CSCI3160 Design and Analysis of Algorithms (2025 Fall)

Divide and Conquer: Fast Fourier Transform

Instructor: Xiao Liang

Department of Computer Science and Engineering
Chinese University of Hong Kong

Motivation: Multiply Polynomials Faster

We have seen divide-and-conquer speeding up many tasks. Next target: polynomials.
Example:

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4.$$

More generally, for degree-d polynomials

$$A(x) = \sum_{i=0}^{d} a_i x^i, \qquad B(x) = \sum_{i=0}^{d} b_i x^i,$$

their product $C(x) = A(x)B(x) = \sum_{k=0}^{2d} c_k x^k$ with

$$c_k = \sum_{i=0}^{k} a_i \, b_{k-i} \quad (\text{treat } a_i, b_i = 0 \text{ if } i > d).$$

Naive time: $\Theta(d^2)$. Can we do better?

Key Idea: Two Representations of a Polynomial

Two equivalent representations of $A(x)$:

1. Coefficients: $(a_0, a_1, \ldots, a_d)$
2. Values at $d+1$ points (aka point-value pairs):

$$(x_0, A(x_0)),\ (x_1, A(x_1)),\ \ldots,\ (x_d, A(x_d))$$

The above is due to a simply fact for polynomials:

Fact: A degree-d polynomial is uniquely determined by its values at *any* $d+1$ distinct points.

Think: How to prove it? (Proof: view coefficients as the variable you want to solve in a linear equation system.)

Why point-value representation?

Why are we interested in point-value representation?

- If $C(x) = A(x)B(x)$ has degree $2d$, then for any point z,

$$C(z) = A(z) \cdot B(z).$$

  Thus in the *value* representation, multiplication is linear time, i.e., just $(2d + 1) = O(d)$ products.

Evaluate–Multiply–Interpolate Paradigm

The discussion so far inspires the following idea for polynomial multiplication:

Input: coefficients of $A(x), B(x)$ of degree d.

1. Selection: pick points $x_0, \ldots, x_{n-1}$, where $n = 2d + 1$.
2. Evaluation: compute $A(x_k)$ and $B(x_k)$ for all $k \in \{0, 1, \ldots, n-1\}$.
3. Pointwise multiply: $C(x_k) = A(x_k)B(x_k)$ for all $k \in \{0, 1, \ldots, n-1\}$.
4. Interpolation: recover $C(x)$ from $\{(x_k, C(x_k))\}_{k \in \{0,1,\ldots,n-1\}}$.

Bottlenecks: how to perform fast Evaluation and Interpolation?

Baseline vs. Goal

Evaluating a degree-n polynomial at one point costs at least $O(n)$ time[1]. At n points, the baseline is $\Theta(n^2)$.

The key intuition behind Fast Fourier Transform: if our goal is to represent a degree-n polynomial by its values at n points, we are free to choose which points to use. So is it possible to pick a "highly structured" set of points that makes evaluation cheaper? By choosing points with special algebraic structure, we might be able to exploit these structures to reduce the total cost from the naive $\Theta(n^2)$.

Spoiler: Yes, we can. FFT successfully implements the above intuition using the set of n-th roots of unity as the evaluation points, reducing the total cost from $\Theta(n^2)$ to $\Theta(n \log n)$.

[1] In fact, achieving $O(n)$ is non-trivial. We need to use Horner's method.

Even-Odd Split

Let us decompose a given polynomial in a special manner: (assume for convenience that n is even):

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_n x^n \\ &= \left(a_0 + a_2 x^2 + a_4 x^4 + \ldots + a_n x^n\right) + \left(a_1 x + a_3 x^3 + a_5 x^5 + \ldots + a_{n-1} x^{n-1}\right) \\ &= \left(a_0 + a_2 x^2 + a_4 x^4 + \ldots + a_n x^n\right) + x \cdot \left(a_1 + a_3 x^2 + a_5 x^4 + \ldots + a_{n-1} x^{n-2}\right) \\ &= A_{even}(x^2) + x \cdot A_{odd}(x^2), \end{aligned}$$

where

$$\begin{aligned} A_{even}(x) &= a_0 + a_2 x^1 + a_4 x^3 + \ldots + a_n x^{n/2} \\ A_{odd}(x) &= a_1 + a_3 x^1 + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1} \end{aligned}$$

Observation: that $A_{even}(x)$ collects all even coefficients of $A(x)$, and A_{odd} collects all odd coefficients of $A(x)$. Both of them a of degree $\leq n/2$.

Even-Odd Split: Example

Let's see a concrete example:

$$\begin{aligned} A(x) &= 3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 \\ &= \left(3 + 6x^2 + x^4\right) + \left(4x + 2x^3 + 10x^5\right) \\ &= \left(3 + 6x^2 + x^4\right) + x \cdot \left(4 + 2x^2 + 10x^4\right) \\ &= A_{even}(x^2) + x \cdot A_{odd}(x^2), \end{aligned}$$

where

$$\begin{aligned} A_{even}(x) &= 3 + 6x + x^2 \\ A_{odd}(x) &= 4 + 2x + 10x^2 \end{aligned}$$

Divide-and-Conquer via Even–Odd Split

Split $A(x)$ into even and odd powers:

$$A(x) = A_{even}(x^2) + x\,A_{odd}(x^2),$$

where A_{even} collects even-indexed coefficients and A_{odd} collects odd-indexed coefficients.

If we evaluate at paired points $\pm x_i$, computations overlap:

$$A(x_i) = A_{even}(x_i^2) + x_i A_{odd}(x_i^2), \quad A(-x_i) = A_{even}(x_i^2) - x_i A_{odd}(x_i^2).$$

Thus, n evaluations at $\{\pm x_0, \ldots, \pm x_{n/2-1}\}$ reduce to evaluations of A_{even} and A_{odd} at $\{x_0^2, \ldots, x_{n/2-1}^2\}$ (two subproblems of size $n/2$) plus linear-time combination.

Recurrence and the Obstacle (1/2)

So far, we've seen that if we choose a special set $S = \{\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}\}$, and:

- **Original problem:** evaluate $A(x)$ on all the points in S.

This problem can be reduced to first solve the same problem for $S' = \{x_0^2, x_1^2, \ldots, x_{n/2-1}^2\}$, i.e.,

- **Reduced problem:** evaluate $A(x)$ on all the points of S'.

Two important observations:

- $|S|' = |S|/2$
- Given the answer to the **reduced problem**, we can solve the **original problem** in $O(n)$ time. (Think: convince yourself this is true, using the equations in the previous slide)

Recurrence and the Obstacle (1/2)

If this reduction could recurse, we would get

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

which solves to $T(n) = O(n \log n)$.

Obstacle: the plus–minus trick only works to reduce S to S'. We cannot reuse the idea to reduce S' further to a S'' of half size. So, the recursion got stuck at S'.

Solution: Let's utilize complex numbers.

Utilizing Complex Numbers (A Toy Example)

Set $S = \{1, -1, i, -i\}$, where i is the **Imaginary unit**, i.e., $i = \sqrt{-1}$.

This S supports two layer of recursion:

- $S' = \{1, -1\}$ (obtained by squaring each element in S)
- $S'' = \{1\}$ (obtained by squaring each element in S')

That is, if we want to evaluate a degree-3 polynomial $A(x)$ on every point of S, we can first do it at S', which can further be reduced to doing the evaluation on S''.

This implement the above recurse $T(n) = 2T(n/2) + O(n)$ for the special case of $n = 4$.

Important: Using a high-dimensional analog of i, we can generalize this idea to any n.

Recalling Complex Numbers: Fundamental Theorem of Algebra

Statement Every polynomial of degree $n \geq 1$ with complex coefficients has *exactly n roots in* $\mathbb{C}$, *counted with multiplicity*. Equivalently, for

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \cdots + a_1 z + a_0 \quad (a_n \neq 0),$$

there exist $\zeta_1, \ldots, \zeta_n \in \mathbb{C}$ such that

$$p(z) = a_n \prod_{k=1}^{n} (z - \zeta_k),$$

where each root appears according to its multiplicity.

Fun Facts: this theorem was first established by Carl Friedrich Gauss in 1799. During his life, Gauss offered not less than four different proofs to the theorem, covering a timespan of fifty years spanning his entire adult life.

Recalling Complex Numbers: Roots of Unity

Solutions to $x^2 = 1$?

- $x = 1$ and $x = -1$

Solutions to $x^3 = 1$?

- $x = 1$, $x = \cos(\frac{2\pi}{3}) + i \cdot \sin(\frac{2\pi}{3})$, and $x = \cos(\frac{4\pi}{3}) + i \cdot \sin(\frac{4\pi}{3})$

In general, there are n solutions to $x^n = 1$, they are:

- $x = \cos(\frac{2k\pi}{n}) + i \cdot \sin(\frac{2k\pi}{n})$, $\forall k \in \{0, 1, \ldots, n-1\}$

They are called the n-th roots of unity

Recalling Complex Numbers: Roots of Unity

In general, there are n *distinct* solutions to $x^n = 1$, they are:

- $x = \cos(\frac{2k\pi}{n}) + i \cdot \sin(\frac{2k\pi}{n})$, $\forall k \in \{0, 1, \ldots, n-1\}$

We denote $\omega_n = \cos(\frac{2\pi}{n}) + i \cdot \sin(\frac{2\pi}{n})$, then

$$\omega_n^0 = \cos(\frac{2 \cdot 0 \cdot \pi}{n}) + i \cdot \sin(\frac{2 \cdot 0 \cdot \pi}{n}) = 1$$
$$\omega_n^1 = \cos(\frac{2 \cdot 1 \cdot \pi}{n}) + i \cdot \sin(\frac{2 \cdot 1 \cdot \pi}{n})$$
$$\omega_n^2 = \cos(\frac{2 \cdot 2 \cdot \pi}{n}) + i \cdot \sin(\frac{2 \cdot 2 \cdot \pi}{n})$$
$$\ldots$$
$$\omega_n^{n-1} = \cos(\frac{2 \cdot (n-1) \cdot \pi}{n}) + i \cdot \sin(\frac{2 \cdot (n-1) \cdot \pi}{n})$$

Therefore, it holds that $\omega_n^k = \cos(\frac{2k\pi}{n}) + i \cdot \sin(\frac{2k\pi}{n})$ for all $k \in \{0, 1, \ldots, n-1\}$.

Thus, the n distinct n-th roots of unity (i.e., solutions to $x^n = 1$) can be equivalently written as:

$$\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-2}, \omega_n^{n-1},$$

where $\omega_n = \cos(\frac{2\pi}{n}) + i \cdot \sin(\frac{2\pi}{n})$. This ω_n is sometimes referred to as the **principal**[2] n-th root of unity. (Note that there are n different n-th roots of unity. But there is only one **principal** n-th root of unity.)

Pictorially, the n-th roots of unity are equally spaced on the unit circle, starting from $\omega_n^0 = 1$, with an angle of $2\pi/n$ between adjacent ω_n^k and ω_n^{k+1} for all $k \in \{0, 1, \ldots, n-1\}$.

[draw the picture on the whiteboard]

[2] We use the word "principal" to refer to the root of unity having smallest positive complex argument. But this is an abuse of the term. See the discussion on Wolfram MathWorld.

Recalling Complex Numbers: Euler's Formula

This is also a good place to recall Euler's formula: for all real number θ,

$$e^{i\cdot\theta} = \cos(\theta) + i \cdot \sin(\theta).$$

Now, let's memorize the following three formats of the n-th roots of unity: for all $k \in \{0, 1, \ldots, n-1\}$:

$$\cos(\frac{2 \cdot k \cdot \pi}{n}) + i \cdot \sin(\frac{2 \cdot k \cdot \pi}{n}) = e^{\frac{2 \cdot k \cdot \pi}{n}} = \omega_n^k,$$

where recall that ω_n is the principal n-th root of unity.

Important Facts about Roots of Unity

The following facts will be crucial to the design of FFT:

- **Fact 1:** For all $n \in \mathbb{N}$, there are exactly n **distinct** n-th roots of unity, which we denote by

$$U_n = \{\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}\}.$$

- **Fact 2:** For all $n \in \mathbb{N}$ that is even, it holds that $\omega_n^{k+\frac{n}{2}} = -\omega_n^k$ for all $k \in \{0, 1, \ldots, \frac{n}{2} - 1\}$.
- **Fact 3:** For all $n \in \mathbb{N}$ that is even, squaring each element of U_n will generate the set of $U_{\frac{n}{2}}$, i.e., the set of $\frac{n}{2}$-th roots of unity. More explicitly, it hols that $\omega_n^{2k} = \omega_{n/2}^{k \bmod (n/2)}$

[We will not discuss the proofs of these three facts. You should be able to prove them yourself, as they involve only high-school–level complex numbers.]

FFT Pseudocode I

Input. Polynomial $A(x)$ of degree at most $n-1$, where $n = 2^m$ for some $m \geq 0$.
Goal. Evaluate $A(x)$ on all points in $U_n = \{\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}\}$ (i.e., the n-th roots of unity)

Base case: If $n = 1$, then $\deg A(x) \leq 0$, so $A(x) = a_0$ for a constant a_0. Output $[a_0]$.
[Time complexity: $O(1)$]

Recursive case: Assume $n \geq 2$.

1. Split $A(x)$ into even and odd parts: $A(x) = A_{even}(x^2) + xA_{odd}(x^2)$. [Time complexity: $O(n)$]
2. Make **two recursive FFT calls (of size $n/2$)** (Think: convince yourself this is true!) to evaluate both $A_{even}(y)$ and $A_{odd}(y)$ on all the points in $U_{n/2} = \{\omega_{n/2}^0, \omega_{n/2}^1, \ldots, \omega_{n/2}^{n/2-1}\}$. In particular, This step yields the following values

$$A_{even}(\omega_{n/2}^0), A_{even}(\omega_{n/2}^1), \ldots, A_{even}(\omega_{n/2}^{n/2-1})$$
$$A_{odd}(\omega_{n/2}^0), A_{odd}(\omega_{n/2}^1), \ldots, A_{odd}(\omega_{n/2}^{n/2-1})$$

[Time complexity: $2f(n/2)$]

FFT Pseudocode II

3. $\forall k \in \{0, 1, \ldots, n-1\}$, compute

$$A(\omega_n^k) = A_{even}(\omega_n^{2k}) + \omega_n^k A_{odd}(\omega_n^{2k}),$$

using the values of $A_{even}(y)$ and $A_{odd}(y)$ at $\omega_n^{2k} = \omega_{n/2}^{k \bmod (n/2)}$ (this is Fact 3!) computed in the previous step. [Time complexity: $O(n)$]

4. Output the values $\left(A(\omega_n^k)\right)_{k=0}^{n-1}$ in order of increasing k.
[Time complexity: $O(n)$]

This finishes the description of the FFT algorithm.

Time complexity analysis:

$$f(n) = \begin{cases} O(1) & n = 1 \text{ (based case)} \\ 2 \cdot f(n/2) + O(n) & n \geq 2 \end{cases},$$

which solves to $f(n) = O(n \log n)$.

Interpolation as "Inverse FFT" I

The FFT algorithm, in $O(n \log n)$ time:

- **Input:** polynomial $A(x)$ in its coefficients representation:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^{n-1}.$$

- **Output:** pair-value representation of $A(x)$ at all the n-th roots of unity:

$$\left((\omega_n^0, A(\omega_n^0)), \quad (\omega_n^1, A(\omega_n^1)), \quad , \ldots, \quad (\omega_n^{n-1}, A(\omega_n^{n-1})), \right)$$

Question: Can we convert the pair-value represention back to the coefficients representation (aka the task of "interpolation") in $O(n \log n)$ time?

More formally:

Interpolation as "Inverse FFT" II

- **Input:** pair-value representation of $C(x)$ at all the n-th roots of unity:

$$\big((\omega_n^0, C(\omega_n^0)),\ \ (\omega_n^1, C(\omega_n^1)),\ \ ,\ldots,\ \ (\omega_n^{n-1}, C(\omega_n^{n-1})),\big).$$

  You are guaranteed that these points come from a polynomial $C(x)$ of degree at most n.

- **Input:** The unique polynomial $C(x)$ in its coefficients representation:

$$C(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^{n-1}.$$

Solution: This interpolation task can also be done in $O(n \log n)$ time, using an algorithm called "inverse FFT." It can be roughly understood as "running FFT in the reverse order." We won't talk about the details during lecture time. Interested students could refer to Section 2.6.3 of [DPV]. [This part will not appear in quizzes/exams.]

Evaluate–Multiply–Interpolate Paradigm, Finalized

Recall the Evaluate–Multiply–Interpolate Paradigm we talked about earlier for polynomial multiplication. Now, we can complete it using FFT

Input: coefficients of $A(x), B(x)$ of degree d.

1. Selection: pick points $x_0, \ldots, x_{n-1}$, where $n = 2d + 1$.
   [in time $O(n)$]
2. Evaluation: compute $A(x_k)$ and $B(x_k)$ for all $k \in \{0, 1, \ldots, n-1\}$.
   [using FFT, in time $O(n \log n)$]
3. Pointwise multiply: $C(x_k) = A(x_k)B(x_k)$ for all $k \in \{0, 1, \ldots, n-1\}$.
   [in time $O(n)$]
4. Interpolation: recover $C(x)$ from $\{(x_k, C(x_k))\}_{k \in \{0,1,\ldots,n-1\}}$.
   [using "inverse FFT", in time $O(n \log n)$]

Congratulations! You now know how to multiply two degree-n polynomials in time $O(n \log n)$.

A Note on Terminology

In this lecture, we use "FFT" to mean the algorithm that converts a polynomial from its coefficient representation to its point-value representation, and "inverse FFT" for the reverse procedure.

This terminology is not universal. While our usage is common (especially in the CS community), some authors—more often in mathematics—swap the names, calling our "FFT" the "inverse FFT," and vice versa.