Polynomial Multiplication via Fast Fourier Transforms

Let us learn about the Fast Fourier Transforms (FFT), and we'll see how it can be applied to efficiently solve the problem of multiplying two polynomials. The Fast Fourier Transform is a very famous algorithm (which is based on Divide-and-Conquer) that has tons of applications in areas like signal processing, speech recognition, and data compression, to name a few. We won't actually say too much about the Fourier transform in its full generality; instead, we will focus on seeing how it can be applied to obtain faster algorithms for polynomial multiplication.

1 Polynomial Multiplication

Suppose that we are given two polynomials of degree $n-1$:[1]

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ B(x) &= b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}. \end{aligned}$$

We wish to find the product of A and B, a polynomial C such that $C(x) = A(x)B(x)$. Suppose that we represent A as a vector of its n coefficients $a = (a_0, \ldots, a_{n-1})$, and likewise B with $b = (b_0, \ldots, b_{n-1})$. Then there is a straightforward algorithm for computing the coefficients of C, by simply expanding out the algebraic expression $A(x)B(x)$ and collecting like terms. This procedure takes $\Theta(n^2)$ operations, since we're essentially multiplying all pairs of coefficients from A and B. In the rest of this tutorial, we'll see a faster method of computing the product C.

1.1 Representing Polynomials

So far, we've assumed that a polynomial is represented by its list of coefficients. However, the following fact will suggest an alternative way of representing polynomials.

Fact 1. *A set of n point-value pairs (say in $\mathbb{R}^2$ or $\mathbb{C}^2$) uniquely determines a polynomial of degree $n-1$.*

This is essentially a generalization of the statement "given two points in the plane, there is a unique line that passes through them." Proving this is an exercise in linear algebra; given n point-value pairs, we can compute the coefficients of a polynomial passing through those points by solving a system of linear equations.[2]

This leads us to the *point-value representation* of a polynomial: Given points $x_0, \ldots, x_{n-1} \in \mathbb{C}$, a degree-$n-1$ polynomial $A(x)$ can be represented by the set

$$\{(x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_{n-1}, A(x_{n-1}))\}.$$

[1] For simplicity, we'll assume that n is a power of 2.

[2] This system of equations is defined using a Vandermonde matrix, which we'll see in more detail later.

Using this representation, it's now much easier to multiply two polynomials A and B, by simply multiplying their values on the sample points $x_0, \ldots, x_{n-1}$ pointwise:

$$\{(x_0, A(x_0)B(x_0)), (x_1, A(x_1)B(x_1)), \ldots, (x_{n-1}, A(x_{n-1})B(x_{n-1}))\}. \tag{$\star$}$$

Therefore, if we use the point-value representation for polynomials, then we can multiply two polynomials of degree $n-1$ using only $\Theta(n)$ arithmetic operations. However, there's still a slight problem: If $A(x)$ and $B(x)$ are both polynomials of degree $n-1$, then their product will be a polynomial $C(x) = A(x)B(x)$ of degree $n-1+n-1 = 2n-2$. But the result computed by $(\star)$ only contains n sample points; not the $2n-1$ points needed to represent a polynomial of degree $2n-2$. To fix this, we can represent A and B using an "extended" point-value representation, by using $2n$ point-value pairs, instead of just[3] n. Asymptotically, this doesn't affect our overall running time, since it only increases our input size by a constant factor of 2.

1.2 Plan of Attack

It's great that we can multiply polynomials in linear time, using their point-value representation, but we'd *really* like to do this using their coefficient representation. This gives us the outline of our algorithm for multiplying polynomials: Given two polynomials A and B in coefficient form, convert them to point-value form by *evaluating* them at $2n$ points, then multiply them pointwise in linear time, and finally convert the result back to coefficient form, via a process called interpolation.

In general, evaluating a polynomial A at a single point x takes $\Theta(n)$ operations, and since we have to evaluate $2n$ points, the evaluation step takes time $\Theta(n^2)$. So it seems like we still aren't doing any better than the ordinary multiplication algorithm.

The key point is that the evaluation step will take $\Theta(n^2)$ operations if we just choose any old points $x_0, \ldots, x_{2n-1}$. But for a particular choice of points, namely, the complex roots of unity, we'll see how we can evaluate A on that set of points using only $\Theta(n \log n)$ operations, using the fast Fourier transform. It'll also be possible to use the FFT to do the interpolation step, once we have the product of A and B in point-value form.

2 Complex Roots of Unity

A number $z \in \mathbb{C}$ is an *nth root of unity* if $z^n = 1$. The *principal* nth root of unity is $\omega_n = e^{\frac{2\pi i}{n}}$. For $n \geq 1$, the n nth roots of unity are $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$. For a bit of geometric intuition, the nth roots of unity form the vertices of a regular n-gon in the complex plane. We'll use a few basic properties of complex roots of unity.

Lemma 2 (Cancellation lemma)**.** *For integers $n \geq 0$, $k \geq 0$, $d > 0$, we have $\omega_{dn}^{dk} = \omega_n^k$.*

Proof.

$$\omega_{dn}^{dk} = \left(e^{\frac{2\pi i}{dn}}\right)^{dk} = e^{\frac{2\pi i k}{n}} = \omega_n^k. \qquad \square$$

[3]We use $2n$ point-value pairs instead of just $2n-1$, since if n is a power of 2, then so is $2n$. This will be useful later.

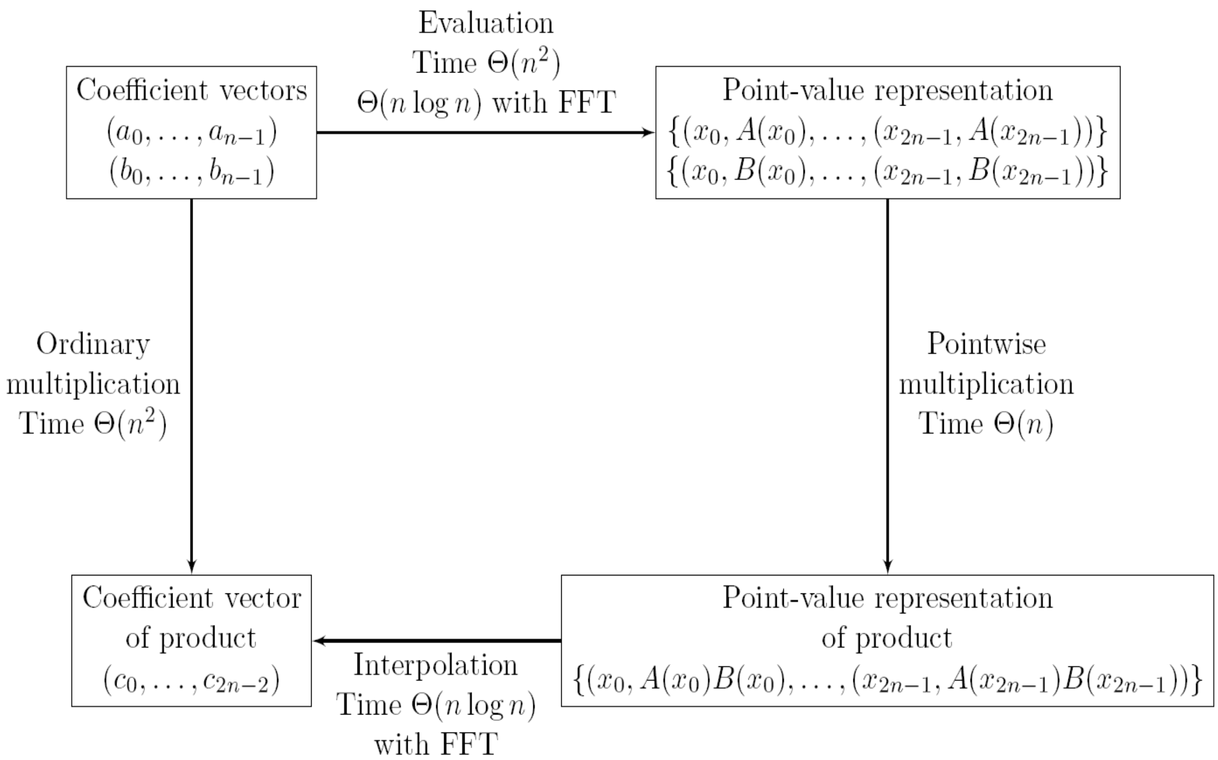

Figure 1: Outline of the approach to efficient polynomial multiplication using the fast Fourier transform.

Lemma 3 (Halving lemma)**.** *If $n > 0$ is even, then the squares of the n complex nth roots of unity are the $\frac{n}{2}$ complex $\frac{n}{2}$th roots of unity.*

Proof. For any nonnegative k, we have $(\omega_n^k)^2 = \omega_n^{2k} = \omega_{n/2}^k$.

Note that squaring all of the nth roots of unity gives us each $\frac{n}{2}$th root of unity twice, since $(\omega_n^{k+n/2})^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = (\omega_n^k)^2$. □

Lemma 4 (Summation lemma)**.** *If $n \geq 1$ and k is not divisible by n, then*

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.$$

Proof. Using the formula for the sum of a geometric series,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} = \frac{1^k - 1}{\omega_n^k - 1} = 0.$$

Since k is not divisible by n, we know that $\omega_n^k \neq 1$, so that the denominator is not zero. □

3 The Discrete Fourier Transform for Polynomial Evaluation

Now we are ready to define the discrete Fourier transform, and see how it can be applied to the problem of evaluating a polynomial on the complex roots of unity.

Definition 5. *Let $a = (a_0, \ldots, a_{n-1}) \in \mathbb{C}^n$. The discrete Fourier transform of a is the vector* $\mathrm{DFT}_n(a) = (\hat{a}_0, \ldots, \hat{a}_{n-1})$*, where*[4]

$$\hat{a}_k = \sum_{j=0}^{n-1} a_j e^{\frac{2\pi ikj}{n}} = \sum_{j=1}^{n-1} a_j \omega_n^{kj} \qquad 0 \leq k \leq n-1.$$

If you look closely, you can see that this definition is in fact exactly what we want. Let $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1}x^{n-1}$ be the polynomial whose coefficients are given by a. Then for $0 \leq k \leq n-1$, the polynomial A evaluated at the root of unity ω_n^k is exactly

$$A(\omega_n^k) = \sum_{j=0}^{n-1} a_j (\omega_n^k)^j = \hat{a}_k.$$

Here, the n is really the $2n$ from previous sections, however we will simply use n to denote the length of the vector a when describing the DFT to simplify matters.

So far though, we still haven't gained much, since applying this definition and directly computing the sum for each k takes $\Theta(n^2)$ operations. This is where the fast Fourier transform comes in: this will allow us to compute $\mathrm{DFT}_n(a)$ in time $\Theta(n \log n)$.

4 Fast Fourier Transform

The fast Fourier transform is an algorithm for computing the discrete Fourier transform of a sequence by using a divide-and-conquer approach.

As always, assume that n is a power of 2. Given $a = (a_0, \ldots, a_{n-1}) \in \mathbb{C}^n$, we have the polynomial $A(x)$, defined as above, which has n terms. We can also define two other polynomials, each with $\frac{n}{2}$ terms, by taking the even and odd coefficients of A, respectively:

$$\begin{aligned} A_e(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{\frac{n}{2}-1}, \\ A_o(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{\frac{n}{2}-1}. \end{aligned}$$

Then, for any $x \in \mathbb{C}$, we can evaluate A at x using the formula

$$A(x) = A_e(x^2) + x A_o(x^2). \tag{$*$}$$

So the problem of evaluating A at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to performing the following steps:

1. evaluating two polynomials of degree $\frac{n}{2} - 1$ at the points $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$, and
2. combining the results using $(*)$.

[4] In the context of signal processing or other areas, you've seen that DFT is defined in terms of ω_n^{-1} rather than ω_n. The form used here simplifies matters a bit for our specific application and the math is almost the same either way.

By the halving lemma, we only actually need to evaluate *Ae* and *Ao* at *n*/2 points, rather than at all *n* points. Thus, at each step, we divide the initial problem of size *n* into two subproblems of size *n*/2.

Now that we have the general idea behind how the FFT works, we can look at some pseudocode.

Algorithm 1 Recursive version of the fast Fourier transform.

```
function RECURSIVE-FFT(a)
    n = a.length
    if n == 1 then
        return a
    a_e = (a_0, a_2, ..., a_{n-2})
    a_o = (a_1, a_3, ..., a_{n-1})
    y_e = RECURSIVE-FFT(a_e)
    y_o = RECURSIVE-FFT(a_o)
    ω_n = e^{2πi/n}
    ω = 1
    for k = 0 to n/2 - 1 do
        y_k = (y_e)_k + ω(y_0)_k
        y_{k+n/2} = (y_e)_k - ω(y_o)_k
        ω = ωω_n.                                        ▷ ω = ω_n^{k+1}
    y = (y_0, ..., y_{n-1})
    return y
```

The pseudocode more or less follows the algorithmic structure outlined above. The main thing that we need to check is that the for loop actually combines the results using the correct formula. For $k = 0, 1, \dots, \frac{n}{2} - 1$, in the recursive calls we compute, by the cancellation lemma,

$$(y_e)_k = A_e(\omega_{n/2}^k) = A_e(\omega_n^{2k}),$$
$$(y_o)_k = A_o(\omega_{n/2}^k) = A_o(\omega_n^{2k}).$$

In the for loop, for each $k = 0, \dots, \frac{n}{2} - 1$, we compute

$$y_k = A_e(\omega_n^{2k}) + \omega_n^k A_o(\omega_n^{2k}) = A(\omega_n^k).$$

We also compute

$$\begin{aligned} y_{k+\frac{n}{2}} &= A_e(\omega_{n/2}^k) - \omega_n^k A_o(\omega_{n/2}^k) \\ &= A_e(\omega_n^{2k}) + \omega_n^{k+\frac{n}{2}} A_o(\omega_n^{2k}) \\ &= A_e(\omega_n^{2k+n}) + \omega_n^{k+\frac{n}{2}} A_o(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+\frac{n}{2}}). \end{aligned}$$

Therefore $y_k = A(\omega_n^k) = \hat{a}_k$ for all $0 \le k \le n-1$, and therefore this algorithm returns the correct result. Moreover, let $T(n)$ denote the running time of RECURSIVE-FFT(a) when a has length n. Since the algorithm makes two recursive calls on subproblems of size $\frac{n}{2}$, and uses $\Theta(n)$ steps to combine the results, the overall running time of this algorithm is

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n).$$

5 Inversion Formula and Interpolation via FFT

Now that, given the coefficients of a polynomial, we can evaluate it on the roots of unity by using the FFT, we want to perform the inverse operation, interpolating a polynomial from its values on the nth roots of unity, so that we can convert a polynomial from its point-value representation, back to its coefficient representation.

The first idea here will be to observe that $\mathrm{DFT}_n : \mathbb{C}^n \to \mathbb{C}^n$ is a linear map. This means that it can be written in terms of a matrix multiplication:

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{bmatrix}.$$

We can denote the matrix on the left-hand side of this equation by $M_n(\omega_n)$. This matrix has a special name in linear algebra, and is called a Vandermonde matrix. To solve the problem of performing the reverse of the discrete Fourier transform, and convert from the point-value representation to the coefficient representation of a polynomial, we simply need to find the inverse of this matrix. It turns out that the structure of this particular matrix makes it particularly easy to invert.

Theorem 6 (Inversion theorem)**.** *For $n \geq 1$, the matrix $M_n(\omega_n)$ is invertible, and*

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

Proof. The (j, j') entry of $M_n(\omega_n)$ is equal to $\omega_n^{jj'}$. Similarly, the (j, j') entry of $\frac{1}{n} M_n(\omega_n^{-1})$ is $\frac{1}{n} \omega_n^{-jj'}$. Therefore, the (j, j') entry of the product $\frac{1}{n} M_n(\omega_n^{-1}) M_n(\omega_n)$ is equal to

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{-kj} \omega_n^{kj'} = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}.$$

If $j' = j$, then this sum equals n, and so the entire expression evaluates to 1. On the other hand, if $j' \neq j$, then this sum equals 0, by the summation lemma. Therefore $\frac{1}{n} M_n(\omega_n^{-1}) M_n(\omega_n) = I_n$, the $n \times n$ identity matrix, and so $M_n(\omega_n)^{-1} = \frac{1}{n} M_n(\omega_n^{-1})$. $\square$

This key takeaway of this inversion formula is that enables us to invert the discrete Fourier transform by using the exact same algorithm as the fast Fourier transform, but with ω_n replaced with ω_n^{-1}, and then dividing the entire result by n. Therefore it is also possible to perform polynomial interpolation (from the complex roots of unity) in time $\Theta(n \log n)$.

Putting it all together, this demonstrates how it is possible to multiply two polynomials, by computing their values on the $2n$th roots of unity, doing pointwise multiplication, and interpolating the result to obtain the coefficients of the product, all in time $\Theta(n \log n)$, a significant improvement over the naive $\Theta(n^2)$ algorithm.