CSE 548: Analysis of Algorithms

Lecture 4
(Divide-and-Conquer Algorithms: Polynomial Multiplication)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2015

Coefficient Representation of Polynomials

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$$= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

$A(x)$ is a polynomial of degree bound n represented as a vector $a = (a_0, a_1, \cdots, a_{n-1})$ of coefficients.

The *degree* of $A(x)$ is k provided it is the largest integer such that a_k is nonzero. Clearly, $0 \leq k \leq n - 1$.

Evaluating $A(x)$ at a given point:

Takes $\Theta(n)$ time using Horner's rule:

$$A(x_0) = a_0 + a_1 x_0 + a_2 (x_0)^2 + \cdots + a_{n-1} (x_0)^{n-1}$$

$$= a_0 + x_0 \left(a_1 + x_0 \left(a_2 + \cdots + x_0 \left(a_{n-2} + x_0 (a_{n-1})\right) \cdots \right)\right)$$

Coefficient Representation of Polynomials

Adding Two Polynomials:

Adding two polynomials of degree bound n takes $\Theta(n)$ time.

$$C(x) = A(x) + B(x)$$

where, $A(x) = \sum_{j=0}^{n-1} a_j x^j$ and $B(x) = \sum_{j=0}^{n-1} b_j x^j$.

Then $C(x) = \sum_{j=0}^{n-1} c_j x^j$, where, $c_j = a_j + b_j$ for $0 \le j \le n-1$.

Coefficient Representation of Polynomials

Multiplying Two Polynomials:

The product of two polynomials of degree bound n is another polynomial of degree bound $2n-1$.

$$C(x) = A(x)B(x)$$

where, $A(x) = \sum_{j=0}^{n-1} a_j x^j$ and $B(x) = \sum_{j=0}^{n-1} b_j x^j$.

Then $C(x) = \sum_{j=0}^{2n-2} c_j x^j$ where, $c_j = \sum_{k=0}^{j} a_k b_{j-k}$ for $0 \le j \le 2n-2$.

The coefficient vector $c = (c_0, c_1, \cdots, c_{2n-2})$, denoted by $c = a \otimes b$, is also called the *convolution* of vectors $a = (a_0, a_1, \cdots, a_{n-1})$ and $b = (b_0, b_1, \cdots, b_{n-1})$.

Clearly, straightforward evaluation of c takes $\Theta(n^2)$ time.

Convolution

$$\begin{array}{ccccccccccccc} & & & & & & \boxed{a_0} & + & \boxed{a_1 x} & + & \boxed{a_2 x^2} & + & \boxed{a_3 x^3} \\ \boxed{b_3 x^3} & + & \boxed{b_2 x^2} & + & \boxed{b_1 x} & + & \boxed{b_0} & & & & & & \\ & & & & & & \boxed{a_0 b_0} & & & & & & \end{array}$$

Convolution

$$
\begin{array}{ccccccccccccc}
 & & & & \boxed{a_0} & + & \boxed{a_1 x} & + & \boxed{a_2 x^2} & + & \boxed{a_3 x^3} \\
\boxed{b_3 x^3} & + & \boxed{b_2 x^2} & + & \boxed{b_1 x} & + & \boxed{b_0} & & & & \\
 & & & & \boxed{a_0 b_1 x} & + & \boxed{a_1 b_0 x} & & & &
\end{array}
$$

Convolution

$$\boxed{a_0} + \boxed{a_1 x} + \boxed{a_2 x^2} + \boxed{a_3 x^3}$$

$$\boxed{b_3 x^3} + \boxed{b_2 x^2} + \boxed{b_1 x} + \boxed{b_0}$$

$$\boxed{a_0 b_2 x^2} + \boxed{a_1 b_1 x^2} + \boxed{a_2 b_0 x^2}$$

Convolution

a_0	$+$	$a_1 x$	$+$	$a_2 x^2$	$+$	$a_3 x^3$
$b_3 x^3$	$+$	$b_2 x^2$	$+$	$b_1 x$	$+$	b_0
$a_0 b_3 x^3$	$+$	$a_1 b_2 x^3$	$+$	$a_2 b_1 x^3$	$+$	$a_3 b_0 x^3$

Convolution

$$\boxed{a_0} + \boxed{a_1 x} + \boxed{a_2 x^2} + \boxed{a_3 x^3}$$

$$\boxed{b_3 x^3} + \boxed{b_2 x^2} + \boxed{b_1 x} + \boxed{b_0}$$

$$\boxed{a_1 b_3 x^4} + \boxed{a_2 b_2 x^4} + \boxed{a_3 b_1 x^4}$$

Convolution

$$\begin{array}{ccccccccccc} \boxed{a_0} & + & \boxed{a_1 x} & + & \boxed{a_2 x^2} & + & \boxed{a_3 x^3} & & & & \\ & & & & \boxed{b_3 x^3} & + & \boxed{b_2 x^2} & + & \boxed{b_1 x} & + & \boxed{b_0} \\ & & & & \boxed{a_2 b_3 x^5} & + & \boxed{a_3 b_2 x^5} & & & & \end{array}$$

Convolution

a_0	$+$	$a_1 x$	$+$	$a_2 x^2$	$+$	$a_3 x^3$						
						$b_3 x^3$	$+$	$b_2 x^2$	$+$	$b_1 x$	$+$	b_0
						$\boxed{a_3 b_3 x^6}$						

Coefficient Representation of Polynomials

Multiplying Two Polynomials:

We can use Karatsuba's algorithm (assume n to be a power of 2):

$$A(x) = \sum_{j=0}^{n-1} a_j x^j = \sum_{j=0}^{\frac{n}{2}-1} a_j x^j + x^{\frac{n}{2}} \sum_{j=0}^{\frac{n}{2}-1} a_{\frac{n}{2}+j} x^j = A_1(x) + x^{\frac{n}{2}} A_2(x)$$

$$B(x) = \sum_{j=0}^{n-1} b_j x^j = \sum_{j=0}^{\frac{n}{2}-1} b_j x^j + x^{\frac{n}{2}} \sum_{j=0}^{\frac{n}{2}-1} b_{\frac{n}{2}+j} x^j = B_1(x) + x^{\frac{n}{2}} B_2(x)$$

Then $C(x) = A(x)B(x)$

$$= A_1(x)B_1(x) + x^{\frac{n}{2}}[A_1(x)B_2(x) + A_2(x)B_1(x)] + x^n\, A_2(x)B_2(x)$$

But $A_1(x)B_2(x) + A_2(x)B_1(x)$

$$= [A_1(x) + A_2(x)][B_1(x) + B_2(x)] - A_1(x)B_1(x) - A_2(x)B_2(x)$$

3 recursive multiplications of polynomials of degree bound $\frac{n}{2}$.

Similar recurrence as in Karatsuba's integer multiplication algorithm leading to a complexity of $\mathrm{O}\left(n^{\log_2 3}\right) = \mathrm{O}(n^{1.59})$.

Point-Value Representation of Polynomials

A point-value representation of a polynomial $A(x)$ is a set of n point-value pairs $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ such that all x_k are distinct and $y_k = A(x_k)$ for $0 \le k \le n-1$.

A polynomial has many point-value representations.

Adding Two Polynomials:

Suppose we have point-value representations of two polynomials of degree bound n using the same set of n points.

$$A: \{(x_0, y_0^a), (x_1, y_1^a), \dots, (x_{n-1}, y_{n-1}^a)\}$$

$$B: \{(x_0, y_0^b), (x_1, y_1^b), \dots, (x_{n-1}, y_{n-1}^b)\}$$

If $C(x) = A(x) + B(x)$ then

$$C: \{(x_0, y_0^a + y_0^b), (x_1, y_1^a + y_1^b), \dots, (x_{n-1}, y_{n-1}^a + y_{n-1}^b)\}$$

Thus polynomial addition takes $\Theta(n)$ time.

Point-Value Representation of Polynomials

Multiplying Two Polynomials:

Suppose we have *extended* (why?) point-value representations of two polynomials of degree bound n using the same set of $2n$ points.

$$A: \{(x_0, y_0^a), (x_1, y_1^a), \dots, (x_{2n-1}, y_{2n-1}^a)\}$$

$$B: \{(x_0, y_0^b), (x_1, y_1^b), \dots, (x_{2n-1}, y_{2n-1}^b)\}$$

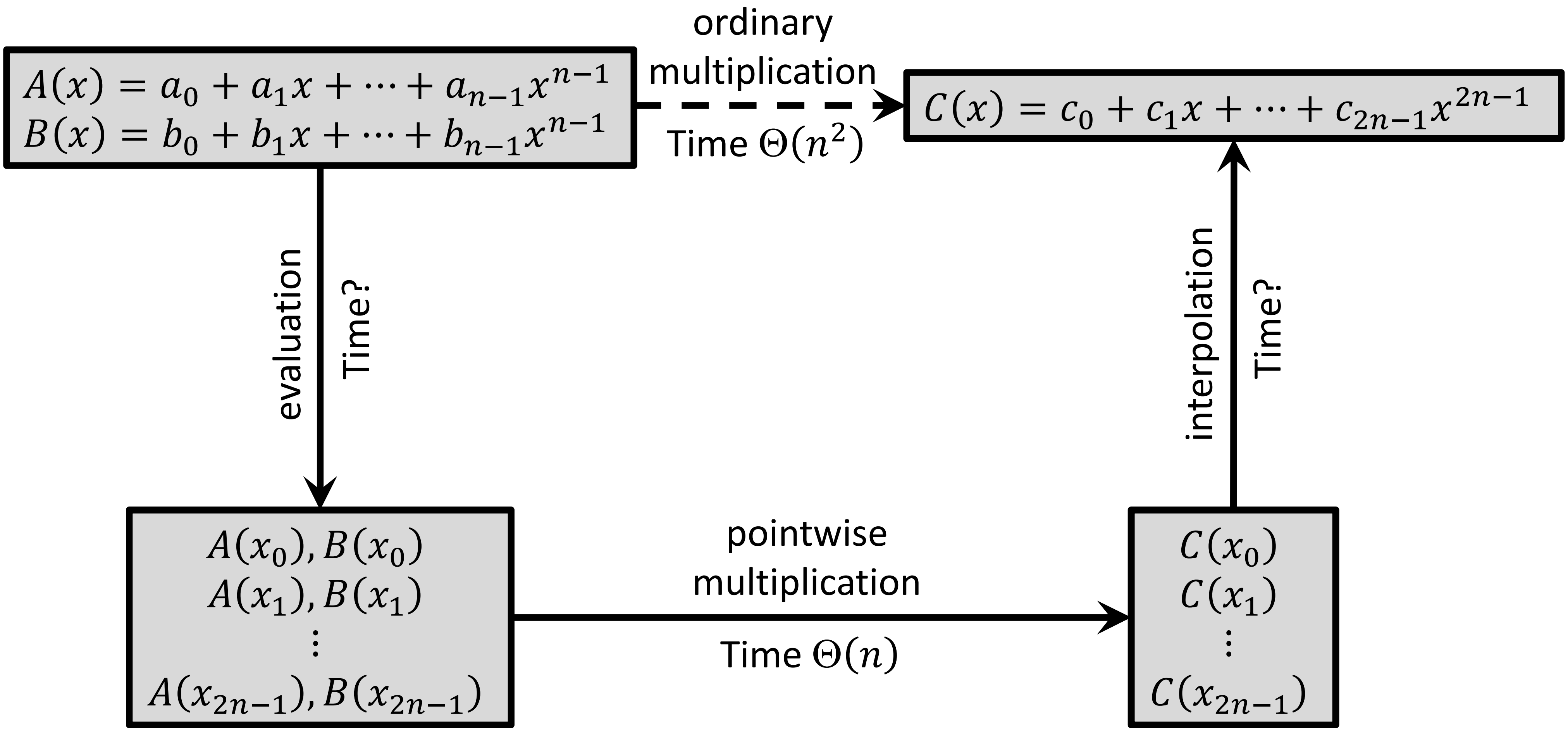
If $C(x) = A(x)B(x)$ then

$$C: \{(x_0, y_0^a y_0^b), (x_1, y_1^a y_1^b), \dots, (x_{2n-1}, y_{2n-1}^a y_{2n-1}^b)\}$$

Thus polynomial multiplication also takes only $\Theta(n)$ time!

(compare this with the $\Theta(n^2)$ time needed in the coefficient form)

Faster Polynomial Multiplication?
(in Coefficient Form)



Faster Polynomial Multiplication? (in Coefficient Form)

Coefficient Representation $\Rightarrow$ Point-Value Representation:

We select any set of n distinct points $\{x_0, x_1, \dots, x_{n-1}\}$, and evaluate $A(x_k)$ for $0 \leq k \leq n-1$.

Using Horner's rule this approach takes $\Theta(n^2)$ time.

Point-Value Representation $\Rightarrow$ Coefficient Representation:

We can interpolate using Lagrange's formula:

$$A(x) = \sum_{k=0}^{n-1} \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)} y_k$$

This again takes $\Theta(n^2)$ time.

In both cases we need to do much better!

Coefficient Form $\Rightarrow$ Point-Value Form

A polynomial of degree bound n: $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$

A set of n distinct points: $\{x_0, x_1, \dots, x_{n-1}\}$

Compute point-value form: $\{(x_0, A(x_0)), (x_1, A(x_1)), \dots, (x_{n-1}, A(x_{n-1}))\}$

Using matrix notation:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \cdot \\ \cdot \\ \cdot \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & (x_0)^2 & \cdots & (x_0)^{n-1} \\ 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & x_{n-1} & (x_{n-1})^2 & \cdots & (x_{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}$$

We want to choose the set of points in a way that simplifies the multiplication.

In the rest of the lecture on this topic we will assume:

n is a power of 2.

Coefficient Form $\Rightarrow$ Point-Value Form

Let's choose $x_{n/2+j} = -x_j$ for $0 \le j \le n/2 - 1$. Then

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \cdot \\ A\left(x_{n/2-1}\right) \\ \hdashline A\left(x_{n/2+0}\right) \\ A\left(x_{n/2+1}\right) \\ \cdot \\ A\left(x_{n/2+(n/2-1)}\right) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & (x_0)^2 & \cdots & (x_0)^{n-1} \\ 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & x_{n/2-1} & \left(x_{n/2-1}\right)^2 & \cdots & \left(x_{n/2-1}\right)^{n-1} \\ \hdashline 1 & -x_0 & (-x_0)^2 & \cdots & (-x_0)^{n-1} \\ 1 & -x_1 & (-x_1)^2 & \cdots & (-x_1)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & -x_{n/2-1} & \left(-x_{n/2-1}\right)^2 & \cdots & \left(-x_{n/2-1}\right)^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}$$

Observe that for $0 \le j \le n/2 - 1$: $\left(x_{n/2+j}\right)^k = \begin{cases} \left(x_j\right)^k, & if\ k = even, \\ -\left(x_j\right)^k, & if\ k = odd. \end{cases}$

Thus we have just split the original $n \times n$ matrix into two almost similar $\frac{n}{2} \times n$ matrices!

Coefficient Form $\Rightarrow$ Point-Value Form

How and how much do we save?

$$A(x) = \sum_{l=0}^{n-1} a_l x^l = \sum_{l=0}^{n/2-1} a_{2l} x^{2l} + \sum_{l=0}^{n/2-1} a_{2l+1} x^{2l+1}$$

$$= \sum_{l=0}^{n/2-1} a_{2l} (x^2)^l + x \sum_{l=0}^{n/2-1} a_{2l+1} (x^2)^l = A_{even}(x^2) + x A_{odd}(x^2),$$

where, $A_{even}(x) = \sum_{l=0}^{n/2-1} a_{2l} x^l$ and $A_{odd}(x) = \sum_{l=0}^{n/2-1} a_{2l+1} x^l$.

Observe that for $0 \leq j \leq n/2 - 1$:

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

$$A(x_{n/2+j}) = A(-x_j) = A_{even}(x_j^2) - x_j A_{odd}(x_j^2)$$

So in order to evaluate $A(x_j)$ for all $0 \leq j \leq n - 1$, we need:

$n/2$ evaluations of A_{even} and $n/2$ evaluations of A_{odd}
n multiplications
$n/2$ additions and $n/2$ subtractions

Thus we save about half the computation!

Coefficient Form $\Rightarrow$ Point-Value Form

If we can recursively evaluate A_{even} and A_{odd} using the same approach, we get the following recurrence relation for the running time of the algorithm:

$$T(n) = \begin{cases} \Theta(1), & if\ n = 1, \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & otherwise. \end{cases}$$

$$= \Theta(n \log n)$$

Our trick was to evaluate A at x (positive) and $-x$ (negative).

But inputs to A_{even} and A_{odd} are always of the form x^2 (positive)!

How can we apply the same trick?

Coefficient Form $\Rightarrow$ Point-Value Form

Let us consider the evaluation of $A_{even}(x_j)$ for $0 \le j \le n/2 - 1$:

$$\begin{bmatrix} A_{even}(x_0) \\ A_{even}(x_1) \\ \cdot \\ \cdot \\ \cdot \\ A_{even}(x_{n/2-1}) \end{bmatrix} = \begin{bmatrix} 1 & (x_0)^2 & (x_0)^4 & \cdots & (x_0)^{n-2} \\ 1 & (x_1)^2 & (x_1)^4 & \cdots & (x_1)^{n-2} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & (x_{n/2-1})^2 & (x_{n/2-1})^4 & \cdots & (x_{n/2-1})^{n-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ \cdot \\ \cdot \\ a_{n-2} \end{bmatrix}$$

In order to apply the same trick on A_{even} we must set:

$$(x_{n/4+j})^2 = -(x_j)^2 \text{ for } 0 \le j \le n/4 - 1$$

Coefficient Form $\Rightarrow$ Point-Value Form

In A_{even} we set: $x_{n/4+j}^2 = -x_j^2$ for $0 \leq j \leq n/4 - 1$. Then

$$\begin{bmatrix} A_{even}(x_0) \\ A_{even}(x_1) \\ \cdot \\ A_{even}\left(x_{n/4-1}\right) \\ \hline A_{even}\left(x_{n/4+0}\right) \\ A_{even}\left(x_{n/4+1}\right) \\ \cdot \\ A_{even}\left(x_{n/4+(n/4-1)}\right) \end{bmatrix} = \begin{bmatrix} 1 & x_0^2 & (x_0^2)^2 & \cdots & (x_0^2)^{\frac{n}{2}-1} \\ 1 & x_1^2 & (x_1^2)^2 & \cdots & (x_1^2)^{\frac{n}{2}-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & x_{n/4-1}^2 & \left(x_{n/4-1}^2\right)^2 & \cdots & \left(x_{n/4-1}^2\right)^{\frac{n}{2}-1} \\ \hline 1 & -x_0^2 & (-x_0^2)^2 & \cdots & (-x_0^2)^{\frac{n}{2}-1} \\ 1 & -x_1^2 & (-x_1^2)^2 & \cdots & (-x_1^2)^{\frac{n}{2}-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & -x_{n/4-1}^2 & \left(-x_{n/2-1}^2\right)^2 & \cdots & \left(-x_{n/4-1}^2\right)^{\frac{n}{2}-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ a_{n-2} \end{bmatrix}$$

This means setting $x_{n/4+j} = ix_j$, where $i = \sqrt{-1}$ (imaginary)!

This also allows us to apply the same trick on A_{odd}.

Coefficient Form $\Rightarrow$ Point-Value Form

We can apply the trick once if we set:

$$x_{n/2+j} = -x_j \text{ for } 0 \leq j \leq n/2 - 1$$

We can apply the trick (recursively) 2 times if we also set:

$$\left(x_{n/2^2+j}\right)^2 = -\left(x_j\right)^2 \text{ for } 0 \leq j \leq n/2^2 - 1$$

We can apply the trick (recursively) 3 times if we also set:

$$\left(x_{n/2^3+j}\right)^{2^2} = -\left(x_j\right)^{2^2} \text{ for } 0 \leq j \leq n/2^3 - 1$$

We can apply the trick (recursively) k times if we also set:

$$\left(x_{n/2^k+j}\right)^{2^{k-1}} = -\left(x_j\right)^{2^{k-1}} \text{ for } 0 \leq j \leq n/2^k - 1$$

Coefficient Form $\Rightarrow$ Point-Value Form

Consider the t^{th} *primitive root of unity*:

$$\omega_t = e^{\frac{2\pi i}{t}} = \cos\frac{2\pi}{t} + i \cdot \sin\frac{2\pi}{t} \qquad \left(i = \sqrt{-1}\right)$$

Then

$$x_{n/2+j} = -x_j \;\Rightarrow\; x_{n/2^1+j} = \omega_{2^1} \cdot x_j$$

$$\left(x_{n/2^2+j}\right)^2 = -\left(x_j\right)^2 \Rightarrow\; x_{n/2^2+j} = \omega_{2^2} \cdot x_j$$

$$\left(x_{n/2^3+j}\right)^{2^2} = -\left(x_j\right)^{2^2} \Rightarrow\; x_{n/2^3+j} = \omega_{2^3} \cdot x_j$$

$$\vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots$$

$$\left(x_{n/2^k+j}\right)^{2^{k-1}} = -\left(x_j\right)^{2^{k-1}} \Rightarrow\; x_{n/2^k+j} = \omega_{2^k} \cdot x_j$$

Coefficient Form $\Rightarrow$ Point-Value Form

If $n = 2^k$ we would like to apply the trick k times recursively.

What values should we choose for $\{x_0, x_1, \dots, x_{n-1}\}$?

Example: For $n = 2^3$ we need to choose $\{x_0, x_1, \dots, x_7\}$.

Choose: $x_0 = 1 \qquad = \omega_8^0$

$k = 3$: $x_1 = \omega_{2^3} \cdot x_0 \qquad = \omega_8^1$

$k = 2$: $x_2 = \omega_{2^2} \cdot x_0 \qquad = \omega_8^2$

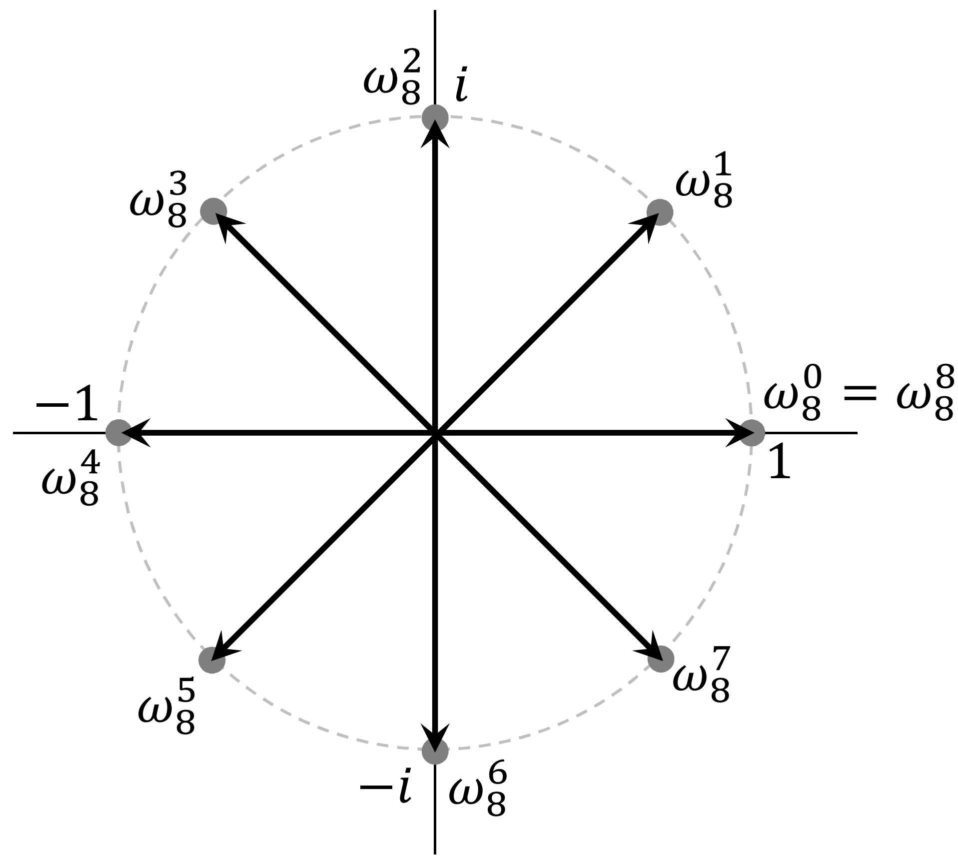
$x_3 = \omega_{2^2} \cdot x_1 \qquad = \omega_8^3$

$k = 1$: $x_4 = \omega_{2^1} \cdot x_0 \qquad = \omega_8^4$

$x_5 = \omega_{2^1} \cdot x_1 \qquad = \omega_8^5$

$x_6 = \omega_{2^1} \cdot x_2 \qquad = \omega_8^6$

$x_7 = \omega_{2^1} \cdot x_3 \qquad = \omega_8^7$



complex 8^{th} roots of unity

Coefficient Form $\Rightarrow$ Point-Value Form

For a polynomial of degree bound $n = 2^k$, we need to apply the trick recursively at most $\log n = k$ times.

We choose $x_0 = 1 = \omega_n^0$ and set $x_j = \omega_n^j$ for $1 \le j \le n-1$.

Then we compute the following product:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} A(1) \\ A(\omega_n) \\ A(\omega_n^2) \\ \cdot \\ \cdot \\ A(\omega_n^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & (\omega_n)^2 & \cdots & (\omega_n)^{n-1} \\ 1 & \omega_n^2 & (\omega_n^2)^2 & \cdots & (\omega_n^2)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & \omega_n^{n-1} & (\omega_n^{n-1})^2 & \cdots & (\omega_n^{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}$$

The vector $y = (y_0, y_1, \cdots, y_{n-1})$ is called the *discrete Fourier transform* (DFT) of $(a_0, a_1, \cdots, a_{n-1})$.

This method of computing DFT is called the *fast Fourier transform* (FFT) method.

Coefficient Form $\Rightarrow$ Point-Value Form

Rec-FFT (($a_0, a_1, \ldots, a_{n-1}$)) { $n = 2^k$ *for integer* $k \geq 0$ }

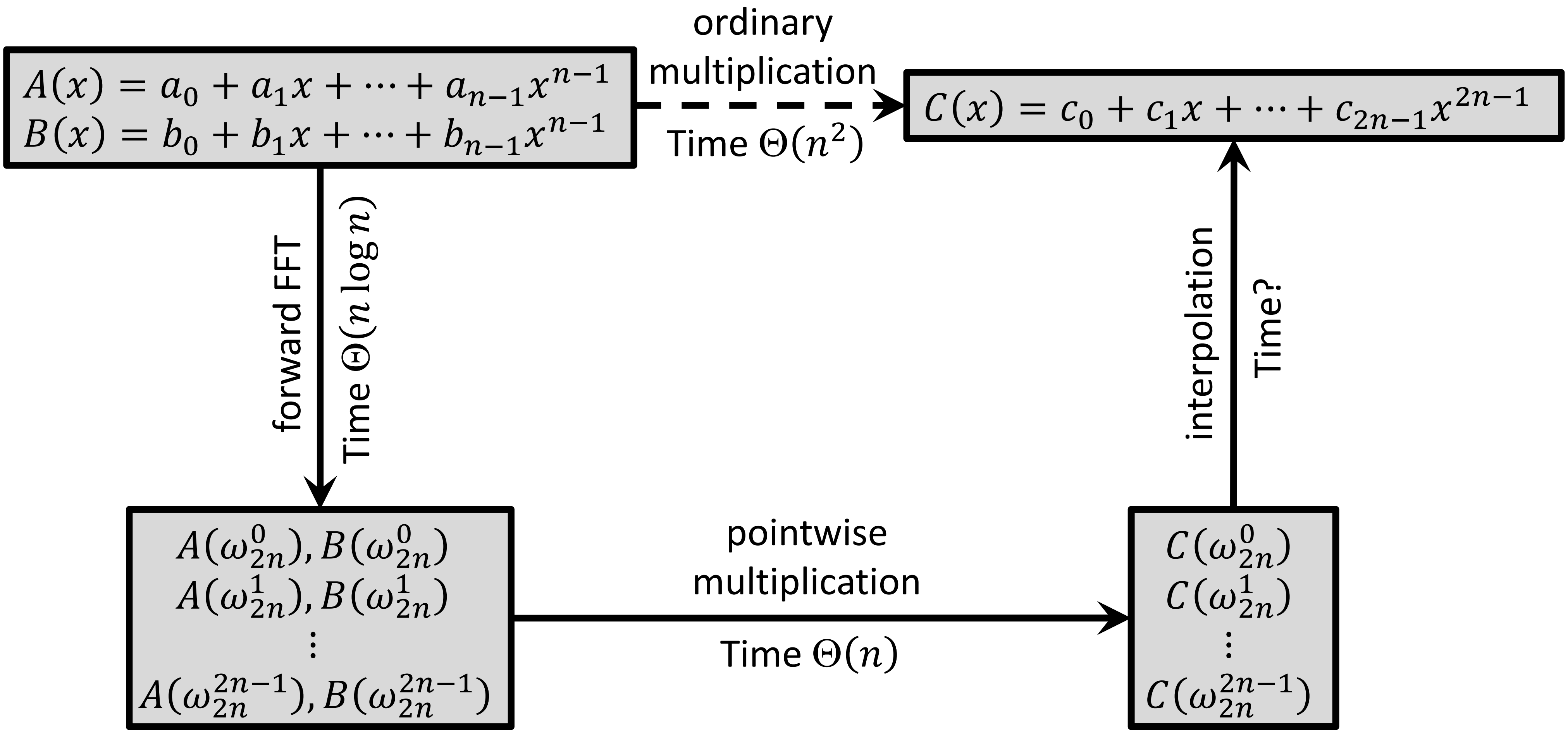
1. *if* $n = 1$ *then*
2. *return* (a_0)
3. $\omega_n \leftarrow e^{2\pi i/n}$
4. $\omega \leftarrow 1$
5. $y^{even} \leftarrow$ *Rec-FFT* (($a_0, a_2, \ldots, a_{n-2}$))
6. $y^{odd} \leftarrow$ *Rec-FFT* (($a_1, a_3, \ldots, a_{n-1}$))
7. *for* $j \leftarrow 0$ *to* $n/2 - 1$ *do*
8. $y_j \leftarrow y_j^{even} + \omega\, y_j^{odd}$
9. $y_{n/2+j} \leftarrow y_j^{even} - \omega\, y_j^{odd}$
10. $\omega \leftarrow \omega\, \omega_n$
11. *return* y

Running time:

$$T(n) = \begin{cases} \Theta(1), & if\ n = 1, \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & otherwise. \end{cases}$$

$$= \Theta(n \log n)$$

Faster Polynomial Multiplication? (in Coefficient Form)



Next Lecture will Cover *Interpolation*