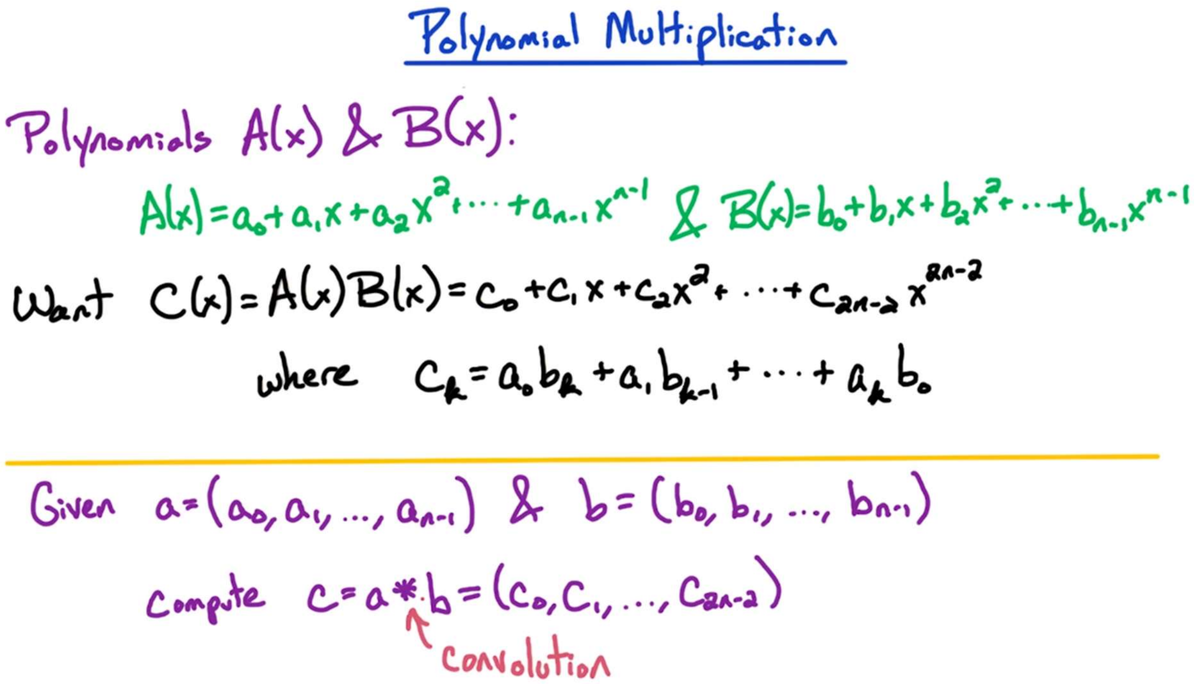
Lesson: DC4: FFT – Part 1

(Slide 1) Polynomial Multiplication

***



Polynomial Multiplication

Polynomials A(x) & B(x):

$A(x)=a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}$ & $B(x)=b_0+b_1x+b_2x^2+\cdots+b_{n-1}x^{n-1}$

Want $C(x)=A(x)B(x)=c_0+c_1x+c_2x^2+\cdots+c_{2n-2}x^{2n-2}$

where $c_k=a_0b_k+a_1b_{k-1}+\cdots+a_kb_0$

Given $a=(a_0,a_1,\dots,a_{n-1})$ & $b=(b_0,b_1,\dots,b_{n-1})$

Compute $c=a*b=(c_0,c_1,\dots,c_{2n-2})$

convolution

What we've seen so far is how to use Divide and Conquer in a clever way to multiply large integers. So for n bit integers, we were able to multiply, compute their product in time better than O(n^2).

A similar idea applies to matrices as well. What we're going to do now is multiply polynomials. And to do this, we're going to use the beautiful Divide and Conquer algorithm known as FFT - FFT stands for Fast Fourier transform.

So here's the set up. We have a pair of polynomials A(x) and B(x). For the polynomial A(x), we will denote its coefficients as a0, a1, a2, …, an-1. So it's of degree at most n-1. And for B(x), we will denote its coefficients as b0, b1,…, bn-1. And it also is of degree of most n-1.

Our goal is to compute the product polynomial, C(x) which is C(x) = A(x) B(x). And the coefficients of C(x), we will denote as c0, c1,…,c2n-2 … since the degree of C(x) is at most 2n-2.

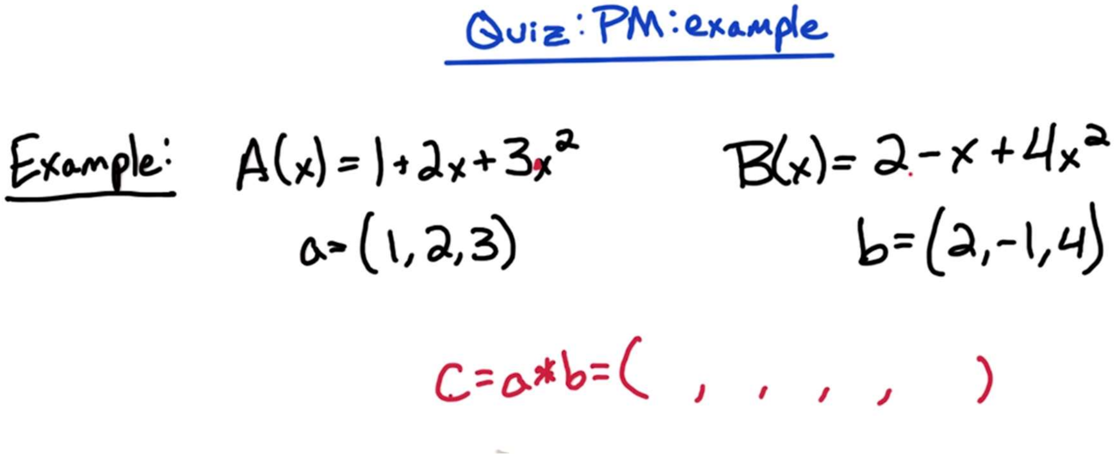
Now recall that the kth coefficient of the polynomial C(x) is denoted by c_k. This is obtained by taking the constant term for A(x), this is a0 times the coefficient for the kth term of B(x), this is b_k. So I look at a0 b_k and I add that to the other possibilities a1 b_k-1, … and so on, up to a_k b0.

We want to solve the following computational problem. We're given the vector of coefficients defining A(x) and we're given the vector of coefficients defining B(x), and we want to compute the vector of coefficients for the product polynomial C(x).

Now this vector c is known as the convolution of a and b. So the star symbol denotes the convolution: c = a * b = (c0,c1,…,c2n-2)

(1st Slide 2) Quiz: PM: Example

***



Quiz: PM: example

Example: $A(x)=1+2x+3x^2$ $B(x)=2-x+4x^2$

$a=(1,2,3)$ $b=(2,-1,4)$

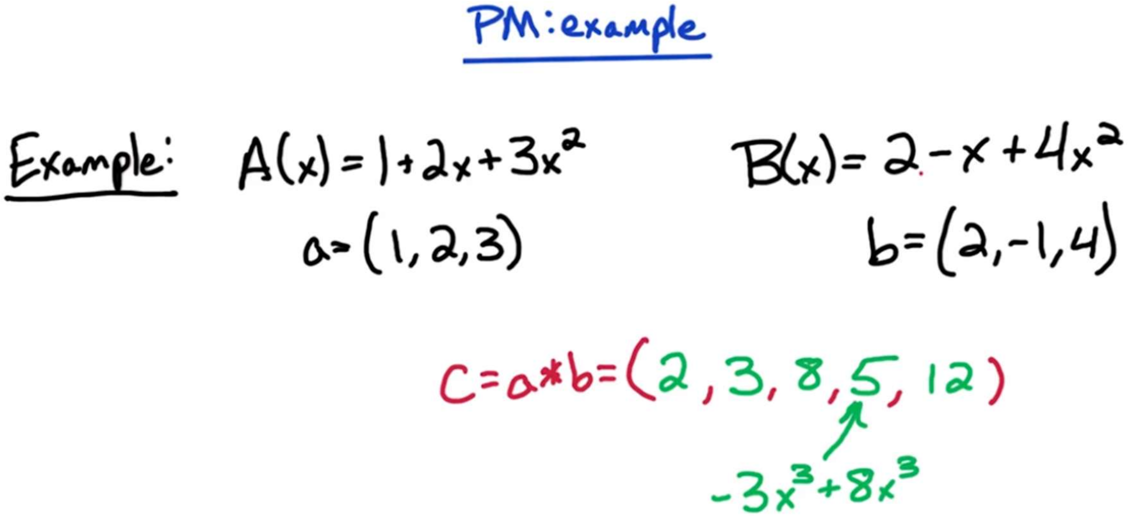
$c=a*b=(\ ,\ ,\ ,\ ,\ )$

Let's take a quick quiz to make sure you're familiar with multiplying polynomials and let's consider the polynomial $A(x) = 1 + 2x + 3x^2$. And $B(x) = 2 - x + 4x^2$. So the vector of coefficients for A(x) is a = (1, 2, 3), and the vector of coefficients for B(x) is b = (2, -1, 4). And now compute the convolution of a and b. c = a * b = (, , , ,)

For vectors a = (1, 2, 3) and b = (2, -1,4), what is their convolution c = a * b?

(2nd Slide 2) Quiz: PM: Example (Answer)

***



PM: example

Example: $A(x)=1+2x+3x^2$ $B(x)=2-x+4x^2$

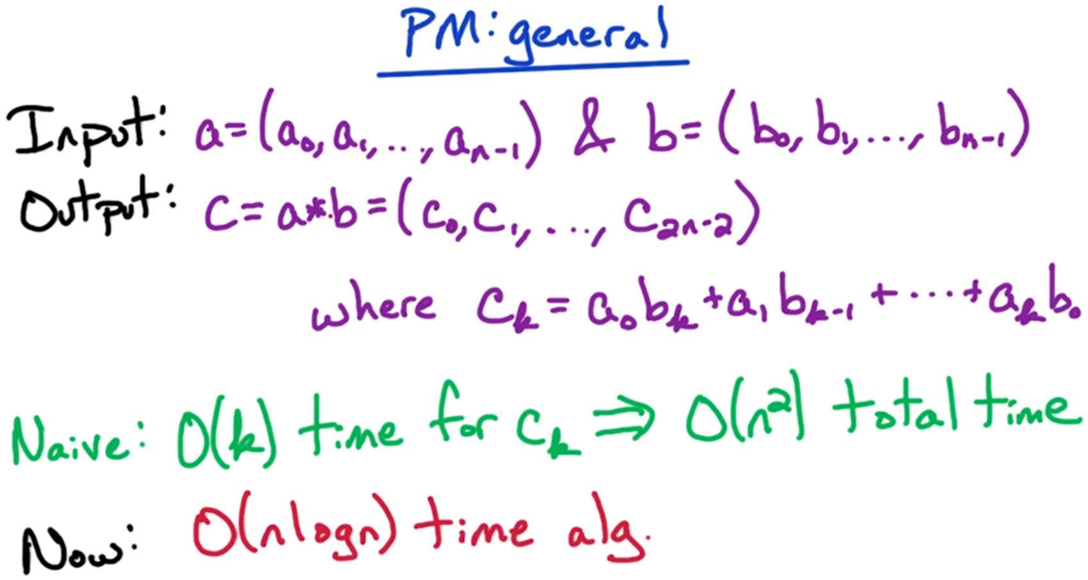
$a=(1,2,3)$ $b=(2,-1,4)$

$c=a*b=(2,3,8,\underset{\uparrow}{5},12)$

$-3x^3+8x^3$

The solution to this problem is (2,3,8,5,12). To get the x^3 coefficient, 5, we obtain it in the following way: multiplying $3x^2$ by $-x$, we get negative $-3x^3$. Multiplying $2x$ by $4x^2$, we get $8x^3$. Adding these up, we get $5x^3$.

(Slide 3) PM: General Problem
***



PM: general

Input: $a=(a_0,a_1,\ldots,a_{n-1})$ & $b=(b_0,b_1,\ldots,b_{n-1})$

Output: $c=a*b=(c_0,c_1,\ldots,c_{2n-2})$

where $c_k=a_0b_k+a_1b_{k-1}+\cdots+a_kb_0$

Naive: $O(k)$ time for $c_k \Rightarrow O(n^2)$ total time

Now: $O(n\log n)$ time alg.

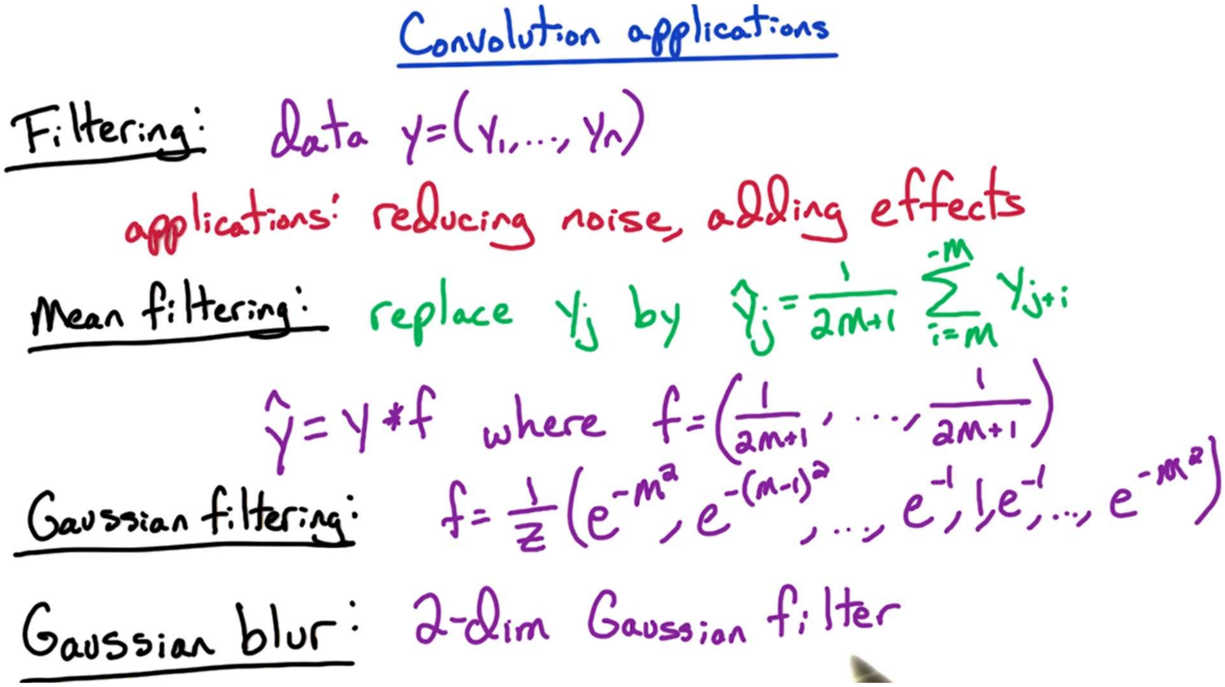
Once again, the general computational problem that we're considering is, given this vector of coefficients for A(x), and this vector of coefficients defining B(x), we want to compute their convolution C.

Now a simple algorithm for computing this convolution is going to compute each of these coefficients, in turn. Now there were O(n) coefficients. How long does it take to compute each coefficient? Well, a naive algorithm takes O(k) time to compute the kth coefficient because there are O(k) terms that we have to sum up. This is going to lead to an O(n^2) algorithm to compute all of the O(n) coefficients of the polynomial C(x).

What we are going to do now in this lecture is an O(n log n) time algorithm for computing this product polynomial C(x), the convolution of A and B.

(Slide 4) Convolution Applications

***



Convolution applications

Filtering: data $y=(y_1,\dots,y_n)$

applications: reducing noise, adding effects

Mean filtering: replace y_j by $\hat{y}_j=\frac{1}{2m+1}\sum_{i=m}^{-m} y_{j+i}$

$\hat{y}=y*f$ where $f=\left(\frac{1}{2m+1},\dots,\frac{1}{2m+1}\right)$

Gaussian filtering: $f=\frac{1}{z}\left(e^{-m^2},e^{-(m-1)^2},\dots,e^{-1},1,e^{-1},\dots,e^{-m^2}\right)$

Gaussian blur: 2-dim Gaussian filter

Before we dive into the algorithm, let us take a look at a few of the many applications of convolutions.

One important application is filtering. So here we have a data set of n points. What we are going to do is we are going to replace each data point by a function of their neighboring points. This is used for such things as reducing noise or adding effects. Let's take a look at a more detailed example of filtering so it becomes clear.

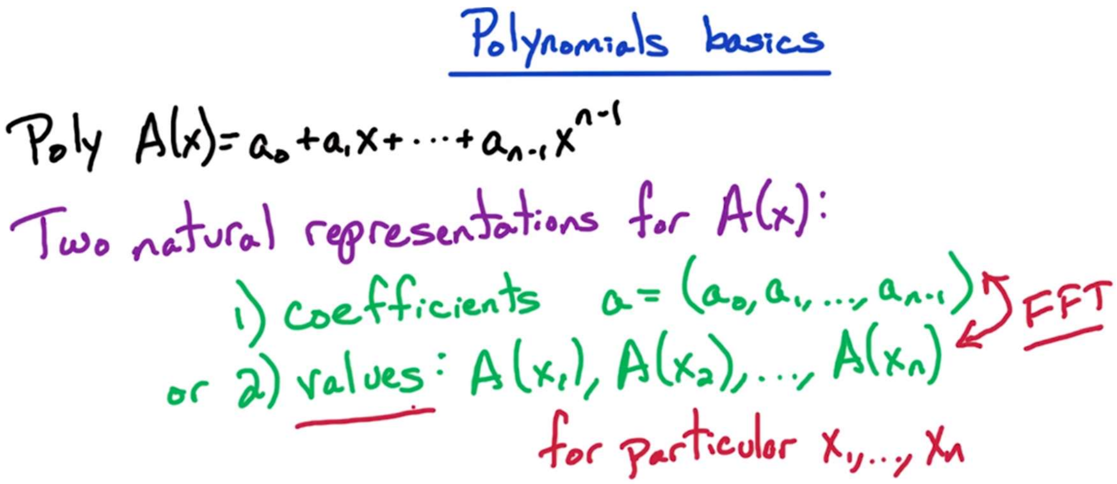
In mean filtering, we have a parameter m and we replace the data point y_j by the mean of the neighboring 2m+1 points and we do this for all j. Now this smooth dataset y_i can be viewed as the convolution of y with a vector f. The vector f is this vector of size 2m+1. To smooth the dataset in this way, we compute its convolution with this simple vector f.

Now, of course, we can smooth in more sophisticated ways. For example, we can replace this factor f by a Gaussian function. In this way, points nearby y_j are given more weight.   In particular, a Gaussian filter uses the following vector f. This quantity, z, is just a normalizing factor so that the sum of the entries in this vector sum up to one.

Now, there are many other filters one might consider. A different type of filter one might consider is Gaussian blur. This is used to add some visual effect to an image, in particular to 'blur' an image. In particular, Gaussian blur applies a two dimensional Gaussian filter to an image. Now let's get back to our original problem of how do we compute the convolution of a pair of vectors.

(Slide 5) Polynomials Basics

***



Polynomials basics

Poly $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$

Two natural representations for A(x):

1) coefficients $a = (a_0, a_1, \ldots, a_{n-1})$

FFT

or 2) values: $A(x_1), A(x_2), \ldots, A(x_n)$

for particulor $x_1, \ldots, x_n$

And let's look at some basic properties of polynomials. Consider the polynomial A(x). There are two natural ways of representing the polynomial A(x).

The first way is by its coefficients - this is the vector A that we've been considering so far.

A second way is by looking at the value of this polynomial A(x) at n points. So, we take n points x1, x2, up to xn and we evaluate this polynomial A(x) at these n points.

The key fact is that a polynomial of degree n-1 is uniquely determined by its values at any n distinct points. So that this statement makes intuitive sense, the example to keep in mind is that of a line. A line is a degree one polynomial, and a line is defined by any two points on that line. And in general, a degree n-1 polynomial is defined by any n points on that polynomial.

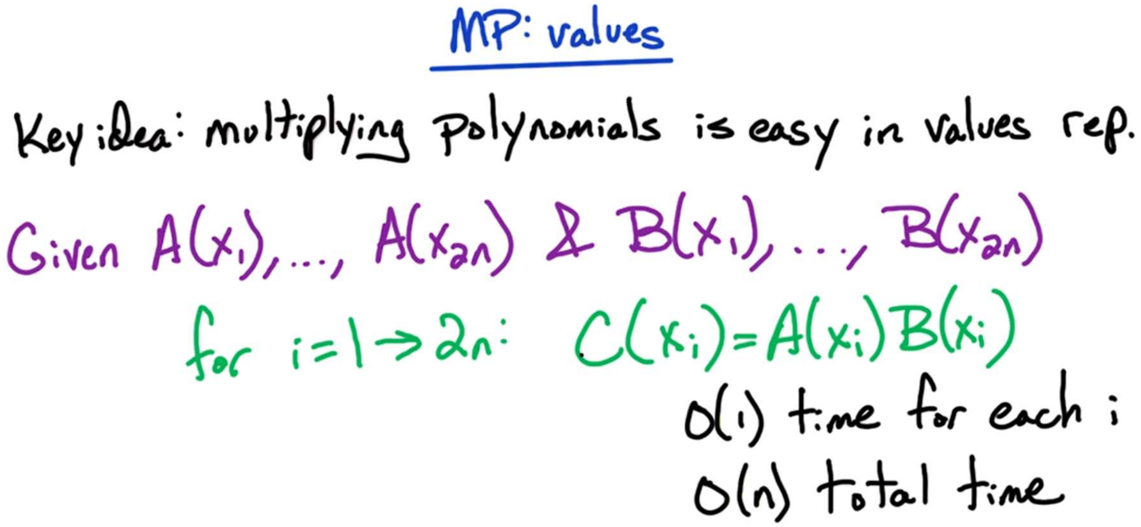
The vector of coefficients is a more natural way to represent a polynomial. However, the values are more convenient for the purposes of multiplying polynomials. We'll see this in a second.

Now, what FFT does … is it converts from coefficients to the values and from values to coefficients. So, it does this transformation between these two representations of the polynomial. And the point is that coefficients is a more convenient way to represent a polynomial oftentimes but the values are more convenient for multiplying polynomials.

So, we'll take the coefficients as input, and we'll convert them to the values. We'll multiply the polynomials and then we'll use FFT to convert it back to the coefficients once again. One important point is that FFT converts from the coefficients to the values … not for any choice of x1 through xn but for a particular, well-chosen, set of points, x1 through xn. And part of the beauty of the FFT algorithm comes from this choice of these points x1 through xn.

Before we dive into FFT, let's take a look at why the values are convenient for multiplying polynomials.

(Slide 6) MP: Values
***



MP: values

Key idea: multiplying polynomials is easy in values rep.

Given $A(x_1),\dots,A(x_{2n})$ & $B(x_1),\dots,B(x_{2n})$

for $i=1\to 2n$: $C(x_i)=A(x_i)B(x_i)$

$O(1)$ time for each i

$O(n)$ total time

One of the key ideas for multiplying polynomials is that multiplying polynomials is easy when we have the polynomials in the values representation. So, suppose that we know the polynomial A(x) evaluated at 2n points, x1 through x_2n, and we know this polynomial B(x) evaluated at the same 2n points. Then we can compute the product polynomial C(x) at these 2n points by just computing the product of two numbers for each i.

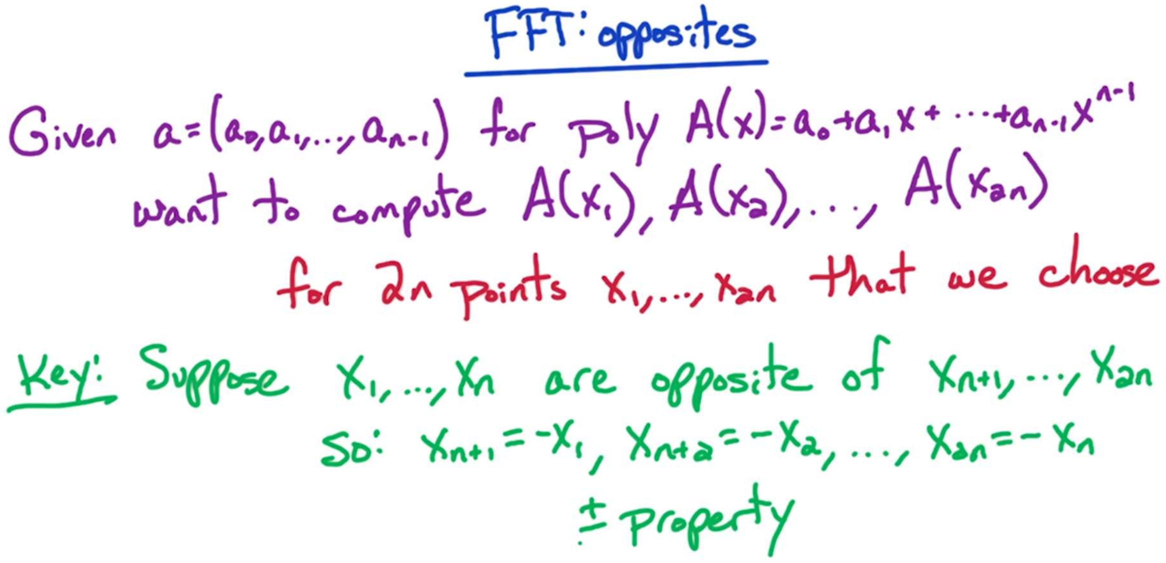
So C(xi) is the product of the number A(xi) and B(xi). This is just a number, and this is just a number. Since C(xi) is just the product of two numbers, it takes O(1) for each i, and therefore it takes O(n) total time to compute this product polynomial.

Now why do we take A and B at 2n points? Well, C is a polynomial of degree at most 2n-2. So we need at least 2n-1 points. So 2n points suffices. The summary is that, if we have the value of these polynomials, A(x) and B(x), at n, or 2n points, then we compute this product polynomial at the same points in O(n) total time.

So what we're going to do is we're going to use FFT to convert from the coefficients to the values - this is going to take O(n log n) to do this conversion, and then I'll take O(n) total time to compute the product polynomial at these 2n points, and then we do FFT to convert back from the value of C(x) at these two n points … back to the coefficients for C(x) …and that again will take O(n log n). So let's dive in now to see how we do FFT, which converts from the coefficients to the values.

(Slide 7) FFT: Opposites

***



FFT: opposites

Given $a=(a_0,a_1,\dots,a_{n-1})$ for poly $A(x)=a_0+a_1x+\cdots+a_{n-1}x^{n-1}$

want to compute $A(x_1), A(x_2), \dots, A(x_{2n})$

for 2n points $x_1,\dots,x_{2n}$ that we choose

Key: Suppose $x_1,\dots,x_n$ are opposite of $x_{n+1},\dots,x_{2n}$

so: $x_{n+1}=-x_1,\ x_{n+2}=-x_2,\ \dots,\ x_{2n}=-x_n$

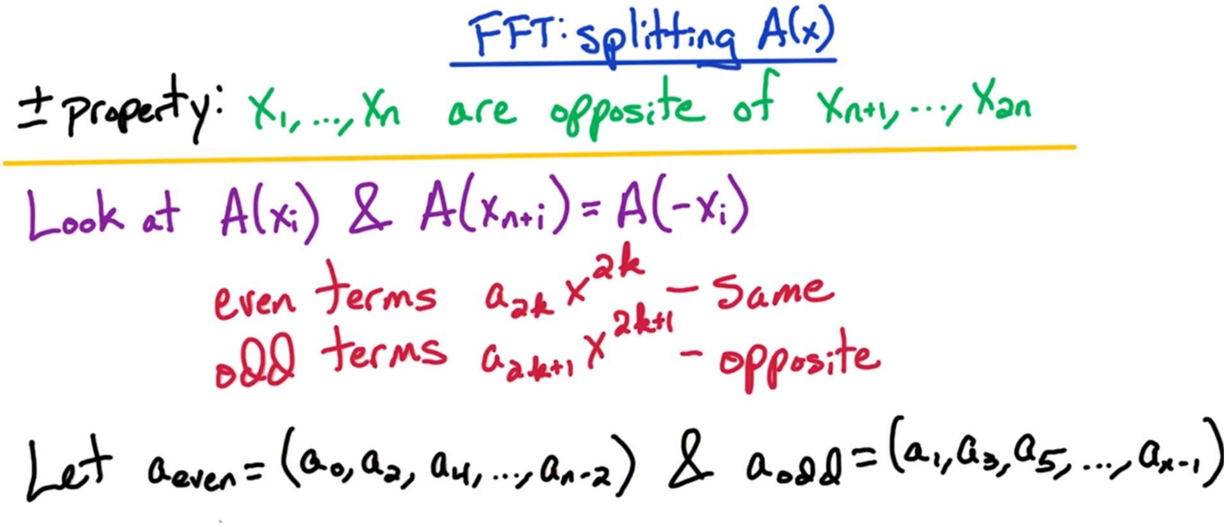
± property

Now here's the smaller computational task that we're focusing on now. We're given this vector a = (a0,a1,…,an-1). These correspond to the coefficients for the polynomial A(x) and we want to compute the value of this polynomial A(x) at 2n points, x1 through x2n. And the key point is that we get to choose these 2n points x1 through x2n.

How do we choose them? Well that's our main task. A crucial observation is … suppose that the first n points are opposite of the next n points. In other words xn+1 = -x1. xn+2 = -x2, and so on. The last one is x2n = -xn. Let's suppose that our 2n points satisfy this property, which we'll call the plus/minus (+/-) property, and let's see how this +/- property works.

(Slide 8) FFT: Splitting A(x)

***



FFT: splitting A(x)

± property: $x_1, \ldots, x_n$ are opposite of $x_{n+1}, \ldots, x_{2n}$

Look at $A(x_i)$ & $A(x_{n+i}) = A(-x_i)$

even terms $a_{2k} x^{2k}$ – Same

odd terms $a_{2k+1} x^{2k+1}$ – opposite

Let $a_{even} = (a_0, a_2, a_4, \ldots, a_{n-2})$ & $a_{odd} = (a_1, a_3, a_5, \ldots, a_{x-1})$

Once again the +/- property says that the first n points are opposite of the next n points. And let's see how this +/- property is useful for recursively computing A(x) at 2n points.

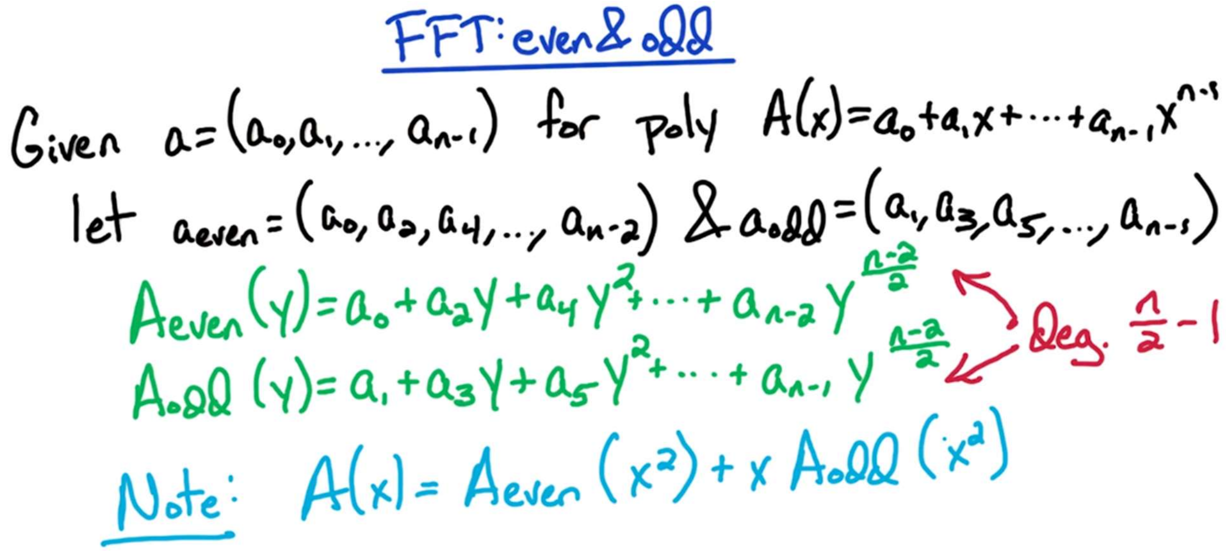
Let's look at A evaluated at the point xi and A evaluated at the point xn+i. Since we're assuming the +/- property, these two points are opposites of each other.

Now let's break up this polynomial A(x) into the even terms and the odd terms. Even terms are of the form a_2k x^2k (i.e., notation for a_{2k} x^{2k}). Since the power is even, this is the same for xi and -xi. So, these even terms are the same for the point xi and the point xn+i. Now, what about for the odd terms? Well. since it's an odd power, it's going to be the opposite for xi and xn+i.

Now given this observation about the even terms and the odd terms, it makes sense to split up A(x) into the even terms and the odd terms. So let's partition the coefficients for the polynomial A(x) into those coefficients for the even terms. Let's call this vector a_even and into the coefficients for the odd terms. Let's call this vector a_odd.

(Slide 9) FFT: Even & Odd

***



FFT: even & odd

Given $a=(a_0,a_1,\dots,a_{n-1})$ for poly $A(x)=a_0+a_1x+\cdots+a_{n-1}x^{n-1}$

let $a_{even}=(a_0,a_2,a_4,\dots,a_{n-2})$ & $a_{odd}=(a_1,a_3,a_5,\dots,a_{n-1})$

$A_{even}(y)=a_0+a_2y+a_4y^2+\cdots+a_{n-2}y^{\frac{n-2}{2}}$

$A_{odd}(y)=a_1+a_3y+a_5y^2+\cdots+a_{n-1}y^{\frac{n-2}{2}}$

Deg. $\frac{n}{2}-1$

Note: $A(x)=A_{even}(x^2)+x\,A_{odd}(x^2)$

So, given this vector of coefficients for this polynomial A(x), define a_even as the coefficients for the even terms and a_odd as the coefficients for the odd terms. Then we can look at the polynomials defined by this vector of coefficients.

This vector a_even defines this polynomial Aeven(y). I have used a variable 'y' to avoid confusion with A(x). Notice that a2 is the quadratic term in A(x), but in Aeven, it's the linear term since it's the second coefficient. And similarly, a4 is going to be the quadratic term.

Similarly this vector a_odd defines this polynomial Aodd(y). Notice that the degree of these two polynomials is n/2 - 1. So we took a polynomial A(x) of degree n-1 and we defined a pair of polynomials of degree n/2 - 1.

How does a polynomial A(x) relate to the polynomial Aeven and the polynomial Aodd? Well, notice if I take the polynomial Aeven and I evaluate it at y=x^2, then I get the even terms of the polynomial A(x). Similarly, if I evaluate the polynomial Aodd at the point y=x^2, then I get the odd terms except it's off by one in the exponent. So I multiply by x. So A(x) = Aeven(x^2) + x Aodd(x^2).

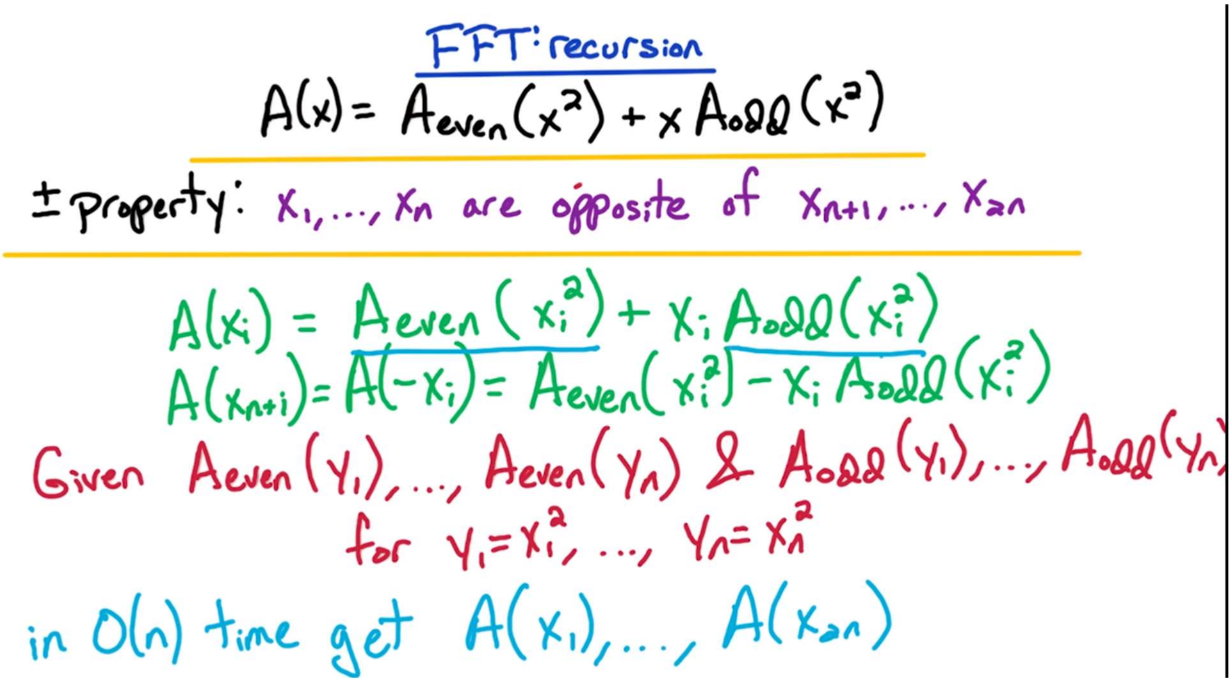
At this point you just start to see the semblance of the idea of the Divide and Conquer approach:

We started with a polynomial of degree n-1. And now we have defined a pair of polynomials of degree n/2 - 1. So we went down from n -> n/2 and we got two subproblems, Aeven and Aodd. And if we want A(x), then it suffices to know Aeven and Aodd at x^2.

So we've got two subproblems of half the size. Now the degree of this polynomial A(x) went down from n-1 to n/2 - 1 for these two smaller polynomials. However if we need A(x) at 2n points, we still need Aeven and Aodd at 2n points (the square of these original 2n points). So, the degree of these polynomials went down by a factor of two. But the number of points we need to evaluate them at hasn't gone down by a factor of two yet. This is where we're going to use the +/- property.

(Slide 10) FFT: Recursion

***



FFT: recursion

$$A(x) = A_{even}(x^2) + x A_{odd}(x^2)$$

± property: $x_1, \dots, x_n$ are opposite of $x_{n+1}, \dots, x_{2n}$

$$A(x_i) = \underline{A_{even}(x_i^2)} + \underline{x_i A_{odd}(x_i^2)}$$

$$A(x_{n+i}) = A(-x_i) = A_{even}(x_i^2) - x_i A_{odd}(x_i^2)$$

Given $A_{even}(y_1), \dots, A_{even}(y_n)$ & $A_{odd}(y_1), \dots, A_{odd}(y_n)$

for $y_1 = x_1^2, \dots, y_n = x_n^2$

in $O(n)$ time get $A(x_1), \dots, A(x_{2n})$

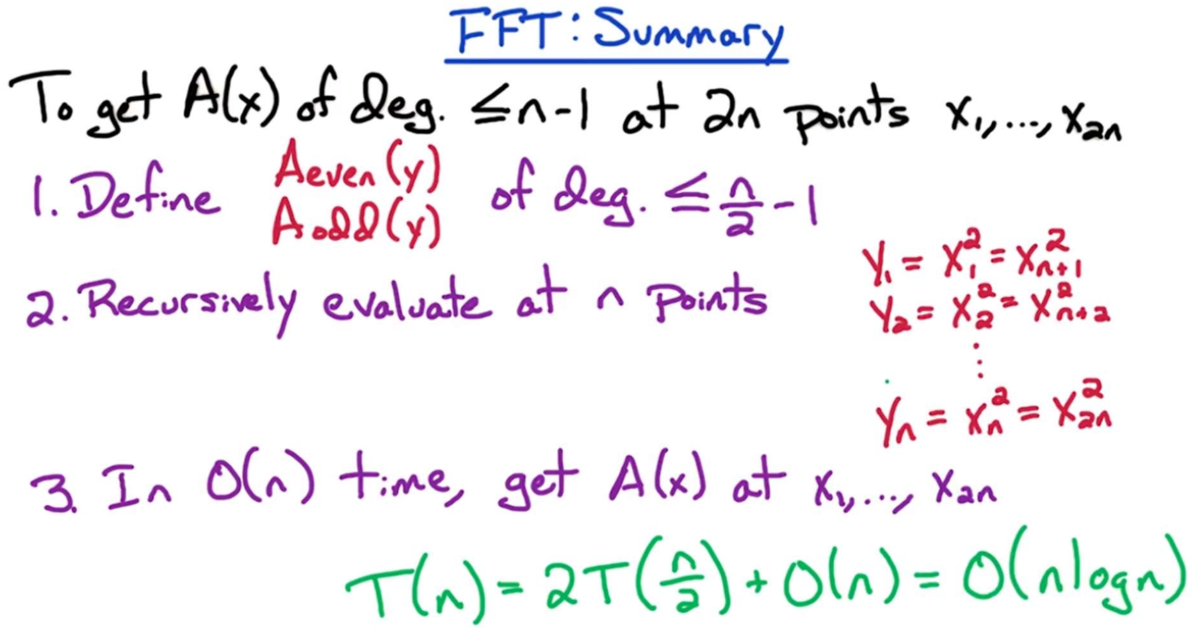
Given this polynomial A(x), we would define this pair of polynomials: Aeven and Aodd. They satisfy the following identity: A(x) = Aeven(x^2) + x Aodd(x^2).

Now let's suppose that the 2n points that we want to evaluate A(x) satisfy the +/- property. So, the first n points are the opposite of the next n points. So let's look at A evaluated at the point xi and A evaluated the point x_n+i. Because of the +/- property, this is A(xi) and A(-xi).

Plugging this into our earlier formula, we have that A(xi) = Aeven(xi^2) + xi Aodd(xi^2). And similarly, for A(-xi) = Aeven(xi^2) – xi Aodd(xi^2). So notice to get A at these two different points, we need Aeven and Aodd at these same points.

Our conclusion is that if we're given Aeven and Aodd at these n points y1 through yn which are the square of these 2n points. Notice that since these 2n points satisfy this +/- property, the square of these 2n points are these n points. Then in O(n), we get A evaluated at these 2n points. In particular to evaluate A at the point xi, it takes O(1) time given the value of Aeven at this point and Aodd at this point. And similarly to evaluate A at the point xn+i, it takes O(1) given the value of Aeven at this point and Aodd at this point.

(Slide 11) FFT: Summary
***



FFT: Summary

To get $A(x)$ of Deg. $\leq n-1$ at $2n$ points $x_1, \dots, x_{2n}$

1. Define $\begin{matrix} A_{even}(y) \\ A_{odd}(y) \end{matrix}$ of deg. $\leq \frac{n}{2} - 1$

2. Recursively evaluate at n points

$$\begin{aligned} y_1 &= x_1^2 = x_{n+1}^2 \\ y_2 &= x_2^2 = x_{n+2}^2 \\ &\vdots \\ y_n &= x_n^2 = x_{2n}^2 \end{aligned}$$

3. In $O(n)$ time, get $A(x)$ at $x_1, \dots, x_{2n}$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

Now, let's summarize our approach. We have this polynomial, A(x) of degree <= n-1, and we want to evaluate this polynomial at 2n points, x1 through x_2n. And we get to choose these 2n points however we want. And we're looking at how we choose these points.

One very minor point that I wanted identify now is why we consider this polynomial at 2n points instead of n points. And, in fact, later we'll go back and we'll look at it at n points, instead of 2n points. But for now, we want this polynomial A(x) at 2n points. Why? Because of our application to polynomial multiplication.

Recall, our first step in our construction is to define this pair of polynormials Aeven and Aodd. We do this by taking the even turns of A(x) to define Aeven, and the odd terms define Aodd. Whereas the original polynomial A(x) was of degree <= n-1. Each of these polynomials is of degree <= n/2 – 1. So, the degree went down by half.

Next, we recursively run the FFT algorithm on this pair of polynomials, and we'll evaluate these pair of polynomials at n points. What are the n points? The n points that we evaluate this pair of polynomials at are these points y1 through yn, which are the squares of these 2n points.

Since the original 2n points satisfy the +/- property, then x1 is the opposite of x_n+1. So these squares are the same. That's our first point y1. x2 is the opposite of x_n+2. So their squares are the same and that's y2, and so on, up to yn, which is (xn)^2 and (x_2n)^2.

Why do we want this pair of polynomials at the square of these 2n points? Well recall … to evaluate this polynomial A at this point x, it's straight forward if we know Aeven and Aodd at the point x^2. So if we know this pair of polynomials at the square of these points then it's straightforward to get A at these 2n points.

In particular, in O(1) per point, we can evaluate this polynomial A(x) at that point, using Aeven and Aodd at the square of that point. So, in O(n) total time, we can get this polynomial A(x) at these 2n points, using Aeven and Aodd at these n points, which are the squares of these 2n points. This is the high level idea of our divide and conquer algorithm for FFT.

What's the running time of this algorithm? Well, let T(n) denote the running time of input of size n, we have two sub problems of exactly half the size. So, 2 T(n/2). It takes us O(n) to form these two polynomials, and it takes us O(n) time to merge their solution together, to get the solution to the original problem. So we get the following recurrence: T(n) = 2 T(n/2) + O(n). This is the same recurrence in the classic mergesort and many of you must recall that it solves to O(n log n).

So it looks like we have an O(n log n) algorithm to solve this problem. All we need is that these 2n points satisfy the +/- property. So the first n are opposite of the second n.

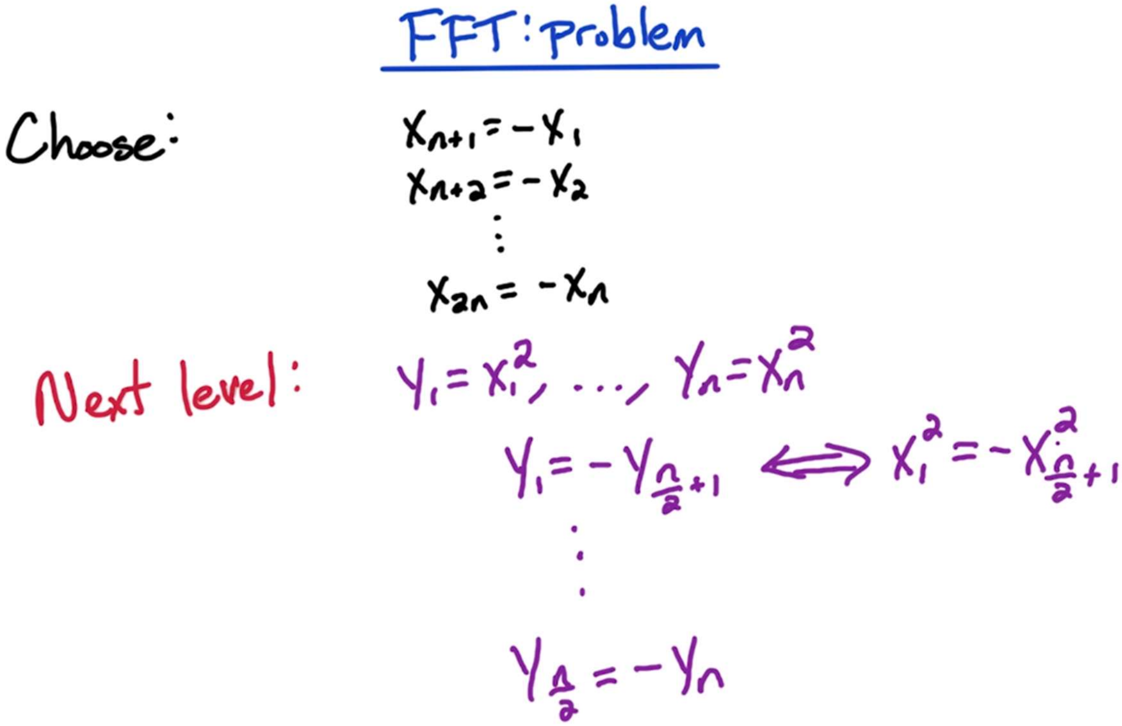
But notice we're going to recursively run this problem on this pair of polynomials Aeven and Aodd with the square of these points. So again, we're going to need the +/- property for this subproblem and for all smaller subproblems.

Looking for this subproblem, we have y1 through yn. We need the first n/2 to be the opposite of the second n/2. It will be straightforward to define 2n points which satisfy the +/- property. The challenge will be to define 2n points which satisfy the +/- property, and for all recursive subproblems to also satisfy the +/- property.

So let's dive into how we achieve this.

(Slide 12) FFT: Recursive Problem

***



FFT: problem

Choose:

$$x_{n+1} = -x_1$$
$$x_{n+2} = -x_2$$
$$\vdots$$
$$x_{2n} = -x_n$$

Next level: $y_1 = x_1^2, \dots, y_n = x_n^2$

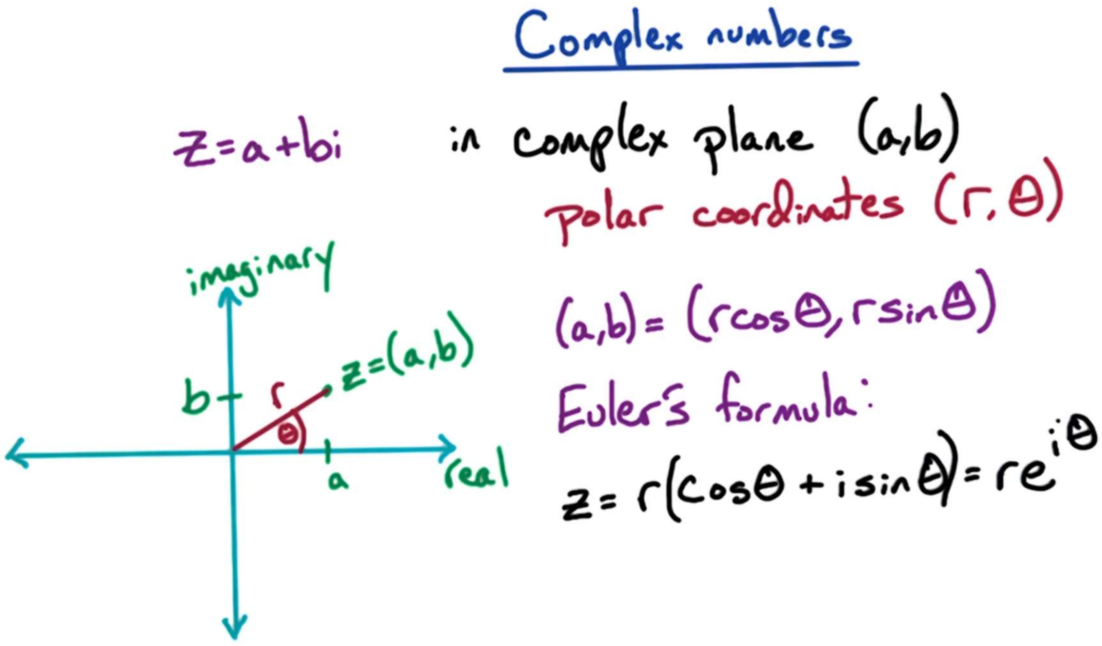
$$y_1 = -y_{\frac{n}{2}+1} \iff x_1^2 = -x_{\frac{n}{2}+1}^2$$
$$\vdots$$
$$y_{\frac{n}{2}} = -y_n$$

In order to get 2n points which satisfy the +/- property, we are going to choose points so that the second n are the opposite of the first n points. What happens in the next level of the recursion? Then the points we're considering are the square of these 2n points. So these are the n points which are x1^2 square up to xn^2 and we want these n points to also satisfy the +/- property. Let's assume that n is a power of two. Then we need that y1 is the opposite of y_n/2+1 and up to y_n/2 is the opposite of yn. What does this mean in terms of x? This means that x1^2 is the opposite of (xn/2+1)^2.

But the square of a number is always positive, so this is a positive number, and this is a positive number. So how can they be opposites of each other? Well, it's impossible if we're working with real numbers. The only way to achieve this is by looking at complex numbers. So, we can easily achieve the plus +/- property at the top level of the recursion. But, for all further levels of the recursion, in order to achieve the +/- property we need to look at complex numbers.

So let's do a quick review of complex numbers and then we'll see the appropriate choice of these 2n numbers.

(Slide 13) Review: Complex Numbers
***



Complex numbers

$z=a+bi$ in complex plane (a,b)

polar coordinates (r,θ)

$(a,b)=(r\cos\theta, r\sin\theta)$

Euler's formula:

$z=r(\cos\theta+i\sin\theta)=re^{i\theta}$

We saw that we need to consider complex numbers for our choice of the n/2 points where we evaluate the polynomial A(x). Here, I'll give a brief review of the relevant concepts regarding complex numbers.

Some of you may get a bit scared at the use of complex numbers; but, the mathematics involved is fairly simple, so don't get intimidated at all. And the final algorithm that we get is very simple and very beautiful.

We have a complex number z = a + bi; a is the real part and b is the imaginary part. It's often convenient to look at complex numbers in the complex plane.

In the complex plane, one of the axis corresponds to the real component and one of the axis corresponds to the imaginary component. So, for this number a+bi, we look at a in the real axis, b in the imaginary axis, and we look at this point (a,b). This number z corresponds to the point (a,b) in the complex plane.

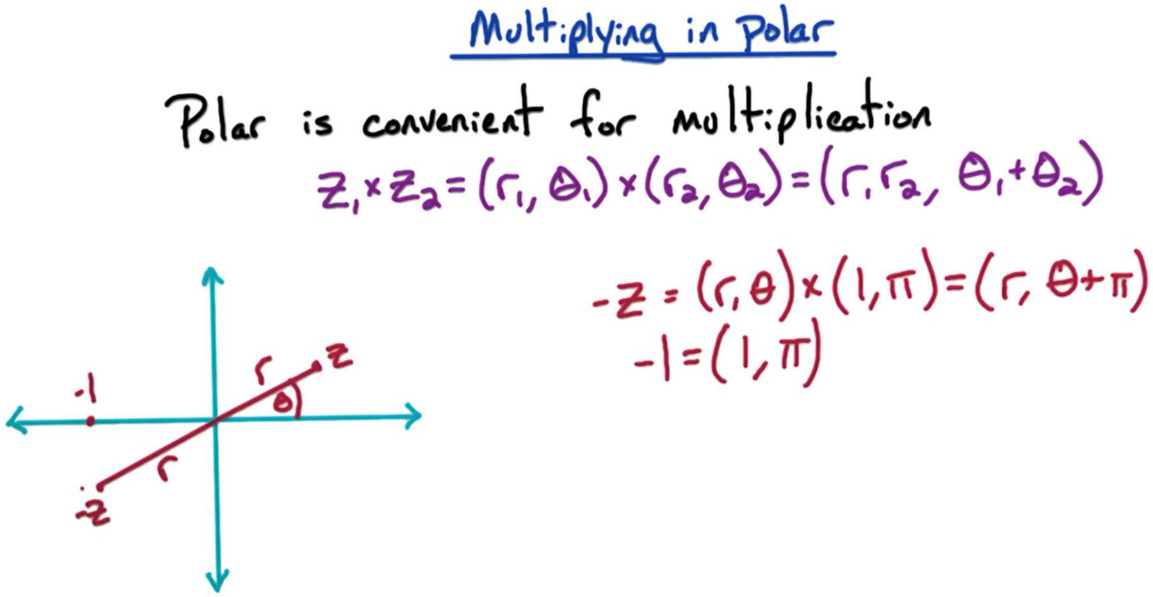
Now, there's another way to describe this point … this number z … and it's called the polar coordinates. In polar coordinates, we look at the length of this vector from the origin to this point z. Let's call r, the length and we look at this angle θ. This number z in polar coordinates is (r, θ). Just as we had two representations of a polynomial … either the coefficients or the values, we also have two representations of a complex number … either with the cartesian coordinates (a,b) or the polar coordinates (r, θ).

Now, it turns out that the polar coordinates are more convenient for certain operations such as multiplication. Before we delve into properties of polar coordinates, let's review some basic properties:

- First off, how do we convert between polar coordinates and complex or cartesian coordinates? Well, if you remember a little bit of trigonometry, you'll recall that a, this length a is equal to r cos(θ) and this length b is equal to r sin(θ). So, this gives us a way to convert between cartesian coordinates and polar coordinates.
- Now there's a third way of representing complex numbers. This is given to us by Euler's formula. This gives us a quite compact representation of complex numbers. The basic fact is Euler's formula which says that cos(θ) + i sin(θ) = $e^{i\theta}$.

  Then we can multiply both sides by r and then we have that this number z, this complex number z, z = r $e^{i\theta}$. The basic idea of the proof of this Euler's formula is by looking at the Taylor expansion of exponential function cosine and sine function.

(Slide 14) Multiplying in Polar
***



Multiplying in Polar

Polar is convenient for multiplication

$z_1 \times z_2 = (r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$

$-z = (r, \theta) \times (1, \pi) = (r, \theta + \pi)$

$-1 = (1, \pi)$

As we mentioned earlier polar coordinates are convenient for multiplication. Let's consider an example. Consider z1 and z2, a pair of complex numbers and let's look at their product. Let's say that z1 in polar coordinates is (r1, θ1) and z2 in polar coordinates is (r2, θ2).

Now let's look at the product of these polar coordinates. We simply multiply the lengths and we add up the angles.

Now suppose that I have a point z. I have a complex number z1 and 1 want to look at its opposite. So I want to look at -z … so this corresponds to multiplying z times -1.

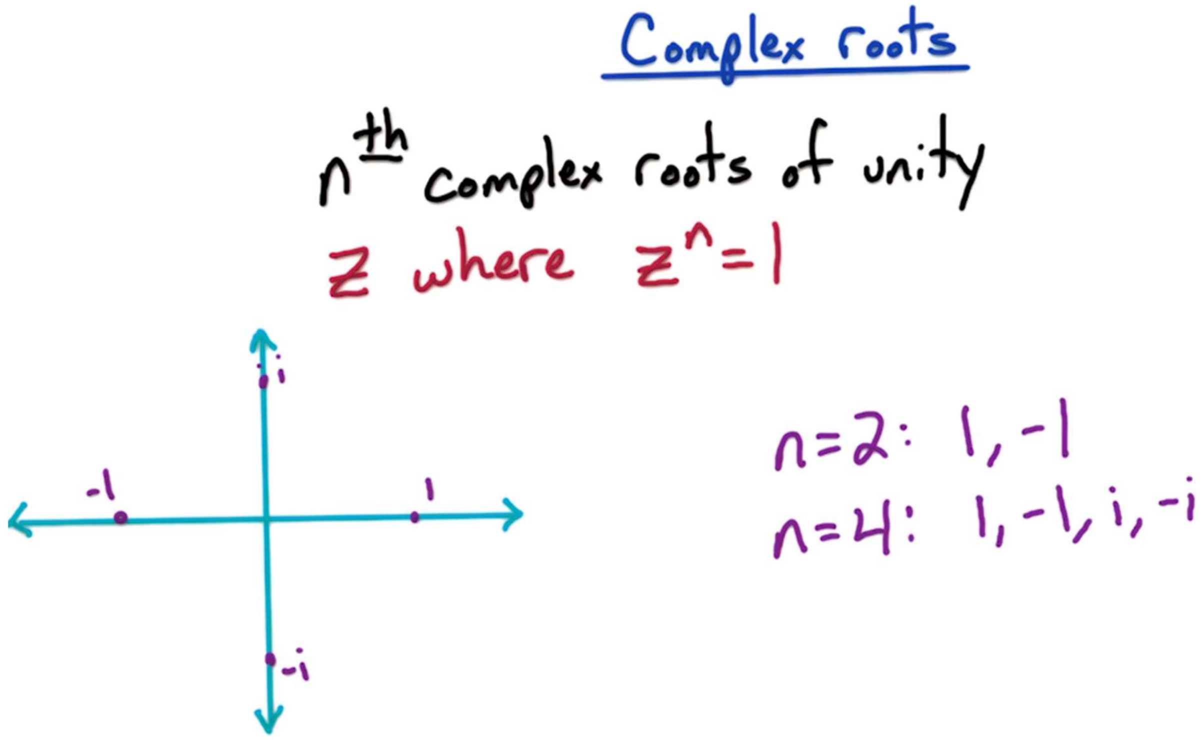
What is -1 in polar coordinates? Here's the point -1 in the complex plane. It's a distance one from the origin and it has angle 180 degrees or pi, so its polar coordinates are (1, pi). And let's say z is this point at polar coordinates (r,θ), then -z corresponds to the product (r,θ) times -1 in polar coordinates which is (1,pi).

Then using this above rule we have that -z is at the polar coordinates (r,θ) + pi. So, the point -z is the reflection of the point z. It's just the same distance from the origin. It's a distance r from the origin, and instead of going in angle θ, we go in angle θ+π.

So, if you view the complex numbers in the complex plane, it's easy to find their opposites by looking at their reflection.

(Slide 15) Complex Roots

***



Complex roots

$n^{\underline{th}}$ complex roots of unity

z where z^n=1

$n=2$: $1, -1$

$n=4$: $1, -1, i, -i$

Now we're going to consider the numbers which are known as the nth complex roots of unity.

Let's break this down. So, by unity we mean one. Instead of nth roots, let's consider the case n equals two. So we're looking at square roots of unity. The square roots of 1 are 1 and -1.

Now consider the case n=4. So we're looking at the fourth roots of unity. What numbers raised to the fourth power equal one? Well, if we're looking at real numbers, the case is 1 and -1, but if we look at complex numbers we have more considerations.

Notice that i and -i, if we raise them to the fourth power we also get one. So the nth roots of unity for real numbers are just 1 and -1, but if we look at complex numbers, then there's much more to consider.

In general, the nth complex roots of unity are defined as those numbers z's … those complex numbers z where when we raise z to the nth power we get 1 (i.e., z where z^n = 1). As we just mentioned, for the case n=2, the square roots of unity are the numbers 1 and -1. So, it doesn't make any difference whether we're doing real numbers or complex numbers for this case n=2.

For the case n=4, the fourth roots of unity are the four numbers 1, -1, and, in addition, i and -i.

Now, what are the eighth roots of unity, and, in general, what are the nth roots of unity?

(Slide 16) Roots: Notation

***

n^{th} roots

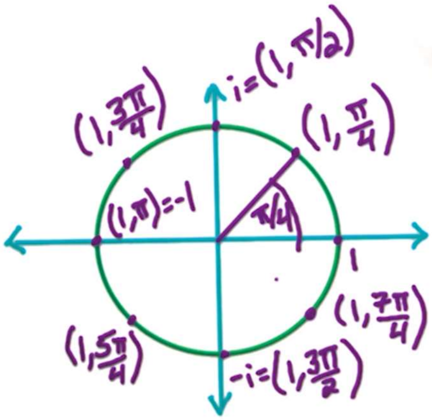
z where $z^n = 1 = (1, 2\pi j)$ for integer j

$z = (r, \theta)$

$(r^n, n\theta) = z^n = 1 = (1, 2\pi j)$

$r^n = 1 \Rightarrow r = 1$

$n\theta = 2\pi j \Rightarrow \theta = \frac{2\pi j}{n}$



Now we're trying to find those z where z^n = 1. And let's draw the number 1 on our complex plane. What is its point in polar coordinates? Its distance 1 obviously, from the origin. What's the angle? The angle is zero. It's also 2 π or 4 π or 6 π or 2 π times j for any integer j. And note that j can also be negative number. So the angle is any multiple of 2 π.

Now let's say z = (r, θ) in polar. We're looking at those z where z^n = 1. So it also equals (1, 2π j). Recall the expression for multiplying two complex numbers in polar coordinates. So look at z^n. So z = (r,θ). So z^n is (r^n, nθ) (we added up the angles - So we get n θ). Now, we're assuming this equals 1. So, what does r equal? and what does θ equal? Well, r^n = 1. So, then we know that r = 1.

So what does that mean? That means that all the complex roots of unity lie on this unit circle, this circle of radius 1. Where do they lie on the circle? Well, we have to look at the angle. n θ = 2 π j. Solving for theta. We have θ = 2πj/n.

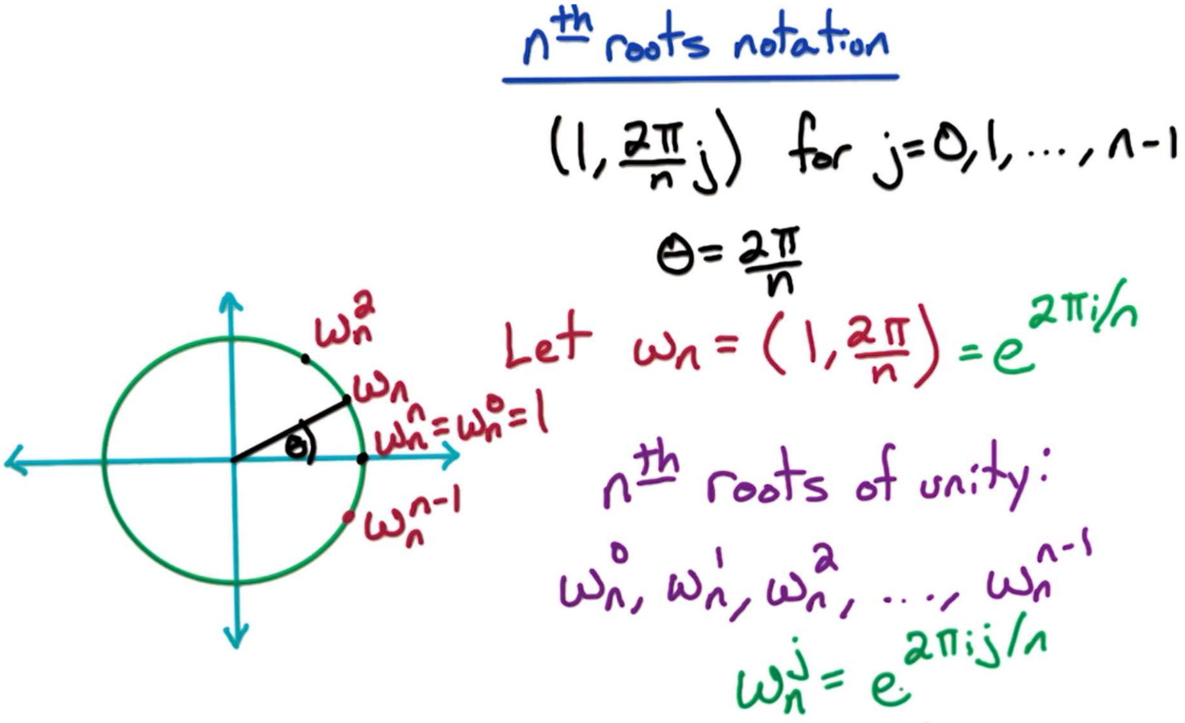
Let's take the case n=8 and see what this looks like on the unit circle:

- The case j=0, corresponds to angle 0. So this is the point 1. That's good because we know that the point 1, that number 1 is an nth root of unity.
- Let's look at j =1. So we got 2π/n. What does this correspond to, this angle 2π/n? It's like you took the whole π, the whole circle, and you subdivided it into equal slices. For the case n=8, this angle is π / 4. So this is the point (1, π / 4).

- j=2 corresponds to this point which is i which is (1, π / 2).
- Next point is (1, 3 π / 4). And we get (1, π) which is -1.
- And we get (1, 5 π / 4), and so on.
- Notice that when we get to j=8, we repeat. We get back to this point 1. j equals 9, j equals 10, and so on.

So notice there are n distinct values, and what we are doing is we're starting at the point 1 and we're taking step sizes of 2π/n.

(Slide 17) Roots: Notation
***



$n^{\underline{th}}$ roots notation

$(1, \frac{2\pi}{n} j)$ for $j = 0, 1, \ldots, n-1$

$\theta = \frac{2\pi}{n}$

Let $\omega_n = (1, \frac{2\pi}{n}) = e^{2\pi i/n}$

$n^{\underline{th}}$ roots of unity:

$\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$

$\omega_n^j = e^{2\pi i j/n}$

And what we just saw are that the nth roots of unity correspond to the points in polar coordinates (1, 2π/n j) for j=0,1,…,n-1.

j=0 corresponds to this point, which is 1. Then we take step sizes of angle θ, where θ equals 2π/n, this gives us j=1 , j=2 and so on and we keep taking the equal size steps around the unit circle.

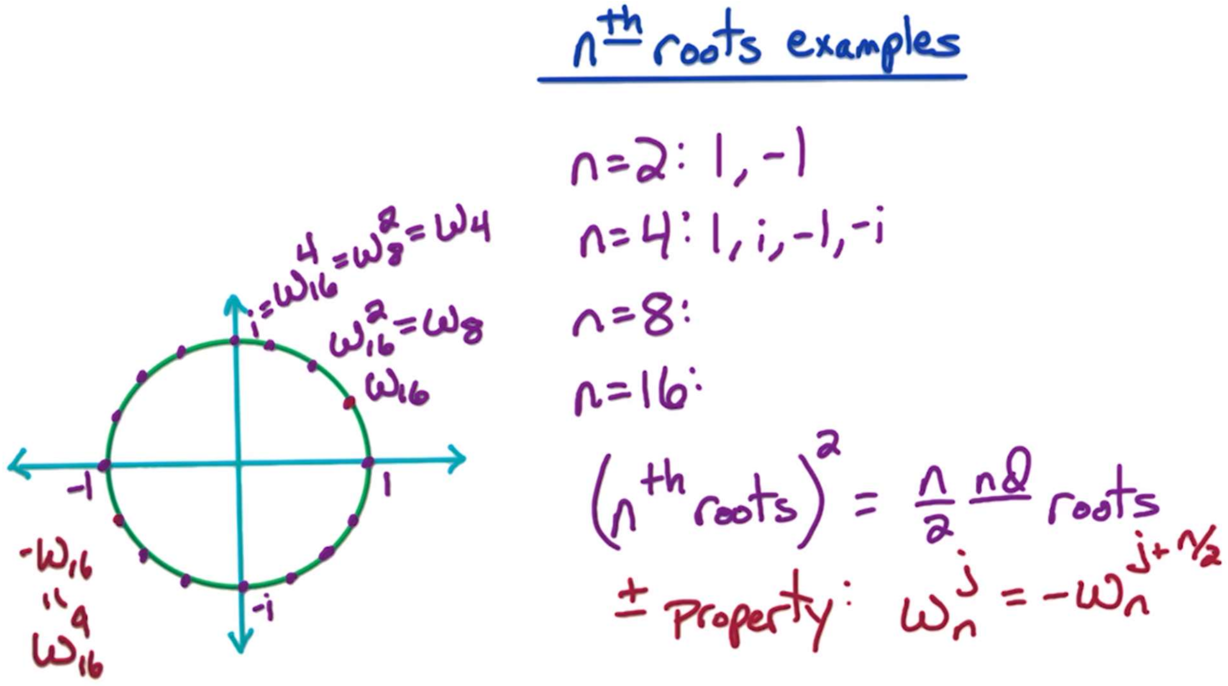
Let's introduce a little bit of notation to make it more convenient for expressing these nth roots of unity. So let's let ω_n denote the point corresponding to j=1. So it's this 1 right here … so this is ω_n. Now what does the next point correspond to? Well, we just doubled the angle so it's just a square of this point ω_n. And the next one is going to be the cube of this point. And the jth point is going to be the jth power. So the last one is going to be (ω_n)^n-1 and then we have (ω_n)^n, which is the same as (ω_n)^0 which is the number 1.

So the nth roots of unity are the following n numbers, it's (ω_n)^0, (ω_n)^1. (ω_n)^2,…, (ω_n)^n-1. These n numbers in polar coordinates are (1, 2π/n j), where j varies from 0 -> n-1.

And recall Euler's formula, so this number ω_n = e^2πi/n. So the jth of the nth roots of unity, (ω_n)^j = e^2πij/n, where j varies between 0 and n-1.

(Slide 18) Roots: Examples

***



$n^{\underline{th}}$ roots examples

$n=2: 1, -1$

$n=4: 1, i, -1, -i$

$n=8:$

$n=16:$

$(n^{th} \text{ roots})^2 = \frac{n}{2} \underline{nd}$ roots

± property: $\omega_n^j = -\omega_n^{j+n/2}$

$i = \omega_{16}^4 = \omega_8^2 = \omega_4$
$\omega_{16}^2 = \omega_8$
ω_{16}
-1
1
$-i$
$-\omega_{16}$
"
ω_{16}^9

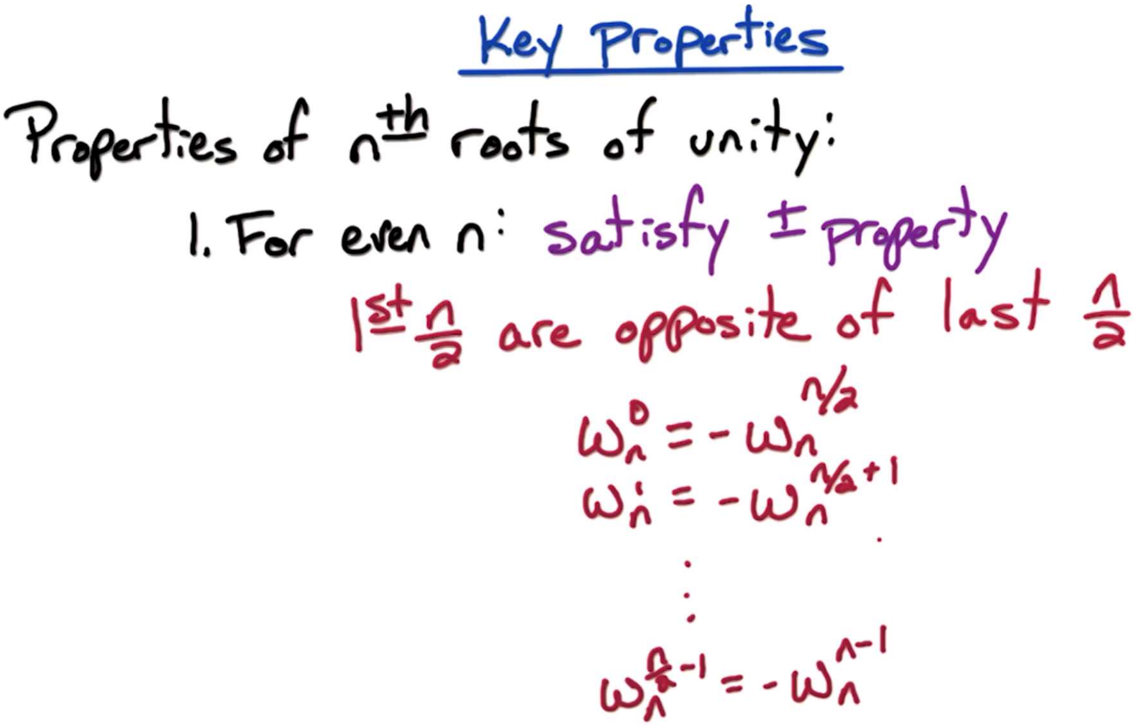
- We saw that for the case n=2, the square roots of unity are the points 1 and -1.
- For the case n=4, we also have the points i and -i so we have the four points 1, i, -1, -i. This corresponds to going around the unit circle in step sizes of π/2.
- And for n=8, we further subdivide the unit circle. Now we go step sizes of π/4
- and then how do we get the points for n=16? We further subdivide. So this corresponds to ω_16. The next one is (ω_16)^2 and notice that it's also ω_8. It's the first one of the eighth roots of unity. And this point i is (ω_16)^4, it's also (ω_8)^2, it's the second one of the 8th roots. And it's the first of the 4th roots.
- So take the nth roots and square them. And, let's suppose that n is a power of 2 as in these examples. So what happens when you take the 16th root squared? Well this guy goes there, this one goes there, this one goes there, this one goes there. What do you get? You get every other one of the 16th roots, which are the 8th roots so the 16th root squared are the 8th roots. And when n is a power of 2, the nth root squared are the n/2nd second roots.

The other key property is this +/- property. Take this point ω_16. What is the opposite of it? It's this point right here. This is - ω_16. It's also a 16th root of unity. Which one is it? Well, this is the 1st first, this is the 8th, this is the 9th. This is the point ω_16^9.

And, in general, for even n, ω_n^j = -ω_n^(j+n/2). I'm going up to n/2, so I'm going an extra π around, so I'm getting the reflection.

See DC4: FFT – Part 1 Complex Roots Practice Quiz in the lectures

(Slide 20) Key Property Opposites
***



Key Properties

Properties of $n^{\underline{th}}$ roots of unity:

1. For even n: satisfy ± property

$1^{\underline{st}}$ $\frac{n}{2}$ are opposite of last $\frac{n}{2}$

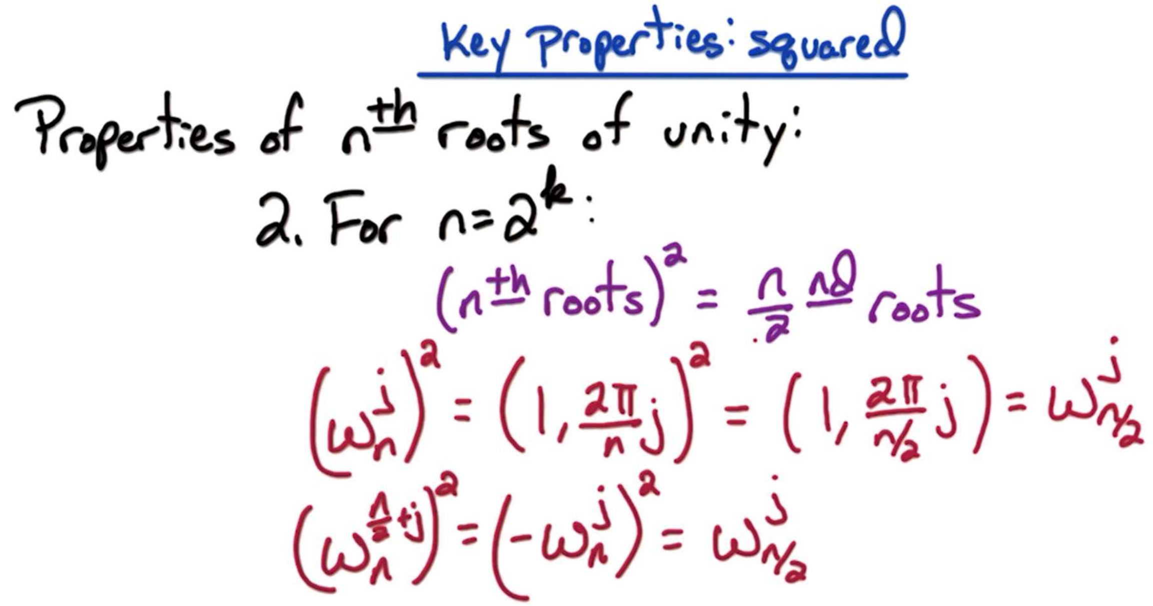
$$\omega_n^0 = -\omega_n^{n/2}$$
$$\omega_n^1 = -\omega_n^{n/2+1}$$
$$\vdots$$
$$\omega_n^{\frac{n}{2}-1} = -\omega_n^{n-1}$$

There are two key properties of the nth roots of unity which we're going to utilize:

- The first holds for even n - the nth root satisfies the +/- property. This was the key property for our Divide and Conquer approach. The first n/2 of the nth roots of unity are opposite of the last n/2. In other words, ω_n^0 = - ω_n^n/2. And in general, ω_n^j = -ω_n^(n/2+j).

  So the nth roots of unity look like a perfect choice for our divide and conquer approach, because they satisfy the +/- property.

(Slide 21) Key Property Squares
***



Key Properties: squared

Properties of n^{th} roots of unity:

2. For $n=2^k$:

$(n^{th}\text{ roots})^2 = \frac{n}{2}^{nd}$ roots

$$(\omega_n^j)^2 = \left(1, \frac{2\pi}{n} j\right)^2 = \left(1, \frac{2\pi}{n/2} j\right) = \omega_{n/2}^j$$

$$\left(\omega_n^{\frac{n}{2}+j}\right)^2 = (-\omega_n^j)^2 = \omega_{n/2}^j$$

The second key property holds when n is a power of two. If we look at the nth roots squared, we get the n over second roots. So we take the jth of the nth roots, so omega sub n to the jth power, and we square it. So in polar, we're taking the point $(1, 2\pi/n\ j)^2$. So we simply double the angle … So we get the point (1, 2π/(n/2) j), which is (ω_n/2)^j … So the jth of the nth roots squared is the jth of the n/2^{nd} roots. And similarly, if we take the (n/2+j)th of the nth roots and we square it, well, this is just the opposite of this. So, when we square the negative of it we get the same thing.

So why do we need this property that the nth root squared are the n/2^{nd} roots? Well, we're going to take this polynomial A(x), and we want to evaluate it at the nth roots. Now these nth roots satisfy the +/- property, so we can do a divide and conquer approach.

But then, what do we need? We need to evaluate Aeven and Aodd at the square of these nth roots, which will be the n/2^{nd} roots. So this subproblem is of the exact same form as the original problem. In order to evaluate A(x) at the nth roots, we need to evaluate these two subproblems, Aeven and Aodd at the n/2^{nd} roots. And then we can recursively continue this algorithm.

Now we're all set to do our FFT algorithm, and the n points where we choose to evaluate the polynomial A(x) are the nth roots of unity.