

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. Lecture 3)
- multiplication of polynomials

⋮

FOCUS TODAY

degree(A)

Polynomial: $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$

$a_{n-1} \neq 0$

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

$$= (a_0, a_1, a_2, \dots, a_{n-1})$$

(coefficient vector)

Dual view:

polynomials \leftrightarrow vectors

G.046

(Today: "time" = # of arithmetic operations $+, -, *, /$) (2)

Operations on polynomials:

① Evaluation:

poly. $A(x)$ & value $x_0 \rightarrow A(x_0)$
 \rightarrow Naïvely $\Rightarrow O(n^2)$ BAD 😞

\rightarrow Horner's Rule $A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \dots + x_0 a_{n-1}))$
 $\Rightarrow O(n)$ GOOD 😊

② Addition: polys $A(x)$ & $B(x) \rightarrow$ poly $C(x) = A(x) + B(x)$ $\forall x$ (i.e., $C_k = a_k + b_k$)

$\rightarrow O(n)$ time easy

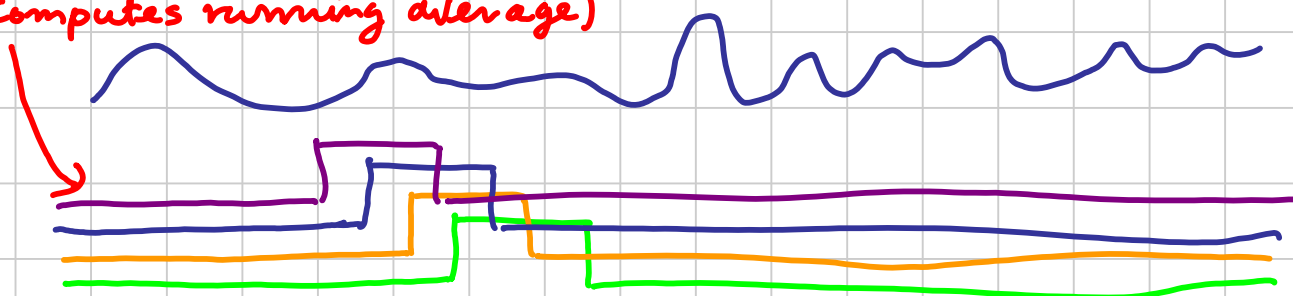
③ Multiplication: polys $A(x)$ & $B(x) \rightarrow$ poly $C(x) = A(x) \cdot B(x)$ $\forall x$ \rightarrow Why interesting? (i.e., $C_k = \sum_{j=0}^k a_j b_{k-j}$ for $0 \leq k \leq 2(n-1)$) Equivalent to convolution of vec. A & B (degree doubles)

\rightarrow inner products of all relative shifts

$$C_k = (0, \dots, 0, a_0, a_1, \dots, a_k, \dots, a_{n-1}, 0, \dots, 0) \text{ A (padded with 0s)} \Rightarrow C_k = \langle A, \text{shift}(\text{rev}(B)) \rangle_{k-(n-1)}$$

$\xrightarrow{\text{shift } b_j(k-n-1)}$ $(b_{n-1}, \dots, b_k, b_{k+1}, \dots, b_0)$ reverse of B
 (negative shift = to the left)

"boxcar filter"
 (computes running average)



A

all shifts
 of $\text{rev}(B)$

6.046

(3)

$\Rightarrow |c_k|$ largest $\Rightarrow k$ th shift of $\text{rev}(B)$ most correlated with A

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

\Rightarrow Fundamental in signal processing

\rightarrow Running time?

\rightarrow Naïve: $O(n^2)$

(just compute each c_k separately)



$\rightarrow O(n^{\log^3})$ or even $O(n^{1+\epsilon}) \forall \epsilon > 0$ [cf Lec 3]

Via Karatsuba-like divide & conquer approach

(But: Not really practical due to large constants)

\rightarrow 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 $A = (a_0, a_1, \dots, a_{n-1})$
 ("monomial basis")

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

$$A(x) = \underbrace{c}_{\substack{\uparrow \\ \text{scale}}} \cdot \underbrace{(x - r_0)}_{\substack{\uparrow \\ \text{roots}}} \cdot \underbrace{(x - r_1)}_{\substack{\uparrow \\ \text{roots}}} \cdot \dots \cdot \underbrace{(x - r_{n-1})}_{\substack{\uparrow \\ \text{roots (can be complex)}}}$$

→ Evaluation: $O(n)$ time

→ Multiplication: just concatenate roots + multiply scales

⇒ $O(n)$ time
 !!!

→ BUT: impossible to find exact roots
 with $+, -, *, /$

⇒ Addition: Hard/impossible



(C) Samples: $(x_0, A(x_0)), (x_1, A(x_1)), \dots, (x_{n-1}, A(x_{n-1}))$
 with all x_i s distinct (and fixed)


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

6.046

(5)

→ Addition / Multiplication:just add / multiply $A(x_i)$ & $B(x_i)$, for each i ⇒ $O(n)$ time
▽▽▽→ Evaluation: Requires interpolationLagrange interpolation formula:

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

⇒ $O(n^2)$ 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 \leftrightarrow samples in $O(n \log n)$ time

Coefficients \rightarrow Samples:

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

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

Idea: Use divide & conquer!

① Divide: How to divide A into smaller polys?

\rightarrow one way: $\overbrace{(a_0, a_1, \dots, a_{\frac{n}{2}})}^{A_L}, \underbrace{(a_{\frac{n}{2}+1}, \dots, a_{n-1})}_{A_R}$

BUT: not useful here

\rightarrow Instead: Divide into even & odd coefficients

$\overbrace{(a_0, a_1, a_2, \dots, a_{n-2}, a_{n-1})}^{A_{\text{even}}}$
 $\underbrace{\hspace{10em}}_{A_{\text{odd}}}$

$$A_{\text{even}}(x) = \sum_{k=0}^{\lfloor \frac{n}{2} - 1 \rfloor} a_{2k} x^k = (a_0, a_2, a_4, \dots)$$

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

② Recursively conquer:

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

③ Combine:

$O(|X|)$ time \rightarrow

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

Already computed these for all $x \in X$

Running time?

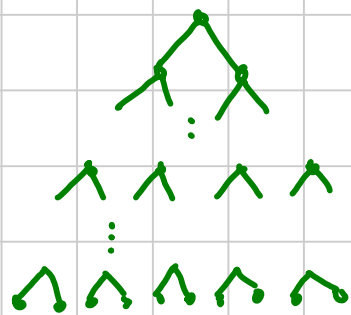
$$T(n, |X|) = 2 \cdot T\left(\frac{n}{2}, |X^2|\right) + O(n + |X|)$$

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

\Rightarrow no reduction in problem size!

\Rightarrow Worst case: $|X^2| = |X|$ on each 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	$n \cdot \Theta(n)$

$\Theta(n^2)$



\rightarrow Dead end ???

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

Crucial observation: WE choose the set X ∇

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

\Rightarrow Define: X is collapsible if

$\rightarrow |X| = 1$, or

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

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

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

→ If X is collapsible then





$$\begin{aligned} T(n, |X|) &= 2 \cdot T\left(\frac{n}{2}, |X|^2\right) + O(n + |X|) = \\ &= 2 \cdot T\left(\frac{n}{2}, \frac{|X|}{2}\right) + O(n + |X|) \\ \Rightarrow T(n, n) &= \boxed{O(n \log n)} \end{aligned}$$

Reduction in size!




Remaining question: How to construct a collapsible X ?

→ Just "invert" the definition (via $\sqrt{}$)

	①	$\{1\}$	(only nonzero starting #)
	①	$\{1, -1\}$	Complex numbers!
	②	$\{1, -1, i, -i\}$	(that's why we avoided using i as index!)
	③	$\{1, -1, \pm i, \pm \frac{\sqrt{2}}{2}(1+i), \pm \frac{\sqrt{2}}{2}(-1+i)\}$	
\vdots			on a unit circle! solve $(p+iq)^2 = i$

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

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

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

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

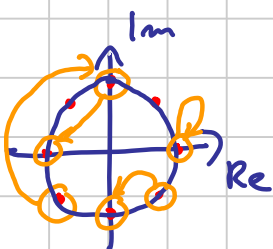
→ if $n = 2^l \Rightarrow$ collapsing:

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

(since $e^{2\pi i} = 1$)

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

Euler's identity



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

How about going sample \rightarrow coefficients?

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

Define A^* = vector form of sample representation
of A over x^*

$$x^* = \{e^{2\pi i \cdot \frac{k}{n}}\}_{k=0}^{n-1}$$

Then, given A , we can compute A^* in $O(n \log n)$ time

Observe: The mapping $A \rightarrow A^*$ is linear, i.e., for any polys A & B

$$(\alpha A + \beta B)^* = \alpha A^* + \beta B^* \quad \& \alpha, \beta \in \mathbb{C}$$

$$\Rightarrow \exists \text{ a matrix } V \text{ s.t. } V \cdot A = A^*$$

matrix-vector product

One can check:

$$V = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} a_0^* \\ a_1^* \\ a_2^* \\ \vdots \\ a_{n-1}^* \end{pmatrix}$$

with $x_j = e^{2\pi i \cdot \frac{j}{n}}$

(example of) Vandermonde matrix V : $v_{jk} = x_j^k$

① Coefficients \rightarrow samples: Compute $V \cdot A$

\rightarrow Matrix-vector product

$$\Rightarrow O(n^2)$$



Sublinear in the size of V !
 \Rightarrow computes the product implicitly

\rightarrow BUT (again) if $x_j = e^{2\pi i \cdot \frac{j}{n}} \Rightarrow O(n \log n)$

Discrete Fourier Transform (DFT)

$$A \rightarrow A^* = V^* \cdot A$$

with $v_{kj}^* = \left(e^{2\pi i \cdot \frac{k}{n}}\right)^j = e^{2\pi i \cdot \frac{k \cdot j}{n}}$



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

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

$\Rightarrow O(n^3)$ via Gaussian elimination \leftrightarrow solve linear system



$\Rightarrow O(n^3)$ via matrix mult. algorithm

$$V \cdot A = Y$$

\nwarrow unknown

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

Inverse (Discrete) Fourier Transform (IDFT):

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

Key fact: $(V^*)^{-1} = \bar{V}^*/n$ ($\bar{\cdot}$ - complex conjugate $p+iq = p-iq$)

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

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

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

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

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

conjugation just switches the sign
(CW orient. \rightarrow CCW orient.)

① If $j=k \Rightarrow P_{jk} = P_{kk} = \sum_{m=0}^{n-1} 1 = n$
(so, all diagonal entries = n)

② else:

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

geometric series \rightarrow

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

(so, all off-diagonal entries = 0)

$\Rightarrow P = n \cdot I$ □

G.046

(Specifically: $\text{IDFT}(Y) = \frac{1}{n} \text{DFT}(\bar{Y})$) (11) \Rightarrow IDFT corresponds to transformation

$$A \rightarrow V \cdot A \quad \text{for} \quad x_k = e^{-2\pi i k/n}$$

\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.)

compute $C(x)$ s.t. $C(x) = A(x) \cdot B(x) \quad \forall x$
 \uparrow 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 $k = 0, 1, \dots, 2(n-1)$
 $O(n)$
- 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

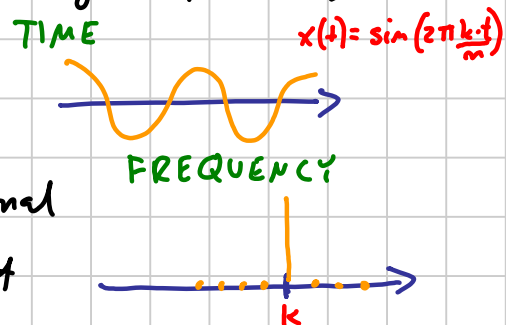
- $|a_k^*|$ = amplitude of frequency- k signal

Used in
e.g. MP3
compression

- $\arg(a_k^*)$ = phase shift

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

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



Side note: Fast Fourier Transform (FFT)

(12)

= $O(n \log n)$ 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)