

The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MITOpenCourseWare@ocw.mit.edu.

PROFESSOR: OK, folks. Welcome back. Hope you had a nice long weekend with no classes. You got caught up on all those problem sets that have been sneaking up on you. You enjoyed watching the Patriots and Tom Brady come back. Oh, sorry, I'm showing my local bias.

Before we talk about today's topic, I want to take a second to set the stage. And I want you to stop and think about what you've seen so far in this course. We're coming up on the end of the first section of the course. And you've already seen a lot. You've certainly learned about fundamentals of computation. You've seen different kinds of data structures, both mutable and immutable, so tuples and lists, dictionaries, different ways of pulling things together.

You've seen a range of algorithms from simple linear code to loops, fors and whiles. You've seen iterative algorithms. You've seen recursive algorithms. You've seen classes of algorithms. Divide and conquer. Greedy algorithms. Bisection search. A range of things. And then most recently, you start pulling things together with classes-- a way to group together data that belongs together along with methods or procedures that are designed to manipulate that data.

So you've had actually a fairly good coverage already of a lot of the fundamentals of computation. And you're starting to get geared up to be able to tackle a pretty interesting range of problems. Today and Monday, we're going to take a little bit of a different look at computation. Because now that you've got the tools to start building up your own personal armamentarium of tools, we'd like to ask a couple of important questions. The primary one of which is how efficient are my algorithms?

And by efficiency, we'll see it refers both to space and time, but primarily to time. And we'd like to know both how fast are my algorithms going to run and how could I reason about past performance. And that's what we're going to do with today's topics. We're going to talk about orders of growth. We'll define what that means in a few minutes. We're going to talk about what's called the big O notation. And we're going to begin to explore different classes of

algorithms.

Before we do that though, let's talk about why. And I want to suggest to you there are two reasons this is important to be considering. First question is how could we reason about an algorithm something you write in order to predict how much time is it going to need to solve a problem of a particular size? I might be testing my code on small scale examples.

And I want to know if I'd run it on a really big one, how long is it going to take? Can I predict that? Can I make guesses about how much time I'm going to need to solve this problem? Especially if it's in a real world circumstance where time is going to be crucial.

Equally important is going the other direction. We want you to begin to reason about the algorithms you write to be able to say how do certain choices in a design of an algorithm influence how much time it's going to take. If I choose to do this recursively, is that going to be different than iteratively? If I choose to do this with a particular kind of structure in my algorithm, what does that say about the amount of time I'm going to need?

And you're going to see there's a nice association between classes of algorithms and the interior structure of them. And in particular, we want to ask some fundamental questions. Are there fundamental limits to how much time it's going to take to solve a particular problem, no matter what kind of algorithm I design around this? And we'll see that there are some nice challenges about that.

So that's what we're going to do over the next two days. Before we do though, let's maybe ask the obvious question-- why should we care? Could be on a quiz, might matter to you. Better choice is because it actually makes a difference. And I say that because it may not be as obvious to you as it was in an earlier generation.

So people with my gray hair or what's left of my gray hair like to tell stories. I'll make it short. But I started programming 41 years ago-- no, sorry, 45 years ago-- on punch cards. You don't know what those are unless you've been to a museum on a machine that filled a half a room and that took about five minutes to execute what you can do in a fraction of a second on your phone. Right. This is to tell you're living in a great time, not independent of what's going to happen on November 8th.

All right. We'll stay away from those topics as well, won't we? My point is yeah, I tell old stories. I'm an old guy. But you might argue look, computers are getting so much faster. Does it really

matter? And I want to say to you-- maybe it's obvious to you-- yes, absolutely it does. Because in conjunction with us getting faster computers, we're increasing the sizes of the problems. The data sets we want to analyze are getting massive.

And I'll give you an example. I just pulled this off of Google, of course. In 2014-- I don't have more recent numbers-- Google served-- I think I have that number right-- 30 trillion pages on the web. It's either 30 trillionaire or 30 quadrillion. I can't count that many zeros there. It covered 100 million gigabytes of data. And I suggest to you if you want to find a piece of information on the web, can you write a simple little search algorithm that's going to sequentially go through all the pages and find anything in any reasonable amount of time? Probably not. Right? It's just growing way too fast.

This, by the way, is of course, why Google makes a lot of money off of their map reduced algorithm for searching the web, written by the way, or co-written by an MIT grad and the parent of a current MIT student. So there's a nice hook in there, not that Google pays MIT royalties for that wonderful thing, by the way. All right. Bad jokes aside, searching Google-- ton of time. Searching a genomics data set-- ton of time. The data sets are growing so fast.

You're working for the US government. You want to track terrorists using image surveillance from around the world, growing incredibly rapidly. Pick a problem. The data sets grow so quickly that even if the computers speed up, you still need to think about how to come up with efficient ways to solve those problems. So I want to suggest to you while sometimes simple solutions are great, they are the easy ones to rate-- too write. Sorry.

At times, you need to be more sophisticated. Therefore, we want to reason about how do we measure efficiency and how do we relate algorithm design choices to the cost that's going to be associated with it? OK. Even when we do that, we've got a choice to make. Because we could talk about both efficiency in terms of time or in terms of space, meaning how much storage do I have inside the computer?

And the reason that's relevant is there's actually in many cases a trade-off between those two. And you've actually seen an example, which you may or may not remember. You may recall when we introduced dictionaries, I showed you a variation where you could compute Fibonacci using the dictionary to keep track of intermediate values. And we'll see in next week that it actually tremendously reduces the time complexity.

That's called a trade-off, in the sense that sometimes I can pre-compute portions of the

answer, store them away, so that when I've tried to a bigger version of the answer I can just look up those portions. So there's going to be a trade-off here. We're going to focus, for purposes of this lecture and the next one, on time efficiency. How much time is it going to take our algorithms to solve a problem?

OK. What are the challenges in doing that before we look at the actual tools? And in fact, this is going to lead into the tools. The first one is even if I've decided on an algorithm, there are lots of ways to implement that. A while loop and a for loop might have slightly different behavior. I could choose to do it with temporary variables or using direct substitution. There's lots of little choices. So an algorithm could be implemented many different ways. How do I measure the actual efficiency of the algorithm?

Second one is I might, for a given problem, have different choices of algorithm. A recursive solution versus an iterative one. Using divide and conquer versus straightforward search. We're going to see some examples of that. So I've got to somehow separate those pieces out. And in particular, I'd like to separate out the choice of implementation from the choice of algorithm. I want to measure how hard is the algorithm, not can I come up with a slightly more efficient way of coming up with an implementation.

So here are three ways I might do it. And we're going to look at each one of them very briefly. The obvious one is we could be scientists-- time it. Write the code, run a bunch of test case, run a timer, use that to try and come up with a way of estimating efficiency. We'll see some challenges with that.

Slightly more abstractly, we could count operations. We could say here are the set of fundamental operations-- mathematical operations, comparisons, setting values, retrieving values. And simply say how many of those operations do I use in my algorithm as a function of the size of the input? And that could be used to give us a sense of efficiency.

We're going to see both of those are flawed somewhat more in the first case than the second one. And so we're going to abstract that second one to a more abstract notion of something we call an order of growth. And I'll come back to that in a couple of minutes. This is the one we're going to focus on. It's one that computer scientists use. It leads to what we call complexity classes. So order of growth or big O notation is a way of abstractly describing the behavior of an algorithm, and especially the equivalences of different algorithms.

But let's look at those. Timing. Python provides a timer for you. You could import the time

module. And then you can call, as you can see right down here. I might have defined a really simple little function-- convert Celsius to Fahrenheit. And in particular, I could invoke the clock method from the time module. And what that does is it gives me a number as the number of some fractions of a second currently there.

Having done that I could call the function. And then I could call the clock again, and take the difference to tell me how much time it took to execute this. It's going to be a tiny amount of time. And then I could certainly print out some statistics. I could do that over a large number of runs-- different sizes of the input-- and come up with a sense of how much time does it take.

Here's the problem with that. Not a bad idea. But again, my goal is to evaluate algorithms. Do different algorithms have different amounts of time associated with them? The good news is that if I measure running time, it will certainly vary as the algorithm changes. Just what I want to measure. Sorry.

But one of the problems is that it will also vary as a function of the implementation. Right? If I use a loop that's got a couple of more steps inside of it in one algorithm than another, it's going to change the time. And I don't really care about that difference. So I'm confounding or conflating implementation influence on time with algorithm influence on time. Not so good.

Worse, timing will depend on the computer. My Mac here is pretty old. Well, at least for computer versions. It's about five years old. I'm sure some of you have much more recent Macs or other kinds of machines. Your speeds may be different from mine. That's not going to help me in trying to measure this.

And even if I could measure it on small sized problems, it doesn't necessarily predict what happens when I go to a really large sized problems, because of issues like the time it takes to get things out of memory and bring them back in to the computer. So what it says is that timing does vary based on what I'd like to measure, but it varies on a lot of other factors. And it's really not all that valuable.

OK. Got rid of the first one. Let's abstract that. By abstract, I'm going to make the following assumption. I'm going to identify a set of primitive operations. Kind of get to say what they are, but the obvious one is to say what does the machine do for me automatically. That would be things like arithmetic or mathematical operations, multiplication, division, subtraction, comparisons, something equal to another thing, something greater than, something less than,

assignments, set a name to a value, and retrieval from memory. I'm going to assume that all of these operations take about the same amount of time inside my machine.

Nice thing here is then it doesn't matter which machine I'm using. I'm measuring how long does the algorithm take by counting how many operations of this type are done inside of the algorithm. And I'm going to use that count to come up with a number of operations executed as a function of the size of the input. And if I'm lucky, that'll give me a sense of what's the efficiency of the algorithm.

So this one's pretty boring. It's got three steps. Right? A multiplication, a division, and an addition-- four, if you count the return. But if I had a little thing here that added up the integers from 0 up to x , I've got a little loop inside here. And I could count operations.

So in this case, it's just, as I said, three operations. Here, I've got one operation. I'm doing an assignment. And then inside here, in essence, there's one operation to set i to a value from that iterator. Initially, it's going to be 0. And then it's going to be 1. And you get the idea.

And here, that's actually two operations. It's nice Python shorthand. But what is that operation? It says take the value of $total$ and the value of i , add them together-- it's one operation-- and then set that value, or rather, set the name $total$ to that new value. So a second operation. So you can see in here, I've got three operations. And what else do I have?

Well, I'm going to go through this loop x times. Right? I do it for i equals 0. And therefore, i equal 1, and so on. So I'm going to run through that loop x times. And if I put that together, I get a nice little expression. $1 + 3x$. Actually, I probably cheated here. I shouldn't say cheated. I probably should have counted the return as one more operation, so that would be $1 + 3x + 1$, or $3x + 2$ operations.

Why should you care? It's a little closer to what I'd like. Because now I've got an expression that tells me something about how much time is this going to take as I change the size of the problem. If x is equal to 10, it's going to take me 32 operations. If x is equal to 100, 302 operations. If x is equal to 1,000, 3,002 operations. And if I wanted the actual time, I'd just multiply that by whatever that constant amount of time is for each operation. I've got a good estimate of that.

Sounds pretty good. Not quite what we want, but it's close. So if I was counting operations, what could I say about it? First of all, it certainly depends on the algorithm. That's great.

Number of operations is going to directly relate to the algorithm I'm trying to measure, which is what I'm after.

Unfortunately, it still depends a little bit on the implementation. Let me show you what I mean by that by backing up for a second. Suppose I were to change this for loop to a while loop. I'll set i equal to 0 outside of the loop. And then while i is less than x plus 1, I'll do the things inside of that. That would actually add one more operation inside the loop, because I both have to set the value of i and I have to test the value of i , as well as doing the other operations down here.

And so rather than getting $3x$ plus 1, I would get $4x$ plus 1. Eh. As the government says, what's the difference between three and four when you're talking about really big numbers? Problem is in terms of counting, it does depend. And I want to get rid of that in a second, so it still depends a little bit on the implementation. I remind you, I wanted to measure impact of the algorithm.

But the other good news is the count is independent of which computer I run on. As long as all my computers come with the same set of basic operations, I don't care what the time of my computer is versus yours to do those operations on measuring how much time it would take. And I should say, by the way, one of the reasons I want to do it is last to know is it going to take 37.42 femtoseconds or not, but rather to say if this algorithm has a particular behavior, if I double the size of the input, does that double the amount of time I need? Does that quadruple the amount of time I need? Does it increase it by a factor of 10?

And here, what matters isn't the speed of the computer. It's the number of operations. The last one I'm not going to really worry about. But we'd have to really think about what are the operations we want to count.

I made an assumption that the amount of time it takes to retrieve something from memory is the same as the amount of time it takes to do a numerical computation. That may not be accurate. But this one could probably be dealt with by just agreeing on what are the common operations and then doing the measurement. So this is closer. Excuse me.

And certainly, that count varies for different inputs. And we can use it to come up with a relationship between the inputs and the count. And for the most part, it reflects the algorithm, not the implementation. But it's still got that last piece left there, so I need to get rid of the last piece.

So what can we say here? Timing and counting do evaluate or reflect implementations? I don't want that. Timing also evaluates the machines. What I want to do is just evaluate the algorithm. And especially, I want to understand how does it scale?

I'm going to say what I said a few minutes ago again. If I were to take an algorithm, and I say I know what its complexity is, my question is if I double the size of the input, what does that say to the speed? Because that's going to tell me something about the algorithm. I want to say what happens when I scale it? And in particular, I want to relate that to the size of the input.

So here's what we're going to do. We're going to introduce orders of growth. It's a wonderful tool in computer science. And what we're going to focus on is that idea of counting operations. But we're not going to worry about small variations, whether it's three or four steps inside of the loop. We're going to show that that doesn't matter. And if you think about my statement of does it double in terms of size or speed or not-- or I'm sorry-- time or not, whether it goes from three to six or four to eight, it's still a doubling. So I don't care about those pieces inside.

I'm going to focus on what happens when the size of the problem gets arbitrarily large. I don't care about counting things from 0 up to x when x is 10 or 20. What happens when it's a million? 100 million? What's the asymptotic behavior of this? And I want to relate that time needed against the size of the input, so I can make that comparison I suggested.

OK. So to do that, we've got to do a couple of things. We have to decide what are we going to measure? And then we have to think about how do we count without worrying about implementation details. So we're going to express efficiency in terms of size of input. And usually, this is going to be obvious. If I've got a procedure that takes one argument that's an integer, the size of the integer is the thing I'm going to measure things in. If I double the size of that integer, what happens to the computation?

If I'm computing something over a list, typically the length of the list is going to be the thing I'm going to use to characterize the size of the problem. If I've got-- and we'll see this in a second-- a function that takes more than one argument, I get to decide what's the parameter I want to use. If I'm searching to see is this element in that list, typically, I'm going to worry about what's the size of the list, not what's the size of the element. But we have to specify what is that we're measuring. And we're going to see examples of that in just a second.

OK. So now, we start thinking about that sounds great. Certainly fun computing something

numeric. Sum of integers from 0 up to x . It's kind of obvious x is the size of my problem. How many steps does it take? I can count that. But in some cases, the amount of time the code takes is going to depend on the input.

So let's take this little piece of code here. And I do hope by now, even though we flash up code, you're already beginning to recognize what does it do. Not the least of which, by the clever name that we chose. But this is obviously just a little function. It runs through a loop-- sorry, a for loop that takes i for each element in a list L . And it's checking to see if i is equal to the element e I've provided. And when it is, I'm going to return true. If I get all the way through the loop and I didn't find it, I'm going to return false. It's just saying is e in my input list L ?

How many steps is this going to take? Well, we can certainly count the number of steps in the loop. Right? We've got a set i . We've got to compare i and potentially we've got to return. So there's at most three steps inside the loop. But depends on how lucky I'm feeling. Right?

If e happens to be the first element in the list-- it goes through the loop once-- I'm done. Great. I'm not always that lucky. If e is not in the list, then it will go through this entire loop until it gets all the way through the elements of L before saying false. So this-- sort of a best case scenario. This is the worst case scenario.

Again, if I'm assigned and say well, let's run some trials. Let's do a bunch of examples and see how many steps does it go through. And that would be the average case. On average, I'm likely to look at half the elements in the list before I find it. Right? If I'm lucky, it's early on. If I'm not so lucky, it's later on.

Which one do I use? Well, we're going to focus on this one. Because that gives you an upper bound on the amount of time it's going to take. What happens in the worst case scenario? We will find at times it's valuable to look at the average case to give us a rough sense of what's going to happen on average. But usually, when we talk about complexity, we're going to focus on the worst case behavior.

So to say it in a little bit different way, let's go back to my example. Suppose you gave it a list L of some length. Length of L , you can call that len if you like. Then my best case would be the minimum running time. And in this case, it will be for the first element in the list. And notice in that case, the number of steps I take would be independent of the length of L . That's great. It doesn't matter how long the list is. If I'm always going to find the first element, I'm done.

The average case would be the average over the number of steps I take, depending on the length of the list. It's going to grow linearly with the length of the list. It's a good practical measure. But the one I want to focus on will be the worst case. And here, the amount of time as we're going to see in a couple of slides, is linear in the size of the problem. Meaning if I double the length of the list in the worst case, it's going to take me twice as much time to find that it's not there. If I increase the length in the list by a factor of 10, in the worst case, it's going to take me 10 times as much time as it did in the earlier case to find out that the problem's not there. And that linear relationship is what I want to capture.

So I'm going to focus on that. What's the worst case behavior? And we're about ready to start talking about orders of growth, but here then is what orders of growth are going to provide for me. I want to evaluate efficiency, particularly when the input is very large. What happens when I really scale this up? I want to express the growth of the program's runtime as that input grows. Not the exact runtime, but that notion of if I doubled it, how much longer does it take? What's the relationship between increasing the size of the input and the increase in the amount of time it takes to solve it?

We're going to put an upper bound on that growth. And if you haven't seen this in math, it basically says I want to come up with a description that is at least as big as-- sorry-- as big as or bigger than the actual amount of time it's going to take. And I'm going to not worry about being precise. We're going to talk about the order of rather than exact. I don't need to know to the femtosecond how long this is going to take, or to exactly one operation how long this is going to take.

But I want to say things like this is going to grow linearly. I double the size of the input, it doubles the amount of time. Or this is going to grow quadratically. I double the size of the input, it's going to take four times as much time to solve it. Or if I'm really lucky, this is going to have constant growth. No matter how I change the input, it's not going to take any more time.

To do that, we're going to look at the largest factors in the runtime. Which piece of the program takes the most time? And so in orders of growth, we are going to look for as tight as possible an upper bound on the growth as a function of the size of the input in the worst case. Nice long definition. Almost ready to look at some examples. So here's the notation we're going to use. It's called Big O notation. I have to admit-- and John's not here today to remind me the history-- I think it comes because we used Omicron-- God knows why. Sounds like something from Futurama. But we used Omicron as our symbol to define this.

I'm having such good luck with bad jokes today. You're not even wincing when I throw those things out. But that's OK. It's called Big O notation. We're going to use it. We're going to describe the rules of it. Is this the tradition of it? It describes the worst case, because it's often the bottleneck we're after. And as we said, it's going to express the growth of the program relative to the input size.

OK. Let's see how we go from counting operations to getting to orders of growth. Then we're going to define some examples of ordered growth. And we're going to start looking at algorithms. Here's a piece of code you've seen before. Again, hopefully, you recognize or can see fairly quickly what it's doing. Computing factorials the iterative way. Basically, remember n factorial is n times $n - 1$ times $n - 2$ all the way down to 1. Hopefully, assuming n is a non-negative integer.

Here, we're going to set up an internal variable called `answer`. And then we're just going to run over a loop. As long as n is bigger than 1, we're going to multiply `answer` by n , store it back into `answer`, decrease n by 1. We'll keep doing that until we get out of the loop. And we're going to return `answer`. We'll start by counting steps. And that's, by the way, just to remind you that in fact, there are two steps here.

So what do I have? I've got one step up there. Set `answer` to one. I'm setting up n -- sorry, I'm not setting up n . I'm going to test n . And then I'm going to do two steps here, because I got a multiply `answer` by n and then set it to `answer`. And now similarly, we've got two steps there because I'm subtracting 1 from n and then setting it to n .

So I've got 2 plus 4 plus the test, which is 5. I've got 1 outside here. I got 1 outside there. And I'm going to go through this loop n times. So I would suggest that if I count the number of steps, it's $1 + 5n + 1$. Sort of what we did before. $5n + 2$ is the total number of steps that I use here.

But now, I'm interested in what's the worst case behavior? Well, in this case, it is the worst case behavior because it doesn't have decisions anywhere in here. But I just want to know what's the asymptotic complexity? And I'm going to say-- oh, sorry-- that is to say I could do this different ways. I could have done this with two steps like that. That would have made it not just $1 + 5n + 1$. It would have made it $1 + 6n + 1$ because I've got an extra step.

I put that up because I want to remind you I don't care about implementation differences. And

so I want to know what captures both of those behaviors. And in Big O notation, I say that's order n . Grows linearly. So I'm going to keep doing this to you until you really do wince at me. If I were to double the size of n , whether I use this version or that version, the amount of time the number of steps is basically going to double.

Now you say, wait a minute. $5n$ plus 2-- if n is 10 that's 52. And if n is 20, that's 102. That's not quite doubling it. And you're right. But remember, we really care about this in the asymptotic case. When n gets really big, those extra little pieces don't matter. And so what we're going to do is we're going to ignore the additive constants and we're going to ignore the multiplicative constants when we talk about orders of growth.

So what does O of n measure? Well, we're just summarizing here. We want to describe how much time is needed to compute or how does the amount of time, rather, needed to computer problem growth as the size of the problem itself grows. So we want an expression that counts that asymptotic behavior. And we're going to focus as a consequence on the term that grows most rapidly.

So here are some examples. And I know if you're following along, you can already see the answers here. But we're going to do this to simply give you a sense of that. If I'm counting operations and I come up with an expression that has n squared plus $2n$ plus 2 operations, that expression I say is order n squared. The 2 and the $2n$ don't matter. And think about what happens if you make n really big. n squared is much more dominant than the other terms. We say that's order n squared.

Even this expression we say is order n squared. So in this case, for lower values of n , this term is going to be the big one in terms of number of steps. I have no idea how I wrote such an inefficient algorithm that it took 100,000 steps to do something. But if I had that expression for smaller values of n , this matters a lot. This is a really big number.

But when I'm interested in the growth, then that's the term that dominates. And you see the idea or begin to see the idea here that when I have-- sorry, let me go back there-- when I have expressions, if it's a polynomial expression, it's the highest order term. It's the term that captures the complexity. Both of these are quadratic. This term is order n , because n grows faster than \log of n .

This funky looking term, even though that looks like the big number there and it is a big number, that expression we see is order $n \log n$. Because again, if I plot out as how this

changes as I make n really large, this term eventually takes over as the dominant term. What about that one? What's the big term there?

How many people think it's n to the 30th? Show of hands. How many people think it's 3 to the n ? Show of hands. Thank you. You're following along. You're also paying attention. How many people think I should stop asking questions? No show of hands. All right. But you're right. Exponentials are much worse than powers.

Even something like this-- again, it's going to take a big value of n before it gets there, but it does get there. And that, by the way, is important, because we're going to see later on in the term that there are some problems where it's believed that all of the solutions are exponential. And that's a pain, because it says it's always going to be expensive to compute. So that's how we're going to reason about these things.

And to see it visually, here are the differences between those different classes. Something that's constant-- the amount of time doesn't change as I change the size of the input. Something that linear grows as a straight line, as you would expect. Nice behavior. Quadratic starts to grow more quickly. The log is always better than linear because it slows down as we increase the size. $n \log n$ or log linear is a funky term, but we're going to see it's a very common complexity for really valuable algorithms in computer science. And it has a nice behavior, sort of between the linear and the quadratic. And exponential blows up.

Just to remind you of that-- well, sorry-- let me show you how we're going to do the reasoning about this. So here's how we're going to reason about it. We've already seen some code where I started working through this process of counting operations. Here are the tools I want you to use. Given a piece of code, you're going to reason about each chunk of code separately. If you've got sequential pieces of code, then the rules are called the law of addition for order of growth is that the order of growth of the combination is the combination of the order of the growth. Say that quickly 10 times.

But let's look at an example of that. Here are two loops. You've already seen examples of how to reason about those loops. For this one, it's linear in the size of n . I'm going to go through the loop n times doing a constant amount of things each time around. But what I just showed, that's order n . This one-- again, I'm doing just a constant number of things inside the loop-- but notice, that it's n squared. So that's order n squared. n times n .

The combination is I have to do this work and then that work. So I write that as saying that is

order of n plus order of n squared. But by this up here, that is the same as saying what's the order of growth of n plus n squared. Oh yeah. We just saw that. Says it's n squared. So addition or the law of addition let's be reasonable about the fact that this will be an order n squared algorithm.

Second one I'm going to use is called the law of multiplication. And this says when I have nested statements or nested loops, I need to reason about those. And in that case, what I want to argue-- or not argue-- state is that the order of growth here is a multiplication. That is, when I have nested things, I figure out what's the order of growth of the inner part, what's the order growth of the outer part, and I'm going to multiply-- bleh, try again-- I'm going to multiply together those orders of growth, get the overall order of growth. If you think about it, it makes sense.

Look at my little example here. It's a trivial little example. But I'm looping for i from 0 up to n . For every value of i , I'm looping for j from 0 up to n . And then I'm printing out A . I'm the Fonz. I'm saying heyyy a lot. Oh, come on. At least throw something, I mean, when it's that bad. Right? Want to make sure you're still awake. OK. You get the idea.

But what I want to show you here is notice the order of growth. That's order n . Right? I'm doing that n times. But I'm doing that for each value of i . The outer piece here loops also n times. For each value of i , I'm doing order n things. So I'm doing order of n times order of n steps. And by that law, that is the same order of n times n or n squared. So this is a quadratic expression.

You're going to see that a lot. Nested loops typically have that kind of behavior. Not always, but typically have that kind of behavior. So what you're going to see is there's a set of complexity classes. And we're about to start filling these in.

Order one is constant. Says amount of time it takes doesn't depend on the size of the problem. These are really rare that you get. They tend to be trivial pieces of code, but they're valuable. $\log n$ reflects logarithmic runtime. You can sort of read the rest of them. These are the kinds of things that we're going to deal with. We are going to see examples here, here, and here. And later on, we're going to come back and see these, which are really nice examples to have.

Just to remind you why these orders of growth matter-- sorry, that's just reminding you what

they look like. We've already done that. Here is the difference between constant log, linear, log linear squared, and exponential. When n is equal to 10, 100, 1,000 or a million. I know you know this, but I want to drive home the difference. Something that's constant is wonderful, no matter how big the problem is. Takes the same amount of time.

Something that is log is pretty nice. Increase the size of the problem by 10, it increases by a factor of 2. From another 10, it only increases by a factor of another 50%. It only increases a little bit. That's a gorgeous kind of problem to have. Linear-- not so bad. I go from 10 to 100 to 1,000 to a million.

You can see log linear is not bad either. Right? A factor of 10 increase here is only a factor of 20 increase there. A factor of 10 increase there is only a factor of 30 increase there. So log linear doesn't grow that badly. But look at the difference between n squared and 2 to the n . I actually did think of printing this out. By the way, Python will compute this. But it was taken pages and pages and pages. I didn't want to do it. You get the point. Exponential-- always much worse. Always much worse than a quadratic or a power expression. And you really see the difference here.

All right. The reason I put this up is as you design algorithms, your goal is to be as high up in this listing as you can. The closer you are to the top of this list, the better off you are. If you have a solution that's down here, bring a sleeping bag and some coffee. You're going to be there for a while. Right? You really want to try and avoid that if you can.

So now what we want to do, both for the rest of today in the last 15 minutes and then next week, is start identifying common algorithms and what is their complexity. As I said to you way back at the beginning of this lecture, which I'm sure you remember, it's not just to be able to identify the complexity. I want you to see how choices algorithm design are going to lead to particular kinds of consequences in terms of what this is going to cost you. That's your goal here.

All right. We've already seen some examples. I'm going to do one more here. But simple iterative loop algorithms are typically linear. Here's another version of searching. Imagine I'll have an unsorted list. Arbitrary order. Here's another way of doing the linear search. Looks a little bit faster.

I'm going to set a flag initially to false. And then I'm going to loop for i from 0 up to the length of L . I'm going to use that to index into the list, pull out each element of the list in turn, and check

to see is it the thing I'm looking for. As soon as I find it, I'm going to send-- sorry-- set the flag to true. OK? So that when I return out of the loop, I can just return found. And if I found it to be true, if I never found it, found will still be false and I'll return it.

We could count the operations here, but you've already seen examples of doing that. This is linear, because I'm looping n times if n is the length of the list over there. And the number of things I do inside the loop is constant. Now, you might say, wait a minute. This is really brain damaged, or if you're being more politically correct, computationally challenged. OK? In the sense of once I've found it, why bother looking at the rest of the list?

So in fact, I could just return true right here. Does that change the order of growth of this algorithm? No. Changes the average time. I'm going to stop faster. But remember the order of growth captures what's the worst case behavior. And the worst case behavior is the elements not in the list I got to look at everything. So this will be an example of a linear algorithm. And you can see, I'm looping length of L times over the loop inside of there. It's taking the order one to test it. So it's order n .

And if I were to actually count it, there's the expression. It's $1 + 4n + 1$, which is $4n + 2$, which by my rule says I don't care about the additive constant. I only care about the dominant term. And I don't care about that multiplicative constant. It's order n . An example of a template you're going to see a lot.

Now, order n where n is the length of the list and I need to specify that. That's the thing I'm after. If you think about it, I cheated. Sorry-- I never cheat. I'm tenure. I never cheat. I just mislead you badly. Not a chance. How do I know that accessing an element of the list takes constant time? I made an assumption about that. And this is a reasonable thing to ask about-- both what am I assuming about the constant operations and how do I know that's actually true?

Well, it gives me a chance to point out something that Python does very effectively. Not all languages do. But think about a list. Suppose I've got a list that's all integers. I'm going to need some amount of memory to represent each integer. So if a byte is 8 bits, I might reserve 4 bytes or 32 bits to cover any reasonable sized integer. When I represent a list, I could simply have each of them in turn. So what do I need to know?

I'm going to allocate out a particular length-- say 4 bytes, 32 bits, 32 sequential elements of memory to represent each integer. And then I just need to know where's the first part of the

list, what's the address and memory of the first part of the list. And to get to the i -th element, I take that base plus 4 bytes times i . And I can go straight to this point without having to walk down the list. That's nice. OK? It says, in fact, I can get to any element of memory-- I'm sorry-- any element of the list in constant time.

OK. Now, what if the things I'm representing aren't integers? They're arbitrary things and they take up a big chunk of space. Well, if the list is heterogeneous, we use a nice technique called indirection. And that simply says we, again, have a list. We know the address of this point. We know the address there for the i -th element of this list.

But inside of here, we don't store the actual value. We store a pointer to where it is in memory. Just what these things are indicating. So they can be arbitrary size. But again, I can get to any element in constant time, which is exactly what I want. So that's great.

OK. Now, suppose I tell you that the list is sorted. It's in increasing order. I can be more clever about my algorithm. Because now, as I loop through it, I can say if it's the thing I'm looking for, just return true. If the element of the list is bigger than the thing I'm looking for, I'm done. I don't need to look at the rest of the list, because I know it can't be there because it's ordered or sorted. I can just return false. If I get all the way through the loop, I can return false.

So I only have to look until I get to a point where the thing in the list is bigger than what I'm looking for. It's the order of growth here. Again, the average time behavior will be faster. But the order of growth is I've got to do order of length of the list to go through the loop, order of one to do the test, and in the worst case, again, I still have to go through the entire list. So the order of growth here is the same. It is, again, linear in the length of the list, even though the runtime will be different depending whether it's sorted or not.

I want you to hold on to that idea, because we're going to come back to the sorted list next week to see that there actually are much more efficient ways to use the fact that a list is sorted to do the search. But both of these versions same order growth, order n . OK. So lurching through a list-- right, sorry-- searching through a list in sequence is linear because of that loop. There are other things that have a similar flavor. And I'm going to do these quickly to get to the last example.

Imagine I give you a string of characters that are all soon to be composed of decimal digits. I just want to add them all up. This is also linear, because there's the loop. I'm going to loop

over the characters in the string. I'm going to cast them into integers, add them in, and return the value. This is linear in the length of the input s . Notice the pattern. That loop-- that iterative loop-- it's got that linear behavior, because inside of the loop constant number of things that I'm executing.

We already looked at `fact iter`. Same idea. There's the loop I'm going to do that n times inside the loop a constant amount of things. So looping around it is order n . There's the actual expression. But again, the pattern I want you to see here is that this is order n . OK.

Last example for today. I know you're all secretly looking at your watches. Standard loops, typically linear. What about nested loops? What about loops that have loops inside of them? How long do they take? I want to show you a couple of examples. And mostly, I want to show you how to reason about them. Suppose I gave you two lists composed of integers, and I want to know if the first list is a subset of the second list. Codes in the handbook, by the way, if you want to go run it.

But basically, the simple idea would be I'm going to loop over every element in the first list. And for each one of those, I want to say is it in the second list? So I'll use the same kind of trick. I'll set up a flag that's initially false. And then I'm going to loop over everything in the second list. And if that thing is equal to the thing I'm looking for, I'll set `match` to true and break out of the loop-- the inner loop. If I get all the way through the second list and I haven't found the thing I'm looking for, when I break out or come out of this loop, `matched` in that case, will still be false and all return false.

But if up here, I found something that matched, `match` would be true. I break out of it. It's not false. Therefore, a return true. I want you look at the code. You should be able to look at this and realize what it's doing. For each element in the first list, I walk through the second list to say is that element there. And if it is, I return true. If that's true for all of the elements in the first list, I return true overall.

OK. Order of growth. Outer loop-- this loop I'm going to execute the length of L_1 times. Right? I've got to walk down that first list. If I call that n , it's going to take that n times over the outer loop. But what about n here? All of the earlier examples, we had a constant number of operations inside of the loop. Here, we don't. We've got another loop that's looping over in principle all the elements of the second list.

So in each iteration is going to execute the inner loop up to length of L_2 times, where inside of

this inner loop there is a constant number of operations. Ah, nice. That's the multiplicative law of orders of growth. It says if this is order length L_1 . And we're going to do that then order length of L_2 times, the order of growth is a product. And the most common or the worst case behavior is going to be when the lists are of the same length and none of the elements of L_1 are in L_2 . And in that case, we're going to get something that's order n squared quadratic, where n is the length of the list in terms of number of operations.

I don't really care about subsets. I've got one more example. We could similarly do intersection. If I wanted to say what is the intersection of two lists? What elements are on both list 1 and list 2? Same basic idea. Here, I've got a pair of nested loops. I'm looping over everything in L_1 . For that, I'm looping over everything in L_2 . And if they are the same, I'm going to put that into a temporary variable.

Once I've done that, I need to clean things up. So I'm going to write another loop that sets up an internal variable and then runs through everything in the list I accumulated, making sure that it's not already there. And as long as it isn't, I'm going to put it in the result and return it. I did it quickly. You can look through it. You'll see it does the right thing.

What I want it to see is what's the order of growth. I need to look at this piece. Then I need to look at that piece. This piece-- well, it's order length L_1 to do the outer loop. For each version of e_1 , I've got to do order of length L_2 things inside to accumulate them. So that's quadratic.

What about the second loop? Well, this one is a little more subtle. I'm only looping over temp, which is at most going to be length L_1 long. But I'm checking to see is that element in a list? And it depends on the implementation. But typically, that's going to take up to the length of the list to do it. I got to look to see is it there or not. And so that inner loop if we assume the lists are the same size is also going to take potentially up to length L_1 steps. And so this is, again, quadratic. It's actually two quadratics-- one for the first nested loop, one for the second one, because there's an implicit second loop right there. But overall, it's quadratic.

So what you see in general-- this is a really dumb way to compute n squared. When you have nested loops, typically, it's going to be quadratic behavior. And so what we've done then is we've started to build up examples. We've now seen simple looping mechanisms, simple iterative mechanisms, nested loops. They tend to naturally give rise to linear and quadratic complexity. And next time, we're going to start looking at more interesting classes. And we'll see you next time.

