# CS 170 Section 2
# Fast Fourier Transform

Owen Jow | owenjow@berkeley.edu

# Agenda

- Logistics
- Fast fourier transform

# Logistics

# Logistics

- Homework 2 due next Monday (02/05)
- Midterm 1 in 13 days (< 2 weeks!)
  - right now, assume that everything up to the midterm (i.e. the first five chapters) are in-scope
  - from the calendar, topics include **D&Q**, **FFT**, **decompositions of graphs**, **paths in graphs**, and **greedy algorithms**
  - for free points, be able to do anything mechanical

# Fast Fourier Transform

# Background: Polynomial Multiplication

- **In this class**, we use the FFT in order to perform efficient polynomial multiplication.

HOW TO COMPUTE $C(x) = A(x) \cdot B(x)$

1. Pick n points, where $n \geq$ [the degree of $C(x)$] + 1.
2. Evaluate $A(x_k)$ at each of the n points.
3. Evaluate $B(x_k)$ at each of the n points.
4. Evaluate $C(x_k) = A(x_k) \cdot B(x_k)$ for each of the n points.
5. Convert our newfound value representation for $C(x_k)$ into a coefficient representation.

# What's Slow?

- Assuming a naive approach,

HOW TO COMPUTE $C(x) = A(x) \cdot B(x)$

1. Pick n points, where $n \geq$ [the degree of $C(x)$] + 1.
2. Evaluate $A(x_k)$ at each of the n points. $O(n^2)$
3. Evaluate $B(x_k)$ at each of the n points. $O(n^2)$
4. Evaluate $C(x_k) = A(x_k) \cdot B(x_k)$ for each of the n points. $O(n)$
5. Convert our newfound value representation for $C(x_k)$ into a coefficient representation. O(wtf)

Naively, polynomial multiplication will take **at least** $O(n^2)$ time!

# Enter the FFT

- With the fast Fourier transform,

HOW TO COMPUTE $C(x) = A(x) \cdot B(x)$

1. Pick n points, where $n \geq$ [the degree of $C(x)$] + 1.
2. Evaluate $A(x_k)$ at each of the n points. $O(n\log n)$
3. Evaluate $B(x_k)$ at each of the n points. $O(n\log n)$
4. Evaluate $C(x_k) = A(x_k) \cdot B(x_k)$ for each of the n points. $O(n)$
5. Convert our newfound value representation for $C(x_k)$ into a coefficient representation. $O(n\log n)$

Using the FFT, polynomial multiplication can be performed in **O(nlogn)** time!

# The Fourier Transform

- The Fourier transform (FT) turns a polynomial in coefficient representation into a value representation.
  - Say we have the polynomial $A(x) = 1 + 2x + 3x^2 + 4x^3$. We can compute the value representation (namely, the polynomial evaluated at the fourth roots of unity 1, i, -1, and -i) as

    $A(1) = 1 + 2(1) + 3(1)^2 + 4(1)^3$
    $A(i) = 1 + 2(i) + 3(i)^2 + 4(i)^3$
    $A(-1) = 1 + 2(-1) + 3(-1)^2 + 4(-1)^3$
    $A(-i) = 1 + 2(-i) + 3(-i)^2 + 4(-i)^3$

    or

    $$\begin{bmatrix} 1 & 1 & 1^2 & 1^3 \\ 1 & i & i^2 & i^3 \\ 1 & -1 & (-1)^2 & (-1)^3 \\ 1 & -i & (-i)^2 & (-i)^3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

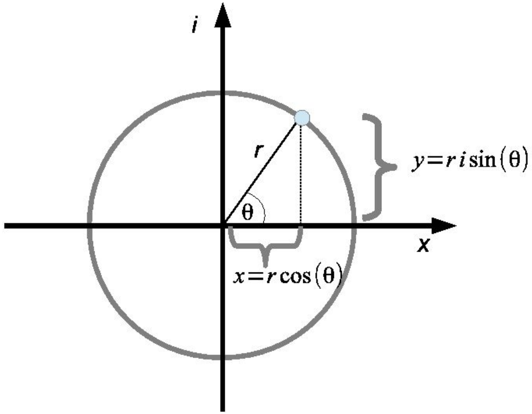
    We find that FT((1, 2, 3, 4)) = (10, -2 - 2i, -2, -2 + 2i).

# The Fourier Transform

- Formally, the discrete Fourier transform is defined as the mapping $DFT : \mathbb{R}^n \to \mathbb{R}^n, (f_0, ..., f_{n-1}) \mapsto (f(1), f(\omega), f(\omega^2), ..., f(\omega^{n-1}))$ where ω is the $n^{th}$ primitive root of unity.

# (side note) Finding ω, the $n^{th}$ Primitive Root of Unity

- The $n^{th}$ primitive root of unity will be $e^{2\pi i/n} = \cos(2\pi/n) + i\sin(2\pi/n)$.



# The Inverse Fourier Transform

- The inverse of the FT transforms a polynomial in *value* representation into *coefficient* representation.
- We can use this for the final step of polynomial multiplication (interpolation).

Mechanics-wise, the inverse of the Fourier transform just runs the FT on the value representation [e.g. (10, -2 - 2i, -2, -2 + 2i)], but substitutes $\omega^{-1}$ for $\omega$ and divides the output by n.

e.g. $FT^{-1}$((10, -2 - 2i, -2, -2 + 2i)) can be computed as

$f_0 = [10 + (-2 - 2i)(1) - 2(1)^2 + (-2 + 2i)(1)^3] / 4$
$f_1 = [10 + (-2 - 2i)(-i) - 2(-i)^2 + (-2 + 2i)(-i)^3] / 4$
$f_2 = [10 + (-2 - 2i)(-1) - 2(-1)^2 + (-2 + 2i)(-1)^3] / 4$
$f_3 = [10 + (-2 - 2i)(i) - 2(i)^2 + (-2 + 2i)(i)^3] / 4$

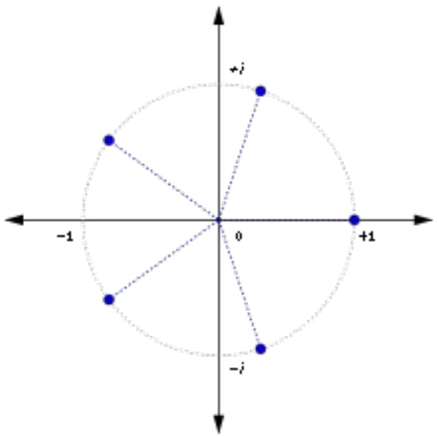
or

$$\frac{1}{4}\begin{bmatrix}1 & 1 & 1^2 & 1^3 \\ 1 & -i & (-i)^2 & (-i)^3 \\ 1 & -1 & (-1)^2 & (-1)^3 \\ 1 & i & i^2 & i^3\end{bmatrix}\begin{bmatrix}10 \\ -2-2i \\ -2 \\ -2+2i\end{bmatrix}$$

which gives (1, 2, 3, 4).

# (side note) Finding $\omega^{-1}$

- The **inverse** of the $n^{th}$ primitive root of unity will be $(e^{2\pi i/n})^{-1} = e^{-2\pi i/n} = \cos(-2\pi/n) + i\sin(-2\pi/n)$.



# The Fast Fourier Transform

- The **fast** Fourier transform is just a faster version of the Fourier transform. I bet you never would have guessed that.
- It does the same thing as the FT.

Its approach? Divide-and-conquer!

# The Fast Fourier Transform, elaborated

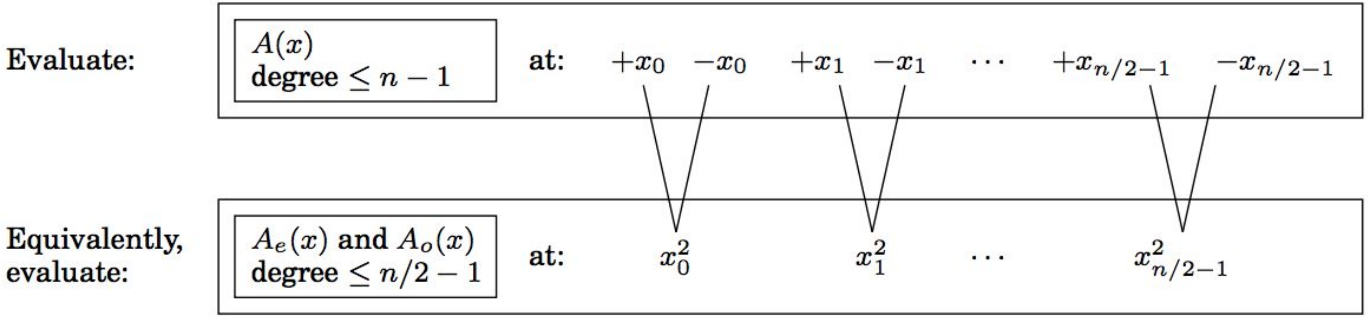
- Observation: any polynomial $A(x)$ is equal to $A_e(x^2) + xA_o(x^2)$
  - e.g. $A(x) = 1 + 2x + 3x^2 + 4x^3 = (1 + 3x^2) + x(2 + 4x^2)$,
    so $A_e(x) = 1 + 3x$ and $A_o(x) = 2 + 4x$

- By splitting polynomials into even and odd components, we end up with two polynomials of degree n / 2, which only need to be evaluated at n / 2 points (because $x^2$ will be the same for plus-minus pairs).

- Thus we have two problems of size n / 2, along with a linear combination step [multiplying $A_o(x^2)$ by x and adding $A_e(x^2)$ and $xA_o(x^2)$ together]. Our recurrence is $T(n) = 2T(n / 2) + O(n)$, and our runtime is **O(nlogn)**.

# The Fast Fourier Transform, elaborated

- This works at every step of the recurrence because
  - the $n^{th}$ roots of unity are always plus-minus paired ($\omega^{n/2+j} = -\omega^j$), and
  - the squares of the $n^{th}$ roots of unity are the $(n/2)^{nd}$ roots of unity



# FFT Pseudocode

**Figure 2.7** The fast Fourier transform (polynomial formulation)

function FFT($A, \omega$)
Input: Coefficient representation of a polynomial $A(x)$
of degree $\leq n-1$, where $n$ is a power of 2
$\omega$, an $n$th root of unity
Output: Value representation $A(\omega^0), \ldots, A(\omega^{n-1})$

if $\omega = 1$: return $A(1)$
express $A(x)$ in the form $A_e(x^2) + xA_o(x^2)$
call FFT($A_e, \omega^2$) to evaluate $A_e$ at even powers of $\omega$
call FFT($A_o, \omega^2$) to evaluate $A_o$ at even powers of $\omega$
for $j = 0$ to $n-1$:
  compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$

return $A(\omega^0), \ldots, A(\omega^{n-1})$