PROFESSOR: I wanted to give everybody a more conceptual idea of what big O notation as well as, hopefully, answer any lingering questions you might have about object-oriented programming. So I have these notes, and I type them up, and they're pretty detailed. So I'm just going to go through some points kind of quickly. Who's still kind of unclear about why we even due big O notation? So who can explain why we do big O notation quickly? What is it? Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Right. Exactly. So big O notation gives us an upper bound on how long something is going to take. Now, something that's important to remember is it's not a time bound. So something that's often confusing is that people say, oh, this is something that'll tell us how long our programs going to run.

That's actually not the case. Big O notation informs how many steps something's going to take. And so why is that important? Well, I mean I look at all you guys, a couple of you have laptops out. Everybody's computer run something at a different speed. Right? But if we say something is big O of n, what we're saying here is we're saying that the worst case number of steps your program is going to take is going to be linear with respect to the size of the input. So if my computer is five times faster than your computer, my computer will probably run it five times faster. As the size of the input grows, I'm going to expect a linear speedup in the amount of time it's going to take.

So why is that important? At the bottom of page one, big O notation, we are particularly concerned with the scalability of our functions. So what the O notation does is it might not predict what's going to be the fastest for really small inputs, for

an array size 10. You guys know a little bit about graphs, right? We have a graph of x-squared, and we have a graph of x-cubed. There's a portion of time where the graph of x-squared is actually bigger than x-cubed. But then all of a sudden, there's a point where x-cubed just goes, whoosh, way bigger than x-squared.

So if we're in some really small amount of input, big O notation might not tell us what's the best function. But in big O notation, we're not concerned about small inputs. We're concerned about really big inputs. We're concerned about filtering the genome. We're concerned about analyzing data from Hubble, really huge blocks of data.

So if we're looking at a program that analyzes the human genome, like three million base pairs, some segment that we're looking at, and we have two algorithms. One runs in order n time, and one runs in order n-cubed time. What this means is regardless of the machine that we're running on, so this is algorithm 1, this is algorithm 2, regardless of the machine that we're running on, we'd expect algorithm 2 to run approximately n-cubed over n approximately n-squared slower. So with big O notation, you can compare two algorithms by just looking at the ratio of their big O run time.

So if I'm looking at something that has an array of size two million as its input, is it clear that this is going to be a much better choice? Ok. So you'll run into that, especially a lot of you guys are taking this for the purposes of scientific computing. So you'll run into big O notation a lot. It's important to have a grasp of what it means.

On the second page of the handout, I have some common ones that you'll see. The first one is constant time. We denote constant time as order 1. But you'll notice that I have here is order 1 is equal to order 10 is equal to order 2 to the 100th. That's unexpected to a lot of people who are learning about big O notation. Why is this true? That seems kind of ridiculous. This is a really big number. This is really small number. Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. Exactly. So we look at a graph of 1 and a graph of 2 to the 100th. Ok. We'll see that even though 2 to the 100th is much higher, much bigger than 1, if this is our input size, as the size of our input grows, do we see any change in these two graphs? No. They're completely constant.

When you're doing big O notation, if you run across an algorithm that does not depend on the size of the input, OK, it's always going to be order 1. Even if it's like 2 to the 100th steps, if it's a constant number of times regardless of the size of the input, it's constant time.

Other ones you'll see are logarithmic time. Any base for logarithmic time is about the same order. So order log base 2 of n is order log base 10 of n. This is the fastest time bound for search. Does anybody know what type of search we'd be doing in logarithmic time? Something maybe we--

AUDIENCE: Bisection time

PROFESSOR: Yeah. Exactly. Bisection search is logarithmic time. Because we take our input. And at every step, we cut in half, cut in half, cut in half, and that's the fastest search we can do. The order n is linear time. Order n log n is the fastest time bound we have for sort. We'll be talking about sort in a couple of weeks. And order n-squared is quadratic time. Anything that is order n to some variable, so order n-squared, order n-cubed, order n-fourth, all of that is going to be less than order something to the power of n. So if we have something that's order 2 to the n, that's ridiculous. That's a computationally very intensive algorithm.

So on page two, I have some questions for you. (1), (2), (3). Does order 100 n-squared equal order n-squared. Who says yes? All right. Very good. How about does order one quarter n-cubed equals order n-cubed? Does order n plus order n equals order n? The answer is yes to all of those. In the intuitive sense behind this is that big O notation deals with the limiting behavior of function.

So I made some nifty graphs for you guys to look at. When we're comparing order 100 n-squared to n-squared n cubed and 1/4 n-cubed, what people often think of is

what I have here in the first figure. So these are the four functions I just mentioned. There's a legend in the top left-hand corner. And the scale of this is up to x equals 80.

So you'll see at this scale, this line right here is 100 x-squared. So this is, I think, often a tripping point is that when people are conceptualizing functions, they're saying, well, yeah, 100 x-squared is much bigger than x-cubed, which is a lot bigger than 1/4 x-cubed. So for very small inputs, yes that's true. But what we're concerned about is the behavior as the input gets very, very large.

So now, we're looking at a size of up to 1,000. So now we see here, x-cubed, even though it's a little bit smaller than 100 x-squared in the beginning, it shoots off. x-cubed is much bigger than either of the 2 x-squared. And even 1/4 x-cubed is becoming bigger than 100 x-squared out of 1,000. So that's an intuitive sense why x-cubed no matter what the coefficient is in front of it is going to dominate any term with x-squared in it, because x-cubed is just going to go, whoosh, real big like that.

And if we go out even further, let's go out to input size of 50,000, we go out to an input size of 50,000, we see that even 100 x-squared versus just x-squared, alright? they're about the same. The x-cubed terms now, they're way above x-squared. So the two x-squared terms, 100 versus just 1, as far as the coefficient goes, they're about the same.

So this is the scale at which we're concerned about when we're talking about big O notation, the limiting behavior as your input size grows very large. 50,000 is not even that large, if you think about the size of the genome. I mean does anybody here bio? What's like the size of the human genome. How many base pairs? Or even one gene or one chromosome.

AUDIENCE: [INAUDIBLE].

PROFESSOR: What's the biggest?

AUDIENCE: It's over 50,000.

PROFESSOR: Yeah, over 50,000. And we're talking about the amount of data that we get back from the Hubble Space Telescope. I mean the resolution on those things are absolutely ridiculous. And you run all sorts of algorithms on those images to try and see if there's life in the universe. So we're very concerned about the big long term behavior of these functions.

How about page three? One last question. Does order 100 n-squared plus 1/4 n-cube equal order n-cubed? Who says yes? And so I have one more graph. Down here, these red dots are 100 x-squared. These blue circles are 1/4 x-cubed. And this line is the sum. We can see that this line is a little bit bigger than the 1/4 x-cubed term. But really, this has no effect at this far out.

So that's why we're just going to drop any lower order terms whenever you're approached with a big O expression that has a bunch of constant factors, it has all sorts of different powers of n and stuff, you're always just going to drop all the constant factors and just pick the biggest thing. So this line right here is order n-cubed. Is that clear to everybody?

So now I've gotten through the basics of how we analyze this and why are we looking at this. Let's look at some code. So the first example, all of these things right here, in Python, we make the assumption that statements like this, x plus 1, x times y, all these mathematical operations are all constant time. That's something that you can just assume. So for this function down here, we have constant time, constant time, constant time, constant time operation. So we'd say, this function bar is what? What's its complexity?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. Constant time. So the complexity of all these functions are just a 1, because it doesn't matter how big the input is. It's all going to run in constant time.

For this multiplication function here, we use a for loop. Oftentimes, when we see for loops that's just going through the input, there's a signal to us that it's going to probably contain a factor of O(n). Why is that? What do we do in this for loop? We

say for i in range y. What does that mean? How many times do we execute that for loop? Yeah, y times.

So if y is really small, we execute that for loop just a few number of times. But if y is really large, we execute that for loop a whole bunch of times. So when we're analyzing this, we see this for loop and we say, ah, that for loop must be O(y). Does that make sense to everybody? OK, good. Let's look at a factorial. Can anybody tell me what the complexity of factorial is?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. Order n. Why is it order n?

AUDIENCE: Because it's self for loop.

PROFESSOR: Yeah. It's the exact same structure. We have a for loop that's going through range 1 to n plus 1. So that's dependent on the size of n. So this for loop is order n. And inside the for loop, we just do a constant time operation. That's the other thing. Just because we have this for loop doesn't mean that what's inside the for loop is going to be constant.

But in this case, if we have order n times, we do a content time operation. Then this whole chunk of the for loop is order n. The rest of everything else is just constant time. So we have constant time plus order n times constant time plus constant time, they're going to be order n. How about this one? Factorial 2.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. Exactly. This is also order n. The only thing that's different in this code is that we initialize its count variable. And inside the for loop, we also increment this count variable. But both result times equals num and count plus equal 1, both of these are constant time operations. So if we do n times 2 constant times operations, that's still going to be order n.

So the takeaway from these two examples that I'm trying to demonstrate here is a single line of code can generate a pretty complex thing. But a collection of lines of

code might still be constant time. So you have to look at every line of code and consider that. I've thrown in some conditionals here. What's the complexity of this guy?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. This is also linear. What's going on here? We initialize a variable count that's constant time. We go through character in a string. This is linear in the size of a string. Now we say if character equal, equal t, this character equal, equal t, that's also a constant time operation. That's just asking if this one thing equals this other thing.

So we're looking at two characters. We're looking at two numbers. Equal, equal or not equal is generally a constant times operation. The exception to this might be a quality of certain types, like if you define a class and you define a quality method in your class, and the equality method of your class is not constant time, then this equal, equal check might not be constant time. But on two strings, equal, equal is constant time. And this is constant time as well.

So linear in the size of a string. Something that's important when you're doing this for exams, it's a good idea to define what n is before you give the complexity bound. So here I'm saying n is equal to the size of a string. So now, I can say this function is order n. What I'm saying is that it's a linear with respect to the size or the length of a string.

Because sometimes, like in the one where there is the input x and y, the running time was only linear in the size of y. So you want to define that n was equal to the size of y to say that it was order n. So always be clear. If it's not clear, be sure to explicitly state what n is equal to. This code's a little more tricky. What's going on here?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. That was perfect. So just to reiterate, the for loop we know is linear with

respect to the size of a string. We have to go through every character in a string. Now, the second is if char in b string, when we're looking at big O notation, we're worried about the worst case complexity in upper bound. What's the worst case?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. If the character is not in b string, we have to look at every single character in b string before we can return false. So that is linear. This one single line, if character in b string, that one line is linear with respect to the size of b string. So how do we analyze the complexity of this? I want to be able to touch the screen.

We have this for loop. This for loop is executed. Let's call n is the length of a string. This for loop is executed n times. Every time we execute this for loop, we execute this inner body. And what's the time bound on the inner body? Well, if we let m equal the length of b string, when we say that this check is order m every time we run it, then we run an order m operation order n times. So the complexity is-- we use something of size m, n times.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. Just order n, m. So we execute an order m check order n time, we say this function is order n, m. Does that make sense to everybody? Because you'll see the nested for loops. Nested for loops are very similar to this. While loops combine the best of conditionals with the best of for loops. Because a while loop has a chance to act like for loop, but a while loop can also have a conditional. It's actually possible to write a while loop that has a complex conditional that also executes a number of times. And so you could have one single line of code generating like an order n-squared complexity. Let's look at factorial 3. Who can tell the complexity of factorial 3?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. It's also linear. It's interesting that factorial is always linear despite its name. We have constant time operations. How many times does the while loop executed?

AUDIENCE: n times.

PROFESSOR: Yeah, n times. And what's inside the body of the while loop? Constant time operations. So we execute a bunch of constant time operation n times order n. How about this char split example? This one's a little tricky because you're like, well, what's the complexity of len? In Python, len's actually a constant time operation. This example's very crafted such that all of the operations that are here are constant time. So appending to a list is constant time. And indexing a string is constant time. So what's the complexity of char split? Constant time.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Who would agree with constant time? And who would say it's linear time? OK, yeah. Very good. It is linear time. That's a correct intuition. We say while the length of the a string is not equal to the length of the result, these are two constant time operations. Well, what do we do? We append a value to the result, and then we add up this index.

So when is this check going to be equal? This check's going to be equal when the length of the result is equal to the length of a string. And that's only going to happen after we've gone through the entire a string, and we've added each of its characters to result. So this is linear with respect to the size of a string.

Something that's important to recognize is that not all string in the list operations are constant time. There's a website here that first off, it says C Python if you go to it. C Python just means Python implemented in C, which is actually what you're running, C Python. So don't worry about that.

There's often two time bound complexities. It says the amortized time and the worst case time. And so if you're looking for big O notation, you don't want to use the amortized time. You want to use the worst case time. And it's important to note that operations like slicing and copying actually aren't constant time.

If you slice a list or a string, the complexity of that operation is going to depend on how big your slice is. Does that makes sense? Does the way that a slice works is

that walks through the list until it gets to the index, and then keeps walking until the final index, and then copies that and returns it to you. So slicing is not constant time. Copying is similarly not constant time. For this little snippet of code, this is just similar to what we-- yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So this is what I was saying. You want to define what n is. So we say something like n equals the length of a string. And then you can say it's order n. It's important to define what you're saying the complexity is related to. So here, I'm saying if we let n equal to the size of z, can anybody tell me what the complexity of this snippet of code is? [UNINTELLIGIBLE].

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, precisesly. Order n-squared. Why? Well, because we execute this for i for loop here order n times. Each time through this for loop, the body of this for loop is, in fact, another for loop. So my approach to problems like this is just step back a minute and ignore the outer loop. Just concentrate on the inner loop. What's the runtime of this inner loop? Yeah. This is order n. We go through this. Now, go to the outer loop. Just ignore the body since we've already analyzed the body. Ignore it. What's the complexity of the outer loop? Also order n.

So now you can combine the analysis. You can say for order n times, I execute this body. This body takes order n times. So if execute something that's order n order n times, that is order n squared complexity. So we just multiply how long it takes the outer body of the loop to take the inner body of the loop. And so in this fashion, I could give you now probably a four or five nested for loop, and you could tell me the complexity of it.

Harder sometimes to understand is recursion. I don't know how important it is to understand this because I've never actually taught this class before. But Mitch did tell me to go over this. So I'd like to touch on it. So consider recursive factorial. What's the time complexity of this? How can we figure out the time complexity over

a recursive function?

The way we want to figure out the time complexity of a recursive function is just to figure out how many times we're executing said recursive function. So here I have recursive factorial of n. When I make a call to this, what do I do? I make a call to recursive factorial n minus 1. And then what does this do? This calls recursive factorial on a sub problem the size n minus 2.

So oftentimes, when you're dealing with recursive problems to figure out the complexity, what you need to do is you need to figure out how many times you're going to make a recursive call before a result is returned. Intuitively, we can start to see a pattern. We can say, I called on n, and then n minus 1, and then n minus 2, and I keep calling recursive factorial until n is less than or equal to 0. When is n going to be less than or equal to 0? Well, when I get n minus n. So how many calls is that?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. This is n calls. So it's a good practice to get into being able to draw this out and work yourself through how many times you're running the recursion. And we see we're making n calls, we can say, oh, this must be linear in time.

How about this one, this foo function? This one's a little harder to see. But what are we doing? We call foo on input of size n, which then makes a call to sub problem the size n/2, which makes the call to a sub problem of size n/4 and so on until I make a call to sub problem of some size. So this is n. This is 2 to the 1st. This is 2-squared. We start to see a pattern-- 2-squared, 2-cubed, 2 to the fourth. So we're going to keep making calls on a smaller, and smaller, and smaller sub problem size. But instead of being linear like before, we're decreasing at an exponential rate.

There's a bunch of different ways to try and work this out in your head. I wrote up one possible description. But when we're decreasing at this exponential rate, what's going to end up happening is that this recursive problem where we make a recursive call in the form to sub problem of size n/b, the complexity of that is always

going to be log base b of n.

So this is just like bisection search, where bisection search, we essentially do in bisection search. We restrict the problem size by half every time. And that leads to logarithmic time, actually log base 2 of n. This problem is also log base 2 of n. If we change this recursive call from n/2 to n/6, we get a cut time complexity of log base 6 of n. So try and work that through. You can read this closer later. Definitely ask me if you need more help on that one.

The last one is how do we deal time complexity of something like Fibonacci? Fibonacci, fib n minus 1 plus fib n minus 2, initially, that kind of looks linear. Right? We just went over the recursive factorial, and it made the call to a sub problem the size n minus 1. And that was linear.

Fibonacci's a little bit different. If you actually draw out in a tree, you start to see like at every level of the tree, we expand the call by 2. Now imagine this is just for Fibonacci of 6. Whenever you're doing big O complexity, you want to imagine it and put 100,000, 50,000. And you could imagine how big that tree grows.

Intuitively, the point to see here is that they're going to be about n levels to get down to 1 from your initial input of 6. So to get down to 1 from an initial input of size n is going to take about n levels. The branching factor of this tree at each level is 2. So if we have n levels, and at each level, we increase our branching factor by another 2, we can say that a loose bound on the complexity of this is actually 2 to the n.

This is something that's even less intuitive, I think, than what we did before with the logarithms. So try and work through it again. Play with it a little bit. There's actually a tighter bound on this, which is like 1.62 to the n, which is a lot more complicated math that you could look up. But for the purposes of this class, it's sufficient to say that Fibonacci is order 2 to the n.

So does that roughly clear up some time complexities stuff for you guys? OK, awesome. Does anybody have the time? I forgot my watch today.

AUDIENCE: 12:42.

PROFESSOR: OK, excellent. That gives us a little bit of time to talk about object-oriented programming. Does anybody had any specific questions that object-oriented programming? How about this? How many of you guys finished the problem set and turned it in already? Or did any of you guys not turn in the problem set yet? I'll talk loosely about it then, not too specifically. Does anybody have any questions from, I guess, at least the first part? We're making some classes, making some trigger classes. Yeah?

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Self dot what?

AUDIENCE: [INAUDIBLE].

PROFESSOR: When we have like self-- we have like the getter methods. So what's important about that? I'll Tell you what's important about that. So we have a class. Let's say we have a class person. So we define our INIT method to just take a name. And so now, what the problems that ask you to do was to define a getter method. Define a getter method called get_name that just returns the attribute.

So what's the point of this? Because I can just say Sally equals person. So here, I defined a person named Sally. And I initialized a person with the string Sally. If I just look at sally.name, that's going to just directly print the attribute. So why do we need this get name function? What's the point of this additional getter method? Does anybody know why that is?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Right. So that's what it does. This get_name does return the attribute name. But we don't need this method to just look at the attribute name. Let's actually code this up. So we have class person. So if we run this code, and over here in the shell, we define Sally equals person with the name Sally. If I just print sally.name, it prints the attribute.

So why did I need to provide this getter method called get_name that does the same thing? That's the question. That seems sort of redundant. But there's actually a pretty big important reason for it. Let's say we set s name equal to the attribute sally.name. If we look at s name, we see Sally. Now if I say-- actually, I'm not sure if this is the correct reasoning. This is going to be better.

Let's say Sally equals a person Sally who's taking what? 1803, 605, 11.1. So now I can look at the attribute classes to show Sally's classes, which are weird flows. And I can also use sally.getclasses to look at Sally's classes. If I set a variable s classes equal to sally.classes, this binds this variable s classes to the attribute sally.classes.

Now if I say sclasses.append 1401, if I now look at the attribute sally.classes, it now has 1401 in it. This is not safe. This is not type safe. Because the reason for that is if you define a class, and you access the classes' attributes directly instead of through a getter method, you can then do this. And sometimes, it's accidental. You'll set some variable equal to some attribute of a class. Then later on in your code, you'll alter that variable. But that variable is not a copy of the attribute.

Yes, you can make copies of that attribute and stuff, but the overall takeaway is that in programming, we try to do something called defensive programming. This isn't defensive. Because it is possible if you code it incorrectly to alter the attribute the instance of the class. But if we use the getter method, if instead of sally.classes, instead of directly accessing the attribute here, we have set s classes equal to sally.getclasses.

And then, we had changed s classes around. That wouldn't have happened, because the getter method, it does return self.classes. But in the way that Python is scoped and when we return something, we're not returning the exact same thing, the reference that we're returning a copy of it.

Does that make sense? All right. Cool. Other questions about classes? We have a little class appear if there's like some basic stuff that you'd like explained again. Now's the time.

AUDIENCE: [INAUDIBLE].

PROFESSOR: So here, I'm setting just some variable s classes equal to the attribute sally classes. it's just like setting any sort of variable equal to some other quantity.

AUDIENCE: So you appending the variable, but it also appended like the attribute of Sally?

PROFESSOR: So what I did here was I set the variable s classes equal to this attribute sallly.classes. And then, because I know this is a list, I appended another value to it. But this is the same as when we have two lists. If we have a list called a, and we say a is equal to 1, 2, 3, then I say b is equal to a. What is b?

Now If I say b.append 1401, what does b look like? What does a look like? Because they're aliases of each other. So what I did here, when I set s classes directly equal to the attribute sally.classes, I made s classes an alias of the attribute. But the problem with that is that then I can change them. And because they're aliases, the attribute itself has changed. And we don't want to do that in object-oriented programming. We need to find an object. The only way you should be able to change an attribute is through some method of the class that allows you to change that attribute.

So if I want to be able to add a class to Sally's class lists, I should define a method called define add class that does self.classes.append new class. While technically, it's possible to directly access an attribute, it's really bad practice to do so simply because this unexpected behavior can result. And also because if you say, oh, well, it's not going to matter for this one time, I'll remember how to do the right thing. The problem with that is it's often the case that you're not the only person using your code. So it's a better practice to provide all the sorts of methods that you would need to do with the class in order to get an access and change attributes as methods within the class. Does that make sense?

So yeah, this is maybe our one violation if you guys have been attending my recitation. Our mantra of programmers are lazy. This is less lazy than just directly accessing the attributes. But even though we know that programmers are super,

super lazy, programmers also like to be super, super safe. So when there's a trade off between defensive programming and being lazy, always pick defensive programming.