[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: All right, so let's get started with today's lecture. We're going to look at a lot more code where we basically figure out the complexity class of that given code. So first, let's remember what we learned at the end of the last lecture. So we introduced this theta notation as a notation to mark the order of growth of a particular function or a particular piece of code. And the theta we preferred over big O notation because the theta allowed us to get this asymptotic upper bound on the worst-case runtime of our function.

So we wanted an asymptotic bound as opposed to an upper bound because that upper bound can be anything that grows faster than our function. So we prefer this theta as the asymptotic bound. So at the end of last lecture, we basically said that given some function, the theta for that function is going to be the dominant term of that function. So if we have a whole bunch of terms, we focus on the one that grows the most. We drop any additive constants, any multiplicative constants, and all the other terms that don't grow as fast as that biggest one.

So we ended up with some classes of algorithms that we're going to go over today. We're going to see a bunch of codes that fall within those classes of algorithms. But before we go into that, I wanted to just quickly recap the end of last lecture. So we saw an example that was pretty similar to this one, if not the same.

So we know that given some function, we can grab the theta of that function by focusing on that dominant term. But how do we get at that function? So given some piece of code, the idea to get at that function was to first start by looking at the inputs to the function. So we have three inputs in this particular case, L, L1, and L2.

Once we figure out the inputs to this function, we go on and look at everything within the code that depends on these input parameters. So they could be direct, like a loop that goes over something related to the input, or it could be indirect, as we're going to see in some examples later today. But we basically look at just the parts of the function that deal with this input.

If we want to be exact, we start by finding out the exact number of operations that we do within this code. That's something that we did when we counted the number of operations given some function. So we're going to count the number of operations given this code in relation to L1, L2, and L. So we've got this relationship that we can come up with that relates the number of operations run as a function of L, L1, and L2.

So the 1 over here is constant because we just have an assignment here for some variable. The next term here is not constant. There are five constant things that we're doing. Assigning I to be a value and range, indexing into L at I, that's two. Indexing into L1 at I, that's three. Checking the equality, that's four. And then setting inL1 to be True, that's five.

So there's five operations, but these are repeated how many times? Well, they're repeated length L1 times because this loop goes through length L1. So this term here, this for loop here is length L1 times 5 number of operations. Then the one here is this assignment over here. And then this loop down at the bottom is exactly the same as the loop up at the top, except that now this bottom loop repeats length L2 times.

So as L2 gets bigger, this loop will take longer to run. That's how we think about that. And then, lastly, the plus 2 at the end of that relationship is finding the and of these two variables and then doing a return. So that leads us to simplify it as 5 times length L1 plus 5 times length L2 plus 3. And this becomes the function that we can then grab the theta.

So now we just use the regular rules of theta, law of addition, and law of multiplication, if there's anything to add or multiply. In this particular case, let's say that L, L1, and L2 are all the same length. And then we can simplify the above function to 10 length L plus 3. And then the theta of that becomes just theta of length L because we drop the 3, we drop the 10 multiply-- the 10 multiplying L. And then we just keep length L.

So this is how we get at the theta of a particular function. This is one we looked at last time. But as we look at more functions today, we're going to get better at just identifying the parts of the code that just deal with our inputs. This inL1 equals False. This inL2 equals False. This return, this and, those are all constant things that are happening. So we don't need to focus in on those. We just maybe glance at them really quickly to make sure there's nothing funky going on that's dependent on the length of our list.

But we can just basically say, well, we've got our inputs. We've got one for loop that goes through the length, another for loop that goes through the length. They're in series, so we use the law of addition to say that this function is theta of length L1 plus length L2. And so then we can quickly tell the theta of that function just by looking at the parts that depend on the input.

So that leads us-- so at the end of last lecture, we ended up with looking at these-- sorry, deciding that these are the complexity classes that we can categorize a lot of our functions in. So theta of 1 is constant. Theta of log n is logarithmic. Here, n is-- assuming n is the input to my function. Theta of n is linear. Theta of n log n is log linear. Theta of n, if n is my input to some constant, like n squared and cubed, runs in polynomial time.

And then theta of some constant to the n, where n is my input, is going to be exponential, like 2 to the n, 3 to the n. Those are all considered exponential time algorithms. And when we write our algorithms, we want to be up in this maybe top four, maybe top five, though polynomial is going to grow pretty quickly as our input gets big.

So if we can take our code and just quickly glance at it and classify it within one of these algorithms, that can guide us towards writing-- towards deciding whether the algorithm we wrote was good or bad. If we glance at it and say, hey, this algorithm is exponential or this function that I wrote is exponential, maybe we want to rethink our approach to the problem and try to get it into one of the upper complexity classes.

So what we're going to do the rest of this class is just go through a bunch of these complexity classes, and we're going to see some codes that belong to these complexity classes and hopefully give you an idea of what code looks like that fits in one of these complexity classes.

So the first one we'll look at is the constant complexity class. It's pretty simple. It's not really very interesting. If your code belongs in this constant complexity class, that means that it does not depend on the input at all. It always runs in constant time. So your code can have loops, or it can have some sort of recursive structure. But that loop or that recursive structure doesn't depend on the input at all.

So it's fine to have loops. It's just as long as it doesn't depend on the input, it's considered constant. So there are some built-in operations that are constant time. So if you see any of these operations, like indexing into a list, appending to a list, grabbing the value associated with a dictionary key, those are all constant time. So if you see them in your code, you don't need to account for them at all. But we're going to see in a few slides that there are some operations on lists and dictionaries that do add some nonconstant complexity. So you can't just brush them off.

All right, let's look at a couple examples of code. So here's a very simple function, add x comma y. So x and y are my inputs. There is no loop or nothing recursive, nothing that takes time here. There's nothing that repeats in this code. So the complexity of this code is theta of 1. That's it.

Here's another example. This is our kilometer example, taking in miles. All it does is a multiplication. Again, theta of 1. There's nothing interesting going on here-- no loop, no recursive. Here's a function that does have a loop within it. First thing we look at, though, is my input. What variable is my input here? It's x, right? So which part of my code here depends on x?

Well, there's something that I'm adding here. So I'm adding x onto some number. And I do have a loop, but does the loop depend on x? No. It depends on some number that is just 100 within my function. If y equals x here, then this code wouldn't be constant, because this loop will go through x times. But here, y is just 100, so this code is theta of 1 complexity. There's nothing here that depends on x as x grows.

All right, so not very interesting examples there. So let's move on to the next simplest class of functions, the linear complexity class. And these functions will be usually denoted by one loop or maybe many loops in series or something like that. But these loops all depend just linearly on n. You could also have a recursive function that repeats that's also linearly in n.

So we're going to see an example of a recursive function in a little bit. But first, we'll start out with just some functions that loop linearly with n. There are some built-in operations though that are linear in time. So if we ever see these operations within our code, we can't ignore them because they will contribute a theta of n complexity to our code. So we have to account for them like if we have some e in n within some other loop, we can't just say this e in n is constant. We'd have to use the law of multiplication or something like that to account for it.

So checking if an element is in a list obviously is linear because you have to look at each element in the list to determine that e is in it or not. Making a copy of your list is also linear in time. Even though we're making a copy of half of our list, so the first half of our list, it's still linear because copying 0.5 times length L is still theta of length L. That multiplicative constant on the front of our length L is 0.5. So if we drop it, that's still theta of length L.

Checking for equality between two lists is also constant because you have to look at each element in the list, compare them to make sure that they're the same or not. And deleting an item in a list is also linear in time. Sorry, this one was constant in time-- sorry, this one was linear in time. Deletion is also linear in time just because of the way lists are stored in memory. So if you delete an item in the end-- from your list, Python will count that as linear time complexity.

So let's look at some examples. First, we'll just start out with just some regular functions with loops, and then we'll look at one recursive function. So here I've got multiply x by y. It loops through range y. And it just adds x plus x plus x plus x y times. So I've got two parameters here. So I need to think about the complexity of this function with regards to both of them.

So the complexity with respect to y is theta of y because I've got one loop that's a function of y. So this loop will repeat however big y is. So if y increases, the time this loop takes will also increase. So the theta complexity of this function is theta of y with respect to y.

But what's the complexity of this function with respect to x? I have no looping structure here that's with respect to x. All I'm doing to x is just adding on to some number. So the complexity with respect to x is just theta of 1. So the overall complexity of this function is just going to be theta of y. X does not contribute anything to this, the runtime of this.

All right, so this and the previous sort of loop function from the constant kind of tells us that we need to be careful about what the inputs are. When we report the complexity, we have to report it with respect to the inputs to our function. We don't always just say theta of n or theta of n squared or theta of length n, whatever it is. We have to relate it to the inputs to our function. And if we have more than one input, we have to be careful to account for all of the inputs that contribute to the complexity.

All right, let's look at another example. So here's one where you take in a string s. We loop through each character in s. We cast each character to an integer. And then we add on to some value. So we're essentially just adding on all of the characters in s, in the string s. So this has one loop that loops through all the elements in s.

Now, if s is a string, what's going to make this program slower? Is it that the string-- so the numerical value of the string is bigger? No, because if I'm looping through the string 1, 0, 0, 0, it's going to take the same amount of time as if I'm looping through the string 9, 9, 9, 9. It's the length of the string that matters. So that's what this loop is doing. It's taking into account the length of the string. So if my string is longer, then it's going to take longer to run.

So the complexity of this function is just theta of length s because that's the length of the string contributes to slowing down my function. Everything else that we do is constant. So the overall complexity is theta of length of s. Or if it's simpler, you can just say theta of n, but then you have to say where n is something, like length of s.

All right, here's another example. This is a factorial program that does it iteratively. So it's going to use a loop to keep adding-- sorry, to keep multiplying on i to calculate the factorial of some n. So in this case, my input is n. So now I'm going to look through my function to see what part of my function depends on n. So here, n is just a number. And I'm looping through from 2 all the way up to n plus 1. So I'm going to loop through n minus 1 times overall.

Since I'm looping through n minus 1 times, there's nothing else really that's contributing to the complexity. So theta of n minus 1 is just theta of n. So the complexity of this function is just theta of n. Everyone OK so far? So very simple programs that just have one loop that just depends on the input linearly.

OK, I will make a little note about the factorial because this is kind of something important. It's going to tell us the difference between theory, which is what this class is mostly about or this set of lectures, and the real world. So I actually ran the iterative version of factorial on the computer. And you can see here I've multiplied the input by 2, so 40, 80, 160, 320, and so on.

So as I'm multiplying the input by 2, if I'm expecting this function to be linearly related to the input, I'm expecting that the time that this function takes to run is going to be approximately twice as long. If the input increases by 2, the time it takes for this program to run should just increase by 2 as well. And it does. It does all the way up to somewhere between 640 and 1,280.

So if we do the math, that's approximately times 2 each time minus-- because we're just doing times here. But then, after somewhere within 640 and 1,280, the time that it takes to run my program no longer follows this linear pattern. In fact, it starts to grow faster than linear. And at a first glance, it looks like it grows squared, polynomially. So instead of-- if you increase the input by 2, it looks like the time it takes for this program to run increases by 4 after some point.

And that's because in the real world, I've got Python running on the machine. There's only some set number of bits that my computer can hold when it stores numbers. And the factorial of some number between 640 and 1,280 becomes so large that when Python and the machine is trying to deal with multiplying these big numbers by these big numbers altogether, it's just taking a really long time to run because it can't store these big numbers as efficiently as it could store these smaller numbers.

And so in the real world what ends up happening is after some-- after I'm trying to store some really large value and doing the operations with some really large values, the time complexity goes down dramatically, right? N to n squared is a pretty big jump. And so this kind of shows the difference between theory and the real world. So in the real world, we can't store these values as efficiently as they get big.

AUDIENCE: So does that mean the, I guess, [INAUDIBLE]?

ANA BELL: Yeah, so if we use a machine that had more bits to store values, then we'd be able to be more efficient farther along. Yeah, exactly. And we could-- I guess we could-- if we had a language that was maybe doing some smarter things and storing these big values in a much smarter way, that could also have an impact in the timing as well.

But for the purposes of this class, we're just interested in the theoretical happenings here. So as the input increases by x, we expect that the time that it takes to run the program will be x times as long because we're looking at values that are really, really big in theory.

OK, so let's look at another example. So this is a factorial function that does it recursively. We've seen this function before. We just looked at the iterative version of factorial. Now we're looking at the recursive version of factorial. So what do we have? We have one base case that our code will eventually get down to and a recursive step, which is just n times factorial n minus 1.

So how do we do the analysis of a recursive algorithm? Because in this recursive algorithm, we don't have a loop. In the previous examples, we had a loop that we could definitively say, hey, this loop will repeat this many times. So clearly, increasing n will increase the time it takes for this loop to run.

So when we're dealing with recursive functions we think about how many times the recursive function is going to be called. Because when we call factorial, we have factorial of some 5 or whatever it is, and this calls factorial of 4, and this calls factorial of 3. And so we have this chain of function calls where we get down to the base case. And once we get down to the base case, we start to kick off the step that returns the result, one at a time.

So when we're talking about recursive functions what we really care about is how many times we call the function. That's our, quote, unquote "loop" for recursive functions. It's just the function calling itself to ask itself to do the work. And it does the work with a slightly changed parameter. So what we need to do is think about, how many times does this function call itself? And on top of that, is there some sort of overhead that's not constant?

In this particular case, when we call factorial recursive, we're going to go essentially theta of n times because we start with n then we do n minus 1, n minus 2, n minus 3, all the way down to 0. So effectively, we've called ourselves about n times, so theta of n. And the overhead for each one of those calls is constant because all I'm doing to n is subtracting by 1. And that's a constant thing. N minus 1 is theta of 1. It's just constant.

So the overall complexity of this is just theta of n, where n is just my input. So what we notice is that the iterative and the recursive versions of factorial are both theta of n, which means that, generally speaking, if we were trying to decide whether to implement factorial recursively or iteratively, it won't really matter in the long run because the worst-case complexity is theta of n. It's the same for both. So it will be your choice which one to actually use. So then it maybe comes down to readability or other factors.

All right, another example. So this is compound. We saw this last lecture. We actually timed it and counted how many-- actually, did we count? I don't remember. I don't think we counted the number of operations, or maybe we did. But we definitely timed it. So this function took in three parameters. So we can have to be careful which one of these parameters-- or which parameters of these actually contribute to my complexity.

So this function calculates the amount of money I have if I invest some monthly amount at some monthly interest over some number of months. So the loop here iterates through a number of months. And then everything else seems to be constant. I have one loop. So the inside of the loop is constant. I do have to double check that. But so far, so good. It's not looping anything else. It's not a function of anything else.

The loop itself, though, is theta of n months. So the overall complexity of this function is theta of n months. Or we could say theta of n, where n is equal to n months. None of the other parameters contribute to my complexity. And that's exactly what we saw when we ran the code. We ran it by changing each one of the parameters, and we saw only n months contributed to a slowing program.

If we really wanted to, we could have done this analysis in depth, as we've done last lecture, to actually count the full number of operations, or as we did at the beginning of this lecture. So total equals 0 is theta of 1. The loop is theta of n months multiplied by for operations. So i grabbing a value in range, taking multiplication, addition, and then saving that into total, that's four, multiplied by theta of n, where n is n months, plus theta of 1 to do the return. So that ends up being theta of 1 plus 4 n plus 1, which just simplifies to theta of n, where n is n months. Yeah.

AUDIENCE: Why did that [INAUDIBLE] to 1-- well, theta 1 [INAUDIBLE]?

ANA BELL: Yeah, so we're just looking at operations. We're doing calculations with interest and invest and multiplying it with total, right? But the fact that interest is bigger-- like, if the interest is $1 or if the interest is $1,000, is this going to make that line of code much slower? No, because all we're doing is a multiplication between two numbers. So that's why the inside is theta 1. But having a loop where we repeat this over and over again is going to slow the program down. Yeah.

OK, how about this Fibonacci function? So this is an iterative version of Fibonacci. I don't know if we've seen this before. Again, we could do sort of a rough, quick analysis, where we just briefly glance at every single line and ask ourselves whether it's contributing theta of 1 or something worse to our total analysis-- total runtime analysis.

So we've got this first part here, which is just constant, state of 1. Nothing here is loopy. There's no recursive going on, nothing that depends on the input in a nonconstant way. In the else, we've got this constant, again, just assigning two parameters. We've got a loop. So now this loop is going to be nonconstant. The stuff inside the loop is constant though. So the loop itself depends on n, my input. So that's going to be theta of n.

But that theta of n is multiplied by theta of 1. The stuff inside the loop is just constant. So that's theta of n times theta of 1, which is just theta of n. And then the return, of course, is theta of 1. So we could do a calculation like this, or you could just quickly scan and say, hey, I've just got a loop that looks-- that depends on n. And that's theta of n. So the overall complexity of this, if we want it to be detailed, is this, theta of 1 plus theta of 1 plus theta of n times theta of 1 plus theta of 1. But overall, that just gives us theta of n because that loop is the only thing that depends on my input.

Everyone all right so far? OK. Perfect. So now let's move on to the second easiest complexity to identify. That's the polynomial complexity. So polynomial complexity generally deals with functions that have nested loops. So if we have two nested loops that linearly depend on my input, that's going to be a function that's n squared. If I've got three nested loops that all depend on my input linearly, that's going to be n cubed.

So let's see some examples. So here I have a really simple nested loop situation. I've got a function called g. And it's going to take in an input n. So I'm going to look for everything that depends on n. Well, I've got a for loop here that's going to iterate n times. So that's theta of n. And I've got an inner for loop. So for each thing in my outer for loop, I'm going to do the inner thing n times as well.

And then the stuff inside my inner for loop is constant. So that's theta of n. And the stuff outside of my loops are-- sorry, the stuff inside my inner for loop is theta of 1. And the stuff outside of any of for loops are theta of 1 as well. So they contribute nothing to this complexity. So the only thing that I need to account for is my outer loop, which is theta of n. And law of multiplication says my inner loop is going to be multiplied-- multiply its complexity to my outer loop's complexity.

All right, so the overall complexity of this function is theta of n squared because the number of times that I'm going to do this operation is going to be n squared times. Perfect. All right, so now let's look at some examples with lists. We haven't seen those yet. So now we have to think about the input. In this case, it's going to be two lists. And when we're dealing with lists, one of the things that-- or the most common thing we're interested in is what happens to the behavior of the function as the lists get bigger?

As we saw in last lecture, the size of the elements within the list typically don't matter. But the fact that I have more elements to do stuff with does matter. So if my list now has twice as many elements, this program or most programs will probably be twice as slow.

So here's a function called is_subset. It takes in two lists, L1 and L2. I've added two little examples up here to help us figure out what this function does. So it's going to tell us whether the elements of L1 are in L2. So in the first example here, elements in L1 are 3 and 5 and 2. And L2 does have the 3 and the 5 and the 2, but it also has other stuff. That's totally fine. All the elements in L1 are in L2.

So this function will return True for those examples, those in L1 and L2. And then here's an example where it will return False. So the elements of L1 are 3 and 5 and 2. And L2 is missing the 3. So then that one will return False. The elements of L1 are not all in L2. So it's not a subset.

All right, so what's this function doing? Well, it's iterating through all the elements in L1. So it's going to first look at the 3, then the 5, then the 2. It's going to look through each element in L2 for every one of those L1 elements. So it's going to look at the 3 and the 2, the 3 and the 3, the 3 and the 5, the 3 and the 9. Then it's going to look at the 5 and the 2, the 5 and the 3, 5 and 5, 5 and 9. It's going to keep doing that.

And it's going to keep track of this Boolean called matched. And it's going to-- as long as it finds this element e1 within my L2 it's going to save matched to be True. And it's going to keep doing this until it does not find-- sorry, until it keeps finding matches-- as long-- sorry, until it finds a match. As soon as it finds a match, it breaks because there's no need for it to keep looking at the remaining elements of L2. It already found one that matches.

So this code could actually be rewritten by saying the inverse. If e1 is not equal to L2, we can just immediately return False because we've already found an element from L1 that's not in L2. So we could have rewritten this code in many different ways, but the ultimate analysis will be the same. So let's look at the analysis for this function. Well, we have two inputs. So we have to be careful about both of these inputs.

Which parts of this function depend on L1 and L2? Well, we've got an outer for loop. So what happens to the complexity with regards to this loop? Well, if I have more elements in L1, then this code-- this loop will go through more times. So this loop will be executed length L1 times. So the theta for this outer loop is going to be theta of length L1.

But there is an inner loop. So for each element in my outer loop, I'm also going to do everything in this inner loop. So in the worst case, I need to look through each element in L2 to find a match. So the inner loop will execute at most length L2 times, again, in the worst case. So the inner loop will be theta of length L2.

So the overall complexity, since I've got this nested loop situation, law of multiplication says that it's going to be the theta of my outer loop multiplied by the theta of my inner loop. So theta of length L1 times length L2. OK, everyone-- yeah, question.

AUDIENCE: It means that if you have a linear if statement, it would be an e?

ANA BELL: Yes. So like here, in this if? Yes, if the if had something like using an in, where in is linear, then yes, there would be another-- it would be like there was another loop at the third level. Yeah, so then it would be n cubed. Still a polynomial but n cubed.

So if L1 and L2 are the same length, which sometimes we put on to simplify the complexity-- we'll put this condition on to simplify the complexity, then we say that it's theta of length L1 squared. It's still polynomial complexity.

OK, let's look-- sorry, question.

AUDIENCE: If they weren't the same length, [INAUDIBLE]?

ANA BELL: Yes, if they were not the same length, you have to denote it in the terms of the both lengths, yeah. OK, let's look at another example. So here's a function that grabs the intersect of two lists. So again, I've got a little example up here, example L1 and L2. So the intersect are going to be the common elements within L1 and L2. But I'm only going to-- I'm not going to do duplicates. So I'm just going to keep the unique numbers.

So here I've got L1 and L2 contain 3, 5, 2 and 2, 3, 5, 9. So notice the 2 and the 3 and the 5 both occur-- all occur in both lists. So the intersect of these two lists is 2, 3, and 5. This example here on the right side is going to be a little bit trickier. It's kind of a unique edge case, but the code still works for that edge case. It's if I have L1 that has duplicates of some number and L2 that has duplicates of that same number, the returned list of the intersect should just be 7, that one number once.

So how does the code achieve this? So you notice a nice little structure here. I've got two blocks of code, right? I've got something here, which is going to actually help us build this list of all of the elements that are common within the two lists and then something down here, where I'm going to cull that list to keep only the unique values. So up here, this has a nested loop situation, just like in the previous example.

I have to look at all of the pairs from L1 and L2 to figure out which are common. So this for loop over L1 is going to go through the 3, the 5, and the 2. And then the inner for loop through L2 is going to basically match-- take a look at, does the 3 match the 2? Does the 3 match the 3? Does the 3 match the 5? Does the 3 match the 9? And then the 5 match the 2, 5 match the 3, and so on.

So that's what those loops are doing. And as soon as we find a match, we're going to append it to this temporary list. And it's OK if we have duplicates in this list. So if you look at the example on the right-hand side there with the 7 duplicated many times, that's actually going to create a temporary list that's going to contain nine times that 7.

So it's going to look at the 7 with the 7, and it's going to say, hey, that's a match. Let me add it. Then it's going to look at the 7 with the middle 7 in L2, and it's going to say, let me add that. And then it's going to look at the first 7 in L1 with the last 7 in L2, and it's going to say, let me add that. And then it's going to do that same thing all over again when it looks at the middle 7 in L1 along with each element in L2. So it's going to add the 7 three more times. And then again when it looks at the last 7 in L1 along with each 7 in L2.

So that's totally fine. That's just what this code is doing. And then the bottom part down here is going to take this temporary list that we created, and it's going to just keep the unique values within it. So it's going to keep that-- create that unique list. And it's going to say, if I haven't seen this value in unique, add it. And if I have, don't do anything. So in the end, this code down here is going to take that big list here and just keep the unique values.

All right, so let's do the analysis for this. So I've got my outer for loop and my inner for loop up in the top half of my code here. That generates my temporary-- long temporary list, potentially long temporary list. So that we already know from the previous example is theta of length L1 times theta of length L2. Pretty simple. Now, what about this bottom half here? Because we have to be careful about this bottom half. This one could also contribute to the complexity.

It's looping through a temporary variable, a list variable that we created. But this list is created by doing something to L1 and L2, by looking at elements in L1 and L2. So it's actually indirectly related to L1 and L2. So we can't just cast it aside because it could potentially contribute to the complexity of my program.

And in the worst case, I create this temporary variable that looks like this. So in the worst possible case, my temporary variables length is going to be length L1 times length L2. I basically added that character every time I compared a value. So this list at worst case is length L1 times length L2 long.

So if I'm iterating through that list, then the complexity of that second half is also theta of length L1 times length L2 in the worst case, right? So the overall complexity of the function is theta of length L1 times length L2 up here plus theta of length L1 times length L2 down here.

So in this particular case, the fact that I'm iterating over temp didn't actually increase my complexity, but you can imagine code where doing something funky like this where you indirectly have some loop over something related to the input could affect the complexity. So in this case, the overall complexity is still theta of length L1 times length L2. Questions about this one? OK. Yeah?

AUDIENCE: Why is it not-- again, because [INAUDIBLE] you're appending a certain number of them, how do you know that's-- like [INAUDIBLE] vary for each problem.

ANA BELL: It varies for each problem, right? But in the analysis, we're interested in the worst-case scenario, like the asymptotic behavior of the worst case. And in the worst case, we've added this number, length L1 times L2 times. Most of the time, of course, it's not going to be this bad. It's just in this one particular case that it is this bad.

AUDIENCE: Oh, I see what you're adding, OK.

ANA BELL: OK, let's look at one more function that's polynomial. So here's diameter. We saw this last lecture. Basically, if we have a bunch of points in a 2D plane, this function tells us the distance-- sorry, the maximum distance between any two points. So I drew that picture in the 2D plane. So this one is going to have nested loops again. So the outer loop iterates through length L times.

So remember, our L is just a list of tuples representing these xy coordinates. So the outer loop easily goes through length L times. But what does the inner loop go through? The inner loop is actually starting at i not 0. If it started at 0, the inner loop would be clearly theta of length L. But it's not. It starts at i. On average, though, how many times does that inner loop go through?

Well, the first time it goes through that inner loop, it's going to look at length L minus 1 elements. Next time it's going to look at length L minus 2 elements. Next time it's going to look at length L minus 3 elements, until we get to the end, where it's going to look at 1 and then 0 elements.

So if we think about how many times that inner loop actually iterates, it's going to be, what is it, like length L minus 1 multiplied by length L over 2. Is that the function, I think, to add all these together? Something like that, which is basically still something that's a function of length L. Like, we can simplify it to be 0.5 length L. So it's still a function of length L because the coefficient in the front of that length L is 0.5.

So the overall complexity of the inner loop is still theta of length L. All right, everything else within this code is constant. So the overall complexity is just theta of length L squared. Yeah. Sorry? Where did the 1/2 come-- oh, it's the formula to add-- like, if you add 1 plus 2 plus 3 plus 4 plus all the way up to n, what's the formula to do that? I think it's like n times n plus 1 over 2, something like that.

So this is not exactly half, but it's something on the order plus, I don't know, something, whatever this calculates to. But in effect, it's something that's smaller than length L, but it's still a function of length L. And so that front coefficient right before length L just goes away. Even if it was 10, we would still cast it away. In this case, it's 0.5 or whatever it is. So it's still less than 1, but we still cast it away because we're interested in the theta of this. Yeah.

AUDIENCE: [INAUDIBLE] nested for loops-- with two for loops, that length [INAUDIBLE] an n squared [INAUDIBLE]?

ANA BELL: I mean, the inner loop could just not depend on the input at all. Like here it's n squared because both of the loops depend linearly on the input. But if the inner loop-- like if the outer loop went through a range length L squared, then the overall complexity would be length L cubed in this case because it's length L squared times length L.

Or if one of the loops doesn't depend on the input at all, then it contributes nothing constant-- nothing linear, so it's constant. OK, let's have you think about this question for a bit.

So think about the input. Think about parts of the function that depend on the input. And then what is the complexity? OK what's the outer loop theta of? Yeah.

AUDIENCE: [INAUDIBLE]

ANA BELL: Yes, nums is a list, so the outer loop is theta of length of nums. Correct. Good. What's the inner loop theta of? Yeah. Is that what you were going to say, theta of 1? Exactly. It's the length of digits, but digits is not my input. Nums is my input. So the inner loop will always just iterate through 10 times. So in the eyes of the inputs to the function, that's just constant.

So the input is nums. The outer loop is theta of nums. The inner loop is theta of 1. So the overall complexity is theta of length of nums. Perfect. How about this one? What are my inputs? Do any loops depend on these inputs? What's the outer loop complexity? Yeah.

AUDIENCE: Length of L1.

ANA BELL: Yeah, theta of length L1, exactly. What's the inner loop complexity?

AUDIENCE: Theta length of L2.

ANA BELL: Theta of length of L2, perfect. And is there anything else that contributes to the complexity here? What's that?

AUDIENCE: The if statement.

ANA BELL: The if statement. Yes, what about it is making you question that the complexity is not constant?

AUDIENCE: So in iterates through the length of L3.

ANA BELL: Exactly, yes, very nice. So in iterates through the length of L3. Looking for an element in L3, e1 in L3, is not constant. You have to look through the whole length of L3 to figure out whether it's there or not. So this inner bit here is not constant. It's theta of length L3. In fact, it's two times length L3. So the overall complexity of this function is theta of length L1 times theta of length L2 times theta of length L3.

OK. Cool. Let's look at the exponential complexity. So this is a complexity that grows really, really quickly. We never want the algorithms that we write to land within this class. Unfortunately, there are just some problems in real life that we have to compute that are just naturally part of this complexity class. There are some techniques to deal with making these algorithms a little bit faster.

But inherently, there are just exponential algorithms that we just can't do any better than exponential in solving these problems. All right, so let's look at Fibonacci again. We looked at Fibonacci a few slides ago, iterative version. And the iterative version was theta of n. But if we look at the recursive version of Fibonacci, it's not theta of n at all. In fact, as you can see, it's in this exponential set of slides, the recursive version of Fibonacci is actually exponential.

So let's recall what this code is doing. So there's two base cases, Fibonacci of 0 and 1. And then the recursive step is Fibonacci of n minus 1 plus Fibonacci of n minus 2. So for every level that we go down, there's going to be times two more paths that we need to explore to grab the values from.

So for the very first n, we've got just one value to grab. For the next n, we've got times two that value to grab. The next level-- for the next n, we've got two times more values to grab. And so on. So the fact that there are two recursive calls in this recursive step leads us to this little inverted tree kind of structure. And we even drew this when we looked at how many function calls are being run.

Remember, when we're figuring out the complexity with a recursive function, we need to figure out how many of these functions-- how many recursive calls are we actually doing. So because of this tree structure, every time we add a new level, we basically have two completely separate paths to explore further. And those two paths have their own two paths and so on. So this leads us to this tree structure, which is actually going to lead to the total number of recursive calls to be exponential, so theta of 2 to the n.

Now, if we looked at the actual recursive call's tree-- right, we looked at this, and it looked something like this a bunch of lectures ago-- you might notice that the tree actually thins out a little bit to the right. It's not a full tree with the leaves nicely all the way down. And that's because, well, the left path calculates Fib 5 but the right path calculates Fib 4, so n minus 1 of the left path. But that's fine.

It's not that we are actually going to speed up anything by some sort of order of magnitude. Just because the tree thins out a little bit on the right-hand side is not going to speed up the overall complexity of this function. It's going to be theta of 2 to the n minus some theta that's less than 2 to the n. So that subtraction is not going to really decrease the overall complexity of our function. So the order of this is still exponential.

All right, here's another example of an exponential code. So this is a function that is going to generate all the subsets of a list. So again, I've added a little example here to help us understand what it's doing. So here I've got three numbers, a list with three numbers, 1, 2, and 3. And to generate subsets, what this means is that I'm going to create a new list of all of the possible combinations of numbers within my original list, of all the possible lengths.

So first, one subset of this list could be just the empty list. So that's not taking any of my original numbers at all. The next one is a list with just one of the numbers in it. So either the 1 or the 2 or the 3. A next subset of my list could be taking just two of the elements. So one and 2, 1 and 3 and 2 and 3 and then lastly I can just grab all the elements. So 1 and the 2 and the 3. I don't care about the order. I just care that I have all of these different combinations of all of the different lengths in my final list.

So does everyone understand the goal of this function? OK, so how do we achieve this? Well, you might not be surprised. We're going to do it recursively. That's really the only reasonable way to write this code. So I'm going to go through this slide just explaining what each line does. But on the next slide, I'll have a little animation that shows step by step how the function creates this subset list.

So first thing, it's recursive, so I've got my base case up there. If I have a list of length 0, then the subset of an empty list is just going to be this list with the empty thing inside it. So if I have no elements, there's only one subset, that's the empty list. Then if I have more than one element inside it, I'm going to do the same idea that we saw when we worked with lists back in the recursion lectures.

I'm going to extract one of my elements. I'm going to work on the remaining list. And then I'm going to do something by taking that element and tacking it back on to the result. So in this particular case, the thing that I'm extracting is the last element in my list. So if my list is 1, 2, and 3, at a step here, I'm going to extract the 3 and make it into its own list. So that's what that step is doing. It extracts the last element in the list.

Then I make a function call to generate subsets on everything except for that last element. So if I'm-- so I say, hey, function that I'm currently writing right now, if you can generate for me the subset of all the elements, the subset for this list, then you're going to come up with something that looks like this. It's going to be the empty list, the 1, the 2, and the 1 and the 2 together.

So the subset of this list is going to be this group of elements here. So that's what this is going to do. So this is, again, us trusting that the function we write will generate something that looks like this. If we've got to this point, then smaller is going to be a list that looks like this. So the next part of the code is going to take that little extra thing that I had saved previously, and it's going to tack on that 3 to every element within this list.

So then I'm going to basically say, I'm going to take this 3 and make a list with the 3 in it, a list with the 1 and the 3 in it, a list with the 2 and the 3 in it, and a list with the 1 and the 2 and the 3 in it. So I've just taken that 3 and added it to everything that resulted from this line of code here, from my function calling itself.

And then all it does is returns smaller plus new. So if I add these two together, this is going to generate for me my final subset that I was interested in. I've got the empty thing. I've got the 1, the 2, and the 3 by itself. I've got the 1, 2; the 1, 3; and the 2, 3 by itself; and then the 1, 2, 3 altogether. So that's the big idea here.

So let's just go through it step by step, recursively calling ourselves. So this is me finding out the-- kicking off my function call, saying, hey, generate the subsets for the list 1, 2, 3. I'm going to keep the extra aside. I need to make another function call because I'm not at my base case. So I'm going to call gen_subsets on 1, 2.

This is also not my base case. So I'm going to take my last element, put it aside, and I'm going to call gen_subsets on just the 1. Still not the base case. I'm going to take this extra, put it aside, and I'm going to call gen_subsets on the empty list. And this is where I reach my base case. So far, nothing has been returned at all. No work has been done.

At my base case, Python will say, I know what this is. It's going to be the list with just the empty thing in it. All right, cool? Next, that gets returned, so this function call goes away. So now what is it going to do? Well, it's going to take that extra I set aside, take the smaller list that I just returned, and basically double that smaller list. So this is my smaller list. And then I'm going to double that by saying I'm going to put this 1 to the end of everything in my smaller list.

Maybe this is not so apparent at this step, but let's go one more step and see what happens. So now this function also terminates. It returns this empty list and 1 in it and says, all right, here, with this function call, I had saved the 2 separately and said, I'm going to now tack on this 2 to the end of everything that I had just returned. So this is smaller. This is smaller over here.

And all I'm going to do is take this extra thing and tack it on to the end of everything that was in smaller. So I'm going to tack it on to the end of this empty list, so it just gives me this 2, and tack it on to the end of this 1 so it gives me the 1, 2.

So I've basically doubled my list at this stage. One more step, this gets returned. And now this is my original function call. The thing that I had extracted was the 3. So now we're basically at this step here. I extracted the 3. The function just below it returned this smaller, right? So that means that this 3 is going to get appended to the end of everything that was in smaller.

So it's going to be appended to the end of this empty list to give me just the 3, to the end of the 1 to give me the 1 and the 3, to the end of the 2 to give me the 2 and the 3, and to the end of the 1, 2 to give me the 1, 2, 3. Now, this is the final answer. So I basically keep what I had returned from the previous function call and concatenate that with the thing that I had just created, where I tacked on my 3. And this is my final answer. It's just out of order to what we intuitively would have written by hand.

But it hits on all of the elements that I wanted to have anyway. So I've got the empty list, everything with just one element in it, everything with the two elements in it, and everything with all three elements in it. So let's look at the complexity analysis of this. We've got two things going on here. One is how many of these function calls are actually being done, like with the inverse tree structure. How many of those function calls do we need to do to get to the end of-- to our base case?

And on top of that-- sorry, that will tell us how many actual elements in the list we will have. And on top of that, we have actually a time complexity that's not constant. That's to copy our list. So copying a list is not constant because it takes some time to take all the elements in a list and make a copy of them.

So if we think about the time it takes to make our list at each step, how many of these sub elements we're creating, well, at the very base case, we have one element; at the case just above it, we had two elements; at the case just above that, we had four elements; at the case just above that, we had eight elements. So at each step, the number of sublists that we were generating was basically twice as much as the previous step.

So the overall number of subsets was on the order of 2 to the n. But there was also a time complexity to make a copy of the list within each one of those subsets. So we're multiplying the complexity it takes to make all those function calls and generate all those subsets by the time it takes to make a copy of the list. So the overall complexity is actually going to be theta of n times 2 to the n because it's a little bit harder-- it's a little bit worse than exponential, just purely for the fact that we're copying the list at each step.

All right, so let's move on to logarithmic complexity. This one's going to be a little bit tricky because right off the bat, we're not going to be able to see a direct relationship between the input and what loop we actually have. So here I've got a function called digit_add. It's going to take in a number, so like 1234, something like that, number 1,234.

The code casts it to a string. So it takes in a pure numerical value. It makes a string out of it and then iterates through the string. So the function here, in terms of time complexity, is theta of length S. Here, we're iterating through the string backward basically, 4, then 3, then 2, then 1. But what's my input? It's n. It's not s.

So the time complexity of this function, while it's linear in s, s is not linear in n, because when my number is 83, my loop only iterates twice. If my number has four digits in it, 4,271, my loop iterates four times. So this relationship is not linear. So what is it exactly? Well, let's think about what that loop is actually doing.

If I have a number with four digits in it, something in the thousands, when I iterate through the number sort of backward, this number as a string, I'm basically taking that 1 and keeping it in my running sum. Then it's kind of like I divided that number by 10. I grabbed the remainder when I divided that number by 10. And that's the thing that I just added. The whole number left over when I divided by 10 is this bit here.

So now think of it like taking this-- to take this last element here, it's like I take this number and divide by 10 again. I grab the remainder when I divide by 10 and add it to my running total. And the whole number I'm left over when dividing by 10 is just this. One more time, I take the 2. The remainder when I divided that 42 is 2, and the whole number I was left over with is 4. And then lastly I can do that last thing again.

So what's the relationship between the magnitude of n, this 4,000-something, or this 80-something to how many times I have to loop through to get every digit in my number? Well, the trick here is to think about taking my magnitude, my n-- the magnitude of n and dividing it by 10 a bunch of times. How many times do I divide by 10 to basically grab every single element, every single digit in my n?

Well, length s times, right? That's kind of like taking each character one at a time. To take each character one at a time, that's like dividing by 10 to grab the remainder. And then I've done that length s times. That's what this loop is doing. So the relationship between the magnitude of n and how many times I go through the loop is this, n divided by 10 some number of times, length s times, is equal to 1. That means I've finished going through this entire element-- this entire number, all the digits within the number.

So the relationship between n and length s is length s is equal to log of n. And now that I have this nice relationship, well, I said that this function was linear in length s. So if it's theta of length s, it's going to be theta of log n. I just mapped those two together.

Questions about this? This trick can work in many different ways. What's important to realize is that here there's kind an indirect relationship between what's actually happening in the code and my input. It's not as clear cut. But there is some relationship, which is not constant and not linear.

OK. So the overall complexity of this function is theta of log n, where I don't actually care about the base when I report the complexity in terms of log. In this case, it's base 10. But if it was base 2, it would be the same, log n.

OK, so we saw a bunch of examples, just one of logarithmic complexity. But we're going to see next that searching for an element in the list will also be logarithmic complexity. Before we get to that, I'd like to just make-- just put this slide up to remind you that there are several functions, built-in functions, with lists and dictionaries that aren't constant.

So like the example you guys did where we used the in operator, we had to be careful-- if you ever see these operations being done in the code, don't just push them aside. You have to account for them within the complexity analysis.

OK, so next we're going to look at some searching algorithms. These algorithms, we're going to see a bunch of different codes that implement searching. These will-- again, they'll be very similar to the ones that we actually timed last lecture. So we're going to look at searching for an element in a list. We're going to look at a bunch of different implementations of the plain brute force searching element in a list.

Whether it's sorted or unsorted, as long as you just brute force your way from the beginning of the list to the end of the list, you'll be able to find the element you're looking for or say that it's not there. So we're going to look at some linear search functions. And then we're going to look at the bisection search-- a couple bisection search implementations. And that's where we divide the list in half and discard one of the halves. And those implementations though will need our list to be sorted.

So the brute forcing our way doesn't really matter whether it's sorted or not, but the bisection search only gives the correct answer if the list is sorted to begin with. All right, so first, let's look at linear search on an unsorted list. This is code that is going to search for element e in list L. It loops through the length of the list and keeps this Boolean flag in mind. If it finds the element we're looking for, it just sets the flag. And at the end of iterating through the whole list, it tells us whether it found it or not.

So the worst-case scenario analysis says that we have to look through the entire list to determine the element is there or not. So the theta of this particular function is theta of length L. There's only one loop. It depends on the length of L, nothing really special about this function. Now, you might notice that there's something inefficient about this function and that once it finds an element, let's say at the beginning of the list, this function actually just sets the flag and keeps going through to the end of the list.

So we can actually do a little bit of a speed-up with this bit here and say that, hey, if we find it, just return True right away. No need to keep going to the end of the list. So what's the analysis for this code? Well, again, we're doing worst-case analysis. So in the worst case, the element is not there. So we still have to search through every single element in the list, beginning to end, to determine it's not there.

So the worst-case theta analysis for this function is that we still have to go through to the end of the list to determine it's not there. So it's still going to be-- sorry, it's still going to be theta of n-- sorry, theta of length L time. So this is on an unsorted list. But what if we look at a sorted list?

OK, so we can do a little something clever in our code. If the list is sorted, we can say we're going to start-- let's say it's increasing sorted. We can start at the beginning of the list, look through each element. If we find it, return True. If we reach an element that's bigger than the one we're looking for, if the list is sorted, so all the remaining elements in the list are also bigger than the one we're looking for, right?

And then we can just return False right away. Well, we think we're pretty clever, but the worst-case analysis says that the element is not even in the list at all. So we still have to go through and look to the end of the list to figure out that that element is not there. So we still have to touch each element in the list to determine it's not there. So the worst-case theta complexity analysis still says that this is theta of length L because everything else is constant.

So now let's look at bisection. So as far as we can tell, just doing a linear, brute force search way is not going to give us anything better than theta of n. But when we looked at the timings in last lecture, we saw that this binary search or bisection search on an element in a list was actually much, much faster. It grew at a faster rate than linear but not quite constant. So let's remember how that code looked.

So we basically had a list with a bunch of elements in it. We looked at the element at the middle of the list. And we said, are you the one we're looking for? In the worst case, it's not. So then we have to ask, are you bigger or smaller than the one we're looking for? If it's bigger, then we know we have to look in the lower half of the list. If it's smaller, we look in the upper half of the list.

And now that we either look in the lower or the upper half, we notice we have the exact same problem to solve. So this should ring a little bell that says we should use recursion. Now we have the same problem to solve, an element e in a slightly smaller list. Is it in that list? So that's exactly what we're going to implement.

So visually speaking, this is what we're going to do. We're going to have an original list with n elements in it. We're going to look at the halfway point. Worst case, it's not the one we're looking for. So we're going to decide on one of the sides to next search through. Now we have n over 2 elements to look through. Again, it's not there, worst case. So we have to decide on which half to look through. Now we have n over 4 elements to look through. We keep doing this. We keep of halving more and more recursive calls until we reach a base case.

And the base case is that we now have a list with one element in it. Either that element is the one we're looking for or, worst case, it's not, and we've determined that the element we're looking for is not in these n elements at all. So our base case is down here. And we started with n elements over here. So the bisection search algorithm will repeat this task of dividing the list in half let's say i times.

So this is quote, unquote "how many iterations we would have made." But since this is recursion, there's no iterations. This is how many function calls we have until we reach the base case, i function calls. So if we take our original n elements and we divide them by 2 so many times that we have only one element left to search for that's when we found our answer.

So we now have a relationship between how many elements we had originally, n elements, and how many times we had to divide our loop to get to our answer, how many of these levels we have. N divided by 2 to the i equals 1, that's our relationship. So in the bisection search algorithm, how many times are we calling this recursive function to get to the base case? Well, i times.

So what is i in terms of n? Well, the relationship between i and n is similar to the one we had over here, where we divided this number by 10 each time, except that now we're dividing a list of n elements by 2 each time. So the relationship is still logarithmic. It relates the number of elements I originally had, n, with how many times I had to divide my list to get to one element, whether it's the one I'm looking for or not.

So the complexity of just the pure bisection search algorithm is theta of log n, where n is the length of the list. That's how many subdivisions I need to do to get to one element to decide it's not the one I'm looking for.

So now we're going to look at two different implementations of the code to do bisection search. One will be more efficient than the other. Let's start with the one that's simpler to write but less efficient. So this code, you can see here it looks for element e in list L. It has two base cases up there, those are both constant, and one recursive step here. So either we do this one or this one.

So this one is if we decided we need to look in the lower half, and this is if we decided we need to look in the upper half for the element. So this is just pure bisection search, which on the previous slide we decided is theta of log of length of the list, theta of log n. Now, that's fine, but what do we have as a parameter here? It's half of my list.

So in addition to doing bisection search and just doing the algorithm, having a bunch of bisection search calls that take me to that list of one element, on top of that, each time I make that bisection search call, I'm copying my list. So this is not constant. It's theta of length L over 2, right? I grab half of my list.

So the complexity of that code is theta of n times log n. Theta of log n for the bisection search bit, but theta of n tacked on to each one of those calls because I have to grab a copy of my list with each function call. So it's not quite that efficient.

Now, let's look at a slightly different implementation. This particular one is going to use integers to keep track of endpoints. So instead of copying my list, let me just keep track of a number for my low endpoint and a number for my high endpoint. The complexity analysis for the bisection search is going to be exactly the same because even though I'm just keeping track of these high and low end points, I'm still dividing the list in half with each call.

But I'm doing it by keeping track of integer indices. So the size of the problem is still reduced by two at each step. I'm keeping track of these integer indices. I'm not copying the list at this point. I'm just changing an integer value from 10 to 5 or whatever it is. So the complexity analysis of the bisection search is theta of log n.

The code looks a little bit messier, but overall, it still does the same sort of things. It's messier because now I want bisection search to look for an element e in list L, but I'd like my recursive call to keep track of two end points, these integers low and the integer high, the thing that I want to search my list between.

So I'm going to create another function that I kick off down here, which looks for an element in list L. But I'm also going to keep track of my low and high end points as parameter to my bisection search function. So bisection_search_helper here is now going to take in these four parameters. The rest of the code is just details. But what's important is everything is constant except for my two bisection search calls.

Here, I'm changing my low-- I'm sorry, I'm changing my high if I want to look in the lower half of the list. And here I'm changing my low if I want to look in the upper half of the list. So the bisection search calls are still going to be theta of log n, but what's the overhead now? The overhead is nothing, right? It's constant. This L is the same one. I'm not making a copy of it. I'm just passing it through. E is just a number. Low is just a number. And mid minus 1 is just a constant operation. There's nothing being copied here.

So the overall complexity of this code, while it looks a little bit messier, is just theta of log n because the overhead is constant on each one of those function calls.

So that brings us to this final question. Clearly bisection search on a sorted list is faster, it's theta of log n, than by pure brute force search on a list that could be sorted or unsorted. So the question is, when does it make sense to sort the list first? So given an unsorted list, when do you sort the list and use this fast binary search versus just using a straight-up linear search?

Well, that's when the time it takes to do the sort, an initial sort, plus the complexity to do binary search is less than doing the straight-up linear search because the list has to be sorted for this to work. Well, when is that true? Well, this implies that the time it takes to do the sort is less than theta of n.

So that means what? Can you sort a list without even looking at all the elements once? No, right? You have to look at all the elements once to even say that, hey, this list is already sorted. So this is actually never true. So what does that mean? Does that mean we never want to do binary search on a list unless it's already sorted? Kind of.

But in fact, there are various situations when it does make sense to do the sort first and then use binary search. And that's the case where you're given a data set, and you want to do a whole bunch of searches on that data set.

So if you can take that sort, do it once, and then amortize the cost it took you to do that sort over K different searches, then it makes sense to pay the price to do the sort once and then do it over-- and then do the binary search over all these searches-- yeah, all these searches. And so as K gets really big, the time it takes for you to do the sort becomes irrelevant.

The theta of doing this thing on the left becomes just the theta to do the search logarithmically than it does to do the search linearly. OK, so if you're only doing the search once, please do not sort your list and then do a binary search. That's going to take longer than just looking at the elements in your list straight through using brute force. But if you're going to do a whole bunch of searches, it makes sense to do the sort and then do the search.

All right. All right, that's all I've got. Next lecture, we're going to look at a bunch of different sorting algorithms, and we'll have a quiz.