Today: Fast Fourier Transform (FFT)

- → operations on polynomials vs. their representations
- → divide & conquer algorithm for manipulating polynomials
- → collapsing sets / roots of unity
- → FFT, DFT, IFFT & fast multiplication of polynomials

FFT: Likely the most taught algorithm at MIT

Shows up in all sorts of contexts:

- → signal processing: compression, speech, images, looking for aliens (!)
- → integer multiplication (cf. last lecture)
- → **multiplication of polynomials**

FOCUS TODAY

Polynomial: $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ (degree(A))

$$= \sum_{k=0}^{n-1} a_k x^k$$

$$= (a_0, a_1, a_2, \ldots, a_{n-1})$$ (coefficient vector)

Dual view: polynomials $\leftrightarrow$ vectors

(Today: "time" = # of arithmetic operations $+, -, *, /, \sqrt{\cdot}$)

Operations on polynomials:

① Evaluation:

poly. $A(x)$ & value $x_0 \rightarrow A(x_0)$

→ Naïvely ⇒ $O(n^2)$ BAD ☹

→ Horner's Rule $A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots + x\, a_{n-1}))$

⇒ $O(n)$ GOOD ☺

② Addition:

polys $A(x)$ & $B(x)$ → poly $C(x) = A(x) + B(x)$ $\forall x$

(i.e., $c_k = a_k + b_k$)

→ $O(n)$ time easy

③ Multiplication:

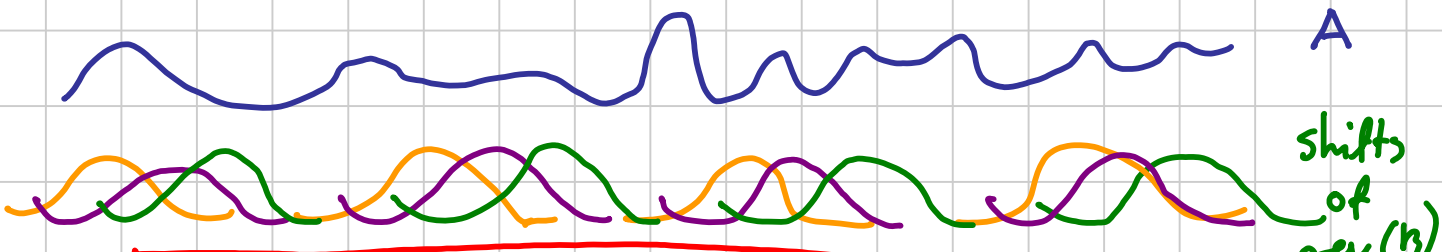
polys $A(x)$ & $B(x)$ → poly $C(x) = A(x) \cdot C(x)$ $\forall x$

(i.e., $c_k = \sum_{j=0}^{k} a_j b_{k-j}$ for $0 \le k \le 2(n-1)$ (degree doubles)

→ Why interesting?

Equivalent to convolution of vec. A & B

↳ inner products of all relative shifts



⇒ $c_k = \langle A, \text{shift}_k(\text{rev}(B)) \rangle$

$\Rightarrow$ $|c_k|$ largest $\Rightarrow$ kth shift of rev(B) most correlated with A

$\Rightarrow$ extremely useful for smoothing, signal detection, denoising, etc.

$\Rightarrow$ Fundamental in signal processing

→ Running time?

→ Naïve: $O(n^2)$ (just compute each c_k separately)

→ $O(n^{\lg 3})$ or even $O(n^{1+\varepsilon})$ $\forall \varepsilon > 0$ [cf Lec 2]

Via Karatsuba-like divide & conquer approach

(But: Not really practical due to large constants)

→ Today: $O(n \log n)$ time!

(and very practical)

Q: Is poly multiplication inherently hard?

No! It all depends on how we represent the polynomials.

Representations of polynomials:

(A) Coefficient vector $A = (a_0, a_1, \ldots, a_{n-1})$
("monomial basis")

(B) Roots + scale: (Fundamental Thm. of Algebra)

$$A(x) = c \cdot (x - r_0) \cdot (x - r_1) \cdot \ldots \cdot (x - r_{n-1})$$

(c: scale; $r_0, r_1, \ldots$: roots (can be complex))

→ Evaluation: $O(n)$ time

→ Multiplication: just concatenate roots + multiply scales

⇒ $O(n)$ time !!!

→ BUT: impossible to find exact roots with $+, -, *, /, \sqrt{}$

⇒ Addition: Hard/impossible

(C) Samples: $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$
with $A(x_i) = y_i$ $\forall i$ & x_is distinct

⇒ Degree-$(n-1)$ polynomial uniquely determined
[Lagrange & Fundamental Thm. of Algebra]

→ Addition / Multiplication:

Just add / multiply each y_j's

⇒ $O(n)$ time

→ Evaluation: Requires interpolation

Lagrange interpolation formula:

$$A(x) = \sum_{j=0}^{n-1} \left(\frac{y_j}{\prod_{k \neq j} (x_j - x_k)} \prod_{k \neq j} (x - x_k) \right)$$

⇒ $O(n^2)$ time

Algorithms vs. Representations:

	(A) Coefficients	(B) Roots	(C) Samples
(1) Evaluation	$O(n)$	$O(n)$	$O(n^2)$
(2) Addition	$O(n)$	∞	$O(n)$
(3) Multiplication	$O(n^2)$	$O(n)$	$O(n)$

Today: Almost "best of all worlds" by converting coefficients ⟺ samples in $O(n \log n)$ time

Coefficients → Samples:

Given: poly $A = (a_0, a_1, \ldots, a_{n-1})$ & $X = \{x_0, \ldots, x_{n-1}\}$

Compute: $A(x)$ for all $x \in X$ $\quad |X| = n$

Idea: Use divide & conquer!

① **Divide:** How to divide A into smaller polys?

→ one way: $(\underbrace{a_0, a_1, \ldots, a_{\frac{n}{2}}}_{A_L}, \underbrace{a_{\frac{n}{2}+1}, \ldots, a_{n-1}}_{A_R})$

BUT: not useful here

→ **Instead:** Divide into even & odd coefficients

$(\underbrace{a_0}_{A_{even}}, \underbrace{a_1}_{A_{odd}}, \underbrace{a_2}_{A_{even}}, \ldots, \underbrace{a_{n-2}}_{A_{even}}, \underbrace{a_{n-1}}_{A_{odd}})$

$$A_{even}(x) = \sum_{k=0}^{\lceil \frac{n}{2}-1 \rceil} a_{2k} x^k = (a_0, a_2, a_4, \ldots)$$

& $$A_{odd}(x) = \sum_{k=0}^{\lfloor \frac{n}{2}-1 \rfloor} a_{2k+1} x^k = (a_1, a_3, a_5, \ldots)$$

② Recursively **conquer**:

Compute $A_{even}(z)$ for all $z \in X^2 = \{x^2 \mid x \in X\}$

& $A_{odd}(z)$ for all $z \in X^2$

③ **Combine:**

$$A(x) = A_{even}(x^2) + x \cdot A_{odd}(x^2) \quad \text{for all } x \in X$$

$O(|X|)$ time →

Already computed these

Running time?

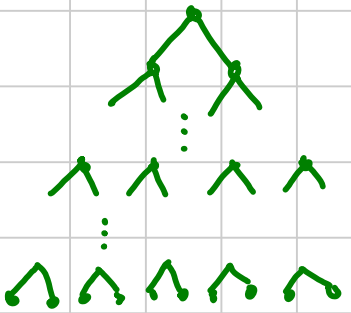
$$T(n, |A|, |X|) = 2 \cdot T\left(\frac{n}{2}, |X^2|\right) + \Theta(n + |X|)$$

Problem: $|X^2|$ could be close to $|X|$

$\Rightarrow$ no reduction in problem size!

$\Rightarrow$ Worst case: $|X^2| = |X|$ on <u>each</u> level

"Worst case" recursion tree:



level:	deg(A):	|X|:	time:
0	n	n	$\Theta(\deg(A) + \|X\|) = \Theta(n)$
1	$\frac{n}{2}$	n	$2 \cdot \Theta(n)$
k	$\frac{n}{2^k}$	n	$2^k \cdot \Theta(n)$
$\lg n$	1	n	$n \cdot \Theta(1)$

$\Theta(n^2)$

$\rightarrow$ Dead end ???

(We can fix the set of samples for our representation as we wish)

Crucial observation: <u>WE</u> choose the set X

$\Rightarrow$ Let's make it so $|X^2| << |X|$

$\Rightarrow$ **Define:** X is <u>collapsible</u> if

$\rightarrow$ $|X| = 1$, or

$\rightarrow$ $|X^2| = |X|/2$ & X^2 is <u>collapsible</u> too (recursively)

$\rightarrow$ **Note:** X collapsible $\Rightarrow |X| = 2^k$

$\rightarrow$ **From now on:** Assume $n = 2^k$ (WLOG: Just pad the input)

→ If X is collapsible then

$$T(n, |X|) = 2 \cdot T\left(\tfrac{n}{2}, |X^2|\right) + O(n + |X|) =$$

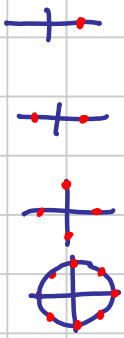
$$= 2 \cdot T\left(\tfrac{n}{2}, \tfrac{|X|}{2}\right) + O(n + |X|)$$

Reduction in size!

$$\Rightarrow T(n, n) = \boxed{O(n \log n)}$$

Remaining question: How to construct a collapsible X?

→ Just "invert" the definition (via ↖)



0. $\{1\}$ (any nonzero starting #)
1. $\{1, -1\}$
2. $\{1, -1, i, -i\}$ ← complex numbers! (that's why we avoided using i as index :)
3. $\{1, -1, \pm i, \pm \frac{\sqrt{2}}{2}(1+i), \pm \frac{\sqrt{2}}{2}(-1+i)\}$

↳ on a unit circle!

solve $(p + i \cdot q)^2 = i$

n-th roots of unity: n x's s.t. $\boxed{x^n = 1}$

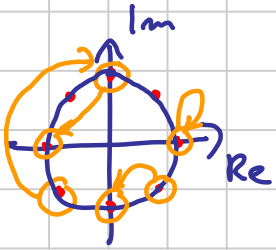
→ uniformly spaced around unit circle in complex plane (& including 1)

→ $\cos\theta + i \cdot \sin\theta = e^{i\theta}$ ← Euler's formula

for $\theta = 0, 1 \cdot \frac{2\pi}{n}, 2 \cdot \frac{2\pi}{n}, \ldots, (n-1) \cdot \frac{2\pi}{n}$ ← full circle

angle w/ real line

→ if $n = 2^\ell \Rightarrow$ collapsing:



$\to (e^{i\theta})^2 = e^{i(2\theta)} = e^{i(2\theta \bmod 2\pi)}$

(since $e^{2\pi i} = 1$) ← Euler's identity

$\Rightarrow$ (even nth root of unity)^2 = $\frac{n}{2}$-nd root of unity

Ok, so we know how to go in $O(n \log n)$ time coefficients → samples (when $x_k = e^{2\pi i \cdot \frac{k}{n}}$)

How about going sample → coefficients?

Matrix view: Given $A = (a_0, a_1, \dots, a_{n-1})$ & sample set X

If all x_js distinct
⇒ V is full rank
⇒ V invertible!

$$\underbrace{\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}}_{\text{Vandermonde matrix } V:\ v_{jk} = x_j^k} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

(A) Coefficients → Samples: Compute $V \cdot A$

→ Matrix-vector product
⇒ $O(n^2)$ ☹

→ BUT: if $x_k = e^{2\pi i \cdot \frac{k}{n}}$ ⇒ $O(n \log n)$ ☺

Sublinear in the size of V!
⇒ Computes the product implicitly

Discrete Fourier Transform (DFT)

$A \to A^* = V^* \cdot A$ with $v^*_{kj} = \left(e^{2\pi i \frac{k}{n}}\right)^j = e^{2\pi i \frac{k \cdot j}{n}}$ ☺

i.e. $a^*_k = \sum_{j=0}^{n-1} e^{2\pi i \frac{kj}{n}} \cdot a_j$

Side note: Fast Fourier Transform (FFT)
= $O(n \log n)$ divide & conquer alg. for DFT
[used by Gauss ca. 1805 (periodic asteroid orbits)
popularized by Cooley & Tukey in 1965]
(detecting Soviet nuclear tests from offshore readings)

→ practical implementation: FFTW (Fastest Fourier Transform in the West)
[Frigo & Johnson @ MIT!]

→ often implemented directly in hardware for maximum performance (for fixed n)

(B) Samples → coefficients: Compute $V^{-1} \cdot Y$

$\Leftrightarrow$ solve linear system $V \cdot A = Y$ (A = unknown)

$\Rightarrow O(n^3)$ via Gaussian elimination

$\Rightarrow O(n^\omega)$ via matrix mult. algorithm, $2 \le \omega \le 2.373$

BUT: How about the special case of computing $(V^*)^{-1} \cdot Y$?

Inverse (Discrete) Fourier Transform (IDFT):

$= A^* \to (V^*)^{-1} A^*$ (from Fourier samples to coefficients)

Key fact: $(V^*)^{-1} = \overline{V^*}/n$ ($\overline{\ }$ = complex conjugate, $\overline{p+i \cdot q} = p - i \cdot q$)

i.e. $P = V^* \cdot \overline{V^*} = n \cdot I$

Proof:

$$P_{jk} = (\text{row } j \text{ of } V^*) \cdot (\text{column } k \text{ of } \overline{V^*})$$

$$= \sum_{m=0}^{n-1} e^{2\pi i j m/n} \cdot e^{2\pi i k \frac{m}{n}}$$

$$= \sum_{m=0}^{n-1} e^{2\pi i j m/n} \cdot e^{-2\pi i k \frac{m}{n}}$$

(conjugation just switches the sign: CW orient. → CCW orient.)

$$= \sum_{m=0}^{n-1} e^{2\pi i (j-k) \frac{m}{n}}$$

① If $j = k$ $\Rightarrow$ $P_{jk} = P_{kk} = \sum_{m=0}^{n-1} 1 = n$

(So, all diagonal entries = n)

② else:

$$P_{jk} = \sum_{m=0}^{n-1} \left(e^{2\pi i (j-k)}\right)^m = \frac{\left(e^{2\pi i (j-k)/n}\right)^n - 1}{e^{2\pi i (j-k)/n} - 1}$$

geometric series

$$= \frac{e^{2\pi i (j-k)} - 1}{e^{2\pi i (j-k)/n} - 1} = 0$$

($e^{2\pi i (j-k)} = 1$; $e^{2\pi i (j-k)/n} - 1 \neq 0$)

(So, all off-diagonal entries = 0)

$\Rightarrow$ $P = n \cdot I$ ∎

(Specifically: $\text{IDFT}(Y) = \frac{1}{n}\overline{\text{DFT}(\bar{Y})}$)

$\Rightarrow$ IDFT corresponds to transformation $A \to V \cdot A$ for $x_k = e^{-2\pi i k/n}$

$\Rightarrow$ We can use an analogous alg. to FFT (IFFT) to get $O(n \lg n)$ running time again!

So, indeed, coefficients $\leftrightarrow$ samples (in Fourier basis) in $O(n \lg n)$ time!

Appl.: Fast polynomial multiplication:

Given polys $A(x)$ and $B(x)$ (in coefficient repr.) ← degree $n-1$

compute $C(x)$ s.t. $C(x) = A(x) \cdot B(x)$ $\forall x$ ← degree $2(n-1)$

Algorithm:

① Compute $A^* = \text{DFT}(A)$ & $B^* = \text{DFT}(B)$ (for degree $2(n-1)$)! using FFT $O(n \lg n)$

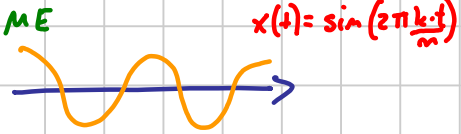
② Set $c_k^* \leftarrow a_k^* \cdot b_k^*$ for $k = 0, 1, \ldots, 2(n-1)$ $O(n)$

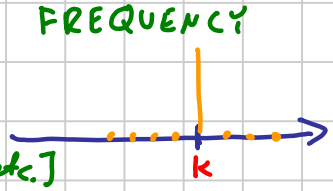
③ Use IFFT to compute $C = \text{IDFT}(C^*)$ $O(n \lg n)$

Appl. 2: Converting between time & frequency domains

(key importance in signal processing!)

- A^* complex
- $|a_k^*|$ = amplitude of frequency-k signal
- $\arg(a_k^*)$ = phase shift





Used in e.g. MP3 compression

Example: Sound [Adobe Audition, Audacity, etc.]

- high-/low-pass filter = zero out high/low frequencies
- pitch shift = shift frequency vector