SRINI DEVADAS: Good morning, everyone. Thanks for making it here through the snow and sleet. You will be, quote unquote, rewarded with a cool little puzzle that has both recreational value as well as a deep connection to the most popular sorting algorithm, or at least the most commonly used sorting algorithm, called quicksort. And so what we have up here is what I call the disorganized handyman puzzle.

This is not a puzzle of my invention, but I called it this because I think at some point when I read this, it was a carpenter who had a bunch of nuts and bolts in his bag, and they were all mixed up. So rather than having these nuts attached to the bolts, he was disorganized and the nuts and bolts were separate. And then they all got mixed up together in a bag,

OK? Now, obviously, all puzzles are a little bit contrived. And so we're going to assume here that there's 100 different nuts and 100 associated bolts. And these 200 objects are all mixed up into this one bag that the carpenter is carrying.

And not only that, each of the bolts is unique in terms of its size. And as I mentioned, there is an associated nut associated with each unique bolt. And so, as you can imagine, the goal of this puzzle is going to be finding the most efficient way of creating some organization in this bag by attaching each nut to the corresponding bolt, or vice versa.

And you can assume that there's no ambiguity because of the uniqueness of the nuts and the bolts. And there's 100 pairs waiting to be discovered. And you can try to make that process as efficient as possible. As with all of the puzzles that we look at here, there's going to be naive slash straightforward way of doing this.

You're going to go ahead and figure that out pretty quickly. We're going to analyze the complexity of that, or how long it takes for a specific example of 100 nuts and 100 bolts, and then we're going to scratch our heads and try and do better, all right? And as I mentioned, obviously, there's going to be some programming associated with this. And we're not going to represent nuts and bolts in programs or codes. We're going to switch gears and talk about sorting once we're done with this puzzle. Good.

The only way that you or the carpenter, if you're the carpenter's helper, have of checking to see that a nut attaches to a bolt is to try it out. And the nuts and bolts are different enough in size that even if you had your eyes closed or it was a dark room, if the nut attaches to the bolt,

it would screw on perfectly. If the bolt is smaller than the nut, it would just go through and it would be obvious that the bolt is smaller than the nut. And if the nut is smaller than the bolt, it would also be obvious because the bolt won't go through it, which makes perfect sense from a physical standpoint, right?

And you all tried this before. We've had-- maybe not all of you, but I've certainly had occasion to discover pairs of nuts and bolts, though it was never 100 of them. So that's kind of a little bit contrived, as I said. All right, good.

So the setup is clear? We're good on the setup? Excellent. So what is the straightforward way of doing this? Someone. Yeah, Fadi back there.

AUDIENCE: So like choose a nut by random and try every different bolt in turn.

SRINI DEVADAS: Right, we could choose a nut at random, try every different bolt, or choose a bolt at random and try every different nut. And we're guaranteed, in this case, that there is a pairing. And so after we do that, we can put this paired nut and bolt aside and we've shrunk the problem down to one less than the original problem. If I started out with 100 pairs that are not paired together, but there's 100 nuts and 100 bolts, then I get one pair which is a correct pair, and I have 99 nuts left and 99 bolts left. And I keep going.

Perfectly reasonable way of doing things. And let me show you how that would work. These slides were made by my daughters many years ago you know, back when they listened to me. I have a fat chance of getting them to do any work for me anymore. But they are busier than they were, I guess, years ago. At least they pretend to be.

And there you go. You end up, in this particular example, which obviously has many fewer than 100 nuts and bolts, you end up checking this. And in the worst case, how many comparisons would I have had to do if I had 100 nuts and 100 bolts? In the worst case, I would have to--

AUDIENCE: Do all 100.

SRINI DEVADAS: I have to do all 100, because I might just have gotten unlucky. Like I said, we're not eyeballing this at all. You could probably prune the search a little bit with respect to putting aside-- not having to do an exact comparison. But let's just say that even looking at a nut, as opposed to even touching it with the bolt that you've chosen, is, in fact, a comparison. You're just

eyeballing it as opposed to physically touching the bolt and nut together, and we're going to call that a comparison.

And if you do that, and if you make that assumption, then obviously, you're going to have to look at all 100 nuts if you have a random bolt, in the worst case, OK? You might get lucky. On an average, it might be only 50. We're not going to do that type of analysis here.

So that's good. And what is the complexity in terms of the growth rate of this particular algorithm? If I had N nuts and bolts, then I do N comparisons in the worst case, to find the first pair. And when I say first pair, I mean the correct pairing that's associated with the nut and the bolt. And what happens-- yeah, Kevin, you have a question?

AUDIENCE: Wouldn't it be less than N squared, because once you find each pair, you have one less?

SRINI DEVADAS: That's exactly right. It's absolutely less than N square in terms of numerics. The growth rate is a slightly different question, right? It's related, obviously. And so if you do this and you get the first pair, you now have a problem that is of size N minus 1, right? And so in this case, how many comparisons am I going to make when I have N minus 1 nuts and N minus 1 bolts, in the worst case?

I'm going to do N minus 1 comparisons, right? And then I keep going down. And you could argue that at the very end, when I have two nuts and two bolts, one comparison, if I assume that this was a perfect set of nuts and bolts, that we had all pairs right at the beginning, you could argue that that small problem corresponding to two nuts and bolts can be solved using one comparison. And you immediately know which nut pairs with which bolt, and the other one as well.

But let's call it a confirmation comparison, and essentially say you need two comparisons here and one here. You can obviously shave a number a little bit out of this. But this goes back to-- Kevin is right. It's less than N square if you look at it numerically. But the growth rate is N square, because we know that N plus N minus 1 plus N minus 2, dot dot dot, 2 plus 1 is N times N plus 1 divided by 2.

And sure, you could take this 1 off and this would become N times N minus 1 divided by 2, but that growth rate is N square. Grows as N square. So if you had 100 nuts and hundreds bolts, you're talking about something of the order of 5,000 comparisons if you want to do it numerically. And it grows as N squared. If you had 1,000, then it would be 1,000 times 1,000,

which is 1 million, divided by 2, that's 500,000. So that is astronomical in terms of its growth. You don't want to do that.

Obviously a contrived problem, but N square in general, when you talk about manual labor, is generally not very good. So we'd like to improve this. And we've been talking about recursion. We've been talking about divide and conquer.

This is not divide and conquer in the true sense in that you're going to a smaller problem, admittedly. But recall I said, divide and conquer is usually used when you break the problem up into fractional pieces, which means in mergesort, for example, we took-- or the tiling puzzle, we took courtyard and we broke it up into four courtyards. So essentially, we had four quarters in the case of the tiling puzzle.

We had two halves in the case of mergesort. Here we are solving, basically, one nut and bolt in the original puzzle. And then we're going from N to N minus 1, and so there's many more steps, right? The one thing to remember when you think about complexity is that when you go N to N divided by 2 to N divided by 4, and then you keep going, and you go all the way down to 1, this, of course, when you go like this, there's a linear number of steps.

But when you do that, how many steps do you have to get all the way to 1 in relation-- if you start with N, how many steps do you have? If this were 64-- or let's just take a simpler one. If this were 4, how many steps would I have? I would have two steps. If this were 8, I'd have three steps. And so what's that formula?

AUDIENCE: Logarithm.

SRINI DEVADAS: Logarithm. Log to the base 2, right? So that's the power of fractional sizes. And this is actually a very fundamental notion that is going to appear over and over if you do any algorithms work or take any algorithms classes. That divide and conquer is very efficient, because the number of steps in order to get down to small problems is relatively small. It's a logarithm.

Now, if you broke this up into three parts and you went to N/3, et cetera, then this would be log to the base something else, log to the base 3. So log to the base 2 came because we broke it up and we went down to half the size.

Now remember, of course, I mean, it's not that the complexity here is just logarithmic. It's that you do have two problems. This is a function of the particular specific algorithm that corresponds to divide and conquer. But in the case of mergesort, it's not that we just went

from N to N/2-- well, we went from N to N/2, but there were two N/2 size problems.

But the beauty of divide and conquer is if I magically broke up the N sized problem-- and let's go back to nuts and bolts-- into two problems of size N/2, so each problem has N/2 nuts and N/2 bolts, each problem-- I'll repeat that-- N/2 nuts and N/2 bolts, then if I used-- and this is magical, right? I don't quite know how to do that yet. But if I did that, then think about what happens with respect to these comparisons.

So I said when N was 100, I needed 5,000 comparisons for this naive algorithm. But now if I did this and I got two N/2's, then I have 50, so I call that M equals 50. And then I have another one which is M equals 50. And so roughly how many comparisons would I need if I had, as problem of size 50, 50 nuts and 50 bolts, using our original naive strategy? Roughly?

AUDIENCE: 1200

SRINI DEVADAS: 1,225, right? Roughly. And this would be 1,225. And so, so that is 2,500. So of course, I haven't quite told you how to do this. This is still magic. But I was upfront about it. But clearly, I've gotten an improvement if I have this magic, OK?

And so let's talk about that. Let's turn this nuts and bolts puzzle, or the solution to this puzzle, and try and figure out a divide and conquer strategy which is distinctly different from the brute force strategy that just reduced by one, all right? Now, we'll-- this is just going on. So the comparisons in the worst case is what I just said. And it grows as N square. So big O N square means it just grows as N square. That's asymptotic notation that we won't go into, but you understand what that means now.

So if I do a straightforward divide and conquer, like I did with mergesort, where I just took the array and I split it in half, if I take these nuts and bolts and I separate the nuts out, and I put 50 on this side, 50 on the other side, take the bolts, put 50 on this side and 50 on the other side, is that going to work? No, that's not going to work.

And the reason is if I do this arbitrary partition, then-- and let's say that I take this, and the two of you get together, you're friends and partners, and you say, let's do this in parallel and save some time, you're still going to be counting the number of comparisons. And so, obviously, you also would like to reduce the number of comparisons.

But you can't even use a helper here in the sense that if you split this up-- obviously, this was

not 50 and 50, but three nuts and three bolts on one side and four and four on the other side. As you can see from this example, you had a situation where the matching nut was in one pile on the left for a bolt that was on the right-hand side. And that could happen not just for one, which would kill the process, but could happen for many nut-bolt pairs. So we can't do that. We cannot use the straightforward divide and conquer approach.

So this comes to the first interesting question that we have here, which is, how do we exploit the fact that all of you are going to help me? So I'm the disorganized carpenter or handyman. And I'd like to break this up, first into two piles, such that I can go off and send one or more of you off and say, OK, I can guarantee that this is a subproblem, in the sense that the original problem had all of the bolts that matched all of the nuts that were in my pile.

I want to break it up into two problems in-- this is the magic that we have. We need to figure out this magic-- such that if I give you 50 nuts and 50 bolts, and I keep 50 nuts and 50 bolts, that you can solve that problem. It is a problem. I mean, there's a matching there, right? For those 50 nuts, you have the 50 bolts in your pile. Same thing with me. So how do I do that? Yeah, go ahead, Josh.

AUDIENCE: I have a question. Can you compare the size of nuts to each other?

SRINI DEVADAS: No, you cannot. Yeah, good question. So the only thing you can do is you have a nut and you have a bolt, and if it goes through, then the nut is bigger. If it doesn't, then the bolt is bigger. And then if it fits exactly, you have a match, all right?

Great, so yeah, go ahead, Ganatra.

AUDIENCE: So once you get one to fit, you can use that to sort of like order the other ones. Because it fits in perfectly, right, it fits, so all the ones that-- you just go through all of them, and if it goes through just the hole, then you know those are smaller-- well, they're bigger. And then if it doesn't fit through, then you know that those--

SRINI DEVADAS: Right, great. Excellent. So Ganatra has discovered this notion of pivoting. And pivoting is essentially something that is best described here in the animation that gives you a divide and conquer strategy. Now, I answered Josh's question, and that was a key question. And I'm not going to violate the answer to that question by using some other strategy that does not correspond to this nut-bolt check, right?

So the only thing I can do in this puzzle is a nut-bolt check. But I do get three potential

possibilities whenever I do a nut-bolt check. I do get the information about a perfect match, whether the nut is smaller, or whether the nut is bigger. And that's all I need in order to do pivoting, right?

And so what's going to happen here is I go ahead and I choose an arbitrary bolt. And I don't even actually need to find the match before I start this process. So it's a small variant on what Ganatra said, but it's really a pretty small variant.

And when I see that this is not a match, and in fact, the nut is bigger, then I put the nut on the right-hand side pile. When I see that the nut is smaller, then I put the nut in the left-hand side pile. And I see a match, I just put that aside. I don't put it in either of the piles, because I'm going to actually-- it turns out I'm not going to get a pile of 50 and 50, or in this case, I'm actually going to get something like three and three, because I'm going to get one match out of it.

So in general, I'm not necessarily going to get equal size piles. So that's actually a little bit of an issue, and we'll get back to that maybe later. But I will get a matching, and I'll get a pile on the left that is the perfect problem, perfect subproblem. There I can hand it off to any of you and you can go off and solve it and it will work. And I'll get a pile on the right, same thing for that.

So this was a match, but I don't stop. I don't put the nut in either of the two piles. It's just going to stay up there. And then I keep going. And I make up my right pile and my left pile.

Now, I'm not quite done yet. What do I do now? Yeah, back there.

AUDIENCE: You would test every screw into that nut to see which one--

SRINI DEVADAS: That's right. So remember, I kept the-- I guess you used a different term here. You used-- I said bolt, you said screw. That's fine. But we're talking about a nut here. So let's go with the nut.

And this nut that I put aside, I'm going to now compare that, not with this pivot bolt that I picked, which is now set aside and never have used that again, but I'm going to put that aside, and I'm going to compare each of the bolts with this pivot nut that I have.

So the pivot bolt was picked and I discover the associated pivot nut. And now I'm going to use that pivot nut, which is the nut that I have up here, the light green one, and then I'm going to

do the same thing. And all of this does not violate the answer to Josh's question. And it's going to do exactly the right thing in terms of giving me two piles that are beautiful problems that are going to be solvable.

There's nothing that's stopping me from repeating this process. When we talk about divide and conquer, usually we talk about recursion, and we talk about repeatedly doing divide and conquer. Then I wrote this up here in terms of the N size problem turned into two N/2 problems. Well, each of those N/2 problems I could turn into two N/4 problems, so I could have four N/4's, and so on and so forth.

And so I could clearly do that. And especially if N is large, then I want to get a reduction in comparisons. And get this to grow, by the way. And we won't do this analysis. But rather than growing as N square, you'd like it to grow as N log N.

And that's kind of the asymptotic analysis that you'll have to do if you take a class like 6006. But the general sense that you should take away from this is that you're going to have a logarithmic number of steps. And obviously, that doesn't imply a logarithmic number of comparisons, because the number of subproblems in each of those steps is doubling.

Initially, you had a problem, then you have two subproblems, and then you have four subproblems. Each of them are small in size, but together, assuming these problems are N/2 and N/2, then you can get something that's substantially smaller than N square, namely N log N, OK? And you can do that numerically. And I don't want to get into that too much, but be happy to talk to you about this after lecture or during office hours.

There's a couple of caveats. One of them is there's no guarantee here if I pick a random pivot bolt that I'm actually going to get N/2 and N/2. It's going to be one less than that. Maybe it'd be in over N/2 and N over 2 minus 1.

I could get something-- if I picked a large pivot bolt, I might get a very skewed pair of piles, right? One of them could have 80 in them-- 80 nuts and bolts in them, and the other one could have 20 nuts and bolts in them, OK? So that's something to worry about. We won't worry about that.

But we'll talk about it when we move to sorting, which is what we're going to do in just a couple of minutes. So people buy the solution to the nuts and bolts puzzle. Clearly, it's going to give you some efficiency. If you pick a middling size bolt, maybe you can eyeball that.

It's a little bit of a violation of what we've talked about, but you could clearly-- I mean, let's assume that you can make out the difference between something that's as thick as this and that finger. And then you pick something in the middle. And then you can get two large piles that are roughly equal in size from the original really large pile.

And at least at the beginning, you could probably get, in the context of this puzzle, piles that are roughly similar in size. And after that, it doesn't really matter once you've broken things up. So N is not 100, but N is more like 5 or 10. At that point, it doesn't really matter what strategy you use, because the numbers aren't really different, regardless of the strategy.

So good, all right. So I promised you a relationship to sorting. And obviously, we want to do some programming here. And it turns out that just like we had mergesort, which was a divide and conquer out algorithm, it turns out this pivoting strategy turns into a strategy for divide and conquer that is quite different from mergesort, but is equally applicable to sorting numbers.

And so let me remind you what mergesort was. And then I'm going to contrast mergesort with quicksort, which is a pivot-based sorting algorithm. So forget about nuts and bolts. We're now down to boring numbers, integers. And we want to just sort these numbers in ascending order.

So if I have a bunch of numbers and I want to sort these in ascending order, mergesort would say, I'm going to go ahead and break it up into halves. And let's just do one level of recursion. With all of these things you can always do more, but these problems aren't large enough that you really want to do that in this example.

And the important thing is you just broke it into half. But it was easy. The split was easy. You slice the list, splice the list, what have you.

And you assume that somehow you can sort this in ascending order. And in this case, you need to go to that. And then you sort this in ascending order. So you go like that.

And then you need to do a merge. And we used what we call the two finger algorithm to do the merge that essentially says, I'm going to assume that this array is sorted and this array is sorted, and I'm going to put pointers up at the beginning of these two arrays. And I'm going to do comparisons.

And I'm essentially going to assume that I have a blank array of size six, blank list of size six. And I'm going to be writing into this blank array the result of the comparison that is, which one

is less in the comparisons. So I'm going to get minus 31. And then I compare 0 with minus 4, and I'm going to get minus 4, 0, et cetera, et cetera.

So that was our merge. And the important thing to remember is that our merge algorithm had an easy divide step, and it had a more difficult step that corresponded to taking the subarrays that were sorted, and the work was all in the merge, putting the things together. Because the divide, obviously, is just like chop.

Now, as you can see from the nuts and bolts, division was non-trivial, right? Division required pivoting. But there was no real merge after that. And one side handed off, let's say, to Ganatra these 50 nuts and bolts after I did the work in pivoting. And I kept 50 nuts and bolts as well.

We were done. I mean, maybe I wanted the nuts and bolts back. I don't want him to run away with it. But it wasn't like I had to process anything that he gave me back, right? If he paired them up, then I didn't have to process that, right?

So this is mergesort. The quicksort algorithm, which I'll now describe to you, is divide and conquer, two-way divide and conquer, same as mergesort, but it kind of flips the work. And it does more work up front before the division, and then this little or no work at the end.

So what happens here is-- the way we're going to think about quicksort is we're going to choose a pivot. You can have some array here. And I can go ahead and just call these a, b, c, d, e, f, g. And I'm going to choose a pivot.

And in the code that I'm going to show you, we're just going to go ahead and choose the last element. We assume that this is in random order. We're going to choose the last element of this array as the pivot.

And what did we do with the pivot back in our nuts and bolts example? We-- someone, what did we do with the pivot? Yeah, go ahead.

AUDIENCE: We spliced around it.

SRINI DEVADAS: Yeah, you just basically spliced around it. You compared it with-- we took the pivot bolt, and in this case of the puzzle, you compared it with the nuts. Here we don't have nuts and bolts, we just have numbers. And so you're going to take that pivot and you're to start comparing it with the other numbers.

And you're exactly right in that we're going to get to the point where we have to splice round it, which isn't your standard Python splicing, because that requires contiguous locations. But what you want to do is you want to get to a situation where you have something like this, where you have g somewhere in the middle and all elements less than g are to the left. And let's assume they're all unique elements. All elements greater than g are to the right, OK?

So this is exactly pivoting around g. So it's referred to as pivoting around g. Now, the nice thing here, when you do this, is that you can go off and you can fix the location of g. In fact, g's location, just like the pair, the nut-bolt pair was corresponding to the pivot nut, and the pivot bolt was determined during the pivoting step. And you never had to check whether-- you never had to discover that pair again. You can put that pair aside.

The location of g, the pivot chosen, in the final sorted array, is determined by this pivoting step, OK? That makes sense? And so this location, whatever that index is, it might be right in the middle. It might be a little bit skewed. But that location is determined.

And if you did this with roughly equal piles, if you will, or equal sides, if there were 100 elements here, you might see 50-odd here and 40-odd over there. Now, of course, if g happened to be the largest number, then g would be all the way to the right. So there's that to worry about.

But this is the divide step. So the divide step is the pivoting step. And then you can go off and sort each of those sublists. Because those less than g are not necessarily in ascending order. You just dumped them. There's still work to be done.

And then the greater than g, likewise. They're not necessarily in ascending order. But you know that there's 42 elements corresponding to less than g, and you can go sort that array. And those are going to be the first 42 elements of your final array.

And the g is going to be at the 43rd position. And then the ones that are greater than g are going to be to the right of that. All make sense? Yeah, go ahead, Fadi.

AUDIENCE: So like if we were very unlucky and at each step, we just picked the largest number, the largest number, the largest number, then it's got to be N squared, correct?

SRINI DEVADAS: That's right. So that's pathological. Now, it turns out that, amazingly, in order to avoid that situation when people use quicksort in practice, which is essentially this algorithm that I describe, they take the array that they're given and they actually randomize it. They actually

make it-- they shake it up.

It's like you get these nuts and bolts, and you want to close your eyes and you want to pick up a nut that's middling in size, and maybe all the small ones are up at the top and the big ones are down at the bottom. You want things in the middle, so you go, you shake, shake, shake. You randomize. And then go stick your hand in About. Halfway through the pile and pick up a nut, or pick up a bolt, right?

That's kind of what happens here. So quicksort, it turns out-- and I'll just say this, and if this doesn't make sense, ask me later. But it should give you some intuition. The randomized quicksort, where you have this random input, and you have some probabilistic guarantee that you're picking a pivot that is middling in size in terms of an integer size, is going to be N log N in complexity. But the worst case complexity of quicksort is exactly as you said, it's N square.

But that's really not something that you need to understand deeply to understand, well, either the puzzle or the rest of this lecture. You do get a sense of-- as long as you have a sense of if you pick something in the middle and these piles are roughly equal in size, I'm going to get some improvement. The guarantee that the piles are roughly equal in size, all the way down to the depths of recursion, is a difficult one to achieve in a deterministic way. But it's not hard to achieve in a probablistic way by doing this randomization.

You had another question? Yeah.

AUDIENCE: So then why do we use this more than mergesort, which is like--

SRINI DEVADAS: Ah, that's exactly-- so that is the rest of the lecture, which is only 10 minutes left. But OK, so that's the rest of the lecture. So one of the things that I said here, I said a blank list of size six.

So what mergesort requires, in order for mergesort to be efficient, you needed auxiliary storage that corresponded to the size of the list. Because right at the first level of recursion, when you got two arrays that were N/2 and N/2, and they were both sorted, and you had to merge the two together to create the final result corresponding to the sorted array, you ended up requiring storage that was N in size in order to actually do this two finger algorithm and compare the minus 31's with the zeros and then write them into this array.

Now, the obvious way of taking this and going to here is code that I'm going to show you that is easy to write. And it requires a blank list as well. I mean, the easy way to do that-- and I'll

just show you the code, because it's easier to show you the code as opposed to waving my hands here and potentially confusing people. But let me show you first the quicksort divide and conquer, which is absolutely trivial. I mean, just like mergesort was.

But pivot partition is this step here. This is the pivoting step. And this is what that procedure does. So we'll get to that in just a second. But the rest of it is simply I'm going to go ahead, and once I do the pivot partition, I have something that looks like this.

And then I just go off and run quicksort on this and quicksort on that. And I'm doing that on the array itself. It's just like I have this array, and I'm just saying I'm going to call quicksort with the indices that are associated with the beginning and the end of this array. And then I'm going to call quicksort with the indices that begin with the beginning and the end of this array.

And that same array is going to be my final output, right? So everything is done-- the term that's used here is "in place." So in place sorting and insertion sort, which is also N square, and a few other things are in place sorting algorithms, which essentially say, look, I need to sort all of you in some way, perhaps by age or something. And I don't want another room, right?

I don't want to say, who's the youngest here, and then go over to that room, and then who's the next youngest and go over to the room, et cetera. I just want you to start swapping positions here. And we're all going to be in this room, and somehow we're going to get sorted. So that's in place.

And when I have an array that corresponds to integers, I don't want another blank list or blank array and use that storage. And so that answers your question at the top level as to why quicksort is used. And it's used because the memory requirements of quicksort are substantially less than the memory requirements of mergesort.

And in fact, you can do this pivoting step using one integer as storage. You need the original array, of course, but you needed that to store all of the numbers. And the auxiliary storage, the extra storage that you need, is exactly the storage for one integer where you store the pivot, OK? And I'm going to show you the naive way, which is the regular quicksort.

And that is a trivial pivot partition. It's not the clever one, which essentially says, look, I'm going to go ahead and I'm going to create a new array. In fact, I have two arrays that are less and more, or two lists. Together the sizes of less and more are going to be the size of the original,

maybe minus 1. I'm adding to less and adding to more, depending on whether the element is less than the pivot or greater than the pivot.

And then I'm good. So there's that algorithm, which would just be, oh yeah, I'm going to create a new list here. And I'm going to compare a to g. And then I'm going to compare b to g, et cetera, et cetera. So this does not give you anything. This is an uninteresting algorithm. I'm sorry. Implementation-- this is an uninteresting implementation, OK?

It is much more-- you need to be much more clever-- I don't need this anymore-- to do this in place. So what I want to do is I'm going to take an example here. I want to take this array. And I'm going to not talk about the recursive sorting or anything like that, because that's easy. We kind of know how to do that.

What I want to do is I want to choose this as the pivot. And I want to translate that somehow into the final pivoted output, which is going to be 0, minus 31, 1, 2, 65, 99. Oops, I have something wrong here. So this should be the other way around. 83, right? OK. Mm, bug.

Oh, I'm sorry, I'm sorry. This is fine. This is fine. I just need it-- I don't need it to be sorted. OK, good. Whew. All right. I just thought I found the first bug in my book.

So there's probably many bugs in the book, but I don't want to know. So I don't need this to be sorted. Yeah, so there's a 0, a minus 31. So the key is that I've discovered this. And everything to the left of that is less than 1. And everything to the right of that is greater than 1.

And that's the only requirement that I have. And as I said, I mean, I said this to begin with and then I forgot, I said we're not going to worry about the recursion. So clearly, I have to do more work here with respect to taking this set of numbers and sorting them in ascending order, et cetera.

But what I've done here is I know that 1 is fixed in place, and 1 will never move. And I just have to turn this into minus 31 and 0. And I have to do some reordering here. But 1 is fixed in place.

Now, the algorithm that I'm going to show you, I'm going to show you the code. And it's probably code that you won't be able to parse, at least in minutes or seconds. And this code is in place pivoting. It's clever code.

And what it does is it has one additional variable worth of storage that we're going to call the pivot. So you see the variable pivot there. And just using that one additional integer storage, it

manages to transform this array using a linear number of steps into this array.

So you can see that this is non-trivial. And I need to move from here to there without having this extra storage. If you had extra storage, it would be easy. We know how to do that. But if you didn't, you need this code that has an outer while loop. You see while not done. And then it's got two inner while loops in it.

And it's got a couple of counters, couple of pointers I should say, to indices. And you go left and right, and you magically get this answer. But there's no magic here, because we have the code up and we can run it. And computers aren't smart, so clearly, that code is doing something clever, and we just need to understand that, all right?

So I'm going to-- in the couple of minutes I have left, this is the last thing I want to do is I want to tell you how this code works. And it's really pretty code. And it's clever code.

So what I have is I have two pointers, top and bottom. So I'm going to, essentially, say that top is-- bottom is 0. Initially, it's minus 1, but we go ahead and increment it. And top is 8. And so how many-- I have 9-- 1, 2, 3, 4, 5, 6, 7, 8, 9.

So top is pointing to the pivot. So this is pointing to pivot, which top is 8. And bottom is pointing to the first element. So these two while loops are essentially going from the left of the array and going this way from the right of the array.

So what we're going to do-- and don't worry about exactly how the code does this. I'm going to show you the way this array is transformed and the way these pointers are changed. And then you'll get a sense of how this code works. And obviously, this transformation happens in one of those while loops.

So we have the pivot corresponding to 1. And bottom equals 0 and top equals 8. What I'm going to do is I'm going to start moving leftward from-- this would be the second inner while loop. I'm going to start moving-- oh, I'm sorry.

I'll start with bottom. I'm going to increment-- bottom is initially minus 1. I increment it to 0. And I start moving rightward from the left of the list and try and find an element that is greater than 1. And that is immediate. I realize that 4 is greater than 1, OK?

So when I realize that 4 is greater than 1, I'm going to copy over 4. And this is doesn't mean that I'm copying the entire array. I'm just going to copy over 4. But I'm writing what the array

looks like, because that's important, over to here.

And you might say-- pivot equals 1. You might say, oh, but I overwrote the location 1. And I'm like, don't worry about it. I do have 1 stored in my pivot. So my pivot has 1 in it.

So I haven't lost anything here. It's not like I threw away any locations. I do have that extra location. So I did this rightward move going from the left.

Now I'm going to go leftward from the right. And I'm going to look for something that is less than 1 and I'm going to try and move it. So basically, what this algorithm does is it tries to find things that are-- if a is greater than g, it moves a to the rightmost part of the array. And the same thing, if d is less than g, then it moves it to the left part of the array.

So depending on whether the comparison to g is greater or less, you want to end up in the edges of the array. And you're going to try and get the edges of the array to be correct in the sense that they have elements on the left that are less than the pivot, and the elements on the right that are greater than the pivot. And you're trying to get to the middle. And when your two pointers converge, your top and bottom pointers converge, you're done.

So let's do a couple more steps and close this lecture. So I did the right step. Now I'm going rightward. And now I'm going to go leftward. What is the first element that is less than 1-- no, less, than 1. 0, right? Yes, but I'm going this way.

AUDIENCE: oh!

SRINI DEVADAS: Yeah, I'm going-- I'm glad you pointed that out. I needed to make sure. So I went rightward the first time, and I want to go leftward this time. So this would be-- so I come here and I see 0.

When I see 0, what I'm going to do is I'm going to transform the array. I'll write this out to make sure I get this right. And so right now I have 4, 65 to minus 31, 0, et cetera. And I'm going to go this way, and I'm going to put-- when I see the 0, I'm going to copy over 0 over here and leave the 0 in here for a second.

And there's no problem here, because 4 was copied over here. And so now I overwrote 4 with 0. OK, now you kind of see, maybe get some sense of how this algorithm works. This is the first interesting step where I took 0, and because 0 is less than 1, as I mentioned, I want to jam it all the way to the left. Just put it all the way to the left.

And I'm cool with losing 4, because 4 is being put all the way to the right, OK? Now, at this point, my bottom and top are going to get modified. My bottom is still 0, but the top is 4, because I moved-- top was 8, and I decremented top all the way to the point where it became 4. And I realized that 0 was less than 1.

And then I copied over that value over to the bottom. You can kind of see the code up there. This is actually an output of that code. And I put a 0 in here. So that's what my counters look like.

And it's only a couple of more steps. We're going to go now go right. And I'm going to go do this again, except that 0 is already taken care of, so now I see 65. Clearly I'm going to have a situation where if I go this way, 65 is greater than the pivot, so I'm going to go 0, 65, 2, minus 31.

And because 65 is greater than the pivot, I'm going to write it into what the top was, which obviously got copied over that way. So I'm going to have 65 in here, 99, 83, 782, and 4. So now after this step, bottom equals 1. And the reason for that was 65 was in the location on 1, and top equals 4.

And then the next one, someone want to tell me what the next one is going to be? If I go leftward. I'm going to start from here, top was 4. So I was pointing at 65. And I'm looking for something that when I go this way, I saw 65 greater than 1. Now I'm looking for something that is less than 1.

And minus 31 is less than 1, correct? When I go this way, I'm looking for something that's less than. So given that it's minus 31, I'm going to copy over minus 31 to what top was pointing to. So I have minus 31, 2, minus 31, 65, 99, 28, 782, and 4. And at this point, I'm going to have bottom equals 1 and top equals 3.

Keep going one more, and I get to the point where I see 2. 2 is greater than 1. 0 and minus 31 are less than 1. And so I take 2 and I copy it over here. And now at this point, when I do an increment, I realize that bottom is 2 and top is 3.

And in the very next step, bottom will equal top. And I'm sitting up here, and so I copy-- I make two-- I'd write the pivot into the location that corresponds to the bottom after top got decremented. And in that case, both bottom and top are 2. And that index, 0, 1, and 2, the pivot gets written into it.

And so that's exactly what I had, which I wrote right at the beginning. So I would say this is probably the most complicated code from a control flow standpoint that I've ever shown you. This algorithm is an in place algorithm that does not require any extra storage.

And that's exactly why it's so popular. People are sorting billions of elements in lists, and you can't use gigabytes of storage during the sorting process. And what this clever strategy tells you is you don't need all of that storage. You can do things in place. And as long as you choose the pivot reasonably well, you get your average case N log N complexity, which is the same as the worst case complexity of mergesort. So you've got to be a little bit careful with that.