[SQUEAKING] [RUSTLING] [CLICKING]

ANA BELL: All right, let's get started. Last lecture, we began talking about an entirely new topic in computer science. And we have begun learning about how to figure out the runtime of our programs. So we did-- we looked at how to actually time the program by figuring out exactly how long it takes. And then, how to count the number of operations in the program.

Today, we're going to do very same thing to begin with. So for the first half of the lecture, we'll time a bunch of programs and then we'll count the number of operations, just like before. But we're going to do them in the context of slightly different, slightly more interesting programs or functions involving just pure numbers as our parameters and then functions that involve lists as our parameters.

That will be the first half of the lecture. And then, from there on, we're going to look at the idea of order of growth, which is kind of what we're building up this set of lectures to. And in the order of growth, there'll be a little bit of math, a little bit of graphing, but not too much. And then, we're just going to see how to actually evaluate the order of growth of functions from there on out.

OK, so let's begin by just figuring out the runtime of our programs, right? This was a really quick and easy way for us to figure out exactly how long our programs take. So last lecture, we imported this time module, and we're doing that again this time.

But instead of actually running the time function that we had seen last lecture here, instead of running the time function which gave us this sort of global absolute time since some date in the past, we're going to run this slightly different function called performance counter. And this is what is typically used in the real world to figure out how long an actual program or a function takes to run. The reason we're using this is because it's more accurate.

So the time dot time function that we used last lecture gave us, maybe, precision to 1 times 10 to negative 3 or something very, very big like that. The performance counter can actually give us precision to something that's a lot, lot smaller. So maybe 1 times 10 to the negative 8 or something very small. So we'll be able to see the timings of some functions that were basically 0 in the last lecture.

OK, so just a quick review of how we actually get the time that a function takes to run. We run this performance counter time, and this one gives us not an absolute time, but more of a shorter time frame, not from some time in the past.

And the performance counter is very useful when we're getting these DTs, right? The difference in some times. So we're running the performance counter to get the quote unquote "starting time." We run the function. We run the performance counter again to get the quote unquote "stopping time." Subtract the starting time to get the DT.

And then, we will print that time to see how long the function actually takes to run. Yeah, that's what I said. OK, so we're going to look at two different functions than last time, but they're going to have the same overarching themes that we saw last lecture.

So the first function we're going to look at is called convert to kilometers taking in some miles and returns the value in kilometers. And the second function is a function named compound. So this one should seem very familiar. It will bring flashbacks to problem set one.

It's a function that takes in a monthly investment, an interest rate for the month, and some number of months to invest that much. And it returns how much money you've made over those number of months. So you can see here you have in total initialized, a loop that goes through that many months, and it updates the total based on the interest rate and how much money you have there right now plus whatever you've invested for that month.

OK, so the three questions we're going to answer, just like we answered last lecture is, how long in actual seconds does it take to run these functions? Which input parameters does the function actually depend on? And do these two functions actually run for different amounts of time? And what is that difference, right? Does one run in 12 seconds and the other one run in 0.5? What is the actual time that it takes for them to run.

So this is our code. So these are the two functions. Before we go on, let me just show you how we're creating the inputs. So just like before, we're creating a list of all of the different inputs we're going to run the function with. So here, I've got this ln. That will contain the numbers 1, 10, 100, 1,000, and so on. And these are going to be the parameters to my function. One at a time, of course.

And then, I've got my loop here for each one of those inputs, 1, 10, 100, 1,000. I'm just going to run my function, right? So here, I'm measuring the time it takes. And then, I'm going to report the time that it took to run the program. And just for fun, I'm also going to report how many times this program could run in one second. Because for me, it was a little bit hard to read 1 times 10 to the negative 8 or something like that. But it was a lot easier for me to see this big number for how many times that function could have run in one second.

So here I've got convert to kilometers. So I'm going to run it. And we're going to see it's this right here. How long the function actually took. So last time we ran a program that was really simple like this, all of it basically said it took zero seconds, right? It was just so fast that time dot time function wasn't able to pick up that precise time difference.

But this performance counter can, which is a lot nicer. So now, we see that no matter what the input it looks like, the time is pretty much the same, right? 3 times 10 to the negative 7 seconds no matter what the input is. That was expected.

Now, what about the compound function? This one's going to be a little bit more interesting. Because there are actually three parameters to this function, right? What we're going to do is change each one and see which one of those parameters actually has an effect on the runtime.

So here, this bit, is going to fix my interest rate and fix the number of months. And I'm going to change the amount I invest every month. OK, so if I run that, that was pretty fast. Again, we look at the results here. And no matter how much I invest every month, it looks like the program doesn't really change how long it takes to run, right? It's always about 1 times 10 to the negative 6 seconds to run.

All right, what if I change the interest rate? So this one was a little bit harder to change. But I settled on this as the thing I'm varying. Sorry, I'm varying it in this way. So it's going to be 1.1 or 1.0 or 1.001. That's what I'm going to invest-- the interest rate for whatever I'm going to invest in. And I'm going to fix $10 as my investment per month and fix the 12 months, again.

So if I run that, same deal. It looks like changing this investment isn't really making much of a difference in how long it takes the program to run. One last parameter to try. So now I'm going to fix the initial investment to $10 a month and I'm going to fix my interest rate to this per month. And I'm going to vary how many months I'm going to invest this.

So again, this n will be 1, 10, 100, 1,000, 10,000, and so on. So let's see what this is going to do. Already it's doing something different than the other two because it hasn't finished running yet, right? So it's still working on this last one down here. But we can see that more interesting things are happening now, right?

So here I've got-- initially, it's a little bit hard to tell for those small numbers, which is fine. But luckily, we're able to run it for a bunch of inputs. So starting from about here when I start investing-- sorry, when I start investing my money over 1,000 months, 10,000 months, 100,000 months, and so on, it looks like we can see a pattern.

Again, for 1,000 months, it takes-- the program takes about 5 times 10 to negative 5 seconds to run. If I increase the number of months by 10, it takes times 10 to the negative 4 seconds to run. And then, as my input increases by 10, the number of months, my time to run seems to increase by 10 as well, right? So 0.005, 0.05, 0.7, 0.8, something like that.

OK, so this is from a previous run. Of course, each run will be different because we're just purely grabbing the time that the program took to run, so the actual time will be different. But a few things to notice. So Python actually reported the time it took the program to run in scientific notation, which is kind of cool.

So this is 4.3 times 10 to the negative 6. So it knew how to show it to me like that. So it's not-- it doesn't have a bunch of zeros in there. And then, the observation as we might have-- as you might have guessed is-- for this convert to kilometers was independent, right? So this is the kilometers, not the compound function.

But then, the compound function here, this is, again, from a previous run, we can make a few observations. So the first was that the time only actually changed with the input when we changed n months. When we changed the initial investment or the interest rate, the program just basically took the same amount of time to run. So it was only n months that actually made a difference for us.

Second observation, again, something we noticed, is as we increase the number of months by 10, the time it takes the program to run also increases by 10. Again, something we've talked about. And the last observation is that we have this relationship very apparent as the input is really big, right?

As the input is small, I think I mentioned this last time, if, for some reason, my computer updates or decides to dedicate some resources to running an app in the background for whatever reason as it's trying to figure out the compound function with an input of 1, this number could be changed dramatically, right? Because 2 times 10 to the negative 6 can be affected a lot by just a little bit of time dedicated to something else.

Whereas, 4 seconds or 14 seconds, if my computer dedicates a little bit of time to something else, that 4 or 14 won't be affected as much, right? So when the numbers are big, that's when we can see the behavior of our function a lot more clearly. Not when the numbers are small.

OK, so now I'd like to look at some more functions. These functions are going to have the input being a list as opposed to just numbers. We've seen a bunch of examples with numbers. But let's see what happens when my input is a list.

So here's a very simple function. It takes in a list L. And it sums all of the elements in the list L. So we've seen this a bunch of times already. We initialize the total to be zero, we iterate through each element in L, and we keep our running total by just adding the element to that total-- pretty simple-- and we return it.

Now, how do we actually run this function with a whole bunch of different inputs? Well, that's what we're going to see next. So this bit here is exactly the same as before. It's actually creating for us the list of 1, 10, 100, 1,000, 10,000, and so on.

But clearly, the number 10 cannot be an input to this function because this function is expecting a list. so L cannot be 10. It needs to be a list with some things in it. So instead, what we're going to do out of that number, 1, 10, 100, 1,000, and so on, we're actually going to make a list with just some random dummy numbers in it. I don't actually care what these numbers are.

So let's just make them be the numbers 0 to 9, 0 to 99, 0 to 999. And in effect, each one of these lists will then have one element in it, 10 elements in it, 100 elements in it, 1,000 elements in it, and so on. All right, everyone OK with that?

All right, so the input is now different. It needs to be a list. We're just creating a bunch of lists of varying lengths. So the relationship between these lengths are that the lists are 10 times as big as the previous list, right?

So then, now I have my input list here. I do the exact same thing as before. Not yet. I run the performance counter to create my starting time. I run my function and I get the DT and I report the exact same thing as before. All right, so let's run that down here.

Running, running, running. Again, we have to wait a little bit. It looks like it's working, but it's just getting slower and slower, which is fine. So what do we notice? So when we had one element in my list, it took 1 times 10 to the negative 5 seconds to run. When I had 10, it took 1 times 10 negative 6 seconds to run.

It was actually shorter to have more elements in it. See this is what I mean when it's very unpredictable for low numbers. But as we get to lists of length 1,000, 10,000, 100,000, a million, and so on, we can start to see the pattern. So with 10,000, it took 4 times 10 to the negative 4 seconds to run. With 100,000, it took times 10 to the negative 3 seconds to run.

And then, as the input increases by 10, that is the length of my list increases by 10, it looks like the program takes 10 times as long to run. OK, a very similar thing as before. So the first observation that we can make out of this function is that the size of the input, obviously, is now the length of our list, right? It's not just the number 10 or the number 1,000. It's a length of-- list of length 10 or a list of length 1,000.

Second observation is that, again, just like in the previous case, the average time increases by 10 as the length of our list increases by 10. Again, very good. And just like before, this relationship between the size and time is more predictable for large sizes than it was for small sizes. As we just saw here, which surprised me a little bit as well is if for us a list that's longer, it took a shorter amount of time to run, which is counterintuitive. But again, that's probably because my computer did something here to take a little bit longer to run it.

And then, the last observation, this is compared to the compound function where we change the number of months. It looks like the time that this program actually takes to run is pretty comparable-- just in terms of pure seconds-- is comparable to how long it took for the compound to run.

So here, when my input was 100 million I think? Yeah, 100 million. It took about 8 seconds to run. And when my list had 100 million elements in it, it took about 7 seconds to run, right? And then, 10 times faster going-- when we decreased our input by 10, OK?

So already, we're starting to see something that we're going to get-- yeah, go ahead.

AUDIENCE: What is the sum_of function?

ANA BELL: Oh, sum_of is just this function that I wrote here. Yep, that's just-- did I name it something differently in here? No. Yeah, so already we're starting to get at this idea where I have two functions that do wildly different things. One sums the elements in a list. The other one just loops over some number of months and does some calculation.

But it looks like they're-- in terms of just algorithmically wise, they are very similar. They take similar amounts of time. They increase at the same rate. And basically, they just have a for loop, or some sort of loop that iterates through the input and does something. So algorithmically, we want to consider both of these functions the same, even though they implement completely different things.

Other questions before we go on to another list function? OK, so let's look at a slightly different problem dealing with lists. So this function, or these three functions, deal with finding an element in a list. And we're going to compare the runtimes of these three functions.

So the first function is going to be a very brute force method to find the element in a list. Was there a question? Yeah.

AUDIENCE: [INAUDIBLE]. Oh, wait. Never mind, I see it. Never mind, I got it.

ANA BELL: All right, no worries. OK, so the first function, we'll do a brute force search to find an element x right here, one of my parameters, within a list of the other one of my parameters. Basically, given a list of a bunch of elements, this function will just painstakingly look at each element one at a time and ask whether that element is the one that I'm looking for. All right, so starting over here, the beginning of my list, and going to the end of my list. That's what is in does.

The next one, binary search, also looks for an element in a list making sure that the list is ordered. But the way that it's going to do it is in a slightly smarter way. So I'm actually going to draw my list this way.

So each one of these is going to be elements in my list. So this is going to do a bisection search to find the element in the list. So again, we're looking for element x within this list L. And remember, bisection search we start with a beginning endpoint and an ending endpoint.

And our first guess for where-- for the element or whatever we're looking for is to just say, what's the-- is it the middle element? So look at the element in the middle and ask, are you the one I'm looking for? In this particular case, you look at the element in the middle and you say, are you the x I'm looking for, right? Good. So that's this one right here.

So the midpoint calculation is right here, right? The reason why we're doing // for integer division is in the case where I have a list where I would actually look at the midpoint here, right? Obviously, I can't ask the I ask Python for the element at index 3.5, right? It doesn't work like that.

So I'm just going to round down. You could also round up if you wanted to. I just made the choice to round down just so I'm actually grabbing the element at an integer index.

So I've grabbed my middle element and then I ask, are you the one I'm looking for? And if not, I ask whether this one is too low or too high. And then, if it's too low, then I know I need to search this part of my list. And if it's too high, I need to search this part of my list. So that's what this little if else is doing, right?

And when I make my decision as to which side to look at, then I reset my endpoints. And I do the process all over again by asking the midpoint there, are you the one I'm looking for? So on and so on. So this does a bisection search, also called binary search for the element in a list.

And the last way for us to search whether an element is in the list is one that we've already been doing. It's this little one liner here is x in L, so using the keyword in. I call that the built-in function, the built-in operator in.

OK, so it would be unfair if we just asked Python to figure out-- or to just pick a random number and ask whether that element was the one we're looking for, OK? So instead, what we're going to do is to take an average for each one of these three functions just to make it fair.

So we're going to say, when we're searching for an element in the list, I'm going to say that I'm going to take the average of the case when the element is the first one in the list. And I can find it maybe right away in this case or maybe not right away in this case.

The average with that plus when the element is the last one in the list and plus when the element is the middle one in my list, right? So in that way, we're actually kind of covering all our bases, kind of best case, worst case, medium case scenarios, right? So each one of these three functions will be run with that in mind.

So that's these three functions. So this is my brute force is in. This is my binary. Search Obviously, I'm just going to type in when I run it. So I'll just show you for one of them, so we're going to uncomment this entire bit here and run it.

But you can see here, so instead of just running the performance counter and making one function call, I'm actually going to run three function calls iterated over this little loop where I'm looking for the element at the zeroth location, the element at the halfway location, and the element at the end of my list. And I'm just averaging those down here, the time it takes to find those three. Does that make sense?

All right, so this is a lot to look at. Luckily, I'm going to summarize it in the next few slides so we don't have to stare at that Python screen there. OK, so we had three functions to run. Let's first look at how each of these three functions did individually. And then, we can start comparing them to each other.

So the first function we ran was the is in. So remember, this was the brute force one. We're painstakingly going through each element and asking if it's the one we're looking for. So no smart way about that, just brute force your way through.

We notice that as the input list grows by 10, the time it takes to find the element in the list, whether it's the first one, last one, or in the middle, on average also grows by 10.

All right, next, let's look at the built-in function. We'll worry about the binary one later. The built-in function, so just using the in operator-- and you see this was down here. So this in. So basically, the function I'm running is purely just asking whether x is in L, and that returns true or false.

So I didn't need to make a function for that. But that built-in in operator also has a very similar trend, right? As the length of my list increases by 10, the time it takes for my program to run is also 10 times as long. I went 0.05 to 0.5 and the next one would be 5 and so on.

OK, so those seem to be doing approximately the same sort of-- they have the same performance. Now, what about the bisection search or binary search? Well, this one is not so clear, right?

If we look at the input, the input clearly increases by 10 from here to here. The time increases 9 times 10 to the negative 6 to 1.1 times 10 to the negative 5. And so the factor, how many more times it took, is very unclear, right? It's not quite 1. If it was 1, that meant it's independent, right? It's constant. It doesn't matter what the input size is, it's always going to give us this amount-- it's always going to run in this amount of time. So it's not quite 1.

So it's almost independent of size, but it's not linear like the other two functions were, right? It's not 10 when the input grows by 10. So we're not quite sure what this function is. But clearly, it's not as bad as the other two, but not as good as no relation, OK?

Observation four will now compare the function that we wrote, the is in, this one here, to the binary search, this one down here. Well, binary search was orders of magnitude faster than brute force, right? Brute force, when the input was-- what is this 10 million or 100 million? I'm not sure. When the input was 100 million, brute force took 1.6 seconds.

But the bisection search, the binary search, took 0.00001 seconds. So it's not like we went from 1.5 seconds to 1.2 seconds or 0.5 seconds. We were orders of magnitude faster, right? 10 to the negative 5, right?

So there's a really big difference between this algorithm, the one that brute forces its way through, and between this algorithm that does something smart about removing half of the search space with each loop, right? All right, so that's important to know.

And lastly, just comparing pure time that it takes these programs to run. Let's compare the function that we wrote, the one that loops one at a time through this list, and the built-in in function.

The built-in in function, while it's still the same sort of-- has the same relationship, linear, with the input size, it seems to do a lot better consistently by about 10 times as fast, right? So when our function took 0.1 seconds, the built-in one took 0.05 seconds. When our function took 1 second, the built in function took 0.5 seconds. So consistently, it's just faster to use the built-in in function than to make our.

All right, questions about any of these observations? Do they make sense? Are they interesting?

[LAUGHS]

OK, so what do we see? Just a quick recap of those three functions, right? The first one we saw is linear in the size of the argument. So when the input list size increases by 10, the program takes 10 times as long to run. But this other one is something less than linear, but not constant. So we're not quite sure what it is.

All right, we'll come back to this in a little bit. We'll end up plotting some of these runtimes. So we'll actually be able to see the relationship in a few slides. The next thing I actually want to do is do one more function.

This one is called the diameter. And I'll explain what it's doing because it looks a little bit scary. But suppose we actually have some points in a 2D plane, right? So it looks like this.

Basically, what this function is going to do is it's going to figure out the biggest distance between all of these points. So the distance between these two points is something. The distance between these two points is something else. Which two points yield the biggest distance? That's what this function aims to tell us and what that distance is.

So the way it works is it has nested for loops. So this is different than what we've seen so far, right? We saw an example of this last lecture. But now, we're seeing it in the context of something actually useful. So in this particular case, we're going to create an input list, all right?

So again, our input list will just have some dummy values in it. I don't actually care what these numbers are. I just want to populate a whole bunch of points in a 2D plane. So what we're going to do is pretty much just iterate from numbers 0 to 10. Sorry, 0 to 9, 0 to 99, 0 to 999, and so on, just like we did before.

And to get us a little coordinate in the 2D plane based on those numbers, I'm just going to take the cosine of that number comma the sine of that number. So that together, so like cosine of 1 or whatever sine of 1 will be this point here, right? Set up as a tuple.

And then, this one might be cosine of 5, sine of 5. Something like that. So I'm just making a whole bunch of coordinates in a 2D plane ensuring that I have n coordinates. OK, now the loop-- sorry, there's going to be two loops.

The outer loop will basically take us through each of these elements. I have five in this particular case. And the inner loop will go through every other element. But notice it starts from i plus 1. And I'll tell you why that is instead of starting from 0.

So let's just walk through. Let's say we start out with this element. This little guy is our first element in our outer for loop. So right now, we've grabbed the first element in our outer for loop. And what we're going to do is figure out the distance between it and everybody else. So now, we're iterating through the inner for loop going through each element except for myself.

So I'm going to get the distance between this one and this one. Since it's the first one, obviously, it's the biggest one. But then, I'm going to get the distance between this one and this one, right? And I'm going to say, are you bigger than this one? It looks like no, so we're still keeping this one as our longest one.

Then I'm going to grab the distance between this and this one and this and this one. And as I'm going through, this little if statement here keeps track of the farthest one. So the one that has the biggest magnitude, right? In this case, that's probably the first one we looked at. And after I've gone through each element, each other element, I've concluded my first iteration in my outer for loop.

So now, the outer for loop goes to the next element in the list. Let's say it's this one. It doesn't actually matter. OK, this one, we'll look at the distance between itself and everybody else except for the one we already looked at because we already know this distance. We kept track of it already when we iterated through this one.

So as I'm going through my outer for loop keeping track of this point here, it figures out the distance between this one, this one, which is suddenly bigger than that one which we had kept track of, and then, this one right here. All right, good. So now we're still keeping track of the biggest distance we've seen. It's probably this one here. And I've concluded the second iteration of my outer for loop.

And now I go to the next element. Let's say, it's this one. Doesn't matter again. Now, this one is going to get the distance between itself and everybody else except for the two that we've already seen, this one and this one, right? So that's why our inner loop starts at i plus 1.

So this one will get the distance between itself and this one all the way back there and this one all the way over here. And then, next iteration in the outer for loop takes a look at this one, let's say. And it finds the distance between itself and everybody else. But you know what? There's only one left, that one there.

And then, the last time through this one doesn't even get a chance to find the distance between itself and anybody else because everybody else already found the distance between it.

And so, in this way, we're basically finding all the possible pairs of all of these points in this 2D plane and keeping track of the longest-- of the biggest distance, OK? So in terms of the list-- the input list, the way that looks like, this i plus 1 business here, the outer loop basically says, I'm going to start with you and I'm going to get the difference between you and the element at index 1, the element at index 2, and the element at index 3.

This outer for loop is done. Next, we're going to get the difference-- the distance between this one and everybody else, right? So obviously, not element index 0 because we already know that distance. So we're going to get the distance between element and index 1 and index 2 and index 3 and then we're done.

And then, the last loop-- outer loop gets the distance between element and index 2 and index 3 and then it's done. So just these two nested loops just does all of this until it finds all of these pairs. It basically pairs up everybody together.

OK, so if we run it, what are we going to see? So my input for this particular function, you'll notice, first of all, is going to be much, much smaller than the inputs for everything we've done so far, right? Some of the inputs we had seen in the past were a million, 10 million, 100 million. In this particular case, I'm only going to go up to 6,000 because it's just going to take way too long to run if I make it go for any much-- any longer than that.

So what do we see already? We've got 100 points. So hundreds of these, right? Finding the distance-- finding the maximum distance between a bunch of these pairs took about 0.03 seconds. If we doubled that to 200 points, it took 0.01 seconds. If we doubled that to 400 points, it took 0.05 seconds and so on and so on.

So just like before, let's take a look at the big numbers to see our trend. So as the numbers increase by 2, if my input increases by 2, it looks like the time that it takes for me to find out the biggest distance increases by 4. So my input increases by 2, the time increases by 4.

And I'm not going to run this, but you can make a new list on your own and change this to be inputs that are multiples of 10, right? Increasing by 10 each time. And you'll see a very similar pattern where the time it takes to run that program will be about 100 times as slow. So the relationship there is an n squared kind of relationship.

All, right so a few observations here as well. First one I already mentioned is this program just takes a lot longer to run in general, right? So here, we were able with a compound and with finding whether an element is in the list and getting the sum of all the elements in the list, we were able to run 100 million-- a list with 100 million elements and it still took about 1 something seconds.

Whereas, with this diameter function, we can barely get to 6,000 and it's already taking 14 seconds. So just way, way, way slower program in general, right? And then, the relationship seems to be an n squared kind of relationship relating the input to how long the program takes.

So let's actually plot-- well, I already did this. But here are the relationships for these three types of algorithms that we've seen so far. So this is the finding the element in a list. Those three versions-- sorry, those two versions that we saw and this is the diameter function.

So if we plot how long it takes the program to run when the input increase-- sorry, when the input is this size, we can see that there is a linear relationship. So the time it takes for the program to run is linear in the input.

The diameter, we'll talk about the binary search in a bit. But the diameter, we, again, notice this just by looking at the pure numbers, but it's a lot easier to see it visually when this is on the x-axis, the size of the problem. So how many points we actually are finding the diameter between and how long it actually takes the program to run. Again, the relationship is quadratic. Now that we plot it, we clearly see the quadratic relationship.

And then, this binary search, we were very unsure of what it was, right? It wasn't quite constant. It definitely wasn't linear. But now that we've plotted it, so this is the input size and this is how long it actually takes the program to run, you can see it drastically increases when the input size is very small.

But then, it kind of asymptotically reaches some sort of value. It's actually a logarithmic relationship. All right, last thing I wanted to mention about timing before we move on to counting is just pure running. Just purely running these functions on different computers will just give us different values just right off the bat.

On this newer-ish computer, how long did it take to run this compound? Well, what? It took 3 seconds or something, 1 point something seconds. On an older laptop, it took 63 seconds. On an even older desktop, it took 126 seconds. You're just purely timing things. The machine you're running it on is going to make a difference, OK?

And then, that's fine, right? It's important to know how long it takes. But if you're just looking at the relationship between input and how long the program takes to run, that's the same. So it doesn't matter what machine you're running it on. When you increase the input by 10, the program will take 10 times as long to run no matter whether you run it on a fast laptop, old laptop, or a super old desktop.

So just timing a program is really important, right? You'd like to know whether the program you wrote you're going to have to wait a month for it to finish or a couple of minutes for it to finish. That's an important thing to know.

But what we're going to get at towards the end of this lecture is something that's complimentary, and that's this idea of asymptotic complexity. So kind of mathematically saying, you know what? This program is not going to be that bad to run, right? You're not going to have to wait for it for months to run without actually running it, of course.

So you'd be able to glance at a program and say, this one is reasonable to run. And so we're going to do that in terms of this idea of order of growth, which we'll get to-- at in a little bit. OK, any questions on timing before we get to counting? Yeah.

AUDIENCE: Should we assume that all built-in functions in Python are the optimal in terms of running time?

ANA BELL: Yes. Can you assume that all built in functions are optimal in terms of running time? Yes. Certainly better than when we write them. Yes, in Python. And then, of course, in other languages, they maybe take advantage of other speed ups as well like putting things in memory efficiently. But yeah, generally, it's better to run something that's already been made than to make it yourself. Yeah.

OK, so now, what we're going to do is we're going to count operations, just like we did last lecture. Clearly, timing is nice, but it doesn't give us a nice relationship besides us spotting it, right? There's no formula. There's no relationship that relates the input to how long it takes the program to run, right?

Counting will get us a little bit closer to that, and we saw that last lecture. Let me remind you the idea of counting. So the idea of counting is that we're going to take a bunch of these operations, like mathematical operations, comparisons, indexing into something, assigning a value to a variable, all of these things, right?

When we run them, yes, they might run for different amounts of time. 1 times 10 negative 9 versus times 10 negative 9, something like that. That's not a very big difference. And so what we're going to say is that every one of these operations will consider to be constant, right? They will take one unit of time.

So if we say that, we can actually come up with a relationship that tells us according to-- that relates the input to how many operations this program will run. So here in the convert to kilometers, what do we have? We have one multiplication.

And just for the heck of it, this lecture, let's say, the return also counts as an operation. So in this convert to kilometer function, we have two operations. Notice that it's not really related to the input at all. So the amount of operations that this program takes to run is always two. It matches what our timing said, right? It basically didn't matter what the input was. It always took approximately the same amount of time to run.

The sum of function. So it takes an input list and it gets the sum of all the elements. This one will do-- we'll have one operation for doing this assignment. It'll have one operation for grabbing an element in my list L and assigning it to i.

It'll have two operations for this total plus equals i. Remember, total plus i on the right hand side is one operation. And total equals that is my second operation, so that's two operations. And then, let's not forget for loop. That's kind of the important part of this function. How many times will these three operations repeat this 1 plus these 2? Well, it's going to repeat however many elements I have in L. So length L times.

And then, again, let's say we count the return. The return will also be one operation. So the total number of operations for the sum of function will be 1 for the total equals 0 plus length of L times 3, because there's three operations being done for each length of L, plus another one for the return. So that's going to be 3 length L plus 2.

That's a nice little formula that relates how many units of time we'll have to wait depending on the size of the list. That's pretty cool. So the way that we're going to count the number of operations-- again, I'm going to do it slightly differently in the last lecture just to show you that there is another way to do it.

So this is our function is in. It's going to count how many operations we have. And I'm going to use something called a global variable. I'll show you, again, the difference between them. So it's just these three lines that I added.

And you should never ever use global variables in your programs except in this situation. The idea of global variables is that you can define variables just in the main program outside of any functions. And you can access those variables within some function purely by saying, if we define count out here, before this function definition count equals 0 or whatever. Inside of any function, we can say, hey Python, I would like to access this variable that I defined outside of this function. You say that-- you tell Python using global and then the name of that variable.

And Python will grab that variable that's basically quote unquote "shared" across the entire program and modify that variable. So in essence, we're basically saying, this is now a shared variable. If I modify it within this counter, within this function, it'll be, obviously, modified for everything else.

It's very tempting to use global variables because all the variables you could ever want to create are going to be accessible by everybody, right? No need to pass in parameters. No need to do-- but it's very, very bad programming. So we won't ever do it except in this particular case because we'd like to keep a counter of things that are happening or for debugging purposes and things like that.

So the count variable will keep track of-- it'll just increment in key places where we have these constant unit of times happening. So I've got count plus equals 1 here because I've got my return value. I've got count plus equals 2 here because I grab an element from L and I do the equality check here and then that's it, OK?

So if I run that, it's down here. What are we going to see? Well, I didn't actually do how much more it ran, but we can see the relationship, right? We go 9 to 37 to 307 to 3007 to 30,007 and so on. So again, the same relationship where we increase the input by 10. The number of operations we do is 10 times as more, exactly like the formula said it would be.

What about the binary search? So again, we're going to use this global variable and we're going to have the counter keep track of all of these operations. So this count incrementing by 3 accounts for setting the low to zero, setting the height of this thing and grabbing this actual value of length.

Incrementing the count within this while loop will keep track of this subtraction as one operation and the test that it's greater than 1 is another operation. Counting increasing by 3 here accounts for high plus low, the integer division, and assigning that value back to mid.

Count plus 3 here accounts for indexing into this L, the less than or equal check. And then, either doing this reassignment of low or this reassignment of high. So that's three operations. And then, lastly, count increases by 3 once more because I've got these operations here. So indexing into low, checking for equality, and then doing the return.

So the actual number of operations will be kept track of in-- by the counter. So all we're doing is just kind of reporting how many times-- how many operations we've done. So as we increase the input by 10, just like with timing, we can't quite tell what the relationship is, right? Again, it's like 1 point something right with each.

All right, so these are the results. So the observation one, as I mentioned, when we increase the input by 10, this brute force-- I erased it, but the brute force is in function also-- does 10 times as many operations. The binary search, again, we don't know what rate it is at, but we can plot them.

So here I have the plots, just like when I plotted the input size, versus how long the program actually took to run. I'm now plotting the input size versus, actually, just the number of operations being done. So the is in function, that brute force way of finding whether an element is in the list, grows linearly. No surprise there.

And how lucky for us? The binary search matches-- the graph matches the one that we had for timing, right? So as I increase my size in the binary search method, the number of operations that I do is logarithmic in time, just like we saw in the actual time.

OK, so timing and counting are really nice, right? Timing gives us pure number of seconds or months or whatever we need to wait for this program to finish. But counting gives us a nice little formula that relates the input to the number of operations that you have to do.

You might have noticed-- I briefly touched upon this-- that throughout this entire lecture and last lecture, we basically just saw something like three different algorithms, right? We saw something that's constant, something that's linear, something that's quadratic, and something that's binary, or logarithmic, in this particular case. So that's four different algorithms. But we saw way more functions run, right?

So what we'd like to do is evaluate the algorithms, not the different implementations, right? And what we'd like to do is evaluate these algorithms as the input gets really, really big. So what we're going to do is figure out a relationship between the program's runtime and the input.

But what we're going to do is focus on the biggest terms that contribute to the program's runtime, right? So we saw these examples last time, right? This my sum, which basically summed all the elements-- sorry, all the numbers from 0 to x. And the silly square function that had nested loops, kind of like this diameter one, right?

We were able to say something like when the input increases by 10, the program is 10 times as long to run, right? So the efficiency of that program was on the order of x. When the input increased by x, the program took x times as long to run. The square had a similar-- we could have said it in a similar way, right?

When the input increases by x, the program took x squared as long to run. So I don't actually care about all of these differences in the exact timings. 1 times 10 negative 6. 1.3 times 10 to the negative 6. I don't care.

What I do care is the order of growth. How does the program run in relation to the input, OK? And I care about that when the input is really, really big. So what we're going to do is express the program's runtime in an order of not exact kind of relationship.

So while counting was really nice, it told us a nice relationship between the input and the number of operations. When the input is really, really big, like 3x plus 4, when x is really, really big, I don't care that the number of operations is 3x plus 4. Because when x is really, really big, that plus 4 might as well be plus 0. And that 3x is basically like x when the input is really, really big. So that's what we're going to try to do, all right?

Now, before we do that, we need to decide what to measure, right? Because when we write functions, we're going to have functions that have a whole bunch of inputs, potentially, right? So the input could be an integer, like in convert to kilometers. It could be a list, in which case, we would be interested in maybe the length of the list. And if you have many parameters, you'd have to decide, right? What is the parameter that contributes to the growth of this function?

So here's an example. This is our is in function. It looks for an element e in list L, right? So there's two parameters to this one. We can ask, does the program take longer to run as e increases? It's one of the parameters. Let's see what happens as we make e bigger.

So we can look at a little example. If we find out whether 0 is in this list containing 1, 2, 3, or whether 1,000 is in the list 1, 2, 3, does the program take longer to run? No, exactly. So e is not really relevant in my runtime calculation here.

Well, let's consider L now. When we say L is going to change, it could change in two ways, right? The elements in L could have different values or the list length itself could be different. So in this particular function, let's say that the elements in L are small numbers versus big numbers, OK? That's certainly something that could happen.

And certainly, with some functions, that's going to make a difference. So let's say, in this particular function, if the elements in L are big versus small, is it going to make an impact on my runtime? Well, here's a little example. Let's say I'm looking for the number 0 inside a list with 1, 2, 3 and the number 0 inside a list with 1,000, 2,000, and 3,000. Is that going to make a difference? No, right?

So the size of the elements themselves don't really matter. And one last thing to ask ourselves is, what about the length of the list? So if L has different lengths, will this make a difference in our runtime? So if I'm looking for zero in a list with three elements or 0 in a list with 10 elements where clearly that 0 is nowhere to be found, is that linked list that's going to have a difference? Yeah, in this case, it will. Exactly.

So here, in this particular function, the input, I'd be interested in reporting the runtime from is the length of the list. Not the elements in the list, not e itself, but the length of the list, OK?

So the last thing that I'll mention is, for this particular class, we're going to talk about the worst case scenario. So you might have noticed in this previous example here, I always looked for an element that wasn't even in the list. So when you're faced with a function, you ask yourself, for this particular class at least, what is the worst case scenario? And finding out whether a function is in a list, the worst case scenario for us is, if it's not in the list at all, right?

So that's another aspect of runtime that we don't actually-- we won't talk about. Because for us, we're always interested in the worst case. But there are certain analyzes where you could look at the best case scenario, which is, well, the element is the first one in the list, right? In that case, you're always going to find it right away, so it's constant. Or an average case scenario, which is kind of what people do in the real world, right? You're not always encountering the worst case.

But for us, we're going to look at the worst case scenario. So our goal is going to be to describe how the runtime grows as the size of the input grows in a really general way. So we're not going to be interested in figuring out the exact number of operations. No 3x plus 2 kind of deal here. We're just going to focus on terms that are really-- that grow the fastest. We're going to eliminate any sort of additive, multiplicative constants, and things like that. So we're just going to focus on terms that grow the fastest and that will give us our order of growth.

So the way we're going to denote the order of growth is using this notation called big O and big theta. OK, now, warning, we're going to have some math coming our way. It's going to be like three slides of just pure math, OK? You can sit back. You won't need to know it. You won't need to know the details. But it will motivate us to give us the idea about this asymptotic order of growth.

All right, so this is the mathematical definition of big O. So what we would like to do-- there will be a drawing, don't worry. So what we would like to do is figure out an upper bound for our function. So the function might look like this. And I know this is just an f, but we relate this to our class by saying, you know what?

If we did the order of calculation-- sorry, the number of operations analysis for a function, we could basically come up with something like this, right? We came up with 3x plus 2. We could come up with 3x squared plus 20x plus 1 for some random function that we wrote, right? So that we consider the function.

Now, the big O is going to be the upper bound on this function. So if I plot this function in my xy axis, this is what it looks like. The big O will be some other function that's going to upper bound this one, the blue one, OK? And it's going to upper bound it for all values beyond some x.

So for all values beyond some x-- some number on the x-axis, some crossover point, this big O of g-- this g will always be bigger than my f. That's the idea here. So clearly, x is not going to upper bound it, right? Because after this crossover point, x will be below my function no matter how big of a constant I tack onto that x. I could have 1,000x. That's still not going to upper bound my little blue line here.

So what we're going to do is we're going to increase the exponential there. So let's take x squared. Well, x squared is getting closer. It looks like they're both quadratics. But this orange line is not above the blue line for some crossover 0.2. 2x, getting closer, 3x, getting closer, 4x is an upper bound on my g because after this little crossover point here at about 20, my orange line, the g, will be always above my blue line, my f. So far so good? Just visually speaking. Yeah.

AUDIENCE: The orange one follow it there or does that matter?

ANA BELL: Yeah, the orange one is below it. That's totally fine. Because what we're interested in is the behavior when the input is really big. So that's why I don't care about weird stuff happening down there. All I care about is when my x is super big.

OK, so now, I've found this g. So I can say that after this point, 20, my orange line will always be above my blue. So I can say that my f is big O of g-- is big O of x squared, OK? Because I don't care about this for so much because it's just a multiplicative constant. Because this 4x is always greater than my function for all x greater than this crossover point here.

That's it. That's the definition, right? So the g here is basically this function without the multiplicative constant in front of it. OK, so I say 3x squared plus 20x plus 1 is big O of x squared. So generally speaking-- that was just an example. Generally speaking, the big O is an upper bound on my function, OK?

And this is now just using variables like constants and things like that. But it's exactly the same situation that we had from before, OK? So I'm going to try to map the blue to the blue and the orange to the orange and the purple to the purple to help you match up what we saw on the previous slide.

So basically, we say that our function f is big O of this orange g if we can find some blue constant where this constant was this 4 here where that constant multiplied by g x squared is greater than my function for all values beyond that crossover point, right? So I found my 4 because 4x squared is always greater than my function beyond 21. That's what we saw in the picture.

So then, we can say that my function f is big O of g of x where that g is x squared, which is matched up, all right? So in terms of the picture here, this is kind of a little zoom in of what happens. Anything can happen down here. But beyond the crossover point, which is here in the big picture, that crossover point-- beyond that crossover point, my orange is always greater than my blue.

OK, so what does this mean? We're going to talk about this in a few slides. But you might have thought about this. I can actually pick any function that grows faster than-- what is this? 3x squared, right? I can pick x factorial. X factorial grows super fast. Or 2 to the x. That also grows super fast. All of those functions that grow way faster than mine are also upper bounds on this function, right?

OK, so that's big O. It's just an upper bound. Then what is theta? For the reason I just stated, I said x squared-- or sorry, x factorial, 2 to the x, all of these functions that grow much faster than my function are all upper bounds. And that's not really helpful for us when we say, oh, this function is big O of whatever, right? Because you can just pick something that's ludicrously fast-- that grows ludicrously fast and say that has no meaning.

So instead, what we usually report is the theta, which is actually an upper bound and a lower bound for our function, all right? So using the exact same reasoning, we're going to find some constant tacked on to that g of x such that that function grows-- is always underneath our function, OK?

So again, I'll put up a lot of math. But basically, these first two lines here, this one here there exists, blah, blah, blah, blah, blah, that first here, this is the big O definition. So we've already know what that means. All we're going to do is tack on another condition, which is that we can find another constant for that same g where that function, beyond some crossover point, is always below my blue line.

So here's an example. 4x squared, we saw that it grew faster than 3x squared beyond a crossover point. Well, we could say 2x squared will always grow slower than its own crossover point. So the constant 4 was the same as we had seen before but this constant 2 now becomes a lower bound, right?

So I'm basically trying to have that same g both upper bound and lower bound my blue function, all right? And that's the definition of theta. So now, I can no longer say that 2 to the x an exponential both upper bounds and lower bounds it because that 2 to the x will grow faster than my function no matter what constant I tack onto it, OK?

So now, what we see is that, really, the g of x is going to be the term that grows the fastest. It's just going to be that term here, right? It's going to be the thing without-- the fastest growing term in my function without the constant behind it.

OK, so yes, we will never remember all that, but we're going to do a bunch of exercises and you're going to see just how easy it is to figure out the order of growth, OK? But I will mention this just, again, because it's very important.

So when we're talking about upper bounds, you can pick any function that grows faster than yours, right? F of x, this 3x squared thing is O of x squared. Yes, but it's also O of x cubed, O of x to the 5, O of x to the 100, 2 to the x of x factorial, all of those things that grow much faster.

But my f of x is only one theta, and it's theta of x squared, right? And that's the term that grows the fastest in my function here. So when we look at a function, based on the number of operations or however you're given the function, when we look at the order of growth of the function, we just focus on the dominant term, right?

So in the first one, the input here is n. And the function is n squared plus 2n plus 2. Which one of these is the dominant term? You tell me. Yes, exactly, n squared. So this function is just going to be theta of n squared. That's it. How about in the next one? What's the dominant term here.

Yeah, exactly, 3x squared. Even though 100,000x is going to be huge for a while and this constant is also going to be huge for a while, as x gets really, really big, this 3x squared and, in fact, just x squared will kind of take over everything else, right? So this next one is theta of x squared.

How about the next one? What's the term that grows the fastest here? Yeah, exactly. Log is slower growth, right? So this theta of this function is just theta of a. So notice what we're doing here is just focusing on the dominant term. We're going to drop the multiplicative constants, drop every other term, and relate the theta in terms of the input.

Don't just use theta of n all the time, right? In the previous one, it's tempting to say, the first one is theta of n squared, the next one is theta of n squared, the last one is theta of n. But n is not always the input to your function, right? If it is, great. If it's not, you always have to relate it according to the input of the function. Maybe its length L. Maybe it's something else.

OK, so let's have you try a couple more. What is the theta of the first one here? What is the term that grows the fastest? Yeah, theta of x. Next one. n cubed. Exactly, theta of n cubed. See, I told you this is going to be so easy. I know that math was scary. How about the next one?

That's the term that grows the fastest, but then, it's theta of drop any multiplicative constants and it's just theta of y. The last one is going to be tricky. What is the theta if the variable is only b? Yeah, 2 to the b. What about if the variable is only a? a cubed. Exactly. And if my function is both a function of a and b?

And a plus a cubed, right? Because both will contribute to the runtime of this function, right? Not just the b. So if this function, whatever this crazy function is that I wrote that takes so long to run had both inputs b and a as its parameters, the theta for that function is in terms of both b and a, right? The dominant terms of each. Yeah.

AUDIENCE: Don't worry about negative coefficients?

ANA BELL: No. No need to worry about negative coefficients.

AUDIENCE: Oh wait, [INAUDIBLE] have a negative.

ANA BELL: Yeah.

[LAUGHS]

Yeah. Yeah, question.

AUDIENCE: I guess it can get confusing. Let's say, they were asking for big data [INAUDIBLE].

ANA BELL: Oh, some different parameters-- variable that's not even here? Yeah, if the parameter the function was c, let's say, for this last one, but the formula was this, then the theta would be just constant theta of 1. Because it doesn't even depend on these variables, so these are just considered constant time. That's a great question. Yeah. If the parameter was c or something else.

OK, so now we can actually look at functions that we write. And we do the exact same thing. We can first start out with just saying how many operations does this function take, come up with that relationship, and just theta that, right? Just like we did on the previous slide.

So here's a function that calculates the factorial. What do we have here? Well, we've got this is constant here, right? We've got just one while loop where there's five things going on here. There's the comparison, there's this times equals, which is two operations, this minus equals it's two operations. So this function is just 5n plus 2, by the same analysis we did a few slides ago, right?

So if we say, what's the theta of this function? Well, what's the theta of this thing 5n plus 2? Super easy, right? It's just theta of n. And in this case, the parameter to our function is truly n.

When we have functions that are slightly more complex and we've got things that are in series, like for example, here, I've got two for loops one right after the other. We basically use this law of addition to take care of that. So that means we figure out what the theta is for the first for loop, the theta for the next for loop, and we just add those two thetas together, OK?

So the first for loop here is theta of n because it's something that depends on parameter n. And the next for loop here is theta of n squared, right? Because the parameter here is n times n. The stuff inside the for loops are constant, so they don't contribute anything to our theta, right? There's no more things to multiply the complexity there.

So if this is my entire function here, the theta for this function is theta of n plus theta of n squared, right? And the law of addition just says, theta of n plus theta of n squared is just theta of sticking those two inside as part of my function n plus n squared. And we know how to do that. That just simplifies to the dominant term, which is n squared.

OK, so that's the law of addition. So that's when we have loops or things like that in series. What about when we have loops that are nested, right? Then, we use the law of multiplication. Because for each one of these things, we're going to have to do this that many times, right?

So in this particular case, we need to be careful. The outer for loop is going to be theta of n. And the inner for loop is also theta of n, even though I'm dividing n by 2, right? 0.5 times n is still theta of n. That multiplicative constant in front of that n is 0.5, which is just-- it's just-- it still leads me to be theta of n.

The print is constant, so there's nothing else to multiply there. So the law of multiplication just says theta of n times theta of n is theta of n squared inside there. OK, so let's look at this program. What is the theta for this?

Well, we could do it sort of in very great detail. We've got x as our parameter. So we only count loops and things like that that are a function of x. If I had a loop that was a function of, I don't know, n or something, that doesn't count because it's not a function of my input. So only look at things that are a function of x.

I've got one outer for loop that goes through x times. So that's theta of x. I've got an inner for loop that starts from i and goes to x. That's a little bit tricky. But in the end, overall, it's going to be theta of x because it's going to be-- the first time it's going to go through x times. The next time it's going to go x minus 1 then x minus 2 then x minus 3. So we're effectively just kind of adding over all of these runs x plus-- sorry, 1 plus 2 plus 3 plus 4 plus 5 all the way up to x. And that's just some function of x. It's definitely not going to be constant. So the inner loop is also theta of x.

Everything else is theta of 1. There's nothing else that depends on x. So this whole function is going to be theta of 1 for this assignment here. Theta of x times theta of x for this nested loop here. And theta of 1 for this return down here. So overall, it's just going to be theta of-- all right, so that's that. And so, overall, it's just going to be theta of x squared just by the laws of multiplication and addition.

All right, think about this. And then, tell me what you think it is. What do you guys think it is? Yeah, theta of length of L. Absolutely right. So this is constant. This stuff inside the loop is constant. The return is constant. The only thing that depends on L is the length of the list, right? This loop. So the answer is theta length. Perfect.

How about this one? So here we're assuming all the inputs are the same length. Yeah, theta of length of pick your favorite one. Theta of length L is reasonable. You could also say theta of length L1 or theta of length L2. Because these are two loops that are in series, right?

So this one just loops through the length of L. But inside, we're not doing anything that costs more than just constant time, right? Here, we're just comparing two numbers like 3 and 2. We're just assigning something to true. So nothing else really depends on the length of the list. So this is theta of length L. This is plus theta of length L. So that's just theta of length L.

All right, so we saw a bunch of different algorithms, right? Sorry, no we didn't see a bunch-- we saw a bunch of different programs. But we could classify them all into one of these categories, right? And this is all-- basically, all the different algorithms that you could ever write in general, right? So something that's constant, theta of 1. Something that's logarithmic is theta log n. Something that's linear, we saw many of these, is theta of n.

Something that's log linear, we haven't seen any yet, but that's theta of n log n. Theta of n to some constant, like n squared, n cubed, is polynomial. And theta of some constant to the n, like 2 to the n, 3 to the n, is exponential. And when we're writing our programs, you can do a quick analysis of the program that you just wrote. Look at the loops. Look at to see how efficient you wrote it.

And you could basically classify your program into one of these categories, right? If you had nested loops that both depend on the input, you probably wrote a polynomial type algorithm. If you just had one loop that depended on the input, you probably wrote a linear time algorithm, right?

And when we write these algorithms, at a first pass, we want to be somewhere up here. You don't want to do anything that's polynomial or definitely not exponential because things get slow really quickly with those numbers, right? And so we never ever want to be in that situation, although sometimes it's unavoidable.

All right, so that's all I've got. Next lecture, we will be going through a bunch of those different complexity classes and looking at different programs that land in those classes, especially the logarithmic and the log linears. All right.