[SQUEAKING] [RUSTLING] [CLICKING]

JUSTIN: OK. So it's a pleasure to see all of you guys. I'm Justin. I'm your third instructor for 6.006. This is my first time with this course. Although, of course, this is material that we all know and love in the computer science department. I'll admit, I find the prospect of teaching sorting to 400 people all at once is mildly, low key terrifying. But we're going to give it a shot. And hopefully, that will subside as the lecture goes on today, all right?

So we're going to pick up where we left off in our last lecture and continue on with a similar theme that we're going to see throughout our algorithms class here in 6.006. I think Jason and colleagues have done a really great job of organizing this class around some interesting themes. And so I thought I'd start with just a tiny bit of review of some key vocabulary words.

Incidentally, typically, I teach the intro graphics class, the geometry course. And last year, I got feedback that said I have serial killer handwriting. I'm not 100% sure what that means. But we're going to use the slides a tiny bit more than normal, just to make sure you guys can read. And when I'm writing on the board, at any point, if you can't tell what I wrote, it's definitely me and not you. So just let me know.

But in any event, in 6.006, all the way back in our lecture 1-- I know that was a long time ago-- we introduced two big keywords that are closely related, but not precisely the same. Hopefully, I've gotten this right. But roughly, there's a theme here which is that there's an object called an interface, which is just a program specification. It's just telling us that there's a collection of operations that we want to implement.

So for example, a set, as we're going to see today, is like a big pile of things. And behind the scenes, how I choose to implement it can affect the runtime and how efficient my set is. But the actual way that I interact with it is the same, whether I use an unsorted array, a sorted array, what have you.

On the other hand, what happens behind the scenes is something called a data structure, which is a way to actually, in some sense, implement an interface. So this is the object that on my computer is actually storing the information and implementing the set of operations that I've laid out in my interface. And so this sort of distinction, I think, is a critical theme in this course because, for instance, in the first couple weeks, we're going to talk about many different ways to implement a set. I'm going to see that there's a bunch of tradeoffs. Some of them are really fast for certain operations and slow for others.

And so essentially, we have two different decisions to make when we choose an algorithm. One is making sure that the interface is correct for the problem that we're concerned with. And the other is choosing an appropriate data structure whose efficiency, and memory usage, and so on aligns with the priorities that we have for the application we have in mind. So hopefully, this high level theme makes sense. And really, spiritually, I think this is the main message to get out of this course in the first couple of weeks, even though these O's, and thetas, and so on are easy to lose the forest through the trees.

In any event, today, in our lecture, we're concerned with one particular interface, which is called a set. A set is exactly what it sounds like. It's a big pile of things. And so a set interface is like an object that just you can keep adding things to it. And then querying inside of my set, is this object here? Can I find it? And then maybe I associate with my objects in my set different information.

So for example, maybe I have a set which represents all the students in our classroom today. Yeah, and all of you guys are associated with your student ID, which I believe at MIT is a number, which has less than sign, which is convenient. So we can sort all of you guys. And that might be the key that's associated to every object in the room.

And so when I'm searching for students, maybe I enter in the student number. And then I want to ask my set, does this number exist in the set of students that are in 6.006? And if it does, then I can pull that student back. And then associated with that object is a bunch of other information that I'm not using to search-- so for instance, your name, your-- I don't know-- your social security number, your credit card number, all the other stuff that I need to have a more interesting profession.

So in any event, let's fill in the details of our set interface a little bit more. So our set is a container. It contains all of the students in this classroom, in some virtual sense at least. And so to build up our set, of course, we need an operation that takes some iterable object A and builds a set out of it. So in other words, I have all the students in this classroom represented maybe in some other fashion. And I have to insert them all into my set. I can also ask my set for how much stuff is in it. Personally, I would call that size. But length is cool, too.

And then of course, there are a lot of different ways that we can interact with our set. So for instance, we could say, is this student taking 6.006? So in set language, one way to understand that is to say that the key-- each person in this classroom is associated with a key. Does that key k exist in my set? In which case, I'll call this find function, which will give me back the item with key k or maybe null or something if it doesn't exist.

Maybe I can delete an object from my set or insert it. Notice that these are dynamic operations, meaning that they actually edit what's inside of my set. And then finally, there are all kinds of different operations that I might want to do to interact with my set beyond is this thing inside of it.

So for instance, so for the student ID example, probably finding the minimum ID number in a class isn't a terribly exciting exercise. But maybe I'm trying to find the student who's been at MIT the longest. And so that would be a reasonable heuristic. I actually have no idea whether MIT student IDs are assigned linearly or not. But in any event, I could find the smallest key, the largest key, and so on in my set. And these are all reasonable operations to query, where my object is just a thing that stores a lot of different entities inside it.

Now, this description here-- notice that I've labeled this as a set interface. This is not a set data structure. And the way to remember that is that I haven't told you how I've actually implemented this. I haven't told you that I'm going to behind the scenes have an array of information, and look inside of it, and that's how I'm going to implement find min or find max with a for loop or whatever. All I'm telling you is that a set is a thing that implements these operations. And behind the scenes, my computer does what it does.

Now, it might sound abstract. But it's more or less what you guys do when you write code in Python. I think in Python what we're calling a set is maybe a dictionary. I'm a Matlab Coder. I'm sorry. I'm a numerical analysis kind of guy. But essentially, one of the beautiful things about coding in these high level programming languages is that they take care of these ugly details. And what you're left with is just the high level interfacing with this object that you need at the end of the day.

So of course, in today's lecture, now that we set out our goal, which is to fill in-- if I wanted to write code for a set, how could I do it? Now, of course, our goal is to give different data structures that implement these, and then understand them in terms of their efficiency, data storage, correctness, all that good stuff. So before we get into all these ugly details, let me pause for a second. Are there any questions about this basic interface? You all should feel free to stop me any time because this is going to be hella boring if you're not getting the first slide or two. Yes?

AUDIENCE: Can you explain how [INAUDIBLE].

JUSTIN: That's a good question. So the question was, what exactly is this insert operation doing? That's why working on the analogy of the students in this classroom is a reasonable one. So I'm going to build up an object, which is a student. So in this lecture notes, I think we've been consistent. I caught one or two typos. We're going to think of x as the object that contains all of the information.

And then associated with that is one piece, which is called the key. That's where we're going to use a letter k. And that's like your student ID. That's the thing I'm going to use to search. So what the Insert operation does that takes this whole student object x, which includes your ID, your name, your phone number, all that good stuff, and inserts it into the set with the understanding that when I search my set, I'm going to be searching by key.

So when I want to find a student, I have to put in my ID number. Does that makes sense? Cool. Any other questions? That's great. Fabulous.

OK. So now, let's talk about how to actually implement this thing. And thankfully, we're already equipped with at least a very simple way that we can implement a set based on what you've already seen in your previous programming classes or even just in the last two lectures, which is one way to understand a set or to implement it rather would be to just store a giant array of objects that are in my set.

I suppose continuing with the theme of the last two lectures, this is not a space in memory, but rather a metaphorical array, a theoretical array. But it doesn't really matter. And so one way to store my set would be to just store a bunch of x's in no particular order.

Does that make sense? So I have a big piece of memory. Every piece of memory is associated with a different object in my set. Obviously, this is quite easy to build. I just make a big array and dump everything in there. And the question is, is this particularly efficient or a useful way to implement a set?

So for instance, let's say that I have a set of all the students in this classroom. There's some ridiculous number of you guys. So actually, asymptotic efficiency maybe actually matters a little bit. And I want to query, does this student exist in my class? Is Erik Demaine taking 6.006? The answer is no, I think. Teaching, taking? I don't know. But in any event, how do I implement it if my set is unordered? We'll think about it for a second. Yeah?

AUDIENCE: [INAUDIBLE]

JUSTIN: It's actually an interesting suggestion, which is going to anticipate what's happening later in this lecture, which was to sort the set and then binary search. But let's say that actually I only have to do this once. For some reason, I built up a whole set of the people in this classroom. And I just want to know, is Erik Demaine in this class?

So then that algorithm would take n log n time because I've got to sort everybody. And then I have to do binary search, which is maybe log n time. But I claim that, if the only thing I care about is building up my entire set and searching it once, there's actually a faster algorithm. This is going to be needlessly confusing because we're going to see that this is really not the right way to implement it in about 38 seconds. Yes?

AUDIENCE: [INAUDIBLE]

JUSTIN: Yeah. Just iterate from beginning to this array and say, is this guy Erik? No. Is this guy Erik? No. Is this guy Erik? Yes. And then return it. So in the worst case, how long will that algorithm take? Well, in the worst case of really bad luck, your instructor is all the way at the end of the list. So in this case, what is that going to mean? That means that I have to walk along the entire array before I find him. So that algorithm takes order n time.

And so your colleague's intuition that somehow this is quite inefficient is absolutely correct. If I know that I'm going to have to search my array many, many times for different people, then probably it makes sense to do a little bit of work ahead of time, like sorting the list. And then my query is much more efficient. But this is all just to say that an unordered array is a perfectly reasonable way to implement this set interface. And then searching that array it will take linear time every single time I search.

And of course, if you go down your list of all of the different operations you might want to do on a set, you'll see that they all take linear time. So for instance, how do I build myself? Well, I have to reserve n slots in memory. And at least according to our model of computation in this class, that takes order n time. Then I'm going to copy everything into the set.

Similarly, if I want to insert or delete, what do I have to do? Well, I have to reserve memory, stick something inside of there. In the worst case, we saw this amortize argument before, if your set is allowed to grow dynamically. And finally, if I wanted to find the minimum student ID in my classroom, the only algorithm I can have if my list of students isn't sorted is to what? Just iterate over every single student in the class. And if the guy that I'm looking at has a smaller ID than the one that I found, replace it.

Does that make sense to everybody? So basically, everything you can do in a set you can implement-- and I think all of you guys are more than qualified to implement-- as an unordered array. It's just going to be slow. Yes?

AUDIENCE: [INAUDIBLE]

JUSTIN: Yeah, that's right. So actually, I don't know in this class. I guess, the interface and the way that we've described it here is dynamic. We can just keep adding stuff to it. In that case, remember this amortized argument from Erik's lecture says that on average that it will take order n time.

AUDIENCE: [INAUDIBLE]

JUSTIN: What was that?

AUDIENCE: [INAUDIBLE]

JUSTIN: Oh, that's true. That's an even better-- sorry. Even if it weren't dynamic. If I wanted to replace an existing key-- like, for some reason, two students had the same ID. This is a terrible analogy. I'm sorry. But in any event, if I wanted to replace an object with a new one, well, what would I have to do? I'd have to search for that object first, and then replace it. And that search is going to take order n time from our argument before. Thank you.

OK. So in some sense, we're done. We've now implemented the interface. Life is good. And of course, this is the difference between existence and actually caring about the details inside of this thing. We've shown that one can implement a set. But it's not a terribly efficient way to do it by just storing a big, hot mess, disorganized list of numbers without any order.

So instead of that, conveniently, our colleague in the front row here has already suggested a different data structure, which is to store our set not as just a disorganized array in any arbitrary order, but rather to keep the items in our set organized by key. So in other words, if I have this array of all of the students in our classroom, the very first element in my array is going to be the student with the smallest ID number, the second is the second smallest number, all the way to the end of the array, which is the student with the biggest ID number. Now, does that mean I want to do arithmetic on student ID numbers? Absolutely not. But it's just a way to impose order on that list so that I can search it very quickly later.

OK. So if I want to fill in the set interface and I have somehow a sorted array of students-- so again, they're organized by student ID number-- then my runtime starts to get a little more interesting. Yeah. So now, insertion, deletion they'd still take the same amount of time. But let's say that I want to find the student with the minimum ID number, this find min function. Well, how could I do it in a sorted array? Keyword is sorted here. Where's the min element of an array? Yes?

AUDIENCE: [INAUDIBLE]

JUSTIN: Yeah. In fact, I can give a moderately faster algorithm, which is just look at the first one. If I want the minimum element of an array and the array is in sorted order, I know that's the first thing. So that's order 1 time to answer that kind of a question. And similarly, if I want the thing with the biggest ID number, I look all the way at the end.

Now, in 6.006-- MIT student class numbers are super confusing to me. In 6.0001, 6.042, you guys already I think learned about binary search and even may have implemented it. So what do we know? If my array is sorted, how long does it take for me to search for any given element? Yes?

AUDIENCE: Log n time.

JUSTIN: Log n time. That's absolutely right because I can cut my array in half. If my key is bigger or smaller, then I look on the left or the right. And so this is a much more efficient means of searching a set. So in particular, 6.006 this year has 400 students. Maybe next year, it has 4,000. And eventually, it's going to have billions.

Then what's going to happen? Well, if I use my unordered array and I have a billion students in this class, what's going to happen? Well, then it's going to take me roughly a billion computations to find any one student in this course, whereas log of a billion is a heck of a lot faster.

On the other hand, I've swept under the rug here, which is how do I actually get a sorted array to begin with. And what we're going to see in today's lecture is that that takes more time than building if I just have a disorganized list. Building a disorganized list is an easy thing to do. You probably all do it at home when you're cleaning house.

But actually, sorting a list of numbers requires a little bit more work. And so this is a great example where there's at least a tiny amount of tradeoff. Now, building my sorted array to represent my set is going to take a little more computation. We're going to see it's n log n time. But then once I've done that step 0, now a lot of these other operations that I typically care about in a set, like searching it for a given key, are going to go a lot faster using binary search.

So this is our basic motivator here. And so now, we've seen the setup interface and two potential data structures. And our goal for the day is going to be to fill in the details of that second one. And since you all have already seen binary search, you've probably also already seen sorting. But in any event, today, we're going to focus mostly on the lower left square here, on just how can I take a disorganized list of objects and put it into sorted order so that I can search for it later.

So in other words, our big problem for lecture today is the second thing here, this sorting. Incidentally, in the next couple of lectures, we're going to see other data sets-- or data structures, rather. Sorry, data sets. I used to teach machine learning class. And we'll see that they have different efficiency operations that we can fill in this table. So we're not done yet. But this is one step forward.

OK. So hopefully, I have ad nauseum justified why one might want to sort things. And indeed, there are a couple of vocabulary words that are worth noting. So one, so remember that your sorting algorithm is pretty straightforward in terms of how you specify it. So in sorting, your input is an array of n numbers. I suppose actually really that we should think of them like keys. It's not going to matter a whole lot.

And our output-- I'm always very concerned that if I write on the board on the back, I have to cover it up-- is going to be sorted array. And we'll call this guy B. We'll call this one A. This classroom is not optimized for short people.

So there's a lot of variations on the basics sorting problem and the different algorithms that are out there. Two vocabulary words are going to highlight really quick-- one is if your sort is destructive, what that means is that rather than reserving some new memory for my sorted array B and then putting a sorted version of A into B, a destructive algorithm is one that just overwrites A with a sorted version of A. Certainly the C++ interface does this. I assume the Python one does, too. I always forget this detail.

In addition to destructive sorts, some sorts are in place, meaning that not only are they destructive, but they also don't use extra memory in the process of sorting. Really, you could imagine a sorting algorithm that has to reserve a bunch of scratch space to do its work, and then put it back into A.

For instance, the world's dumbest destructive sort might be to call your non-destructive and then copy it back into A. But that would require order n space to do. So if my algorithm additionally has the property that it doesn't reserve any extra space, at least up to a constant, then we call that in place. OK. So those are our basic vocabulary words. And they're ways to understand the differences between different sorting algorithms. Yes?

AUDIENCE: [INAUDIBLE]

JUSTIN: Why do they end up using extra O(1) space? Oh yeah, sure. Any time I just make a temporary variable like a loop counter, that's going to count toward that order 1. But the important thing is that the number of variables I need doesn't scale in the length of the list.

OK. So I present to you the beginning and end of our sorting lecture, which is the world's simplest sorting algorithm. I call it permutation sort. I think it's very easy to prove correctness for this particular technique. So in permutation sort, what can I do? Well, I know that if I have an input that's a list of numbers, there exists a permutation of that list of numbers that is sorted by definition because a sort is a permutation of your original list.

So what's a very simple sorting algorithm? Well, list every possible permutation, and then just double check which one's in the right order. So there's two key pieces to this particular technique, if we want to analyze it. I don't see a reason to belabor it too much. But one is that we have to enumerate the permutations. Now, if I have a list of n numbers, how many different permutations of n numbers are there? Yes?

AUDIENCE: n factorial.

JUSTIN: n factorial. So just by virtue of calling this permutation's function, I know that I incur at least n factorial time. It might be worse. It might be that like actually listing permutations takes a lot of time for some reason, like every permutation itself takes order n time. But at the very least, each one of these things looks like n factorial. I warned you my handwriting is terrible. So that's what this omega thing is doing, if I recall properly.

And then secondarily, well, we've got to check if that particular permutation is sorted. How are we going to do that? There's a very easy way to check if a list is sorted. I'm going to do maybe for i equals 1 to n minus 1. Notice not a Python coder. It's going to look different. Then check, is Bi less than or equal to Bi plus 1?

And so if this relationship is true for every single i-- that's supposed to be a question mark. This was less than or equal to with a question mark over it. There's my special notation. So if I get all the way to the end of this for loop and this is true everywhere, then my list is sorted and life is good.

So how long does this algorithm take? Well, it's staring you right in the face because you have an algorithm, which is looping from 1 to n minus 1. So this step incurs order n time because theta of n time because it's got to go all the way to the end of the list. So when I put these things together, permutation sort-- well, remember that this check if sorted happens for every single permutation.

So at the end of the day, our algorithm takes at least n factorial times n time. It's a great example of something that's even worse than n factorial, which somehow in my head is like the worst possible algorithm. So do you think that Python implements permutation sort? I certainly hope not. Yes?

AUDIENCE: [INAUDIBLE]

JUSTIN: Right. So the question was, why is it omega and not big O? Which is a fabulous question in this course. So here's the basic issue. I haven't given you an algorithm for how to compute the set of permutations for a list of numbers. I just called some magic function that I made up.

But I know that that algorithm takes at least n factorial time in some sense. Or if nothing else, the list of permutations is n factorial big because that's all the stuff has to compute. So I haven't told you how to solve this problem. But I'm convinced that it's at least this amount of time.

So remember that omega means lower bound. So when I put it all together, in some sense-- OK, this isn't satisfying in the sense that I didn't give you precisely the runtime of this algorithm. But hopefully, I've convinced you that it's super useless. Yeah, OK. Any other questions about that?

But great. So if we go back to our table for the set interface, well, in some sense, if we implemented it using this goofy algorithm, then the lower left entry in our table would be n factorial times n, which wouldn't be so hot. But notice that actually all the rest of our operations are now quite efficient. I can use binary search. I just obtained the algorithm that-- rather, I obtained the sorted array in a funny fashion.

OK. So let's fill in some more interesting algorithms. As usual, I'm talking too much. And I'm nervous about the time. But we can skip one of them if we need to. So how many of us have seen selection sort before? I see your hand. But we're going to defer for a little bit. I'm sorry?

AUDIENCE: [INAUDIBLE]

JUSTIN: That's fabulous. Why don't we defer to the end of lecture? And we'll do it then. OK. So the first algorithm that we'll talk about for sorting, which is somewhat sensible, is something called selection sort. Selection sort is exactly what it sounds like. So let's say that we have a list of-- whoops, my laptop and the screen are not agreeing.

OK. Let's say I have a list of numbers-- 8, 2, 4, 9, 3. There's a message that Jason I think is sending me in the course notes. But I haven't figured it out. But in any event, I want to sort this list of numbers. Here's a simple algorithm for how to do it, which is I can find the biggest number in this whole list and stick it at the end.

So in this case, what's the biggest number in this list everybody? 9. Good. See, this is why you go to MIT. So I'm going to take that 9. I find it. And then swap it out with the 3, which is at the end.

And now, what's my inductive hypothesis? Well, in some sense, it's that everything to the right of this little red line that I've drawn here is in sorted order, in this case because there's only one thing. So now, what am I going to do? I'm going to look to the left of the red line, find the next biggest thing. What's that? Come on.

AUDIENCE: 8.

JUSTIN: There we go. Yeah, wake up. OK. So right, the next biggest one is the 8. So we're going to swap it with the 3, put it at the end, and so on. I think you guys could all finish this off. I suppose there should be one last line here where everything is green and we're happy. But in some sense, we're pretty sure that an array of one item is in sorted order.

And so essentially, from a high level, what does selection sort do? Well, it just kept choosing the element which was the biggest and swapping it into the back and then iterating. Now, in 6.006, we're going to write selection sort in a way that you might not be familiar with. In some sense, this is not so hard to implement with two for loops. I think you guys could all do this at home. In fact, you may have already.

But in this class, because we're concerned with proving correctness, proving efficiency, all that good stuff, we're going to write it in kind of a funny way, which is recursive. Now, I can't emphasize strongly enough how little you guys should implement this at home. This is mostly a theoretical version of selection sort rather than one that you would actually want to write in code because there's obviously a much better way to do it. And you'll see that in your recitation this week, I believe.

But in terms of analysis, there's a nice, easy way to write it down. So we're going to take the selection sort algorithm. And we're going to divide it into two chunks. One of them is find me the biggest thing in the first k elements of my array. I shouldn't use k because that means key. The first i elements of my array.

And the next one is to swap it into place and then sort everything to the left. That's the two pieces here. So let's write that down. So what did I do? Well, in some sense, in step 1 here, I found the biggest with index less than or equal to i. So I started at the end of the list, and then moved backward.

And then step 2 was to swap that into place. Notice when I say swap-- so for instance, when I put the 8 there, well, I had to do something with that 3. So I just put it where the 8 used to be. And then finally, well, am I done? No, I just put the biggest thing at the end of my array. So now, I have to sort from index 1 to i minus 1 because now I know that the last guy is in sorted order. I see you. I'll turn it over to you in just a sec. Yes?

AUDIENCE: [INAUDIBLE]

JUSTIN: You can't read the handwriting?

AUDIENCE: [INAUDIBLE]

JUSTIN: This is index less than or equal to i. Great question. I warned you. It's going to be a problem. So let's do step 1 first. So I'm going to put code on the board. And then we're going to fill in the details. Erik is posting on Facebook. I'm going to turn that feature off on my watch later. So right, let's implement this helper function here. This is something we're going to call prefix max. And this is going to find me the biggest element of array between index 0 and index i inclusive, I believe. Yeah?

AUDIENCE: [INAUDIBLE]

JUSTIN: Well, here's an interesting observation, really a deep one, which is that the biggest element from 0 to i-- that's an i, sorry. There's two cases. Either it's at index i, meaning I have the first 10 elements of my right-- either it is element number 10 or what's the other case? It ain't, Yeah?

In other words, it has index less than i. This is a tautology, rate? Either the biggest thing is at this index or it's not. In which case, it has to be to the left. Does that makes sense? So this gives us a really simple algorithm for finding the biggest element in the array between index 0 and index i, which is what I've shown you on the screen here. I'd write it on the board. But I am a slow writer and already low on time.

And so essentially, what did I implement? Well, I found the biggest element between index 0 and index i minus 1. So let's say that I have an array-- I forget the sequence of numbers-- 8, 3, 5, 7, 9. That'll do it. And so like I give a pointer here, which is i. And the very first thing that I do is I compute the biggest number all the way to the left of this stuff. In this case, that is?

AUDIENCE: 8.

JUSTIN: 8. There we go. Now, I look at the very last element of my array, which is-- 9. You're killing me today, guys. And then what do I return? Well, I want the biggest one between 0 and index i. So in this case, I return the 9. Does that make sense?

So I know Jerry Cain at Stanford likes to talk about the recursive leap of faith that happens. Another term for this is induction. So we want to prove that our algorithm works. Well, what do we have to do? We have to show that when I call this function, it gives me the max of my array between index 0 and index i for all i.

So let's maybe do this inductive proof a little bit carefully. And then the rest, we'll be sloppy about it. So the base case is i equals 0. Well, in this case, there's only one element in my array. So it's pretty clear that it's the max.

And now, we have to do our inductive step, which means that if I call prefix max with i minus 1, I really do get the max of my array between 0 and index i minus 1. And then really, I can just look at my very deep statement, which is that either my object is at the end of the array or it's not.

And this is precisely what we need to justify the inductive step. Essentially, there are two cases. Either the biggest element of my arrays the last one or it's not. We already, by our inductive hypothesis, have argued that our code can find the biggest element between index 0 and index i minus 1. So as long as we take the max of that and the very last guy, we're in good shape. So this is our very informal proof of correctness.

OK. So now, we have to justify runtime for this algorithm. And that's actually not 100% obvious from the way I've written it here. There's no for loop. But what do I do? Well, in some sense, if my run time is a function s, well, for one thing, if my array has one element in it, well, my run time might be 7, might be 23. But at the end of the day, it only does one thing. It just returns i.

So in other words, it's theta of 1. This isn't terribly insightful. But what else do we know? Well, when I call my function, I call it recursively on one smaller index. And then I do a constant amount of work. So I know that s of n is equal to s of n minus 1 plus theta of 1. I do a little bit of extra computation on top of that. Can anybody guess what this total runtime is going to be? Yes?

AUDIENCE: [INAUDIBLE]

JUSTIN: Yeah, order n. So let's say that we hypothesize that this takes n time. You can see that because at step n we call n minus 1, we call it minus 2, and so on, all the way down to 1. If we want to prove this, one of the ways that we-- I think, in theory, you guys have learned in the past-- and you're going to cover it in recitation-- is a technique called substitution. What we do is we're going to look at this relationship. And we're going to hypothesize that we think s of n maybe look something like cn for some constant c that doesn't depend on n.

Then all we have to do is double check that that relationship is consistent with our inductive hypothesis, or rather just as a recursive function. And if it is, then we're in good shape. So in this case, well, what do I know? I've guessed that s of n is theta of n. In particular, if I plug into this recursive relationship here, on the left-hand side, I'm going to get cn. On the right-hand side, I'm going to get c n minus 1 plus theta of 1. We just have to make sure that this is an OK equal sign.

So what can I do? I can subtract cn from both sides, maybe put that 1 on the other side here. Then we get the c equals big I of 1. c is, of course, a constant. So we're in good shape. My undergrad algorithms professor told me never to write a victory mark at the end of a proof. You have to do a little square. But he's not here.

So now, I see you. But we're a little low on time. So we'll save it for the lecture.

OK. So if we want to implement the selection sort algorithm, well, what do we do? Well, we're going to think of i as the index of that red line that I was showing you before. Everything beyond i is already sorted. So in selection sort, the first thing I'm going to do is find the max element between 0 and i. And then I'm going to swap it into place.

So this is just a code version of the technique we've already talked about. Hopefully, this makes sense. So you find the biggest element between 0 and index i. That's what we're going to call j here. I swap that with the one in index i. That's step 2. And then step 3 is I still have to sort everything to the left of index i and that's that recursive call.

So if I want to justify the runtime of this particular technique, well, now let's call that t for time. Well, what do I do? Well, for one, I call selection sort with index i minus 1. So that incurs time that looks like this. But I also call that prefix max function. And how much time does that take? That takes order n time.

So at the end of the day, I have some relationship that looks like this. Does that makes sense? So by the way, notice that this order n swallowed up the order 1 computations that I had to do to swap and so on.

So remember, there's this nice relationship, which you probably learned in your combinatorics class, which is that 1 plus 2 plus dot, dot, dot plus n. OK. I can never remember exactly the formula. But I'm pretty sure that it looks like n squared. So based on that and taking a look at this recursive thing, which is essentially doing exactly that-- n plus n minus 1 plus n minus 2, and so on-- I might hypothesize that this thing is really order n squared.

So if I'm going to do that, then again if I want to use the same technique for proof, I have to plug this relationship in, and then double check that is consistent. So maybe I hypothesize that t of n equals cn squared. In which case, I plug it in here. I have cn squared equals with a question mark over it cn minus 1 squared plus big O or even theta n here. So if I expand the square, notice I'm going to get c times n squared plus a bunch of linear stuff.

This is really cn squared-- I should be careful with that-- minus 2 cn plus c plus theta of n. Notice that there's a cn squared on both sides of this equation. They go away. And what I'm left with is a nice, consistent formula that theta of n equals 2 cn minus c. And indeed, this is an order n expression. So there's order in the universe. Life is good. Yeah, this is the substitution method. And again, I think you'll cover it more in your recitation.

So what have we done? We have derived the selection sort. We've checked that it runs in n squared time. And by this nice, inductive strategy, we know that it's correct. So life is pretty good.

Unfortunately, I promised for you guys on the slides that sorting really takes n log n time. And this is an order n squared algorithm. So we're not quite done yet. I'm way over time. So we're going to skip a different algorithm, which is called insertion sort, also runs on n time.

Essentially, insertion sort runs in the reverse order. I'm going to sort everything to the left, and then insert a new object, whereas, in selection, I'm going to choose the biggest object and then sort everything to the left. But I'll let you guys piece through that at home. It's essentially the same argument.

And instead, we should jump to an algorithm that actually matters, which is something called merge sort. How many of us have encountered merge sort before? Fabulous. Good. So then I'm done. So let's say that I have a list. Now, I'm sending a message back to Jason. I made this one up last night.

So I have 7, 1, 5, 6, 2, 4, 9, 3. This is not in sorted order. But I can make a very deep observation, which is that every number by itself is in sorted order if I think of it as an array of length 1. It's really deep, like deep learning deep.

So now, what can I do? Well, I could take every pair of numbers, draw a little red box. Well, now, they're not in sorted order any more inside of the red boxes. So I'm going to sort inside of every box. In this case, it's not too exciting because it's just pairs. And now, they're in sorted order because they said they were.

Now, I'm going to keep doubling the size of my boxes. So now, let's say I have box of length 4. What do I know about the left and right-hand sides of the dotted lines here? On the two sides of the dotted lines, the array is in sorted order. There's a 1 and then a 7. Those are in sorted order, 5 and a 6. That's because, in the previous step, I sorted every pair.

So when I merge these two sides together, I have an additional useful piece of information, namely that the two sides of the dotted line are already in sorted order. That's going to be our basic inductive step here. So in this case, I merge the two sides. I get 1, 5, 6, 7, and 2, 3, 4, 9. Then finally, I put these two things together. And I have to sort these two. I have to merge these two sorted lists. But they're in sorted order. And that's going to give me a big advantage because-- oops, I lost my chalk.

I suppose I've got space on this board here. Oh no. So if I want to merge 1, 5, 6, 7 and 2, 3, 4, 9, there's a nice, clever technique that we can do that's going to take just linear time. Jason tells me it's the two finger algorithm. I think that's a cute analogy here. So here are my two fingers. They're going to point at the end of the list. And I'm going to construct the merged array backwards.

So how many elements are in my merged array, if I'm merging two things of length 4? I don't ask you guys hard questions. It's 8, yeah? 4 plus 4. 8, yeah?

So what do I know? I know that my merge array-- 5, 6, 7-- has eight elements. And now, I'm going to have two fingers at the end of my array. Which one should I put at the end of the merged guy? The 7 of the 9?

AUDIENCE: The 9

JUSTIN: The 9. Right, thank you. So now, I can move my lower finger to the left because I've already added that. Notice that I never need to look to the left of where my finger is because they're already in sorted order. Now what should I add, the 4 or the 7?

AUDIENCE: 7.

JUSTIN: The 7. And so on, dot, dot, dot, yeah? So that's going to be the basic idea of the merge sort. I'm going to take two sorted lists. And I'm going to make a new sorted list, which is twice as long, by using two fingers and moving from the and backward.

So that's the basic intuition here. Indeed, there's our sorted list. It's stressing me out that there's no eight. I need the power of 2. So I think merge sort, we're going to present it in a backward way from the previous one, where I'm going to give you the high level algorithm. And then actually, the headache is that merging step, which I have four minutes for. And I apologize for it.

So what does the merger sort do? Well, it computes an index c, which is the middle of my array. And it's going to make a recursive call which is sort the left, which is everything between index A and index C. And then sort everything on the right, which is everything from index C to index B. I know this is confusing because usually letters appear in order. But C, if you think of as standing for center, then it makes sense like.

Here's my array. I'm going to choose an index right in the middle. I've done myself a disservice by not using a power of 2. But that's OK. I'm going to say sort everything to the left of the dotted line first. Sort everything to the right of the dotted line second. Now, I have two sorted lists on the two sides of the dotted line. And then I'm going to use my two fingers to put them together.

So that's what this is implementing here. See, there's two recursive calls-- sort from A to C, and then sort from C to B. Oops, I didn't actually label this. So this is A, C, B. And then I've got to call merge.

Now, our implementation of merge-- well, we can also do this in a recursive fashion. But personally, I find this a little complicated. I'm going to admit. But here's the basic idea here, which I'm now rushing. So I'm going to think of my upper finger as finger i and my lower finger as finger j. Does that makes sense?

So I have two sorted lists. So maybe like that. I don't know, 1, 3, 5, 7. And then I have a second sorted list here, which is maybe 2, 4, 6, 72, as one does. Then I'm going to have one pointer like this, which is i, and a pointer down here, which is j. And my goal is to construct an array A with a bunch of elements in it.

And the way that I'm going to do it is I'm going to use exactly the same kind of recursive argument, that I can either have the biggest element of my be the last element of the first guy or be the last element of the second one. So here's going to be our recursive call. And in addition to that, for convenience, we'll have a third index, which is B, which is pointing to this thing inside of my sorted array that I'm currently processing Yeah? It's going to start at A, go to B.

Incidentally, I see a lot of people taking photos of the slides. These are just copy pasted from the notes. OK. So in this case, what should I put in B for my two arrays? I have 1, 3, 5, 7; 2, 4, 6, 72. 72, yeah? Great. So now, what am I going to do? I'm just going to call the merge function. But I'm going to decrement B because now I'm happy with that last element.

And in addition to that, I'm going to decrement j because I already used it up. And so that's our recursive call here. It's saying, if j is less than or equal to 0-- so in other words, I have an element to use in one of the lists of the other. And maybe the left one is bigger than the right one. That's our first case. That does not apply in this example here.

Well, then I should make the last element of a from the first list and then recurse with one fewer element i, and similarly the reverse case for j. So if we do our runtime in two minutes or less-- bare with me guys-- well, what is this merge function going to do? Well, in some sense, there's two branches. There's an if statement with two pieces. But both of those pieces call merge with one fewer piece in it.

So in some sense, we have s of n equals s of n minus 1 plus theta of 1, which we already know from our previous proof means that s of n is equal to theta of n. So in other words, it takes linear time to merge. It makes sense intuitively because essentially you're touching every one of these things once with your two fingers.

And now, probably the hardest part of the lecture, which I left zero time for, is deriving the runtime for the actual merge sort algorithm. And what does that look like? Well, that one's a little bit trickier because, of course, I call the merge sort algorithm twice, each time on a list that's half the size. In this class, we're going to assume that our list is always a power of 2 in its length. Otherwise, this analysis is a itty bitty bit more of a headache.

So first of all, how long does it take to sort an array of length 1? I am not going to ask hard questions. Everybody? Yeah, it's just 1, right? Because there's nothing to do. An array of length 1 has one element and it's sorted. It's also the biggest element and the smallest element.

And now, what does our algorithm do? Well, it makes two recursive calls on lists that are half the length. And then it calls that merge function. And we know that the merge function takes theta of n time. Does that make sense?

So one thing we might do, because we have some intuition from your 6042 course, is that we think that this thing is order n log n because it makes the two recursive calls. And then it puts them together. And let's double check that that's true really quick using the substitution method.

So in particular, on the left-hand side here, maybe I have cn log n. Now, I have 2 c. Well, I have to put an n over 2 log n over 2 plus theta of n. And I want to double check that this expression is consistent. I've got about a foot to do it in.

So remember-- let's see. If we use our favorite identities from high school class that you probably forgot, remember that log of 2 things divided by each other is the difference of the logs. So this is really 2. OK. 2 divided by 2 is 1. So this is c times n times log n minus log of 2 plus theta n.

I'm already out of time. But notice that there's a c n log n on the right-hand side. There's a c n log n on the left-hand side. So those two things go away. And what am I going to be left with? I'm going to be left with theta of n equals cn log of 2. Notice that c and log 2 are both constants.

We have a theta event on the left-hand side. So there's order in the universe. And we've derived our runtime. So I know I rest a little bit through merge sort. I'm sure that Erik and Jason can review this a little bit next time. But with that, we'll see you, what? Thursday and Friday. And it's been a pleasure to talk to you all.