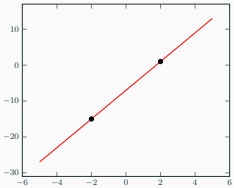
CSE 317: Design and Analysis of Algorithms
Fast Fourier Transform

Shahid Hussain

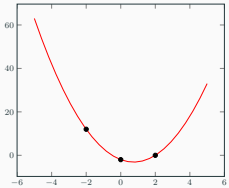
Fall 2025

SMCS, IBA

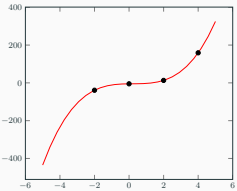
Representating Polynomials



$y = 4x - 7$



$y = 2x^2 - 3x - 2$



$y = 3x^3 - 2x^2 + x - 5$

- We need $n + 1$ distinct points to determine any n degree polynomial
- We can specify any n degree polynomial $A(x) = a_0 + a_1 x + \cdots + a_n x^n$ by either one of the following:
  - Its coefficients: $\langle a_0, a_1, \ldots, a_n \rangle$
  - The values: $A(x_0), A(x_1), \ldots, A(x_n)$
  - Roots: $r_1, r_2, \ldots, r_n$

Multiplying polynomials

- Let $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ and $B(x) = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1}$ be two $n-1$ degree polynomials
- Then the product: $C(x) = A(x) \cdot B(x)$ is a $2n - n$ degree polynomial: defined as

$$C(x) = c_0 + c_1 x + \cdots + c_{2n-2} x^{2n-2}$$

- The polynomial $C(x)$ can be computed in $O(n^2)$ time
- For example:

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4$$

- Generally: if $A(x) = \sum_{j=0}^{d} a_j x j$ and $B(x) = \sum_{j=0}^{d} b_j x^j$ be two degree d poly.
- $C(x) = A(x) \cdot B(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d}$ has coefficients:

$$c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0 = \sum_{j=0}^{k} a_j b_{j-k}$$

Polynomial Multiplication

- For two n degree polynomials $A(x)$ and $B(x)$ represented using their coefficients $\langle a_0, \ldots a_n \rangle$ and $\langle b_0, \ldots b_n \rangle$
- The polynomial $C(x) = A(x) \cdot B(x)$ can easily be computed using $A(x)$ and $B(x)$ as following:

Algorithm: PolynomialMultiplication

Input: Coefficients of $A(x)$ and $B(x)$

Output: The product $C(x) = A(x) \cdot B(x)$

1. Pick some $k \geq 2n + 1$ points $x_0, \ldots, x_{k-1}$
2. Compute $A(x_0), \ldots, A(x_{k-1})$ and $B(x_0), \ldots, B(x_{k-1})$
3. Compute $C(x_j) = A(x_j) \cdot B(x_j)$ for all $j = 0, \ldots k - 1$
4. Recover $C(x) = c_0 + c_1 x + \cdots + c_{2n} x^{2n}$

Polynomial Multiplication

- How to pick these points? randomly?
- One idea is to use positive-negative pairs i.e,

$$\pm x_0, \pm x_1, \ldots \pm x_{n/2-1}$$

  This will allow us to reuse some parts of computation e.g., $A(x_i)$ and $A(-x_i)$ will be same for all even powers of x_i
- For example: let $A(x)$ be

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

- We see that:

$$A(x) = A_e(x^2) + xA_o(x^2)$$

  where $A_e(\cdot)$ and $A_o(\cdot)$ are polynomials of degree $\leq n/2 - 1$ of even- and odd-numbered coefficients, respectively

Polynomial Multiplication

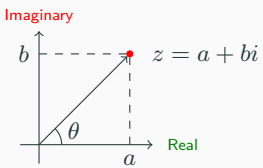
- Assuming n is even and the $\pm$ pairs of points $\pm x_i$

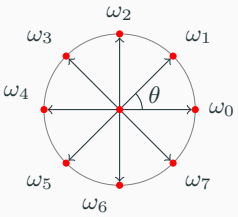
$$\begin{aligned} A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2) \end{aligned}$$

- So, evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ at just $n/2$ points $x_0^2, \ldots, x_{n/2-1}^2$
- Therefore, applying it recursively, the time complexity is now as $T(n) = 2T(n/2) + O(n) = O(n \log n)$
- Unfortunately, this only works for the top-level recursion, we need $n/2$ evaluation points $x_0^2, \ldots, x_{n/2-1}^2$ to be themselves plus-minus pairs, which is impossible, unless we use *complex numbers*

Complex Numbers and Roots of Unity

- Let $z = a + bi$ a complex number
- Let $x^8 = 1$ gives us 8 roots of unity, namely $\omega_0, \omega_1, \ldots, \omega_7$





Polynomial Multiplication

- For $x^n = 1$ we get $\omega_0, \omega_1, \ldots, \omega_{n-1}$ as n *roots of unity*
- Here $\omega = e^{2\pi i/n} = \cos(2\pi/n) + i\cos(2\pi/n)$
- The complex conjugate of $\omega = \overline{\omega}$

$$\begin{aligned} \overline{\omega} &= e^{-2\pi i/n} \\ &= \cos(-2\pi/n) + i\sin(-2\pi/n) \\ &= \cos(2\pi/n) - i\sin(2\pi/n) \qquad (\cos \text{ is even and } \sin \text{ is odd}) \end{aligned}$$

- We see that squaring a root of unity actual doubles the angle

$$\omega = e^{2\pi i/n} \implies \omega^2 = \overline{\omega}^2 = \left(e^{\pm 2\pi i/n}\right)^2 = e^{4\pi i/n}$$

The Fast Fourier Transform

Algorithm: FFT

Input: Coefficient rep. of $A(x)$ of degree $n = 2^k$ for $k \geq 0$ and ω, an n-th root of unity

Output: Value representation $A(\omega_0), \ldots, A(\omega_{n-1})$

1. **if** $\omega = 1$ **return** $A(1)$ ..... {*if base case*}
2. $A(x) \leftarrow A_e(x^2) + xA_o(x^2)$ ..... {*divide in even and odd parts*}
3. $A_e^* \leftarrow \mathrm{FFT}(A_e, \omega^2)$ ..... {*compute FFT of even parts with* ω^2}
4. $A_o^* \leftarrow \mathrm{FFT}(A_o, \omega^2)$ ..... {*compute FFT of odd parts with* ω^2}
5. **for** $j \leftarrow 0$ **to** $n - 1$
6. $\quad A^*(\omega_j) \leftarrow A_e^*(\omega^{2j}) + \omega^j A_o(\omega^{2j})$
7. **return** A^*

Interpolation

- Given a polynomial in *coefficient representation* $a_0, a_1, \ldots, n-1$ we can *evaluate* the polynomial at different values, called *value representation*, $A(x_0), A(x_1), \ldots, A(x_{n-1})$
- The value representation makes it trivial to multiply polynomials
- But we cannot ignore the coefficient representation since it is the form in which the input and output of our overall algorithm are specified
- So, with FFT algorithm we move from coefficient to values in $O(n \log n)$ time

$$\langle \text{values} \rangle = \text{FFT}(\langle \text{coefficients} \rangle, \omega)$$

- We use *interpolation* to find:

$$\langle \text{coefficients} \rangle = \frac{1}{n} \text{FFT}(\langle \text{values} \rangle, \omega^{-1})$$

- This step also takes $O(n \log n)$ time

Matrix representation: Evaluation

Vandermone Matrix

Let $A(x)$ be a polynomial of degree $\leq n-1$.

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

If $x_0, \ldots, x_{n-1}$ are distinct numbers then M (the matrix in the middle) is invertible.

Matrix representation: Interpolation

Vandermone Matrix

Let $A(x)$ be a polynomial of degree $\leq n-1$.

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

Matrix representation: Interpolation

Inversion formula

If we evaluate the matrix M (the matrix in the middle) for ω

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1\cdots & 1 & \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^j & \omega^j & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

Now, $M_n(w)^{-1} = \frac{1}{n} M_n(w^{-1})$