in that case, in general lists, I need to figure out what's the access time.

And here I've got a choice. OK, one of the ways I could do would be the following. I could have a pointer to the beginning of the list where the first element here is the actual value, and this would point to the next element in the list. Or another way of saying it is, the first part of the cell could be some encoding of how many cells do I need to have to store the value, and then I've got some way of telling me where to get the next element of the list. And this would point to value, and this would point off someplace in memory.

Here's the problem with that technique, and by the way, a number of programming languages use this, including Lisp. The problem with that technique, while it's very general, is how long does it take me to find the i'th element of the list? Oh fudge knuckle. OK. I've got to go to the first place, figure out how far over to skip, go to the next place, figure out how far over to skip, eventually I'll be out the door. I've got to count my way down, which means that the access would be linear in the length of the list to find the i'th element of the list, and that's going to increase the complexity.

There's an alternative, which is the last point I want to make, which is instead what I could do, I should have said these things are called linked lists, we'll come back to those, another way to do it, is to have the start of the list be at some point in memory, and to have each one of the successive cells in memory point off to the actual value. Which may take up some arbitrary amount of memory. In that case, I'm back to this problem. And as a consequence, access time in the list is constant, which is what I want.

Now, to my knowledge, most implementations of Python use this way of storing lists, whereas Lisp and Scheme do not.

The message I'm trying to get to here, because I'm running you right up against time, is I have to be careful about what's a primitive step. With this, if I can assume that accessing the i'th element of a list is constant, then you can't see that the rest of that analysis looks just like the log analysis I did before, and each step, no matter which branch I'm taking, I'm cutting the problem down in half. And as a consequence, it is log. And the last piece of this, is as said, I have to make sure I know what my primitive elements are, in terms of operations.

Summary: I want you to recognize different classes of algorithms. I'm not going to repeat them. We've seen log, we've seen linear, we've seen quadratic, we've seen exponential. One of the things you should begin to do, is to

recognize what identifies those classes of algorithms, so you can map your problems into those ranges.

And with that, good luck on Thursday.