[SQUEAKING]

[RUSTLING]

[CLICKING]

PROFESSOR: All right, folks. I think it's about time to get started. Thank you guys for making it to our problem session and I guess talking to many of you guys virtually, given the number of people in the room. But that's perfectly fine.

Right. So we're in problem session number two for 6.006. Hopefully you're all in the right place. I'm Justin. I was looking at a little spreadsheet of who's teaching what, and I think you guys are stuck with me in a lot of these problem sessions, which actually, I think in terms of hours of video, means that you're stuck listening to my voice the most. So sorry in advance.

But in any event, it's always fun to do this kind of thing. And it gave me the opportunity to do homework, which I have not done in a very long time. So last night, I worked through all these problems and came up with my own hacked-together answers for these, which thankfully more or less agreed with what was in the answer key, which I have and you don't.

I think that we release on-- so there's a bit of a known bug in 6.006, which is that this instructor has horrible handwriting. I'm working on it. I went to the classroom and practiced the other day. But in the meantime, there are horrible handwritten notes that are posted in the learning module, at least for the duration of the talk today. And we'll decide whether we want to leave them up or not because they're not super exciting.

Just so you can make sure you can follow along with that, I'm writing on the board. And as with before, if there's ever anything that you can't read, just stop me. And I'll gladly fix.

Cool. So right, so hopefully, does everybody have a copy of the problems? It'll look a little different from this. The green stuff is the answers. I get those, and you don't. Or sorry, I always ask this question when I'm teaching, and you never get an answer. Does anybody not have a copy of the problems and the problem session? Fabulous. OK. If you don't, there's a pile of handouts up here. Here, I'll grab one for you.

Yeah. Great. So the basic goal of the problem sessions is to go over a bunch of example problems, which I believe are mostly lifted from past years' homeworks and exams, and work through how we went about thinking about them and a solution or at least just kind of sketch out enough that we're confident you could fill in the blanks of a solution, given our limited time together.

Right. And of course, one of the pleasures of teaching this problem session is this is not normally the class that I teach. So these really are problems I haven't seen before. And indeed, in the middle of the night last night, I woke up and realized my answer to the last problem was totally wrong. So if you feel that way sometimes, the feeling is mutual.

So in any event, there are four problems on your handout that roughly, I think, are in correspondence with the stuff we've taught over the last week or so. So the first problem involves recurrences. The second involves Infinity Stones. The third involves some kind of a queue stack structure. And the fourth involves all kinds of things talking to each other in a way that I got wrong.

So I guess for lack of a more creative thing, we'll just go through these guys in order. And essentially, this is supposed to be an interactive session, although it doesn't really look that way. But since you guys are the people in the room, you have the advantage of being able to stop me the second that I, A, make a mistake or, B, say something that confuses you. And I'm relying on you to do that, OK? Nods. That's our deal today. Some kind of acknowledgment. Noses out of the laptops, maybe, for a second.

Cool. Thank you. See, thumbs up. That's what I'm looking for. OK. All right, folks. Right, so let's start with the problem number one here. So problem number one is solving recurrences, which is our favorite way to torture undergrads in algorithms classes in the first couple of weeks.

And in particular, applying this master theorem. So I thought I'd spent about 82 seconds reviewing master theorem because this is sort of the giant sledgehammer for solving recurrences without understanding why you got the answer that you did.

And then for each of these problems, conveniently, this homework problem asks you to do each big O thing twice, right? Once using master theorem to kind of get you used to just applying this little set of rules, which is sort of the most efficient way to solve recurrences. It just gives you a formula to look at. In addition to that, drawing a tree of computation and counting the number of computations you do.

So the first two are basically straightforward applications of master theorem. The third one requires a little bit of thinking. So naturally, I got stuck and wasted an hour yesterday trying to convince myself the answer was right.

OK. So for a tiny bit of review, I know that you also did this in recitation. But it can't hurt to bring it up again. Remember that master theorem-- I believe that "master" really is just because it's general, right? There's not a guy named Master, but I notice that it was capitalized a lot in the course notes.

But in any event, the basic idea is that it has some recurrence that looks like the following, which is T of n equals aT of n/b plus f of n. So for instance, in merge sort, for a tiny bit of review, remember that we kind of split into two pieces. So both a and b in merge sort would be 2. Yeah.

And the amount of work that I do in the merge step is kind of like n. So that's how to fill in these different constants there. And the question is, asymptotically, what does this function look like? And of course, it's not totally obvious from this formula because I've defined T in terms of itself, right? And that's the sort of annoying part of solving recurrences.

And so master theorem, and I'll let you guys review your section notes about precisely how this gets proved, essentially divides into three cases. And the relevant numbers here are as follows, right? There's an a, a b, and an f. And so the a is kind of like a branching factor, right?

So if you remember your computation tree, like in merge sort, you split into two pieces, right? So that number is kind of like a. It's the number of splits that you make, right? b is the amount that your problem size reduces when you go to each of those leaves. And then finally, f is the amount of work that you do at each node. Yeah, so hopefully that basic language makes sense.

And then essentially what you learn from master theorem is that there's three cases. The first is that f of n looks like-- by the way, in the recitation notes, for some reason, it's kind of written backward. They give the answer to master theorem and then the condition. I'm going to write in the other order.

Right. Looks like the following, which is n to the log base b of a minus epsilon for some positive epsilon. I'm going to try to be very conscientious about reading my writing out loud as I write it down. All right. And in that case, what we find is notice there's something kind of magic here, which is that this is just an upper bound. There's not a theta here. So it's OK if f is below this thing.

But in any event, the conclusion then is that T of n is big theta of n to the log base b of a. It's kind of cool if you think about it. You only have an upper bound for f, but you get a nice theta bound for T, which is pretty cool. The reason for that is essentially the f is sort of insignificant relative to the work of just traversing up and down the tree.

That's case one. Case two-- I'm going to try and leave this on the board as we actually solve these problems-- is the following, which is f of n is big theta of n to the log base b of a multiplied by log to the k-th power of n. This is a super weird form, but for example, it's perfectly kosher to take k equals 0, in which case it starts to look like case one, right?

For some positive k or non-negative k, rather, in which case what we learn is that T of n is theta of n to the log base b of a times log to the k plus 1 of n. You can see why we don't love applying this theorem in this course because I feel just staring at these formulas is totally unenlightening. But it is a giant sledgehammer for solving recurrences quickly.

And then finally, case three-- I'm going to slide the board up, and I'm going to slide it back down because I don't want to bend over. OK-- is the following, which is f of n is omega, right, meaning that it's lower bounded by n to the log base b of a plus epsilon for some epsilon greater than 0.

And we needed a second case. What's it called? af of n over b is less than cfn for some c between 0 and 1. And what's the conclusion there? Then it turns out that the f term sort of dominates, and what we get is that T of n is theta of f of n.

OK. So essentially, this covers three major cases that you see in recurrences. There are other more generic versions of the master theorem out there where essentially, rather than just having one term with the T in it, maybe use more than one term with a T in it. I don't think those are covered in this class.

I always confuse the name of the more general one. I want to say it's Arzela-Ascoli, but I know that that's from functional analysis. [INAUDIBLE] theorem, if you want to Google that, learn more. But in any event, that's going to be enough for most of the recurrences we care about in 006.

So I've at least written our conditions down. And now they're going to sit on the left-hand side while we do three example problems to show how they show up in practice. Are there any questions about what this theorem is telling us about life or how to apply it? Don't all speak at once now. Yes.

AUDIENCE: Sorry, I have a handwriting question.

PROFESSOR: No problem.

AUDIENCE: What's after "some," the last line?

PROFESSOR: Oh, for some c in 0, 1, not inclusive. Fabulous question. Any others? OK. So let's do an example problem here. So let's do part a.

So in part a, they give you a recurrence, which isn't all that different from merge sort or any of the other ones. It looks something like this. T of n is equal to 2T of n/2 plus-- and then they have some other term, which they don't tell you anything about beyond that it's big O of square root of n.

By the way, this might mean that it actually does order 1 work, right? It just has to be upper bounded by a square root of n. That's the only thing that problem's telling you. I'm going to keep kind of driving home that point because I think everybody, including myself, gets really sloppy about, like, when is it O? When is it theta? When's it omega?

And so every single time you write one of those letters down, you should step back 50 feet and look at it and think, like, did I do that right? Or did I just write a Greek letter, right? And make sure that you're actually thinking through it logically.

OK. So when I apply master theorem in my everyday life, what I like to do is to say, OK, somehow, the really key quantity in master theorem is this dude, right, n to the log base b of a. He just keeps showing up in all these different cases.

So I might as well figure what that is for my recurrence, yeah, and then plug it in and check which case I'm in. Does that make sense? So let's do that.

So first of all, what is a for this particular recurrence?

AUDIENCE: 2.

PROFESSOR: 2. What's b? You're killing me, guys. On 3. 1, 2, 3.

AUDIENCE: 2.

PROFESSOR: OK. The enthusiasm is overwhelming. OK, and what do we know about the function f?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, it's big O of root n. We don't know that it equals root n. But we do know that it's at most upper bounded by something that kind of looks like root n.

Eric gave me a whole lesson on this chalk, and I'm still failing at it. OK. So now that gives us everything we need to compute n to the log base b of a. We're going to do this one real slowly, and then we're going to do the next one a little more fastly, quickly.

So n to the log base b of a. Well, that's n to the log base 2 of 2. Anybody have an idea? What's the log base 2 of 2?

AUDIENCE: 1.

PROFESSOR: 1. OK, I'll take it this time. OK. Right. And so then what does master theorem tell you? Well, in some sense, I want to know what f is, which in this case looks like square root of n, compared to what this is, n to the first. Yeah? Now, first of all, which of these two things grows faster? This guy, right?

So what we know is that f is really upper bounded by this n here. "Upper bounded" is not quite the right term because big O allows for some wiggle room, but asymptotically. Hopefully you guys get this concept.

OK, so in particular, let's see. So remember that f of-- let's maybe just write it again. So f of n is big O of square root of n. But let's write that in a suggestive way as n to the 1/2. Yeah.

In particular, this is equal to big O of n to the 1 minus 1/2. It's the beautiful thing about 1/2. Yeah. But what is n to the 1? Well, that's n to the log base b of a, right? That's what we just showed here, right?

So this is really O of n to the log b of a-- that's a complicated way of writing the number 1, right-- minus 1/2. Let's give 1/2 a special name, too, while we're at it. Let's call him epsilon or her, yeah, where we take epsilon equals-- see what I did there?

Now, what does it tell me? There are three different cases of master theorem. And take a look at what I just showed. I showed that f of n is equal to big O of n to the log base b of a minus epsilon for some epsilon that's equal to 1/2.

The beautiful thing about 1/2 is that it is greater than 0. Yeah. So I think we have somewhat laboriously checked that we are in case one. Any dissidents here? Fabulous. OK. So in that case, master theorem, basically, you're just done, right? So what is the conclusion?

AUDIENCE: T of n is theta of n.

PROFESSOR: Exactly. So T of n is theta of n to the log base b of a. But we already showed that's equal to n. So we're done. All right. Any questions about how we apply master theorem here?

OK. So now here's the thing about master theorem. Did we learn anything about what this recursive function is doing? No, we just did a bunch of work, plugged in some Greek symbols, and out came a big theta. Yeah.

So what we might want to do instead is to use a method that I'm still learning myself because I'm not used to presenting it this way, but that's OK, which is to actually draw out the tree of computation. Yeah. So let's actually do that.

So let's say that I call my function T of n. So what does T of n do? Well, in some sense, it does work that kind of looks like the square root of n, right, and then it makes two function calls, each of which has n/2 amount of data. Yeah.

So let's draw what this looks like. So the first thing that my function might do is work square root of n much. Going to put a little square root of n there. And now it makes two function calls. And how much work does each of these guys do? So when it calls T, what goes into T? n/2.

AUDIENCE: So square root of n/2.

PROFESSOR: Exactly. So it does square root of n/2 amount of work at each of these two nodes. That make sense?

AUDIENCE: That's determined by the branching number, right?

PROFESSOR: You have to be careful. So the branching number, in this case, it happened to be the same. The next problem we do, they'll be different. So they're both 2 here. The outside 2 refers to the fact that there's two children.

AUDIENCE: Yeah, never mind.

PROFESSOR: And inside 2 refers to the fact that I divided by 2. It's a fabulous question because it's exactly what I got wrong when I did this problem. OK.

So now each of these guys calls one of their children. Yeah. And so again, the data divides by 2. So now I have the square root of n/4, like this. Yeah. If you're wondering, inside of each of these circles, it says square root of n/4. OK.

So let's say that-- so now meanwhile, in our function call tree here, how many nodes are in just the first level at the very top? One, right? Just this dude. Yeah. One node.

How many nodes are in the second level? Two. Or if we want, 2 to the 1 nodes. Here, there's four, yeah, which is 2 squared nodes, and so on.

OK. So now we have some pictorial representation of what's going on inside of our recurrence. If you look at the course notes I wrote out, I wrote out one more layer just for fun and profit. So now let's see how this helps us actually solve our recurrence.

So what do we know? How many levels are in this tree? Well, our algorithm kind of stops when the input to T looks like 1, right? So since this divides by 2 each time, the tree has log base 2 of n levels in the tree. Yeah. And each level does how much work?

So be careful. So each level-- let's call that level l, right? So we'll say l equals 0, l equals 1, and so on. So there's two different things we have to account for when we account for the work in this level. One is the square root of n/2 in a single node. And the other is the fact that there's two different nodes in the level.

So in level l, so the amount of work in level l is equal to the product of those two things. So how many nodes are in level l? 2 to the l, yeah? OK. And how much work does each one of those nodes do?

It's like n/2 to the l. So a different way of writing that is the square root of n times 2 to the minus l, like that. This is just n/2 to the l. Fabulous. So now we have everything we need to actually work out our solution to the recurrence. OK. Yes?

AUDIENCE: What does that say?

PROFESSOR: This is n times 2 to the minus l. I appreciate the [INAUDIBLE]. I'm going to work on this more. And that's why I also share it online. The handwritten notes is-- this is literally a scan of the page that I'm holding in front of me.

OK. So if we want the total amount of work in our tree, right, if we want to account for all of T of n, then what are we going to do? Well, we have to sum starting at level 0. I'm using l instead of i because I'm an analysis person, and i is the square root of minus 1. And I don't like i's in my algorithms.

But in any event, how many total levels are there in my tree? Well, we know that there's log base 2 of n, right? And now we have to sum this quantity for all those levels. 2 to the l, the square root of n times 2 to the minus l, like that.

OK. And then what do we know secretly from the master theorem is that we're suspecting that this is going to be theta of n, right? So that's all we've got to check. OK, so let's do that real fast.

So first of all, notice that this n term doesn't depend on the summand. So I can just pull them out, right? So this is really the square root of n times the sum over l equals 0 to the log 2 of n of what? Oh, and there's a mistake in my notes. Oh, man.

Oh, no, this is--

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh, right, duh. OK. So you have 2 to the l, and you have a square root of 2 to the minus l. So this is really 2 to the minus l/2. So you have l minus l/2. And this quantity is really 2 to the l/2, like that.

Does that make sense? So this is just properties of exponents from high school class. And in fact, I don't like this because it's l/2, and I want to blindly apply a formula without thinking about it. And so of course, this is really the same as the square root of 2 to the l-th power. This is just properties of exponents.

OK. So what is this sum called? Do you recognize this? So there's some constant here. You're taking it to the l-th power, and then you're summing it over l's. This is called a geometric series, right?

So it would be like 1 plus x plus x squared plus x to the third and dot dot dot, x to the, I guess in this case, log base 2 of n. Yeah? So conveniently, there's a geometrics series formula out there. I wrote it down in the handwritten notes, but maybe for now, because I'm moving slowly as always, so we'll skip that part.

And what you could is as follows, that this is the square root of n, right-- that's just this outer part-- times-- and the inside are going to get the square root of 2 to the log base 2 of n plus 1 minus 1 divided by the square root of 2 minus 1. And this is the geometric series formula. So I encourage you guys to go back home and Google that one if you forgot. Yes.

AUDIENCE: Wait, why are we calculating the sum of the series?

PROFESSOR: Because what we're trying to do is figure out the total amount of work in this tree, right? So what we've done so far, we know that a single level in this tree is this value. And now we have to sum over all the levels from 0 to log base 2 of n.

AUDIENCE: OK, and what do we use this number for?

PROFESSOR: Because we're trying to actually approximate or at least bound T of n, right? And T of n is the sum of all these values.

AUDIENCE: Oh, OK. And theta of n is not a sufficient bound.

PROFESSOR: It is a sufficient bound. We just got it from kind of a boring way. This is a second way that we could have proved the same formula.

AUDIENCE: Oh, OK.

PROFESSOR: Yeah. I see a question over here.

AUDIENCE: Can you go over the root n, 2 minus l part? I understand the 2 to the l is the number of nodes second part. Does that work?

PROFESSOR: Oh. OK, this is the number of nodes. This is the quantity inside of the circle. So notice that it's square root of n and then square root of n/2 and then square root of n/4 and then square root of n/8, right? So 2 to the minus l here is like 1/2 and then 1/4 and then 1/8. Yeah. Fabulous. Any other questions? Yes.

AUDIENCE: So if we have log base 2n levels and we started indexing at 0, doesn't it go from l equals 0 to l equals 2n minus 1?

PROFESSOR: Oh, boy. I'm bad at this stuff. No, right? I think this is correct. Let's see. So let's say that I have n equals 2, right? Then what's going to happen?

Yeah, no, this is correct. So as a sanity check, think about n equals 2. So how many levels should it be? It should go from l equals 0 and then l equals 1. Then I'm done because I have T.

And then I think-- yeah, so the log base 2 of 2 is 1. So yeah, this formula checks out. OK, any other questions? These are all great questions. You guys are keeping me honest.

OK. So we have this giant ugly expression here. And so our final job here is just to simplify it. That's it. And then what we're going to find is that this is secretly just theta of n. A little surprising, given that this is kind of ugly. OK.

And by the way, this was just the constants. So this term on the denominator is not going to end up mattering for our asymptotic calculation. OK. Cool. So let's see here.

So remember that the square root of 2 is 2 to the 1/2. Yeah. So I can do a little bit of reshuffling on our formula. And what I'm going to find is that this is exactly the square root of n times 1 over the square root of 2 minus 1. That's just this constant here.

Now I just have to cope with this funny term inside of here. I'm going to do the following. I'm going to say, well, square root of 2 is the same as 2 to the 1/2. So I'm going to write this by moving that 1/2 upstairs here, right?

So what am I going to get at the end of the day is that this is the same as 2 to the log base 2 of n plus 1 and all of that-- oops, plus 1/2, right, because there's the 1/2 exponent that got multiplied by the 1-- minus 1.

So this is just an identical expression to this one. All I've done is move the 1/2 upstairs. And finally, now there's some order in the universe, because what is 2 to the log base 2 of n?

AUDIENCE: n.

PROFESSOR: n, exactly. That's a very complicated way of writing n. So this is equal to the square root of n times 1 over the square root of 2 minus 1, which is just a constant. I see your hand. I'm going to write, and then I'll catch up with you.

And now this whole quantity is n times the square root of 2, right? Oops, wait.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh, I'm sorry. That should be-- right, because we moved the 1/2 into the exponent, and I didn't account for it. So this is the square root of n multiplied by the square root of 2.

AUDIENCE: [INAUDIBLE]

PROFESSOR: What's that?

AUDIENCE: It'd be n/2 to the square root of n.

PROFESSOR: n/2 to the square root of n. No, I think this is right.

AUDIENCE: Oh, sorry.

PROFESSOR: Yeah. You guys are making me nervous. OK, so finally, now we can start to see the big O come out, right? Because now we have the square root of n times itself plus some other stuff, which is obviously going to grow slower than the remaining terms here. So we're good.

If you're wondering, the official solutions are actually incorrect. They, too, get this a little bit wrong. I'm going to fix that tonight, and then we'll post them for you guys.

And this unfortunately is the kind of thing you do a lot of, where you're going to take your tree. You write down this giant tree formula here. You write down a geometric series of some sort and then start bounding terms until you get to the expression you want. So now you can see why master theorem maybe is kind of valuable because it saves you a lot of headache. Yes.

AUDIENCE: When we write up our piece sets and we have to write up trees like this, can we hand draw them and take pictures?

PROFESSOR: Yeah, I don't see why not. They don't have to draw it in Dixie or something.

AUDIENCE: You can take pictures of diagrams as long as the math doesn't [INAUDIBLE].

PROFESSOR: Yeah. I think if you're just writing out your math, taking a photo of it, and then, like, backslash include graphics of your photo-- yeah. But just for the figure, I think that's fine.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah. In fact, when I write research papers, that's how I do it until the final draft. Yeah. OK. So essentially, what did we just do in this problem, which is part a of one problem, is essentially just showed two different ways of solving recurrence, right? One is the sledgehammer that's very efficient but not terribly illustrative.

The other is really working out all of the work that happens at every level of our tree, doing a giant sum, and then just actually deriving the proper formula. And both of these are just different ways to skin the same cat. Is that a weird phrase? I don't know.

OK. So right, so actually, part two, I believe, ends up being easier by a funny fluke even though it looks like it's harder. So let's do that next here. So I'm going to leave master theorem on the left-hand side. I really want to say "master's theorem," but I guess that's not right.

I'm going to leave the master theorem on the left-hand side. Just do the problem over here. I'm going to do this one a little bit quicker because I prefer not to spend the whole session on one problem. OK. But I do think it's worth spending a few minutes going over this theorem and how to apply it carefully because, well, otherwise, you won't. OK.

All right, folks. It's a workout up here. So in part b, we have a version where the branching factor and the amount that the work reduces are not the same. So now we have to be a little bit more careful in the way that we apply master theorem, right?

So now in part b, we have that T of n here is equal to 8T of n/4 plus-- and now in the problem, they write big O of n times the square root of n. We're all grown-ups here. That's the same as big O of n to the 3/2. Yeah. That's just arithmetic.

OK. So first, let's apply the master theorem. Let's see. So what is n to the log base b of a, in this case? Well, what is b? 4. What is a? 8.

Anybody know log base 4 of 8?

AUDIENCE: 3/2.

PROFESSOR: Wow, you're good. Yeah, so this is equal to n to the 3/2. And notice that these two things agree now. Yeah. So which of the three cases of the master theorem are we in? Are we in the first one, where one of these kind of dwarfs the other? No. We're in case two, right, where they both kind of behave similarly.

Yeah. So this is case two. And what is the value of k for case two that's relevant here? Do I need a log factor? No, these two terms are just the same, right? They're both n to the 3/2.

So this is case two with k equals 0. And in particular, immediately, we get that T of n is-- let's see. We're in case two, so we have theta of n to the log base b of a times log to the k plus 1 of n. So you have to account for that term, so it's like that. That make sense, how we applied master theorem in that case? Yes.

AUDIENCE: So how can we apply case two if it's big O and not big theta?

PROFESSOR: How can we apply case two if f of n is big O and not big theta? Well, that's coming from a grad student. So why don't I ping it back to the students in the class rather than answering myself?

AUDIENCE: Can you repeat the question?

PROFESSOR: Sure. So let's say that we just have a big O. We don't have a big theta here. Or at least a simple answer, which maybe is the one that you're looking for.

So what do we know? In that case, we know that f of n might grow more slowly than this function. But it doesn't grow more quickly than this thing. Yeah.

AUDIENCE: Would T of n just be big O instead of big theta?

PROFESSOR: That's exactly right. So you can just replace big theta with big O, and that's perfectly fine. So I don't know if that's the answer you're looking for. But at least you get a loose bound this way. Yeah.

AUDIENCE: You should probably write it big O in the answer then.

PROFESSOR: Oops. Oh, I'm sorry. I see. We're being pedantic. OK. You can just tell me if I made a mistake. It happens. I think it's correct in my notes. It is. OK.

So the reason our colleague brings it up in the back is that I made a slight mistake, which is that this is a big O, not a big theta, right? In which case, the only thing I can draw down here is that this is also a big O. So I made exactly the mistake I told you guys to avoid before. Yes.

AUDIENCE: So then over there, would those also be big O?

PROFESSOR: No. Yeah, now I've managed to confuse you. This is correct. This entire statement of the master theorem, assuming I copied it correctly, is right. But there's a bit of a difference.

Remember, we wanted to apply case two. And look at case two. So there's a theta here. But in the problem, in the homework problem, they were sneaky. And they only said that f of T is big O of something.

So remember, what's the difference between big O and big theta? Well, intuitively, big theta says that my function really does look like this guy. Somehow, it's bounded above and below as I go far enough out.

In big O, there's just a bound above, right? So this is somehow looser. And so the way to apply master theorem in this case is say, well, at least f of n is upper bounded. It looks like this. So the best that I can do is to replace this guy also with an upper bound. Yeah. That's a great question.

AUDIENCE: So the reason you chose two is that [INAUDIBLE] construction of the math is similar to [INAUDIBLE]?

PROFESSOR: Exactly. You got to kind of stand back and squint at it a little bit, make sure that it fits. But certainly, it's the case that the other two parts of this theorem don't apply. So we might as well try to squeeze this part into the right form. All right.

So let's do the tree version of this. And then I think what we're going to do is skip part three for now of this problem because it's mostly a fun kind of vectorial problem rather than something that actually is going to help your understanding of algorithms. And then if we have time at the end, we'll come back to it. OK.

All right. So that was the easy way to solve the problem. Now let's do the painful one where we draw the tree. I shouldn't say that because I think it's something we want to encourage-- the more enlightening version, yeah?

So now at the top of my tree, I do n to the 3/2 work. And now how many children does this guy have?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Careful. You have 1 in 2 chances, either 8 or 4. 8, right? So the outer coefficient is the number of children because it's like the number of function calls you make, right? That's the way to think about it, right?

So there's eight of these guys. OK. And each one of these guys-- I'm only going to do one level of these. Wait, that should not be connected-- does how much work? Well, remember that it's divided by 4 here, right? So it looks like n/4 to the 3/2.

That make sense, how we got this picture from our recurrence? Let me switch to another piece of chalk. OK. Right. So in other words, if I look at level l of my tree, assuming we're going to index from 0 at the top, how many nodes are in level l?

Remember, it branches by 8 each time. So it has 8 to the l nodes. Yeah. And how much work does each do?

Well, in each level, I divide by 4, right? So this is going to look like n times 1/4 to the l to the 3/2. Right. Where am I?

There we are. So that's n times 4 to the minus l. That's fancy notation for 1/4 to the l to the 3/2. Notice that this is exactly the same pattern as the previous problem. I'm just plugging in some different constants. Yes.

AUDIENCE: I have a question about the diagram. If you have a really large a and you're trying to do the recursion tree method, is this type of work OK with where you have--

PROFESSOR: If I were grading your papers, it would be. But you should probably check with your TAs on Piazza on that one.

AUDIENCE: All right.

PROFESSOR: But I've learned my license. I'm not sure you guys have yet. And also, I am not sure I have, either. All right, folks. So now we need to do our total amount of work here, right?

So how many levels are there in our tree here? Well, the amount of data divides by 4 each time. And when the amount of data is 1, then I'm done, right? So the total number of levels is equal to the log base 4 of n.

The basic thing to get right in this problem is just where there should be 4's and where there should be 8's. OK. So now what am I going to do? Well, again, as our colleague in the back points out, we only have big O. So we can only upper bound our work.

But our work in the tree is less than or equal to-- really, there's a proportion here because this is just up to a multiple. I guess that's why, in the notes, they put a little c in front, which I can do. Let's do c like that.

Well, we're going to have the sum from l equals 0 to log base 4 of n of exactly the quantity, this guy. Multiply it by the number of nodes in the row. Yes.

AUDIENCE: Is the reason why you made this work an inequality because of the--

PROFESSOR: Because of the big O. That's exactly right. Yeah. OK. All right. So what do I get here?

Well, let's write it out the big ugly way first. So this is n times 4 to the minus l. This is to the 3/2. And then this is multiplied by 8 to the l. But whoever designed this problem was really sneaky, right? So what's 4 to the minus l to the 3/2? Any guesses?

AUDIENCE: 8 to the minus l.

PROFESSOR: Yeah, this is exactly 8 to the minus l if we work through all of your exponent arithmetic. Yeah. Why does that matter? Well, it exactly cancels this 8 to the l term here. This is actually exactly what case two in master theorem is trying to capture.

So this whole thing is equal to, really, just n to the 3/2 from l equals 0 to the log base 4 of n. All these constants go away. Isn't that beautiful? Yes.

AUDIENCE: Quick question about the tree again.

PROFESSOR: No problem. Uh-huh.

AUDIENCE: So do people ever use these for optimization in terms of trying to write a better algorithm, trying to minimize the work per level? Yeah, I'm just curious if that's an application that's--

PROFESSOR: Sure, I guess so. I mean, this is somehow a visualization of how your algorithm is making function costs, right? So every node here kind of looks like a call to your piece of code.

So every time you make a new one, it's like making a call. And then the divide here is kind of like how much data goes into that function call. So I think this is a useful way to kind of visualize what's going on inside of your code. And then if you're trying to optimize your code, you're trying to basically reduce the number of nodes and/or the amount of work that each node does, right?

Yeah. So this is just a nice way to kind of visualize what's going on in a recursive algorithm. Yeah. Great question. All right. So here's a really nice thing. Does n to the 3/2 depend on l? No, right? So this whole thing is equal to c times n to the 3/2.

And my colleague in the back is going to catch me if I don't make the off by 1 error here. So this is log base 4 of n plus 1 because my sum started at 0. Yeah.

And notice that this is exactly big O of n to the 3/2 log n, which agrees with what we did in a much more easy case with the master theorem. Does that make sense? Yeah.

AUDIENCE: So if I'm doing the exact bound, can we say that it's theta of n to the [INAUDIBLE]?

PROFESSOR: Oh, great question. This is where this notation is going to be misleading. This expression in a vacuum is theta of this value. But the reason that I wrote big O is because I only have an inequality all the way up here. So if I'm worried about bounding this work, there's no reason to write a theta there because it's just telling me some kind of intermediate piece of information. Yeah. OK, fabulous. So yes.

AUDIENCE: Does the base of the log matter?

PROFESSOR: Does the base of the log matter? Ah. This is a nice formula here. So remember that log base b of a is the same as log a over log b, if I got that right. Yeah. So the base of the log when it comes to big O doesn't matter because it's a constant factor. Yeah.

OK. So that's part two of this problem. Part three is a little bit annoying because it has two branches in it. I wrote out a careful solution. It does not apply the master theorem because it's irrelevant, at least the version that we know in this class.

So we're going to kind of skip that for now because I don't think it's terribly relevant to most of the algorithms that we'll see in 6.006. But we'll come back to it if we have time. Yes.

AUDIENCE: Can we make that a lower bound and an upper bound using master theorem, knowing that T of n/4 is smaller than T of n/3?

PROFESSOR: I'm going to refer you to the solutions rather than talking about that problem right now. Yeah. OK, cool. All right. So that's how you apply master theorem, which is mildly painful.

The good news is the rest of this problem set, actually, I consider to be much easier than the first one, which is why I think it's OK to spend a little bit extra time here because I think this stuff is confusing to get right. Mostly for me, and hopefully I've conveyed my confusion to the rest of you.

OK. So let's erase this. And while we're doing that, why don't you guys all give problem number two a read? So one of the big skills that we need to cover in 6.83-- not 6.837, 6.006 and 6.837, but I'm not quite as evil about it in that class-- is as follows, which is you read a problem.

And for some reason, your instructors have some sick sense of humor. And they encode it in this totally weird, goofy language which somehow, to a theoretician, makes your problem feel more practical. So in any event, when you read all of this paragraph, the very first skill that you have to do is to figure out, OK, like, this is cute notation, and it's about Infinity Stones. And if I watch *Star Wars* or whatever, I would know what that meant.

But in any event, what really matters is understanding, OK, but algorithmically, what are they asking? Yeah. So I'm going to try and talk about this problem as I erase the board. So I believe, what, like, Mickey Mouse or whatever has a bunch, has some planet that he's looking for.

I'm going to open the problem now. Right, she's a super villain on a quest. She's looking for a stone on a planet, and the planet has some index k. And unfortunately for us, the number of planets is quite large. It's infinity, in fact, because it's the Infinity Stone.

Or, sorry, did I say that? I think it's the Infinity Phone or something. I forget. But in any event, the only thing that you can do when you land on a planet is ask an oracle, is the index of my planet bigger or smaller than the index of the planet that I'm standing on, right?

And then the question is in log k time where k is the index of the planet. Notice that's already a little weird because it's not the size of your data, quite. Can you find the planet?

Now, what does this kind of problem-- like, what is it screaming out for you to use? You see a log k. You're looking for something.

AUDIENCE: Bisection.

PROFESSOR: Bisection, right?

AUDIENCE: Binary search.

PROFESSOR: Or binary search. That's absolutely right. Those are both great answers. But there's a bit of a problem, which is that the number of planets is unbounded. We don't know how many planets there are in this little universe that problem 1.2 sets up.

Yeah. So our intuition is that we want to use binary search. But in order to do binary search, I need to have a left- and a right-hand side and divide, and I have no right-hand side. So what can I do? How's a supervillain to solve this problem?

Well, remember that each planet has an oracle on it telling me. Is there something to my left or my right? Yeah. So I could start at planet number one, and I could just start walking from planet one to planet two to planet three, planet four, and asking, am I there yet? Am I there yet? How much time is that going to take?

AUDIENCE: Infinite, eventually.

PROFESSOR: Actually, it won't take infinite time. I caught you. How much time will it take? When will I stop? When I hit planet k, right, because I know that planet k is out there somewhere. The truth is out there. And when I find it and I step on it, I stop. And I took exactly k steps, maybe k minus 1, depending on how you count.

But we need a log k algorithm. Yeah. So what can I do?

AUDIENCE: You start at some k, and then if it's to your right, you multiply by another k?

PROFESSOR: Multiply by another k.

AUDIENCE: [INAUDIBLE] essentially perceived by another k planets. So it's like you're at index k squared.

PROFESSOR: Interesting. Hmm.

AUDIENCE: Couldn't you go from i to i squared?

AUDIENCE: Yeah.

PROFESSOR: OK, so you got the right-- you're in the right church, wrong pew here. So the basic intuition here is that stepping one planet at a time doesn't step fast enough. If you work out the details on that one, you're going to find that you'll end up with a runtime that still goes a little bit too quickly in k here.

In some sense, if you were to reverse engineer this problem, which isn't really a great way to problem set, you really do expect there to be powers of 2 in every step of your algorithm. Yes.

AUDIENCE: Could you do the binary search and start in the middle and then ask the oracle if it's higher or lower and then [INAUDIBLE]?

PROFESSOR: It's the right intuition. But I have a philosophical question for you, which is, what is the middle of an infinite set of planets?

AUDIENCE: Infinite.

PROFESSOR: Infinite, exactly. It's a problem. Yes.

AUDIENCE: We could use-- and I don't know who it was back there who was suggesting doing i squareds-- except for doing i squared, replace it with 2 to the i.

PROFESSOR: That's exactly right. That's a great intuition. So let's formalize that a little bit, which is like, I could do binary search if I had a right-hand side. I have a left-hand side because it's 1. So what I'm going to do, I'm going to keep trying right-hand sides.

And the oracle is going to tell me, is this a valid right-hand side, right, because the oracle is telling me, is there a planet to my left? Yeah. So here's what I'm going to do. Again, I don't like the index i. In the official solution, there's an i. But I'm going to use the letter m because I can-- which is the following.

I'm going to visit-- I hate teaching. No, that's not true. I like teaching. I just don't like chalk. I'm going to visit planet 2 to the m for each m starting with m equals 0 and essentially until-- remember that k is the index of the planet I'm looking for.

So eventually, I'm going to reach the point where k is less than or equal to 2 to the m. And I know that I can query my oracle, and they're going to tell me when that's the case. So how much time does this take? So I'm going to try planet 1 and then planet 2 and then planet 4 and then 8 and 16 all the way up until k.

AUDIENCE: Log k.

PROFESSOR: So it's going to take log k time.

AUDIENCE: Can't you also have a stronger or lower bound now to the m minus 1?

PROFESSOR: Yeah, that's sort of right. Right, so eventually, what's going to happen when you stop here is that 2 to the m minus 1 is going to be less than or equal to k. This one equals the 2 to the m because this is the condition for stopping. This is a condition for not stopping.

So when you stop k, this little sandwich is true. Yeah. So if I take the log, essentially, what I've shown is that I've taken, right-- because m is the number of steps in this part of my algorithm. So if I take the log, I'm going to find that I took log k steps. Yeah, nods, acknowledgment. Yeah. OK.

And then, well, now I have an upper bound and a lower bound. So now what can I do?

AUDIENCE: Binary search.

PROFESSOR: Yeah. Now I can binary search. And that also takes log k time.

AUDIENCE: Is it 2 times log k [INAUDIBLE]?

PROFESSOR: That's exactly right. Oops. So now I have step two is also binary search. And it's also a log k time, and our problem is solved. So problem number two is not so hard. Is everybody with me? Any questions about that one? That's a quick one. OK.

Now, problem three really spoke to me as a computer graphics professor. Right. So now I am running Fadobe. I collaborate a lot with Fadobe Fresearch.

Right. So Fadobe is trying to make a piece of software for image editing. And what does my piece of software do? Well, my image, or I guess my document, consists of a bunch of images that are overlaid with one another. OK.

And essentially, what's happening inside of the software is I want to keep all the images in order from top to bottom, OK, because when I render my photo, what do I do? I render the bottom one and then the next one and so on and just layer them on top of each other, like if you ever played with PowerPoint or Photoshop or I guess whatever fake name they're giving here. That's a pretty common user interface to encounter.

And so what they've asked you to do is to come up with essentially a data structure. And your data structure has to support a few different operations. In particular, you have to be able to make a document, import an image and then stick it on top, display, which returns all the IDs in the order that you've stored them.

And then there's the real kicker, which is that you need to be able to take one of the layers and stick it underneath another one. But you have to do that in log n time. It's that "but" that makes this whole problem kind of a pain in the tuchus. Yeah.

Right. So again, here our operations. We've got to make an empty doc. This is supposed to take order 1 time. We're going to import, which adds an x on top, and that this should take order n time. Notice that this is already a little suspicious.

If I were trying to psychologically diagnose my professors, I would look at this order n with some-- like, with a raised eyebrow, because probably what would you have in mind if you're talking about stacks of photographs? It would be a stack or a queue or some data structure like that. But then insertion would be like order 1 time. So there's clearly something a little more complicated going on.

OK. What's number three is display. This has to happen in order n time. This one kind of makes sense, right, because in order to display n things, you kind of expect to take at least n time.

And then finally, we have to move below. And this has to take order log n time. And this is going to be the kicker, right, because we really-- this is somehow not totally obvious from the way that we set up our problem.

So everybody understand the problem setup so far? We just keep adding objects with IDs, and we need to be able to insert them on the top and then kind of reorder them. And the problem has given you runtime for each of these different operations.

OK. Right. So here's the thing. There's kind of a sequence aspect to our problem and kind of a set aspect to our problem. Does that make sense? The sequence aspect to our problem is that we're going to have to display stuff in order n time, right? We've got to iterate over our whole list, put stuff on top, and so on.

And the set aspect is that we'd like to move stuff on top of each other in log n time. And the reason that I say that this is somehow set aspect is that I need to be able to put any x underneath any y, which means that I need to be able to quickly find what layer the x and the y is in. Any ideas how we can solve this problem? How about from some folks I haven't heard from yet? Yes.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, that's a really great idea. So maybe I'm going to maintain-- so remember, we've talked about-- let's see. In my lecture two lectures ago, we kind of thought of a sorted list as a set data structure.

So yeah, that's a fabulous idea. Why don't we, for the set part of our problem-- we're going to store in particular a sorted array of-- and for now, let's think of it like a sorted array of x's.

And so this is going to be able to help us answer questions like, is this ID in my set of images or not? But there's also going to be a second part of my problem, which is in addition to that, I need to be able to keep a different ordering other than being just ordered by x, which is the ordering that they're being drawn on the screen, right? That's different than the ordering of their IDs. So for that--

AUDIENCE: Quick question. We're going to be building a sorted array with nothing in it and it's going to be O of n log n time. But that's undefined. So can we build a sorted array with nothing in it?

PROFESSOR: Hmm. OK. No, I'm not sure I quite follow.

AUDIENCE: It's building an empty sorted array O of n log n time, which is O of 0 log 0 time. And log 0 time [INAUDIBLE].

AUDIENCE: Empty sorted array?

PROFESSOR: Yeah, building an empty anything that takes 1 time.

AUDIENCE: Oh.

PROFESSOR: Yeah.

AUDIENCE: OK, cool.

PROFESSOR: Cool. All right. So in addition to this, we have a sequence aspect of our problem. And for that, maybe we'll use a linked list, right? That's a pretty reasonable sequence data structure.

And in fact, let's think for a second. Now, we're going to need this move below operation, right? So our linked list is going to store the order of the images in our document. In order to move something below something else, we're going to have to splice an image in between two other guys.

So for convenience, maybe we have a doubly-linked list so that we can move backward and forward so that we can insert stuff. Yeah. So we'll have a doubly-linked list.

OK. Right. So let's start solving our problem and then thinking about where things are going to go wrong with our setup here. So first of all, I don't think we really have to write anything for part one because making an empty anything is a pretty easy algorithm. On your homework, of course, you should write that and tell us that it takes order 1 time.

Now, how about importing an object? Remember, that puts it on the top of our linked list. And we have to insert it into our sorted array, right? So when we do that, our insert algorithm is super simple, right?

We're going to add x to the set, which we talked about in class for a sorted array. How long does this take for a sorted array to add something?

AUDIENCE: Order n time.

PROFESSOR: Exactly, order n time. But that's actually OK because that's our criterion for number two. Yeah. And in addition to that, we'll put x on the top of the linked list, right? That's order 1 time. That's easy.

I'm adding space because we're going to see that we made a slight mistake and that we're going to need to modify our algorithm a tiny bit to solve this problem. OK, so how do we display? I think that's the simplest one, right? I'm just going to iterate over the entire linked list and just, one by one, output the order because, remember, the whole point of the linked list is to keep track of all of our documents in order.

So display, all you have to do is loop over the linked list. OK. And the real kicker is the last part, right, which is how do you move something below? So let's think for a minute. What's moving below going to entail?

So is it going to affect the set of keys that are in my document? No, actually, right? It's just changing their ordering, but both of those keys are already there. Yeah. So the only thing it's going to do is effect where they are in the sequence.

But there's something really, really annoying. How do I find something in this sequence? Let's say that I tell you that document 75 has to move below document 352.

Well, so far in my data structure, what's my only option? Does the set tell me anything about where stuff is in the linked list? No. So how much time is it going to take for me to find a key?

AUDIENCE: Order n.

PROFESSOR: Order n, right, because I've got to loop over this whole linked list and find the item that I'm missing. Is that allowed? No, the problem tells me that I have to do this in order log n time. So somehow in our head, we should be thinking, OK, well, we want to kind of use this set here to help us find stuff in the doubly-linked list.

Does that make sense, that intuition? So here's how we're going to do it. Actually, before I do that, any ideas? How can we solve that? Yes.

AUDIENCE: Maybe the [INAUDIBLE].

PROFESSOR: That's a fabulous idea. So here's what I'm going to do. Remember that when we make a set, we don't just have to sort the keys. We can attach data to our keys. Yeah.

And in particular, I am going to attach a pointer into my linked list. Really sneaky answer. So let's say that I have-- let's do a quick example. So let's say that I have my documents are 1, 5, 3, 2, and 7, like that. So my linked list is going to be real simple. It's doubly linked. See, there's two arrows here, just like that, right?

And now my sorted array is going to store all the keys in order. Let's see if I can do this with me. 1, 2, 3, 5, 7, right? So this is my linked list. And here's my sorted array is going to be 1, 2, 3, 5, 7.

But then what I'm going to attach to each of these guys is a pointer to the element of the linked list, right? So he is going to additionally contain an arrow here. The 2 will contain an arrow there. The 3 will contain an arrow there. The 5-- this is ugly, I'm sorry.

Now, here's the thing. Let's say that I remove this 3 here. Does that actually affect 7's pointer? No, right? So these pointers remain valid even if I edit other parts of the list. Really sneaky trick. OK.

So that is going to be our solution to this problem is to store not just the sorted array of x's but x's and pointers. So now we have to modify a tiny bit, right? And now what can we do to move something below? Well, it's going to happen in sort of three steps, right?

So our first step, well, we've got to first find keys x and y, right? So we're going to find x, vx, and y, vy in the set. How long does this take? Now we can do it in binary search, right? Exactly. So this is O, like that.

And now what's my next step? Well, remember, the move below operator sort of removes x from where it currently is and then puts it underneath y, right? And both of those are like linked list editing operations, right?

So I'm going to kind of-- if I have a linked list, like 1, 2, 3, 4, right, and let's say I want to erase the 2, I think you guys have all coded this up at some point in your lives. Then what I'll do is add links like that, which I can do because I can move forward and backward in the list, right?

And similarly, if I want to add something, then I'll erase the links, put it in there, and update the links. How much time does that take? That's just order 1 time, right, because now that I have the pointers to these two locations, I'm just doing a lot of rewiring in my linked list. But it's all kind of local stuff.

If you're coding in C, this is where your memory leak happens, and your company gets hacked. OK. Right, so that is what we're going to do next is update the linked list. And that takes order 1 time. On your homework, if you're writing out your answer, you should have an answer closer to what I've written on my page than the two words that I've written on the board here.

And then finally, in step three-- well, actually, there is no step three that I've written here. Sorry. There's actually two parts to updating the linked list, right? One is to remove the old x position and then insert in the proper position, right?

And when I do that, my whole update happened. Notice that there's something kind of interesting about this move below operation, which is that, did I actually edit the set at all? No, right? That kind of makes sense because the set is just a set of keys in my document.

And just by changing the ordering, it doesn't affect what's in my document. So somehow, it's a sanity check that that works out. So this entire algorithm takes how long? Well, there's order log n. There's order 1. And there's another order 1. So the whole thing is log n. And life is good for our problem here. Yes.

AUDIENCE: So how we can put such pointers in real life?

PROFESSOR: How do we input such point--

AUDIENCE: [INAUDIBLE] like a set with pointers in it in real life?

PROFESSOR: Got it. So I only know the C++ version of this. I'm going to get it wrong for Python. But I'm not sure that I quite understand the question. So it's just like pointers anywhere else in real life.

So I guess in Python, what you'll do is you'll create a new, right-- as you add these guys to your set, you'll create a new object here. And in addition to that, you're going to create a new vx that you add to the linked list. And the pointer is just an address in memory, right? So essentially, what you'll store--

AUDIENCE: [INAUDIBLE] linked list nodes are the pointers.

PROFESSOR: The linked list nodes are pairs, a pair of an x value and a-- well, oh, I'm sorry. Yeah, actually, I guess that's right. No. The linked list just contains a long list of x's, right?

But they don't have to be in contiguous memory. That's the only difference. So you make them one linked list item a time. But it's just like constructing any other linked list. Yeah.

In fact, there's pseudocode in the solutions that are distributed. So you can take a look there. Yeah.

AUDIENCE: Yeah, speaking of which, for implementing these database operations, can we pseudocode the algorithms for them?

PROFESSOR: I think the official answer is no, that you really are supposed to write out in words what your thing is doing. Now, there's a weird gray area, of course, which is that the words that you write out are going to look an awful lot like pseudocode.

But you should make an effort to try and write it out a little bit in paragraph form, be descriptive about it. This is a good skill to learn. At the beginning of this course, I think it'll feel a little pedantic at points. And that's good because it should, and you should deal with it. Yeah. But really, try and write things out in a way that captures your logic rather than just Python code. Yeah.

Cool. All right. So let's see. We've got approximately 17 minutes left in class, which is approximately 1/8 of the time it took me to figure out the solution to this last homework problem. But we'll get started. And I've written out a careful answer because essentially, in problem four, I found myself getting stuck in a few little details.

And so I figured the way that I write out my little hand-scribbled chicken scratch answer online was to just basically give you an internal dialogue of my brain and how I went about solving these things and all the stupid missteps that I made. Yeah. Because I think that's actually quite valuable. Essentially, it's often you see in these problems sessions just like, oh, here's a problem. Here's the answer. Here's a problem. Here's the answer.

But the reality is even your professor, especially because he's used to thinking about rendering, occasionally gets confused about these algorithms problems, especially when they're frankly written in a bit of a confusing way. And I heard some war stories about this problem actually being on the homework. Apparently, I'm in good company.

So let's take a look at this last homework problem, 1-4. So this is on brick blowing. And this is a great exercise in taking a problem that is described in really long, nasty, useless language and then extracting the two words that matter.

By the way, I'm making a joke about that. I can tell you that about 82% of my time as a professor is spent with people from industry visiting my office with exactly this kind of scenario where it has a really complicated thing. They've been thinking about their construction problem for years, and they have all these details and tables and flow charts. And then it turns out that their problem can be captured in about two sentences.

So it's a good skill to have and one that can get you a lot of money as a consultant. So it's one that's worth practicing in this class even if we're doing it for Porkland with a wolf that blows only to the east. OK. So what's going on in this problem? There's a wolf, and the wolf can blow on houses.

But for some reason, the wolf likes to blow to the right. And to make matters even more complicated, whoever wrote this problem occasionally ordered things from east to west and other times ordered them from west to east. If I were your instructor at the time, that, I would have at least gotten rid of because that's just mean.

But in any event, I believe the story goes as the apocryphal house-blowing pork wolf story that we all know and love from our childhood is there's a row of houses, each of which has a different number of bricks. Yeah. And now there's a wind which is blowing from west to east.

I remember all this because I was staring at it this morning in my office. And essentially, the wolf, being the big, bad wolf that he is, says, aha, if I, too, blow from west to east, I can knock over more houses more efficiently because the wind is helping me. You don't want to blow into the wind. You get your spit in your face.

OK, so the wolf does that. And then what happens is that the wolf not only knocks down the house upon which the wolf blows but also all the houses to the east of that house that have fewer bricks. Why, you might ask? I don't know, because whoever wrote this problem was being a goofball.

But that's the basic setup of this problem. And you can see that, essentially, this is just a long, convoluted way of describing something pretty straightforward. And so at the end of the day, the wolf, being kind of an adversarial wolf, wants to blow down as many bricks as possible. And we are providing the wolf with the analytical tools needed to do so. Yeah.

So in case you didn't think we covered anything practical in 6.006, now you know that we actually are doing state-of-the-art wolf brick-blowing pig analysis. OK. Right. So let's do an example because this is weirdly complicated.

So in the first part of the problem, it asks you to just do an example out. And I think if I were psychologically diagnosing the person who wrote this problem, probably the reason is that they stood back, and they said, nobody's going to understand this unless I force them to do one example by hand. Right?

And so the basic problem is asking, I could choose to blow on each house. Which one should I choose to cause the most damage? Or in fact, they actually ask a slightly different problem, which is if I were to blow on each house, how much damage would I cause?

OK, so let's step through the problem. So they give you an example, a set of numbers, 34, 57, 70, 19, 48, 2. Actually, let's just stop there. I don't think there's any reason to do a list of a million numbers like in the problem. That was just like, the more times you do it, the more you convince yourself how useless 6.006 is. OK. Right.

So how much damage does blowing on house number one cause? Well, let's make a little table. So if I blow on house number one, just by default, house number one falls. I'll pull an x there, meaning I blew it over.

And then the rule is any house to the right which has fewer bricks-- the physics of this problem drives me nuts. I thought about it for quite a long time, and I can't think of a wind that actually has this property. But any house that has fewer bricks and is to the east also gets blown down.

So 57 does not, right? That's bigger than 34. Similarly for 70. Aha, the 19 goes with the wind. 48's bigger, and the 2 goes, right? So the element number one of my array would be 3, right? Three things get blown over.

OK, for the next guy, right? So the 57 gets blown over. It does not blow over the 34 because my wolf only knows how to blow to the right. Uh. So it does not blow over the 70. It does blow over the 19. It does blow over the 48. It does, right? So here, there'd be a 4.

These numbers do not match the ones in the notes because this is a shorter list of numbers, I should point out. OK, let's do one more, and then we're going to stop because this is laborious and boring. So the 70 blows over everything. Yeah. But only everything to its right because, as we all know, wolves only blow to the-- OK, I'll shut up.

So there's four things that get blown over here. But you guys get the point, right? And so essentially, this problem is asking you to fill in arrays that look like this for your problem. Everybody on board with the problem?

Now, notice that just by virtue of doing this example, let's think about this abstractly because I think this is a problem that's just coded in so many words and made-up weird TA garbage. But at the end of the day, this problem as an algorithmic problem is not so bad.

Essentially, what I'm asking you to do is I look at that 57, and I look to the right, and I count how many things are smaller than 57. And I add 1 to it. And that's the number that should go in that element in the array. So notice that those three paragraphs, that's what they're communicating. You didn't need wolves blowing and winds and trees and houses and Dorothy and all that kind of stuff.

Yeah. Right. So at the end of the day, that's the problem we can think about. And we no longer have to think about Porkland ever again. OK. So maybe in the remaining 10 minutes here, I think we can feasibly do part b. And then part c is an extension of part b.

I got myself confused on part c, so the handwritten solutions have an incorrect answer, and then an oops, and then a correct answer in it. So you can see where my logic went wrong. To be fair, I already knew the full answer and wrote out both because I thought that was useful.

OK. So in case our problem wasn't contrived enough, we're going to add a little bit of condition to it. And then what we're going to see in part c is that's somehow a stepping stone toward solving the bigger problem. So in part c, they say that a house in Porkland is special if it has one of two properties. Either it has no easterly neighbor, meaning it's just the rightmost house, or its adjacent neighbor to the east contains at least as many bricks as it does.

Right. So what does that mean in terms of this list of numbers? Which one of these guys is special? So the easterly guy is special. We just know that by default. And other than that, this special property is saying that the guy immediately to the right has a bigger number or at least equal to or larger to the one right before it. Yeah.

So 57 is bigger than 34, so this guy is special. By the way, the problem doesn't ask you to do this as an example. But I totally would if I were solving this. Yeah.

70 is bigger than 57, so it's special. 19 is smaller than 70, so it's not special. I'm sorry. 40 is bigger than 19. And I guess 2 is special just because things on the right are special. OK, so do we all get the definition? Fabulous.

So in this problem, now what we're given is that I guess, what, our evil wolf takes a walk down the block and takes a look at-- counts the number of bricks at each house. And he or she makes an observation, which is that all but one house is special.

Now, let's think about what that means. Let's think about the structure of our array. So what happens as I walk along my right? If a house is special, what do I know about the next house?

AUDIENCE: It won't get blown if you blow a special house down.

PROFESSOR: That's true, but we're already solving the problem there. Let's think qualitatively for a second, like just in terms of the number of bricks, right? I'm walking down the street. I'm looking. And the number of bricks increases until I get to the one house that isn't special.

AUDIENCE: [INAUDIBLE]

PROFESSOR: And then it decreases, potentially, or it's the last one. And then it starts increasing again. Yeah. So in other words, my array kind of looks like the following, which is that it basically has two different pieces that are both sorted. Yeah. That's what this long paragraph is saying. It's saying suppose that the list of houses is sorted except for one thing. Yeah. That's how you could have said it.

OK, so here's an example. So maybe we have 1, 2, 3, 4, and then, I don't know, 2, 3, 4, 5. So all of these is special with the exception of this guy, sadly for him.

And so there's kind of a little dividing line that happens, to the left of which, it's sorted, and to the right. Yeah. OK, and then the question is, can we predict the damage for each house based on this information? So I want as output to my code an array that says, this guy causes so much damage, this guy causes so much damage, and so on.

And I want to get it in order n time. Let's think for a second. So how are we going to do that? So order n time means that I can't afford to do sorting and all that kind of fancy stuff. So basically, all I can do is walk up and down the array.

So we should think a bit about, what's special about special-- oy vey. What's special about this configuration that we wouldn't normally have?

AUDIENCE: Anything before the non-special node should have the same-- or I'm sorry.

PROFESSOR: Not quite.

AUDIENCE: Everything after the non-special node causes only one damage.

PROFESSOR: Yeah, so let's think about this carefully. I think that's right, but I'm bad at parsing sentences. Remember that I always blow to the right, and I only blow short stuff. So if I blow on this 2 here, will anything before that vertical line ever get blown over? No, because by property of being special, we know that the list is increasing. But it has to decrease to blow something over. That make sense?

So for each one of these guys, the only stuff we can blow is in the second array. Does that make sense? So we made an observation. Notice that it is special to the structure that we gave to our problem. Yeah.

And moreover, so in both parts of this problem, we're going to look at two parts of an array and then merge them together in a smart way. Let's look on the right-hand side. Does any of these things-- if I blow on any of these houses, what's the damage that I do? 1. You see that?

Because this is an increasing list, I only blow to the right, and it has to get shorter. Yeah. So if I want partial credit on my homework problem-- probably very little partial credit, but spiritually, it's half of your credit-- we know the second half of my array is just a bunch of 1's.

Incidentally, one detail which I bet would get a minus 1 on your homework, we didn't tell you that-- all we told you is that there existed one house that wasn't special, but we didn't tell you which one. But notice that I can find it in order n time, right? I can just walk along my array and find the first place where it decreases, and that's a special house. So I can kind of assume that I know where this line is. That's kosher.

So then the only question is, how do I fill in my array here? How do I count-- and so let's think about this mathematically because porks and blowing and stuff is kind of confusing. So let's think about it in terms of this array.

Essentially, what am I trying to do? So for this 2, I'm trying to look in the second half of my array and find all the numbers here that are less than 2, right? That's essentially what this problem is asking you to do. You see that?

OK. I'm going to make one-- so here's a simple kind of n squared algorithm, right, which is I could iterate over every item here and just loop over here and count the number of things that are less than that, right? That's a double for loop, each of which could be up to n. So that's n squared. That's against the rules. Can't do it.

So we need one more additional observation. OK. Let's think about this for a second. Let's say I'm the wolf, the pork-blowing wolf. And I walk from one house to the next. And I keep track of what houses get blown over here.

Now, what happens? As I walk from one house to the next on the left-hand side, they're only getting taller, right? That's what it means to be special. So the houses that get blown over here, the set of those only grows larger. Does that make sense? Because as these guys get taller, more stuff falls over on the right-hand side.

It's like a total goofball talking about this problem. Moreover, these guys are in sorted order. Yeah. So what's going to end up happening is that, always, there's just some interval of houses that gets knocked over. So for example, the 4 blows over the 2 and the 3, I guess.

Notice that that always starts on the left-hand side. And that's because these guys are sorted. The 3 just blows over the 2, which is kind of a subset of the 4. This isn't a great example because those first two guys don't blow over anything. That make sense?

Any ideas? How could I use that observation to make an order n algorithm for counting blowiness damage? I've heard a lot from you. Let's hear from some of our other colleagues here. Another person. Yes.

AUDIENCE: Binary search.

PROFESSOR: Binary search. Tell me more. What would I search for?

AUDIENCE: So let's say you have 4, and you move by yourself from the right, take the index of that, and then you take the number.

PROFESSOR: It's an interesting intuition. And it's, like, 82% right. So to repeat our colleague's suggestion, which is a good one and will make your code a lot faster-- in fact, in practical terms, it would probably be about the same. It would be I'm going to move from the left to the right-hand side unless there's a very large neighborhood of pigs.

I'm going to move across this array. I'm going to binary search this other side and find the first house that shouldn't get blown over or something. And now I know the total number of things that get blown over. Now, unfortunately for you, what's the runtime of that algorithm? n log n, right? Because for every single guy here, I incur a binary search there, which is log n time. So you're in the church, wrong pew here.

But there's an observation we haven't used yet, which is as I move to the right, these numbers increase. Yeah. So what's going to happen to the index of the stuff that gets blown over? It's always going to move this way. Is it ever going to move to the left? No.

So there is a term that we used in two lectures ago, which is a term I never heard before, but I kind of like it. It's called a two-finger algorithm. Yeah. What does the two-finger algorithm do? Looks like this. Two fingers. Yeah.

So what I'm going to do is I'm going to keep my finger on the right pointing at the very first thing that doesn't get blown over. OK. So here, right, the 1 doesn't blow over anything. So it stays here. I'm going to iterate to the next guy. Does the 2 blow over anything? No, by the way the problem is written.

Now I'm going to move this guy here. Well, now the 3 blows over the 2 here. So I'm going to move my right finger one thing to the right and move the guy to 4. The 4 also doesn't blow over anything, and then I guess I'm done.

But the basic point here is that I'm always going to be incrementing two different pointers and moving to the right. And so what am I going to do? If I call this finger i and this finger j, will i or j ever decrease? No.

So if my algorithm is looping over all of these guys and they only ever touch each of these indices once, I'm going to get an order n technique. Yeah. And that's going to be the basic trick to solving this problem. Now, I've managed to talk myself out of time. But basically, that's it.

So what I'm going to do if we have 10 extra seconds is I'm going to initialize. Essentially, for each i, I'm going to increment j until the number of bricks at j is greater than the number of bricks at i. I guess that should be greater than or equal to. Right?

And the reason to do that is that now the number of houses that get blown over is just this index j minus the index of the first guy, right? And then I'll increment i and continue. That make any sense? Fabulous.

So I managed by some miracle to make it to almost the end of this problem set minus one part of one problem. Of course, that's the hardest part of the whole problem set. But there's an answer written out where, essentially, now the question is, can you do exactly the same algorithm but I don't give you this special assumption that it increases and then decreases?

And it's going to be basically an extension of this idea, as we're going to use this plus a little bit of merge sort. OK. So with that, we'll see you guys next week. It's always a pleasure. And yeah, have a lovely weekend.