Fast Fourier Transforms

Prof. Michel Goemans and Peter Shor

1 Introduction: Fourier Series

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 the product of these sines as half of the difference between $\cos(\pi(k-j)x/L)$ and $\cos(\pi(k+j)x/L)$, 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 for a periodic function, we need $f(0) = f(L)$ and $f'(0) = f'(L)$. For the sum of sines above, the terms with odd k such as $\sin(\pi x/L)$ are not themselves periodic with period L. For periodic functions, a better Fourier expansion is

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

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 a_j)

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

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

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

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

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

(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 Finite 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 jL/n, 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 the y_j's by

$$a_k = \sum_j y_j e^{-2\pi i j k/n}, \tag{3}$$

for $k = 0, \cdots, n-1$. Observe that it would not make sense to define (these complex Fourier coefficients) a_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 a_k's, so we keep the same number of degrees of freedom. The a_k's here are the finite Fourier transform of the y_j's.

How do we recover the y_j's, given the a_k's? It's not hard to see that it works in essentially the same way that it did for the complex Fourier series we talked about earlier, only we have to replace an integral with a sum. Thus, we get

$$y_j = \frac{1}{n} \sum_k a_k e^{2\pi i j k/n}, \tag{4}$$

for $j = 0, \cdots, n-1$. (4) is known as the inverse discrete Fourier transform. The expression is similar to (4), except for the + sign (rather than −) in the exponent, and for the scaling factor $\frac{1}{n}$. (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.)

How do we prove that the formulas (3) and (4) are inverse transforms of each other? The proof works the same way as it does for the Fourier series, and in fact this formula can be derived from (2) and (1). The orthogonality relations turn into the sum

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

and the derivation of (4) from (3) is left as an exercise.

3 Computing the finite 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, $e^{2\pi i k/n}$, 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 a_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

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

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 a_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 a_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 a_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 $z^{(2k+1)s} = -1$. Why? Again, $z^{2ks} = 1$, and z^s is a square root of 1, because when we square it we get $z^{2s} = z^n = 1$. Since z is a primitive n-th root of 1 (as we said, z is either $e^{2\pi i/n}$ or $e^{-2\pi i/n}$), z^s is not 1, so it must be -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

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

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 recusion 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.

0	1	2	3	4	5	6	7
coefficients of p							
$p_{\text{e}}(z^{2k}) = p(z^{2k})$				$p_{\text{o}}(z^{2k}) = p(z^{2k+1})$			
$p_{\text{e,e}}(z^{4k}) = p(z^{4k})$		$p_{\text{e,o}}(z^{4k}) = p(z^{4k+2})$		$p_{\text{o,e}}(z^{4k}) = p(z^{4k+1})$		$p_{\text{o,o}}(z^{4k}) = p(z^{4k+3})$	
$p(z^0)$	$p(z^4)$	$p(z^2)$	$p(z^6)$	$p(z^1)$	$p(z^5)$	$p(z^3)$	$p(z^7)$

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 lablling 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 Multiplication and Convolution

Let's go back to the complex Fourier series. Suppose we have two functions f and g. Suppose we have the Fourier series for these two functions, that is,

$$f(x) = \frac{1}{L} \sum_{j=-\infty}^{\infty} a_j e^{2\pi i j x/L}$$

and

$$g(x) = \frac{1}{L} \sum_{k=-\infty}^{\infty} b_k e^{2\pi i k x/L}$$

How do we find the Fourier series of the sum of these two functions? It's easy. We take the sum of the Fourier coefficients.

$$f(x) + g(x) = \frac{1}{L} \sum_{k=-\infty}^{\infty} (a_k + b_k) e^{2\pi i k x/L}$$

How about the product? This requires some calculation.

$$f(x)g(x) = \frac{1}{L^2} \sum_{j,k=-\infty}^{\infty} a_j b_k e^{2\pi i (j+k) x/L}$$

Now, what is the $e^{2\pi i l x/L}$ term of this Fourier series? We get a contribution to the l term exactly when $j + k = l$. So,

$$f(x)g(x) = \frac{1}{L^2} \sum_{l=-\infty}^{\infty} \left(\sum_{j=-\infty}^{\infty} a_j b_{l-j} \right) e^{2\pi i l x/L}$$

This sequence, $c_l = \sum_j a_j b_{l-j}$, is known as the *convolution* of sequences a_j and b_k. We recently saw how convolutions come up in generating functions. If you multiply two generating functions together, you take the convolution of their respective sequences. The same thing is true in the Fourier transform, in Fourier series and the finite Fourier transform: taking the Fourier transform turns pointwise multiplication into convolution, and vice versa, modulo the right constant factor.

This was not done in class, but it might be worth going through in these notes. Let's check that the convolution of functions turns into multiplication of the Fourier series. Remember that

$f(x)$ and $g(x)$ are both periodic with period L. Their convolution is

$$\begin{aligned} h(y) &= \int_0^L f(x)g(y-x)dx \\ &= \frac{1}{L^2}\int_0^L \sum_j a_j e^{2\pi ijx/L} \sum_k b_k e^{2\pi ik(y-x)/L} dx \\ &= \frac{1}{L^2}\int_0^L \sum_j \sum_k a_j b_k e^{2\pi i(ky+(j-k)x)/L} dx \\ &= \frac{1}{L}\sum_k a_k b_k e^{2\pi iky/L} \end{aligned}$$

where we only get the terms with $j = k$ because $\int_0^L e^{2\pi i(j-k)x/L}dx$ is 0 if $j \neq k$.

Now, let's work out the details of the fact that convolution turns into pointwise multiplication for finite Fourier transforms. Suppose we have two sequences a_j and b_j, and their finite Fourier transforms, i.e.,

$$\begin{aligned} f_k &= \sum_j a_j e^{-2\pi ijk/n} \\ g_k &= \sum_j b_j e^{-2\pi ijk/n}. \end{aligned}$$

What do we get when we take the inverse Fourier transform of the pointwise product $f_k g_k$? Let

$$f_k g_k = \sum_j c_j e^{-2\pi ijk/n}.$$

Then taking the inverse Fourier transform, we get (for $l = 0, \cdots, n-1$):

$$\begin{aligned} c_l &= \frac{1}{n}\sum_k e^{2\pi ilk/n} f_k g_k \\ &= \frac{1}{n}\sum_k e^{2\pi ilk/n} \sum_j a_j e^{-2\pi ijk/n} \sum_{j'} b_{j'} e^{-2\pi ij'k/n} \\ &= \frac{1}{n}\sum_j \sum_{j'} a_j b_{j'} \sum_k e^{2\pi ik(l-j-j')/n} \\ &= \sum_{j=0}^{n-1} a_j b_{l-j} \end{aligned}$$

where $l - j$ has to be understood modulo n, and the last equality holds because the sum over k is 0 unless $l \equiv j + j'$ (mod n). Thus, (wrapped around) convolution turns into pointwise multiplication for the finite Fourier transform. The inverse is almost true. Consider the pointwise multiplication of the two series a_j's and b_j's. Its finite Fourier transform is given by

$$h_k = \sum_j a_j b_j e^{-2\pi ijk/n}.$$

Therefore, we can derive that

$$\begin{aligned} h_k &= \sum_j a_j b_j e^{-2\pi ijk/n} \\ &= \sum_j \frac{1}{n}\left(\sum_l f_l e^{2\pi ijl/n}\right)\frac{1}{n}\left(\sum_{l'} g_{l'} e^{2\pi ijl'/n}\right) e^{-2\pi ijk/n} \\ &= \frac{1}{n^2}\sum_l \sum_{l'} f_l g_{l'} \sum_k e^{2\pi ij(l+l'-k)} \\ &= \frac{1}{n}\sum_l f_l g_{k-l}, \end{aligned}$$

and again indices have to be understood modulo n. Thus, the finite Fourier transform of the pointwise multiplication is equal to $\frac{1}{n}$ times the (wrapped around) convolution of the finite Fourier transforms.

There's actually an other way of seeing these relations for finite Fourier transforms. If you think of the Fourier series as the coefficients of a polynomial, the Fourier transform is what you get if you evaluate the polynomial at a root of unity. Now, suppose you have two polynomials, $a(x)$ and $b(x)$, and you multiply them? What happens to the coefficients of the polynomial? You take the convolution of them to get $a(x)b(x)$. And what happens to the pointwise evaluation when you multiply? You multiply pointwise. So the FFT turns convolution into multiplication, and vice versa; this is modulo a scaling factor of n.

We can use this fact that convolution turns into pointwise multiplication 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 a_0, a_1, $\ldots a_{n-1}$ 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 generalize them to work over any field with a primitive n-th root of unity. Suppose z is a primitive n-th root of unity (primitive means that $z^k \neq 1$ for $0 < k < n$). The main fact we used to derive the finite Fourier transform is that if $z \neq 0$ and $z^n = 1$, then

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

This holds for any field with a primitive n-th root of unity. The transforms work the same way as in Eqs. (4) and (3). The Fourier transform is

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

while the inverse Fourier transform is

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

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

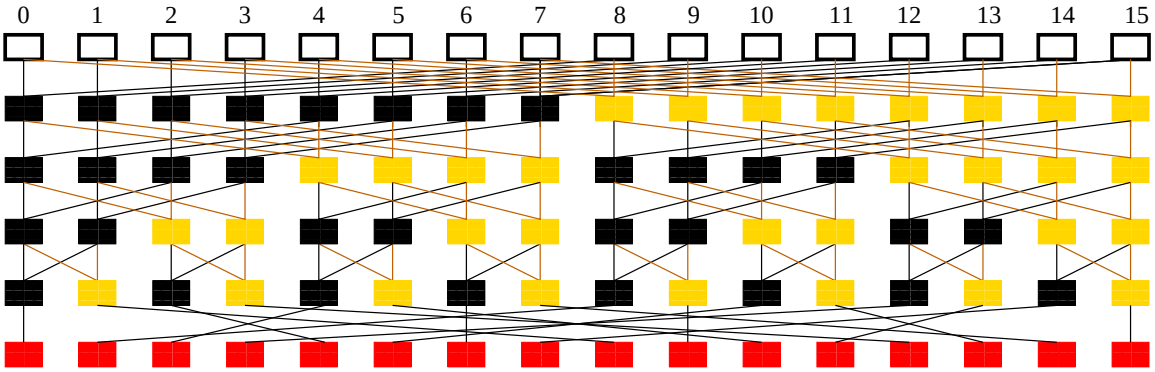
growth rate. We thus get that p has around $3\log_2 N$ bits. We then set n to be a power of 2 larger than $2\frac{N}{b}$, so that our finite Fourier transform involves $O(n \log n) = O(N)$ operations, each of which may be an operation on a $(3\log_2 N)$-bit number. If we use longhand multiplication and 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 $O(N \log N)$.

6 Details of making a spreadsheet

It seems at first like the FFT is perfectly suited for a spreadsheet. It can be illustrated beautifully by a two-dimensional diagram. However, once you start looking at the details, it gets fairly tricky. The diagram illustrating the FFT is:



FFT Figure

Here, the outlined boxes in the top row contain our input. In every black box we put $y_j + y_{j+s}$ where y_j and y_{j+s} are the entries in the two boxes in the row connected to this black box by black lines. In each yellow box, we put $(y_j - y_{j+s})z^j$, where now y_j and y_{j+s} are the boxes in the row above connected to this yellow box by brown lines. One thing to remember is that z (as well as y) changes from row to row. In calculating the boxes in the second row, we use our original z. In the third row, we use z^2. (So in terms of our original z, we are really multiplying by z^{2j} here and not z^j.) In the fourth row, we use z^4. In the fifth row, it turns out we don't need to use z at all, since in all these yellow boxes we have $j = 0$.

One tricky thing about this is that we have to start counting j from the first box of the contiguous row of yellow boxes it's in, and not from the column. So, for example, in row 3 you need to get j by taking the column number mod 4 and not just the column number.

After row 5, we're done with all the addition steps, and have only one or two steps left. In row 6, we shuffle all the numbers into the red boxes according to the reversal-of-binary-digits permutation.

Finally, depending on whether we're doing the FFT or the inverse FFT, we may need to make a row 7 where we divide row 6 by n, which in this case is 16.

If we're doing an FFT mod p for some prime p, all of these arithmetic operations are going to have to be done mod p, and in the last step, we will have to multiply by n^{-1}, which is the integer satisfying

$$n \cdot n^{-1} = 1 \bmod p.$$

This multiplicative inverse can be found by Euclid's gcd algorithm.

There are a number of issues that can arise when you construct the spreadsheet for the homework. The first is the issue of copying formulas in spreadsheets. Suppose you want to give a spreadsheet for the FFT in which you can easily change the prime p and/or the root of unity z. If you put z or p into the formulas, then when you copy the spreadsheet you'll have to change them all by hand. Furthermore, if you get them by using an absolute reference like \$A\$3 (say to a cell containing p or z) then when you copy this cell, the reference in the copy will still points to A3. If you want to change the values of p and z, you will need to replace all these references.

Of course, you can argue that this is what the find-and-replace tool in the spreadsheet editor is intended for. Here is another way. Suppose we start our spreadsheet with the three rows

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	
1	0	1	2	3	4	5	6	7	8	9	10	11	...
2	p	p	p	p	p	p	p	p	p	p	p	p	...
3	z	z	z	z	z	z	z	z	z	z	z	z	...

Then if we want to make a reference to p, we can use a reference like C\$2. This uses the value in the same column, but the second row. Now, we can copy our spreadsheet horizontally. This changes C to column farther to the right, and we can put a new prime p in the second row of our copy. As long as we don't change the rows our spreadsheet occupies, but just move it horizontally, we're fine.

We now need to calculate the values of b_j using the equations

$$\begin{aligned} b_j &= y_j + y_{j+s} \\ \tilde{b}_j &= (y_j - y_{j+s})z^j. \end{aligned}$$

The first case (b_j) is easy. The second case ($\tilde{b}_j$) is trickier, because of the z^j term. The obvious way is to calculate it is to use a statement like

$$= \text{MOD}(\ (\text{J5} - \text{N5}) * z^{(2*j)},\ p).$$

to calculate N6 in row 6, where $s = 4$ and we multiply by $(z^2)^j$. There are two problems with something like this. The first is that, if row 2 is all p's and row 3 is all z's, we can get p and z by N\$2 and N\$3, but where do we get the j from? In row 3, we have two stretches of four yellow cells, and we want j to start counting from 0 each time. If we just reference j up in the first stretch,

we'll have to change the formula if we copy it to the second stretch. This can be solved by using MOD(j, 4) in the formula.

The second problem is somewhat more serious. This is that using the spreadsheet's arithmetic to compute z^{2j} is very likely to give you an integer overflow, if z or j is large. What we could do is to create an extra row that contains powers of z mod p, and use the OFFSET function to find the right cell in it. This works fine, but it's kind of complicated and messy, and it's easy for mistakes to sneak in.

Here is an easier solution: create *another* $(n \times \log_2 n)$ array in your spreadsheet that contains the numbers z^j that you need to multiply by $y_j - y_{j+s}$ in the $\tilde{b}$ cells. For $n = 16$, the numbers you need to multiply them by look like

–	–	–	–	–	–	–	–	1	z	z^2	z^3	z^4	z^5	z^6	z^7
–	–	–	–	1	z^2	z^4	z^6	–	–	–	–	1	z^2	z^4	z^6
–	–	1	z^4	–	–	1	z^4	–	–	1	z^4	–	–	1	z^4
–	1	–	1	–	1	–	1	–	1	–	1	–	1	–	1

where all of the powers are performed mod p. Here – means that we have b instead of $\tilde{b}$ in the corresponding cell, so there's no z^j term needed for that cell.

How can we construct this second array? Since we'll never actually be looking at the – cells, we can fill them in anyway we want, and it's easiest to fill them in with the same sequence we find in $\tilde{b}$ cells.

For example, for $n = 16$, if $p = 193$, and $z = 3$ is our 16th root of 1, we get

1	3	9	27	81	50	150	64	1	3	9	27	81	50	150	64
1	9	81	150	1	9	81	150	1	9	81	150	1	9	81	150
1	81	1	81	1	81	1	81	1	81	1	81	1	81	1	81
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

How do we create this table? It's easy. We get the first eight cells in the top row by sticking in a 1 on the left, and then repeatedly multiplying by 3 mod 193. We can get the rest of the cells by copying the cell 8 positions to the left. In the second row, we can get the first four cells by squaring the cells above them mod 193, and the rest by looking 4 positions to the left. In the third row, we can get the first two cells by squaring the celle above them, and the rest by looking 2 positions to the left. Larger FFT's will work similarly.

Unless you want to use a search-and-replace function, you should remember to get 3 and 193 by referencing cells that contain them, rather than coding them into the formulas.

Finally, we need to shuffle all the entries around, by copying the entry in each cell to the column obtained by reversing the digits of the column's number in binary. For example, if $n = 16$, you would move the entry in the 5th (= 0101) position to the 10th (= 1010) position. it is not too painful to construct this shuffle by hand. You can also do it by taking an additional $\log n$ rows. Write the column's number in binary in these rows, and then add these up in reverse order to get the column the answer was shuffled to. This can be done several ways; one way is to use the OFFSET function.

You also need to remember to divide by n (if it is an inverse transform). You do this by multiplying by n^{-1} (mod p). Calculating n^{-1} can be done using the Euclidean algorithm, which we have covered earlier in 18.310.

There are a lot of ways to make errors in this process. One thing you can do to make it easier is to start out by making a spreadsheet which uses $z = 3$ and $p = 17$ or 193, and change the numbers to be bigger later. You can also put in test sequences where you can easily figure out what the FFT should be, such as putting in one 1 and the rest 0's. It is useful to color the cells using the b formula one color and those using the $\tilde{b}$ formula a different color, so that one can visually see if the formulas have been misplaced by mistake.

We have explained how to do the FFT on a spreadsheet using two different types of cells in each row: one type that computes b and another that computes $\tilde{b}$. If you use slightly more complicated formulas, you might be able to use just two formulas total for all of these rows, again, one for $\tilde{b}$ and one for b, but where these two types don't need to be customized for each rows. It is probably hard to do this without using use the OFFSET function (which generally makes things more complicated), and you also need some way to figure out which row you are in (which also makes things more complicated, although not by much), so unless you can figure out some clever way of doing it without the OFFSET function, it's probably not worth the extra trouble.