Algorithm Analysis and Design

Pratyay Suvarnapathaki, 2020111016

Week 3, Lecture 1

The Polynomial Product Problem

The problem statement is quite simple.
Given two polynomials $A(x)$ and $B(x)$, their product $C(x) = A(x)B(x)$ is to be found, as efficiently as possible.
Writing out the polynomials:
$A(x) = a_0 + a_1x + \cdots + a_dx^d$
$B(x) = b_0 + b_1x + \cdots + b_dx^d$
$C(x) = c_0 + c_1x + \cdots + c_{2d}x^{2d}$,
where the coefficients $c_k = \sum_{i=0}^{k} a_ib_{k-i}$
Here,
The naïve approach is intuitive, just multiply the individual coefficients to get the answer in $O(d^2)$

The question now is,
Can we do better?
The answer is yes, it is possible to do this in $O(d\ logd)$, using the Fast Fourier Transform (FFT), which is yet another algorithm that uses the 'divide-and-conquer' methodology.

Polynomial Representation

The polynomial $A(x) = a_0 + a_1x + \cdots + a_dx^d$ can be represented in two ways:

1. **Coefficient representation**
   All the coefficients a_i are stored in a list $[a_0, a_1, \cdots, a_d]$, where d is the desgree of the polynomial.
2. **Point-Value Representation**
   A polynomial can be uniquely identified using $d + 1$ point-vaue pairs $(x_i, A(x_i))$

Conversion from coefficient to point-value representation is called **evaluation**, and the reverse is called **interpolation**.

Since polynomial multiplication is clearly more efficient (linear time) in the point-value form, we come up with the following rudimentary plan:

- Given $A(x)$, pick n such that $n > 2d + 1$ and $n = 2^k, k\epsilon Z$ points $x_0, \cdots, x_{n-1}$
- Compute $A(x_0), \cdots, A(x_{n-1})$ and $B(x_0), \cdots, B(x_{n-1})$
- Then compute $C(x_i) = A(x_i)B(x_i)\ \ \forall i = 0, \cdots n - 1$
- Interpolate to obtain $C(x) = c_0 + c_1x + \cdots + c_{2d}x^{2d}$

However, upon inspection, this plan is not very promising because:
Although the multiplication itself is now faster owing to the point-value form, evaluation costs $O(n^2)$ time and interpolation costs even more time, even if we were to use Strassen's method.

Therefore, we now apply the '**divide and conquer**' methodology to the evaluation and interpolation steps to hopefully make the above solution more efficient.

Evaluation using Divide and Conquer

Naturally, evaluating $A(x)$ at a point, say x_k, takes $O(d)$ time, where d is the degree of the polynomial.
The way we can make this method more efficient is by searching for points which result in a lot of *overlap* in computation, thereby reducing the requisite number of computations.

Therefore, intuitively, it makes sense to split $A(x)$ into two $d/2$ degree polynomials $A_e(x) and A_o(x)$, consisting of the even and odd powers of x, resprectively.
For eg.
For $A(x) = 5 + 2x + 6x^2 + 3x^3 + 7x^4 + 8x^5$,
$A_e(x) = 5 + 6x + 7x^2$ and $A_o = 2 + 3x + 8x^2$
$A(x) = A_e(x^2) + xA_o(x^2)$

The advantage of doing this is that the polynomial can now be evaluated for **two** points using just two evaluations of degree $d/2$, since
$A(x) = A_e(x^2) + xA_o(x^2)$ and $A(-x) = A_e(x^2) - xA_o(x^2)$
i.e. Evaluating $A(x)$ at n points has now reduced to evaluating two $d/2$ degree polynomials $A_e(x)$ and $A_o(x)$ at just $n/2$ points.
Recursively, $T(n) = 2T(n/2) + O(n)$

The problem now is,
This plus/minus trick only works for the *first step of recursion*. The square numbers obtained in the first recursive step inherently cannot be plus/minus pairs.
Therefore here, we have to utilise **complex numbers**
But the thing is, the complex numbers that we will use will be such that their evaluation matrix *exactly maches* their **fourier transform**.
In other words, the fourier transform can be viewed as a transformation of a polyomial in coefficient form to its point-value form.

The Fast Fourier Transform

As concluded above, in order to get plus-minus pairings at each step of recursion, we choose our initial d points to be the n$^{\text{th}}$ roots of unity represented by $1, \omega, \omega^2, .. \omega^{n-1}$ where $\omega = e^{2\pi i/n}$.
This makes the evaluation step have $(d\ logd)$ or, more precisely, $O(d\ logn)$ complexity.

Pseudocode for Evaluation

function FFT (A, ω)
Input: Coefficient representation of a polynomial $A(x)$ of degree $d \leq n-1$ where n is a power of 2, and ω: an n$^{\text{th}}$ root of unity
Output: Value representation $A(\omega), \cdots, A(\omega^{n-1})$
Logic:
if ω=1: return $A(1)$
express $A(x)$ in the form $A_e(x^2) + xA_o(x^2)$
call $FFT(A_e, \omega^2)$ to evaluate A_e at even powers of ω
call $FFT(A_o,\omega^2) to evaluate A_o, at even powers of ω
for $j = 0$ to $n-1$:
compute $A(\omega^j) = A_e(w^{2j}) + \omega^j A_o(\omega^{2j})$
return $A(\omega), \cdots, A(\omega^{n-1})$

Evaluation and Interpolation Matrices

For polynomial
$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ A(x_2) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

As mentioned above, we put $x_i = \omega^i$
In the above equation, the middle Vandermonde matrix M plays an important role.
Evaluation corresponds to multiplication by M while interpolation corresponds to multiplication by M^{-1}

Thus,

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

To calculate inverse, we have: $M_n(\omega)^{-1} = \frac{1}{n} M_n\left(\omega^{-1}\right)$

Thus,

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} A\left(\omega^0\right) \\ A\left(\omega^1\right) \\ A\left(\omega^2\right) \\ \vdots \\ A\left(\omega^{n-1}\right) \end{bmatrix}$$

Hence, in summary,

Coefficient form $\overset{FFT}{\Longrightarrow}$ Point-Value form
Multiply in Point-Value form
Point-Value form $\overset{FFT}{\Longrightarrow}$ Coefficient form

---