[SQUEAKING]

[RUSTLING]

[CLICKING]

**ANA BELL:** Let's get started. So today's lecture will be super short. We've got a 45-minute quiz on object-oriented programming classes, that kind of stuff. So I wanted to give you guys an extra bit of time to work through three programming problems. But the actual lecture part-- we're going to switch gears a little bit. And we're going to start talking about something more theoretical, which is how to figure out whether the programs we write are efficient and how efficient are they. OK. So we're going to do that today using the idea of timing our programs and then counting the number of operations, as I'll describe in a little bit.

But first of all, a little bit of motivation. So why do we actually care about this topic? It's a topic that's a high research area in computer science. So far in this class though, we've emphasized correctness, right? In problem sets, the unit tests check for whether that the programs you wrote were correct. In quizzes, we basically look at how many test cases you pass, right, and to determine the grade.

But these days, we actually have a whole bunch of data coming at us, right? So we have a lot of data that we need to analyze, we need to read, we need to visualize, we need to make sense of. And so the programs that we write, yes, they have to be correct, which is a large part of it. But they also have to be fast, right? So if it takes a year to analyze a bunch of information on YouTube videos, nobody's going to really want to wait that long, right?

And so we're going to emphasize in the next three or four lectures-- I forget exactly how many-- but the next little section in this class, the idea of how to determine the efficiency of our programs. So when we're talking about efficiency, we can talk about the time efficiency of programs and also the space efficiency of programs. And usually, there's going to be a tradeoff between these two. So very rarely these days can you come up with an algorithm that's both efficient in time and space compared to algorithms that are already out there.

So usually, there's a tradeoff. And the best example is the one that we saw with Fibonacci. So we saw a code that was recursive to calculate Fibonacci. So Fibonacci of n was Fibonacci of n minus 1 plus Fibonacci of n minus 2, right? That was our recursive step. That program that was recursive took something like 30 million steps to calculate Fibonacci of 30-something, right? 30 million recursive calls, which was pretty slow. It took a couple seconds for it to run.

But then we saw a version with memoization. No, there's no R missing there. It's just memoization, sort of the process of keeping a memo through a dictionary in that particular case. And the memoization idea was that we would take some values that we calculate. And as we calculate them, store them in the memo. So in the memoization example, we had given up some of our memory, right, to store these values so that we didn't have to compute them.

And in the process of doing, so we had a program that ran really, really quick, much quicker than the plain recursive version that we had originally seen. So there's this tradeoff, where you have a program that's fast but might store some values and take up more memory, or a program that doesn't store anything but then is not going to be as fast. It's going to be slower because it needs to keep computing a bunch of different values.

So what we're going to do in this lecture is show you a very simple way of figuring out how efficient our programs are, which is we're just going to time them. And then we're going to count the number of operations that these programs take. But we're going to do so sort of with the idea in the back of our mind that there's going to be a better way to figure out the efficiency of these programs. And ultimately, we don't really want to figure out the efficiency of an implementation.

An implementation means you implement a program that finds a sum using a while loop. I implement the program to find a sum using a for loop. Those are two different implementations. But at their core, the algorithms behind the scenes is going to be the same. And so what we want to do is to try to figure out how to evaluate algorithms as opposed to these different implementations, because each one of you is going to come up with a completely different implementation for today's quiz. But I don't want to evaluate that. I would like to evaluate the algorithms behind the scenes.

OK. So we're going to do-- like I mentioned, we're going to, today, look at measuring how long our program takes with an actual timer. And then we're going to also count how many operations our program takes. And then we're not going to look at this other abstract notion, we're going to look at that next lecture.

So today's lecture, we're going to use another module. We've been looking at modules in the past couple of lectures already. We've seen the random module, which helps us deal with random numbers. We've seen the datetime module, which helps us deal with, or was it dateutil, something like that, which helps us deal with datetime objects and converting dates into objects that were nicely usable. Today, we're going to use the time module-- right here-- which will help us deal with the system clock.

So if we're timing functions that we run, we're going to want to access the system clock to figure out exactly what time we started this function and what time we ended the function. So just a little thing-- you probably already know this-- how to call these functions within these modules. So the modules basically bring in a whole bunch of functions and maybe objects and things like that related to one topic or one subject into your code. And then to run the functions in your code, you just use this dot notation on the module name.

So if I wanted to use the sine function from the math module, I would just say math.sine. And then I have access to that sine function. OK. So let's start looking at timing of program. The simplest way to figure out how fast the program runs. So we're going to use the time module. So I'm importing it here. And when I do that, Python is going to bring in all of these functions related to the time.

Now we're going to look in this particular lecture at three different functions. And we're going to time them, each of them. Next lecture, we're going to look at a whole bunch more functions, just to give you a little bit more practice with timing and counting operations. And then we'll introduce a more abstract notion of this order of growth. So the three functions we're going to look at are these ones-- so Celsius to Fahrenheit, mysum, and square.

So Celsius to Fahrenheit-- pretty self-explanatory. It takes in one parameter, the number for a Celsius temperature, and converts it to Fahrenheit. So we did this lecture one. Just using the formula. This function, mysum, will take in a number x-- so 7, or 10, or 100, whatever it is. And it uses a loop, so computationally uses this loop that iterates through each number from 0 all the way up to including x, and keeps a running total. So it adds I to itself to the total and returns it.

So of course, we could have rewritten this in a more efficient way by using the formula to calculate the sum what is n times n plus 1 over 2. But here, we're just doing it using this for loop. And then lastly, is this function called square. And this one is going to be even more inefficient. We're going to calculate n squared. So the parameter here n will be squared.

But we're not doing return n times n or return n star, star two. We're not doing any of that. We're actually going to use two nested loops. So I've got an outer for loop that goes through the numbers 0 to n, not including. An inner for loop that goes through the numbers 0 to n, not including. And this square sum here adds 1 to itself every time. So effectively, we're going through and adding 1 to that sum n squared times. So super inefficient. But this is where we're at.

And so how do we actually time these functions? So here's-- this is basically some lines of code in a file. So I've got the time module imported here. I've got the function here. I'm going to call the time module, and the time function within the datetime module. So this tells me the number of seconds that have passed since January 1, 1970. That's called the epoch. So the beginning of time in computer speak.

So if I grab how many seconds have passed since that time, then tstart stores that number of seconds. Then I'm going to run my function, Celsius to Fahrenheit 37. And then I'm going to get the time again down here and subtract the time right now, after the function has finished, minus the time it was right before I started my function. So that gives me the DT. And then I just print that out.

So we can run it together. The way I'm going to run it is by actually doing a little bit of modularization to this code. So I have this function, and this is the only function I'm actually going to run down here. It's my-- I call it a time wrapper. It's a wrapper function. And it takes in two parameters. The first is the actual function I want to run. So I'll show you down here, you can see I'm running the time wrapper with the name, literally the name of the function I want to run. This is not a function call. It's just the name of my function. So that's the first parameter.

And the second parameter is a whole bunch of different inputs I want to run the function with. So this LN is created up here. And it just makes for me the list of all of these inputs. So I'm going to run Celsius to Fahrenheit with a number 1, Celsius to Fahrenheit with the number 10, Celsius to Fahrenheit with 100 and so on. So these will be all my inputs to my function.

And so when I call this wrapper, Python is just going to replace f with the function that I'm actually running, so Celsius to Fahrenheit, or mysum or square. And you can see here for each one of the different inputs I'm going to grab the time, run the function, grab the time again to get the DT, and then print how long it took. So I'll show you what that looks like. So here I ran Celsius to Fahrenheit with inputs 1, 10, 100, 1,000, 10,000, and so on.

It was really fast. It took zero seconds every single time. So no matter what the input, 0 seconds. So fast that Python didn't even tell me exactly how slow it was, 10 to the negative 9 or whatever, just 0 seconds. And that's in part to the time function, but we'll leave it at that. It's just very fast.

OK, how about the next function? Let's do mysum. So mysum is not just doing calculations, it has a loop. That's a function of the input. So our input changes. And you might have noticed that as our input got bigger, we actually had to wait a little while for this result to come by. So we see down here, or up here, when the input is pretty small-- yes, it took 0 seconds. It's so fast that it didn't even register it.

But at some point, we started to get actual numbers. So with 10,000, it took 0.00099 seconds. With 100,000, it took 0.01. With-- what is this, 1 million, yeah, with 1 million it took 0.05 seconds. So we can actually see a little pattern if we stare at it long enough, especially for the bigger numbers. So down here, these first two are iffy. But when we get to a big number like 1 million, we say it took 0.05 seconds.

When we increase the input by 10 to 10 million, the input took 0.5 seconds. And when we increase the input by 10 again, it took 5 seconds. So we could guess that when we increase the input again by 10, it will take about 50 seconds to run. And you can even try that out if you'd like to wait for 50 seconds. That's the mysum function.

Now what about the square? Remember the square had the two nested for loops-- for, for, and then just a regular addition in there. So let's run that. Pretty fast. Pretty fast. Square of 1,000 is already taking 0.05-- or 0.06 seconds. Square of 10,000 is now taking 6 seconds. What do we notice?

With one more round, if we waited for square of 100,000, we might be able to see a pattern, or we can guess the pattern. Does anyone want to wager a guess what the next number should be here when you think about it?

**AUDIENCE:** About 600.

**ANA BELL:** Yeah, about 600. We're going from 0.06 to maybe about 6. So, I don't know, we could say about 600. I'm certainly not going to wait for 600 seconds. And I'm actually not going to make my computer do that, just in case it crashes. But yeah, we could guess something like, on the order of some hundreds, 600, something like that. So that's one thing to notice.

The other thing to notice is that already at 10,000, where the input is just 10,000, this took 5 seconds already. In the previous function here, mysum, we had to get to 100 million as my input to run for 5 seconds. So that's also a big difference here. Already, this program square is taking a really long time to run when the input is not very big.

So some things to notice about timing. And as I said, we're going to look at some more programs next lecture. I just wanted to give you a general sense of timing programs. First of all, the green check is good. We want all these to be green checks. The green check is good because if we have different algorithms, they're going to take a different amount of time. The time that it takes for these algorithms to run will be different, which is good.

But if we have different implementations for the same sort of program, for the same algorithm, that's also going to give us different timings. And really in the long run, I don't really care about that. What I would really like to evaluate is just the algorithm itself. Because when we're talking about algorithms, there's probably only a handful of algorithms out there in the world that we can apply to a given problem. Whereas there's probably thousands of different implementations we can apply to a problem.

So for example, you could have a for loop versus a while loop. You could have creating intermediate variables as an implementation. Or you could have a list comprehension version of an implementation. But underlying all of that is going to be just some algorithm that you're trying to implement.

So the running time will vary between different implementations, which is not really something I want. The running time will also vary between computers. If I ran the same programs on an older computer, it's probably not going to take 5 seconds to run with an input of 100 million. It might take 10 or it might take 11. So the timing is also going to differ between different computers.

It will also differ between different languages, so Java versus Python versus C. If C is very efficient at memory management, it's going to run very fast. Whereas if Python is a little bit slower, it's going to run slower. So again, we're actually capturing-- the timing is capturing variations between languages.

And the timing is not very predictable for small inputs. So if for some reason, when I was running this square function here with 1, I was also running Netflix in the background, or my computer decided to update something. And it decided to just dedicate resources to doing that task at that moment when I'm trying to run this square of 1, this 0.0 seconds might not be 0.0 seconds. It might take away from the time that it allocates to running my square program. And then what we'll see is that this will no longer be 0.0. It might be 0.1 or something like that.

So timing programs is not very good. It's not very consistent with our goal here, which is to evaluate algorithms. Let's see if we can do better with the idea of counting the number of operations. So rather than focusing on describing our program in terms of human time, 1 second, 0.5 seconds, things like that, let's come up with some operations in Python that take one time unit. And we're going to say that all of these really basic operations-- we can say that they take the same amount of time. I don't care if they're like 10 to negative 9 seconds, or 2 times 10 to the negative 9 seconds. I don't care about that. I just know that they're really fast.

And if they're really fast, I can say that each of them just take one unit of time. So I'll just count them all as one unit of time. So the examples of those are mathematical operations. They're pretty fast. So no matter whether I'm multiplying, dividing, adding, subtracting, taking something to the power of something else, I'm going to say that each one of those takes one unit of time.

Comparing something. So A less than B, 3 greater than 4, things like that, equality, also super fast to do, also takes one unit of time. Assigning something. So A is equal to 3. That assignment statement right there-- also pretty fast to do. It takes one unit of time. And then accessing objects in memory. Also pretty fast, takes one unit of time.

So with this new definition of "time," quote, unquote, where we have these units of time. Let's figure out what these functions-- how long these functions actually take. So our Celsius to Fahrenheit function has three operations in it. I got a multiplication, a division, and an addition. I don't care-- so the little variations that each one of these take to actually do inside computer memory, I'm going to say that the Celsius to Fahrenheit program takes three units of time.

So no matter what the input is, if I'm converting 0 Celsius or 1 million Celsius, the program will still just take three units of time to complete. How about mysum? So we'll go through step by step. So in mysum, I've got one assignment statement here. So that's going to be one operation. The for loop here is going to take I and assign it to one of the values in the range. That's just internally what it does. So that's going to be one operation each time through the loop.

And then total plus equals i is going to be two operations, because I have total plus i on the right hand side. That's one operation. And then assigning that back to total is my second operation. So that's two operations there. And that's it. But notice our for loop. These three operations here-- the one for assigning i to be a value here. And these two operations here repeat x plus 1 times. 0 all the way up to x. That's x plus 1 total times.

So how long does this program actually take? Well, we count all that up. So the one for the total equals 0 plus, and we're multiplying x plus 1 times what? The 1 plus the 2, which gives us 3x plus 4 total operations. So now we're noting this in terms of the input, which is kind of cool. So now I have this nice little formula, where if I know my input is 10, I can actually tell you how many quote, unquote, "units of time" this program will take.

How about the square? It's going to be very similar. So I've got one operation for assignment here. This is one operation for grabbing the i and making it one of the values in the range. Similarly for the inner loop, one operation there. And then square sum plus equals 1 for the same reason as this, is two operations. One for the right hand side doing the addition and two for making the assignment.

Let's not forget for loops. We've got two for loops here. So the inner one will repeat n times. And for each one of those outer n times, we do the inner n times. This nested for loop situation here. So the total number of time units that this square will take is the 1 over here for this square sum equals 0, plus, and then I've got these nested for loops. So the other one goes through n times-- sorry, n times the one operation, multiplied by the inner for loop, also n times, times what is the operations done in the inner for loop? Well, it's this 1 plus these 2. So the 1 plus the 2. So in total, 3n squared plus 1 operations.

So let's run this. And now that we're counting operations, we should be able to see a better pattern. So here's my Celsius to Fahrenheit. Mysum and square slightly changed because I've got this little counter variable within each function. That is going to increment each time I see an operation. So obviously, for Celsius to Fahrenheit, it's always 3. So when I do my return, I'm just going to return the counter variable and then the regular thing that this function should return as a tuple.

For mysum, this counter equals 1 stands for this assignment statement. And then each time through the loop, I'm going to increment my counter for the three operations, assigning the i to be one of the values in the range. And then two more for this total plus equals i. So that's going to get incremented each time through the loop. And then the square similarly. So here's my counter equals 1 for this statement here. Counter plus equals 1 for grabbing the i as one of these values. And then counter plus equals 3 for grabbing the j to be one of these values and incrementing this mysum.

So because of where I've placed these counters, Python will automatically count it all up no matter how many loops I've got. So here's my wrapper for counting. Slightly different than the timing one, because now I'm actually going to also keep track of how many more operations I've done compared to the previous input. So let me show you what that means. Let's run Celsius to Fahrenheit with the following inputs.

So I'm first of all reporting the total number of operations just like I did with timing. So always three operations. No surprise there. That's what we coded up basically. But then I'm also reporting here. And that's done inside the wrapper function, the count wrapper, how many more operations is this based on the previous one. So the first one is a little weird. But when my input is 10 times more, I went from 100 to 1,000, I've done one more operation. No change, obviously, because it's always three operations done in total.

So just for completion sake, this is the slide. So no matter what happens to the input here, the number of operations in these sort of units of time, which we're just counting the number of operations, is 3. What about the sum? So remember the sum had that for loop in it. Let's run that and see how many operations are here. So first I'm going to report the number of operations. So when the input is 100, it's 304. When the input is 1,000, it's 3,004. When the input is 10,000, it's 30,004. So that matches up the formula we came up with-- 3x plus 4. So that's pretty cool.

And then you can see now here I'm reporting how many more operations is this line based on the previous line. So it's about 9.8 times more. So when my input increases by 10, from 100 to 1,000, I am doing approximately 9.88 times more operations. When my input increases from 1,000 to 10,000, again, by 10, I'm doing 9.988 times more operations.

So we see a nice little steady state going on here, where when my input gets really, really big, it looks like I'm approaching approximately 10 times as many operations, when my input is 10 times more. This is obviously more apparent when the input is big, because the tiny variations in my formula, the plus 4 specifically, makes less of an impact when my input is really large. And this is kind of going in line with our motivation.

When the input data is really, really big, what I'd like to report is the algorithm and how long it takes. I don't care that the algorithm takes 3x plus 4, or 3x times 3x as operations. When the input is really big, all I care is that it's sort of on the order of x. And that's something we'll get at next lecture. But this is the big idea here. When the input increases by 10, it seems like at steady state, our number of operations increases by 10 as well. So it's this linear relationship.

What about the last function, the square? So I'm doing something a little special here. I have two different inputs I'm going to run the square with. So the first one is L2_a. So I'm going to run square with input 128, 256, 512, 1024-- so I'm basically increasing my input by 2. I'm multiplying my input by 2 each time. And then I'm going to run it with L2_b, where my input increases by 10 each time.

So we're going to see if we can figure out a relationship between these for the square. Because that one was a little hard to figure out in just pure timing without actually waiting for minutes or days. So we've got something to work with here. So here I've got my square. So this first bit here is when my input increased by 2. And down here, it just finished, is when my input increases by 10.

So number of operations when my input increases by 2 are not so important. Yes, they're big. But what I'm really interested in, just like what we saw in the mysum example, is what happens to the steady state as the input gets really big. How many more operations are we doing? And what we can see is that the number of operations as the input gets really big is approximately 4 times more, in the case where I increase my input by 2 every round.

So when I increase my input by 2, the number of operations are going to be 4 times more. Well, what about when I increase my input by 10? 1, 100, 1,000, and so on. Again, I'm not so much interested in number of operations. But what happens to the steady state? With very few operations, it's hard to tell. But as we increase it, we see that it goes towards approximately 100. So when my input increases by 10, that takes me to about 100-fold increase in the number of operations.

So now do you guys-- can you guys see the relationship between the input for the square and the number of operations? You can, right? So it's approximately a sort of an n squared relationship. When my input is n, the number of operations is going to be on the order of n squared more.

So counting operations is actually a lot better than timing, as we can see. We've eliminated a bunch of those red X's. We no longer have to deal with variations between computers, because if we're counting this on a computer that's slow or fast, we're still counting the same amount of stuff. Languages-- again, it's not going to matter, because you'll implement it in a similar way. Small inputs is still sort of iffy. We saw the square was a little bit unpredictable when the input was pretty small, right down here-- 60 then straight up to 90.

But we didn't take long to see the steady state. So it's actually better than before, better than timing. It's not 0 at least. But now the problem becomes sort of what's the definition of which operations to count? Notice our functions have a return value. Do we count the return as an operation? Technically, you should. That's a value that's being passed between functions. So that's going to take some time to run.

But we didn't actually count it in our example. But you could if you wanted to. So that's where we stand. We've got timing and counting, just as an initial example. Next lecture, we're going to look at a few more examples with lists and things like that. Just again, timing and counting those functions.

But again, the big idea here is that we're trying to get at evaluating just a handful of different algorithms, sort of what's the order of growth as the input becomes really, really big? Because all we're interested in is how scalable are these programs that we're running when the input is really big, when we're dealing with big data. And so that's what we're going to start talking about next lecture.