SRINIVAS DEVADAS: All right. Good morning, everyone. Let's get started. A new module today-- we're going to spend a few lectures on randomized algorithms.

And so not only will we look at slightly different ways of solving old problems like sorting, we'll also look at how we can analyze this new kind of algorithm that generates random numbers in order to actually make decisions as it's executing and that we'll end up obviously with the analysis that gives us the expected run time of the algorithm-- for example, whether the algorithm is going to produce a correct result or not, with what probability will this algorithm produce a correct result.

So I'll talk a little bit about why we're interested in randomized algorithms in a couple of minutes, but let me define what a randomized algorithm, or a probabilistic algorithm, is to start things off.

And so randomized algorithm is something that generates a random number.

Now, this would be a coinage flip, but more often than not, you're generating a real number that comes from a sudden range. Sometimes you're generating a vector. You'll see a couple of different examples here in today's lecture and in section. And it's going to make decisions based on this value, based on r's actual value.

Now, you can imagine that an algorithm would be recursive, and at every level of recursion, it's going to generate a random r. So when you're executing at a particular level of recursion, you may be doing different things based on r. And not only that, if you re-run the algorithm again on the same input, the execution will be different because you're resuming a true random number generator as opposed to a pseudo random one. And the r's that you're going to get at different levels of recursion or through the execution of the algorithm are going to be different from the first time to the second time.

So on the same input on different executions, two things might happen. The algorithm may run for a different number of steps.

So you might get lucky on the first execution, and the algorithm finishes, let's say at 100 time units. The second time around, it takes a long time. It takes 700 time units.

Our goal here is to try and analyze what this probabilistic runtime would be to ask for an expectation, to be able to compute an expectation for the runtime, or-- if you're talking about a different scenario where different executions-- I could actually produce different outputs.

And in this case, it's possible that one or more of these outputs are incorrect. You actually get the wrong answer. And obviously, that's going to happen with a certain probability. You're going to have to decide or analyze what that probability is.

And generally speaking, we won't be happy with a high probability of error, as you can imagine. And we'd like to set up an algorithm such that you can reduce that probability of incorrect output to be something really, really small. And it might take you longer to get to that low level of incorrect output in one case for a certain set of inputs versus another case.

So that's this set up here in terms of randomized. You're going to have algorithms that-- you can think of them as probably correct. So these are algorithms-- you want to think of them as probably correct, and they do have a name. They're called Monte Carlo algorithms.

And then you have algorithms that are probably fast.

So-- indicates a probably correct-- you could have a constant probability that they're going to give you the correct answer, 99%. And you could obviously try and parametrize that. In the case of probably fast, you say things like, it runs an expected polynomial time. And really what that means is that you may have to run it for more information. So rather than taking 100 iterations or 100 steps to sort something, it might take you 110.

But in the case of probably fast, you do get the sorted result at the end. And when the algorithm has finished execution, you do get that sorted result at the end. So it's correct and probably fast or probably correct and deterministically fast. OK. And this is Las Vegas. So you have Monte Carlo versus Las Vegas here.

So yesterday, it occurred to me-- and I've taught this class a bunch of times-- but it occurred to me for the first time last night that there should be algorithms that are probably correct and probably fast, which means that they're incorrect and slow some of the time. Right? So what do you think those algorithms are called? Sorry. What?

AUDIENCE: T?

SRINIVAS DEVADAS: The T? Oh. Oh! That deserves a Frisbee. Oh my goodness! [LAUGHS] All right. There you go. There you go.

All right.

Now, they're not called the T. So we should write that down so everyone knows. Probably correct and probably fast, which is I guess they don't get you anywhere. I don't know what that means-- incorrect and so in the case of the T.

So the MB/TA.

Any guesses? I mean, think about what we have for Monte Carlo, Las Vegas. Extrapolate. These are the kinds of questions you're going to get on your quiz.

I guess you guys don't gamble you. Go ahead.

AUDIENCE: Atlantic City.

SRINIVAS DEVADAS: Atlantic City. That deserves a Frisbee. Yeah. Absolutely right. That It turns out Atlantic City isn't a name that's really caught on, but it was in terms of being used in this context.

Most of the time, if you do have a probably correct probably fast algorithm, you can convert it into a Monte Carlo algorithm or a Las Vegas algorithm. There are some prime testing algorithms to test whether a particular number is a prime or not that run in probabilistic polynomial time, and they may incorrectly tell you that the number is a prime. So that's an example of an Atlantic City algorithm.

We won't actually do Atlantic City. What we'll do is we'll take a look at a couple of different algorithms, and both of these will motivate why randomized algorithms are interesting.

The Monte Carlo example is checking matrix multiply. So you've gotten a couple of square matrices. Both of them are n by n matrices, and you multiply them out-- A times B, and you

produce C. And so you got the C matrix. And rather than re-multiplying and checking the result, you'd like to do something better. You'd like to verify with some probability that you can parametrize that the output matrix is in fact the product of the two input matrices.

And so that's a randomized algorithm that's a Monte Carlo because you're not guaranteeing that that output matrix is in fact the product of the first two matrices or the operand matrices, but you're getting a good sense of how likely that is. And you can kind of squish that probability of error down to however low you want it to be except you have to run the algorithm for longer. So that's an example of Monte Carlo.

Now, quicksort. It doesn't make sense to say-- I guess you could-- but it doesn't make too much sense to say that you have an almost sorted array. What does that mean exactly? You have to categorize that. So quicksort is an example where you're guaranteed to get a sorted array at the end of it. So it's correct. You will get a sorted ray. That's what you wanted-- descending order, ascending order. But it might not run in order n log n time. That's expected time. Order n log n is expected time. And so that's what probably fast would correspond to.

All right? So that's the set up. You can kind of see why these are interesting because you could imagine that in practical scenarios, you might want to do some checking in a probabilistic way. And you want to do that without having to redo all the work.

Obviously you don't want your checker for matrix multiply to be as slow as multiplying two matrices. Otherwise it makes no sense. So let's dive into matrix product and our first example of a probably correct algorithm, or Monte Carlo algorithm.

So what I want to do here is C equals A times B. And the simple algorithm-- I guess, those of us who went to high school, myself included, did my four years-- know of an n cube algorithm-- or learned it back then.

It simply corresponds to taking rows and columns, and you get an entry. You have n square entries that you need to compute corresponding to the output matrix C. And you're going to do order n multiplications and additions, but we're really going to consider multiplications here. When I talk about n here, it's not the total number of operations. It's the number of multiplications. And the reason for that is-- this may have gone away a little bit, but it's still probably true-- that multiplication, in computers, it takes longer to multiply two numbers, integers are floating point numbers, than adding numbers.

It used to be much more dramatic, the differences between multiplying and add in computers. But thanks to pipelining and lots of optimizations, multiplies are actually very fast. But they are, obviously, a more sophisticated operation than addition. So we'll be counting multiplies.

So when you've seen Karatsuba divide and conquer for multiply, back end in 006. Remember that we were counting multiplications, and we were actually trading off multiplications for additions. We were trying to shrink that number associated with the complexity of the algorithm when counting the number of multiplies. And we actually counted the number of additions that were going up-- at least from a constant factor standpoint, not necessarily from an asymptotic complexity standpoint.

And so that's simple algorithm. You probably heard of Strassen. Some of you might have seen it. Essentially what happens with Strassen is you multiply two two by two matrices using seven multiplications as opposed to eight.

Now, if you do that-- and this is similar to the Karatsuba analysis-- you can do this in n raised to log 2 7 time, which is essentially n raised to 2.81 time.

And so rather than n cubed, you can go down to n raised 2.81.

Now it turns out people have obviously not stopped with this. You can go to n raised to 2.70 by doing something of the order of 143,000 multiplications for 70 by 70 matrices. So you can play around. Just like you had Toom-Cook. I don't know if you remember that or it got covered-- but Karatsuba could get generalized into this thing called Toom-Cook. And the same thing, Strassen-- you could go off and divide and conquer whose base case is not two by two, but 70 by 70, and that improves things.

But it turns out there's other arithmetic series summation ways. And so a famous algorithm that's up until 2010 was the best complexity algorithm known. It's Coppersmith-Winograd, which is 2.376.

And then at some point, we had a faculty candidate here who either shrunk this from 2.376 to 2.373. And it turns out that there were two different researchers who came up with a 2.373, but this particular candidate in the sixth decimal place won. So she had an eight. Person had a nine or something.

But anyway, all of these are impractical. OK. You don't want to use them. The constant factors associated with these things are much larger than what you have here. The constant factors

here, I guess it's one, right? Makes sense that it would be one, forgetting the additions of course.

So if you have large constant factors, then you need a billion by billion matrix in order to win. And if you have billion by billion matrices that you want to multiply, do something else. OK. You don't want to go there. Even in the day of the internet, it's not going to work.

So what we'd like to do now is do something better. So we will try-- given that theoretical computer science class, it makes sense to say that our verification algorithm should be better than n raised to 2.376 or 2.3-whatever. Right? Otherwise, it doesn't feel good.

So what we'd like to do-- and we can do this-- is try and get an order n square algorithm-- that's this. So it's probably correct Monte Carlo algorithm where if you have A times B equals C, then the probability of the output equals yes is 1. So in fact, if you got it right, then the verifier is going to not give you a false negative. It's not going to say-- no, you got it wrong-- when you got it right.

But it could give you up a false positive with some probability where you have the probability of output equals yes, and that's a false positive. But you can bound that to be less than half. OK. So it's going to say, yes.

So obviously, if the verifier kept saying yes all the time, you wouldn't have this. It wouldn't be very interesting. I would be constant time, but it wouldn't be very interesting. What is interesting here is that when they're not equal, you're going to get an incorrect result with some operand on the probability.

So you say, about one half seems kind of high-- 50% flipping a coin.

The good news is that these algorithms, you can run them over and over. You can run this checker over and over. And as long as executions are independent, and you can certainly ensure that they're independent by ensuring that the randomness from one execution to another-- the flipping of the coins-- are independent. OK. And so that's relatively easy to do, in certainly all of the scenarios we'll be looking at. In 046, it's relatively easy to do.

You can now drive this probability down to one quarter with two executions because you'll just check different things. And then one eighth with three and so on and so forth. So that's what's cool about it.

And now, if you can look at the runtime, you say, well, runtime still order n square.

That's the beauty of this because I'm just putting an extra constant factor here where I have k n square, where k is a constant.

And effectively, I have this nice relationship in terms of the probability of error going down to 1 divided by 2 raised to k. And what I have here is a k n square.

So that's what's cool about it. And obviously, k n square is 2 order n square in a polynomial time, and but the probably correct aspect of it gets better and better. OK.

Any questions so far?

All right. Good.

So what we're going to do is this algorithm actually works for arbitrary matrices-- the structure at least. We're going to assume that the matrix entries are Boolean. They're going to work in the finite field mod 2. And it's just is an easier proof. It's easier to see. So the complexities are all the same. You're still multiplying numbers. They happen to be small. And multiplication cost you one operation. And you need to do n cubed multiplies to actually get the C matrix, and you have to verify it order n square time.

And so the number of multiplies, again, that you want to use in your verification algorithm has to be order n square. We're ignoring the additions.

So that's what we'd like out of our matrix product checker, and the algorithm we're going to look at is called Freivald's algorithm, cute little algorithm, that does the following.

So the algorithm itself is very straightforward, in couple of lines, a minute or so to describe, and the interesting aspect of it is the analysis-- the fact that you can show this. That's the cool part. If you couldn't show that, there's nothing cool about this algorithm.

So we're going to choose a random binary vector. So there you go. Here's your randomness. And this binary vector, every time you run it, as the k increases here, the random binary vector is different from one to another. That's important. You can't run the same thing again and then expect a different result. That's called insanity.

But you are going to assume that given that we are working in the binary space and this is a binary vector, you're going to assume that r i equals 1 is half independently for i equals 1 through n.

And the algorithm essentially is this-- we're going to do a bunch of matrix vector multiplies. An n by n matrix multiplied by an n by n matrix gives you an n by n matrix, and that's your n cubed.

So these are all-- I think I said this, but I should've written this down-- these are all square matrices that are n by n. And that's where you get your n cube.

A matrix vector would be something where you have-- typically we'd have a column vector here. You're going to get something like that, and you have n square multiplications here.

You're going to grab one of these and then multiply it by that and get an entry here. And that obviously is n multiplications, but you only have n elements to produce here in this vector. So you only got n square. That make sense?

And so what we're going to do is, we're going to take this r, and we're going to compute A times B r. And so the brackets are important because it says that you're going to compute what's inside the brackets first. Otherwise, it would be a problem because you'd be multiplying A times B. And obviously, that's order n cube. Right? You don't want that.

So A times Br equals Cr. OK.

So r, remember, is a column vector. And C is an n by n matrix as our A and B. We're going to output yes.

Else-- if these two are not equal, you're going to output no. OK?

And so that's it. That's one run of the algorithm, generator random r and do the multiplication as you see here.

So let's be clear about complexity, and let's make sure we understand the simpler aspects of the algorithm before we get into the analysis associated with bounding the false positive probability. The hard part is going to be bounding the false positive probability.

But the easy part is first, the complexity. So how many matrix vector products am I doing

here? How many matrix vector products am I doing here in this check on one iteration of algorithm? Yeah.

AUDIENCE: Three.

SRINIVAS DEVADAS: Three. All right. All right. You need to stand up. This is fun. This is the hardest throw I've had to make in 6046. I got to put this down. Warm up a little bit. It's kind of cold.

Whoa. Terrible!

All right. The person who gets up and gets that owns it, and we're going to do this again. All right. Let's see how long this takes.

AUDIENCE: Is this part of my trial?

SRINIVAS DEVADAS: Yes. Well, the first one failed. False whatever, right?

[LAUGHTER]

I got a few more.

[LAUGHS]

All right. Let me see. I think I need to go here. This is good. And I need to be-- all right. Number three. Thank you. Thank you.

[CLAPPING]

So it was three. Three. Perfect. Three matrix vector products because I got to do this. That's the matrix vector product. Remember I'm getting a column vector out of this, which is important, and then I'm going to multiply this matrix with a column vector, matrix vector product number two. And then there's a matrix vector product over here. So then at that point-- do you remember I have a vector and a vector. And checking the equivalence of two vectors is simply checking the equivalence of each element in the vector 1 by one.

So first one, same as the first one. Second one, same as the second one. Et cetera.

And so this is order n square, but three is something that is worth thinking about simply because every once in a while, we're interested in constant factors. And the other thing that's interesting about this-- make sure I write this-- let me write this over here-- that you are going to-- if A B equals C, then there's no issue associated with error here. So there's no notion of a false negative because if AB equals C, then you know, thanks to the associativity of matrix multiplication-- be it whether they're n by n matrices or columns-- you have this relationship here.

And I hope you can read it at the back. Essentially what I have here is if AB equals C. So if in fact, the matrix multiply happened correctly, I'm in a situation where it is clear that A Br equals this, thanks to this associativity of matrix multiply. And that of course, is exactly the same as Cr.

So that should convince you, thanks to associativity of matrix multiply that you don't have any false negatives in this algorithm. Make sense?

So we're all good. All we have to do, given what we have with respect to Frievald is to do this part here, which is going to take a little bit of doing. And the challenge always with simple algorithm is you don't quite know why they work. And then of course, you have sophisticated algorithms, and you don't quite know why they work.

So this will take a few minutes. It's not super complicated, and there's a little insight, as always, with these things that are not immediately obvious.

But we'll have to look at the number of r's. So you have an r vector that you've generated randomly, and it may be a bad vector. It may be a vector that doesn't show you that the product matrix has an incorrect entry. Remember there's n square entries in this matrix. Exactly one of them may be wrong, and you need to find it. Right?

So there may be a lot of entries which are all correct, but you've got to find that one entry that's incorrect. And so you could miss it. A given r vector might miss it, and of course, if you keep generating the r's, you'd like to find it and declare that the matrices weren't multiplied correctly and that probability is what we have to compute.

So we want to get this result where we are analyzing the correctness in the case. You've already analyzed the correctness in the case where AB equals C, but now we have to analyze the correctness in the case where AB is not equal to C. Right?

And so the claim is that if AB is not equal to C, then the probability of ABr not equal to Cr is greater than or equal to half.

So this is greater than or equal to. Over there, I'm just talking about the false negative probability where I'm actually getting an incorrect yes when you have the matrices being multiplied wrongly, incorrectly. And so that's why I get-- this is what I want. I want there to be a greater than one half probability for r to have discovered that, for r to have discovered that. OK?

I'll stop for questions in a second, but let me do a little bit more.

I'm going to compute the difference matrix, and I'm not computing this because obviously this would take a while to compute. It's just for the purpose of analysis.

I'm going to look at the difference matrix D equals AB minus C because you want D to be 0. And that we're going to do some analysis that says-- we are going to try and find these non-zero entries in D because, clearly, the non-zero entries in D tell you if there's non-zero entry in D, you got a problem here. The matrices weren't multiplied properly. So that's why we have D here. Don't think of it as we're actually computing that.

So what we'd like is to, as I said, discover these entries where our hypothesis now is that D is not equal to 0 because that's the case we're considering. We know that D is not equal to 0 if the matrices were multiplied incorrectly. And when I say D is not equal to 0, it means that there are n square entries in D, and one of them is not 0. They all have to be identically 0. That's all it means. D not equal to 0 means one entry at least is not 0.

So now what we need to do is we need to show that there are many r-- it's a binary vector of length n, and you can obviously think about two ways to n possibilities with respect to r. And what we really want to show is that there's a large fraction-- more than half of the r's are going to actually discover that the matrices were multiplied incorrectly. OK.

So we want to show that there are many r's such that Dr is not equal to 0.

And because if Dr is not equal to 0, then you're obviously going to discover that ABr is not equal to Cr. So if ABr is not equal to Cr, that's identical to saying the Dr is not equal to 0. That

make sense?

So specifically, if you look at the claim and writing it in terms of Dr, you want to say that the probability of Dr not equal to 0 is greater than or equal to half for a randomly chosen r.

And so that's it. That's the setup that we have to show. We have to do a counting argument corresponding to these r vectors that are being generated randomly. So let's do that.

So the general argument we're going to make here is simply that we're going to-- roughly speaking-- if you're going to look at a bad r-- what's a bad r? And bad r is something that doesn't discover the incorrect multiplication. That's what a bad r is.

So you're D is not equal to 0, but Dr equals 0. OK. That's a bad r, right? It's quite possible that that would be the case. And so you want to try and figure out how many of these bad r are there because those are the ones that are causing the false negatives. Right? So that counting argument is the crux of the proof of the claim.

So let's look at that. And what we're going to do is, we're going to pick a bad r, and we're going to say I there are these good r's that are associated with this bad r. And for every bad r, there's a good r. And a good r is something that actually discovers the incorrect multiply.

And given that for every bad r there's a good r, half of the arts are good r. That's it. So I'll write it down. That's the essence of the argument. And I'm go a little more slowly so hopefully you'll get that.

So let's look at the case where Dr equals 0 case because that's the interesting case. That's the case where the r is bad even though we had an incorrect multiply, and you get this-- I should have said you get a false positive. So I'm sorry. I think just before, I said false negative, but I meant false positive.

So you have a false positive in this case, and D equals AB minus C not equal to 0 implies there exists an i and j such that Dij is not equal to 0. OK? And there's just one entry at least-- if you say the matrix is not equal to 0, there's got to be an entry that's not equal to 0.

So let's take a look at that entry, and let's just draw it out. That's my D matrix. And there's going to be an i and a j. So that's my ith row and my jth column. And there you go. I have an entry here which is Dij, and I'm just picking that. I don't care what i and j are, but there's got to

be an entry that's not equal to 0.

Now I'm going to create a vector, v. So this vector is not r. It's a vector v that is chosen deterministically given the Dij where it's got 0's everywhere except it at vj.

So if this the jth entry column-wise, everywhere else you got 0. And you just got the one associated with the-- going downward-- the jth entry. OK? So it's a one one-hot vector, if you will. It's got one, one.

So now, if you multiply these two things out, you know that you're going to get something, and we're going to can call this Dv. So you take D and you multiply it by v-- matrix multiplied by a vector.

You're guaranteed, given that all of these are 0, when I do my this times that plus this times this plus this times this, all of these are going to produce 0. This times 1 is going to produce something that's non-zero, and then all of the other ones are going to produce 0. So I'm just adding a bunch of 0's to this non-zero multiplied by one. So I'm going to get something that's non-zero. Right? All make sense?

So I'm going to see something here, which is the jth entry that's not equal 0. And so that implies that Dv is not equal to 0. And in particular, what I'm saying is Dv of j-- so if I just look at that entry that is identically Dij, which is not equal to 0. Because I'm multiplying it by 1 and I'm adding a bunch of 0's to it. That's it. OK.

Yeah. A question.

AUDIENCE: Is it Dv of j or of i?

SRINIVAS DEVADAS: So I picked the j here. So I think I'm going j, j, right? That make sense?

If j was 7 and this is 7 down, then it would be the seventh [INAUDIBLE] because this is going to turn into that. OK?

Now, either way, if I picked it in the middle, it doesn't really matter. The point is there's going to be one entry. So hang in there. There's going to be one entry that's nonzero if you didn't quite get that.

So Dij is not 0. And this is one more observation we're going to make in order to do the counting of these bad r's because this is a bad r that we're looking at if you say that Dr equals 0. You've created a v that has nothing to do with r, but we're going to use the v to go from a bad r, which is our example here, to a good one. That's pretty much it. That's the last step here.

So what we're going to do, is we're going to take any r that can be chosen by our algorithm such that Dr equals 0 because that's the case that we're looking at. And we're going to compute r prime, which is r plus v.

And just remember this is mod 2 arithmetic. You're only going to get 0's and 1's. So if you have 1 plus 1, it gives you 0. Obviously 0 plus 0 gives you 0. And the other cases are clear.

And this plus here, remember, is also-- the other thing that's important is this is not only mod 2. These are all vectors. So r is a vector, and you can think of it as a column vector. That's how I drew it. You're adding up a column vector with a v. That's the column vector the way I drew that.

You could do it with rows if you like, but it's just notation. And you're going to compute an r prime here.

What can you say about Dr prime? Someone?

Yeah. Go ahead.

AUDIENCE: It's not 0.

SRINIVAS DEVADAS: It's not 0. And I'll give you a Frisbee, but then you can explain-- can you stand up a little? I don't want to take this lady's head off.

So can you explain why?

AUDIENCE: Because r prime is r plus v and Dr gives you Dr plus DB is not 0.

SRINIVAS DEVADAS: Absolutely right. So essentially what we have is this is simply Dr plus v. 0 plus Dv, not equal to 0. Do we like yellow or do we like white? Yellow's fine.

So that's pretty much it.

So what's cool about this- is this final step, which I think you've gotten, but I'm just going to say it out loud now, which is that r to r prime is 1 to 1 for any given r such that Dr equals 0, given the situation where capital D is not equal to 0, and there's some Dij-- and there could be many Dij's. I just need one. I've constructed, based on that Dij, this v vector which has the jth entry corresponding to the v vector being a 1 with all of the other entries being a 0. But I can now create an r to r that is 1 to 1 in the sense that if r prime equals r plus v and that equals r double prime plus v-- so if I ever have a situation where in order to show there's one-to-one, I want to say that it's not too many-to-one or even two-to-one.

So if I have an r prime that equals r plus v and you tell me that r prime also equals r double prime plus v, I can make the argument that r and r double prime are exactly the same.

So then r equals r double prime.

So what am I saying there? I'm just saying that for any given r that has Dr equals 0, I can twiddle the jth element of that r and go from 0 to 1 or 1 to 0.

If you tell me that there's a Dij somewhere in that matrix that is nonzero and I do that little twiddle-- remembers it's all 0's and 1's Boolean matrices-- so if I do one twiddle, it's one-to-one. If I do two twiddles and I go 1 to 0, I'm back to 1 again. And that's all this says. Because you have mod 2. That's all that says.

So one little tweak-- and I'm going to be able to take a bad r and turn it into a good r because the good r, the r prime in this case, had Dr prime not equal to 0.

And that's it. That's my counting argument, and all that remains is to essentially close this by sayiny-- just to write this out to get to the final claim and get the one half-- the one-to-one essentially gives you the one half. At least half of these things are going to be good r's.

If you had I Dr that's not equal to 0-- and that's the case that you have here-- then we're going to discover an r prime such that the Dr prime is not equal to 0 and r to r prime is a one-to-one mapping.

So the number of r prime for which Dr prime is not equal to 0 is greater than or equal to the number of r for which Dr equals zero.

And so that implies that the probability of Dr not equal to zero-- so if you just choose an r, this is now a randomly chosen r. Not that others weren't, but I'm treating it a little bit differently here.

This was a specific r for which Dr was equal to 0. I made an argument that you can always get this r prime one-to-one such that Dr prime is not equal to 0. And now going back to what I had initially with respect to the claim here where the r here was a randomly chosen r, I'm saying, thanks to this little argument-- this line up top-- I'm going to be able to say this is greater than or equal to one half. OK?

Cool. Any questions? Yeah.

AUDIENCE: I think the Dr squared times column equal column on the board.

SRINIVAS DEVADAS: Yeah.

AUDIENCE: On the last column, it should be i, not j.

SRINIVAS DEVADAS: This should be i?

AUDIENCE: Yes.

SRINIVAS DEVADAS: People agree with that? Majority vote. All right. I'm good. Let's make that an i.

AUDIENCE: The iteration of that as well, Dv sub j.

SRINIVAS DEVADAS: Oh, yeah. Of course. Yeah. Once you do that, you have to have an i there. Good.

So you're looking at a particular entry-- it makes a difference whether you used a column or a row. If I'd done-- now that I remember-- if you turn this into a row matrix and this becomes a row matrix, you'll essentially get the Dvj. So it depends on which way you look at it, but thanks for pointing that out.

The specifics of i and j weren't particularly important to the proof itself. The key thing is you

zoom in on a particular entry that is not equal to 0, and then you tweak that entry corresponding to the r. So once you tweak that-- you make that 0 or 1 or 1 to 0-- you can get this result.

I'm sorry. I'm pointing to the wrong spot.

This result-- and then get your claim. OK?

So summarize-- we have a bound. We run it over and over, and we get it to the point where we can have a 0.0001 probability that, if the matrices were multiplied incorrectly, that you wouldn't discover that because you ran it for enough hours independently chosen that that probability becomes as low as possible. OK?

So that was Monte Carlo. Let's do a Las Vegas algorithm. And you guys are probably thinking, my goodness. Another sorting algorithm after, I don't know, 17 different sorting algorithms.

This all sorting algorithms that you've ever learned so far. Right? So merge sort doesn't work, and the reason it doesn't work in practice-- if you're really into performance-- is because of the auxiliary space that merge sort requires.

So if you recall there's the notion of in-place sorting. So let's move onto the next thing here, which is quicksort, which is a new sorting algorithm. And I want to motivate it for just a couple of minutes.

And the primary motivation really is practical performance, not asymptotic complexity. So I'll be upfront about that. It's all about practical performance corresponding to quicksort. And quicksort is a divide and conquer randomized algorithm invented in '62.

Unlike merge sort, it's got two interesting properties. The first is that it's in place, like I just said. So no auxiliary space. In Mozart, you can try and get around this. I should say order n auxiliary space. You need a little temporary variable in order to do a swapping.

But you don't have the order n auxiliary space. So you don't have to constantly allocate. And remember, n could be large. It could be in the billions or trillions. So, from that standpoint, quicksort ends up winning simply because of relatively mundane things like memory allocation in your computer.

And the other interesting thing about quicksort in relation to merge sort is that all the work is in

the divide step.

So in merge sort, remember we just split, and we recurse. And what happens when you come back is you have to do the finger emerging algorithm by looking at the two sorted arrays and looking at what the new merger is going to look like. So the work is in the merge. But in quicksort, the work is going to be in the divide because we're going to have to do a bunch of work associated with figuring out how to keep the partitions balanced-- a little bit like we had to do when we did median finding back a couple of weeks ago.

I'm going to talk about three different variants of quicksort. The. variant that we're going to spend the most time on is the Las Vegas quicksort where we'd like to show that it's probably fast and make a statement about the expected runtime. But we'll get to that by talking about a couple of other interesting variants, and this'll be elaborated on to some extent in section tomorrow.

So before we get to variants of course, let's try and set up the structure corresponding to quicksort. And as always, we have an n element array A. You have divide that corresponds to picking a pivot element, x in A. And then we're going to partition the array into sub-arrays.

And what we have here-- this little picture should make things clearer. And you kind of saw this in the median finding, but here we go again. Let's assume all the array elements are unique. We have L, E, and G. L is less than. G is greater than.

And so your pivot element is going to break this array up into L and G, where you got all the elements that are less on the left and all the elements that are greater on the right. And you're going to recurs on the L and G. So recursively sort sub-arrays L and G. Combine is trivial-- or merge is trivial-- because you've already broken things up thanks to the pivoting. And you just concatenate those arrays.

And that's why you can do this in place. There's no issues. You're really recursively sorting sub-arrays. You are moving things around a little bit when you do the partition. Obviously, the initial array may have all the elements. You may pick the pivot such that the pivot is all the way on the right-hand side in the sense that it's a very large element. That is not necessarily a good thing. I will talk about that. But if you pick an interesting pivot or a good pivot, you're going to have to move the elements in the array to the left of the pivot if they're less than the pivot, and you got to move the elements to the right if they're to the right of the pivot.

Nontrivial piece of code, not super complicated, but you can look at the CLRS page 171 to look at in-place partitioning where you don't have to use another order n space to move these elements around such that they look like that picture that I have up there, starting from some random starting point.

So you want to have the picture that you have here, and you need to go from-- the very same array, it needs to-- and x is somewhere here, and you got x plus 1 here and x minus 1 here, for example. And you need to move those things around so they look like L, E, and G, and that's something that you can do in place. And you can look at the code for that in the CLRS. I won't cover that here.

So let's look at a bunch of different variants responding to quicksort. And there's some real simple ones. Each of these, we can knock off with respect to complexity and runtime fairly easily with the one exception that we'll spend some time on, which is the Las Vegas quicksort.

But we'll call these different names. Let's talk about the basic quicksort, which is also a useful algorithm that people use in practice. And amazingly, this algorithm is simply something that says, I'm just going to constantly pivot on either the first entry or the last entry. So I'm going to pick my pivot to be A1. And when I pick my pivot to be A1, it's a value that I'm talking about here. X is a value. It's not an index. The A1 value-- maybe that's 75. Then I'm going to create my L matrix corresponding to this pivot where all the entries are strictly less than 75, and G would be strictly greater than 75.

And I could do that for A1. I could do that for An. So remember that the pivot is a value.

Now, if I look at this, I'm going to do the partition, given x, just like you saw there. And this is going to be done in order n time. It makes sense that you're going to look at every element, and you're going to move it to an appropriate location to the left of x, which is the e array, or to the right. And you'll do that. That takes order n time. And, as I mentioned, you can look at this to see how this is done in place.

So let's take a look at the analysis of basic quicksort, and what I'm interested in, of course, is the worst case analysis. And I asked this question, I think, before when we were doing median finding, but what is the worst case complexity of the basic quicksort algorithm that chooses the pivot as A1? What is the complexity?

[INTERPOSING VOICES]

Order n square. It's order n square. And the reason for that is that you may have an array that is sorted or reverse sorted-- depending on whether you're picking A1 or An. You can have a worst case situation where one side, L or G, has n minus 1 elements, and the other has 0 elements.

And so if you look at our recurrence associated with this, you could have Tn, which is T0, plus T n minus 1 plus theta n. And why do I have a theta n here? Well, remember that I still have to do this divide step or this partition step in order to compute this up unbalanced array. So I do have to look at each of these elements and do the comparison. And maybe I don't actually have to move them, but I have to do the comparison with the A1, which is the x pivot.

And in some cases, if I'm doing the wrong thing reverse sorted, I also have to do the move. Either way, I have a theta n complexity associated with the divide step.

And so if you go off and you look at what happens with this, well, you've got Tn equals Tn minus 1 plus theta n, which ends up with theta n square complexity.

So a hand waved a little bit two weeks ago for a similar analysis, but you can kind of look at it a little more precisely here by writing the actual recurrence out. And you see that you get the recurrence Tn equals Tn minus 1 plus theta n, which is an n square result, or the solutions is n square. OK?

So basic quicksort look bad. It's got a worst case complexity of theta n square. It works well on random inputs and practice. And it turns out that it's a fashion of algorithm, partly because it's in place and it's easy to code, that what people do is they take their inputs, and they shuffle them.

You might get a bad input, and it might take you a long time to run. But if you take an input and you shuffle it and you do that in theta n time, you just move things around and randomize the input. Then effectively, you have a random input, and this thing works pretty well in practice.

Now, what is pretty well? Well, we're going to do an analysis that is going to not be exactly the analysis that you'd have to do on basic quicksort on random inputs, but essentially, you can say that basic quicksort on random inputs is going to run in expected theta m log n time. OK?

It's something that you'll see a little bit of how to do that today and in section, perhaps for

median finding in section tomorrow. But that's all I wanted to say about basic quicksort.

It's a practical algorithm. It does require a little bit of shuffling up at the beginning, and then you can simply use the pivot A1. And because you've done the shuffle, generally you get balance partitions. The L and G's look balanced, and you don't end up with theta n square. If you have any sort of balance associated with the two partitions L and G, you're going to get a nice divide and conquer, which is going to give you your theta n log n. OK? So that's basic quicksort.

There's another way to do this, and so this is a question for you guys. Suppose I wanted to use the quicksort strategy and get a worst case theta n log n through an intelligent pivot selection. So I want to do a pivot selection intelligently.

So how would I get under the structure of quicksort that you see up there on the left there? How would I select a pivot such that I get worse case theta n log n complexity?

Go ahead.

AUDIENCE: Linear median finding.

SRINIVAS DEVADAS: Linear medium finding. Perfect. That's exactly right. There's a gentleman at the back who'd raised his hand, and I decided I'd chicken out.

I think one time to the back of the room is enough for a day. I'll have a Frisbee left. Hopefully you can get one.

So the intelligence pivots selection algorithm is the median finding algorithm because that's going to guarantee me that I'm going to get balanced partitions.

If you tell me that A1-- and remember, we're talking medians of values-- so don't get confused with indices. When I say something is a median, I'm talking about the value, that given its value, there are all these other n over 2 values that are less than it, roughly speaking and n over 2 values that are greater than it.

And so A1, I have no idea whether it's large or small. So I couldn't say much about it. But if I want to be worst case and I want to guarantee that I have balanced partitions, I can choose the median. And if I choose the median every time, I'm going to get perfectly balanced

partitions. They're going to half on the left and half on the right.

And we do know a way of getting balanced partitions. We can guarantee that balanced L and G using median selection that runs in theta n time. And we showed that a couple of weeks ago.

Now, that median selection algorithm was nontrivial. OK? It had this weird thing where you broke things up into five sub-arrays of size 5, and you found a median of medians et cetera, et cetera. But we argued that the whole thing ran in theta n time, which is important.

And so now, if you look at what happens with quicksort and if I write the recurrence for quicksort, thanks to selecting a median, I effectively have balanced partitions. So I have 2T n over 2.

This is thanks to the median based pivoting. That's important. Otherwise it won't work.

And then, just to be very clear here, I got two theta n terms. OK?

The first theta n term is the recursive median selection.

And then the second theta n term is of course the divide, or partition.

But it's important to realize that now I have a lot of work in the divide. A lot of work. I have to do an intelligent selection using this recursive median finding algorithm. And I also have to do the moves comparing and then generate and the G arrays. OK?

So those are the two theta n's. They're obviously theta n, but I wanted to make it clear that there's two things going on here. And we all know that that is theta n log n worst case. All right?

So there is a way of using the quicksort structure template and getting a theta n log n worst case algorithm, which doesn't work in practice because it's just too complicated. What's going on here is at every level of recursion, you're calling another recursive algorithm to find the median. So if you go code this up, it loses to merge sort in practice. You can do all of this in place, but because of all these recursive calls, it doesn't work well in practice. But it's good to know.

And so this is a good example I think, which we don't do a lot of in 046, but you get a sense of the difference between asymptotic complexity and performance.

So while the median finding algorithm has better asymptotic complexity worst case, it really loses in practice to the basic quicksort, which essentially is a bit of a hack, where you take an input and you randomize the input and you run it with A1 as the pivot or An at the pivot.

Is there a different way that you can actually get to a Las Vegas algorithm? And it turns out randomized quicksort is something that you can build and use, which is a bit different from basic quicksort and certainly different from median finding. But it kind of has a little bit in common with them, and it's our example of a Vegas algorithm.

So what happens at randomized quicksort? An x is chosen at random from the array, A.

So you're not choosing A1 or An. You might just flip-- well, effectively an n-sided die-- and pick a particular index, and then go grab the pivot corresponding to the value at that index. You're not going to randomize over values. You don't know what these values are, but you can pick a random index and then grab the pivot based on the value at that index.

And so at each recursion, a random choice is made. And the expected time-- so now we're saying something different. We're making a stronger theoretical statement that the expected time, when you do this, for all inputs arrays A is order n log n.

And so now, this is not worst case time. It's expected time. So this is going to be our analysis in the last few minutes here to analyze not randomized quicksort, but a slight variant of randomized quicksort that is going to show you that you can run randomize quicksort and this variant in order n log n time.

So not quite sure what's going to happen in section tomorrow, but the full analysis is in the book. You should read it. As you can see, it's a couple of pages that includes the description of a quicksort that I have already. But what we're going to do here is analyze a variant quicksort, which is a little bit easier to analyze, and it gives you the sense of why in fact the randomized quicksort is going to run in expected time. And this analysis is easy to do it in a few minutes. So we'll do that.

And tomorrow, you'll see either a median finding analysis that's similar to that analysis in CLRS or precisely that analysis, depending on what your TAs want to do.

So this particular variant, we're going to call paranoid quicksort. And so this quicksort is paranoid in the sense that it's going to be afraid of getting unbalanced partitions, and it's going to keep trying to get balanced partitions. So it's going to try to get a balanced partition. It's going to check, and then if it fails, it's going to try again. And so at the end of it, there's obviously an expectation associated with the number of tries that you need in order to get a balanced partition.

But it just sort of flips the problem on its head and says, you know what? I'm just going to guarantee a balanced partition from a probabilistic standpoint and it might take me a little bit longer to get there. But that's what Las Vegas algorithms are all about. They're probably fast. And once I get a balanced partition, I'm in good shape because I can go do my recursion, and I get my divide and conquer working properly.

So what is paranoid quicksort? Absolutely straightforward. You could probably guess given my description. Let's just choose a pivot to be a random element of A. Perform the partition, and then values will repeat.

So we're going to go off, and we say until the resulting partition is such that the cardinality of L less than or equal to 3/4 of cardinality of A. And the cardinality of G is less than or equal to 3/4 the cardinality of A.

So I'm allowing you a certain amount of imbalance, but not a lot. Right? And that's it. That's paranoid quicksort. You obviously are doing that in each level of the recursion. And at each level of the recursion, your L and G are going to be, at most, a factor of three apart.

So you might get 1/4 and 3/4. If you're lucky, you'll get 1/2 and 1/2. But the worst case, given that you're going to be exiting out of this loop, is 1/4 and 3/4. OK?

So, as always, you have a simple algorithm, and it's not completely clear how you're going to get to expected n log n time. But it's not difficult.

Basically, what we had to do is we have to try and figure out what the probability of a good call is, over here, a good pivot choice, and what the probability of a bad pivot choice is. And we have to obviously-- given the potential imbalance, we have to write the recurrence associated with that, but let's take a look at the pivots here.

And what can we say about the size as of L and G if you just did a random pivot? Well, a bad

call is when you get something in L or G that is less than 1/4. And a good call is when you get somewhere between 1/2-- well, roughly, if you look at the choice of the pivot. So what I have up here, is the choice of the pivot.

If my pivot is out here, I have a very small L, and all of the thing on the right is G. If the pivot is here, I have a relatively small L and a large G. The pivot is over here, I'm good. I got 1/4 and 3/4. If the pivot is over here, I got 1/2 and 1/2 and so on and so forth.

And so this part is bad, this part is bad, and the middle part is good. So that's all that this picture shows.

So a call is good with what probability? Given that picture, a call is good with what probability?

It's greater than or equal to 1/2.

And so what you can now write simply is if Tn is the time required to sort the array, essentially you can say Tn is T of n divided by 4 plus T of 3n divided by 4 plus expected number of iterations in terms of getting a good partition times C times n.

And there is a reason why I'm putting C in here as opposed to theta. That will become clear in just a second. Because I can't really apply the master theorem to this given what I have with respect to Tn over 4 and 3n over 4.

So what I have here is, I'm looking at the case where I could get an imbalanced partition, but the imbalance is bounded. So I'd have n over 4 on one side and 3n over 4 on the other side. But I'm not going to have n over 5 and 4n over 5 or what have you.

And so that's the two recursive calls. So that's hopefully easy to see. The part that is new here is simply the complexity of this code that you see here, which is obviously the randomized algorithm. That's exactly where the randomness comes in because you're picking a random pivot, and you're checking it. And so this is going to run a certain number of times. And we can figure out what the expectation is in just a minute.

But I have C times n because this is constant time to choose a random number. We'll assume that performing the partition is C times n or theta n, and that's why I have this up there. So this, we're going to call this Cn.

And so expected number of iterations given what I have-- what can I say about the expected

number of iterations using simple probability rules? What is that?

2, right? 1 over p. All of them are independent. So this is 2.

So what I have here is something that I think you might have seen this before, but it's worth drawing the tree out and seeing it one more time in case it didn't fully registered the first time or you didn't actually see it in 006 or recitation.

But what I now have is T of n. I want to solve T of n equals T of n over 4 plus T of 3 n over 4 plus 2 times Cn.

And, again, like I said, I didn't put theta n in here because, as you'll see, when I draw this tree out-- because it's not a massive theorem invocation-- it's worth looking at it from a constant factor standpoint to really get the sense of how all of this works out.

And so if I draw that tree of execution and I start counting, basically what I have is 2Cn up at the top. I have 1 over 4 times 2Cn over here. I have 3 over 4 times 2Cn over here. And then this 1 over 4 might go 1 over 16 times 2Cn over here. And this might go 3 over 16 times 2Cn over here. And this would go, I guess it would be 3 over 16 times 2Cn. And then 9 over 16 times 2Cn et cetera.

So this is an unbalanced tree because you have an unbalanced partition up on top, and now you want to count up all the work that this tree does. If you collect up all of the operations, then that's going to tell you what T of n is because that's all the work that you have to do in order to finish up the top level of recursion.

And what you can say is, if you look at this side here are all the way to the right-hand side, you're going to have log to the base 4 over 3 times 2Cn levels.

So that's just simply every time you're multiplying by 3 over 4, when you get down to the number 1, and that's log of 4 over 3. And then over here, it's a little bit easier to think about because it's a power of 2. You're going to have log of 4 to the base 4 times 2Cn levels.

And really, it doesn't really matter honestly when we go to asymptotics. But is worth seeing, I think, just to get a sense of why it all works out, regardless of whether it's n over 4 or a different constant here or whether it's balanced or unbalanced. The tree looks a little bit different. It's sort of weird. It's got fewer levels here and more levels there. So it's sort of tilted this way.

But eventually, you get down to theta 1 constants down below. And basically what you can see-- if you just add it up-- is 1 over 4 plus 3 over 4 is 1. 1 over 16. 3 over 16. Obviously, those all end up being 1.

So you have 2Cn work at each level. And if you just go ahead and be pessimistic about it, there's a maximum of log 4 over 3 times 2Cn levels. And that's pretty much it.

Obviously, now you can start ignoring the constants. You just keep the log here. You don't care what the base is. You got an n here. So drop the 2C. Drop the 4 over 3. Drop the 2C. And you get your n log n. OK?

So that's pretty much it. I'll stick around here for questions. But you got an example of a Monte Carlo algorithm. You got an example of a Las Vegas algorithm.

And tomorrow in section, you'll see a slightly more involved analysis for something that looks a lot closer to the randomized quicksort. So see you next time.