Fast Fourier Transform for Polynomial Multiplication

by

Kevin Kim

Math 4490 - Applied Math

Department of Mathematics & Statistics

Contents

1. **Introduction**

Naive approach polynomial multiplication is $O(n^2)$ where n is the number of terms. However, if we apply the Fast Fourier Transform to polynomial multiplication we can achieve $O(nlogn)$ performance. The following will detail the ideas behind this elegant algorithm.

2. Polynomials

A polynomial is a function A(x) of the form

$$A(x) = \sum_{j=0}^{n-1} a_j x^j.$$

The highest nonzero coefficient a_k of $A(x)$ is said to have **degree** k and any integer greater than k is considered the **degree-bound** of the polynomial. Important properties to note in polynomial multiplication is for polynomials $A(x), B(x)$ of degree bound n, $A(x)B(x) = C(x)$ is degree bound $2n-1$ and the degree of C is the sum of degrees of A and B.

2.1 Polynomial representations

There are two ways of uniquely representing polynomials. The first is one most familiar with namely **coefficient representation** $A(x) = \sum_{j=0}^{n-1} a_j x^j$. The other is **point-value representation** which consists of sampling a distinct degree-bound number of points that results in a vector of values. For example, a polynomial $A(x)$ of degree bound n in point-value representation is $\{A(x_0), A(x_1), ..., A(x_{n-1})\}$ where $x_1, x_2, ...x_{n-1}$ is all distinct. A question then natually arises: from point-value form, how will we know interpolating back to coeffiecient form will be valid? This must be answered since we will most likely want to evaluate our polynomial at points that have not been sampled.

Theorem. *Uniqueness of interpolating polynomials*
For any set $\{(x_0, y_0), (x_0, y_0), ..., (x_0, y_0)\}$ of n point-value pairs such that all x_k values are distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, ..., n-1$.

Proof. For a degree-bound n polynomial of the form

$$A(x) = \sum_{j=0}^{n-1} a_j x^j.$$

sampled at $x_1, x_2, ...x_{n-1}$ points can be represented as the following Vandermonde matrix:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

which is known to have the determinant

$$\prod_{0 \leq j < k \leq n-1} (x_k - x_j).$$

Since the points must be distinct, the determinant is non-zero and nonsingular thus able to uniquely solve for the coefficients given the point-value form with the inverse Vandermonde matrix. $\square$

The benefits of the point value representation lies in linear time (i.e $O(n)$) polynomial multiplication. For example, given polynomials in point-value representation vectors $\bar{A}$, $\bar{B} \in \mathbb{R}^n$, $\bar{A} * \bar{B}$ will require exactly n multiplicatons. There is still a problem however and that is the algorithmic complexity of conversion between the two forms. Even with the polynomial evaluation technique called $Horner's\ Method$, at least $n-1$ operations will be required for a degree-bound n polynomial and given that we must sample $n-1$ points, converting between coefficient to point-value form will still take $O(n^2)$. Mind as well implement the naive approach polynomial multplication but as we shall see, by carefully selecting our points of evaluation we can achieve the desired $O(nlogn)$ runtime.

3. Complex n^{th} Roots of Unity

A complex n^{th} root of unity is a complex number ω such that $\omega^n = 1$. For every n^{th} roots of unity, there are exactly n roots and they form a cyclic group with the generator called the **principle nth root of unity**. This is of great benefit because when we wish to find all the nth roots of unity, we can do so by taking the princple n^{th} root of unity to the k^{th} power.

n complex n^{th} roots of unity:

$$\omega_k = e^{2\pi ik/n} \qquad \text{for } k = 0, 1, ..., n-1$$

Principle n^{th} root of unity:

$$\omega_n = e^{2\pi i/n}$$

There are important properties of the complex n^{th} root of unity that allows for the desired time complexity of $O(nlogn)$. Although not very obvious, evaluating polynomials at these complex roots of unity will form the Discrete Fourier Transform and the following properties will allow for the application of the Fast Fourier Transform!

3.1 Halving Theorem

Theorem. *If $n > 0$ and even, then the squares of the n complex n^{th} roots of unity are the $\frac{n}{2}$ complex $\frac{n}{2}^{th}$ roots of unity.*

Proof.
Observe we can square two nth roots of unity to get the same results. For $k = 0, 1, ..., n-1$:

$$(\omega_n^k)^2 = (e^{2\pi ik/n}) = e^{2\pi ik/\frac{n}{2}} = \omega_{\frac{n}{2}}^k$$

and

$$(\omega_n^{k+\frac{k}{n}})^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = \omega_n^{2k} = (\omega_n^k)^2$$

Thus by squaring all the nth roots of unity, we get exactly each of the (n/2)th rots of unity exactly twice. □

3.2 Summation Theorem

Theorem. *For any integer $n \geq 1$ and nonzero integer k not divisible by n*

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

Proof.

$$\begin{aligned} \sum_{j=0}^{n-1} (\omega_n^k)^j &= \frac{1-(\omega_n^k)^n}{1-\omega_n^k} \qquad \text{by geometric series} \\ &= \frac{1-(\omega_n^n)^k}{1-\omega_n^k} = \frac{1-1^k}{1-\omega_n^k} = 0 \end{aligned}$$

□

4. Dicrete Fourier Tranform

By taking the polynomial at the nth complex roots of unity, we arrive at the Discrete Fourier Trasnform. Given $A(x) = \sum_{j=0}^{n-1} a_j x^j$, for $k = 0, 1, ..., n-1$

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj} = \sum_{j=0}^{n-1} a_j e^{2\pi i \frac{jk}{n}}$$

we get the vector $\bar(y) = (y_0, y_1, ..., y_{n-1})$

4.1 Fast Fourier Transform

In order to achieve $O(nlogn)$, we derive the recurrence relation of the coefficient to point-value representation where we use the properties of the nth roots of unity. Once the recurrence relation is found, we apply the divide & conquer approach so we half the problem size during each recursive call.

We first observe a polynomial can be of the form:

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$$

where $A^{[0]}$ are the even indexed terms and $A^{[1]}$ are the odd indexed terms. For example

$$A(x) = 1 + 3x^2 + 5x^2 + 7x^3$$
$$A^{[0]} = 1 + 5x$$
$$A^{[1]} = 3 + 7x$$

Notice how $A^{[0]}$ and $A^{[1]}$ have their x values squared. Thus by the halving theorem, we can take the $\frac{n}{2}$ roots of unity of $A^{[0]}$ and $A^{[1]}$ and they themselves become smaller subproblems!

4.2 Fast Fourier Transform^{-1}

So we have found how to convert from coefficient representation to point-value representation but what about the reverse? With vector multplication of results we get the point value form of the multiplied polynomial but we must now interpolate the coefficient form. We have already showed such interpolating exists and it too can be done is $O(nlogn)$ time.

Theorem. *Let V be the Vandermonde matrix. Then for $j, k = 0, 1, ..., n$, the j, k entry of V_n^{-1} is $\frac{\omega_n^{-kj}}{n}$*

Proof. We show we get the identity matrix I when multiplying VV^{-1}. Consider the (j, j') entry of VV^{-1}

$$\begin{aligned}[VV^{-1}]_{jj'} &= \sum_{k-0}^{n-1}(\frac{\omega_n^{-kj}}{n})(\omega_n^{kj'}) \\ &= \sum_{k-0}^{n-1}\omega_n^{k(j'-j)/n}\end{aligned}$$

thus by the summation theorem, the entry (j, j') if $j = j'$ and 0 otherwise which corresponds to the identity matrix. □

In summary, for polynomials of the form

$$A(x) = \sum_{j=0}^{n-1} a_j x^j,$$

forward fourier transform is

$$y_k = \sum_{j=1}^{n-1} a_j \omega_n^{kj}$$

and with the theorem above, the inverse fourier as

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj}$$

so one can conclude not a lot of change has to be done to the interpolating fast fourier algorithm than from the initial fast fourer transform.

5. Algorithm Implementation

5.1 Additional details

Some detail was left out in hashing out the idea for the fast fourier transform of polynomial multiplication. Firstly, given polynomials A(x), B(x) of degree bound n and C(x) such that A(x)B(x) = C(x), C must be degree bound 2n since deg(A)+deg(B) = deg(C). But the point value representations discussed thus far only converns n points. We can resolve this by taking 2n point-value conversion of polynomials A and B.
Lastly, the halving lemma requires n to be even so we 0-pad the polynomials so the number of terms is a power of 2.

5.2 Source Code (R)

```r
fib <- function(n) {
  if (n < 2)
    n
  else
    fib(n - 1) + fib(n - 2)
}
fib(10) # => 55
```