CSE 548: Analysis of Algorithms

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

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2019
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 \( a = (a_0, a_1, \ldots, a_{n-1}) \) 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:

\( \Theta(n) \) 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 $0 \leq j \leq n - 1$. 
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 $0 \leq j \leq 2n - 2$.

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

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

\[ a_0 + a_1 x + a_2 x^2 + a_3 x^3 \]

\[ b_3 x^3 + b_2 x^2 + b_1 x + b_0 \]

\[ a_0 b_0 \]
Convolution

\[ b_3 x^3 + b_2 x^2 + b_1 x + b_0 + a_0 + a_1 x + a_2 x^2 + a_3 x^3 \]
Convolution

\[ a_0 + a_1 x + a_2 x^2 + a_3 x^3 \]

\[ b_3 x^3 + b_2 x^2 + b_1 x + b_0 \]

\[ a_0 b_2 x^2 + a_1 b_1 x^2 + a_2 b_0 x^2 \]
Convolution

\[
\begin{align*}
& a_0 + a_1 x + a_2 x^2 + a_3 x^3 \\
& b_3 x^3 + b_2 x^2 + b_1 x + b_0 \\
& a_0 b_3 x^3 + a_1 b_2 x^3 + a_2 b_1 x^3 + a_3 b_0 x^3
\end{align*}
\]
Convolution

\[ a_0 + a_1 x + a_2 x^2 + a_3 x^3 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 + a_1 b_3 x^4 + a_2 b_2 x^4 + a_3 b_1 x^4 \]
Convolution

\[a_0 + a_1 x + a_2 x^2 + a_3 x^3 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 + a_2 b_3 x^5 + a_3 b_2 x^5\]
Convolution

\[ a_0 + a_1 x + a_2 x^2 + a_3 x^3 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 + a_3 b_3 x^6 \]
Coefficient Representation of Polynomials

Multiplying Two Polynomials:

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

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

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

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 \( O(n^{\log_2 3}) = O(n^{1.59}) \).
Point-Value Representation of Polynomials

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

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.

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

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

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

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

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.

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

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

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

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

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 )

\[ A(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} \]
\[ B(x) = b_0 + b_1x + \cdots + b_{n-1}x^{n-1} \]
\[ C(x) = c_0 + c_1x + \cdots + c_{2n-1}x^{2n-1} \]

ordinary multiplication  
Time \( \Theta(n^2) \)

evaluation

Time?

\[ A(x_0), B(x_0) \]
\[ A(x_1), B(x_1) \]
\[ \vdots \]
\[ A(x_{2n-1}), B(x_{2n-1}) \]

pointwise multiplication

Time \( \Theta(n) \)

interpolation

Time?

\[ C(x_0) \]
\[ C(x_1) \]
\[ \vdots \]
\[ C(x_{2n-1}) \]
Faster Polynomial Multiplication? (in Coefficient Form)

Coefficient Representation ⇒ Point-Value Representation:

We select any set of \( n \) distinct points \( \{x_0, x_1, \ldots, x_{n-1}\} \), 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 ⇒ 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!
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: \( \{x_0, x_1, \ldots, x_{n-1}\} \)

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

Using matrix notation:

\[
\begin{bmatrix}
A(x_0) \\
A(x_1) \\
\vdots \\
A(x_{n-1})
\end{bmatrix} =
\begin{bmatrix}
1 & x_0 & (x_0)^2 & \cdots & (x_0)^{n-1} \\
1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{n-1} & (x_{n-1})^2 & \cdots & (x_{n-1})^{n-1}
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
\vdots \\
a_{n-1}
\end{bmatrix}
\]

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 $0 \leq j \leq n/2 - 1$. Then

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

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

Thus we have just split the original $n \times n$ matrix into two almost similar $\frac{n}{2} \times n$ matrices!
How and how much do we save?

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

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

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

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

\[
A(x_j) = A_{\text{even}}(x_j^2) + x_j A_{\text{odd}}(x_j^2)
\]

\[
A(x_{n/2+j}) = A(-x_j) = A_{\text{even}}(x_j^2) - x_j A_{\text{odd}}(x_j^2)
\]

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

- \( n/2 \) evaluations of \( A_{\text{even}} \) and \( n/2 \) evaluations of \( A_{\text{odd}} \)
- \( n \) multiplications
- \( n/2 \) additions and \( n/2 \) subtractions

Thus we save about half the computation!
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:

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

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

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 ⇒ Point-Value Form

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

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

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

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

$$
\begin{bmatrix}
A_{even}(x_0) \\
A_{even}(x_1) \\
\vdots \\
A_{even}(x_{n/4-1}) \\
A_{even}(x_{n/4+0}) \\
A_{even}(x_{n/4+1}) \\
\vdots \\
A_{even}(x_{n/4+(n/4-1)})
\end{bmatrix}
\begin{bmatrix}
1 \\
x_0^2 \\
x_1^2 \\
\vdots \\
x_{n/4-1}^2 \\
-x_0^2 \\
-x_1^2 \\
\vdots \\
-x_{n/4-1}^2
\end{bmatrix}
=
\begin{bmatrix}
1 \\
(x_0^2)^2 \\
(x_1^2)^2 \\
\vdots \\
(x_{n/4-1}^2)^2 \\
(-x_0^2)^2 \\
(-x_1^2)^2 \\
\vdots \\
(-x_{n/4-1}^2)^2
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_2 \\
a_4 \\
\vdots \\
a_{n-2}
\end{bmatrix}
$$

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 ⇒ 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:

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

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

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

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

\[ \left(x_{n/2^k+j}\right)^{2^{k-1}} = -(x_j)^{2^{k-1}} \text{ for } 0 \leq j \leq n/2^k - 1 \]
Consider the $t^{th}$ primitive root of unity:

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

Then

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

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

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

$$\vdots \quad \vdots \quad \vdots$$

$$\left(x_{n/2^k+j}\right)^{2^{k-1}} = -(x_j)^{2^{k-1}} \implies x_{n/2^k+j} = \omega_{2^k} \cdot x_j$$
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 $\{x_0, x_1, \ldots, x_{n-1}\}$?

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

Choose: $x_0 = 1$

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

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

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

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

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

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

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

complex $8^{th}$ roots of unity
**Coefficient Form ⇒ 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 $1 \leq j \leq n - 1$.

Then we compute the following product:

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

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 ⇒ Point-Value Form**

**Example:** For $n = 2^3 = 8$:

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7$$

We need to evaluate $A(x)$ at $x = \omega_8^i$ for $0 \leq i < 8$.

![Complex 8th roots of unity](image)

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

where $A_{even}(y) = a_0 + a_2 y + a_4 y^2 + a_6 y^3$

and $A_{odd}(y) = a_1 + a_3 y + a_5 y^2 + a_7 y^3$
**Coefficient Form ⇒ Point-Value Form**

Observe that:

\[ \omega_8^0 = \omega_8^8 = \omega_4^0 \]
\[ \omega_8^2 = \omega_8^{10} = \omega_4^1 \]
\[ \omega_8^4 = \omega_8^{12} = \omega_4^2 \]
\[ \omega_8^6 = \omega_8^{14} = \omega_4^3 \]

Also:

\[ \omega_8^4 = -\omega_8^0 \]
\[ \omega_8^5 = -\omega_8^1 \]
\[ \omega_8^6 = -\omega_8^2 \]
\[ \omega_8^7 = -\omega_8^3 \]

\[ A(\omega_8^0) = A_{even}(\omega_8^0) + \omega_8^0 \cdot A_{odd}(\omega_8^0) \]
\[ A(\omega_8^1) = A_{even}(\omega_8^2) + \omega_8^1 \cdot A_{odd}(\omega_8^2) \]
\[ A(\omega_8^2) = A_{even}(\omega_8^4) + \omega_8^2 \cdot A_{odd}(\omega_8^4) \]
\[ A(\omega_8^3) = A_{even}(\omega_8^6) + \omega_8^3 \cdot A_{odd}(\omega_8^6) \]
\[ A(\omega_8^4) = A_{even}(\omega_8^8) + \omega_8^4 \cdot A_{odd}(\omega_8^8) \]
\[ A(\omega_8^5) = A_{even}(\omega_8^{10}) + \omega_8^5 \cdot A_{odd}(\omega_8^{10}) \]
\[ A(\omega_8^6) = A_{even}(\omega_8^{12}) + \omega_8^6 \cdot A_{odd}(\omega_8^{12}) \]
\[ A(\omega_8^7) = A_{even}(\omega_8^{14}) + \omega_8^7 \cdot A_{odd}(\omega_8^{14}) \]
Coefficient Form $\Rightarrow$ Point-Value Form

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

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

Running time:

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

$$= \Theta(n \log n)$$
Faster Polynomial Multiplication?  
(in Coefficient Form)

\[ A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \]
\[ B(x) = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1} \]
\[ C(x) = c_0 + c_1 x + \cdots + c_{2n-1} x^{2n-1} \]

ordinary multiplication  
Time \( \Theta(n^2) \)

\[ A(\omega_0^{2n}), B(\omega_0^{2n}) \]
\[ A(\omega_1^{2n}), B(\omega_1^{2n}) \]
\[ \vdots \]
\[ A(\omega_{2n-1}^{2n}), B(\omega_{2n-1}^{2n}) \]

forward FFT  
Time \( \Theta(n \log n) \)

pointwise multiplication  
Time \( \Theta(n) \)

\[ C(\omega_0^{2n}) \]
\[ C(\omega_1^{2n}) \]
\[ \vdots \]
\[ C(\omega_{2n-1}^{2n}) \]

interpolation  
Time?
Point-Value Form $\Rightarrow$ Coefficient Form

Given:

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

$V(\omega_n)$

Vandermonde Matrix

$$
\Rightarrow V(\omega_n) \cdot \bar{a} = \bar{y}
$$

We want to solve: $\bar{a} = [V(\omega_n)]^{-1} \cdot \bar{y}$

It turns out that: $[V(\omega_n)]^{-1} = \frac{1}{n} V \left( \frac{1}{\omega_n} \right)$

That means $[V(\omega_n)]^{-1}$ looks almost similar to $V(\omega_n)$!
Point-Value Form $\Rightarrow$ Coefficient Form

Show that:  $[V(\omega_n)]^{-1} = \frac{1}{n} V \left( \frac{1}{\omega_n} \right)$

Let $U(\omega_n) = \frac{1}{n} V \left( \frac{1}{\omega_n} \right)$

We want to show that $U(\omega_n)V(\omega_n) = I_n$, where $I_n$ is the $n \times n$ identity matrix.

Observe that for $0 \leq j, k \leq n - 1$, the $(j, k)^{th}$ entries are:

$[V(\omega_n)]_{jk} = \omega_n^{jk}$ and $[U(\omega_n)]_{jk} = \frac{1}{n} \omega_n^{-jk}$

Then entry $(p, q)$ of $U(\omega_n)V(\omega_n)$,

$[U(\omega_n)V(\omega_n)]_{pq} = \sum_{k=0}^{n-1} [U(\omega_n)]_{pk} [V(\omega_n)]_{kq} = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{k(q-p)}$
**Point-Value Form \Rightarrow Coefficient Form**

\[ [U(\omega_n)V(\omega_n)]_{pq} = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{k(q-p)} \]

**CASE** \( p = q \):

\[ [U(\omega_n)V(\omega_n)]_{pq} = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^0 = \frac{1}{n} \sum_{k=0}^{n-1} 1 = \frac{1}{n} \times n = 1 \]

**CASE** \( p \neq q \):

\[ [U(\omega_n)V(\omega_n)]_{pq} = \frac{1}{n} \sum_{k=0}^{n-1} (\omega_n^{q-p})^k = \frac{1}{n} \times \frac{(\omega_n^{q-p})^n - 1}{\omega_n^{q-p} - 1} \]

\[ = \frac{1}{n} \times \frac{(\omega_n^{n})^{q-p} - 1}{\omega_n^{q-p} - 1} = \frac{1}{n} \times \frac{1^{q-p} - 1}{\omega_n^{q-p} - 1} = 0 \]

Hence \( U(\omega_n)V(\omega_n) = I_n \)
We need to compute the following matrix-vector product:

\[
\begin{bmatrix}
  a_0 \\
  a_1 \\
  a_2 \\
  \vdots \\
  a_{n-1}
\end{bmatrix} = \frac{1}{n} \times \begin{bmatrix}
  1 & 1 & 1 & \cdots & 1 \\
  1 & \frac{1}{\omega_n} & \left(\frac{1}{\omega_n}\right)^2 & \cdots & \left(\frac{1}{\omega_n}\right)^{n-1} \\
  1 & \frac{1}{\omega_n^2} & \left(\frac{1}{\omega_n^2}\right)^2 & \cdots & \left(\frac{1}{\omega_n^2}\right)^{n-1} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  1 & \frac{1}{\omega_n^{n-1}} & \left(\frac{1}{\omega_n^{n-1}}\right)^2 & \cdots & \left(\frac{1}{\omega_n^{n-1}}\right)^{n-1}
\end{bmatrix} \begin{bmatrix}
  y_0 \\
  y_1 \\
  y_2 \\
  \vdots \\
  y_{n-1}
\end{bmatrix}
\]

This inverse problem is almost similar to the forward problem, and can be solved in \(\Theta(n \log n)\) time using the same algorithm as the forward FFT with only minor modifications!
Faster Polynomial Multiplication? (in Coefficient Form)

Two polynomials of degree bound $n$ given in the coefficient form can be multiplied in $\Theta(n \log n)$ time!
Some Applications of Fourier Transform and FFT

• Signal processing
• Image processing
• Noise reduction
• Data compression
• Solving partial differential equation
• Multiplication of large integers
• Polynomial multiplication
• Molecular docking
Some Applications of Fourier Transform and FFT

Any periodic signal can be represented as a sum of a series of sinusoidal (sine & cosine) waves. [1807]
Spatial (Time) Domain $\Leftrightarrow$ Frequency Domain

Source: The Scientist and Engineer’s Guide to Digital Signal Processing by Steven W. Smith
Spatial (Time) Domain $\Leftrightarrow$ Frequency Domain

Function $s(x)$ (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, $S(f)$ (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Spatial (Time) Domain $\Leftrightarrow$ Frequency Domain

Function $s(x)$ (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, $S(f)$ (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Function $s(x)$ (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, $S(f)$ (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Spatial (Time) Domain $\iff$ Frequency Domain

Function $s(x)$ (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, $S(f)$ (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Function $s(x)$ (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, $S(f)$ (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Let $s(t)$ be a signal specified in the time domain.

The strength of $s(t)$ at frequency $f$ is given by:

$$S(f) = \int_{-\infty}^{\infty} s(t) \cdot e^{-2\pi i f t} \, dt$$

Evaluating this integral for all values of $f$ gives the frequency domain function.

Now $s(t)$ can be retrieved by summing up the signal strengths at all possible frequencies:

$$s(t) = \int_{-\infty}^{\infty} S(f) \cdot e^{2\pi i f t} \, df$$
Why do the Transforms Work?

Let’s try to get a little intuition behind why the transforms work. We will look at a very simple example.

Suppose: \( s(t) = \cos(2\pi h \cdot t) \)

\[
\frac{1}{T} \int_{-T}^{T} s(t) \cdot e^{-2\pi ift} \, dt = \begin{cases} 
1 + \frac{\sin(4\pi fT)}{4\pi fT}, & \text{if } f = h, \\
\frac{\sin(2\pi (h-f)T)}{2\pi (h-f)T} + \frac{\sin(2\pi (h+f)T)}{2\pi (h+f)T}, & \text{otherwise}.
\end{cases}
\]

\[
\Rightarrow \lim_{T \to \infty} \left( \frac{1}{T} \int_{-T}^{T} s(t) \cdot e^{-2\pi ift} \, dt \right) = \begin{cases} 
1, & \text{if } f = h, \\
0, & \text{otherwise}.
\end{cases}
\]

So, the transform can detect if \( f = h \)!
Noise Reduction

Data Compression

- Discrete Cosine Transforms (DCT) are used for lossy data compression (e.g., MP3, JPEG, MPEG)

- DCT is a Fourier-related transform similar to DFT (Discrete Fourier Transform) but uses only real data (uses cosine waves only instead of both cosine and sine waves)

- Forward DCT transforms data from spatial to frequency domain

- Each frequency component is represented using a fewer number of bits (i.e., truncated/quantized)

- Low amplitude high frequency components are also removed

- Inverse DCT then transforms the data back to spatial domain

- The resulting image compresses better
**Data Compression**

Transformation to frequency domain using cosine transforms work in the same way as the Fourier transform.

Suppose: \( s(t) = \cos(2\pi h \cdot t) \)

\[
\frac{1}{T} \int_{-T}^{T} s(t) \cdot \cos(2\pi f t) \, dt = \begin{cases} 
1 + \frac{\sin(4\pi f T)}{4\pi f T}, & \text{if } f = h, \\
\frac{\sin(2\pi (h - f) T)}{2\pi (h - f) T} + \frac{\sin(2\pi (h + f) T)}{2\pi (h + f) T}, & \text{otherwise.}
\end{cases}
\]

\[ \Rightarrow \lim_{T \to \infty} \left( \frac{1}{T} \int_{-T}^{T} s(t) \cdot \cos(2\pi f t) \, dt \right) = \begin{cases} 
1, & \text{if } f = h, \\
0, & \text{otherwise.}
\end{cases} \]

So, this transform can also detect if \( f = h \).
Protein-Protein Docking

Knowledge of complexes is used in
- Drug design
- Studying molecular assemblies
- Structure function analysis
- Protein interactions

Protein-Protein Docking: Given two proteins, find the best relative transformation and conformations to obtain a stable complex.

Docking is a hard problem
- Search space is huge (6D for rigid proteins)
- Protein flexibility adds to the difficulty
To maximize skin-skin overlaps and minimize core-core overlaps

- assign positive real weights to skin atoms
- assign positive imaginary weights to core atoms

Let \( A' \) denote molecule \( A \) with the pseudo skin atoms.

For \( P \in \{ A', B \} \) with \( M_P \) atoms, affinity function: 
\[
f_P(x) = \sum_{k=1}^{M_P} w_k \cdot g_k(x)
\]

Here \( g_k(x) \) is a Gaussian representation of atom \( k \), and \( w_k \) its weight.
Let $A'$ denote molecule $A$ with the pseudo skin atoms.

For $P \in \{A', B\}$ with $M_P$ atoms, affinity function:

$$f_P(x) = \sum_{k=1}^{M_P} w_k \cdot g_k(x)$$

For rotation $r$ and translation $t$ of molecule $B$ (i.e., $B_{t,r}$), the interaction score, $F_{A,B}(t, r) = \int_x f_{A'}(x) f_{B_{t,r}}(x) \, dx$
For rotation $r$ and translation $t$ of molecule $B$ (i.e., $B_{t,r}$),

the interaction score, $F_{A,B}(t, r) = \int_x f_A'(x)f_{B_{t,r}}(x) \, dx$

$Re\left( F_{A,B}(t, r) \right) = \text{skin-skin overlap score} - \text{core-core overlap score}$

$Im\left( F_{A,B}(t, r) \right) = \text{skin-core overlap score}$
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Docking: Rotational & Translational Search
Translational Search using FFT

∀z ∈ Ω = [−n, n]^3, h(z) = \int_{x \in \Omega} f_{A'}(x)f_{B_r}(z - x)dx