Computer Sciene 2: Advanced Algorithms Fall 2026

Lecture 2 — September, 1, 2026

Prof. Jarosław Błasiok *Scribe: Francesco Agrimonti*

1 Overview

In the last lecture we learned about the problem of *polynomial identity testing* (PIT), which consists of determining whether an arithmetic formula that is given in input, representing a polynomial $p(x_1, \ldots, x_n) \in F[x_1, \ldots, x_n]$ where F is a field, represents the zero polynomial of $F[x_1, \ldots, x_n]$ or not. After observing that a naive approach to tackle this problem (namely expanding the input formula as a sum of monomials and checking whether all coefficients are 0) requires a non-negligible running time, we turned to one of the sharpest tools in theoretical computer science: *randomness*. In particular, the DeMillo-Lipton-Schwartz-Zippel lemma was proved, which essentially states that a subset $S \subseteq F$ can be chosen of size large enough to ensure that the probability that a non-zero polynomial p vanishes on n elements $X_1, \ldots, X_n$, chosen independently and uniformly at random from S, is not greater than, say, $\frac{1}{2}$. If p is not the zero polynomial, then, as the amount k of independent samples of n elements increases, the probability that p *always* vanishes on all of them decreases as 2^{-k}. Once this is clear, a randomized algorithm with one-sided error for the PIT naturally comes to mind: perform k independent samples of $X_1, \ldots, X_n$, and output with full confidence NonZero if p doesn't vanish *for at least one* of the k independent samples, otherwise output Zero knowing that an error could occur with a probability that is upper bounded by 2^{-k}.

Moreover, two applications of this randomized one-sided error algorithm for PIT were described: a PIT-based algorithm was designed to check whether a *bipartite* graph contains a perfect matching or not, and a second one for checking whether a *general* graph does. The interest behind these applications (besides their obvious aesthetic value) is driven by how *simple* their implementations are, especially when compared to the much more sophisticated deterministic state-of-the-art alternatives. In fact, it was proved that checking whether a $2n$-vertex bipartite graph contains a perfect matching amounts to design a matrix T having its (i, j)-entry containing a variable x_{ij} if the vertices i and j are joined by an edge and 0 otherwise, and checking whether its determinant, which actually is a polynomial, is the zero polynomial or not (and this is precisely where the randomized one-sided algorithm for PIT comes into play): if it is, then the graph doesn't contain any perfect matching, otherwise it does. That's it. When the input graph is not necessarily bipartite, the situation is slightly more nuanced, but an algorithm can be designed that shares the same spirit: a suitable matrix (different from that of the bipartite case) can be defined such that its determinant is the zero polynomial if and only if the graph doesn't contain any perfect matching.

In conclusion: what we learned shows that if we tolerate some uncertainty on the correctness of the output, then some concrete problems can actually be tackled efficiently through very simple-to-define algorithms, and these are based on tools that would probably not seem related to the problem at hand (at least) at a first glance.

In this lecture we will carry on the leitmotiv of this course of efficiently solving computational problems with techniques that do not even seem related to the problems at all, and we will get pretty

carried away in doing so. In fact, we will end up showing how the fast Fourier transform (FFT) algorithm can be employed in solving an ubiquitous computational problem: *integer multiplication*. Whichever notion of *basic* we may have in our mind, it is very likely that integer multiplication is one of the most basic computational problems we can imagine. When facing computational problems that are *this kind* of basic, one could be tempted to think that the most simple, out-of-the-box algorithms would also be pretty much the most efficient ones: after all, one could think, the problem is so basic that maybe it doesn't even *leave room* to do something non-trivial. Well, put very simply: this is wrong. We will show pretty strong evidence that this mindset is in general misleading.

In fact, first we will see how far we expect simple algorithms for Integer Multiplication can go by computing the running time of the grade-school multiplication algorithm (which shouldn't fall short of being considered basic); then, we will show how increasingly more sophisticated techniques allow to obtain running times that asymptotically beat that of the grade-school algorithm by an increasingly larger margin. Interestingly, we will show that no extremely sophisticated technique is needed to *simply beat* the grade-school algorithm: Karatsuba's one [1] will be based on a clever "rearrangement trick" that exploits the fact that a product can be performed in time $O(n)$ once *only* 3 suitable products have been computed (and not 4, as one could think to be necessary at a first glance). This approach also exploits the fact that the cost of sums is rather cheap in the (pretty realistic) word-RAM computational model, upon which our analyses will be based. After this, we will show arguably the core technique of the lecture: integers can be encoded into polynomials in such a way that multiplying two integers amounts to multiplying their two corresponding encoding polynomials. This will lead to a further discussion: is it possible to multiply two polynomials more efficiently, with respect to the trivial method?

Given that multiplying integers can be reduced to multiplying polynomials, we will see how algorithmic improvements for solving the latter problem imply algorithmic improvements also for solving the former one. This will lead us to understand that there exist at least *two* forms in which any single polynomial can represented: one of them, called *coefficient form*, will be useful for *framing* the Integer Multiplication problem we are interested in (and efficiently performing additions), while the other one, called *value form*, will turn out to be very effective for computing the value of the product of two polynomials in specific, and carefully chosen a priori, *evaluation points*. The good news will be that there exists a correspondence between these two forms, but the bad news will be that it won't be immediately clear how to exploit it (i.e. switch between coefficient and value forms as needed) efficiently. It will turn out that the fast Fourier transform (FFT) algorithm designed by Cooley and Tukey [2] (some similar ideas had already been exploited by Gauss [3] previously), allows to switch both from coefficient to value form (*evaluation*) and from value to coefficient form (*interpolation*) with almost the *exact same technique* and *very efficiently*.

This very successful application will lead us to talk a little bit more deeply about Fourier transforms in general, and we will end up considering $\mathbb{C}$-valued functions defined on a finite cyclic group. We will observe that this set (which forms a vector space) can be equipped with a scalar product that allows to define a very specific orthonormal basis; this orthonormal basis will be a nice tool to define extremely useful notions such as those of Fourier coefficients and Fourier expansions. These considerations will set the perfect ground to define the notion of *cyclic convolution* between two functions of the space: we will discover a posteriori that, when exploiting the FFT in designing an efficient algorithm for Integer Multiplication, we will have had actually performed a convolution in disguise between polynomials, without knowing it.

Finally, we will mention some specific examples of phenomena (intended from both pure and applied perspectives) that can actually be captured by convolutions: from sums of independent random variables, to random walks to isolating single signals from a mixture of them.

2 Preliminaries

Before we stick to the plan, let us be clear on some details so that we will avoid any ambiguity on how the problem is defined and how the efficiency of the algorithms is evaluated.

2.1 Expansions of Numbers w.r.t. Bases

To study the multiplication of two integers, we will actually study the multiplication of their respective *expansions* with respect to one *same*, *fixed* base β.

Definition 1. *Let $N, n, \beta \in \mathbb{N}$. The n-**digit expansion with respect to the base** β of the number N is* the *sum $\sum_{k=0}^{n-1} c_k \beta^k$ such that:*

$$N = \sum_{k=0}^{n-1} c_k \beta^k$$

where $c_k \in \{0, \ldots, \beta - 1\}$ for every $k \in \{0, \ldots, n-1\}$.

In the previous definition, the article "*the*" preceding the term "sum" presumes that for any $N \in \mathbb{N}$ an expansion as defined exists and that it is actually unique once n is large enough and β has been fixed (and these facts should be provable without too much effort).

Example. For the sake of completeness, the number $N = 8204$ can be expressed with respect to the base $\beta = 10$ by the $n = 4$-digit expansion:

$$8204 = \sum_{k=0}^{3} c_k 10^k$$

where $c_0 = 4$, $c_1 = 0$, $c_2 = 2$ and $c_3 = 8$.

Notice that we defined the notion of expansion with respect to a base only for *natural* numbers, and not for all *integers*: the reason is that handling the sign of the integers would make the text unnecessarily heavier, because any sign-careful version of the algorithms will have the same asymptotic properties, and because the presence of sign is not that difficult to handle. So, in short: even though we will refer to integers we will actually restrict to natural numbers, knowing that asymptotically identical results could be proved for integers (with a bit of care).

2.2 Computational Model

To evaluate the running time of the algorithms, we will employ a word-RAM-like computation model, with the following features:

1. the two input strings representing the respective two numbers to be multiplied, call them $A, B \in \mathbb{N}$, are assumed to be their n-digits expansion w.r.t. a fixed base β, that is:

$$A = \sum_{i=0}^{n-1} a_i \beta^i \quad \text{and} \quad B = \sum_{j=0}^{n-1} b_j \beta^j$$

for suitable $a_0, \ldots, a_{n-1}$ and $b_0, \ldots, b_{n-1}$ all belonging to $\{0, \ldots, \beta - 1\}$;

2. numbers are expressed as *arrays* of *words* of $\log n$ digits each, and arithmetic operations on these words are assumed to have an $O(1)$ cost.

Some "bit-level" nuances concerning word arithmetic and numerical precision have to be considered when translating results on this word-RAM-like model to a Turing machine. They will be highlighted when considered relevant.

Finally, in the rest of the text we will always assume that the two input numbers are called A and B, and that their corresponding input strings are determined by their respective unique n-digits representations with respect to a fixed base β, that we will always denote with:

$$A := \sum_{i=0}^{n-1} a_i \beta^i \quad \text{and} \quad B := \sum_{j=0}^{n-1} b_j \beta^j$$

3 Algorithms for Integer Multiplication: Some First Examples

Let us finally proceed with the descriptions of the algorithms for the Integer Multiplication problem. A general guiding question that we could keep in mind as a sort of benchmark (which will make us better appreciate the results that are coming) is:

Question: *Does there exist an algorithm that solves Integer Multiplication in time $\Theta(n)$? If it is not the case, how near can we get?*

Notice that this $\Theta(n)$ benchmark is not that arbitrary: it comes from the (rather fair) assumption that an algorithm is supposed to at least *read* the whole input strings before performing any operation, and this requires time $\Theta(n)$.

Let us proceed with the grade-school multiplication algorithm.

3.1 Grade-School Multiplication Algorithm

We all know how the grade-school multiplication algorithm works and it should not be too difficult to formalize it, so we avoid to do so (writing its pseudocode could seem a bit of an overkill). For our purposes, instead, it is sufficient to observe that the following formula highlights the operations that are performed by it:

$$AB = \sum_{k=0}^{2n-2} \left(\sum_{i+j=k} a_i b_j \right) \beta^k$$

In fact, all the possible products of the kind $a_i b_j$ (where i and j range among $0, \ldots, n-1$) are performed, which already yields an $O(1)$ cost for each one of the n^2 products, which implies that the running time is bounded from below by $\Omega(n^2)$. The other necessary steps, like handling carries and adding partial products, can be checked to require a cost of $O(n^2)$ too. This implies that the running time is bounded also from above by $O(n^2)$. These considerations can be summarized in the following proposition.

Proposition 2. *The running time T_{school} of the grade-school algorithm for the multiplication of two n-digits numbers is $\Theta(n^2)$, i.e.:*

$$T_{school}(n) = \Theta(n^2)$$

As premised, we will consider this running time:

$$T_{\text{school}}(n) = \Theta(n^2)$$

the *baseline* from which we will find algorithms with better running times, aiming to get as near as possible to the natural *barrier* of $\Theta(n)$.

3.2 Karatsuba's Algorithm

The first non-trivial algorithm we are going to present is Karatsuba's one [1]. It is probably fair to say that it is impressive to see how large of an improvement can be given by an elementary, let us call it, *trick* like the one we are going to see.

Karatsuba's algorithm is based on the *divide and conquer* technique, whose idea we briefly recall.

3.2.1 Divide and Conquer

The high-level idea behind this algorithm-design technique is that, sometimes, a generic instance of a problem can be *decomposed* in multiple instances of the exact same problem, but of *smaller size* (call these *sub-instances* of the original instance), in such a way that:

1. the sub-instances of the problem are solvable *much faster* if their sizes are small enough;
2. a solution for the original instance can be easily found by combining together the solutions of the sub-instances, *once* they have all been computed.

In order to get to sub-instances whose sizes are small enough that they can be solved efficiently, one can utilize *recursion*: for example, if one is able to decompose the instance I of size s of a given problem in, say, two sub-instances I_1 and I_2 both of size $\frac{s}{2}$, then they can decompose also I_1 into two (sub-)sub-instances I_{11} and I_{12} of size $\frac{s}{4}$, and I_2 into other two (sub-)sub-instances I_{21} and I_{22} of size $\frac{s}{4}$, and so on until the (sub-sub-...-)sub-instances are of size so small that they can be solved efficiently enough.

Let us see the divide and conquer approach in action in the actual definition of Karatsuba's algorithm. We will first show that a straightforward application of the divide and conquer approach doesn't lead to any improvement w.r.t. the running time of the grade-school algorithm, and this will set the ground for introducing (and appreciating) Karatsuba's technique.

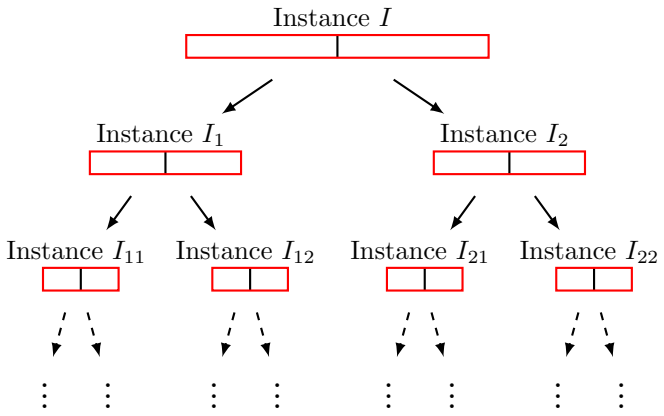


Figure 1: A very primitive scheme of a divide and conquer algorithm in which every (non-leaf) instance is decomposed into 2 sub-instances.

3.2.2 Design of The Algorithm: A First (Wrong) Direction

First of all, we can safely assume that n is *even*, because otherwise we can simply work with the $2n$-digits expansions of A and B (which for A would simply be $\sum_{i=0}^{2n-1} a_i\beta^i$, with $a_n = \ldots = a_{2n-1} = 0$, instead of $\sum_{i=0}^{n-1} a_i\beta^i$, while for B would simply be $\sum_{i=0}^{2n-1} b_i\beta^i$, with $b_n = \ldots = b_{2n-1} = 0$, instead of $\sum_{i=0}^{n-1} b_i\beta^i$). Since n is even, then $\frac{n}{2}$ is an integer, which we denote with $m := \frac{n}{2}$.

Observe that the expansions of A and B can be rewritten in a very convenient way. In fact, if we call:

$$a'_i := a_{i+m} \quad \forall i \in \{0, \ldots, m-1\}$$

it holds that:

$$\begin{aligned} A &= \sum_{i=0}^{n-1} a_i\beta^i \\ &= \sum_{i=0}^{m-1} a_i\beta^i + \sum_{i=m}^{n-1} a_i\beta^i \\ &= \sum_{i=0}^{m-1} a_i\beta^i + \beta^m \sum_{i=0}^{m-1} a_{i+m}\beta^i \\ &= \sum_{i=0}^{m-1} a_i\beta^i + \beta^m \sum_{i=0}^{m-1} a_i'\beta^i \\ &= A_0 + \beta^m A_1 \end{aligned} \tag{1}$$

where A_0 and A_1 are two m-digits expansions of some numbers (these are respectively the expansions corresponding to the m *least significant digits* and the m *most significant* ones). In a completely analogous way it can be proved that also for B it holds that:

$$B = B_0 + \beta^m B_1 \tag{2}$$

where, again, B_0 and B_1 are two m-digits expansions of some numbers. We can exploit the expres-

sions in 1 and 2 to find a suggestive form of the product AB:

$$\begin{aligned} AB &\overset{1\,2}{=} (A_0 + \beta^m A_1)(B_0 + \beta^m B_1) \\ &= A_0B_0 + \beta^m A_0B_1 + \beta^m A_1B_0 + \beta^{2m} A_1B_1 \\ &= A_0B_0 + \beta^m(A_0B_1 + A_1B_0) + \beta^{2m} A_1B_1 \end{aligned} \tag{3}$$

Now that we have reduced computing the product of two n-digits expansions (namely AB) to computing 4 products of $m = \frac{n}{2}$-digits expansions (namely A_0B_0, A_0B_1, A_1B_0 and A_1B_1), one could think of a straightforward application of the divide and conquer method:

1. recursively decompose the instance of the problem into 4 sub-instances (as of 3) until you cannot decompose any other sub-instance further (all sub-instances have size 1);
2. solve the instances of size 1 with a direct computation, and utilize them to compute the solution.

Could this algorithm beat the $\Theta(n^2)$ barrier? No, unfortunately. In fact, calling $T_{\text{wrong}}(n)$ the running time of this self-admittedly doomed approach, remembering that additions can be performed in time $O(n)$, we get that:

$$\begin{aligned} T_{\text{wrong}}(n) &= 4 \cdot T_{\text{wrong}}\left(\frac{n}{2}\right) + O(n) \\ &= 4 \cdot 4 \cdot T_{\text{wrong}}\left(\frac{n}{4}\right) + O(n) + O\left(4 \cdot \frac{n}{2}\right) \\ &= \dots \\ &= \Theta(4^{\log n}) + O(n^2) \\ &= \Theta(2^{2\log n}) + O(n^2) \\ &= \Theta(n^2) \end{aligned}$$

which is (asymptotically speaking) precisely the running time of the grade-school algorithm: no improvement whatsoever. (P.S.: even if seeing written $O\left(4 \cdot \frac{n}{2}\right)$ instead of $O(2n)$ aesthetically hurts, it hopefully made more clear the reason why the term $O(n^2)$ appeared in the end).

3.2.3 Karatsuba's (Right) Direction

There is a precise point of the analysis of the previous approach suggesting that an important factor that prevented us from beating the $\Theta(n^2)$ barrier (with the divide and conquer approach) was the *number of sub-instances* into which a generic instance was decomposed. In fact, as it was clear from the expression of T_{wrong}, they were 4; maybe it is not a coincidence that we got to the bound of $\Theta(n^2)$ precisely through the equalities:

$$\dots = \Theta(4^{\log n}) = \Theta(2^{2\log n}) = \Theta(n^2) \tag{4}$$

Where are these insinuations leading to? To the fact that if we were ever able to design a divide and conquer algorithm in which an instance of the problem could be decomposed in *strictly less than* 4 half-sized sub-instances (say 3 sub-instances), then we would be able to almost replicate the

analysis of the previous approach, but (this time) with a very different conclusion w.r.t. 4, as it would hold that:

$$\ldots = \Theta(3^{\log n}) = \Theta((2^{\log 3})^{\log n}) = \Theta((2^{\log n})^{\log 3}) = \Theta(n^{1.585})$$

yielding a significant improvement on the $\Theta(n^2)$ barrier. So, we have an explicit strategy in mind:

Find some way to decompose every instance in strictly less than 4 *sub-instances*

And here is precisely where Karatsuba's idea sets in: Rearrange the expression of the product AB in such a way that, at the (very much acceptable) cost of computing a few extra additions in each sub-instance, *only* 3 *products have to be computed*, instead of 4.

How to do this? First, recall that we expressed A and B as:

$$A = A_0 + \beta^m A_1 \quad \text{and} \quad B = B_0 + \beta^m B_1$$

We have that:

$$\begin{aligned} AB &= (A_0 + \beta^m A_1)(B_0 + \beta^m B_1) \\ &= A_0B_0 + \beta^m A_0B_1 + \beta^m A_1B_0 + \beta^{2m} A_1B_1 \\ &= A_0B_0 + \beta^m (A_0B_1 + A_1B_0) + \beta^{2m} A_1B_1 \\ &= A_0B_0 + \beta^m (A_0B_1 + A_1B_0 + A_0B_0 - A_0B_0 + A_1B_1 - A_1B_1) + \beta^{2m} A_1B_1 \\ &= A_0B_0 + \beta^m \left[(A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1\right] + \beta^{2m} A_1B_1 \end{aligned}$$

and if we define $P_0 := A_0B_0$, $P_2 := A_1B_1$ and $P_1 := (A_0 + A_1)(B_0 + B_1) - P_0 - P_2$, then AB can be expressed as:

$$AB = P_0 + \beta^m P_1 + \beta^{2m} P_2$$

This is amazing news, because it means that *only* 3 products have to be computed. In fact, even though P_0 and P_2 are actual products but P_1 is not, *once the values* P_0 *and* P_2 *have been computed*, P_1 can be computed by performing:

- the addition $A_0 + A_1$;
- the addition $B_0 + B_1$;
- **the product** $(A_0 + A_1)(B_0 + B_1)$;
- the subtraction of A_0B_0
- the subtraction of A_1B_1

So *only one product*, also for P_1. Now that the strategy has set in, we are able to almost repeat

the analysis for the first (wrong) direction, to get that:

$$\begin{aligned}
T_{\text{Karatsuba}}(n) &= 3 \cdot T_{\text{Karatsuba}}\left(\frac{n}{2}\right) + O(n) \\
&= 3 \cdot 3 \cdot T_{\text{Karatsuba}}\left(\frac{n}{4}\right) + O(n) + O\left(3 \cdot \frac{n}{2}\right) \\
&= \dots \\
&= \Theta(3^{\log n}) + O(n^{\log 3}) \\
&= \Theta((2^{\log 3})^{\log n}) + O(n^{\log 3}) \\
&= \Theta((2^{\log n})^{\log 3}) + O(n^{\log 3}) \\
&= \Theta(n^{\log 3}) + O(n^{\log 3}) \\
&= \Theta(n^{1.585})
\end{aligned}$$

(Apologies again for the $O\left(3 \cdot \frac{n}{2}\right)$). What we proved can be synthesized in the following theorem.

Theorem 3. *Let $T_{Karatsuba}(n)$ be the running time of Karatsuba's algorithm. It holds that:*

$$T_{Karatsuba} = \Theta(n^{1.585})$$

Now that Karatsuba's technique allowed us to beat the $O(n^2)$ barrier, it is probably time to see where we can get with more sophisticated tools.

4 Integer Multiplication Through Polynomials

A great way to think about the Integer Multiplication problem is to think about another problem that turns out to be a generalization of it: *Polynomial Multiplication*. First of all, we recall a formula for the product $C(x)$ of two (formal) polynomials $A(x)$ and $B(x)$ of degree $n-1$ on a (formal) variable x. If we call:

$$A(x) := \sum_{i=0}^{n-1} a_i x^i \quad \text{and} \quad B(x) := \sum_{j=0}^{n-1} b_j x^j$$

their *product polynomial* $C(x)$ (or $A(x)B(x)$) has the following form:

$$C(x) := \sum_{k=0}^{2n-2} \left(\sum_{i+j=k} a_i b_j \right) x^k$$

which, calling $c_k := \sum_{i+j=k} a_i b_j$ for every $k = 0, \dots, 2n-2$, can be also expressed more concisely as:

$$C(x) = \sum_{k=0}^{2n-2} c_k x^k$$

It shouldn't be too difficult to see that, also in this case, computing the product between two formal polynomials in a straightforward manner requires $O(n^2)$ time. As premised, the Polynomial Multiplication problem is a generalization of the Integer Multiplication one, in the sense of the following proposition.

Proposition 4. *The Polynomial Multiplication problem reduces to the Integer Multiplication one, up to handling the $O(n)$ cost of carry operations. More formally, consider the running time $T_{poly}(n)$ of a specific algorithm for multiplying two polynomials of degree $n-1$. Then, there exists an algorithm $\mathcal{A}$ for multiplying two numbers, having n-digits expansion w.r.t. to a fixed base β, whose running time is:*

$$T_{\mathcal{A}}(n) = O(T_{poly}(n)) + O(n) \tag{5}$$

Proof. The algorithm $\mathcal{A}$ behaves in the following way:

1. gets the n-digits expansions $\sum_{i=0}^{n-1} a_i\beta^i$ of A and $\sum_{j=0}^{n-1} b_j\beta^j$ of B;
2. defines two corresponding (formal) polynomials $A(x) := \sum_{i=0}^{n-1} a_i x^i$ and $B(x) := \sum_{j=0}^{n-1} b_j x^j$;
3. computes the product $C(x) = A(x)B(x)$;
4. evaluates $C(\beta) = \sum_{k=0}^{2n-2} c_k\beta^k$;
5. performs the carry operations on the coefficients $(c_k)_{k=0}^{2n-2}$, ensuring that the unique $(2n-2)$-digits expansion of $C(\beta)$ w.r.t. the base β is obtained;
6. outputs this proper, newly obtained expansion of $C(\beta)$.

The correctness of the algorithm is a direct consequence of the fact that AB (the product of the two *integers*) coincides with $A(\beta)B(\beta)$ (the product of the two *evaluations in β of formal polynomials*), and of the fact that the carry operations guarantee that what is returned in output is a proper expansion of $C(\beta)$.

In terms of running time, points 3 and 5 of the algorithm are those that require some considerations. First, point 3 is the responsible of the occurrence of $T_{\text{poly}}(n)$ in equality 5 in the statement: any improvement in the efficiency of multiplying two polynomials implies an improvement in the efficiency of multiplying two integers (at least up to $\Omega(n)$, given the cost of the carry operations). Moreover, the fact that carry operations can be performed in time $O(n)$ is a consequence of the fact that, although the various c_ks are not guaranteed to be valid digits of an expansion (i.e., nothing guarantees that $c_k \in \{0, \dots, \beta-1\}$ for every $k \in \{0, \dots, 2n-2\}$), at least they are guaranteed to be suitably bounded:

$$0 \leq c_k \leq n(\beta-1)^2 \quad \forall k \in \{0, \dots, 2n-2\}$$

and thus, since the base β is fixed, this guarantees in turn that every c_k requires $O(\log n)$ to be represented. Remembering that arithmetic operations on words of $\log n$ length are performed in $O(1)$ time, we conclude that performing the carry operations on the $O(n)$ coefficients requires $O(n)$. □

The moral behind this proposition is clear: if you were ever able to compute the product of *polynomials* efficiently, then you would be able to compute also the product of *integers* efficiently (up to the $\Omega(n)$ cost required by the carry operations, as usual).

4.1 Two Useful Representations of Polynomials

Now that we have highlighted how useful an efficient way to multiply two polynomials would be for our goal, let us reason a little bit on how polynomials can be *represented*, and more in particular if and how some particular representations of them make some operations more efficient. Let us consider a generic degree $D-1$ polynomial:

$$A(x) := \sum_{i=0}^{D-1} a_i x^i \in F[x]$$

where F is some field (this is actually the setting we will restrict ourselves to work in). A natural way to represent $A(x)$ is clearly through the vector of its coefficients $(a_0, \ldots, a_{D-1})$. This takes the name of *coefficient form* of the polynomial $A(x)$.

Definition 5. *Let F be any field. The **coefficient form** of a $D-1$ degree polynomial:*

$$A(x) := \sum_{i=0}^{D-1} a_i x^i \in F[x]$$

consists of the vector of its coefficients $\mathbf{c} := (a_0, \ldots, a_{D-1})$.

However, a less trivial question is whether a polynomial can be identified by how it *behaves* on the elements of F, i.e. by which are the *values* it assumes when it is evaluated on specific elements of F. The answer to this question, under suitable conditions, is *yes*. Let us give a definition that is functional to prove this.

Definition 6. *Let F be any field, $x_0, \ldots, x_{D-1} \in F$ be elements of F, and $A(x) \in F[x]$ be a degree $D-1$ polynomial in F. The vector:*

$$\mathbf{v} := (A(x_0), \ldots, A(x_{D-1})) \in F^D$$

*is called the **value form** of $A(x)$ with respect to the points $x_0, \ldots, x_{D-1}$.*

So, reformulated in new terms, the question becomes:

Do there exist D points of F such that it is possible to identify the polynomial $A(x)$ by its value form with respect to these D points?

To prove this under suitable conditions (spoiler: that the D points are all distinct), it will be sufficient to prove that there exist D elements of F, call them again $x_0, \ldots, x_{D-1}$, such that there exists a *bijective* correspondence between coefficient forms $\mathbf{c}$ of degree $D-1$ polynomials and value forms $\mathbf{v} \in F^D$ of degree $D-1$ polynomials w.r.t. the points $x_0, \ldots, x_{D-1}$. More specifically we will prove two points:

1. if a coefficient form $(a_0, \ldots, a_{D-1})$ of a polynomial $A(x)$ of degree $D-1$ is given, then there exists a *unique* value form $\mathbf{v} \in F^D$ of $A(x)$ w.r.t. the points $x_0, \ldots, x_{D-1}$.

   In this case, the value form $\mathbf{v}$ is also said to **evaluate** the polynomial $A(x)$ in the D points;

2. if a vector $\mathbf{v} := (v_0, \ldots, v_{D-1}) \in F^D$ is given, then there exists a *unique* coefficient form $\mathbf{c} := (a_0, \ldots, a_{D-1})$ inducing a polynomial $A(x) := \sum_{i=0}^{D-1} a_i x^i$ such that $\mathbf{v}$ is the value form of $A(x)$ w.r.t. the points $x_0, \ldots, x_{D-1}$.

   In this case, the polynomial $A(x)$ induced by the coefficients of the coefficient form $\mathbf{c}$ is said to **interpolate** the values of the value form $\mathbf{v}$ w.r.t. the points $x_0, \ldots, x_{D-1}$.

Once we will have proved these two points, we will have proved the correspondence between coefficient forms and value forms:

$$\text{coefficient forms} \iff \text{value forms}$$

But since we are already aware of the canonical correspondence between polynomials and coefficient forms:

$$\text{polynomials} \iff \text{coefficient forms}$$

then we will have obtained an indirect correspondence between polynomials and value forms *via* coefficient forms:

$$\text{polynomials} \iff \text{coefficient forms} \iff \text{value forms}$$

Or, in other words, we will have proved that, once D suitable points of F are fixed, *evaluations and interpolations are each others' inverses*. This is our goal. So, let us focus now on proving the previous two points.

4.1.1 First Point: Evaluation

The proof of the first point is a matter of definitions more than anything. In fact, given a coefficient form of a polynomial (thus the polynomial itself) $A(x) \in F[x]$, then no matter which points $x_0, \ldots, x_{D-1} \in F$ are chosen: there exists one, and *only* one, vector $\mathbf{v}$ that is a value form of $A(x)$ w.r.t. $x_0, \ldots, x_{D-1}$. The existence is certified by the fact that the vector:

$$(A(x_0), \ldots, A(x_{D-1})) \tag{6}$$

is a value form of $A(x)$ w.r.t. $x_0, \ldots, x_{D-1}$, while the uniqueness is guaranteed by the fact that if by contradiction there existed a value form of $A(x)$ w.r.t. $x_0, \ldots, x_{D-1}$ different (on at least one entry $\tilde{i} \in \{0, \ldots, D-1\}$) from that of 6, then, by the very definition of value form, it would amount to say that $A(x_{\tilde{i}}) \neq A(x_{\tilde{i}})$, clearly a contradiction.

4.1.2 Second Point: Interpolation

The second point is slightly more involved to deal with.

Lemma 7 (Interpolation Lemma)**.** *Let F be any field, $x_0, \ldots, x_{D-1}$ be* distinct *elements of F and $y_0, \ldots, y_{D-1}$ be any other D elements of F. Then, there exists a polynomial $p(x) \in F[x]$ of degree $D-1$ such that:*

$$p(x_i) = y_i \quad \forall i \in \{0, \ldots, D-1\}$$

and it is unique*.*

Proof. We first prove the existence of such a polynomial $p(x)$. In order to do this, we define the following *Lagrange basis* of polynomials:

$$L_i(x) := \prod_{\substack{0 \leq j \leq D-1 \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

which is properly defined since F is a field (thus $(x_i - x_j)^{-1}$ is properly defined) and since the denominator $x_i - x_j$ of any factor cannot vanish for any $i \neq j$ (as $x_0, \ldots, x_{D-1}$ are all distinct by hypothesis). Any factor of the kind:

$$\frac{x - x_j}{x_i - x_j}$$

vanishes if $x = x_j$, while it is equal to 1 if, instead, $x = x_i$, implying that for every $i, j \in \{0, \ldots, D-1\}$ it holds that:

$$L_i(x_j) = \delta_{ij}$$

where δ_{ij} denotes the usual Kronecker's delta. So, if we define a new polynomial $p(x)$ as:

$$p(x) := \sum_{i=0}^{D-1} y_i L_i(x)$$

we observe that $p(x)$ cannot have degree strictly larger than $D-1$, and that for every $i \in \{0, \ldots, D-1\}$ it holds that:

$$p(x_i) = y_i$$

which completes the proof of the existence part.

Let us prove uniqueness now. Suppose there exists another polynomial $q(x) \in F[x]$ such that $p(x_i) = q(x_i)$ for every $i \in \{0, \ldots, D-1\}$, and define a new polynomial $r(x) := p(x) - q(x) \in F[x]$. Then, the degree of $r(x)$ cannot be strictly larger than $D-1$, moreover for every $i \in \{0, \ldots, D-1\}$ it holds that:

$$r(x_i) = p(x_i) - q(x_i) = 0$$

which means that $r(x)$, a polynomial of degree $D-1$, has at least D roots. This is not possible for a non-zero polynomial, thus $r(x)$ must be the zero polynomial; this means that $p(x)$ and $q(x)$ coincide in the whole F, which concludes the proof of uniqueness. □

4.1.3 Summary

We synthesize the results obtained in this section in the following theorem.

Theorem 8. *Let F be any field, and $x_0, \ldots, x_{D-1} \in F$ be any D* distinct *elements of F. Then, there exists a* bijective correspondence *between* coefficient forms *of polynomials of degrees up to $D-1$ and* value forms *of these polynomials w.r.t. the elements $x_0, \ldots, x_{D-1}$.*

Now it is finally time to show why we are interested in this correspondence.

4.2 Why This Correspondence?

As we know, having *two* different representations of one same mathematical object can be very helpful. In fact, for example, one kind of representation could have the advantage of being the right tool for *framing* a computational problem (regarding the mathematical object it represents), but maybe it could have the disadvantage of not being suitable for efficiently performing computations that we *need* to perform efficiently. On the other hand, maybe the other representation is such that it is more complicated to use in order to frame the problem, but if it became somehow possible, then it would be efficient for performing specific kinds of operations for the problem at hand.

The importance of a nice correspondence that allows to efficiently *switch* from one representations to the other (and back) is that it allows to benefit of *both* the respective advantages of the representations. In fact, one could frame the problem regarding a mathematical object in the representation that allows to do it comfortably, and then switch to the other representation for performing certain computations more efficiently.

If we think about it, this is precisely the situation we find ourselves stuck in. We have already observed that the coefficient form of polynomials is the one in which we are able to reformulate the original problem of Integer Multiplication: however, despite being suitable for additions (having cost $\Theta(n)$), it certainly is not for multiplications (having cost $\Theta(n^2)$).

On the other hand, the value form of polynomials possesses the exact *opposite* features. It is not clear how to reformulate in terms of value forms the Integer Multiplication problem (even *via* the Polynomial Multiplication one), but one thing is for sure: in this form, the product of polynomials in $O(n)$ points can be computed *very* efficiently, as it requires time $O(n)$ (because every product of the kind $C(x_i) = A(x_i)B(x_i)$ for $i \in \{0, \dots, D-1\}$ requires time $O(1)$).

So, this is our plan. Let us simply take the best of both forms by defining the following algorithm, called FFT-*Based Integer Multiplication*.

FFT-Based Integer Multiplication Algorithm

1. take the n-digits expansions of A and B in input;
2. choose $D \geq 2n-1$ *distinct* elements $x_0, \dots, x_{D-1}$ of (a suitable) F;
3. transform the n-digits expansions of A and B into D-digits expansions by zero-padding the missing coefficients;
4. reformulate the Integer Multiplication problem of AB into a Polynomial Multiplication problem through the *coefficient forms* of A and B;
5. *evaluate* the polynomials $A(x)$ and $B(x)$ into the D points, yielding the corresponding value forms $(A(x_0), \dots, A(x_{D-1}))$ and $(B(x_0), \dots, B(x_{D-1}))$;
6. compute the value form $(C(x_0), \dots, C(x_{D-1}))$ of the degree-$D-1$ product polynomial $C(x) = A(x)B(x)$ w.r.t. the D elements $x_0, \dots, x_{D-1}$;
7. compute back the coefficient form of the polynomial $C(x)$ that *interpolates* the values of the value form $(C(x_0), \dots, C(x_{D-1}))$;

8. compute the carry operations to the coefficients of the coefficient form newly obtained;

9. return the D-digits expansion of the resulting vector as the output.

This allows us to actually exploit the fact that multiplications between polynomials can be performed in time $O(n)$ *once* they are expressed in their value forms, as we can see in point 5. However, there is a major flaw in this approach: the *evaluation* and *interpolation* steps (respectively, steps 5 and 7) require a prohibitive time $\Theta(n^2)$ when performed naively. This $\Theta(n^2)$ bottleneck completely *neutralizes* the idea of exploiting the fact that multiplication on $\Theta(n)$ points can be performed in time $O(n)$ (in the realm of value forms).

But luckily not everything is lost. In the next subsections, we will prove that FFT [2] heroically salvages this approach. In particular, we will prove that, thanks to FFT, these two very delicate steps of evaluation and interpolation can be performed in time $O(D \log D) \in O(n \log n)$ (remembering that it is necessary to choose $D = \Theta(n)$ distinct points in F). Putting everything together, this yields a quasi-linear time algorithm for Integer Multiplication that gets significantly close to the $\Theta(n)$ barrier.

4.3 FFT For Efficient Evaluations

In this subsection we want to show that FFT allows us to perform the evaluation step in time $O(n \log n)$. To do this, we fix $\mathbb{C}$ as the field F to work in (even though it is not strictly necessary: a field that *behaves* like $\mathbb{C}$, in a sense that will be clear later, would suffice), and we show that, by choosing D very specific evaluation points, the divide and conquer approach will be applied very successfully. Let us proceed step by step.

Let us assume without loss of generality that D is a power of two (reasons analogous to those why this is not restrictive have already been pointed out previously), and let us *fix* straight from the start which are the D evaluation points: we choose them to be the inverses of the D-th complex roots of unity. More precisely, if we call ζ_D the first D-th root of unity:

$$\zeta_D := e^{\frac{2\pi i}{D}}$$

then the evaluation numbers we choose are all the powers of ζ_D, up to the D-th one, that is:

$$\zeta_D, \dots, \zeta_D^{D-1}, \underbrace{1}_{=\zeta_D^D} \tag{7}$$

remembering that $\zeta_D^D = 1$. It is of course natural to stop at the D-th one, as for any integer $k \in \mathbb{Z}$ it holds that:

$$\zeta_D^{D+k} = \zeta_D^k \tag{8}$$

So, these are actually the evaluation points, but for convenience we will index them through negative exponents, exploiting 8. In short, the set 7 can be equivalently rewritten as:

$$\zeta_D^{1-D}, \dots, \zeta_D^{D-1-D}, \zeta_D^{D-D}$$

that is:

$$\zeta_D^{-(D-1)}, \dots, \zeta_D^{-1}, 1$$

and this latter form is the one in which we require our evaluation numbers to be expressed (which amounts to simply considering the exact same points in the reverse order). We define now some central notions related to the (unnormalized) *discrete Fourier transform* (DFT) on these D evaluation points.

Definition 9. *Let* $\mathbf{a} := (a_0, \ldots, a_{D-1}) \in \mathbb{C}^D$*. For every integer* $k \in \{0, \ldots, D-1\}$*, we call:*

$$\mathcal{F}_D(\mathbf{a})_k := \sum_{j=0}^{D-1} a_j \zeta_D^{-kj}$$

the k*-th* ***Fourier coefficient of the*** DFT *of* $\mathbf{a}$ *on points* $\zeta_D^{-D}, \ldots, \zeta_D^{-1}, 1$*. Moreover, we call the vector:*

$$\mathcal{F}_D(\mathbf{a}) := (\mathcal{F}_D(\mathbf{a})_0, \ldots, \mathcal{F}_D(\mathbf{a})_{D-1})$$

the DFT $\mathcal{F}$ *of* $\mathbf{a}$ *on the points* $\zeta_D^{-D}, \ldots, \zeta_D^{-1}, 1$*.*

Notice that Fourier coefficients directly represent the core of what we need to compute efficiently to make the *whole* evaluation step efficient. In fact, given the usual degree D polynomial $A(x) := \sum_{j=0}^{D-1} a_j x^j$, if we denote with the vector $\mathbf{a} \in \mathbb{C}^D$ its coefficient form, then it holds that:

$$\mathcal{F}_D(\mathbf{a})_k = A(\zeta_D^{-k})$$

and remember that (for every $k \in \{0, \ldots, D-1\}$) these $A(\zeta_D^{-k})$ are precisely the only ingredient we need to efficiently perform the multiplication through value forms, as pointed out earlier.

So, putting everything together: if we were able to compute all these D Fourier coefficients efficiently we would call it a day (for the evaluation step, at least). However, this is not a trivial task: doing this in the naive way would (once again...) require to compute a sum of D terms for D times, that is a $\Theta(n^2)$ cost that we are not willing to accept. The FFT is the tool that allows us to get to our $\Theta(n \log n)$ goal.

4.3.1 How FFT Works

Let us see what is so special about FFT. Remember that D, being a large enough power of 2, is even. This means that we are able to *split* any k-th Fourier coefficient expression in two sums of

terms, those of *even* indices and those of *odd* ones, in the following way:

$$\begin{aligned}
\mathcal{F}_D(\mathbf{a})_k &= \sum_{j=0}^{D-1} a_j \zeta_D^{-kj} \\
&= \sum_{j=0}^{\frac{D}{2}-1} a_{2j} \zeta_D^{-k(2j)} + \sum_{j=0}^{\frac{D}{2}-1} a_{2j+1} \zeta_D^{-k(2j+1)} \\
&= \sum_{j=0}^{\frac{D}{2}-1} a_{2j} \zeta_D^{-k(2j)} + \sum_{j=0}^{\frac{D}{2}-1} a_{2j+1} \zeta_D^{-2kj-k} \\
&= \sum_{j=0}^{\frac{D}{2}-1} a_{2j} \zeta_D^{-k(2j)} + \zeta_D^{-k} \sum_{j=0}^{\frac{D}{2}-1} a_{2j+1} \zeta_D^{-2kj} \\
&= \sum_{j=0}^{\frac{D}{2}-1} a_{2j} (\zeta_D^2)^{-kj} + \sum_{j=0}^{\frac{D}{2}-1} a_{2j+1} (\zeta_D^2)^{-kj}
\end{aligned} \tag{9}$$

Now, call E and O the DFTs of (respectively the *even* and *odd* entries of) $\mathbf{a}$ on the $\frac{D}{2}$ points given by the $\frac{D}{2}$-roots of unity:

$$(\zeta_D^2)^{-\frac{D}{2}}, (\zeta_D^2)^{-\frac{D}{2}-1}, \ldots, 1$$

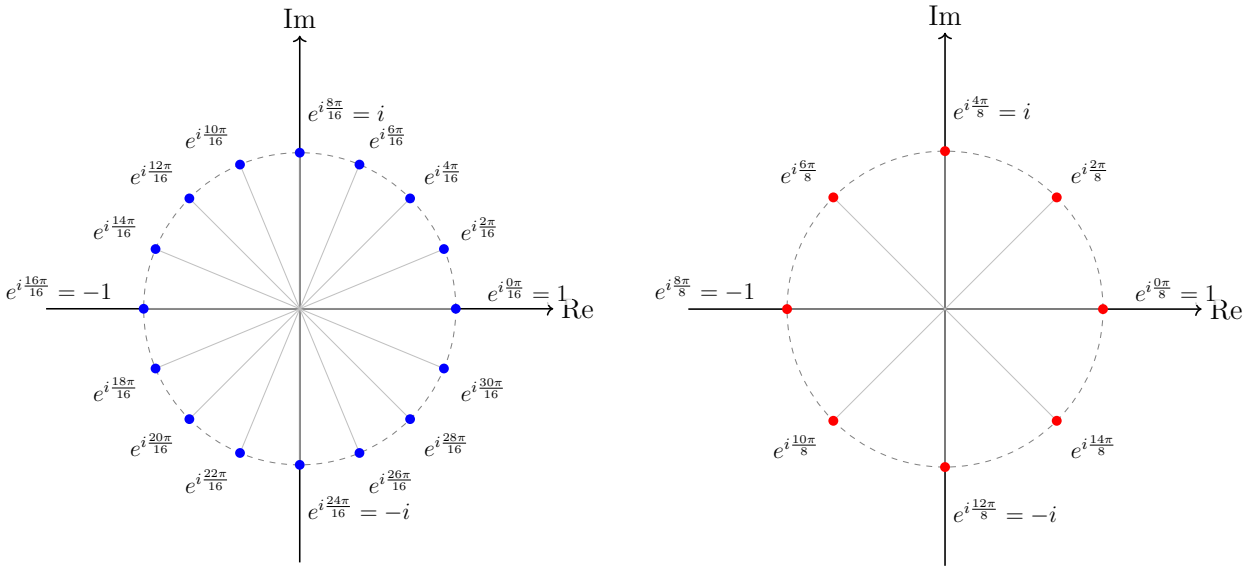


Figure 2: On the left: the representation of the $D = 16$ D-th complex roots of unity. On the right: the representation of the $\frac{D}{2} = \frac{16}{2} = 8$ $\frac{D}{2}$-th complex roots of unity.

that is:

$$E := \mathcal{F}_{\frac{D}{2}}(a_0, a_2, \ldots, a_{D-2}) \quad \text{and} \quad O := \mathcal{F}_{\frac{D}{2}}(a_1, a_3, \ldots, a_{D-1})$$

Now: *only* for $k \in \{0, \ldots, \frac{D}{2} - 1\}$, the k-th Fourier coefficients of E and O, call them E_k and O_k,

can be defined. In fact, let us define E_k as:

$$E_k := \mathcal{F}_{\frac{D}{2}}(a_0, a_2, \ldots, a_{D-2})_k = \sum_{j=0}^{\frac{D}{2}-1} a_{2j}(\zeta_D^2)^{-kj}$$

while O_k as:

$$O_k := \mathcal{F}_{\frac{D}{2}}(a_1, a_3, \ldots, a_{D-1})_k = \sum_{j=0}^{\frac{D}{2}-1} a_{2j+1}(\zeta_D^2)^{-kj}$$

We are going to discover very soon that it is useful to generalize these E_k and O_k to capture the periodicity of their expressions with respect to k, in the following way. For every $k \in \{0, \ldots, D-1\}$, we define $\tilde{E}_k$ as:

$$\tilde{E}_k := E_{k \bmod \frac{D}{2}}$$

which, in layman's terms, means that $\tilde{E}_k$ equals E_k if $k \in \{0, \ldots, \frac{D}{2} - 1\}$ and equals E_s if k exceeds $\frac{D}{2} - 1$ and thus there exists some $s \in \{0, \ldots \frac{D}{2} - 1\}$ such that $k = \frac{D}{2} + s$. These $\tilde{E}_k$ and $\tilde{O}_k$ have the very nice feature of being $\frac{D}{2}$-periodic; for every $k \in \{0, \ldots, \frac{D}{2} - 1\}$ it holds that:

$$\tilde{E}_{\frac{D}{2}+k} = \tilde{E}_k$$

and:

$$\tilde{O}_{\frac{D}{2}+k} = \tilde{O}_k$$

4.3.2 Towards an Effective Divide and Conquer Approach

Why do these $\tilde{E}_k$ and $\tilde{O}_k$ matter so much to us? Because they allow us to conclude (thanks to 9) that the following two *crucial* relations, which we are going to heavily exploit in the design of an efficient divide and conquer approach, hold. In fact, if $k \in \{0, \ldots \frac{D}{2} - 1\}$, by putting everything together it holds that:

$$\mathcal{F}_D(\mathbf{a})_k := \tilde{E}_k + \zeta_D^{-k}\tilde{O}_k \qquad \text{(first crucial relation)}$$

and if we observe that $\zeta_D^{\frac{D}{2}} = -1$ (which holds because $(\zeta_D^{\frac{D}{2}})^2 = 1$ but $\zeta_D^{\frac{D}{2}} \neq 1$), it also holds that:

$$\mathcal{F}_D(\mathbf{a})_{k+\frac{D}{2}} := \tilde{E}_k - \zeta_D^{-k}\tilde{O}_k \qquad \text{(second crucial relation)}$$

Can you see why this could be very useful? If we were given the values $\tilde{E}_k$ and $\tilde{O}_k$, thanks to the two crucial relations, we would be able to efficiently compute not only one but *two* Fourier coefficients, namely $\mathcal{F}_D(\mathbf{a})_k$ *and* $\mathcal{F}_D(\mathbf{a})_{\frac{D}{2}+k}$. This very clever reuse of $\tilde{E}_k$ and $\tilde{O}_k$, made possible by the periodicity that derives from the fact we are working with elements in $\mathbb{C}$, allows us to *simultaneously* compute *all* the D Fourier coefficients $\{\mathcal{F}_D(\mathbf{a})_k\}_{k=0}^{D-1}$ with the following divide and conquer algorithm.

$$\mathsf{FFT}(a_0, \ldots, a_{D-1})$$

1. if $D = 1$, return a_0;
2. $\tilde{E} \leftarrow \mathsf{FFT}(a_0, a_2, \ldots, a_{D-2})$;

3. $\tilde{O} \leftarrow \mathsf{FFT}(a_1, a_3, \ldots, a_{D-1})$;
4. for $k = 0, \ldots, \frac{D}{2} - 1$:
   4.1. $t \leftarrow \zeta_D^{-k} \tilde{O}_k$;
   4.2. $\mathcal{F}_D(\mathbf{a})_k \leftarrow \tilde{E}_k + t$;
   4.3. $\mathcal{F}_D(\mathbf{a})_{\frac{D}{2}+k} \leftarrow \tilde{E}_k - t$;
5. **return** $\{\mathcal{F}_D(\mathbf{a})_k\}_{k=0}^{D-1}$;

Running time: Remember that we are assuming D to be a power of 2. Notice that, at the top level of the recursion, a cost $\Theta(D)$ must be spent to combine the values of $\tilde{E}_k$ and $\tilde{O}_k$ to determine the values of all the $\mathcal{F}_D(\mathbf{a})_k$ (for the suitable values of k) and for other minor tasks. Pairing this consideration with the fact that, as the two crucial relations show, every instance is divided into 2 sub-instances only, we conclude that the recurrence relation that describes the running time $T_{\mathsf{FTT}}(D)$ of the FFT algorithm is:

$$\begin{aligned} T_{\mathsf{FFT}}(D) &= 2 \cdot T_{\mathsf{FFT}}\left(\frac{D}{2}\right) + \Theta(D) \\ &= 2 \cdot 2 \cdot T_{\mathsf{FFT}}\left(\frac{D}{4}\right) + \Theta(D) + \Theta\left(2 \cdot \frac{D}{2}\right) \\ &= \ldots \\ &= \Theta(D \log D) \end{aligned}$$

implying that:

$$T_{\mathsf{FFT}}(D) = \Theta(D \log D) = \Theta(n \log n)$$

since $D = \Theta(n)$ as a consequence of our requirements. We synthesize this achievement in the following theorem.

Theorem 10. *The running time of the evaluation step (5-th step) of the* FFT*-Based Integer Multiplication algorithm is:*

$$T_{\mathsf{FFT}}(D) = \Theta(D \log D) = \Theta(n \log n)$$

Now, let us try to get the same asymptotic result for the *interpolation* step.

4.4 FFT For Efficient Interpolations

One may think that inverting what FFT allowed us to compute would require some different techniques, but surprisingly it is not the case: the FFT algorithm allows us to *also* efficiently perform the interpolation step (the 7-th one) of the FFT-Based Integer Multiplication algorithm. We are going to show why, and we begin by proving the following *Orthogonality Lemma*

Lemma 11 (Orthogonality Lemma)**.** *Let $D \in \mathbb{N}$ and $\zeta_D := e^{\frac{2\pi i}{D}}$ be the first D-th complex roots of unity (as before). Then, for every integer m it holds that:*

$$\sum_{j=0}^{D-1} \zeta_D^{jm} = \begin{cases} D & \text{if } m \bmod D = 0 \\ 0 & \text{if } m \bmod D \neq 0 \end{cases}$$

Proof. If the first case holds, then $m \bmod D = 0$, which implies that $\zeta_D^{jm} = 1$ for every value of the index j, so it actually holds that:

$$\sum_{j=0}^{D-1} \zeta_D^{jm} = \sum_{j=0}^{D-1} 1 = D$$

as in the statement. Suppose now that the second case holds instead, i.e. suppose that $\zeta_D^m \neq 1$. Then, it holds that:

$$(\zeta_D^m)^D = (\zeta_D^D)^m = 1^m = 1$$

implying, thanks to the finite geometric-series identity, that:

$$\begin{aligned}\sum_{j=0}^{D-1} (\zeta_D^m)^j &= \frac{1 - (\zeta_D^m)^D}{1 - \zeta_D^m} \\ &= \frac{1 - 1}{1 - \zeta_D^m} \\ &= 0\end{aligned}$$

which concludes the proof. □

The Orthogonality Lemma is the main ingredient of the proof of the following proposition, telling us, essentially, that the role the coefficient form had in determining the value form through the FFT, is now the role that the value form have in computing *back* the coefficient form, *again through the* FFT.

Proposition 12. *Let $D \in \mathbb{N}$, $\zeta_D := e^{\frac{2\pi i}{D}}$ be the first of the D-th complex roots of unity, and $\mathbf{c} := (c_0, \ldots, c_{D-1})$ be the coefficient form of a polynomial $C(x)$ of degree not exceeding $D-1$. For any $k \in \{0, \ldots, D-1\}$, let us denote with F_k the k-th Fourier coefficient of $\mathbf{c}$ on the points $\zeta_D^{-D}, \ldots, \zeta_D^{-1}, 1$, i.e.:*

$$F_k := \sum_{r=0}^{D-1} c_r \zeta_D^{-kr}$$

then, it holds for every $j \in \{0, \ldots, D-1\}$ that:

$$c_j = \frac{1}{D} \sum_{k=0}^{D-1} F_k \zeta_D^{kj}$$

Proof. It holds that:

$$\begin{aligned}
\frac{1}{D}\sum_{k=0}^{D-1} F_k \zeta_D^{kj} &= \frac{1}{D}\sum_{k=0}^{D-1}\sum_{r=0}^{D-1} c_r \zeta_D^{-kr}\zeta_D^{kj} \\
&= \sum_{r=0}^{D-1} c_r \left(\frac{1}{D}\sum_{k=0}^{D-1}\zeta_D^{k(j-r)}\right) \\
&= c_j \cdot \frac{1}{D}\sum_{k=0}^{D-1}\zeta_D^{k(j-j)} + \sum_{\substack{r\in\{0,\dots,D-1\},\\ r\neq j}} c_r \left(\frac{1}{D}\sum_{k=0}^{D-1}\zeta_D^{k(j-r)}\right) \\
&\overset{\text{Orth. Lem.}}{=} c_j \cdot \frac{\not{D}}{\not{D}} + \cancel{\sum_{\substack{r\in\{0,\dots,D-1\},\\ r\neq j}} c_r \left(\frac{1}{D}\sum_{k=0}^{D-1}\zeta_D^{k(j-r)}\right)} \\
&= c_j
\end{aligned}$$

and this concludes the proof. □

Put in other words, counting on the fact that $F = \mathbb{C}$, given the coefficient form $\mathbf{c}$ of a degree $D-1$ polynomial (having in mind the coefficient form of the product polynomial $C(x) = A(x)B(x)$ of the algorithm), the following relation holds:

$$\mathcal{F}_D^{-1}((\mathcal{F}_D(\mathbf{c})_0), \dots, \mathcal{F}_D(\mathbf{c})_{D-1}) = \frac{1}{D}\overline{\mathcal{F}_D(\overline{\mathcal{F}_D(\mathbf{c})_0}, \dots, \overline{\mathcal{F}_D(\mathbf{c})_{D-1}})}$$

highlighting once again that, in order to perform the *interpolation* step, another application of the FFT algorithm suffices, requiring a cost of only $\Theta(n \log n)$ as already pointed out. We synthesize also this achievement in a theorem.

Theorem 13. *The running time of the interpolation step (7-th step) of the* FFT*-Based Integer Multiplication algorithm is:*

$$T_{\mathsf{FFT}}(D) = \Theta(D \log D) = \Theta(n \log n)$$

4.4.1 Some Final Observations

By proving that both the evaluation and interpolation steps require time $\Theta(n \log n)$, we have also finally proved the following theorem.

Theorem 14. *The running time of the* FFT*-Based Integer Multiplication Algorithm (*$\mathsf{FFT-BIM}$*) is* $\Theta(n \log n)$*, i.e.:*

$$T_{\mathsf{FFT-BIM}} = \Theta(n \log n)$$

One little caveat is that performing operations between complex numbers introduces errors, which must be taken care of if one wants to recover a meaningful output. In fact, the actual coefficients (entries of the vector $\mathbf{c}$) that we want the algorithm to compute are integers, but a numerical FFT actually computes a vector $\tilde{\mathbf{c}}$ of floating point numbers that approximate the entries of $\mathbf{c}$.

However, fortunately this inconvenient can be handled in at least two different ways:

1. either by applying a provably exact recovery criterion, which consists in rounding the entries of $\mathbf{c}$ to their nearest integers after having suitably kept track of all the various cases in which errors could have occurred during the computation;

2. or by avoiding to take $F = \mathbb{C}$, preferring instead a suitable *finite* field having all the properties of $\mathbb{C}$ that we needed, but allowing us to define and utilize *number theoretic transforms*, (and thus allowing us to avoid floating-point errors by working modulo suitable primes).

Finally, we observe that despite the fact that the $\mathsf{FFT} - \mathsf{BIM}$ algorithm asymptotically dominates both Karatsuba's and the grade-school algorithms (thus it should be preferable w.r.t. the other two for inputs of large enough size), the grade school algorithm dominates for small inputs, while Karatsuba's does for certain medium sizes that lie in between.

5 Still Fourier Transforms, but More in General

After observing how central the FFT algorithm proved to be for the Integer Multiplication problem, we devote this last section to get some sense of how (discrete) Fourier transforms occur in other fields, with the hope of sharpening our understanding of it. Before we proceed, we first give some definitions and prove some basic facts related to them.

Let us focus here on *finite cyclic groups* of finite cardinality $D \in \mathbb{N}$. A well-known algebraic result states that every finite cyclic group of size D is isomorphic to $\mathbb{Z}_D$ and thus, without loss of generality, we restrict our attention to the finite cyclic group $\mathbb{Z}_D$. We are interested in studying the functions of the kind:

$$f : \mathbb{Z}_D \longrightarrow \mathbb{C}$$

whose behaviours are of course completely described by the D images yielded by the D elements of $\mathbb{Z}_D$:

$$(f(0), f(1), \ldots, f(D-1)) \in \mathbb{C}^D$$

In particular, the set of all these functions forms a $\mathbb{C}$-vector space.

Definition 15. *Let $D \in \mathbb{N}$. Consider the set of functions:*

$$\mathbb{C}^{\mathbb{Z}_D} := \{f : \mathbb{Z}_D \longrightarrow \mathbb{C}\}$$

and consider the $\mathbb{C}$-vector space obtained by equipping $\mathbb{C}^{\mathbb{Z}_D}$ with the point-wise sum operation ($(f+g)(x) := f(x) + g(x)$ for every $x \in \mathbb{Z}_D$ and any two $f, g \in \mathbb{C}^{\mathbb{Z}_D}$) and the scalar product ($(\lambda f)(x) := \lambda(f)(x)$ for every $x \in \mathbb{Z}_D$, every $\lambda \in C$ and every $f \in \mathbb{C}^{\mathbb{Z}_D}$). Then, it is possible to define on it the scalar product:

$$\langle f, g \rangle := \frac{1}{D} \sum_{x \in \mathbb{Z}_D} f(x)\overline{g(x)} \quad \forall f, g \in \mathbb{C}^{\mathbb{Z}_D}$$

Many orthonormal bases can be defined. A trivial example is given by $\{b_k\}_{k \in \mathbb{Z}_D}$ in which every function b_k is such that:

$$b_k(x) := \begin{cases} \sqrt{D} & \text{if } x = k \\ 0 & \text{if } x \neq k \end{cases}$$

It is trivial to check that it forms an orthonormal basis:

$$\begin{aligned}\langle b_i, b_j \rangle &= \sum_{x \in \mathbb{Z}_D} b_i(x)\overline{b_j(x)} \\ &= \frac{1}{D} \sum_{x \in \mathbb{Z}_D} \sqrt{D} \cdot \mathbb{I}[x = i] \cdot \sqrt{D} \cdot \mathbb{I}[x = j] \\ &= \frac{\cancel{D}}{\cancel{D}} \cdot \delta_{ij} \\ &= \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}\end{aligned}$$

where we denoted with $\mathbb{I}[E]$ the indicator function of E. However, a different specific basis is one that gives emphasis to the results we have previously proved involving FFT. The elements of this basis are called *characters*.

Definition 16. *Let $D \in \mathbb{N}$ and ζ_D be the first D-th complex root of unity. For any $k \in \mathbb{Z}_D$, we call the function $\chi_k : \mathbb{Z}_D \to \mathbb{C}$ such that:*

$$\chi_k(x) = \zeta_D^{kx}$$

*the **character** of x with respect to k (or simply k**-th character** of x).*

Interestingly, characters (kind of) turn sums into products, in the sense that for any three $x, y, k \in \mathbb{Z}_D$ it holds that:

$$\chi_k(x + y) = \chi_k(x)\chi_k(y)$$

Moreover, they form an orthonormal basis of $\mathbb{C}^{\mathbb{Z}_D}$ as a vector space equipped with the scalar product $\langle \cdot, \cdot \rangle$ we previously introduced.

Proposition 17. *The family of characters $\{\chi_k\}_{k \in \mathbb{Z}_D}$ forms an orthonormal basis of the $\mathbb{C}$-vector space defined by $\mathbb{C}^{\mathbb{Z}_D}$ equipped with the scalar product $\langle \cdot, \cdot \rangle$.*

Proof. Let k, k' be any two elements of $\mathbb{Z}_D$. Then, it holds that:

$$\begin{aligned}\langle \chi_k, \chi_{k'} \rangle &= \frac{1}{D} \sum_{x=0}^{D-1} \zeta_D^{kx} \overline{\zeta_D^{k'x}} \\ &= \frac{1}{D} \sum_{x=0}^{D-1} \zeta_D^{(k-k')x} \\ &= \delta_{kk'}\end{aligned}$$

which implies that the D distinct characters $\chi_0, \ldots, \chi_{D-1}$ are all orthogonal (thus independent), and so (since they are D and $\dim(\mathbb{C}^{\mathbb{Z}_D}) = D$) they span the whole $\mathbb{C}^{\mathbb{Z}_D}$. Moreover, for every $k \in \mathbb{Z}_D$ it holds that $\langle \chi_k, \chi_k \rangle = 1$, which allows us to conclude that $\{\chi_k\}_{k \in \mathbb{Z}_D}$ actually is an orthonormal basis. □

A very important property, tightly connected to what we saw when FFT was utilized, is that the basis $\{\chi_k\}_{k \in \mathbb{Z}_D}$ of characters allows to identify, through the simple scalar product with the function f, a "normalized version" of the Fourier coefficients of f whose properties we heavily exploited in the previous section.

Definition 18. *Let $f \in \mathbb{C}^{\mathbb{Z}_D}$. We call:*

$$\hat{f}(k) := \langle f, \chi_k \rangle = \frac{1}{D} \sum_{x=0}^{D-1} f(x) \zeta_D^{-kx}$$

*the **normalized k-th Fourier coefficient of** f. Moreover, we call **Fourier expansion of** f the (unique) expression of f w.r.t. the basis $\{\chi_k\}_{k \in \mathbb{Z}_D}$:*

$$f(x) = \sum_{k=0}^{D-1} \hat{f}(k) \chi_k(x) = \sum_{k=0}^{D-1} \hat{f}(k) \zeta_D^{kx}$$

Finally, we call:

$$\mathcal{F}f(k) := \sum_{x \in \mathbb{Z}_D} f(x) \zeta_D^{-kx}$$

*the **discrete Fourier transform (DFT)** of f w.r.t. k.*

Notice that the Fourier expansion of f is precisely the formula (although expressed through the basis expansion) that allowed us to efficiently compute the interpolation step in the previous section. Since, up until now, it hasn't been necessary to talk about normalizations, it is the case to point out that for every $f \in \mathbb{C}^{\mathbb{Z}_D}$ and every $k \in \mathbb{Z}_D$ it holds that:

$$\mathcal{F}f(k) = D \cdot \hat{f}(k)$$

5.1 Convolutions

It is possible to define a very important operation in $\mathbb{C}^{\mathbb{Z}_D}$: that of *(cyclic) convolution* of two functions.

Definition 19. *Let f, g be any two functions in $\mathbb{C}^{\mathbb{Z}_D}$. Then, we call the **(cyclic) convolution** of f and g the function $f * g \in \mathbb{C}^{\mathbb{Z}_D}$ such that for every $x \in \mathbb{Z}_D$ it holds that:*

$$(f * g)(x) = \sum_{y \in \mathbb{Z}_D} f(y) g(x - y)$$

Without mentioning it, we have already met something that shared this same spirit when we described the formula representing a generic coefficient c_k of the degree-k monomial of the product $C(x) = A(x)B(x)$ of two polynomials (say, of degree $D-1$). In fact, the product of $A(x)$ and $B(x)$ can be expressed not only as:

$$A(x)B(x) = \sum_{i=0}^{D-1} a_i x^i \sum_{j=0}^{D-1} b_j x^j = \sum_{i=0}^{D-1} \sum_{j=0}^{D-1} a_i b_j x^{i+j}$$

but also as:

$$\sum_{k=0}^{2D-2} \left(\sum_{j=0}^{k} a_j b_{k-j} \right) x^k$$

where what is contained in the parentheses clearly recalls the definition of convolutions we have just given. One of the main reasons to be interested in convolutions is that whenever some operation can be expressed as a convolution, it is immediately expressed simply as a sum of products, and sums of products can be rather convenient to handle in many situations (as we have also seen in the previous section, for example). A core result concerning convolutions is the following, called *Convolution Theorem*.

Theorem 20 (Convolution Theorem)**.** *Let f and g be any two functions in $\mathbb{C}^{\mathbb{Z}_D}$. Then, for every $k \in \mathbb{Z}_D$ it holds that:*

$$\mathcal{F}(f * g)(k) = \mathcal{F}f(k)\mathcal{F}g(k)$$

Proof. It holds that:

$$\begin{aligned}\mathcal{F}(f * g)(k) &= \sum_{x \in \mathbb{Z}_D} \sum_{y \in \mathbb{Z}_D} f(y)g(x-y)\zeta_D^{-kx} \\ &\overset{z:=x-y}{=} \sum_{y \in \mathbb{Z}_D} \sum_{z \in \mathbb{Z}_D} f(y)g(z)\zeta_D^{-k(z+y)} \\ &= \left(\sum_{y \in \mathbb{Z}_D} f(y)\zeta_D^{-ky}\right)\left(\sum_{z \in \mathbb{Z}_D} g(z)\zeta_D^{-kz}\right)\end{aligned}$$

where the substitution $z := x - y$ is justified by the fact that subtraction and addition are modulo D, implying that the correspondence between (x, y) and (z, y) is a bijection in $\mathbb{Z}_D^2$. □

The Convolution Theorem sheds some light on (or better: gives a name to) what we actually did when we utilized FFT in order to efficiently multiply polynomials. In fact, if we call $\mathbf{a}$ and $\mathbf{b}$ the coefficient forms of two degree $n-1$ polynomials (suitably padded into size-D vectors), then the Convolution Theorem tells us that:

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

where $\mathbf{a} \odot \mathbf{b} := (a_0 b_0, \ldots, a_{D-1} b_{D-1})$. Since, if $D \geq 2n-1$, ordinary and cyclic convolutions agree on the padded vectors $\mathbf{a}$ and $\mathbf{b}$, we discover now, a posteriori, that what we were interested in finding in the first place was simply the *convolution* of the (suitably padded vectors of the) two polynomials $A(x)$ and $B(x)$.

Now, let us see the very heterogeneous contexts in which (more or less direct) *manifestations* of convolutions naturally occur.

5.1.1 Sums of Independent Variables

Let X and Y be two random variables taking values in $\mathbb{Z}_D$, and call:

$$p_X(x) := \mathbb{P}[X = x] \quad \text{and} \quad p_Y(x) := \mathbb{P}[Y = x]$$

their respective probability mass functions. Then, the probability that the sum $X + Y$ attains a certain value can be expressed as the convolution $p_X * p_Y$, as:

$$\begin{aligned}\mathbb{P}[X + Y = x] &= \sum_{y \in \mathbb{Z}_D} \mathbb{P}[X = y]\,\mathbb{P}[Y = x - y] \\ &= (p_X * p_Y)(x)\end{aligned}$$

The Convolution Theorem can be fruitfully applied here; once the probability mass function p_{X+Y} is seen through the lens of convolutions and then $\mathcal{F}$ is applied to it, for every $k \in \mathbb{Z}_D$ it holds that:

$$\mathcal{F}p_{X+Y}(k) = \mathcal{F}p_X(k)\mathcal{F}p_Y(k)$$

which allows to treat the probability mass function of a *sum* of independent variables as a *product*, that is often easier to handle. This and other related facts can be utilized to prove a vast array of important results in probability theory.

5.1.2 Random Walks

After being convinced that convolutions and Fourier transforms are nice tools for studying the distributions of sums of independent variables, a natural step forward is to consider *random walks* $S_t := X_1 + \ldots + X_t$, where $X_1, \ldots, X_t$ are independent and identically distributed random variables having all probability mass function p. By reiterating the previous observations we observe that p_{S_t} can be expressed as:

$$p_{S_t} = \underbrace{p * p * \cdots * p}_{t \text{ times}}$$

and consequently that:

$$\mathcal{F}p_{S_t}(k) = (\mathcal{F}p(k))^t$$

this last relation allows to deduce important properties of the random walk S_t by simply checking whether $\mathcal{F}p(k)$ satisfies certain conditions or not (for suitably selected $k \in \mathbb{Z}_D$). An important fact is that this framework can actually be extended successfully to deal with problems that do not have a discrete nature, like those concerning (for example) diffusion and heat equation.

5.1.3 Separation of Single Signals From A Mixture of Them

Convolutions turn out to be very useful also for solving very concrete problems. For example, suppose that (say) three musical instruments are emitting three different notes, and that a microphone is recording all of them, *together*. When recording, a microphone only samples variations in air pressure over time, which in an ideal world could be described by a function of the kind:

$$f(t) = 0.85\cos(2\pi \cdot 220t) + 0.6 \cdot \cos(2\pi \cdot 330t) + 0.4 \cdot \cos(2\pi \cdot 440t)$$

For many various reasons, one could be interested in being able to *distinguish* the three notes that are played by having just *the graph* of $f(t)$ at their disposal. It turns out that FFT is helpful also in this situation. In fact, a generic character can be directly related to a wave with frequency $k\omega_0$ (where $\omega_0 := \frac{2\pi}{D}$) as:

$$\chi_k(j) = \zeta_D^{kj} = \cos(jk\omega_0) + i\sin(jk\omega_0) \tag{10}$$

Fortunately, FFT allows us to decompose f as a linear combination of waves with different frequencies (which, as 10 show, form an orthonormal basis):

$$f(x) = \sum_{k \in \mathbb{Z}_D} \hat{f}(k)\zeta_D^{kx}$$

With this approach, the frequencies that are needed to deduce the notes played are identified through local maxima of the Fourier transform. Of course, the idealized setting we are considering doesn't take into account many important nuances that (in actual *real* applications) should be taken care of.

AI use disclosure. Gemini Pro 3.1 (Extended Reasoning) has been utilized exclusively for checking for mistakes and for generating TikZ elements.

References

[1] Karatsuba A., Ofman Y.. Multiplication of Many-Digital Numbers by Automatic Computers *Doklady Akademii Nauk SSSR*, Vol. 145, No. 2, 1962, pp. 293-294.

[2] Cooley J.W., Tukey J.W. An algorithm for the machine calculation of complex Fourier series, *Math. Comp.*, 19 (1965), 297-301.

[3] Gauss C.F. Nachlass: Theoria interpolationis methodo nova tractata, *Werke*, Band 3, pp. 265-327. Göttingen: Königliche Gesellschaft der Wissenschaften, 1866.