PROFESSOR: So, for the last two lectures we've been talking about analyzing algorithms, complexity, orders of growth. How do we estimate the cost of an algorithm as the size of the input grows? And as I've said several times, I'll say at least once more, how do we also turn it the other direction? How do we use thoughts about choices of pieces of algorithm in terms of implications on the cost it's going to take us to compute?

We saw last time a set of examples-- constant algorithms, linear algorithms, logarithmic algorithms, linear algorithms, quadratic algorithms, exponential algorithms. Today, what I'm going to do is fill in one more piece, a log linear algorithm-- something that's really a nice kind of algorithm to have-- and use it to talk about one last class of algorithms that are really valuable, and those are searching and sorting algorithms.

So a search algorithm. Kind of an obvious statement. You use them all the time when you go to Google or Bing or whatever your favorite search mechanism on the web is. It's just a way to find an item or a group of items from a collection. If you think about it, that collection could be either implicit or explicit. So way back at the beginning of the term, we saw an example of a search algorithm when you were looking for square roots. And we saw simple things like exhaustive enumeration. We'd go through all the possibilities. We saw our first version of bisection search there, where you would do approximations. Newton-Raphson-- these are all examples of a search algorithm where the collection is implicit. So all the numbers between some point that some other point.

More common is a search algorithm where the collection is explicit. I don't know. For example, I've got all the data records of students and I want to know how do I find a particular student, so I can record that A plus that everybody in this room is going to get next Tuesday on that exam? That's not a promise. Sorry. But we'll work on it. So could do it implicit, could do it explicit. Today I want to focus on doing search explicitly. And it could be on different kinds of collections, but I'm going to focus-- just as an example-- on search over lists. And to make it a little easier, let's just do search over lists of numbers. But it could obviously be other kinds of elements.

Now you've already seen some of this, right? We did search where we said, we can do linear search. Brute force. Just walk down the list looking at everything till we either find the thing we're looking for or we get to the end of the list. Sometimes also called British Museum algorithm or exhaustive enumeration. I go through everything in the list. Nice news is, the list doesn't have to be sorted. It could be just in arbitrary order. What we saw is that the expected-- sorry, not expected. The worst case behavior is linear. In the worst case, the element's not in the list. I got to look at everything. So it's going to be linear in terms of complexity.

And then we looked at bisection search, where we said the list needs to be sorted. But if it is, we can actually be much more efficient because we can take advantage of the sorting to cut down the size of the problem. And I'll remind you about both of those. There was our simple little linear search. Right? Set a flag that says, I haven't yet found it. And then just loop over the indices into the list. I could have also just looped directly over the list itself, checking to see if the ith member of the list is the thing I'm looking for. If it is, change the flag to true so that when I come out of all of this I'll return the flag-- either false because it was set that way initially or true because I found it.

And of course what we knew is we have to look at everything to see if it's there or not. I could speed this up by just returning true at this point.

While that would improve the average case, doesn't improve the worst case. And that's the thing we usually are concerned about, because in the worst case I've got to go through everything. And just to remind you, we said this is order length of the list. To go around this part-- the loop right here-- and inside the loop, it's constant work. I'm doing the same number of things each time. That's order n times order 1. And by our rules, that's just order n. So it's linear in the size of the problem.

OK. We said we could do it on sorted lists. But just again, we'll walk down the list. Again, here I could loop over everything in the list, checking to see if it's the thing I want. Return true. And if I ever get to a point where the element of the list is bigger than the thing I'm looking for, I know it can't be in the rest of the list because all the things to the right are bigger yet. I could just Return false and drop out. In terms of average behavior, this is better because it's going to stop as soon as it gets to a point where it can rule everything else out. But in terms of complexity, it's still order n. Because I still on average have-- not average. In the worst case, I'm still going to be looking n times through the loop before I get to a point where I can decide to bail out of it. So order n.

And then finally-- last piece of recap-- bisection search. Repeat again. The idea here is, take the midpoint of the list. Look at that element. If it's the thing I'm looking for, great. I just won the lottery. If it isn't, decide is the thing I'm looking for bigger or less than that middle point. If it's bigger than that, I only use the upper half of the list. If it's less than that, I only use the lower half of the list. And the characteristic here was, at each step, I'm reducing the size of the problem in half. I'm throwing away half of the remaining list at each step.

And I'll just remind you of that code. I know it's a lot here, but just to remind you. It said, down here if I've got an empty list, it can't be there. I'm going to Return false. Otherwise call this little helper function with the list, the thing for which I'm searching, and the beginning and end point indices into the list. Initially the start and the very end. And this code up

here basically says, if those two numbers are the same I'm down to a list of one. Just check to see if it's the thing I'm looking for. Otherwise, pick something halfway in between. And ignore this case for the moment. Basically then check to see, is the thing at that point bigger than e? In which case, I'm in general going to call this only with from the low point to the midpoint. Otherwise I'm going to call this with the midpoint to high. And that was just this idea of, keep cutting down in half the size of the list.

Last piece of the recap-- the thing we wanted you to see here-- is there are the two recursive calls. I'm only going to do one because I'm making a decision. At each step, I'm cutting down the problem by half. And that says the number of steps, the number of times I'm going to iterate through here, will be log in the length of the list. And if that still doesn't make sense to you, it says, I need to know when 1 over 2 to the k-- where k is the number of steps-- is equal to 1. Because in each step, I'm reducing by half. And that's when k is log base 2 of n. So that's why it's log linear.

And so this just reminds you. Again, that recap. Number of calls reduced-- or, sorry. The call gets reduced by a factor or two each time. I'm going to have a log n work going around it. And inside it's a constant amount of work because I'm just passing the pointers, I'm not actually copying the list. And that's a nice state to be.

OK, so-- sounds good. Could just use linear search. It's going to be linear. When you use binary search or bisection search, we can do it in log time. That's great. We assumed the list was sorted, but all right. So that lens basically says, OK. So when does it make sense to sort the list and then do the search? Right? Because if I can sort the list cheaply, then the search is going to be logarithmic. That's really what I would like.

This little expression basically says, let's let sort be the cost of sorting the list. I want to know when that cost plus something that's order log n-- which is what it's going to cost me to do this search. When is that less than something that's order n? Because then it's going to be better to do

the sort first than do the search. And so I can just rearrange it. It needs to be, when does the cost of sorting-- when is it last than this expression? Which basically says, when is sorting going to be less expensive than the linear cost?

Crud. Actually, good news for you, right? This is a really short lecture. Because it says it's never true. Ouch. Don't worry. We've got more to go on the lecture. The reason it can't be true-- if you think about it just informally-- is, if I've got a collection of n elements and I want to sort it, I've got to look at each one of those elements at least once. Right? I have to look at them to decide where they go. Oh, that's n elements. So sorting must be at least order n, because I got to look at everything. And in fact as it says there, I'm going to have to use at least linear time to do the sort.

Sounds like we're stuck, but we're not. And the reason is, often when I want to search something I'm going to do multiple searches, but I may only want to sort the list once. In fact, I probably only want to sort the list once. So in that case, I'm spreading out the cost. I'm amortizing the expense of the sort.

And now what I want to know is, if I'm going to do k searches, the cost of those k searches I know is going to be k log n-- because it's log to do the search. And I simply need to know, is the cost of sorting plus this-- can I have something where it's less than k searches just using linear search? And the answer is, yes. There are going to be, for large k's, ways in which we can do the sort where the sort time becomes irrelevant, that the cost is really dominated by this search.

And so what I want to do now is look at-- all right. How could we do the sort reasonably efficiently? It's going to have to be at least linear. We're going to see it's going to be a little more than linear. But if I could do it reasonably, I'm going to be in good shape here. So what I want to do is show you a number of ways in which we can do sorting-- take a list of elements and sort them from, in this case, smaller to higher or increasing order. So here's my goal. I want to efficiently sort a list. I want

to see if we can do this as efficiently as possible.

I'm going to start, you might say, with a humorous version of sort. You're all convinced that my humor is non-existent. You're right. But it sets the stage for it. This is a sort. You can look it up. It's called monkey sort, BOGO sort, stupid sort, slow sort, permutation sort, shotgun sort. And here's how it works. Anna has nicely given me a set of numbers on cards here. Here's how you do BOGO sort. I got to do that better. I got to spread them out randomly, like this. Oh good. I'm going to have to-- sorry, Tom. I'm not walking. And now I pick them up, saying, is that less than this? Which is less than-- oh, crud. They're not sorted. All right. I pick them all up and I do it again. A little brain damage, right?

Now it's intended to get your attention. I did. I heard a couple of chuckles. Those are A students, by the way. I heard a couple of chuckles here. We could actually do this exhaustively. Basically it's called permutation sort because you could search through all possible permutations to see if you find something that's sorted. That, by the way-- the complexity of that is something like n factorial, which for large n is n to the nth power. And if n's anything bigger than about 2, don't do it. Right? But it would be a way to think about doing this.

All right. Now, having caught the humorous version of this, how could we do this a little bit better? Oh sorry. I should say, what's the complexity? There's a nice crisp definition of BOGO sort. Its best case is order n, because I just need to check it's sorted. Its average case is n factorial and its worst case, if I'm just doing it randomly, is God knows. Because I could be doing it here forever. So we're going to move on.

Here's a second way to do it called bubble sort. I'm going to do this with a small version of this. I'm going to put out a set. I'll turn these up so you can see them in a second. The idea of bubble sort is, I'm going to start at-- I'm going to call this the front end of the list. And I'm going to walk down, comparing elements pairwise. And I'm always going to move the larger one over. So I start here and I say, 1 is less than 11. I'm OK. 11's bigger than five. I'm going to bubble that up. 11's bigger than 6. I'm

going to bubble that up. 11's bigger than 2. I've basically bubbled 11 to the end.

Now I go back here. I say, 1 is less than 5. That's good. 5 is less than 6. That's good. Ah, 6 is bigger than 2. Bubble that. 6 is less than 11. You get the idea-- comparison, comparison, and swap. Comparison, comparison. And now if I go back to this part and do it, you'll notice that's in the right order. That's in the right order. That's in the right order. That's in the right order. I'm done. Small round of applause, please. I was able to sort five elements. Thank you.

The little video is showing the same thing. You can see the idea here. It's called bubble sort because you're literally bubbling things up to the end of the list. It's pretty simple to do. You're just swapping pairs. And as you saw, when I get to the end of the list I go back and do it until I have a pass where I go all the way through the list and I don't do any swaps. And in that case I know I'm done because everything's in order, and I can stop. One of the properties of it is that the largest unsorted element is always at the end after the pass. In other words, after the first one I know that the largest element's at the end. After the second one, the largest thing left is going to be in the next place. And that tells me, among other things, that this is going to take no more than n times through the list to succeed. It might actually take fewer than that.

OK. Again let's look at some code for it. Let's look at its complexity and let's actually run this. So here is a little simple version of bubble sort. I'm going to set a flag up here. I'm going to call its swap initially to false. That's going to let me tell when I'm done, when I've gone through everything in the list without doing a swap. And then I'm going to loop. As long as swap is false-- so the first time through it's going to do that loop. I set swap initially to true, and notice what I then do. I let j range from 1 up to the length of the list, and I look at the jth element and the previous element. If the previous element is bigger, I'm going to flip them. Right there. And that's just doing that swap, what I just did down here.

And if that's the case, I'm going to set the flag to false. Which says, I've done at least one bubble as part of this. Which means when I come out of here and go back around to the loop, it's going to do it again. And it will do it until all of this succeeds without this ever being true, in which case that's true, which makes that false. And it will drop out. OK?

Let's look at an example of this running. It's just to give you a sense of that, assuming I can find the right place here. So there is, again, a version of bubble sort on the side. And I'm going to bring this down to the bottom I've got a little test list there. And I've put a print statement in it. So you can see each time through the loop, what's the form of the list as it starts. And assuming I've done this right-- here you go.

There's the list the first time through. Notice after one pass, 25's at the end of the list-- the biggest element. Exactly what I like. But you can also see a few other things have flipped. Right? Right in there, there have been some other swaps as it bubbled through. And in fact, you can see it's-- well, you can see that idea. You can see 25 moving through. Notice on the next step, a whole bunch of the list is actually in the right order. It's just because I got lucky. All I can guarantee is that the second largest element is the second from the end of the list. But you can see here. Even though the list is, I think, nine long, it only took us four passes through. So this is nice. It says, at most, n times through the list. And at the end, we actually get out something that's in the right form.

OK. So let's go back to this and basically say, what's the complexity? Well that's length n, right? Has to be. I'm going through the entire list. And inside of there is just constant work. Four operations. I'm doing a test. Sorry, five. I'm doing a test. And then depending whether that test is true or not, I'm setting a flag and doing some movement of things around. But it's just constant. I don't care about the five. And there, how many times do I go around the loop? In the worst case, n. All I can guarantee is, after the first pass the biggest thing is here. After the second pass, the second biggest thing is there. After the third pass-- you get the idea.

So I've got order and things inside the loop, and I'm doing that loop n times. And I hope that looks familiar. We've talked about this. Right? This is nested loops. What's this? Quadratic. So it's order n squared, where n is the length of the list. Now as you also saw, on average, it could be less than that. But it's going to be order n squared.

OK. That's one possibility. Here's a second nice, simple, sort algorithm. It's called selection sort. You can kind of think of this as going the other way. Not completely, but going the other way. And when I say going the other way, the idea here is that I'm going to find the smallest element in the list. And I'm going to stick it at the front of the list when I'm done, and simply swap that place with whatever was there. Flip them. I might do a few other flips along the way, depending how I implement this.

Next, pass. I'm just going to look at everything but the first element, because I know that one's done. I'm going to do the same thing. Find the smallest element remaining in the list, put it in the second spot, and keep doing that. What I know is, if I implement this correctly, after i steps the first i elements of the list will be sorted. And everything in the rest of the list has to be bigger than the largest thing in the first part of the list.

OK. So we could build that. Before we do it, I'm going to show you a little video starring Professor Guttag. This is his cameo performance here. But I want to just show you an example of this using not numbers, but people.

[VIDEO PLAYBACK]

- All right. So now we're going to do selection sort. The idea here is that each step we're going to select the shortest person and put them next in line of the sorted group. So we'll bring the leftmost person forward, and we will compare her to everybody else. So one at a time, step forward. You're still the winner. You go

back. Please step forward.

PROFESSOR: And watch the number of comparisons that go on, by the way. We're going to come back to that.

- Next. Still the winner. Next. Ah. A new winner. All right. So you can take her place.

PROFESSOR: So here, we're choosing to actually insert into the spot in the Line We could have put her back at the front, but either one will work.

- Now we'll compare. Same old winner. Same winner. No change. It's getting kind of boring. Don't fall, that-- same winner. Please.

PROFESSOR: This is a tough one.

- Oh. Close, but I think you're still shorter. All right. Next. No change, which means you are the first in line. Congratulations.

PROFESSOR: So, smallest element now going to be the first slot.

- Now you step forward, and we'll compare you.

PROFESSOR: I would invite you to watch the left hand of the list. Notice how it is slowly building up at each stage to have that portion sorted. And we deliberately admit students to be of different heights, so John can do this demo.

- You are the winner. Take your place in line. Next. It's you. And once again, we have a lovely group of students sorted in height order.

[END PLAYBACK]

[APPLAUSE]

PROFESSOR: And check out-- I want you to remember number of comparisons-- 55. Not that the [INAUDIBLE], but I want you to see a comparison as we go on in a second.

So again, selection sort. This is this idea of, find the smallest element. Put it at the front. I might do a little number of flips, as you can see, here along the way. But this is the same animation of that. So let's first of all convince ourselves it will do the right thing, and then look at some code, and then run the code.

So to convince ourselves that this is going to do the right thing, we could talk about something that we often refer to as a loop invariant. We're going to write a loop, but we're going to walk through this. And the invariant here-- and we want to just demonstrate if it's true at the beginning and it's true at each step. Therefore, by induction as we did earlier, I can conclude it's true always. Is that if I'm given the prefix or the first part of a list from 0 up to i, and a suffix or a second part of the list from i plus 1 up to the end of the overall list-- given that, then I want to assert that the invariant is that the prefix is sorted and no element of the prefix is larger than the smallest element of the suffix. Just what I said earlier. It says, at any stage here-- if this is the amount of sort I've done so far-- I can guarantee, I'm going to claim, this will be sorted. And everything here is bigger than that thing there.

How do I prove it? Well the base case is really easy. In the base case, the prefix is empty. I don't have anything, so it's obviously sorted. And everything in the suffix is bigger than anything in the prefix. So I'm fine. And then I just want to say, as long as I write my code so that this step is true, then I'm going to move the smallest element from the suffix-- the second part of the list-- to the end of the prefix. Since the prefix was sorted, this is now sorted. And everything in the suffix is still going to be bigger than everything in the prefix. And as a consequence, by

induction, this is going to give me something that says it's always going to be correct.

So here's code that would do that. Here. I'm just going to set a little thing called the start of suffix, or soft start. Initially it's going to point to the beginning of the list. And then I'm going to run a loop. And as long as I still have things to search in the list, that that pointer doesn't point to the end of the list, what am I going to do? I'm going to loop over everything from that point to the end of the list, comparing it to the thing at that point. If it's less than, I'm going to do a swap because I wanted to move it up. And you can see, by the time I get through this loop I will have found the smallest element in the remainder of the list. And I would have put it at that spot, whatever suffix start points to.

And when I've done all of that, I just change this by one. Having found the smallest element, I've stuck it at spot zero. I'll do the same thing. Having found the next smallest element, I know it's at point one. And I'll just continue around. One of the things you can see here is, as opposed to bubble sort, this one is going to take n times around the loop because I'm only moving this pointer by one So it starts at 0, and then 1, and then 2, all the way up to n minus 1.

You can also see in this particular implementation, while I'm certainly ensuring that the smallest element goes into that spot, I may do a few other flips along the way. I'm going to find something I think is the smallest element, put it there and put that element here. And then when I find another smaller element, I may do that flip. I could have implemented this where I literally search for the smallest element and only move that. Doesn't make any difference in terms of the complexity.

All right. What's the complexity here? Already said this part. I will loop n times, because I start at 0 and then 1. You get the idea. Inside of the loop I'm going to walk down the remainder of the list, which is initially n. And then n minus 1, and then n minus 2 times. But we've seen that before as well. While they get shorter, that complexity is still quadratic. Order n times going through this process. Within the process, order n

things that I have to compare. And yes, n gets smaller. But we know that that n term, if you like to dominate. So again, this is quadratic.

OK. Before you believe that all sorting algorithms are quadratic, I want to show you the last one, the one that actually is one of the-- I think-- the prettiest algorithms around, and a great example of a more efficient algorithm. It's called merge sort. Merge sort takes an approach we've seen before. We talked about divide and conquer. Break the problem down into smaller versions of the same problem. And once you've got those solutions, bring the answer back together. For merge sort, that's pretty easy. It says, if I've got a list of 0 or 1 elements, it's sorted. Duh.

OK. If I got a list of more than 1 element, here's my trick. I'm going to split it into two lists. I'm going to sort them. And when I'm done, I'm just going to merge those two lists into one list. And the merge is easy. Because if I've got two lists that are sorted, I just need to look at the first element of each, take the one that's smaller. Add it to my result. And keep doing that until one of the lists is empty. And then just copy the remainder of the other list. You can probably already get a sense of what the cost is going to be here, because this is cutting the problem in half.

Now I've got two pieces. So I need to think about both of them. I want to give you a couple of visualizations of this. Here's the first one. It says, basically, I've got a big unsorted list. I'm going to split it. And I'm going to split it. And I'm going to split it. Until I get down to just lists that are either 0 or 1, which by definition are sorted. And once I'm at that level, then I just have to merge them into a sorted list and then merge them pairwise into a sorted list. And you get the idea.

So it's divide and conquer. The divide is dividing it up into smaller pieces. The conquer is merging them back together. And we have Professor Guttag back for an encore, together with his students. So let's show you an example of merge sort.

[VIDEO PLAYBACK]

- So we're about to demonstrate merge sort. And we're going to sort this rather motley collection of MIT students by height. So the first thing we need to do is, we're going to ask everyone to split into a group of two. So you split a little bit. You two are together. You two are together. You two are together. You two are together. You two are together. And you are all by yourself. I'm sorry.

PROFESSOR: Poor Anna.

- All right. So now let's take the first group. Take a step down. And what we do is, we sort this group by height, with the shortest on the left. And look at this. We don't have to do anything. Thank you. Feel free to go back up. We then sort the next pair. Please. And it looks to me like we need to switch. All right. Take a step back. Ladies-- OK. Ladies, gentlemen-- also OK. And again, OK.

PROFESSOR: Notice each subgroup is now sorted. Which is great.

- And I think you're in the correct order. Now what we do is, we take these groups and merge the groups. So let's have these two-- going to sort these groups, have them step forward. And now what we're doing is, we're doing a merge of the two sorted groups. So we start by merging them. We'll take the leftmost person in this group and compare her to the first person in this group, and decide.

She's still the shortest. Take a step back. Now we're going to look at you and say, you're actually taller than this fellow. So you now step up there. And we're good here. Both of you take a step back.

Now we'll take these two groups and follow the same procedure. We'll merge them. Let's see. We'll compare you-- the first person in this group to the first person in this group. Now it's a little tricky. So let's see, the two of you compare. Let's see, back to back. We have a winner. Step back. And now we need to compare the shortest person in this group to the shortest person in this group. We have a winner. It's you. I'm sorry. And now we just-- we're OK. Please step back.

Now we'll have these two groups come forward. We'll compare the shortest person in this group to the shortest person in that group. I actually need you guys to get back to back here. You are the winner. And it's pretty clear that the shortest person in this group is shorter than the shortest person in that group. So you go there and you step back.

PROFESSOR: Notice the groups. Now all sorted.

- And now we repeat the same process.

PROFESSOR: And notice how the whole subgroup now goes up once we know that one group is empty.

- And you can see that we have a group of students sorted in order by height.

[END PLAYBACK]

[APPLAUSE]

PROFESSOR: Remember the first number, right? 55, 28. Now it's just numbers but you can see the expectation is, this is going to take less time. And it certainly did there. So again just to demo another way visually. I'm sorting-- sorry.

I am splitting down until I get small things, and then just merging them up. I may have to do multiple passes through here, but it's going to be hopefully faster than the other methods we looked at.

I'm going to show you code in a second, and then we're going to run it just to see it. But let me stress one more time just the idea of merging. You can see the idea. I keep splitting down till I got something small enough. And I want to merge them back. The idea of merging-- you've seen it from Professor Guttag. But I just want to highlight why this is going to be efficient. If I've got two lists: list 1 and list 2, the things left there. Process is very simple. I pull out the smallest element of each. I compare them. And I simply put the smallest one into the result, move on in that first list. So the 1 disappears from that left list. And now again I pull up just the smallest element of each one, do the comparison. Smallest one goes to the end of my result. And I drop that element from its list. So I've now taken 1 from list 1 and one from list 2. You get the idea.

The reason I want to give you this visualization-- sorry. Let me do the last step. Once I get to a place where one of the lists is empty, just copy the rest of the list onto the end. You can see already a hint of the code. And that is, that I'm only going to ever look at each element of each sublist once as I do the merge. And that's a nice property. Having had them sorted, I don't need to do lots of interior comparisons. I'm only comparing the ends of the list. I only, therefore, look at each element-- the number of comparisons, rather, I should say. I may look at each element more than once. The number of comparisons is going to be, at most, the number of elements in both lists. And that's going to be a nice Q as we think about how to solve it.

So here's the code to merge, and then we'll write Merge Sort. And I know there's a lot of code here, but we can walk through it and get a good sense of it. I'm going to set up a variable called Result that's going to hold my answer. And I'm going to set up two indices, i and j, that are initially 0. They're pointing to the beginning. And remember, the input

here is two lists that we know are sorted-- or should be sorted, or we screwed up in some way. So initially, i and j are both pointing to the beginning of the left and right list. And look at what we do. We say, as long as there's still something in the left list and still something in the right list-- i is less than the length of left, j is less than the length of right.

Do the comparison. If the left wants smaller, add it to the end of result. To the end of result, right? I'm appending it because I want it to be in that sorted order. And increase i. If it's not, add the right one to the end of result and increase j. And I'll just keep doing that until I exhaust one of the lists. And when I do I can basically say, if the right list is empty, I know if I get out of here they can't both be true. In other words, if there's still something in the left list, just put it on the end. Otherwise if the only things left are in the right list, just put them on the end. So I'm just walking down the list, doing the comparison, adding the smallest element to my result. And when I'm done, I just return result.

Complexity we can already begin to see here, right? This says the left and right sublists are ordered, so I'm just moving the indices depending on which one holds the smaller element. And when I get done, I'm just returning the rest of the list. So what's the complexity here? I'm going to do this a little more informally. You could actually do that kind of relationship I did last time. But what am I doing? I'm going through the two lists, but only one time through each of those two lists. I'm only comparing the smallest elements. So as I already said, this says that the number of elements I copy will be everything in the left list and everything in the right list. So that order is just the length of left plus the length of right.

And how many comparisons do I do? The most I have to do is however many are in the longer list. Right? That's the maximum number I need to have. Oh, that's nice. That says, if the lists are of order n-- I'm doing order n copies, because order n plus order n is just 2n, which is order n-- then I'm doing order n comparisons. So it's linear in the length of the lists.

OK. Sounds good. That just does the merge. How do I do merge sort? Well we said it. Break the problem in half. Keep doing it until I get sorted lists. And then grow them back up. So there's merge sort. It says, if the list is either empty or of length 1, just return a copy of the list. It's sorted. Otherwise find the middle point-- there's that integer division-- and split. Split the list everything up to the middle point and do merge sort on that. Split everything in the list from the middle point on. Do merge sort on that. And when I get back those two sorted lists, just merge them.

Again, I hope you can see what the order of growth should be here. Cutting the problem down in half at each step. So the number of times I should have to go through this should be to log n the size of the original list. And you can see why we call it divide and conquer. I'm dividing it down into small pieces until I have a simple solution and then I'm growing that solution back up. So there is the base case, there's the divide, and there's the nice conquer [INAUDIBLE] piece of this.

OK. I'm going to show you an example of that. But let's actually look at some code-- sorry about that. Let's look at some code to do this. And in fact I meant to do this earlier and didn't. I also have a version of bubble sort here. Sorry-- selection sort. I've already done bubble sort. There is selection sort. Let's uncomment this. And let's run both of those and just see the comparison between them. Yeah, sorry-- just make that a little easier to read. There we go.

So we saw a bubble sort. It only went through four times, so less than n times. There's selection sort. And as I said to you, it has to do n passes it because it can only ever guarantee that it gets one element at the beginning. So you can in fact see, in this case, from the first or after the initial input until the end of the first step, it looks like it didn't do anything because it determined eventually that one was in the right spot. And similarly I think there's another one right there where it doesn't do any-- or appears not to do anything. All it's guaranteeing is that the next smallest element is in the right spot. As we get through to the end of it, it in fact ends up in the right place.

And then let's look at merge sort and do one more visualization of this. Again let me remove that. If we run it-- again, I've just put some print statements in there. Here you can see a nice behavior. I start off calling Merge Sort with that, which splits down into doing Merge Sort of this portion. Eventually it's going to come back down there and do the second one. It keeps doing it until it gets down to simple lists that it knows are sorted. And then it merges it. Does the smaller pieces and then merges it. And having now 2 merged things, it can do the next level of merge. So you can see that it gets this nice reduction of problems until it gets down to the smallest size.

So let's just look at one more visualization of that and then get the complexity. So if I start out with this list-- sorry about that. What I need to do is split it. Take the first one, split it. Keep doing that until I get down to a base case where I know what those are and I simply merge them. Pass it back up. Take the second piece. Split it until I get down to base cases. Do the merge, which is nice and linear. Pass that back up. Having done those two pieces, I do one more merge. And I do the same thing.

I want you to see this, because again you can notice how many levels in this tree log. Log in the size. Because at each stage here, I went from a problem of 8 to two problems of 4. Each of those went to two problems of 2, and each of those went to two problems of size 1.

All right. So the last piece is, what's the complexity? Here's a simple way to think about it. At the top level, I start off with n elements. I've got two sorted lists of size n over 2. And to merge them together, I need to do order n work. Because as I said I got to do at least n comparisons where n is the length of the list. And then I've got to do n plus n copies, which is just order n. So I'm doing order n work.

At the second level, it gets a little more complicated. Now I've got problems of size n over 4. But how many of them do I have? 4. Oh, that's nice. Because what do I know about this? I know that I have to copy each element at least once. So not at least once. I will copy each

element exactly once. And I'll do comparisons that are equal to the length of the longer list. So I've got four sublists of length n over 4 that says n elements. That's nice. Order n. At each step, the subproblems get smaller but I have more of them. But the total size of the problem is n. So the cost at each step is order n. How many times do I do it? Log n. So this is log n iterations with order n work at each step. And this is a wonderful example of a log linear algorithm. It's n log n, where n is the length of the list.

So what you end up with, then, is-- all right, a joke version, some reasonable ways of doing sort that are quick and easy to implement but are quadratic, and then an elegant way of doing the search that's n log n. And I'll remind you I started by saying, as long as I can make the cost of sorting small enough I can amortize that cost. And if you go back and look at last lecture's notes, you'll see n log n grows pretty slowly. And it's actually a nice thing to do. It makes it reasonable to do the sort. And then I can do the search in order n time.

And here's the last punchline. It's the fastest we can do. I'm going to look at John again. I don't think anybody has found a faster sort algorithm. Right? This is the best one can do. Unless you do-- sorry, the best worst case. I'm sorry. John is absolute right. There are better average cases. Again, our concern is worst case. So this is as good as we're going to do in terms of a worst case algorithm. So there you now have sorting algorithms and searching algorithms, and you've now seen-- excuse me, sorry-- constant, log, linear, log linear, quadratic, and exponential algorithms. I'll remind you, we want things as high up in that hierarchy as possible.

All right. I have six minutes left. Some of you are going to leave us. We're going to miss you, but that's OK. I'm sure we'll see later on. For those of you hanging around, this isn't a bad time just to step back and say, so what have we seen? And I want to do this just very quickly. I'm sorry. And I'll remind you, we started by in some sense giving you a little bit of a contract of things we were going to show you. And I would simply

suggest to you, what have we done? We've given you a sense of how to represent knowledge with data structures, tuples, lists, dictionaries, more complicated structures. We've shown you some good computational metaphors, iteration, and loops. Recursion has a great way of breaking problems down into simpler versions of the same problem. And there really are metaphors. There are ways of thinking about problems.

We've given you abstraction, the idea of capture a computation, bury it in a procedure. You now have a contract. You don't need to know what happens inside the procedure as long as it delivers the answer it says it would. Or another way of saying it, you can delegate it to somebody and trust that you're going to get what you like out of it. We've seen classes and methods as a wonderful way to modularize systems, to capture combinations of data and things that operate on them in a nice, elegant way. And we just spent a week and a half talking about classes of algorithms and their complexity.

If you step up a level, what we hope you've gotten out of this are a couple of things. You've begun to learn computational modes of thinking. How do I tackle a problem and divide and conquer? How do I think about recursion as a tool in dealing with something? You've begun to-- begun, I will use that word deliberately-- to master the art of computational problem solving. How can you take a problem and turn it into an algorithm? And especially, you've begun to have the ability to make the computer do what you want it to. To say, if I've got a problem from biology or chemistry or math or physics or chemical engineering or mechanical engineering, how do I take that problem and say, here's how I would design an algorithm to give me a simulation and a way of evaluating what it does.

And so what we hope we've done is, we've started you down the path to being able to think and act like a computer scientist. All right. Don't panic. That doesn't mean you stare at people's shoes when you talk to them. Not all computer scientists do that, just faculty. Sorry, John. So

what do computer scientists do? And this is actually meant to be serious. And I put up two of my famous historical figures of computer scientists. They do think computationally. They think about abstractions, about algorithms, about automated execution. So the three A's of computational thinking. And in the same way that traditionally you had the three R's of reading, writing, and arithmetic, computational thinking we hope is becoming a fundamental that every well-educated person is going to need.

And that says, you think about the right abstraction. When you have a problem in your [INAUDIBLE] what's the right abstraction? How do I pull apart the pieces? How do I think about that in terms of decomposing things into a relationship that I can use to solve problems? How do I automate? How do I mechanize that abstraction? How do I use what I know happens inside of the machine to write a sequence of steps in a language I'm using to capture that process? And then finally, how do I turn that into an algorithm? And that not only means I need a language for describing those automated processes, and if you like allowing the abstraction of details, but frankly also a way to communicate. If you have to think crisply about how do I describe an algorithm, it's actually giving you a way to crystallize or clarify your thinking about a problem. This is not to say you should talk to your friends in Python. I don't recommend it. But it does say you should use that thinking as a way of capturing your ideas of what you're going to do.

And that leads, then, to this idea of, how difficult is a problem? How best can I solve it? We've shown you these complexity classes and we've hinted at the idea that in fact some problems are inherently more difficult than others. That's something I hope you come back to as you go along. And especially we want you to start thinking recursively. We want you to think about how do I take a hard problem, break it up into simpler versions of the same problem, and then construct the solution. And that shows up lots of places. Right? Recursion is in all sorts of wonderful places. So just to give you an example, I could say to you recursively, "This lecture will end when I'm done talking about this lecture, which will

end when I'm done talking about this lecture, which will end when I'm done--"

All right. You don't like infinite recursion. Good luck on the exam.