Fast Fourier Transform: A Walkthrough

Tianchen Liu

1 Motivation

Overarching Goal: we want to do polynomial multiplication quickly.

Given two polynomials in coefficient representation. The brute force to multiply the two takes $O(n^2)$.

From CS 70 we know there are two ways to represent a polynomial: the *coefficient representation* and *value representation*.

Review: The *coefficient representation* of a degree d polynomial is

$$f(x) = c_0 + c_1 x + \cdots + c_d x^d$$

The *value representation* of a degree d polynomial is $d+1$ points that the polynomial passes through. Because we know that given $d+1$ points, there is a unique degree d polynomial that passes through the given $d+1$ points (Proved in CS 70).

What we could do is to first evaluate the polynomials in a sufficient amount of points, and then multiply the two polynomials at those points. For example, if we know that polynomial f passes through $(x_1, f(x_1))$, $(x_2, f(x_2))$, and we know polynomial g passes through $(x_1, g(x_1))$, $(x_2, g(x_2))$, then we already know two points $f \circ g$ passes through - namely $(x_1, f(x_1)g(x_1))$, $(x_2, f(x_2)g(x_2))$.

Once we know f and g's values at the same $d+1$ points, we can just multiply them one by one to to know $d+1$ points that the result polynomial $(f \circ g)$ pass through (this is an $O(n)$ operation). That's enough values to recover the result polynomial! Transform the result polynomial from the value representation back to coefficient representation via interpolation.

Recap of our procedure:

1. Evaluate degree d polynomials f and g at the same x-values $(x_1 \dots x_n)$, where $n \geq d+1$.
2. Multiply the n points to get $\{f \circ g(x_i)\}_{i=1}^n$.
3. If needed, interpolate $f \circ g$ back to the coefficient representation.

Multiplying the two polynomials in value representation is $O(n)$ as described above, so this procedure is bottlenecked by the speed of evaluation/interpolation[1].

The brute force way to do evaluation is still $O(n^2)$ - you evaluate the polynomial at an arbitrary n points, each evaluation takes $O(n)$.

Is there a better way to evaluate? That's where FFT comes in.

2 A Trick

Let n be odd, $A(x) = c_0 + c_1 x + \cdots + c_n x^n$.

Define $A_e(x) = c_0 + c_2 x + c_4 x^2 + \cdots + c_{n-1} x^{\frac{n-1}{2}}$ and $A_o(x) = c_1 + c_3 x + c_5 x^2 + \cdots + c_n x^{\frac{n-1}{2}}$.

Fact:

$$A(x) = A_e(x^2) + x A_o(x^2)$$

[1] We will only talk about evaluation in this note. Once you understood using `FFT` in evaluation, feel free to read the textbook for a clear and succinct explanation of interpolation using `FFT`.

The best way to convince yourself about this fact is just to try out some low degree (*e.g.* 4) polynomial and see for yourself.

If we just evaluate this by just grabbing any x-values, then we need to evaluate A_e and A_o at $n+1$ points - namely $x_1^2, x_2^2, \ldots, x_{n+1}^2$ - for this to work.

But what if we carefully select the x-values we want to evaluate such that the array of x-values has some *nice properties*? What if the x-values are paired - that is, we want to evaluate A at $x_1, -x_1, x_2, -x_2 \cdots$? In that case, we only need to evaluate A_e and A_o at $\frac{n+1}{2}$ points, because $x^2 = (-x)^2$, therefore $f(x^2) = f((-x)^2)$ for all functions f.

Yay! Using this trick we just sped up our computation by 2x because this trick only works in the first layer. Why only the first layer? Because this *nice property* of the x-values - they're paired - are gone in the next layer of recursion: we can't make sure $x_1^2, x_2^2 \cdots$ are also *paired*...

Or can we?

3 Roots of Unity

What if we select our x-values even more carefully? Now the goal is clear: we want to select an array of x-values that preserves this nice property of being paired in all layers of recursion. That is, no matter how many times we square the numbers in this array, the resulting array - now half the length - are still paired.

Theorem: *n-th roots of unity does the job, where n is a power of 2.*

Proof Sketch: We first prove that all even roots of unity have this property of being paired (1). Then we prove no matter how many times we square this array, the result is still an array of 2^k-th roots of unity (2) - meaning that this nice property is still there.

Claim 1: Let n be even, $S = \{w_n^0, w_n^1, \ldots, w_n^{n-1}\}$. $x \in S \implies -x \in S$.

Proof. Notice that

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

Therefore for all $k < \frac{n}{2}$:

$$w_n^{k+\frac{n}{2}} = w_n^k w_n^{\frac{n}{2}} = -w_n^k$$

□

Claim 2: Let n be even. The squares of the n-th roots of unity are the $\frac{n}{2}$-th roots of unity.

Proof. The proof is left as an exercise to the reader. □

4 Putting It All Together

Now we figured out everything: We first start out with a trick to speed up the computation a little bit. It only works in the first layer because the paired property of the x-values is lost in subsequent layers of recursion as we square the x-values. By carefully choosing the x-values to be n-th roots of unity, this nice

property is preserved no matter how many times we square them.

Algorithm 1: Fast Fourier Transform

Input: A, n

// A is the polynomial in coefficient representation.

Output: $A(w_n^0), A(w_n^1), \ldots, A(w_n^{n-1})$

// A evaluated at n-th roots of unity.

Evaluate A_e at $\frac{n}{2}$ roots of unity by calling `FFT`$(A_e, \frac{n}{2})$.
Evaluate A_o at $\frac{n}{2}$ roots of unity by calling `FFT`$(A_o, \frac{n}{2})$.
for $k = 0$ **to** $\frac{n}{2} - 1$ **do**
    $A(w_n^k) = A_e(w_{\frac{n}{2}}^k) + w_n^k A_o(w_{\frac{n}{2}}^k)$ // $w_{\frac{n}{2}}^k = (w_n^k)^2$
    $A(w_n^{k+\frac{n}{2}}) = A_e(w_{\frac{n}{2}}^k) - w_n^k A_o(w_{\frac{n}{2}}^k)$ // $w_n^{k+\frac{n}{2}} = -w_n^k$, $w_{\frac{n}{2}}^k = (-w_n^k)^2$

5 Time Complexity

Two calls to `FFT` with parameter halved + linear procedure.

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

By Master Theorem, the total time complexity is $O(n \log n)$.