ERIC GRIMSON: Last time, we started talking about complexity. And I want to quickly remind you of what we're doing, because we're going to talk some more about that today. And when I say "complexity," it was the question of, can we estimate the amount of resources-- typically time-- that we're going to need to get an algorithm to solve a problem of a particular size? And we talked about that both in terms of estimating, how much time is it going to take and using it to go the other direction and think about how design choices in an algorithm have implications for the cost that's going to be associated with that.

So we introduced the idea of what we call big O notation, orders of growth, a way of measuring complexity. And we started talking about different classes of algorithms. So today, I'm going to quickly recap those basic ideas. And what we're going to do is we're going to see examples of standard classes of algorithms with the idea that you're going to want to begin to recognize when an algorithm is in that class.

So very quick recap-- I've already said part of this. We want to have a mechanism, a method for being able to estimate or reason about, how much time do we think an algorithm's going to take to solve a problem of a particular size. And especially as we increase the size of the input to the algorithm, what does that do in terms of the increase in the amount of time that we need?

Some ways, we don't care about exact versions. In a second, we're going to see the definition of this. But what we care about is that notion of, how does it grow as we increase the problem size. And what we're going to focus in going, if you like, in the forward direction-- as I said, a lot of the interest is actually thinking about what you might think of as the backwards or reverse direction. How does a choice in algorithm design impact efficiency of the algorithm?

And really, what we want you to do is to begin to recognize standard patterns, that, if you make a particular choice, this fits in a class of algorithms you've seen before. And I know, in essence, how much time-- what the cost is going to be as I do that.

All right, so just to recap, as it says on the top, we talked about orders of growth. And the idea is, we want to be able to evaluate a program's efficiency when the input is very big. We talked about timing things just with a timer. We suggested that, unfortunately, conflates the algorithm with the implementation with the particular machine. We want to get rid of those latter two and just focus on the algorithm. So we're going to talk about counting operations, but in a very general sense. So we're going to express what we call the growth of the program's runtime as the input size grows very large.

And because we're only interested not in the exact number, but in, if you like, the growth of that, we're going to focus on putting an upper bound on the growth, an expression that grows at least as fast as what the cost of the algorithm is. Now, you could just cheat and pick a really big upper bound. That doesn't help us very much. So in general, we're going to try and use as tight an upper bound as we can. What's the class of algorithm it falls in?

But as we've seen before, we care about the order of growth, not being exact. So if something grows as 2 to the n plus 5n, I don't care about the 5n. Because when n gets big, that 2 to the n is the really big factor. And therefore we're going to look at the largest factors when we think about that.

We've seen some examples. We're going to do a bunch more examples today to fill those in. One of the things that we want to now do is say, with that idea in mind, there are classes of complexity of algorithms. So in some sense, the best ones are way up here, O of 1, order 1, constant. It says cost doesn't change as I change the size of the input. That's great. It's always going to be the same cost.

Order log n says the cost grows logarithmically with the size of the input. It's a slow growth, and I'm going to remind you of that in a second. We saw lots of examples of linear running time-- we're going to see a few more today-- what we call log-linear, polynomial, and exponential.

And the thing I want to remind you is that, ideally-- whoops, sorry-- we'd like our algorithms to be as close to the top of this categorization as we can. This is actually described in increasing order of complexity. Something that takes the same amount of time no matter how big the input is, unless that amount of time is a couple of centuries, seems like a really good algorithm to have. Something that grows linearly is not bad. Something that grows, as we've seen down here, exponentially tends to say, this is going to be painful.

And in fact, you can see that graphically. I'll just remind you here. Something that's constant says, if I draw out the amount of time it takes as a function of the size of the input, it doesn't change. Logarithmic glows-- gah, sorry-- grows very slowly. Linear will grow, obviously, in a linear way.

And I actually misspoke last time. I know it's rare for a professor to ever admit they misspeak, but I did. Because I said linear is, if you double the size of the input, it's going to double the amount of time it takes. Actually, that's an incorrect statement. Really, what I should have said was, the increment-- if I go from, say, 10 to 100, the increase in time-- is going to be the same as the increment if I go from 100 to 1,000. Might be more than double depending on what the constant is. But that growth is linear.

If you want to think of it, take the derivative of time with respect to input size. It's just going to be a constant. It's not going to change within. And of course, when we get down to here, things like exponential, it grows really fast.

And just as a last recap, again, I want to be towards the top of that. There was my little chart just showing you things that grow constant, log, linear, log-linear, quadratic, and exponential. If I go from n of 10, to 100, to 1,000, to a million, you see why I want to be at the top of that chart.

Something up here that grows logarithmically, the amount of time grows very slowly as I increase the input. Down here, well, like it says, good luck. It's going to grow up really quickly as I move up in that scale. I want to be at the top of this chart if I can.

OK, with that in mind, when I'm going to do today is show you examples filling in most of this chart. We've already seen some examples. We've seen examples of linear. We've seen examples of quadratic. I'm going to just remind you of those. What I want to do is show you how you can begin to recognize a choice of an algorithm in terms of where it lies.

So algorithms that are constant complexity, they're kind of boring. They tend to be pretty simple. Because this says, this code is going to run in, basically, the same amount of time independent of the size of the input. Now, notice the bottom thing here. It doesn't say you can't-- blah, try again. It doesn't say you couldn't have a loop or a recursive call. You could, it's just that that loop cannot depend on the size of the input.

So there aren't many interesting algorithms here. We're going to see pieces of code that fit into this when we do our analysis. But something that's constant in complexity says, independent of the size of the input.

All right, a little more interesting-- not a little, a lot more interesting-- are algorithms that are logarithmic in their complexity. So they're going to grow with the logarithm of the size of the input. You saw an example much earlier in the term when Ana showed you bisection search. It was searching for a number with a particular property.

I want to show you another example, both to let you recognize the form of the algorithm, but especially, to show you how we can reason about the growth. And that's another trick called binary search or, again, it's a version of bisection search. Suppose I give you a list, a list of numbers, integers. And I want to know if a particular element is in that list. We saw, last time, you could just walk down the list, just iterate through the entire list looking to see if it's there. In the worst case, which is what we worry about, it's going to be linear. You're going to have to look at every element in the list till you get to the end. So complexity was linear in that case.

And then we said, suppose we know that the list is sorted. It's ordered from smallest to largest. And we saw, a simple algorithm says, again, walk down the list checking to see if it's there. But when you get to an element that's bigger than the thing you're looking at, you can just stop. There's no reason to look at the rest of the list. They've all got to be bigger than the thing you're searching for.

Practically, in the average case, that's going to be faster than just looking at an unsorted list. But the complexity is still linear. Because in the worst case, I've got to go all the way through the list before I deduce that the thing is not there. OK, so even sequential search in an ordered list is still linear. Can we do better? And the answer is, sure.

So here's how we do better. I'm going to take that list. I'm going to assume it's sorted. And I'm going to pick an index that divides the list in half, just pick the midpoint in the list. And I'm going to check that value. I'm going to ask, is the element in the list at that point the thing I'm looking for? If it is, great, I'm done.

If I'm not that lucky, I'm then going to ask, is it larger or smaller than the thing I'm looking for? And based on that, I'm either going to search the front half or the back half of the list. Ooh, that's nice, OK? Because if you think about it, in something that was just a linear algorithm, at each step, I reduced the size the problem by 1. I went from a problem of size n to a problem of size n minus 1 to a problem of size n minus 2.

Here, I'm taking a problem of size n. I'm reducing it to n/2 in one step, because I can throw half the list away. So this is a version of divide and conquer, things we've seen before. I'm breaking it down into smaller versions of the problem. So let's look at that. And then, let's write some code. And then, let's analyze it.

So suppose I have a list of size n, all right? There are n elements in there. I'm going to look at the middle one, say, is it the thing I'm looking for. If it's not, is it bigger than or less than the thing I'm looking for? And in this case, let's assume that, in fact, the thing I'm looking for is smaller than that element. Great. I'm going to throw away half the list. Now I only have to look at the lower half of the list.

I'll do the same thing. I'll look at the element in the middle here. And I'll say, is it the thing I'm looking for? If not, is it bigger than or smaller than the thing I'm looking for? OK, and I'm down to n/2 elements. And after I do that, I throw away half the list again. In this case, I'm assuming that the thing I'm looking for is bigger than that middle point. Until I find it, at each step, I'm looking at the middle element. And I'm either throwing away the left half or the right half of that list.

So after i steps, I'm down to a list of size n over 2 to the i. Now, what's the worst case? The worst case is the element's not in the list. I'm going to have to keep doing this until I get down to just a list of one element. And at that point, if it's not the thing I'm looking for, I know I'm done, and I can stop.

Different pattern-- notice how I'm cutting down the size of the problem by 2. So I can ask, before we look at the code, what's the complexity of this? How many steps do I have to go through in the worst case? And I know I'm going to be done looking at the list when n over 2 to the i is equal to 1, meaning there's only one element left that I'm still looking at. And I can solve that. It says I'm going to have to take, at most, i equal to log n steps, all right? So logarithmically, I'm cutting this down.

And so the complexity of the recursion-- we haven't talked about the code yet, but in terms of the number of steps I have to do in the worst case-- is just logarithmic in the length of the list. That's nice. It's a lot better than looking at everything inside the list. And in fact, you can see it, right? I don't look at everything inside the list here. I'm throwing half the things away at a time.

OK, so let's look at some code to do that. Bisection search-- I'm going to give it a list of numbers. I'm going to give it something I'm looking for. We can walk through this code. Hopefully it's something that you're going to be able to recognize pretty clearly.

It says if the list is empty, there's nothing there, the thing I'm looking for can't be there, I return False. If there's exactly one element in the list, then I just check it. If that thing's the thing I'm looking for, return True. Otherwise, return False. So I'm just going to return the value there. Otherwise, find the midpoint-- notice the integer division here-- find the midpoint in that list and check it. In particular, say, if the thing at the midpoint is bigger than the thing I'm looking for, then I'm going to return a recursive call to this function only looking at the first half of the list. I'm just slicing into it. Otherwise, I'll do the same thing on the second half of the list.

Nice, this is implementing exactly what I said. We could actually try it. I'll do that in a second if I remember. But let's think about complexity here. That's constant, right? Doesn't depend on the size of the list. That's constant, doesn't depend on the size of the list. That's consonant. Sounds good. And what about that?

Well, it looks like it should be constant, right, other than the number of times I have to go through there. Remember, I know I'm going to have order log n recursive calls. I'm looking at what's the cost to set it up. It looks like it should be constant. So does that. But I'm going to claim it's not. Anybody see why it's not? You can look at the slides you've already printed out.

Right there-- I'm actually copying the list, all right? When I slice into the list like that, it makes a copy of the list. Oh, crud. I was about to say something different, but I won't, because this is going to cost me a little bit as I think about the work. So let's look at that a little more carefully.

I've got order log n search calls. We just deduced that. I've just repeated it here, right? On each call, I'm reducing the size of the list in half. So it goes from n, to n/2, to n/4, to n/8, to n/16. I'll be done, in the worst case, when I get down to having only a list of size 1. That takes a log n steps, because n over 2 to the log n is n/n, which is 1.

But to set up the search for each cell, I've got to copy the list. And the list starts out n long, so in principle, I've got order n work to do to set up the recursive call. And so by the things we saw last time, I got order log n for the number of recursive calls times order n work inside of each call. And that's order n log n. So it's not what I wanted.

Now, if you're thinking about this carefully, you'll realize, on each step, I'm not actually copying the whole list. I'm copying half the list, and then, a quarter of the list, and then, an eighth of the list. So if I was actually really careful-- I'm not going to do the math here-- and in fact, what we'll see-- and if you like, in your copious spare time, you can go off and work this through-- what you'll discover is that you're actually doing order n work to do the copying. But that's still a problem, because then, I've got something that's order n plus log n. And the n is going to dominate, so this is still linear.

Sounds like I led you down a primrose path here. Can we fix this? Sure. Because we could do the following.

We could say, when I want to look at that list, do I need to copy everything? What about if, instead, I said, here's the beginning and the end of the list. When I test the middle, I'll move one of the pointers to the middle of the list. When I test the middle again, I'll move another pointer in. So in other words, I can test the middle. And based on that, I could say, I only need to search this part of the list. Just keep track of that point and that point in the list. And when I test the middle again, same idea.

Now I'm not actually copying the list, I am simply keeping track of, where are the pieces of the list that bound my search. Ha, all right? I'm still reducing the size of the problem by a factor of 2 at each step. That's great. All I'd need to do now, though, is just keep track of which portion of the list I'm searching. I'm going to avoid copying the list. So the number of recursive calls, again, will be logarithmic. Let's see if that actually fixes my problem.

A little bit of code, not as bad as it looks-- I've got an internal function here that I'm going to come back to. But let's look at what happens in this case. I'm going to say, again, if there's nothing in the list, just return False. Element can't possibly be there. Otherwise, call this function with the list, the element for whom I'm searching, and the beginning and end of the list-- so 0 at one end, length of n l minus 1 at the other end. It's just that idea of, I'm keeping track of the two pieces, OK?

Now let's look at what this does. It says, here's the low part of the list, the high part of the list. Initially, it's 0 and length of list minus 1. It says, if they are the same, oh cool, then I've got a list of length 1. Just test to see if it's the thing I'm looking for. If they're not, find the midpoint. And the midpoint's just the average of low plus high, integer division by 2. Think about it. If it's 0 and n, midpoint is n/2.

But if it's, for example, n/2 and n, midpoint is 3/4 n. So that mid picks the middle point. If it's the thing I'm looking for, great, I'm done. Otherwise, check to see, is the thing at the middle bigger than or less than the thing I'm looking for. And based on that-- I'm going to skip this one for a second-- I'm either going to search everything from the low point up to the middle point or from the middle point up to the high point. And the last piece here is, if, in fact, the low point and the middle point are the same, I've got a list of size 1. There's nothing left to do. I'm done.

OK, I know it's a lot of code. I would invite you just to walk through it. But I want to take you back again to just, simply, this point and say, here's what we're doing. We're starting off with pointers at the beginning and end of the list. We're testing the middle point. And based on that, we're giving a call where, now, the pointer is to the beginning and the middle of the list, simply passing it down, and same as I go through all of these pieces.

So that code now gives me what I'd like. Because here, in the previous case, I had a cost. The cost was to copy the list. In this case, it's constant. Because what am I doing? I'm passing in three values. And what does it take to compute those values? It's a constant amount of work, because I'm simply computing mid right there, just with an arithmetic operation.

And that means order log n steps, because I keep reducing the problem in half. And the cost at each point is constant. And this is, as a consequence, a really nice example of a logarithmic complexity function.

Now, if you think about it, I'm cheating slightly-- second time today. Because we said we really don't care about the implementation. We want to get a sense of the complexity of the algorithm. And that's generally true. But here is a place in which the implementation actually has an impact on that complexity. And I want to be conscious of that as I make these decisions. But again, logarithmic in terms of number of steps, constant work for each step, because I'm just passing in values. And as a consequence, the overall algorithm is log.

Notice one other thing. I said I want you to see characteristics of algorithms that tell you something about the complexity of that algorithm. Something that's iterative and reduces the problem by size 1 each time, from n, to n minus 1, to n minus 2-- linear. Something that reduces the size of the problem in half, or in thirds, or in quarters each time-- logarithmic, generally, unless I've got a hidden cost somewhere.

Here's another little example just to give you a sense of log. I want to convert an integer to a string. I know I can just call str() on it. But how might we do that inside of the machine?

Well, here's a nice little algorithm for it. I'm going to set up something I call digits. It's just a string of all the digits. If the thing I'm trying to convert is 0, I just return the string "0". Otherwise, let's run through a little loop where I take that integer divided by 10, the remainder of that. What is that? Oh, that's the zeroth or the 1-order, the first order bit. And I'm going to index into digits to find that. And I'm going to add it on to a string that I'm [INAUDIBLE]. And I'll divide i by 10.

So this says, given an integer, I want to convert it to a string. I divide the integer by 10, take the remainder. That gives me the zeroth, or if you like, the ones element. I index into the string, and I record it. And then I add it to what I get by dividing i by 10 and doing the same thing. So I'll just walk down each of the digits, converting it into a string.

What I care about is the order of growth here. This is all constant. All I want to worry about here is, how many times do I go through the loop. And inside of the loop, this is just constant. It doesn't depend on the size of the integer.

So how many times do I go through the loop? Well, how many times can I divide i by 10? And that's log of i, right? So it's not i itself. It's not the size of the integer. It's the number of digits in the integer. And here's another nice example of log. I'll point you, again, right here. I'm reducing the size of the problem by a constant factor-- in this case, by 10-- each time-- nice characteristic of a logarithmic algorithm.

OK, we've got constant. We've got log. What about linear? We saw it last time, right? Something like searching a list in sequence was an example of something that was linear. In fact, most of the examples we saw last time were things with iterative loops.

So for example, fact, written intuitively-- factorial, right-- n times n minus 1 times n minus 2 all the way down to 1. I set product to 1. I go for a loop where i goes from 1 up to n minus 1, or just below n minus 1-- incrementally multiplying product by i and restoring that back away.

Again, we know that this loop here-- how many times do I go through it? I go through it n times. The cost inside the loop, there are three steps, changing i, I'm multiplying product times i, I'm storing that value back in product. And as we saw, that constant doesn't matter. This is linear. So n times around the loop, constant cost each time-- order n.

What about recursive? I could write fact recursively. I actually prefer it this way, right? If n is less than or equal to 1, return 1. Otherwise, multiply n by whatever I get by calling this on n minus 1. The cost inside the loop is just constant. I'm doing one subtraction, one multiplication. How many times I go through it? Again, n times, because I've got to go from n to n minus 1 to n minus 2. So again, this is linear.

Now, if you were to time it, you'd probably see a difference. My guess is-- I'm sure Professor Guttag will correct me if I get it wrong-- is that the factorial one probably takes a little more time, because you've got to set up the frame for the recursive call. But in terms of what we care about, they're the same. They're both linear. They're order n. And so interestingly, both iterative and recursive factorial have same order of growth.

Again, I want you to notice, what's the key here. Reducing the size of the problem by 1 is indicative, generally, of something that's going to have linear growth. I say in general. If it's a loop inside of a loop, as we saw, it might be a little bigger. But this is generally linear.

Constant, log, linear, log-linear-- that is, n log n-- we're going to see this next time. I'm certainly going to push it ahead. It invites you to come back on Wednesday and see this. It's actually something that's a really powerful algorithm. It's going to be really useful. We're going to look at something called merge sort, which is a very common sorting algorithm and has that property of being log-linear. So we'll come back to this next time.

How about polynomial? Well, we saw this last time as well. This commonly occurs when we have nested loops or where we have nested recursive function calls-- nested loop meaning I'm looping over some variable, and inside of there, I've got another loop. And what we saw is the outer loop, if it's a standard iterative thing, will be linear. But inside of the loop, I'm doing a linear amount of work each time. So it becomes n times n, so order n squared.

OK, exponential-- these are things-- sorry, yes, I did that right. I'm going to go back to it. Exponential-- these are things that we'd like to stay away from, but sometimes, we can't. And a common characteristic here is when we've got a recursive function where there's more than one recursive call inside the problem.

Remember Towers of Hanoi, that wonderful demonstration I did. I was tempted to bring it back, because it's always fun to get a little bit of applause when I do it. But I won't do it this time. But remember, we looked at that problem of solving the Towers of Hanoi. How do I move a stack of size n of different-sized disks from one peg to another where I can only move the top disk onto another one and I can't cover up a smaller disk?

Want to remind you, we saw, there was a wonderful recursive solution to that. It said, move a stack of size n minus 1 onto the spare peg. Move the bottom one. And then, move that stack over onto the thing you were headed towards, OK? What's the complexity of that?

Well, I'm going to show you a trick for figuring that out. It's called a recurrence relation for a very deliberate reason. But it'll give us a little, handy way to think about, what's the order of growth here.

So I'm going to let t sub n denote the time it takes to move a tower of size n. And I want to get an expression for, how much time is that going to take. What do I know? I know that's 2 times t to the n minus 1, right? I've got to move a stack of size 1 less onto the spare peg, and then, 1 to move that bottom thing over, and then, whatever it takes me to move a stack of size n minus 1 over to that peg. OK, so how does that help me?

Well, let's play the same game. What's t of n minus 1? Oh, that's 2t of n minus 2 plus 1. I'm just substituting in. I'm using exactly the same relationship here. All right, let's just do a little math on that. That's 4t to the n minus 2 plus 2 plus 1. And you're still going, OK, who cares. Well, let's do the same thing one more time. t of n minus 2-- that's 2t of n minus 3 plus 1. Oh, see the pattern? You can start to see it emerge here, right?

Each time I reduce this, I'm adding another power of 2, and I'm increasing the coefficient out front. And so, in fact, after k steps, I'll have 1 plus 2 plus 4 all the way up to 2 to the k minus 1 plus 2 to the k times t sub n minus k. Hopefully you can see it if you just look. This expression is capturing all of those up there. I'm just pulling it out each time.

When am I done? When this is size 0, when k is equal to n. And so that's when I get that expression. If this is going by too fast, just walk it through yourself later on. But I'm literally just using this expression to do the reduction until I see the pattern. All right, what's that?

Well, if your Course 18 major, you've seen it before. If you haven't, here's a nice, little trick. Let me let a equal that sum, 2 to the n minus 1 plus 2 to the n minus 2 all the way down to 1. Let me multiply both the left and the right side by 2. That gives me, 2a is equal to 2 to the n plus 2 to the n minus 1 all the way down to 2. I'm just taking each of the terms and multiplying them by 2.

Now subtract this from that. And then on the left side, you get a. And on the right side, you get that term. These all cancel out minus 1-- geometric series, cool. So that sum is just 2 to the n minus 1. And if I plug that back in there, ah, I've got my order of growth, exponential, 2 to the n.

OK, I was a math/physics undergrad. I like these kinds of things. But I wanted you to see how we can reason through it, because this is letting us see the growth. What I want you to pull away from this is, notice the characteristic. In Towers of Hanoi-- we're going to do another example in a second-- the characteristic was, at the recursive step, I had not one, but two recursive calls. And that is characteristic of something with exponential growth, which I just saw here, 2 to the n.

That, by the way, I'll remind you of the story of Towers of-- Towers of Hanoi, right? When the priests in that temple move the entire stack from one peg to another, we all reach nirvana, and the world ends. n is equal to 64 here. Go figure out what 2 to the 64 is. And if you're doing one move per second, which they will, I think we're certainly going to be here a while before the universe ends and we reach nirvana, probably several times over.

AUDIENCE: I thought we were already in nirvana.

ERIC GRIMSON: We are in nirvana, we're at MIT. You're right, John. But we're worrying about the rest of the world. So, OK, we'll keep moving on. Nirvana will be next week when they do the quiz, John. So we'll keep moving quickly. All right.

I want to show you one more example. It's a cool problem from math, but mostly to see that characteristic of exponential growth. And then we're going to pull all of this together. This is something called the power set. So if I have a set of things-- well, let's assume I have a set of integers-- with no repeats-- so 1 through n, 1, 2, 3, 4, for example-- I want to generate the collection of all possible subsets-- so subset with no elements, with one element, with two elements, with three amounts, all the way up to n elements.

So for example, if my set is 1 through 4, then the power set would be the empty set with no elements in it, all of the instances with one element, all of them with two, all of them with three, and all of them with four. I'd like to write code to generate this. It's actually handy problem in number theory or in set theory. By the way, the order doesn't matter. I could do it this way, but this would be a perfectly reasonable way of generating it as well. And I'm going to come back to that in a second as we think about solving this. The question is, how would I go about finding all of these.

I'm going to use-- well, we could stop and say, you could imagine writing a big iterative loop. You start with n, and you decide, do I include it or not. And then you go to n minus 1. Do I include it or not? And you could think about writing a big loop that would generate all of these-- actually, a bunch of nested loops. But there's a nice recursive solution. And I want to encourage you to think that way.

So here's the way I'm going to do it. What did we do when we said we want to think recursively? We say, let's assume we can solve a smaller size problem. If I want to generate the power set of all the integers from 1 to n, I'm going to assume that I can generate the power set of integers from 1 to n minus 1.

If I have that solution, then I can construct the solution to the bigger problem really easily. Wow. Well, all of the things that were in that solution to the smaller problem have to be part of the solution to the bigger problem. They're all subsets of 1 to n, because they're all subsets of 1 to n minus 1.

So I'm going to add all those in. And then I'm going to say, let's take each one of those and add n to each of those subsets. Because that gives me all the rest of the solutions. I've got all the ways to find solutions without n. I get all the ways to find solutions with n. That may sound like a lot of gobbledygook, but let me show you the example.

There is the power set of the empty set. It's just the empty set. Get the power set of 1, I include that, and I include a version of everything there with 1 added to it. There's the power set of 1. Now, given that, how do I get the power set of 2? Well, both of those are certainly things I want. And for each one of them, let me just add 2.

And if you look at that, right, that's the set of all ways of getting nothing, 1, 2, or both of them. And you get the idea. Now, having that solution, I can get the solution for 3, because all of those have to belong. And I simply add 3 to each one of those. Oh, that's cool, right?

All right, you don't have to be a math geek to admit it's cool. It is kind of cool. Because it says, gee, got a solution to the smaller problem. Generating the next piece is a natural step. And you can also see, the size of that set's doubling each time. Because you get to 4, I'm going to add everything in to all of those pieces-- really nice recursive description. Let's write some code.

So I'll also hand it out to you, but here's the code. And I'm going to walk through it carefully.

And then we're going to analyze it. But it's actually, for me, a beautiful piece of code. I did not write it, by the way, John did. But it's a beautiful piece of code.

I want to generate all the subsets with a power set of some list of elements. Here's how I'm going to do it. I'm going to set up some internal variable called res, OK? And then, what am I going to do? Actually, I don't know why I put res in there. I don't need it. But we'll come back to that.

If the list is empty, length of the list is 0, I'm going to just return that solution. And this is not a typo. What is that funky thing there? It is a list of one element, which is the empty list, which I need. Because the solution in this case is a set with nothing in it. So there is the thing I return in the base case.

Otherwise, what do I do? I take all the elements of the list except the last one, and I call it recursively. I generate all of the subsets of that. Perfect, so I'm going to call that smaller. I then take the last element, and I make a list of just the last element. And what did I say I need to do? I need all of these guys, plus I need all of them where I add that element in-- oh, nice.

I'll set up new as a variable here. And I'll loop over all of the elements from the smaller problem, where I basically add that list to that list. And I put it into new. That's simply taking all of the solutions of subsets of up to n minus 1 and creating a new set of subsets where n is included in every one of them. And now I take this, and I take that. I append them-- or concatenate them, rather. I should say "append them"-- concatenate them together and return them.

That's a crisp piece of code. And I'm sorry, John, I have no idea why I put res up there. I don't think I need that anywhere in this code. And I won't blame it on John. It was my recopying of the code.

AUDIENCE: [INAUDIBLE] .

ERIC GRIMSON: Sorry?

AUDIENCE: Maybe.

ERIC GRIMSON: Maybe, right. Look, I know I'm flaming at you. I get to do it. I'm tenured, as I've said multiple times in this course. That's a cool piece of code. Imagine trying to write it with a bunch of loops iterating over indices. Good luck. You can do it. Maybe it'll be on the quiz. Actually, no, it won't.

That's way too hard to ask.

But it's a cool piece of code, because I can look at it and say, what's the solution, solve the smaller problem, and then, given that, take every one of the things in that smaller problem, add that element into it, and put the two pieces together. Wonderful. OK, with that in mind, let's see if we can figure out the complexity of this.

Up here, that's constant. That's OK. Right there, I've got the recursive call. So I know, first of all, that this is going to call itself n times, right? Because each stage reduces the size of the problem by 1. So if I'm trying to get the power set of n, I'm going to have to do it to get n minus 1, and then, n minus 2. So I know the recursion of genSubsets() to genSubsets(). This is going to go around n times.

That's not so bad. But right down here, I've got to figure out, what's the cost of this, all right? This is constant. That's setting up as constant. That's constant. But there, I've got another loop. And the loop depends on how big smaller is. And "smaller's" a bad choice of term here, because it's going to grow on me. But let's think about it.

By the way, I'm assuming append is constant time, which, generally, it is. The time I need to solve this problem includes the time to solve the smaller problem. That recursive call, I know that's going to be linear. But I also need the time it takes to make the copy of all the things in that smaller version.

So how big is that? Oh, crud number two-- number of things in the power set grows as a factor of 2, right? If I've got something of, you know, 1 through 3, I've got all the things with nothing in it, all the things with one in it, all the things with two things in it, all the things with three things in it. That's 8. And each time around, I'm doubling the size of it.

So for a set of size k, there are 2 the k cases. And that says that this loop right here is going to be growing exponentially. Because I've got to go down that entire list to find all of the pieces. So what's the overall complexity? I'm going to play the same game.

Let's let t sub n capture the time it takes to solve a problem of size n. Just temporarily, I'm going to let s sub n denote the size of the solution for a problem of size n. How big is that thing, smaller? And what do I know?

The amount of time it takes me to solve the problem of size n is the amount of time it takes me to solve the slightly smaller problem-- that's the recursive call to genSubsets()-- plus the amount of time it takes me to run over that loop looking at everything in smaller and adding in a new version, plus some constant c, which is just the number of constant operations, the constant steps inside that loop, OK? And if I go back to it, t sub n is the cost here. t sub n minus 1 is the cost there. s sub n is the size of this. And then I've got, one, two, three, four, five constant steps. So c is probably 5 in this case.

So what can I say? There's the relationship. Because I know s of n minus 1 is 2 to the n minus 1. There are 2 to the n minus 1 elements inside of that.

How do I deal with this? Let's play the same game. What's t sub n minus 1? That's t of n minus 2 plus 2 to the n minus 2 plus c. And I could keep doing this. You can see what the pattern's going to look like. I'm going to have k times c constant steps. For each reduction, I'm going to get another power of 2. And I'm going to reduce this overall term, after k steps, to t the n minus k.

When am I done? When that's down to something of size 0. And there's the expression. And what you can see is what I wanted you to see, order n-- or sorry, order 2 to the n-- is exponential in the size of the problem.

What's the characteristic? Something that has a recursive call-- sorry, multiple recursive calls at each step-- is likely to lead to exponential. But that can also be buried inside of how I grow the size of the problem. And that was the case here. There's only one recursive call, but that loop grows in size each time around. So the complexity is exponential.

I'm going to pull this together. I said one of the things I'd like to start to recognize is, what are the characteristics of a choice in algorithm that leads to a particular complexity class. And you now have some of them.

If the code doesn't depend on the size of the problem, that's constant. And in fact, we've been using that as we look at pieces of the code. If we can reduce the problem-- I said, in this case-- by half each time, by some constant factor, from n, to n/2, to n/4, to n/8, that tends to be characteristic-- unless there's a hidden cost somewhere else-- of a logarithmic algorithm. These are really nice.

Simple things that reduce the size of the problem by 1 at each step-- an iterative call that goes from n, to n minus 1, and then to n minus 2, and then to n minus 3-- characteristic of linear algorithms. Log-linear we're going to see next time. Polynomial-- typically quadratic n squared when we have nested loops or nested recursive calls. I'm looping over something. Inside of there, I'm looping over something else on a size that depends on the size of the problem.

And then, we just saw this last one. Multiple recursive calls at each level tends to be characteristic of exponential. And as I said, we'd like to be as high up in this list as we can, because those are really nice algorithms to have.

Let me give you one more example of looking at this, and then we'll be done. Fibonacci-- standard problem, right? The nth Fibonacci number is the sum of the previous two Fibonacci numbers. This was the example we saw of multiplying rabbits, if you like. Here's an iterative version of Fibonacci, which says if n is 0, it's just 0. If it's 1 is just 1.

Otherwise, I'm going to set up, initially, the two previous Fibonacci numbers. And then I'm just going to run through a loop where I temporarily keep track of that number. I move the second previous one into the last previous one. I add those two. That becomes the second previous number. And I just keep running through that loop.

You can go run it. You see it does the right thing. What I want to look at is the complexity.

So that's constant. That's constant. That's linear, because the work inside the loop is constant, but I'm doing it n times. So this is nice. [INAUDIBLE] I should say, the bottom thing is constant. The overall algorithm, the worst case is just order n. Great.

What about the recursive version? For me, this is much nicer code. It's nice and clean. It says if n is equal to 0, fib is 0, 0. If n is equal to 1, fib of 1-- or the first and second Fibonacci numbers-- are 0 and 1. Otherwise, just return what I get by summing both of those pieces.

And you can probably already guess what the complexity is going to be here, right? Because I've now got two recursive calls inside of this call. So one way to think about it is, if I'm going to solve the problem up here, I've got to solve two versions of the problem below, which has got to solve two versions of the problem below. And in general, this is going to be exponential, 2 to the n.

Now you say, wait a minute. I was paying attention when this guy was yattering on a couple of weeks ago. Honest, I was. And in fact, what we saw was that fib isn't balanced in terms of how it goes, right? It's not that, on the right-hand side of the tree, I have to solve all of those portions, because the problem gets smaller. Does that change the complexity?

Well, the answer is, it changes the base, but it's actually still exponential. And if you want to go look this up, I'm sure you can find Wikipedia very quickly. This actually has a very cool exponential growth. It's the golden ratio to the nth power. And in fact, I encourage you to go look at it in the even more copious spare time you have. It's a very cool proof to see it. But the bottom line is, while we can do a little bit better than 2 to the n, it still grows exponentially with n.

So what do we have? We've got big O notation as a way of talking about comparing efficiency of algorithms. What I want you to see here is that you ought to be able to begin to reason about what's the cost of an algorithm by recognizing those common patterns. I keep saying it, but it's going to be really valuable to you.

And you should be able to therefore work the other direction. When you're given a new problem, how do I get this into a linear algorithm if I can? Log-linear, if I can, would be really great. But you know, if I can't, how do I stay away from exponential algorithms? And finally, what we're going to show later on is that, in fact, there are some problems that, as far as we know, are fundamentally exponential. And they're expensive to compute.

The very last thing is, you might have decided I was cheating in a different way. So I'm using a set of built-in Python functions. I'm not going to go through all of these. But this is just a list, for example, for lists, of what the complexity of those built-in functions are. And if you look through the list, they kind of make sense.

Indexing, you can go straight to that point. Computing the length, you compute it once, you've stored it. Comparison-- order n, because I've got to compare all the elements of the list. Similarly, to remove something from the list, I've got to find where it is in the list and remove it. Worst case, that's going to be order n. So you can see that these operations are typically linear in the size of a list. These are constant.

For dictionaries, remember, dictionaries were this nice thing. They weren't ordered. It gave me a power in terms of storing them. But as a consequence, some of the costs then go up.

For a list, indexing, going to a particular point, I just go to that spot and retrieve it. Indexing into a dictionary, I have to find that point in the dictionary that has the key and get the value back. So that's going to be linear, because I have to, in principle, walk all the way down it.

It's a slight misstatement, as we'll see later on. A dictionary actually uses a clever indexing scheme called a hash. But in the worst case, this is going to be linear. So you see a trade-off. For dictionaries, I get more power. I get more flexibility. But it comes as a cost.

And so these, basically, are what let me reason on top of the things I've been doing to figure out complexity. And next time, we'll do the last piece of this when we look at sorting. So we'll see you all on Wednesday.