Polynomial Multiplication (Fast Fourier Transform)

In this problem we have two polynomials

\[ A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_{n-1}x^{n-1} \]
\[ B(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + \ldots + b_{n-1}x^{n-1} \]

We assume \( n \) is a power of 2 and that \( a_{n/2} = a_{n/2+1} = \ldots = a_{n-1} = 0, b_{n/2} = b_{n/2+1} = \ldots = b_{n-1} = 0 \). This is without loss of generality, since if the actual degrees are \( d \) then we can round \( 2(d+1) \) up to the nearest power of 2 and let that be \( n \), at most roughly quadrupling the degree (coefficients of \( x^{d+1}, x^{d+2}, \ldots, x^{n-1} \) will just be set to 0).

The goal in this problem is to compute the coefficients of the product polynomial

\[ C(x) = c_0 + c_1x + c_2x^2 + \ldots + c_{n-2}x^{n-1}. \]

where

\[ c_k = \sum_{i=0}^{k} a_i b_{k-i}, \text{ treating } a_i, b_i \text{ as } 0 \text{ for } i \geq n. \]

Note that by our choice of rounding degrees up to \( n \), the degree of \( C \) is less than \( n \) (it is at most \( n - 2 \) in fact).

The straightforward algorithm is to, for each \( k = 0, 1, \ldots, n-1 \), compute the sum above in \( 2k + 1 \) arithmetic operations. The total number of arithmetic operations is thus \( \sum_{k=0}^{n-1}(2k + 1) = \Theta(n^2) \). Can we do better? We shall, by making use of the following interpolation theorem!

**Theorem 10.1** (Interpolation) A degree-\( N \) polynomial is uniquely determined by its evaluation on \( N + 1 \) points.

**Proof:** Suppose \( P(x) \) has degree \( N \) and for \( N + 1 \) distinct points \( x_1, \ldots, x_{N+1} \) we know that \( y_i = P(x_i) \). Then we have the following equation:

\[
\begin{pmatrix}
1 & x_1 & x_1^2 & \ldots & x_1^N \\
1 & x_2 & x_2^2 & \ldots & x_2^N \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
1 & x_{N+1} & x_{N+1}^2 & \ldots & x_{N+1}^N \\
\end{pmatrix}
\begin{pmatrix}
p_0 \\
p_1 \\
\vdots \\
p_N \\
\end{pmatrix}
= 
\begin{pmatrix}
y_1 \\
y_2 \\
\vdots \\
y_{N+1} \\
\end{pmatrix}
\]
That is, $Vp = y$. Thus, if $V$ were invertible we would be done: $p$ would be uniquely determined as $V^{-1}y$. Well, is $V$ invertible? A matrix is invertible if and only if its determinant is non-zero. It turns out this matrix $V$ is well studied and is known as a Vandermonde matrix. The determinant is known to have the closed form $\prod_{1 \leq i < j \leq N+1} (x_i - x_j) \neq 0$. The proof of this determinant equality is by induction on $N$, using elementary row operations. We leave it as an exercise (or you can search for a proof online).

Thus we will employ a different strategy! Rather than directly multiply $A \times B$, we will pick $n$ points $x_1, \ldots, x_n$ and compute $\alpha_i = A(x_i), \beta_i = B(x_i)$ for each $i$. Then we can obtain values $y_i = C(x_i) = \alpha_i \cdot \beta_i$, which by the interpolation theorem uniquely determines $C$. Given all the $\alpha_i, \beta_i$ values, it just takes $n$ multiplications to obtain the $y_i$'s, but there is of course a catch: computing a degree $n-1$ polynomial such as $A$ on $n$ points naively takes $\Theta(n^2)$ time. We also have to translate the $y_i$ into the coefficients of $C$. Doing this naively requires computing $V^{-1}$ then multiplying $V^{-1}y$ as in the proof of the interpolation theorem, but matrix inversion on an arbitrary $n \times n$ matrix itself must clearly take $\Omega(n^2)$ time (just writing down the inverse takes this time!), and in fact the best known algorithm for matrix inversion is as slow as matrix multiplication.

Our approach for getting around this is to not pick the $x_i$ arbitrarily, but to pick them to be algorithmically convenient, so that multiplying by $V, V^{-1}$ can both be done quickly. In what follows we assume that our computer is capable of doing infinite precision complex arithmetic, so that adding and multiplying complex numbers takes constant time. Of course this isn’t realistic, but it will simplify our presentation (and it turns out this basic approach can be carried out in reasonably small precision).

We will pick our $n$ points as $1, w, w^2, \ldots, w^{n-1}$ where $w = e^{2\pi i/n}$ (here $i = \sqrt{-1}$) is a “primitive $n$th root of unity”. While this may seem magical now, we will see soon why this choice is convenient. Also, a recap on complex numbers: a complex number is of the form $a + ib$ where $a, b$ are real. Such a number can be drawn in the plane, where $a$ is the $x$-axis coordinate, and $b$ is the $y$-axis coordinate. Then we can also represent such a number in polar coordinates by the radius $r = \sqrt{a^2 + b^2}$ and angle $\theta = \tan^{-1}(b/a)$. A random mathematical fact is that $e^{i\theta} = \cos \theta + i \sin \theta$, and thus we can write such an $a + ib$ as $re^{i\theta}$. Note that if one draws $1, w, w^2, \ldots, w^{n-1}$ in the plane, we obtain equispaced points on the circle (and note the next power in the series $w^n = (e^{2\pi i/n})^n = e^{2\pi i} = \cos(2\pi) + i \sin(2\pi) = 1$ just returns back to the start).

Now, I claim that divide and conquer lets us compute all the evaluations $A(1), A(w), \ldots, A(w^{n-1})$ in $O(n \log n)$
time via divide and conquer! This is via an algorithm known as the Fast Fourier Transform (FFT). Let us write

\[
A(x) = \frac{A_{\text{even}}(x^2)}{A_{\text{even}}(x^2)} + x \frac{A_{\text{odd}}(x^2)}{A_{\text{odd}}(x^2)}
\]

Note the polynomials \(A_{\text{even}}, A_{\text{odd}}\) each have degree \(n/2 - 1\). Thus evaluating a degree-\((n-1)\) polynomial on \(n\) points reduces to evaluating two degree-\((n/2 - 1)\) polynomials on the \(n\) points \(\{1^2, w^2, (w^2)^2, (w^2)^3, \ldots, (w^{n-1})^2\}\). Note however that \(w^2\) is just \(e^{4\pi i/n} = e^{2\pi i/(n/2)}\) an \((n/2)\)th root of unity! Thus, this set of “\(n\)” points is actually only a set of \(n/2\) points, since \((w^2)^{n/2}, (w^2)^{n/2+1}, \ldots, (w^2)^{n-1}\) again equals \(1, w^2, (w^2)^2, \ldots, (w^2)^{n/2-1}\). Thus what we have is that the running time to compute \(A(x)\) satisfies the recurrence

\[
T(n) = 2T(n/2) + \Theta(n).
\]

This is the same recurrence we’ve seen before for MergeSort, and we know it implies \(T(n) = \Theta(n \log n)\). Evaluating \(B\) at these points takes the same amount of time.

Ok, great, so we can get the evaluations of \(C\) on \(n\) points. But how do we translate that into the coefficients of \(c\)? Basically, this amounts to computing \(V^{-1}y\). So what’s \(V^{-1}\)?

**Lemma 10.2** For \(x_1, \ldots, x_{n+1} = 1, w, \ldots, w^{n-1}\) for \(w\) an \(n\)th root of unity, we have that for \(V\) the corresponding Vandermonde matrix, \((V^{-1})_{i,j} = \frac{1}{n} \cdot w^{-ij}\).

**Proof:** First of all, we should observe that \(V_{i,j} = w^j\). Define \(Z\) to be the matrix \(Z_{i,j} = \frac{1}{n} \cdot w^{-ij}\). We need to verify that \(Z = V^{-1}\), i.e. that \(VZ\) is the identity matrix, so we should have \((VZ)_{i,j}\) being 1 for \(i = j\) and 0 otherwise.

\[
(VZ)_{i,j} = \frac{1}{n} \sum_{k=0}^{n-1} w^j w^{-kj} = \frac{1}{n} \sum_{k=0}^{n-1} (w^{i-j})^k
\]

If \(i = j\) then this is \(\frac{1}{n} \cdot n = 1\) as desired. If \(i \neq j\), then we have a geometric series which gives us

\[
\frac{1}{n} \cdot \frac{(w^{i-j})^n - 1}{w^{i-j} - 1} = \frac{1}{n} \cdot \frac{(w^n)^{i-j} - 1}{w^{i-j} - 1} = \frac{1}{n} \cdot \frac{1 - 1}{w^{i-j} - 1} = 0
\]

as desired. —

Now, recall for a complex number \(z = a + ib = re^{i\theta}\), its complex conjugate \(\bar{z}\) is defined as \(\bar{z} = a - ib = re^{-i\theta}\). Let us define for a vector \(x = (x_1, \ldots, x_N)\) of complex numbers its complex conjugate \(\bar{x} = (\bar{x}_1, \ldots, \bar{x}_N)\). Then the above lemma implies that \(V^{-1}y = \overline{V\bar{y}}\) (check this as an exercise!). The complex conjugation operator can be applied
to $y$ in linear time, then we can multiply by $V$ in $O(n \log n)$ time (this is just the FFT again, on a polynomial whose coefficients are given by $\overline{y}!$), then we can apply conjugation to the result again in linear time.

This completes the description and analysis of the FFT.

### Applications of the FFT

**Integer multiplication.** Suppose we have two integers $a, b$ which are each $n$ digits and we want to multiply them. For example, we might have $a = 1254, b = 2047$. We can treat an integer as simply a certain polynomial evaluated at $10$. For example, define $A(x) = 4 + 5x + 2x^2 + x^3$ and $B(x) = 7 + 4x + 2x^3$. Then $a \cdot b = c$ is just $(A \cdot B)(10)$. Thus we just need to compute $C = A \cdot B$ and evaluate it at the point 10, which we can using $O(n \log n)$ arithmetic operations via the FFT.

Note that we have not discussed precision at all, which actually should be carefully considered. First, there is the issue of the precision to which we represent $w$ in the FFT and do our FFT arithmetic, so that our accumulated errors are sufficiently small; we do not analyze that here though it is known that it is possible to do this reasonably (see [S96]). There is also the issue though that, while the coefficients of $A, B$ are each digits 0–9 and can be concatenated to form the integers $a, b$, this is not true for obtaining $c$ from the coefficients of $C$. This is because if we write

$$A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$

and similarly for $B$, then we have

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$ 

That is, the coefficients of $c$ are not in the range 0–9. In fact they can be as big as $9^2 \cdot n = 81n$. Thus when we evaluate $C(10)$, we cannot simply concatenate the coefficients of $C$ to obtain $c$: we have to deal with the carries. The key thing to note is that if a number is at most $81n$, then writing it base 10 takes at most $D = \lceil \log_{10}(81n) \rceil = O(\log n)$ digits. Thus, evaluating $C(10)$ even naively requires $O(nD) = O(n \log n)$ arithmetic operations.

**Pattern matching.** Suppose we have binary strings $P = p_0p_1\cdots p_{m-1}$ and $T = t_0t_1\cdots t_{n-1}$ for $n \geq m$, and the $p_i, t_i$ each being 0 or 1. We would like to report all indices $i$ such that the length-$m$ string starting at position $i$ in $T$ matches the smaller string $P$. That is, $T[i, i+1, \ldots, i+m-1] = P$. The naive approach would try all possible $n-m+1$ starting positions $i$ and check whether this holds character by character, taking time $\Theta((n-m+1)m) = O(nm)$. It turns out that we can achieve $O(n \log n)$ using the FFT!
First we double the lengths of $P, T$ by the following encoding: for each letter which is 0, we map it to 01, and for each that is 1 we map it to 10. So for example, to find all occurrences of $P = 101$ in $T = 10101$, we would actually search for all occurrences of $P' = 100110$ in $T' = 10011001$. Write $P' = a_0a_1 \cdots a_{2m-1}$, $T' = b_0b_1 \cdots b_{2n-1}$. We now define the polynomials

$$A(x) = a_2m^{-1} + a_2m^{-2}x + a_2m^{-3}x^2 + \cdots + a_0x^{2m-1}$$

and

$$B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{2n-1}x^{2n-1}. $$

(Note that the coefficients in $A$ are reversed from the order they appear in $P'$!) Now let us see what $C(x) = A(x) \cdot B(x)$ gives us. If $i \geq 2m - 1$, let us write $i = 2m - 1 + j$ for $j \geq 0$. Then we have that the coefficient of $x^i$ in $C(x)$ is

$$c_i = \sum_{k=0}^{2m-1} a_{2m-1-k}b_{2m-1-k+j}.$$ 

That is, we are taking the dot product of $T'[j, j+1, \ldots, j+2m-1]$ with $P'$. Because of our encoding of 0 as 01 and 1 as 10, note that two matching characters in $T, P$ contribute 1 to the dot product, and mismatched characters contribute 0. Thus the sum above for $c_i$ will be exactly $m$ if $T[j/2, j/2+1, \ldots, j/2+m-1]$ matches $P$ and will be strictly less than $m$ otherwise (the division by 2 is since we blew up the lengths of $T, P$ each by a factor of 2). Thus by simply checking which $c_{2m-1+j}$ for even $j$ are equal to exactly $m$, we can read off the locations of occurrences of $P$ in $T$. The time is dominated by the FFT, which is $O(n \log n)$.

**References**