1 Preliminaries

The Fast Fourier Transform (FFT) is a widely used algorithm in areas like signal processing, digital audio, x-ray tomography, physics simulations, and multiplying large numbers. As we'll see, it's also a useful took to have in your algorithms tool box.

The algorithm is derived, and most easily understood in the context of the problem of multiplying polynomials. So let $A(x)$ and $B(x)$ be two polynomials of degree d

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d \\ B(x) &= b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d \end{aligned}$$

Our goal is to compute the polynomial $C(x)$:

$$C(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2d} x^{2d}$$

Where

$$c_k = \sum_{\substack{0 \le i,j \le k \\ i+j=k}} a_i * b_j = \sum_{0 \le i \le k} a_i * b_{k-i}$$

If we think of A and B as vectors of numbers, then the C-vector is called the *convolution* of A and B. Note that if the sum refers to an element that does not exist (such as a_{2d}) a zero is used for that value.

The FFT is the basis of a fast algorithm for computing this convolution. Given vectors $A = [a_0, \ldots, a_d]$ and $B = [b_0, \ldots, b_d]$ we want to compute $C = [c_0, \ldots, c_{2d}]$ where c_i is defined as above.

This can be done naively in $O(n^2)$, or via Karatsuba's algorithm in $O(n^{1.58})$. We'll use the FFT to do it in $O(n \log n)$. This is then used in the Schonhage-Strassen integer multiplication algorithm that multiplies two n-bit integers in $O(n \log n \log\log n)$ time. (Which is not covered in this lecture.)

2 The High Level Idea of the Algorithm

In the introduction we represented the polynomials in the coefficient representation. But there's an alternative representative for a polynomial. If we know the value of a degree d polynomial at $d+1$ distinct points, that uniquely determines the polynomial. Not only that, but if we had A and B in this point-value representation, we could, in $O(d)$ time compute the polynomial C in that same representation: simply mulitply the values of A and B at the specified points together.

This leads to the following outline of an algorithm for this problem:

Let $N = 2d + 1$ so the degree of C is less than N.
(1) Pick N points $x_0, \ldots, x_{N-1}$ according to a secret formula.
(2) Evaluate A at each of the points: $A(x_0), \ldots, A(x_{N-1})$.
(3) Same for B.
(4) Now compute $C(x_0), \ldots, C(x_{N-1})$ where $C(x) = A(x) * B(x)$.
(5) Interpolate to get the coefficients of C.

This whole thing seems patently crazy since it looks like steps 2 and 3 should take $\Theta(N^2)$ time just in themselves. However, the FFT will allow us to quickly move from the "coefficient representation" of polynomial to the "value on N points" representation, and back, for our special set of N points. (It doesn't work for N **arbitrarily** chosen points.)

The reason we like this is that multiplying is easy in the "value on N points" representation. So step 4 is only $O(N)$ time.

Let's focus on forward direction first. In that case, we've reduced our problem to the following:

> **GOAL:** Given a polynomial A of degree $< N$, evaluate A at N points of our choosing in total time $O(N \log N)$. Assume N is a power of 2.

First here's a little intuition for how this is going to work. Consider the case where the degree of A is 7, and we want to evaluate it at two points: 1 and -1.

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7.$$

So:

$$\begin{aligned} A(1) &= a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 \\ A(-1) &= a_0 - a_1 + a_2 - a_3 + a_4 - a_5 + a_6 - a_7 \end{aligned}$$

These two computations take a total of 14 additions.

Suppose instead we were to compute $Z = a_0 + a_2 + a_4 + a_6$ and $W = a_1 + a_3 + a_5 + a_7$. Now $A(1) = Z + W$ and $A(-1) = Z - W$. This can be done in a total of only 8 additions. This optimization may not seem like much, but if you can figure out how to apply it recursively, it solves the problem – it gets us from $O(N^2)$ down to $O(N \log N)$.

3 Evaluation at the Roots of Unity

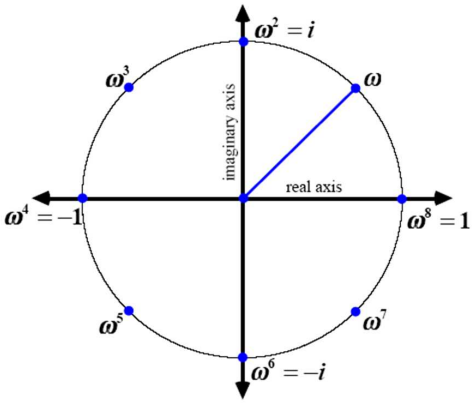
The way to get this to work recursively is to find a number that is a *primitive Nth root of unity*. This number, call it ω_N, has the following properties:

$$\begin{aligned} \omega_N^N &= 1 \\ \omega_N^j &\neq 1 \qquad \text{for } 0 < j < N \end{aligned}$$

If our coefficients are real numbers, we will not find any primitive roots of unity (other than for $N = 2$) among the reals. So we have to find a larger field that contains the reals, and also has primitive Nth roots of unity. Luckly such a field exists – the complex numbers.[1]

So let $\omega_N = \exp(2\pi i/N)$. So $1, \omega_N, \omega_N^2, \ldots, \omega_N^{N-1}$ are N are complex numbers, starting at 1, and evenly spaced around the unit circle. The following figure shows the eighth roots of unity.

[1] If we are operating over the integers, then there are fields modulo certain primes that have appropriate primitive roots of unity. They are not hard to find, but they have to be choosen sufficiently large so that the answer coefficients of the product polynomial do not exceed the modulus.



These are the special magic numbers at which we will evaluate our polynomial. (In the following derivation if we omit N and write ω, then we mean ω_N.)

Recall that A is a polynomial of degree $N-1$:

$$A(x) = a_0 + a_1 x + a_2 x_2 + \cdots + a_{N-1} x^{N-1}$$

Define the DFT (Discrete Fourier Transform) of the vector $(a_0, \ldots, a_{N-1})$ to be another vector of N numbers as follows:

$$F_N(a_0, a_1, \ldots, a_{N-1}) = (A(\omega^0), A(\omega^1), \ldots, A(\omega^{N-1}))$$

(As we will see later, this can be expressed as a matrix multiplication.)

Now we can derive the fast algorithm for computing the DFT. (This is called the FFT algorithm.) Let A be the vector $(a_0, a_1, \ldots, a_{N-1})$. Let $F_N(A)_j$ denote the jth component of the DFT of the vector A.

$$\begin{aligned}
F_N(A)_j &= A(\omega^j) \\
&= \sum_{i=0}^{N-1} a_i \omega^{ij} \\
&= \sum_{\substack{i\,=\,0 \\ i \text{ even}}}^{N-1} a_i \omega^{ij} + \sum_{\substack{i\,=\,1 \\ i \text{ odd}}}^{N-1} a_i \omega^{ij} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_{2i} \omega^{2ij} + \sum_{i=0}^{\frac{N}{2}-1} a_{2i+1} \omega^{(2i+1)j} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_{2i} (\omega^2)^{ij} + \omega^j \sum_{i=0}^{\frac{N}{2}-1} a_{2i+1} (\omega^2)^{ij} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_{2i} (\omega_{N/2})^{i(j \bmod N/2)} + \omega_N^j \sum_{i=0}^{\frac{N}{2}-1} a_{2i+1} (\omega_{N/2})^{i(j \bmod N/2)}
\end{aligned}$$

In the last step we've simply used the fact that $\omega_N^2 = \omega_{N/2}$, and the observation that $\omega_{N/2}^{ij} = (\omega_{N/2}^j)^i = (\omega_{N/2}^{j \bmod N/2})^i$ because $\omega_{N/2}^j$ is a periodic funciton of j with period $N/2$.

Now the key point is that these two summations are in fact just DFTs half the size. Let's let A_{even} denote the vector of as with even subsubscripts (of length $N/2$) and A_{odd} be the odd ones. We can write the final equation above using these vectors as follows:

$$F_N(A)_j = F_{N/2}(A_{\text{even}})_{j \bmod N/2} + \omega_N^j F_{N/2}(A_{\text{odd}})_{j \bmod N/2}$$

Notice that in this derivation we did use the fact that ω is a root of unity, but we *never used* the fact that it is a primitive root of unity. That will be needed when we compute the inverse.

So we can rewrite the above recurrence as pseudocode as follows:

FFT$([a_0, \ldots, a_{N-1}], \omega, N)$
  if $N = 1$ **then return** $[a_0]$
  $F_{\text{even}} \leftarrow$ **FFT**$([a_0, a_2, \ldots, a_{N-2}], \omega^2, N/2)$
  $F_{\text{odd}} \leftarrow$ **FFT**$([a_1, a_3, \ldots, a_{N-1}], \omega^2, N/2)$
  $F \leftarrow$ a new vector of length N
  $x \leftarrow 1$
  for $j = 0$ to $N - 1$ **do**
    $F[j] \leftarrow F_{\text{even}}[j \mod (N/2)] + x * F_{\text{odd}}[j \bmod (N/2)]$
    $x \leftarrow x * w$
  return F

The algorithm clearly follows this recurrence:

$$T(N) = 2T(N/2) + O(N),$$

which solves to $O(N \log N)$.

4 The Inverse of the DFT

Remember, we started all this by saying that we were going to multiply two polynomials A and B by evaluating each at a special set of N points (which we can now do in time $O(N \log N)$), then multiply the values point-wise to get C evalauated at all these points (in $O(N)$ time) but then we need to interpolate back to get the coefficients. In other words, we're doing $F_N^{-1}(F_N(A) \cdot F_N(B))$.

So, we need to compute F_N^{-1}. We'll develop this now.

First, we can view the forward computation of the DFT (evaluating the polynomial $A()$ at $1, \omega, \omega^2, \ldots, \omega^{N-1}$) as a matrix-vector product:

$$\begin{pmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \cdots & \omega^{0\cdot(N-1)} \\ \omega^{1\cdot 0} & \omega^{1\cdot 1} & \cdots & \omega^{1\cdot(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^{(N-1)\cdot 0} & \omega^{(N-1)\cdot 1} & \cdots & \omega^{(N-1)\cdot(N-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} A(\omega^0) \\ A(\omega^1) \\ \vdots \\ A(\omega^{N-1}) \end{pmatrix}$$

Note that the matrix on the left is the one where the contents of the ith row and the jth column is ω^{ij}. Let's call this matrix $\mathbf{DFT}(\omega, N)$. What we need is $\mathbf{DFT}(\omega, N)^{-1}$.

Let's see what happens if we multiply $\mathbf{DFT}(\omega^{-1}, N)$ by $\mathbf{DFT}(\omega, N)$.

$$\text{The } (i,j) \text{ entry of } \mathbf{DFT}(\omega^{-1}, N)\ \mathbf{DFT}(\omega, N) = \sum_{k=0}^{N-1} \omega^{-ik}\omega^{kj}$$

So let's try to evaluate the summation on the right in the two cases of $i = j$ and $i \neq j$.

If $i = j$ then we get:

$$\sum_{k=0}^{N-1} \omega^{-ik}\omega^{kj} \;=\; \sum_{k=0}^{N-1} \omega^{0} \;=\; N$$

If $i \neq j$ then we get:

$$\sum_{k=0}^{N-1} \omega^{-ik}\omega^{kj} \;=\; \sum_{k=0}^{N-1} \omega^{(j-i)\cdot k} \;=\; \sum_{k=0}^{N-1} (\omega^{j-i})^{k}$$

Now let's make use of the fact that ω is a primitive root of unity. This means that $\omega^{j-i} \neq 1$. The sum is now a geometric series. So we can use the standard formula for summing a geometric series:

$$1 + r + r^2 + \cdots + r^{N-1} = \frac{1 - r^N}{1 - r}$$

So

$$\sum_{k=0}^{N-1} (\omega^{j-i})^{k} = \frac{1 - (\omega^{j-i})^{N}}{1 - \omega^{j-i}} = \frac{1 - 1}{1 - \omega^{j-i}} = 0$$

So, summarizing what we just learned:

$$\text{The } (i,j) \text{ entry of } \mathbf{DFT}(\omega^{-1}, N)\ \mathbf{DFT}(\omega, N) = \begin{cases} N & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

This is just the identity matrix times N. So what we've just proven is that:

$$\mathbf{DFT}(\omega, N)^{-1} = \frac{1}{N}\mathbf{DFT}(\omega^{-1}, N)$$

This means that we can compute the inverse of the DFT by using the FFT algorithm described above except running it with ω^{-1} instead of ω, and then multiplying the result by $1/N$. It still runs in $O(N \log N)$ time. Putting this all together we have an algorithm to multiply polynomials, or compute the convolution in $O(N \log N)$ time.

5 Applications of the FFT

Here are some simplified examples of what can be done with the FFT.

5.1 Digital Filtering

Suppose you wanted to simulate what it would sound like to be in a dungeon (or concert hall, or a forest, or anywhere else). You want to take a recorded sound and transform it to what it sounds like in some other environment. Here's one way to go about doing that.

You could go to the dungeon and measure what is called an *impulse response* of the environment. This is done by generating a loud very short bang (the impulse) and recording for the next several seconds what comes back (the response) as a result of that impulse.

The input to this problem is a signal of length n, which is a sequence of numbers, $S = s_0, s_1, \ldots, s_{n-1}$. Let the response also be a sequence of n numbers (pad it out if it's too short), and call it $R = r_0, r_1, \ldots, r_{n-1}$. Now the convolution of the two sequences is

$$c_k = \sum_{\substack{0 \le i,j \le k \\ i+j=k}} s_i * r_j$$

(A picture is needed here.)

So to apply our algorithm to compute this, we would let N be the next power of two greater than $2n-2$. Then we would pad both of the vectors S and R with zeros, then we would compute the convolution using the FFT technique.

Making a practial digital filter for applications is much more complicated. But at the core of these algorithms is an FFT-based convolution algorithm. This is needed to make it run fast enough.

5.2 Three Evenly Spaced 1s within a Binary String

Given a binary string S of length n, let $S[i]$ denote the ith bit, for $0 \le i < n$. And let $L = \{i \mid S[i] = 1\}$, be the indices with ones. For this input we have to decide whether there are three evenly spaced 1s within S (we don't need to find the locations of the 1s should they exist). For example, $11100000, 110110010$ both have three evenly spaced 1s, while 1011 does not.

1. What is a naive $O(n^2)$ solution to this problem? Just try all starting indices i and all gap lengths d, and for each (i, d) pair, check in $O(1)$ if $S[i] = S[i+d] = S[i+2d] = 1$. There are at most n choices for i and for d, so total runtime is $O(n^2)$.

2. Here's the $O(n \log n)$ solution using convolutions. Construct a polynomial $p(x) = \sum_{i=0}^{n-1} S[i]x^i$. So for the second example above this is:

   $$p(x) = x^0 + x^1 + x^3 + x^4 + x^7.$$

   This step is $O(n)$.

   Compute $p(x)^2$ using FFT multiplication. For the example above:

   $$p(x)^2 = x^0 + 2x^1 + x^2 + 2x^3 + 4x^4 + 2x^5 + x^6 + 4x^7 + 3x^8 + 2x^{10} + 2x^{11} + x^{14}.$$

The coefficient of x^t is the number of ordered pairs (a, c) such that $a, c \in L$ and $a + c = t$. This step is $O(n \log n)$.

Now we look at the terms x^{2b} for $b \in L$. Note that the pair (b, b) always contributes 1 to the coefficient of x^{2b}. If there exists some other pair (a, c) where $a \neq c$, $a + c = 2b$, and $a, c \in L$, then the coefficient of x^{2b} will be at least 3. Thus if the coefficient of x^{2b} is 3 or more for any $b \in L$ then output YES. Otherwise, output NO. In the example above, $4 \in L$ we see that the coefficient of x^8 is 3 implying that there exists three evenly spaced 1s, centered at 4.

5.3 String Matching (Optional Material)

1. Suppose we are given a text $t = t_0 t_1 \dots t_{n-1}$ and a pattern $p = p_0 p_1 \dots p_{m-1}$. Find all occurrences of p in t in $O(n \log n)$ time using FFT.

   Let $X[i] = \sum_{j=0}^{m-1} (t_{i+j} - p_j)^2$ for all $0 \leq i < n - m$. There is a match at index i if and only if $X[i] = 0$. Now $X[i] = \sum_{j=0}^{m-1} (t_{i+j} - p_j)^2 = \sum_{j=0}^{m-1} (p_j^2 - 2p_j t_{i+j} + t_{i+j}^2)$.

   So define $Y[i] = \sum_{j=0}^{m-1} p_j t_{i+j}$ for all $0 \leq i < n - m$. It remains to calculate Y.

   Let t' be the reverse of t. Then

   $$Y[i] = \sum_{j=0}^{m-1} p_j t_{i+j} = \sum_{j=0}^{m-1} p_j t'_{n-1-i+j}$$

   Let $k = n - 1 - i - j$. Then

   $$Y[i] = \sum_{j+k=n-1-i} p_j t'_k$$

   This can now be calculated using FFT. Let f be the polynomial with coefficients $p_1, p_2, \dots, p_{m-1}$ and g be the polynomial with coefficients $t'_0, t'_1, \dots, t'_{n-1}$. Then $Y[i]$ is the coefficient of x^{n-1-i} in $f \cdot g$.

2. Now suppose p contains wildcard characters, which can match any character t. Finds all matches of p in t in $O(n \log n)$ time.

   Replace each wildcard with 0. Let

   $$X[i] = \sum_{j=0}^{m-1} t_{i+j} p_j (t_{i+j} - p_j)^2$$

   Then there is a match at index i if and only if $X[i] = 0$. The rest is the same as in part (a).

   Specifically, suppose we want to calculate $Y[i] = \sum_{j=0}^{m-1} p_j^a t_{i+j}^b$ for some constants a and b. Let f be the polynomial with coefficients $p_1^a, p_2^a, \dots, p_{m-1}^a$ and g be the polynomial with coefficients $t'^b_0, t'^b_1, \dots, t'^b_{n-1}$. Then $Y[i]$ is the coefficient of x^{n-1-i} in $f \cdot g$.

3. Suppose p contains no wildcards. Compute, in $O(n \log n)$ time, for each index $0 \leq i < n - m$, the number of characters that agree between p and $t[i, i + m - 1]$.

   Let's solve an easier version of this problem, where for each i, we want to calculate the number of agreements between p and $t[i, i + m - 1]$ where both letters are 'a'.

   Replace each letter in p and t with 1 if it is an 'a', and 0 otherwise.

Again define $Y[i] = \sum_{j=0}^{m-1} p_j t_{i+j}$ for all $0 \leq i < n - m$. Clearly $Y[i]$ is the value we want for index i. Y can be computed with FFT as in part a.

Now repeat for each of the other 25 letters of the alphabet. Sum together all Y arrays to obtain the final answer.

6 Cyclic Convolutions

Consider these two vectors:

$$\begin{aligned} a &= [1,1,1,1,0,0,0,0] \\ b &= [1,1,1,1,0,0,0,0] \end{aligned}$$

their convolution is:

$$c = [1,2,3,4,3,2,1,0]$$

And this is precisely what you'd expect if you multiplied the corresponding polynomials together. But a natural question to ask is: what would the result be if there were some non-zeros in all those zero positions of a and b? What is the algorithm doing with those? Here's an example of it.

$$\begin{aligned} a &= [1,2,3,4,0,0,0,0] \\ b &= [0,0,0,0,0,1,0,0] \\ c &= [4,0,0,0,0,1,2,3] \end{aligned}$$

You can see that it just wraps the results around modulo N. This is called the *cyclic convolution*. And it's defined as follows:

$$c_k = \sum_{\substack{0 \leq i,j < N \\ i+j \bmod N = k}} a_i * b_j = \sum_{0 \leq i < N} a_i * b_{(k-i) \bmod N}$$

So this is what our FFT-based polynomial multiplication algorithm is really doing with the two input vectors.

Here are some other examples of cyclic convolutions.

```
       A =    1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0
       B =    1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
conv A B =    1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0


       A =    1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0
       B =    0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0
conv A B =    0  0  0  0  0  0  0  1  2  3  4  5  6  7  8  0


       A =    1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0
       B =    0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0
conv A B =    4  5  6  7  8  0  0  0  0  0  0  0  0  1  2  3


       A =    1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0
       B =    0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0
conv A B =    7  8  0  0  0  1  2  3  4  5  7  9 11  4  5  6


       A =    1  2  3  4  5  6  7  8  9 10 11 12  0  0  0  0
       B =    0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0
conv A B =   19  8  9 10 11 13  2  3  4  5  7  9 11 13 15 17


       A =    1  0  1  1  0  0  1  1  1  1  1  0  1  0  0  0
       B =    1  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0
conv A B =    1  1  1  3  1  1  2  2  2  3  3  2  2  2  0  1
```