Generic Reductions for In-place Polynomial Multiplication

Pascal Giorgi
LIRMM, Univ. Montpellier, CNRS
Montpellier, France
pascal.giorgi@lirmm.fr

Bruno Grenet
LIRMM, Univ. Montpellier, CNRS
Montpellier, France
bruno.grenet@lirmm.fr

Daniel S. Roche
United States Naval Academy
Annapolis, Maryland, U.S.A.
roche@usna.edu

ABSTRACT

The polynomial multiplication problem has attracted considerable attention since the early days of computer algebra, and several algorithms have been designed to achieve the best possible time complexity. More recently, efforts have been made to improve the space complexity, developing modified versions of a few specific algorithms to use no extra space while keeping the same asymptotic running time.

In this work, we broaden the scope in two regards. First, we ask whether an arbitrary multiplication algorithm can be performed in-place generically. Second, we consider two important variants which produce only part of the result (and hence have less space to work with), the so-called middle and short products, and ask whether these operations can also be performed in-place.

To answer both questions in (mostly) the affirmative, we provide a series of reductions starting with any linear-space multiplication algorithm. For full and short product algorithms these reductions yield in-place versions with the same asymptotic time complexity as the out-of-place version. For the middle product, the reduction incurs an extra logarithmic factor in the time complexity only when the algorithm is quasi-linear.

KEYWORDS

arithmetic, polynomial multiplication, in-place algorithm, self reduction

ACM Reference Format:
Pascal Giorgi, Bruno Grenet, and Daniel S. Roche. 2019. Generic Reductions for In-place Polynomial Multiplication. In *International Symposium on Symbolic and Algebraic Computation (ISSAC '19), July 15–18, 2019, Beijing, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3326229.3326249

1 INTRODUCTION

1.1 Polynomial multiplication

Polynomial multiplication is a fundamental problem in mathematical algorithms. It forms the basis (and key bottleneck) for other fundamental problems such as division with remainder, GCD computation, evaluation/interpolation, resultants, factorization, and structured linear algebra (see, e.g., [10, §8–15] and [3, §2–7,10,12]).

As such, significant effort has gone to improving the time to multiply two size-n polynomials, mainly following ideas from integers multiplication most notably with Karatsuba's algorithm [17], Toom-Cook multiplication [9], and Schönhage-Strassen [22]. More recent results have improved the complexity further with dedicated polynomial approaches over finite field [13, 15] or over any algebra [7], yielding the best known quasi-linear time algorithms.

1.2 Space complexity

After minimizing the runtime, an important question both in theory and in practice is how much extra *space* these algorithms require. While the classical algorithm can be made to use only a constant number of temporary values, all the faster algorithms mentioned above require $O(n)$ space to multiply two size-n polynomials. In fact, proven time-space trade-offs in the algebraic circuit and branching program models indicate that space at least polynomial in n is required for any sub-quadratic multiplication algorithm [1, 21].

But in a model where the output space admits both random writes *and* reads, these time-space lower bounds can be broken. [20] developed a variant of Karatsuba's algorithm using only $O(\log n)$ space. Later, an FFT-based multiplication algorithm using $O(n \log n)$ time and constant space was developed for the case that the coefficient ring contains a suitable root of unity [14]. Space-saving versions of Karatsuba's algorithm can also be found in [6, 8, 23, 24].

1.3 Short and middle products

Besides the usual *full product* computation, two variants have also been extensively studied: the *short product* which truncates the output to the first n terms, and the *middle product* which truncates the result on both ends. These variants are important especially for power series, and specific variants of Karatsuba's algorithm and others have been developed, usually gaining a constant factor compared to a full product followed by a truncation [11, 12, 19]. For space efficiency, dedicated algorithms are even mandatory as using full product increases the output size whence the space complexity.

In some specific model of straight line programs, [4] shows that transposing full product yields a middle product with the same time and space complexities. Nevertheless, such a result does not directly apply to the more general context of algorithms, since their model allows computations with fixed size inputs only.

1.4 Our work

In this paper, we develop reductions which transform any multiplication algorithm which uses $O(n)$ extra space into full, short, and middle product algorithms which use only $O(1)$ extra space. For this we provide two kinds of reductions: time-space preserving reductions that link the different product variants; and self-reductions for each variant that reduce space complexity to $O(1)$. The time complexity for full and short product is the same as that of the original, while that for middle product incurs an additional $\log n$ factor. This improves the $O(\log n)$ space of the most space-constrained

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ISSAC '19, July 15–18, 2019, Beijing, China
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6084-5/19/07…$15.00
https://doi.org/10.1145/3326229.3326249

Karatsuba algorithm [20], and implies for the first time: in-place versions of Toom-Cook multiplication; in-place FFT-based multiplication even when the ring does not contain a root of unity; in-place subquadratic short and middle product algorithms.

We begin by carefully stating our space complexity model and then defining the multiplications problems in Sections 2 and 3. Our time-space preserving reductions and some equivalences are presented next in Section 4, followed by the critical self-reductions in Section 5 which prove our main results.

2 COMPLEXITY MODEL

We use the model of an *algebraic-RAM* that is equipped with two kinds of registers: the standard registers store integers as in the classical Word-RAM model, whereas the *algebraic registers* store elements from the base field $\mathbb{K}$ of coefficients. As in Word-RAM, we assume that the standard registers can store integers of size $O(\log n)$ where n is the number of coefficients in the inputs.

Word-RAM machines are a classical model in computational complexity, in particular for *fine-grained complexity* that classifies the difficulty of polynomial-time problems [25]. We use it in order to distinguish between the space needed to store indices (that is thus hidden in the standard registers) from the space needed to store elements from the base field.

Time complexity. As mentioned, we use the number of arithmetic operations as the time complexity measure since the cost of the operations on indices is negligible with respect to arithmetic operations. Formally, we assume that any ring operation on the algebraic registers has cost 1.

Space complexity. We divide the registers into three categories: the input space is made of the (algebraic) registers that store the inputs, the output space is made of the (algebraic) registers where the output must be written, and the work space is made of (algebraic and non-algebraic) registers that are used as extra space during the computation. The space complexity is then the maximum number of work registers used simultaneously during the computation. An algorithm is said to be "in-place" if its space complexity is $O(1)$, and "out-of-place" otherwise.

One can then distinguish different models depending on the read/write permissions on the input and output registers:

(1) Input space is read-only, output space write-only;
(2) Input space is read-only, output space is read/write;
(3) Input and output spaces are both read/write.

The first model is the classical one from complexity theory [2]. Despite its theoretical interest, it does not reflect low-level computation where output is typically in some DRAM or Flash memory on which reading is no more costly than writing. Furthermore, polynomial multiplication here has a quadratic lower bound for time × space [1], limiting the possibility for meaningful improvements.

The second model has been used in the context of in-place polynomial multiplication [14, 20]. This is a very reasonable model since it matches the paradigm of parallel computing with shared memory. This is the model in which we develop our algorithms.

The third model has been used in the context of straight line programs to provide a generic approach for preserving memory in Tellegen's transposition principle [4]. However, this memory model allows to erase the input registers, which can be problematic especially for recursive algorithms that re-use their operands. Hence, we will not use this too-permissive model.

Notation. The output space in our algorithms is denoted by R and registers are indexed from 0 to $n-1$. We write $\mathsf{R}_{[k..\ell[}$ to denote the registers of indices k to $\ell-1$.

3 POLYNOMIAL MULTIPLICATIONS

Define the *size* of a univariate polynomial as the number of coefficients in its (dense) representation; a polynomial of size n has degree at most $n-1$. Importantly, we allow zero padding: a size-n polynomial could have degree strictly less than $n-1$; the size indicates only how it is represented.

Let $f = \sum_{i=0}^{n-1} f_i X^i$ and $g = \sum_{i=0}^{n-1} g_i X^i$ be two size-n polynomials. Their product $h = fg$ is a polynomial of size $2n-1$, what we call a *balanced full product*. More generally, if f has size m and g has size n, their product has size $m+n-1$. We call this case the *unbalanced full product* of f and g.

We now define precisely the short product, middle product, and half-additive full product.

Definition 3.1. *Let f and g be two size-n polynomials. Their* low short product *is the size-n polynomial defined as*

$$\mathsf{SP}_{\mathsf{lo}}(f,g) = (f \cdot g) \bmod X^n$$

and their high short product *is the size-$(n-1)$ polynomial defined as*

$$\mathsf{SP}_{\mathsf{hi}}(f,g) = (f \cdot g) \operatorname{quo} X^n.$$

The low short product is actually the meaningful notion of product for truncated power series. Note also that the definition of the high short product that we use implies that the result does not depend on all the coefficients of f and g. The rationale for this choice is to have the identity $fg = \mathsf{SP}_{\mathsf{lo}}(f,g) + X^n \mathsf{SP}_{\mathsf{hi}}(f,g)$.

Definition 3.2. *Let f and g be two polynomials sizes $n+m-1$ and n, respectively. Their* middle product *is the size-m polynomial made of the central coefficients of the product fg, that is*

$$\mathsf{MP}(f,g) = \left((f \cdot g) \operatorname{quo} X^{n-1}\right) \bmod X^m.$$

If $f = \sum_{i<n+m-1} f_i X^i$ and $g = \sum_{j<n} g_j X^j$, then

$$\mathsf{MP}(f,g) = \sum_{n-1 \le i+j < n+m-1} f_i g_j X^{i+j-n+1}.$$

The middle product, most commonly in the special case $n = m$, arises naturally in several algorithms manipulating polynomials or power series which are based on Newton's iteration, such as division or square root [11].

The most efficient algorithms for middle product are obtained by *Tellegen's transposition principle* from full product algorithms [4, 11], saving constant factor in time complexity compared to the naive approach. It is not yet known if this transposition or similar one can preserve space complexity. This question has been established as an open problem in [16], and only answered partially in [4] in a restricted model.

Definition 3.3. *Let f and g be two size-n polynomials, and h be a size-$(n-1)$ polynomial. The* (low-order) half-additive full product *of f and g given h is $\mathsf{FP}^+_{\mathsf{lo}}(f,g,h) = h + fg$. Similarly, their* high-order

half-additive full product *is* $\mathsf{FP}^+_{\mathsf{hi}}(f,g,h) = X^n h + fg$. *An* in-place *half-additive full product algorithm is an algorithm computing a half-additive full product with h initially stored in the output space.*

This variant of the full product which has a partially-initialized output space will be useful to derive other in-place algorithms.

3.1 Multiplications as linear maps

For ease of explanation, we will use the linear property of polynomial multiplications when an operand is fixed. Let $f = \sum_{i=0}^{n-1} f_i X^i$ and $g = \sum_{i=0}^{n-1} g_i X^i$ be two size-n polynomials. If f is fixed, the product $h = fg$ can be described as a linear map from $\mathbb{K}^n$ to $\mathbb{K}^{2n-1}$. The matrix, denoted $\mathfrak{M}_{\mathsf{FP}(f)}$, for this map is to a Toeplitz matrix built from the coefficients of f, and the product fg corresponds to the following matrix-vector product:

$$\underbrace{\begin{pmatrix} f_0 & & \\ \vdots & \ddots & \\ f_{n-1} & & f_0 \\ & \ddots & \vdots \\ & & f_{n-1} \end{pmatrix}}_{\mathfrak{M}_{\mathsf{FP}(f)}} \times \underbrace{\begin{pmatrix} g_0 \\ \vdots \\ g_{n-1} \end{pmatrix}}_{\vec{g}} = \underbrace{\begin{pmatrix} h_0 \\ \\ \vdots \\ \\ h_{2n-1} \end{pmatrix}}_{\vec{h}} \tag{1}$$

where $\mathfrak{M}_{\mathsf{FP}(f)} \in \mathbb{K}^{(2n-1)\times n}$, $\vec{g} \in \mathbb{K}^n$ and $\vec{h} \in \mathbb{K}^{2n-1}$.

The low and high short products being defined as part of the result of the full product, their corresponding linear maps are from $\mathbb{K}^n$ to $\mathbb{K}^n$ and from $\mathbb{K}^n$ to $\mathbb{K}^{n-1}$ respectively, given by submatrices of $\mathfrak{M}_{\mathsf{FP}(f)}$ as follows:

$$\underbrace{\begin{pmatrix} f_0 & & \\ \vdots & \ddots & \\ f_{n-1} & \dots & f_0 \end{pmatrix}}_{\mathfrak{M}_{\mathsf{SP}_{\mathsf{lo}}(f)}} \quad \underbrace{\begin{pmatrix} 0 & f_{n-1} & \dots & f_1 \\ \vdots & & \ddots & \vdots \\ 0 & & & f_{n-1} \end{pmatrix}}_{\mathfrak{M}_{\mathsf{SP}_{\mathsf{hi}}(f)}} \tag{2}$$

Lastly, the middle product corresponds to a linear map from $\mathbb{K}^n$ to $\mathbb{K}^m$ with larger operand fixed, given by the $m \times n$ Toeplitz matrix

$$\underbrace{\begin{pmatrix} f_{n-1} & f_{n-2} & \dots & f_1 & f_0 \\ f_n & f_{n-1} & & f_2 & f_1 \\ \vdots & \vdots & & \vdots & \vdots \\ f_{n+m-2} & f_{n+m-3} & \dots & f_{m-2} & f_{m-1} \end{pmatrix}}_{\mathfrak{M}_{\mathsf{MP}(f)}}. \tag{3}$$

4 TIME-SPACE PRESERVING REDUCTIONS

In this section, we compare the relative difficulties of the full product, the half-additive full product, the low and high short products, and the balanced middle product, in the framework of time and space efficient algorithms. To this end, we define a notion of *time and space preserving reduction* between problems.

We say that a problem A is TISP-reducible to a problem B if, given an algorithm for B that has time complexity $t(n)$ and space complexity $s(n)$, one can deduce an algorithm for A that has time complexity $O(t(n))$ and space complexity $s(n) + O(1)$. We write $A \leq_{\mathsf{TISP}} B$ is A is TISP-reducible to B and $A \equiv_{\mathsf{TISP}} B$ if both $A \leq_{\mathsf{TISP}} B$ and $B \leq_{\mathsf{TISP}} A$. Note that the TISP-reduction is transitive. The reduction we use can be defined using oracles and is an adaptation of the notion of *fine-grained reduction* [25, Definition 2.1] adapted to time-space fine-grained complexity classes [18]. In this section, MP denotes the balanced middle product.

Theorem 4.1. *Half-additive full products and short products are equivalent under* TISP*-reductions, that is*

$$\mathsf{FP}^+_{\mathsf{hi}} \equiv_{\mathsf{TISP}} \mathsf{FP}^+_{\mathsf{lo}} \equiv_{\mathsf{TISP}} \mathsf{SP}_{\mathsf{hi}} \equiv_{\mathsf{TISP}} \mathsf{SP}_{\mathsf{lo}}.$$

Furthermore, if SP *denotes either* $\mathsf{SP}_{\mathsf{lo}}$ *or* $\mathsf{SP}_{\mathsf{hi}}$,

$$\mathsf{FP} \leq_{\mathsf{TISP}} \mathsf{SP} \leq_{\mathsf{TISP}} \mathsf{MP}.$$

Proof. The equivalences $\mathsf{SP}_{\mathsf{hi}} \equiv_{\mathsf{TISP}} \mathsf{SP}_{\mathsf{lo}}$ and $\mathsf{FP}^+_{\mathsf{hi}} \equiv_{\mathsf{TISP}} \mathsf{FP}^+_{\mathsf{lo}}$ are proved below in Lemmas 4.3 and 4.4. The equivalence $\mathsf{SP} \equiv \mathsf{FP}^+$ (where SP denotes any of $\mathsf{SP}_{\mathsf{lo}}$ and $\mathsf{SP}_{\mathsf{hi}}$, and FP^+ any of $\mathsf{FP}^+_{\mathsf{lo}}$ and $\mathsf{FP}^+_{\mathsf{hi}}$) is proved in Section 4.2. The reduction $\mathsf{FP} \leq_{\mathsf{TISP}} \mathsf{SP}$ simply amounts to the identity $\mathsf{FP}(f,g) = \mathsf{SP}_{\mathsf{lo}}(f,g) + X^n \mathsf{SP}_{\mathsf{hi}}(f,g)$. The reduction $\mathsf{SP} \leq_{\mathsf{TISP}} \mathsf{MP}$ follows from the following equalities where 0_n denotes the zero polynomial stored in size n:

$$\mathsf{SP}_{\mathsf{lo}}(f,g) = \mathsf{MP}(0_{n-1} + X^{n-1} f, g), \text{ and}$$
$$\mathsf{SP}_{\mathsf{hi}}(f,g) = \mathsf{MP}((f \operatorname{quo} X) + X^{n-1} 0_{n-2}, g \operatorname{quo} X).$$

Hence, one can compute the full product, the low and high short products of f and g simply by calling a middle product algorithm on f padded with zeroes and g. In our model of read-only inputs, an actual padding is not required. It is sufficient to use some kind of *fake padding* where the data structure storing f is responsible for returning 0 when needed. □

The relative order of difficulty $\mathsf{FP} \leq_{\mathsf{TISP}} \mathsf{SP} \leq_{\mathsf{TISP}} \mathsf{MP}$ makes intuitive sense based on the size of the output compared to the size of the inputs since the output can be used as work space: The full product maps $2n$ coefficients to $2n-1$ coefficients, the short products map $2n$ coefficients to n coefficients and the middle product maps $3n$ coefficients to n coefficients. In Section 5, we shall give a partial converse to $\mathsf{SP} \leq_{\mathsf{TISP}} \mathsf{MP}$: There exists a reduction from SP to MP which preserves space and either maintains the asymptotic complexity or increases it by a logarithmic factor.

4.1 Equivalences based on reverse polynomials

Definition 4.2. *The* size-n reversal *of a polynomial f is defined as* $\mathrm{rev}_n(f) = X^{n-1} f(1/X)$.

We note that any algorithm whose input is a size-n polynomial f can be turned into a new algorithm that computes the same function with input $\mathrm{rev}_n(f)$, simply by replacing a query to any coefficient with index i with one of index $n-i$, not affecting the number of ring operations. Let us now prove that $\mathsf{SP}_{\mathsf{hi}} \equiv_{\mathsf{TISP}} \mathsf{SP}_{\mathsf{lo}}$.

Lemma 4.3. *Let f and g be two size-n polynomials. Then we get* $\mathsf{SP}_{\mathsf{hi}}(f,g) = \mathrm{rev}_{n-1}(\mathsf{SP}_{\mathsf{lo}}(\mathrm{rev}_{n-1}(f \operatorname{quo} X), \mathrm{rev}_{n-1}(g \operatorname{quo} X)))$.

Proof. Let $\tilde{f} = \mathrm{rev}_{n-1}(f \operatorname{quo} X)$ and $\tilde{g} = \mathrm{rev}_{n-1}(g \operatorname{quo} X)$. Then

$$\mathsf{SP}_{\mathsf{lo}}(\tilde{f}, \tilde{g}) = \sum_{\substack{0 \leq i,j < n-1 \\ i+j < n-1}} f_{n-1-i} g_{n-1-j} X^{i+j},$$

whence

$$\mathrm{rev}_{n-1}\left(\mathsf{SP}_{\mathsf{lo}}(\tilde{f},\tilde{g})\right) = \sum_{\substack{0\le i,j<n-1\\ i+j<n-1}} f_{n-1-i}g_{n-1-j}X^{n-2-(i+j)}.$$

One can change the indices of summation using $k = n-1-i$ and $\ell = n-1-j$. Then $n-2-(i+j) = k+\ell-n$ and the indices i and j such that $0 \le i+j < n-1$ are mapped to indices k and ℓ such that $2n-1 > k+\ell \ge n$. In other words,

$$\mathrm{rev}_{n-1}\left(\mathsf{SP}_{\mathsf{lo}}(\tilde{f},\tilde{g})\right) = \sum_{\substack{0<k,\ell\le n-1\\ n\le k+\ell<2n-1}} f_k g_\ell X^{k+\ell-n} = \mathsf{SP}_{\mathsf{hi}}(f,g). \quad \square$$

Similarly, we can prove that $\mathsf{FP}^+_{\mathsf{hi}} \equiv_{\mathsf{TISP}} \mathsf{FP}^+_{\mathsf{lo}}$.

Lemma 4.4. *Let f and g be two size-n polynomials and h be a size-$(n-1)$ polynomial. Then*

$$\mathsf{FP}^+_{\mathsf{hi}}(f,g,h) = \mathrm{rev}_{2n-1}\left(\mathsf{FP}^+_{\mathsf{lo}}(\mathrm{rev}_n(f),\mathrm{rev}_n(g),\mathrm{rev}_{n-1}(h))\right).$$

Proof. Let $f^* = \mathrm{rev}_n(f)$, $g^* = \mathrm{rev}_n(g)$ and $h^* = \mathrm{rev}_{n-1}(h)$. First note that $\mathrm{rev}_{2n-1}(h^*) = X^n h$ by definition. Since $\mathrm{rev}_{2n-1}(f^*g^*) = \mathrm{rev}_n(f^*)\,\mathrm{rev}_n(g^*)$ we get $\mathrm{rev}_{2n-1}(f^*g^*+h^*) = \mathrm{rev}_n(f^*)\,\mathrm{rev}_n(g^*) + \mathrm{rev}_{2n-1}(h^*) = fg + X^n h = \mathsf{FP}^+_{\mathsf{hi}}(f,g,h)$. $\square$

4.2 Equivalence between short products and half-additive full products

Reduction from SP *to* FP^+. Let f and g be two size-n polynomials and h be a size-$(n-1)$ polynomial. The half-additive full product $\mathsf{FP}^+_{\mathsf{lo}}(f,g,h)$ equals $h+fg$. Note that $fg = \mathsf{SP}_{\mathsf{lo}}(f,g) + X^n\mathsf{SP}_{\mathsf{hi}}(f,g)$. This already proves that the non-additive full product can be computed using algorithms for low and high short products. For the half-additive full products, it is sufficient to store an intermediate result in the free registers of the output space.

Assuming $\mathsf{R}_{[0..n-1[}$ holds the value of h, the following instructions reduces the computation of $\mathsf{FP}^+_{\mathsf{lo}}(f,g,h)$ to two short products plus $(n-1)$ additions.

1: $\mathsf{R}_{[n-1..2n-1[} \leftarrow \mathsf{SP}_{\mathsf{lo}}(f,g)$
2: $\mathsf{R}_{[0..n-1[} \leftarrow \mathsf{R}_{[0..n-1[} + \mathsf{R}_{[n-1..2n-2[}$
3: $\mathsf{R}_{n-1} \leftarrow \mathsf{R}_{2n-1}$
4: $\mathsf{R}_{[n..2n-1[} \leftarrow \mathsf{SP}_{\mathsf{hi}}(f,g)$

Reduction from FP^+ *to* SP. Let f and g be two size-n polynomials. Splitting f and g by half such that $f = f_0 + X^{\lceil n/2\rceil}f_1$ and $g = g_0 + X^{\lceil n/2\rceil}g_1$, we have

$$\mathsf{SP}_{\mathsf{lo}}(f,g) = f_0g_0 + X^{\lceil n/2\rceil}(f_0g_1 + f_1g_0) \bmod X^n.$$

What is needed is the full product of f_0 and g_0, and the low short products of f_0 and g_1, and f_1 and g_0. Actually, since f_0 is larger than g_1 when n is odd (and g_0 larger than f_1), one only needs the short products $\mathsf{SP}_{\mathsf{lo}}(f_0^-, g_1)$ and $\mathsf{SP}(f_1, g_0^-)$ where $f_0^- = f \bmod X^{\lfloor n/2\rfloor}$ and $g_0^- = g \bmod X^{\lfloor n/2\rfloor}$.

To avoid any recursive call that would imply storing a call stack, we can actually use full products instead of short products: We first compute $f_0^-g_1 + f_1g_0^-$ using a full product and a half-additive full product. Then we can forget about the higher order terms, and add f_0g_0 to this sum using a second half-additive full product. The following instructions summarize this approach:

1: $\mathsf{R}_{[0..2\lfloor n/2\rfloor-1[} \leftarrow \mathsf{FP}(f_0^-, g_1)$ ▹ half-additivity not needed
2: $\mathsf{R}_{[0..2\lfloor n/2\rfloor-1[} \leftarrow \mathsf{FP}^+_{\mathsf{lo}}(f_1, g_0^-, f_0^-g_1 \bmod X^{\lfloor n/2\rfloor-1})$
3: $\mathsf{R}_{[\lceil n/2\rceil..n[} \leftarrow \mathsf{R}_{[0..\lfloor n/2\rfloor[}$ ▹ keep lower part of $f_0^-g_1 + f_1g_0^-$
4: $\mathsf{R}_{[0..2\lceil n/2\rceil-1[} \leftarrow \mathsf{FP}^+_{\mathsf{hi}}(f_0, g_0, (f_0^-g_1 + f_1g_0^-) \bmod X^{\lceil n/2\rceil-1})$

The correctness is clear. The complexity of the algorithm is the cost of three full products in size approximately $n/2$: One non-additive full product in size $\lfloor n/2\rfloor$ and two half-additive full products in size $\lfloor n/2\rfloor$ and $\lceil n/2\rceil$, respectively.

As direct consequence of Lemmas 4.3 and 4.4, one obtains the same reductions to $\mathsf{SP}_{\mathsf{hi}}$ and from $\mathsf{FP}^+_{\mathsf{lo}}$ or $\mathsf{FP}^+_{\mathsf{hi}}$.

4.3 From half-additive full product to unbalanced full product

The unbalanced full product can be computed using any algorithm for the (balanced) full product. Nevertheless, the space complexity increases since intermediate results must be stored. Given an algorithm for the balanced full product of space complexity $s(n)$, one obtains an algorithm with space complexity $s(n) + (n-1)$ for the unbalanced full product. In this section, we prove that if the original full product algorithm is *half-additive*, the resulting unbalanced full product algorithm has the same space complexity.

Let f be a size-m polynomial and g be a size-n polynomial with $m > n$. Write $f = \sum_{k=0}^{\lceil m/n\rceil-1} X^{kn}f_k$, where each sub-polynomial $f_0, \ldots, f_{\lceil m/n\rceil-1}$ has size at most n. The computation of $f\cdot g$ reduces to the computations of each $f_k \cdot g$. The following instructions prove that using half-additivity, the intermediate results $f_k \cdot g$ can be computed directly in the output space.

1: $\mathsf{R}_{[(\lceil m/n\rceil-1)n..m+n[} \leftarrow \mathsf{FP}(f_{\lceil m/n\rceil-1}, g)$
2: **for** k from $\lceil m/n\rceil - 2$ down to 0 **do**
3: $\quad \mathsf{R}_{[kn..(k+2)n-1[} \leftarrow \mathsf{FP}^+_{\mathsf{hi}}(f_k, g, f_{k+1}g \bmod X^n)$

Note that at step 1, the size of $f_{\lceil m/n\rceil-1}$ can be strictly smaller than n. To ensure a balanced product, fake padding can be done on this input and the output can be placed anywhere in the free output space. Thus, the time complexity is at most $\lceil m/n\rceil\mathsf{M}(n)$ where $\mathsf{M}(n)$ is the complexity of the half-additive full product.

5 IN-PLACE ALGORITHMS FROM OUT-OF-PLACE ALGORITHMS

In this section, we show how to obtain in-place algorithms from out-of-place algorithms. The theorem below summarizes the main results described in this section.

Theorem 5.1.

(i) Given a full product algorithm with time complexity $\mathsf{M}(n)$ and space complexity $\le cn$, one can build an in-place algorithm for the half-additive full product with time complexity $\le (2c+7)\mathsf{M}(n) + o(\mathsf{M}(n))$.

(ii) Given a (low or high) short product algorithm with time complexity $\mathsf{M}(n)$ and space complexity $\le cn$, one can build an in-place algorithm for the same problem with time complexity $\le (2c+5)\mathsf{M}(n) + o(\mathsf{M}(n))$.

(iii) Given a middle product algorithm with time complexity $\mathsf{M}(n)$ and space complexity $\le cn$, one can build an in-place algorithm for the same problem with time complexity $\le \mathsf{M}(n)\log_{\frac{c+1}{c+2}}(n) + O(\mathsf{M}(n))$ if $\mathsf{M}(n)$ is quasi-linear, and $O(\mathsf{M}(n))$ otherwise.

Actually, our reductions work for any space bound $s(n) \le O(n)$. Smaller space bounds yield better time bounds though we do not have a general expression in terms of $s(n)$. Yet sublinear space bounds still imply an increase of the time complexity by a multiplicative constant for full and short products.

Formally, we give self-reductions for the three problems. That is, we use an out-of-place algorithm for the problem as building block of our in-place version. The general idea is similar in the three cases. In a first step, we use the out-of-place algorithm to compute some part of the output, using the unused output space as temporary work space. Then a recursive call finishes the work. The (constant) amount of space needed in our in-place algorithms correspond the space needed to process the base cases. As depicted in section 3.1, we aim to apply some specific matrix to a vector. Our general construction consists in first applying the top or bottom rows of the matrix to the vector using the out-of-place algorithm, and applying the remaining rows using a recursive call (*cf.* Fig. 1). Similar techniques have been already used for in-place fast matrix multiplication [5].

In the cases of full and short products, the diamond and triangular shapes of the corresponding matrices imply that the recursive call is made on two smaller inputs: For instance, to apply the first rows of a lower triangular matrix to a vector, one only needs to apply it to the first entries of the vector. For the middle product, the square shape imply that one input remains of the same size in the recursive call. This difference explains the difference in the time complexities in Theorem 5.1.

Our complexity analyses are based on the following Proposition which is proved in Section 5.4.

Proposition 5.2. *Let $T, \mathsf{M} : \mathbb{N} \to \mathbb{R}_+$ such that M is super-linear.*

(i) If for all n, $T(n) \le \sum_k a_k \mathsf{M}(\lfloor \lambda_k n + \mu_k \rfloor) + bn + c + T(\lfloor \alpha n + \beta \rfloor)$ where $a_k, \lambda_k, \mu_k, b, c, \alpha, \beta \in \mathbb{R}_+^*$ with $\alpha < 1$ and $\lambda_k n + \mu_k < n$ for all k. Then

$$T(n) \le \sum_k \left(\frac{a_k \lambda_k}{1-\alpha}\right) \mathsf{M}(n) + \frac{bn}{1-\alpha} + o(\mathsf{M}(n)).$$

(ii) If for all $m \le n$, $T(m) \le (\lambda m/n + \mu/(m - \frac{1}{1-\alpha}) + 1)\mathsf{M}(n) + T(\lfloor \alpha m + 1 \rfloor)$ where $\lambda, \mu, \alpha \in \mathbb{R}_+^*$ such that $\alpha < 1$. Then for $m = n$,

$$T(n) \le \mathsf{M}(n) \log_{1/\alpha}(n) + \frac{\lambda + \mu\alpha}{1-\alpha} \mathsf{M}(n) + o(\mathsf{M}(n)).$$

(iii) If for all $m \le n$, $T(m) \le \lambda(\frac{n}{(1-\alpha)m-1)} + 1)((1-\alpha)m)^\gamma + T(\lfloor \alpha m + 1 \rfloor)$ where $\alpha, \gamma, \lambda \in \mathbb{R}_+^*$ such that $\alpha < 1$ and $\gamma > 1$. Then for $m = n$, $T(n) \le O(n^\gamma)$.

5.1 In-place full product algorithm

Our aim is to build an in-place (low-order) half-additive full product algorithm $\mathtt{iFP}^+_{\mathtt{hi}}$ based on an out-of-place full product algorithm $\mathtt{oFP}$ that has space complexity cn. That is, we are given two size-n polynomials f and g in the input space and a size-$(n-1)$ polynomial h in the $(n-1)$ low-order registers of the output space R and we aim to compute $fg + h$ in R. The algorithm is based on a tiling of the matrix $\mathfrak{M}_{\mathsf{FP}(f)}$ given in Eq. (1), see Fig. 1 (left).

For some $k < n$ to be fixed later, let $f = \hat{f}X^k + f_0$ and $g = \hat{g}X^k + g_0$ where $\deg f_0, \deg g_0 < k$. Then we have

$$h + fg = h + f_0 g + \hat{f} g_0 X^k + \hat{f}\hat{g} X^{2k}. \tag{4}$$

Recall that the output R has size $2n-1$ with its $n-1$ lowest registers containing h. Then Eq. (4) can be evaluated as follows:

1: $\mathsf{R}_{[0..n+k-1[} \leftarrow h + f_0 g$
2: $\mathsf{R}_{[k..n+k-1[} \leftarrow \mathsf{R}_{[k..n+k-1[} + \hat{f} g_0$
3: $\mathsf{R}_{[2k..2n[} \leftarrow \mathsf{R}_{[2k..2n[} + \hat{f}\hat{g}$

The first two steps correspond exactly to two *additive* unbalanced full products, that is unbalanced full products that must be added to some already filled output space. One can describe an algorithm $\mathtt{oFP}^+_{\mathsf{u}}$ for this task, based on a (standard) full product algorithm $\mathtt{oFP}$: If f has size k and g has size n, $n > k$, we write $g = \sum_{i=0}^{\lceil n/k \rceil - 1} g_i X^{ki}$ with $\deg(g_i) < k$. Then $fg = \sum_i f g_i$: The algorithm computes the $\lceil n/k \rceil$ products $f g_i$ in $2k-1$ extra registers and adds them to the output. If $\mathtt{oFP}$ has time complexity $\mathsf{M}(n)$ and space complexity cn, the time complexity of $\mathtt{oFP}^+_{\mathsf{u}}$ is $\lceil n/k \rceil (\mathsf{M}(k) + 2k - 1)$ and its space complexity $(c+2)k - 1$.

The last step computes $h + fg$ and corresponds to a half-additive full product on inputs of size $n-k$, since only the $n-k-1$ first registers of $\mathsf{R}_{[2k..2n[}$ are filled: Indeed, $\deg(h + f_0 g + \hat{f} g_0 X^k) < n + k - 1$. This last step is thus a recursive call.

In order to make this algorithm run in place, k must be chosen so that the extra memory needed in the two calls to $\mathtt{oFP}^+_{\mathsf{u}}$ fits exactly in the unused part of R. This is the case when

$$(c+2)k - 1 \le 2n - 1 - (n + k - 1)$$

which gives $k \le \frac{n+1}{c+3}$. The resulting algorithm is depicted below.

Algorithm 1 $\mathtt{iFP}^+_{\mathtt{hi}}\mathtt{_from_oFP}$

Input: f and g of size n in the input space, h of size $n-1$ in the output space R
Output: R contains $fg + h$
Required: Full product algorithm $\mathtt{oFP}$ of space complexity $\le cn$

1: **if** $n < c + 2$ **then**
2: $\quad \mathsf{R} \leftarrow \mathsf{R} + fg$ ▹ using a naive algorithm
3: **else**
4: $\quad k \leftarrow \lfloor (n+1)/(c+3) \rfloor$
5: $\quad \mathsf{R}_{[0..n+k-1[} \leftarrow \mathtt{oFP}^+_{\mathsf{u}}(f_0, g, h)$ ▹ work space: $\mathsf{R}_{[n+k-1..2n[}$
6: $\quad \mathsf{R}_{[k..n+k-1[} \leftarrow \mathtt{oFP}^+_{\mathsf{u}}(f, g_0, h + f_0 g)$ ▹ same work space
7: $\quad \mathsf{R}_{[2k..2n[} \leftarrow \mathtt{iFP}^+_{\mathtt{hi}}\mathtt{_from_oFP}(f \operatorname{quo} X^k, g \operatorname{quo} X^k)$

Complexity analysis. The algorithm uses two calls to $\mathtt{oFP}^+_{\mathsf{u}}$ with inputs of sizes (k, n) and $(n-k, k)$ respectively. The total complexity amounts to $\lceil n/k \rceil \mathsf{M}(k) + (\lceil n/k \rceil - 1)\mathsf{M}(k) + (2\lceil n/k \rceil - 1)(2k-1)$ plus a recursive call in size $n-k$. Let $T(n)$ be the complexity of $\mathtt{iFP}^+_{\mathtt{hi}}$, we thus have

$$T(n) = T(n-k) + (2\lceil n/k \rceil - 1)\left[\mathsf{M}(k) + (2k-1)\right].$$

Note that k depends upon n, this implies that the analysis must be done without k. Since $k = \lfloor (n+1)/(c+3) \rfloor$, $\lceil n/k \rceil \le c + 4$ for $n \ge (c+2)(c+4)$. Therefore,

$$T(n) \le T\left(\left\lfloor \tfrac{c+2}{c+3}(n+1) \right\rfloor\right) + (2c+7)\left[\mathsf{M}\left(\left\lfloor \tfrac{n+1}{c+3} \right\rfloor\right) + 2\tfrac{n}{c+3} - \tfrac{c+1}{c+3}\right].$$

Proposition 5.2(i) yields $T(n) \le (2c+7)\mathsf{M}(n) + o(\mathsf{M}(n))$.

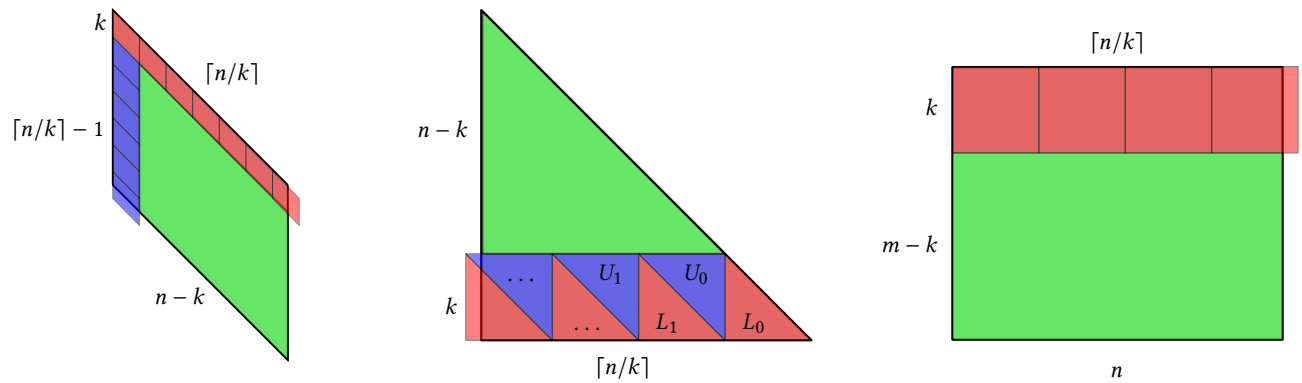

Figure 1: Tilings of the matrices $\mathfrak{M}_{\mathsf{FP}(f)}$ (left), $\mathfrak{M}_{\mathsf{SP}_{\mathsf{lo}}(f)}$ (center) and $\mathfrak{M}_{\mathsf{MP}(f)}$ (right).

5.2 In-place short product algorithm

Our goal is to describe an in-place (low) short product algorithm based on an out-of-place one. It corresponds to tiling $\mathfrak{M}_{\mathsf{SP}_{\mathsf{lo}}(f)}$ given in Eq. (2), see Fig. 1 (center). Let $f = \sum_{i=0}^{n-1} f_i X^i$ and $g = \sum_{i=0}^{n-1} g_i X^i$, and let $h = \sum_{i=0}^{n-1} h_i X^i = \mathsf{SP}_{\mathsf{lo}}(f, g)$. The idea is to fix some $k < n$ and to have two phases. The first phase corresponds to the bottom k rows of $\mathfrak{M}_{\mathsf{SP}_{\mathsf{lo}}(f)}$ and computes h_{n-k} to h_{n-1} using the out-of-place algorithm on smaller polynomials. The second phase corresponds to the top $(n-k)$ rows and is a recursive call to compute h_0 to h_{n-k-1}: Indeed, $h \bmod X^{n-k} = \mathsf{SP}_{\mathsf{lo}}(f \bmod X^{n-k}, g \bmod X^{n-k})$.

For the second phase, we remark that the bottom k rows can be tiled by $\lceil n/k \rceil$ lower triangular matrices (denoted $L_0, \dots, L_{\lceil n/k \rceil - 1}$ from the right to the left), and $\lceil n/k \rceil - 1$ upper triangular matrices (denoted $U_0, \dots, U_{\lceil n/k \rceil - 2}$). One can identify the matrices L_i and U_i as matrices of some low and high short products. More precisely, the coefficients that appear in the lower triangular matrix L_i are the coefficients of degree ki to $k(i+1)-1$ of f. Thus, $L_i = \mathfrak{M}_{\mathsf{SP}_{\mathsf{lo}}(f_{ki,k(i+1)})}$ where $f_{ki,k(i+1)} = \sum_{j=ki}^{k(i+1)-1} f_j X^{j-ki}$. Similarly, $U_i = \mathfrak{M}_{\mathsf{SP}_{\mathsf{hi}}(f_{ki,k(i+1)})}$. The matrices $L_{\lceil n/k \rceil - 1}$ and $U_{\lceil n/k \rceil - 2}$ must be padded if k does not divide n. Altogether, this proves that this part of the computation reduces to $\lceil n/k \rceil$ low short products and $\lceil n/k \rceil - 1$ high short products, in size k.

In order for this algorithm to actually be in place, k must be small enough. If the out-of-place short product algorithm uses ck extra space, since we also need k free registers to store the intermediate results, k must satisfy $n - k \geq (c+1)k$, that is $k \leq n/(c+2)$.

Complexity analysis. The algorithm performs $\lceil n/k \rceil$ low short products and $\lceil n/k \rceil - 1$ high short products plus one recursive call in size $n-k$. Let $\mathsf{M}(k)$ be the complexity of a low short product algorithm. Then the high short product can be computed in time $\mathsf{M}(k-1)$. Let $T(n)$ be the complexity of the recursive algorithm. Then $T(n) = \lceil n/k \rceil \mathsf{M}(k) + (\lceil n/k \rceil - 1)\mathsf{M}(k-1) + (2\lceil n/k \rceil - 1)k + T(n-k)$ (the linear time is for the additions). Since $k = \lfloor n/(c+2) \rfloor$, $\lceil n/k \rceil \leq c+3$ for $n \geq (c+3)(c+2)$ and $n - k \leq \frac{c+1}{c+2} n + 1$. Thus,

$$T(n) \leq (c+3)\mathsf{M}\left(\left\lfloor \frac{n}{c+2} \right\rfloor\right) + (c+2)\mathsf{M}\left(\left\lfloor \frac{n}{c+2} - 1 \right\rfloor\right) + 2n + T\left(\left\lfloor \frac{c+1}{c+2} n + 1 \right\rfloor\right).$$

Proposition 5.2(i) implies $T(n) \leq (2c+5)\mathsf{M}(n) + o(\mathsf{M}(n))$.

Algorithm 2 `iSP_lo_from_oSP`

Input: f and g of size n
Output: R contains $\mathsf{SP}_{\mathsf{lo}}(f, g)$
Required: Two short product algorithms $\mathsf{oSP}_{\mathsf{lo}}$ and $\mathsf{oSP}_{\mathsf{hi}}$ of space complexity $\leq cn$

```
if n < c + 2 then
    R ← SP_lo(f, g)                                  ▷ using a naive algorithm
else
    k ← ⌊n/(c + 2)⌋
    for i = 0 to ⌈n/k⌉ − 1 do                        ▷ work space: R[k..n−k[
        R[0..k[ ← oSP_lo(f_{ki,k(i+1)}, g_{n−k(i+1),n−ki})
        R[n−k..n[ ← R[n−k..n[ + R[0..k[
    for i = 0 to ⌈n/k⌉ − 2 do                        ▷ same work space
        R[0..k[ ← oSP_hi(f_{ki,k(i+1)}, g_{n−k(i+2),n−k(i+1)})
        R[n−k..n[ ← R[n−k..n[ + R[0..k[
    R[0..n−k[ ← iSP_lo_from_oSP(f mod X^{n−k}, g mod X^{n−k})
```

5.3 In-place middle product algorithm

To build an in-place (unbalanced) middle product algorithm, we assume that we have an algorithm for the (balanced) middle product that uses cn extra space for inputs of size $2n-1$ and n respectively.

The in-place algorithm is based on a tiling of $\mathfrak{M}_{\mathsf{MP}(f)}$ given in Eq. (3), see Fig. 1 (right) : The top k rows correspond to the matrix $\mathfrak{M}_{\mathsf{MP}(f \bmod X^k)}$ and the bottom $m-k$ rows to the matrix $\mathfrak{M}_{\mathsf{MP}(f \operatorname{quo} X^k)}$. The algorithm computes $\mathfrak{M}_{\mathsf{MP}(f \bmod X^k)}\vec{g}$ using the out-of-place algorithm and then $\mathfrak{M}_{\mathsf{MP}(f \operatorname{quo} X^k)}\vec{g}$ using a recursive call.

To make this algorithm work in place, the value of k has to be adjusted so that the work space is large enough. The result of a middle product in size k has degree $< k$ and needs ck extra work space by hypothesis. Therefore, if $m - k \geq (c+1)k$, that is $k \leq m/(c+2)$, the computation can be performed in place.

Complexity analysis. Let $\mathsf{M}(k)$ be the cost of an out-of-place balanced middle product algorithm. The cost of an unbalanced middle product is thus $\lceil n/k \rceil \mathsf{M}(k)$ for $k < n$. The in-place algorithm computes first a middle product using an out-of-place algorithm and then makes a recursive call on the remaining part. Note that

Algorithm 3 iMP_from_oMP

Input: f and g of size $n+m-1$ and n respectively
Output: R contains $\mathsf{MP}(f,g)$
Required: Middle product algorithm oMP of space complexity $\le cn$

```
if m < c + 2 then
    R ← oMP(f, g)                          ▷ using a naive algorithm
else
    k ← ⌊m/(c + 2)⌋
    R[0..k[ ← oMP(f mod X^{n+k}, g)        ▷ work space: R[k..m[
    R[k..m[ ← iMP_from_oMP(f quo X^k, g)   ▷ recursive call
```

n does not change during the algorithm and can be viewed as a large constant, while m is the parameter that varies. Then the cost of the algorithm verifies $T(m) \le \lceil n/k \rceil \mathsf{M}(k) + T(m-k)$. Since $k = \lfloor m/(c+2) \rfloor$, $\lceil n/k \rceil < n(c+2)/(m-c-2)+1$ and $m-k \le (c+1)m/(c+2)+1$. Furthermore, $\mathsf{M}(k) \le m/n(c+2)\mathsf{M}(n)$, thus $\lceil n/k \rceil \mathsf{M}(k) \le (m/(m-c-2) + m/n(c+2))\mathsf{M}(n)$. That is,

$$T(m) \le \left(\frac{m}{n(c+2)} + \frac{c+2}{m-c-2} + 1 \right) \mathsf{M}(n) + T\left(\left\lfloor \tfrac{c+1}{c+2} m + 1 \right\rfloor \right).$$

Proposition 5.2(ii) implies $T(n) \le \mathsf{M}(n) \log_{\frac{c+2}{c+1}}(n) + O(\mathsf{M}(n))$ for $m = n$, and $T(n) \le O(\mathsf{M}(n))$ if $\mathsf{M}(n)$ is not quasi-linear.

If $\mathsf{M}(n) \le \lambda n^\gamma$ for some $\gamma > 1$, $\mathsf{M}(k) \le \left(\frac{m}{c+2}\right)^\gamma$ and the recurrence becomes $T(m) \le \left(\frac{n(c+2)}{m-c-2} + 1\right) \lambda \left(\frac{m}{c+2}\right)^\gamma + T(\left\lfloor \frac{c+1}{c+2} m + 1 \right\rfloor)$. Proposition 5.2(iii) implies $T(n) \le O(n^\gamma)$ for $m = n$.

Reduction from short products to middle product. The middle product of f and g can be computed as the sum of the low short product of $f \operatorname{quo} X^n$ with g and the high short product of $f \bmod X^n$ with g. Yet this reduction does not preserve the space complexity since one needs to store the results of the two short products in two zones of size n before summing them. Actually, the reduction given above from oMP to iMP can easily be adapted to a reduction from SP to MP that is space-preserving. Yet, the complexity also worsens with a logarithmic factor. Thus, we cannot conclude that $\mathsf{MP} \le_{\mathsf{TISP}} \mathsf{SP}$.

5.4 Proof of Proposition 5.2

Lemma 5.3. *Let $T(n)$ be a function satisfying $T(n) \le f(n) + T(\lfloor \alpha n + \beta \rfloor)$ for some $\alpha < 1$, β and non-decreasing f. Then*

$$T(n) \le T(\lfloor n_K \rfloor) + \sum_{i=0}^{K-1} f(n_i)$$

where $n_i = \alpha^i n + \beta \frac{1-\alpha^{i+1}}{1-\alpha}$ and $K \le \log_{1/\alpha}(n)$.

Proof. By definition of n_i, $n = n_0$ and $T(\lfloor n_i \rfloor) \le f(n_i) + T(\lfloor n_{i+1} \rfloor)$. Then by recurrence, $T(n) \le T(\lfloor n_{i+1} \rfloor) + \sum_{j=0}^{i} f(n_i)$. □

Lemma 5.4. *Let $n_i = \alpha^i n + \beta \frac{1-\alpha^{i+1}}{1-\alpha}$. Then*

$$\sum_{i=0}^{K-1} n_i \le \frac{n + \beta K}{1-\alpha}.$$

Proof. Since $0 < \alpha < 1$, $\sum_{i=0}^{K-1} \alpha^i < \sum_{i=0}^{\infty} \alpha^i = 1/(1-\alpha)$. Also, $\sum_{i=0}^{K-1} (1-\alpha^{i+1})/(1-\alpha) \le \sum_{i=0}^{K-1} 1/(1-\alpha) = K/(1-\alpha)$. □

Lemma 5.5. *Let $n_i = \alpha^i n + \beta \frac{1-\alpha^{i+1}}{1-\alpha}$. Then*

$$\sum_{i=0}^{K-1} \frac{1}{n_i - \beta/(1-\alpha)} = \frac{\alpha(\alpha^{-K}-1)}{(1-\alpha)n - \alpha\beta}.$$

Proof. Since $n_i = \alpha^i(n - \beta\alpha/(1-\alpha)) + \beta/(1-\alpha)$, $n_i - \beta/(1-\alpha)$ is a multiple of α^i. Thus,

$$\sum_{i=0}^{K-1} \frac{1}{n_i - \beta/(1-\alpha)} = \frac{1}{n - \beta\alpha/(1-\alpha)} \sum_{i=0}^{K-1} \alpha^{-i}.$$

Then, $\sum_i \alpha^{-i} = (1-\alpha^{-K})/(1-1/\alpha) = \alpha(\alpha^{-K}-1)/(1-\alpha)$, and $\sum_i 1/(n_i - \beta/(1-\alpha)) = \alpha(\alpha^{-K}-1)/((1-\alpha)n - \alpha\beta)$. □

Lemma 5.6. *If $\mathsf{M}(n)/n$ is non-decreasing, and $n_i = \alpha^i n + \beta(1-\alpha^{i+1})/(1-\alpha)$ for some $\alpha < 1$, then*

$$\sum_{i=0}^{K-1} \mathsf{M}(\lfloor \lambda n_i + \mu \rfloor) = \frac{\lambda}{1-\alpha} \mathsf{M}(n) + o(\mathsf{M}(n))$$

for $K \le \log_{1/\alpha}(n)$ and any λ and μ such that $\lambda n_i + \mu \le n$ for all n_i.

Proof. Since $\mathsf{M}(n)/n$ is non-decreasing, we get $M(\lfloor \lambda n_i + \mu \rfloor) \le \frac{\lambda n_i + \mu}{n} \mathsf{M}(n)$. Therefore, $\sum_i \mathsf{M}(\lfloor \lambda n_i + \mu \rfloor) \le \mathsf{M}(n)/n \sum_i (\lambda n_i + \mu)$. By Lemma 5.4, $\sum_i \mathsf{M}(\lfloor \lambda n_i + \mu \rfloor) \le \lambda \mathsf{M}(n)/(1-\alpha) + \lambda\beta K \mathsf{M}(n)/n(1-\alpha) + \mu K \mathsf{M}(n)/n$. Since $K = O(\log n)$, $K\mathsf{M}(n)/n = o(\mathsf{M}(n))$. □

Proof of Proposition 5.2. Proof of (i): By hypothesis, we have $T(n) \le \sum_k a_k \mathsf{M}(\lfloor \lambda_k n + \mu_k \rfloor) + bn + c + T(\lfloor \alpha n + \beta \rfloor)$ with $\alpha < 1$ and $\lambda_k n + \mu_k < n$ for all k. By Lemma 5.3, $T(n) \le T(\lfloor n_K \rfloor) + \sum_i f(n_i)$ with n_i defined as in the lemma and $f(n) = \sum_k a_k \mathsf{M}(\lfloor \lambda_k n + \mu_k \rfloor) + bn + c$. Then

$$\begin{aligned}
\sum_{i=0}^{K-1} f(n_i) &= \sum_k a_k \sum_{i=0}^{K-1} \mathsf{M}(\lfloor \lambda_k n_i + \mu_k \rfloor) + b \sum_{i=0}^{K-1} n_i + Kc \\
&\le \sum_k a_k \left(\frac{\lambda_k}{1-\alpha} \mathsf{M}(n) + o(\mathsf{M}(n)) \right) + b \frac{n + \beta K}{1-\alpha} + Kc \\
&= \sum_k \frac{a_k \lambda_k}{1-\alpha} \mathsf{M}(n) + \frac{bn}{1-\alpha} + o(\mathsf{M}(n))
\end{aligned}$$

since $K = o(\mathsf{M}(n))$ and the sum over k is of fixed size.

Proof of (ii): Since $T(m) \le (\lambda m/n + \mu/(m - \frac{1}{1-\alpha}) + 1)\mathsf{M}(n) + T(\lfloor \alpha m + 1 \rfloor)$ with $\alpha < 1$ and $m \le n$, Lemma 5.3 implies

$$T(m) \le T(\lfloor m_K \rfloor) + \mathsf{M}(n) \sum_i \left(\frac{\lambda m_i}{n} + \frac{\mu}{m_i - 1/(1-\alpha)} + 1 \right)$$

where $m_i = \alpha^i m + (1-\alpha^{i+1})/(1-\alpha)$. By Lemma 5.4, $\sum_i m_i \le \frac{m+K}{1-\alpha}$ and by Lemma 5.5, $\sum_i 1/(m_i - \frac{1}{1-\alpha}) \le \alpha^{-K+1}/((1-\alpha)m - \alpha)$. Altogether,

$$\begin{aligned}
T(m) \le T(\lfloor m_K \rfloor) + K\mathsf{M}(n) + \frac{\lambda(m+K)}{n(1-\alpha)} \mathsf{M}(n) \\
+ \frac{\mu\alpha}{1-\alpha} \cdot \frac{(1/\alpha)^K}{m - \alpha/(1-\alpha)} \mathsf{M}(n).
\end{aligned}$$

If we plug $K = \log_{1/\alpha}(m)$ and fix $m = n$, we get

$$T(n) \le T(\lfloor n_K \rfloor) + \mathsf{M}(n) \log_{1/\alpha} n + \frac{\lambda + \mu\alpha}{1-\alpha} \mathsf{M}(n) + o(M(n)).$$

Proof of (iii): Let $T(m) \leq \lambda(n/((1-\alpha)m-1))+1)((1-\alpha)m)^\gamma + T(\lfloor \alpha m+1 \rfloor)$. We claim that there exist constants μ and ν such that $T(m) \leq \mu m^{\gamma-1}n + \nu m^\gamma + o(m^{\gamma-1}n + m^\gamma)$ and prove it by induction. Using the recurrence relation and the induction hypothesis,

$$T(m) \leq \lambda n((1-\alpha)m)^{\gamma-1} + \lambda((1-\alpha)m)^\gamma + \mu(\alpha m)^{\gamma-1}n + \nu(\alpha m)^\gamma + o(m^{\gamma-1}n + m^\gamma)$$

The result follows as soon as $\lambda(1-\alpha)^{\gamma-1} + \mu\alpha^{\gamma-1} \leq \mu$ and $\lambda(1-\alpha)^\gamma + \nu\alpha^\gamma \leq \nu$. We can thus fix

$$\mu = \frac{\lambda(1-\alpha)^{\gamma-1}}{1-\alpha^{\gamma-1}} \text{ and } \nu = \frac{\lambda(1-\alpha)^\gamma}{1-\alpha^\gamma}.$$

Finally, taking $m = n$, we conclude that $T(n) \leq (\mu+\nu)n^\gamma + O(n^{\gamma-1})$. □

6 PERSPECTIVES

We have presented algorithms for polynomial products which are efficient in terms of both time and space. Our results show that any algorithm for the full and short products can be turned into another algorithm with the same asymptotic time complexity while using only $O(1)$ extra space. We obtain similar results for the middle product but only proved it for algorithms that do not have a quasi-linear time complexity. In the latter case, an increase of the time complexity by a logarithmic factor occurs. We provided analysis of our reductions that make their constants explicit. In particular, their values ensure that our reductions are practicable.

In a future work, we plan to address some remaining issues. By examining the constants in the already known algorithms, we can choose the algorithms to use as starting points of our reductions to optimize the complexity. For instance three variants of Karatsuba's algorithm with different time and space complexities are known [17, 20, 24]. Furthermore, it seems possible to improve on the complexity of low-space versions of Karatsuba's and Toom-Cook's algorithm, yielding faster in-place algorithms through our reductions. Another promising approach is to slightly relax the model of computation and work in model in which one can write on the input space as long as the original inputs are restored by the end of the computation. Preliminary results for Karatsuba's algorithm suggest that this could also yield a lower constant in the time complexity.

Finally, we have started to explore the design of in-place algorithms for a broader range of problems of polynomials, such as division or evaluation/interpolation. The use of in-place middle and short products becomes crucial since one needs to avoid any increase in the size of the intermediate results.

ACKNOWLEDGMENTS

This work was begun while the last author was graciously hosted by the LIRMM at the Université Montpellier. This work was supported in part by the National Science Foundation under grants 1319994 (http://www.nsf.gov/awardsearch/showAward?AWD_ID=1319994), 1618269 (http://www.nsf.gov/awardsearch/showAward?AWD_ID=1618269); and also by the Région Occitanie.

REFERENCES

[1] K. Abrahamson. 1986. Time-space tradeoffs for branching programs contrasted with those for straight-line programs. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 402–409. https://doi.org/10.1109/SFCS.1986.58
[2] Sanjeev Arora and Boaz Barak. 2009. *Computational Complexity: A Modern Approach* (1st ed.). Cambridge University Press.
[3] Alin Bostan, Frédéric Chyzak, Marc Giusti, Romain Lebreton, Grégoire Lecerf, Bruno Salvy, and Éric Schost. 2017. *Algorithmes Efficaces en Calcul Formel* (1 ed.).
[4] A. Bostan, G. Lecerf, and É. Schost. 2003. Tellegen's Principle into Practice. In *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation (ISSAC '03)*. ACM, 37–44. https://doi.org/10.1145/860854.860870
[5] Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. 2009. Memory Efficient Scheduling of Strassen-Winograd's Matrix Multiplication Algorithm. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation (ISSAC '09)*. ACM, 55–62. https://doi.org/10.1145/1576702.1576713
[6] Richard Brent and Paul Zimmermann. 2010. *Modern Computer Arithmetic*. Cambridge University Press, New York, NY, USA.
[7] David G. Cantor and Erich Kaltofen. 1991. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica* 28 (1991), 693–701. Issue 7. https://doi.org/10.1007/BF01178683
[8] Yiping Cheng. 2016. Space-Efficient Karatsuba Multiplication for Multi-Precision Integers. http://arxiv.org/abs/1605.06760.
[9] Stephen A. Cook. 1966. *On the minimum computation time of functions*. Ph.D. Dissertation. Harvard University.
[10] J. von zur Gathen and J. Gerhard. 2013. *Modern Computer Algebra* (3rd ed.). Cambridge University Press.
[11] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann. 2004. The Middle Product Algorithm I. *Applicable Algebra in Engineering, Communication and Computing* 14, 6 (01 Mar 2004), 415–438. https://doi.org/10.1007/s00200-003-0144-2
[12] G. Hanrot and P. Zimmermann. 2004. A long note on Mulders' short product. *Journal of Symbolic Computation* 37, 3 (2004), 391–401. https://doi.org/10.1016/j.jsc.2003.03.001
[13] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. 2017. Faster Polynomial Multiplication over Finite Fields. *J. ACM* 63, 6 (Jan. 2017), 52:1–52:23. https://doi.org/10.1145/3005344
[14] David Harvey and Daniel S. Roche. 2010. An in-place truncated Fourier transform and applications to polynomial multiplication. In *ISSAC '10: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*. ACM, 325–329. https://doi.org/10.1145/1837934.1837996
[15] David Harvey and Joris Van Der Hoeven. 2019. Polynomial multiplication over finite fields in time O(n log n). (March 2019). https://hal.archives-ouvertes.fr/hal-02070816 preprint.
[16] Erich Kaltofen. 2000. Challenges of symbolic computation: my favorite open problems. *Journal of Symbolic Computation* 29, 6 (2000), 891–919.
[17] Anatolii Karatsuba and Yuri Ofman. 1963. Multiplication of Multidigit Numbers on Automata. *Soviet Physics-Doklady* 7 (1963), 595–596.
[18] Andrea Lincoln, Virginia Vassilevska Williams, Joshua R. Wang, and R. Ryan Williams. 2016. Deterministic Time-Space Trade-Offs for k-SUM. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi (Eds.), Vol. 55. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 58:1–58:14. https://doi.org/10.4230/LIPIcs.ICALP.2016.58
[19] Thom Mulders. 2000. On Short Multiplications and Divisions. *Applicable Algebra in Engineering, Communication and Computing* 11, 1 (2000), 69–88. https://doi.org/10.1007/s002000000037
[20] Daniel S. Roche. 2009. Space- and Time-efficient Polynomial Multiplication. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation (ISSAC '09)*. ACM, 295–302. https://doi.org/10.1145/1576702.1576743
[21] John Savage and Sowmitri Swamy. 1979. Space-time tradeoffs for oblivious integer multiplication. In *Automata, Languages and Programming*, Hermann Maurer (Ed.). Lecture Notes in Computer Science, Vol. 71. Springer Berlin / Heidelberg, 498–504. https://doi.org/10.1007/3-540-09510-1_40
[22] A. Schönhage and V. Strassen. 1971. Schnelle Multiplikation großer Zahlen. *Computing* 7 (1971), 281–292. Issue 3. https://doi.org/10.1007/BF02242355
[23] Chen Su and Haining Fan. 2012. Impact of Intel's new instruction sets on software implementation of GF(2)[x] multiplication. *Inform. Process. Lett.* 112, 12 (2012), 497–502. https://doi.org/10.1016/j.ipl.2012.03.012
[24] Emmanuel Thomé. 2002. Karatsuba multiplication with temporary space of size ≤ n. (2002). online.
[25] Virginia Vassilevska Williams. 2018. On some fine-grained questions in algorithms and complexity. In *Proceedings ICM*, Vol. 3. 3431–3472.