ERIK DEMAINE: All right, let's get started. I am Erik Demaine. You can call me Erik. Today we're going to do another divide and conquer algorithm called the fast Fourier transform. It's probably the most taught algorithm at MIT. It's used in all sorts of contexts, especially digital signal processing like MP3 compression, and all sorts of things. But we're going to think about it today in the context of divide and conquer and polynomials. So let me remind you-- I mean, this class is all about polynomial time, but usually with polynomial time we only care about the lead term.

Today-- and today only, pretty much-- we're going to be thinking about all the terms in a polynomial. So I'll talk about polynomials mostly a and b. You have a constant term, and then a linear term, and a quadratic term, and so on up to-- I will say that there are n terms-- which means the last one is a n minus 1. Normally the degree the polynomial here is n minus 1. I wish the degree was defined to be n here, but whatever. That's-- I'm not-- I can't change the definitions in algebra. So this is the traditional algebraic way of thinking about a polynomial. Of course, you can write it with summation notation. akx to the k, k equals 0 to n minus 1. We'll jump back and forth between them.

I'm also going to introduce a vector notation for polynomials because-- so the ai's are real numbers, typically. We might change that at some point, but usually-- a common reason-- maybe you don't care much about polynomials, but you definitely care about vectors. Any kind of one-dimensional data set is a string of real numbers, like if you're sampling audio-- like right now we're recording this microphone-- you're seeing lots of different-- the movement of the membrane in this microphone over time. You're sampling-- whatever-- 40,000 times the second. Each one you're measuring a real number about where that thing is. That is a sequence of real numbers. Now, you can convert it into a polynomial if you want. They're the same thing. x is not necessarily meaningful here. We really care about other coefficients.

Now, given such a polynomial there are three typical things we want to do. So these are the operations on polynomials. I'll say why we want to do them in a second. So the obvious one, if you do care about x, is some kind of evaluation. So maybe I give you a polynomial-- a of x--

and I give you a number-- let's call it x0-- and I want to compute what is a of x0. So if I plug in-- x here is a general variable, but if I give you an actual real number, say, for the x's, what does that add up to?

So, how would you solve evaluation before we go to the other operations? There's an obvious way. We, you know-- you compute all the terms and add them up, but if you do that naively, computing x to the k-- in this form, maybe-- computing x to the k maybe takes k multiplications, and so the total runtime of the quadratic, but we can do better than that.

I know it's 11 o'clock-- too early in the morning think. Yeah.

AUDIENCE: When you've already calculated x to the i, you can calculate x to the i plus 1 by multiplying it.

ERIK DEMAINE: Good. Once you've computed x to the k, you can compute x to the k plus 1 with one multiplication, and so you can compute all x to n's in linear time and then you're basically doing a dot product between x to the k's and the a vector. Cool, my first Frisbee.

So, that's one way to do it. There's a slightly slicker way to write it called Horner's Rule, but it's doing exactly the same thing that you said. It's a-- this is just a nice algebraic way of writing it. a of x is the same as a0 plus x times a1 plus x times a2 plus, and so on. x times a n minus 1, and then lots of-- close parentheses. So this is, of course, equivalent to that expression by the law of distribution, and this is essentially doing the x-products one at a time. So, this is clearly order n additions and multiplication, so we get order n time. In this lecture, time is the number of arithmetic operations. That's our model. Assume it takes constant time to multiply or add two real numbers. OK. Cool. So, evaluation-- that's easy. Linear time, for today, is good. Quadratic is bad. We want to beat quadratic.

OK, second thing you might want to do with polynomials is add them. The third thing is multiply them.

So, we're given two polynomials-- a of x and b of x-- and we want to compute a new polynomial-- c of x-- that is the summation. How do you define the summation? Well, you would like c of x to equal a of x plus b of x for all x. That's the definition. Of course, we can do it algebraically as well because these are numbers in the end-- for any x this evaluates to a number-- so if we add two polynomials of this form-- one with ai's, one with the b i's-- all we're doing is adding corresponding ai's and b i's. So, this is easy. We just need ck to equal ak plus bk for all k. So, again, linear time-- no problem.

Third operation is the exciting one, the hard one to get good, otherwise this lecture would be over in a couple more minutes. So multiplication-- same deal. We're given a of x and b of x, and we want to convert that into some c of x that, for all x, evaluates to the product of those two polynomials.

How do we do this? We can't just multiply corresponding ak's and bk's. In fact, if you take a big thing like this and you multiply it by corresponding big thing-- eh, let's do it. This doesn't look like fun. We get-- let's see. So, the constant term is just the product-- that's easy, the constant terms-- but then, if I take this product or this product, I get linear terms. So it's going to be a1b0 plus a0b1 times x, and then there's a quadratic term which I get from-- switch colors-- this and this and this. So there's three things times x squared, and that's where I get tired.

I'm going to switch to the summation notation. I didn't go to high school, but I assume in high school algebra you learn this. ck is the sum of j equals 0 to k. ajbk minus j. That's the general form because aj came from an x to the j term, bk minus j came from x to the k minus j term. When you multiply those together, you get x to the k, so this is the coefficient of x to the k. Cool? So, that's what we'd like to compute. Given a and b we want to compute this polynomial c. How long does it take? We have to do this for all k. So, to compute the k-th term takes order k time, so the total time is n squared. So, that's not good for this lecture. We want to do better.

In fact, today we will achieve n log n. That's our goal-- polynomial multiplication in n log n. Why do we care about polynomial multiplication? Because it's equivalent to another operation which we use all the time in digital signal processing, image editing, all sorts of different things, which is convolution.

Convolution is usually thought of as an operation on vectors. So, remember this vector notation, where we're just thinking about the coefficients. x's are kind of irrelevant. We're just thinking about a sequence of real numbers. So, maybe that sequence of real numbers for a represents waveform. Maybe this is the audio I'm speaking now.

And then I take some other waveform. Here I have a Gaussian function-- e to the minus x squared-- and I want to take-- for all possible shifts of this Gaussian I want to compute the dot product between the blackboard and the piece of paper. That's some kind of smoothing function, if I wanted to clean up noise or something like that.

You can do the same thing on a two-dimensional image. It's a little harder to think about, but

you can map a two-dimensional image to a one-dimensional vector. And you have a two-dimensional Gaussian-- if you ever do Gaussian blur in Photoshop this is what you're doing-- a convolution-- and it's used to pretend that your lens is out of focus when you do that. It's done in audio processing, and all sorts of things.

So, formally, you're given two vectors and you want to take all possible shifts of one vector and take the dot product with the other one. I have that written down. I just-- dot product, same thing as inner product, which just means multiply corresponding positions and add them up. And, if you ignore this minus sign, that's exactly what this is doing. This is taking aj versus bk, pretend it's plus j. So, that's the bj vector, but with all possible shifts k. We compute this for all k. That's really cool. We're going to compute it in n log n time. All different n shifts of b will take the dot product with a. It's kind of magical because it looks like you're doing n squared work, but we will do it in n log n time.

The only issue is we have to reverse b. Then the minus signs turn into plus signs. And there's some boundary conditions, but it's basically the same. If we can solve polynomial multiplication, we can solve convolution. Cool? So, that's why we care about multiplication.

So, how are we going to solve this? What I'd like to do is talk about alternate representations of polynomials. That's the next thing here. We just did operations. Let's talk about different representations. So, we talked about this one representation-- it's one way to represent polynomial-- but it's not the only one. You probably know others. So, on the one hand we have representation a is a coefficient vector. So we can write down the ai's. That was just one way to represent a polynomial.

Can anyone give me another way to represent a polynomial? I have two ways in mind. Yeah.

AUDIENCE: Generating function?

ERIK DEMAINE: Generating function. Isn't that the same-- oh, I guess in principle you could imagine writing a recurrence on the generating function or something. It's plausible. In general, generating functions are polynomials if you know what that-- they are cool. So it doesn't quite answer the question. Yeah?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Sure.

AUDIENCE: Representation.

ERIK DEMAINE: Sorry?

AUDIENCE: Points?

ERIK DEMAINE: Point representation, yeah. I call them samples. I'm going to put that under C. A bunch of samples, a bunch of points on the polynomial. So like xk, yk for-- how many do we need-- I think n minus 1 should do it. We'll check. Yep. And if we are told that a of xk equals yk, and we're told that all the xk's are distinct, then this uniquely determines the polynomial. You have a degree n minus one polynomial, and you have n samples. There's only one polynomial that passes through all those points. It's a consequence of the fundamental theorem of algebra that gives you uniqueness. Existence, I think, was proved by Legendre in 1700s, 1800s-- a long time ago. This is good. This is what we're going to use.

There's another answer. I should give you a Frisbee first. You look so excited. Just wait till I hit you, then you'll be less excited. OK, so. Samples, coefficients, anything else? Yeah.

AUDIENCE: Roots?

ERIK DEMAINE: Roots, yeah. Roots is the other answer I was looking for, but it's not going to be so good algorithmically, as we'll see. Sorry.

So, I can give you a sequence of roots. This is the fundamental theorem of algebra that every polynomial is uniquely determined by its set of roots. If you allow roots with multiplicity then every polynomial of degree n has exactly n roots. So, this would be some sequence of r1 up to rn minus 1, and the polynomial would be given as-- you actually need a constant multiplier-- but x minus r0, x minus r1. That would be polynomial. The trouble with roots is that if I give you a coefficient vector and I want to compute the roots, not only is it hard to do, it's impossible to do in our model. If you're only allowed to add, subtract, multiply, divide, take square roots, take k-th roots, there is no way to solve a polynomial of degree 5 or larger. There's the quadratic formula, cubic formula, quartic formula. There is no quintic formula. That's an old result. 1800s.

So, going from coefficient vector to roots takes infinite time. It's not so good. And, in particular, if we think about our operations, addition becomes really difficult. Multiplication is easy if I have two polynomials represented as a sequence of roots. I want to multiply them. That's just concatenating the vectors-- taking union the vectors of their root lists. So that's cool. I'm

multiplying the c's, I guess. But addition is really hard because addition is sort of fundamentally about coefficient vectors. And then, once you go there-- you can go from roots to coefficient vectors and add them up, but then there's no relation between the roots of the sum of the polynomials versus the roots of the original. I don't know for sure that that's impossible. It's definitely very, very hard, probably impossible.

Let me draw a little table. Each of these representations has some advantages and a disadvantage in terms of these three operations. So, on the one hand, we have the algorithms we care about-- evaluation, addition, and multiplication-- and on the other axis we have our representations, which are coefficient vectors, roots, and samples. You'll see why I chose this order in a moment. It makes for a nice, pretty matrix. We've talked about almost every cell in this matrix, but let me just summarize.

We started out just thinking about coefficient vector, and evaluation was linear time. Addition was linear time. Multiplication, so far, was quadratic, although our goal is to make n log n. For roots, I just said multiplication is easy. That's linear time, addition is really hard-- like, infinite time-- and evaluation, I guess that's linear time. In fact, the way it's written, you only have a linear number of subtractions and multiplications, so it's really easy to evaluate.

And then sample vectors-- we haven't talked much about that. The idea is, suppose you're given two polynomials with the same xk's. We're going to fix xk's. All we need is that they're distinct. So xk could equal k, for example, just a bunch of integers. And then we are told what polynomial a evaluates to at every xk, and we're told what polynomial b evaluates to at every xk.

So, we're given some yk's and some zk's, and then we want to compute, say, the sum or the product of those two vectors. What do we do? Just add or multiply the corresponding yk and zk's, because if we're told we want c of x to equal a of x times b of x, or c of x to equal a of x plus b of x for all x, Well, now we know what x's we care about. We just do it at the xk's. That's what we're told for a and for b, and so to compute c of xk it's just the sum or the product of yk and zk.

So multiplication is really easy in the sample view, and this is why we are going to use this view. We're also going to use this view because, as we'll see, there's a problem. Addition is easy, multiplication is easy, evaluation is annoying. I can evaluate a of x at xk for any k, but I can't evaluate it at some arbitrary value of x. That's annoying. I'm told at these finite sample

points, but now I have to somehow interpolate. This is called polynomial interpolation, well studied in numerical analysis and so on. You can do it, but it takes quadratic time, in general. The best known algorithms are quadratic. So, this is bad, this is bad, and this is bad, so no representation is perfect. Life sucks.

What we'd like is to get the best-- now this one is really hard to work with because converting inter roots is impossible in an arithmetic model, so we're going to focus on column A and column C. We kind of like to take the min of those two columns. We won't quite get that. What we will get is an algorithm for converting between these two representations in n log n time-- it's not quite linear, but close-- and once we can do that, if we want to multiply two things in the coefficient land we can convert to sample land, do it in linear time, and then convert back. So that's the magical transformation we're going to cover, and it is called the fast Fourier transfer.

Fast Fourier transform is the algorithm. Discrete Fourier transform is that transformation mathematically. Cool. So the whole name of the game is converting from coefficient representation to samples, or vice versa. Turns out they're almost the same, though that won't be obvious for a long time-- till the end of the class. Any questions before we proceed? Yeah.

AUDIENCE: [INAUDIBLE] multiply repetitions, why not we evaluate a and b first then multiply?

ERIK DEMAINE: Ah. So, OK, the question is, if I want to-- we'll get there in a second-- but if I want to multiply, and it's so easy to do in sample land, why don't I just sample a and b and then multiply them? That's right, but [? effect ?] sampling is not so easy. It takes quadratic time. Let's go there now.

Because we have n samples to do, each one will cost linear time. Remember, to evaluate a polynomial takes linear time. If you want to think of it in a matrix-- let's enter the matrix-- then we get a big matrix. So we're given the xi's and we just want to evaluate a given polynomial whose coefficients are given by a0, a1, a2, and minus 1. Our goal is to compute the yi's-- y0, y1, y2, to yn minus 1-- and if you know a matrix-vector product, you take this row with that column. You take the dot product that multiplies corresponding entries. You get y0. That is the definition of the polynomial evaluation. I'm just going to write a bunch of these rows so you get the pattern. Pretty simple. This is called the Vandermonde matrix. I'll call it V. And in general-- I don't have room for it, so let me go--

In general, if we look at V ij, it's just-- sorry. You may notice I'm not using the letter i. We will get to why in a moment. V jk. Row j, column k. That's going to be x sub j to the power k. That's

the Vandermonde matrix. We can compute it in quadratic time. It has quadratic entries. We can use the trick we suggested earlier-- compute each term from the previous one by multiplying by xj-- and then we want to compute this matrix-vector product, and you can clearly do it in quadratic time-- I'm just computing each thing correspondingly-- and that's sort of the best you can do without any further assumptions. So this takes-- if I want to compute this product, that's the coefficients to samples problem. This is the same thing as computing V times the A vector, so this is a matrix-vector multiplication, which takes n squared time. OK?

On the other hand-- so, that's a problem because we're trying to beat quadratic multiplication, so if we spend quadratic time to convert over here it doesn't matter if this is linear time. There are two problems. One is that conversion costs too much. The other is we don't yet know how to convert backwards. But this matrix field gives us also the reverse transformation. If we want to convert samples to coefficients, this is-- the best notation I know is from MATLAB. How many people know MATLAB? A bunch. So for you, it's V backslash A, but usually in linear algebra like 18.06 you see you have some matrix V times some unknown vector-- usually it's called x, here it's called a-- and you know the right-hand side. You want to solve for this. How do you do it?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Sorry?

AUDIENCE: Multiply by the inverse?

ERIK DEMAINE: Multiply by the inverse. Yeah. How do you do it in computer science?

AUDIENCE: Gaussian elimination.

ERIK DEMAINE: Gaussian elimination. Turns out inverse is the right answer here, but Gaussian elimination would be the standard way to solve a linear system like that. The trouble with Gaussian elimination is it takes cubic time in its normal form. In this case it's a little bit special because this matrix is essentially fixed. The xi's don't need to change. It could be xi is just i or something, so we can-- in this case it's a little better to compute the inverse first.

So we could also just do v inverse times a. From a numerical analysis standpoint this is very bad, but don't worry about it for now. We're going to get a better algorithm today. Anyway, it doesn't involve matrices at all, but the nice thing is, if computing the inverse, that takes n

cubed time, but you only have to do it once. So if you have to do this many times you can do this product in n squared time. You just have to maintain that v inverse once and for all. OK, great.

So, we've got quadratic algorithms to go back and forth between these representations. That, at least, tells us it's doable, but we, of course, need better than quadratic to improve on the naive multiplication algorithm, so that's what we're going to do. In general, we can't do any better, but we have one freedom, which is we have said nothing about the x case. We can choose them to be whatever we want them to be. I keep saying xk equals k. That seems fine. It's actually really bad choice for a reason we will get to, but there is a choice where, magically, this transformation becomes easy and you can do it in n log n time.

Before we get there, I want to give you some motivation for how this could possibly work. As you might expect, even just from that n log n running time, we're going to be using divide and conquer, so let's just think about how divide and conquer could work. I'm going to show you an idea and then we'll figure out how that idea could possibly work. It doesn't work at the moment, but we will be able to choose the xk so that it works.

So, let's say the goal-- I mean, what we want to do is compute this v times a. I'm going to convert-- think of that back into polynomial land. So our goal is to compute a of x for all x in some set x. This is taking a bunch of samples. Set x is just a set of the xk's, but I'm going to change that set in a moment using recursion. So the input to this algorithm is a polynomial a of x, and it's a set capital X of positions that I'd like to evaluate that polynomial at. This is clearly more general than the problem we're trying to solve, and I'm going to solve it with divide and conquer. In divide and conquer there are three steps. Divide, conquer, and combine.

Let's start with divide. Here's the big idea. I would say there are two natural ways to divide a vector. One is in the middle, that's what we've always seen with merge sort and convex hull from last time, but there's another way which will work better here, which is the even entries and the odd entries. So I'm going to divide into even and odd coefficients.

Let me write that down. One of them is called a sub even of x-- that's a polynomial. It's going to have half the degree, so it's going to be sum from k equals 0 to-- I wrote n here, but I think I want something like n over 2 minus 1, n minus 1 over 2, one of those things-- of a to k x to the k.

So, really, what I want-- which is easier to write-- is, in the vector notation, I want all the even

entries. I won't try to figure out what the last one is, but it's roughly n over 2-ish. I'll just write that. You can go a little bit extra. It's fine if you define a sub n to be 0, and a sub n plus 1 to be zero, and all those to be 0, those terms will disappear.

So the key thing here is I'm taking the even entries, but I don't have 2k up here. This is the x to the 0 term. This is the x to the 1 term. This is the x to the 2 term. So there's a difference between x to the k's and the sub 2k, but, I mean, just think about it in vector form. Don't worry about the algebra for now. We're going to have to worry about it in a second, but, intuitively, what I want to do is extract from the vector of all the ai's these two vectors-- the odd coefficients in order and the even coefficients in order-- but I'm going to need the algebraic form in a moment for the combined step. It should be 2k plus 1 x to k. That's step one. Easy to do. Linear time, of course.

Let's jump ahead to step three, combine. In order to compute a of x from-- what I'd like to do is recursively compute a even of x and a odd of x for some values x. It's not going to be x and x. It's going to be some other set. Let's think about how I can compute a of x given some solutions to a even of x and a odd of x. So this is step three, combine.

So I would like a of x over here, and I want a even of something and a odd of something and something in here. Anyone see the algebra? Maybe start with this? I hear a mumble. Yeah.

AUDIENCE: x squared.

ERIK DEMAINE: x squared. Exactly. Time for a purple one. I'm getting better.

Why x squared? Because we have this mismatch here. We have a sub 2k. We want x to the 2k. How could we do that? Well, we could put x squared to the 2k, and x squared to the k is the same thing as x to the 2k. And so magically this transforms into the even entries of a of x. That's half of them. We do the same thing for the odd ones and we're almost there. Now we have a sub 2k plus 1 times x to the 2k, no plus 1. So how can I add a plus 1?

AUDIENCE: Multiply by x.

ERIK DEMAINE: Multiply by x here. Take the whole thing, multiply by x, then I get all of the odd terms of a of x. I add these together. I get a of x. I mean, you could prove this more carefully, but that's just algebra to see that this is correct. Once you have this, it tells you what I need to do is compute a even of x squared for all x in x, so this is 4x and x. There's a for loop for you. So that's going

to take linear time. If I already know this value and I already know this value, I do one multiplication, one addition, and boom. I get a of x.

So in the conquer step I want to recursively compute-- I think I'll call it-- a even of y, and a odd of y for y in x squared. x squared is the set of squares of all numbers in x. So I'm changing my set x. I started with a polynomial a and a set x. Recursively, I'm doing a different polynomial of half the degree-- half the number of terms-- but with a different set of the same size. I started with x. x squared has the same size as x, right?

So let's try to figure out how fast or slow this algorithm is. But that isn't divide and conquer. That is going to be our golden ticket. It's pretty simple, but we're going to need another trick. So I'm going to write a recurrence. Now, this recurrence depends on two things. One is how many terms are there in a that we've been calling n? And the other is how many numbers are there in x? How many different places do I have to evaluate my polynomial? So we've got t of n, and I always call it size of x. So divide and conquer goes hand-in-hand with recurrences. Generally you've got the recursive part, so that's just how big are the subproblems. How many are there? There are two subproblems. They have half the size in terms of n, but they have the same size in terms of x. So, all right, 2 times-- because there's two subproblems-- each with size n over 2 and size x, plus what goes here is however much it costs to do the divide step, plus however much it costs to do the combined step-- all the non-recursive parts. So this is just partitioning the vectors-- linear scan, linear time. This is-- we talked about it-- it's a constant number of arithmetic operations for each x. So this cause-order x time, this cause-order n time. So, in general, we get n plus x.

Now, this is, again, not a recurrent solvable by the master method because it has two variables. So usually when you're faced with this sort of thing, you want to do back of the envelope picture. Draw a recursion tree. That's a good way to go. So at the root-- now, I know that initially x equals n. When I start out I have n coefficients, I have n different positions I want to evaluate them at because that's what I want to do to do this conversion from coefficients to samples. So at the root of the recursion tree I'm just going to write n. Order n work to get started and to do the recursions.

There are two recursive calls. One has som-- both have size n over 2 in terms of a, and they have the same x-- which is also known as n-- in those two recursions. So in fact, the linear work here will be n and n, and then we'll get n and n. x never goes down, so x always remains n-- the original value of n. This is a bad recurrence. There are n-- sorry, there are log n levels.

That's the good news. Once we get down to constant size we can kind of stop. When there's only one coefficient I know how to evaluate the polynomial with just a 0. That's easy. So down at the bottom here, at the last level-- this is the height log n-- the last level is just, again, going to be a whole bunch of n's-- all the same n. Here n is the original value of x because we haven't changed that. How many n's are there down here?

AUDIENCE: 2 to the log n.

ERIK DEMAINE: 2 to the log n, also known as n. Good. So we had-- because we had log n levels, we had binary branching, so it's 2 to the number of levels, which is just n. So this is n squared. All this work, still n squared.

Clearly what we need is for x to get smaller, too. If x-- if, in this recursion-- let me, in red, draw the recursion I would like to have. If x became x over 2 here, that's the only change we'd need. Then n and x change in exactly the same way, and so then we can just forget about x-- it's going to be the same as n. Then we get 2 times n over 2 plus order n. Look familiar? It is our bread and butter recurrence-- merge sort recurrence. That's n log n. That's what we need to do. Somehow, when we convert our set x to x squared, I want x to get smaller. Is that at all plausible? Let's think about it.

What's the base case? To keep things simple let's say the base case when x equals 1, I'll just let x be-- let's say I want to compute my a at 1. Keep it simple. What if I want two values in x? I'd like to have the feature that when I square all the values in x-- so I want two values, but when I square them I only have one value. Solve for x. Yeah.

AUDIENCE: Negative 1 and 1?

ERIK DEMAINE: Negative 1 and 1. Whew, tough one. Ehh. Not bad. Good catch. Negative 1 and 1. That could work. What are negative 1 and 1? They're the square roots of 1. Negative 1 squared is 1, 1 squared is 1. There's two square roots for every number. Two square roots for every number. Interesting. So that means, if I just keep taking square roots, when I square them it collapses by a factor of 2. Let me go to another board and define something.

You're all anticipating what's going to happen, but I'm going to say, a collapsing set x-- or, a set is collapsing if either the size of x squared is the size of x divided by 2, and, recursively, x squared is collapsing. So I need this to work all the way down the recursion. Or I need a base case, which is just x equals 1. There's a single item in x. So I happened to start with x equals

the item 1. It didn't have to be 1. It could have been 7. It couldn't be 0 because 0-- you won't get two numbers. There's only one square root of 0. OK, so I lied a little bit. Other than 0, every number has exactly two square roots, so what's the square root of negative 1?

AUDIENCE: i.

ERIK DEMAINE: i. So complex numbers. If I take square roots of these guys I get i and negative i, and again I get minus 1 and 1. That's when x equals 4. Turns out this is only going to work for powers of 2, but, hey, if n isn't a power of 2 just round up to the next power of 2. That only hurts me by a factor of 2.

Complex numbers. Every time I said real number in the past, pretend I said complex number. Everything I said is still true. Actually, the root thing is only true when you allow complex numbers. Some polynomials have complex roots, so pretend I said complex. We're going to need complex numbers. This is why. Because, when we'd start taking square roots, we immediately get complex numbers.

Next would be x equals 8. Square root of i. Square root of i-- let's see if I can do it-- should be root 2 over 2 times 1 plus i. And then this one is just the-- square root of negative i is going to be root 2 over 2 times 1 minus i, and then we have all our old guys. Oh, and then I could write plus or minus in front of each of these. I was like, there weren't enough terms there. Now I've got eight-- sorry, I've got four numbers here. I've got four numbers there.

How in the world could I remember this? Maybe I memorized it. No, I didn't memorize it. It's actually really easy to figure this out if you know geometry. Geometry. Let's do geometry over here. It's convenient. I'm actually a geometer.

You know, complex numbers have two parts, right? The real part and the complex part. I'm going to draw that in what's called the complex plane, where we draw the real part here, and I guess it's usually called the imaginary part on the y-axis. Every point in this plane is a complex number, and vice versa-- the same thing.

So what did we start with? We started with the number 1. Number 1 will be here. It has no imaginary part, so it's on the x-axis-- this is the real line down here-- and it's at position 1, which I'm going to just define to be right there. Then we've got negative 1. That's over here. Then we got i. That's here, 1 times i. Then we've got negative i. That's here. Then we've got root 2 over 2 times i plus 1. That's here. Root 2 over 2 by root 2 over 2. What is a property of

root 2 over 2 comma root 2 over 2? It has distance exactly 1 to the origin.

If I draw this triangle, root 2 over 2, root 2 over 2. I square this. I get a half. I square this I get a half. I add them together, I get what? Take the square root, I still get 1, so this distance is 1. Interesting. And then I got the negative of that, which is over here, and a negative of that. Then this is the-- with a negative-- did I get it wrong? Yep, sorry. It doesn't matter, but I'll think of it as i minus-- killed a chalk-- i minus 1. It's the same because I have the plus and minus, but I like the geometry. So this point is negative root 2 over 2 by root 2 over 2, and then there's the negative over here. What property do these points have? I heard a word.

AUDIENCE: Unit circle?

ERIK DEMAINE: Unit circle. Good. Who said unit circle? Nice. It's a unit circle, right? Clearly that deserves a Frisbee end.

Unit circle. Hm. Circle. It seem good. What's going on here is I took this number. I claimed it was the square root of i because it turns out, if you take points on the unit circle in the complex plane, when you square a number it's like doubling the angle relative to the x-axis.

This is angle 0. This is angle-- what do you call it-- 45 degrees. This is angle 90 degrees. So when I square this number I get 90 degrees. That's why this number is a square root of i. I probably should have labeled some of these. This is i. This is minus i. This is minus 1, and this is 1. In general, we get something called-- so these are called the n-th roots of unity.

Unity is just a fancy word for 1. 1 is here, and first we computed the square roots of 1. They were minus 1 and 1. Then we computed the fourth roots of 1. All of these numbers, if you take the fourth power, you get 1. Then we computed the eighth roots of 1.

All of these numbers, if you take the 8th power, you get 1 again. So, in general, n-th roots of unity. We're going to assume n is a power of 2, but this notion actually makes sense for any n. And they're just uniformly spaced around the circle, and if you know some geometry and how it relates to trig, you know that a general point on the circle is cos theta comma sine theta. x-coordinate is cos theta. y-coordinate is sine theta. This is also a funny notation for complex numbers-- not so funny, this is the geometric interpretation of cos theta plus i sine theta.

And if I want them uniformly spaced around the circle, and I want to include this point-- also known as theta equals 0-- because when theta equals 0, cos theta is 1, sine theta 0. So I want to say 4 theta equals to 0, and then-- here I'm going to get fancy-- tau over n to tau over n up

to n minus 1 over n times tau. What's tau?

AUDIENCE: 2pi.

ERIK DEMAINE: 2pi, thank you. This is modern notation just over the last couple years. I believe in tau so much I got it tattooed on my arm. Tau is the fundamental constant. Screw pi. None of that 3 and change. Six and change is where it's at. So tau. Clearly this is much nicer. Tau over n, not 2pi over n. Tau is a whole circle. This is one n-th of a circle, 2n-ths of a circle, n minus 1 over n-ths of a circle. I didn't do n n-ths of a circle because that's also the same as 0.

Now, why did I introduce this notation? Because there's this other great thing called Euler's formula, which is that this equals e to the i theta. Double check. It's so rare that I get to do real calculus. This is Euler's formula-- e for Euler-- another number-- 2 and change. e to the i-- this is funny because it's complex-- times theta is equal to cos theta plus i sine theta. This is the relation between exponentials and trigonometry. That's a big thing Euler did. Cool. So what? Because this lets us understand how squares work-- not squares the shape, squares the operation.

When I take squares-- so if I take e to the i theta, this is one of my roots of unity. Let me expand out what theta is. So theta is some k times tau over n. Let's do it this way first. So in reality we have k tau over n, but when I square it, that's the same thing as putting the 2 right here. This is the same thing as e to i times 2 theta.

Bingo. I get what I was claiming, that if I start at some angle theta relative to the x-axis, when I square the number I just double the angle. This is why. This is obvious just from regular algebra. And then this thing, Euler's formula, tells me that corresponds to doubling the angle on a circle. So it only works with points on a circle. So when I go here I get e to the i2k times tau over n-- twice as far around circle. All right. Fine.

What happens if I take this number and I square it? This has a really big angle. This is 1/2 plus 1/8, whatever that is-- 5/8 times tau. When I double that angle I go to here. Now, this, you might call it, 10/8, or you might also call it 1/4 because when you go around the circle you stay on the circle.

So there's another thing going on, which is really-- this is e to the i times 2 theta mod tau. Usually we think of mods relative to integers, but what I mean is every time I add a multiple of tau nothing changes. If I go around the circle five times and then do something, it's the same

as just doing the something. So I kind of need that.

And this is true because e to the i tau equals 1. You may know it as e to the i pi equals negative 1, but clearly this is a superior formula-- so superior I got it tattooed on my other arm.

[LAUGHTER]

It's amazing what you can do with a laser printer and a temporary tattoo kit. Sadly, these won't last, but definitely try it at home. Cool.

So e to the i tau equals 1, so going around in circles-- same thing as not. All right. So you can draw on this picture for every number, what is its square? And, in general, if you look at these four guys, their squares are just going to be these two guys. If you look at these four guys, their squares are going to be among these four guys.

So I started with 8 guys. I square them, I get 4. I square them, I get 2. I square them, I get 1. That's how we constructed it, but you can see it works not only for this 8-point set that we constructed sort of by hand, but it works for the n-th roots of unity. As long as n is a power of 2 this set of points will be collapsing.

So if n is the power of 2 for some integer k, then n-th roots of unity are collapsing according to this definition. And that's what we want. Then this divide and conquer algorithm runs in n log n time because every time we square the set x, we reduce its size by a factor of 2. We get t of n equals 2 times t of n over 2 plus n. Plus order n and that's order n log n.

And so this whole thing we compute-- in other words, we set xk to be e to the ik tau over n. Now, I guess I should say how to compute that, but let's just say that's given to you for free. It takes constant time to compute each roots of unity. In fact, again, we only need to do this once and for all for each value of n, so you can think of it as just being part of the algorithm. These are the xk's we use. I said xk could be anything we want as long as they're all different, so I'm going to choose them to be n uniformly spaced points around the complex unit circle. And then magically this algorithm runs in n log n time. It's pretty cool.

Intuitively, you think of real numbers x squared, of course it has the same sides as x. But once you go to complex numbers there's this nifty trick where you start with one n-th of the circle and you'd uniformly space points after the first level of recursion. You only have to be dealing with n over two of those points, namely the even ones, also known as the n over [? 2th ?] roots

of unity. And after the next level of recursion is the n over fourth roots of unity. And so on. So this recursion is well defined. When you have a vector of size n you just have to deal with the n-th roots of unity, and you're happy.

That is fast Fourier transform. That algorithm with these xi's is FFT. So let me write this somewhere. It kind of snuck up on us. This is the algorithm we were aiming to find.

Fast Fourier transform is that divide and conquer algorithm on the right. I'll write it abstractly. For something called the DFT-- the discrete Fourier transform-- is the corresponding mathematical transformation. The fast is about an algorithm. Discrete is about discrete. So DFT is this thing that we wanted to compute, which was basically the product v times a.

Remember, v was the Vandermonde matrix, and it depended on all these xk's. And we're going to set xk to be this e to the ik tau over n. So if you remember the vjk here was xj to the k-th power, so this just becomes e to the ijk tau over n. It's a little funny these are all consecutive because this is a totally different, but-- that's the matrix. If you take that matrix times a vector, that is called a discrete Fourier transform of the vector, and FFT is a way to compute it in n log n time. You just run this algorithm and for those xk's it'll just work in n log n time. Cool.

But if you remember way back to the beginning, what we needed is a way-- this converts a coefficient vector into a sample vector. Then we can multiply the polynomials, then we need to transform the sample vector that results back into a coefficient vector. So we're only half done. I only have 15 minutes. Luckily, the other half is almost identical to this half, so let's do that next. Maybe over here.

So we've got our great divide and conquer algorithm, what we need now-- let me give you the polynomial multiplication algorithm. Let's call this-- sound more exotic-- fast polynomial multiplication, fast meaning n log n. Let's say that we're given two polynomials, a and b, represented in coefficient form. What we need is-- I'll call it a star, which is the result of running FFT on a. It's the discrete Fourier transform of a. b star, do the same thing.

So we know what this means is convert a from a coefficient vector into a sample vector. Convert b from a coefficient vector into a sample vector. Now I have the samples of a star at the n-th roots of unity-- these guys-- and I have the samples of b at the exact same points-- the n-th roots of unity-- so I can compute c star, the transform version-- the DFT of c-- is just the, I mean, c star k equals a star k times b star k. This is the multiplication algorithm for sample vectors. We started with that. That's linear time.

And then the missing piece is I need to re-compute c, which is the inverse fast Fourier transform of c star. So this is the missing link, so to speak. We need to be able to go backwards in this transformation. Good news is the algorithm is not going to change. What we're computing isn't going to change. All that's going to change are the xk's. Why? Because, remember from the top-- right now we know how to compute v times a, but what we now need to compute is the inverse times a. So the only question is, what is the inverse?

This matrix has a super special structure-- it's symmetric, lots of points on a circle-- maybe the inverse has a nice structure. And it does. Claim is the inverse is v complex conjugate divided by n. What's complex conjugate? For a geometer, it's reflection through the x-axis. For an algebraic person, a plus ib, the complex conjugate is a minus ib. So just apply that to every entry in the matrix, and then divide all the entries by n. You get the inverse. Cool.

Very cool because what this tells us is we run exactly the same algorithm and do exactly the same transformation. If we want to do the inverse we can actually just use v, but with a different choice of xk. Namely, for the inverse, we'll call it xk inverse. We just take the complex conjugate of this thing, which turns out to be e to the minus ijk tau over n, and then divide the whole thing by n. I'm using a fact here, which is that the complex conjugate of this number is actually, just, you put a minus sign here. Why does that hold? Because geometry.

Theta is usually measuring the counterclockwise angle from the x-axis. If you take the complex conjugate you go from up here to down here-- the reflection to the x-axis. That's the same thing as if you measure theta as a clockwise angle from the x-axis. That's the same thing as negating the angle. So that's just a little geometry. You can prove it algebraically, although I don't know how off-hand. It's not hard. So if I want to take some angle and then flip it through the x-axis, it's the same as the negative of the angle. So the complex conjugate of this number is the same thing with a minus sign. And then we have to divide by n. If I just-- did I lie?

I can't divide by n, sorry. It's in the wrong spot. I use these xk's and then I apply the transform. I take this thing. I get not quite the inverse, but I end up getting n the inverse. I multiply that by my a star. That's going to give me n times n. So then I just take that vector and divided by n. Boom, I've got the inverse transform. So this is how you do inverse fast Fourier transform. You just flip the sign and the exponent in the xk's, then you do the same matrix-vector product, and then you divide by n. And then you've done the inverse. So that's how you do-- if you believe this claim-- that's how you do this last step. Question?

AUDIENCE: What's the j in the x j [INAUDIBLE].

ERIK DEMAINE: Whoops, no j, sorry. I was imagining these guys. Just k. Thank you. Other questions?

So what remains is to prove this claim. I'd be very happy if I-- I should probably have better-- anyway, that's not the best. I'm going to call this xk prime. Then when I plug that in, I get a different matrix, v prime. And what this claim says is that v prime is equal to n times the inverse. So I apply the same f of t algorithm, but with v prime instead of v, then I get not quite the product I want, but just n times the product I want. Divide every term by n and we get the inverse. This is the cool thing about complex numbers. Here we get another cool thing. All right, but we have to prove this claim. So let's do a little bit of algebra. No pain, no gain. Good.

So let's look at vjk prime. Sorry. So let's look at p, the product of v, and v complex conjugate-- what I was calling v prime up there. I claim that this thing is n times the identity matrix, with 1's down the diagonal and 0's everywhere else. So let's look at this product. In general, let's look at the the jk-th item in the product. That's going to come from row j of V, dot product with column k of the complex conjugate. Now the matrices here are symmetric, so actually rows and columns are the same, but that's the general definition of the cell and the product of two matrices. So let's write it out and a summation. We have-- from m equals 0-- it's so hard not to use i for my summations. It's the only class I have to do it because i is already taken, but I guess I can still use capital I, but we'll use m because i is complex number today.

So we have e to the i tau jm over n times e to the minus i tau mk k over n. I don't know why I changed the order. I put the tau here instead of there, but same thing. This is just the for every position m in the cell and corresponding position m in-- sorry, m in the row, corresponding position m in the column, I have jm and I have mk-- again, the order doesn't matter. It's symmetric, but I'm getting it right here. This is-- we're using this formula. I put a minus sign here because this is the complex conjugate.

So now I just do some algebra. These share a lot. They share i. They share m, and they share the divided by n. So this is sum m equals 0, n minus 1 e to the i-- oh, they also share tau-- tau m over n times j minus k. Please correct me if I make any mistakes. Cool.

So it depends how and k relate, of course. If j equals k, that's also known as something on the diagonal. I want the matrix n, n, n, n, 0, 0. So j equals k. That's the diagonal. That's where I want to get n, and, indeed, if j equals k, this becomes 0. So all this becomes 0. e to the 0 is 1,

and so I'm summing up 1 n times. I get n. Cool. That's one case. This is n if j equals k, and somehow I claim that everywhere else I get 0's. So let's prove that.

So if j does not equal k-- I'm going to rewrite this a little bit. j and k are fixed. m is changing in the sum, so I want to write this as sum m equals 0 to n minus 1, e to the i tau j minus k, over m, times m. In other words, this thing raised to the n-th power. What is this series? One word.

AUDIENCE: Geometric.

ERIK DEMAINE: Geometric. Thank you. This is a geometric series, and I guess I should have waited to give you the Frisbee until you tell me how do you solve a geometric series with a finite term? So think of this as z to the m. Do you know the formula?

AUDIENCE: That's the e to the i tau--

ERIK DEMAINE: Just z.

ERIK DEMAINE: Oh, z to the n minus 1.

ERIK DEMAINE: Z to the n minus 1. Almost.

AUDIENCE: Over z minus 1.

ERIK DEMAINE: Over z minus 1. Yep. That's-- If you have sum of z to the k, k equals 0 to n minus 1. That's that. It's in the appendix of your textbook. So we just plug that in and we get-- oh boy-- e to the-- better chalk-- e to the i tau j minus k over n to the n-th power minus 1 over e to the i. Actually, the denominator doesn't really matter because this is supposed to be 0, so it's all about the numerator, but it's the same thing. Minus 1.

Where's my red? Red.

The n cancels with the n. This is an integer not equal to zero, so we have e to the i tau to an integer power. What's e to the i tau? 1. Convenient that I have that there. 1 minus 1 is 0, so we get 0. Satisfying.

So that proves this claim. We prove that, on the diagonal, we get n because we had n copies of 1. Off the diagonal we get 0, like this. Therefore, v complex conjugate times n is the v inverse. Therefore, this algorithm actually computes the inverse fast Fourier transform.

So that's it for algorithms today, but let me quickly tell you about some applications. You've

probably taken other classes that use applications of fast Fourier transform, so I will just summarize. If you've ever edited audio, you probably did it in-- unless you're just pasting audio clips together-- you probably did it in what's called frequency space.

So you know that-- as I talked about in the beginning-- when we're measuring where the membrane on this microphone goes over time, that is in the time domain for every time we sample where, physically, this thing is. If you apply-- I think the way I defined it here is the inverse fast Fourier transform, usually it's called Fourier transform-- to that time domain vector, you get a new vector. Now it's a complex vector. You may have started with real numbers. You get complex numbers.

So for every position in the vector-- what it corresponds to-- the x-axis is no longer time. Now it's frequency. For every frequency you're measuring, essentially, you're viewing this vector-- the waveform-- as a bunch of trigonometric functions-- say, sine of something times theta. If you look at one of the entries in the vector, and it's a complex number-- if you compute the magnitude of the complex number-- the length of the vector, of the length of that two-coordinate vector-- that is how much stuff-- of that frequency-- you have. And then the angle of the vector, in 2D, is how that trigonometric function shifted in time.

So if you take a pure note, like if I was playing a bell and it's exactly C major, it looks really wavy. It's actually a nice perfect sine curve-- some offset depending on when I hit it-- and then if you apply the Fourier transform what you get is 0's everywhere except for the one frequency that's appearing, and there you get 1, and everywhere else you get 0. Well, 1 possibly rotated, depending on the phase.

And you can take any audio stream, convert it by a Fourier transform, do manipulations there. For example, you've probably heard of high-pass filters that removes all the high frequencies, or low-pass filters remove all of the low frequencies. You just convert to this space and zero out the parts you want, and then you convert back with inverse Fourier transform.

If you've used Adobe Audition or Audacity, they can all do these things. And there are tons of contexts where converting to Fourier space makes life easy. And, in general, if you have any time-based signal you should always think about what do you get with FFT when you transform to frequency-based signal, and you can do lots of cool things you couldn't do otherwise. And it only takes n log n time.

Plus people do it in hardware. There's a fast implementation called FFTW-- the fastest Fourier

transform in the west-- which is made here at MIT a bunch of years ago, and is still the best software implementation of FFT. So people use it everywhere. Your noise-cancelling headsets probably use it, MP3 uses it. It's a cool algorithm.