

Session 2

Corbet Elkins

October 23, 2026

Purdue University

Questions over lecture content?

See title

Lemma

Any n degree polynomial is uniquely determined by its value on $n + 1$ points of $\mathbb{C}$.

¹Based on Prof. Quanrud's lecture notes

Fast Fourier Transform¹

Lemma

Any n degree polynomial is uniquely determined by its value on $n + 1$ points of $\mathbb{C}$.

Fact

Any n degree polynomial, which is not the 0 polynomial, has at most n zeros in $\mathbb{C}$.

¹Based on Prof. Quanrud's lecture notes

Fast Fourier Transform¹

Lemma

Any n degree polynomial is uniquely determined by its value on $n + 1$ points of $\mathbb{C}$.

Fact

Any n degree polynomial, which is not the 0 polynomial, has at most n zeros in $\mathbb{C}$.

Proof.

Suppose $f \neq g$ are n degree polynomials, which agree on $a_0, \dots, a_n \in \mathbb{C}$, then $(f - g)(a_i) = 0$ for all i . Thus, $f - g = 0$ by the fact above, and $f = g$, a contradiction. $\square$

¹Based on Prof. Quanrud's lecture notes

Fast Fourier Transform

Motivation

Suppose we have two $n - 1$ degree polynomials f, g , and we want to compute $f \cdot g$. If we know f and g on $2n$ points, $a_1 \dots a_{2n}$, then by the previous lemma, we can determine $f \cdot g$ by $f(a_i) \cdot g(a_i)$.

Fast Fourier Transform

Motivation

Suppose we have two $n - 1$ degree polynomials f, g , and we want to compute $f \cdot g$. If we know f and g on $2n$ points, $a_1 \dots a_{2n}$, then by the previous lemma, we can determine $f \cdot g$ by $f(a_i) \cdot g(a_i)$.

Question

What points should we use?

Fast Fourier Transform

Motivation

Suppose we have two $n - 1$ degree polynomials f, g , and we want to compute $f \cdot g$. If we know f and g on $2n$ points, $a_1 \dots a_{2n}$, then by the previous lemma, we can determine $f \cdot g$ by $f(a_i) \cdot g(a_i)$.

Question

What points should we use?

Most natural is the roots of unity $\omega_{n'}^j$, which are the elements such that $(\omega_{n'}^j)^{n'} = 1$. For example 1 is always a root of unity, and -1 is if n' is even.

Additionally, we have $\omega_n^j = e^{2\pi ji/n}$ by Euler's formula

Fast Fourier Transform

The n th discrete Fourier transform (for n as a power of 2) takes a polynomial $p(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$, and returns $(p(1), p(\omega_n), p(\omega_n^2), \dots, p(\omega_n^{n-1}))$, we denote this as $F_n p$.

Exploiting the fact that $\omega_{2n}^2 = \omega_n$ (why?) we can build a recursive algorithm for computing the n th discrete Fourier transform.

Fast Fourier Transform

$$(F_{2n}p)_k = a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1}$$

Fast Fourier Transform

$$\begin{aligned}(F_{2n}p)_k &= a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1} \\ &= (a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{2n-2}) \\ &\quad + (a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1})\end{aligned}$$

Fast Fourier Transform

$$\begin{aligned}(F_{2n}p)_k &= a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1} \\&= (a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{2n-2}) \\&\quad + (a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1}) \\&= (a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1}) \\&\quad + \omega_{2n}^k(a_1 + a_3\omega_n^k + \cdots + a_{2n-1}(\omega_n^k)^{n-1})\end{aligned}$$

Fast Fourier Transform

$$\begin{aligned}(F_{2n}p)_k &= a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1} \\&= (a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{2n-2}) \\&\quad + (a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1}) \\&= (a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1}) \\&\quad + \omega_{2n}^k(a_1 + a_3\omega_n^k + \cdots + a_{2n-1}(\omega_n^k)^{n-1}) \\&= (F_n p_{\text{even}})_{k \bmod n} + \omega_{2n}^k (F_n p_{\text{odd}})_{k \bmod n}\end{aligned}$$

Fast Fourier Transform

$$\begin{aligned}(F_{2n}p)_k &= a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1} \\&= (a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{2n-2}) \\&\quad + (a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1}) \\&= (a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1}) \\&\quad + \omega_{2n}^k(a_1 + a_3\omega_n^k + \cdots + a_{2n-1}(\omega_n^k)^{n-1}) \\&= (F_n p_{\text{even}})_{k \bmod n} + \omega_{2n}^k (F_n p_{\text{odd}})_{k \bmod n}\end{aligned}$$

This can be implemented as a recursive divide and conquer algorithm.

Fast Fourier Transform

$$\begin{aligned}(F_{2n}p)_k &= a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1} \\&= (a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{2n-2}) \\&\quad + (a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1}) \\&= (a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1}) \\&\quad + \omega_{2n}^k(a_1 + a_3\omega_n^k + \cdots + a_{2n-1}(\omega_n^k)^{n-1}) \\&= (F_n p_{\text{even}})_{k \bmod n} + \omega_{2n}^k (F_n p_{\text{odd}})_{k \bmod n}\end{aligned}$$

This can be implemented as a recursive divide and conquer algorithm.

Question

What's the runtime?

Fast Fourier Transform

$$\begin{aligned}(F_{2n}p)_k &= a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1} \\&= (a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{2n-2}) \\&\quad + (a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1}) \\&= (a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1}) \\&\quad + \omega_{2n}^k(a_1 + a_3\omega_n^k + \cdots + a_{2n-1}(\omega_n^k)^{n-1}) \\&= (F_n p_{\text{even}})_{k \bmod n} + \omega_{2n}^k (F_n p_{\text{odd}})_{k \bmod n}\end{aligned}$$

This can be implemented as a recursive divide and conquer algorithm.

Question

What's the runtime?

$$T(n) = 2T(n/2) + O(n)$$

Fast Fourier Transform

$$\begin{aligned}(F_{2n}p)_k &= a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1} \\&= (a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{2n-2}) \\&\quad + (a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1}) \\&= (a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1}) \\&\quad + \omega_{2n}^k(a_1 + a_3\omega_n^k + \cdots + a_{2n-1}(\omega_n^k)^{n-1}) \\&= (F_n p_{\text{even}})_{k \bmod n} + \omega_{2n}^k (F_n p_{\text{odd}})_{k \bmod n}\end{aligned}$$

This can be implemented as a recursive divide and conquer algorithm.

Question

What's the runtime?

$$T(n) = 2T(n/2) + O(n), \quad O(n \log n)$$

Inverse Fourier Transform

Now that we have calculated the Fourier transform, we want some way to recover the original polynomial, i.e. we want to find the inverse of a Fourier transform.

Inverse Fourier Transform

Now that we have calculated the Fourier transform, we want some way to recover the original polynomial, i.e. we want to find the inverse of a Fourier transform.

Definition

Let $F_n^* p = (p(1), p(\omega_n^{-1}), p(\omega_n^{-2}), \dots, p(\omega_n^{1-n}))$

Note that as $\omega_n^k = \omega_n^{n+k}$, these are the same values we have already calculated in the Fourier transform.

Fact

For all $a \in \mathbb{C}^n$, $F_n^* F_n p = np$

Inverse Fourier Transform

Lemma

For all $a \in \mathbb{C}^n$ $F_n^* F_n a = na$

Proof.

$$(F_n^* F_n a)_h = \sum_{j=0}^{n-1} (F_n a)_j (\omega_n^{-h})^j = \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} a_k (\omega_n^j)^k \right) (\omega_n^{-h})^j$$



Inverse Fourier Transform

Lemma

For all $a \in \mathbb{C}^n$ $F_n^* F_n a = na$

Proof.

$$\begin{aligned}(F_n^* F_n a)_h &= \sum_{j=0}^{n-1} (F_n a)_j (\omega_n^{-h})^j = \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} a_k (\omega_n^j)^k \right) (\omega_n^{-h})^j \\ &= \sum_{k=0}^{n-1} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j\end{aligned}$$

□

Inverse Fourier Transform

Lemma

For all $a \in \mathbb{C}^n$ $F_n^* F_n a = na$

Proof.

$$\begin{aligned}(F_n^* F_n a)_h &= \sum_{j=0}^{n-1} (F_n a)_j (\omega_n^{-h})^j = \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} a_k (\omega_n^j)^k \right) (\omega_n^{-h})^j \\&= \sum_{k=0}^{n-1} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j \\&= na_h + \sum_{k \neq h} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j\end{aligned}$$

□

Inverse Fourier Transform

Lemma

For all $a \in \mathbb{C}^n$ $F_n^* F_n a = na$

Proof.

$$\begin{aligned}(F_n^* F_n a)_h &= \sum_{j=0}^{n-1} (F_n a)_j (\omega_n^{-h})^j = \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} a_k (\omega_n^j)^k \right) (\omega_n^{-h})^j \\&= \sum_{k=0}^{n-1} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j \\&= na_h + \sum_{k \neq h} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j = na_h + \sum_{k \neq h} a_k \frac{1 - (\omega_n^{k-h})^n}{1 - \omega_n^{k-h}}\end{aligned}$$

□

Inverse Fourier Transform

Lemma

For all $a \in \mathbb{C}^n$ $F_n^* F_n a = na$

Proof.

$$\begin{aligned}(F_n^* F_n a)_h &= \sum_{j=0}^{n-1} (F_n a)_j (\omega_n^{-h})^j = \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} a_k (\omega_n^j)^k \right) (\omega_n^{-h})^j \\&= \sum_{k=0}^{n-1} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j \\&= na_h + \sum_{k \neq h}^{n-1} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j = na_h + \sum_{k \neq h}^{n-1} a_k \frac{1 - (\omega_n^{k-h})^n}{1 - \omega_n^{k-h}} \\&= na_h\end{aligned}$$

□

Putting it together

Theorem

For any two polynomials f, g of degree at most n , we can calculate the $f \cdot g$ in $n \log n$ time. Let $r = f \cdot g$

Putting it together

Theorem

For any two polynomials f, g of degree at most n , we can calculate the $f \cdot g$ in $n \log n$ time. Let $r = f \cdot g$

$$(f, g) \xrightarrow{O(n \log n)} (F_{2n}f, F_{2n}g) \xrightarrow{O(n)} F_{2n}r \xrightarrow{O(n \log n)} F_{2n}^* F_{2n}r = 2n \cdot r$$

Note this general idea is used to get a multiplication algorithm that takes $O(n \log n \log \log n)$ time.

Problem

Suppose you have two sets of positive integers, X, Y whose elements are between 0 and M for some fixed M . Design and analyze an $O(M \log M)$ algorithm to compute the unique sums of X, Y , that is all z such that $z = x + y$ for exactly one pair of $(x, y) \in X \times Y$.

Problem

Design an algorithm, as fast as possible, to multiply n degree-1 polynomials:

$$a_1 + b_1x, a_2 + b_2x, \dots, a_n + b_nx$$