SRINI DEVADAS: We're going to do something quite different in terms of a puzzle. And while it's a recursive puzzle, it's going to play out to be a recursive solution. It looks quite different from the N-queens puzzle. And so this is going to be a tiling puzzle.

And so I set up the puzzle, and then we're going to take a little segue off into a canonical recursive algorithm called merge sort because I think it's good for you to see that algorithm before you dive into solving this particular puzzle because it kind of shows you the way. All right, and but I'll set up the puzzle. I guess I didn't have to erase this square.

It's a courtyard that has a bunch-- of tiles. You're supposed to tile it, and it's all square tiles. And I'll just make it eight by eight like a chessboard. And in general, we are going to say that this is a 2 raised to n by 2 raised to n courtyard. And I want to tile this courtyard.

So this is a three tile-- some people call it a tromino. You can think of it's really an L-shaped tile, and it can be oriented in any different way. It could be an inverted L, et cetera.

And you're supposed to tile this courtyard using L-shaped tiles or trominos. And they're called trominos because there's three tiles in there-- three squares in there. And so that's your job. I guess kind of a weird job, but this is a puzzle.

And so you're out there with 2 raised to n by 2 raised to n. And all you have are these L-shaped tiles, and you're not allowed to break them. If you break them, then that's not considered a valid tiling. That's bad, and that's your job.

Oh, and obviously, it needs to look like this, and these look like the floor. Maybe you'd like to see the markings because these are tiles, after all. But you can't have something that is uneven. You can't have a tile on top of another tile. That's a definite no-no.

And you can't not have a tile in any part of the courtyard. So I think that was kind of obvious, but it's worth explicitly stating just so you don't get confused here. So that's the question that we're going to try and answer.

But as I said, I want to talk about something a little bit different for about 5 or 10 minutes and actually show you some code because I want to introduce this notion that's a little bit different from the N-queens recursive paradigm. It is recursive, but it's called recursive divide and conquer. And usually, when we use the term recursive divide and conquer, you might think

that it's applicable, even to the N-queens recursive code. But from a technical term standpoint when we say recursive divide and conquer, we're really saying that we're taking the problem, and we are dividing it up into smaller problems.

So you take the problem, and you divide it into two problems that are half the size or three problems that are a third of the size et cetera. Which is not what happened in the case of the N-queens problem.

And in the case of the N-queens problem, you took a column, and you actually did iterations on the column. So it potentially exploded on you. And then you had a problem that is slightly smaller than the N-queens. It was effectively n minus 1 queens.

And so it's not n over 2 or n over 2 or 3-- n over 3 in the case of a three-way recursive divide and conquer. So there's a qualitative difference. And there's also a quantitative difference in the sense that these kinds of algorithms are going to be efficient in that they are going to run very quickly in polynomial time, perhaps quadratic time. We can do some analysis for both of the merge sort that we're going to do now and for the tiling puzzle, once we get to that. But that's something to keep in mind.

So merge sort is the canonical recursive divide and conquer approach. And I'll put up code for it, but I'm not going to show it to you. You can look at it in the website.

But I'm going to describe the algorithm to you using a trivial example. And so I want to sort these numbers in ascending order.

So I have these four numbers that I want to sort in ascending order. And obviously, I could do this many different ways. We looked at selection sort. But this is a large array, and I want to do something more efficient than that quadratic algorithm and n squared algorithm. And merge sort, which is recursive divide and conquer, fits the bill.

And what merge sort says is I'm going to split the array. I'm just going to take this array, and I'm going to split it up into two sub-arrays. And I could do this recursively, but let's just talk about the first step.

So I'm going to go ahead and split this array up, and I have 76 and 32. And I'm going to call merge sort on this smaller array. And the first thing that merge sort does is split the array up into something smaller, and you end up getting that.

So this is the first thing, and that's the first split. And then you end up splitting 48 and 32, and I'm sorry-- or 12, 76, 32, 48, 12. Yep, I got that right. And so far, you've done nothing. I mean, you've just sort of broken things up and turned this into four one arrays or one list or what have you.

But then the fun starts where after you do the split, after you return, and so 76 as singleton is trivially sorted so there's no work to do. So there's no more splits that you could do. There's nothing that says you can-- I mean, 76 is a monolithic element in their array, so you can't do anything with it. And so you just leave it be. And the same thing with 32, and so on and so forth.

But after your two calls return, you have to merge the two together. And the way you merge the two together-- the invariant here is that when you merge the two together, they have to be sorted in the order that you want the overall array to be sorted, which is ascending order. So in this case, you have down at the bottom, it's trivial code because you have two singleton lists. And you just decide which order they go into.

OK, so the merge, in this case, would simply be produced, you would produce 32 and 76. OK, and in this case, the merger produced 12 and 48.

So at this level, you're going to return 32 and 76 back. You're going to return 32 and 76 back. And here you're going to return 12 and 48 back. Now it gets a little more interesting. Now it gets a little more interesting because now you have 32 and 76 and 12 and 48, and those are both sorted in ascending order, and you need to merge them together.

And so if you had a 128 element list, you would go, go, go all the way down, and then you would do some merging. Come up all the way and then the most time-consuming step really would be taking two 64 element lists that the left one is in ascending order, and the right one is in descending order. And then you need to merge them together so the result is in ascending order. That makes sense?

And so in this case, the way we're going to do this is if you have-- and I can do this for this example or a different example. But I'll just stick with 32, 76, and this could be larger. This is 12 and 48. So I need to merge these two to produce in ascending order.

And we use a straightforward algorithm that's called the two-finger algorithm, which says I'm going to start with my fingers pointing to the beginning of the two arrays, and I'm going to

choose the element that's smaller. And I'm going to write it into my output array. So I'm going to compare 32 and 12. 12 is smaller, so I'm going to write 12 down. And because I picked 12 from this array, and I didn't pick 32, I'm going to leave my top finger where it is, and I'm going to move my bottom finger-- my right-hand finger-- over to the next element.

And then I'm going to compare 32 with 48 because this finger didn't move. And I'm going to figure out which one is smaller, and it's 32. And I'm going to put that over here. And then I'm going to move this finger because I took away 32.

And then at this 48 is smaller, and then I'm off this way, assuming the array ended here. But if it didn't, I keep going.

And then the last thing I do is put 76 down. So that is a recursive divide and conquer algorithm. It's the simplest algorithm I can think of, and that is still interesting. I'll show you code for it. I'm not going to explain it.

It is worth looking at the overall code. And then the merge, which is this two-finger algorithm, is actually the most amount of code. But the split itself, as you can see, is you see as with our recursion, we say we want to have a base case. And so when the length of the list becomes 2, I'm just going to do that little flip. This is a sorting algorithm that only works for a list of length 2.

But the beauty of this is that I can keep going down to get smaller and smaller lists, such that the sorting algorithm actually works. And this is it. This is a fine sorting algorithm for a base case. And we do have a base case here.

We also have the other property that I'm actually calling. I'm making two recursive calls with significantly smaller problems. In this case, I'm using list splicing to get the first part of the list and the second part of the list, and I'm just calling merge sort on those two lists.

And then finally, the workhorse in merge sort is the merge step-- this two finger thing that I just described to you. That is this merge thing. And this merge thing, as you can see-- there's not a lot of code.

But it's doing this movement and the comparisons. And that's essentially what you have there. So given that context-- right given that context, let's go back to our tiling puzzle.

And let's talk about this tiling puzzle and how we could we could solve this tiling puzzle. So you're going to break this courtyard up clearly because that's really what I've been trying to tell

you in terms of recursive divide and conquer.

First question-- look at this 2 raised to n, and 2 raised to n, and look at this three tile. I am going to ask you the question. Is there a solution that is a perfect tiling with no overlapping tiles and no squares with holes in them? Is someone?

Fadi had his hand up first. Go for it.

AUDIENCE: Well, if we look at the case of n is equal to 1, you have a two by two courtyard, then it's impossible, you can see that trivially. But in a general case, then we will have 2 to the 2n squares. And every time we're adding 3, 3, 3, the number of tiles is a multiple of 3.

SRINI DEVADAS: Correct. So 2 to the 2n-- and Josh probably had the same answer-- 2 to the 2n is not a multiple of 3. The only thing that divides it is 2.

So OK so we're going to change the problem. We're going to say that there's a statue of your favorite person that is going to be out here. And so you don't need to tile that.

So now you have 2 raised to 2n minus 1 let me write that out properly. 2 raised to 2n minus 1 squares. Now can you do this for all n? Clearly, it works for n equals 1.

And then for n equals 2, you have what? 2 raised to 4, which is 16 minus 115. That seems to work.

And then you get up to, I think 80-- what is it? 2 raised to 6 would be 64. And so that will be 63. That works too.

In general, does it work? It does. So it turns out if you think about it, you can write this out as 2 raised to n minus 1 times 2 raised to n plus 1. And in the middle there, pretend that there's 2 raised to n. So you have three consecutive numbers.

And when you multiply three consecutive numbers, they're consecutive. Doesn't matter what the numbers are. The product is always going to be a multiple of 3 because one of those things is going to be a multiple of 3. And you know that 2 raised to n is not a multiple of 3. This is not a product of three consecutive numbers.

But if it were, and you pretended that 2 raised to n was in here, either this one is a multiple of 3, or that one is a multiple of 3. So this is good. So we know that at least from a standpoint of volume-- our surface area, if you will-- that this is going to work. Whereas the original problem

didn't work from a surface area standpoint.

So it comes down to how am I going to solve this problem using recursive divide and conquer? And so I can break this up. And you want to have a similar feel to the merge sort, where when you go down to the base case of merge sort, you had this trivial problem to solve. That's the beauty of divide and conquer. It becomes obvious when you have such a small problem like one queen.

That's not technically not divide and conquer, but you do get to the smaller part. But clearly, in this case, this is a classic divide and conquer algorithm merge sort. And you can see that the first two or three lines of code are the trivial case of two elements that needed to be sorted. So let's go back and forget that.

Let's look at this picture and think about how you'd break this up. How would I bring this up?

So think two-dimensionally. And merge sort was a single dimension. OK, think two dimensional. This is two dimensional.

This is a two-dimensional problem. It's in that sense, closer to N-queens. But merge sort was simply a single dimension. All right, I see Kevin had-- Fadi had his hand up.

Kevin has a hand up. Someone else? All right, go ahead, Kevin.

AUDIENCE: Do you keep cutting the square into quarters?

SRINI DEVADAS: Quarters-- good. OK, yeah. So all right, so let's do that. Let's do that. You're on the right track.

And by the way, this particular thing could be over there. And I'll just do this just to make sure. Your algorithm needs to work, regardless of where that statue is.

So maybe the statue is over here. Maybe the statue is over there. There's only one statue that is covering one square, but it's not necessarily in near the center. It could even be anywhere. You need a general-purpose scheme that would work, regardless of where the statue is as long as it only occupies one square.

That's the only constraint we have. So you're on the right track, Kevin. Someone else or, Fadi, you want to add to that?

AUDIENCE: So in something that's similar to the merge sort so you divide it into quarters and then you

reach the base case and then we tile that obviously with an L-shaped tile.

SRINI DEVADAS: Right, that's exactly right. So yeah. So that there's one thing that's missing in what both you and Kevin have put up. The thing with merge sort was I had exactly the same problem to solve.

It was exactly the same problem. It was just smaller. I needed to sort the smaller array in ascending order. And I had two of them, and it was exactly the same problem. But when I split this up into four quarters like you guys are suggesting, do I have exactly the same four problems to solve?

In fact, can I even solve this the 16 problem with L-shaped tiles? I can't. So at some level, I have three problems that I can't solve. And then a smaller problem that I don't quite know how to solve-- other than maybe splitting it up even more, which is kind of interesting, which is good. But then the next time, if I go ahead and split that, then wonderful. I will find one problem that's two by two that I can solve, but that doesn't solve my puzzle.

So I have to do one more thing in order to make this look like merge sort or any recursive divide and conquer. One of the paradigms of recursive divide and conquer is you're not creating new types of problems. You're creating the same type of problem, except smaller. And therefore, eventually easier.

So I have to do one more thing. And this is an aha moment here that I'm hoping-- I don't want to take an answer yet because I want you to think about it and perhaps arrive at it. But it's one of those things where you're going to go "oh!" if you don't know the answer. But it is worth spending another minute-- another minute for each of you thinking about it.

How am I going to set it up so these quarters-- so all of this is good? I'm going to split it up into quarters, but I want all four problems to look exactly the same in the sense that they look-- this obviously looks different from these other ones. Well, there's a tile missing here, but there is no tile missing over in the other three. So how do I make all problems look the same? I know you know, but we need more. I'll give you 30 more seconds, and I'll give you one more hint.

All! right, the hint is place one tile-- place one L-shaped tile. All right. Yeah, go for it.

AUDIENCE: Yeah, you know that with those last four quarters-- you know that that L-shape has to go around?

SRINI DEVADAS: Yes. Yeah, that's right. So you set it up so you tile.

The reason I picked this L-shape this way is because this was the statue. So I wanted to remove a tile from each of the other 3/4. I'm sorry, remove a square. A tile is L-shaped.

And so this was a missing square, which was part of the specification of the problem. It has nothing to do with the tile. It's just a missing square because there's a statue on it. But what I could do is if I oriented the tile in this way, now I have this being a smaller problem, which is one-quarter of the size of the original problem with one square missing. This one is one square missing.

This one is one square missing. This one is one square missing. Now that makes all of the problems look the same. Now, at first, they're not exactly the same in the sense that the missing square is in a different position.

But there's some generality that we need in our algorithm that says that eventually, if I keep breaking this up in this fashion, I just need to orient my L-shaped tile in the appropriate way. So there's only four different ways that an L-shaped tile is going to be oriented. It could be oriented like this.

And I won't draw all of them. It could be oriented like that. And it could also be that's the same. So what else is there?

Oh, it could be flipped this way. Yeah, like that and the fourth one. So that's what I did here. This one is what I put up there.

And then as I break this up, let's just say that now I'm going to make four recursive calls. I'm going to place one tile-- one L-shaped tile-- and I'm going to mark off of these things. And I'm going to make four recursive calls to basically myself, except that the arguments are different. It's the same code, obviously.

So when I go up here, I go like that, and hopefully, this will become clearer if you're going to do a tiling here. I'm going to go focus on the top left, and I'm going to do this. What am I going to do with respect to this at this particular quarter of the courtyard?

How am I going to orient the L-shaped tile for this one? Point with your fingers. Exactly right. So this is the same as this. I'm going to go ahead and do that.

So now I have this problem. So I'm going to mark that with a different color. So I'm focused on this problem, and I've done that break. And now I'm good because I have four two by two courtyards, and each of these four has one square missing in each of these four little two by two courtyards. So clearly, I can solve that.

And then once I solve that, I move on to this one, and I move onto this one-- whatever order you want. That's it. And so the code for this is-- from a control flow standpoint, from a standpoint of the overall algorithm, it's extremely straightforward. It's almost exactly like merge sort, except that you have four-way recursion because it's two dimensional, and you end up getting quarters.

And the only thing that's annoying about this code to write and also to understand, which you guys will-- I'll put this up, and I'll show you what to look for-- is how you represent this as such that you get all the values right. So you get an arbitrary square, to begin with, that is missing. And then you have to go to the middle. That's clear.

You're going to go to the middle of this courtyard, and you're going to split it. And you're going to decide which way this L-shaped tile is oriented, and then you got to do the splitting again and so it's just bookkeeping. That's really what this is all about. It's not algorithmically interesting. It's just bookkeeping in terms of the orientation of these L-shaped tiles.

So if you look at the code, and if you ignore a bunch of these indices of these different variables-- let me show you. Well, first, I'll show you what happens when you run the code. I'm running it for two different kinds of courtyards. And I've used A through, I guess, something like S, in this case, to signify a tile.

So you see AAA here. AAA is an L-shaped tile. BBB is an L-shaped tile. FFF is an L-shaped tile, et cetera.

The original specification of the problem-- this is just simply an eight by eight problem-- had this tile that you see here. That's a blank that I've covered up that was missing. I'm sorry, this square missing.

And then I broke this up into two or four 4 by 4's. And then I went all the way up and then the first tile that I placed, I pretended to place this U tile over here. I could've placed it and then called it the A tile.

But this is essentially what you think of the one in the center exactly the way this is oriented.

That's your U tile over here. And then you go all the way down. And explicitly, it's just the A tile that displays first, and then the B tile, and so on and so forth.

And if you take a look at the code, what's the best place to start? So this is the base case-- same thing as we've talked about before. You always need to have a base case. The base cases is a two by two. And this code places one L-shaped tile on a two by two yard.

This is going to make four recursive calls, as you can see from range four, and there's two different kinds of recursive calls in the sense that they're both calling recursive tile, but the arguments of the recursive style are different for this recursive call because it already had the missing square in it. And these three recursive calls that are going to have the missing square in one of their corners. So that's why you see the if and the else here.

And if the original for the recursive call that already had the missing square because it had the statue on it, that's the if statement. And the other ones are the ones that are in the corners. So the first things it would be a corner for this one. This is the corner of that, one and that's the corner of that one. So that's pretty much all I had.

This code-- you need to gaze at it for a few minutes to figure out that I got all the arguments correct. But from a standpoint of, as I said, the control flow and the algorithm, it is hopefully completely clear to you at this point. If not, ask me questions. Come to office hours. Have a good long weekend.