PROFESSOR: Today, we're moving on to what will be a major unit of the course, which is the topic of efficiency. Thus far, we focused our attention on the admittedly more important problem, getting our programs to work, i.e., to do what we want them to do. For the next several lectures, I want to talk about how do we get them to work quickly enough to be useful.

It is in practice often a very important consideration in designing programs. The goal is not to make you an expert in this topic. It's hard to be an expert in this topic. I'm certainly not an expert. But I want to give you some intuition about how to approach the question of efficiency, how to understand why some programs take much longer to run than others, and how to go about writing programs that will finish before you die. And we'll see that if you write things wrong, the programs could, in principle, run longer than you can.

So why is efficiency so important? Earlier in the term, I started to spend some time talking about how really fast computers are and showing you that we can use brute force algorithms to solve fairly large problems. The difficulty is that some of the computational problems we're confronted with are not fairly large but enormous.

So for example, in my research group where we work at the intersection of computer science and medicine, we have a big database of roughly a billion and half heart beats. And we routinely run computations that run for two weeks on that data. And the only reason they complete it in two weeks and not two years is we were really careful about efficiency. So it really can matter. And increasingly, it matters is we see the scale of problems growing.

The thing I want you to take home to remember is that efficiency is rarely about

clever coding. It's not about some little trick that saves one instruction here or two instructions there. It's really about choosing the right algorithm. So the take home message is that efficiency is about algorithms, not about coding details.

Clever algorithms are hard to invent. A successful computer scientist might invent maybe one in his or her whole career. I have to say I invented zero important algorithms in my whole career. Therefore, we don't depend upon being able to do that. Instead what we depend upon is problem reducing. When confronted with a problem, we want to reduce it to a previously solved problem. And this is really often the key to taking some problem and fitting into a useful computation. We sit back, say, well, this looks a little bit like this other problem. How come I transform my problem to match a problem that some clever person already knows how to solve?

Before I spend time on problem reduction, however, I want to draw back and look at the general question of how do we think about efficiency. When we think about it, we think about it in two dimensions, space and time. And as we'll see later in the term, we can often trade one for the other. We can make a program run faster by using more memory or use less memory at the cost of making it run more slowly. For now, and the next few lectures, I'm going to focus on time. Because really, that's mostly what people worry about these days when they're dealing with complexity.

So now, suppose I ask you the question, how long does some algorithm implemented by a program take to run? How would you go about answering that question? Well, you could say, all right, I'm going to run it on some computer on some input and time it. Look at my watch. That took three minutes. I ran this other algorithm, and it took two minutes. It's a better algorithm.

Well, that would be really a bad way to look at it. The reasons we don't think about computational complexity, and that's really what people call this topic in terms of how long a program takes to run on a particular computer, and it's not a stable measure. To do that, it's influenced by the speed of the machine. So a program that took 1 minute on my computer might take 30 seconds on yours. It has to do with the

cleverness of the Python implementation. Maybe I have a better implementation of Python than you do, so my programs will run a little bit faster.

But most importantly, the reason we don't depend upon running programs is it depends upon the input. So I might choose one input for which the program took 2 minutes and another seemingly similar input in which it took 1 hour. So I need to get some way to talk about it more abstractly. The way we do that is by counting the number of basic steps.

So we define some function, say time, which maps the natural numbers to the natural numbers. The first n, in this case, the first natural number, the argument corresponds to the size of the input, how big an input do we want to run the program on. And the result of the function is the number of steps that the computation will take for an input of that size. I'll come back to this in a little bit more precise detail momentarily.

A step is an operation that takes constant time. And that's important. So steps are not variable, but they're constant. So we have lots of these, for example, an assignment, a comparison, an array access, et cetera. In looking at computational complexity in this course, we're going to use a model of the computer. It's known as random access, a random access machine, frequently abbreviated as RAM.

In a random access machine, instructions are executed one after another, that is to say they're sequential. Only one thing happens at a time. And we assume constant time required to access memory. So we can access at random any object in memory in the same amount of time as any other object.

In the early days of computers, this model was not accurate, because memory was often say, a tape. And if you wanted to read something at the end of the tape, it took a lot longer to read than something at the beginning of the tape. In modern computers, it's also not quite accurate. Modern computers have what's called a memory hierarchy where you have levels of memory, the level one cache, the level two cache, the actual memory. And it can differ by say a factor of a 100, how long it takes access data depending upon whether it's in the cache. The cache keeps track

of recently accessed objects.

Nevertheless, if we start going into that level of detail, we end up losing the forest for the trees. So almost everybody when they actually try and analyze algorithms typically works with this model. We also know in modern computers that some things happen in parallel. But again, for most of us, these will be second order effects. And the random access model is quite good for understanding algorithms.

Now when we think about how long an algorithm will take to run, there are several different ways we could look at it. We could think of the best case. And as we think about these things, as a concrete example, we can think about linear search. So let's say we have an algorithm that's using linear search. We've looked at that before to find out whether or not an element is in the list. Well, the best case would be that the first element is three, and I'm searching for 3, and I find it right away, and I stop. So that would be my best case complexity. It's the minimum running time over all possible inputs. Is the best case.

I can also look at the worst case. What's the worst case for linear search? It's not there. Exactly. So I go and I have to look at every element, and whoops, it's not there. So the worst case is the maximum over all possible inputs of a given size. The size here is the length of the list.

And then I can ask what's the expected or average case, what would happen most of the time. The expected case seems, in principle, like the one we should care about. But the truth is when we do algorithmic analysis, we almost never deal with the expected case because it's too hard. We think about the expected case for say linear search, we can't talk about it without some detailed model of what the list itself looks like, what elements are in it, and what the distribution of queries looks like.

Are we most of the time asking for elements that are not in the list in which case the expected value is out here somewhere? Or are we most of the time looking for things that are in the list in which case the expected value would be somewhere near halfway through the length of the list? We don't know those things. We have a tough time modeling expected value.

And one of the things we know is that frequently we don't -- when we release a program -- have a good sense of how people will actually use it. And so we don't usually focus on that. Similarly, we don't usually focus on the best case. It would be nice. But you could imagine that it's not really what we care about, what happens when we get really lucky. Because we all believe in Murphy's law. If something bad can happen, it will. And that's why complexity analysis almost always focuses on the worst case.

What the worst case does is it provides an upper bound. How bad can things possibly get? What's the worst that can happen? And that's nice because it means that there are no surprises. You say the worst that this thing can do is look at every element of the list once. And so if I know that the list is a million elements, I know, OK, it might have to do a million comparisons. But it won't have to do any more than a million. And so I won't be suddenly surprised that it takes overnight to run. Alas, the worst case happens often. We do frequently end up asking whether something is in a list, and it's not. So even though it seems pessimistic to worry about the worst case, it is the right one to worry about.

All right. Let's look at an example. So I've got a little function here, f. You can see it here. It's on the handout as well. First of all, what mathematical function is f computing, just to force you to look at it for a minute? What's it computing? Somebody? It is a function that should be familiar to almost all of you. Nobody? Pardon.

AUDIENCE: Exponentiation.

PROFESSOR: Exponentiation? Don't think so. But I appreciate you're trying. It's worth some candy, not a lot of candy, but a little candy. Yeah?

AUDIENCE: Factorial.

PROFESSOR: Factorial. Exactly. It's computing factorial. Great grab.

So let's think about how long this will take to run in terms of the number of steps.

Well, the first thing it does is it executes an assertion. And for the sake of argument for the moment, we can assume that most instructions in Python will take one step. Then, it does an assignment, so that's two steps. Then, it goes through the loop. Each time through the loop, it executes three steps, the test at the start of the loop and the two instructions inside the loop. How many times does it go through the loop? Somebody? Right. n times. So it will be 2 plus 3 times n. And then it executes a return statement at the end.

So if I want to write down the function that characterizes the algorithm implemented by this code, I say it's 2 plus 3 times n plus 1. Well, I could do that. But it would be kind of silly. Let's say n equals 3,000. Well, if n equals 3,000, this tells me that it takes 9,000-- well, what does it take? 9,003 steps. Right. Well, do I care whether it's 9,000 or 9,003? I don't really.

So in fact, when I look at complexity, I tend to-- I don't tend to I do ignore additive constants. So the fact that there's a 2 here and a 1 here doesn't really matter. So I say, well, if we're trying to characterize this algorithm, let's ignore those. Because what I really care about is growth with respect to size. How does the running time grow as the size of the input grows?

We can even go further. Do I actually care whether it's 3,000 or 9,000? Well, I might. I might care whether a program take say 3 hours to run or 9 hours to run. But in fact, as it gets bigger, and we really care about this as things get bigger, I probably don't care that much. If I told you this was going to take 3,000 years or 9,000 years, you wouldn't care. Or probably, even if I told you it was going to take 3,000 days or 9,000 days, you'd say, well, it's too long anyway.

So typically, we even ignore multiplicative constants and use a model of asymptotic growth that talks about how the complexity grows as you reach the limit of the sizes of the inputs. This is typically done using a notation we call big O notation written as a single O. So if I write order n, O(n), what this says is this algorithm, the complexity, the time grows linearly with n. Doesn't say whether it's 3 times n or 2 times n. It's linear in n is what this says.

Well, why do we call it big O? Well, some people think it's because, oh my God, this program will never end. But in fact, no. This notion was introduced to computer science by Donald Knuth. And he chose the Greek letter omicron because it was used in the 19th century by people developing calculus. We don't typically write omicron because it's harder to types. So we usually use the capital Latin letter O, hence, life gets simple.

What this does is it gives us an upper bound for the asymptotic growth of the function. So formerly, we would write something like f of x, where f is some function of the input x, is order, let's say x squared. That would say it's quadratic in the size of x. Formally what this means is that the function f-- I should probably write this down-- the function f grows no faster than the quadratic polynomial x squared.

So let's look at what this means. I wrote a little program that talks about some of the-- I should say probably most popular values we see. So some of the most popular orders we would write down, we often see order 1. And what that means is constant. The time required is independent of the size of the input. It doesn't say it's one step. But it's independent of the input. It's constant.

We often see order log n, logarithmic growth. Order n, linear. One we'll see later this week is nlog(n). This is called log linear. And we'll see why that occurs surprisingly often. Order n to the c where c is some constant, this is polynomial. A common polynomial would be squared as in quadratic. And then, if we're terribly unlucky, you run into things that are order c to the n exponential in the size of the input.

To give you an idea of what these classes actually mean, I wrote a little program that produces some plots. Don't worry about what the code looks like. In a few weeks, you'll be able to write such programs yourself. Not only will you be able to, you'll be forced to. So I'm just going to run this and produce some plots showing different orders of growth. All right. This is producing these blocks. Excuse me. I see.

So let's look at the plots. So here, I've plotted linear growth versus logarithmic growth. And as you can see, it's quite a difference. If we can manage to get a

logarithmic algorithm, it grows much more slowly than a linear algorithm. And we saw this when we looked at the graded advantage of binary search as opposed to linear search. Actually, this is linear versus log linear. What happened to figure one? Well, we'll come back to it. So you'll see here that log linear is much worse than linear. So this factor of nlog(n) actually makes a considerable difference in running time.

Now, I'm going to compare a log linear to quadratic, a small degree polynomial. As you can see, it almost looks like log linear is not growing at all. So as bad as log linear looked when we compared it to linear, we see that compared to quadratic, it's pretty great. And what this tells us is that in practice, even a quadratic algorithm is often impractically slow, and we really can't use them. And so in practice, we worked very hard to avoid even quadratic, which somehow doesn't seem like it should be so bad. But in fact, as you can see, it gets bad quickly. Yeah, this was the log versus linear, not surprising.

And now, if we look at quadratic versus exponential, we can see hardly anything. And that's because exponential is growing so quickly. So instead, what we're going to do is I'm going to plot the y-axis logarithmically just so we can actually see something. And as you can see on input of size 1,000, an exponential algorithm is roughly order 10 to the 286th. That's an unimaginably large number. Right? I don't know what it compares to, the number of atoms in the universe, or something ridiculous, or maybe more. But we can't possibly think of running an algorithm that's going to take this long. It's just not even conceivable.

So exponential, we sort of throw up our hands and say we're dead. We can't do it. And so nobody uses exponential algorithms for everything, yet for anything. Yet as we'll see, there are problems that we care about that, in principle, can only be solved by exponential algorithms. So what do we do? As we'll see, well, we usually don't try and solve those problems. We try and solve some approximation to those problems. Or we use some other tricks to say, well, we know the worst case will be terrible, but here's how we're going to avoid the worst case. We'll see a lot of that towards the end of the term. The moral is try not to do anything that's worse than

log linear if you possibly can.

Now some truth in advertising, some caveats. If I look at my definition of what big O means, I said it grows no faster than. So in principle, I could say, well, what the heck, I'll just write 2 to the x here. And it's still true. It's not faster than that. It's not what we actually want to do. What we actually want is a type bound. We'd like to say it's no faster than this, but it's no slower than this either, to try and characterize the worst cases precisely as we can.

Formally speaking, a theorist used something called big Theta notation for this. They write a theta instead of an O. However, most of the time in practice, when somebody writes something like f of x is order x squared, what they mean is the worst case is really about x squared. And that's the way we're going to use it here. We're not going to try and get too formal. We're going to do what people actually do in practice when they talk about complexity.

All right, let's look at another example now. Here, I've written factorial recursively. Didn't even try to disguise what it was. So let's think about how we would analyze the complexity of this. Well, we know that we can ignore the first two lines of code because those are just the additives pieces. We don't care about that-- the first line, and the if, and the return. So what's the piece we care about? We care about the number of times the factorial is called.

In the first implementation of factorial, we cared about the number of iterations of a loop. Now instead of using a loop, you use recursion to do more or less the same thing. And so we care about the number of times fact is called. How many times will that happen? Well, let's think about why I know this doesn't run forever, because that's always the way we really think about complexity in some sense.

I know it doesn't run forever because each time I call factorial, I call it on a number one smaller than the number before. So how many times can I do that if I start with a number n? n times, right? So once again, it's order n. So the interesting thing we see here is that essentially, I've given you the same algorithm recursively and iteratively. Not surprisingly, even though I've coded it differently, it's the same

complexity.

Now in practice, the recursive one might take a little longer to run, because there's a certain overhead to function calls that we don't have with while loops. But we don't actually care about that. Its overhead is one of those multiplicative constants I said we're going to ignore. And in fact, it's a very small multiplicative constant. It really doesn't make much of a difference.

So how do I decide whether to use recursion or iteration has nothing to do with efficiency, it's whichever is more convenient to code. In this case, I kind of like the fact that recursive factorial is a little neater. So that's what I would use and not worry about the efficiency.

All right. Let's look at another example. How about g? What's the complexity of g? Well, I can ignore the first statement. But now I've got two nested loops. How do I go and think about this? The way I do it is I start by finding the inner loop. How many times do I go through the inner loop? I go through the inner loop n times, right? So it executes the inner for statement is going to be order n. The next question I ask is how many times do I start the inner loop up again? That's also order n times. So what's the complexity of this? Somebody?

AUDIENCE: n-squared.

PROFESSOR: Yes. I think I heard the right answer. It's order n-squared. Because I execute the inner loop n times, or each time around is n, then I multiply it by n because I'm doing the outer loop n times. So the inner loop is order n-squared. That makes sense?

So typically, whenever I have nested loops, I have to do this kind of reasoning. Same thing if I have recursion inside a loop or nested recursions. I start at the inside and work my way out is the way I do the analysis. Let's look at another example. It's kind of a different take. How about h? What's the complexity of h? First of all, what's h doing? Kind of always a good way to start. What is answer going to be? Yeah?

AUDIENCE: The sum of the [UNINTELLIGIBLE].

PROFESSOR: Right. Exactly. It's going to be the sum of the digits. Spring training is already under way, so sum of the digits. And what's the complexity? Well, we can analyze it. Right away, we know we can ignore everything except the loop. So how many times do I go through this loop? It depends upon the number of digits in the string representation of the int, right?

Now, if I were careless, I would write something like order n, where n is the number of digits in s. But really, I'm not allowed to do that. Why not? Because I have to express the complexity in terms of the inputs to the program. And s is not an input. s is a local variable. So somehow, I'm going to have to express the complexity, not in terms of s, but in terms of what? x. So that's no go. So what is in terms of x? How many digits? Yeah?

AUDIENCE: Is it constant?

PROFESSOR: It's not constant. No. Because I'll have more digits in a billion than I will in four. Right. Log -- in this case, base 10 of x. The number of decimal digits required to express an integer is the log of the magnitude of that integer. You think about we looked at binary numbers and decimal numbers last lecture, that was exactly what we were doing. So that's the way I have to express this.

Now, what's the moral here? The thing I really care about is not that this is how you talk about the number of digits in an int. What I care about is that you always have to be very careful. People often think that they're done when they write something like order n. But they're not until they tell you what n means, Because that can be pretty subtle. Order x would have been wrong because it's not the magnitude of the integer x. It's this is what controls the growth. So whenever you're looking at complexity, you have to be very careful what you mean, what the variables are. This is particularly true now when you look at functions say with multiple inputs.

Ok. Let's look at some more examples. So we've looked before at search. So this is code you've seen before, really. Here's a linear search and a binary search. And in fact, informally, we've looked at the complexity of these things before. And we can

run them, and we can see how they will grow. But it won't surprise you. So if we look at the linear search-- whoops, I'm printing the values here, just shows it works. But now, the binary search, what we're going to look at is how it grows. This is exactly the search we looked at before.

And the thing I want you to notice, and as we looked at before, we saw it was logarithmic is that as the size of the list grows, doubles, I only need one more step to do the search. This is the beauty of a logarithmic algorithm. So as I go from a 100, which takes 7 steps, to 200, it only takes 8. 1,600 takes 11. And when I'm all the way up to some very big number, and I'm not even sure what that number is, it took 23 steps. But very slow growth. So that's a good thing.

What's the order of this? It's order n where n is what? Order log n where n is what? Let's just try and write it carefully. Well, it's order length of the list. We don't care what the actual members of the list are.

Now, that's an interesting question to ask. Let's look at the code for a minute. Is that a valid assumption? Well, it seems to be when we look at my test. But let's look at what I'm doing here. So a couple of things I want to point out. One is I used a very common trick when dealing with these kinds of algorithms. You'll notice that I have something called bsearch and something called search.

All search does is called bsearch. Why did I even bother with search? Why didn't I just with my code down here call bsearch with some initial values? The answer is really, I started with this search. And a user of search shouldn't have to worry that I got clever and went from this linear search to a binary search or maybe some more complex search yet. I need to have a consistent interface for the search function.

And the interface is what it looks like to the caller. And it says when I call it, it just takes a list and an element. It shouldn't have to take the high bound and the lower bound as arguments. Because that really is not intrinsic to the meaning of search. So I typically will organize my program by having this search look exactly like this search to a caller. And then, it does whatever it needs to do to call binary search. So that's usually the way you do these things. It's very common with recursive

algorithms, various things where you need some initial value that's only there for the initial call, things like that.

Let me finish-- wanted to point out is the use of this global variable. So you'll notice down here, I define something called NumCalls. Remember we talked about scopes. So this is now an identifier that exists in the outermost scope of the program. Then in bsearch, I used it. But I said I'm going to use this global variable, this variable declared outside the scope of bsearch inside bsearch. So it's this statement that tells me not to create a new local variable here but to use the one in the outer scope.

This is normally considered poor programming practice. Global variables can often lead to very confusing programs. Occasionally, they're useful. Here it's pretty useful because I'm just trying to keep track of a number of times this thing is called. And so I don't want a new variable generated each time it's instantiated. Now, you had a question? Yeah?

AUDIENCE: Just checking. The order len L, the size of the list is the order of which search?

PROFESSOR: That's the order of the linear search. The order of the binary search is order log base 2 of L-- sorry, not of L, right? Doesn't make sense to take the log of a list of the length of the list. Typically, we don't bother writing base 2. If it's logarithmic, it doesn't really very much matter what the base is. You'll still get that very slow growth. Log base 10, log base 2, not that much difference. So we typically just write log. All right. People with me?

Now, for this to be true, or in fact, even if we go look at the linear search, there's kind of an assumption. I'm assuming that I can extract the elements from a list and compare them to a value in constant time. Because remember, my model of computation says that every step takes the same amount of time, roughly. And if I now look say a binary search, you'll see I'm doing something that apparently looks a little bit complicated up here. I am looking at L of low and comparing it to e, and L of high and comparing it to e.

How do I know that's constant time? Maybe it takes me order length of list time to extract the last element. So I've got to be very careful when I look at complexity, not to think I only have to look at the complexity of the program itself, that is to say, in this case, the number of recursive calls, but is there something that it's doing inside this function that might be more complex than I think. As it happens, in this case, this rather complicated expression can be done in constant time. And that will be the first topic of the lecture on Thursday.