15-252 SPRING 2021 – Lecture 6

Fast Integer and Polynomial Multiplication (via "Fast Fourier Transform")

Multiplying two n-bit integers

Grade school method (3000 BC) : $O(n^2)$ time

1962 Karatsuba $O(n^{\log_2 3})$

$A \cdot B$ n-bit numbers

$A = A_1 2^{n/2} + A_0$ | A_1 | A_0 |

$B = B_1 \cdot 2^{n/2} + B_0$ | B_1 | B_0 |

$$AB = A_1B_1 \cdot 2^n + (A_1B_0 + B_1A_0)2^{n/2} + A_0B_0$$

$$A_1B_0 + B_1A_0 = (A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1$$

$\Rightarrow$ 3 multiplications suffice to compute AB (of $\frac{n}{2}$-bit integers)

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

$$3^{\log_2 n}$$

$A(X) = A_1 X + A_0$

$B(X) = B_1 X + B_0$

$$A(X)B(X) = A_1B_1 X^2 + (A_0B_1 + A_1B_0)X + A_0B_0$$

3 multiplications from coefficient space suffice to compute AB.

$A_0 B_0 = A(0)\ B(0)$

$(A_0 + A_1)(B_0 + B_1) = A(1)\ B(1)$

Check: Can deduce $A_1 B_1$ from $A(2)\ B(2)$ and above two products $A(0)B(0)$, $A(1)B(1)$

Integer multiplication $\longrightarrow$ Polynomial multiplication

Karatsuba: Split number into 2 parts (or degree 1 polys)

Split number into r parts (or degree $(r-1)$ polynomials)

1963: Toom: $n^{\frac{\log(2r-1)}{\log r}}$

(Pick r as a function of n & optimize

$\hookrightarrow n \cdot 2^{O(\sqrt{\log n})} = n^{1+o(1)}$

$\hookrightarrow$ near-linear time

Improvements 1966, 69: Schonhage, Knuth

($n \cdot 2^{c\sqrt{\log n}}$ for better c)

1971: Schonhage-Strassen: $O(n \log n \log\log n)$ (bit operations)

$\hookrightarrow$ Conjecture $\Theta(n \log n)$ is the "right" complexity.

2007 : Fürer $O(n \log n \, K^{\log^* n})$

↳ K: fixed unspecified constant

$$\log^* x = \begin{cases} 1 & \text{if } x \le 1 \\ 1 + \log^*(\log x) & \text{if } x > 1 \end{cases}$$

For all practical purposes, $\log^* n$ is a constant

$\log^* n$ — super slowly growing.

Improvements : [Harvey, van der Hoeven, Lecerf]

2014-2019. $O(n \log n \cdot 4^{\log^* n})$

Finally, [2019] [Harvey & van der Hoeven]

$O(n \log n)$ time !!

A : $a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \cdots + a_1 \cdot 2 + a_0$

B : $b_{n-1} 2^{n-1} + b_{n-2} 2^{n-2} + \cdots + b_1 \cdot 2 + b_0$

Multiply these as polynomials $\cdot \; a_{n-1} x^{n-1} + \cdots + a_0$

$\cdot \; b_{n-1} x^{n-1} + \cdots + b_1 x + b_0$.

If we could this in $O(n \log n)$ multiplications of $O(\log n)$ bit numbers

Recurse on these $(\log n)$ bit numbers.

(over complex numbers)

Naive algo: $O((\log n)^2)$ time

$\Rightarrow$ n poly$(\log n)$ time algs.

Recurse: $(n \log n) \log n (\log\log n)^2$

$= n(\log n)^2 \text{poly}(\log\log n)$

Push this to limit to get

$O(n \log n)(\log\log n)(\log\log\log n) \ldots)$

One of Schonhage–Strassen algs.

$O(n \log n \log\log n)$ is a different alg, using more clever algebra.

Multiplying Polynomials

Input: $A(X) = \sum_{i=0}^{d} a_i X^i$ $\quad B(X) = b_0 + b_1 X + \cdots + b_d X^d$

(degree d polys)

Output: Coefficients $c_0, \ldots c_{2d}$ of poly

$C(X) = A(X) B(X)$.

$$C(X) = \sum_{i=0}^{2d} \left(\sum_{k=0}^{i} a_k b_{i-k} \right) X^i$$

Convolution

Compute this naively

Total $O(d^2)$ operations

Q: Can we do better than $O(d^2)$ operations?

Answer: Resounding Yes!

$d^2 \longrightarrow O(d \log d)$ (complex operations)

Key tool underlying this huge speedup is the Fast Fourier Transform (FFT)

FFT: divide-and-conquer algorithm for fast polynomial **evaluation**

Why evaluation?

Observation: Multiplication of polynomials is easy.

For every u, $C(u) = \underbrace{A(u)\,B(u)}_{1 \text{ multiplication}}$

Idea:

- Evaluate A, B on many pts
- Pointwise multiplications
- Interpolate to get C.

Representations of poly

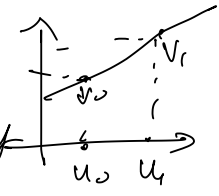
coeff rep ⇄ evaluation rep.

line
deg 1

$A(x) = a_0 + a_1 x$

$A(x) = v_0 + \frac{v_1 - v_0}{u_1 - u_0}(x - u_0)$ any u_0, u_1 distinct

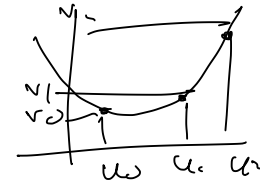
(u_0, v_0)
(u_1, v_1)



Parabola
deg 2

$a_0 + a_1 x + a_2 x^2$

(u_0, v_0)
(u_1, v_1)
(u_2, v_2)



u_0, u_1, u_2 distinct

deg d — $d+1$ coefficients — $(u_0, v_0) \dots (u_d, v_d)$ distinct u_i's

Fix a domain S, $|S| \geq d+1$ (any such S will do)

- coefficients to evaluations:

$$A(x) = \sum_{i=0}^{d} a_i x^i \mapsto \{v_u = A(u)\}_{u \in S}$$

- evaluations to coefficients

$$\{(u, v_u)\}_{u \in S} \mapsto A(x) = \sum_{i=0}^{d} a_i x^i$$

s.t. $v_u = A(u)$ $\forall u \in S$

- Interpolation
- Linear system of $|S|$ equations in variables $a_0, a_1, \dots, a_d$.
- UNIQUE SOLUTION

Poly Multiplication

Multiply: $A(X), B(X)$ $\quad A(x) = \sum_{i=0}^{d} a_i x^i$

$(a_0 \ldots a_d)$ $(b_0 \ldots b_d)$ $\quad B(x) = \sum_{i=0}^{d} b_i x^i$

1. Choose domain S with $|S| \geq 2d+1$
2. Compute $\{A(u)\}_{u \in S} := \text{Evaluate}((a_0 \ldots a_d), S)$
3. Compute $\{B(u)\}_{u \in S} := \text{Evaluate}((b_0 \ldots b_d), S)$
4. For $u \in S$, compute $C(u) = A(u)B(u)$.
   (uniquely determines $C(x) = A(x)B(x)$)
5. Compute $(c_0, c_1 \ldots c_d) := \text{Interpolate}(\{(u, C(u)) \mid u \in S\})$

$$M(d) = 2\,T_{eval}(d) + O(d) + T_{interp}(2d)$$

FFT allows to do these in $O(d \log d)$ time (for a clever, special choice of S)

Focus on poly evaluation

Naively would be $O(|S| d) = O(d^2)$

Idea: Choose S cleverly, to support a divide & conquer approach.

$$A(x) \longmapsto A(u) \mid_{u \in S}$$

View $A(x)$ as even & odd powers:

$$A(x) = A_e(x^2) + x A_o(x^2)$$

$$A(x) = 4 + 9x + 6x^2 - 3x^3 + 6x^4 + 5x^5$$

$$= \underbrace{(4 + 6x^2 + 6x^4)}_{A_e(x^2)} + x \underbrace{(9 - 3x^2 + 5x^4)}_{A_o(x^2)}$$

Obs. To compute $A(x)$ on domain S, it suffices to compute A_e, A_o on

$$S^2 = \{ \alpha^2 \mid \alpha \in S \}$$

Idea: Choose S so that S^2 is half the size!!

$S = \{-1, 1\}$

$$A(1) = A_e(1) + 1 \cdot A_o(1)$$
$$A(-1) = A_e(1) - 1 A_o(1)$$

$\Rightarrow$ need A_e & A_o on $S^2 = \{1\}$.

$S = \{1, -1, i, -i\}$

$$A(1) = A_e(1) + 1 A_o(1)$$
$$A(-1) = A_e(1) - A_o(1)$$
$$A(i) = A_e(-1) + i A_o(-1)$$

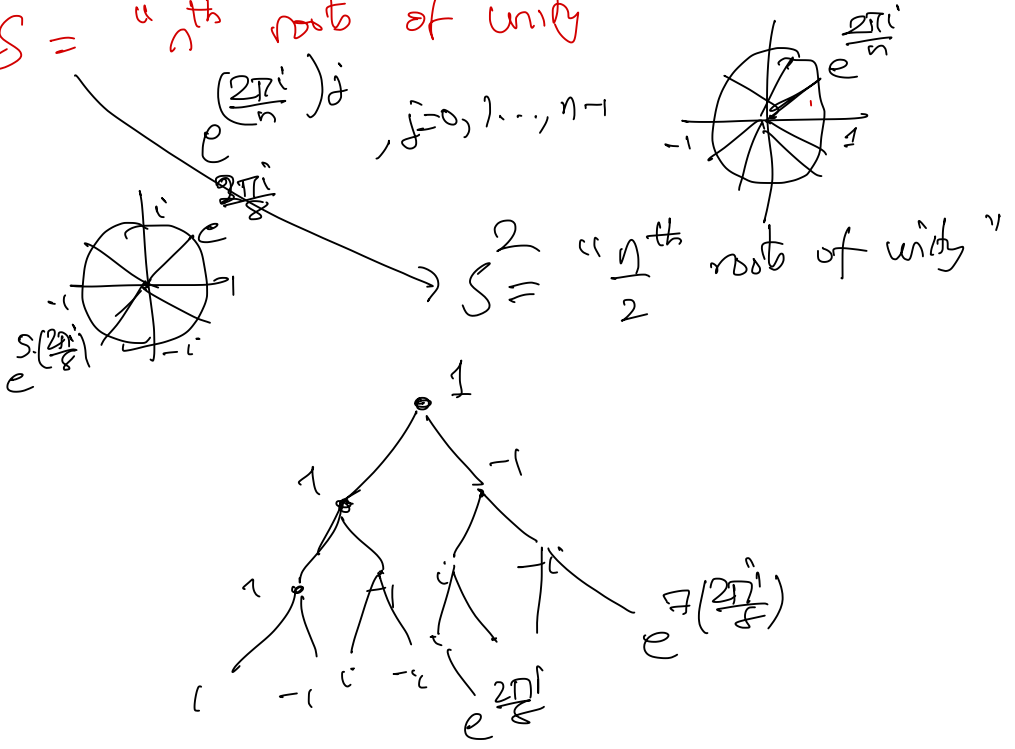
$A(-i) = A_e(-1) - i A_o(-1)$

$\Rightarrow$ need A_e & A_o on $S^2 = \{1, -1\}$.

Good choice of S: Roots of unity!

n = power of 2 (that's $\geq d$)

S = "n^{th} roots of unity"



FFT:

$$T_{eval}(d) = 2\, T_{eval}\left(\frac{d}{2}\right) + O(d)$$

FFT:

- Input: Coeffs $a_0 \ldots a_{n-1}$ (n power of 2)

  Output: $A(\omega^0)\ A(\omega^1) \ldots A(\omega^{n-1})$ $\omega = e^{\frac{2\pi i}{n}}$

Algo

1. $A_e = (a_0, a_2 \ldots, a_{n-2})$

   $A_o = (a_1, a_3 - a_{n-1})$

2. $(A_e(\omega^{2\cdot 0}), A_e(\omega^2), \ldots, A_e(\omega^{2(\frac{n}{2}-1)}))$

   $:= FFT(A_e, \omega^2)$

3. Similar for A_o

4. for $i = 0, \ldots \frac{n}{2} - 1$

   $A(\omega^i) = A_e(\omega^{2i}) + \omega^i A_o(\omega^{2i})$

   $A(\omega^{(i+\frac{n}{2})}) = A_e(\omega^{2i}) - \omega^i A_o(\omega^{2i})$

   $\omega^{i+\frac{n}{2}} = -\omega^i$

5. Output $(A(\omega^0), A(\omega^1) \ldots A(\omega^{n-1}))$

Running time: $T(n) = 2T(\frac{n}{2}) + O(n)$

$\Rightarrow T(n) = O(n \log n)$

Interpolation at $\{\omega^i \mid i = 0, 1 \ldots, n-1\}$

is also an FFT computation

(at ω^{-1})