Fast Fourier Transform

Lecturer: Michel Goemans

In these notes we define the Discrete Fourier Transform, and give a method for computing it fast: the *Fast Fourier Transform*. We then use this technology to get an algorithms for multiplying big integers fast. Before going into the core of the material we review some motivation coming from the classical theory of Fourier series.

1 Motivation: Fourier Series

In this section we discuss the theory of Fourier Series for functions of a real variable. In the next sections we will study an analogue which is the "discrete" Fourier Transform.

Early in the Nineteenth century, Fourier studied sound and oscillatory motion and conceived of the idea of representing periodic functions by their coefficients in an expansion as a sum of sines and cosines rather than their values. He noticed, for example, that you can represent the shape of a vibrating string of length L, fixed at its ends, as

$$y(x) = \sum_{k=1}^{\infty} a_k \sin(\pi k x/L).$$

(Observe that indeed $y(0) = y(L) = 0$.) The coefficients, a_k, contain important and useful information about the quality of the sound that the string produces, that is not easily accessible from the ordinary $y = f(x)$ description of the shape of the string.

This kind of representation is called a *Fourier Series*, and there is a tremendous amount of mathematical lore about properties of such series and for what classes of functions they can be shown to exist. One particularly useful fact about them is the *orthogonality* property of sines:

$$\int_{x=0}^{L} \sin(\pi k x/L) \sin(\pi j x/L) dx = \delta_{j,k} \frac{L}{2},$$

for nonnegative integers j and k. Here $\delta_{j,k}$ is the Kronecker delta function, which is 0 if $j \neq k$ and 1 if $j = k$. The integral above, then, is 0 unless $j = k$, in which case it is $L/2$. To see this, you can write

$$\sin(\pi k x/L) \sin(\pi j x/L) = \frac{1}{2} \cos(\pi(k-j)x/L) - \frac{1}{2} \cos(\pi(k+j)x/L),$$

and realize that unless $j = \pm k$, each of these cosines integrates to 0 over this range.

By multiplying the expression for $y(x)$ above by $\sin(\pi j x/L)$, and integrating the result from 0 to L, by the orthogonality property everything cancels except the $\sin(\pi j x/L)$ term, and we get the expression

$$a_j = \frac{2}{L} \int_{x=0}^{L} f(x) \sin(\pi j x/L) dx.$$

Now, the above sum of sines is a very useful way to represent a function which is 0 at both endpoints. If we are trying to represent a function on the real line which is periodic with period L,

it is not quite as useful. This is because the sum of sines above is not periodic with period L but only periodic with period $2L$. For periodic functions, a better Fourier expansion is

$$y(x) = a_0 + \sum_{j=1}^{\infty} a_j \cos(2\pi jx/L) + \sum_{k=1}^{\infty} b_k \sin(2\pi kx/L).$$

It is fairly easy to rewrite this as a sum of exponentials (over the complex numbers), using the identity $e^{ix} = \cos(x) + i\sin(x)$ which implies

$$\begin{aligned} \cos x &= \frac{e^{ix} + e^{-ix}}{2} \\ \sin x &= \frac{e^{ix} - e^{-ix}}{2i}. \end{aligned}$$

This results in the expression (with a different set of coefficients c_j)

$$y(x) = \frac{1}{L} \sum_{j=-\infty}^{\infty} c_j e^{2\pi ijx/L}, \tag{1}$$

where i is the standard imaginary unit with $i^2 = 1$. The scaling factor $\frac{1}{L}$ is introduced here for simplicity; we will see why shortly. The orthogonality relations are now

$$\int_{x=0}^{L} e^{2\pi ijx/L} e^{2\pi ikx/L} dx = \delta_{-j,k} L,$$

and thus, after dividing by L, we get that the integral is 0 or 1. This means that we now can recover the c_j coefficient from y by calculating the integral

$$c_j = \int_{x=0}^{L} y(x) e^{-2\pi ijx/L} dx. \tag{2}$$

(2) is referred to as the Fourier transform and (1) to as the inverse Fourier transform. If we hadn't introduced the factor $1/L$ in (1), we would have to include it in (2), but the convention is to put it in (1).

2 The Discrete Fourier Transform

Suppose that we have a function from some real-life application which we want to find the Fourier series of. In practice, we're not going to know the value of the function on every point between 0 and L, but just on some finite number of points. Let's assume that we have the function at n equally spaced points, and do the best that we can. This gives us the finite Fourier transform, also known as the Discrete Fourier Transform (DFT).

We have the function $y(x)$ on points j, for $j = 0, 1, \ldots, n-1$; let us denote these values by y_j for $j = 0, 1, \cdots, n-1$. We define the *discrete Fourier transform* of $y_0, \ldots, y_{n-1}$ to be the coefficients $c_0, \ldots, c_{n-1}$, where

$$c_k = \sum_{j=0}^{n-1} y_j e^{-2\pi ijk/n}, \tag{3}$$

for $k = 0, \cdots, n-1$.

Observe that it would not make sense to define (these complex Fourier coefficients) c_k for more values of k since the above expression is unchanged when we add n to k (since $e^{2\pi i} = 1$). This makes sense — if we start with n complex numbers y_j's, we end up with n complex numbers c_k's, so we keep the same number of degrees of freedom.

Can we recover the y_j's, given the c_k's? Yes, this is known as the *inverse Fourier transform*, and is stated below.

Theorem 1. *If $c_0, c_1, \ldots, c_{n-1}$ is the discrete Fourier transform of $y_0, \ldots, y_{n-1}$. Then*

$$y_j = \frac{1}{n} \sum_{k=0}^{n-1} c_k e^{2\pi i j k/n}, \tag{4}$$

for $j = 0, \cdots, n-1$.

The equation (4) is known as the inverse discrete Fourier transform. Observe that this is similar to (1), except that the scaling factor $\frac{1}{n}$ which replaces $\frac{1}{L}$ is not at the same place.[1]

The proof of Theorem 1 will be based on the following lemma.

Lemma 1. *If z is a complex number satisfying $z^n = 1$ and $z, z^2, \ldots, z^{n-1} \neq 1$ then we have the following "orthogonality relation": for all $j, k \in \{0, 1, \ldots, n-1\}$,*

$$\sum_{l=0}^{n-1} z^{jl} z^{-kl} = n\delta_{j,k},$$

where $\delta_{j,k}$ is the Kronecker delta function, which is 0 if $j \neq k$ and 1 if $j = k$.

Observe that $z = e^{-2\pi i/n}$ satisfies the condition of the Lemma 1 so the orthogonality relations turn into the sum

$$\sum_{l=0}^{n-1} e^{-2\pi i j l/n} e^{2\pi i k l/n} = n\delta_{j,k}.$$

Proof of Lemma 1.

$$\sum_{l=0}^{n-1} z^{jl} z^{-kl} = \sum_{l=0}^{n-1} (z^{j-k})^l = \sum_{l=0}^{n-1} w^l$$

where $w = z^{j-k}$. If $k = l$ then $w = 1$ and the above sum equals n. On the other hand, if $k \neq l$ then our assumption on $z^i \neq 1$ for $i \in \{1, 2, \cdots, n-1\}$ means that $w \neq 1$. Thus, we have $\sum_{l=0}^{n-1} w^l = (w^n - 1)/(w - 1) = 0$, since $w^n = 1$. □

We are now ready to prove the Theorem.

[1] This is just a matter of convention. Actually, to avoid the confusion that this $\frac{1}{n}$ factor may create, sometimes this factor of $1/n$ is distributed equally, with a $1/\sqrt{n}$ on both the forward and the inverse Fourier transforms; we will not use this.

Proof of Theorem 1. The definition of c_k can be written as

$$c_k = \sum_{j=0}^{n-1} y_j z^{kj},$$

where $z = e^{-2\pi i/n}$.

Now we can compute

$$\begin{aligned}
\frac{1}{n}\sum_{l=0}^{n-1} c_l e^{2\pi ijl/n} &= \frac{1}{n}\sum_{k=0}^{n-1} c_l z^{-jl} \\
&= \frac{1}{n}\sum_{l=0}^{n-1}(\sum_{k=0}^{n-1} y_k z^{kl}) z^{-jl} \\
&= \sum_{k=0}^{n-1} y_k \left(\frac{1}{n}\sum_{l=0}^{n-1} z^{kl} z^{-jl}\right) \\
&= y_j.
\end{aligned}$$

where the last equality comes from applying Lemma 1 to z (which shows that all but one of the inner sums are 0).

□

3 Computing the discrete Fourier transform

It's easy to compute the finite Fourier transform or its inverse if you don't mind using $O(n^2)$ computational steps. The formulas (4) and (3) above both involve a sum of n terms for each of n coefficients. However, there is a beautiful way of computing the finite Fourier transform (and its inverse) in only $O(n \log n)$ steps.

One way to understand this algorithm is to realize that computing a finite Fourier transform is equivalent to plugging into a degree $n-1$ polynomial at all the n n-th roots of unity, $e^{2\pi ik/n}$, for $0 \le k \le n-1$. (Recall that an n-th root of unity is any (complex) number such that $z^n = 1$; for example, the 4th root of unity are 1, $e^{i\pi/2} = i$, $e^{i\pi} = -1$ and $e^{i3\pi/2} = -i$.) The Fourier transform and its inverse are essentially the same for this part, the only difference being which n-th root of unity you use, and that one of them has to get divided by n. So, let's do the forward discrete Fourier transform (3).

Suppose we know the values of y_j and we want to compute the c_k using the Fourier transform, (3). Let the polynomial $p(x)$ be

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

Now, let $z = e^{-2\pi i/n}$. Then, it is easy to check that we have

$$c_k = p(z^k).$$

This shows we can express the problem of computing the Fourier transform as evaluating the polynomial p (of degree $n-1$) at the n-th roots of unity. (If we were computing the inverse one

(i.e. exchange the role of y_j and c_k), we would use the root $z = e^{2\pi i/n}$ and divide the overall result by $1/n$.)

What we will show is that if n is even, say $n = 2s$, it will be possible to find two degree $s-1$ polynomials (thus of degrees roughly half the degree of $p(x)$), p_{even} and p_{odd}, such that we get all n of the values c_k for $0 \le k \le n-1$ by plugging in the s-th roots of unity (rather than the n-th roots of unity) into p_{even} and p_{odd}. The evaluation of p at even powers of z will appear when evaluating p_{even}, and the odd powers of z will appear in p_{odd}. If n is a multiple of 4, we can then repeat this step for each of p_{even} and p_{odd}, so we now have our n values of c_k appearing as the values of four polynomials of degree $n/4-1$, when we plug the $\frac{n}{4}$-th units of unity, i.e., the powers of z^4, into all of them. If n is a power of 2, we can continue in the same way, and eventually reduce the problem to evaluating n polynomials of degree 0. But it's really easy to evaluate a polynomial of degree 0: the evaluation is the polynomial itself, which only has a constant term. So at this point we will be done.

The next question we address is: how do we find these two polynomials p_{even} and p_{odd}? We will do the case of p_{even} first. Let us consider an even power of z, say z^{2k}, at which we want to evaluate $p(\cdot)$. We look at the j-th term and the $(j+s)$-th term. These are

$$y_j z^{2kj} \quad \text{and} \quad y_{j+s} z^{2kj+2ks}.$$

But since $z^{2s} = z^n = 1$, we have

$$z^{2kj+2ks} = z^{2kj}.$$

Thus, we can combine these terms into a new term in the polynomial p_{even}, with coefficients

$$b_j = y_j + y_{j+s}.$$

If we let

$$p_{\text{even}}(x) = \sum_{j=0}^{s-1} b_j x^j$$

we find that

$$p(z^{2k}) = p_{\text{even}}(z^{2k}).$$

Observe furthermore that since z^k is an n-th root of unity, z^{2k} is an s-th root of unity (since $n = 2s$).

Now, let us do the case of the odd powers. Suppose we are evaluating p at an odd power of z, say z^{2k+1}. Again, let's consider the contribution from the j-th and the $(j+s)$-th terms together. This contribution is

$$y_j z^{(2k+1)j} + y_{j+s} z^{(2k+1)(j+s)}.$$

Here we find that $z^{(2k+1)s} = e^{(2\pi i)(2k+1)s/n} = e^{(\pi i)(2k+1)} = -1$. We now have

$$\begin{aligned} y_j z^{(2k+1)j} + y_{j+s} z^{(2k+1)(j+s)} &= (y_j z^j) z^{2kj} + (y_{j+s} z^j) z^{2kj} (-1) \\ &= (y_j - y_{j+s}) z^j \, z^{2kj}. \end{aligned}$$

Setting the j-th coefficient of p_{odd} to

$$\tilde{b}_j = (y_j - y_{j+s}) z^j$$

and letting

$$p_{\text{odd}}(x) = \sum_{j=0}^{s-1} \tilde{b}_j x^j$$

we see that

$$p(z^{2k+1}) = p_{\text{odd}}(z^{2k}).$$

What we just did was reduce the problem of evaluating one degree $n-1$ polynomial, p, at the n-th roots of unity to that of evaluating two degree $\frac{n}{2}-1$ polynomials, p_{odd} and p_{even} at the $\frac{n}{2}$-th roots of unity. That is, we have taken a problem of size n and reduced it to solving two problems of size $\frac{n}{2}$. We've seen this type of recursion before in sorting, and you should recognize that it will give you an $O(n \log n)$ algorithm for finding the finite Fourier transform.

So now, we can show how the Fast Fourier transform is done. Let's take $n = 2^t$. Now, consider an $n \times t$ table, as we might make in a spreadsheet. Let's put in our top row the numbers y_0 through y_{n-1}. In the next row, we can, in the first $\frac{n}{2}$ places, put in the coefficients of p_{even}, and then in the next $\frac{n}{2}$ places, put in the coefficients of p_{odd}. In the next row, we repeat the process, to get four polynomials, each of degree $\frac{n}{4}-1$. After we have evaluated the second row, we treat each of p_{even} and p_{odd} separately, so that nothing in the first $\frac{n}{2}$ columns subsequently affects anything in the last $\frac{n}{2}$ columns. In the third row, we will have in the first $\frac{n}{4}$ places the coefficients of $p_{\text{even,even}}$, which give us the value of $p(z^{4k})$ when we evaluate $p_{even,even}(z^{4k})$. Then in the next $\frac{n}{4}$ places, we put in the coefficients of $p_{\text{even,odd}}$. This polynomial will give the value of $p(z^{4k+2})$ when we evaluate $p_{\text{even,odd}}(z^{4k})$. The third $\frac{n}{4}$ places will contain the coefficients of $p_{\text{odd,even}}$, which gives us the values of $p(z^{4k+1})$. The last $\frac{n}{4}$ places will be occupied by the coefficients of $p_{\text{odd,odd}}$, which gives the values of $p(z^{4k+3})$. From now on, we treat each of these four blocks of $\frac{n}{4}$ columns separately. And so on.

There are two remaining steps we must remember to carry out. The first step arises from the fact that is that the values of $p(z^k)$ come out in the last row in a funny order. We have to reshuffle them so that they are in the right order. I will do the example of $n = 8$. Recall that in the second row, the polynomial p_o, giving odd powers of z, followed p_e, giving even powers of z. In the third row, first we get the polynomial giving z^{4k}, then z^{4k+2}, then z^{4k+1}, then z^{4k+3}. So in the fourth row (which is the last row for $n = 8$), we get the values of $p(z^k)$ in the order indicated below.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td colspan="8">coefficients of p</td></tr>
<tr><td colspan="4">$p_{\text{e}}(z^{2k}) = p(z^{2k})$</td><td colspan="4">$p_{\text{o}}(z^{2k}) = p(z^{2k+1})$</td></tr>
<tr><td colspan="2">$p_{\text{e,e}}(z^{4k}) = p(z^{4k})$</td><td colspan="2">$p_{\text{e,o}}(z^{4k}) = p(z^{4k+2})$</td><td colspan="2">$p_{\text{o,e}}(z^{4k}) = p(z^{4k+1})$</td><td colspan="2">$p_{\text{o,o}}(z^{4k}) = p(z^{4k+3})$</td></tr>
<tr><td>$p(z^0)$</td><td>$p(z^4)$</td><td>$p(z^2)$</td><td>$p(z^6)$</td><td>$p(z^1)$</td><td>$p(z^5)$</td><td>$p(z^3)$</td><td>$p(z^7)$</td></tr>
</table>

You can figure out where each entry is supposed to go is by looking at the numbers in binary, and turning the bits around. For example, the entry in column 6 (the 7th column as we start labeling with 0) is $p(z^3)$. You can figure this out by expressing 6 in binary: 110. You then read this binary number from right to left, to get 011, which is 3. Thus, the entry in the 6 column is $p(z^3)$. The reason this works is that in the procedure we used, putting in the even powers of z first, and then the odd powers of z, we were essentially sorting the powers of z by the 1's bit. The next row ends up sorting them by the 2's bit, and the next row the 4's bit, and so forth. If we had sorted starting with the leftmost bit rather than the rightmost, this would have put the powers in numerical order. So, by numbering the columns in binary, and reversing the bits of these binary numbers, we get the right order of the transformed sequence.

The other thing we have to do is to remember to divide by n if it is necessary. We only need do this for the inverse Fourier transform, and not the forward Fourier transform.

4 Computing convolutions of sequences using Fast Fourier Transform

Suppose you have two sequences $f_0, f_1, \ldots, f_{n-1}$ and $g_0, g_1, \ldots, g_{n-1}$ and want to compute the sequence $h_0, h_1, \ldots, h_{n-1}$ defined by

$$h_k = \sum_{j=0}^{n-1} f_j g_{k-j}$$

where the index $k-j$ is taken modulo n. Clearly it is possible to compute the numbers $h_0, \ldots, h_{n-1}$ in n^2 arithmetic operations. We will now explain how to do it faster.

Let a_k and b_k be the discrete Fourier transform of f_k and g_k and their finite Fourier, that is,

$$\begin{aligned} a_k &= \sum_j f_j e^{-2\pi ijk/n} \\ b_k &= \sum_j g_j e^{-2\pi ijk/n}. \end{aligned}$$

Now let's compute the inverse Fourier transform of the sequence $a_k b_k$. For all $l = 0, \cdots, n-1$ we get :

$$\begin{aligned} \frac{1}{n}\sum_k e^{2\pi ilk/n} a_k b_k &= \frac{1}{n}\sum_k e^{2\pi ilk/n} \sum_j f_j e^{-2\pi ijk/n} \sum_{j'} g_{j'} e^{-2\pi ij'k/n} \\ &= \frac{1}{n}\sum_j \sum_{j'} f_j g_{j'} \sum_k e^{2\pi ik(l-j-j')/n} \\ &= \sum_{j=0}^{n-1} f_j g_{l-j} \\ &= h_l \end{aligned}$$

where the second last equality holds because the sum over k is 0 unless $l \equiv j + j' \pmod{n}$.

We have just found a way of computing the sequence $h_0, h_1, \ldots, h_{n-1}$ by first applying Fourrier transform to the sequences $f_0, f_1, \ldots, f_{n-1}$ and $g_0, g_1, \ldots, g_{n-1}$ and then taking the inverse Fourier transform of the sequence $a_k b_k$. Since the Fourier transforms and inverse Fourrier transform can be computed in $O(n \log(n))$ operations, the sequence $h_0, h_1, \ldots, h_{n-1}$ can be computed in $O(n \log(n))$ instead of $O(n^2)$ operations.

We can now use this method in order to multiply polynomials efficiently. Suppose we have two degree d polynomials, and we want to multiply them. This corresponds to convolution of the two series that make up the coefficients of the polynomials. If we do this the obvious way, it takes $O(d^2)$ steps. However, if we use the Fourier transform, multiply them pointwise, and transform back, we use $O(d \log d)$ steps for the Fourier transforms and $O(d)$ steps for the multiplication. This gives $O(d \log d)$ total, a great savings. We must choose the n for the Fourier series carefully. If we multiply two degree d polynomials, the resulting polynomial has degree $2d$, or $2d+1$ terms. We

must choose $n \geq 2d+1$, because we need to have room in our sequence f_0, f_1, $\ldots$ f_{n-1} for all the coefficients of the polynomial; if we choose n too small, the convolution will "wrap around" and we'll end up adding the last terms of our polynomial to earlier terms.

5 Fourier transforms modulo p and fast integer multiplication

So far, we've been doing finite Fourier transforms over the complex numbers. We can actually work over any field with a *primitive n-th root of unity*, that is, a number z such that $z^n = 1$ and $z, z^2, \ldots, z^{n-1} \neq 1$. Indeed if such a z exists, we can define the Fourier transform of some number $y_0, \ldots, y_{n-1}$ as

$$c_k = \sum_{j=0}^{n-1} y_j z^{-jk}.$$

In this case we can prove similarly as in Section 2 that the inverse Fourier transform is

$$y_j = n^{-1} \sum_{k=0}^{n-1} c_k z^{jk}.$$

The factor n^{-1} is the multiplicative inverse of n over this field, and comes from the fact that $\sum_{k=0}^{n-1} z^0 = n$.

If we take a prime p, then the field of integers mod p has a primitive n-th root of unity if $p = mn + 1$ for some integer m. In this case, we can take the Fourier transform over the integers mod p. Thus, 17 has a primitive 16-th root of unity, one of which can be seen to be 3. (By Fermat's little theorem, any $a \neq 0$ satisfies $a^{16} \equiv 1 \pmod{17}$, but for many a's, a smaller power than 16 will give 1. For example, modulo 17, 1 is a primitive 1st root of unity, 16 is a primitive 2nd root of unity, 4 and 13 are primitive 4-th root of unity, 2, 8, 9 and 15 are primitive 8th roots of unity and 3, 5, 6, 7, 10, 11, 12 and 14 are primitive 16-th root of unity.) So if we use $z = 3$ in our fast Fourier transform algorithm, and take all arithmetic modulo 17, we get a finite Fourier transform. And we have seen how to compute n^{-1} modulo a prime p by the Euclidean gcd.

We can use this for multiplying polynomials. Suppose we have two degree d polynomials, each of which has integer coefficients of size less than B. The largest possible coefficient in the product is $(B-1)^2(d+1)$. If we want to distinguish between positive and negative coefficients of this size, we need to make sure that $p > 2(B-1)^2(d+1)$. We also need to choose $n \geq 2d+1$, so as to have at least as many terms as there are coefficients in the product. We can then use the Fast Fourier transform (mod p) to multiply these polynomials, with only $O(d \log d)$ operations (additions, multiplications, taking remainders modulus p), where we would have needed d^2 originally.

Now, suppose you want to multiply two very large integers. Our regular representation of these integers is as $\sum_k d_k 10^k$, where d_k are the digits. We can replace this by $\sum_k d_k x^k$ to turn it into a polynomial, then multiply the two polynomials using the fast Fourier transform.

How many steps does this take? To make things easier, let's assume that our large integers are given in binary, and that we use a base B which is a power of 2. Let's assume the large integers have N bits each and that we use a base B (e.g., 10 in the decimal system, 2 in binary) that has b bits. We then have our number broken up into N/b "digits" of b bits each. How large does our prime have to be? It has to be larger than the largest possible coefficient in the product of our two

polynomials. This coefficient comes from the sum of at most N/b terms, each of which has size at most $(2^b - 1)^2 < 2^{2b}$. This means that we are safe if we take p at least

$$(\frac{N}{b})2^{2b}$$

or taking logs, p must have around $2b + \log_2 \frac{N}{b}$ bits.

Rather than optimizing this perfectly, let's just set the two terms in this formula to be approximately equal by letting $b = \log_2 N$; this is much simpler and will give us the right asymptotic growth rate. We thus get that p has around $3 \log_2 N$ bits. We then set n to be a power of 2 larger than $2\frac{N}{b}$, so that our finite Fourier transform involves $O(n \log n) = O(N)$ operations, each of which may be an operation on a $(3 \log_2 N)$-bit number. If we use longhand multiplication and division (taking $O(b^2)$ time) to do these operations, we get an $O(N \log^2 N)$-time algorithm.

There's no reason that we need to stop there. We could always use recursion and perform these operations on the $3b$-bit numbers using fast integer multiplication as well. If we use two levels of recursion, we get an $O(N \log N (\log \log N)^2)$ time algorithm. If we use three levels of recursion, we get an $O(N \log N (\log \log N)(\log \log \log N)^2$ time algorithm, and so forth.

It turns out, although we won't go into the details, that you can get a $O(N \log N \log \log N)$ time algorithm. The main difference from what we've done is that you choose the number you use to do the FFT not of size around $\log N$, but of a number of the form $2^{2^k} + 1$ of size around $\sqrt{N}$ (it actually doesn't have to be prime). You then carefully compute the time taken by applying this algorithm recursively, making sure that you use the fact that mod $2^{2^k} + 1$, multiplication by small powers 2 can be accomplished fairly easily by just shifting bits. Details can be found in Aho, Hopcroft and Ullman's book "Design and Analysis of Computer Algorithms." In fact, very recently, still using the finite Fourier transform, Fürer found a way to speed up multiplication even further so that the running time is only a tiny bit more than $O(N \log N)$.

18.310 Principles of Discrete Applied Mathematics
Fall 2013