Lecture 5

Fast Fourier Transform

Supplemental reading in CLRS: Chapter 30

The algorithm in this lecture, known since the time of Gauss but popularized mainly by Cooley and Tukey in the 1960s, is an example of the **divide-and-conquer** paradigm. Actually, the main uses of the fast Fourier transform are much more ingenious than an ordinary divide-and-conquer strategy—there is genuinely novel mathematics happening in the background. Ultimately, the FFT will allow us to do n computations, each of which would take $\Omega(n)$ time individually, in a total of $\Theta(n \lg n)$ time.

5.1 Multiplication

To motivate the fast Fourier transform, let's start with a very basic question:

> How can we efficiently multiply two large numbers or polynomials?

As you probably learned in high school, one can use essentially the same method for both:

$$\begin{array}{r} 385 \\ \times 426 \\ \hline 2310 \\ 770 \\ +1540 \\ \hline 164010 \end{array} \qquad \begin{array}{r} (3x^2 + 8x + 5) \\ \times\, (4x^2 + 2x + 6) \\ \hline 18x^2 + 48x + 30 \\ 6x^3 + 16x^2 + 10x \\ 12x^4 + 32x^3 + 20x^2 \\ \hline 12x^4 + 38x^3 + 54x^2 + 58x + 30 \end{array}$$

Of these two, polynomials are actually the easier ones to work with. One reason for this is that multiplication of integers requires carrying, while multiplication of polynomials does not. To make a full analysis of this extra cost, we would have to study the details of how large integers can be stored and manipulated in memory, which is somewhat complicated.[1] So instead, let's just consider multiplication of polynomials.

Suppose we are trying to multiply two polynomials p, q of degree at most n with complex coefficients. In the high-school multiplication algorithm (see Figure 5.1), each row of the diagram is

[1] More ideas are required to implement efficient multiplication of n-bit integers. In a 1971 paper, Schönhage and Strassen exhibited integer multiplication in $O(n \lg n \cdot \lg\lg n)$ time; in 2007, Fürer exhibited $O\left(n \lg n \cdot 2^{O(\lg^* n)}\right)$ time. The iterated logarithm $\lg^* n$ is the smallest k such that $\lg\lg\cdots\lg n$ (k times) ≤ 1. It is not known whether integer multiplication can be achieved in $O(n \lg n)$ time, whereas by the end of this lecture we will achieve multiplication of degree-n polynomials in $O(n \lg n)$ time.

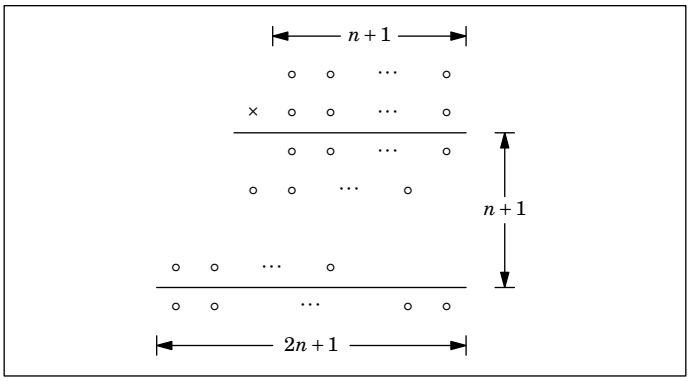

Figure 5.1. The high-school polynomial multiplication algorithm.

obtained by multiplying p by a monomial. Thus any single row would take $\Omega(n)$ time to compute individually. There are $n+1$ rows, so the computation takes $\Omega(n^2)$ time.

The key to improving this time is to consider alternative ways of representing p and q. Fix some $N > n$. We'll choose N later, but for now all that matters is that $N = O(n)$. A polynomial of degree at most $N-1$ is uniquely determined by its values at N points. So, instead of storing the coefficients of p and q, we could represent p and q as the lists

$$p_z = \langle p(z_0), \ldots, p(z_{N-1}) \rangle \quad \text{and} \quad q_z \langle p(z_0), \ldots, p(z_{N-1}) \rangle$$

for any distinct complex numbers $z_0, \ldots, z_{N-1}$. In this representation, computing pq is very cheap—the list

$$(pq)_z = \langle p(z_0)q(z_0), \ldots, p(z_{N-1})q(z_{N-1}) \rangle$$

can be computed in $O(n)$ time. If $N > \deg pq$, then pq is the unique polynomial whose FFT is $(pq)_z$.

Algorithm: MULTIPLY(p, q)

1. Fix some $N = O(\deg p + \deg q)$ such that $N > \deg p + \deg q$.
2. Compute the sample vectors $p_z = \langle p(z_0), \ldots, p(z_{N-1}) \rangle$ and $q_z = \langle q(z_0), \ldots, q(z_{N-1}) \rangle$.
3. Compute the unique polynomial r such that $r_z = p_z \cdot q_z$, where $\cdot$ denotes entrywise multiplication.*
4. Return r.

* For example, $\langle 1,2,3 \rangle \cdot \langle 4,5,6 \rangle = \langle 4,10,18 \rangle$.

Thus, the cost of finding the coefficients of pq is equal to the cost of converting back and forth between the coefficients representation and the sample-values representation (see Figure 5.2). We are free to choose values of $z_0, \ldots, z_{N-1}$ that make this conversion as efficient as possible.

5.1.1 Efficiently Computing the FFT

In the fast Fourier transform, we choose $z_0, \ldots, z_{N-1}$ to be the Nth roots of unity, $1, \omega_N, \omega_N^2, \ldots, \omega_N^{N-1}$, where $\omega_N = e^{2\pi i/N}$. We make this choice because roots of unity enjoy the useful property that

$$\omega_N^2 = \omega_{N/2}.$$

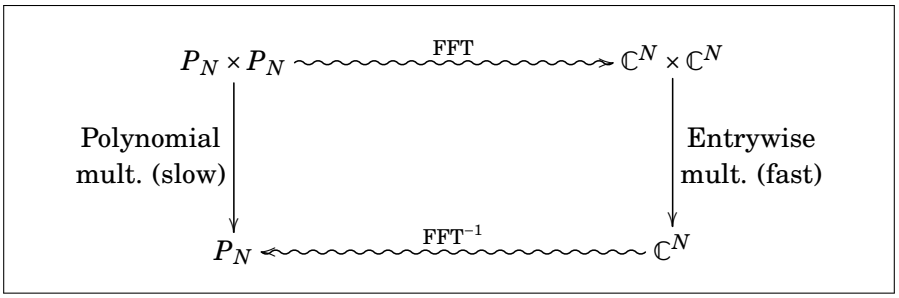

Figure 5.2. Commutative diagram showing the cost of multiplication on either side of a fast Fourier transform. As we will see, the fastest way to get from the top-left to the bottom-left is through the FFT.

(We can choose N to be a power of 2.) This allows us to *divide* the task of computing an FFT of size N into computing two FFTs of size $N/2$, and then combining them in a certain way.[2]

Algorithm: $\text{FFT}\left(N,\ p(x) = a_{N-1}x^{N-1} + a_{N-2}x^{N-2} + \cdots + a_1 x + a_0\right)$

1 ▷ As part of the specification, we assume N is a power of 2
2 **if** $N = 1$ **then**
3 **return** a_0
4 Let

$$p^{\text{even}}(y) \leftarrow a_0 + a_2 y + \cdots + a_{N-2} y^{N/2-1},$$
$$p^{\text{odd}}(y) \leftarrow a_1 + a_3 y + \cdots + a_{N-1} y^{N/2-1}$$

be the even and odd parts of p
5 ▷ Thus

$$p(x) = p^{\text{even}}\left(x^2\right) + x p^{\text{odd}}\left(x^2\right) \tag{5.1}$$

6 $p^{\text{even}}_{\omega_{N/2}} \leftarrow \text{FFT}(N/2,\ p^{\text{even}})$
7 $p^{\text{odd}}_{\omega_{N/2}} \leftarrow \text{FFT}\left(N/2,\ p^{\text{odd}}\right)$
8 ▷ That is,

$$p^{\text{even}}_{\omega_{N/2}} = \left\langle p^{\text{even}}(1),\ p^{\text{even}}(\omega_{N/2}),\ \ldots,\ p^{\text{even}}\left(\omega_{N/2}^{N/2-1}\right)\right\rangle$$
$$p^{\text{odd}}_{\omega_{N/2}} = \left\langle p^{\text{odd}}(1),\ p^{\text{odd}}(\omega_{N/2}),\ \ldots,\ p^{\text{odd}}\left(\omega_{N/2}^{N/2-1}\right)\right\rangle.$$

9 ▷ Because $\omega_{N/2} = \omega_N^2$, we can calculate the vector $p_{\omega_N} = \left\langle p(1),\ p(\omega_N),\ \ldots,\ p\left(\omega_N^{N-1}\right)\right\rangle$ very quickly using (5.1):
10 $\boldsymbol{\omega} \leftarrow \left\langle 1,\ \omega_N,\ \ldots,\ \omega_N^{N-1}\right\rangle$ ▷ the left side is a bold omega
11 **return** $p^{\text{even}}_{\omega_{N/2}} + \boldsymbol{\omega} \cdot p^{\text{odd}}_{\omega_{N/2}}$, where $\cdot$ denotes entrywise multiplication

[2] I suppose the "conquer" stage is when we recursively compute the smaller FFTs (but of course, each of these smaller FFTs begins with its own "divide" stage, and so on). After the "conquer" stage, the answers to the smaller problems are combined into a solution to the original problem.

Above, we compute p_{ω_N} by computing $p^{\text{odd}}_{\omega_{N/2}}$ and $p^{\text{even}}_{\omega_{N/2}}$ and combining them in $\Theta(N)$ time. Thus, the running time of an FFT of size N satisfies the recurrence

$$T(N) = 2T(N/2) + \Theta(N).$$

This recurrence is solved in CLRS as part of the Master Theorem in §4.5. The solution is

$$T(N) = \Theta(N \lg N).$$

5.1.2 Computing the Inverse FFT

Somewhat surprisingly, the inverse FFT can be computed in almost exactly the same way as the FFT. In this section we will see the relation between the two transforms. If you don't have a background in linear algebra, you can take the math on faith.

Let P_N be the vector space of polynomials of degree at most $N-1$ with complex coefficients. Then the FFT is a bijective linear map $P_N \to \mathbb{C}^N$. If we use the ordered basis $1, x, \ldots, x^{N-1}$ for P_N and the standard basis for $\mathbb{C}^N$, then the matrix of the FFT is

$$A = \left(\omega_N^{ij}\right)_{0 \le i,j \le N-1} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N & \omega_N^2 & \cdots & \omega_N^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \cdots & \omega_N^{(N-1)^2} \end{pmatrix}.$$

The jth column of A^{-1} contains the coefficients of the polynomial which, when evaluated on ω_N^j, gives 1 and, when evaluated on ω_N^i ($i \neq j$), gives zero. This polynomial is

$$\frac{\prod_{i \neq j}\left(x - \omega_N^i\right)}{\prod_{i \neq j}\left(\omega_N^j - \omega_N^i\right)}.$$

We will not show the details here, but after about a page of calculation you can find that

$$A^{-1} = \tfrac{1}{N}\overline{A} = \frac{1}{N}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N^{-1} & \omega_N^{-2} & \cdots & \omega_N^{-(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{-(N-1)} & \omega_N^{-2(N-1)} & \cdots & \omega_N^{-(N-1)^2} \end{pmatrix},$$

where the bar indicates entrywise complex conjugation. You can certainly check that the i,j entry of $\frac{1}{N}A\overline{A}$ is

$$\sum_{\ell=0}^{N-1} \omega_N^{i\ell} \cdot \tfrac{1}{N}\omega_N^{-j\ell} \quad = \quad \tfrac{1}{N}\sum_{\ell=0}^{N-1} \omega_N^{\ell(i-j)} \quad = \quad \begin{cases} 1, & i = j \\ 0, & i \neq j, \end{cases}$$

and similarly for $\frac{1}{N}\overline{A}A$. Thus

$$\boxed{\text{FFT}^{-1}(\mathbf{v}) = \tfrac{1}{N}\overline{\text{FFT}\left(\overline{\mathbf{v}}\right)}.} \tag{5.2}$$

Again, this is surprising. A priori, we would consider P_N and $\mathbb{C}^N$ to be "different worlds"—like two different implementations of the same abstract data structure, the coefficients representation and

the sample-values representation of polynomials serve essentially the same role but have different procedures for performing operations. Yet, (5.2) tells us that inverting an FFT involves creating a new polynomial whose coefficients are the *sample values* of our original polynomial. This of course has everything to do with the fact that roots of unity are special; it would not have worked if we had not chosen $1, \omega_N, \ldots, \omega_N^{N-1}$ as our sample points.

Algorithm: INVERSE-FFT$\big(N,\ \mathbf{v} = \langle v_0, \ldots, v_{N-1}\rangle\big)$

1 $\overline{\mathbf{v}} \leftarrow \langle \overline{v_0}, \ldots, \overline{v_{N-1}}\rangle$
2 Let $p_{\overline{\mathbf{v}}}(x)$ be the polynomial $\overline{v_{N-1}}x^{N-1} + \cdots + \overline{v_1}x + \overline{v_0}$
3 **return** the value of

$$\frac{1}{N}\overline{\mathrm{FFT}\big(N,\ p_{\overline{\mathbf{v}}}\big)}$$

With all the details in place, the MULTIPLY algorithm looks like this:

Algorithm: FFT-MULTIPLY(p, q)

1. Let N be the smallest power of 2 such that $N - 1 \geq \deg p + \deg q$.
2. Compute $\mathbf{v} = \mathrm{FFT}(N, p)$ and $\mathbf{w} = \mathrm{FFT}(N, q)$, and let $\mathbf{u} = \mathbf{v} \cdot \mathbf{w}$, where $\cdot$ denotes entry-wise multiplication.
3. Compute and return INVERSE-FFT$(N,\ \mathbf{u})$.

It runs in $\Theta(N \lg N) = \Theta\big((\deg p + \deg q)\lg(\deg p + \deg q)\big)$ time.

5.2 Convolution

Let $p(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ and $q(x) = b_{n-1}x^{n-1} + \cdots + b_1 x + b_0$. The coefficient of x^k in pq is

$$\sum_{\ell \in \mathbb{Z}_{2n-1}} a_\ell b_{k-\ell},$$

where the subscripts are interpreted modulo $2n-1$. This sort of operation is known as **convolution**, and is written as $*$. Convolution can be applied quite generally: if f and g are any functions $\mathbb{Z}_{2n-1} \to \mathbb{C}$, then one common definition of the convolution operation is

$$(f * g)(k) = \frac{1}{2n-1}\sum_{\ell \in \mathbb{Z}_{2n-1}} f(\ell)g(k-\ell).$$

For each i, computing $(f * g)(i)$ requires linear time, but the fast Fourier transform allows us to compute $(f * g)(i)$ for all $i = 0, \ldots, 2n-2$ in a total of $\Theta(n \lg n)$ time.

Convolution appears frequently, which is part of the reason that the FFT is useful. The above notion of convolution can easily be generalized to allow f and g to be functions from any group G to any ring in which $|G|$ is a unit. There is also a continuous version of convolution which involves an integral. This continuous version is useful in signal compression. The idea is to let $f : \mathbb{R} \to \mathbb{R}$ be an arbitrary signal and let g be a smooth bump function which spikes up near the origin and is zero elsewhere. (See Figure 5.3.) Then the convolution of f and g is a new function $f * g$ defined by

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau.$$

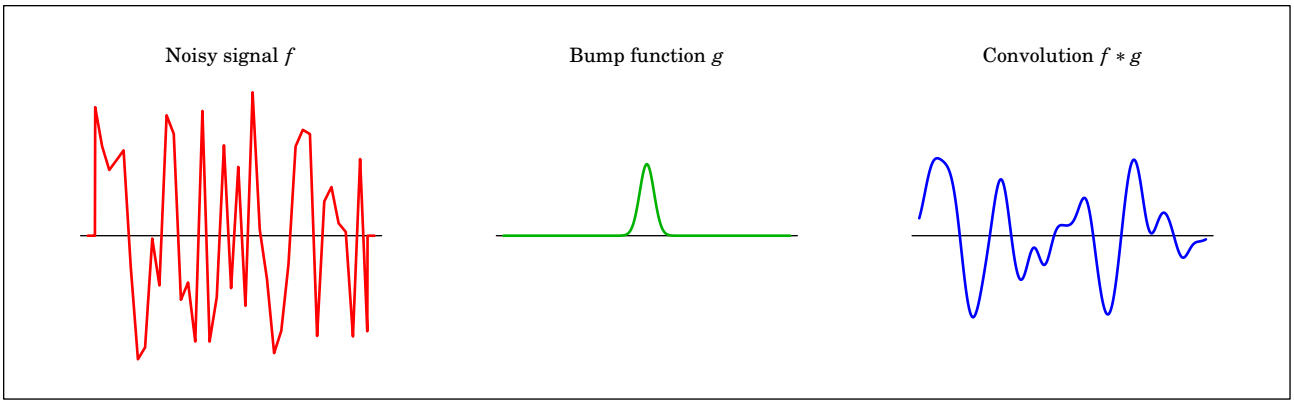

Figure 5.3. A complicated, noisy signal f, a bump function g, and the convolution $f * g$.

The value $(f * g)(t)$ is essentially an average of the values of f at points close to t. Thus, the funtion $f * g$ inherits smoothness from g while still carrying most of the information from f. As a bonus, if f comes with some unwanted random noise, then g will have much less noise. For example, f may be the raw output of a recording device, and computing $f * g$ may be the first step towards encoding f into a compressed audio format such as `MP3`. The point is that smooth functions are much easier to describe concisely, so creating a smooth version of f is useful in (lossy) data compression.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012