Fourier Transforms and Theoretical Computer Science

February 5, 1999

Lecturer: Umesh Vazirani

Scribe: Steve Chien

Today’s topics: The Discrete Fourier Transform=20

- DFT over \( \mathbb{Z}_2^n \) and the mixing time of random walks.
- DFT over \( \mathbb{Z}_n \) and applications to polynomial/integer multiplication.

1 DFT over \( \mathbb{Z}_2^n \)

We consider the group \( \mathbb{Z}_2^n \) and functions \( f : \mathbb{Z}_2^n \rightarrow \mathbb{C} \). These functions form a vector space of dimension \( 2^n \), and a natural basis to take would be that comprised of the indicator functions \( \{ f_x : f_x(y) = 1 \Leftrightarrow x = 3Dy \} \). Another basis is the parity basis, which we can define as the set \( \{ \chi_x : \chi_x(y) = 3D(\phi_1)^{x \cdot y} \} \), where \( x \cdot y \) is the inner product of \( x \) and \( y \). The effect of applying \( \chi_x \) is to determine the parity of those bits of \( y \) selected by the 1 bits in \( x \).

If we define \( \chi_x \cdot \chi_y \) as \( \sum_z \chi_x(z) \chi_y(z) \), then we see that the \( \chi_x \) are orthogonal. It is obvious that \( \forall x \chi_x \cdot \chi_x = 3D \sum_y 20(\phi_1)^{x \cdot y} = 3D \sum_y (\phi_1)^{x \cdot y} = 3D(\phi_1)^{x \cdot z} \). Since \( x + z \) must be nonzero in at least one of its bits, we see that as \( y \) varies, we will achieve exact cancellation and \( = \end{equation} \) end up with 0. Thus, by a counting argument, the \( 2^n \) parity functions form a basis.

Furthermore, since these functions \( \{ \chi_x \} \) are homomorphisms, they are in fact the Fourier basis. (Another way to say this is that they are the characters of \( \mathbb{Z}_2^n \), a term that comes from group representation theory.) The Fourier transform then takes us from the basis of the indicator functions to the parity basis:

\[
\hat{f}(y) = 3D \sum_z (\phi_1)^{x \cdot z} f(z).
\]

There are several nice properties that come from these definitions.

Convolution

The first is that convolution in the indicator basis is equivalent to \( = \) multiplication in the Fourier basis. Recall that the convolution of two functions \( f \) and \( g \) is defined as \( f \ast g = g(x \Leftrightarrow y) = 3D \sum_y f(y)g(x \Leftrightarrow y) \) (over an arbitrary group, we replace the \( g(x \Leftrightarrow y) \) with \( g(y^{-1}x) \)). We can quickly verify that \( f \ast g = 3D \hat{f}(x) \hat{g}(x) \):

\[
f \ast g(x) = 3D \sum_z \sum_y f(y)g(z \Leftrightarrow y)(\phi_1)^{x \cdot z} = 3D \sum_y f(y)(\phi_1)^{x \cdot y} \sum_z g(z \Leftrightarrow y)(\phi_1)^{x \cdot (z-y)} = 3D \hat{f}(x) \hat{g}(x)
\]

Plancherel’s Theorem

\[
\sum_x f(x)g(x) = 3D \frac{1}{2^n} \sum_x \hat{f}(x) \hat{g}(x)
\]
Subspaces of $\mathbb{Z}_2^n$

Consider $\mathbb{Z}_2^n$ as a vector space, and a subspace $X \subseteq \mathbb{Z}_2^n$. Define $X^\perp$ (pronounced "x-perp") as $\{y \in \mathbb{Z}_2^n : x \cdot y = 3D0 \forall x \in X\}$. Consider now $\xi$ the indicator function $\xi(x) = 20 f(x) = 3D \begin{cases} 1 & \text{if } x \in X \\ 0 & \text{otherwise} \end{cases}$ We claim that the Fourier transform of $\xi$ behaves well $\iff 20 \hat{\xi}(y) = 3D \begin{cases} 1 & \text{if } y \in X^\perp \\ 0 & \text{otherwise} \end{cases}$.

Proof. Note that $\hat{\xi}(y) = 3D \sum_{x \in X} e^{-ix \cdot y}$. If $y \in X^\perp$, then $\forall x \cdot y = 3D0$ and so $\hat{\xi}(y) = 3D|X|$. If $y \not\in X^\perp$ then $\exists z \in X$ such that $z \cdot y = 3D1$. We can then divide $X$ into two equal sets, $= X' = 3D\{x : x \cdot y = 3D0\}$ and $X' + Z$. From this we can conclude that $\hat{\xi}(y) = 3D0$ as needed.

Mixing Time of Random Walks over $\mathbb{Z}_2^n$

[Insert reference.]

Consider a random walk over $\mathbb{Z}_2^n$ in which a particle begins at the origin and at each time step will either remain at its current position or move to one of its $d$ neighbors, all with equal probability $\frac{1}{d+1}$. We wish to know how quickly the particle’s position as a probability distribution approaches uniform.

More formally, the probability distribution of the particle’s position after one time step can be described as:

$$Q(x) = 3D \begin{cases} \frac{1}{d+1} & \text{if } x = 3D0 \text{ or } x = 3D\epsilon_i \\ 0 & \text{otherwise} \end{cases}$$

Phrasing it this way allows to see the happy fact that after two steps, the probability distribution is simply the convolution of $Q(x)$ with itself, which by definition is $Q \ast Q(x) = 3D \sum_{y} Q(y)Q(x \oplus y)$. One way to read this is that the probability of the particle being at $x$ after two steps is exactly the sum over all $y$ of the probability of both being at $y$ after one step, $Q(y)$ and moving from $y$ to $x$ in the second step, $Q(x \oplus y)$. From this we conclude that after $n$ steps, the probability distribution is the convolution power of $Q$, denoted by $Q^n$.

Using the above fact about convolutions and the Fourier transform, we can see that $Q^n = 3DQ^n$. We wish to determine how close to uniform this is. Notice that if $U(x) = 3D \frac{1}{d+1}$ is the uniform distribution, then the Fourier transform of it is $20 U(x) = 3D \begin{cases} 1 & \text{if } x = 3D0 \\ 0 & \text{otherwise} \end{cases}$.

We now prove the following theorem:

**Theorem.** $\|Q \ast k \Rightarrow U\|_1 \leq \sqrt{2(\epsilon^{-k} \Rightarrow 1)} = \sqrt{3D \frac{(d+1)^{\frac{\ln d}{4}} + cd}}$.

In fact this is a very tight bound; a lower bound of $\frac{d\ln d}{4} + = cd$ can be proven.

To prove this theorem we first need the following lemma:

**Lemma.** For any function $P(x)$ such that $\hat{P}(0) = 3D1$ then $\|P \ast k \Rightarrow U\|_1 \leq \sum_{x \neq y} |\hat{P}(x)|^2$.

Proof. The proof relies on the Cauchy-Schwarz inequality, whose results are often disastrous but fortunately not so in this case, as well as Plancherel’s result.

$$\|P \ast k \Rightarrow U\|_1^2 = 3D \sum_y (|P \ast k(y) \Rightarrow U(y)|^2) =$$
\[
\leq 2d \sum_y |P^{*k}(y)|^2 \leq U(y)|^2 \leq \sum_{y \not\equiv 0} (P^{*k}(y))^2 \\
\leq \sum_{y \not\equiv 0} [\hat{P}(y)]^{2k}
\]

Now, since \( \hat{U}(0) = 3D\hat{P}(0) = 3D1 \) but \( \hat{U}(x) = 3D0 \) otherwise, using Plancherel’s theorem we can write

\[
\|P^{*k} \leftrightarrow U\|_1^2 \leq \sum_{y \not\equiv 0} (P^{*k}(y))^2 \\
\leq \sum_{y \not\equiv 0} [\hat{P}(y)]^{2k}
\]

which is exactly what we need.

**Proof of Theorem.** What we need to do now is to calculate the actual Fourier coefficients \( \tilde{=} \) and apply our lemma. Fortunately, our task is made much easier by the structure of \( Q(x) \), namely the \( \tilde{=} \) properties that \( Q(x) \) is nonzero for only \( d+1 \) values of \( x \), and that these values of \( x \) have simple structure.

\[
\hat{Q}(x) = 3D \sum_y (\varepsilon_1) \varepsilon_y Q(y) \\
= 3D \frac{1}{d+1} [1 + \#0’s \text{ in } x \leftrightarrow \#1’s \text{ in } x] \\
= 3D 1 \varepsilon \frac{2|x|}{d+1} \text{ where } |x| = 3D \#1’s \text{ in } x
\]

The second equality comes about since \( Q(y) \) is nonzero only for \( y = 3D0 \) or \( y = 3De \). When \( y = 3D0 \), \( \varepsilon \varepsilon_y \) is 1, and when \( y = 3De \), then \( (\varepsilon_1)^{x=y} \) is 1 when the \( i \)th bit of \( x \) is 0, and \( \varepsilon \) otherwise.

Applying the lemma, we obtain

\[
\|Q^{*k} \leftrightarrow U\|_1^2 = 3D \sum_{x \not\equiv 0} (1 \varepsilon \varepsilon_1)^{2|y|} = 3D \sum_{j=3D1}^d \left( \frac{d}{j} \right) = 20(1 \varepsilon \varepsilon_1)^{2d} \leq \sum_{j=3D1}^d \frac{d^j}{j!} = e^{ \frac{2d}{e} } 
\]

Considering only the \( \epsilon \) term and substituting in for \( k \), we see that

\[
e^{ \frac{2d}{e} } = 3De^{-d \ln d} = 3D e^{-d \ln d} \leq d^{-d} e^{-c_d}
\]

Substituting this back in, we find that

\[
\|Q^{*k} \leftrightarrow U\|_1^2 \leq \sum_{j=3D1}^d \frac{d^j}{j!} = d^{-d} e^{-c_d}
\]
This gives us the following:

\[ \sum_{j=3D1}^{d} e^{-\epsilon j} j! \]
\[ \sum_{j=3D1}^{\infty} e^{-\epsilon j} j! \]
\[ e^{\epsilon^{-\epsilon}} \Leftrightarrow 1 \]

This finishes the proof.

2 DFT over \( \mathbb{Z}_n \)

We now consider the Discrete Fourier Transform over the group \( \mathbb{Z}_n \). Following the same principle as before, we consider the functions \( f : \mathbb{Z}_n \rightarrow \mathbb{C} \). This time the \( \mathbb{C} \) homomorphisms, and thus the proper Fourier basis functions, are \( \{ \chi^k(j) = 3D\omega^{jk} \} \), where \( \omega = 3D e^{2\pi i/n} \) is a primitive \( n \)th root of unity. (Once again, these are the characters of \( \mathbb{Z}_n \).) Consequently, we can once again define the Fourier transform of \( f \) as \( \hat{f}(x) = 3D \sum_y \omega^{xy} f(y) \).

Notice that the Fourier transform in this case has an elegant matrix representation:

\[
\begin{bmatrix}
\hat{f}(1) \\
\hat{f}(\omega) \\
\hat{f}(\omega^2) \\
\vdots \\
\hat{f}(\omega^n)
\end{bmatrix} = 3D \\
\begin{bmatrix}
1 & 1 & 1 & \cdots & 1 \\
201 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
1 & \omega & \omega^2 & \cdots & \omega^{(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2}
\end{bmatrix}
\begin{bmatrix}
\hat{f}(1) \\
\hat{f}(\omega) \\
\hat{f}(\omega^2) \\
\vdots \\
\hat{f}(\omega^n)
\end{bmatrix}
\]

In other words, the transformation matrix \( F \) can be expressed as \( \left[ [ \omega^{jk} ] \right] \). Also important is that \( F \) is invertible and that \( F^{-1} \) has a nice form, namely that \( 20 \hat{F}^{-1} = 3D \frac{1}{2} \hat{F} = 3D \frac{1}{2} \left[ [ \omega^{-jk} ] \right] \). We also have the usual property concerning convolution: \( f * g(x) = 3D \hat{f}(x) \hat{g}(x) \). (These last two properties are fairly easy exercises.)

FFT

Calculating the \( \hat{f} \) vector using straightforward matrix multiplication would require time \( O(n^2) \), but the well-known FFT algorithm requires only time \( O(n \log n) \). FFT relies on the fact that a clever rearrangement of the columns of the \( F \) matrix significantly reduces the number of needed operations. More precisely, we move the columns representing the even powers of \( \omega \) to the left, = and those representing the odd powers to the right, and rearrange the \( \hat{f} \) vector to place the even values on top = and the odd values on bottom. This gives \( 20 \) us the following:

\[
\begin{bmatrix}
\hat{f}(1) \\
\hat{f}(\omega) \\
\hat{f}(\omega^2) \\
\vdots \\
\hat{f}(\omega^n)
\end{bmatrix} = 3D = 20
\begin{bmatrix}
\{ \omega^{jk} \} = 3DA & \{ \omega^j \omega^{2jk} \} = 3DB = \\
20 \{ \omega^{2(j+n/2)k} \} = 3DA & \{ \omega^{j+n/2} \omega^{2(j+n/2)k} \} = 3D \Leftrightarrow B
\end{bmatrix}
\]
The operations now required include the recursive calls of computing = each of \( Af_0 \) and \( Bf_1 \) and then=20 performing the additions. The total running time now becomes \( T(n) = 3D = 2T(n/2) + O(n) \) =\( 3DO(n \log n) \).

**Application: Polynomial multiplication**

The FFT algorithm leads to an immediate application in polynomial = multiplication. Given two polynomials of degree \( n \gg 1 \), \( P(x) = 3Da_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \) and \( Q(x) = 3Db_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1} \), we notice that the coefficients = of their product \( P(x)Q(x) \) is the convolution of their original coefficients. Thus to compute the = product, all we need do is take the FFT of the original coefficients, multiply them, and take the inverse FFT, all = of which can be done in time \( O(n \log n) \).

More explicitly, if we write \( P(x)Q(x) = 3Dc_0 + c_1 x + \cdots + c_{2n-2} x^{2n-2} \), then our algorithm will first compute FFT(\( a_0, \ldots, a_{n-1}, 0, \ldots, 0 \)) and \( = \) FFT(\( b_0, \ldots, b_{n-1}, 0, \ldots, 0 \)) to obtain \( (\hat{a}_0, \ldots, \hat{a}_{2n-2}) \) and \( = (\hat{b}_0, \ldots, \hat{b}_{2n-2}) \). We then multiply these componentwise to obtain \( (\hat{c}_0, \ldots, \hat{c}_{2n-2}) \) and then = do the inverse FFT to obtain the \( \{c_i\} \).

**Application: Integer multiplication**

A more interesting application of FFT is the integer multiplication = algorithm of Schönhage and Strassen, which allows us to multiply two \( n \) =bit integers in time \( O(n \log n \log \log n) \) rather than the naive \( O(n^2) \). The=20 idea is to treat \( n \) =bit integers as polynomials and use the \( O(n \log n = n) \) polynomial multiplication algorithm. If we are still allowed to assume that primitive operations still take = \( O(1) \) time, then we might imagine that given two \( n \) =bit integers \( A \) and \( B \) we might divide each of them into the = concatenation of \( n/\log n \) pieces. For example, we would write \( A = 3D_{n-1} a_{n-1} \cdots a_0 \). We can = then write the polynomial 

\[
A(x) = 3D_{n-1} a_{n-1} x^{n-1} + \cdots + a_1 x + a_0
\]

and realize that \( A(n) = 3DA \). After = following a similar procedure for \( B \), we would then apply the FFT polynomial multiplication algorithm to recover the = \( c_i \). We can then evaluate the polynomial at \( n \) in time \( O(n \log n) \), as the \( c_i x^i \) terms have small overlap.

The problem, of course, is that the primitive operations in the FFT take = longer than \( O(1) \). Furthermore, this approach requires precise arithmetic computations. The solution to = these problems is to work in the ring \( \mathbb{Z}_m \) and to divide the \( n \) =bit numbers in a different manner.

Fortunately for our purposes, it is not difficult to choose appropriate = values for \( m \) and \( n \) so that performing FFT is possible. If we let \( n \) and \( \omega \) be some powers of \( 2 \) and = let \( m = 3D \omega^{n/2} + 1 \), then we have the following properties, which turn out to be sufficient for FFT: \( \varepsilon=20 \)

- \( \omega^n = 3D1 \)
- \( \omega^0, \omega^1, \ldots, \omega^{n-1} \) are distinct
- \( \forall 1 \leq k \leq n \sum_{j=0}^{n-1} \omega^{jk} = 3D0 \)
- \( n^{-1} \) exists.

Notice that the time to perform FFT has increased, however. Since our = addition operations are potentially=20 over \( n \) =bit numbers, we have an extra factor of \( n \), so that the cost = of FFT is now \( O(n^2 \log n) \).

=20 (Note that as the only multiplications in FFT involve powers of = \( \omega \), multiplications consist only of shifts and additions.)
The algorithm is now straightforward. We now divide each of our \( n \)-numbers into \( \sqrt{n} \) pieces each of size \( \sqrt{n} \). As before, we do an FFT on these \( \sqrt{n} \) values, perform a componentwise multiplication, and then do the inverse FFT. Each FFT now takes time \( O(n \log \sqrt{n}) = 3D = O(n \log n) \). If \( T(n) \) is the time required to multiply two \( n \)-bit integers, we can write the recurrence relation \( T(n) = 3D \sqrt{n}T(\sqrt{n}) + O(n \log n) \), which works out to \( O(n \log n \log \log n) \).