Fast Fourier Transform

Polynomial Multiplications and Fast Fourier Transform

Polynomial Multiplication

- Problem: Given two polynomials $p(x)$ and $q(x)$ with degree $d-1$, compute its product $r(x) = p(x)q(x)$.
- Each polynomial is encoded by its coefficients:
  - $p(x) = \sum_{i=0}^{d-1} a_i x^i \rightarrow (a_0, a_1, \dots, a_{d-1})$
  - $q(x) = \sum_{i=0}^{d-1} b_i x^i \rightarrow (b_0, b_1, \dots, b_{d-1})$
- Need to compute

$$r(x) = \sum_{i=0}^{2d-2} c_i x^i \quad \text{where} \quad c_i = \sum_{k=0}^{i} a_k\, b_{i-k}$$

- Naïve computation: $O(d^2)$

Polynomial Multiplication

Given $p(x) = \sum_{i=0}^{d-1} a_i x^i$ and $q(x) = \sum_{i=0}^{d-1} b_i x^i$

Compute $r(x) = \sum_{i=0}^{2d-2} c_i x^i$ where $c_i = \sum_{k=0}^{i} a_k \, b_{i-k}$

- Class Discussion: Can we do better than $O(d^2)$?

Divide and Conquer

- Adapt Karatsuba Algorithm
- Assume d is an integer power of 2.
- Write $p(x) = p_1(x) + p_2(x) \cdot x^{\frac{d}{2}}$ where

$$p_1(x) = a_0 + a_1 x + \cdots + a_{\frac{d}{2}-1} x^{\frac{d}{2}-1} \quad \text{and} \quad p_2(x) = a_{\frac{d}{2}} + a_{\frac{d}{2}+1} x + \cdots + a_{d-1} x^{\frac{d}{2}-1}$$

- Similarly, write $q(x) = q_1(x) + q_2(x) \cdot x^{\frac{d}{2}}$
- Then, $r = p_1 q_1 + (p_1 q_2 + p_2 q_1) x^{\frac{d}{2}} + p_2 q_2 x^d$. We need to compute

$$p_1 q_1, \qquad (p_1 q_2 + p_2 q_1), \qquad p_2 q_2$$

Adapting Karatsuba Algorithm

- Need to compute $p_1 q_1, p_2 q_2$, and $p_1 q_2 + p_2 q_1$
- $(p_1 q_2 + p_2 q_1) = (p_1 + p_2)(q_1 + q_2) - p_1 q_1 - p_2 q_2$
- Compute
  - $p_1 q_1$
  - $p_2 q_2$
  - $(p_1 + p_2)(q_1 + q_2)$
- One size-d multiplication → Three size-$\frac{d}{2}$ multiplications
- Time Complexity

$$T(d) = 3T\left(\frac{d}{2}\right) + O(d) \Longrightarrow \ T(d) = O\left(d^{\log_2 3}\right)$$

Fast Fourier Transform (FFT)

- In this lecture, we will learn a new divide and conquer algorithm with time complexity $O(d \log d)$!

- Fast Fourier Transform (FFT)
- Polynomial Interpolation
- Complex Numbers

Another Interpretation of A Polynomial

Polynomial Interpolation

- Represent a polynomial $p(x)$ of degree $d-1$ by d points
$$\big(\alpha_0, p(\alpha_0)\big), \big(\alpha_1, p(\alpha_1)\big), \dots, \big(\alpha_{d-1}, p(\alpha_{d-1})\big)$$

where $\alpha_0, \alpha_1, \dots, \alpha_{d-1}$ are distinct.

Framework for FFT

- Interpolation Step (FFT):
  - Choose $2d-1$ distinct numbers $\alpha_0, \alpha_1, \dots, \alpha_{2d-2}$, and
  - compute the values of $p(\alpha_0), p(\alpha_1), \dots, p(\alpha_{2d-2})$, $q(\alpha_0), q(\alpha_1), \dots, q(\alpha_{2d-2})$
- Multiplication Step:
  - For each $i = 0,1, \dots, 2d-2$, compute $r(\alpha_i) = p(\alpha_i)q(\alpha_i)$
  - Obtain interpolation for $r(x)$: $\big(\alpha_0, r(\alpha_0)\big), \big(\alpha_1, r(\alpha_1)\big), \dots, \big(\alpha_{2d-2}, r(\alpha_{2d-2})\big)$
- Recovery Step (inverse FFT):
  - Recover $(c_0, c_1, \dots, c_{2d-2})$, the polynomial $r(x) = \sum_{i=0}^{2d-2} c_i x^i$, from the interpolation obtained in the previous step.

Framework for FFT

$$p(x) = a_0 + a_1 x + \cdots + a_{d-1} x^{d-1}$$
$$q(x) = b_0 + b_1 x + \cdots + b_{d-1} x^{d-1}$$

Objective

$$r(x) = p(x) \cdot q(x)$$
$$= c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2}$$

Interpolation Step
(FFT)

Recovery Step
(Inverse FFT)

$$(\alpha_0, p(\alpha_0)), (\alpha_1, p(\alpha_1)), \ldots, (\alpha_{2d-2}, p(\alpha_{2d-2}))$$
$$(\alpha_0, q(\alpha_0)), (\alpha_1, q(\alpha_1)), \ldots, (\alpha_{2d-2}, q(\alpha_{2d-2}))$$

Multiplication
$r(\alpha_i) = p(\alpha_i) q(\alpha_i)$

$$(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \ldots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$$

Before we move on…

- Let's prove that d distinct points can indeed uniquely determine a polynomial of degree $d-1$.

Interpolation Theorem. Given d points $(x_0, y_0), (x_1, y_1), \ldots (x_{d-1}, y_{d-1})$ such that $x_i \neq x_j$ for any $i \neq j$, there exists a unique polynomial $p(x)$ with degree at most $d-1$ such that $p(x_i) = y_i$ for each i.

Proof of Interpolation Theorem

- Let $p(x) = \sum_{t=0}^{d-1} a_t\, x^t$. We have $y_i = \sum_{t=0}^{d-1} a_t\, x_i^t$ for each i.

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{d-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{d-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{d-1} & x_{d-1}^2 & \cdots & x_{d-1}^{d-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \end{bmatrix}$$

- We want to show: $(a_0, a_1, \dots, a_{d-1})$ satisfying the above equation is unique.
- The yellow matrix is a $\mathit{Vandermonde\ \ matrix}$ with determinant $\prod_{0 \le i < j \le d-1}(x_j - x_i)$, which is nonzero given $x_i \neq x_j$.
- Uniqueness is proved: $\mathbf{y} = X\mathbf{a} \Rightarrow \mathbf{a} = X^{-1}\mathbf{y}$

Step 1: Interpolation

Interpolation Step (FFT):

Choose $2d - 1$ distinct numbers $\alpha_0, \alpha_1, \dots, \alpha_{2d-2}$, and compute the values of $p(\alpha_0), p(\alpha_1), \dots, p(\alpha_{2d-2})$, $q(\alpha_0), q(\alpha_1), \dots, q(\alpha_{2d-2})$

Interpolation Step

- Interpolation Step (FFT):
  - Choose $2d-1$ distinct numbers $\alpha_0, \alpha_1, \dots, \alpha_{2d-2}$, and
  - compute the values of $p(\alpha_0), p(\alpha_1), \dots, p(\alpha_{2d-2})$, $q(\alpha_0), q(\alpha_1), \dots, q(\alpha_{2d-2})$

- Computing each $p(\alpha_i)$ or $q(\alpha_i)$ requires $O(d)$ time.
- We need to compute $4d-2$ of them.
- Overall time complexity: $O(d^2)$.
- Can we do faster by divide and conquer?

Some Notations

- Let $D = 2d - 1$.
- Assume D is an integer power of 2.

- Interpolation Step (FFT):
  - Choose D distinct numbers $\alpha_0, \alpha_1, \dots, \alpha_{D-1}$, and
  - compute the values of $p(\alpha_0), p(\alpha_1), \dots, p(\alpha_{D-1})$, $q(\alpha_0), q(\alpha_1), \dots, q(\alpha_{D-1})$

A Naïve Divide and Conquer Algorithm

- "Left-right decomposition": $p(\alpha_i) = p_1(\alpha_i) + p_2(\alpha_i) \cdot \alpha_i^{\frac{D}{2}}$
- Compute $p_1(\alpha_i)$ and $p_2(\alpha_i)$ recursively.
- Time complexity: $T(D) = 2T\left(\frac{D}{2}\right) + O(1) \Longrightarrow T(D) = O(D)$
- No faster than direct computation!
- Reason: no sophistication in it! We merely compute the $D - 1$ additions in different order…

Lesson we learned

- Computing each $p(\alpha_i)$ requires $O(D)$ time.
  - We need to compute $D-1$ additions, and there is no way to simplify it!
- We need to choose $\alpha_0, \alpha_1, \dots, \alpha_{D-1}$ in a clever way so that, for example, $p(\alpha_0)$ and $p(\alpha_1)$ can be computed together!

An Idea to Compute $p(\alpha_0)$ and $p(\alpha_1)$ Together

- Instead of the "left-right decomposition", we use "even-odd decomposition":
$$p(x) = p_e(x^2) + x \cdot p_o(x^2)$$
where
$$p_e(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{D-2} x^{\frac{D-2}{2}}$$
$$p_o(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{D-1} x^{\frac{D-2}{2}}$$
- Choose α_0 and α_1 such that $\alpha_1 = -\alpha_0$. We have
$$p_e(\alpha_0^2) = p_e(\alpha_1^2) \quad \text{and} \quad p_o(\alpha_0^2) = p_o(\alpha_1^2)$$

An Idea to Compute $p(\alpha_0)$ And $p(\alpha_1)$ Together

$$p(\alpha_0) = \boxed{p_e(\alpha_0^2)} + \alpha_0 \cdot \boxed{p_o(\alpha_0^2)}$$

$$p(\alpha_1) = p_e(\alpha_1^2) + \alpha_1 \cdot p_o(\alpha_1^2) = \boxed{p_e(\alpha_0^2)} - \alpha_0 \cdot \boxed{p_o(\alpha_0^2)}$$

Two size-D computations $\rightarrow$ ~~four~~ two size-$\frac{D}{2}$ computations, great!

A Divide and Conquer Attempt

1. Choose $\alpha_0, \alpha_1, \dots, \alpha_{D-1}$ such that $\alpha_0 = -\alpha_1$, $\alpha_2 = -\alpha_3$, …, $\alpha_{D-2} = -\alpha_{D-1}$.
2. Compute $p_e(\alpha_0^2), p_e(\alpha_2^2), \dots, p_e(\alpha_{D-2}^2)$ and $p_o(\alpha_0^2), p_o(\alpha_2^2), \dots, p_o(\alpha_{D-2}^2)$ recursively.
3. For each $i = 0,1, \dots, D-1$, compute $p(\alpha_i) = p_e\left(\alpha_i^2\right) + \alpha_i \cdot p_o\left(\alpha_i^2\right)$.

- Let $T(D)$ be the time complexity for computing $p(\alpha_0), p(\alpha_1), \dots, p(\alpha_{D-1})$.
- Step 2 above requires $2T\left(\frac{D}{2}\right)$ time.
- Step 3 above require $O(D)$ time.
- Overall time complexity: $T(D) = 2T\left(\frac{D}{2}\right) + O(D) \Longrightarrow T(D) = O(D\log D)$

Are We Done?

1. Choose $\alpha_0, \alpha_1, \dots, \alpha_{D-1}$ such that $\alpha_0 = -\alpha_1,\ \alpha_2 = -\alpha_3,\ \dots,\ \alpha_{D-2} = -\alpha_{D-1}$.
2. Compute $p_e(\alpha_0^2), p_e(\alpha_2^2), \dots, p_e(\alpha_{D-2}^2)$ and $p_o(\alpha_0^2), p_o(\alpha_2^2), \dots, p_o(\alpha_{D-2}^2)$ recursively.
3. For each $i = 0,1,\dots,D-1$, compute $p(\alpha_i) = p_e\left(\alpha_i^2\right) + \alpha_i \cdot p_o\left(\alpha_i^2\right)$.

NO!

To compute $p_e(\alpha_0^2), p_e(\alpha_2^2), \dots, p_e(\alpha_{D-2}^2)$ “recursively”, we need that

$$\begin{array}{cccc} \alpha_0^2 = -\alpha_2^2, & \alpha_4^2 = -\alpha_6^2, & \dots, & \alpha_{D-4}^2 = -\alpha_{D-2}^2 \\ \alpha_0^4 = -\alpha_4^4, & \alpha_8^4 = -\alpha_{16}^4, & \dots, & \alpha_{D-8}^4 = -\alpha_{D-4}^4 \end{array}$$

and so on...

We need complex numbers!

Complex Numbers

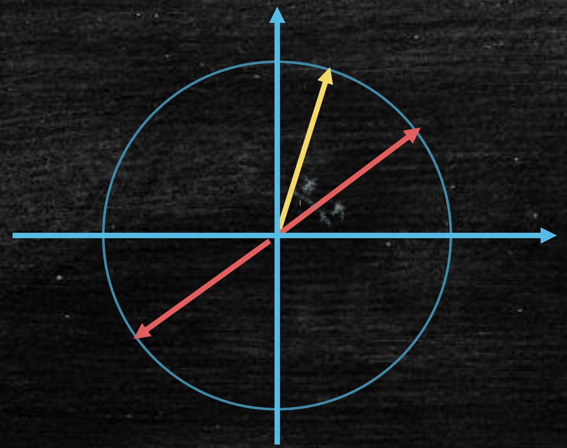
- $z = a + bi$
  - a: real part
  - b: imaginary part
  - $i = \sqrt{-1}$: imaginary unit
- Polar form: $z = r(\cos\theta + i\ \sin\theta)$
  - r: the length of the 2-dimensional vector (a, b)
  - θ: the angle between the vector (a, b) and the x-axis (the real axis)
- Euler's formula: $z = r(\cos\theta + i\ \sin\theta) = r \cdot e^{\theta i}$

Squares and Square Roots of Unit Length Complex Numbers

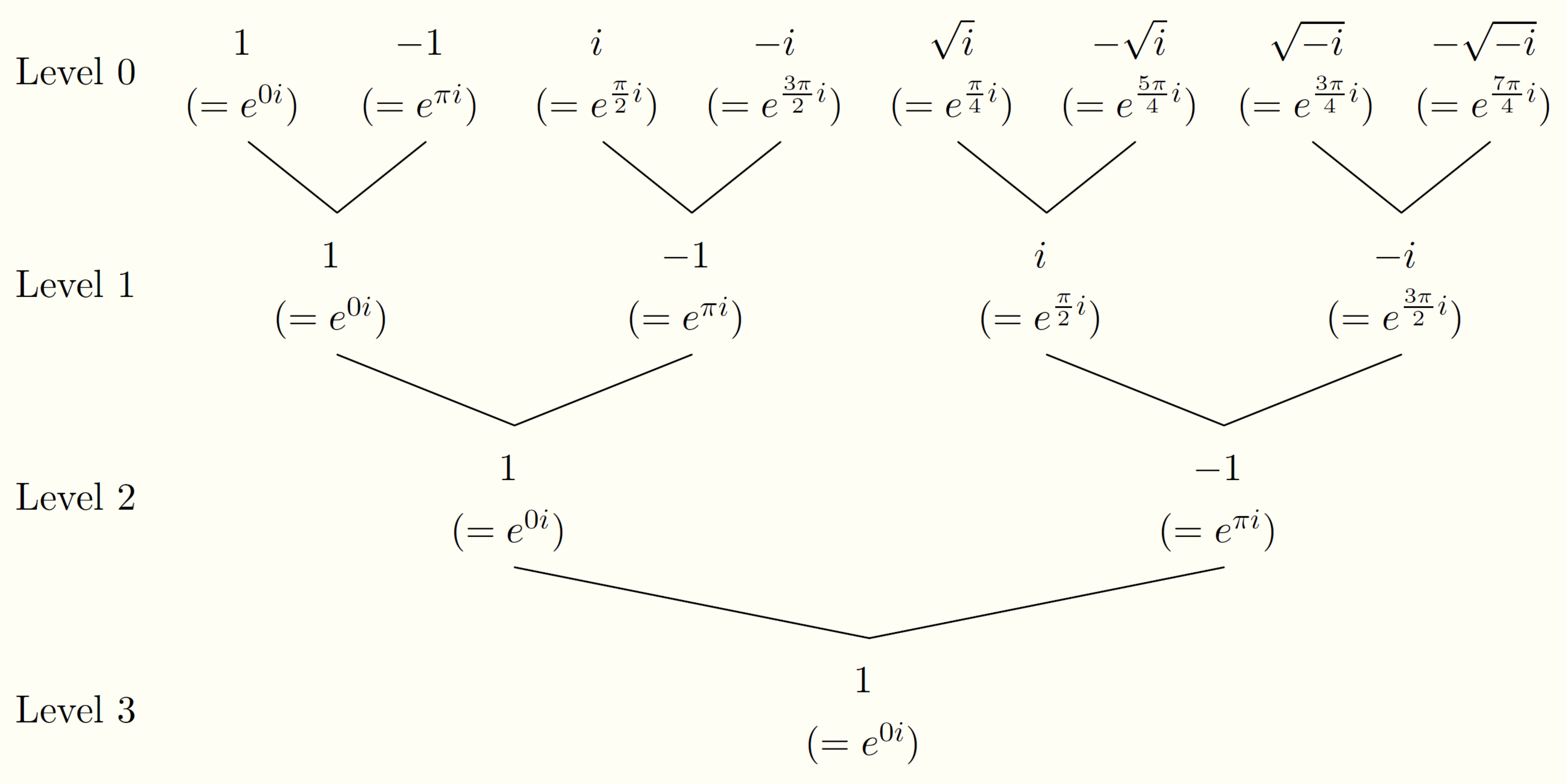
- The square of $e^{\theta i}$ is $e^{2\theta i}$: we have just rotated $e^{\theta i}$ by an angle θ.
- Two complex numbers of unit length opposite to each other have the same square:

$$\left(e^{(\theta+\pi)i}\right)^2 = e^{2\theta i} \cdot e^{2\pi i} = e^{2\theta i} = \left(e^{\theta i}\right)^2$$

- The square roots of $e^{\theta i}$ are $e^{\frac{\theta}{2}i}$ and $e^{\left(\frac{\theta}{2}+\pi\right)i}$



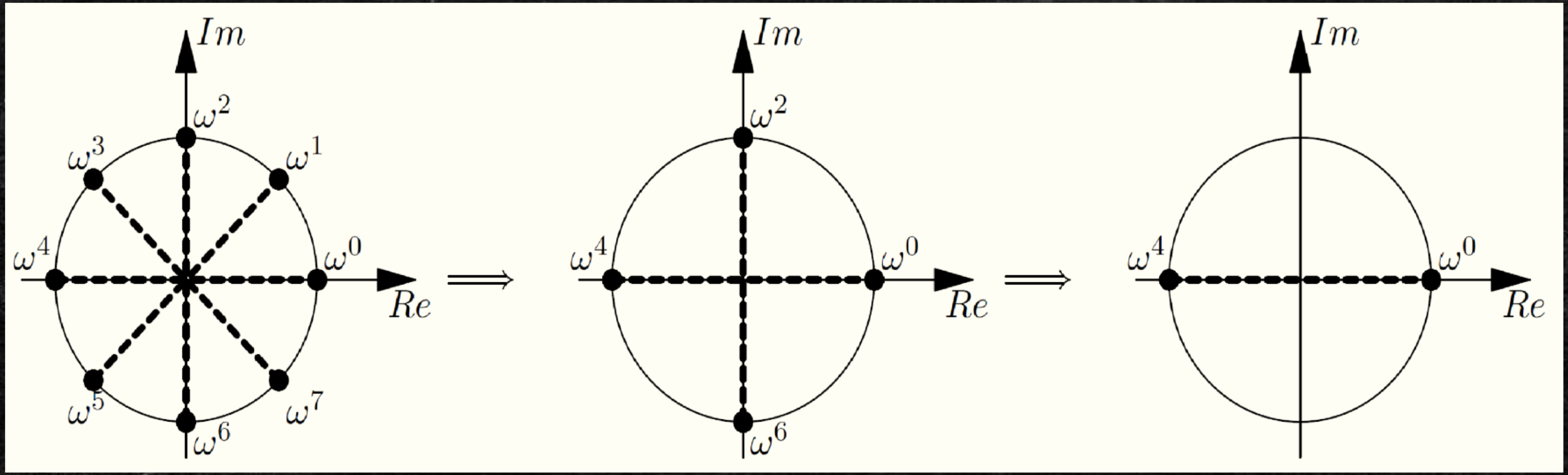
Example for $D = 8$



Example for $D = 8$

$$\omega_0 = 1, \quad \omega_1 = e^{\frac{\pi}{4}i}, \quad \omega_2 = e^{\frac{\pi}{2}i}, \quad \omega_3 = e^{\frac{3\pi}{4}i}$$

$$\omega_4 = e^{\pi i}, \quad \omega_5 = e^{\frac{5\pi}{4}i}, \quad \omega_6 = e^{\frac{3\pi}{2}i}, \quad \omega_7 = e^{\frac{7\pi}{4}i}$$



Interpolation: Putting Together

Algorithm 1: Fast Fourier Transform

FFT(p, ω): // p is a polynomial of degree $D-1$ and $\omega = e^{\frac{2\pi}{D}i}$

1. if $\omega = 1$, return $(p(1))$;
2. $p_e(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{D-2}x^{\frac{D-2}{2}}$
3. $p_o(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{D-1}x^{\frac{D-2}{2}}$
4. $\left(p_e(\omega^0), p_e(\omega^2), \ldots, p_e(\omega^{D-2})\right) \leftarrow$ FFT(p_e, ω^2);
5. $\left(p_o(\omega^0), p_o(\omega^2), \ldots, p_o(\omega^{D-2})\right) \leftarrow$ FFT(p_o, ω^2);
6. for $t = 0, 1, \ldots, D-1$:
7. $\qquad p(\omega^t) = p_e(\omega^{2t}) + \omega^t \cdot p_o(\omega^{2t})$
8. endfor
9. return $\left(p(\omega^0), p(\omega^1), \ldots, p(\omega^{D-1})\right)$;

Time Complexity for Interpolation Step

- Let $T(D)$ be the time complexity for computing **FFT**(p, ω), where p has degree $D-1$.
- We have $T(D) = 2T\left(\frac{D}{2}\right) + O(D) = O(D \log D)$.
- Interpolation step requires $T(D) = O(d \log d)$ time.

Framework for FFT

$$p(x) = a_0 + a_1 x + \cdots + a_{d-1} x^{d-1}$$
$$q(x) = b_0 + b_1 x + \cdots + b_{d-1} x^{d-1}$$

Objective

$$r(x) = p(x) \cdot q(x)$$
$$= c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2}$$

Interpolation Step
(FFT)

$O(d \log d)$

Recovery Step
(Inverse FFT)

$$(\alpha_0, p(\alpha_0)), (\alpha_1, p(\alpha_1)), \dots, (\alpha_{2d-2}, p(\alpha_{2d-2}))$$
$$(\alpha_0, q(\alpha_0)), (\alpha_1, q(\alpha_1)), \dots, (\alpha_{2d-2}, q(\alpha_{2d-2}))$$

Multiplication
$r(\alpha_i) = p(\alpha_i) q(\alpha_i)$

$$(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \dots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$$

Step 2: Multiplication

Multiplication Step:

For each $i = 0,1,\dots,2d-2$, compute $r(\alpha_i) = p(\alpha_i)q(\alpha_i)$

Obtain interpolation for $r(x)$: $(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \dots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$

It's easy! Just compute it one-by-one…

- For each $i = 0, 1, \dots, 2d - 2$, compute $r(\alpha_i) = p(\alpha_i)q(\alpha_i)$
- Time complexity: $O(d)$

Framework for FFT

$$p(x) = a_0 + a_1 x + \cdots + a_{d-1} x^{d-1}$$
$$q(x) = b_0 + b_1 x + \cdots + b_{d-1} x^{d-1}$$

Objective

$$r(x) = p(x) \cdot q(x) = c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2}$$

Interpolation Step (FFT)

$O(d \log d)$

Recovery Step (Inverse FFT)

$$(\alpha_0, p(\alpha_0)), (\alpha_1, p(\alpha_1)), \ldots, (\alpha_{2d-2}, p(\alpha_{2d-2}))$$
$$(\alpha_0, q(\alpha_0)), (\alpha_1, q(\alpha_1)), \ldots, (\alpha_{2d-2}, q(\alpha_{2d-2}))$$

Multiplication

$O(d)$

$$(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \ldots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$$

Step 3: Recovery

Recovery Step (inverse FFT):

Recover $(c_0, c_1, \ldots, c_{2d-2})$, the polynomial $r(x) = \sum_{i=0}^{2d-2} c_i x^i$, from the interpolation obtained in the previous step.

We Have Interpolation of $r(x)$ Now…

- We have $(1, r(1)), (\omega, r(\omega)), (\omega^2, r(\omega^2)), \ldots, (\omega^{D-1}, r(\omega^{D-1}))$, where $\omega = e^{\frac{2\pi}{D}i}$.

$$\begin{bmatrix} r(1) \\ r(\omega) \\ r(\omega^2) \\ \vdots \\ r(\omega^{D-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{D-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(D-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{D-1} & \omega^{2(D-1)} & \cdots & \omega^{(D-1)(D-1)} \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{D-1} \end{bmatrix}$$

What we want…

Complex Matrices Recap

- The complex conjugate of $z = a + bi$ is $z = a - bi$.
- Given two complex vectors $\mathbf{a}, \mathbf{b} \in \mathbb{C}^n$, their inner product is
$$\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=1}^{n} \overline{a_i} \cdot b_i$$
- $\mathbf{a}, \mathbf{b}$ are orthogonal if $\langle \mathbf{a}, \mathbf{b} \rangle = 0$; $\mathbf{a}, \mathbf{b}$ are orthonormal if $\langle \mathbf{a}, \mathbf{b} \rangle = 0$ and $\langle \mathbf{a}, \mathbf{a} \rangle = \langle \mathbf{b}, \mathbf{b} \rangle = 1$.
- A square matrix A is an orthonormal (unitary) matrix if every pair of its columns is orthonormal.
  - If columns are pairwise orthonormal, so are the rows.
- Conjugate transpose of A, denoted by A^*, is defined as $(A^*)_{i,j} = \overline{A_{j,i}}$.
- If A is an orthonormal, then A is invertible and $A^{-1} = A^*$.

Let's come back…

- We have $(1, r(1)), (\omega, r(\omega)), (\omega^2, r(\omega^2)), \ldots, (\omega^{D-1}, r(\omega^{D-1}))$, where $\omega = e^{\frac{2\pi}{D}i}$.

$$\begin{bmatrix} r(1) \\ r(\omega) \\ r(\omega^2) \\ \vdots \\ r(\omega^{D-1}) \end{bmatrix} = \underbrace{\boxed{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{D-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(D-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{D-1} & \omega^{2(D-1)} & \cdots & \omega^{(D-1)(D-1)} \end{bmatrix}}}_{A(\omega)} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{D-1} \end{bmatrix}$$

$A: \mathbb{C} \to \mathbb{C}^{D\times D}$ is a function.

Proposition. $\frac{1}{\sqrt{D}}A(\omega)$ is orthonormal for $\omega = e^{\frac{2\pi}{D}i}$.

Proof.

- Let $\mathbf{c}_i, \mathbf{c}_j$ be two arbitrary columns of $\frac{1}{\sqrt{D}}A(\omega)$.

$$\langle \mathbf{c}_i, \mathbf{c}_j \rangle = \sum_{k=1}^{D} \frac{1}{D} \cdot \overline{\omega^{(k-1)(i-1)}} \cdot \omega^{(k-1)(j-1)} = \frac{1}{D} \sum_{k=1}^{D} \omega^{(k-1)(j-i)}$$

- If $i = j$, we have $\langle \mathbf{c}_i, \mathbf{c}_j \rangle = \frac{1}{D} \sum_{k=1}^{D} \omega^0 = 1$;
- If $i \neq j$, then

$\omega^D = e^{2\pi i} = 1$

$$\langle \mathbf{c}_i, \mathbf{c}_j \rangle = \frac{1}{D} \sum_{k=1}^{D} \omega^{(k-1)(j-i)} = \frac{1}{D} \frac{1 - \omega^{(j-i)D}}{1 - \omega^{j-i}} = 0$$

- Thus, $\frac{1}{\sqrt{D}}A(\omega)$ is orthonormal.

Inverting $A(\omega)$...

- **Theorem.** If A is an orthonormal, then A is invertible and $A^{-1} = A^*$.
- **Proposition.** $\frac{1}{\sqrt{D}} A(\omega)$ is orthonormal for $\omega = e^{\frac{2\pi}{D} i}$.
- We have

$$A(\omega)^{-1} = \left(\sqrt{D} \cdot \frac{1}{\sqrt{D}} \cdot A(\omega)\right)^{-1} = \frac{1}{\sqrt{D}} \left(\frac{1}{\sqrt{D}} \cdot A(\omega)\right)^{-1} = \frac{1}{\sqrt{D}} \left(\frac{1}{\sqrt{D}} \cdot A(\omega)\right)^{*} = \frac{1}{D} A(\omega)^*$$

- Therefore,

$$\left(A(\omega)^{-1}\right)_{i,j} = \frac{1}{D} \overline{\left(A(\omega)\right)_{j,i}} = \frac{1}{D} \cdot \omega^{-(i-1)(j-1)} = \frac{1}{D} \left(\omega^{-1}\right)^{(i-1)(j-1)},$$

which implies

$$A(\omega)^{-1} = \frac{1}{D} \cdot A(\omega^{-1}).$$

Putting $A(\omega)^{-1} = \frac{1}{D} \cdot A(\omega^{-1})$ back

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{D-1} \end{bmatrix} = \frac{1}{D} \cdot A(\omega^{-1}) \cdot \begin{bmatrix} r(1) \\ r(\omega) \\ r(\omega^2) \\ \vdots \\ r(\omega^{D-1}) \end{bmatrix}$$

- This is very similar to the first step!
- Let $s(x)$ be a polynomial with coefficients $r(1), r(\omega), \dots r(\omega^{D-1})$. Can we just apply FFT$(s, \omega^{-1})$?
- $(\omega^{-1}, \omega^{-2}, \dots, \omega^{-(D-1)})$ is just the same as $(\omega^1, \omega^2, \dots, \omega^{(D-1)})$ with a clockwise orientation!
- Yes, we can just apply FFT(s, ω^{-1})!

Framework for FFT

$$p(x) = a_0 + a_1 x + \cdots + a_{d-1} x^{d-1}$$
$$q(x) = b_0 + b_1 x + \cdots + b_{d-1} x^{d-1}$$

Objective

$$r(x) = p(x) \cdot q(x)$$
$$= c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2}$$

Interpolation Step (FFT)

$O(d \log d)$

Recovery Step (Inverse FFT)

$O(d \log d)$

$$(\alpha_0, p(\alpha_0)), (\alpha_1, p(\alpha_1)), \ldots, (\alpha_{2d-2}, p(\alpha_{2d-2}))$$
$$(\alpha_0, q(\alpha_0)), (\alpha_1, q(\alpha_1)), \ldots, (\alpha_{2d-2}, q(\alpha_{2d-2}))$$

Multiplication

$O(d)$

$$(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \ldots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$$

Putting 3 Steps Together

Putting Three Steps Together

Algorithm 2: Polynomial multiplication by FFT

Multiply(p, q): // p, q are two polynomials with degrees at most d

1. let D be the smallest integer power of 2 such that $d \leq \frac{D}{2}$;
2. let $\omega = e^{\frac{2\pi}{D}i}$;
3. $(p_0, p_1, \dots, p_{D-1}) \leftarrow$ **FFT**(p, ω); // where $p_i = p(\omega^i)$
4. $(q_0, q_1, \dots, q_{D-1}) \leftarrow$ **FFT**(q, ω); // where $q_i = q(\omega^i)$
5. for each $t = 0,1, \dots, D-1$: compute $r_t \leftarrow p_t \cdot q_t$
6. let $s(x) = \sum_{t=0}^{D-1} r_t x^t$
7. $(c_0, c_1, \dots, c_{D-1}) \leftarrow$ **FFT**(s, ω^{-1});
8. let $r(x) = \sum_{t=0}^{D-1} \frac{c_t}{D} x^t$;
9. return r;

Overall Time Complexity

$$O(d \log d) + O(d) + O(d \log d) = O(d \log d)$$

Recap

Three Steps:

$$p(x) = a_0 + a_1 x + \cdots + a_{d-1} x^{d-1}$$
$$q(x) = b_0 + b_1 x + \cdots + b_{d-1} x^{d-1}$$

Objective

$$r(x) = p(x) \cdot q(x)$$
$$= c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2}$$

Interpolation Step
(FFT)

Recovery Step
(Inverse FFT)

$$(\alpha_0, p(\alpha_0)), (\alpha_1, p(\alpha_1)), \dots, (\alpha_{2d-2}, p(\alpha_{2d-2}))$$
$$(\alpha_0, q(\alpha_0)), (\alpha_1, q(\alpha_1)), \dots, (\alpha_{2d-2}, q(\alpha_{2d-2}))$$

Multiplication
$r(\alpha_i) = p(\alpha_i) q(\alpha_i)$

$$(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \dots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$$

Step 1: Interpolation

- Naïve computation: $O(d^2)$
- Even-odd decomposition: $p(x) = p_e(x^2) + x \cdot p_o(x^2)$
- "Tree structure" for $\alpha_i, \alpha_i^2, \alpha_i^4, \dots, \alpha_i^D$
- Choose $\alpha_i = \omega^i$ where $\omega = e^{\frac{2\pi}{D}i}$
- FFT to compute $\mathbf{p} = A(\omega) \cdot \mathbf{a}$ and $\mathbf{q} = A(\omega) \cdot \mathbf{b}$

$$p(x) = a_0 + a_1 x + \cdots + a_{d-1} x^{d-1}$$
$$q(x) = b_0 + b_1 x + \cdots + b_{d-1} x^{d-1}$$

Interpolation Step (FFT)

$$(\alpha_0, p(\alpha_0)), (\alpha_1, p(\alpha_1)), \dots, (\alpha_{2d-2}, p(\alpha_{2d-2}))$$
$$(\alpha_0, q(\alpha_0)), (\alpha_1, q(\alpha_1)), \dots, (\alpha_{2d-2}, q(\alpha_{2d-2}))$$

Step 2: Multiplication

- Just perform $2d - 1$ normal complex number multiplications.

$(\alpha_0, p(\alpha_0)), (\alpha_1, p(\alpha_1)), \ldots, (\alpha_{2d-2}, p(\alpha_{2d-2}))$

$(\alpha_0, q(\alpha_0)), (\alpha_1, q(\alpha_1)), \ldots, (\alpha_{2d-2}, q(\alpha_{2d-2}))$

$\xrightarrow{\text{Multiplication}}$

$(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \ldots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$

Multiplication

$r(\alpha_i) = p(\alpha_i)q(\alpha_i)$

Step 3: Recovery

- We have $\mathbf{r} = A(\omega) \cdot \mathbf{c}$, and we want to recover $\mathbf{c}$ from $\mathbf{r}$ and $A(\omega)$.
- Nice property of A:

$$A(\omega)^{-1} = \frac{1}{D} \cdot A(\omega^{-1})$$

- Thus, $\mathbf{c} = \frac{1}{D} \cdot A(\omega^{-1}) \cdot \mathbf{r}$, and we can compute $A(\omega^{-1}) \cdot \mathbf{r}$ by FFT again.

$$\begin{aligned} r(x) &= p(x) \cdot q(x) \\ &= c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2} \end{aligned}$$

Recovery Step
(Inverse FFT)

$$(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \ldots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$$

Polynomial Multiplications vs Integer Multiplications

- $23341 = 2 \times 10^4 + 3 \times 10^3 + 3 \times 10^2 + 4 \times 10 + 1$
- $p(x) \quad = 2x^4 \qquad + 3x^3 \qquad + 3x^2 \qquad + 4x \qquad + 1$
- Polynomials and integers are similar!
- Perhaps the only difference in multiplications is "carry".
- FFT-based algorithms for integer multiplications:
  - Schonhage-Strassen (1971): $O(n \log n \log\log n)$
  - Furer (2007): $O(n \log n \log^* n)$
  - Harvey and van der Hoeven (2019): $O(n \log n)$