Multiplication using the FFT

We previously saw that the simple algorithm of *long multiplication* is equivalent to a convolution for each output digit, and thus has complexity of $O(N^2)$ where N is the number of bits in the multiplicands. Later we saw that the Toom-Cook algorithm lowered the complexity to $O(N^{\log_2 3})$ but remarked that using the FFT we can lower the complexity even further. What is the connection between an algorithm for converting signals form the time domain representation to the frequency domain one and simple multiplication? That is the subject of this note.

Remember how we multiply two N-bit integers $a = a_{N-1}a_{N-2}\ldots a_3a_2a_1a_0$ and $b = b_{N-1}b_{N-2}\ldots b_3b_2b_1b_0$. we saw that the formula for the n^{th} bit of the output c is formally a convolution $c_n = \sum_l a_l b_{n-l}$ where l runs over all integral values that make sense. Convolutions remind us of filtering signals in the time domain. Indeed, we can think of each integer as a signal in the time domain representation; a signal that for each time may take only one of two values – 0 or 1.

Let's make this concrete by taking $N = 4$. The 4-bit numbers are 0 = 0000, 1 = 0001, 2 = 0010, ... 14 = 1110, 15 = 1111. We can represent the number $a = a_{N-1}a_{N-2}\ldots a_3a_2a_1a_0$ by the signal $a = (a_0, a_1, a_2, \ldots a_{N-2}a_{N-1})$. Note that since the index 0 represents the earliest time, it appears first in the time representation, although as LSB it appeared last in the bit representation. All the 4-bit numbers with their representations are given in the following table.

a	bit representation	time representation
0	0000	(0, 0, 0, 0)
1	0001	(1, 0, 0, 0)
2	0010	(0, 1, 0, 0)
3	0011	(1, 1, 0, 0)
4	0100	(0, 0, 1, 0)
5	0101	(1, 0, 1, 0)
6	0110	(0, 1, 1, 0)
7	0111	(1, 1, 1, 0)
8	1000	(0, 0, 0, 1)
9	1001	(1, 0, 0, 1)
10	1010	(0, 1, 0, 1)
11	1011	(1, 1, 0, 1)
12	1100	(0, 0, 1, 1)
13	1101	(1, 0, 1, 1)
14	1110	(0, 1, 1, 1)
15	1111	(1, 1, 1, 1)

Now we can check that the convolution formula really works. We will limit ourselves to products that fit into 4 bits $0 * s = 0$, $1 * s = s$, $2 * 3 = 12$, $2 * 4 = 8$, $2 * 5 = 10$, $2 * 6 = 12$,$2 * 7 = 14$, $3 * 4 = 12$, and $3 * 5 = 15$.

We will further use superscripts to indicate which number the time representation represents; e.g., $s^{[10]}$ represents the number 10, so that $s^{[10]} = (0, 1, 0, 1)$, i.e., $s_0^{[10]} = 0$, $s_1^{[10]} = 1$, $s_2^{[10]} = 0$, and $s_3^{[10]} = 1$. Using this notation the formula for the n^{th} bit of the output is easily given by $s_n^{[a*b]} = \sum_{l=0}^{n} s_l^{[a]} s_{n-l}^{[b]}$.

For example, let's see how the convolutions of long multiplication compute $2*3$.

$$\begin{aligned}
s_0^{[2*3]} &= s_0^{[2]}s_0^{[3]} &&= 0*1 = 0\\
s_1^{[2*3]} &= s_0^{[2]}s_1^{[3]} + s_1^{[2]}s_0^{[3]} &&= 0*0 + 1*1 = 1\\
s_2^{[2*3]} &= s_0^{[2]}s_2^{[3]} + s_1^{[2]}s_1^{[3]} + s_2^{[2]}s_0^{[3]} &&= 0*1 + 1*1 + 0*1 = 1\\
s_3^{[2*3]} &= s_0^{[2]}s_3^{[3]} + s_1^{[2]}s_2^{[3]} + s_2^{[2]}s_1^{[3]} + s_3^{[2]}s_0^{[3]} &&= 0*0 + 1*0 + 0*1 + 0*1 = 0
\end{aligned}$$

So $s^{[2*3]} = (0, 1, 1, 0)$ which is the time representation of 6. The computation only took 10 multiplications and not the full $4^2 = 16$, since only the last output bit required all 4 multiplications.

Now that we understand how to convert integer numbers into time domain signals let's see how the FFT helps perform multiplications. We know from our study of filters that all filters obey the *law of filters* $Y_k = H_k X_k$. The long multiplication convolution in the time domain $a*b$ can be considered a MA filter, and so can be computed as a simple multiplication in the frequency domain. Unfortunately, we have the multiplicands in the time domain representation and desire the product in that domain, so we do what we always do in DSP, we go back and forth between the representations using the FFT. So instead of performing the convolution between $s^{[a]}$ and $^{[b]}$, we perform a first FFT to convert $s^{[a]}$ into its frequency domain representation $S^{[a]}$, and a second FFT to convert $s^{[b]}$ into its frequency domain representation $S^{[b]}$. Now we only need to perform N simple index-by-index multiplications $S_k^{[a]}S_k^{[b]}$ to obtain the answer in the frequency domain representation, and a final inverse FFT to get the desired result.

That sounds complicated – is it worth it? Well, the two FFTs and the one iFFT take $O(N \log N)$ each, and the simple multiplication takes $O(N)$, so altogether the complexity is $O(N \log N)$. Since this is less than $O(N^2)$, and even less than $O(N^{\log_2 3})$, at least for very large N the FFT approach will be faster than our previous approaches ($N \log N$ is less than N^x for all $x > 1$).

Let's return to our 4-bit numbers to see how this works. Remember that the DFT matrix for $N = 4$ is

$$W_4 = \begin{pmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^1 & W_4^2 & W_4^3 \\ W_4^0 & W_4^2 & W_4^4 & W_4^6 \\ W_4^0 & W_4^3 & W_4^6 & W_4^9 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \end{pmatrix}$$

We can use this matrix to find all the frequency domain representation of the 4-bit numbers. All we have to do is to multiply the vectors previously given in the table by this matrix. Of course, using the DFT requires N^2 complex multiplications (in this case 16), but we *could* use the FFT to obtain the same result. In any case for $N = 4$ there are no true multiplications at all! Only negations and multiplications by i which merely interchange the real and imaginary parts.

Without further ado we present the desired representations in the following table, and invite the reader to verify the entries.

a	time representation	frequency representation
0	(0, 0, 0, 0)	(0, 0, 0, 0)
1	(1, 0, 0, 0)	(1, 1, 1, 1)
2	(0, 1, 0, 0)	(1, -i, -1, +i)
3	(1, 1, 0, 0)	(2, 1-i, 0, 1+i)
4	(0, 0, 1, 0)	(1, -1, 1, -1)
5	(1, 0, 1, 0)	(2, 0, 2, 0)
6	(0, 1, 1, 0)	(2, -1-i, 0, -1+i)
7	(1, 1, 1, 0)	(3, -i, 1, +i)
8	(0, 0, 0, 1)	(1, +i, -1, -i)
9	(1, 0, 0, 1)	(2, 1+i, 0, 1-i)
10	(0, 1, 0, 1)	(2, 0, -2, 0)
11	(1, 1, 0, 1)	(3, 1, -1, 1)
12	(0, 0, 1, 1)	(2, -1+i, 0, -1-i)
13	(1, 0, 1, 1)	(3, +i, 1, -i)
14	(0, 1, 1, 1)	(3, -1, -1, -1)
15	(1, 1, 1, 1)	(4, 0, 0, 0)

We can note a few interesting things here. The first element in the frequency domain is the sum of all elements (i.e., the un-normalized DC component). Hence $s^{[15]}$ which is constant and has only a DC component, is zero for all $S^{15}_{k>0}$, and of course $S^{15}_0 = 4$. Also, $s^{[5]}$ is similar to the Nyquist signal $s^{[Nyquist]} = (+1, -1, +1, -1)$, except for an offset which is equivalent to a DC component; hence $S^{[5]}_k$ is only non-zero for $k = 0$ and $k = 2$ which corresponds to the Nyquist frequency (the standard order is from DC to sampling frequency). Finally, note that the four signals $s^{[1]}$, $s^{[2]}$, $s^{[4]}$, and $s^{[8]}$ all have the same energy for all frequencies, making them white noise.

Now let perform in the frequency domain the same calculation $2 * 3$ that we performed above in the time domain.

$$\begin{aligned}
S^{[2*3]}_0 &= S^{[2]}_0 S^{[3]}_0 &&= 1 * 2 = 2 \\
S^{[2*3]}_1 &= S^{[2]}_1 S^{[3]}_1 &&= -\mathrm{i} * (1 - \mathrm{i}) = -1 - \mathrm{i} \\
S^{[2*3]}_2 &= S^{[2]}_2 S^{[3]}_2 &&= -1 * 0 = 0 \\
S^{[2*3]}_3 &= S^{[2]}_3 S^{[3]}_3 &&= \mathrm{i} * (1 + \mathrm{i}) = -1 + \mathrm{i}
\end{aligned}$$

We see that the answer is $S^{[2*3]} = (2, -1 - \mathrm{i}, 0, -1 + \mathrm{i})$ which is indeed $S^{[6]}$. It took us just 4 complex multiplications to do this, but of course we needed to prepare the table first.

We leave it to the reader to check all the possible multiplications that fit into 4 bits. It is obvious that multiplication by $S^{[0]} = (0, 0, 0, 0)$ always gives $S^{[0]}$ and that multiplication by $S^{[1]} = (1, 1, 1, 1)$ returns the multiplicand. The remaining cases are straightforward to verify.

One final question – what happens when the product doesn't fit into 4 bits? For example, it is easy to see that $S^{[4*4]} = S^{[4]} \cdot S^{[4]} = (1, -1, 1, -1) \cdot (1, -1, 1, -1) = (1, 1, 1, 1) = S^{[1]}$ which corresponds to 1 and not 16. What's going on? (Hint: to accommodate *all* products of 4-bit numbers we need 16 bits; what does the uncertainty theorem tell us about the frequency domain representation?)