[SQUEAKING]

[RUSTLING]

[CLICKING]

ZACHARY ABEL: All right, let's get into it. Last lecture of material before the quiz. So let's make sure we make the most of it. Today we're talking about recurrences.

"Recurrence" is a fancy word for saying sequence described by induction. So it's a sequence defined inductively. And we've seen multiple examples. A great example is the Fibonacci numbers. Are there two B's? "Fibonnaci."

Let's pretend I spelled that right. No, it's two B's and one N isn't it? Are there two C's? I don't know. [CHUCKLES] No one's helping me. Doesn't matter.

Defined by F0 is 0, F1 is 1. And the important part is that fn is defined as Fn minus 1 plus Fn minus 2 when n is at least 2. And this is the part where we're describing the sequence inductively.

Once you know the first two terms, the next term is the sum of those. So F2 is 0 plus 1 is 2. F3 is 1 plus 2 is 3. Then 2 plus 3 is 5. 3 plus 5 is 8, then 13, 21, 34, and so on. It's just a sequence defined inductively. That's the idea of a recurrence.

And these show up all the time when analyzing algorithms for many of the same reasons that summations show up when analyzing algorithms. And we need a lot of the same things out of them. If we end up with an algorithm whose runtime we can describe as the Fibonacci numbers or as a similar recurrence, well, that's not as useful as it could be.

A closed form would be a lot more helpful, right, if we could have it, if we could find it. And indeed, Fibonacci numbers do have a closed form. The n-th Fibonacci number is 1 over square root of 5 times 1 plus square root of 5 over 2 to the n minus 1 minus square root of 5 over 2 to the n. And there's a closed form.

You just take two exponentials, subtract them, divide by root 5, and you get the n-th Fibonacci number exactly. I think this is a really cool formula, also non-obvious. How do we possibly go from this to that? Well, we have some techniques that might work sometimes, and other times we won't.

But certainly, once we know this formula, if we believe this formula or want to check whether it's true-- now that we know the formula, we can just try the induction and see if the induction works out. In this case, it will work out. And then you'll not only know that this is the right formula, but you'll have a proof for it.

So just like with summations, if you know a closed form, then usually it's easy to prove it by induction. So finding the closed form is the hard part, proving it is the easy part. So this guess-and-check-- guess and check-- technique that we use for sums works just as well for recurrences.

Now, recurrences very often show up, especially when analyzing recursive algorithms. So let's talk about recursive algorithms and show some examples. So let's start with a very classic, fun example called the "Towers of Hanoi."

So this comes with its own myth. It was described by Edouard Luca in 1883. He said there is a temple in Hanoi that has three large pillars. I have some helpers today acting as my pillars. We've got "P-Town," "Wire," and Totoro. And on the leftmost pillar are 64 golden disks stacked on top of each other, getting bigger as you go down.

And the monks there are moving the disks around one at a time but according to a couple rules. You're only allowed to move one disk at a time from the top to some other peg. And you're never allowed to put a big box on top of a small box-- I mean big golden disk on top of a small golden disk.

Those are the rules-- one peg at a time. So I can't pick these both up and move them elsewhere. And always small on top of big, never the opposite. The goal is if all the disks start over here on the left peg, we want to get them all to a different peg. We want to move them all to the right peg by moving one disk at a time.

So let me write that down quickly. So we have, let's say, n disks on the left peg. We have two other pegs. And we're going to move one at a time, always keeping smaller disks on top of bigger ones.

Goal-- all disks on the right peg. And as the myth goes, when the monks finish, when they get all 64 disks over to the other peg, the universe ends. I don't know why. That's just the story.

Maybe there's a weight-sensitive detonation switch over there. And you have to get all that weight on there, and then the universe blows up. It's a myth. You don't need actual answers here.

But that's the game. And the question is, how many moves will it take? Is it even possible? How long does the universe have to live? Do we have a year? Do we have a century?

Let's see if we can figure it out. Let's see if we can find an algorithm to get all of the disks from one side to the other. Are we clear on the game? Nice.

Let's start small. I always recommend starting small. What happens if we have just one disk and we need to move it from "P-Town" to Totoro? How are we going to do that? Is that a hand? So how do I move a disk from here to here according to the rules of the game? Yes?

AUDIENCE: You just do it.

ZACHARY ABEL: You just do it. Yeah, we're allowed to move one disk at a time. And there aren't any bigger or smaller disks to get in the way. So you just do it in this case.

So if we're trying to move just disk number 1 from, let's say, peg A to peg C-- we'll call the pegs A, B, C-- well, you do it just by doing it. You move disk 1 from A to C. Great, OK.

What if we have two disks? Here we go. How am I going to get both of these from there to there? Who can help me out? Yes, please.

AUDIENCE: Small one in the middle.

ZACHARY ABEL: Small one in the middle.

AUDIENCE: Second one--

ZACHARY ABEL: Second one goes over here. Small one on the big one. Absolutely, perfect. So if we're trying to move disks 1 and 2, the two smallest disks, again from peg A to peg C, as your colleague pointed out, first let's move disk 1 from A to B. Then we'll move disk 2 from A to C. Then we'll move disk 1 from B to C, just as we acted out over here.

Next case-- three boxes. Now, this-- there we go-- this might take a little bit of planning ahead. And the thing to remember is that, at some point, we want to be able to make this move, where we take the biggest box from the left to the right. And so to do that, we need the other boxes here in the middle.

So somewhere along the way, we're going to need to go from here to here. Of course, that's not a valid move. We can't move two boxes at once. Who can help me out? How are we going to do this first part of the algorithm? Please.

AUDIENCE: Move the small box to the middle.

ZACHARY ABEL: Small box to the middle. We're trying to move these two to the middle. So here to here is our goal.

AUDIENCE: Wait. I need to think about this.

ZACHARY ABEL: Yes.

AUDIENCE: Move the small box to the right.

ZACHARY ABEL: Small box to the right.

AUDIENCE: And then the medium box--

ZACHARY ABEL: Medium box to the middle. Small box on top of it. Perfect. Does that look familiar to anyone? That's kind of exactly what we just described for moving boxes 1 and 2 from one peg to another.

Instead of moving from the left peg to the right peg, we're moving from the left peg to the middle peg. But that's the only thing that's changing. We can use the same algorithm to move these two boxes from here to here in three moves.

Now we can do this. And now we can do the same algorithm again to move these two boxes from here to here. And if we think about what that algorithm says, it says 1, 2, 3. Let's write that one down, as well.

So if our goal is to move the interval 1 through 3, so boxes 1, 2, and 3, from peg A to peg C, well, then the first thing we're going to do is move boxes 1 and 2 from A to B. Then we're going to move box 3 on its own from A to C, then boxes 1 and 2 from B to C.

And here, when I'm saying move two boxes at a time from one peg to another, I'm really invoking this strategy. We already know how to do it with two boxes. So let's use that as the first part of our algorithm. Then make one move. Then use that again.

Does that make sense? Let's do one more example together. OK, great. We have four disks. We need to move them from the left to the right. And we're going to follow-- hey, everyone. We're going to follow the same idea.

Somewhere along the way, we're going to need to get to this position-- to this position-- so that we can move the bigger box over here. And then we're going to need to, again, move these three boxes together somehow, with some sequence of moves, to finish the job.

And I claim we have all the tools we need to do exactly that. So step 1-- in fact, let's write this down first. If we're trying to move the interval from 1 to 4-- so boxes 1, 2, 3, and 4-- from A to C, let's follow the same strategy here.

Let's put the top three boxes-- let's move those from A to B somehow. Then we'll move box 4 on its own from A to C. Then we'll move boxes 1 through 3 from B to C.

And here, where I'm saying move three boxes from one peg to another, we have an algorithm for that. It's this algorithm. Yeah? So let's see what this looks like in practice.

So first things first, we're trying to move these top three boxes from here to here. Well, we have a way to do that. That's 1, 2, 3, 4, 5, 6, 7.

If we followed the algorithm that's listed there, those are the seven moves it says to do. And I want to point out one important thing. We put this one over here.

Back when we were just working with three boxes, we were assuming that all of the pegs were empty, that there was nothing else to worry about other than those three boxes. But now there's a fourth box in the way. So can we really run our algorithm for the top three boxes even when there's a fourth box?

And, yeah, we can thankfully because the fourth box will always be underneath. The fourth box doesn't affect the movement of 1 through 3. In fact, any larger boxes don't affect the movement of boxes 1 through 3.

So we can use our algorithm for three boxes and just temporarily ignore the fact that there's a fourth box. It doesn't matter. And that's how we were able to get here.

Then we're going to do this. And then, again, we're going to follow our three-box strategy to get these three over to here. So that looks like 1, 2, 3, 4, 5, 6, 7. And we've done four boxes. Yes?

AUDIENCE: How are you able to claim that that sequence of moves, especially as the number of boxes becomes bigger, is the most optimal?

ZACHARY ABEL: Great question. How are we able to claim that this sequence of moves, especially with more and more boxes or disks, is the optimal algorithm? We've started describing an algorithm to do it, but is it the best?

That's a great question. It turns out it is the best. We're not going to prove it right now. But it's not much harder than what we've already done.

The basic idea is, at some point, you have to move the smallest-- or the largest box. So at some point, you have to get to either this position or this position. And we know how many steps that takes, assuming we're doing some proof by induction or something.

So we know we have to spend some number of steps for n minus 1 boxes. Then we have to make a move. And then we have to put these back on top, which, again, takes-- a landslide.

So with an induction very similar to that, you can prove that, in fact, this is the optimal algorithm. OK, but how many moves does it take? Let's see if we can figure that out.

So let's let T of n be the number of moves with n disks. And we want to figure out T of n somehow. So here, T of 1, if we're just moving one disk, we make one move. So T of 1 is 1.

T of 2-- well, T of 2 is written here. It's 1, 2-- three moves. If we're moving two disks, we go 1, 2, 3-- three moves. How about T of 3?

Well, it says do the two-disk algorithm. So that's T of 2. Then do a single move. Then do the two-disk algorithm again. So that's plus T of 2 again. So that's 3 plus 1 plus 3 is 7.

Likewise, T of 4 is-- well, you do the three-disk algorithm. That's T of 3. We do one move. Then we do the three-disk algorithm again. That's another T of 3. So we're looking at 2T of 3 plus 1, which is 14-- 15.

And now we've exactly described the number of moves that this algorithm takes. And just to really be clear about it, if we're trying to move disks 1 through n, so n disks together from peg A to peg C, we're going to first move the first n minus 1 disks from A to B.

Then we're going to move disk n on its own from A to C, then the remaining n minus 1 disks from B to C-- exactly the same strategy we've just described. And this is going to be our general strategy.

To move n disks, we do our n minus 1 disk algorithm, then a move, then our n minus 1 disk algorithm again. So now we've described an algorithm by induction. [WHISPERING] And secretly that's all recursion is. We're just inducting.

But in general, the number of moves we take is, well, however many moves we needed to take for the n minus 1 disk algorithm plus 1 plus Tn minus 1 again. So it's 2T of n minus 1 plus 1. And this is our sequence. This is our recurrence.

And we're getting to the place where these are in the way. So let's move those. Success. OK, we've described our algorithm. We've described its runtime.

But like I said earlier, closed form would be so much nicer, right? I want to know how long the universe takes with 64 disks. I don't want to have to compute 64 terms of my sequence. So what does this look like? Well, let's see.

We have our numbers 1, 3, 7, 15. 2 times that plus 1 is 31, then 63, then 127. Is anyone starting to see a pattern? Yes?

AUDIENCE: Isn't it just 2 to the n minus 1.

ZACHARY ABEL: We have a guess. T of n is 2 to the n minus 1. This looks suspiciously like 2, 4, 8, 16, 32, 64, 128. And especially since we really are doubling every time, it seems likely that it might be related to powers of 2 somehow.

Well, the good thing about having a guess is that we know what to do with it. We check it. We try to use it to prove this formula by induction. And if it works, we have the right formula. Turns out it works.

You can prove it with a pretty straightforward induction. And it turns out this is, indeed, the correct formula. So this is a clear example of what we said about guess and check. If you can guess the formula, you check it with induction.

But we had to recognize that pattern to be able to have something to check. But finally, we can answer our question-- how long does the universe have left? 64 disks-- so T of 64. That's 2 to the 64 minus 1 steps.

Let's assume these monks are really strong and they can move one disk per second. Turns out, if I flip to the correct page in my notes, that many seconds is about a half trillion years.

Universe has got some time left. And I find it surprising that with just 64 disks, you can get all the way up into the trillions of years. That's very quick growth. All right, any questions about Towers of Hanoi? Awesome.

I thought the camera in the back was a hand for a second. I apologize for that. Let's move on to our next recursive algorithm. I promised last time that we would talk about sorting. So let's talk about sorting.

And before we get to merge sort, which is the fast algorithm we're going to talk about today, let's talk about a simpler algorithm called selection sort. The problem is you have a list of n numbers. And you want to arrange them in increasing order.

So your list is in whatever order it's in. And you want to figure out the correct increasing order. So put them in sorted order. And the idea for selection sort is a very simple one. Find the smallest. Pull it out. Repeat until done.

So let's see this in action. And I have some helpers today. I'm going to need a few more volunteers. Thankfully, I brought some. Nice. All right, so we have a bunch of helper friends. And we need to somehow put them in sorted order.

We're going by height here. And for selection sort, we said find the smallest. Well, clearly that's "P-Town" here. Pull them out, and then repeat. OK, who's the smallest left? Well, that's going to be "Wire."

Who's the smallest left? Well, it's maybe a little less clear. But let's talk more precisely about how we're going to find the smallest every time. And we can do this with just a bunch of comparisons. So let's compare the first two. Between Ralph and "Wire," whose shorter?

AUDIENCE: Wire.

ZACHARY ABEL: "Wire," yeah, OK. So "Wire" is the smallest one we've seen so far. Between "Wire" and "P-Town," whose shorter? "P-Town." Between these two, whose shorter? Still "P-Town." Still "P-Town." Still "P-Town." Still "P-Town."

So we got through the end of the list. "P-Town" is the smallest one we've seen. So "P-Town" is our smallest, softest, roundest boy. And that is how we are going to find the smallest. We can do that with just n minus 1 comparisons.

We compared the first two. Then whoever won that, we compared to the third, then compared that to the fourth and the fifth and so on, so n minus 1 comparisons. And then you pull them out, and we're going to do it again.

So now with n minus 2 comparisons, because we have one fewer animal up front, we're going to find that "Wire" is the smallest. Then we're going to find-- I think it's Bubs, is the smallest, then Totoro, then Ralph, then Coop-- get out of here-- then Butternut.

And now we have our sorted list. How many comparisons did we have to do to implement that? Well, we've already said that we did n minus 1 comparisons in the first round. And then the second round, we had n minus 2 comparisons.

The third round, n minus 3 comparisons, down to the last round, where we had just two animals left. We need one comparison to figure out who's shorter. And then the last one is just the last one. We don't need more comparisons for that.

And we've seen this sum before. This is n minus 1 times n divided by 2 comparisons. So this is n squared over 2 plus change comparisons for selection sort.

This is not the fast sort that was promised. This is still quadratic runtime. It is the easiest thing-- like when you have a small number of things in front of you and you just-- sorry for calling you "things"-- small number of animals in front of you--

[LAUGHTER]

--and you just want to sort them, it's very easy to just scan through, find the smallest, find the smallest. That's what I do by hand. But for big lists, for lots and lots of numbers, this is not the way you want to go.

But a much better algorithm-- not the only better algorithm, but a much better algorithm-- is pretty similar to this. And to talk about it, first let's talk about the idea of a merge. The idea is we have two sorted lists, A and B. And we want to combine them into a single sorted list.

As an example, so maybe we have these, and that goes over there. There. Totoro, stay. There we go. So for example, if we have this sorted list, which is already in increasing order, and this separate sorted list that's already in increasing order, let's try to combine them into a single sorted list.

And let's just do something like selection sort. Let's find the smallest and just keep finding the smallest until we have our answer. So who's the smallest between these two lists? Where do we have to look? I saw this. Was that an indication?

AUDIENCE: [INAUDIBLE]

ZACHARY ABEL: All right, so the beginning of here or the beginning of here. Yeah, the smallest on the table is either going to be the smallest on this side or the smallest on this side. And we know where the smallest is because the lists are sorted.

The smallest on this side is "Wire." The smallest on this side is "P-Town." We just have to look at the first from each of these two sorted lists. And with just one comparison, we know who the shortest is overall-- still "P-Town."

And then we can continue-- first over here compared to the first over here. "Wire" versus Bubs, I think "Wire" is smaller. Now it's Bubs versus Totoro. And I think Bubs is smaller this time.

Now it's Coop versus Totoro-- Totoro. Coop versus Ralph-- Ralph is shorter. And now we get to an interesting point where there is no first over here anymore. This side of the table is empty. Well, that just means that we're done.

We know this side is already sorted and bigger than everything else we've seen. So we don't have to do any more comparisons. We just pull them back, and we're done. So that's the idea of merging two sorted lists.

The algorithm says we compare smallest in each list, pull out the smallest, continue until one list becomes empty. And we put the other list at the end.

So we got it down to where this list was empty. This list had two animals left. And we didn't have to do any more work. We're just done with those.

Is this algorithm clear? Does it make sense why it works? How many comparisons does it take? Well, each round, every time we identify the next smallest animal, we're doing a single comparison. Yeah? In the worst case, we're going to have to do that for all of the first n minus 1 animals.

If it comes down to one on the left versus one on the right, then everyone needed a comparison. We only got one freebie at the end. So worst case, this takes less than or equal to size of A plus size of B minus 1 comparisons. Sound good? Awesome.

So now that we have this merge algorithm, let's talk about merge sort. So you're given a list with n elements. And we want to sort it.

This list doesn't have any promises. It's not half sorted or anything like that. It's just some random list of random data. We want to put it in sorted order.

And by the way, let's assume for now that n is a power of 2 so we can just keep cutting in half and everything works out nicely. We'll remove that assumption later. But for now, we'll just solve this for lists whose lengths are powers of 2.

So the first step, we are going to sort the first-- oh, sorry, I forgot an important step. If n equals 1, done. If you're sorting a list with just a single element, there's nothing to do. Your list is already sorted, so we'll call it done.

Otherwise, we're going to the first n over 2 elements. Somehow we'll get them sorted. Then we will sort the last n over 2 elements somehow. And then you merge them together.

So as long as we're somehow able to sort the first half and somehow able to sort the second half, the merge step will take care of the rest. Does this strategy make sense? Cool.

And you might ask, well, how are we going to sort the first half and the last half? Maybe we'll use selection sort. Turns out, if you do that, you're still going to get quadratic runtime. That's not the best plan.

The strategy I'm going to propose, and the way that merge sort is described, we're going to sort the first n over 2 elements using merge sort. And we're going to sort the last n over 2 elements using merge sort.

OK, I claim I have written an algorithm here, though it might not be obvious. Recursion in general is one of those bogeyman topics. It's seen as the bane of all computer science students. 'Woe unto me.' 'The end is nigh.' It's the hardest-ever topic.

And I am here to tell you it's not some new, complicated, scary topic. It's an old scary topic. No, it's an old familiar topic. It's just induction. I claim everything we're doing here is just going to be induction. And let's talk about what we mean by that.

So let's run through merge sort and see what it looks like. So we have our scrambled list, which is a power of 2-- which is not a power of 2-- which is a power of 2. Whoops.

[LAUGHTER]

All right, last helper for the day. Thank you, Charlie. Now let's scramble this up and see if we can sort them together. So here's Butternut over here. All right. So let's run through our algorithm and see what it tells us to do.

So merge sort on eight elements. So we're calling merge sort on eight elements. I'm going to abbreviate over here just to keep track of where we are. And it says the first thing to do is merge sort the first four elements, then merge sort the last four elements, then merge them together.

So we're going to merge sort 1 through 4. And then we're going to merge sort 5 through 8. And then we're going to merge. That's what the algorithm tells us to do. So it tells us, take these four-- I did not do a good job scrambling these, did I? Let's do a little more.

OK, we'll pretend we did that. So it tells us to merge sort the first four elements, then merge sort the last four elements, then merge them together. OK, well, here we're merge sorting the first four elements.

What does our algorithm tell us to do when we're merge sorting four elements? Well, it says merge sort elements 1 through 2, then merge sort elements 3 through 4, then merge them together. Yeah? I'm just reading the code here. And that's what it tells me to do when I call merge sort on these four elements.

So it tells me to call merge sort on these two, then call merge sort on these two, then merge these together into a single list-- a single sorted list of four elements. OK, well, what does merge sort with two elements mean? OK, sort the first one element, sort the next one element, then merge. Merge sort on two elements.

Well, sorting 1 and 1 on their own. There's nothing to do. A list with one element is already sorted. So all I have to do is the merge. Merge this list with that list. And who's shorter. "P-Town," then Bubs.

OK, now this list is sorted. Merge sort on these two elements next. Here's our-- you're kind of heavy. This one over here, this one over here. We get to do our merge. So who's shorter? It's Coop. Then it's Charlie.

So where are we again? We've done this and this. Now we have to merge them. I'm just following what the code says to do. We have to merge this list with this list.

OK, between these two, who's shorter? "P-Town." Between these two, who's shorter? Bubs. Now one list is empty, so the other list just falls in place. And now we have successfully called merge sort on these four elements.

Let's do the same on the other side. So split into two and two. No, Butternut. Split into one and one. Merge them together. Totoro first. Split into one and one. Then merge them back together. "Wire," then Ralph.

Now we have two verses two that we're merging. We have "Wire" versus Totoro. That's "Wire." We have Totoro versus Ralph. That's Totoro. Oops, there we go. Then Ralph versus Butternut. Ralph, then butternut. And now we have sorted these next four elements as well.

And the last thing we're supposed to do is merge these four with those four. OK, we've got these two. It's this. We've got these two. It's this. These two, I think it's Bubs. These two, Totoro. These two-- what was it again? I think it's Ralph.

Then these two is Coop, then Butternut, then Charlie. And so we just ran through merge sort on these eight elements just by following this code. And it kind of worked, right? We got our answer in the end, and it worked.

But how can we be confident that it's going to work? More than that, how can we be confident that we've even described an algorithm? Because how can we describe the merge sort algorithm using merge sort before we finished describing merge sort?

How can we know that this recursive call to merge sort correctly sorts our list before we've proven that merge sort correctly sorts your lists? This is the kind of circular thinking that makes recursion hard to think about. But thankfully we don't have to think about it in that way. We can think about it as just induction.

We're inducting on the size of the lists that we're sorting. Merge sort on one element. We know what it does, and we know it works. It does nothing. And it's correct. If we're sorting one element, we're good.

Merge sort on two elements says we're just doing that one comparison, like we saw. This also works. Merge sort on four elements-- to describe merge sort on four elements, we call merge sort with two elements, then merge sort with two elements, then merge them together.

But by the time we're here, we already know what it means to merge sort two elements. And we know that it does the job correctly. So these recursive calls, don't think of them as calling back to merge sort, which we haven't finished describing yet. Think of them as calling a previous incarnation of merge sort.

We're calling a smaller size, which we've already constructed, because we're building them up inductively, starting from the smaller sizes and working bigger. So we know that merge sort 2 exists and is correct. And we use that to prove that merge sort 4 exists and is correct.

We use that to prove that merge sort 8, which is using merge sort 4, both its description and its correctness, to make sure that merge sort 8 works and is correct. Once we know how to merge sort eight elements, we can use that to know how to merge sort 16 elements. We can use that to merge sort 32 elements.

So all this circular reasoning, we can break that cycle by remembering that the recursive calls are actually your inductive hypothesis. We're allowed to assume that it works for eight elements when we're constructing and proving it for 16 elements, and so on and so forth. Does that idea make sense? Awesome.

All right. It can be helpful to see it all at once and not with cute fuzzy helper friends. So here's a quick animation that's just running through the whole algorithm. And every time it has an unsorted list, it's breaking it into two halves. And then by the time they come back sorted, it's doing the merge.

So we're about to merge 5, 7 with 1, 2. Looks like 1 is smaller than 5. Then 2, then 5, and 7 have no friends, so they go back in. Now we're going to call merge sort on the right half.

Here we're merging one and one to get two. Now we're splitting. And now we're merging back 0 and 6. Now let's merge 3, 4 with 0, 6.

Looks like 0 won, then 3. Then 4 is the winner, and then 6. And finally, we merge the two sorted halves with our familiar merge algorithm again.

Next, 3 beats 5. 4 beats 5. 5 beats 6. Then 6, then 7. And that is a quick view of merge sort with just eight elements. Are there any questions on how this algorithm works?

All right, still restricting to just powers of 2 for now. Well, how long does it take? We have an algorithm. I want a runtime. And just like before, we can describe our runtime with a recurrence.

Let's call it M of n, which this is an upper bound on the number of comparisons when merge sorting n elements. And we're trying to figure out m of n for now just when n is a power of 2. And we have our algorithm. It's up here.

The runtime for M of n, well, we're calling the algorithm for n over 2. And we're calling it twice. So 2 times M of n over 2. And then we're merging n over 2 elements with n over 2 elements. And that merge takes at most n minus 1 more comparisons.

And so M, which is described by this recurrence, is an upper bound on the number of comparisons we'll need. Does this make sense, how we were able to translate this recursive algorithm into this recurrence for a bound on the runtime? This isn't the exact runtime because, like we said, merge sometimes takes fewer comparisons.

If one of the lists is entirely smaller than the other lists, then you only need n over 2 comparisons. And then the other list falls into place for free. So it's going to be somewhere between n over 2 comparisons and n minus 1 comparisons. Worst case, it's going to be n minus 1. And since we're looking for an upper bound here, we'll call it n minus 1.

OK, well, we have a recurrence. We can compute some terms and aim for a closed form, see if we can get an answer there. And let me write down some terms for us. Here's n. Here's selection sort. And here's merge sort, just for comparison.

n is 1, 2, 4, 8, 16, and 32. That should be enough to find a pattern, right? So for selection sort-- that was the slow quadratic one-- we had 0, 1, 6, 28, 120, and 496. And we already know the formula for this. That's n squared minus n divided by 2.

For merge sort, we get 0, 1, 5, 17, 49, 129. And we don't know this formula yet. Gratifying to see that it's smaller than selection sort. So this looks like an improvement on our algorithm. Can everyone see, or is Charlie in the way? Sorry about this.

But looking at that, I don't really see the pattern. I can't think of a formula that gets me those numbers just by staring at them, which brings us to our other technique for trying to find a closed form when we don't know one yet. Just like with sums, we had the perturbation method when we were trying to figure out a closed form.

Well, here, with recurrences, we have what we call "plug and chug"-- not to be confused with "guess and check." Whoever named these, I don't like them-- the names, not the people. They're fine, I assume.

We have the plug and chug technique, which looks like the following. So roughly speaking, substitute the recurrence into itself. So we get expanded and expanded versions of the formula. And look for a pattern in that expansion.

Let's see if we can do that with merge sort. So we have M of n. We know I'm going to put the n minus 1 first. This is n minus 1 plus 2 times M of n over 2. That's just our recurrence exactly as it was described before.

But this M of n over 2, M still satisfies our recurrence. So let's apply this recurrence to n over 2 and replace this term with the recurrence version of that. So M over n is going to equal n minus 1 plus 2 times n over 2 minus 1 plus 2 times M of n over 4.

So I applied our recurrence to the input of n over 2 and substituted that in here. Are we clear on how I got from here to here? This is the key step.

We can expand this out a little bit. So this is n minus 1. This is-- 2 times that is going to be n minus 2. And then we have a 2 and 2 plus 4 M of n over 4.

All right, so we expanded it out one step. And now we have this recurrence, which still describes the same sequence. And we can do it again. Let's do it again.

This is n minus 1 plus n minus 2 plus 4 times n over 4 minus 1 plus 2 times M of n over 8. Pretty sure I did that right. We can expand that out a little bit.

So M of n is still equal to n minus 1 plus n minus 2 plus-- and now we have 4 times this, which is going to give us r minus 4. Plus, there's a 4 and 2. We have 8 times M of n over 8.

And if we keep doing this, we can guess at the pattern we're going to see when we expand this further and further. We can guess that M of n is going to look like n minus 1 plus n minus 2 plus n minus 4 plus n minus 8 plus n minus 2 to the t minus 1 plus 2 to the t times M of n over 2 to the t.

If we expand this many times, it looks like we're going to end up with this. This was just me guessing at the formula we're going to get, but informed guessing by how this keeps expanding. And remember, we're already assuming that n is a power of 2. We're assuming that n is 2 to the k.

So what happens if we choose t equals k? Then M of n would equal n minus 1 plus n minus 2 plus n minus 4 plus r minus 2 to the k minus 1 plus 2 to the k M of n over 2 to the k.

OK, but 2 to the k is n. And n over 2 to the k is 1. So this term here is n times M of 1. Yeah? All right, let me make sure I'm still matching my algebra.

And then, from here, we have a sum to deal with. Let's take a look at all of these n terms. How many are there? Well, it's 2 to the 0 up to 2 to the k minus 1. So there should be k of them. Yes?

AUDIENCE: Why is that n times M of 1 instead of erase the k [INAUDIBLE] by 1?

ZACHARY ABEL: Instead of what?

AUDIENCE: Erase the k [INAUDIBLE]

ZACHARY ABEL: 2 to the k? So 2 to the k is n. We are assuming that n is a power of 2 and k is that exponent. So these two are the same term. Good question.

So let's see what we got. How many n's did we have. We have k of them. This is k times n. Now let's deal with these parts that are being subtracted. Minus 1 plus 2 plus 4 plus up to 2 to the k minus 1.

And then this term here, what was M of 1 again? Yeah? I see someone going like this. Excellent. 0, yeah. There's nothing to do when we have just one element. M of 1 is 0.

So that whole term, in fact, disappears. And we're just left with this, which has kn and a geometric series. And we know formulas for geometric series. I'll spare us the details.

This is going to be equal to kn minus 2 to the k minus 1. And if we remember what k is, k is log 2 of n. This is n times log 2 of n. That's this term.

And then this 2 to the k is just n. So minus n plus 1 is our formula. And we got that not by looking at the numbers but by expanding out the recurrence and looking for a pattern there.

Of course, this is still just a guess. We didn't really explain why this thing is true. We would maybe want to prove that by induction. But easier option-- once we have our actual guess, let's just prove this by induction.

Once we have our guess, we can check it. And it turns out this is the correct formula. There's your check. Questions on that? Plug and chug method-- really useful method. Doesn't always bear fruit, but often does. And it's a nice tool to have in your back pocket.

One bit of advice when using plug and chug, by the way, is notice when I had this n minus 1 and n minus 2, I didn't combine them. I could have written that as 2n minus 3. And then the next one would have been, what, 3n minus 7? But maybe the pattern is less clear if we did that.

I tried to keep terms from the different levels of recursion separate from each other so I can keep track of what's coming from each stage. So when you're doing plug and chug, try not to simplify everything at the beginning. It might help you see the pattern more readily.

All right, back to merge sort. Finally, what if n isn't a power of 2? I claim the algorithm works pretty much as described. We don't really need to change it all that much. At least we don't have to change the idea, just the measurements.

So we're going to sort the first n over 2 floor elements. We're going to sort the last ceiling of n over 2 elements. And then merge them together.

So this notation, floor notation, this means n over 2 rounded down to the nearest integer. So if n was 11, we would be sorting the first five elements and then sorting the last six elements. And we'd again have two sorted lists, now with different sizes. But that doesn't matter.

Merge doesn't care what size the lists are. We can merge them together. And this is our actual implementation of merge sort no matter what the size is.

And we can similarly bound its runtime. Now m of n, the number of comparisons when merge sorting n elements-- or at least an upper bound on the number of comparisons-- well, now we're recursively calling floor of n over 2 and ceiling of n over 2, and then still doing a merge with n total elements.

We can be conservative and say the number of comparisons is at most 2 times the amount of comparisons it takes to sort the bigger of the two halves. If the other half is smaller, then presumably that's even fewer comparisons-- so this recurrence and then, still, plus n minus 1 for the merge at most.

So this is a conservative measurement of how many comparisons we need in the worst case for n elements. Are we OK with how we got here? Nice. And now we could try the same thing.

We could try to plug and chug or something or guess and check and see if we can get a closed form for this recurrence here. I don't want to, especially the floors and-- or the ceilings, rather. They get in the way. They make it a lot harder to do something like that.

Thankfully, we have yet another tool. Turns out algorithms like merge sort, more generally what we'll call "divide and conquer" recurrences-- sorry, algorithms-- there are lots of divide and conquer algorithms very similar to merge sort that give runtime recurrences very similar to merge sort.

"Divide and conquer" means that usually whatever set of data you have, you're dividing it into smaller groups. There might be two. There might be multiple. You're recursively calling your algorithm on each group, potentially. Possibly your group's overlap. So you need more recursive calls.

And then, after you get information about each of your subsets, you somehow combine them back together to get your answer for the full list. So merge sort is a good example there. You'll see some others in recitation and problem set for other divide and conquer algorithms.

We're not so concerned with algorithm design in this class. But tools that will be used for designing algorithms in the future are very appropriate. And I'm singling out divide and conquer algorithms because we have a really nice tool that helps us analyze the runtime of divide and conquer algorithms like this. It's called the master theorem.

It's called the master theorem, and it looks like that. It's a lot of text, so I didn't want to write it by hand. What the heck is going on here? Well, the point is the following.

Suppose we have a recurrence. So we'll call it a divide and conquer recurrence, is going to be one that looks like this. It's going to be T of n equals some constant a times T of the ceiling of n over b plus some function f of n, where a is-- let's see, what are our conditions? a is at least 1 and b is greater than 1, strictly greater than 1, and f of n is some function.

Merge sort certainly fits this. a is 2. b is 2. f of n is n minus 1. And the master theorem says if you have a divide and conquer recurrence that looks like this, if you check just a couple quick conditions on how quickly f grows, I can use that to tell you a theta bound on T.

It's not giving me an exact closed form. It's giving me an asymptotic bound. It's going to say that T of n is in theta of something. There are three cases where the master theorem can tell you explicitly T of n is theta of this nice simple thing.

All right, so that's the idea. Let's not prove that right now. But let's talk about what the three cases mean and where they come from and get some intuition for the master theorem.

And if we think about a divide and conquer algorithm like this, it says, well, at the very top, we have n elements in our list. And we're going to do some work with recursive calls. That's the a times T of thing term.

But then we're going to do some work with these n elements themselves. And that's f of n. So we're going to do f of n work at this node. And all the rest will be done in the children if we're thinking about this as the recursive call stack, call tree.

Here's where we're dealing with T of n. Here's where we're dealing with T of n over b. And there are "a" different children. In merge sort, we're splitting it into halves. We're making two recursive calls. In this example, maybe there are three recursive calls. So a is 3.

And at each of these nodes, we are calling f of n over b. And each of these is making some recursive calls. Each of these nodes is doing work f of n over b squared. Then, again, f of n over b cubed. And so on all the way down-- f of n over b to the h, however tall this tree is.

So each of these nodes is doing this amount of work in itself. And the rest of the work is done recursively. How many nodes do we have at each level? Well, we have one node here, "a" nodes here, "a" squared nodes here, "a" cubed nodes here, a to the h nodes here at the bottom.

So we're getting more and more subproblems. But the sizes of those subproblems are getting smaller and smaller. To put this in context, for merge sort, we called merge with n elements at the very top. We had two cases where we called merge with n over 2 elements, four times where we called merge with n over 4 elements, eight times we called merge with n over 8 elements, and so on and so on.

Does this idea make sense? This is a conceptual view of all of the different recursive calls in our divide and conquer algorithm. And this sum here, fn times 1 plus f of n over b times a, this times a squared, this times a cubed, all the way down to the bottom, this is our runtime.

I'm being sketchy here. I'm not dealing with the floors and ceilings. This is just intuition anyway. So I'm going to be a bit sloppy. I apologize in advance. But this is conceptually our runtime.

First of all, how tall is this tree? What is this h? Well, we're going down until-- well, with merge sort, we're going all the way until we get a single element, we sort a single element. And that's our base case. And we're certainly not going beyond that.

Well, what does that say here? That says n and b to the h are equal to each other. In other words, the height of the tree is log b of n. All right, and now we have the sum. And this is our runtime.

The master theorem says that this sum often has three familiar forms, one of three familiar forms. In the first case, in case 1 of the master theorem, it says if fn is big O of this thing-- ignore what this thing is-- if fn is big O, if fn is small-- so if f of n is small, imagine in the extreme case fn is just 1. There's only a single step to do at each of these nodes.

Well, then our runtime looks like 1 plus a plus a squared plus a cubed-- cubed-- plus a to the h. If the f's are insignificant compared to the a's, then our sum is basically this. And this is a geometric series, which is basically as big as a to the h itself, up to constant factors, lower-order terms. We have a geometric series formula for that.

But basically it's saying if fn is small, then this last term dominates the whole sum. We have these two conflicting effects where the f's are getting smaller because the inputs are getting smaller. But the powers of a are getting bigger. And so case 1 is describing the case where the a's grow faster than the f's shrink.

And it turns out that in that case, the runtime is basically just a to the h itself, that T to the n is in theta of a to the h, which is-- a to the h is, what? a to the log b of n, which, by log rules, is the same as n to the log b of a.

And that's exactly the result from case 1. If the a's overpower the f's, then this last term wins. And that basically gives us our asymptotic runtime.

On the other extreme, if f grows fast, if f is big, so it grows really quickly, then the f terms over here are going to overpower the a terms. And we end up getting the opposite effect, where this first term is the biggest, and all the rest don't really contribute much asymptotically.

So if f is big, then the first term in our sum dominates. And we end up getting T of n-- T of n is in theta of f. And that is what case 3 says.

So if f grows really fast, then f alone is where most of the work happens. In terms of our tree, that means that the work we had to do in this top node, fn, that's the bulk of it. And everything else we did down here is small in comparison.

All right, now there's a case 3-- I want to keep that tree. So let's come over here. Sorry, there's a case 2, the middle case, of master theorem that says, what if these two effects balance each other out?

And in this case, it's basically asking, what if all of these terms, so fn times 1, fn over b times a, fn over b squared times a squared, all of these terms in the sum, what if they're all basically equal to each other? So in the middle, all terms of the sum are basically equal.

And so we end up getting that the sum is-- well, there are h terms here, h plus 1 terms. And each of the terms is basically the same. So I'm going to take this where it's a to the h times-- well, n over b to the h is 1, right?

So this is basically, well, h plus 1 times-- what was that last term again? a to the h times f of 1. If all the terms are basically equal, then it's just the number of terms times this last one.

We're doing asymptotics. I don't care about that plus 1. f of 1 is a constant. I don't care about that. So this ends up being basically a to the h times h. This is theta of.

And if we remember what h was-- sorry, if we remember what a to the h was, that's n to the log b of a. And if we remember what h was-- times another factor, log b. Sorry, yeah, log b of n.

So a to the h is this power of n. h is log of n. It's basically saying in, case 2, if things balance out nicely, you just incur an extra factor of log n from what you would otherwise expect from case 1. And turns out merge sort is in that middle ground.

It's in case 2 because fn here is n minus 1 for merge sort. And it's asking, is this theta of n to the log base 2 of 2? That's the condition for case 2 to apply, according to master theorem.

And, yeah, n minus 1 is in theta of n. Its exponent is 1. So, yes. So we're in case 2. And this says that master theorem of n is theta of n to that exponent-- that exponent is 1-- times an extra factor of log n.

So that's how we can use the master theorem just by pattern matching on the shape of the recurrence. a is 2. b is 2. fn is n minus 1. Make sure it satisfies the condition for case 2. And in that case, we can conclude that Mn is theta of n log n, which is the formula we found earlier, at least when we were only dealing with powers of 2.

We had n log n plus a linear term, minus a linear term. And so this matches what we had in general. It's less informative. It's only a theta bound, not an exact formula. But it still gives us what we would expect, which is nice.

Last thing I want to say about this, merge sort does have gaps-- sorry, not merge sort. Master theorem does have gaps. If you have a divide and conquer recurrence like this, you can't always just apply master theorem.

Sometimes you have cases that don't fall into any of the three cases of master theorem. Case 1, 2, and 3 are all false. Those conditions are not valid. So you can't apply master theorem at all.

So master theorem is not snake oil that will solve every problem. It'll solve many problems. And in 6.121, you'll be using it quite a bit. And you'll see how useful it can be in practice.

Final thought. Let's come back and compare some-- yeah, I'll just use this-- compare some of our recurrences and the growth rates that they describe. So we had, for example, Fibonacci numbers. Fn is Fn minus 1 plus Fn minus 2.

We had Towers of Hanoi, where Hn was 2H of n minus 1 plus 1. This was Fibonacci. This was Hanoi. We have merge sort-- merge sort of n, which is 2 M of n over 2 plus n minus 1. And other divide and conquer recurrences-- I'll use "D" for divide and conquer. This is a times D of n over b plus f of n.

And if we take a look at what formulas we got for these, this Fibonacci, this was theta of 1.6 to the n, that 1 plus root 5 all divided by 2 to the n. The Fibonacci numbers grow exponentially. The Towers of Hanoi grow exponentially.

Merge sort is very small-- n log n. And the main difference here is that, in each of these recurrences, we're making recursive calls to smaller terms. But how quickly are those smaller terms decreasing?

In merge sort, every time we make a recursive call, we cut the size in half. We cut n in half. So there are only logarithmically many times that we can cut it in half before we bottom out. So that's why our call tree over there was only logarithmic height. And so the runtime ends up being pretty small because there's not much of a tree to deal with.

By contrast, with Hanoi and Fibonacci, every time we make a recursive call, we're only subtracting 1 or 2. And so we have to do that linearly many times, like n times, before we get down to the bottom. And every time we make this call, the tree is getting exponentially bigger. It's doubling every time we do this, basically.

And that's why, in these two cases, we're seeing exponential growth, because the tree is huge. So just thinking about this recursive call tree, which terms are calling which other terms and how far down does that go and how many nodes are there, is a great way to get intuition and a rough idea of how quickly these things are growing, and a great reason why divide and conquer algorithms in general are often really good, because they bottom out quickly.

That is everything I wanted to say. Thank you so much. We'll see you in recitation tomorrow. And good luck studying for your quiz.