LECTURE #4

CS 170

Spring 2021

We have applied divide and conquer to arithmetic problems:

- Karatsuba: faster integer multiplication ($n^2 \to n^{\log_2 3}$)
- Strassen: faster matrix multiplication ($n^3 \to n^{\log_2 7}$)

Today we study faster polynomial multiplication ($n^2 \to n \log n$).

The key tool underlying this (huge) speedup is the

Fast Fourier Transform (FFT).

The FFT has numerous applications across all sciences and engineering.
Examples include signal processing and coding theory.

The FFT a divide-and-conquer algorithm for polynomial evaluation.

Polynomial Multiplication

- input: coefficients $a_0, a_1, \dots, a_d$ of polynomial $A(x) = \sum_{i=0}^{d} a_i x^i$
  coefficients $b_0, b_1, \dots, b_d$ of polynomial $B(x) = \sum_{i=0}^{d} b_i x^i$
- output: coefficients $c_0, c_1, \dots, c_d$ of polynomial $C(x) := A(x) \cdot B(x)$

The naive algorithm: follow the definition of polynomial multiplication

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

The algorithm is:

$$\left.\begin{array}{l} \text{For } i = 0, 1, \dots, 2d \\ \quad \text{compute } c_i := \sum_{k=0}^{i} a_k b_{i-k} \end{array}\right\} O(d^2)$$

Also, any algorithm for polynomial multiplication must run in time $\Omega(d)$.

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

Observation: multiplication of polynomial evaluations is easy

For every u, $C(u) = A(u) \cdot B(u)$ is obtained in 1 mult of $A(u)$ and $B(u)$.

Idea: evaluate A, B on many points, then pointwise multiply, then interpolate to get C.

Representations of Polynomials

coefficient representation $\rightleftarrows$ evaluation representation

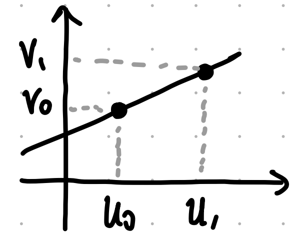
line (poly of deg $d=1$)

$$A(x) = \overset{\text{intercept}}{a_0} + \overset{\text{slope}}{a_1} x$$

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

(u_0, v_0), (u_1, v_1)

any u_0, u_1 distinct

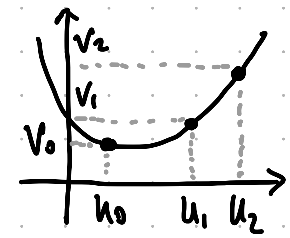


parabola (poly of deg $d=2$)

$$A(x) = a_0 + a_1 x + a_2 x^2$$

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

any u_0, u_1, u_2 distinct

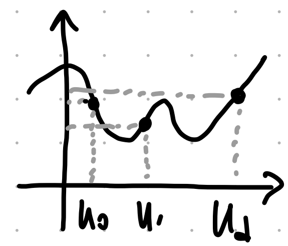


poly of deg d

$$A(x) = a_0 + a_1 x + \cdots + a_d x^d$$

$(u_0, v_0), \ldots, (u_d, v_d)$

any $u_0, \ldots, u_d$ distinct



Fix a domain S with $|S| \geq d+1$. (There are many options for choosing S.)

- coefficients to evaluations: $A(x) = \sum_{i=0}^{d} a_i x^i \mapsto \{v_u := A(u)\}_{u \in S}$
- evaluations to coefficients: $\{(u, v_u)\}_{u \in S} \mapsto \underbrace{A(x) = \sum_{i=0}^{d} a_i x^i \text{ s.t. } v_u = A(u)\ \forall u \in S}_{\text{linear system of } |S| \text{ equations in variables } a_0, a_1, \ldots, a_d}$

  aka INTERPOLATION

Polynomial Multiplication by Evaluation & Interpolation

Multiply$((a_0, a_1, .., a_d), (b_0, b_1, .., b_d))$:

1. Choose domain S with $|S| \geq 2d+1$.
2. Compute $\{A(u)\}_{u \in S} :=$ Evaluate$((a_0, a_1, .., a_d), S)$.
3. Compute $\{B(u)\}_{u \in S} :=$ Evaluate$((b_0, b_1, ..., b_d), S)$.
4. For $u \in S$, compute $C(u) := A(u)B(u)$. \\ uniquely determines $C(x) := A(x) \cdot B(x)$
5. Compute $(c_0, c_1, ..., c_{2d}) :=$ Interpolate$(\{(u, C(u))\}_{u \in S}, S)$.

The running time is $2\,T_{\mathrm{Eval}}(d) + O(d) + T_{\mathrm{Interp}}(2d)$.

The Fast Fourier Transform (FFT) enables us to do Eval/Interp in $O(d \log d)$ operations.
This leads to Polynomial Multiplication in $O(d \log d)$ operations, improving on $O(d^2)$.

Today we focus on Polynomial Evaluation only.

The naive algorithm is: For $u \in S$, compute $A(u) := \sum_{i=0}^{d} a_i u^i$.

This uses $O(|S| \cdot d)$ operations, which is $O(d^2)$ when $|S| = \Theta(d)$.

How to evaluate polynomials faster?

Idea: make a clever choice of S to support a Divide & Conquer approach

View polynomial as even and odd powers: $A(x) = A_e(x^2) + x \cdot A_o(x^2)$.

Ex: $A(x) = 4 + 12x + 20x^2 + 13x^3 + 6x^4 + 7x^5 = \underbrace{(4 + 20x^2 + 6x^4)}_{A_e(x^2)} + x \cdot \underbrace{(12 + 13x^2 + 7x^4)}_{A_o(x^2)}$

$\Rightarrow$ To compute A on S it suffices to compute A_e, A_o on S^2, and then recurse.

Let's choose S so that S^2 is half the size!

$S = \{-1, 1\}$: $\quad A(1) = A_e(1) + 1 \cdot A_o(1)$

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

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

$S = \{1, -1, i, -i\}$: $\quad A(1) = A_e(1) + 1 \cdot A_o(1)$

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

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

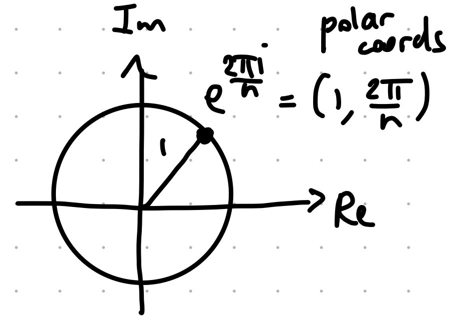
$A(-i) = A_e(-1) - i\, A_o(-1)$

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

We can use roots of unity for the set S.

Roots of unity

The n-th roots of unit are all solutions to the equation $x^n = 1$.

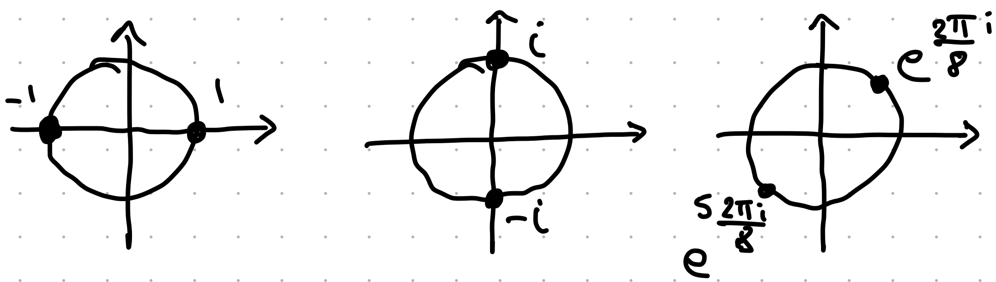


They are $\left\{1, e^{\frac{2\pi i}{n}}, e^{2\cdot\frac{2\pi i}{n}}, e^{3\cdot\frac{2\pi i}{n}}, \dots, e^{(n-1)\cdot\frac{2\pi i}{n}}\right\}$.

Recall that $e^{\pi i} = -1$, so that $e^{2\pi i} = 1$.

This means that α and $\alpha \cdot e^{\pi i}$ have the same square.

Ex:

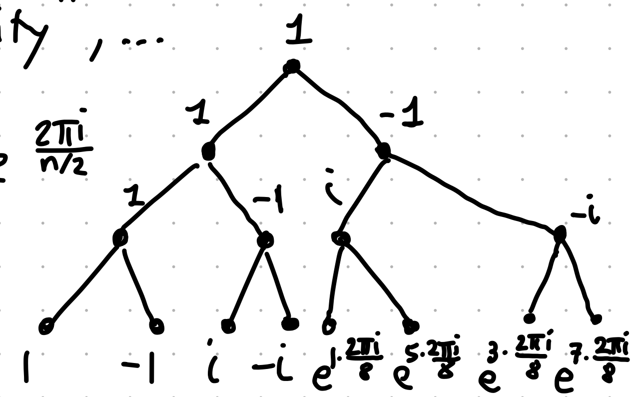


We can use roots of unity to create a set S s.t. $|S| = n$, $|S^2| = \frac{n}{2}$, $|S^4| = \frac{n}{4}$, $|S^8| = \frac{n}{8}$, ...

Fix n to be a power of 2, and let $S :=$ "n-th roots of unity".

Then $S^2 :=$ "$\frac{n}{2}$-th roots of unity", $S^4 :=$ "$\frac{n}{4}$-th roots of unity", ...

Why? $e^{\frac{2\pi i}{n}}$ and $e^{\frac{2\pi i}{n}} e^{\pi i} = e^{(1+\frac{n}{2})\cdot\frac{2\pi i}{n}}$ both square to $e^{\frac{2\pi i}{n/2}}$



Fast Fourier Transform

- input: coefficients $a_0, a_1, \ldots, a_{n-1}$ (n power of 2)
- output: $A(\omega^0), A(\omega^1), \ldots, A(\omega^{n-1})$ where $\omega := e^{2\pi i/n}$
- algorithm:
  1. $A_e := (a_0, a_2, \ldots, a_{n-2})$, $A_o := (a_1, a_3, \ldots, a_{n-1})$
  2. $\left(A_e(\omega^{2\cdot 0}), A_e(\omega^{2\cdot 1}), \ldots, A_e(\omega^{2\cdot(\frac{n}{2}-1)})\right) :=$ FFT(A_e, ω^2)
  3. $\left(A_o(\omega^{2\cdot 0}), A_o(\omega^{2\cdot 1}), \ldots, A_o(\omega^{2\cdot(\frac{n}{2}-1)})\right) :=$ FFT(A_o, ω^2)
  4. for $i = 0, 1, \ldots, n/2 - 1$:

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

     $A(\omega^{i+n/2}) := A_e(\omega^{2\cdot i}) + \omega^{i+n/2} A_o(\omega^{2\cdot i})$
  5. output $\left(A(\omega^0), A(\omega^1), \ldots, A(\omega^{n-1})\right)$

$\omega^2 = \left(e^{\frac{2\pi i}{n}}\right)^2 = e^{\frac{2\pi i}{n/2}}$

$\omega^{i+\frac{n}{2}} = \omega^i \cdot \omega^{n/2} = \omega^i e^{\pi i} = -\omega^i$

Via Master Theorem:

$a=2, b=2, d=1 \rightarrow \frac{a}{b^d} = \frac{2}{2^1} = 1$

$\rightarrow O(n^1 \log_2 n)$

The running time is $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$.

What about interpolation?

We view it as the inverse of evaluation, which is a linear transformation:

$$\begin{bmatrix} A(u_0) \\ A(u_1) \\ \vdots \\ A(u_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & u_0^1 & u_0^2 & \cdots & u_0^{n-1} \\ 1 & u_1^1 & u_1^2 & \cdots & u_1^{n-1} \\ \vdots & & & & \\ 1 & u_{n-1}^1 & u_{n-1}^2 & \cdots & u_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Vandermonde matrix on $u_0, u_1, \ldots, u_{n-1}$

Even if we are given the matrix inverse for free, multiplication on the left costs $O(n^2)$ ops.
So we again leverage the fact that the evaluation points are special:

$$\begin{bmatrix} A(\omega^0) \\ A(\omega^1) \\ \vdots \\ A(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & & & & \\ 1 & \omega^{n-1} & \omega^{2\cdot(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Call the matrix $M_n(\omega)$.
The FFT is an $O(n\log n)$ algo to (right) multiply by $M_n(\omega)$.

lemma: $M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$

proof: Show that $M_n(\omega) \cdot M_n(\omega)^{-1} = n \cdot I$.
The (i,j)-th entry is $\sum_{k=0}^{n-1} \omega^{ik}\omega^{-kj} = \sum_{k=0}^{n-1} (\omega^{i-j})^k$.
If $i=j$ then $C_{ij} = n$. If $i \neq j$ then $C_{ij} = \frac{1-(\omega^{i-j})^n}{1-\omega^{i-j}} = \frac{1-(\omega^n)^{i-j}}{1-\omega^{i-j}} = 0$.

$\Rightarrow$ To interpolate $\{(\omega^i, v_i)\}_{i=0}^{n-1}$,
run $\frac{1}{n} \cdot \mathrm{FFT}((v_0, v_1, \ldots, v_{n-1}), \omega^{-1})$. $[\omega^{-1} = e^{-\frac{2\pi i}{n}} = e^{(\frac{n-1}{n})2\pi i}]$

We can also interpolate in $O(n\log n)$ ops.