Algorithm Design and Analysis

The Fast Fourier Transform

Goals for today

- Review some math, i.e., **polynomials** and **complex numbers**
- Derive the **Fast Fourier Transform** algorithm, and use it to produce a fast algorithm for polynomial multiplication
- (Optional) time permitting, FFT over finite fields

Quick review: polynomials

- A polynomial of degree d is a function p that looks like

$$p(x) \coloneqq \sum_{i=0}^{d} c_i\, x^i = c_d x^d + c_{d-1} x^{d-1} + \cdots + c_1 x + c_0$$

- Uniquely described by its coefficients $\langle c_d, c_{d-1}, \ldots, c_1, c_0 \rangle$
- Uniquely described by its value at $d + 1$ distinct points (the unique reconstruction theorem)

Quick review: polynomials

Given polynomials $A(x)$ and $B(x)$,

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d$$

$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d$$

Their product is

$$C(x) = c_0 + c_1 x + c_2 x^2 + \cdots c_{2d} x^{2d}$$

where

$$c_k = \sum_{i+j=k} a_i b_j = \sum_{i=0}^{k} a_i b_{k-i}$$

Review: complex numbers

- The field of **complex numbers** consists of numbers of the form
$$a + bi$$
- $i^2 = -1$ by definition
- Useful because every polynomial equation has a solution over the complex numbers. Not true over reals.

Roots of unity

- An $\boldsymbol{n}^{\mathbf{th}}$ **root of unity** is an n^{th} root of 1, i.e.,

$$\omega^n = 1$$

- There are exactly n complex n^{th} roots of unity, given by

$$e^{\frac{2\pi i k}{n}}, \qquad k = 0,1,\dots,n-1$$

- Can also write

$$e^{\frac{2\pi i k}{n}} = \left(\boldsymbol{e^{\frac{2\pi i}{n}}}\right)^k$$

Roots of unity

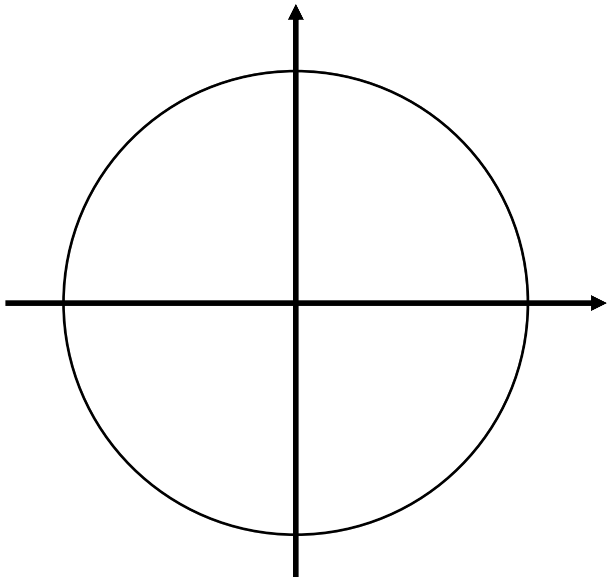
- The number $e^{\frac{2\pi i}{n}}$ is called a **primitive n^{th} root of unity**

$$e^{\frac{2\pi ik}{n}} = \left(e^{\frac{2\pi i}{n}}\right)^k$$

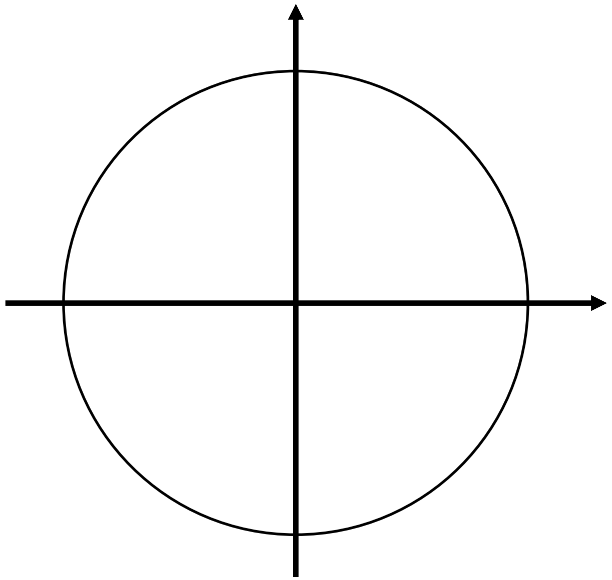
- Formally, ω

$$\begin{cases} \omega^n = 1 \\ \omega^k \neq 1 \ \text{ for } \ 0 < k < n \end{cases}$$

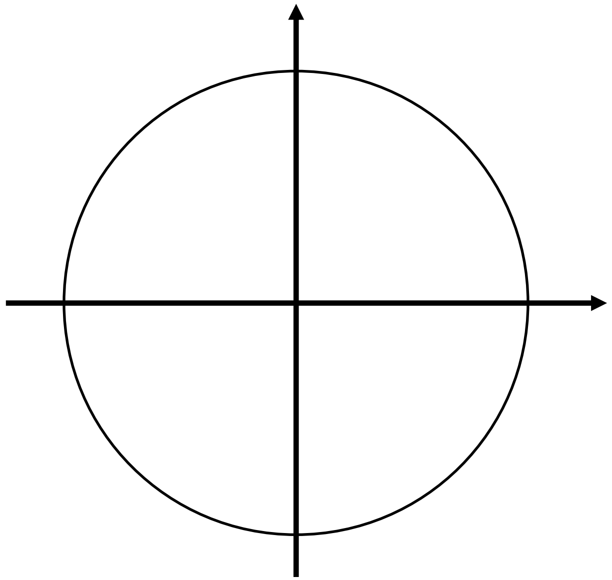
Roots of unity



2^{nd} roots of unity



4^{th} roots of unity



8^{th} roots of unity

Back to polynomial multiplication

- Directly using the definition of the product of two polynomials would give us an $O(d^2)$ algorithm
- Karatsuba can bring this down to $O(d^{1.58})$
- What if we used a different representation?

A: $\boldsymbol{A(x_0), A(x_1), A(x_2), \dots, A(x_d), \dots, A(x_{2d})}$

B: $\boldsymbol{B(x_0), B(x_1), B(x_2), \dots, B(x_d), \dots, B(x_{2d})}$

$\Downarrow$

C: $\boldsymbol{C(x_0), C(x_1), C(x_2), \dots, C(x_d), \dots, C(x_{2d})}$

Fast polynomial multiplication

1. Pick $N = 2d + 1$ points $x_0, x_1, \dots, x_{N-1}$
2. Evaluate $A(x_0), A(x_1), \dots, A(x_{N-1})$ and $B(x_0), B(x_1), \dots, B(x_{N-1})$
3. Compute $C(x_k) =$
4. Interpolate $C(x_0), \dots, C(x_{N-1})$ to get the coefficients of C

How do we do steps 2 and 4 efficiently???

To Point-Value Form

- Consider the polynomial A of degree 7

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

- Suppose we want to evaluate $A(1)$ and $A(-1)$

$$A(1) = a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7$$

$$A(-1) = a_0 - a_1 + a_2 - a_3 + a_4 - a_5 + a_6 - a_7$$

How to make it recursive?

- Consider the polynomial A of degree 7

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

- What if we split in half (like last slide) but keep it as a polynomial?

$$\begin{aligned} Z &= a_0 + a_2 + a_4 + a_6 \\ W &= a_1 + a_3 + a_5 + a_7 \end{aligned} \Rightarrow \begin{aligned} A_{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + a_6 x^3 \\ A_{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + a_7 x^3 \end{aligned}$$

$$\boxed{A(x) = \qquad\qquad\qquad\qquad}$$

A divide-and-conquer idea

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

- This formula gives us a key ingredient for ***divide-and-conquer***
  - We want to evaluate an N-term polynomial at N points
  - Break into two $N/2$-term polynomials and evaluate at $N/2$ points
  - Combine the two halves using the formula above

What points should we use for x?

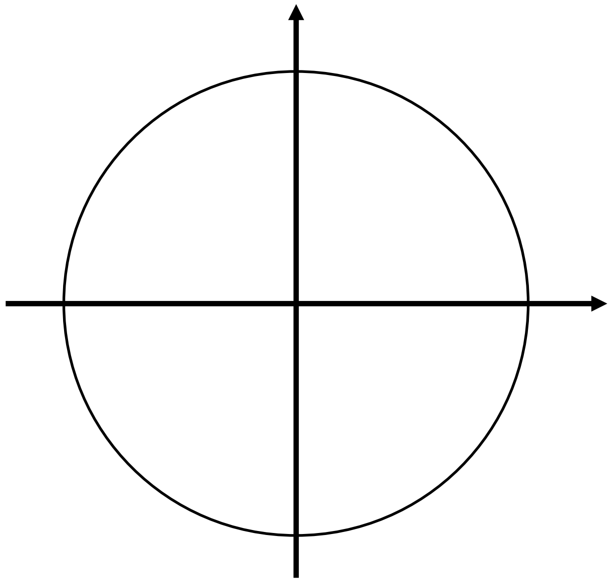
- But what to do about the x^2
- We want to evaluate N points and recurse on a problem that evaluates $N/2$ points… such that the squares of the N points are the $N/2$ points…

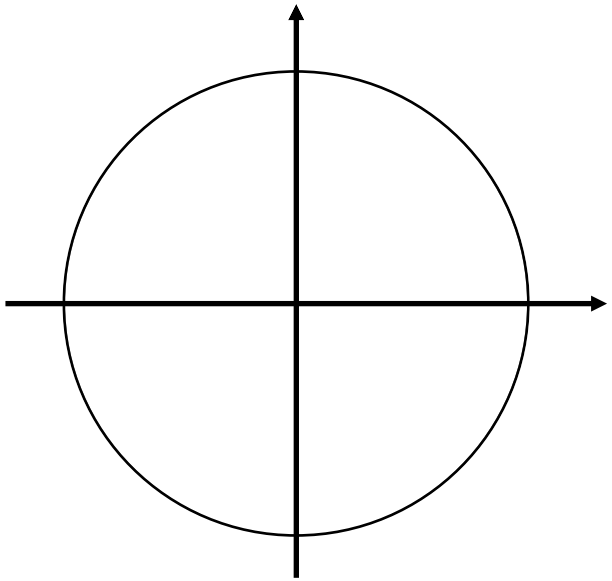
Roots of unity to the rescue!!!

- Recall the n^{th} roots of unity over the complex field are

$$\omega^k \qquad \text{for } k = 0,1,\ldots,n-1$$

where $\omega = e^{\frac{2\pi i}{n}}$ is our “primitive” n^{th} root of unity





The Fast Fourier Transform

- Assume N is a power of two (pad with zero coefficients)
- **Choose $x_0, x_1, \ldots, x_{N-1}$ to be N^{th} roots of unity**
- In other words, set $\omega = \exp\left(\frac{2\pi i}{N}\right)$ then set $x_k = \omega^k$
- To evaluate $A(x)$ at $\omega^0, \omega^1, \omega^2, \ldots, \omega^N$
  - Break into $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$
  - Evaluate those at $\omega^0, \omega^2, \omega^4, \ldots$ ⟵ ***The $(N/2)^{th}$ roots of unity!!!***
  - Combine using $A(\omega^k) = A_{\text{even}}(\omega^{2k}) + \omega^k\, A_{\text{odd}}(\omega^{2k})$

FFT($[a_0, a_1, \ldots, a_{N-1}], \omega, N$) $= \{$ *// Returns F =* $[A(\omega^0), A(\omega^1), \ldots, A(\omega^{N-1})]$

if $N = 1$ **then return** ______

$F_{\text{even}} \leftarrow$ **FFT(** ________________________________

$F_{\text{odd}} \leftarrow$ **FFT(** ________________________________

$x \leftarrow 1$ *// x stores* ω^k

for $k = 0$ **to** $N - 1$ **do** $\{$ *// Compute* $A(\omega^k) = A_{\text{even}}(\omega^{2k}) + \omega^k A_{\text{odd}}(\omega^{2k})$

$F[k] \leftarrow$ ____

$x \leftarrow x \times \omega$ *// In practice, beware rounding errors…*

$\}$ **return** F

$\}$

Back to multiplication

1. Pick $N = 2d + 1$ points $x_0, x_1, \dots, x_{N-1}$ (Pick N^{th} roots of unity)
2. Evaluate $A(x_0), \dots, A(x_{N-1})$ and $B(x_0), \dots, B(x_{N-1})$ (Using FFT)
3. Compute $C(x_k) = A(x_k)\, B(x_k)$
4. Interpolate $C(x_0), \dots, C(x_{N-1})$ to get the coefficients of C

One step to go…

Question break

Inverse FFT

- Given $C(\omega^0), C(\omega^1), \dots, C(\omega^{N-1})$ where $N = 2d + 1$
- We want to get the N coefficients of $C(x)$ back
- We're going to do it with… maths!

Observation: Evaluating a polynomial at a point can be represented as a vector-vector product:

Inverse FFT continued

Corollary: Evaluating a polynomial at many points can be represented as a matrix-vector product

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \cdots & x_{N-1}^{N-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{bmatrix} = \begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{N-1}) \end{bmatrix}$$

Theorem (Vandermonde): This matrix is invertible iff the x_i are distinct

Inverse FFT continued

- In our case, $x_k = \omega^k$ where ω is a principle N^{th} root of unity, so

$$FFT(\omega, N) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ & & & \ddots & \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)^2} \end{bmatrix}$$

- Element in row k, column j, is $\left(\omega^k\right)^j = \omega^{kj}$

We want to figure out $FFT^{-1}(\omega, N)$

Inverse FFT continued

Idea: Consider FFT with inverse root of unity, i.e.

$$FFT(\omega^{-1}, N)$$

What is the product of $FFT(\omega, N) \times FFT(\omega^{-1}, N)$? The (k, j) entry is

Inverse FFT continued

- Entry (k, j) of $FFT(\omega, N) \times FFT(\omega^{-1}, N)$ is:

$$\sum_{s=0}^{N-1} \omega^{-ks} \omega^{sj}$$

- What does the diagonal of the product look like? $(k = j)$

Inverse FFT continued

- Entry (k, j) of $FFT(\omega, N) \times FFT(\omega^{-1}, N)$ is:

$$\sum_{s=0}^{N-1} \omega^{-ks} \omega^{sj}$$

Reminder: *ω is a primitive root of unity*

$$\begin{cases} \omega^N = 1 \\ \omega^k \neq 1 \text{ for } 0 < k < N \end{cases}$$

- What do the non-diagonal entries of the product look like? $(k \neq j)$

Inverse FFT continued

- So, we've just showed that

$$FFT(\omega, N) \times FFT(\omega^{-1}, N) = \begin{bmatrix} N & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & N \end{bmatrix} = N \begin{bmatrix} 1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Therefore

$$FFT^{-1}(\omega, N) =$$

Back to multiplication

1. Pick $N = 2d + 1$ points $x_0, x_1, \dots, x_{N-1}$ (Pick N^{th} roots of unity)
2. Evaluate $A(x_0), \dots, A(x_{N-1})$ and $B(x_0), \dots, B(x_{N-1})$ (Using FFT)
3. Compute $C(x_k) = A(x_k)\, B(x_k)$
4. Interpolate $C(x_0), \dots, C(x_{N-1})$ to get the coefficients of C (Inverse FFT)

Runtime:

Question break #2

FFT over finite fields (optional)

- We defined FFT in terms of roots of unity over complex numbers
- Did we really ***need*** to use complex numbers?
- We needed N N^{th} roots of unity to do divide-and-conquer
- Other fields have roots of unity too!
- E.g. integers $\bmod p$ for a prime p

FFT over finite fields (optional)

Caveats:

- Need to pick a sufficiently large prime p.
- Not all primes work for any N. A good choice is $(cN + 1)$.
- The field must have N N^{th} roots of unity (guaranteed if $p = cN + 1$).
- Must find a primitive N^{th} root of unity (doable with number theory)

Take-home messages

- FFT is super cool
- The first key idea was to divide a polynomial into odd and even terms and use ***divide-and-conquer.***
- To make the points line up in the recursive case, we had to evaluate the polynomials at ***roots of unity***.