CMPSCI611: **The Fast Fourier Transform** Lecture 3

The Fast Fourier Transform (FFT) is a divide-and-conquer algorithm to multiply two polynomials in $O(n \log n)$ time rather than the $O(n^2)$ of the most obvious algorithm. In this lecture we will:

- Set up the context of **polynomial arithmetic**,
- See how fast **evaluation** and **interpolation** will allow us to multiply quickly,
- Review **complex numbers** and **roots of unity**.
- Present and analyze the **FFT algorithms** for evaluation and interpolation, and finally
- See what this has to do with ordinary **Fourier transforms**

Polynomial Arithmetic:

Suppose we are given two polynomials over the complex numbers (or over a subset like the reals), $A = \Sigma_i a_i x^i$ and $B = \Sigma_i b_i x^i$, each of degree at most $n - 1$. Each polynomial can be represented by a vector, or array, of its n coefficients.

Adding A and B is easy to do in $O(n)$ time: If $C = A+B$ then $c_i = a_i + b_i$ for each i. We can't hope to do better because we have to look at the entire input to be sure of the right answer and it takes $\Omega(n)$ time to do this.

Multiplying A and B looks like a harder problem. If $C = AB$, C may have degree as large as $2n - 2$, and each coefficient of C depends on many of, maybe all of, the coefficients of A and B:

$$c_k = \sum_i a_i b_{k-i}$$

where the range of the sum is such that both a_i and b_{k-i} exist.

Computing each c_k separately takes $O(n)$ each for most of them, or $O(n^2)$ in all. Can we do better?

Evaluation and Interpretation:

We can also represent a polynomial by giving its *value* on sufficiently many inputs. If we fix n distinct values $x_0, \ldots, x_{n-1}$, then the n values $A(x_0), \ldots, A(x_{n-1})$ determine the n coefficients $a_0, \ldots, a_{n-1}$. We've been told this in various math courses – why is it true?

The mapping from coefficients to functional values is a *linear tranformation,* and can thus be represented by a matrix. For example, if $n = 4$:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & x_0^3 \\ 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} A(x_0) \\ A(x_1) \\ A(x_2) \\ A(x_3) \end{pmatrix}$$

To tell whether this linear transformation is *invertible,* we need to look at the *determinant* of its matrix and see whether it is nonzero. It turns out that for general n the determinant of this *Vandemonde matrix* is

$$\prod_{i<j} (x_j - x_i)$$

which is nonzero iff the x_i's are all distinct.

If we have two polynomials A and B represented by their values at the same n points, and $C = AB$, then we can calculate the values of C at each of those points by the rule $C(x_i) = A(x_i)B(x_i)$. We can get all these values in $O(n)$ total time. (Actually, if A and B are arbitrary polynomials of degree at most $n-1$, we will want at least $2n-1$ points, so that we will have enough to determine all the coefficients of C).

This gives us an alternate way to compute the coefficients of C. If we have fast ways to *evaluate* a polynomial with given coefficients at given points, and to *interpolate* the coefficients from sufficiently many functional values, we can compute the mapping:

$$A, B \text{ (coefficients)} \rightarrow C \text{ (coefficients)}$$

by a three-step process:

$$\begin{tikzcd}
A, B \text{ (coefficients)} \arrow[r] & A, B \text{ (values)} \arrow[d] \\
C \text{ (coefficients)} & C \text{ (values)} \arrow[l]
\end{tikzcd}$$

(This is the intent of the garbled Figure 2.9 on page 18.)

Complex Numbers and Roots of Unity:

Our divide-and-conquer algorithms for evaluation and interpretation will take advantage of the *particular values* we choose for the points $x_0, \ldots, x_{n-1}$. For any positive number n, there are exactly n complex numbers ω that satisfy the equation $\omega^n = 1$. These are called the *n'th roots of unity*.

Recall the geometric meaning of multiplication of complex numbers. If we write two nonzero numbers as $\omega_1 = \rho_1 e^{i\theta_1}$ and $\omega_2 = \rho_2 e^{i\theta_2}$, then their product $\omega_1\omega_2$ is equal to $\rho_1\rho_2 e^{i(\theta_1+\theta_2)}$. Thinking of the numbers as vectors, we multiply their lengths and add their angles.

So the n roots of unity are the numbers $e^{j\cdot 2\pi i/n}$ for j from 0 through $n-1$. These are unit-length vectors evenly spaced around the origin. For example, with $n = 4$ the four roots of unity are 1, i, -1, and -1. Figure 2-10 in the notes shows the $n = 8$ case.

The **Halving Lemma** says that if we square each of the n'th roots of unity, where n is even, we get the $n/2$'th roots of unity, twice each.

An Evaluation Example:

Suppose we want to evaluate $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ at the four roots of unity 1, i, -1, and $-i$. The four values we need are:

$$\begin{gathered} a_0 + a_1 + a_2 + a_3 \\ a_0 + ia_1 - a_2 - ia_3 \\ a_0 - a_1 + a_2 - a_3 \\ a_0 - ia_1 - a_2 + ia_3 \end{gathered}$$

Just as with the two divide-and-conquer multiplication algorithms, we can identify common pieces of these sums:

$$\begin{gathered} (a_0 + a_2) + (a_1 + a_3) \\ (a_0 - a_2) + i(a_1 - a_3) \\ (a_0 + a_2) - (a_1 + a_3) \\ (a_0 - a_2) - i(a_1 - a_3) \end{gathered}$$

Normally it would take us twelve additions to get these four numbers, but if we calculate the four in parentheses first, we can do it with eight. The FFT algorithm will use the same idea.

FFT Evaluation:

Let $A(x)$ be $a_0 + \ldots + a_{n-1}x^{n-1}$ and write it as follows:

$$\begin{aligned} A(x) &= (a_0 + a_2x^2 + \ldots + a_{n-2}x^{n-2}) \\ &\quad +x(a_1 + a^3x^2 + \ldots + a_{n-1}x^{n-2}) \\ &= A_{even}(x^2) + xA_{odd}(x^2) \end{aligned}$$

Here A_{even} is the polynomial whose coefficients are the even-numbered coefficients of A, and similarly for A_{odd}.

$$\begin{aligned} A_{even}(y) &= a_0 + a_2y + a_4y^2 + \ldots + a_{n-2}y^{\frac{n}{2}-1} \\ A_{odd}(y) &= a_1 + a_3y + a_5y^2 + \ldots + a_{n-1}y^{\frac{n}{2}-1} \end{aligned}$$

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

With a recursive call to our FFT evaluation algorithm, we will be able to evaluate a polynomial with $n/2$ coefficients at $n/2$ points. The polynomials we evaluate will be $A_{even}(x)$ and $A_{odd}(x)$, and we will evaluate them at the $n/2$'th roots of unity. We will need some arrays to store the answers:

- y is an array such that $y_k = A(\omega_n^k)$
- y^{even} is an array such that $y_k^{even} = A_{even}(\omega_n^{2k})$
- y^{odd} is an array such that $y_k^{odd} = A_{odd}(\omega_n^{2k})$

By the Halving Lemma, the $n/2$ points at which the recursive call evaluates the functions A_{even} and A_{odd} are exactly the points ω^{2k} for each k.

The FFT Evaluation Algorithm:

```
yEven = fft (a[0],...,a[n-2]);
yOdd  = fft (a[1],...,a[n-1]);
  for (int k=0; k < n/2; k++) {
    y[k] = yEven[k] +
              (OMEGAˆk)*yOdd[k];
    y[k + n/2] = yEven[k] -
              (OMEGAˆk)*yOdd[k];}
return y;
```

Recall that $A(x) = A_{even}(x^2) + xA_{odd}(x^2)$. The value returned by this algorithm for $A(\omega_n^k)$ is:

- $A_{even}(\omega_n^{2k}) + \omega_n^k A_{odd}(\omega_n^{2k})$, if $k < n/2$
- $A_{even}(\omega_n^{2k}) - \omega_n^k A_{odd}(\omega_n^{2k})$, if $k \geq n/2$

which is correct because $\omega_n^{k+n/2} = -\omega_n^k$, as $\omega_n^{n/2} = -1$.

The time analysis is just as for Mergesort: $T(n) = 2T(n/2)+ \Theta(n)$, so $T(n) = \Theta(n \log n)$.

FFT Interpolation:

We argued that the matrix $\mathbf{V}_n$, whose (i, j) entry is ω_n^{ij}, is invertible.

In fact its inverse is almost the same as itself. Its (i, j) entry is $\frac{1}{n}\omega_n^{-ij}$.

To check this, let's compute the product of these two matrices. The (i, j) entry of the product is

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega_n^{ik}\omega_n^{-kj}.$$

If $i = j$, each entry of this sum is $\omega_n^{ij-ij} = 1$, and the sum is n. But if $i \neq j$, each entry is $\omega_n^{k(i-j)}$, and these entries are an equal number of copies of each of the powers of ω_n^{i-j}, which will add to zero because they are evenly spaced around the unit circle.

The interpolation algorithm is thus very similar to the evaluation algorithm. We need only switch the roles of the arrays y and a, replace each ω_n with ω_n^{-1}, and divide the answer by n before returning it. Of course the timing analysis is exactly the same.

Ordinary Fourier Transforms:

A function f from the reals to the reals is *periodic* if for some nonzero number T, it satisfies the rule $f(x+T) = f(x)$ for all x. The most familiar periodic functions are the sine and cosine from trigonometry.

The theory of *Fourier Analysis* tells us that any continuous periodic function on the reals may be expressed as an infinite linear combination of sine and cosine functions whose period is an integer fraction of T, T/k.

If we think of $f(x)$ as being $g(e^{2\pi ix/T})$, then we might try to approximate g by giving it the correct values on the n roots of unity. As we have just seen, given any set of values of g on those roots of unity, there is a unique polynomial with n coefficients that achieves those values. The FFT algorithm can get us the coefficients from the values, or vice versa.

If we could write g as $a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$, we break g down as the sum of functions of the form a_kx^k, and thus break down f as the sum of functions of the form $a_k(e^{2\pi ix/T})^k$, or $a_k(\cos(2\pi kx/T) + i\sin(2\pi kx/T)$. These are the sine and cosine functions of the Fourier transform. As n increases, the approximation is correct on more values and closer in general to the original function.