Data Structures and Algorithms

---

12 Multiplication of Polynomials

If we have two polynomials:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

and

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

then, we can form a product:

$$C(x) = A(x)B(x) = \sum_{0}^{2n-2} c_j x^j$$

where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

Clearly, this algorithm is $\Theta(n^2)$.

12.1 Point-Value Representation

In section 12, we used the *coefficient form* to represent the polynomials. There is an alternative way to represent polynomials, using the *point-value representation*. A polynomial of degree $n-1$ can be represented by n point-value pairs:

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

where

- All the x_k are distinct
- $y_k = A(x_k)$

12.2 Uniqueness Theorem

For any set of n points, there is a unique polynomial.
(For a proof, *see* Cormen *et al.*).

12.3 Operations in the Point-Value Representation

If we have two polynomials, A and B:

$$A(x) = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

and

$$B(x) = \{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$$

where all the x_k are the same, then

$$C(x) = A(x) + B(x) = \{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\}$$

and

$$C(x) = A(x)\dot{B}(x) = \{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{n-1}, y_{n-1} y'_{n-1})\}$$

giving a $\Theta(n)$ procedure for multiplying two polynomials. But note that we now need $2n$ points in our representations of A and B.
Unfortunately, conversion between coefficient and point-value representations requires generating n or $2n$ points. Evaluating $y_k = A(x_k)$ requires $\Theta(n)$ operations using Horner's rule, so the conversion is $\Theta(n^2)$!
However, note that we can choose any x_k as long as they are distinct. If we choose the *n complex n^{th} roots of unity*, then we are performing a Discrete Fourier Transform. As we will show, this takes $\Theta(n \log n)$ time. The reverse transformation (from point-value to coefficient representation), called *interpolation*, is also a Fourier Transform, also requiring $\Theta(n \log n)$ time.
This leads us to the following polynomial multiplication algorithm:

Augment polynomials A and B by adding n higher order 0 coefficients	$\Theta(n)$
Evaluate Compute point-value representation of $A(x)$ and $B(x)$ by two applications of the FFT $\Rightarrow$ values at each $(2n)^{th}$ root of unity.	$\Theta(n \log n)$
Point-wise multiply	$\Theta(n)$
Interpolate Create coefficient representation by application of FFT	$\Theta(n \log n)$
Total	$\Theta(n \log n)$

12.4 Complex Roots of Unity

The complex n^{th} root of unity, ω, is defined by:

$$\omega^n = 1$$

There are exactly n such roots:

$$e^{\frac{2\pi i k}{n}} = \cos\frac{2\pi k}{n} + i \sin\frac{2\pi k}{n} \quad for\ k = 0, 1, \ldots, n-1$$

The *principal n^{th} root of unity* is

$$\omega_n = e^{\frac{2\pi i}{n}}$$

The other roots are

$$\omega_n^2, \omega_n^3, \ldots, \omega_n^{n-1}$$

12.5 Properties of ω

Cancellation lemma

For any $n \geq 0, k \geq 0, d > 0$:

$$\omega_{dn}^{dk} = \omega_n^k$$

For any even $n > 0$:

$$\omega_n^{\frac{n}{2}} = \omega_2 = -1$$

Halving lemma

If n is even, the squares of the n complex n^{th} roots of unity are the $\frac{n}{2}$ complex $(\frac{n}{2})^{th}$ roots.

12.6 The Fast Fourier Transform

To evaluate $A(x)$, we divide it into two parts:

$$A^{even}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{\frac{n}{2}-1}$$

and

$$A^{odd}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{\frac{n}{2}-1}$$

now

$$A(x) = A^{even}(x^2) + A^{odd}(x^2) x$$

So evaluating $A(x)$ at

$$\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$$

reduces to

1. Evaluating $A^{even}(x)$ and $A^{odd}(x)$ at

   $$(\omega_n^0)^2, (\omega_n^1)^2, (\omega_n^2)^2, \ldots, (\omega_n^{n-1})^2$$

   but these are the complex $(\frac{n}{2})^{th}$ roots of unity and there are only $\frac{n}{2}$ of them.

2. Combining them.

So we have divided our $\Theta(n^2)$ evaluation into two sub-problems of size $\frac{n}{2}$. As usual, we can divide the problem $\log n$ times, giving a $\Theta(n \log n)$ evaluation algorithm.
Thus the time complexity of FFT is $\Theta(n \log n)$ and we have developed a $\Theta(n \log n)$ algorithm for multiplying polynomials.

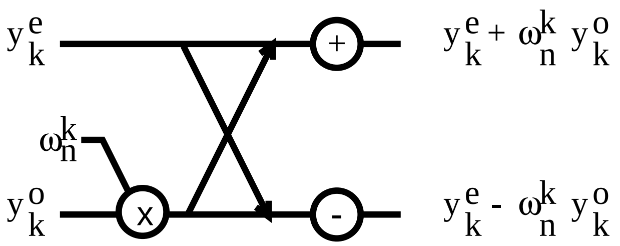
12.7 Recursive FFT

A recursive FFT algorithm has the following form:

$$
\begin{array}{l}
fft(a, n) \\
\quad if\ n = 1\ then\ return\ a; \\
\quad \omega_n = e^{\frac{2\pi i}{n}} \\
\quad \omega = 1 \\
\quad a^e = (a_0, a_2, a_4, \ldots, a_{n-2}) \\
\quad a^o = (a_1, a_3, a_5, \ldots, a_{n-1}) \\
\quad y^e = fft(a^e, \frac{n}{2}) \\
\quad y^o = fft(a^o, \frac{n}{2}) \\
\quad for\ k = 0\ to\ \frac{n}{2} - 1 \\
\quad\quad y_k = y^e_k + \omega y^o_k \\
\quad\quad y_{k+\frac{n}{2}} = y^e_k - \omega y^o_k \\
\quad\quad \omega = \omega\omega_n \\
\quad return\ y
\end{array}
$$

12.8 The Butterfly

To save the overhead of recursive calls, efficient FFT routines are iterative, replacing the n values input to each of the $\log n$ stages with the new values calculated in that stage.
The combination stage in the recursive algorithm may be represented diagrammatically:



Standard iterative algorithms simply stack n of these 'butterflies' (so-called because of the shape of the diagram) vertically and then replicate this stack for $\log n$ stages. The results from each stage overwrite those from the previous one. A 'bit-twiddling' function determines which butterfly in the stage $k+1$ the outputs of the butterfly in stage k are sent.