Optimizing the computation time of the product of two “large” numbers

Alexis Lecordier

30/01/26 – 06/02/26

Abstract

Multiplying two numbers is an elementary task, but its cost quickly becomes very high when their size increases. This article analyzes the complexity limits of naive methods, then presents a more efficient method and its improvement.

1. The problem

Let us consider two real numbers A and B that can be represented computationally, i.e. decimals with a finite expansion. There then exist two integers $a, b \in \mathbb{Z}$ and two natural integers $e, f \in \mathbb{N}$ such that

$$A = a \times 10^{-e}, \qquad B = b \times 10^{-f}.$$

It follows immediately that

$$AB = (a \times b) \times 10^{-(e+f)}.$$

The values a, b, e and f are those stored in memory.

Thus, computing AB amounts to multiplying two integers a and b, then adding the exponents e and f to determine the final power of 10.

Evaluating the cost of computing AB therefore requires studying both the complexity of an integer multiplication algorithm and that of adding two natural integers.

2. Elementary methods for addition and multiplication

2.1 Addition of two natural integers

Let m and p be two natural integers written in base 10 and represented in memory as lists of digits. Let n be the maximum number of digits of m and p. These lists are reversed so that the least significant digits appear in the first position.

Addition is then performed by a simultaneous traversal, consisting in adding the corresponding digits term by term while propagating successive carries.

Finally, the resulting list is reversed to recover the usual decimal order, then converted back into a string representing the integer $m + p$. Since each digit is processed only once, the algorithmic complexity of addition is linear, on the order of $O(n)$.

We introduce the notation M_n to denote the algorithmic cost of multiplying two (signed) integers having at most n digits in base 10. According to the reductions above, computing the product of two finite-decimal numbers (of size at most n) reduces to an addition and an integer multiplication, hence a total cost of the form

$$O(n) + O(M_n).$$

It therefore remains to determine M_n for the **elementary multiplication method**.

2.2 The "elementary" multiplication of two signed integers

Let m and p be two signed integers. We begin by isolating the sign :

$$m = \sigma_m |m|, \qquad p = \sigma_p |p|, \qquad \sigma_m, \sigma_p \in \{-1, +1\}.$$

The sign of the product is then $\sigma_m \sigma_p$, which is a constant-time operation $O(1)$. Thus, the cost study reduces to the multiplication of two natural integers $|m|$ and $|p|$.

We then represent $|m|$ and $|p|$ in base 10 as lists of digits, as in the previous subsection, and reverse these lists so that the least significant digits appear in the first position. We then write

$$\widetilde{L}_m = [a_0, a_1, \ldots, a_{n-1}], \qquad \widetilde{L}_p = [b_0, b_1, \ldots, b_{n-1}],$$

where n denotes the maximum number of digits (possibly after completion with zeros). We therefore have

$$|m| = \sum_{i=0}^{n-1} a_i \, 10^i, \qquad |p| = \sum_{j=0}^{n-1} b_j \, 10^j.$$

Multiplication as a discrete convolution. Expanding, we obtain

$$|m| \, |p| = \left(\sum_{i=0}^{n-1} a_i \, 10^i \right) \left(\sum_{j=0}^{n-1} b_j \, 10^j \right) = \sum_{k=0}^{2n-2} \left(\sum_{i+j=k} a_i b_j \right) 10^k.$$

We define the *discrete convolution* of the sequences $a = (a_i)_{i=0}^{n-1}$ and $b = (b_j)_{j=0}^{n-1}$ by

$$(a * b)_k = \sum_{\substack{i+j=k \\ 0 \le i \le n-1 \\ 0 \le j \le n-1}} a_i \, b_j, \qquad k = 0, \ldots, 2n-2.$$

Setting

$$c_k = (a * b)_k,$$

we obtain a compact expression for the product :

$$|m|\,|p| = \sum_{k=0}^{2n-2} c_k\, 10^k.$$

Renormalization (carries) and link with the "schoolbook" method. The coefficients c_k are not necessarily digits (they may exceed 9). We therefore perform a renormalization in base 10 : we traverse $k = 0, 1, \ldots$ while propagating carries. For each rank k, we write

$$c_k + r_k = d_k + 10\, r_{k+1}, \qquad d_k \in \{0, \ldots, 9\},$$

where r_k is the incoming carry (with $r_0 = 0$), d_k is the final digit at rank 10^k, and r_{k+1} the outgoing carry. This step formalizes exactly the multiplication learned at school : the products $a_i b_j$ are first grouped by rank 10^{i+j} (which corresponds to the convolution), then carries are performed to obtain a valid decimal expansion.

After renormalization, we obtain a list of digits $(d_0, d_1, \ldots)$ such that

$$|m|\,|p| = \sum_{k} d_k\, 10^k.$$

Finally, we apply the sign $\sigma_m \sigma_p$.

Elementary convolution and complexity. The convolution $c = a * b$ can be computed naively by two nested loops :

```
def convolution(L1, L2):
    L = [0] * (len(L1) + len(L2) - 1)
    for i in range(len(L1)):
        for j in range(len(L2)):
            L[i+j] += L1[i] * L2[j]
    return L
```

If $|L1| \sim n$ and $|L2| \sim n$, this computation performs on the order of n^2 elementary operations. Thus, for the "elementary" method, the multiplication of two n-digit integers satisfies

$$M_n = O(n^2).$$

At this stage, the total cost of computing the product (for finite-decimal numbers of size at most n) is dominated by elementary multiplication, and is therefore on the order of $O(n^2)$. In other words, the problem essentially reduces to studying the complexity of an algorithm that computes the convolution of two digit lists. In the sequel, we present an algorithm that makes it possible to replace the naive $O(n^2)$ convolution by a method in $O(n \ln n)$.

3. The Fast-Fourier-Transform (FFT) algorithm of Cooley and Tukey

To understand the FFT algorithm, one must first know what a discrete Fourier transform is, and relate it to the computation of convolutions.

3.1 Definitions and properties of the discrete Fourier transform

We work in the very general setting of (complex) finite-support sequences.
Let $\mathcal{S}_f$ denote the set of sequences $x = (x_k)_{k\in\mathbb{Z}}$ with values in $\mathbb{C}$ and finite support

Definition 3.1

Fix $N \in \mathbb{N}^*$. The *discrete Fourier transform* (DFT) of size N is the linear map

$$\mathcal{F}_N : \mathbb{C}^N \longrightarrow \mathbb{C}^N$$

defined by

$$(\mathcal{F}_N(x))_k = \sum_{j=0}^{N-1} x_j\, \omega_N^{jk}, \qquad k = 0, 1, \ldots, N-1.$$

If $x \in \mathcal{S}_f$ is supported in $\{0, \ldots, N-1\}$, we then define $\mathcal{F}_N(x)$ by identification with its vector $(x_0, \ldots, x_{N-1})$.

Remark 3.1

Polynomial link (evaluation at roots of unity). To $x = (x_0, \ldots, x_{N-1})$, associate the polynomial

$$P_x(X) = \sum_{j=0}^{N-1} x_j X^j.$$

Then the previous definition can be written simply as :

$$(\mathcal{F}_N(x))_k = P_x(\omega_N^k).$$

Thus, computing $\mathcal{F}_N(x)$ amounts to evaluating P_x at the N points $\omega_N^0, \ldots, \omega_N^{N-1}$.

Proposition 3.1

Proposition (convolution by DFT after "zero-padding"). Let $a, b \in \mathcal{S}_f$ be two finite-support sequences, and $c = a*b$ their convolution. Assume that $\mathrm{supp}(a) \subset \{0, \ldots, n-1\}$ and $\mathrm{supp}(b) \subset \{0, \ldots, m-1\}$, so that $\mathrm{supp}(c) \subset \{0, \ldots, n+m-2\}$. Fix an integer $N \geq n+m-1$, and consider the zero-extensions ("zero-padding")

$$a^{(N)} = (a_0, \ldots, a_{N-1}) \in \mathbb{C}^N, \qquad b^{(N)} = (b_0, \ldots, b_{N-1}) \in \mathbb{C}^N,$$

(where $a_k = 0$ for $k \geq n$ and $b_k = 0$ for $k \geq m$). Then, still denoting by $c^{(N)} \in \mathbb{C}^N$ the

zero-extension of c, we have the relation

$$\mathcal{F}_N(c^{(N)}) = \mathcal{F}_N(a^{(N)}) \ \odot \ \mathcal{F}_N(b^{(N)}),$$

where $\odot$ denotes pointwise multiplication.

Proof (polynomial argument). Associate to $a^{(N)}$ and $b^{(N)}$ the polynomials

$$P_a(X) = \sum_{i=0}^{N-1} a_i X^i, \qquad P_b(X) = \sum_{j=0}^{N-1} b_j X^j.$$

Expanding,

$$P_a(X)P_b(X) = \sum_{k=0}^{2N-2} \left(\sum_{i+j=k} a_i b_j \right) X^k.$$

For $k \leq n+m-2$, the terms with $i \geq n$ or $j \geq m$ are zero, so the coefficient of X^k is

$$\sum_{i+j=k} a_i b_j \ = \ c_k.$$

Thus, setting $P_c(X) = \sum_{k=0}^{n+m-2} c_k X^k$, we indeed have

$$P_c(X) = P_a(X)P_b(X) \qquad \text{(at the level of coefficients up to degree } n+m-2).$$

Evaluating at $X = \omega_N^r$ ($\omega_N = e^{-2\pi i/N}$), for all $r = 0, \ldots, N-1$, we obtain

$$P_c(\omega_N^r) = P_a(\omega_N^r)\, P_b(\omega_N^r).$$

But $(\mathcal{F}_N(a^{(N)}))_r = P_a(\omega_N^r)$, $(\mathcal{F}_N(b^{(N)}))_r = P_b(\omega_N^r)$ and $(\mathcal{F}_N(c^{(N)}))_r = P_c(\omega_N^r)$, hence

$$(\mathcal{F}_N(c^{(N)}))_r = (\mathcal{F}_N(a^{(N)}))_r\, (\mathcal{F}_N(b^{(N)}))_r,$$

which is equivalent to

$$\mathcal{F}_N(c^{(N)}) = \mathcal{F}_N(a^{(N)}) \ \odot \ \mathcal{F}_N(b^{(N)}).$$

Inverse transform. The map $\mathcal{F}_N$ is invertible, and its inverse is naturally given by :

$$(\mathcal{F}_N^{-1}(X))_j \ = \ \frac{1}{N} \sum_{k=0}^{N-1} X_k\, \omega_N^{-jk}, \qquad j = 0, 1, \ldots, N-1.$$

We deduce the computation formula :

$$(a * b)^{(N)} = \mathcal{F}_N^{-1}\Big(\mathcal{F}_N(a^{(N)}) \odot \mathcal{F}_N(b^{(N)})\Big)$$

We note here that to obtain the convolution of a and b, it suffices :
(i) to compute $\mathcal{F}_N(a^{(N)})$ and $\mathcal{F}_N(b^{(N)})$;
(ii) to perform their **pointwise product** (of complexity $O(N)$) ;

(iii) to apply the **inverse transform** $\mathcal{F}_N^{-1}$.

Thus, if we have an algorithm of complexity $O(N \ln N)$ to compute $\mathcal{F}_N(s)$ and $\mathcal{F}_N^{-1}(s)$ (where s is a finite-support sequence of size N), then the convolution computation has complexity :

$$\underbrace{O(N \ln N)}_{\mathcal{F}_N(a^{(N)})} + \underbrace{O(N \ln N)}_{\mathcal{F}_N(b^{(N)})} + \underbrace{O(N)}_{\text{pointwise product}} + \underbrace{O(N \ln N)}_{\mathcal{F}_N^{-1}} = O(N \ln N).$$

3.2 Principle of the algorithm

Let $a = (a_0, \ldots, a_{N-1}) \in \mathbb{C}^N$. By Remark 3.1, computing the discrete Fourier transform

$$\mathcal{F}_N(a) = \big((\mathcal{F}_N(a))_k\big)_{0 \le k \le N-1}$$

amounts to evaluating the polynomial

$$P_a(X) = \sum_{j=0}^{N-1} a_j X^j$$

at the N N-th roots of unity :

$$(\mathcal{F}_N(a))_k \;=\; P_a(\omega_N^k), \qquad k = 0, \ldots, N-1, \quad \text{where } \omega_N = e^{-2\pi i/N}.$$

A naive evaluation of P_a at one point costs $O(N)$, so evaluation at N points costs $O(N^2)$.

Cooley–Tukey's idea is to exploit a recursive structure when N is a power of 2. We therefore assume $N = 2^s$ (if needed, we *extend by zeros* the sequence a up to the next power of 2, which amounts to completing P_a with zero coefficients).

We then write the *even/odd* decomposition of P_a :

$$P_a(X) = P_{\text{pair}}(X^2) + X\, P_{\text{impair}}(X^2),$$

where

$$P_{\text{pair}}(Y) = \sum_{r=0}^{N/2-1} a_{2r} Y^r, \qquad P_{\text{impair}}(Y) = \sum_{r=0}^{N/2-1} a_{2r+1} Y^r.$$

We want to evaluate P_a at $X = \omega_N^k$. But $(\omega_N^k)^2 = \omega_{N/2}^k$, so

$$P_a(\omega_N^k) = P_{\text{pair}}(\omega_{N/2}^k) + \omega_N^k\, P_{\text{impair}}(\omega_{N/2}^k).$$

Thus, evaluating P_a on the N N-th roots reduces to evaluating two polynomials P_{pair} and P_{impair} on the $N/2$ $(N/2)$-th roots.

Moreover, for any d-th root of unity ω_d, we have the relation

$$\omega_d^{i+d/2} = -\omega_d^i, \qquad i = 0, \ldots, d/2 - 1.$$

In particular, setting $d = N$, we obtain for $k = 0, \ldots, N/2 - 1$:

$$\omega_N^{k+N/2} = -\omega_N^k.$$

We can then compute simultaneously the two values

$$P_a(\omega_N^k) \quad \text{and} \quad P_a(\omega_N^{k+N/2})$$

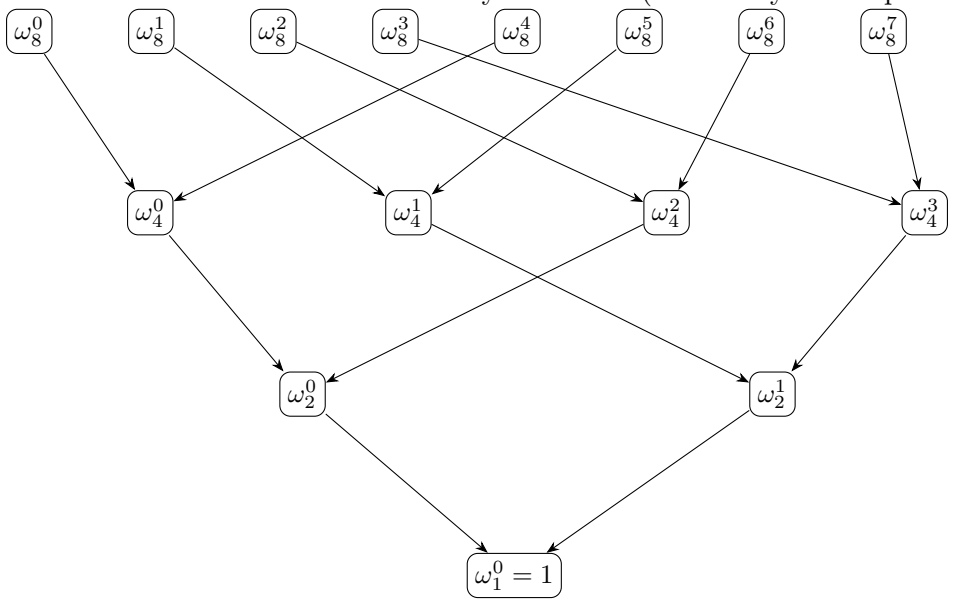
from the same quantities $P_{\text{pair}}(\omega_{N/2}^k)$ and $P_{\text{impair}}(\omega_{N/2}^k)$:

$$\begin{aligned} P_a(\omega_N^k) &= P_{\text{pair}}(\omega_{N/2}^k) + \omega_N^k \, P_{\text{impair}}(\omega_{N/2}^k), \\ P_a(\omega_N^{k+N/2}) &= P_{\text{pair}}(\omega_{N/2}^k) - \omega_N^k \, P_{\text{impair}}(\omega_{N/2}^k). \end{aligned}$$

We then iterate exactly the same procedure on P_{pair} and P_{impair} (separating again their even/odd coefficients), which produces a recursive decomposition until reaching degree-0 polynomials (trivial evaluation). This recursion corresponds to a tree whose level t manipulates evaluations on the 2^{s-t}-th roots of unity :

$$U_N \longrightarrow U_{N/2} \longrightarrow \cdots \longrightarrow U_1 = \{1\}.$$

FIGURE 1 – Reduction tree of roots of unity for $N = 8$ (descent by the map $x \mapsto x^2$).



Algorithm (Python) :

Listing 1 – Recursive implementation of the FFT (Cooley–Tukey)

```
def FFT(L):
    n = len(L)

    if n == 1:
        return L

```

```
    w = np.exp(2j * np.pi / n)

    L_pair = L[0::2]
    L_impair = L[1::2]
    y_pair, y_impair = FFT(L_pair), FFT(L_impair)
    y = [0] * n

    for i in range(n // 2):
        y[i] = y_pair[i] + (w ** i) * y_impair[i]
        y[i + n // 2] = y_pair[i] - (w ** i) * y_impair[i]

    return y
```

3.3 Complexity computation (counting by levels)

We assume $n = 2^s$ (after *zero-padding* if necessary). The recursion tree then has

$$s = \left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor = \log_2(n)$$

levels.

Descent (even/odd split). At level k (with $1 \leq k \leq s$), we manipulate sub-lists of size $n/2^k$. The total cost of the descent is bounded above by

$$\sum_{k=1}^{s} \left(\frac{n}{2^k} + 1 \right),$$

where $\frac{n}{2^k}$ corresponds to the splitting work (building the even/odd sub-lists), and 1 to the computation of the root w associated with the considered level.

Ascent The ascent goes through the same s levels. At level k, we reconstruct a list of size $n/2^{\,s-k}$. We therefore perform $\frac{n}{2^{\,s-k}}$ *butterfly* combinations, each containing two lines :

$$y[i] = y_{\text{pair}}[i] + (w^i)\, y_{\text{impair}}[i], \qquad y[i + n/2] = y_{\text{pair}}[i] - (w^i)\, y_{\text{impair}}[i].$$

For one line, the term 1 corresponds to the addition/subtraction, while the term $\text{cplx}\big((w^i)\, y_{\text{impair}}[i]\big)$ corresponds to computing the product. Thus, the cost of the ascent is bounded above by

$$\sum_{k=1}^{s} \left(\frac{n}{2^{\,s-k+1}} \cdot 2\Big(1 + \text{cplx}\big((w^i)\, y_{\text{impair}}[i]\big)\Big) \right).$$

Cost of $(w^i)\, y_{\text{impair}}[i]$ and implementation choice. In our case, we have already computed and stored the root w associated with the considered size.

Consequently, computing the term $(w^i)\, y_{\text{impair}}[i]$ boils down to an exponentiation of w (which is an addition of arguments) and a multiplication by an integer $(y_{\text{impair}}[i])$, thus we can consider that its complexity is on the order of

$$\text{cplx}\big((w^i)\, y_{\text{impair}}[i]\big) = O(1).$$

Summary. By combining descent and ascent, we obtain the upper bound

$$\boxed{\sum_{k=1}^{s}\left(\frac{n}{2^k}+1\right)+\sum_{k=1}^{s}\left(\frac{n}{2^{\,s-k+1}}\cdot 2\Big(1+O(1)\Big)\right)}$$

with $s=\log_2(n)$. Since $\sum_{k=1}^{s}\frac{n}{2^k}=2n$ and $\sum_{k=1}^{s}\frac{n}{2^{\,s-k+1}}=2n$, we deduce that the total complexity is on the order of

$$O(n\log_2 n).$$

We thus obtain that computing the convolution of a and b via FFT decomposes into :

$$\underbrace{O(n\ln n)}_{\text{FFT of } a}+\underbrace{O(n\ln n)}_{\text{FFT of } b}+\underbrace{O(n)}_{\text{pointwise product}}+\underbrace{O(n\ln n)}_{\text{inverse FFT}}=O(n\ln n).$$

Conclusion

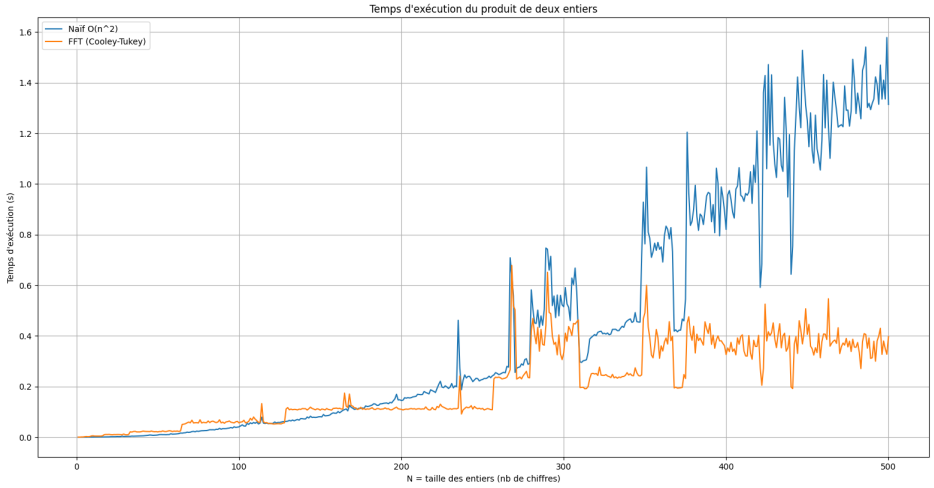


FIGURE 2 – Product of two integers : comparison of execution times between the naive method and the FFT (Cooley–Tukey).

The orange curve represents the execution time of our FFT-based algorithm, while the blue curve corresponds to the naive $O(n^2)$ method. We observe that from about $N > 300$, the FFT approach becomes more advantageous.