PROFESSOR JOHN GUTTAG: In the example we looked at, we had a list of ints. That's actually quite easy to do in constant time. If you think about it, an int is always going to occupy the same amount of space, roughly speaking, either 32 or 64 bits, depending upon how big an int the language wants to support.

So let's just, for the sake of argument, assume an int occupies four units of memory. And I don't care what a unit is. Is a unit 8 bits, 16 bits? It doesn't matter. 4 units. How would we get to the i-th element of the list? What is the location in memory of L-th of i?

Well, if we know the location of the start of the list-- and certainly we can know that because our identifier, say L in this case, will point to the start of the list-- then it's simply going to be the start plus 4 times i.

My list looks like this. I point to the start. The first element is here. So, that's start plus 4 times 0. Makes perfect sense. The second element is here. So, that's going to be start plus 4 times 1. Sure enough, this would be location 4, relative to the start of the list, et cetera.

This is a very conventional way to implement lists. But what does its correctness depend upon? It depends upon the fact that each element of the list is of the same size. In this case, it's 4. But I don't care if it's 4. If it's 2, it's 2 times i. If it's 58, it's 58 times i. It doesn't matter. But what matters is that each element is the same size. So this trick would work for accessing elements of lists of floats, lists of ints, anything that's of fixed size.

But that's not the way lists are in Python. In Python, I can have a list that contains ints, and floats, and strings, and other lists, and dicts, almost anything. So, in

Python, it's not this nice picture where the lists are all homogeneous. In many languages they are, by the way. And those languages would implement it exactly as I've outlined it on the board here. But what about languages where they're not, like Python?

One possibility-- and this is probably the oldest way that people used to implement lists-- is the notion of a linked list. These were used way back in the 1960s, when Lisp was first invented. And, effectively, there, what you do is a list. Every element of the list is a pointer to the next element. And then the value.

So what it looks like in memory is we have the list. And this points to the next element, which maybe has a much bigger value field. But that's OK. This points to the next element. Let's say this one, maybe, is a tiny value field. And then at the end of the list, I might write none, saying there is no next element. Or nil, in Lisp speak.

But what's the cost here of accessing the nth element of the list, of the i-th element of the list? Somebody? How many steps does it take to find element i?

AUDIENCE: i.

AUDIENCE: i?

PROFESSOR JOHN GUTTAG: i steps, exactly. So for a linked list, finding the i-th element is order i. That's not very good. That won't help me with binary search. Because if this were the case for finding an element of a list in Python, binary search would not be log length of the list, but it would be order length of the list. Because the worst case is I'd have to visit every element of the list, say, to discover something isn't in it. So, this is not what you want to do.

Instead, Python uses something like the picture in your handout. And the key idea here is one of indirection. So in Python, what a list looks like is it is a list, a section of memory, a list of objects each of the same size. Because now what each object is a pointer. So we've now separated in space the values of the members of the list and the pointers to, if you will, the next one.

So now, it can be very simple. This first element could be big. Second element could be small. We don't care. Now I'm back to exactly the model we looked at here. If, say, a pointer to someplace in memory is 4 units long, then to find l-th of i, I use that trick to find, say, the i-th pointer. And then it takes me only one step to follow it to get to the object.

So, I can now, in constant time, access any object into a list, even though the objects in the list are of varying size. This is the way it's done in all object-oriented programming languages. Does that makes sense to everybody?

This concept of indirection is one of the most powerful programming techniques we have. It gets used a lot. My dictionary defines indirection as a lack of straightforwardness and openness and as a synonym uses deceitfulness. And it had this pejorative term until about 1950 when computer scientists discovered it and decided it was a wonderful thing.

There's something that's often quoted at people who do algorithms. They say quote, "all problems in computer science can be solved by another level of indirection." So, it's sort of, whenever you're stuck, you add another level of indirection. The caveat to this is the one problem that can't be solved by adding another level of indirection is too many levels of indirection, which can be a problem.

As you look at certain kinds of memory structures, the fact that you've separated the pointers from the value fields can lead to them being in very far apart in memory, which can disturb behaviors of caches and things like that. So in some models of memory this can lead to surprising inefficiency. But most of the time it's really a great implementation technique. And I highly recommend it.

So that's how we do the trick. Now we can convince ourselves that binary search is indeed order log n. And as we saw Tuesday, logarithmic growth is very slow. So it means we can use binary search to search enormous lists and get the answer very quickly.

All right. There's still one catch. And what's the catch? There's an assumption to binary search. Binary search works only when what assumption is true?

AUDIENCE: It's sorted.

PROFESSOR JOHN GUTTAG: The list is sorted because it depends on that piece of knowledge. So, that raises the question, how did it get sorted? Or the other question it raises, if I ask you to search for something, does it make sense to follow the algorithm of (1) sort L, (2) use binary search? Does that make sense?

Well, what does it depend upon, whether this makes sense from an efficiency point of view? We know that that's order log length of L. We also know if the list isn't sorted, we can do it in order L. We can always use linear search. So, whether or not this makes a good idea depends upon whether we can do this fast enough.

It has the question is order question mark plus order log len of L less than order L? If it's not, it doesn't make sense to sort it first in sums, right? So what's the answer to this question? Do we think we can sort a list fast enough? And what would fast enough mean? What would it have to be?

For this to be better than this, we know that we have to be able to sort a list in sublinear time. Can we do that? Alas, the answer is provably no. No matter how clever we are, there is no algorithm that will sort a list in sublinear time.

And if you think of it, that makes a lot of sense. Because, how can you get a list in ascending or descending order without looking at every element in the list at least once? Logic says you just can't do it. If you're going to put something in order, you're going to have to look at it.

So we know that we have a lower bound on sorting, which is order L. And we know that order L plus order log length L is the same as order L, which is not better than that.

So why do we care? If this is true, why are we interested in things like binary search at all? And the reason is we're often interested in something called amortized complexity. I know that there are some course 15 students in the class who will know what amortization means. But maybe not everybody does.

The idea here is that if we can sort the list once and end up searching it many times, the cost of the sort can be allocated, a little bit of it, to each of the searches. And if we do enough searches, then in fact it doesn't really matter how long the sort takes.

So if we were going to search this list a million times, maybe we don't care about the one-time overhead of sorting it. And this kind of amortized analysis is quite common and is what we really end up doing most of the time in practice.

So the real question we want to ask is, if we plan on performing k searches-- who knows how long it will take to sort it-- what we'll take is order of whatever sort of the list is, plus k times log length of L. Is that less than k times len of L? If I don't sort it, to do k sort searches will take this much time. If I do sort it, it will take this much time.

The answer to this question, of course, depends upon what's the complexity of that and how big is k. Does that make sense? In practice, k is often very big. The number of times we access, say, a student record is quite large compared to the number of times students enroll in MIT.

So if at the start of each semester we produce a sorted list, it pays off to do the searches. In fact, we don't do a sorted list. We do something more complex. But you understand the concept I hope.

Now we have to say, how well can we do that? That's what I want to spend most of the rest of today on now is talking about how do we do sorting because it is a very common operation. First of all, let's look at a way we don't do sorting.

There was a famous computer scientist who opined on this topic. We can look for him this way. A well-known technique is bubble sort.

Actually, stop. We're going to need sound for this. Do we have sound in the booth? Do we have somebody in the booth? Well, we either have sound or we don't. We'll find out shortly. Other way. Come on. You should know. Oh there. Thank you.

[VIDEO PLAYBACK]

-Now, it's hard to get a job as President. And you're going through the rigors now. It's also hard to get a job at Google. We have questions, and we ask our candidates questions. And this one is from Larry Schwimmer.

[LAUGHTER]

-You guys think I'm kidding? It's right here. What is the most efficient way to sort a million 32-bit integers?

[LAUGHTER]

-Well, uh.

-I'm Sorry, maybe that's not a--

-I think the bubble sort would be the wrong way to go.

[LAUGHTER]

-Come on, who told him this? I didn't see computer science in your background.

-We've got our spies in there.

-OK, let's ask a different interval--

[END VIDEO PLAYBACK]

PROFESSOR JOHN GUTTAG: All right, so as he sometimes is, the President was correct. Bubble sort, though often discussed, is almost always the wrong answer. So we're not going to talk about bubble sort.

I, by the way, know Larry Schwimmer and can believe he did ask that question. But yes, I'm surprised. Someone had obviously warned the President, actually the then future president I think.

Let's look at a different one that's often used, and that's called selection sort. This is about as simple as it gets. The basic idea of selection sort-- and it's not a very good way to sort, but it is a useful kind of thing to look at because it introduces some ideas. Like many algorithms, it depends upon establishing and maintaining an invariant.

An invariant is something that's invariantly true. The invariant we're going to maintain here is we're going to have a pointer into the list. And that pointer is going to divide the list into a prefix and a suffix. And the invariant that we're going to maintain is that the prefix is always sorted.

We'll start where the prefix is empty. It contains none of the list. And then each step through the algorithm, we'll decrease the size of the suffix by one element and increase the size of the prefix by one element while maintaining the invariant. And we'll be done when the size of the suffix is 0, and therefore the prefix contains all the elements. And because we've been maintaining this invariant, we know that we have now sorted the list.

So, you can think about it. For example, if I have a list that looks like 4, 2, 3, I'll start pointing here. And the prefix, which contains nothing, obeys the invariant. I'll then go through the list and find the smallest element in the list and swap it with the first element.

My next step, the list will look like 2, 4, 3. I'll now point here. My invariant is true. The prefix contains only one element, so it is in ascending order. And I've increased its size by 1. I don't have to look at this element again because I know by construction that's the smallest.

Now I move here, and I look for the smallest element in the suffix, which will be 3. I swapped 3 and 4. And then I'm going to be done. Does that make sense?

It's very straightforward. It's, in some sense, the most obvious way to sort a list. And if you look at the code, that's exactly what it does. I've stated the invariant here. And I just go through and I sort it. So we can run it. Let's do that.

I'm going to sort the list 3, 4, 5, et cetera, 35, 45. I'm going to call selection sort. And I don't think this is in your handout, but just to make it obvious what's going on, each iteration of the loop I'm going to print the partially sorted list so we can see what's happening.

The first step, it finds 4 and puts that in the beginning. It actually finds 0, puts it in the beginning, et cetera. All right? So, people see what's going on here? It's essentially doing exactly what I did on the board over there. And when we're done, we have the list completely sorted.

What's the complexity of this? What's the complexity of selection sort? There are two things going on. I'm doing a bunch of comparisons. And I'm doing a bunch of swaps. Since I do, at most, the same number of comparisons as I do swap or swaps as I do comparisons-- I never swap without doing a comparison-- we can calculate complexity by looking at the number of comparisons I'm doing. You can see that in the code as well.

So how many comparisons might I have to do here? The key thing to notice is each time I look at it, each iteration, I'm looking at every element in what? In the list? No, every element in the suffix.

The first time through, I'm going to look at-- let's just say n equals the length of the list. So the first time through, I'm going to look at n elements. Then I'm going to look at n minus 1. Then I'm going to look at n minus 2. Until I'm done, right? So that's how many operations I'm doing.

And what is the order of n plus n minus 1 plus n minus 2? Exactly. Order n. So, selection sort is order n. Is that right? Somebody said order n. Do you believe it's n? Is this really n? It's not n. What is it? Somebody raise your hand, so I can throw the candy out. Yeah.

AUDIENCE: [INAUDIBLE]

PROFESSOR It's not n factorial.

JOHN GUTTAG:

AUDIENCE: n-squared?

PROFESSOR JOHN GUTTAG: You said that with a question mark at the end of your voice.

AUDIENCE: No, it's like the sum of the numbers is, like, n times n minus 1 over 2 or something like that.

PROFESSOR JOHN GUTTAG: It's really exactly right. It's a little smaller than n-squared, but it's order n-squared. I'm doing a lot of these additions. So I can't ignore all of these extra terms and say they don't matter. It's almost as bad as comparing every element to every other element.

So, selection sort is order n-squared. And you can do it by understanding that sum or you can look at the code here. And that sort of will also tip you off.

Ok. so now, can we do better? There was a while where people were pretty unsure whether you could do better. But we can.

If we think about it now, it was a method invented by John von Neumann, a very famous guy. And he, back in the '40s amazingly enough, viewed this as a kind of divide and conquer algorithm. And we've looked at divide and conquer before.

What is the general form of divide and conquer? A phrase you've heard me use many times, popularized, by the way, I think, by Machiavelli in *The Prince*, in a not very nice context.

So, what we do-- and they're all of a kind, the same-- we start with 1. Let me get over here and get a full board for this. First, we have to choose a threshold size. Let's call it n0. And that will be, essentially, the smallest problem.

So, we can keep dividing, making our problem smaller-- this is what we saw with binary search, for example-- until it's small enough that we say, oh the heck with it. We'll stop dividing it. Now we'll just solve it directly. So, that's how small we need to do it, the smallest thing we'll divide things into.

The next thing we have to ask ourselves is, how many instances at each division? We have a big problem. We divide it into smaller problems. How many are we going to divide it into? We divide it into smaller problems until we reach the threshold where we can solve it directly.

And then the third and most important part is we need some algorithm to combine the sub-solutions. It's no good solving the small problem if we don't have some way to combine them to solve the larger problem. We saw that before, and now we're going to see it again. And we're going to see it, in particular, in the context of merge sort.

If I use this board, can people see it or is the screen going to occlude it? Is there anyone who cannot see this board if I write on it? All right then, I will write on it.

Let's first look at this problem. What von Neumann observed in 1945 is given two sorted lists-- and amazingly enough, this is still the most popular sorting algorithm or one of the two most popular I should say-- you can merge them quickly.

Let's look at an example. I'll take the lists 1, 5, 12, 18, 19, and 20. That's list one. And I'll try and merge it with the list 2, 3, 4, and 17.

The way you do the merge is you start by comparing the first element to the first element. And then you choose and say all right, 1 is smaller than 2. So that will be the first element of the merge list. I'm now done with 1, and I never have to look at it again.

The next thing I do is I compare 5 and 2, the head of the two remaining lists. And I say, well, 2 is smaller than 5. I never have to look at 2 again. And then compare 5 and 3. I say 3 is smaller. I never have to look at 3 again. I then compare 4 and 5. 4 is smaller. I then compare 5 and 17. 5 is smaller. Et cetera.

Now, how many comparisons am I going to do this time? Well, let's first ask the question, how many elements am I going to copy from one of these lists to this list?

Copy each element once, right? So, the number of copies is order len of the list. That's pretty good. That's linear. That's sort of at the lower bound. But how many comparisons? That's a little trickier to think about.

AUDIENCE: [INAUDIBLE]

PROFESSOR JOHN GUTTAG: Pardon?

AUDIENCE: At most, the length of the longer list.

PROFESSOR JOHN GUTTAG: At most, the length of the longer list, which would also be, we could claim to be, order len of-- I sort of cheated using L when we have two lists. But just think of it as the longer list. So, you'd think that many comparisons. You think we can do this whole thing in linear time? And the answer is yes. That's our merge. That's a good thing. Now, that takes care of this step.

But now we have to ask, how many times are we going to do a merge? Because remember, this worked because these lists were sorted. And so I only had to compare the front of each list.

When I think about how I'm going to do the binary or the merge sort, what I'm going to do is take the original list, break it up, break it up, break it up, break it up, until I have a list of length 1. Well, those are all sorted, trivially sorted. And then I'll have, at the end, a bunch of lists of length 1. I'll merge pairs of those.

Now I'll have sorted lists of length 2. Then I'll merge those, getting sorted lists of length 4. Until at the end, I'll be merging two lists, each half the length of the original list. Right. Does that make sense to everybody?

Now I have to ask the question, how many times am I going to call merge? Yeah.

AUDIENCE: Base 2 log of one of the lists.

PROFESSOR JOHN GUTTAG: Yes, I'm going to call merge log length of the list times. So, if each merge is order n where n is length of the list, and I call merge log n times, what's the total complexity of the merge sort?

AUDIENCE: nlog(n).

PROFESSOR JOHN GUTTAG: nlog(n). Thank you. Let's see, I have to choose a heavy candy because they carry better. Not well enough though. All right, you can relay it back.

Now let's look at an implementation. Here's the implementation of sort. And I don't think you need to look at it in detail. It's doing exactly what I did on the board. Actually, you do need to look at in detail, but not in real time. And then sort.

Now, there's a little complication here because I wanted to show another feature to you. For the moment, we'll ignore the complication, which is-- it's, in principle, working, but it's not very bright. I'll use the mouse.

What we see here is, whenever you do a sort, you're sorting by some ordering metric. It could be less than. It could be greater than. It could be anything you want. If you're sorting people, you could sort them by weight or you could sort them by height. You could sort them by, God forbid, GPA, whatever you want.

So, I've written sort to take as an argument the ordering. I've used this funny thing called lambda, which you don't actually have to be responsible for. You're never going to, probably, need to use it in this course. But it's a way to dynamically build a function on the fly.

The function I've built is I've said the default value of LT is x less than y. Lambda x, lambda xy says x and y are the parameters to a function. And the body of the function is simply return the value x less than y. All right? Nothing very exciting there.

What is exciting is having a function as an argument. And that is something that you'll be doing in future problem sets. Because it's one of the very powerful and most useful features in Python, is using functional arguments.

Right. Having got past that, what we see is we first say if the length of L is less than 2-- that's my threshold-- then I'm just going to return L, actually a copy of L.

Otherwise, I'm going to find roughly the middle of L.

Then I'm going to call sort recursively with the part to the left of the middle and the part to the right of the middle, and then merge them. So I'm going to go all the way down until I get to list of length 1, and then bubble all the way back up, merging as I go.

So, we can see that the depth of the recursion will be log(n), as observed before. This is exactly what we looked at when we looked at binary search. How many times can you divide something in half -- log(n) times? And each recursion we're going to call merge. So, this is consistent with the notion that the complexity of the overall algorithm is nlog(n).

Let's run it. And I'm going to print as we go what's getting merged. Get rid of this one. This was our selection sort. We already looked at that. Yeah.

So what we'll see here is the first example, I was just sorting a list of integers. Maybe we'll look at that all by itself. I didn't pass it in the second argument, so it used the default less than.

It was first merge 4 and 5. Then it had to merge 35 with 4 and 5, then 29 with 17, 58 and 0. And then the longer list, 1729 with 058, 04535 with 0172958. And then we were done. So, indeed it did a logarithmic number of merges.

The next piece of code, I'm taking advantage of the fact that this function can sort lists of different kinds. And I'm calling it now with the list of floats. And I am passing in the second argument, which is going to be-- well, let's for fun, I wonder what happens if I make this greater than. Let's see what we get.

Now you note, it's sorted it in the other order. Because I passed in the ordering that said I want to use a different comparison then less than, I want to use greater than. So the same code did the sort the other way.

I can do more interesting things. So, here I'm assuming I have a list of names. And I've written two ordering functions myself, one that first compares the last names and then the first names. And a different one that compares the first names and then the last names. And we can look at those. Just to avoid cluttering up the screen, let me get rid of this.

What we can see is we got-- we did the same way of dividing things initially, but now we got different orderings. So, if we look at the first ordering I used, we start with Giselle Brady and then Tom Brady and then Chancellor Grimson, et cetera. And if we do the second ordering, we see, among other things, you have me between Giselle and Tom. Not a bad outcome from my perspective.

But again, a lot of flexibility. By using this functional argument, I can define whatever functions I want, and using the same sort, get lots of different code. And you will discover that in fact the built in sort of Python has this kind of flexibility.

You will also find, as you write your own programs, increasingly you'll want to use functions as arguments. Because it allows you to write a lot less code to accomplish the same tasks.