CS 3510: Design and Analysis of Algorithms, Summer 2023
Instructor: He Jia, Georgia Institute of Technology.

Lecture 4, 5/25/23. Scribed by Abdullah AlJasser, Andrew McBurnett, Robert Terpin.

1 Poll Questions

1.1 Questions

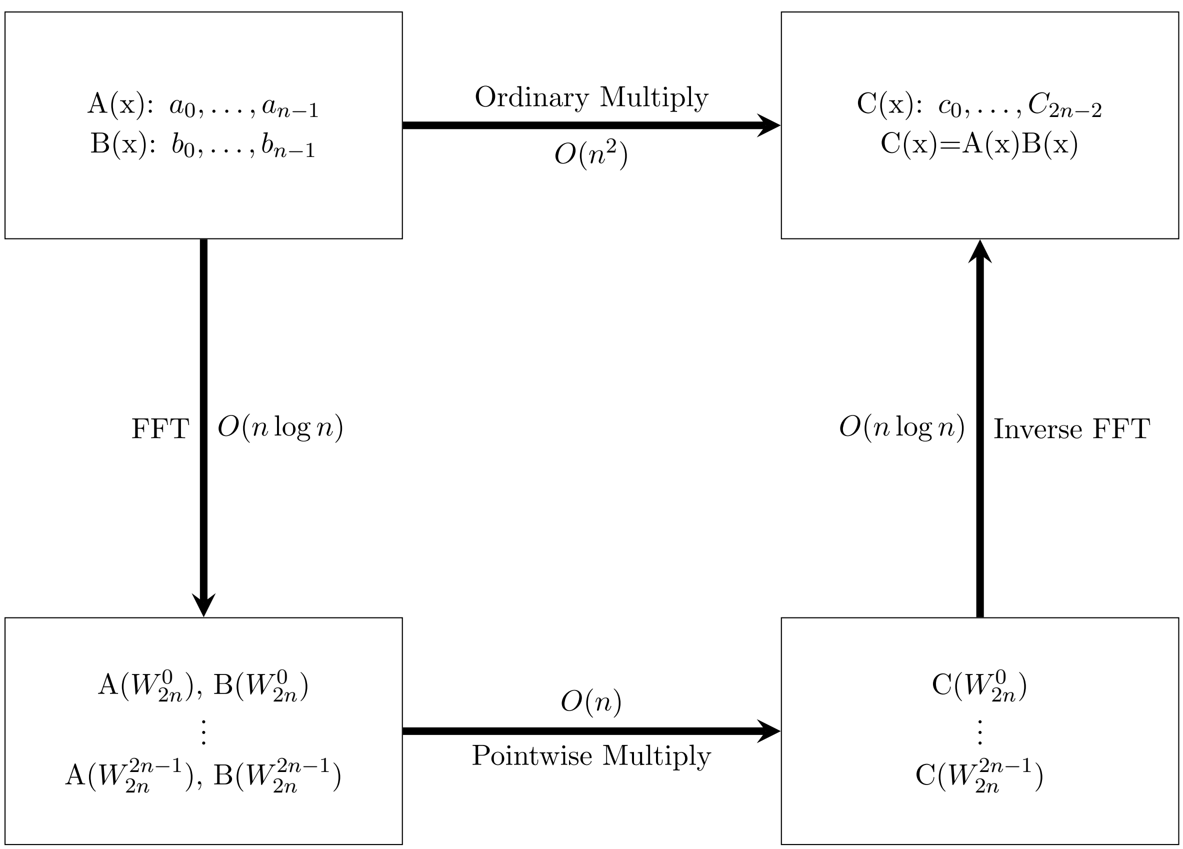
- How many distinct points are needed to uniquely determine a degree-k polynomial?
  - 2
  - k-1
  - k
  - k+1
- How many coefficients are needed to uniquely determine a degree-k polynomial?
  - 2
  - k-1
  - k
  - k+1
- What is the degree of the product of two polynomials of degree k?
  - k
  - k+1
  - 2k
  - k^2
- How many complex roots does $z^n = 1$ have?
  - 1
  - 1 or 2
  - n
  - n+1

1.2 Answers

- How many distinct points are needed to uniquely determine a degree-k polynomial?
  - k+1
- How many coefficients are needed to uniquely determine a degree-k polynomial?
  - k+1
- What is the degree of the product of two polynomials of degree k?
  - 2k
- How many complex roots does $z^n = 1$ have?
  - n

2 Recap

2.1 Polynomial Multiplication Process



In this problem, we take the coefficients of two polynomials A(x) and B(x) each with a degree of $n-1$ at the most, and we want to multiply them to get C(x). Instead of using ordinary multiplication which is $O(n^2)$, we can use the FFT polynomial multiplication algorithm to get a better efficiency.

2.2 Algorithm: Polynomial-Multiplication-FFT

Input:
$a = (a_0, ..., a_{n-1})$
$b = (b_0, ..., b_{n-1})$

Output:
$c = (c_0, ..., c_{2n-2})$

Algorithm Steps:

1. Run FFT(a, ω_{2n}) and FFT(b, ω_{2n}) to get $A(x)$ and $B(x)$ at the $(2n)^{th}$ roots of unity.
2. Multiply to get $C(x) = A(x)B(x)$ at the $(2n)^{th}$ roots of unity.
3. Run InverseFFT(C) to get $c = (c_0, ..., c_{2n-2})$.

2.3 Algorithm: FFT

Input:
$a = (a_0, ..., a_{n-1})$
ω, an n^{th} root of unity

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

Algorithm Steps:

1. if $\omega = 1$, return $A(1)$
2. Let $a_{even} = (a_0, a_2, ..., a_{n-2})$ and $a_{odd} = (a_1, a_3, ..., a_{n-1})$
3. $(s_0, s_1, ..., s_{\frac{n}{2}-1}) = FFT(a_{even}, \omega^2)$
4. $(t_0, t_1, ..., t_{\frac{n}{2}-1}) = FFT(a_{odd}, \omega^2)$
5. For $j = 0 \rightarrow \frac{n}{2} - 1$
   - $r_j = s_j + \omega^j t_j$
   - $r_{\frac{n}{2}+j} = s_j - \omega^j t_j$
6. Return $(r_0, r_1, ..., r_{n-1})$

NOTE: in the above algorithm, r_j represents $A(\omega^j)$, and $r_{\frac{n}{2}+j}$ represents $A(\omega^{j+\frac{n}{2}})$

3 Matrix View of FFT

For a polynomial A(x), when we apply $FFT(a, \omega_n)$, we get $A(\omega_n^0), A(\omega_n^1), ..., A(\omega_n^{n-1})$. The following is a matrix representation of the FFT algorithm.

1. For points $x_0, x_1, ..., x_{n-1}$

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

2. $x_j = \omega_n^j$ for $j = 0, ..., n-1$

$$\begin{bmatrix} A(\omega_n^0) \\ A(\omega_n^1) \\ \vdots \\ A(\omega_n^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$A = M_n(\omega_n)a$$

3. Inverse FFT: $a = (M_n(\omega_n))^{-1}A$

<u>Lemma</u>

$$(M_n(\omega_n))^{-1} = \frac{1}{n}M_n(\omega_n^{-1})$$

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

$$m_n(\omega_n^{-1}) =$$

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \dots & \omega_n^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \dots & \omega_n^{-(n-1)(n-1)} \end{bmatrix}$$

4 Inverse FFT

Inverse FFT(C)
Input:
$C(\omega^0), C(\omega^1), ..., C(\omega^{2n-1})$ is a set of point-values

Output:
$c = (c_0, c_1, ..., c_{2n-1})$

$(S_0, S_1, ..., S_{2n-1}) = FFT(C, \omega_{2n}^{2n-1})$
return $\frac{1}{n}(S_0, S_1, ..., S_{2n-1}) \rightarrow$ co-efficients of polynomial C(x)

5 Example

$$A(x) = 3 + x, B(x) = 2 + 2x$$

Find C(x), where C(x) = A(x)B(x).
Based on the information of A(x) and B(X), we know that C(x) has degree of 2 (z^2) and will have 3 co-efficents. The power of 2 above 3 is 4 (2^2), so we will have 4 points of unity (ω_4).

Let $a = (3, 1)$, which is the co-efficients of degree 0 and 1 of A(x).
Let $b = (2, 2)$, which is the co-efficients of degree 0 and 1 of B(x).

Now we run FFT with the inputs $(3, 1)$ and ω_4.

We know that $a_{even} = (3)$ which corresponds to the 0th degree of a. We also know that $a_{odd} = (1)$ which corresponds to the 1st degree of a. Both of these correspond to ω_2.

$$FFT(a_{even}, \omega_2) = (3, 3)$$

$$FFT(a_{odd}, \omega_2) = (1, 1)$$

The result of these (3,3) correspond to (s_0, s_1) and (1,1) correspond to (t_0, t_1).
We can now calculate (r_0, r_1, r_2, and r_3).
Additionally, ω^n corresponds to ($1, i, -1$, and $-i$) for n = 0,1,2, and 3 respectively.

Remember that:

- $r_j = s_j + \omega^j t_j$
- $r_{\frac{n}{2}+j} = s_j - \omega^j t_j$

So for a(x),

$$r_0 = 3 + 1 * 1 = 4$$

$$r_1 = 3 + i * 1 = 3 + i$$

$$r_2 = 3 - 1 * 1 = 2$$

$$r_3 = 3 - i * 1 = 3 - i$$

thus, $a(x) = (4, 3 + i, 2, 3 - i)$.
And for b(x),

$$r_0 = 2 + 1 = 4$$

$$r_1 = 2 + 2 * i = 2 + 2i$$

$$r_2 = 2 + 2 * (-1) = 0$$

$$r_3 = 2 + 2 * (-i) = 2 - 2i$$

thus, $b(x) = (4, 2+2i, 0, 2-2i)$. We can now multiply the two together to get c(x), so r_0 from a(x) * r_0 from b(x) to get r_0 for c(x).

$$r_0 = 4 * 4 = 16$$

$$r_1 = (3 + i) * (2 + 2i) = 4 + 8i$$

$$r_2 = 2 * 0 = 0$$

$$r_3 = (3 - i) * (2 - 2i) = 4 - 8i$$

This can now be run again on FFT, which looks like:

$$FFT((16, 4 + 8i, 0, 4 - 8i), \omega_4^3)$$

We now have $a_{even} = (16, 0)$ and $a_{odd} = (4 + 8i, 4 - 8i)$ we can then run FFT on both with ω^n $= \omega_4^{3*2} = \omega_4^2$

$$FFT(a_{even}, \omega_4^2) = (16, 16)$$

$$FFT(a_{odd}, \omega_4^2) = (8, 16i)$$

$$r_0 = 16 + \omega_2^2 * 8 = 24$$

$$r_1 = 16 + \omega_4^3 * 16i = 32$$

$$r_2 = 16 - \omega_2^2 * 8 = 8$$

$$r_3 = 16 - \omega_4^3 * 16i = 0$$

We then return $(\frac{1}{4}(24, 32, 8, 0))$ This gives us the coefficients for $C(x) = 6 + 8x + 2x^2$

6 Inverse Matrix Lemma

We can prove the lemma $(M_n(\omega_n))^{-1} = \frac{1}{n}M_n(\omega_n^{-1})$

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \dots & \omega_n^{(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{bmatrix}$$

Lemma: For any integer $n \geq 1$, ω is a n^{th} root of unity and ω is not equal to 1, then $1 + \omega + \omega^2 + \cdots + \omega^{n-1} = 0$

Proof: $(z-1)(z^{n-1} + z^{n-2} + \cdots + z + 1) = z^n - 1$

$= z^n + z^{n-1} + \cdots + z^2 + z$

$-z^{n-1} - z^{n-2} - \cdots - z^2 - z - 1$

Everything will cancel except the first and last term giving $z^n - 1$

If we set $z = \omega$ then we have $\omega^n - 1$ which equals 0

Proof: To show $\frac{1}{n}M_n(\omega_n)M_n(\omega_n^{-1}) = I$ where I is the identity matrix

Diagonal: (k,k) entry of $M_n(\omega_n)*M_n(\omega_n^{-1})$ $(1, \omega_n^k, \omega_n^{2k}, \dots, \omega_n^{(n-1)k})*(1, \omega_n^{-k}, \omega_n^{-2k}, \dots, \omega_n^{-(n-1)k})$

$= 1 + 1 + \cdots + 1 = n$

Off-diagonal: (k,j) entry $(k \neq j)$ $(1, \omega_n^k, \omega_n^{2k}, \dots, \omega_n^{(n-1)k}) * (1, \omega_n^{-j}, \omega_n^{-2j}, \dots, \omega_n^{-(n-1)j})$

$= 1 + \omega_n^{k-j} + \omega_n^{2(k-j)} + \cdots + \omega_n^{(n-1)(k-j)}$

if we set $\omega = \omega_n^{k-j}$, we have a n^{th} root of unity and $\omega \neq 1$ since $k \neq j$

if we then apply the previous lemma $1 + \omega + \omega^2 + \cdots + \omega^{n-1} = 0$ then entry $(k,j) = 0$

This proves that all off-diagonal entries are 0

7 Dynamic Programming

-Fibonacci numbers

$$F_0 = 0, F_1 = 1$$

for any $n > 1$

$$F_n = F_{n-1} + F_{n-2}$$

We could solve this with a natural recursive algorithm

Fib(n)

if $n = 0$, return 0

if $n = 1$, return 1

if $n > 1$, return Fib(n-1) + Fib(n-2)

This produces an algorithm with run time

$$T(n) = T(n-1) + T(n-2) + O(1)$$

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0$$

$$F_1 = 1$$

So we have $T(0) = O(1)$ and $T(1) = O(1)$

$T(n) \geq F_n$ therefore we have exponential time

Because we have to calculate the previous Fib(n) multiple times per call we can take advantage of this using dynamic programming. If we were to store these Fib(n) into a table we could create an algorithm that runs in O(n) time.