TOM LEIGHTON: This week we're going to talk about recurrences, how to set them up, how to model a problem as a recurrent problem, and then how to solve them. This stuff is really useful when you get into algorithm design and algorithm analysis. 6006 and 6046, we you use this stuff a lot.

We're going to start with a very famous recurrent problem known as the Towers of Hanoi problem. And I'm guessing that many of you have seen this problem in one form or another. In the Tower of Hanoi problem, you've got three pegs and n disks, these little guys here. All different sizes. And the object is to start with a stack like this and move the disks around so they wind up with the stack on another disk.

Now, the rules are you can only move one disk at a time. And you can never put a big disk over a little disk. So that you can't do. So that would have to go there. Then you could do this. And now I've gotten the top two moved over to here. And the goal is to take the minimum number of moves to get the whole stack moved over to here, and to even figure out if that's doable.

Now, this was invented by a famous French mathematician named Edward Lucas in 1883. And in the original legend around this puzzle, there were 64 disks. This one has just seven on it. And the story was at the beginning of time, God placed the 64 disks on one of the three golden pegs.

And he got an order of monks whose lifelong mission was to move the disks one at a time so at the end all 64 disks were lined up on an adjacent peg. And according to legend, when that task was done the tower would crumble and the world would end. Sort of explains why monks are never smiling, because they work forever and then just something bad happens.

Now, the question we want to be able to answer is, how long till the world ends? So we can start with a simpler version. Say let's take three disks and start with that. And let's see, is it doable? Well, we could. That's one, that's two, three, four, five, six, seven.

So it's doable with three disks. And we took seven moves. Maybe there's a better way, and that's part of what we'll try to study today. In fact, what we'd like to know is how many moves for n disks? And is it even finite? So we're going to define t sub n to be the minimum number of moves for n disks. So to take a stack of n disks and move them to an adjacent peg.

So, for example, what's t1? One. I got one peg, I just move it. That's easy. t2, 3. Move the little guy, move the base, move the little guy back on top. That's the best you can do. And we've seen that t3 is at most 7. Maybe there's a better way.

But let's try to first get an upper bound on this. Any thoughts about how we might go about a general upper bound for n for this problem? Like say I had four disks. Any thoughts about a strategy where we might be able to analyze or get an upper bound the number of steps for four disks here? Or more disks. Any thoughts? Yeah.

AUDIENCE: You can look at the previous problem and then add a certain number of [INAUDIBLE].

TOM LEIGHTON: Look at the three disk problem. That's a good idea. In fact, you know, I could look at the three disk problem recursively. I got these three disks. Say I move them over a series of steps, seven steps at most, to put them here. What would I do next? Move the lower one and then do this recurrent problem again back.

All right, so we can take a problem with n disks and solve it recursively. Let's see what happens when we do that. So let's look at a recursive solution. And let's draw it for n. So the first phase that you described is I start, here are my three pegs, and I start with this, 1, 2, 3 down to end here, and nothing on these pegs.

And I'm going to use recursion to take the top and minus 1 and move them out here. So I leave the nth guy there. Nothing here. And I go 1, 2, to n minus 1. How many steps does that move take, that series of moves take, that phase? In terms of t. t of n minus 1, because I'm taking an n minus 1 disk stack and moving it. So that is tn minus 1 steps.

All right, now my next move is to take this guy and stick him here. So we'll draw that over here. So the nth guy moves there and I leave all these guys here. That's going to be phase two. I move the largest disk. And how many steps did that take? One step. Just move one disk. One step for that. And then my last phase is to take all these and move them onto here. And so then it looks like the final result, 1, 2, over n minus 1 over n. And how many steps this phase three take? tn minus 1. Good.

All right, so we've got an algorithm now, a recursive algorithm. And the total time, the number of moves tn is at most tn minus 1 plus tn minus 1 is 2tn minus 1 plus the one step up there. For example, which we already know, t3 is at most 2t2 plus 1, which is 2 times 3 plus 1 is 7. So we already knew we could do it in seven. What's an upper bound on the time to move four disks? 15. OK? All right, so t4 is at most 15.

But maybe there's a better way. This is just one way to do it. How many people think there's a better way than this approach? How many people think there is no better way? All right, very good. Yeah, this is the best you can do. And let's prove that. Let's get a lower bound.

All right, to see why this is optimal, think about the step where the nth disk moves. At some point in our procedure, the biggest disk has to move. Now, it could move to this peg or that peg, doesn't really matter. But when it moves, what do I know had to have happened before that guy moved? Everything on top of it had to move to the other peg. I started with 1 through n minus 1 here. None of them can be here. They all have to be here. OK? So that means that I have to have had tn minus 1 steps. So at least tn minus 1 steps before the big disk moves. All right?

Then there is the step when the big guy moves. One step for the big guy to move. Because it's got to move at least once. And then after it moves, I got to get everything else, one way or another, on top of it. At some point after the last time the big guy moves, everything else has to go on top. And that's another tn minus 1 steps.

So after the last time the big guy moves, you've got to take tn minus 1 steps. OK, so that means there's a lower bound that no matter what you do tn is at least 2tn minus 1 plus 1. And so in fact, it's equal. tn equals 2tn minus 1 plus 1. So t3 is 7, t4 is 15. Any questions about that?

All right, well we'd like to know how long till the world ends, which is t64. So we'd like to get a formula for t of n. We could just compute it 64 times and see. But much nicer just to get a formula. And that's what we're going to learn how to do this week is to get formulas, closed form expressions, to solve recurrences. And there's a bunch of different methods you can use and we're going to talk about them. The simplest is to guess the answer and then check it. It's called guess and verify, also known as the substitution method.

Now, what's a good guess for t of n? 2 to the n minus 1. You got t of 1 is 1, t of 2 is 3, t of 3 is

7, t of 4 is 15. You won't compute too many before you guess that t of n is 2 to the n minus 1. Now, that doesn't mean it's true.

So how are we going to verify that guess? What proof technique? Induction. It just keeps coming back. You just never finish with induction. So we're going to verify this by induction.

What's our induction hypothesis? Any thoughts? Right there. p of n is going to be that tn equals 2 to the n minus 1. Very typical. So the predicate is that t of n is 2 to the n minus 1. Base case. We'll pick n equals 1. T of 1 is 1 and that is 2 to the 1 minus 1. So the base case works.

Now we'll do the inductive step. We're going to assume p of n. We're going to assume that tn equals 2 to the n minus 1 to prove pn minus 1, which is tn plus 1 equals 2 to the n plus 1 minus 1. So a very standard induction. So let's look at tn plus 1 now. tn plus 1 equals 2n plus 1. Yeah, 2tn plus 1. Now we plug in here. That's just the recurrence. Now we plug in using our assumption from the induction. This is 2 to the n minus 1 plus 1. Multiply the 2. That's 2 to the n plus 1. Minus 2 plus 1 is minus 1. And so we're done.

All right, so we guessed the answer. It was a pretty easy guess. Very simple to prove it by induction. Any questions?

All right, how long would it take me to do all seven disks on this puzzle here? How many steps? 127 steps. How long till the end of the world? A long time. 2 to the 64th minus 1 is 18 billion, billion moves. So going to keep those monks busy for a little while before the world ends.

Now in general, this method is sort of the best method for solving a recurrence. We'll see lots of examples where it works great. The only problem is it requires that divine insight thing, that guess to guess the right answer. Obviously if we guess the wrong answer, it's not going to work. Guessing is many times easy, but not always. So you need other methods when you can't guess it.

The next most common method has a lot of names. It's called plug and chug, expansion, iteration, brute force, exhaustion. Let me show you how that works in this example.

OK, so plug and chug. So we'll solve the same recurrence. So we've got tn equals, and I'm going to write it just reversed, 1 plus 2tn. Because I'm going to expand out the recurrent term.

That's why it's called expansion also. Well, that's equal to 1 plus 2. Now I expand this out using the recurrence. 1 plus 2 tn minus 1. All right, now I do a little chugging here. So I plugged here. Now I'll do chug. That's 1 plus 2 plus 4 tn minus 1. Now I'm going to plug in again. What's that?

AUDIENCE: [INAUDIBLE].

TOM LEIGHTON: You're right. You're right, I've got to get minus 1 here and minus 2 here. That is correct. Very good. I would have run into trouble. That's good. Now I plug in here. 1 plus 2 plus 4, 1 plus 2 tn minus 3. Then we do the chug. 1 plus 2 plus 4 plus 8 tn minus 3.

All right, and now we're going to try to observe the pattern we're getting. So there's a little bit of a guess going on in this context still. But you see the pattern. We've got powers of 2. Pretty easy pattern. So you might say if I keep on doing this, I'm going to get 1 plus 2 plus 4 plus dot dot dot plus 2 to the i minus 1 plus 2 to the i tn minus i. Got 2 to the 3 tn minus 3 there.

And then I take this all the way down to i equals n minus 1, 1 plus 2 plus 4 plus 2 to the n minus 2 plus 2 to the n minus 1 t1. And t1 is just 1. So I'm just getting the sum of the powers of 2 up to the n minus 1. And we know what that is from last time, for sure. That's just 2 to the n minus 1.

So we've derived the answer with only a little bit of a-- we had to observe the pattern here, which is often easy to do. But now we've derived the answer. Now to be really safe, we've got to go back and still do verify, just to make sure that we didn't make a mistake in how we did the derivation here. OK? Any questions about that approach? You just keep plugging it back into the recurrence, look at the pattern you get, and try to guess it from that point and then solve.

All right, let's do a little more interesting recurrence. This is a fairly famous one that comes up in a well known algorithm for sorting called merge sort. This is an algorithm you'll look at in 6046, one of the many sorting algorithms you'll study when you take an algorithms class.

So you're given n numbers and your task is to sort them, put them in order. And you're allowed to do comparisons between the numbers. You can think of them as you've got a list of names, putting them in alphabetical order. How many comparisons do you have to do to make that happen? And this is one way to do it that's very efficient.

So to sort n items. In fact, we're going to look at n being a power of 2. Call the items x1, x2, xn.

n's going to be a power of 2 to make it simple for now. The first thing we're going to do is recursively sort the first half of the items and the second half. So we use recursion to short the first n over 2 items and the last n over 2 items. And then we're going to merge those two sorted lists together. So we're going to merge this list and that list.

So let's see how this works and figure out how many comparisons we have to do to sort the list. For example, let's say we're sorting this list. 10, 7, 23, 5, 2, 4, 3, and 9. So I've got eight numbers that I'm going to sort. So the first step is to recursively sort that list. And when I do that, I produce, so I'm sorting 10, 7, 23, 5. And that produces, let's see, I'm going to get 5, 7, 10, 23. And then I recursively sort these guys. And that produces 2, 3, 4, and 9.

I've done comparisons, but haven't counted them because they're in the recursion. Now I'm going to use comparisons to merge the lists, And the way I'm going to do that is to look at the smallest item in each list and compare them. Because I know the smallest item overall is either this one or that one.

And when I compare 2 to 5, 2's the smaller one, so I start with 2. And I strike it. What two items do I compare next to get the next smallest item? 5 and 3. Smallest remaining guys. I get 3. Scratch that guy.

And I compare 5 and 4. Get 4, scratch him. And 5 and 9. 5 is the smallest. 7 and 9. 7. 9 and 10. And actually now I don't really have to do any more comparisons. All I've got is one list left. So I can just send these guys down. And I produced a sorted list.

Now, how many comparisons do I have to do in the worst case if this is n over 2 items here and this is n over 2 here? How many comparisons do I have to do to produce the n item sorted list after I have the sub list sorted? What is it? n. Very close. n minus 1 is right, because every time I do a comparison I pull an item and stick it in the list.

But once I get down to the very last item, just one guy sitting there, I don't have to do a comparison. It was just the last guy. I know he's the largest. So it could go up to n minus 1 in the worst case. In this situation, I actually did n minus 2 because I was left with two guys at the end, I just bring them down. But it might have been that the next largest thing to 23 was here and then I would have done n minus 1. Any questions? Yeah.

AUDIENCE: [INAUDIBLE]. Would that make it faster?

TOM LEIGHTON: Well, that's what's going to happen with the recursion. Because the first thing we're going to do is when we unfold the recursion is you're going to take the first half, split it in half, recursively sort them and then merge them. So the answer is that's going to happen, but we're sort of going to hide it by the recursion. All right, you didn't see me produce that sorted list. But I would have done it by taking 10 and 7 sorting them, 5 and 23 and sorting them and then merging them. OK, any other questions?

All right, well let's set up the recursion to count how many comparisons are needed for the whole thing, not just the last merge phase. All right, so let's define. And instead of t sub n, I'm going to write t of n as a function. Makes no difference, but you'll see people do recurrences both ways. t of n or t sub n. It's going to be the number of comparisons. Use my merge sort to sort n numbers. In the worst case, worst case.

All right, so we already know that merging takes n minus 1 comparisons in the worst case. And you always want to prepare for the worst case. How many comparisons are used in the recursive steps to sort this list and to sort that list, in general. tn over 2 to do this list times 2. Because you've got two lists of size n over 2. Two sub problems besides n over 2. Each takes t of n over 2.

All right, so you've got 2 tn over 2 comparisons for the recursive sorting. That means that t of n equals 2 t of n over 2 plus n minus 1. And we should say what t of 1 is. What's that? What's the time to sort? 0. You don't do any comparisons if there's one item. So we'll set that as our base case.

All right, so let's find the answer. So the first thing to always do when you see a recurrence is to take a few values, compute a few values, and try to guess the answer. So what's t of 2? 1. Plug it in. I get 2t of 1, t of 1's 0, plus 2 minus 1. That's 1. And we're only doing powers of 4 here. Sorry, powers of 2. So I go to t of 4 next. What's t of 4? 5. Right, because I got double this is 2 plus 4 minus 1. That's adding 3. So I get 5.

All right, let's look at t of 8. That's twice 5 plus 8 minus 1. 17. Not so clear what to guess here. We can try one more. T of 16 equals 2 times 17 plus 16 minus 1 equals 34 and 15 is 49. Is that right? 34 and 15.

Any guesses? You're not going to guess this. That's not going to work. That happens. And a miracle could happen, you get a divine insight and guess it, but I think it's not likely.

All right, so let's go to plug and chug. See if that works. All right, so now plug and chug. All right, so we write down t of n is-- and you always write their current part last for plug and chug to keep it simpler. 2 t of n over 2. And then I substitute, I plug in for t of n over 2. So I have n minus 1 plus 2 n over 2 minus 1 plus 2 t of n over 4.

Then I chug. So I have n minus 1 plus that's n minus 2. Plus 4 tn over 4. All right, then we'll plug in here. And I get n over 4 minus 1 plus 4 tn over 8. And now we chug. That's an n here minus a 4. I did something wrong. That should be 2. 4 times 2 is 8 tn over 8.

All right, can you start to guess the pattern here maybe? We probably should do one more to be safe. But you got an n minus 1 plus an n minus 2 plus an n minus 4. And if we do one more, what's the next term going to be? n minus 8. And this is looking pretty simple. So we could guess now that the pattern is n minus 1 plus n minus 2 plus n minus 4 all the way out to plus n minus 2 to the i minus 1 plus 2 to the i t n over 2 to the i.

And now we go all the way down where i equals log n, base 2, and see what we get. 1 plus n minus 2 plus all the way down n minus 2 to the log n minus 1 plus 2 to the log n t of 1. What's t of 1? 0. That's nice. That goes away. All right, so this is not looking so bad. This equals the sum i equals 0 to log n minus 1 of n minus 2 to the i. All right, and this will be good practice maybe for towards the test. Well, I can split that sum up. i equals 0 to log n minus 1 of n minus the i equals 0 log n minus 1 of 2 to the i.

Well, this is easy. I'm just adding n up a total of log n times. So I have n log n. This is just something the powers of 2 which we know how to do. So that's going to be 2 to the log n minus 1. And so we get our answer is n log n minus 2 the log n is just n minus 1 is a plus 1.

Well, we derived an answer. Let's just go back and check some of the values. n equals 1. That's 0 and I get 1 minus 1. It works. n equals 2. 2 log 2 is 2 minus 2 plus 1. Works. Plug in 4, I get 4 log 4 is 8 minus 4 is 4 plus 1 is 5. So it works for those examples.

Now, to be really careful we should prove it by induction. Because when you do enough of these equations, you make mistakes like I invariably do even up here. And even though it's written down here, I'll make mistakes writing it on the board. And so you want to check by induction.

OK, so this is the answer. This is how many comparisons are done in merge sort. And the nice thing is this is growing nearly linearly. It grows as n log n. And that's why this algorithm is used

a lot in practice. Much better than comparing every item to every other item, which would be more like n squared steps. So much better than a naive sorting algorithm. Any questions about what we did there?

Just one rule of thumb when you're doing the plug and chug. Notice that I didn't try to collapse terms along the way. It makes it easier sometimes to see the pattern. I left it as n minus 1 plus n minus 2 and then plus n minus 4 rather than saying, oh, this is 3 n minus 7, which might have made it harder to guess the pattern. So you don't always, when you're doing the chugging, collapse everything down. You want to try and see if there are some. Because this is the amount of work done at each step, and you want to see if you can see a pattern in that.

All right, so let's write down our two recurrences and sort of compare them. So first we did Towers of Hanoi, which was t of n equals 2 t of n minus 1 plus 1. And the answer was t of n is tilde 2 to the n. It was actually 2 to the n minus 1. Then we did merge sort, which is t of n equals 2 t of n over 2. Difference there. Plus n minus 1. And now the answer was t of n.

What's the tilde for this one? For merge sort. n log n. Yeah, I can forget the rest if I know the tilde, n log n. Wow. These are pretty different answers, right? This one's nearly linear, that one's exponential. The recurrences looks pretty similar, in fact, this one sort of looks worse because I'm adding in n minus 1 every time instead of 1. But the answer's a lot better, a lot smaller. What's the critical difference between these recurrences? Yeah?

AUDIENCE: You're dropping the argument of t a lot faster. You're cutting it in half every time instead of subtracting one.

TOM LEIGHTON: Right.

AUDIENCE: So it's going to go away faster.

TOM LEIGHTON: That's right. Here I've got two problems of size one less. Here I've got two problems of half a size. And that makes an enormous difference in the answer. Even though I'm doing more work after did the recursive problems, much faster running time or answer if these are algorithms that I'm analyzing. All right?

In fact, let's take a look at what happens if I didn't use that n minus 1, I just put a 1 there. Let's see what impact that n minus 1 had. So let's do the following recurrence. This one is going to be a little bit different because I'm not going to stick to powers of 2. I'm going to get a little

messier, because that often happens in reality. And I'm going to define s of 1 is 0. s of n is s of the floor function of n over 2 plus s of the ceiling function of n over 2 plus 1.

Now, the ceiling function means the smallest integer that's at least that big. So you're sort of rounding up to the next integer. And the floor function means you're rounding down to the next integer. So it's the biggest integer less than equal to n over 2. All right.

Now, if we had just powers of 2, this would be much easier to write. It would just be sn equals 2 sn over 2 plus 1. So it's analogous to what we're doing up there where we're just adding 1 instead of n minus 1. But I'm being careful. now. I'm going to make this recurrence work for all natural numbers. Because sometimes in reality, you're dealing with non powers of 2.

All right, let's see if we can solve this. What's the first thing to try to solve this? What do you do? What's the first step when you get a recurrence? You try to solve. What method should you try? Guess. And to guess, well, you got to plug some values in. It's sort of hard to guess. I mean, looking at that, I guess nothing comes to mind other than panic looking at that. But if we plug in some values, then maybe it won't be so bad.

So we've got s1 equals 0. What's s2? It's not 3, because this is 2 over 2 is 1, floor of 1 is 1. s of 1 is 0. Ceiling of 1 is 1 because it's an integer. s of 1 is 0 plus 1. So s of 2 is 1. What's s of 3? Yikes, all right. 3 over 2 is 1 and 1/2. What's the floor of 1 and 1/2? 1. s of 1 is 0. That's gone. What's the ceiling of 1 and 1/2? 2. s of 2 is 1 plus 1. What's the answer for s of 3? 2, because I've got 0, 1, and 1.

s of 4. This one's easy because it's even. I get s of 2, which is 1 plus s of two which is 1 plus 1. So what's s of 4? 3.

You might try a guess. What would you guess? Yeah, n minus 1. So we're going to guess s of n equals n minus 1. And that's not so bad, actually. It's pretty simple. Looking at that mess, you'd never think of that right away. But once you plug in some values it's not so bad.

All right, let's verify it. And we're going to verify by strong induction. The induction hypothesis is our guess. The base case is easy. Because I've got s of 1 is 0. And that equals 1 minus 1. So that's good. So let's do the induction step.

All right, so the induction step. We're going to assume that p1 is true, p2 is true, all the way up to pn, to prove pn plus 1 is true. For n better than or equal to 1. And then we look at the n plus first term. That equals s of floor n plus 1 over 2 plus s of ceiling n plus 1 over 2 plus 1.

All right, now I don't know what these are exactly depending on whether n is even a rod. First let me plug in using the induction hypothesis. I know that this is n plus 1 over 2 minus 1. This is ceiling n plus 1 over 2 minus 1. And then I carry the plus 1 down. That's by the induction hypothesis.

All right, now I don't know exactly what either of these values is unless I know the parity of n. But I do know the sum of these two things. The floor of something plus the ceiling of that something is twice the something. All right, so this plus that equals n plus 1. Minus 1 minus 1 plus 1 is minus 1. And so in fact, we prove that s of n plus 1 equals n plus 1 minus 1. So we verified the induction hypothesis. Yeah?

AUDIENCE: [INAUDIBLE]. Taking the n [INAUDIBLE].

TOM LEIGHTON: Let's see, I'm not sure. So are you asking why I started sn plus 1 here?

AUDIENCE: [INAUDIBLE].

TOM LEIGHTON: Doesn't matter. But the expression is s of whatever this is, the induction hypothesis would be, s of whatever's inside there is what's inside there minus 1. And so the induction hypothesis says s of this is that minus 1. Any other questions?

So let's write that on our table up here. So this is sn equals s of floor n over 2 plus s of ceiling n over 2 plus 1. And the answer is, sn is tilde. What's the tilde of? n. The answer is n minus 1. So it's tilde of n. Now, this is the same expression is this if n's a power of 2. I'd have 2 sn over 2 plus 1 instead of n minus 1. And by going from n minus 1 to 1, I got rid of the log term. But it's still not exponential. And that's because I'm cutting down the insides here by a factor of 2 roughly.

All right, so all the recurrences we've seen so far have been pretty simple. If guess and verify doesn't work after seeing a few terms, you can use plug and chug and you can figure it out. The good news is, that happens a lot in practice. The bad news is, sometimes it doesn't and you can get nasty recurrences.

So let me show you a nasty one that actually was once asked in 6046, I think, on a quiz with disastrous results. All right, so this one is t of x equals 2 t of x over 2 plus 8/9 t of 3x over 4 plus x squared for x bigger and equal to 1, and otherwise it's 0 when x is less than 1.

Now, this is nasty for a lot of reasons. First, it's not just a find on the integers. We can make an integer version by putting floors or ceilings here. But the way you set it up now, it's defined on real numbers. You can, in theory, solve it with plug and chug.

The nasty part is, you've got a bunch of things going down by powers of 2 and a bunch of things going down by 3/4 here. And so you have to keep track of all the terms that are decreasing, and that's a nightmare plug and chug. Because it's not just one thing that's unfolding, it's two things that are unfolding. So very painful.

Now, it used to be that was the state of life. 20 years ago, I think we started teaching this course sometime around then, there just was nothing else to say other than life's tough and sometimes you've got to do these things and it's a pain. Just had to make the best of it.

And then in the mid-90s, one day two students at the American University in Beirut came by my office. They were named Akra and Bazzi. And they'd actually come to Boston to escape the daily bombings. This was when all heck was breaking loose in Beirut. It was a mess. And they come into my office and they claim to have a simple formula for solving any recurrence like this. In fact, you can even add a 5 t of x over 7. They said no problem. Their simple formula just will always work.

And the general form that they claim to solve was the following. It's the general class of divide and conquer recurrences. So any recurrence you get from an algorithm where you break it into smaller pieces by a constant factor, they claim to solve. So let me define this class. It's a little painful.

So a divide and conquer recurrence has the form. There's a simple version and the hard version. The simple version is it looks like that. You got a bunch of terms. Inside each term, you cut it down by a constant factor and then you add something at the end. That's the simple version of what a divide and conquer recurrence is. And you can throw in floors and ceilings and other junk too.

Now I'm going to give you the formal version. So it has the form t of x equals some constant a1 times t of a constant less than b1 of x, plus some slop you're allowed to have like floors and ceilings and who knows what else.

And then you get more of these. You get a2, t of b2x plus epsilon 2x and so forth. And you can have any number of these you want up to a constant. So maybe there's a sub k t b sub kx plus

epsilon kx. Plus some function of x sitting out here, like the x squared. And this happens for x bigger than some fixed value, some constant.

Now, the constraints are, well, the ai's are positive constants. The bi's are positive and less than 1. You gotta cut it down by a constant factor to be a divide and conquer recurrence. That's critical. k is fixed. It's constant.

And these epsilon functions can't be too big. They can't be bigger than 0 of x over log squared of x. Almost as big as x, but you can't let them be too big or you'll not be cutting things down by a constant factor. Now, you don't get too hung up over this because you don't get too deep in the details. But this is that most 0 of x over log squared of x.

And finally, the g thing has got to be polynomial. It can't be an exponent. You can't put 2 to the x here. And the formal way that's expressed is the derivative of g of x is less than x to the c for some constant c.

That's it. So this is a mouthful. But like I said before, anything that looks like that, basically. As long as you don't throw in exponentials or do something too wild to it, it works.

So let's see which of our guys over here are divide and conquer recurrences. How about this one? Towers of Hanoi, is that a divide and conquer recurrence? Some yes, some no. Who thinks no can tell me why it's not? Why isn't that?

AUDIENCE: [INAUDIBLE].

TOM LEIGHTON: That's right. The recurrence problem is n minus 1 and we've got to make it down by a constant factor here. The b1 is less than 1. It can't be equal to 1. And this is not big enough to make up for the x over 2 term or something we're missing.

So whenever you have an n minus 1 and minus 2, that is not divide and conquer. We'll discuss those next time, on Thursday, how to solve those. So that's not divide and conquer.

What about this one? Is that divide and conquer here? Yes. You've got two problems of half the size and you're adding a linear function. It's not exponential. It's polynomial. That is divide and conquer.

What about this nasty guy? Is that divide and conquer? Yeah. You've got basically n over 2 here. Two n over 2 terms now. This is where that epsilon thing comes in handy. Because

epsilon 1 of n, in this case, is floor n over 2 minus n over 2. And that's, at most, 1.

So maybe we're adding a half here or something like that. And we're allowed to add anything up to n over log squared of n to fit. So you can mess around with the low order terms in here. And adding 1 is fine as well. So that qualifies.

Let's check that this thing qualifies. Let's check that. Well, yeah. I got 2 constant a1 t of 1/2 x. That's OK. 8/9 t of 3/4 x. That's OK. My g function is a polynomial. The derivative is 2x. That's less than x to a power or of x to a power. So this fits.

Any questions about the definition of a divide and conquer recurrence? And you can see why sort of they come up, because whenever you're solving a problem with recursive pieces that are a constant factor smaller, you're going to have something that looks like that.

OK, so getting back to Akra and Bazzi. B They show up in my office. They claim to have solved anything like this. Now needless to say, I'm a little skeptical. I got to tell you, you're a professor at MIT, especially mathematics, and people are always coming by with proofs of the Riemann hypothesis, p equals np, p doesn't equal np. And this was a classic problem.

Hundreds of researchers had studied recurrences for decades. And not only, it wasn't even believed that there was possibly a solution. It's not like it's one of these open questions where we all believe p doesn't equal np and it's just a matter of proving it. People didn't think a solution existed that would be clean. I mean, look at just the definition's a mess. And how could there possibly be a clean solution to that?

So it's sort of like the holy grail. It doesn't exist, but it's fun to think about, that fantasy, maybe there's a solution and you could find it. And much less, these aren't reputable researchers in my office. These are two students from Beirut named Akra and Bazzi claiming they've figured out this thing that all these professors been teaching this stuff in 6046 for years said you've got to do plug and chug.

Anyway, they'd been kicked out of everybody else's office, so I said, OK, let's do it. Show me the solution. And they did. And I go, holy cow. It's right. It looks good. Now, they had a few details wrong, but my goodness, the answer was correct and it's amazingly simple. So now we teach it. And in fact, you can write the solution down easier than you can write the definition of divide and conquer recurrence. Really remarkable.

So the theorem by Akra and Bazzi in '96. Set p, real number, So that the sum i equals 1 to k of

ai times bi to the p equals 1. So the ai's are these things here. The bi's are the fractions here. All right, find a p such that this sum is 1. Then the answer is this. Then t of x equals theta. We're going to use our asymptotic notation. x to the p plus x to the p times the integral from 1 to x g of u. G is that polynomial thing, that thing at the end. Over u to the p minus 1 du.

That's it. This is really great news for you guys, especially if you go on to algorithms, because you don't have to deal with plug and chug on nasty things. You just solve the answer.

Now, the proof is by guess and verify. And they made a really good guess. You wouldn't think of looking at that, oh, I'm going to guess this. Probably not. And there's no real numbers to try to guess this. It's asymptotic notation. But it works. You can verify it by induction. It's a little painful to verify by induction, but you can do it. We won't do it in class. I did try one year, and it was a disaster, so we don't do it in class anymore.

But let's see some examples of how to apply this. So let's start with merge sort, which is the second one here. Let's do that. All right, so you got an example here. t of x equals 2 t of x over 2 plus n minus 1, or x minus 1, x now. And I'm not even going to worry about powers of 2 anymore. I don't care. It's going to be fine.

All right, what is p in this case? I've only got one term. I don't even need a sum from that definition over there. What's the value of p for this guy? So I've got a1 is 2. b1 is 1/2. Just one term. What's the value of p such that 2 times 1/2 to the p equals 1? 1. Doesn't get much simpler. So 2 times 1/2 to the p equals 1 implies p is 1.

All right, now I just plug into the integral over there. All right, let's do that. So t of x equals theta. Well, x to the p is x to the 1 plus just take off the 1, plus x times the integral 1 to x. What's the g function? What's g of x in this case? x minus 1. So I take the integral of g of u is just u minus 1 over u to the p plus 1 p minus 1.

Did I write the formula right? Is it plus 1 or minus 1 here? Did I get it? p plus 1. Sorry, didn't write that right. p plus 1 over here. All right, so u to the p plus 1 is u squared du. This is theta of x plus x integral 1 over u minus 1 over u squared. The integral of 1 over u is the log of u. The integral of 1 over u squared is 1 over u, but I changed the sign there. 1 to x. All right, so I get theta of x plus x times log of x plus 1 over x minus log of 0. That's log of 1 is nothing. Minus 1.

And I'm doing thetas here so I can forget the low order terms. So I get x log x plus 1 minus x.

This is the only term that survives, x log x. And that's right. t of n is in fact tilde n log n, so in fact, it's theta n log n.

Now, in this case, going through the integral and all that, maybe that was harder than just guessing and verifying. And by the way, guess and verify got it really the exact answer. The nice thing is this works for the nasty guys too. In fact, let's figure out the solution to this thing, which I guarantee you is a pain to do without this method.

The first thing to do is compute p. So to do that, we need to find a p such that 2 times 1/2 to the p plus 8/9 times 3/4 to the p equals 1. I gotta find a p for which that's true.

Any guesses what p works? p equals 1 not going to do it. p equals 2? Let's try that. I'd have 2 times 1/4 plus 8/9 times 9/16. That looks pretty good. I get 1/2 plus 1/2 equals 1. So p equals 2 works.

All right, so let's do the integral. tx is theta of x to the p plus x to the p times the integral. All right, what is g of x in this thing? x squared. So the integral is 1 to x. g of u is u squared divided by u to the 1 plus p. u to the 1 plus 2 is just u cubed.

This is theta x squared plus x squared times, well, the integral of 1 over u is just log of you. And that's just log of x. So the answer is, x square ln of x. Done. So that's pretty good what these guys did. In fact, that is the correct answer. So really easy to pug in. Any questions on that? Yeah.

AUDIENCE: [INAUDIBLE] or does it give the actual number?

TOM LEIGHTON: It doesn't give you the actual number. It just gives you the asymptotic growth. All you know is that the limit of tx over x squared ln of x is less than infinity and bigger than 0. So it's growing as this. You don't know if it's 10x squared lnx or a million squared lnx. On the other hand, for algorithms usually you're forgetting the constant factors anyway because they depend on things to do with the CPU you're using and stuff like that. What you really care about, often, when you're studying algorithms is the asymptotic growth, not the actual value. Yeah?

AUDIENCE: [INAUDIBLE] log base e. And then above I see you using log, which I assume is base two? Does it matter?

TOM LEIGHTON: It does up there because I've got a tilde. And in the tilde, the constant factor matters. It doesn't matter in here because log base two and log base E are within a constant factor, namely log of

2, ln of 2 is the constant factor. So logs don't matter what the base is once you're inside a theta or an O or that kind of thing. In a tilde, they matter because the constant factor matters. Yeah.

AUDIENCE: [INAUDIBLE].

TOM LEIGHTON: Yeah. Because x squared is small compared to x squared ln of x because it's all by the theta.

AUDIENCE: [INAUDIBLE].

TOM LEIGHTON: No. You could put all sorts of other stuff in here and it doesn't matter because it still equals theta of x squared ln of x. There's no hidden meanings by having extra stuff in here. That's important. In this case, when you put more stuff into theta, doesn't mean a thing. Any other questions? Yeah.

AUDIENCE: [INAUDIBLE].

TOM LEIGHTON: Ah, good question. Well, let's see. It goes back to over here. See, the b's are between 0 and 1. By making p big enough, I drive the value down. By making p be large and negative, I drag the value up. So there is some value of p. But that's a good question. Let's see what happens when p is not so nice. Because the examples I gave you, in one case p was 1 and one p was 2, sort of nice. Let's look at a bad case there.

Let's look at this recurrence. 3 t of x over 3 plus 4 t of x over 4 plus x squared. All right, so that's the recurrence. And first step is to compute p. And this time it's not going to be nice. So I need to find a p such that 3 times 1/3 of the p plus 4 times 1/4 to the p equals 1.

And you might say, OK, let's try p equals 1. I get 1 plus 1 is 2. That's too big. So which way do I have to go on p? Do I need a bigger p or a smaller p? Bigger. Because I got to get from 2 down to 1. So I know that p is bigger than 1.

All right, so the next thing to try would be hoping the world is nice, p equals 2. Maybe I'll do that over here. Let's try p equals 2. So I get three times 1/9 plus 4 times 1/16 and that equals a 1/3 plus 1/4. That's less than 1. That didn't work. What do I have to do now with p? Is it bigger than 2 or less than 2? Less than 2. That's sort of a pain. P is between one and two, so it's not an integer.

Now, you can sort of do divide and conquer here. Next I'd try p is 3/2, get my calculator out

and see which way it goes. And you could sort of home in on it. But there's something really nice that happens with Akra Bazzi. And that is if you ever conclude that p is less than your exponent here, you don't even have to compute it.

Let's see why. Let's do an example. This will happen a lot. There's some p, but it's just going to be a pain to evaluate. But let's see what happens here. So I'm just going to try to solve it, write the solution anyway. t of x equals theta x to the p plus x to the p integral 1 to x. What's g of x? x squared. So that's u squared over u 1 plus p. And I still don't know what it is, but I'm going to give it a try here. That's theta x to the p plus x to the p.

Well, this is now the integral of u to the 1 minus p du. And p is bigger than 1. I know that. This is theta x to the p plus x to the p. Well, the integral here is x to the 2 minus p. Up to constants. I don't even care about dividing by 2 minus p, because I got big theta. Just put that p up here. OK

Well, this equals theta x to the p plus cancelled there, x squared. And what's the answer going to be? x squared, because p is less than 2. So this is a small order term. So I didn't even know what p was, just that it was less than two. And I got the answer. And what do you know? The answer was just this thing. That's sort of nice and simple.

And in fact, that's a theorem. Which will state, in general, if g of x equals theta x to the t, for some t bigger and equal to 0. And when you plug that value in and take your sum. Trying to compute the p value. So I take ai times bi to the t.

And that comes out to be less than 1 like it did here when I plugged in p equals 2 and I computed it and it was less than 1. But if that happens in general, then the answer is theta g of x. Plain and simple. It just turns out to be the g of x term up to constant factors.

So you don't even have to compute it. Just do that check on the power here to see is it smaller than that. I won't prove that there. It's actually not hard to do. The proof is pretty much what I did with this example, so it's not too hard. OK, any questions?

I want to show you one more thing, but just make sure that's OK. And this is something never to do with an asymptotic notation, like we talked about last time. When you're using Akra Bazzi and when you're doing occurrences with algorithm, you always are using asymptotic notation. Your answer is theta of something.

And in fact, what will start happening is you'll do this stuff because the constant factors don't

matter. Because you're wiping them out in the theta notation, what you'll see is things like this. Your recurrence will be set up to be t of n is 2 t of n over 2 plus o of n or plus theta of n. And then you'll conclude that t of n is theta of n log n using Akra Bazzi or whatever method you want. Because it doesn't really matter what the constant is on g because the constants disappear. Doesn't matter. And so people stop worrying about it early on.

All right, now here's the bad example. Let me show you a false proof like last time. And this time I think it'll be pretty easy to spot the flaw. Theorem not. If t of n equals 2 t of n over 2 plus n minus 1, this is the recurrence we had for merge sort, and t of 1 equals 0, then I'm going to prove that t of n is o event.

Now, that can't be right because we've proved it's theta then log n, tilde then log n. And that is not o of n, right? n log n grows faster than n. But let's see the proof. By strong induction. The induction hypothesis is going to be what we're trying to prove. Base case. n equals 1. t of 1 is 0. 0 is surely o of 1. The induction step. We're going to assume p1, p2, p3 up to pn minus 1 to prove pn.

And let's look at p of n. So I look at tn is 2 t of n over 2 plus n minus 1. Induction hypothesis on n over 2 says this is 2 times o of n over 2 plus n minus 1. Twice o of n over 2 is o of n plus n is o of n. This is o of n. And I'm done. I just proved that t of n is o of n.

What's the bad step? What's the bug? See the bug? What is it?

AUDIENCE: [INAUDIBLE].

TOM LEIGHTON: Induction hypothesis. What's wrong with that?

AUDIENCE: [INAUDIBLE].

TOM LEIGHTON: Right. The end from the predicate cannot be in the o of n. Because remember, o of n means, this statement here by itself means the limit as n goes to infinity of t of n over n is less than infinity. But this makes no sense if I specified n in a predicate.

So what's the rule? Same rule as last time. Same bug. What's the rule? Never do asymptotic notation in induction hypothesis. Never mix big o and a predicate. Never, ever. It looks so nice to do, but it is so wrong. It just makes nonsense.

All right, very good. So remember ice cream tonight, recitation optional tomorrow. But study

session is there.