Lecture 3: Divide and Conquer and FFT

- Learn/review Divide and Conquer
- Fast Fourier Transform: The Magical Algorithm

Resources

- CLRS, *Introduction to Algorithms*, Chap 30
- Erikson, *Algorithms*, Chapter A online
- CMU 15-451, Introduction to Algorithms, *Fast Fourier Transform*

PRINCETON COMPUTER SCIENCE

Divide and Conquer: Easy like 1,2,3

1. **Divide:** *the problem instance into smaller subinstances*
2. **Recurse:** *to solve each subinstance recursively*
3. **Combine:** *subsolutions into a solution for the original instance*

Canonical Example: Merge Sort

Input: list of unsorted integers.
Goal: list in the sorted order.

pivot

Sort left half

Sort right half

Combine!!

Canonical Example: Merge Sort

Input: list of unsorted integers.
Goal: list in the sorted order.

function MergeSort (**list** L) {
    if ($|L| = 1$) **then return** L
    else {
        let $L_\ell \leftarrow$ left half of L
        let $L_r \leftarrow$ right half of L
        let sorted-$L_\ell \leftarrow$ MergeSort (L_ℓ) $T(\frac{n}{2})$
        let sorted-$L_r \leftarrow$ MergeSort (L_r) $T(\frac{n}{2})$

        return combine(sorted-L_ℓ, sorted-L_r)
        }
}

$T(n)$

$O(n)$

Running Time Recurrences

$T(n)$ ← *running time on instances of size n*

Goal: express $T(n)$ recursively in terms of $T(k)$ for $k < n$

Merge Sort Runtime Recurrence

$T(n) \leftarrow$ *running time on instances of size n*

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

$$T(1) = O(1)$$

"Master Theorem"

"recurrence relation":

Fact: all recurrences of the form $T(n) \leq a \cdot T\left(\frac{n}{b}\right) + f(n)$ have a known mechanical sol^{n}

Analyzing the recurrence

$$T(n) \le 2T\left(\frac{n}{2}\right) + cn \qquad T(1) = O(1)$$

$$\le 2\cdot\left(2\cdot T\left(\frac{n}{4}\right) + c\cdot\frac{n}{2}\right) + cn$$

$$= 4\cdot T\left(\frac{n}{4}\right) + cn + cn$$

$$\le 4\left(2T\left(\frac{n}{8}\right) + c\cdot\frac{n}{4}\right) + cn + cn$$

$$= 8\cdot T\left(\frac{n}{8}\right) + cn + cn + cn$$

$$\vdots$$

$$\le 2^i\cdot T\left(\frac{n}{2^i}\right) + i\cdot cn$$

plug in $i = \log_2 n$

can always apply master theorem.

But "unfolding" is easier & don't need to remember!

Analyzing the recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn \qquad T(1) = O(1)$$

$$\leq 2\cdot\left(2\cdot T\left(\frac{n}{4}\right) + c\cdot\frac{n}{2}\right) + cn$$

$$= 4\cdot T\left(\frac{n}{4}\right) + cn + cn$$

$$\leq 4\left(2T\left(\frac{n}{8}\right) + c\cdot\frac{n}{4}\right) + cn + cn$$

$$= 8\cdot T\left(\frac{n}{8}\right) + cn + cn + cn$$

$$\vdots$$

$$\leq 2^i\cdot T\left(\frac{n}{2^i}\right) + i\cdot cn$$

plug in $i = \log_2 n$

$\leq O(n\cdot\log n)$

TIP: assume n is a power of 2 when needed. simplifies calculation. Later remove assumption.

Another recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn^2$$

$$\leq 2\cdot\left(2\cdot T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)^2\right) + cn^2$$

$$= 4\cdot T\left(\frac{n}{4}\right) + \frac{c\cdot n^2}{2^2} + cn^2$$

$$\leq 8\,T\left(\frac{n}{8}\right) + cn^2/4 + cn^2/2 + cn^2$$

$$\vdots$$

$$\leq 2^i\cdot T\left(\frac{n}{2^i}\right) + cn^2\left(1 + 1/2 + 1/2^2 + \ldots + 1/2^{i-1}\right)$$

$$\leq O(n^2).$$

$\boxed{\text{Plug in } i = \log_2 n}$

Note: we got n^2 & not $n^2 \log n$ here. Why?

Anotherrrrrrr recurrence

$$T(n) \leq 2T(n-1)$$

$T(1) = 1$

$\vdots$

$\leq 2^n$

The *Fast Fourier Transform*

“the most … *ang*

Top 10 a

(IEEE C

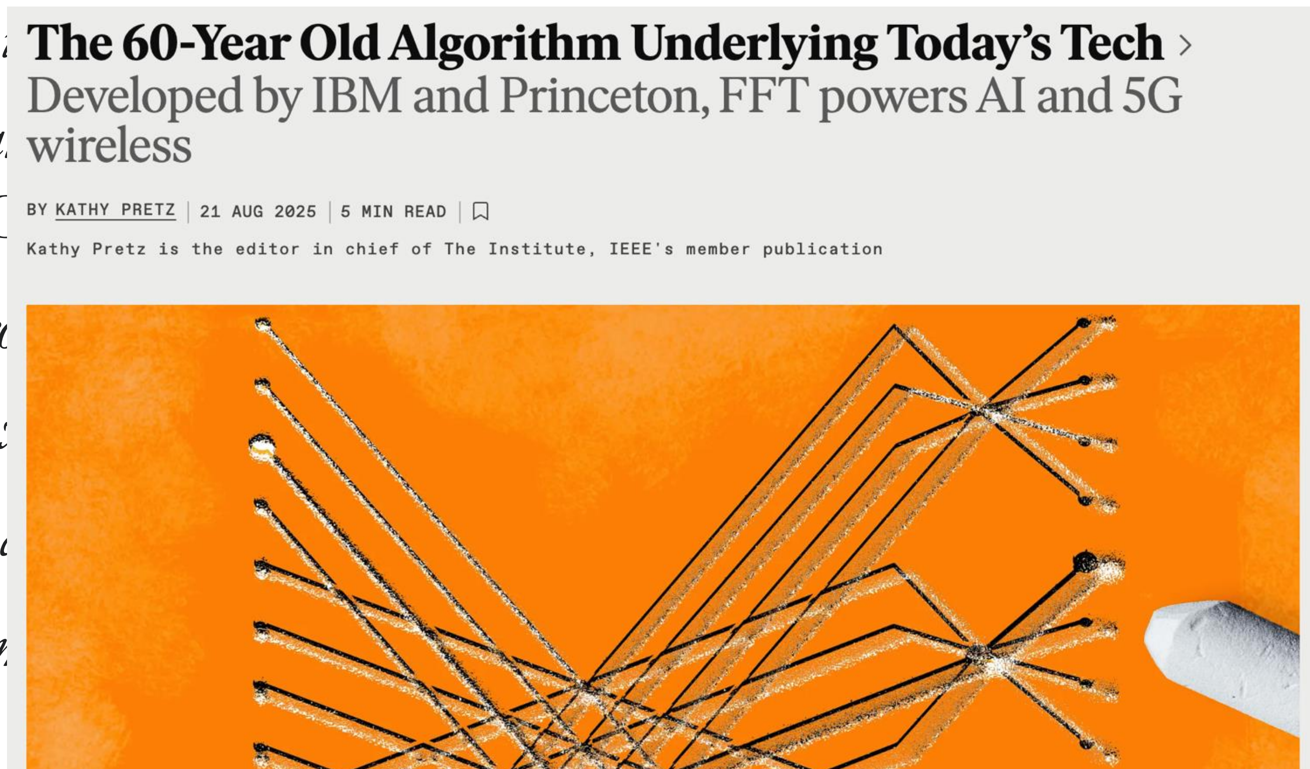
Signal pr

“multiple

MRI Im

Image co

Detecting



The 60-Year Old Algorithm Underlying Today’s Tech ›

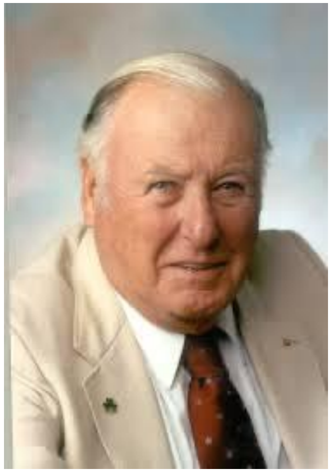
Developed by IBM and Princeton, FFT powers AI and 5G wireless

BY KATHY PRETZ | 21 AUG 2025 | 5 MIN READ

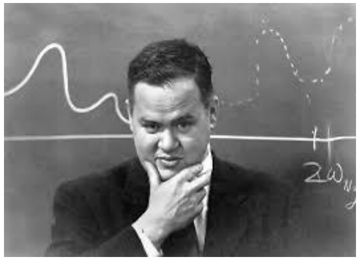
Kathy Pretz is the editor in chief of The Institute, IEEE's member publication

The *Fast Fourier Transform*

GAUSS!



COOLEY!



TUKEY!

NACHLASS.

THEORIA INTERPOLATIONIS

METHODO NOVA TRACTATA.

1.

PROBLEMA. *Invenire summam seriei*

$$\frac{a^n}{(a-b)(a-c)(a-d)(a-e)\ldots}+\frac{b^n}{(b-a)(b-c)(b-d)(b-e)\ldots}+\frac{c^n}{(c-a)(c-b)(c-d)(c-e)\ldots}+\frac{d^n}{(d-a)(d-b)(d-c)(d-e)\ldots}+\frac{e^n}{(e-a)(e-b)(e-c)(e-d)\ldots}+\text{etc.}$$

ubi a, b, c, d, e *sunt* m *quantitates diversae, atque* n *numerus integer quicunque positivus, negativus sive etiam* 0.

Solutio. Faciendo brevitatis caussa

An Algorithm for the Machine Calculation of Complex Fourier Series

By James W. Cooley and John W. Tukey

An efficient method for the calculation of the interactions of a 2^m factorial experiment was introduced by Yates and is widely known by his name. The generalization to 3^m was given by Box et al. [1]. Good [2] generalized these methods and gave elegant algorithms for which one class of applications is the calculation of Fourier series. In their full generality, Good's methods are applicable to certain problems in which one must multiply an N-vector by an $N \times N$ matrix which can be factored into m sparse matrices, where m is proportional to $\log N$. This results in a procedure requiring a number of operations proportional to $N \log N$ rather than N^2. These methods are applied here to the calculation of complex Fourier series. They are useful in situations where the number of data points is, or can be chosen to be, a highly composite number. The algorithm is here derived and presented in a rather different form. Attention is given to the choice of N. It is also shown how special advantage can be obtained in the use of a binary computer with $N = 2^m$ and how the entire calculation can be performed within the array of N data storage locations used for the given Fourier coefficients.

Consider the problem of calculating the complex Fourier series

$$X(j) = \sum_{k=0}^{N-1} A(k) \cdot W^{jk}, \qquad j = 0, 1, \cdots, N-1, \tag{1}$$

The *Fast Fourier Transform*

We will see it as part of a method to multiply two polynomials super fast.

"converts a $\underbrace{\text{sliding sum of products}}_{\text{Convolution}}$ into a single product"

(same operation that gives Convolutional neural networks their name)

But first…

"the tastiest dishes have a pinch (or even a handful) of an important ingredient"

Review: Polynomials

Definition: A polynomial of degree d is a function p of the form:

$$p(x) = c_d x^d + c_{d-1} x^{d-1} + \cdots + c_0$$

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

Proof: Suppose there are two distinct deg $\leq d$ polynomials p & p' s.t. $p(x_0) = p'(x_0)$
$\vdots$
$p(x_d) = p'(x_d)$.

Then $p - p'$ is a deg $\leq d$ polynomial that is zero on $d+1$ distinct points ← Contradicts fund. thm of algebra

Review: *Multiplying* 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

$$c_k = \sum_{0 \le i, j \le d : i+j=k} a_i b_j$$

convolution of sequences a & b

Review: Complex Numbers

Definition: **Complex numbers** are all numbers of the form $a + bi$ where i is a ***root*** of unity.

- $i^2 = -1$ (this is a definition!)
- **Fundamental fact**: every polynomial equation has a solution over the complex numbers! Not true over reals.

Roots of Unity: Magical Complex Numbers

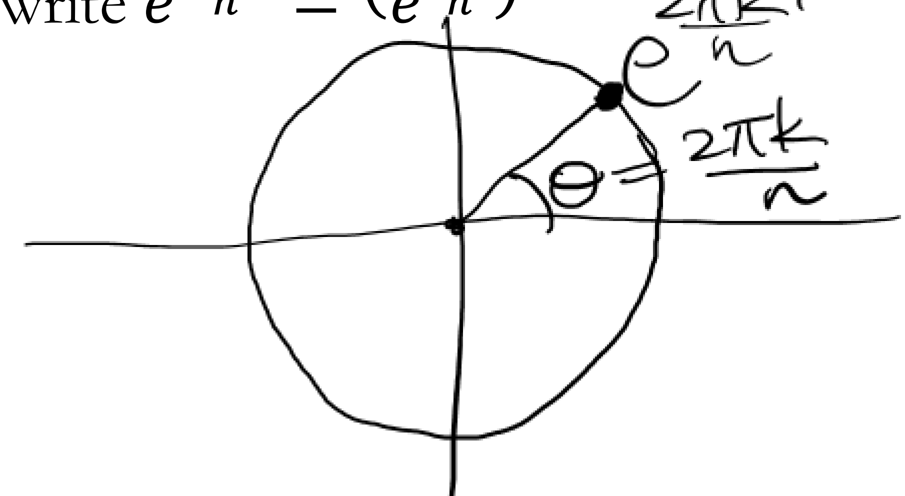
Definition: An n^{th} **root of unity** is an n^{th} root of 1, i.e.,

$$\omega^n = 1$$

- There are exactly n complex n^{th} roots of unity, given by $e^{\frac{2\pi i k}{n}}$.

$$= \cos\left(\frac{2\pi k}{n}\right) + i \sin\left(\frac{2\pi k}{n}\right)$$

- Can also write $e^{\frac{2\pi i k}{n}} = \left(e^{\frac{2\pi i}{n}}\right)^k$



Primitive Roots of Unity

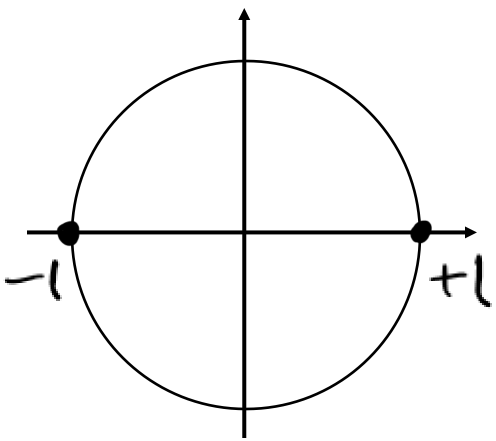
- The number $e^{\frac{2\pi i}{n}}$ is called a **primitive n^{th} root of unity**

- **Definition:** Formally, ω is a primitive n^{th} root of unity if

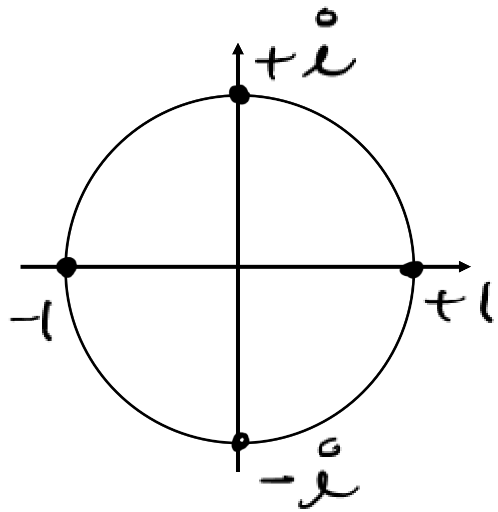
$$\omega^n = 1$$
$$\omega^k \neq 1 \quad \text{for} \quad 0 < k < n$$

"no smaller power of ω is 1".

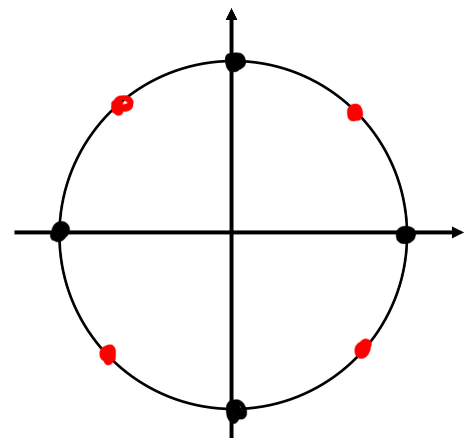
Primitive Roots of Unity



2nd roots of unity



4th roots of unity



8th 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?

Anatoly
Karatsuba

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

$\times \quad \times \quad \times \quad \times$

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

$\Downarrow$

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

Fast Polynomial Multiplication

1. Pick $N = 2d + 1$ points $x_0, x_1, \ldots, x_{N-1}$
2. *Evaluate* $A(x_0), A(x_1), A(x_2), \ldots, A(x_{N-1}), B(x_0), B(x_1), B(x_2), \ldots, B(x_{N-1})$
3. Compute $C(x_0), C(x_1), \ldots, C(x_{N-1})$
4. *Interpolate* $C(x_0), C(x_1), \ldots, C(x_{N-1})$ to get the coefficients of C

How do we do steps 2 and 4 efficier

FFT!

To Point-Value Form

- Consider the polynomial A of degree 7

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

- 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?

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

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

Let's divide and conquer!

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

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

    …and evaluate at $N/2$ points
  - Combine the two halves using the formula above

We might be in a pickle still…

- We need to evaluate the two "even" and "odd" polynomials on the *squares* of the N points to implement our plan.
- So, it seems like we need to evaluate the smaller degree polynomials at N points still… ☹
- **Idea:** choose a ***structured*** set of N evaluation points so that the squares of the points form a set of $\frac{N}{2}$ points…

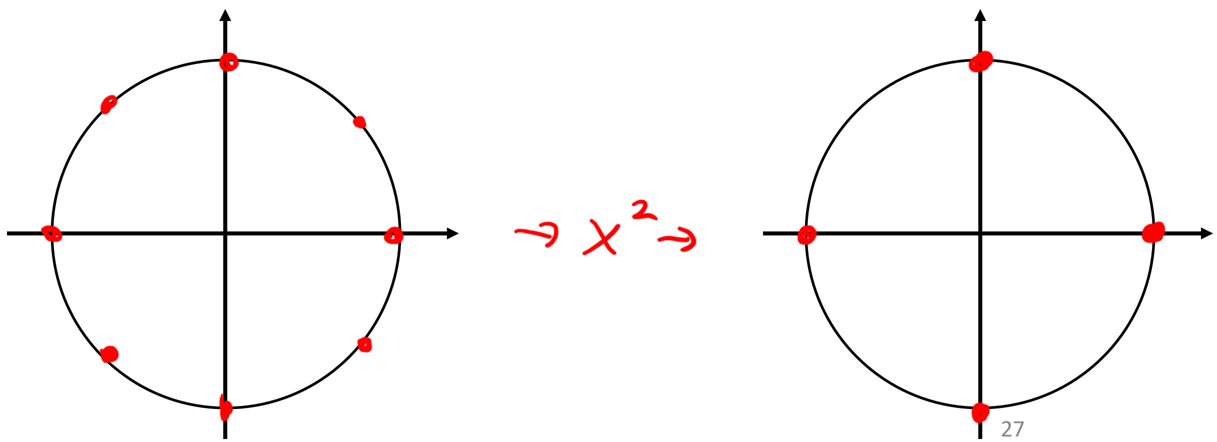
That should sound insane!

Roots of unity to the rescue!

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

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

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



Magical Idea 1: Suppose N is a power of 2. Squares of N-th roots of unity are (N/2)-th roots of unity!

Fast Fourier Transform: Coeff to Point-Value

- Assume N is a power of two (pad with zero coefficients)
- **Choose $x_0, x_1, \dots, x_{N-1}$ 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 $\omega^0, \omega^1, \omega^2, \dots, \omega^N$
  - Break into $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$ ← ***The $\left(\frac{N}{2}\right)$ th roots of unity!!!***
  - Evaluate those at $\omega^0, \omega^2, \omega^4, \dots$
  - Combine using $A(\omega^k) = A_{even}(\omega^{2k}) + \omega^k A_{odd}(\omega^{2k})$

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(** $[a_0, a_2, \ldots, \ldots], \omega^2, N/2)$

$F_{\text{odd}} \leftarrow$ **FFT(** $[a_1, a_3, \ldots], \omega^2, N/2)$

$x \leftarrow 1$ *// x stores* ω^k

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

$F[k] \leftarrow F_{\text{even}}[k \bmod \frac{N}{2}] + x \cdot F_{\text{odd}}[k \bmod \frac{N}{2}]$

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

$\}$ **return** F

$\}$

Back to multiplication

1. Pick $N = 2d + 1$ points $x_0, x_1, \dots, x_{N-1}$
2. *Evaluate* $A(x_0), A(x_1), A(x_2), \dots, A(x_{N-1}), B(x_0), B(x_1), B(x_2), \dots, B(x_{N-1})$
3. Compute $C(x_0), C(x_1), \dots, C(x_{N-1})$
4. *Interpolate* $C(x_0), C(x_1), \dots, C(x_{N-1})$ to get the coefficients of C

Inverse FFT: Point-Value to Coefficients

- Given $C(\omega_0), C(\omega_1), \ldots, C(\omega_{N-1})$ where $N = 2d + 1$
- We want to get the N coefficients of $C(x)$ back
- We're going to do it with…

…math!

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

$$(x^0, x^1, x^2, \ldots, x^{N-1}) \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{pmatrix} = p(x)$$

monomial vector

coeff vector

Inverse FFT

Observation: Evaluating a polynomial at a set of points can be represented as a *matrix-vector* product ⟶ Extremely important computational primitive)

Real life still of GPUs at an LLM startup working hard to compute matrix-vector products.

Inverse FFT

Observation: Evaluating a polynomial at a set of points can be represented as a *matrix-vector* product

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

We need to "invert" this operation. When can we do this?

Inverse FFT

Observation: Evaluating a polynomial at a set of points can be represented as a *matrix-vector* product

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

Alexander Théophile Vandermonde

Theorem: This matrix is invertible iff the x_i are distinct

Inverse FFT

- In our case, $x_k = \omega^k$ where ω is a primitive 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 $\left(\omega^k\right)^j = \omega^{kj}$

- Why are these numbers distinct?

Inverse FFT

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

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

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

if $j, k \le N-1$ & $\omega^k = \omega^j \Rightarrow \omega^{k-j} = 1$

- Why are these numbers distinct?

But ω is primitive! so $k-j=0$

Magical Idea 2: FFT Matrix is invertible on powers of a primitive N-th root of unity!

Inverse FFT

Magical Idea 3: Inverse of the FFT is an FFT on inverse eval points!

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

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

$$(AB)_{kj} = \sum_{s=0}^{N-1} a_{ks} \cdot b_{sj}$$

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

Inverse FFT

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

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

- How do the diagonal (i.e., $k \neq j$) entries of the product look?

$$\sum_{s=0}^{N-1} \omega^{-js} \cdot \omega^{sj} = \sum_{s=0}^{N-1} 1 = \underline{\underline{N}}$$

Inverse FFT

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

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

- How do the off-diagonal (i.e., $k \neq j$) entries of the product look?

$$\sum_{s=0}^{N-1} \omega^{(j-k)s} = \sum_{s=0}^{N-1} \left(\omega^{(j-k)}\right)^{s} = \frac{1-\left(\omega^{j-k}\right)^{N}}{1-\omega^{j-k}} = \frac{1-(\overbrace{\omega^{N}}^{=1})^{j-k}}{1-\omega^{j-k}} = \frac{1-1}{1-\omega^{j-k}} = 0.$$

Geometric Sum

Sum of GP

Inverse FFT

- 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

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

Back to multiplication

1. Pick $N = 2d + 1$ points $x_0, x_1, \ldots, x_{N-1}$ $\nearrow$ $O(N \cdot \log N)$
2. *Evaluate* $A(x_0), A(x_1), A(x_2), \ldots, A(x_{N-1})$, $B(x_0), B(x_1), B(x_2), \ldots, B(x_{N-1})$
3. Compute $C(x_0), C(x_1), \ldots, C(x_{N-1})$ $\rightarrow O(N)$
4. *Interpolate* $C(x_0), C(x_1), \ldots, C(x_{N-1})$ to get the coefficients of C

$\rightarrow O(N \log N)$

Running Time: $O(N \log N) = O(d \log d)$

The Magic of FFT

- Switch between coefficient & point-value representations in O(n log n) time!
- **Idea 1**: Divide and Conquer
- **Magic 1:** *Needed a set of points such that taking their squares shrinks the set by half – roots of unity!*
- **Idea 2:** *Invert* the Point-Value representation of the product. Interpret FFT as matrix-vector product.
- **Magic 2:** *Needed the FFT matrix to be invertible. Vandermonde shows Matrix invertible iff eval points distinct.*
- **Idea 3:** Compute the inverse-matrix-vector product to recover coeff representation.
- **Magic 3:** *The inverse matrix is also an FFT just at the inverses of the original eval points.*

Takeaways

FFT is super cool!