CMPSCI 611 Advanced Algorithms **Fall 2001**

Lecture 3: September 12

Lecturer: Micah Adler *Scribe: Marc Pickett I and Özgür Şimşek*

These scribe notes describe lecture 3 of Computer Science 611. They borrow heavily from [SN2] and [SN3].

In this lecture, we learn about the fast Fourier transform (FFT) which is another example of a divide and conquer algorithm. We learn how the FFT can be used to do multiplication of polynomials of degree $n-1$ in $\Theta(n \log n)$ time, where a more naive method would take $\Theta\left(n^2\right)$ time.

3.1 Polynomial Arithmetic

Suppose $A(x)$ and $B(x)$ are polynomials of degree $n-1$.

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

Addition of two polynomials can be done in time $\Theta(n)$ by adding similar terms:

$$A(x) + B(x) = (a_0 + b_0) + (a_1 + b_1) x + (a_2 + b_2) x^2 \ldots (a_{n-1} + b_{n-1}) x^{n-1}$$

Multiplication is not quite as simple as addition. The obvious algorithm is to multiply each pair of terms. This takes time $\Theta\left(n^2\right)$.

$$A(x) \cdot B(x) = (a_0 b_0 + a_0 b_1 x + \ldots) + (a_1 x b_0 + a_1 x b_1 x + \ldots) + \ \ldots$$

3.2 Point-Value Representation

The above polynomials are in coefficient representation. We can also represent polynomials using point-value representation. In this representation, a degree $n-1$ polynomial is evaluated at (any) n distinct points. For example, we would represent $A(x)$ as follows:

$$\begin{gathered} A(x) = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} \\ \text{where } y_i = A(x_i), \text{ and if } i \neq j \text{ then } x_i \neq x_j \end{gathered}$$

Claim 3.1 *A point-value representation with n distinct points defines a unique degree $n-1$ polynomial [CLR, pp 779–780].*

If we have n points, each point-value pair gives a linear equation in the coefficient representation. Thus, we get n linear equation with n unknowns. These equations are in a special form that guarantees a unique solution.

Addition with point-value representation can be done by using the same x values to specify $A(x)$ and $B(x)$, and adding the corresponding y values. This takes time $\Theta(n)$.

$$\begin{aligned} A(x) &= \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} \\ B(x) &= \{(x_0, y_0'), (x_1, y_1'), \ldots, (x_{n-1}, y_{n-1}')\} \\ A(x) + B(x) &= \{(x_0, y_0 + y_0'), (x_1, y_1 + y_1'), \ldots, (x_{n-1}, y_{n-1} + y_{n-1}')\} \end{aligned}$$

Multiplication with point value representation is the same as addition except that we multiply corresponding terms instead of adding them. This also takes time $\Theta(n)$.

$$\begin{aligned} A(x) &= \{(x_0, y_0), (x_1, y_1), \ldots\} \\ B(x) &= \{(x_0, y_0'), (x_1, y_1'), \ldots\} \\ A(x) \cdot B(x) &= \{(x_0, y_0 \cdot y_0'), (x_1, y_1 \cdot y_1'), \ldots\} \end{aligned}$$

Note that $A(x) \cdot B(x)$ is a polynomial with degree $2n-2$, which needs $2n-1$ points in order to be specified. Therefore, we should start with point-value representations of $A(x)$ and $B(x)$ that specify $2n-1$ points.

3.3 A Framework for Polynomial Multiplication

We have seen that multiplication using point-value representation takes linear time. We can take advantage of this fact when multiplying two polynomials in coefficient representation by using the following procedure:

Evaluation Convert $A(x)$ and $B(x)$ to point-value representation by evaluating each polynomial at $2n-1$ points.

Multiplication Multiply the polynomials in point-value representation.

Interpolation Convert the result back to coefficient representation.

If evaluation and interpolation can be done in a lesser time than quadratic, then we've found a faster way to do multiplication than the obvious algorithm. This idea is shown in Figure 3.1. We will use the FFT to do both evaluation and interpolation in $\Theta(n \log n)$ time. Thus, we will do polynomial multiplication in time $\Theta(n \log n)$.

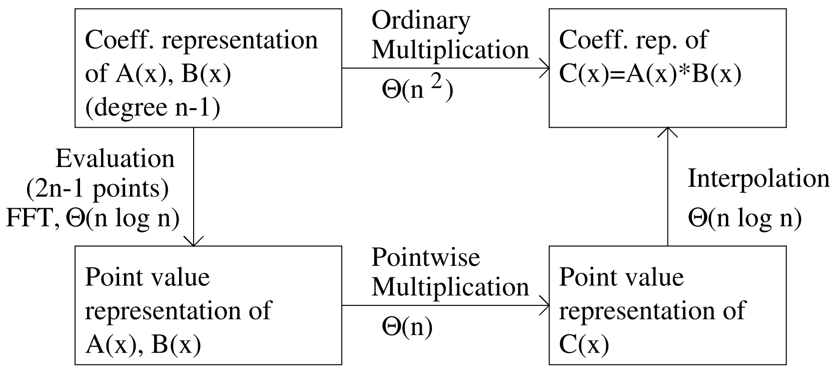

Figure 3.1: A graphical outline of polynomial multiplication using FFT.

3.4 Fast Polynomial Evaluation

We can choose the points at which we want to evaluate the polynomials $A(x)$ and $B(x)$ in the evaluation step. The basic idea of the FFT is to exploit this by choosing **complex roots of unity** because these have properties that can be used to make evaluation inexpensive.

3.4.1 Complex Roots of Unity

Definition 3.2 Complex nth roots of unity *are solutions to $x^n = 1$. These are n equally spaced points around the unit circle.*

For example, the first root of unity is 1. It is the solution to $x = 1$.
The second roots of unity are 1 and -1. They are the solutions to $x^2 = 1$.
The fourth roots of unity are 1, -1, i, and $-i$, where $i = \sqrt{-1}$. They are the solutions to $x^4 = 1$.

Definition 3.3 The principle nth root of unity (ω_n) *is the number that makes an angle of $\frac{360}{n}$ degrees with the real axis on the complex plane (going counter-clockwise).*

In general, ω_n^k makes an angle of $\frac{k*360}{n}$ degrees with the real axis. Note that the n complex roots of unity can be written as $\omega_n^0, \omega_n^1, \omega_n^2, \cdots, \omega_n^{n-1}$.

Figure 3.2 shows the 8th roots of unity on the complex plane, also identifying the principle 8th root.

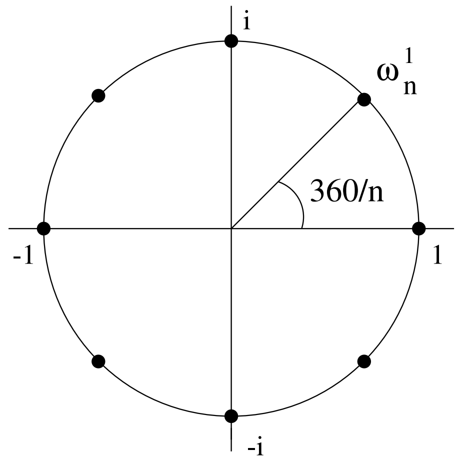

Figure 3.2: Complex 8th roots of unity (n=8).

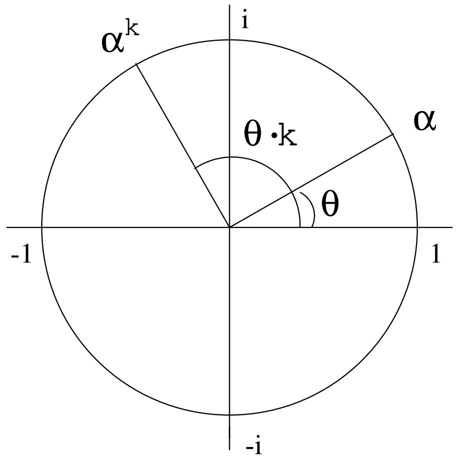

Figure 3.3: A property of the complex plane. If a complex number α makes an angle of θ degrees with the real axis, then the complex number α^k makes $\theta \cdot k$ degrees with the real axis.

The complex roots of unity have the following useful properties:

$$\omega_n^{k+n} = \omega_n^k \tag{3.1}$$

$$(\omega_n)^{n/2} = -1 \tag{3.2}$$

Equation (3.2) follows directly from the following property of the complex plane:

If a complex number α makes an angle of θ degrees with the real axis on the complex plane, then the complex number α^k makes $\theta \cdot k$ degrees with the real axis. This property is illustrated in figure 3.3.

Lemma 3.4 "The Halving Lemma" *If n is even, squares of n complex roots of unity are $\frac{n}{2}$ complex $\frac{n}{2}$nd roots of unity repeated twice. That is,*

$$\left(\omega_n^0\right)^2, \left(\omega_n^1\right)^2, \left(\omega_n^2\right)^2, \ldots, \left(\omega_n^{n-1}\right)^2 \quad = \quad \omega_{\frac{n}{2}}^0, \omega_{\frac{n}{2}}^1, \omega_{\frac{n}{2}}^2, \ldots, \omega_{\frac{n}{2}}^{\frac{n}{2}-1},\ \ \omega_{\frac{n}{2}}^0, \omega_{\frac{n}{2}}^1, \omega_{\frac{n}{2}}^2, \ldots, \omega_{\frac{n}{2}}^{\frac{n}{2}-1}$$

3.4.2 The Fast Fourier Transform for Evaluation

Now that we have some properties of complex roots of unity, we can use the n nth complex roots of unity as our points to evaluate a $n-1$ degree polynomial at n points.

Example:

Let us consider the case when n=4, and $A(x)$ is a degree 3 polynomial.

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

Let us evaluate $A(x)$ at the 4th roots of unity (i.e., $\omega_4^0, \omega_4^1, \omega_4^2, \omega_4^3$) which are

$$\begin{aligned} \omega_4^0 &= 1 \\ \omega_4^1 &= i \\ \omega_4^2 &= -1 \\ \omega_4^3 &= -i \end{aligned}$$

Thus, A can be written in point value representation as follows:

$$\begin{aligned} A(x) = \{ \ & \left(\omega_4^0,\ a_0 + a_1 + a_2 + a_3\right), \\ & \left(\omega_4^1,\ a_0 + a_1\omega_4 + a_2\omega_4^2 + a_3\omega_4^3\right), \\ & \left(\omega_4^2,\ a_0 + a_1\omega_4^2 + a_2\omega_4^4 + a_3\omega_4^6\right), \\ & \left(\omega_4^3,\ a_0 + a_1\omega_4^3 + a_2\omega_4^6 + a_3\omega_4^9\right) \ \} \end{aligned}$$

This evaluation requires 12 additions and 9 multiplications to solve. But the number of operations can be reduced by substituting the complex values from above and reordering to get a point-value representation as shown below.

$$\begin{aligned} A(x) = \{ \ & \left(1,\ (a_0 + a_2) + (a_1 + a_3)\right), \\ & \left(\omega_4,\ \left(a_0 + a_2\omega_4^2\right) + \omega_4\left(a_1 + a_3\omega_4^2\right)\right), \\ & \left(\omega_4^2,\ (a_0 + a_2) - (a_1 + a_3)\right), \\ & \left(\omega_4^3,\ \left(a_0 + a_2\omega_4^2\right) - \omega_4\left(a_1 + a_3\omega_4^2\right)\right) \ \} \end{aligned}$$

We see that, in the above representation, certain terms are repeated. By taking advantage of these repetitions, the work can be brought down to 8 additions and 2 multiplications. For large values of n, the savings can be quite substantial.

Let us see how this concept can be extended to evaluate a degree $n-1$ polynomial at the nth roots of unity. Let $A(x)$ be a degree $n-1$ polynomial. Grouping the even-indexed coefficients and the odd-indexed coefficients together we get the following:

$$\begin{aligned} A(x) &= a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} \\ A(x) &= a_0 + a_2x^2 + a_4x^4 + \ldots + a_{n-2}x^{n-2} \quad + x\left(a_1 + a_3x^2 + a_5x^4 + \ldots + a_{n-1}x^{n-2}\right) \\ A(x) &= A_{even}\left(x^2\right) + xA_{odd}\left(x^2\right) \end{aligned}$$

where A_{even} is a polynomial of degree $\frac{n}{2}-1$, with coefficients $(a_0, a_2, a_4, \ldots, a_{n-2})$, and A_{odd} is a polynomial of degree $\frac{n}{2}-1$, with coefficients $(a_1, a_3, a_5, \ldots, a_{n-1})$.

From the Halving Lemma, evaluation of A_{even} at $(1)^2, (\omega_n)^2, \left(\omega_n^2\right)^2, \ldots, \left(\omega_n^{n-1}\right)^2$ is equivalent to the evaluation of A_{even} at $1, \omega_{\frac{n}{2}}, \omega_{\frac{n}{2}}^2, \ldots, \omega_{\frac{n}{2}}^{\frac{n}{2}-1}, 1, \omega_{\frac{n}{2}}, \omega_{\frac{n}{2}}^2, \ldots, \omega_{\frac{n}{2}}^{\frac{n}{2}-1}$.

The same holds for A_{odd}. Thus, the problem of evaluating a degree $n-1$ polynomial at the nth roots of unity reduces to the evaluation of two degree $\frac{n}{2}-1$ polynomials at the $\frac{n}{2}$nd roots of unity.

Evaluating A_{even} and A_{odd} recursively at $\frac{n}{2}$nd roots of unity, we have:

$$\begin{aligned} A\left(\omega_n^k\right) &= A_{even}\left(\omega_n^{2k}\right) + \omega_n^k \cdot A_{odd}\left(\omega_n^{2k}\right), \quad \text{for } 0 \le k < \frac{n}{2} \\ A\left(\omega_n^{k+\frac{n}{2}}\right) &= A_{even}\left(\omega_n^{2k+n}\right) + \omega_n^{k+\frac{n}{2}} \cdot A_{odd}\left(\omega_n^{2k+n}\right), \quad \text{for } \frac{n}{2} \le k < n \\ &= A_{even}\left(\omega_n^{2k}\right) - \omega_n^k \cdot A_{odd}\left(\omega_n^{2k}\right) \end{aligned}$$

The last equation follows because $\omega_n^{k+\frac{n}{2}} = \omega_n^{\frac{n}{2}}\omega_n^k$ and $\omega_n^{\frac{n}{2}} = -1$.

This gives us the following algorithm:

Algorithm: FFT $(a_0, a_1, a_2, \ldots, a_{n-1})$

if $n = 1$, return a_0

else //divide and conquer

$y^{even} = FFT(a_0, a_2, a_4, \ldots, a_{n-2})$

$y^{odd} = FFT(a_1, a_3, a_5, \ldots, a_{n-1})$

for $k = 0$ to $\frac{n}{2} - 1$ //combine

$y_k = y_k^{even} + \omega_n^k \cdot y_k^{odd}$

$y_{k+\frac{n}{2}} = y_k^{even} - \omega_n^k \cdot y_k^{odd}$

return y

Running time: FFT has the following recurrence relation since we are solving 2 problems each of size $\frac{n}{2}$ and the cost of combining is linear in n:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n) \\ T(1) &= \Theta(1) \end{aligned}$$

Solving the recurrence relation yields

$$T(n) = \Theta(n \log n)$$

3.5 Summary

We've almost shown how to multiply two $n-1$ degree polynomials in $\Theta(n \log n)$ time. We've shown the general scheme, and how the first two steps can be taken (introducing the fast Fourier transform (FFT) in the process). In the next lecture, we'll see how the FFT can be used to do the interpolation step in $\Theta(n \log n)$ time.

References

SN2 G. Holness and T. Brunette Scribe notes for lecture 2 of Computer Science 611, University of Massachusetts at Amherst, 2000.

SN3 S. Vasudevan and P. Dickson Scribe notes for lecture 3 of Computer Science 611, University of Massachusetts at Amherst, 2000.

CLR T. Cormen and C. Leiserson and R. Rivest, Introduction to Algorithms, MIT Press, 1990