# Fast Fourier Transform Notes
# 18.310, Fall 2005, Prof. 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

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

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:

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

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 a constant multiple 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 $\sin \pi x$ 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, using the identities

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

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

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

The orthogonality relations are now

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

This means that we now can recover the $a_j$ coefficient from $y$ by performing the integral

$$a_j = \frac{1}{L}\int_{x=0}^{L} y(x) e^{-2\pi i j x/L} \tag{2}$$

## 2 The Fourier transform

Given a function $f(x)$ defined for all real $x$, we can give an alternative representation to it as an integral rather than as an infinite series, as follows

$$f(x) = \int e^{ikx} g(k) dk$$

Here $g(x)$ is called the Fourier transform of $f(x)$, and $f(x)$ is the inverse Fourier transform of $g(x)$. This is a very important tool used in physics.

One reason for this is that exponential functions $e^{ikx}$, which $f$ is written as an integral sum of, are eigenfunctions of the derivatives. That is, the derivative, acting on an exponential, merely multiplies the exponential by $ik$. This makes the Fourier transform a useful tool in dealing with differential equations.

Another example of the Fourier transform's applications can be found in quantum mechanics. We can represent the state of a particle in a physical system as a *wave function* $\phi(x)$, and the probability that the particle in this state has a position lying between $x$ and $x + dx$ is $|\phi(x)|^2 dx$.

The same state can also be represented by its wave function in *momentum space*, and that wave function of the variable $p$ is a constant multiple of the Fourier transform of $\phi(x)$:

$$\psi(p) = c \int e^{ikx} \phi(x) dx$$

We can derive a formula for computing the Fourier transform in much of the same way we can compute Fourier series. The resulting formula is

$$g(x) = \frac{1}{2\pi} \int e^{-ikx} f(x) dx$$

where the integration is over all real values of $x$.

The Fourier transform can be obtained by taking the Fourier series and letting $L$ go to $\infty$. This turns both the function and its Fourier series into functions defined over the real line. The finite Fourier transform arises by turning these both into a finite sequence, as shown in the next section.

## 3 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.

We have the function $y(x)$ on points $jL/n$, for $j = 0, 1, \ldots, n-1$. We would like to represent the function

$$y(x) = \sum_k a_k e^{2\pi ikx/L}$$

but, of course, we only have knowledge of $y$ at values of $x = jL/n$. If we plug in $x = jL/n$ for the value of $x$, we get

$$y_j = \sum_k a_k e^{2\pi ijk/n}. \tag{3}$$

There is one more thing to notice here. Namely, that adding $n$ to $k$ doesn't change any of the values of $e^{2\pi ijk/n}$ because $e^{2\pi i} = 1$. Thus, if we just have $n$ equally spaced points, we only need the first $n$ terms from the Fourier series to perfectly match the values of $y$ at our points. This makes sense — if we start with $n$ complex numbers $y_j$, we end up with $n$ complex numbers $a_k$, so we keep the same number of degrees of freedom. The $a_k$ here are the finite Fourier transform of the $y_j$.

How do we compute the $a_k$, given the $y_j$. 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

$$a_k = \frac{1}{n} \sum_j y_j e^{-2\pi ijk/n}. \tag{4}$$

Here, again the $a_k$ are the finite Fourier transform of the $y_j$. The inverse Fourier transform given above (Eq. 3) is the same as the finite Fourier transform, except we have put a $-$ sign on the exponent, and left out the factor of $1/n$. In fact, you will sometimes see the factor of $1/n$ distributed equally, with a $1/\sqrt{n}$ on both the forwards and the inverse Fourier transforms.

How do we prove that the formulas (4) and (3) 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

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

I'll let you figure out the rest of it.

## 4 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 (3) and (4) 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 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 $n$ roots of unity, $e^{2\pi ik/n}$, for $0 \le k \le n-1$. 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. So, let's be consistent with Prof. Kleitman's notes and do the inverse Fourier transform.

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

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

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

$$y_j = p(z^j).$$

This shows we can express the problem of the inverse Fourier transform as evaluating the polynomial $p$ at the $n$-th roots of unity.

What we will show is that if $n$ is even, say $n = 2s$, it will be possible to find two degree $s-1$ polynomials, $p_{\text{even}}$ and $p_{\text{odd}}$, such that we get all $n$ of the values $y_j$ for $0 \le j \le n-1$ by plugging in the $s$-th roots of unity into $p_{\text{even}}$ and $p_{\text{odd}}$. The even powers of $z$ will appear in $p_{\text{even}}$, and the odd powers of $z$ will appear in $p_{\text{odd}}$. If $n$ is a multiple of 4, we can then repeat this step for each of $p_{\text{even}}$ and $p_{\text{odd}}$, so we now have our $n$ values of $y_j$ 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 $p_{\text{even}}$ and $p_{\text{odd}}$? We will do the case of $p_{\text{even}}$ first. Let us consider an even power of $z$, say $z^{2k}$. We will look at the $j$-th term and the $(j+s)$-th term. These are

$$a_j z^{2kj} \quad \text{and} \quad a_{j+s} z^{2kj+2ks}$$

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 $p_{\text{even}}$, with coefficients

$$b_j = a_j + a_{j+s}$$

If we let

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

we find that

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

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

$$a_j z^{(2k+1)j} + a_{j+s} z^{(2k+1)(j+s)}.$$

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 $z^2 s = z^n = 1$. Since $z$ is a primitive $n$-th root of 1, $z^s$ is not 1, so it must be $-1$. We now have

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

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

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

and letting

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

we see that

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

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 $a_0$ through $a_{n-1}$. In the next row, we can, in the first $\frac{n}{2}$ places, put in the coefficients of $p_{\text{even}}$, and then in the next $\frac{n}{2}$ places, put in the coefficients of $p_{\text{odd}}$. 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 $p_{\text{even}}$ and $p_{\text{odd}}$ 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 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 the 6 column 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 reversing the bits and sorting, we get the right order.

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 Fourier transform, and not the inverse Fourier transform.

## 5 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,

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

and

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

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.

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

How about the product? This requires some calculation.

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

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

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

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 turns into convolution, and vice versa.

I didn't do this in class, but I think 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

$$\begin{aligned} h(y) &= \int_0^L f(x)g(y-x)dx \\ &= \int_0^L \sum_j a_j e^{2\pi ijx/L} \sum_k b_k e^{2\pi ik(y-x)/L} dx \end{aligned}$$

$$\begin{aligned} &= \int_0^L a_j b_k e^{2\pi i(ky+(j-k)x)/L} dx \\ &= L \sum_k a_k b_k e^{2\pi iky/L} \end{aligned}$$

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

Wait a minute! Where did that $L$ come from? We didn't get it when we multiplied $f$ and $g$. I don't have any good intuitive explanation for it right now, but if you look at the equations, you can see it's really there. Remember this extra $L$. We'll see it again in a couple of paragraphs.

Now, let's work out the details of the fact that pointwise multiplication turns into convolution for finite Fourier series. Suppose we have two sequences 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 Fourier transform, we get

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

where the last equality holds because the sum over $k$ is 0 unless $l = j + j'$. Thus, pointwise multiplication turns into convolutions for the finite Fourier transform as well.

We can use this fact that pointwise multiplication turns into convolution 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.

One can ask the question: when you multiply polynomials by taking the Fourier transforms and multiplying pointwise, how many times do you have to divide by $n$? At first sight, it seems like the answer should be inconsistent. If we first take the inverse Fourier transform, Eq. (3), of each of the polynomials, multiply then, and then take the Fourier transform, we end up dividing by $n$ when we take the forwards Fourier transform, so we divide by $n$ once. On the other hand, if we take the forwards Fourier transform of the polynomials first, we have to divide each of these polynomials by $n$. Then, when we multiply them together, we have divided each of these by $n$, so we've ended up dividing by $n^2$ total. Sot it seems like, when we take the inverse Fourier transform, we should end up with an extra division by $n$ than we did if we took the inverse Fourier transform first. Both of these methods can't be right, since the two answers differ by a factor of $n$. What's going on?

The correct answer, as you should be able to figure out if you sit down and write everything out, is that you only divide by $n$ once. This is related to the extra $L$ term we mentioned above, which turns into an $n$ when you turn Fourier series into finite Fourier transforms, and cancels out one of the extra factors of $n$.

## 6 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 our $n$-th root of unity. The main fact we need to have the finite Fourier transforms work 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. (3) and (4). The inverse Fourier transform is

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

and the forward Fourier transform is

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

If we take a prime $p$, then the field of integers mod $p$ has an $n$-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 16th roots of unity, one of which can be seen to be 3. So if we use $z = 3$ in our fast Fourier transform algorithm, and take all arithmetic modulo 17, we get a finite Fourier transform. We can use this for multiplying polynomials. Suppose we have two degree $d$ polynomials, each of which has integer coefficients of size at most $B$. 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 $p > 4(B-1)^2(d+1)$. 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 $m$ 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 $m/b$ "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 $m/b + 1$ terms, each of which has size at most $(2^b - 1)^2 < 2^{2b}$ This means that we're safe if we take $p$ at least

$$(\frac{m}{b} + 1)2^{2b}$$

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

Rather than optimizing this perfectly, let's set $b = \log_2 m$; this is simpler and will give us the right asymptotic growth rate. We thus get that $p$ has around $3 \log_2 m$ bits. We then set $n = 2\frac{m}{b} + 1$, so that our finite Fourier transform involves $O(n \log n) = O(m)$ operations, each of which may be an operation on a $(3 \log_2 m)$-bit number. If we use longhand multiplication and division (taking $O(b^2)$ time) to do these operations, we get an $O(m \log^2 m)$-time algorithm.

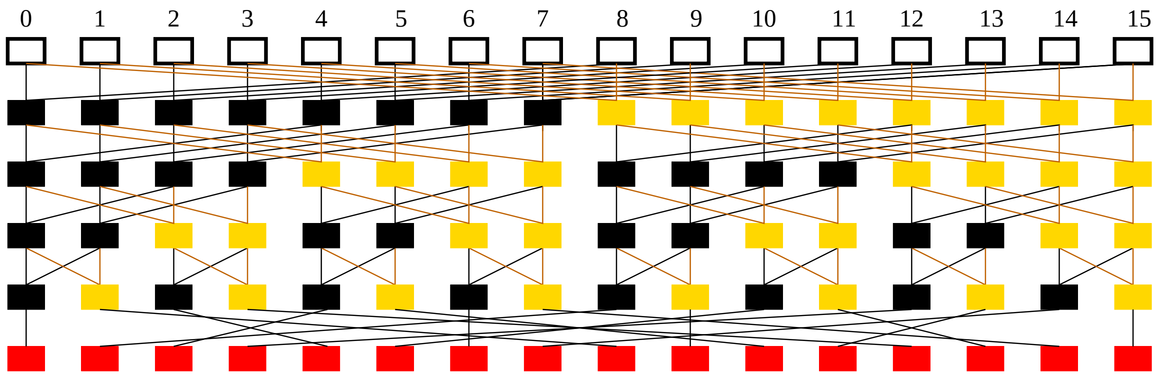
There's no reason that we need to stop there. We could always use recursion and perform these operations on the $3b$-bit numbers using fast integer multiplication as well.

If we use two levels of recursion, we get an $O(\log n(\log\log n)^2)$ time algorithm. If we use three levels of recursion, we get an $O(\log n(\log\log n)(\log\log\log n)^2$ time algorithm, and so forth.

It turns out, although I won't go into the details, that if you work hard enough you can get a $O(n \log n \log\log n)$ time algorithm. The details can be found in Aho, Hopcroft and Ullman's book "Design and Analysis of Computer Algorithms." This algorithm uses the Chinese remainder theorem to get a recursion that produces a running time just a little bit more efficient than what you get using recursion with the method above.

## 7 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 $a_i + a_{i+s}$ where $a_i$ and $a_{i+s}$ are the entries in the two boxes in the row connected to this black box by black lines. In each yellow box, we put $(a_i - a_{i+s})z^i$, where now $a_i$ and $a_{i+s}$ 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 $s$) 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 $z^{2i}$ here and not $z^i$.) 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 $i = 0$.

One tricky thing about this is that we have to start counting $i$ 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 $i$ 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. If you're constructing the spreadsheet according to the instructions in Prof. Kleitman's lecture notes, you don't actually need row 6, because he folds this shuffle into rows 2 through 5. However, you haven't really gotten rid of any complexity, because he has made constructing rows 2 through 5 are a little more complicated.

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 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. But Prof. Kleitman has also figured out how to construct a spreadsheet cleverly so that we don't have to use this tool. 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 $b_i$ using the equations

$$\begin{aligned} b_i &= a_i + a_{i+s} \\ \tilde{b}_i &= (a_i - a_{i+s}) z^i \end{aligned}$$

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

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

to calculate N6 in row 6, where $s = 4$ and we multiply by $(z^2)^i$. 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 $i$ from? In row 3, we have two stretches of four yellow cells, and we want $i$ to start counting from 0 each time. If we just reference $i$ 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($i$, 4) in the formula.

The second problem is somewhat more serious. This is that using the spreadsheet's arithmetic to compute $z^{2i}$ is very likely to give you an integer overflow, if $z$ or $i$ 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's what I think might be the easiest solution: create *another* ($n \times \log_2 n$) array in your spreadsheet that contains the numbers $z^i$ that you need to multiply by $a_i - a_{i+s}$ 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 $z^i$ term needed for that cell.

How can we constrct 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 | 50 | 1 | 9 | 81 | 50 | 1 | 9 | 81 | 50 | 1 | 9 | 81 | 50 |
| 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. I didn't find it too painful to construct this shuffle by hand. I also don't see any easy way of making the spreadsheet do it for you, but I wouldn't be too surprised if someone thinks of a clever way.

You also need to remember to divide by $n$. You do this by multiplying by $n^{-1}$ (mod p). Calculating $n^{-1}$ can be done using the Euclidean algorithm, which you should have gone over 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. I find it useful to color the cells using the $b$ formula one color and those using the $\tilde{b}$ formula a different color, so that I could visually see if I stuck these formulas in the wrong columns by mistake.

I've 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. Here, I don't see any way to do it without using use the OFFSET function (which generally makes things more complicated), and you also need some way to figure out which row you're 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.