Fast Fourier Transform

1 Overview

This material, in somewhat different form, is in §30.1-2 of the text.

Our goal is to give a rapid algorithm to multiply two binary numbers $\alpha = \sum_i a_i 2^i$, $\beta = \sum_i b_i 2^i$ and output the product $\gamma = \sum_i d_i 2^i$. Recall that we consider numbers represented by their array $a[i], b[i], d[i]$. Our algorithm will take time $O(n \lg n)$ (our mantra!) where all exponents of 2 (in α, β, γ) are less than n. We shall take n a power of two and write

$$n = 2^t \tag{1}$$

(If, say, the largest exponent is 37 we will take $n = 64$. This incurs a small loss but the algorithm is considerably more complicated when n is not a power of two.) Sums will be for $0 \leq i < n$, though there are likely to be many zeroes near the top.

2 Reduction to Polynomials

We first reduce to polynomial multiplication. Here the input is two polynomials $A(x) = \sum_i a_i x^i$, $B(x) = \sum_i b_i x^i$ and the output is $C(x) = \sum_i c_i x^i$ where $C(x) = A(x)B(x)$. (Similar to binary, the polynomials are represented by the array $a[i], b[i], c[i]$.) *Given* $C(x)$ we plug in $x = 2$. So $\alpha = A(2), \beta = B(2)$ and therefore $\gamma = C(2) = \sum c_i 2^i$. We're not quite done as we may, and generally will, have some $c_i \geq 2$. We get to the final product $d[i]$ by a simple CARRY routine:

```
CARRY = 0
FOR i = 0 TO (n-1)
  c[i]=c[i]+CARRY (* add carry *)
  CARRY = c[i]/2 (* new  carry *)
  c[i]=c[i]-2*CARRY (* remainder, zero or one *)
END FOR
```

(We are assuming integer division so, e.g., $7/2 = 3$. So when $c[i] = 7$ we set $CARRY = 3$ and reset $c[i]$ to 0.) The CARRY routine is a simple FOR loop, taking $O(n)$ times.

3 Complex Roots of Unity

FFT depends critically on complex numbers, in particular on the solutions to the equation $z^n = 1$. We set

$$\epsilon = e^{2\pi i/n} = \cos[2\pi/n] + i\sin[2\pi/n] \tag{2}$$

(If you aren't familiar with the $e^{i\theta}$ notation just use the $\sin, \cos$ notation.) This point is on the unit circle at angle $2\pi/n$ with the X-axis. The n solutions to the equation $z^n = 1$ are then given by

$$z = 1, \epsilon, \epsilon^2, \ldots \epsilon^{n-1} \tag{3}$$

Geometrically, these n points are the vertices of a regular n-gon on the unit circle. For $n = 4$, $\epsilon = i$, the points $1, i, -1, -i$ form a square.

`Background:` Complex multiplication is best understood using polar coordinates. Write nonzero $z = x + iy$ in polar coordinates (r, θ). (Or $z = re^{i\theta}$.) When z_1, z_2 have $(r_1, \theta_1), (r_2, \theta_2)$ their product $z = z_i z_2$ has $(r_1 r_2, \theta_1 + \theta_2)$. That is, multiply the r-values and add the θ-values. When points are on the unit circle, so $r = 1$, this is particularly nice. When $z = e^{i\theta} = (1, \theta)$, its s-th power is $z^s = e^{i(s\theta)} = (1, s\theta)$. When $z = \epsilon$, given above, the powers ϵ^s form the regular n-gon.

A key fact about ϵ we shall use is

$$\sum_{s=0}^{n-1} \epsilon^s = 0 \tag{4}$$

and, more generally, for any $0 < t < n$,

$$\sum_{s=0}^{n-1} (\epsilon^t)^s = 0 \tag{5}$$

We give two arguments for (4), (5 is similar). By the formula for sum of a geometric series

$$\sum_{s=0}^{n-1} \epsilon^s = \frac{\epsilon^n - 1}{\epsilon - 1} = \frac{1 - 1}{\epsilon - 1} = 0 \tag{6}$$

Or, geometrically, the points of the regular n-gon average out to their center, which is 0.

4 Discrete Fourier Transform

Let $A(x)$ be a polynomial of degree less than n. Recall A is given by an array $a[i] : 0 \leq i < n$. The Discrete Fourier Transform of A, written $DFT_n[A]$ is the array of values

$$DFT_n[A] = (A(1), A(\epsilon), A(\epsilon^2), \ldots, A(\epsilon^{n-1})) \tag{7}$$

That is, $DFT_n[A]$ is the values $A(z)$ where z ranges over the n n-th roots of unity.

Mathgeeks may compare this (others can ignore this!) with the standard Fourier Transform of a function $f(x)$ given by

$$\hat{f}(\theta) = \int_x f(x) e^{ix\theta} dx \tag{8}$$

Now we give the idea of polynomial multiplication. Input is $A(x), B(x)$ and output should be $C(x) = A(x)B(x)$, all polynomials of degree less than n.

1. Find
$$DFT_n[A] = (A(1), A(\epsilon), A(\epsilon^2), \ldots, A(\epsilon^{n-1})) \tag{9}$$

2. Find
$$DFT_n[B] = (B(1), B_n(\epsilon), B(\epsilon^2), \ldots, B(\epsilon^{n-1})) \tag{10}$$

3. Now as $C(\epsilon^s) = A(\epsilon^s)B(\epsilon^s)$ we multiply termwise to get
$$DFT_n[C] = (C(1), C_n(\epsilon), C(\epsilon^2), \ldots, C(\epsilon^{n-1})) \tag{11}$$

4. n values of a polynomial of degree at most n determine the polynomial (more on that later). Given $DFT_n[C]$, find $C = (c_0, \ldots, c_{n-1})$. This is called finding the *inverse* discrete Fourier Transform.

How long do these steps take. Consider finding $DFT_n[A]$. Calculating each $A[\epsilon^s]$ would involve the sum of n terms so would take time $O(n)$ so it *appears* that calculating the n different values would take time $O(n^2)$. **However,** the special plugin values $1, \epsilon, \ldots, \epsilon^{n-1}$ shall allow us to do the calculation in time our mantra $O(n \lg n)$.

Efficient implementation of DFT (e.g., keeping track of the parts A_1, A_2) can be done using ingenious methods of §30.3, which we do not cover. However, even clumsy implementation will yield the $O(n \lg n)$ final result – albeit with a poorer constant.

5 Calculating Discrete Fourier Transform

Our input is $A(x) = \sum_{i=0}^{n-1} a_i x^i$, given as an array $[a[i] : 0 \le i < n]$. The key is to split $A(x)$ into odd and even powers of x. The even powers are written as a function of x^2. The odd powers are written as x times a function of x^2. More precisely we set

$$A_1(x) = \sum_{j=0}^{\frac{n}{2}-1} a_{2j} x^j \tag{12}$$

and

$$A_2(x) = \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} x^j \tag{13}$$

so that

$$A(x) = A_1(x^2) + x A_2(x^2) \tag{14}$$

`Example:` With $n = 4$ and $A(x) = 3 + 5x + 2x^2 + 7x^3$ we set $A_1(x) = 3 + 2x$ and $A_2(x) = 5 + 7x$.

Now we come to a special property of the 2^t-th roots of unity. As x ranges over the 2^t-th roots of unity, x^2 ranges of the 2^{t-1}-st roots of unity. For example, with $t = 2$, the squares of $1, i, -1, -i$ and $1, -1, 1, -1$, the square roots of unity. So A_1 and A_2 need only be evaluated at the 2^{t-1}-st roots of unity, which is precisely $DFT_{n/2}$. This gives a *recursive* evaluation for $DFT_n[A]$:

1. Find, recursively $DFT_{n/2}[A_1]$. This gives the values $A_1(1), A_1(\epsilon^2), A_1(\epsilon^4) \ldots, A_1(\epsilon^{n-4}), A_1(\epsilon^{n-2})$.

2. Find, recursively $DFT_{n/2}[A_2]$. This gives the values $A_2(1), A_2(\epsilon^2), A_2(\epsilon^4), \ldots, A_2(\epsilon^{n-4}), A_2(\epsilon^{n-2})$.

3. For each $0 \le s < n$ find $A(\epsilon^s)$ by the formula

$$A(\epsilon^s) = A_1(\epsilon^{2s}) + \epsilon^s A_2(\epsilon^{2s}) \tag{15}$$

Let $T(n)$ be the time to calculate $DFT_n(A)$. Equation (15) takes time $O(1)$ (once the A_1, A_2 have been calculated) for each s for a total time $O(n)$. We have two recursive calls taking time $2T(n/2)$. This gives the recursion:

$$T(n) = 2T(n/2) + O(n) \tag{16}$$

This is "just right overhead" with solution our mantra

$$T(n) = O(n \lg n) \tag{17}$$

Going back to the idea in §4, Equations (9,10) each take time $O(n \lg n)$. Equation (11) takes time $O(n)$. This leaves us with the inverse Discrete Fourier Transform.

6 Calculating Inverse Discrete Fourier Transform

We'll take $n = 4$ as an example. Let $C(x) = a + bx + cx^2 + dx^3$. We know $DFT_4[C]$ – that is, we know $C(1), C(i), C(-1), C(-i)$ – and we want to find a, b, c, d. This gives four equations in the four unknowns a, b, c, d

```
C(1)  =    a  +   b  +  c  +   d
C(i)  =    a  +  ib  + -c  + -id
C(-1) =    a  +  -b  +  c  +  -d
C(-i) =    a  + -ib  + -c  +  id
```

Normally n equations in n unknowns takes *lots* of time to solve. But these equations are *very* special. If we add them up the coefficients of b, c, d all cancel! So

$$4a = C(1) + C(i) + C(-1) + C(-i) \tag{18}$$

How about b? Multiply each equation in order to make the coefficient of b equal to 1 – by $1, -i, -1, i$ respectively. (In the general pattern below think of this as multiplying by $1, i^{-1}, i^{-2}, i^{-3}$.) This gives

```
  C(1)  =    a  +   b  +   c  +   d
-iC(i)  =  -ia  +   b  +  ic  +  -d
 -C(-1) =   -a  +   b  +  -c  +   d
 iC(-i) =   ia  +   b  + -ic  +   d
```

Then add. Now the coefficients of a, c, d all cancel. So

$$4b = C(1) + (-i)C(i) + (-1)C(-1) + iC(-i) \tag{19}$$

and $4c, 4d$ are similar.

What is the general pattern? Write $C(x) = \sum_{i=0}^{n-1} c_j x^i$. The n equations in the n unknowns c_j become:

$$C(\epsilon^s) = \sum_{j=0}^{n-1} c_j \epsilon^{sj} \tag{20}$$

for $0 \le s < n$. Now we want to solve for some c_t. We first multiply the s-th equation by ϵ^{-st} giving

$$\epsilon^{-st}C(\epsilon^s) = \sum_{j=0}^{n-1} c_j \epsilon^{sj} \epsilon^{-st} \tag{21}$$

Now add equation (20) over $0 \le s < n$. We have made the coefficient of c_t equal to one in each equation. So when we add we get an nc_t term. What about terms c_u with $u \ne t$. The coefficient of c_u is

$$\sum_{s=0}^{n-1} \epsilon^{su}\epsilon^{-st} = \sum_{s=0}^{n-1} (\epsilon^{(u-t)})^s \tag{22}$$

From (5), with t replaced by $u-t$, this geometric sum is zero. That is, the coefficients of c_u all cancel. The only thing left is the nc_t term. That is:

$$nc_t = \sum_{s=0}^{n-1} \epsilon^{-st}C(\epsilon^s) \tag{23}$$

It is helpful here to think of the subscripts t as calculated modulo n so that we can write $-t$. For example, with $n = 16$, -5 would be 11. Now replacing t by $-t$, (23) becomes

$$nc_{-t} = \sum_{s=0}^{n-1} \epsilon^{st}C(\epsilon^s) \tag{24}$$

This is *nearly* the formula for Discrete Fourier Transform. We make a little massaging to get from one to the other. Given $(\alpha_0, \ldots, \alpha_{n-1})$ we wish to find $(c_0, \ldots, c_{n-1})$ so that $DFT_n(c_0, \ldots, c_{n-1}) = (\alpha_0, \ldots, \alpha_{n-1})$. We do this in three steps:

1. Calculate $(d_0, \ldots, d_{n-1}) = DFT_n(\alpha_0, \ldots, \alpha_{n-1})$
2. Divide each term by n giving $(e_0, \ldots, e_{n-1})$ with $e_t = d_t/n$.
3. Reverse the e array giving $(c_0, \ldots, c_{n-1})$ with $c_0 = e_0$ and $c_t = e_{n-t}$ for $0 < t < n$.

Then $(c_0, \ldots, c_{n-1})$ is the inverse Discrete Fourier Transform of $(\alpha_0, \ldots, \alpha_n)$.

Division and reversal are both $O(n)$ so the main time is the DFT_n which takes $O(n \lg n)$.

We have completed all the steps for polynomial multiplication. The total time is $O(n \lg n)$. Hence we also have integer multiplication, taking time $O(n \lg n)$.

7 Rumination

The Fourier Transform is a powerful technique in mathematics that has been developed over the centuries. How remarkable that it is applicable to one of the most basic algorithmic challenges – multiplication!