Lecture 9

Multiplication

Chinmay Nirkhe | CSE 421 Spring 2025

The next couple of weeks

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	4/21 Lecture 10	Sets 4 & 4 ¾ released	4/23 Lecture 11 Set 3 due		4/25 Lecture 12	
	4/28 Lecture 13		4/30 Lecture 14 Set 4 due	Midterm Q&A 5:30-7:30pm	5/2 Lecture 15	
	5/5 **Midterm**	Set 5 released	5/7 Lecture 16 Set 4 ¾ due		5/9 Lecture 17	
	5/12 Lecture 18		5/14 Lecture 19 Set 5 due		5/16 Lecture 20	

Problem set 4 ¾

- A set with one 10 point question
  - The problem is about *dynamic programming*
  - Dynamic programming <u>is</u> covered on the midterm
- It is due on Wednesday May 7th 11:59pm
  - But, I’m posting solutions on Saturday May 3rd (12:01am) before its due
  - You can look at the solution <u>after</u> you upload your solution to Gradescope
  - Not doing so is academic dishonesty. I’m trusting each of you here

Previously in CSE 421…

Principles of divide and conquer

- Identity a division of the problem into a self-similar parts of size n/b
- Recursively solve each subpart of the problem
- Stitch the solutions from each subpart together

- Runtime is defined by the following recursively defined formula:

$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ and $T(< b) = O(1)$

Analysis divide and conquer runtimes

The master theorem

- For solving recursive equations of the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^k) \text{ and } T(<b) = O(1)$$

- Different cases based on how $f(n)$, a, and b compare:
    - If $a < b^k$, then $T(n) = O(n^k)$
    - If $a = b^k$, then $T(n) = O(n^k \log n)$
    - If $a > b^k$, then $T(n) = O(n^{\log_b a})$

Today: Matrix, integer, and polynomial multiplication

Matrix multiplication

- **Input**: Two matrices $A, B \in \mathbb{R}^{n \times n}$
- **Output**: The matrix $AB \in \mathbb{R}^{n \times n}$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

where

$$c_{ij} = \sum_k a_{ik} b_{kj}.$$

Trivial algorithm for matrix multiplication

- **Algorithm**:
  - Initialize $n \times n$ array C as zeroes
  - For $i \in [n], j \in [n], k \in [n]$, $\quad C_{ij} \leftarrow C_{ij} + A_{ik} \cdot B_{kj}$
  - Return C.

- **Runtime**: n^3 multiplications + n^3 additions
- Can we improve this with divide and conquer?

Matrix multiplication naturally decomposes

- **Matrix multiplication of matrices**

terms do not commute

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

- **Divide and conquer:**
  - Decompose into 8 matrix multiplications of $n/2 \times n/2$ matrices and 4 matrix additions of $n/2 \times n/2$ matrices
  - $T(n) = 8T\left(\frac{n}{2}\right) + 4\left(\frac{n}{2}\right)^2 \implies T(n) = O(n^{\log_2 8}) = O(n^3)$

$a = 8$
$b = 2$
$k = 2$

$a > b^k$

leaf-heavy computation

Strassen's divide and conquer (1968)

- Can we decrease the number of mini-multiplications at the cost of increasing the number of mini-additions?
- If we were to somehow decrease to 7 multiplications but 18 additions …
- $T(n) = 7T\left(\frac{n}{2}\right) + \frac{18}{4}n^2 \implies T(n) = \frac{18}{4} \cdot O(n^{\log_2 7}) = O(n^{2.8074})$

$a = 7$ $\quad$ $a > b^k$ but

$b = 2$ $\quad$ $\log_b a$ is smaller…

$k = 2$

- But how do we achieve this decrease?
- **Find repeated terms**.

A clever decomposition

We know that if

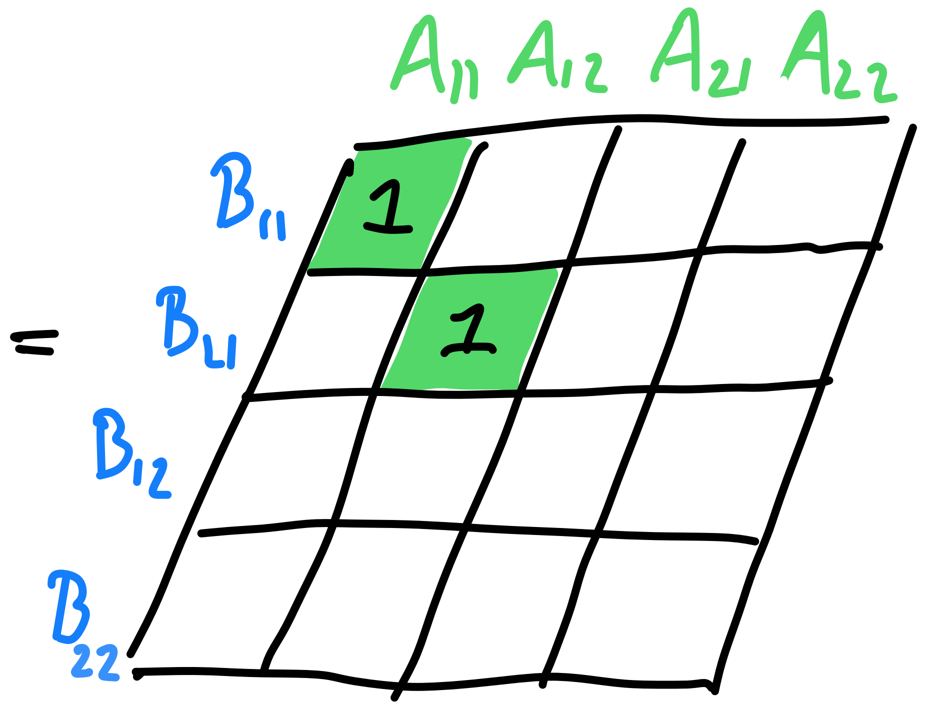
$$A \cdot B = C$$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

then $C_{11} = A_{11}B_{11} + A_{12}B_{21}$.

Pictorally, let's represent this fact by

$C_{11} =$



A clever decomposition

Similarly,

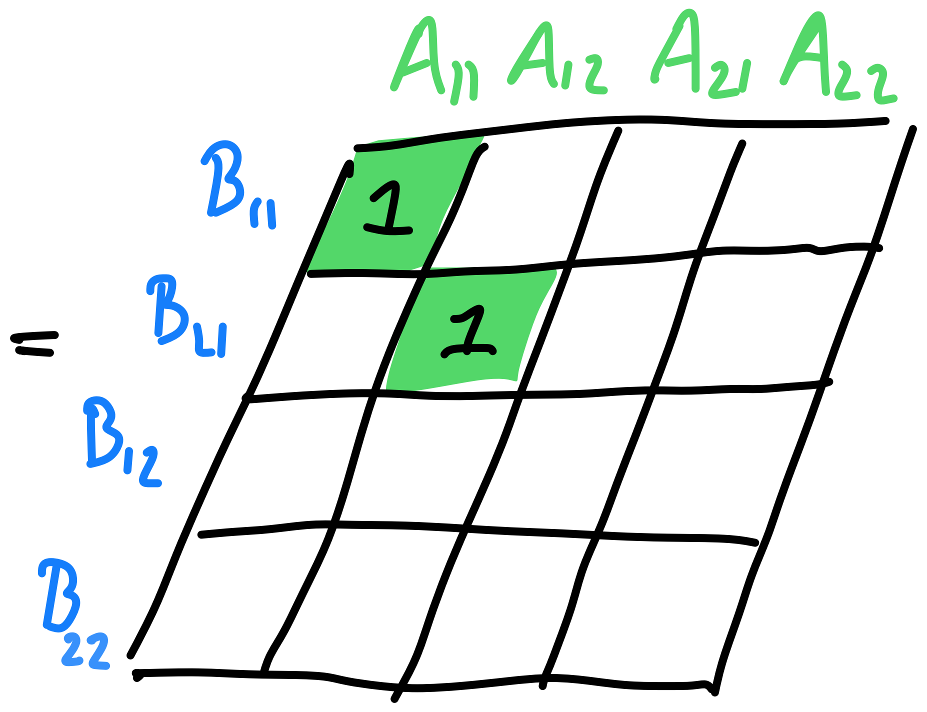
$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

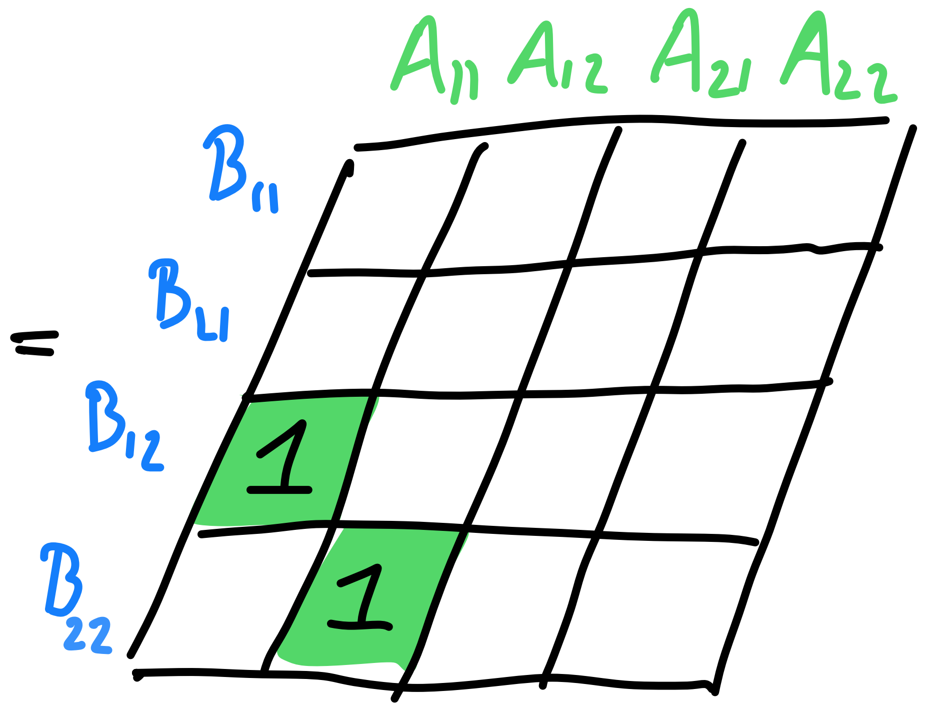
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

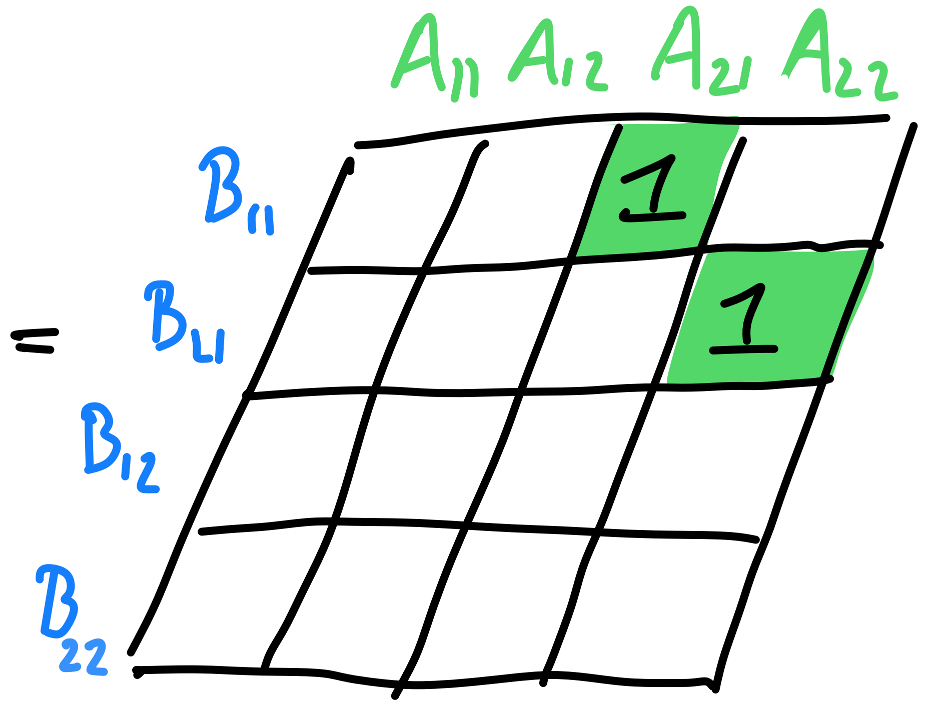
$C_{11} =$



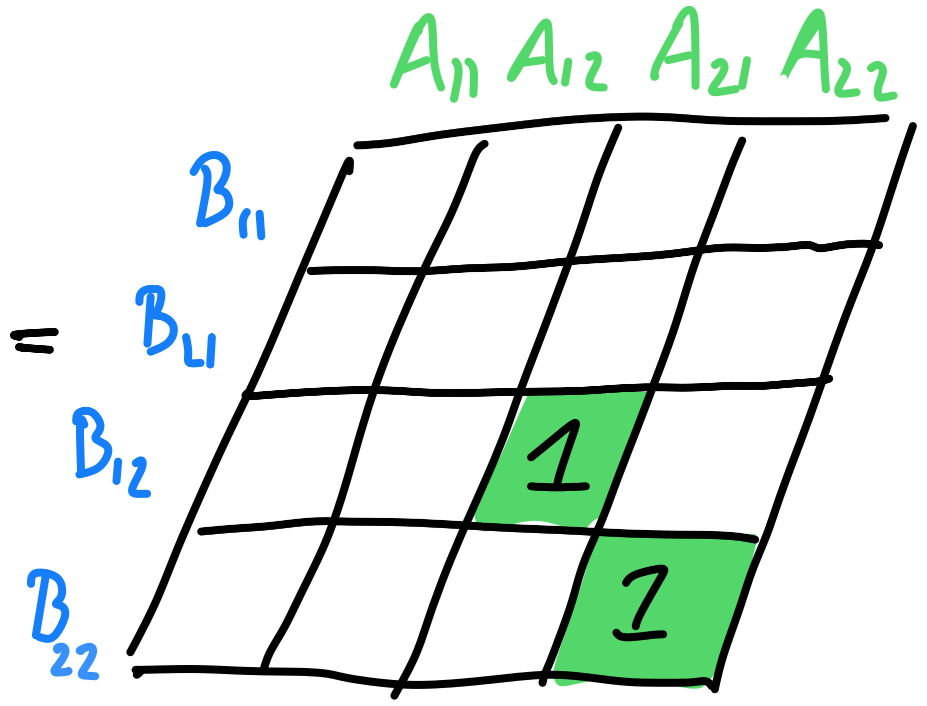
$C_{12} =$



$C_{21} =$



$C_{22} =$



A clever decomposition

Now, what happens if we want to calculate

$$M = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22}$$

$$M = \begin{array}{c|cccc} & A_{11} & A_{12} & A_{21} & A_{22} \\ \hline B_{11} & 1 & & & 1 \\ B_{21} & & & & \\ B_{12} & & & & \\ B_{22} & 1 & & & 1 \end{array}$$

A clever decomposition

Another example...

$$M' = A_{11}(B_{12} - B_{22})$$

$$= A_{11}B_{12} - A_{11}B_{12}$$

$$M' = \begin{array}{c|cccc} & A_{11} & A_{12} & A_{21} & A_{22} \\ \hline B_{11} & & & & \\ B_{21} & & & & \\ B_{12} & 1 & & & \\ B_{22} & -1 & & & \end{array}$$

A clever decomposition

We can add these diagrams...

$$M = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M' = A_{11}(B_{12} - B_{22})$$

$$M = \begin{array}{c|cccc} & A_{11} & A_{12} & A_{21} & A_{22} \\ \hline B_{11} & 1 & & & 1 \\ B_{21} & & & & \\ B_{12} & & & & \\ B_{22} & 1 & & & 1 \end{array}$$

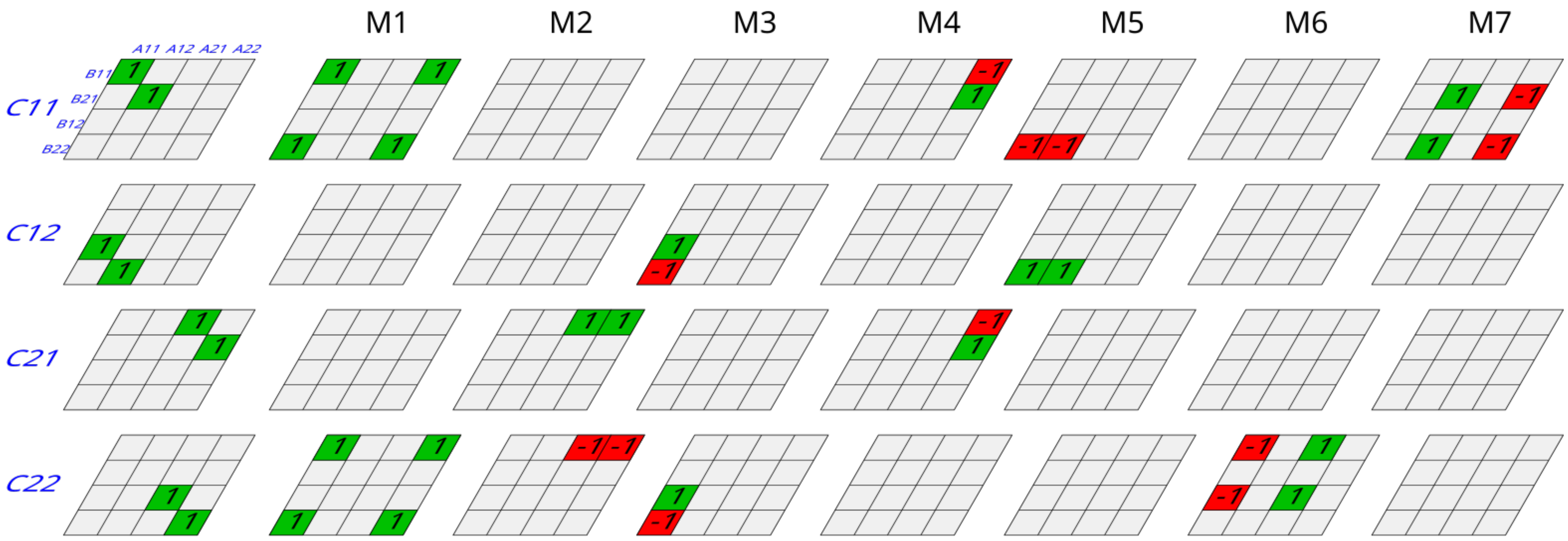
$$M' = \begin{array}{c|cccc} & A_{11} & A_{12} & A_{21} & A_{22} \\ \hline B_{11} & & & & \\ B_{21} & & & & \\ B_{12} & 1 & & & \\ B_{22} & -1 & & & \end{array}$$

$$M + M' = \begin{array}{c|cccc} & A_{11} & A_{12} & A_{21} & A_{22} \\ \hline B_{11} & 1 & & & 1 \\ B_{21} & & & & \\ B_{12} & 1 & & & \\ B_{22} & & & & 1 \end{array}$$

A clever decomposition

7 multiplications + 10 additions

M1	M2	M3	M4	M5	M6	M7
1 mult + 2 additions	1 mult + 1 addition	1 mult + 1 addition	1 mult + 1 addition	1 mult + 1 addition	1 mult + 2 addition	1 mult + 2 additions



$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 - M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

8 additions

Wikipedia article for Strassen's algorithm

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{12})$$

$$M_5 = (A_{11} + A_{12})B_{22}$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

Strassen's algorithm details

- Best for matrices of size $2^m \times 2^m$. Pad the matrix with zeroes until it is.
- Strassen's has 18 mini-additions. Only beneficial if $n \geq 32$.
  - For smaller matrices, use $O(n^3)$ algorithm.
  - Still a base case for the recursive definition. Only adjust $O(\cdot)$ constants.
- Is there an even cleverer decomposition into fewer mini-multiplications?
  - Not for dividing into $n/2 \times n/2$ mini-matrices
  - Other divisions plus clever tricks have gotten algorithms down to $O(n^{2.371339})$ [May 2024]
  - **Major open question**: $O(n^{2+\epsilon})$ time algorithm possible for all $\epsilon > 0$.

Integer multiplication

- **Input:** Two n-bit binary numbers $x, y \in \{0,\ldots,2^n - 1\}$
- **Output:** A $2n$-bit binary number

- Complexity is <u>not</u> measured in RAM model!
- Instead by number of binary operations required.

$$
\begin{array}{cccccccc}
 & & & & 1 & 0 & 0 & 1 \\
 & \times & & & 1 & 1 & 0 & 1 \\
\hline
 & & & & 1 & 0 & 0 & 1 \\
 & & & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 1 & 0 & 0 \\
+ & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & 1 & 1 & 1 & 0 & 1 & 0 & 1
\end{array}
$$

- Gradeschool multiplication algorithm takes $O(n^2)$ time

The Karatsuba method

$\boxed{x_1 \mid x_0} \times \boxed{y_1 \mid y_0}$

$= \left(2^{n/2} x_1 + x_0\right)\left(2^{n/2} y_1 + y_0\right)$

$= 2^n x_1 y_1 + 2^{n/2}\left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 .$

$= 2^n\left(\boxed{x_1} \times \boxed{y_1}\right) + 2^{n/2}\left(\boxed{x_1} \times \boxed{y_0} + \boxed{x_0} \times \boxed{y_1}\right) + \boxed{x_0} \times \boxed{y_0} .$

left shifts

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n) \implies T(n) = O(n^{\log_2 4}) = O(n^2)$$

no improvements.

The Karatsuba method

$$\boxed{x_1 \mid x_0} \times \boxed{y_1 \mid y_0}$$

$$= \left(2^{n/2} x_1 + x_0\right)\left(2^{n/2} y_1 + y_0\right)$$

$$= 2^n x_1 y_1 + 2^{n/2}\left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 .$$

$$= 2^n x_1 y_1 + 2^{n/2}\left((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0\right) + x_0 y_0 .$$

Identify 3 multiplications of size $\frac{n}{2}$

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) \implies T(n) = O(n^{\log_2 3}) = O(n^{1.58})$$

Improving integer multiplication

- Fast integer multiplication is used in high-precision arithmetic
- Storing a number to n-bits of precision is equal to 2^{-n} precision
- Karatsuba's algorithm is not the fastest
  - Fastest is $O(n \log n)$ based on the fast Fourier transform (next!)
  - These are galactic algorithms (not useful in practice)

Polynomial multiplication

- **Input:** polynomials $p, q \in \mathbb{C}[x]$ of deg $< n$ expressed by their coefficients

i.e.
$$\begin{aligned} p(x) &= a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0 \\ q(x) &= b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \ldots + b_1 x + b_0 \end{aligned}$$

- **Output:** The poly. $pq \in \mathbb{C}[x]$ of deg $< 2n - 1$ expressed in coefficients.

$$pq(x) = c_{2n-1}x^{2n-1} + c_{2n-2}x^{2n-2} + \ldots + c_1 x + c_0 \text{ where } c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

Why is polynomial multiplication useful?

- This algorithm is used as a subroutine in
  - signal processing, image processing, audio compression
  - Many public-key cryptosystems rely on polynomial arithmetic
  - Computing Reed-Solomon (5G) error-correcting codes
  - Polynomial-based error-detection codes
- The major subroutine is equivalent to **convolution**, a fundamental mathematical computation

Polynomial multiplication

- **Output:** The poly. $pq \in \mathbb{C}[x]$ of deg $< 2n-1$ expressed in coefficients.

$$pq(x) = c_{2n-1}x^{2n-1} + c_{2n-2}x^{2n-2} + \ldots + c_1 x + c_0 \text{ where } c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

- Can be solved using $1 + 2 + \ldots + n + \ldots + 2 + 1 = O(n^2)$ multiplications
- Can we be any faster? Perhaps using divide and conquer?

When is polynomial multiplication fast?

- **Fundamental theorem of algebra**: A degree $< n$ polynomial p is **uniquely** specified by *any* n distinct evaluation points.
  - Let $\xi_1, \xi_2, \ldots, \xi_n \in \mathbb{C}$ be distinct. Then $\{(\xi_i, p(\xi_i))\}$ uniquely define p.
- Let p be the poly defined by $\{(\xi_i, y_i)\}$.
  Let q be the poly defined by $\{(\xi_i, z_i)\}$.
  For every x, $(pq)(x) = p(x) \cdot q(x)$. So, $(pq)(\xi_i) = y_i \cdot z_i$.
- Then pq is a poly defined by $\{(\xi_i, y_i \cdot z_i)\}$.

degree $< 2n$ poly

only n interpolation points.

When is polynomial multiplication fast?

- **Fundamental theorem of algebra:** A degree $< n$ polynomial p is **uniquely** specified by *any* n distinct evaluation points.
    - Let $\xi_1, \xi_2, \ldots, \xi_{2n} \in \mathbb{C}$ be distinct. Then $\{(\xi_i, p(\xi_i))\}_{i\in[2n]}$ uniquely define p.
- Let p be the poly defined by $\{(\xi_i, y_i)\}_{i\in[2n]}$.
  Let q be the poly defined by $\{(\xi_i, z_i)\}_{i\in[2n]}$.
- Then pq is the unique poly defined by $\{(\xi_i, y_i \cdot z_i)\}_{i\in[2n]}$.

Polynomial multiplication algorithm

compute $\xi_i^1, \xi_i^2, \xi_i^3, \ldots \xi_i^n$ in time $O(n)$

compute $p(\xi_i) = \sum_{j=0}^{n} a_j \xi_i^j$ in time $O(n)$.

- **New algorithm**:
  - Pick evaluation points $\xi_1, \ldots, \xi_{2n}$.
  - Evaluate p and q to compute $y_i \leftarrow p(\xi_i), z_i \leftarrow q(\xi_i)$ for $i \in [2n]$. total time $O(n^2)$.
  - Calculate $w_i \leftarrow y_i \cdot z_i$. $\leftarrow$ $O(n)$ total time.
  - Compute the coefficients of polynomial uniquely defined by $\{(\xi_i, w_i)\}_{i \in [2n]}$.

we still haven't discussed how to do this.

this is prohibitive. We need to do this faster to improve on naïve alg.

Polynomial multiplication algorithm

- **New idea:** Pick interpolation points $\{\xi_j\}$ intelligently.
  - If done correctly, we can speed up computing $p(\xi_j), q(\xi_j)$.
  - Also will give us a way of un-doing interpolation after multiplication.
- Choose related points to parallelize the computation
- Writing down all the partial computations will take too long

Change of basis

- Let's observe that $p(\xi) = \sum_{j=0}^{2n} a_j \xi^j$ is the **inner product** between two vectors.

- $$p(\xi) = \underbrace{\begin{bmatrix} 1 & \xi & \xi^2 & \dots & \xi^{2n} \end{bmatrix}}_{\text{only depends on } \xi \text{ and not polynomial coefficients.}} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{2n} \end{bmatrix}$$

set $a_{n+1} = \dots = a_{2n} = 0$

This only depends on the polynomial coefficients and not ξ.

Change of basis

$$\begin{bmatrix} p(\xi_1) \\ p(\xi_2) \\ \vdots \\ p(\xi_{2n}) \end{bmatrix} = \overbrace{\begin{bmatrix} 1 & \xi_1 & \xi_1^2 & \dots & \xi_1^{2n} \\ 1 & \xi_2 & \xi_2^2 & \dots & \xi_2^{2n} \\ \vdots & & & \ddots & \vdots \\ 1 & \xi_{2n} & \xi_{2n}^2 & \dots & \xi_{2n}^{2n} \end{bmatrix}}^{V} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{2n} \end{bmatrix}$$

special type of matrix called

a Vandermonde matrix

$$\det V = \prod_{i \neq j} (\xi_i - \xi_j) \leftarrow$$

non-zero iff ξ_i are distinct.

if $\det V$ is non-zero, V is invertible.

Polynomial multiplication algorithm

- **New new algorithm**:
  - Pick interpolation points $\{\xi_j\}_{j\in[2n]}$.
  - Compute Vandermonde matrix V and it's inverse V^{-1}.
  - Compute $\vec{y} \leftarrow V\vec{a}, \vec{z} \leftarrow V\vec{b}$.
  - Compute $\overrightarrow{w} \leftarrow \vec{a} \odot \vec{b}$, point-wise multiplication.
  - Return $V^{-1}\overrightarrow{w}$.

Still too slow if we are spending the time writing out V and V^{-1}. Need to do the $V\vec{a}$, $V\vec{b}$, $V^{-1}\vec{w}$ computations "in place".

Choice of interpolation points

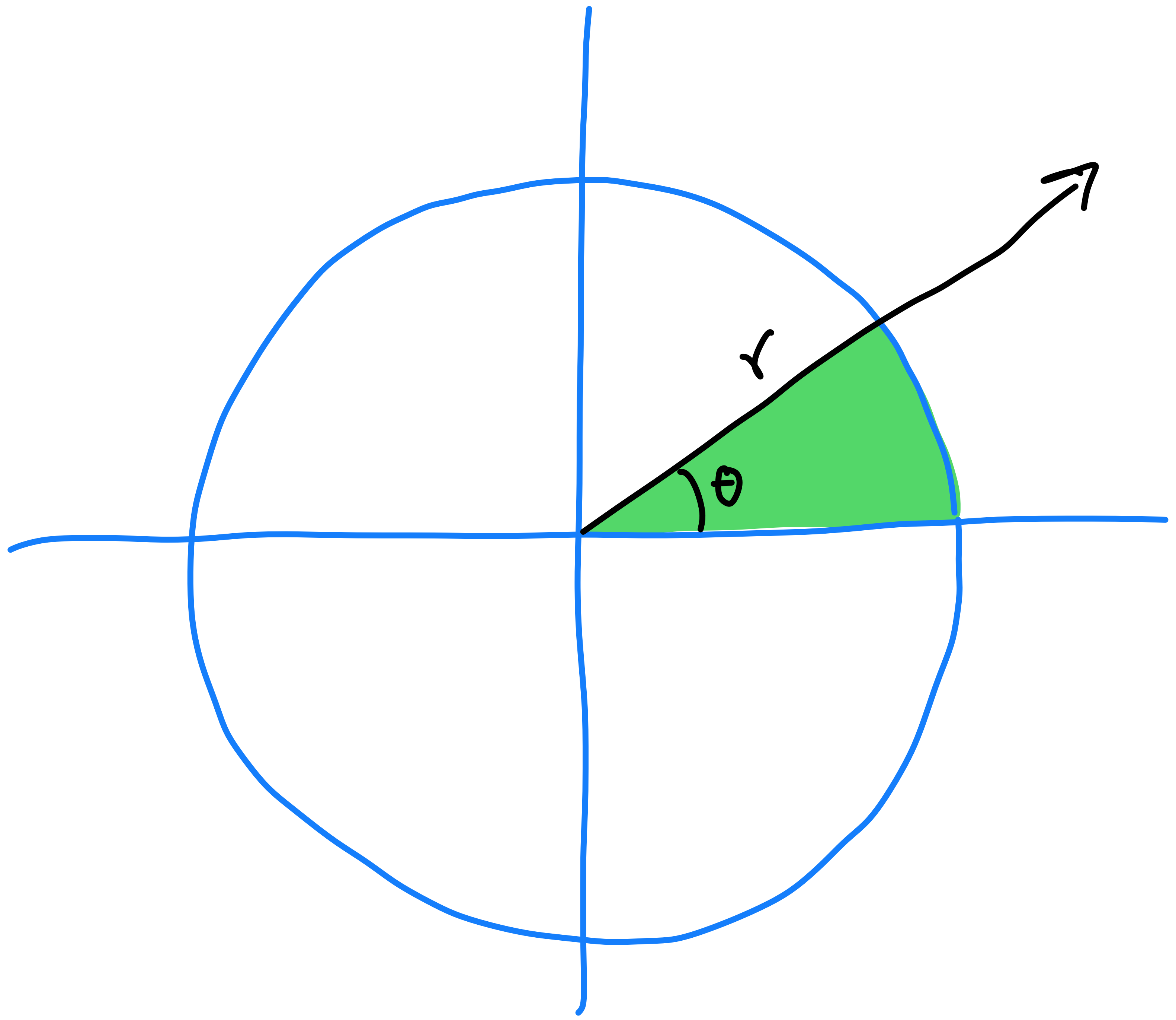
- To speed up the algorithm, we are going to pick $\{\xi_j\}$ creatively.
- Let m be the smallest power of 2 that is $\geq 2n$.
- Choose $\omega := e^{2\pi i/m}$, a primitive m-th root of unity.
- Define $\xi_j = \omega^{j-1}$.

$$V = \underbrace{\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{m-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(m-1)} \\ 1 & \omega^3 & \omega^6 & \dots & \omega^{3(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{m-1} & \omega^{2(m-1)} & \dots & \omega^{(m-1)^2} \end{bmatrix}}_{\text{known as } F_m(\omega)\text{, the Fourier transform over } \mathbb{Z}/m.}$$

Complex number review

- A complex number can be expressed as $a + bi$ with $a, b \in \mathbb{R}$
- Alternatively, it can be expressed as $re^{i\theta}$ with $r \in \mathbb{R}^{\geq 0}, \theta \in [0,2\pi)$
- Multiplying complex numbers is easy

$$((r_i e^{i\theta_1})(r_2 e^{i\theta_2}) = (r_1 r_2) e^{i(\theta_1 + \theta_2)})$$



Complex number review

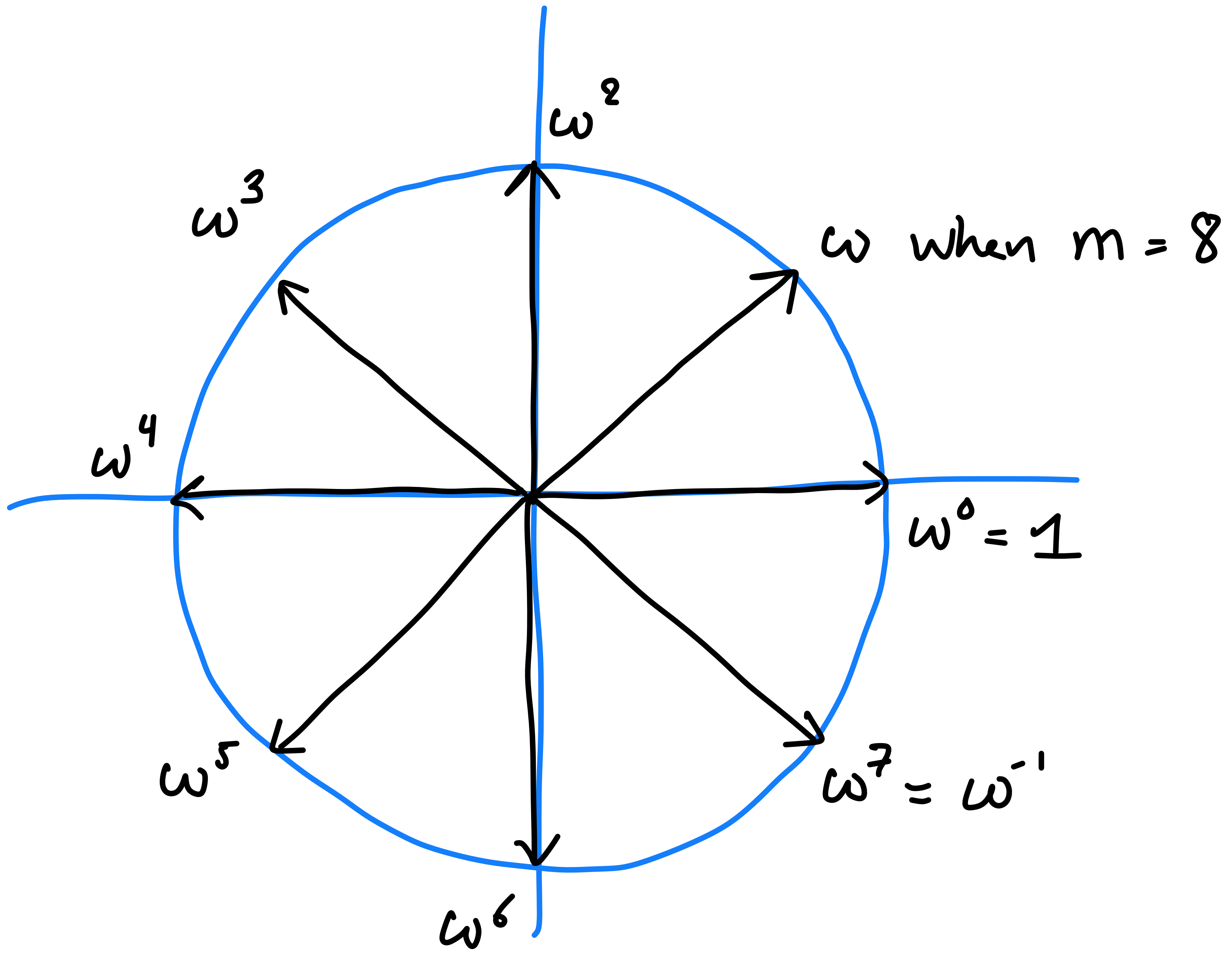
- A complex number can be expressed as $a + bi$ with $a, b \in \mathbb{R}$

- Alternatively, it can be expressed as $re^{i\theta}$ with $r \in \mathbb{R}^{\geq 0}, \theta \in [0,2\pi)$

- Multiplying complex numbers is easy

$$((r_i e^{i\theta_1})(r_2 e^{i\theta_2}) = (r_1 r_2) e^{i(\theta_1 + \theta_2)})$$

- ω, the m-th root of unity can be expressed as $r = 1, \theta = 2\pi/m$



Polynomial multiplication algorithm

- New new new algorithm:
  - Let m be the smallest power of 2 that is $\geq 2n$. $\omega = e^{2\pi i/m}$.
  - Pad vectors $\vec{a}$ and $\vec{b}$ with zeroes till length m.
  - Compute $\vec{y} \leftarrow F_m(\omega)\vec{a}$, $\vec{z} \leftarrow F_m(\omega)\vec{b}$.
  - Compute $\vec{w} \leftarrow \vec{a} \odot \vec{b}$, point-wise multiplication.
  - Return $F_m(\omega)^{-1}\vec{w}$.

Theorem: $F_m(\omega)^{-1} = \frac{1}{m} F_m(\omega^{-1})$.

Polynomial multiplication algorithm

- **New new new algorithm**:
  - Let m be the smallest power of 2 that is $\geq 2n$. $\omega = e^{2\pi i/m}$.
  - Pad vectors $\vec{a}$ and $\vec{b}$ with zeroes till length m.
  - Compute $\vec{y} \leftarrow F_m(\omega)\vec{a}, \vec{z} \leftarrow F_m(\omega)\vec{b}$.
  - Compute $\overrightarrow{w} \leftarrow \vec{a} \odot \vec{b}$, point-wise multiplication.
  - Return $\frac{1}{m}F_m(\omega^{-1})\overrightarrow{w}$.

Computing $F_m(\omega)\vec{a}$ efficiently

- Goal is to use **divide and conquer** to do this computation efficiently
- To do so, we need to find similar components to break the problem into smaller parts
- Let's analyze this for $m = 8$ and then generalize.

$$\begin{bmatrix} p(1) \\ p(\omega) \\ p(\omega^2) \\ p(\omega^3) \\ p(\omega^4) \\ p(\omega^5) \\ p(\omega^6) \\ p(\omega^7) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}$$

Computing $F_m(\omega)\vec{a}$ efficiently

- Nothing says we have to calculate the evaluations in this order!
- Is there a better order in which a pattern emerges?

$$\begin{bmatrix} p(1) \\ p(\omega) \\ p(\omega^2) \\ p(\omega^3) \\ p(\omega^4) \\ p(\omega^5) \\ p(\omega^6) \\ p(\omega^7) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}$$

Computing $F_m(\omega)\vec{a}$ efficiently

- Nothing says we have to calculate the evaluations in this order!
- Is there a better order in which a pattern emerges?
- Even rows then odd rows

$$\begin{bmatrix} p(1) \\ p(\omega^2) \\ p(\omega^4) \\ p(\omega^6) \\ p(\omega) \\ p(\omega^3) \\ p(\omega^5) \\ p(\omega^7) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ a_6 \\ a_1 \\ a_3 \\ a_5 \\ a_7 \end{bmatrix}$$

Computing $F_m(\omega)\vec{a}$ efficiently

- Nothing says we have to calculate the evaluations in this order!
- Is there a better order in which a pattern emerges?
- Even rows then odd rows

$$\begin{bmatrix} p(1) \\ p(\omega^2) \\ p(\omega^4) \\ p(\omega^6) \\ p(\omega) \\ p(\omega^3) \\ p(\omega^5) \\ p(\omega^7) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ a_6 \\ a_1 \\ a_3 \\ a_5 \\ a_7 \end{bmatrix}$$

$$\begin{bmatrix} F_{\frac{m}{2}}(\omega^2) & F_{\frac{m}{2}}(\omega^2) \\ D\, F_{\frac{m}{2}}(\omega^2) & D'\, F_{\frac{m}{2}}(\omega^2) \end{bmatrix}$$

Computing $F_m(\omega)\vec{a}$ efficiently

$$\begin{bmatrix} 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^3 & \omega^6 & \omega^1 \\ 1 & \omega^5 & \omega^2 & \omega^7 \\ 1 & \omega^7 & \omega^6 & \omega^5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega & 0 & 0 \\ 0 & 0 & \omega^2 & 0 \\ 0 & 0 & 0 & \omega^3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \end{bmatrix}$$

Computing $F_m(\omega)\vec{a}$ efficiently

$$\begin{bmatrix} 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^3 & \omega^6 & \omega^1 \\ 1 & \omega^5 & \omega^2 & \omega^7 \\ 1 & \omega^7 & \omega^6 & \omega^5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega & 0 & 0 \\ 0 & 0 & \omega^2 & 0 \\ 0 & 0 & 0 & \omega^3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \end{bmatrix}$$

$$D\,F_{\frac{m}{2}}(\omega^2) \qquad D \qquad F_{\frac{m}{2}}(\omega^2)$$

Computing $F_m(\omega)\vec{a}$ efficiently

$$\begin{bmatrix} p_{even} \\ \hline p_{odd} \end{bmatrix} = \left[\begin{array}{c|c} \mathbb{1} & \mathbb{1} \\ \hline D & D' \end{array}\right] \left[\begin{array}{c|c} F_{\frac{m}{2}}(\omega^2) & \\ \hline & F_{\frac{m}{2}}(\omega^2) \end{array}\right] \begin{bmatrix} a_{even} \\ \hline a_{odd} \end{bmatrix}$$

where $D = \begin{bmatrix} 1 & & & & \\ & \omega & & & \\ & & \omega^2 & & \\ & & & \ddots & \\ & & & & \omega^{m/2-1} \end{bmatrix}$, $D' = \begin{bmatrix} \omega^{m/2} & & & \\ & \omega^{m/2+1} & & \\ & & \ddots & \\ & & & \omega^{m-1} \end{bmatrix}$.

Computing $F_m(\omega)\vec{a}$ efficiently

- **Divide and conquer algorithm**:
  - Split $\vec{a}$ into $\vec{a}_{\text{even}}$ and $\vec{a}_{\text{odd}}$ coordinates.
  - Compute $\vec{y}_{\text{even}} \leftarrow F_{m/2}(\omega^2)\vec{a}_{\text{even}}$ and $\vec{y}_{\text{odd}} \leftarrow F_{m/2}(\omega^2)\vec{a}_{\text{odd}}$.
  - Compute $D\vec{y}_{\text{even}}$ and $D'\vec{y}_{\text{odd}}$.
  - Compute $\begin{bmatrix} \vec{y}_{\text{even}} + \vec{y}_{\text{odd}} \\ D\vec{y}_{\text{even}} + \overrightarrow{D}'y_{\text{odd}} \end{bmatrix}$.
  - Rearrange coordinates to original format.

$$\begin{bmatrix} | \\ p_{even} \\ | \\ \hline | \\ p_{odd} \\ | \end{bmatrix} = \left[\begin{array}{c|c} \mathbb{1} & \mathbb{1} \\ \hline D & D' \end{array}\right] \underbrace{\left[\begin{array}{c|c} F_{\frac{m}{2}}(\omega^2) & \\ \hline & F_{\frac{m}{2}}(\omega^2) \end{array}\right] \begin{bmatrix} | \\ a_{even} \\ | \\ \hline | \\ a_{odd} \\ | \end{bmatrix}}_{\begin{bmatrix} | \\ y_{even} \\ | \\ \hline | \\ y_{odd} \\ | \end{bmatrix}}$$

Computing $F_m(\omega)\vec{a}$ efficiently

Total time: $T(m) = 2T\left(\frac{m}{2}\right) + O(m)$

$T(m) = O(m \log m)$.

- **Divide and conquer algorithm**:
  - Split $\vec{a}$ into $\vec{a}_{\text{even}}$ and $\vec{a}_{\text{odd}}$ coordinates. ← $O(m)$ time.
  - Compute $\vec{y}_{\text{even}} \leftarrow F_{m/2}(\omega^2)\vec{a}_{\text{even}}$ and $\vec{y}_{\text{odd}} \leftarrow F_{m/2}(\omega^2)\vec{a}_{\text{odd}}$. ← Recursion: $2T\left(\frac{m}{2}\right)$ time.
  - Compute $D\vec{y}_{\text{even}}$ and $D'\vec{y}_{\text{odd}}$. ← D, D' diagonal so $O(m)$ time.
  - Compute $\begin{bmatrix} \vec{y}_{\text{even}} + \vec{y}_{\text{odd}} \\ D\vec{y}_{\text{even}} + \vec{D}'y_{\text{odd}} \end{bmatrix}$. ← $O(m)$ time.
  - Rearrange coordinates to original format. ← $O(m)$ time.

Returning to the full algorithm

- **New new new algorithm**:
  - Let m be the smallest power of 2 that is $\geq 2n$. $\omega = e^{2\pi i/m}$.
  - Pad vectors $\vec{a}$ and $\vec{b}$ with zeroes till length m.
  - Compute $\vec{y} \leftarrow F_m(\omega)\vec{a}, \vec{z} \leftarrow F_m(\omega)\vec{b}$.
  - Compute $\overrightarrow{w} \leftarrow \vec{a} \odot \vec{b}$, point-wise multiplication.
  - Return $F_m(\omega)^{-1}\overrightarrow{w}$.

Theorem: $F_m(\omega)^{-1} = \frac{1}{m} F_m(\omega^{-1})$.

Returning to the full algorithm

- **New new new algorithm**:
  - Let m be the smallest power of 2 that is $\geq 2n$. $\omega = e^{2\pi i/m}$.
  - Pad vectors $\vec{a}$ and $\vec{b}$ with zeroes till length m.
  - Compute $\vec{y} \leftarrow F_m(\omega)\vec{a}, \vec{z} \leftarrow F_m(\omega)\vec{b}$. ← Runtime $O(n \log n)$
  - Compute $\vec{w} \leftarrow \vec{a} \odot \vec{b}$, point-wise multiplication. ← Runtime $O(n)$
  - Return $\frac{1}{m}F_m(\omega^{-1})\vec{w}$. ← Runtime $O(n \log n)$

Fast Fourier Transform

- The algorithm for computing $F_m(\omega)\vec{a}$ is known as the **Fast Fourier Transform (FFT)**
- It serves as a major component in many algorithms particularly dealing with polynomial computations
- Quantum computers are capable of “implementing” the FFT on matrices in time $O(\text{polylog } m)$. This is the major step in the quantum factoring algorithm and the hype behind quantum computers.

Final remaining theorem

- **Theorem:** $F_m(\omega)^{-1} = \dfrac{1}{m} F_m(\omega^{-1})$.
- **Proof:** Suffices to show that $F_m(\omega) \cdot F_m(\omega^{-1}) = m\mathbb{1}$.

$$\underbrace{\left(F_m(\omega) \cdot F_m(\omega^{-1})\right)_{ij}}_{\text{matrix coordinate } i,j} = \sum_{k=0}^{m-1} \left(F_m(\omega)\right)_{ik} \left(F_m(\omega^{-1})\right)_{kj}$$

$$= \sum_{k=0}^{m-1} \omega^{ik} \cdot (\omega^{-1})^{kj}$$

$$= \sum_{k=0}^{m-1} \omega^{k(i-j)} = \begin{cases} m & \text{if } i=j \\ 0 & \text{if } i \neq j \end{cases}$$

sum of all roots of unity is 0.

General algorithm: Convolution

- **Input:** Two vectors $f = (f_0, \ldots, f_{n-1}), g = (g_0, \ldots, g_{n-1}) \in \mathbb{C}^n$
- **Output:** The convolution vector $f * g$ is defined by

$$(f * g)_k := \sum_{j=0}^{n} f_j \cdot g_{n-j}$$

- **Algorithm:** Exact same alg. as that of poly. mult.:

$$f * g = \frac{1}{n} F_n(\omega^{-1}) \big(F_n(\omega) f \odot F_n(\omega) g \big) \text{ for } \omega = e^{2\pi i/m}.$$

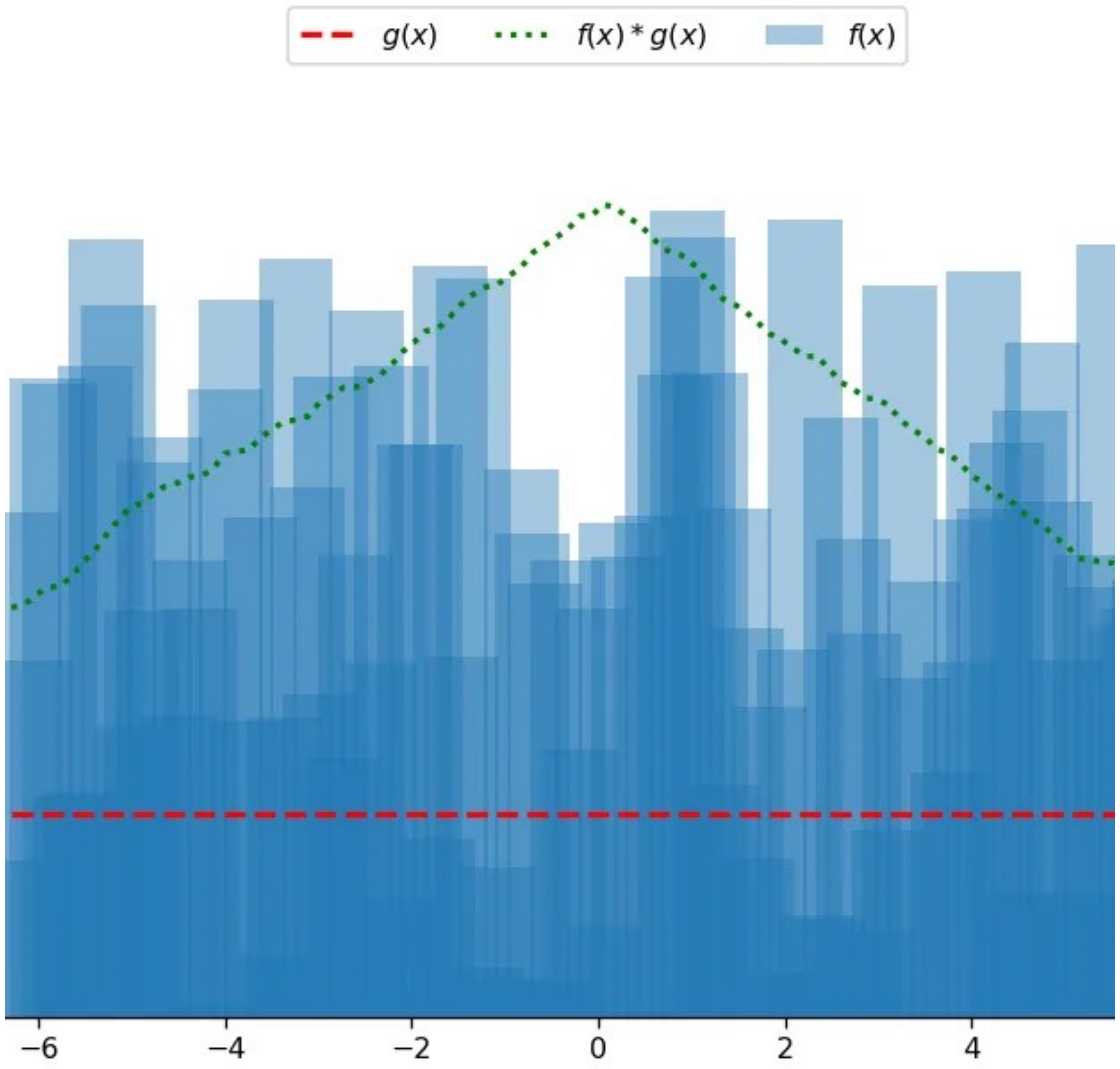
- **Runtime:** $O(n \log n)$

- **Proof:**
  - Let p, q be the poly. with coeffs. given by f, g
  - Then $f * g$ are the coeffs. of the poly. $r = pq \pmod{x^n}$
  - Then $r(\omega^j) = p(\omega^j) q(\omega^j)$ for $j = 0$ to $n - 1$
  - So, prev algorithm will find the coefficients of the unique polynomial

Overtime (stay if interested)

Convolution

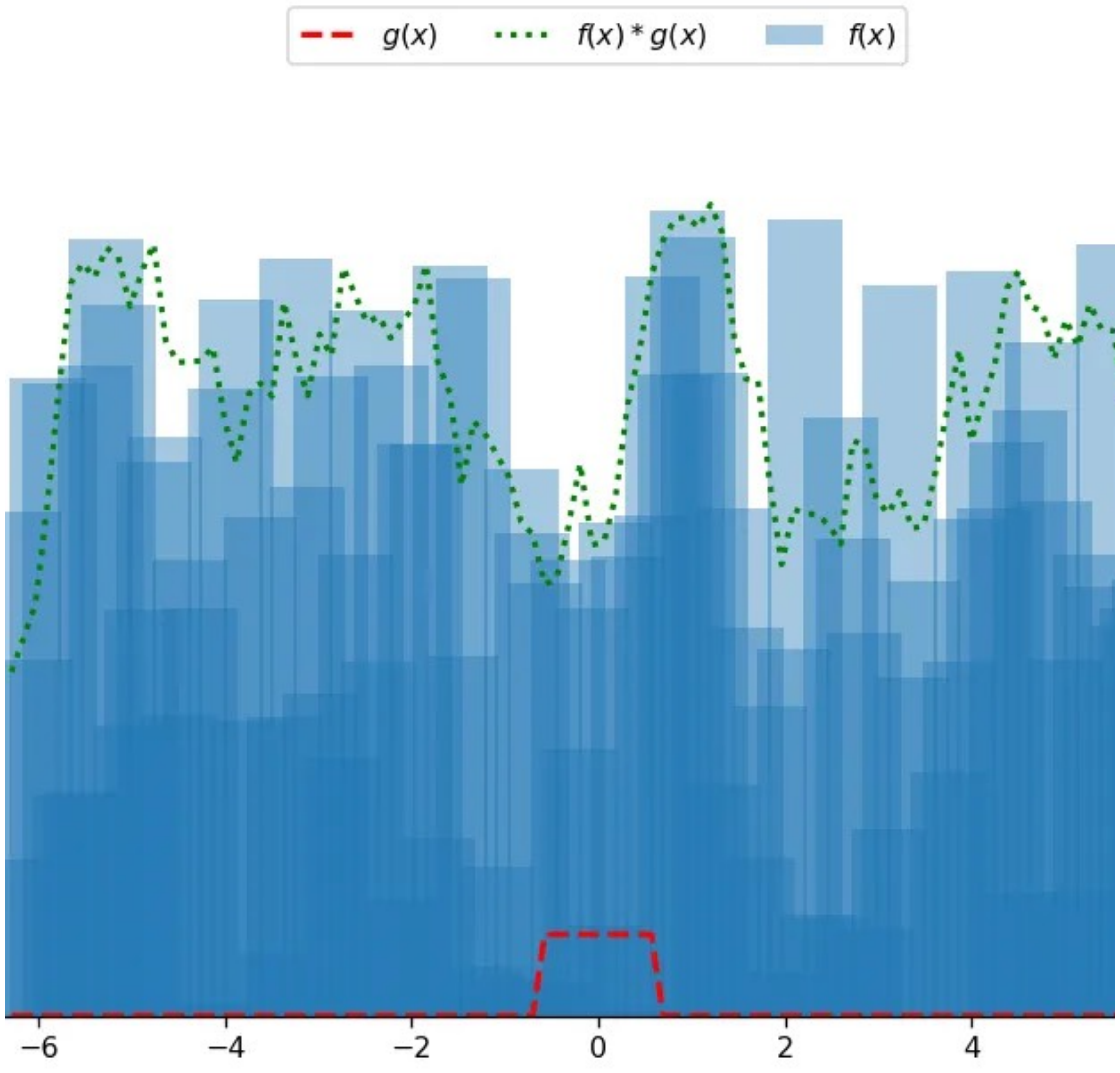
- An algorithm for combining two signals to form a third signal
- Shows up most commonly now in *convolution neural networks*
- $(f * g)_k := \sum_{j=0}^{n} f_j \cdot g_{n-j}$ vs
- $(f * g)(x) = \int_{-\infty}^{\infty} f(\tau)g(x-\tau)d\tau$
    - This is the area under the curve f with weights defined by g
    - Let's you smooth out the curve f by picking g



Source: Medium post by TDS archive.

Convolution

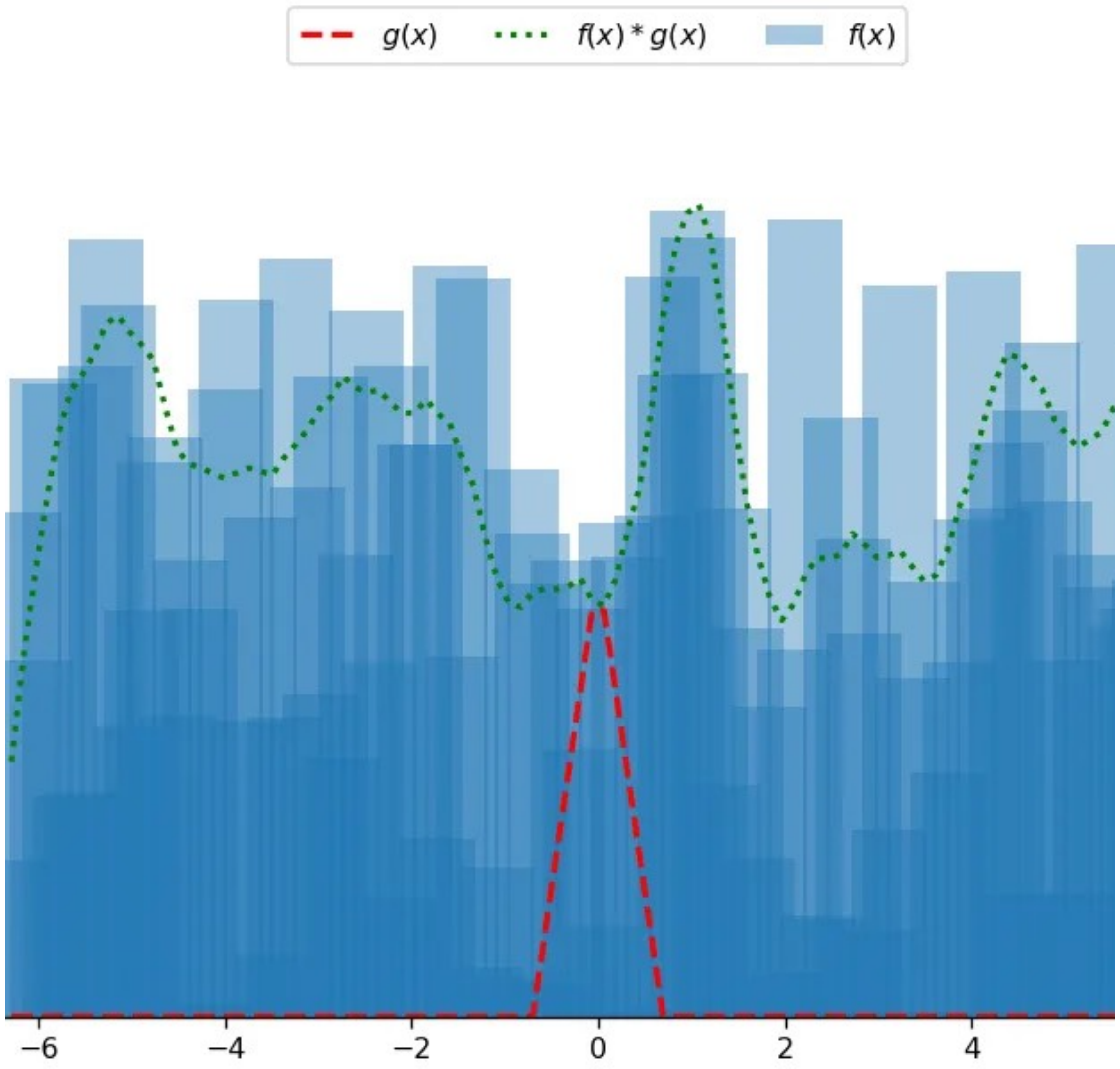
- An algorithm for combining two signals to form a third signal
- Shows up most commonly now in *convolution neural networks*
- $(f * g)_k := \sum_{j=0}^{n} f_j \cdot g_{n-j}$ vs
- $(f * g)(x) = \int_{-\infty}^{\infty} f(\tau)g(x-\tau)d\tau$
    - This is the area under the curve f with weights defined by g
    - Let's you smooth out the curve f by picking g



Source: Medium post by TDS archive.

Convolution

- An algorithm for combining two signals to form a third signal
- Shows up most commonly now in *convolution neural networks*
- $(f * g)_k := \sum_{j=0}^{n} f_j \cdot g_{n-j}$ vs
- $(f * g)(x) = \int_{-\infty}^{\infty} f(\tau)g(x - \tau)d\tau$
  - This is the area under the curve f with weights defined by g
  - Let's you smooth out the curve f by picking g



Source: Medium post by TDS archive.

Convolution

Gaussian blurring and edge detection

- Ex. We can also apply a 2D version of convolution for image processing

Source: Stanford 315b lectures

Convolution

- Filtering signals (low-pass, high-pass)
  - Convolve with a signal to filter out certain frequencies
- Audio effects (reverb, echo, suppression)
- Image processing
- And more!