FFT and Schönhage–Strassen

Yuval Filmus

July 7, 2017

1 Polynomial multiplication and convolution

Consider the following problem. Given two univariate polynomials

$$P = \sum_{i=0}^{n} p_i x^i, \quad Q = \sum_{i=0}^{n} q_i x^i,$$

compute the product polynomial

$$R = \sum_{i=0}^{2n} r_i x^i, \quad r_i = \sum_{j,k \colon j+k=i} p_j q_k.$$

When we say "compute R", we really mean "compute the coefficients of R".

Each r_i is the sum of at most n products, so we can compute R using $O(n^2)$ operations. Surprisingly (or perhaps, at this state of the course, not so surprisingly), this can be significantly improved to $O(n \log n)$ using the fast Fourier transform (FFT).

The Fourier transform is often approached from a physics perspective, but here we will take the point of view of representation theory[1]. According to this point of view, we think of P and Q as linear combinations of the "formal monomials" $x^0, \ldots, x^n$. If we identify x^i with the natural number i, then this views P, Q as members of the vector space $\mathbb{C}[\mathbb{N}]$ (we will see later that it is advantageous to work over the complex numbers rather than the real numbers). We turn this vector space into an algebra by defining a multiplication operation. It is defined on basis elements as $x^i \cdot x^j = x^{i+j}$, and extended linearly for arbitrary vectors (this ensures that $P(aQ + bR) = aPQ + bPR$). The resulting algebraic structure is known as the *monoid algebra* of $\mathbb{N}$ (over $\mathbb{C}$).

Since we are looking for algorithmic solutions, it is probably not such a good idea to look at the infinite-dimensional $\mathbb{C}[\mathbb{N}]$. Since $\deg R \leq 2n$, it suffices to work over a basic structure which has analogs of $x^0, \ldots, x^{2n}$. One possible choice is the group $\mathbb{Z}_m$ for some $m > 2n$. We thus look at the vector space $\mathbb{C}[\mathbb{Z}_m]$, and endow it with the multiplication operation defined on basis elements by $x^i \cdot x^j = x^{i+j \bmod m}$, and extended linearly (this operation is sometimes called *convolution*). We obtain the so-called *group algebra* of $\mathbb{Z}_m$ (over $\mathbb{C}$). The reader can check that if we consider P, Q to be elements of $\mathbb{C}[\mathbb{Z}_m]$ then their product PQ in the group algebra encodes the polynomial PQ.

We have thus reduced the problem of multiplying univariate polynomials to that of multiplying two elements in the group algebra $\mathbb{C}[\mathbb{Z}_m]$. Representation theory tells us[2] that there is a basis $\chi_0, \ldots, \chi_{m-1}$ of

[1] Our presentation will be somewhat unorthodox even from that perspective.

[2] Each element x of the group algebra corresponds to the linear operator T_x which maps y to xy. Since the group algebra $\mathbb{C}[Z_m]$ is m-dimensional whereas the dimension of the space of $m \times m$ matrices is m^2, not every linear operator on the matrix algebra is realizable as T_x for some element T_x in the group algebra. Since $\mathbb{Z}_m$ is abelian, representation theory tells us that there is a common basis of eigenvectors for the T_x. With respect to this basis, the T_x become diagonal operations. Since the dimension of the space of $m \times m$ diagonal matrices is m, this fixes the dimension issue: every diagonal matrix in this basis is realizable as T_x, and vice versa. The non-abelian case is more complicated, but well-understood.

the group algebra which satisfies the following identity:

$$\left(\sum_{j=0}^{m-1} \alpha_j \chi_j\right)\left(\sum_{j=0}^{m-1} \beta_j \chi_j\right) = m \sum_{j=0}^{m-1} \alpha_j \beta_j \chi_j.$$

It also tells us what the basis is:

$$\chi_j = \sum_{i=0}^{m-1} \omega^{ij} x^i,$$

where $\omega = e^{2\pi i/m}$ is a primitive mth root of unity (that is, $\omega^m = 1$, and $\omega^r \neq 1$ for all $1 \leq r < m$). In a subsection below, we demystify this, and some subsequent, formulas.

Given this explicit form, we can prove the identity directly:

$$\begin{aligned}
&\left(\sum_{j=0}^{m-1} \alpha_j \chi_j\right)\left(\sum_{k=0}^{m-1} \beta_k \chi_k\right) \\
&= \left(\sum_{j,s=0}^{m-1} \alpha_j \omega^{sj} x^s\right)\left(\sum_{k,t=0}^{m-1} \beta_k \omega^{tk} x^t\right) \\
&= \sum_{i=0}^{m-1} x^i \left[\sum_{j,k=0}^{m-1} \sum_{s=0}^{m-1} \alpha_j \omega^{sj} \beta_k \omega^{(i-s)k}\right] \\
&= \sum_{i=0}^{m-1} x^i \left[\sum_{j,k=0}^{m-1} \alpha_j \beta_k \omega^{ik} \sum_{s=0}^{m-1} \omega^{s(j-k)}\right].
\end{aligned}$$

The point now is that when $j = k$, we have $\omega^{s(j-k)} = 0$, and so $\sum_{s=0}^{m-1} \omega^{s(j-k)} = m$. When $j \neq k$, we have $\omega^{j-k} \neq 1$ (since ω is a *primitive* mth root), and so we can use the formula for the sum of a geometric series to deduce that

$$\sum_{s=0}^{m-1} \omega^{s(j-k)} = \frac{\omega^{m(j-k)} - 1}{\omega^{j-k} - 1} = 0,$$

since $\omega^m = 1$. The sum above is thus equal to

$$\sum_{i=0}^{m-1} x^i \left[\sum_{j=0}^{m-1} m \alpha_j \beta_j \omega^{ij}\right] = \sum_{j=0}^{m-1} m \alpha_j \beta_j \sum_{i=0}^{m-1} \omega^{ij} x^i = \sum_{j=0}^{m-1} m \alpha_j \beta_j \chi_j.$$

If we have an element of $\mathbb{C}[\mathbb{Z}_m]$ expressed in the basis $\chi_0, \ldots, \chi_{m-1}$, we can compute its expression in the usual basis using the definition of χ_j:

$$\sum_{j=0}^{m-1} \alpha_j \chi_j = \sum_{j=0}^{m-1} \alpha_j \sum_{i=0}^{m-1} \omega^{ij} x^i = \sum_{i=0}^{m-1} x^i \left[\sum_{j=0}^{m-1} \alpha_j \omega^{ij}\right].$$

(This is usually known as the *Fourier inversion formula*.) Going in the other way is known as the *Fourier transform*:

$$\sum_{i=0}^{m-1} a_i x^i = \sum_{j=0}^{m-1} \chi_j \left[\frac{1}{m} \sum_{i=0}^{m-1} \alpha_i \omega^{-ij}\right].$$

We can check the validity of this formula by evaluating the right-hand side:

$$\begin{aligned}
\sum_{j=0}^{m-1} \chi_j \left[\frac{1}{m}\sum_{i=0}^{m-1} \alpha_i \omega^{-ij}\right] &= \sum_{j=0}^{m-1}\sum_{k=0}^{m-1} \omega^{jk} x^k \left[\frac{1}{m}\sum_{i=0}^{m-1} \alpha_i \omega^{-ij}\right] \\
&= \sum_{k=0}^{m-1} x^k \left[\frac{1}{m}\sum_{i,j=0}^{m-1} \alpha_i \omega^{j(k-i)}\right] \\
&= \sum_{k=0}^{m-1} x^k \sum_{i=0}^{m-1} \alpha_i \left[\frac{1}{m}\sum_{j=0}^{m-1} \omega^{j(k-i)}\right] \\
&= \sum_{k=0}^{m-1} x^k \alpha_k,
\end{aligned}$$

just as before.

All of the above suggests the following algorithm for polynomial multiplication:

- Express P as a linear combination $P = \sum_{j=0}^{m-1} \alpha_j \chi_j$.
- Express Q as a linear combination $Q = \sum_{j=0}^{m-1} \beta_j \chi_j$.
- Compute $R = PQ = \sum_{j=0}^{m-1} m\alpha_j\beta_j\chi_j$.
- Convert R to a linear combination of the basis x^i.

In the first two steps we are computing Fourier transforms, and the last step computes an inverse Fourier transform. The formulas for both are very similar (they differ by replacing ω with ω^{-1}, another primitive mth root of unity), so they have the same complexity. How fast can we compute the inverse Fourier transform?

1.1 A more traditional view

Before turning to the Fast Fourier Transform, let us propose a different exposition of the foregoing. We can think of elements of $\mathbb{C}[\mathbb{Z}_m]$ as functions from $\mathbb{Z}_m$ to $\mathbb{C}$. For example, the basis $\chi_0, \ldots, \chi_{m-1}$ corresponds to the functions

$$\chi_j(i) = \omega^{ij}.$$

These functions are *homomorphisms* from $\mathbb{Z}_m$ to $\mathbb{C}^\times$, that is, $\chi_j(i_1 + i_2) = \chi_j(i_1)\chi_j(i_2)$. In fact, they are *all* such homomorphisms. Another important property they satisfy is $\overline{\chi_j(i)} = \chi_j(-i)$.

We define an inner product on $\mathbb{C}[\mathbb{Z}_m]$ by the formula

$$\langle f, g\rangle = \frac{1}{m}\sum_{i=0}^{m-1} f(i)\overline{g(i)}.$$

Under this inner product, the basis $\chi_0, \ldots, \chi_{m-1}$ is orthonormal:

$$\langle \chi_j, \chi_k\rangle = \frac{1}{m}\sum_{i=0}^{m-1} \omega^{i(j-k)} = \begin{cases} 1 & \text{if } j = k, \\ 0 & \text{otherwise,} \end{cases}$$

a calculation we have seen several times. This property allows us to deduce the Fourier transform formula almost immediately: if $f = \sum_{j=0}^{m-1} \alpha_j \chi_j$ then

$$\langle f, \chi_j\rangle = \sum_{i=0}^{m-1} \alpha_i \langle \chi_i, \chi_j\rangle = \alpha_j.$$

We recover the formula

$$\alpha_j = \langle f, \chi_j \rangle = \sum_{i=0}^{m-1} f(i)\overline{\chi_j(i)} = \sum_{i=0}^{m-1} f(i)\omega^{-ij}.$$

In the literature, α_j is often denoted $\hat{f}(j)$.

What about convolution? Suppose that $f = \sum_{j=0}^{m-1} \alpha_j \chi_j$, $g = \sum_{j=0}^{m-1} \beta_j \chi_j$, and $h = f * g$ is given by

$$h(i) = \sum_{r=0}^{m-1} f(r)g(i-r),$$

which is the same as our polynomial multiplication. To find the Fourier expansion of h (its expansion in the basis $\chi_0, \ldots, \chi_{m-1}$), we first consider the case in which $f = \chi_j$ and $g = \chi_k$. In that case

$$(\chi_j * \chi_k)(i) = \sum_{r=0}^{m-1} \chi_j(r)\chi_k(i-r) = \chi_k(i) \sum_{r=0}^{m-1} \chi_j(r)\overline{\chi_k(r)} = m\chi_k(i)\langle \chi_j, \chi_k \rangle.$$

We conclude that $\chi_j * \chi_k = 0$ if $j \neq k$, and $\chi_j * \chi_j = m$. Linearity then implies the convolution formula

$$h = \sum_{j=0}^{m-1} m\alpha_j\beta_j\chi_j.$$

2 Fast Fourier Transform

It turns out that when $m = 2^M$, the Fourier transform and its inverse can be computed very quickly, in time $O(m \log m)$. The corresponding algorithm is known as the *Fast Fourier Transform*.

Recall that the inverse Fourier transform asks us to compute the coefficients $p_0, \ldots, p_{m-1}$ given the coefficients $\alpha_0, \ldots, \alpha_{m-1}$, using the formula

$$p_i = \sum_{j=0}^{2^M-1} \alpha_j \omega^{ij}.$$

The basic idea is to break the sum into two parts. There are two natural ways of doing it: according to LSB and according to MSB (these are known as decimation-in-time and decimation-in-frequency, respectively). We choose breaking according to the LSB:

$$p_i = \sum_{j=0}^{2^{M-1}-1} \left(\alpha_{2j}\omega^{2ij} + \alpha_{2j+1}\omega^{2ij+i}\right) = \sum_{j=0}^{2^{M-1}-1} \alpha_{2j}(\omega^2)^{ij} + \omega^i \sum_{j=0}^{2^{M-1}-1} \alpha_{2j+1}(\omega^2)^{ij}.$$

The expressions in the two sums are the same ones as the inverse Fourier transform for $\mathbb{Z}_{m/2}$! There is a slight complication: the inverse Fourier transform for $\mathbb{Z}_{m/2}$ has $i < 2^{M-1}$, but here the indices i could be as large as $2^M - 1$. This is no big deal, however, since $(\omega^2)^{2^{M-1}} = 1$, and so $(\omega^2)^{ij} = (\omega^2)^{(i \bmod 2^{M-1})j}$. With this correction in place, we can explain the FFT algorithm:

- Compute (recursively) the inverse Fourier transform $s_0, \ldots, s_{m/2-1}$ of $\alpha_0, \alpha_2, \ldots, \alpha_{m-2}$.
- Compute (recursively) the inverse Fourier transform $t_0, \ldots, t_{m/2-1}$ of $\alpha_1, \alpha_3, \ldots, \alpha_{m-1}$.
- For $i = 0, \ldots, m-1$, compute

$$p_i = s_{i \bmod (m/2)} + \omega^i t_{i \bmod (m/2)}.$$

The base of the recursion is when $m = 1$, in which case the formula is just $p_0 = \alpha_0$. The number of arithmetic operations performed, as a function of m, satisfies the recurrence

$$T(m) = 2T(m/2) + \Theta(m),$$

whose solution is $T(m) = \Theta(m \log m)$.

The FFT in hand, we can calculate the running time of the polynomial multiplication algorithm described above. We take $m = 2^{\lceil \log_2(2n+1) \rceil}$, which satisfies $2n+1 \le m \le 4n+2$. Given polynomials P, Q, we compute their Fourier transforms in time $O(m \log m) = O(n \log n)$. We then compute the Fourier transform of $R = PQ$ in linear time $O(m)$ by multiplying the *Fourier coefficients* (the coefficients of the *characters* χ_j). Finally, we compute the inverse Fourier transform of R in time $O(m \log m) = O(n \log n)$. The entire algorithm takes time $O(n \log n)$.

We can multiply multivariate polynomials using multidimensional Fourier transforms. For example, we can multiply bivariate polynomials using the Fourier transform in the group $\mathbb{Z}_m^2$. An extreme case is the *Walsh transform*, which is the Fourier transform in $\mathbb{Z}_2^n$, useful in theoretical computer science. Multidimensional Fourier transforms are computed essentially by computing the Fourier transform across each dimension in sequence. The generalization to arbitrary (non-abelian) groups is the subject of representation theory.

2.1 In-place algorithms

What happens if we want to implement FFT in place? To see what happens, let us work out the algorithms for small m. We will use the notation ω_m for $e^{2\pi i/m}$, which is a primitive mth root of unity.

When $m = 2$, the original algorithm is:

- $s_0 \leftarrow \alpha_0$.
- $t_0 \leftarrow \alpha_1$.
- $p_0 \leftarrow s_0 + \omega_2^0 t_0 = \alpha_0 + \alpha_1$.
- $p_1 \leftarrow s_0 + \omega_2^1 t_0 = \alpha_0 - \alpha_1$.

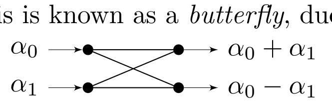
If we want to implement this algorithm in-place, we need to execute the last two lines simultaneously:

$$\alpha_0, \alpha_1 \leftarrow \alpha_0 + \alpha_1, \alpha_0 - \alpha_1.$$

We can also express this as a matrix-vector multiplication:

$$\begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix}.$$

This is known as a *butterfly*, due to its diagrammatic shape:



When $m = 4$, the algorithm is:

- $s_0, s_1 \leftarrow \alpha_0 + \alpha_2, \alpha_0 - \alpha_2$.
- $t_0, t_1 \leftarrow \alpha_1 + \alpha_3, \alpha_1 - \alpha_3$.
- $p_0 \leftarrow s_0 + \omega_4^0 t_0$.
- $p_1 \leftarrow s_1 + \omega_4^1 t_1$.
- $p_2 \leftarrow s_0 + \omega_4^2 t_0$.

- $p_3 \leftarrow s_1 + \omega_4^3 t_1$.

If we want to implement this algorithm in-place, we first need to perform the first two lines in-place:

- $\alpha_0, \alpha_2 \leftarrow \alpha_0 + \alpha_2, \alpha_0 - \alpha_2$.
- $\alpha_1, \alpha_3 \leftarrow \alpha_1 + \alpha_3, \alpha_1 - \alpha_3$.
- $p_0 \leftarrow \alpha_0 + \omega_4^0 \alpha_1$.
- $p_1 \leftarrow \alpha_2 + \omega_4^1 \alpha_3$.
- $p_2 \leftarrow \alpha_0 + \omega_4^2 \alpha_1$.
- $p_3 \leftarrow \alpha_2 + \omega_4^3 \alpha_3$.

To implement the second half in-place, we need to perform pairs of lines simultaneously:

- $\alpha_0, \alpha_2 \leftarrow \alpha_0 + \alpha_2, \alpha_0 - \alpha_2$.
- $\alpha_1, \alpha_3 \leftarrow \alpha_1 + \alpha_3, \alpha_1 - \alpha_3$.
- $\alpha_0, \alpha_1 \leftarrow \alpha_0 + \omega_4^0 \alpha_1, \alpha_0 - \omega_4^0 \alpha_1$.
- $\alpha_2, \alpha_3 \leftarrow \alpha_2 + \omega_4^1 \alpha_3, \alpha_2 - \omega_4^1 \alpha_3$.

(We used $\omega_4^2 = -1$.) If we look at the correspondence between the entries of p and the entries of α, we see something strange:

$$p_0, p_1, p_2, p_3 = \alpha_0, \alpha_2, \alpha_1, \alpha_3.$$

The two middle values got switched! This phenomenon is known as *bit reversal*.

Let us take this one step further, and consider the case $m = 8$. The original algorithm is:

- Compute the transform of $\alpha_0, \alpha_2, \alpha_4, \alpha_6$, and put the results in s_0, s_1, s_2, s_3.
- Compute the transform of $\alpha_1, \alpha_3, \alpha_5, \alpha_7$, and put the results in t_0, t_1, t_2, t_3.
- For $i = 0, 1, 2, 3$, compute $p_i, p_{i+4} = s_i + \omega_8^i t_i, s_i - \omega_8^i t_i$.

(We used $\omega_8^4 = -1$.) When performing the first part in-place, we get:

- Compute the transform of $\alpha_0, \alpha_2, \alpha_4, \alpha_6$ in-place, obtaining s_0, s_2, s_1, s_3.
- Compute the transform of $\alpha_1, \alpha_3, \alpha_5, \alpha_7$ in-place, obtaining t_0, t_2, t_1, t_3.
- For $i = 0, 1, 2, 3$, compute $p_i, p_{i+4} = s_i + \omega_8^i t_i, s_i - \omega_8^i t_i$.

Implementing the second part in-place, we get:

- Compute the transform of $\alpha_0, \alpha_2, \alpha_4, \alpha_6$ in-place.
- Compute the transform of $\alpha_1, \alpha_3, \alpha_5, \alpha_7$ in-place.
- $\alpha_0, \alpha_1 \leftarrow \alpha_0 + \omega_8^0 \alpha_1, \alpha_0 - \omega_8^0 \alpha_1$.
- $\alpha_2, \alpha_3 \leftarrow \alpha_2 + \omega_8^2 \alpha_3, \alpha_2 - \omega_8^2 \alpha_3$.
- $\alpha_4, \alpha_5 \leftarrow \alpha_4 + \omega_8^1 \alpha_5, \alpha_4 - \omega_8^1 \alpha_5$.
- $\alpha_6, \alpha_7 \leftarrow \alpha_6 + \omega_8^3 \alpha_7, \alpha_6 - \omega_8^3 \alpha_7$.

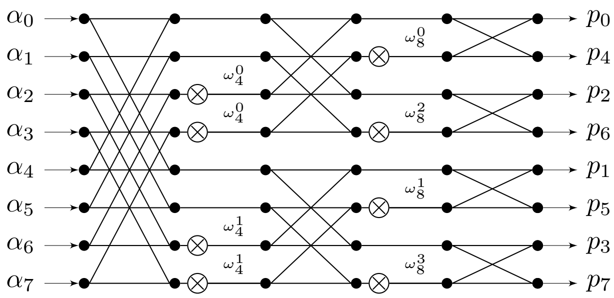
The correspondence this time is:

$$p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7 = \alpha_0, \alpha_4, \alpha_2, \alpha_6, \alpha_1, \alpha_5, \alpha_3, \alpha_7.$$

The bit-reversal phenomenon is more apparent if we use binary indices:

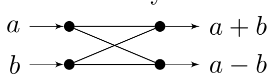
$$\begin{array}{cccccccc} p_{000} & p_{001} & p_{010} & p_{011} & p_{100} & p_{101} & p_{110} & p_{111} \\ \alpha_{000} & \alpha_{100} & \alpha_{010} & \alpha_{110} & \alpha_{001} & \alpha_{101} & \alpha_{011} & \alpha_{111}. \end{array}$$

Here is a diagram describing the entire algorithm:



Multiplication nodes $\otimes$ multiply the wire by the stated constant.

Each butterfly has the same semantics as before:



We can now describe the general in-place FFT algorithm on $m = 2^M$ points, using the notation $\mathrm{br}_M(x)$ for the bit reversal of the M-bit index x:

- Compute the in-place FFT of the 2^{M-1} points α_{2i} (where $0 \le i < 2^{M-1}$).
- Compute the in-place FFT of the 2^{M-1} points α_{2i+1} (where $0 \le i < 2^{M-1}$).
- For $0 \le i < 2^{M-1}$, let $I = \mathrm{br}_M(i)$, and compute

$$\begin{bmatrix} \alpha_{2i} \\ \alpha_{2i+1} \end{bmatrix} = \begin{bmatrix} 1 & \omega_m^I \\ 1 & -\omega_m^I. \end{bmatrix} \begin{bmatrix} \alpha_{2i} \\ \alpha_{2i+1} \end{bmatrix}.$$

- For $0 \le i < 2^M$, let $I = \mathrm{br}_m(i)$, and if $i < I$ then exchange α_i and α_I.

The base case, $m = 1$ or $M = 0$, is the trivial "do-nothing" algorithm.

We can also insert the input bit-reversed, changing the twiddle factors (the factors ω_m^i) accordingly, and then the output will be in the correct order.

2.2 Number-theoretic transform

The Fast Fourier transform, as we have described it, requires floating point computations when implemented on a real computer. This invariably introduces calculation errors, and so the answer is not exact. We can avoid this by replacing $\mathbb{C}$ with another ring which contains mth roots of unity. The ring has to contain the coefficients of the polynomials P, Q. Assuming that these coefficients are small integers, we can choose the ring $\mathbb{Z}_{2^m-1}$, which contains a primitive mth root of unity, namely 2. This ring has to be large enough so that the coefficients of the product polynomial R are at most 2^{m-1} (or $2^m - 1$ if they are non-negative), as this will allow their decoding at the end of the process. To this end we might want to choose a ring $\mathbb{Z}_{2^{km}-1}$ for a large enough k, which also contains a primitive mth root of unity, namely 2^k. In practice, $2^{km} + 1$ is better than $2^{km} - 1$, since it allows several optimizations. The primitive mth roots of unity are 4 and 4^k, respectively.

During the FFT algorithm, we have to multiply by powers of ω. Since for us ω is a power of 2 (both in the initial step and in the recursive steps), we can implement this by the fast operation of *bit rotation* (or using shifts). When counting only bit operations, bit rotations are for free. Therefore if we are using the ring $2^{km} \pm 1$, computing the direct and inverse Fourier transforms takes $\Theta(km^2 \log m)$ bit operations (since each addition takes $\Theta(km)$ bit operations).

3 Schönhage–Strassen algorithm

The idea of the Schönhage–Strassen algorithm is to express integer multiplication as polynomial multiplication. Suppose that $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$ are two n-bit non-negative integers. Then

$$ab = \left(\sum_{i=0}^{n-1} a_i x^i \right) \left(\sum_{i=0}^{n-1} b_i x^i \right) \Bigg|_{x=2}.$$

Using the number theoretic transform with $m = 2^{\lceil \log_2(2n+1) \rceil} = \Theta(n)$, we can multiply the polynomials in time $\Theta(m^2 \log m) = \Theta(n^2 \log n)$, which is worse than the trivial algorithm. The point of difficulty is that FFT uses an mth root of unity, and so we need to work in a ring containing an mth root of unity with a simple form, and such rings are large. To contravene this difficulty, we will partition a, b into groups of ℓ bits:

$$ab = \left(\sum_{i=0}^{n/\ell-1} \sum_{j=0}^{\ell-1} 2^j a_{\ell i+j} x^i \right) \left(\sum_{i=0}^{n/\ell-1} \sum_{j=0}^{\ell-1} 2^j b_{\ell i+j} x^i \right) \Bigg|_{x=2^\ell}.$$

We need to work over a ring $\mathbb{Z}_{2^{k(2n/\ell+1)} \pm 1}$ with $2^{k(2n/\ell+1)} > (n/\ell + 1)2^{\ell+1}$, so we choose the smallest k such that $k(2n/\ell + 1) > \log(n/\ell + 1) + \ell + 1$. Our choice of ℓ will be larger than $\log n$, so we will have $k(2n/\ell + 1) = \Theta(\ell)$. The FFT algorithm thus takes time $\Theta(k(n/\ell) \cdot (n/\ell) \log(n/\ell)) = \Theta(n \log(n/\ell))$. Having computed the Fourier transforms, we need to multiply the Fourier coefficients. These are n/ℓ products of coefficients of length $\Theta(\ell)$ bits. After running FFT again to compute the inverse transform and converting the Fourier coefficients to integers, we are left with computing

$$\sum_{i=0}^{2n/\ell} c_i 2^{\ell i},$$

where each c_i has length $\ell + \log n$ bits. Each of these staggered additions takes time $\Theta(\ell + \log n)$, for a total of $\Theta(n + (n/\ell) \log n)$. The total running time thus satisfies the recurrence

$$T(n) = (n/\ell) T(\Theta(\ell)) + \Theta(n \log(n/\ell)).$$

How should we choose ℓ? Let us assume that $\Theta(\ell) = B\ell$ for some $B > 1$, and ignore the other Θ. Opening up the recursion, we obtain

$$\begin{aligned} T(n) &= n \log \frac{n}{\ell_1} + \frac{n}{\ell_1} \cdot B\ell_1 \log \frac{B\ell_1}{\ell_2} + \frac{n}{\ell_1} \cdot \frac{B\ell_1}{\ell_2} \cdot B\ell_2 \log \frac{B\ell_2}{\ell_3} + \cdots \\ &= n \left(\log \frac{n}{\ell_1} + B \log \frac{B\ell_1}{\ell_2} + B^2 \log \frac{B\ell_2}{\ell_3} + \cdots \right). \end{aligned}$$

Finding the best choice for ℓ_i is somewhat tiresome. A good choice is $\ell_i = n^{(1/B)^i}$, and substituting this we obtain

$$\begin{aligned} T(n) &= n \left(\log n^{1-1/B} + B \log(Bn^{1/B-1/B^2}) + B^2 \log(Bn^{1/B^2-1/B^3}) + \cdots \right) \\ &= n \left((1 - 1/B) \log n (1 + B/B + B^2/B^2 + \cdots) + (\log B)(1 + B + B^2 + \cdots) \right). \end{aligned}$$

The dots continue until ℓ_i becomes constant, which happens for $B^i \approx \log n$, that is, at $i = \Theta(\log \log n)$. The overall complexity is thus $\Theta(n \log n \log \log n)$.