Algorithms Analysis and Design

Week 3 - Diary

Ayan Agrawal (2020101034)

Lecture 5 : FFT

Problem for the class :

> ***Given 2 polynomials, $P(x)$ and $Q(x)$ of degree 'd' each, we need to find the product $R(x) = P(x).Q(x)$.***
>
> - Now, We will firstly discuss a naive approach for the following task. Then like always, we will see if "We can do better?".

1. Naive approach

- It is very intuitive one, we multiply every term of polynomial $P(x)$ with every term of polynomial $Q(x)$. This would result in d^2 operations. As each of them cost $O(1)$, therefore time complexity of naive approach would be $O(d^2)$. This is most basic method, we can surely think of something innovative and less-time consuming.
- Again, we have question infront of us that "Can we do better?". Yes, we surely can do better.
- ***Fast Fourier Transform*** Algorithm which is a very famous and genius algorithm helps us do better in our problem. But, it is a quite complex algorithm and involves solving the problem from the very root of polynomials. Therefore, Prof. first taught us a less-known method of representing the polynomials.

2. Unique way of representing polynomials

- In this method, unlike representing polynomial in form of coeffecients and powers of variable, we represent polynomial by taking any `d+1` points on the graph of polynomial where `d` is the degree of the polynomial.
- We can understand the method as, we will have `d+1` points which satisfy the polynomial so we can find `d+1` coeffecients of the polynomial by substituting these values in $A(x) = a_0x^0 + a_1x^1 + ... a_dx^d$.
- For example, if our polynomial is say, $P(x) = x^2 - 5x + 6$, then we could represent it in a set of tuples. As $degree(P(x)) = 2$, we would need $d + 1 = 3$ tuples. They can be $\{(2,0),(1,2),(5,6)\}$. This method would help us in understanding and implementing FFT.
- This method can be called as ***Value-representation*** of a polynomial.

3. Fast Fourier Transform Algorithm (Divide & Conquer method)

- We would firstly divide the polynomial in 2 parts - even powers terms and odd power terms.

$$P(x) = P_e(x^2) + xP_o(x^2)$$
$$\text{where, } P_e(x^2) \text{ represents the group of even power terms of polynomial } P(x)$$
$$\text{and } P_o(x^2) \text{ represents the group of odd power terms.}$$

> *Can you figure out the advantage of this step?*

As we saw in the method to represent a polynomial, we need `d+1` points to represent a polynomial. Here, we actually need to find only `(d+1)/2` points. How? Let's see,

$$P(x_i) = P_e(x^2) + x_iP_o(x^2)$$
$$P(-x_i) = P_e(x^2) - x_iP_o(x^2)$$

Here, we can pair up $x_i, -x_i$ but we need a way to find x_i for all i , we cannot directly just use any x_i as $x_i, -x_i$ are transforming into x_i^2 after performing single recursion step. This method would therefore fail after one step.

We need to find a such that some x_i gives us $-x_i$ even on squaring. Think what would be this... It is definitely complex numbers which gives us negative quantities even after squaring them. This will help us in making pairs of $x_i, -x_i$ even after a recursion step. The pairs here would be therefore $x_i, x_{i+\frac{d}{2}}$.

Now, we need to decide such complex numbers which makes our task easier and not gives us hard time in handling complex numbers rather than focussing on the algorithm. ***Roots of Unity*** *are the simplest complex numbers which are easy to handle.*

We can now move on to our actual FFT algorithm, as we have made whole base for it.

Algorithm :

Given 2 polynomials $P(x), Q(x)$ we can represent them as Value-represent polynomials in complexity $O(n \log n)$. How? Let's see,

> By divide and conquer strategy, which is being used here, Recurrence relation formed is :
>
> $$T(n) = 2T(\tfrac{n}{2}) + O(n)$$
>
> Can you recall which else algorithm had same relation. Yes, Merge Sort algorithm had similar recurrence relation and therefore, this algorithm also has complexity $O(n \log n)$.

Though, representing a polynomial $R(x)$ from 2 value-represented $P(x), Q(x)$ takes $O(n)$, as there we just need to multiply corresponding value of $P(x_i), Q(x_i)$ and obtain $R(x_i)$.

Pseudo code for FFT algorithm is :

```
def fft(poly,ω):
    # poly is a vector having coeffecients of polynomial
    n = len(poly)
    if n == 1:
        return poly
    # ω(omega) is n-th primitive root of unity

    poly_e = [poly[0],poly[2],...poly[n-2]]
    poly_o = [poly[1],poly[3],...poly[n-1]]

    #express poly(x) in the form poly_e(x^2) + x*poly_o(x^2)

    v_e = fft(poly_e,ω^2)
    v_o = fft(poly_o,ω^2)

    vector v(n)

    for k in range(n/2):
        v[k] = v_e[k] + ω^(k)*v_o[k]
        v[n/2 + k] = v_e[k] - ω^(k)*v_o[k]

    return poly(ω^0), . . . , poly(ω^n-1)
```

Interpolation :

The interpolation step deals with reverse of what we did till now. It converts value representation of the polynomial into coefficient representation.

> $$\begin{aligned} < values > &= FFT(< coefficients >, \omega) \\ < coefficients > &= \frac{1}{n} FFT(< values >, \omega^{-1}) \end{aligned}$$

Now let's consider what we did in matrix form :

$$\underbrace{\begin{bmatrix} 1 & \omega_0 & \omega_0^2 & \dots & \omega_0^{n-1} \\ 1 & \omega_1 & \omega_1^2 & \dots & \omega_1^{n-1} \\ & & . & & \\ & & . & & \\ & & . & & \\ 1 & \omega_{n-1} & \omega_{n-1}^2 & \dots & \omega_{n-1}^{n-1} \end{bmatrix}}_{\uparrow \text{ solution matrix}} \underbrace{\begin{bmatrix} r_0 \\ r_1 \\ . \\ . \\ . \\ r_n \end{bmatrix} }_{\uparrow \text{ coefficient matrix}} = \begin{bmatrix} R(x_0) \\ R(x_1) \\ . \\ . \\ . \\ R(x_n) \end{bmatrix} \leftarrow \text{ value matrix}$$

Here, x_i is ω_i. Matrix in RHS is obtained by using FFT. We can get matrix of coefficients by multiplying inverse of solution matrix with the matrix containing the values.

The following equation shows how we do it,

$$\begin{bmatrix} r_0 \\ r_1 \\ . \\ . \\ . \\ r_n \end{bmatrix} = \begin{bmatrix} 1 & \omega_0 & \omega_0^2 & \dots & \omega_0^{n-1} \\ 1 & \omega_1 & \omega_1^2 & \dots & \omega_1^{n-1} \\ & & . & & \\ & & . & & \\ & & . & & \\ 1 & \omega_{n-1} & \omega_{n-1}^2 & \dots & \omega_{n-1}^{n-1} \end{bmatrix}^{-1} \begin{bmatrix} R(x_0) \\ R(x_1) \\ . \\ . \\ . \\ R(x_n) \end{bmatrix}$$

A thing we got to observe was coefficients a_{ij} of inverse matrix was nothing but $a_{ij} = \frac{1}{n} x_{ij}^{-1}$. (x_{ij} are values in solution matrix)

Now, we can again apply FFT using a_{ij} and coefficients being values this time. It will give result array as actual coefficient list of multiplication.

Therefore, This algorithm takes $O(n \log n)$ time complexity which is much much quicker than naive algorithm having complexity $O(n^2)$.

> **SUMMARY :** *We have a $O(d \log d)$ algorithm to multiply 2 degree d polynomials.*