PROFESSOR ERIC GRIMSON: Let's recap where we were. Last lecture, we talked about, or started to talk about, efficiency. Orders of growth. Complexity. And I'll remind you, we saw a set of algorithms, and part of my goal was to get you to begin to recognize characteristics of algorithms that map into a particular class.

So what did we see? We saw linear algorithms. Typical characterization, not all the time, but typical characterization, is an algorithm that reduces the size of a problem by one, or by some constant amount each time, is typically an example of a linear algorithm. And we saw a couple of examples of linear algorithms.

We also saw a logarithmic algorithm. and we like log algorithms, because they're really fast. A typical characteristic of a log algorithm is a pro-- or sorry, an algorithm where it reduces the size of the problem by a constant factor. Obviously-- and that's a bad way of saying it, I said constant the previous time-- in the linear case, it's subtract by certain amount. In the log case, it's divide by an amount. Cut the problem in half. Cut the problem in half again. And that's a typical characterization of a log algorithm.

We saw some quadratic algorithms, typically those are things with multiple nested loops, or iterative or recursive calls, where you're doing, say, a linear amount of time but you're doing it a linear number of times, and so it becomes quadratic, and you'll see other polynomial kinds of algorithms.

And finally, we saw an example of an exponential algorithm, those Towers of Hanoi. We don't like exponential algorithms, or at least you shouldn't like them, because they blow up quickly. And we saw some examples of that. And unfortunately, some problems are inherently exponential, you're sort of stuck with that, and then you just have to try be as clever as you can.

OK. At the end of the lecture last time, I also showed you an example of binary search. And I want to redo that in a little more detail today, because I felt like I did that a little more quickly than I wanted to, so, if you really got binary search, fall asleep for about ten minutes, just don't snore, your neighbors may not appreciate it, but we're going to go over it again, because it's a problem and an idea that we're going to come back to, and I really want to make sure that I do this in a way that makes real good sense you.

Again. Basic premise of binary search, or at least we set it up was, imagine I have a sorted list of elements. We get, in a second, to how we're going to get them sorted, and I want to know, is a particular element in that list.. And the basic idea of binary search is to start with the full range of the list, pick the midpoint, and test that point. If

it's the thing I'm looking for, I'm golden. If not, because the list is sorted, I can use the difference between what I'm looking for and that midpoint to decide, should I look in the top half of the list, or the bottom half of the list? And I keep chopping it down.

And I want to show you a little bit more detail of that, so let's create a simple little list here. All right? I don't care what's in there, but just assume that's my list. And just to remind you, on your handout, and there it is on the screen, I'm going to bring it back up, there's the little binary search algorithm. We're going to call search, which just calls binary search.

And you can look at it, and let's in fact take a look at it to see what it does. We're going to call binary search, it's going to take the list to search and the element, but it's also going to say, here's the first part of the list, and there's the last part of the list, and what does it do inside that code? Well, it checks to see, is it bigger than two? Are there more than two elements there? If there are less than two elements there, I just check one or both of those to see if I'm looking for the right thing.

Otherwise, what does that code say to do? It says find the midpoint, which says, take the start, which is pointing to that place right there, take last minus first, divide it by 2, and add it to start. And that basically, somewhere about here, gives me the midpoint.

Now I look at that element. Is it the thing I'm looking for? If I'm really lucky, it is. If not, I look at the value of that point here and the thing I'm looking for. And for sake of argument, let's assume that the thing I'm looking for is smaller than the value here.

Here's what I do. I change-- oops! Let me do that this way-- I change last to here, and keep first there, and I throw away all of that. All right? That's just the those-- let me use my pointer-- that's just these two lines here. I checked the value, and in one case, I'm changing the last to be mid minus 1, which is the case I'm in here, and I just call again. All right?

I'm going to call exactly the same thing. Now, first is pointing here, last is pointing there, again, I check to see, are there more than two things left? There are, in this case. So what do I do? I find the midpoint by taking last minus first, divide by 2, and add to start. Just for sake of argument, we'll assume it's about there, and I do the same thing.

Is this value what I'm looking for? Again, for sake of argument, let's assume it's not. Let's assume, for sake of argument, the thing I'm looking for is bigger than this. In that case, I'm going to throw away all of this, I'm going to hit that bottom line of that code.

Ah. What does that do? It changes the call. So in this case, first now points there, last points there. And I cut around.

And again, notice what I've done. I've thrown away most of the array-- most of the list, I shouldn't say array-- most of the list. All right? So it cuts it down quickly as we go along.

OK. That's the basic idea of binary search. And let's just run a couple of examples to remind you of what happens if we do this.

So if I call, let's [UNINTELLIGIBLE], let's set up s to be, I don't know, some big long list. OK. And I'm going to look to see, is a particular element inside of that list, and again, I'll remind you, that's just giving me the integers from zero up to 9999 something or other.

If I look for, say, minus 1, you might go, gee, wait a minute, if I was just doing linear search, I would've known right away that minus one wasn't in this list, because it's sorted and it's smaller than the first elements. So this looks like it's doing a little bit of extra work, but you can see, if you look at that, how it cuts it down at each stage. And I'll remind you, what I'm printing out there is, first and last, with the range I'm looking over, and then just how many times the iteration called.

So in this case, it just keeps chopping down from the back end, which kind of makes sense, all right? But in a fixed number, in fact, twenty-three calls, it gets down to the point of being able to say whether it's there.

Let's go the other direction. And yes, I guess I'd better say s not 2, or we're going to get an error here. Again, in twenty-three checks. In this case, it's cutting up from the bottom end, which makes sense because the thing I'm looking for is always bigger than the midpoint, and then, I don't know, let's pick something in between. Somebody want-- ah, I keep doing that-- somebody like to give me a number? I know you'd like to give me other things, other expression, somebody give me a number. Anybody? No? Sorry. Thank you. Good number.

OK, walks in very quickly. OK? And if you just look at the numbers, you can see how it cuts in from one side and then the other side as it keeps narrowing that range, until it gets down to the place where there are at most two things left, and then it just has to check those two to say whether it's there or not.

Think about this compared to a linear search. All right? A linear search, I start at the beginning of the list and walk all the way through it. All right, if I'm lucky and it's at the low end, I'll find it pretty quickly. If it's not, if it's at the far end, I've got to go forever, and you saw that last time where this thing paused for a little while while it actually searched a list this big.

OK. So, what do I want you to take away from this? This idea of binary search is going to be a really powerful tool.

And it has this property, again, of chopping things into pieces.

So in fact, what does that suggest about the order of growth here? What is the complexity of this? Yeah. Logarithmic. Why?

STUDENT: [UNINTELLIGIBLE]

PROFESSOR ERIC GRIMSON: Yeah. Thank you. I mean, I know I sort of said it to you, but you're right. It's logarithmic, right? It's got that property of, it cuts things in half.

Here's another way to think about why is this log. Actually, let me ask a slightly different question. How do we know this always stops? I mean, I ran three trials here, and it did. But how would I reason about, does this always stop? Well let's see. Where's the end test on this thing? The end test-- and I've got the wrong glasses on-- but it's up here, where I'm looking to see, is last minus first less than or equal to 2?

OK. So, soon as I get down to a list that has no more than two elements in it, I'm done. Notice that. It's a less than or equal to. What if I just tested to see if it was only, say, one? There was one element in there. Would that have worked?

I think it depends on whether the list is odd or even in length. Actually, that's probably not true. With one, it'll probably always get it down there, but if I've made it just equal to two, I might have lost.

So first of all, I've got to be careful about the end test. But the second thing is, OK, if it stops whenever this is less than two, am I convinced that this will always halt? And the answer is sure. Because what do I do? At each stage, no matter which branch, here or here, I take, I'm cutting down the length of the list that I'm searching in half. All right?

So if I start off with a list of length n, how many times can I divide it by 2, until I get to something no more than two left? Log times, right.? Exactly as the gentleman said.

Oh, I'm sorry. You're patiently waiting for me to reward. Or actually, maybe you're not. Thank you.

OK. So this is, in fact, log. Now, having said that, I actually snuck something by you. And I want to spend a couple of minutes again reinforcing that. So if we look at that code, and we were little more careful about this, what did we say to do? We said look an-- sorry. Count the number of primitive operations in each step.

OK. So if I look at this code, first of all I'm calling search, it just has one call, so looks like search is constant, except I don't know what happens inside of b search. So I've got to look at b search.

So let's see. The first line, that print thing, is obviously constant, right? Just take it as a constant amount of operations But. let's look at the next one here, or is that second line?

OK. If last minus first is greater than or equal to 2-- sorry, less than 2, then either look at this thing or look at that thing. And that's where I said we've got to be careful. That's accessing an element of a list. We have to make sure that, in fact, that operation is not linear.

So let me expand on that very slightly, and again, we did this last time but I want to do one more time. I have to be careful about how I'm actually implementing a list.

So, for example: in this case, my list is a bunch of integers. And one of the things I could take advantage of, is I'm only going to need a finite amount of space to represent an integer.

So, for example, if I want to allow for some fairly large range of integers, I might say, I need four memory cells in a row to represent an integer. All right, if it's a zero, it's going to be a whole bunch of ones-- of zeroes, so one, it may be a whole bunch of zeroes in the first three and then a one at the end of this thing, but one of the way to think about this list in memory, is that I can decide in constant time how to find the i'th element of a list.

So in particular, here's where the zero-th element of the list starts, there's where the first element starts, here's where the third element starts, these are just memory cells in a row, and to find the zero-th element, if start is pointing to that memory cell, it's just at start.

To find the first element, because I know I need four memory cells to represent an integer, it's at start plus 4. To get to the second element, I know that that's-- you get the idea-- at the start plus 2 times 4, and to get to the k'th element, I know that I want to take whatever the start is which points to that place in memory, take care, multiply by 4, and that tells me exactly where to go to find that location.

This may sound like a nuance, but it's important. Why? Because that's a constant access, right? To get any location in memory, to get to any value of the list, I simply have to say which element do I want to get, I know that these things are stored in a particular size, multiply that index by 4, add it to start, and then it's in a constant amount of time I can go to that location and get out the cell.

OK. That works nicely if I know that I have things stored in constant size. But what if I have a list of lists? What if I have a homogeneous list, a list of integers and strings and floats and lists and lists of lists and lists of lists of lists and all that sort of cool stuff? In that case, I've got to be a lot more careful.

So in this case, one of the standard ways to do this, is to use what's called a linked list. And I'm going to do it in the following way.

Start again, we'll point to the beginning of the list. But now, because my elements are going to take different amounts of memory, I'm going to do the following thing. In the first spot, I'm going to store something that says, here's how far you have to jump to get to the next element.

And then, I'm going to use the next sequence of things to represent the first element, or

the zero-th element, if you like. In this case I might need five. And then in the next spot, I'm going to say how far you have to jump to get to the next element. All right, followed by whatever I need to represent it, which might only be a blank one.

And in the next spot, maybe I've got a really long list, and I'm going to say how to jump to get to the next element.

All right, this is actually kind of nice. This lets me have a way of representing things that could be arbitrary in size. And some of these things could be huge, if they're themselves lists.

Here's the problem. How do I get to the nth-- er, the k'th element in the list, in this case? Well I have to go to the zero-th element, and say OK, gee, to get to the next element, I've got to jump this here. And to get to the next element, I've got to jump to here, and to get to the next element, I've got to jump to here, until I get there.

And so, I get some power. I get the ability to store arbitrary things, but what just happened to my complexity? How long does it take me to find the

k'th element? Linear. Because I've got to walk my way down it. OK? So in this case, you have linear access. Oh fudge knuckle. Right? If that was the case in that code, then my complexity is no longer log, because I need linear access for each time I've got to go to the list, and it's going to be much worse than that.

All right. Now. Some programming languages, primarily Lisp, actually store lists these ways. You might say, why? Well it turns out there's some trade-offs to it. It has some advantages in terms of power of storing things, it has some disadvantages, primarily in terms of access time.

Fortunately for you, Python decided, or the investors of Python decided, to store this a different way. And the different way is to say, look, if I redraw this, it's called a box and pointer diagram, what we really have for each element is two things. And I've actually just reversed the order here.

We have a pointer to the location in memory that contains the actual value, which itself might be a bunch of pointers, and we have a pointer to the actual-- sorry, a pointer the value and we have a pointer to the next element in the list. All right?

And one of the things we could do if we look at that is, we say, gee, we could reorganize this in a pretty straightforward way. In particular, why don't we just take all of the first cells and stick them together? Where now, my list is a list of pointers, it's not a set of values but it's actually a pointer off to some other piece of memory that contains the value.

Why is this nice? Well this is exactly like this. All right? It's now something that I can search in constant time. And that's what's going to allow me to keep this thing as being log.

OK. With that in mind, let's go back to where we were. And where were we? We started off talking about binary search, and I suggested that this was a log algorithm, which it is, which is really kind of nice.

Let's pull together what this algorithm actually does. If I generalize binary search, here's what I'm going to stake that this thing does.

It says one: pick the midpoint.

Two: check to see if this is the answer, if this is the thing I'm looking for.

And then, three: if not, reduce to a smaller problem, and repeat.

OK, you're going, yeah, come on, that makes obvious sense. And it does. But I want you to keep that template in mind, because we're going to come back to that. It's an example of a very common tool that's going to be really useful to us, not just for doing search, but for doing a whole range of problems. That is, in essence, the template the describes a log style algorithm. And we're going to come back to it.

OK. With that in mind though, didn't I cheat? I remind you, I know you're not really listening to me, but that's OK. I reminded you at the beginning of the lecture, I said, let's assume we have a sorted list, and then let's go search it. Where in the world

did that sorted list come from? What if I just get a list of elements, what do I do? Well let's see. My fall back is, I could just do linear search, walk down the list one at a time, just comparing those things. OK. So that's sort of my base. But what if I wanted, you know, how do I want to get to that sorted list? All right?

Now. One of the questions, before we get to doing the sorting, is even to ask, what should I do in a search case like that? All right, so in particular, does it make sense, if I'm given an unsorted list, to first sort it, and then search it? Or should I just use the basically linear case? All right?

So, here's the question. Should we sort before we search?

OK. So let's see, if I'm going to do this, how fast could we sort a list? Can we sort a list in sublinear time? Sublinear meaning, something like log less than linear time? What do you think? It's possible? Any thoughts? Don't you hate professors who stand here waiting for you to answer, even when they have candy?

Does it make sense to think we could do this in less than linear time? You know, it takes a little bit of thinking. What would it mean-- [UNINTELLIGIBLE PHRASE] do I see a hand, way at the back, yes please? Thank you. Man, you're going to really make me work here, I have no idea if I can get it that far, ah, your friend will help you out. Thank you.

The gentleman has it exactly right. How could I possibly do it in sublinear time, I've got to look at least every element once. And that's the kind of instinct I'd like you to get into thinking about. So the answer here is no.

OK. Can we sort it in linear time? Hmmm. That one's not so obvious. So let's think about this for a second. To sort a list in linear time, would say, I have to look at each element in the list at most a constant number of times. It doesn't have to be just once, right? It could be two or three times.

Hmm. Well, wait a minute. If I want to sort a list, I'll take one element, I've got to look at probably a lot of the other elements in the list in order to decide where it goes. And that suggests it's going to depend on how long the list is. All right, so that's a weak argument, but in fact, it's a way of suggesting, probably not.

All right. So how fast could I sort a list? How fast can we sort it? And we're going to come back to this, probably next time if I time this right, but the answer is, we can do it in n log n time. We're going to come back to that. All right? And I'm going to say-- sort of set that stage here, so that-- It turns out that that's probably about the best we can do, or again ends at the length of the list.

OK, so that's still comes back to my question. If I want to search a list, should I sort it first and then search it? Hmmm. OK, so let's do the comparison. I'm just going to take an unsorted list and search it, I could do it in linear time, right? One at a time. Walk down the elements until I find it. That would be order n. On the other hand, if I want to sort it first, OK, if I want to do sort and search, I want to sort it, it's going to take n log n time to sort it, and having done that, then I can search it in log n time.

Ah. So which one's better? Yeah. Ah-ha. Thank you. Hold on to that thought for second, I'm going to come back to it. That does not assume I'm running a search it wants, which one's better?

The unsorted, and you have exactly the point I want to get to-- how come all the guys, sorry, all the people answering questions are way, way up in the back? Wow. that's a Tim Wakefield pitch right there, all right. Thank you.

He has it exactly right. OK? Is this smaller than that? No. Now that's a slight lie. Sorry, a slight misstatement, OK? I could run for office, couldn't I, if I can do that kind of talk. It's a slight misstatement in the sense that these should really be orders of growth. There are some constants in there, it depends on the size, but in general, n log n has to be bigger than n.

So, as the gentleman back there said, if I'm searching it once, just use the linear search. On the other hand, am I likely to only search a list once? Probably not. There are going to be multiple elements I'm going to be looking for, so that suggests that in fact, I want to amortize the cost.

And what does that say? It says, let's assume I want to do k searches of a list. OK. In the linear case, meaning in the unsorted case, what's the complexity of this? k times n, right? Order n to do the search, and I've got to do it k times, so this would be k times n.

In the [GARBLED PHRASE] sort and search case, what's my cost? I've got to sort it, and we said, and we'll come back to that next time, that I can do the sort in n log n, and then what's the search in this case? Let's log n to do one search, I want to do k of them, that's k log n, ah-ha!

Now I'm in better shape, right? Especially for really large n or for a lot of k, because now in general, this is going to be smaller than that. So this is a place where the amortized cost actually helps me out. And as the gentleman at the back said, the question he asked is right, it depends on what I'm trying to do. So when I do the analysis, I want to think about what am I doing here, am I capturing all the pieces of it? Here, the two variables that matter are what's the length of the list, and how many times I'm going to search it? So in this case, this one wins, whereas in this case, that one wins.

OK. Having said that, let's look at doing some sorts. And I'm going to start with a couple of dumb sorting mechanisms. Actually, that's the wrong way saying it, they're simply brain-damaged, they're not dumb, OK? They are computationally challenged, meaning, at the time they were invented, they were perfectly good sorting algorithms, there are better ones, we're going to see a much better one next time around, but this is a good way to just start thinking about how to do the algorithm, or how to do the sort. Blah, try again. How to do this sort.

So the first one I want to talk about it's what's called selection sort. And it's on your handout, and I'm going to bring the code up here, you can see it, it's called cell sort, just for selection sort. And let's take a look at what this does. OK. And in fact I think the easy way to look at what this does-- boy. My jokes are that bad. Wow-- All right.

I think the easiest way to look at what this does, is let's take a really simple example-- I want to make sure I put the right things out-- I've got a simple little list of values there. And if I look at this code, I'm going to run over a loop, you can see that there, i is going to go from zero up to the length minus 1, and I'm going to keep track of a

couple of variables. Min index, I think I called it min val.

OK. Let's simulate the code. Let's see what it's doing here. All right, so we start off. Initially i-- ah, let me do it this way, i is going to point there, and I want to make sure I do it right, OK-- and min index is going to point to the value of i, which is there, and min value is initially going to have the value 1. So we're simply catting a hold of what's the first value we've got there.

And then what do we do? We start with j pointing here, and we can see what this loop's going to do, right? j is just going to move up.

So it's going to look at the rest of the list, walking along, and what does it do? It says, right. If j is-- well it says until j is at the less than the length of l-- it says, if min value is bigger than the thing I'm looking at, I'm going to do something, all right? So let's walk this. Min value is 1,. Is 1 bigger than 8? No. I move j up. Is 1 bigger than 3? No. 1 bigger than 6? No. 1 bigger than 4? No.

I get to the end of the loop, and I actually do a little bit of wasted motion there. And the little bit of wasted motion is, I take the value at i, store it away temporarily, take the value where min index is pointing to, put it back in there, and then swap it around.

OK. Having done that, let's move i up to here. i is now pointing at that thing. Go through the second round of the loop.

OK. What does that say? I'm going to change min index to also point there n value is 8, j starts off here, and I say, OK, is the thing I'm looking at here smaller than that? Yes.

Ah-ha. What does that say to do? It says, gee, make min index point to there, min value be 3. Change j. Is 6 bigger than 3? Yes. Is 4 bigger than 3? Yes. Get to the end.

And when I get to the end, what do I do? Well, you see, I say, take temp, and store away what's here, all right? Which is that value, and then take what min index is pointing to, and stick it in there, and finally, replace that value.

OK. Aren't you glad I'm not a computer? Slow as hell.

What's this thing doing? It's walking along the list, looking for the smallest thing in the back end of the list, keeping track of where it came from, and swapping it with that spot in the list. All right?

So in the first case, I didn't have to do any swaps because 1 was the smallest thing. In the second case, I found in the next smallest element and moved here, taking what was there and moving it on, in this case I would swap the 4 and the 8, and in next case I wouldn't have to do anything.

Let's check it out. I've written a little bit of a test script here, so if we test cell sort, and I've written this so that it's going to print out what the list is at the end of each round, OK.

Ah-ha. Notice what-- where am I, here-- notice what happened in this case. At the end of the first round, I've got the smallest element at the front. At the end of the second round, I've got the smallest two elements at the front, in fact I got all of them sorted out. And it actually runs through the loop multiple times, making sure that it's in the right form.

Let's take another example. OK. Smallest element at the front. Smallest two elements at the front. Smallest three elements at the front. Smallest four elements at the front, you get the idea. Smallest five elements at the front.

So this is a nice little search-- sorry, a nice little sort algorithm . And in fact, it's relying on something that we're going to come back to, called the loop invariant. Actually, let me put it on this board so you can see it.

The loop invariant what does the loop invariant mean? It says, here is a property that is true of this structure every time through the loop. In the loop invariant here is the following: the list is split, into a prefix or a first part, and a suffix, the prefix is sorted, the suffix is not, and basically, the loop starts off with the prefix being nothing and it keeps increasing the size of the prefix by 1 until it gets through the entire list, at which point there's nothing in the suffix and entire prefix is sorted.

OK? So you can see that, it's just walking through it, and in fact if I look at a couple of another-- another couple of examples, it's been a long day, again, you can see that property. You'll also notice that this thing goes through the entire list, even if the list is sorted before it gets partway through.

And that you might look at, for example, that first example, and say, man by this stage it was already sorted, yet it had to go through and check that the third element was in the right place, and then the fourth and then the fifth and then the six.

OK. What order of growth? What's complexity of this? I've got to get rid of this candy. Anybody help me out? What's the complexity of this? Sorry, somebody at the back.

n squared. Yeah, where n is what? Yeah, and I can't even see who's saying that. Thank you. Sorry, I've got the wrong glasses on, but you're absolutely right, and in case the rest of you didn't hear it, n squared.

How do I figure that out? Well I'm looping down the list, right? I'm walking down the list. So it's certainly at least linear in the length of the list. For each starting point, what do I do? I look at the rest of the list to decide what's the element to swap into the next place.

Now, you might say, well, wait a minute. As I keep moving down, that part gets smaller, it's not always the initial length of the list, and you're right. But if you do the sums, or if you want to think of it this way, if you think about this more generally, it's always on average at least the length of the list. So I've got to do n things n times. So it's quadratic, in terms of that sort.

OK. That's one way to do this sort. Let's do another one.

The second one we're going to do is called bubble sort. All right? And bubble sort is also on your handout. And you want to take the first of these, let me-- sorry, for a second let me uncomment that, and let me comment this out--

All right, you can see the code for bubble sort there. Let's just look at it for a second, then we'll try some examples, and then we'll figure out what it's actually doing. So bubble sort, which is right up here. What's it going to do? It's going to let j run over the length of the list, all right, so it's going to start at some point to move down, and then it's going to let i run over range, that's just one smaller, and what's it doing there?

It's looking at successive pairs, right? It's looking at the i'th and the i plus first element, and it's saying, gee, if the i'th element is bigger than the i'th plus first element, what's the next set of three things doing? Just swapping them, right? I temporarily hold on to what's in the i'th element so I can move the i plus first one in, and then replace that with the i'th element.

OK. What's this thing doing then, in terms of sorting? At the end of the first pass, what could I say about the result of this thing? What's the last element in the list look like? I hate professors who do this.

Well, let's try it. Let's try a little test. OK? Test bubble sort-- especially if I could type-- let's run it on the first list. OK, let's try it on another one. Oops sorry. Ah, I didn't want to do it this time, I forgot to do the following, bear with me. I gave away my punchline. Let's try it again. Test bubble sort.

OK, there's the first run, I'm going to take a different list. Can you see a pattern there? Yeah.

STUDENT: The last cell in the list is always going to [INAUDIBLE]

PROFESSOR ERIC GRIMSON: Yeah. Why? You're right, but why?

STUDENT: [UNINTELLIGIBLE PHRASE]

PROFESSOR ERIC GRIMSON: Exactly right. Thank you. The observation is, thank you, on the first pass through, the last element is the biggest thing in the list. On the next pass through, the next largest element is at the second point in the list.

OK? Because what am I doing? It's called bubble sort because it's literally bubbling along, right? I'm walking along the list once, taking two things, and saying, make sure the biggest one is next. So wherever the largest element started out in the list, by the time I get through it, it's at the end. And then I go back and I start again, and I do the same thing.

OK. The next largest element has to end up in the second last spot. Et cetera. All right, so it's called bubble sort because it does this bubbling up until it gets there.

Now. What's the order of growth here? What's the complexity? I haven't talked to the side of the room in a while, actually I have. This gentleman has helped me out. Somebody else help me out. What's the complexity here? I must have the wrong glasses on to see a hand. No help.

Log? Linear? Exponential? Quadratic? Yeah. Log. It's a good think, but why do you think it's log? Ah-ha. It's not a bad instinct, the length is getting shorter each time, but what's one of the characteristics of a log algorithm? It drops in half each time. So this isn't-- OK. And you're also close. It's going to be linear, but how many times do I go through this? All right, I've got to do one pass to bubble the last element to the end. I've got to do another pass to bubble the second last element to the end. I've got to do another pass. Huh. Sounds like a linear number of times I've got to do-- oh fudge knuckle. A linear number of things, quadratic. Right?

OK. So this is again an example, this was quadratic, and this one was quadratic. And I have this, to write it out, this is order the length of the list squared, OK? Just to make it clear what we're actually measuring there. All

right. Could we do better? Sure. And in fact, next time we're going to show you that n log n algorithm, but even with bubble sort, we can do better. In a particular, if I look at those traces, I can certainly see cases where, man, I already had the list sorted much earlier on, and yet I kept going back to see if there was anything else to bubble up.

How would I keep track of that? Could I take advantage of that? Sure. Why don't I just keep track on each pass through the algorithm whether I have done any swaps? All right? Because if I don't do any swaps on a pass through the algorithm, then it says everything's in the right order.

And so, in fact, the version that I commented out-- which is also in your handout and I'm now going to uncomment, let's get that one out, get rid of this one-- notice the only change. I'm going to keep track of a little variable called swap, it's initially true, and as long as it's true, I'm going to keep going, but inside of the loop I'm going to set it to false, and only if I do a swap will I set it to true.

This says, if I go through an entire pass through the list and nothing gets changed, I'm done. And in fact if I do

that, and try test bubble sort, well, in the first case, looks the same. Ah. On the second case, I spot it right away. On the third case, it takes me the same amount of time. And the fourth case, when I set it up, I'm done.

OK. So what's the lesson here? I can be a little more careful about keeping track of what goes on inside of that loop. If I don't have any more work to do, let me just stop.

All right. Nonetheless, even with this change, what's the order growth for bubble sort? Still quadratic, right? I'm looking for the worst case behavior, it's still quadratic, it's quadratic in the length of the list, so I'm sort of stuck with that.

Now. Let me ask you one last question, and then we'll wrap this up. Which of these algorithms is better? Insertion sort or bubble sort?

STUDENT: Bubble.

PROFESSOR ERIC GRIMSON: Bubble. Bubble bubble toil and trouble. Who said bubble? Why?

STUDENT: Well, the first one was too inefficient [UNINTELLIGIBLE] store and compare each one, so [UNINTELLIGIBLE]

PROFESSOR ERIC GRIMSON: It's not a bad instinct. Right. So it-- so, your argument is, bubble is better because it's is essentially not doing all these extra comparisons. Another way of saying it is, I can do this stop when I don't need to. All right?

OK. Anybody have an opposing opinion? Wow, this sounds like a presidential debate. Sorry, I should reward you. Thank you for that statement. Anybody have an opposing opinion? Everybody's answering these things and sitting way up at the back. Nice catch. Yeah.

STUDENT: [INAUDIBLE]

PROFESSOR ERIC GRIMSON: I don't think so, right? I think selection sort, I still have to go through multiple times, it was still quadratic, OK, but I think you're heading towards a direction I want to get at, so let me prime this a little bit. How many swaps do I do in general in bubble sort, compared to selection source?

God bless. Oh, sorry, that wasn't a sneeze, it was a two? How many swaps do I do in bubble sort? A lot. Right. Potentially a lot because I'm constantly doing that, that says I'm running that inner loop a whole bunch of times.

How many swaps do I do in selection sort? Once each time. Right? I only do one swap potentially, it-- though not one potentially, each time at the end of the loop I do a swap.

So this actually suggests again, the orders of growth are the same, but probably selection sort is a more efficient algorithm, because I'm not doing that constant amount of work every time around. And in fact, if you go look up, you won't see bubble sort used very much. Most-- I shouldn't say most, many computer scientists don't think it should be taught, because it's just so inefficient. I disagree, because it's a clever idea, but it's still something that we have to keep track of.

All right. We haven't gotten to our n log n algorithm, we're going to do that next time, but I want to set the stage here by pulling out one last piece.

OK. Could we do better in terms of sorting? Again, remember what our goal was. If we could do sort, then we saw, if we amortized the cost, that searching is a lot more efficient if we're searching a sorted list.

How could we do better? Let me set the stage. I already said, back here, when I used this board, that this idea was really important. And that's because that is a version of a divide and conquer algorithm.

OK. Binary search is perhaps the simplest of the divide and conquer algorithms, and what does that mean? It says, in order to solve a problem, cut it down to a smaller problem and try and solve that one.

So to just preface what we're going to do next time, what would happen if I wanted to do sort, and rather than in sorting the entire list at once, I broke it into pieces, and sorted the pieces, and then just figured out a very efficient way to bring those two pieces and merge them back together again? Where those pieces, I would do the same thing with, I would divide them up into smaller chunks, and sort those. Is that going to give me a more efficient algorithm?

And if you come back on Thursday, we'll answer that question.