ECS 122A Lecture 5

Integer multiplication (Karatsuba's Alg)

Setting: Given 2 n-bit integers x, y,
compute $z = xy$.

Standard alg: $O(n^2)$ time.
w/ long multiplication

$$
\begin{array}{r}
1101 \\
\times\ 0101 \\
\hline
1101 \\
0000 \\
1011 \\
+\ 0000 \\
\hline
0110001
\end{array}
$$

Can we do better (like Strassen's)?

First step, D+C view:

$x = x_{low} + 2^{\lfloor \frac{n}{2} \rfloor} x_{high}$

$y = y_{low} + 2^{\lfloor \frac{n}{2} \rfloor} y_{high}$

where $x_{low} = x \,\%\, 2^{\lfloor \frac{n}{2} \rfloor}$ (bottom bits)

$x_{high} = x \text{ div } 2^{\lfloor \frac{n}{2} \rfloor}$

bit shift

$$xy = x_{low}y_{low} + 2^{\lfloor \frac{n}{2} \rfloor}(x_{high}y_{low} + y_{high}x_{low}) + 2^{2\lfloor \frac{n}{2} \rfloor} x_{high}y_{high}$$

Simple method, compute 4 products then do bit shift to compute product w/ $2^{\lfloor \frac{n}{2} \rfloor}$ and $2^{2\lfloor \frac{n}{2} \rfloor}$

Recurrence:

$$T(n) \le 4T(\tfrac{n}{2}) + O(n)$$

$$\Rightarrow T(n) \approx \Theta(n^2)$$

How to improve to 3 multiplications?

Compute
- $x_{low}y_{low} \leftarrow a$
- $x_{high}y_{high} \leftarrow b$
- $(x_{low}+y_{low})(x_{high}+y_{high}) \leftarrow c$

Then $x_{high}y_{low} + y_{high}x_{low} = c - a - b$

$$T(n) \le 3T(\tfrac{n}{2}) + \alpha n$$

$\Rightarrow T(n) \le n^{\lg 3} - \beta n$ by induction

for some $\beta \gg \alpha$

Polynomial multiplication

Setting: Given two degree-n polynomials

$$P(x) = p_0 + p_1 x + \cdots + p_n x^n$$

$$q(x) = q_0 + q_1 x + \cdots + q_n x^n$$

Compute the product $(p \cdot q)(x)$.

Same alg!

Let $p_{low}(x) =$ terms of p up to degree $\lfloor \frac{n}{2} \rfloor$

$p_{high}(x) = (p(x) - p_{low}(x)) / x^{\lfloor \frac{n}{2} \rfloor + 1}$

(i.e., terms with degree $> \lfloor \frac{n}{2} \rfloor$)

Compute $p \cdot q$

$$= p_{low}\, q_{low} + x^{\lfloor \frac{n}{2} \rfloor + 1}\left(p_{low}\, q_{high} + q_{low}\, p_{high}\right) + x^{2\lfloor \frac{n}{2} \rfloor + 2}\, p_{high}\, q_{high}$$

Exactly same structure as before

$O(n^{\log_2 3})$ alg!

Can we do even better? YES.

Can improve (basically) to $O(n \log n)$

Convolution

Setting: Given 2 length-n seq

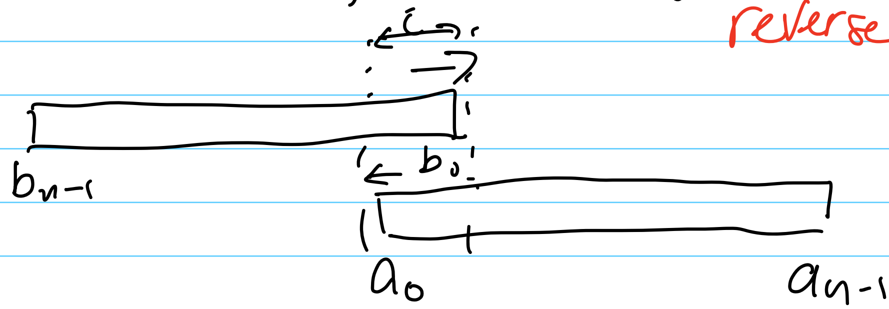
$$a = [a_0\ a_1\ a_2 \dots a_{n-1}]$$

$$b = [b_0\ b_1\ b_2 \dots b_{n-1}]$$

Compute the convolution $a * b$ (of length $n+1$)

$$(a*b)_i = \sum_{j=0}^{i} a_j b_{i-j}$$

reverse then swipe



Naive algorithm: $O(n^2)$

n terms : each term $O(n)$ work

Improve to $O(n \log n)$ time with

Fast Fourier Transform

Discrete Fourier Transform

Given a length-n seq (vector) s, the Discrete Fourier Transform of s is

$$\tilde{F}_n s$$

where
$$\tilde{F}_n = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & & & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \ddots & & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & & & \vdots \\ \vdots & \vdots & \vdots & & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & & \cdots & \omega^{(n-1)^2} \end{pmatrix}$$

ω is an n^{th} root of unity (that isn't 1)

i.e. $\omega^n = 1$ (say $e^{-\frac{2\pi i}{n}}$)

This is a library call you can find in most languages.

Why?

$$\tilde{F}_n^{-1} = \frac{1}{n}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \ddots & & \omega^{-2(n-1)} \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)^2} \end{pmatrix}$$

Essentially same as Fourier Transform

pointwise multiply

$$\underbrace{a * b}_{\text{seq}} = \underbrace{\mathcal{F}_n^{-1}\Big(\underbrace{\underbrace{(\mathcal{F}_n a)}_{\text{seq}} \bullet \underbrace{(\mathcal{F}_n b)}_{\text{seq}}}_{\text{seq}}\Big)}_{\text{seq}}$$

Convolution in normal domain = ptwise mult in Fourier domain

Naive computation of DFT: $O(n^2)$ time (matrix vector product)

Fast Fourier Transform: $O(n \log n)$ time but numerically unstable

Useful how?

Polynomial multiplication is a convolution

$$p \cdot q = p * q$$

$$(p \cdot q)(x) = \sum_i x^i \sum_j p_j q_{i-j}$$

So we can multiply polynomials in $O(n \log n)$ time

MATLAB implements conv() naively ...

Fast Fourier Transform

How to compute

$\tilde{F}_n v$ in $O(n \log n)$ time?

By defn,

$$(\tilde{F}_n v)_k = \sum_{j=0}^{n-1} v_j e^{-i\frac{2\pi kj}{n}}$$

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

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

$$= \boxed{\sum_{j=0}^{n/2-1} v_{2j} e^{-i\frac{2\pi k}{n}(2j)}} + e^{\frac{-i2\pi k}{n}} \boxed{\sum_{j=0}^{n/2-1} v_{2j+1} e^{-i\frac{2\pi}{n}k(2j)}}$$

$FFT(\frac{n}{2})$

Pointwise multiplication

$$T(n) \le 2T(\tfrac{n}{2}) + O(n)$$

$$\Rightarrow T(n) \le O(n \log n).$$