Fast integer multiplication using generalized Fermat primes

Svyatoslav Covanov, Emmanuel Thomé

FAST INTEGER MULTIPLICATION USING GENERALIZED FERMAT PRIMES

SVYATOSLAV COVANOV AND EMMANUEL THOMÉ

Abstract. For almost 35 years, Schönhage-Strassen's algorithm has been the fastest algorithm known for multiplying integers, with a time complexity $O(n \cdot \log n \cdot \log\log n)$ for multiplying n-bit inputs. In 2007, Fürer proved that there exists $K > 1$ and an algorithm performing this operation in $O(n \cdot \log n \cdot K^{\log^* n})$. Recent work by Harvey, van der Hoeven, and Lecerf showed that this complexity estimate can be improved in order to get $K = 8$, and conjecturally $K = 4$. Using an alternative algorithm, which relies on arithmetic modulo generalized Fermat primes (of the form $r^{2^\lambda} + 1$), we obtain conjecturally the same result $K = 4$ via a careful complexity analysis in the deterministic multitape Turing model.

1. Introduction

The first nontrivial algorithm for multiplying n-bit integers is Karatsuba's divide-and-conquer algorithm [KO63], which reaches the complexity $O(n^{\log_2 3})$, with $\log_2$ denoting the logarithm in base 2. The Karatsuba algorithm can be viewed as a simple case of a more general evaluation-interpolation paradigm. In the form of the Toom-Cook algorithm [Too63], this paradigm can be extended so as to reach the complexity $O(n^{1+\epsilon})$ for any $\epsilon > 0$.

The first algorithm to achieve what is called *quasi-linear* complexity is Schönhage and Strassen's [SS71, Sch82]. First, the Schönhage-Strassen algorithm uses the fast Fourier transform (FFT) as a means to quickly evaluate a polynomial at the powers of a primitive root of unity [vzGG99, §8]. Second, the complexity is obtained by an appropriate choice of a ring $\mathcal{R}$ in which this evaluation is to be carried out. Namely, the choice $\mathcal{R} = \mathbb{Z}/(2^t + 1)$, for t a suitable power of two, yields the complexity $O(n \cdot \log n \cdot \log\log n)$, while other natural choices for $\mathcal{R}$ appeared to yield inferior performance at the time.

In 2007, M. Fürer observed that the ring $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$, for P a suitable power of two, is particularly interesting [Für09]. Using this ring $\mathcal{R}$, it is possible to take advantage of large-radix FFT to obtain the improved complexity $O(n \cdot \log n \cdot 2^{O(\log^* n)})$ (in contrast, radix-2 FFT is sufficient for the Schönhage-Strassen algorithm). The notation $\log^*$ denotes the iterated logarithm (see §2.1). Fürer's result was an acclaimed improvement on the complexity of the Schönhage-Strassen algorithm which had remained unbeaten for 35 years.

Fürer's algorithm, as it stands, is perceived as a theoretical result. The last decade has seen various articles explore potential improvements on Fürer's work, either meant to make the complexity more explicit, or to provide possibly more practical variants. An early extension of Fürer's work, proposed in [DKSS08], replaces the field $\mathbb{C}$ in the definition of $\mathcal{R}$ by a p-adic ring and reaches an identical asymptotic complexity. This p-adic variant can be expected to ease precision issues for potential implementations. Harvey, van der Hoeven and Lecerf in [HvdHL16], and later Harvey and van der Hoeven in [HvdH16] propose new algorithms and a

2010 *Mathematics Subject Classification.* Primary 68W30; Secondary 11A41.

sharper complexity analysis that allows one to make the complexity more explicit, namely $O(n \cdot \log n \cdot 8^{\log^* n})$ and even $O(n \cdot \log n \cdot 4^{\log^* n})$ conjecturally. In comparison, they also show that a careful analysis of Fürer's original algorithm reaches the complexity $O(n \cdot \log n \cdot 16^{\log^* n})$

This article presents another variant of Fürer's algorithm. Our algorithm reaches the complexity $O(n \cdot \log n \cdot 4^{\log^* n})$ and relies on a conjecture which can be regarded as an explicit version of the Bateman-Horn conjecture [BH62], supported by numerical evidence. Namely, our assumption is as follows.

Hypothesis 4.5. *Let $\lambda \geq 2$ be an integer. For any real number R such that $2^\lambda \leq R \leq 2^{2\lambda}$, there exists a generalized Fermat prime $p = r^{2^\lambda} + 1$ such that $R \leq r < \lambda^{2.5} R$.*

The key concept of our algorithm is the use of a chain of generalized Fermat primes (of the form $r^{2^\lambda}+1$) to handle recursive calls. We therefore differ significantly from the approach followed by Harvey, van der Hoeven and Lecerf in [HvdHL16, HvdH16]. In a sense however, some lineage can be drawn between our work and an early article by Fürer [Für89] (from 1989), which is dependent on the assumption that there exist infinitely many Fermat primes. The latter assumption, however, is widely believed to be wrong, so our variant fills a gap here.

The way we obtain a complexity formula with $4^{\log^* n}$ and not $16^{\log^* n}$ as for Fürer's algorithm is original. In fact, two improvements stack onto one another. First, we encode integers to be multiplied as integers modulo generalized Fermat primes, and not as polynomials. This saves a factor of two in the sizes of intermediate products. Second, generalized Fermat primes allow to avoid the Kronecker substitution, and therefore we use less padding in the intermediate products.

This article is organized as follows. Section 2 reviews classical facts about quasi-linear integer multiplication algorithms. Fürer's algorithm in particular is introduced in Section 3. Section 4 studies generalized Fermat primes, and their relation to the Bateman-Horn conjecture. We then proceed to define a chain of generalized Fermat primes which is crucial to tackle sizes above a certain threshold. Section 5 uses material developed in the previous sections and presents our new algorithm (in fact two algorithms), with the corresponding recursive complexity equations. We derive an asymptotic complexity estimate in Section 6. Section 7 discusses how practical our algorithm could be, and proposes projected timings. Appendix A gives the proof of Proposition 4.3.

2. Background

2.1. **Notations.** Throughout the article, $\log_2 x$ denotes the logarithm in base 2, and $\log x$ denotes the natural logarithm. We use the notation $\log^{(m)}$ to denote the m-th iterate of the log function, so that $\log^{(m+1)} = \log \circ \log^{(m)}$ (and likewise for $\log_2$).

We denote by $\log^*$ the iterated logarithm function, defined recursively by $\log^* x = 0$ for any real number $x \leq 1$, and by $\log^* x = 1 + \log^*(\log x)$ for $x \geq 1$.

The notation $[\![u, v]\!]$ denotes the set of integers x such that $u \leq x \leq v$.

The notation $u = \Theta(v)$ denotes: ($u = O(v)$ and $v = O(u)$).

2.2. **Integers to polynomials.** Let a and b be positive n-bit integers to be multiplied and $c = ab$. Standard substitution techniques (see e.g. [Ber01]) allow one to compute c via the computation of the product $C(x) = A(x)B(x)$, where A and B are univariate polynomials related to a and b. Polynomials are taken over some well-chosen ring $\mathcal{R}$. Such a procedure is described in Algorithm 1, where we highlight the possibility of computing the product $C(x) = A(x)B(x)$ by multipoint

evaluation and interpolation if the ring $\mathcal{R}$ in which computations take place provides a nice and sufficiently large set of interpolation points. (In this section, we do not explicitly fix a choice for $\mathcal{R}$. We will do so later on in this article.)

Algorithm 1 Multiply in $\mathbb{Z}$ via multipoint evaluation of polynomials

function MultiplyIntegersViaMultipointEvaluation$(a, b, \eta, \mathcal{R})$
  Input: a, b two positive n-bit integers;
  η a power of two; we let $N = \lceil 2n/\log_2 \eta \rceil$
  $\mathcal{R}$ a ring where integers below $N\eta^2$ are unambiguously represented
  $\mathcal{S} \subset \mathcal{R}$ a set of N evaluation points
  Output: $c = a \cdot b$
  Let $A(x) \in \mathbb{Z}[x]$, with all coefficients in $[0, \eta)$, be such that $A(\eta) = a$.
  Define $B(x)$ likewise.
  $\hat{A} \leftarrow$ MultiEvaluation$(A, \mathcal{S})$; define $\hat{B}$ likewise.
  $\hat{C} \leftarrow$ PointwiseProduct$(\hat{A}, \hat{B})$.
  $C \leftarrow$ Interpolation$(\hat{C}, \mathcal{S})$
  Reinterpret C as a polynomial in $\mathbb{Z}[x]$.
  return $c = C(\eta)$.
end function

The procedure followed by Algorithm 1 is in fact quite general, and can be applied to a wider range of bilinear operations than just integer multiplication. For example, one can imitate this algorithm to multiply polynomials or power series in various rings, or to compute other operations such as middle products or dot products. The latter example of the dot product is archetypal of the situation where results of the MultiEvaluation step (as e.g. $\hat{A}$ in Algorithm 1) are used more than once. The conditions on $\mathcal{R}$ that are used to guard against possible overflow must be adjusted accordingly.

2.3. **Cooley-Tukey FFT.** We now discuss how multi-evaluation can be performed efficiently. This depends first and foremost on the number of evaluation points N and on the ring $\mathcal{R}$. FFT algorithms are special-purpose algorithms adapted to evaluation points chosen among roots of unity in $\mathcal{R}$. In order to allow $\mathcal{R}$ to be a non-integral ring, we need the following definition.

Definition 2.1. Let $N \geq 1$ be an integer, and $\mathcal{R}$ be a ring of characteristic zero or characteristic coprime to N, containing an N-th root of unity ω. We say that ω is a principal N-th root of unity if $\forall i \in [\![1, N-1]\!], \sum_{j=0}^{N-1} \omega^{ij} = 0$.

The notion of principal root of unity is stricter than the classical notion of primitive root, and provides the suitable generalization to non-integral rings. For example in $\mathbb{C} \times \mathbb{C}$, the element $(1, i)$ is a primitive 4-th root of unity but not a principal 4-th root of unity.

Using the set of powers of ω as a set of evaluation points, we define the discrete Fourier transform (DFT).

Definition 2.2 (Discrete Fourier Transform (DFT))**.** Let $\mathcal{R}$ be a ring with ω a principal N-th root of unity. The DFT of length N and base root ω over $\mathcal{R}$ is the ring isomorphism $\mathrm{DFT}_{N,\omega}$ defined as:

$$\begin{cases} \mathcal{R}[x]/(x^N - 1) & \to & \mathcal{R}[x]/(x-1) \times \mathcal{R}[x]/(x-\omega) \times \cdots \times \mathcal{R}[x]/(x-\omega^{N-1}) \\ P & \mapsto & (P(1), P(\omega), \ldots, P(\omega^{N-1})). \end{cases}$$

We customarily write a DFT of length N of a polynomial P as the polynomial $\hat{P}$ of degree at most $N-1$ defined as

$$\hat{P} = \mathrm{DFT}_{N,\omega}(P) = P(1) + XP(\omega) + \cdots + X^{N-1}P(\omega^{N-1}).$$

Cooley and Tukey showed in [CT65] how a DFT of composite order $N = N_1N_2$ can be computed. This algorithm is also sometimes called "matrix Fourier algorithm", alluding to the fact that it performs N_2 "column-wise" transforms of length N_1, followed by N_1 "row-wise" transforms of length N_2. It is described in Algorithm 2. We note that Algorithm 2 implicitly rearranges data (e.g. when computing B_j and S_i), and some work is needed to perform the required matrix transpositions in a satisfactory way on a multitape Turing machine. Using an algorithm proposed in [BGS07], it is shown in [HvdHL16, §2] that this extra cost is small enough that it is subsumed within the cost of multiplications by roots of unity in $\mathcal{R}$.

Algorithm 2 General Cooley-Tukey FFT of order $N = N_1N_2$

function CooleyTukeyFFT(N_1,N_2,ω,A)
  Input: $A = \sum_{i=0}^{N-1} a_iX^i \in \mathcal{R}[X]/(X^N - 1)$;
   ω a principal N-th root of unity. Let $\omega_1 = \omega^{N_2}$ and $\omega_2 = \omega^{N_1}$.
  Output: $\hat{A} = \mathrm{DFT}_{N,\omega}(A) = A(1) + A(\omega)X + \cdots + A(\omega^{N-1})X^{N-1}$
  Let $(B_j(X))_j \in \mathcal{R}[X]^{N_2}$ be such that $A(X) = \sum_{j<N_2} B_j(X^{N_2})X^j$
  for $j \in [\![0, N_2 - 1]\!]$ **do**
    $\hat{B}_j \leftarrow \mathrm{DFT}_{N_1,\omega_1}(B_j)$ ▷ $\omega_1 = \omega^{N_2}$ is a principal N_1-st root
    $\Gamma_j \leftarrow \hat{B}_j(\omega^j X)$ ▷ $(\Gamma_j)_j$ are "twisted transforms" of the $(B_j)_j$
  end for
  Let $(S_i(Y))_i \in \mathcal{R}[Y]^{N_1}$ be such that $\sum_{j<N_2} \Gamma_j(X)Y^j = \sum_{i<N_1} S_i(Y)X^i$
  for $i \in [\![0, N_1 - 1]\!]$ **do**
    $\hat{S}_i \leftarrow \mathrm{DFT}_{N_2,\omega_2}(S_i)$ ▷ $\omega_2 = \omega^{N_1}$ is a principal N_2-nd root
  end for
  return $\sum_{i<N_1} \hat{S}_i(X^{N_1})X^i$
end function

The notation $\mathrm{DFT}_{N,\omega}$ denotes a mathematical object rather than an algorithm. Therefore, we need to detail how recursive computations of $\mathrm{DFT}_{N_1,\omega_1}$ and $\mathrm{DFT}_{N_2,\omega_2}$ are handled in Algorithm 2. Two approaches are rather typical instantiations of the Cooley-Tukey algorithm when the length N is a power of two:

- "radix-two FFT": For a length $N = 2^k$, compute $N_2 = 2^{k-1}$ transforms of length $N_1 = 2$ (often called "butterflies"), then recurse with two transforms of length 2^{k-1}. We use the notation Radix2FFT(N, ω, A) for this algorithm.
- "large-radix FFT": More generally, for a length $N = 2^{uq+r}$ with $r < u$, and $q > 0$, compute $N_2 = N/2^u$ transforms of length $N_1 = 2^u$, then recurse with transforms of length $N_2 = N/2^u$. When all recursive calls are unrolled, we see that the computation is based on transforms of length $N_1 = 2^u$ (or $N = 2^r$ at the very end of the recursion). Those are done with Radix2FFT. We use the notation LargeRadixFFT$(N, \omega, 2^u, A)$ for this algorithm.

It is clear that the latter approach specializes to the former when $u = 1$.

Large-radix FFT is often used for practical purposes, as it typically improves application performance. As we observe later on in this article, this has a stronger impact in the context of Fürer's algorithm, since the overall complexity is very dependent on this technique.

The computational interest of using FFT algorithms for multi-evaluation follows from the count $C(N)$ of operations in $\mathcal{R}$ that are required for an FFT of length

$N = 2^k$. Using radix 2^u as an example (u being a constant), we have $C(N)/N = C(2^u)/2^u + C(N/2^u)/(N/2^u) + O(1)$, from which it follows that asymptotically we have $C(N) = O(N \log_2 N)$.

Two additional comments are worth mentioning. First, we define a similar isomorphism, denoted Half-DFT$_{N,\omega}$, by the multi-evaluation at odd powers of a $2N$-th root of unity ω:

$$\begin{cases} \mathcal{R}[x]/(x^N+1) & \to & \mathcal{R}[x]/(x-\omega) \times \mathcal{R}[x]/(x-\omega^3) \times \cdots \times \mathcal{R}[x]/(x-\omega^{2N-1}) \\ P & \mapsto & (P(\omega), P(\omega^3), \ldots, P(\omega^{2N-1})). \end{cases}$$

A half-DFT of length N can be computed at the same cost as a DFT of length N, plus N extra multiplications for scaling. More precisely, to multi-evaluate $P(X)$ (an element of $\mathcal{R}[x]/(x^N+1)$) at $\omega, \omega^3, \ldots, \omega^{2N-1}$, we compute Half-DFT$_{N,\omega}(P(X)) =$ DFT$_{N,\omega^2}(P(\omega X))$. Half-DFTs are used for polynomial products modulo $X^N + 1$, as opposed to $X^N - 1$. Such convolutions are called *negacyclic*.

Also, it is straightforward to verify that the task of interpolating a polynomial A from its multi-evaluation $\hat{A}$ can be done with essentially the same algorithm (see e.g. [vzGG99, §8]). The inverse transforms are written as

$$\text{IFT}_{N,\omega}(A(X)) = \frac{1}{N} \text{DFT}_{N,\omega^{-1}}(A(X)),$$
$$\text{Half-IFT}_{N,\omega}(A(X)) = \frac{1}{N} \text{IFT}_{N,\omega^2}(A(X))(\omega^{-1}X).$$

We shall not discuss this point further.

2.4. **Complexity of integer multiplication.**

Notation 2.3. *We denote by* $\mathsf{M}(n)$ *the cost of the multiplications of two* n*-bit integers in the deterministic multitape Turing model* [Pap94]*, also called bit complexity.*

By combining the evaluation-interpolation scheme of §2.2 with FFT-based multi-evaluation and interpolation as in §2.3, we obtain quasi-linear integer multiplication algorithms. We identify several tasks whose cost contributes to the bit complexity of such algorithms.

- conversion of the input integers to polynomials in $\mathcal{R}[X]$;
- multiplications by roots of unity in the FFT computation;
- linear operations in the FFT computation (additions, etc);
- pointwise products of elements of $\mathcal{R}$.
- recovery of the resulting integer from the computed polynomial.

Algorithm 1 chooses η a power of two so that the first and last steps above have linear complexity (at least provided that elements in $\mathcal{R}$ are represented in a straightforward way). If we go into more detail, $\mathsf{M}(n)$ then expresses as $\mathsf{M}(n) = C(N) \cdot K_{\text{FFT}}(\mathcal{R}) + N \cdot K_{\text{PW}}(\mathcal{R}) + O(n)$, with the following notations.

- $K_{\text{FFT}}(\mathcal{R})$ denotes the cost for the multiplication by powers of ω in $\mathcal{R}$ that occur within the FFT computation.
- $K_{\text{PW}}(\mathcal{R})$ denotes the binary cost for the pointwise products in $\mathcal{R}$.

The costs $K_{\text{PW}}(\mathcal{R})$ and $K_{\text{FFT}}(\mathcal{R})$ are not necessarily equal. Of course, both may involve recursive calls to fast multiplication algorithms.

2.5. **Choice of the base ring.**

Depending on $\mathcal{R}$, the bit complexity estimates of §2.4 can be made more precise. Some rings have special roots of unity that allow faster operations (multiplication, in $\mathcal{R}$, most importantly) than others. Several choices for $\mathcal{R}$ are discussed in [SS71]. We describe their important characteristics when the goal is to multiply two n-bit integers.

The choice $\mathcal{R} = \mathbb{C}$ might seem natural because roots of unity are plenty. The precision required calls for some analysis.

- A precision of $t = \Theta(\log_2 n)$ bits is compatible with a transform length $N = \Theta(n/\log_2 n)$ (see [SS71, §3]), in the sense that the polynomials that we multiply can be represented on tN bits and the product would not be correct if t were smaller (thus, $t = \Theta(\log_2 n)$ is optimal).
- Costs for operations in $\mathcal{R}$ are $K_{\mathrm{FFT}}(\mathcal{R}) = K_{\mathrm{PW}}(\mathcal{R}) = O(\mathsf{M}(\log_2 n))$.

This yields $\mathsf{M}(n) = O(N \log_2 N \cdot \mathsf{M}(\log_2 n)) = O(n \cdot \mathsf{M}(\log_2 n))$, so that

$$\mathsf{M}(n) = 2^{O(\log_2^* n)} \cdot n \cdot \log_2 n \cdot \log_2^{(2)} n \cdot \log_2^{(3)} n \cdot \ldots,$$

where the number of recursive calls is $\log_2^* n + O(1)$.

Schönhage and Strassen (originally in [SS71], later changed to a simpler variant in [Sch82]) proposed the alternative $\mathcal{R} = \mathbb{Z}/(2^t + 1)\mathbb{Z}$, in which 2 is a principal $2t$-th root of unity. Their algorithm multiplies n-bit integers modulo $2^n + 1$, for suitable n (to fix ideas, take n a power of two). This algorithm can be adapted to the general integer multiplication by multiplying n-bit integers modulo $2^{2n} + 1$.

- We pick a transform length N slightly below $\sqrt{n}$, and divide both inputs in chunks of $\lceil n/N \rceil$ bits.
- We choose the ring $\mathcal{R} = \mathbb{Z}/(2^t + 1)\mathbb{Z}$ with t subject to several constraints, namely that t be a multiple of N, and that $t \geq 2n/N + \log_2 N + O(1)$. The algorithm uses a negacyclic convolution in $\mathcal{R}[X]/(X^N + 1)$.
- The cost $K_{\mathrm{FFT}}(\mathcal{R})$ is linear in t, as all multiplications by power of ω reduce to binary shifts. We thus have $K_{\mathrm{FFT}}(\mathcal{R}) = O(\sqrt{n})$.
- The cost $K_{\mathrm{PW}}(\mathcal{R})$ is the cost of a recursive multiplication modulo $2^t + 1$. Thus, $K_{\mathrm{PW}}(\mathcal{R}) = \mathsf{M}(t)$.

For the complexity analysis, write $m(n) = \mathsf{M}(n)/(n \log_2 n)$. We then have $m(n) = O(1) + (1 + o(1)) \cdot \frac{2 \log_2 t}{\log_2 n} m(t)$. Dealing with $(1 + o(1))$ with due care (see in particular [Sch82]), we eventually obtain $\mathsf{M}(n) = O(n \cdot \log n \cdot \log\log n)$.

3. Fürer-type bounds

The choices mentioned in §2.5 have orthogonal advantages and drawbacks. The complex field allows larger transform length, shorter recursion size, but suffers, when looking at the cost $K_{\mathrm{FFT}}(\mathbb{C})$, from expensive roots of unity. Those account for the term $\log_2 n \cdot \log_2^{(2)} n \cdots \log_2^{(\log^* n)} n$ in the complexity of the multiplication of n-bit integers using this base ring.

Fürer proposed two distinct algorithms: one in [Für89] and, some 20 years later, in [Für09]. The scheme proposed in [Für89] relies on the assumption that there exist infinitely many Fermat primes, which is unfortunately widely believed to be wrong. We briefly review here the algorithm proposed later in [Für09].

3.1. A ring with convenient roots of unity.

Fürer proposed in [Für09] to use the ring $\mathcal{R} = \mathbb{C}[x]/(x^{2^\lambda} + 1)$, which has a natural principal $2^{\lambda+1}$-th root of unity, namely x. Notice that $\mathcal{R}$ is also isomorphic to $\prod_{j=0}^{2^\lambda - 1} \mathcal{R}_j$, where the component $\mathcal{R}_j$ is $\mathbb{C}[x]/(x - \exp((2j+1)i\pi/2^\lambda))$. For any integer N which is a multiple of $2^{\lambda+1}$ (and in particular for powers of two of higher order), we define ω_N as the unique element of $\mathcal{R}$ that maps to $\exp(2(2j+1)i\pi/N)$ in $\mathcal{R}_j$. Lagrange interpolation can be used to compute ω_N explicitly. We verify easily that:

- ω_N is a principal N-th root of unity.
- $\omega_N^{N/2^{\lambda+1}}$ maps to $x = \exp((2j+1)i\pi/2^\lambda)$ in $\mathcal{R}_j$, so that $\omega_N^{N/2^{\lambda+1}} = x$ in $\mathcal{R}$.

The latter point implies that among powers of ω_N, some enjoy particularly easy operations.

Consider now how an FFT of length $N = 2^{(\lambda+1)\cdot q+r}$ can be computed with Algorithm 2 (CooleyTukeyFFT). For $q > 0$, we write $N = N_1 N_2$ with $N_1 = 2^{\lambda+1}$. This way, Algorithm 2 calls an external algorithm (say, radix-two FFT) for the transform of length $N_1 = 2^{\lambda+1}$, and calls itself recursively for the transform of length N_2. The key observation is that in the many transforms of length N_1 that are computed within the recursion, multiplications by roots of unity are then multiplications by powers of $x \in \mathcal{R}$, and therefore inexpensive. We can count the remaining multiplications that occur within the recursion. We call them "expensive" although in truth some might actually be accidentally cheap. Those correspond to the scaling operation $\Gamma_j \leftarrow \hat{B}_j(\omega^j X)$ in Algorithm 2. Their count $E(N)$ satisfies $E(N) = 2^{\lambda+1}E(N/2^{\lambda+1}) + N$, from which it follows that $E(N) = N(\lceil \log_{2^{\lambda+1}} N \rceil - 1)$.

3.2. **Impact on the complexity of integer multiplication.** To multiply integers of at most n bits, where n is a power of two, Fürer selects $2^\lambda = 2^{\lceil \log_2^{(2)} n \rceil}$ and proves that precision $O(\log n)$ is sufficient for the coefficients of the elements of $\mathcal{R}$ that occur in the computation. The integers to be multiplied are split into pieces of $2^{2\lambda-1}$ bits. Each piece of $2^{2\lambda-1}$ bits is transformed into a polynomial of degree $2^{\lambda-1}$ whose coefficients are encoded on 2^λ bits. These polynomials are seen as elements of $\mathcal{R}$. Moreover, the transform length is $N \le 4n/\log_2^2 n$. This decomposition is described in Algorithm 3 (FurerComplexMul)[1].

Algorithm 3 Multiplication of integers with Fürer's algorithm

1: **function** FurerComplexMul(a,b,n)
2: **Input:** a and b two positive n-bit integers, where n is a power of two
3: **Output:** $a \cdot b \mod 2^{2n}+1$
4: Let $\lambda = \lceil \log_2^{(2)} n \rceil$, $\eta = 2^{2^{2\lambda-1}}$, $N = 2n/\log_2 \eta = n/2^{2\lambda-2}$,
5: Let $\mathcal{R} = \mathbb{C}[x]/(x^{2^\lambda}+1)$, and $\omega = \omega_{2N}$ as in §3.1.
6: Let $A_0(X) \in \mathbb{Z}[X]$, with all coefficients in $[0, \eta)$, be such that $A_0(\eta) = a$.
7: Let $\tilde{A}(X, x) \in \mathbb{C}[X, x]$, with all coefficients integers in $[0, 2^{2^\lambda})$, be such that $\tilde{A}(X, 2^{2^\lambda}) = A_0(X)$. ▷ $\deg_X \tilde{A} < N/2$, $\deg_x \tilde{A} < 2^{\lambda-1}$.
8: Define $B_0(X)$ and $\tilde{B}$ likewise.
9: Map $\tilde{A}$ and $\tilde{B}$ to polynomials A and B in $\mathcal{R}[X]/(X^N+1)$
10: $\hat{A} \leftarrow$ LargeRadixFFT$(N, \omega^2, 2^{\lambda+1}, A(\omega X))$ ▷ $\hat{A} = \text{Half-DFT}_{N,\omega}(A)$
11: $\hat{B} \leftarrow$ LargeRadixFFT$(N, \omega^2, 2^{\lambda+1}, B(\omega X))$ ▷ $\hat{B} = \text{Half-DFT}_{N,\omega}(B)$
12: $\hat{C} \leftarrow$ PointwiseProduct$(\hat{A}, \hat{B})$
13: $C \leftarrow \frac{1}{N}$LargeRadixFFT$(N, \omega^{-2}, 2^{\lambda+1}, \hat{C})(\omega^{-1}X)$ ▷ $C = \text{Half-IFT}_{N,\omega}(\hat{C})$
14: Lift C to $\tilde{C} \in \mathbb{C}[X, x]$ with $\deg_X \tilde{C} < N$, $\deg_x \tilde{C} < 2^\lambda$, and integer coefficients (rounding if necessary).
15: **return** $\tilde{C}(\eta, 2^{2^\lambda})$
16: **end function**

Some non-trivial multiplications by elements of $\mathcal{R}$ are needed in Algorithm 3: $3E(N)$ multiplications in recursive calls, and $4N$ multiplications by scaling factors (because of the negacyclic convolution) and pointwise products. For these, we use Kronecker substitution: we encode elements of $\mathcal{R}$ as integers of bit length $O((\log_2 n)^2)$, and then call recursively FurerComplexMul. Other multiplications by roots of unity are cheap. Their number is $O(N \log N)$, and their cost is linear in the

[1]In line 14 of Algorithm 3, the rounding is an acknowledgement that complex numbers may be represented with restricted precision: if we were to reason only on the mathematical definition of $\hat{A}$, $\hat{B}$, $\hat{C}$, and C, we could be content with the observation that C has integer coefficients.

size of elements of $\mathcal{R}$, that is $O(2^\lambda \log n)$. Additionally, all implicit rearrangement costs of Algorithm 3 (see §2.3) are also within this same bound. We get the following equation for $\mathsf{M}(n)$:

$$\mathsf{M}(n) = N(3\lceil \log_{2^{\lambda+1}} N \rceil + 1) \cdot \mathsf{M}(O(\log n)^2) + O(N \log N \cdot 2^\lambda \log n). \tag{3.1}$$

Fürer proves that this recurrence leads to $\mathsf{M}(n) \leq n \log n (2^{d \log^* \sqrt[4]{n}} - d')$ for some constants $d, d' > 0$, so that

$$\mathsf{M}(n) = n \cdot \log n \cdot 2^{O(\log^* n)}.$$

Various directions improve on the above complexity. One of them is to take advantage of precomputations of transforms of roots of unity. Briefly put, this transforms the constant 3 in Equation (3.1) to 2. We do not detail here how this can be done. In fact, this precomputation strategy is one of the ingredients (but certainly not the most original one) that the present article develops in §5 to obtain an improved complexity.

As mentioned in §1, Harvey, van der Hoeven and Lecerf in [HvdHL16], and Harvey and van der Hoeven in [HvdH16] propose other ways to obtain a better complexity. They propose new algorithms that achieve complexity bounds similar to the one that Fürer gets, and improve on the constant d. Their improvement yields an asymptotic equation similar to the improvement in this article. The algorithms in [HvdHL16, HvdH16] rely on Bluestein's chirp transform [Blu70]. They are unrelated to the present work, and will not be detailed.

4. Admissible generalized Fermat numbers and primes

This section defines admissible generalized Fermat numbers. Our main use case will be when such numbers are prime, and we define a descending chain of such primes. This section is independent of the previous sections.

Definition 4.1. A *generalized Fermat number* is an integer of the form $r^{2^\lambda} + 1$, where λ and r are two positive integers. We use the shorthand notation $P(r, \lambda)$ for such numbers.

For notational ease, throughout this article, whenever we mention a generalized Fermat number p, we actually consider the pair (r, λ) rather than the number p alone. For this reason, it shall be understood without further mention that r and λ are implicit data that is unequivocally attached to p, which is underlined by the fact that we favor the expression "let $p = P(r, \lambda)$ be a generalized Fermat number".

4.1. Abundance of generalized Fermat primes.

Asymptotically, the existence of generalized Fermat *primes* in integer intervals can be obtained via the Bateman-Horn conjecture [BH62]. For real numbers $A < B$ and an integer $\lambda \geq 1$, we let $\Delta(\lambda, A, B)$ denote the number of integers $r \in [A, B)$ such that $p = f(r) = P(r, \lambda) = r^{2^\lambda} + 1$ is a generalized Fermat prime. The following lemma captures the asymptotic behaviour of Δ in specific intervals. However it will be of little use *per se* but to define some notations.

Lemma 4.2. *Fix an integer $\lambda \geq 1$. Let $\alpha > 1$ be a real number (possibly depending on λ). If the Bateman-Horn conjecture holds for $f(x) = x^{2^\lambda} + 1$, then*

$$\Delta(\lambda, R, \alpha R) \sim \frac{C_\lambda}{2^\lambda}(\mathrm{li}(\alpha R) - \mathrm{li}(R))$$

as $R \to \infty$, where we used the notations:

$$\mathrm{li}(x) = \int_2^x \frac{\mathrm{d}t}{\log t}, \quad C_\lambda = \frac{1}{2} \prod_{p \text{ prime}} \frac{1 - \chi_\lambda(p)/p}{1 - 1/p}, \quad \chi_\lambda(p) = \begin{cases} 2^\lambda & \text{if } 2^{\lambda+1} \mid p - 1, \\ 0 & \text{otherwise.} \end{cases}$$

λ	C_λ	λ	C_λ	λ	C_λ	λ	C_λ	λ	C_λ
1	1.37	5	3.61	9	7.49	13	8.47	17	11.00
2	2.68	6	3.94	10	8.02	14	8.01	18	13.01
3	2.09	7	3.11	11	7.23	15	5.80	19	13.06
4	3.67	8	7.43	12	8.43	16	11.20	20	14.45

TABLE 1. Approximations of the infinite product C_λ, as defined by Lemma 4.2. The computation was done by enumerating all primes below 10^{11}, with resulting values rounded to nearest. Proposition 4.3 shows that $\frac{1}{\lambda} = O(C_\lambda)$.

Proof. Bateman and Horn [BH62] define the constant C_λ as above, and conjecture that as R grows, we have

$$\Delta(\lambda, 1, R) \sim \frac{C_\lambda}{2^\lambda} \operatorname{li}(R) \sim \frac{C_\lambda}{2^\lambda} \cdot \frac{R}{\log R}.$$

Let $\epsilon > 0$. Assuming the Bateman-Horn conjecture holds, we have for R large enough

$$\left| \frac{\Delta(\lambda, R, \alpha R)}{\frac{C_\lambda}{2^\lambda}(\operatorname{li}(\alpha R) - \operatorname{li}(R))} - 1 \right| < \epsilon \cdot \frac{1 + \operatorname{li}(R)/\operatorname{li}(\alpha R)}{1 - \operatorname{li}(R)/\operatorname{li}(\alpha R)}.$$

Now since $\operatorname{li}(x) \sim x/\log x$ and $\alpha > 1$, the right-hand side above converges to a positive constant as $R \to \infty$. This proves the claim. □

We now go through several steps to provide heuristic arguments supporting the existence of sufficiently many generalized Fermat primes in our ranges of interest. Our attention first goes to the asymptotic estimate on the right-hand side in Lemma 4.2, and to how it evolves as $\lambda \to \infty$, for some specific choices of α and R. Table 1 indicates some experimental values for the constant C_λ (the same data has also been collected by [DG02]). While the observation of Table 1 would support the empirical claim that C_λ increases as λ increases, a proof of such a statement has eluded us. In Appendix A, we prove Proposition 4.3 below, which is a much weaker statement. We then choose α and R so that the estimate of Lemma 4.2 can be shown to tend to infinity (Proposition 4.4).

Proposition 4.3. *Let C_λ be as in Lemma 4.2. We have $\frac{1}{\lambda} = O(C_\lambda)$.*

Proof. See Appendix A □

Proposition 4.4. *We use the same notations as in Lemma 4.2. Let $a(\lambda)$ be a real-valued function such that $a(\lambda) \geq \kappa\lambda^{2+\epsilon}$ for two positive constants κ, ϵ. Then the asymptotic estimate of Lemma 4.2, when formulated for $R = 2^\lambda$ and $\alpha = a(\lambda)$ is such that, as $\lambda \to \infty$:*

$$\frac{C_\lambda}{2^\lambda}\left(\operatorname{li}(a(\lambda) \cdot 2^\lambda) - \operatorname{li}(2^\lambda)\right) \longrightarrow \infty.$$

Proof. A lower bound for $(\operatorname{li}(\alpha R) - \operatorname{li}(R))$ is $(\alpha - 1)R/\log(\alpha R)$. We have

$$\frac{C_\lambda}{2^\lambda}\left(\operatorname{li}(a(\lambda) \cdot 2^\lambda) - \operatorname{li}(2^\lambda)\right) \geq \frac{C_\lambda}{2\log 2} \cdot \frac{\kappa\lambda^{2+\epsilon} - 1}{\lambda} \geq \lambda C_\lambda \cdot \frac{\kappa\lambda^\epsilon - 1/\lambda^2}{2\log 2}.$$

The claim follows, since $\frac{1}{\lambda} = O(C_\lambda)$ implies that $\lambda C_\lambda \lambda^\epsilon$ tends to ∞. □

Our heuristic claim is that for $\alpha = \lambda^{2.5}$ (which fulfills the conditions of Proposition 4.4), the estimate of Lemma 4.2 is accurate enough, as early as for $R = 2^\lambda$.

λ	Candidates	Primes	Estimate	λ	Candidates	Primes	Estimate
1	0	0	0	7	8233	42	46
2	9	3	5	8	2.3e4	126	138
3	58	1	8	9	6.2e4	184	170
4	248	24	22	10	1.6e5	224	218
5	878	31	30	11	4.1e5	227	230
6	2789	57	45	12	1.0e6	≥ 307	312

TABLE 2. Number of generalized Fermat primes $r^{2^\lambda}+1$ with $r \in [R, \lambda^{2.5}R)$ with $R = 2^\lambda$ (only even r are counted as candidates), compared to the asymptotic estimate of Lemma 4.2. Hypothesis 4.5 asserts that the third column is never zero for $\lambda \geq 2$.

Hypothesis 4.5. *Let $\lambda \geq 2$ be an integer. For any real number R such that $2^\lambda \leq R \leq 2^{2\lambda}$, we have $\Delta(\lambda, R, \lambda^{2.5}R) \geq 1$. In other words, there exists a generalized Fermat prime $p = P(r, \lambda)$ such that $R \leq r < \lambda^{2.5}R$.*

Both the constant C_λ, as well as the accordance of the prime count $\Delta(\lambda, 1, B)$ with the asymptotic estimate given by the Bateman-Horn conjecture, have been studied by [DG02]. While the experiments of [DG02] do support the validity of the Bateman-Horn conjecture even for primes not very large, we provide independent experimental data to support Hypothesis 4.5. We computed numerically the value $\Delta(\lambda, 2^\lambda, \lambda^{2.5}2^\lambda)$, as well as the estimate given by Lemma 4.2. We chose to restrict the verification to $R = 2^\lambda$ because this is empirically the hardest case. To obtain $\Delta(\lambda, 2^\lambda, \lambda^{2.5}2^\lambda)$, we used a simple primality proof algorithm based on Pocklington's theorem, in Las Vegas probabilistic time. The result of our experiments is given in Table 2.

Hypothesis 4.5 is in fact stronger than what would be strictly necessary to reach the asymptotic complexity we claim in this article. Proposition 4.4 led us to choose α as a polynomial of degree at least two, and our particular choice $\alpha = \lambda^{2.5}$ has the advantage that the data in Table 2 has no corner cases for small values of λ (in particular for $\lambda = 3$).

Throughout the rest of the article, Hypothesis 4.5 is tacitly assumed.

4.2. **Chains of generalized Fermat primes.** Some generalized Fermat numbers, defined below, play a key role in this article.

Definition 4.6 (Admissible generalized Fermat number)**.** A generalized Fermat number $p = P(r, \lambda)$ is called *admissible* whenever $\lambda \geq 4$ and r is such that $2^\lambda \leq r < 2^{2\lambda}\lambda^{2.5}$.

Definition 4.6 captures the primes whose existence is asserted by Hypothesis 4.5 (it is easy to observe that these are admissible when $\lambda \geq 4$), as well as generalized Fermat numbers that are subject to the same bounds.

The following proposition shows how from admissible generalized Fermat numbers (not necessarily prime), we can build smaller generalized Fermat primes. For large enough inputs, these smaller primes are in turn admissible, so that this construction can be used another time.

Proposition 4.7. *Let $\lambda \geq 4$, and let $p = P(r, \lambda) = r^{2^\lambda}+1$ be an admissible generalized Fermat number. A smaller generalized Fermat prime denoted* smallerprime(p) *and an integer* batchsize(p) *are defined as follows.*

Let $\lambda' = \left\lceil \log_2^{(3)} p \right\rceil$. Let $\phi(k) = 2^{k+1}\log_2 r + \lambda - k$. There exists a power of two β such that the following conditions hold:

λ	λ'
$3 \le \lambda \le 4$	3
$\lambda = 5$	3, 4
$6 \le \lambda \le 11$	4
$\lambda = 12$	4, 5
$13 \le \lambda \le 26$	5
$\lambda = 27$	5, 6
$28 \le \lambda \le 56$	6
$57 \le \lambda \le 58$	6, 7
$59 \le \lambda$	≥ 7

TABLE 3. Possible values for $\lambda' = \left\lceil \log_2^{(3)} p \right\rceil$ for $p = P(r, \lambda)$ an admissible generalized Fermat number, using the bounds $\log_2(\lambda + \log_2 \lambda) \le \lambda' \le 1 + \log_2^{(2)}(1 + 2^\lambda(2\lambda + 2.5 \log_2 \lambda))$.

(*i*) $0 \le \log_2 \beta < \lambda'$,
(*ii*) $\lambda' 2^{\lambda'} \le \phi(\log_2 \beta) \le 2\lambda' 2^{\lambda'}$,
(*iii*) *Given* $R' = 2^{\phi(\log_2 \beta)/2^{\lambda'}}$, *there exists an integer* $r' \in [R', {\lambda'}^{2.5} R')$ *such that* $p' = P(r', \lambda') = {r'}^{2^{\lambda'}} + 1$ *is a generalized Fermat prime.*

Given β *and* p' *as above, we let* smallerprime$(p) = p'$ *and* batchsize$(p) = \beta$. *Furthermore, if* $\lambda' \ge 4$, *then* p' *is admissible too.*

In anticipation for the proof of Proposition 4.7, we prove the following bounds.

Lemma 4.8. *Let* λ *and* λ' *be as in Proposition 4.7. We have*

$$\log_2(\lambda + \log_2 \lambda) \le \lambda' < 3 \log_2 \lambda - 1 < \lambda.$$

Proof. Since p is admissible, we have

$$2^{\lambda'} \ge \log_2(2^\lambda \log_2 r) \ge \lambda + \log_2^{(2)} r \ge \lambda + \log_2 \lambda.$$

In the other direction, the condition on p being admissible gives the following uniform bound on λ' (we first bound p by $2r^{2^\lambda}$):

$$\lambda' \le 1 + \log_2^{(2)}(1 + 2^\lambda(2\lambda + 2.5 \log_2 \lambda)).$$

An unilluminating calculation shows that this right hand side is indeed bounded by $3 \log_2 \lambda - 1$ for all $\lambda \ge 3$, and then by λ for all $\lambda \ge 4$. □

The lower bound given by Lemma 4.8 is most useful now, and gives in fact the correct order of magnitude for λ'. The upper bound is much coarser and will be used in §6. Possible values for λ' are given in Table 3. In particular, $\lambda \ge 4$ implies $\lambda' \ge 3$.

Proof of Proposition 4.7. The function ϕ is easily seen to satisfy $\phi(k) \le 2\phi(k-1)$ for any integer $k \le \lambda + 2$. As a consequence, the intervals $[\phi(k), 2\phi(k)]$, for k ranging from 0 to $\lambda' - 1$, form a covering of the interval $[\phi(0), \phi(\lambda')]$.

We prove $\phi(0) \le 2\lambda' 2^{\lambda'} \le \phi(\lambda')$, which will directly entail that $2\lambda' 2^{\lambda'}$ is within one of the above intervals that form a covering.

The bound $2\lambda' 2^{\lambda'} \le \phi(\lambda')$ is a consequence of $\lambda \ge \lambda'$:

$$\phi(\lambda') \ge 2^{\lambda'+1} \log_2 r \ge 2^{\lambda'+1} \lambda \ge 2^{\lambda'+1} \lambda'.$$

The proof that $2\lambda' 2^{\lambda'} \ge \phi(0)$ is based on calculus. Lower and upper bounds for $2\lambda' 2^{\lambda'}$ and $\phi(0)$ are

$$2\lambda' 2^{\lambda'} \ge 2(\lambda + \log_2 \lambda) \log_2(\lambda + \log_2 \lambda) \ge (\lambda + \log_2 \lambda) \log_2(36),$$
$$\phi(0) \le \lambda + 2(2\lambda + 2.5 \log_2 \lambda) = 5(\lambda + \log_2 \lambda).$$

We have proved that there exists an integer k such that $0 \leq k < \lambda'$, and that $\phi(k) \leq 2\lambda' 2^{\lambda'} \leq 2\phi(k)$. Let $\beta = 2^k$, so that (i) holds. We have that

$$\lambda' \leq \frac{\phi(\log_2 \beta)}{2^{\lambda'}} \leq 2\lambda'.$$

This implies (ii). Finally, $R' = 2^{\frac{\phi(\log_2 \beta)}{2^{\lambda'}}}$ is such that $2^{\lambda'} \leq R' \leq 2^{2\lambda'}$. Hypothesis 4.5 then implies (iii), and concludes the proof. Admissibility of p' follows from Definition 4.6. □

The following technical lemma provides useful bounds for $p' = \text{smallerprime}(p)$.

Lemma 4.9. *Let $p = P(r, \lambda)$ be as in Proposition 4.7. Let $\beta = \text{batchsize}(p)$ and $p' = \text{smallerprime}(p)$. We have*

(i) $1 \leq \frac{\log_2 p'}{2\beta \log_2 r} \leq \min(1 + \frac{4 \log_2 \lambda'}{\lambda' - 1}, \frac{7}{2})$. *In particular,* $\frac{\log_2 p'}{\beta \log_2 r} = 2 + o(1)$.

(ii) $\lambda' + \log_2 \lambda' \leq \log_2^{(2)} p' \leq \lambda' + \log_2 \lambda' + 2$.

Proof. We follow the notations of Proposition 4.7. The lower bound in (i) is easy:

$$\log_2 p' \geq 2^{\lambda'} \log_2 R' \geq \phi(\log_2 \beta) \geq 2\beta \log_2 r.$$

The upper bound requires more work. On the one hand, Lemma 4.8 gives $2^{\lambda'} > \lambda$, whence

$$2\beta \log_2 r + 2^{\lambda'} \geq \phi(\log_2 \beta) \geq \lambda' 2^{\lambda'}$$
$$2\beta \log_2 r \geq (\lambda' - 1) 2^{\lambda'}.$$

And on the other hand, we can bound $\log_2 p'$ as follows.

$$\begin{aligned} \log_2 p' &= \log_2((p'-1)+1) = \log_2(p'-1) + \log_2(1 + 1/(p'-1)) \\ &\leq 2^{\lambda'} \log_2 r' + 1 \leq 2^{\lambda'} \log_2 R' + 2^{\lambda'} \log_2(\lambda'^{2.5}) + 1 \\ &\leq \phi(\log_2 \beta) + 2^{\lambda'} \log_2(\lambda'^{2.5}) + 1 \qquad (4.1) \end{aligned}$$

by the definition of R'. Using now Lemma 4.8 and $2^{\lambda'} \geq \lambda + \log_2 \lambda \geq \lambda + 2$ we have

$$\begin{aligned} \log_2 p' &\leq 2\beta \log_2 r + (2^{\lambda'} - 2) + 2^{\lambda'} \log_2(\lambda'^{2.5}) + 1 \\ &\leq 2\beta \log_2 r + 2^{\lambda'} \cdot \min(4 \log_2 \lambda', 5(\lambda' - 1)/2) \text{ since } \lambda' \geq 3. \end{aligned}$$

The upper bound on the last line is obtained by calculus. We have thus proved (i).

The lower bound in statement (ii) is trivial. The upper bound is derived from inequality (4.1) above. By (ii) in Proposition 4.7, we have $\frac{\phi(\log_2 \beta)}{2^{\lambda'}} \leq 2\lambda'$, whence

$$\log_2 p' \leq 2\lambda' 2^{\lambda'} + 2^{\lambda'} \log_2(\lambda'^{2.5}) + 1.$$
$$\begin{aligned} \log_2^{(2)} p' &\leq \log_2\left(1 + 2^{\lambda'}(2\lambda' + \log_2(\lambda'^{2.5}))\right) \\ &\leq \lambda' + \log_2 \lambda' + 2 \text{ since } \lambda' \geq 3. \end{aligned}$$

Again, this last upper bound is verified by calculus. □

5. Two new algorithms

We now see how we can design an asymptotically fast integer multiplication algorithm that uses rings of integers modulo generalized Fermat primes.

Throughout this section, our preferred representation for elements of a ring $\mathcal{R}$ of integers modulo a generalized Fermat number $p = P(r, \lambda)$ is the representation *in radix r*. Namely, $a \in \mathcal{R}$ is represented as a 2^λ-uple $(a_0, \ldots, a_{2^\lambda - 1})$ such that $a = \sum_{j < 2^\lambda} a_j r^j$ and $0 \leq a_j < r$. This representation does not cover the case

$a = -1$, and we need an *ad hoc* exceptional representation for this case (possible representation choices are plenty – one extra bit is enough). Conversions between binary representation and radix r representation can be done in linear time when r is a power of two, but we also need to deal with the general case. Recursive base conversion algorithms (see [BZ10, §1.7.2]), do this in quasi-linear time $O(\lambda \mathsf{M}(\log p))$ (this holds both for ways, both *to* and *from* representation in radix r). Additions and subtractions in $\mathcal{R}$ using this representation are linear. This section is concerned with the complexity of multiplication in $\mathcal{R}$. We denote this cost by $\mathsf{M}_{\mathcal{R}}$.

5.1. **Preliminaries: transforms.** The following definition extends concepts defined in Proposition 4.7 and defines useful data for our algorithms.

Definition 5.1 (smallerring($\mathcal{R}$))**.** Let $\lambda \geq 4$. Let $p = P(r, \lambda) = r^{2^\lambda} + 1$ be an admissible generalized Fermat number, and let $\mathcal{R} = \mathbb{Z}/p\mathbb{Z}$. Following Proposition 4.7 we let smallerring($\mathcal{R}$) be the triple $(\mathcal{R}', N', \omega')$ defined as follows:

- $\mathcal{R}' = \mathbb{Z}/p'\mathbb{Z}$, with $p' = P(r', \lambda') =$ smallerprime(p).
- $N' = 2^\lambda /$ batchsize(p). (N' is a power of two.)
- ω' is a primitive $2N'$-th root of unity in $\mathcal{R}'$.

For the root ω' to be well defined above, we need the following property.

Lemma 5.2. *Using the notations above, $\mathcal{R}'$ has a primitive $2N'$-th root of unity.*

Proof. Notice first that $\lambda' \geq 3$ so that $p' \geq 2^{3 \cdot 2^3}$, and that $p' = r'^{2^{\lambda'}} + 1$ is prime, so that in particular r' is even. For $2N'$ to divide $p' - 1$, it suffices to check that $2N'$ divides $2^{2^{\lambda'}}$. We have

$$\log_2(2N') = \lambda + 1 - \log_2 \beta$$
$$2^{\lambda'} \geq \lambda + \log_2^{(2)} r,$$

so that it is sufficient to check that $\log_2^{(2)} r \geq 1$, which holds as soon as $\lambda \geq 2$. □

The algorithms described in the remainder of this section all assume that the sequences of rings and auxiliary data defined by Definition 5.1 are computed in advance, for all levels of the recursion. We assume that a tape of our Turing machine is devoted to that data, stored one level after another. The size of the data smallerring($\mathcal{R}'$) is clearly $O(\log p')$.

5.2. **New algorithms.** We now describe two new algorithms that are dependent on each other. Both aim at computing products of elements of $\mathcal{R}$.

- One algorithm that computes "transforms" of elements of $\mathcal{R}$. Internally, this algorithm multiplies elements of $\mathcal{R}'$.
- One algorithm that multiplies elements of $\mathcal{R}$. This algorithm uses the transforms computed by the previous algorithm.

We begin with Algorithm 4 (TransformR), which computes transforms. We can state it thanks to Lemma 5.2.

Our complexity analysis will need to reason on the set of transforms of roots of unity that are used by Algorithm 4. We define it as follows:

Definition 5.3 ($\mathcal{W}(\mathcal{R})$, vector of precomputed transforms useful for $\mathcal{T}_{\mathcal{R}}$)**.** Fix notations as in Definition 5.1. We let $\mathcal{W}(\mathcal{R})$ denote the vector defined as:

$$\mathcal{W}(\mathcal{R}) = \{\mathcal{T}_{\mathcal{R}'}(\omega'^{2i+1}),\ i \in [\![0, \frac{N'}{2^{\lambda'+1}} - 1]\!]\}$$

where $\mathcal{T}_{\mathcal{R}'}$ is defined as in Algorithm 4 (albeit using $\mathcal{R}'$ as an input ring).

Algorithm 4 Transform $\mathcal{T}_{\mathcal{R}}(a)$ of $a \in \mathcal{R} = \mathbb{Z}/p\mathbb{Z}$, with $p = r^{2^\lambda} + 1$ admissible (not necessarily prime), $\lambda \geq 4$. (Algorithm without precomputations.)

1: **function** TransformR(a)
2: **Input:** $a \in \mathcal{R}$, represented in radix r.
3: **Output:** $\mathcal{T}_{\mathcal{R}}(a)$, a vector of N' elements of $\mathcal{R}'$, represented in radix r'
4: Let $\beta = \text{batchsize}(p)$, and $(\mathcal{R}', N', \omega') = \text{smallerring}(\mathcal{R})$.
5: Let $\tilde{A}(X) \in \mathbb{Z}[X]$ with positive coefficients below r^β be such that $\tilde{A}(r^\beta) = a$;
6: Map $\tilde{A}$ to $A \in \mathcal{R}'[X]/(X^{N'} + 1)$.
7: Rewrite coefficients of A in radix r'.
8: **return** Half-DFT$_{N',\omega'}(A) = \mathsf{LargeRadixFFT}(N', \omega'^2, 2^{\lambda'+1}, A(\omega' X))$.
9: **end function**

Complexity of Algorithm 4, with or without precomputations. We define the following costs. The analysis of $\mathsf{M}_{\mathcal{R}}$ and $\mathsf{M}'_{\mathcal{R}}$ will be done in §5.3.

- $\mathsf{M}_{\mathcal{R}}$: cost of multiplying $a \in \mathcal{R}$ by $b \in \mathcal{R}$, with no auxiliary inputs.
- $\mathsf{M}'_{\mathcal{R}}$: cost of the same computation, with $\mathcal{T}_{\mathcal{R}}(b)$ known.
- $\mathsf{T}_{\mathcal{R}}$: cost of computing $\mathcal{T}_{\mathcal{R}}$ with Algorithm 4.
- $\mathsf{T}'_{\mathcal{R}}$: cost of computing $\mathcal{T}_{\mathcal{R}}$ with Algorithm 4, aided with the auxiliary knowledge of $\mathcal{W}(\mathcal{R})$.
- $\mathsf{W}_{\mathcal{R}}$: cost of computing $\mathcal{W}(\mathcal{R})$.

We begin with $\mathcal{T}_{\mathcal{R}}$. Algorithm 4 uses base conversions on lines 5 and 7. Both operations perform $N' = 2^\lambda/\beta$ conversions, and the respective costs per conversion in each case are $O(\log\beta \cdot \mathsf{M}(\beta \log r))$ and $O(\lambda' \cdot \mathsf{M}(\beta \log r))$ (in these complexity estimates, $\mathsf{M}(n)$ can be taken as the complexity obtained for mutiplying integers by the Schönhage-Strassen algorithm, for example). By Proposition 4.7 we have $\log\beta \leq \lambda'$, and by Lemma 4.9 we have $\log p' = \Theta(\beta \log r)$, so that the overall base conversion costs in Algorithm 4 can be expressed as $O(N'\lambda' \cdot \mathsf{M}(\log p'))$.

The computation of the Half-DFT on line 8 of Algorithm 4 involve $N' \log N'$ multiplication by roots of unity in $\mathcal{R}'$, of which only $(E(N') + N')$ exceed a linear cost (using the notation of §3.1). We have

$$\mathsf{T}_{\mathcal{R}} = (E(N') + N')\mathsf{M}_{\mathcal{R}'} + O\left(N' \log N' \log p' + N'\lambda' \cdot \mathsf{M}(\log p')\right).$$

We now turn to the analysis of $\mathsf{T}'_{\mathcal{R}}$. If the vector $\mathcal{W}(\mathcal{R}) = \{\mathcal{T}_{\mathcal{R}'}(\omega^{2i+1}),\ i \in [\![0, \frac{N'}{2^{\lambda'+1}} - 1]\!]\}$ is known, then the computation of $\mathcal{T}_{\mathcal{R}}$ can be done a bit faster: the $(E(N') + N')$ "expensive" multiplications by roots of unity in $\mathcal{R}'$ do not need to recompute the transforms of the roots. They may thus use a somewhat faster algorithm for multiplication in $\mathcal{R}'$. We defined above its cost as $\mathsf{M}'_{\mathcal{R}}$, and we have:

$$\mathsf{T}'_{\mathcal{R}} = (E(N') + N')\mathsf{M}'_{\mathcal{R}'} + O\left(N' \log N' \log p' + N'\lambda' \cdot \mathsf{M}(\log p')\right).$$

Finally, we give the cost $\mathsf{W}_{\mathcal{R}}$ of computing $\mathcal{W}(\mathcal{R})$. Here, we do *not* recursively use $\mathcal{W}(\mathcal{R}')$ to compute the different elements. We do however use the knowledge of the root of unity ω' (it belongs to the precomputed data smallerring($\mathcal{R}$)). To compute $\mathsf{W}_{\mathcal{R}}$, we first compute $\mathcal{T}_{\mathcal{R}'}(\omega')$ and $\mathcal{T}_{\mathcal{R}'}(\omega'^2)$, which cost $2\mathsf{T}_{\mathcal{R}'}$. Then we do successive pointwise multiplications by the vector $\mathcal{T}_{\mathcal{R}'}(\omega'^2)$ to obtain the transforms of the other roots. For each of the $N'/2^{\lambda'+1} - 1$ transforms to be inferred this way, we need N'' multiplications in $\mathcal{R}''$, where we temporarily set $(\mathcal{R}'', N'', \omega'') = \text{smallerring}(\mathcal{R}')$. Therefore we have

$$\mathsf{W}_{\mathcal{R}} \leq 2\mathsf{T}_{\mathcal{R}'} + (N'/2^{\lambda'+1} - 1)N''\mathsf{M}_{\mathcal{R}''} \leq N'\mathsf{T}_{\mathcal{R}'}.$$

Without further detail, we also claim that the inverse transform $\mathcal{T}_{\mathcal{R}}^{-1}$ can be computed with the same cost as $\mathcal{T}_{\mathcal{R}}$.

Algorithm 5 Multiplication in $\mathcal{R} = \mathbb{Z}/p\mathbb{Z}$, with $p = r^{2^\lambda} + 1$ admissible, $\lambda \geq 4$. p is not necessarily prime.

We use the notations $\mathcal{T}_\mathcal{R}$, $\mathcal{W}_\mathcal{R}$ as in §5.2.
1: **function** MulR(a,$T_\mathcal{R}(b)$)
2: **Input:** $a \in \mathcal{R}$, represented in radix r; $\mathcal{T}_\mathcal{R}(b)$ for some $b \in \mathcal{R}$.
3: **Output:** $a \cdot b \mod p$, represented in radix r
4: Let $\beta = \text{batchsize}(p)$, and $(\mathcal{R}', N', \omega') = \text{smallerring}(\mathcal{R})$.
5: Compute $W = \mathcal{W}(\mathcal{R})$ using Algorithm TransformR.
6: Compute $\mathcal{T}_\mathcal{R}(a)$ using Algorithm TransformR and W as auxiliary data.
7: Compute $\gamma = \mathcal{T}_\mathcal{R}(a) * \mathcal{T}_\mathcal{R}(b)$ ▷ pointwise products of elements of $\mathcal{R}'$.
8: Compute $c = \mathcal{T}_\mathcal{R}^{-1}(\gamma)$ as follows:
9: $C \leftarrow \text{Half-IFT}_{N',\omega'}(\gamma) \in \mathcal{R}'[X]/(X^{N'}+1)$ using W as auxiliary data.
10: Lift C to $\tilde{C} \in \mathbb{Z}[X]$ as follows:
11: **for** $i \in [\![0, N'-1]\!]$ **do**
12: Lift coefficient of degree i to $[\![-(N'-1-i)r^{2\beta}, (i+1)r^{2\beta}[\![$.
13: **end for**
14: Rewrite coefficients of $\tilde{C}$ as signed integers in radix r.
15: Compute $\tilde{C}(r^\beta) = c$. ▷ The result is defined modulo $(r^\beta)^{N'} + 1 = p$.
16: **return** c
17: **end function**

5.3. **Multiplication modulo generalized Fermat numbers.** Using Algorithm 4 (TransformR), we can now state Algorithm 5 (MulR). Its validity depends on the following lemma:

Lemma 5.4. *Let notations be as in Algorithm 5. Let $\tilde{A}, \tilde{B}$ be polynomials in $\mathbb{Z}[X]$ of degree less than N' and with positive coefficients below r^β such that, A and B being their respective images in $\mathcal{R}'[X]/(X^{N'}+1)$, we have $\mathcal{T}_\mathcal{R}(a) = \text{Half-DFT}_{N',\omega'}(A)$ and $\mathcal{T}_\mathcal{R}(b) = \text{Half-DFT}_{N',\omega'}(B)$ on line 7 of Algorithm 5.*

(i) Both $\tilde{A}$ and $\tilde{B}$ are uniquely defined from $\mathcal{T}_\mathcal{R}(a)$ and $\mathcal{T}_\mathcal{R}(b)$.
(ii) The polynomial $\tilde{C}$ is equal to $\tilde{A} \cdot \tilde{B} \bmod X^{N'}+1$.
(iii) c is equal to $ab \bmod p$.

Proof. We prove (i) for $\mathcal{T}_\mathcal{R}(a)$, the same reasoning holds for $\mathcal{T}_\mathcal{R}(b)$. The polynomial $A \in \mathcal{R}'[X]/(X^{N'}+1)$ is uniquely defined because Half-DFT is an isomorphism. Now since $\mathcal{T}_\mathcal{R}(a)$ is computed from an element a of $\mathcal{R}$, line 5 of Algorithm 4 has unambiguously computed a polynomial $\tilde{A}$, which meets the conditions. Since there is a unique lift of A to $\mathbb{Z}[X]$ that has degree less than N' and positive coefficients below p', this lift is then necessarily the same as $\tilde{A}$.

Statement (ii) holds modulo p' by construction, but we must make sure that the lift on lines 10-13 of Algorithm 5 computes the correct product over the integers. To do so, we compute a bound for the coefficients of the product $\tilde{A} \cdot \tilde{B} \bmod X^{N'}+1$. Both operands have at most N' coefficients. The coefficient of degree i of their product modulo $X^{N'}+1$ lies within the interval $[\![-(N'-1-i)(r^\beta)^2, (i+1)(r^\beta)^2[\![$ (actually with the lower endpoint open for $i < N'-1$), which has width $N'(r^\beta)^2$. The base 2 logarithm of this latter value is $2\beta \log_2 r + \lambda - \log_2 \beta = \phi(\log_2 \beta)$, following the notation of Proposition 4.7. Now again following notations of Proposition 4.7, we have $p' \geq R'^{2^{\lambda'}} \geq 2^{\phi(\log_2 \beta)} \geq N'(r^\beta)^2$. Thus, the coefficient

c_i of degree i of $\tilde{A}\cdot\tilde{B} \bmod X^{N'}+1$ is lifted to a unique signed representative modulo p' on line 10. This proves the claim.[2]

Statement (*iii*) follows: by (*ii*), we have that $\tilde{C} = \tilde{A}\cdot\tilde{B} \mod X^{N'}+1$. By evaluating at r^β, we obtain the result $c = ab$ modulo $(r^\beta)^{N'}+1 = p$. □

Complexity analysis of Algorithm 5. We first mention that the relative costs of multiplications and transforms, with or without precomputations, satisfy the following equations.

$$2\mathsf{T}'_{\mathcal{R}} \le \mathsf{M}'_{\mathcal{R}} \le \mathsf{M}_{\mathcal{R}} \le \mathsf{M}'_{\mathcal{R}} + \mathsf{T}'_{\mathcal{R}} \le \frac{3}{2}\mathsf{M}'_{\mathcal{R}} \quad \text{and} \quad \mathsf{T}_{\mathcal{R}} \le \mathsf{M}_{\mathcal{R}}.$$

(To get $\mathsf{M}_{\mathcal{R}} \le \mathsf{M}'_{\mathcal{R}} + \mathsf{T}'_{\mathcal{R}}$, it suffices to *first* compute $\mathcal{W}(\mathcal{R})$, and then $\mathcal{T}_{\mathcal{R}}(b)$.)

On line 10, Algorithm 5 converts between representation in radix r' and binary representation. On line 14 the conversion is between binary representation and representation in radix r. As with Algorithm 4, we can do this in time $O(N'\lambda'\cdot \mathsf{M}(\log p'))$. Pointwise products, on line 7, use a variation of Algorithm 5, where there is no auxiliary input, recursively (thus exploiting the fact that the coefficients of $\mathcal{T}_{\mathcal{R}}(a)$ and $\mathcal{T}_{\mathcal{R}}(b)$ are represented in radix r'). And last but not least, the most important aspect of the complexity of Algorithm 5 is that since we compute $\mathcal{W}(\mathcal{R})$, the transforms $\mathcal{T}_{\mathcal{R}}(a)$ and $\mathcal{T}_{\mathcal{R}}^{-1}(\gamma)$ can take advantage of it. We thus have:

$$\mathsf{M}'_{\mathcal{R}} = \mathsf{W}_{\mathcal{R}} + 2\mathsf{T}'_{\mathcal{R}} + N'\mathsf{M}_{\mathcal{R}'} + O(N'\lambda'\cdot\mathsf{M}(\log p')) + O(\log p).$$

We now use the various expressions obtained in §5.2 to rewrite this. We use the coarse bounds $\mathsf{T}_{\mathcal{R}'} \le \mathsf{M}_{\mathcal{R}'} \le \frac{3}{2}\mathsf{M}'_{\mathcal{R}'}$. We have

$$\begin{aligned}
\mathsf{M}'_{\mathcal{R}} &\le N'\mathsf{M}_{\mathcal{R}'} + 2E(N')\mathsf{M}'_{\mathcal{R}'} + 2N'\mathsf{M}'_{\mathcal{R}'} + N'\mathsf{M}_{\mathcal{R}'} \\
&\qquad + O(N'\cdot\lambda'\cdot\mathsf{M}(\log p')) + O(N'\cdot\log N'\cdot\log p') + O(\log p) \\
&\le 5N'\mathsf{M}'_{\mathcal{R}'} + 2E(N')\mathsf{M}'_{\mathcal{R}'} \\
&\qquad + O(N'\cdot\lambda'\cdot\mathsf{M}(\log p')) + O(N'\cdot\log N'\cdot\log p') + O(\log p) \\
&\le 2N'\cdot(3+\log_{2^{\lambda'+1}} N')\cdot\mathsf{M}'_{\mathcal{R}'} \\
&\qquad + O(N'\cdot\lambda'\cdot\mathsf{M}(\log p')) + O(N'\cdot\log N'\cdot\log p') + O(\log p)
\end{aligned}$$

where we used $E(N') \le N'\log_{2^{\lambda'+1}} N'$ and $5/2 < 3$. Algorithm 5 also needs to move the head of tape of precomputed data by the size of the current data smallerring($\mathcal{R}$). The corresponding overhead $O(\log p')$ is easily subsumed within the lower-order terms above.

5.4. **Multiplication in $\mathbb{Z}$ using multiplication in $\mathcal{R}$.** We can build on Algorithm 5 to obtain an integer multiplication algorithm for n-bit integers a and b.

Note however that we avoid the following simple approach because it does not work complexity-wise: we do not multiply a and b by considering them as elements of $\mathbb{Z}/p\mathbb{Z}$ for p an admissible generalized Fermat number such that $p \ge 2^{2n}$. There are two reasons for that. First, doing so for p an admissible generalized Fermat *prime* is out of question: unless we consider that p is given beforehand, computing it is likely to be more expensive than computing a product of bit length $\log_2 p$, and would therefore appear dominant, maybe prohibitive even for a precomputation. Fortunately, Algorithm 5 (MulR) does not require that p be prime, and therefore this difficulty can easily be circumvented. For example we may select λ such that $\lambda 2^\lambda \ge 2n$, and then set $p = P(2^\lambda, \lambda)$. The second issue is harder to deal with: in the ring $\mathcal{R}'$ used by Algorithm 5, we need to find $2N'$-th roots of unity, and for

[2] On lines 10-13 of Algorithm 5, intervals depend on the degree so that we can do without a needlessly coarse lower bound $2N'(r^\beta)^2 \le p'$. It would be possible to adjust the definition of ϕ in Proposition 4.7, as well as the corresponding proofs, so that that coarser inequality holds.

this we need a quadratic nonresidue in $\mathcal{R}'$ (which generates the 2-Sylow subgroup of $\mathcal{R}'$). Alas, if our first (non-prime) modulus p is such that $\log_2 p \geq 2n$, then in Proposition 4.7 we have $\lambda' = \lceil \log_2^{(3)} p \rceil \geq \log_2^{(2)} n$, so that the upper bound on $\log_2 p'$ that we obtain from Lemma 4.9 is at least as large as $\log_2 n \cdot \log_2^{(2)} n$. If we can use only deterministic exponential-time algorithms to search for a quadratic nonresidue in $\mathcal{R}'$, then the complexity of this search exceeds the overall complexity of integer multiplication.

Similar (but subtly different) issues were already encountered by Harvey, van der Hoeven and Lecerf. The workarounds proposed in [HvdHL16, §8] also apply here.

- Either we assume the generalized Riemann hypothesis, in which case a quadratic nonresidue in $\mathcal{R}'$ can be found in polynomial time.
- Or we do the top-level multiplication with one round of Fürer's algorithm. Multiplication in the ring $\mathbb{C}[X]/X^{2^\lambda}+1$ that is used by Algorithm 3 (FurerComplexMul) reduces to multiplication of integers of bit length $n_0 = O((\log n)^2)$, with n denoting the bit length of the integers a and b (see Equation (3.1)). These integers are then multiplied by Algorithm 5 (MulR), for a suitable modulus p_0 (not necessarily prime).

The latter strategy is given by Algorithm 6 (MulZ). Note that since we build upon Algorithm 3, we force the bit length n to be rounded up to a power of two.

Algorithm 6 Multiplication of integers in $\mathbb{Z}$

1: **Input:** a, b two positive n-bit integers, n being a power of two.
2: **Output:** $c = a \cdot b$
3: **function** MulZ(a,b)
4: Let n_0 be such that all internal multiplications in FurerComplexMul$(\cdot,\cdot,n)$ may be done by multiplying two n_0-bit integers. (As per the analysis of FurerComplexMul, we have $n_0 = O((\log_2 n)^2)$.)
5: Let λ_0 be the smallest integer such that $2n_0 \leq \lambda_0 2^{\lambda_0}$.
6: Let $p_0 = P(2^{\lambda_0}, \lambda_0) = 2^{\lambda_0 2^{\lambda_0}} + 1$.
7: **return** $c =$ FurerComplexMul(a, b, n), where all internal multiplications are done with Algorithm 5 (MulR), in the ring $\mathcal{R}_0 = \mathbb{Z}/p_0\mathbb{Z}$.
8: **end function**

It is easy to see that p_0 in Algorithm 6 is an admissible generalized Fermat number. As for the determination of *prime* moduli as well as the computation of primitive roots of unity of the desired order in the recursive multiplication levels of Algorithm 5, we have that $\log_2(\text{smallerprime}(p_0))$ is polynomial in $\log_2^{(2)} n$. This is small enough so that simple algorithms are fit for the task of testing smallerprime(p_0) for primality, as well as for finding primitive roots. Thus the complete chain of precomputed triples defined by smallerring in Definition 5.1 can be computed in advance and stored on an auxiliary tape of the Turing machine, as suggested in §5.1.

The complexity of computing n-bit products with Algorithm 6 (MulZ), which we denote by $\mathsf{M}_{\text{new}}(n)$, can be expressed as follows. The equation below is naturally very similar to Equation (3.1).

$$\mathsf{M}_{\text{new}}(n) = N(3\lceil \log_{2^{\lambda+1}} N \rceil + 1) \cdot \mathsf{M}_{\mathcal{R}_0} + O(N \log N \cdot 2^\lambda \log n);$$

6. Solution of the recursive complexity equations

6.1. **Summary of the recursive complexity equations.** In Algorithm 5 (MulR), multiplication in $\mathcal{R}$ uses $(\mathcal{R}', N', \omega') = \text{smallerring}(\mathcal{R})$. In turn, multiplication in

$\mathcal{R}'$ may use $(\mathcal{R}'', N'', \omega'') = \text{smallerring}(\mathcal{R}')$ if recursion is used again. We define $(\mathcal{R}_i)_{i\geq 0}$ as well as $(N_i)_{i\geq 1}$ and $(\omega_i)_{i\geq 1}$ by:

$$\mathcal{R}_0 = \text{as in Algorithm 6,}$$
$$(\mathcal{R}_{i+1}, N_{i+1}, \omega_{i+1}) = \text{smallerring}(\mathcal{R}_i) \text{ for } i \geq 0.$$

Likewise, we let p_i be such that $\mathcal{R}_i = \mathbb{Z}/p_i\mathbb{Z}$, for $i \geq 0$. Of course, since Definition 5.1 as well as Algorithms 4 and 5 are only valid for $\lambda_i \geq 4$, only a finite number of terms of the above sequences are defined for a given input size n. Part of the work towards determining our final complexity will be to determine this number of terms (the recursion depth). We briefly recall the key equations for the complexity analysis:

$$\begin{aligned}\mathsf{M}'_{\mathcal{R}_i} &\leq 2N_{i+1} \cdot (3 + \log_{2^{\lambda_{i+1}+1}} N_{i+1}) \cdot \mathsf{M}'_{\mathcal{R}_{i+1}} \\ &\quad + O(N_{i+1} \cdot \lambda_{i+1} \cdot \mathsf{M}(\log p_{i+1})) \\ &\quad + O(N_{i+1} \cdot \log N_{i+1} \cdot \log p_{i+1}) \\ &\quad + O(\log p_i).\end{aligned}$$
$$\mathsf{M}_{\mathcal{R}_i} \leq \frac{3}{2}\mathsf{M}'_{\mathcal{R}_i}.$$
$$\mathsf{M}_{\text{new}}(n) = N(3\lceil \log_{2^{\lambda_0+1}} N \rceil + 1) \cdot \mathsf{M}_{\mathcal{R}_0} + O(N \log N \cdot 2^{\lambda_0} \log n).$$

We first prove the following that lemma bounds the transform length N'.

Lemma 6.1. *Using the notations as above, we have*

$$N_{i+1} \leq \min\left(2\left(1 + \frac{4\log_2 \lambda_{i+1}}{\lambda_{i+1} - 1}\right), 7\right) \cdot \frac{\log_2 p_i}{\log_2 p_{i+1}}.$$

Proof. Let $\beta = \text{batchsize}(p_i)$. We have $2^{\lambda_i} \log_2 r_i \leq \log_2 p_i$, therefore

$$N_{i+1} = \frac{2^{\lambda_i}}{\beta} \leq \frac{\log_2 p_i}{\beta \log_2 r_i} \leq 2\frac{\log_2 p_i}{\log_2 p_{i+1}} \frac{\log_2 p_{i+1}}{2\beta \log_2 r_i}.$$

Then (i) in Lemma 4.9 allows to conclude. □

The following result plays a central role in the asymptotic analysis.

Proposition 6.2. *We keep the above notations. Let $i \geq 0$ be such that p_i is admissible. Let $\epsilon_{0,i} = \frac{4\log_2 \lambda_{i+1}}{\lambda_{i+1}}$, $\epsilon_{1,i} = \frac{8\log_2 \lambda_i}{\lambda_i}$, and $\epsilon_{2,i} = \frac{2+\log_2 \lambda_{i+1}}{\lambda_{i+1}}$. Let $m_i = \frac{\mathsf{M}'_{\mathcal{R}_i}}{\log_2 p_i \cdot \log_2^{(2)} p_i}$. We have*

$$m_i \leq 4 \cdot (1 + \epsilon_{0,i}) \cdot (1 + \epsilon_{1,i}) \cdot (1 + \epsilon_{2,i}) \cdot m_{i+1} + O(1).$$

Proof. We first bound the second and third lines in the equation for $\mathsf{M}'_{\mathcal{R}_i}$, and compare them to $\log p_i \cdot \log^{(2)} p_i$. The third line uses Lemma 6.1. We have

$$N_{i+1}/\log_2 p_i \leq 7/\log_2 p_{i+1} = O(1)$$

which obviously also implies $(\log_2 N_{i+1})/(\log_2^{(2)} p_i) = O(1)$. Then

$$\frac{N_{i+1} \log_2 N_{i+1} \log_2 p_{i+1}}{\log_2 p_i \log_2^{(2)} p_i} \leq 7\frac{\log_2 N_{i+1}}{\log_2^{(2)} p_i} = O(1).$$

For the second line, it suffices to assume that $\mathsf{M}(\log p_{i+1})$ is bounded by the complexity of the Schönhage-Strassen algorithm. We have

$$\frac{N_{i+1}\lambda_{i+1}\mathsf{M}(\log p_{i+1})}{\log_2 p_i \log_2^{(2)} p_i} \leq 7\frac{\lambda_{i+1} \log_2^{(2)} p_{i+1} \log_2^{(3)} p_{i+1}}{\log_2^{(2)} p_i} = O(1).$$

In the expression above, we obtain the upper bound by bounding the numerator by a polynomial in λ_{i+1} (because p_{i+1} is admissible), while the denominator is exponential in λ_{i+1}.

The most important calculation for the analysis is the comparison of the first term of $\mathsf{M}'_{\mathcal{R}_i}$ with $\log p_i \cdot \log^{(2)} p_i$. Lemma 6.1 gives the bound $N_{i+1} \leq 2(1+\epsilon_{0,i})\frac{\log_2 p_i}{\log_2 p_{i+1}}$, and we also have the coarse bound $\log_2 N_{i+1} = \log_2(2^{\lambda_i}/\beta_i) \leq \lambda_i$. This implies

$$\begin{aligned} m_i &\leq 4(1+\epsilon_{0,i})\frac{\log_2 p_i}{\log_2 p_{i+1}} \cdot \left(3 + \frac{\lambda_i}{\lambda_{i+1}+1}\right) \cdot m_{i+1} \frac{\log_2 p_{i+1} \log_2^{(2)} p_{i+1}}{\log_2 p_i \log_2^{(2)} p_i} + O(1) \\ &\leq 4(1+\epsilon_{0,i}) \left(3 + \frac{\lambda_i}{\lambda_{i+1}+1}\right) \frac{\log_2^{(2)} p_{i+1}}{\log_2^{(2)} p_i} m_{i+1} + O(1) \end{aligned}$$

By Lemma 4.8 we have $\left(3 + \frac{\lambda_i}{\lambda_{i+1}+1}\right) \leq \frac{\lambda_i + 9\log_2 \lambda_i}{\lambda_{i+1}+1} \leq \frac{\lambda_i + 9\log_2 \lambda_i}{\lambda_{i+1}}$. Furthermore by statement *(ii)* from Lemma 4.9 for $i > 0$, we have $\log_2^{(2)} p_i \geq \lambda_i + \log_2 \lambda_i$, so that

$$\begin{aligned} m_i &\leq 4 \cdot (1+\epsilon_{0,i}) \cdot \frac{\lambda_i + 9\log_2 \lambda_i}{\lambda_i + \log_2 \lambda_i} \cdot \frac{\log_2^{(2)} p_{i+1}}{\lambda_{i+1}} \cdot m_{i+1} + O(1) \\ &\leq 4 \cdot (1+\epsilon_{0,i}) \cdot (1+\epsilon_{1,i}) \cdot (1+\epsilon_{2,i}) \cdot m_{i+1} + O(1). \end{aligned}$$

where we used again Lemma 4.9 to bound $\log_2^{(2)} p_{i+1}$. This proves our claim. □

It is easy to convince oneself that the three quantities $\epsilon_{0,i}$, $\epsilon_{1,i}$, and $\epsilon_{2,i}$ all tend to zero as λ_i grows (that is, as we deal with larger and larger input numbers). The final asymptotic formula needs the following stronger result, however.

Lemma 6.3. *Let λ_0 be an arbitrarily large integer. Let K be the first integer such that $\lambda_K < 4$. We have $K = \log^* \lambda_0 + O(1)$. Furthermore, for $j = 0, 1, 2$:*

$$\prod_{i=0}^{K-1} (1+\epsilon_{j,i}) < \infty \quad \textit{(independently of } K\textit{)}$$

Proof. The expression of K follows from the inequality $\lambda' < 3\log\lambda - 1$ proved in Lemma 4.8. To see that, let $\Phi(\lambda) = 3\log_2 \lambda - 1$, defined for $\lambda \geq 4$. Let $\Phi^*(x)$ be the function defined similarly to $\log^*$, by $\Phi^*(x) = 0$ for $x < 4$, and $\Phi^*(x) = 1 + \Phi^*(\Phi(x))$ otherwise. It is clear that $K \leq \Phi^*(\lambda_0)$. Now using the terminology defined in [HvdHL16, §5], the function Φ^* is an iterator for the logarithmically slow function Φ. As such, it satisfies $\Phi^*(x) = \log^* x + O(1)$, which corresponds to our claim.

To bound the product, it suffices to bound $\sum_i |\epsilon_{j,i}|$. Let $f_0(x) = \frac{4\log_2 x}{x}$, $f_1(x) = \frac{8\log_2 x}{x}$, and $f_2(x) = \frac{2+\log_2 x}{x}$, so that $\epsilon_{0,i} = f_0(\lambda_{i+1})$, $\epsilon_{1,i} = f_1(\lambda_i)$, $\epsilon_{2,i} = f_2(\lambda_{i+1})$. The functions f_j are decreasing for $x \geq \exp(1)$. In particular, we have $\epsilon_{1,i} \leq f_1(\lambda_{i+1})$. Consider the sequence of real numbers defined by $u_0 = 3$, $u_1 = 4$, $u_2 = 6$, $u_3 = 27$, and $u_{k+1} = 2^{u_k/3}$ for $k \geq 3$. This sequences diverges to infinity. Independently of the starting value λ_0, we have

$$\begin{aligned} \lambda_K &\geq 3 = u_0, \\ \lambda_{K-1} &\geq 4 = u_1, \\ \lambda_{K-2} &\geq 6 = u_2 \text{ by observing Table 3,} \\ \lambda_{K-3} &\geq 27 = u_3 \text{ again by Table 3,} \\ \lambda_{K-4} &\geq 2^{\lambda_{K-3}/3} \geq u_4 \text{ by Lemma 4.8,} \\ \lambda_{K-k} &\geq u_k \text{ for all } k \leq K. \end{aligned}$$

This yields

$$\sum_{i=0}^{K-1}\sum_{j=0}^{2}|\epsilon_{j,i}| = \sum_{k=0}^{K-1}\sum_{j=0}^{2}|\epsilon_{j,K-1-k}| \le \sum_{k=0}^{K-1}(f_0(\lambda_{K-k}) + f_1(\lambda_{K-1-k}) + f_2(\lambda_{K-k}))$$
$$\le \sum_{k=0}^{K-1}\sum_{j=0}^{2} f_j(\lambda_{K-k}) \le \sum_{k=0}^{K-1}\sum_{j=0}^{2} f_j(u_k) \le \sum_{k=0}^{\infty}\sum_{j=0}^{2} f_j(u_k).$$

The latter sum converges to an absolute constant. □

6.2. **Complexity of integer multiplication.**

Theorem 6.4. *The complexity* $\mathsf{M}_{new}(n)$ *of the algorithm presented in §5.4 to multiply* n*-bit integers is*

$$\mathsf{M}_{new}(n) = O(n \cdot \log n \cdot 4^{\log^* n}).$$

Proof. This theorem is a consequence of the results obtained thus far. Recall that in Algorithm 3 (FurerComplexMul), we have $N = O(n/(\log_2 n)^2)$ and $2^\lambda = O(\log_2 n)$. The input size of Algorithm 6 (MulZ) is $n_0 = \Theta((\log_2 n)^2)$ bits. We have

$$O(N \log N \cdot 2^\lambda \log n) = O((n/(\log_2 n)^2)(\log_2 n)^3) = O(n \log n).$$
$$N(3\lceil \log_{2^{\lambda+1}} N\rceil + 1) \cdot \mathsf{M}_{\mathcal{R}_0} \le O\left(\frac{n}{(\log_2 n)^2}\right) \cdot O\left(\frac{\log_2 n}{\log_2^{(2)} n}\right) \mathsf{M}_{\mathcal{R}_0}$$
$$\le O(n \log n) \cdot \frac{\mathsf{M}_{\mathcal{R}_0}}{n_0(\log_2 n_0)}.$$

We thus have, using $\log p_0 = \Theta(n_0)$:

$$\frac{\mathsf{M}_{\text{new}}(n)}{n \log n} = O(1) + O\left(\frac{\mathsf{M}_{\mathcal{R}_0}}{\log_2 p_0 \log_2^{(2)} p_0}\right) = O(1) + m_0$$

using the notation of Proposition 6.2. Let now A be a constant bounding the $O(1)$ in Proposition 6.2, let $C = A/3$, and let $e(i) = \prod_{0\le j\le 2}(1+\epsilon_{j,i})$. We have $4e(i) - 1 \ge 3$ so that $A \le (4e(i)-1)C$. Proposition 6.2 implies

$$m_i \le 4e(i)m_{i+1} + (4e(i)-1)C$$
$$(m_i + C) \le 4e(i)(m_{i+1} + C),$$

so that we get $m_0 = O(4^{\log^* n})$ by Lemma 6.3. Finally, this gives

$$\frac{\mathsf{M}_{\text{new}}(n)}{n \log n} = O(4^{\log^* n}).$$

□

7. Practical considerations

While our algorithm is mostly of theoretical interest, several points are worth mentioning, as an answer to the natural question of its practicality. Despite the title of this section, we are not reporting data on an actual implementation of our algorithm, but rather measurements that shed some light on its practical value.

7.1. **Adaptation of the asymptotically fast algorithm to practical sizes.** At the beginning of §5.4, we briefly alluded to a way to multiply two n-bit integers: pick a generalized Fermat number (not a priori prime) of the form $p_0 = P(2^{\lambda_0}, \lambda_0)$, for λ_0 such that $\lambda_0 2^{\lambda_0} \geq 2n$. Then use Algorithm 5 (MulR). This does not work asymptotically because computing roots of unity modulo $p_1 = \text{smallerprime}(p_0)$ cannot be done deterministically with good complexity. However, in practice, for say $n \leq 2^{64}$, Table 3 and Lemma 4.9 imply that p_1 would then be at most a 2048-bit prime, for which both the primality proof and the computation of roots can reasonably be assumed to be done once and for all. Therefore, the stumbling blocks that are relevant for the asymptotic analysis need not be considered as such for a practical implementation. This implies in particular that resorting to Algorithm 3 (FurerComplexMul), as we do in Algorithm 6 (MulZ) for asymptotic reasons, is not needed in practice.

Going further in this direction, we may in fact consider as a practical instance of our algorithm the more general procedure that follows Algorithm 1 with $\mathcal{R}_1 = \mathbb{Z}/p_1\mathbb{Z}$ as a base ring, where p_1 is a generalized Fermat prime. The aforementioned strategy can be regarded as Algorithm 1 with $\eta = 2^{\lambda_0}$, $N = 2^{\lambda_0}$ (still with $\lambda_0 2^{\lambda_0} \geq 2n$), at least in the case where $\beta = \text{batchsize}(p_0) = 1$.

Another alteration that we wish to make in practice is that our top-level multiplication need not use a negacyclic transform: whether we compute a product modulo $2^{2n}+1$ or $2^{2n}-1$ makes no difference when both inputs are less than or equal to $2^n - 1$. On the other hand, a "full" DFT of length N instead of a Half-DFT saves $3N$ multiplications in the base ring, which is not entirely negligible.

Finally, we note that for all sizes of practical interest, arithmetic in $\mathcal{R}_1 = \mathbb{Z}/p_1\mathbb{Z}$ will not be done with a Fourier-transform-based algorithm, because p_1 is only of very moderate size.

Taking into account all the remarks above, the only link that remains between the practical procedure that we envision and the algorithms (in particular, Algorithm 5 (MulR)) described in this article is that p_1 is a generalized Fermat prime. The developments in this article show that computing with generalized Fermat prime is asymptotically feasible, and yields a good complexity.

7.2. **Parameter choices for various input sizes.** In this section, we consider various input sizes n, and various candidate generalized Fermat primes $p_1 = {r_1}^{2^{\lambda_1}} + 1$. For combinations of these, we find values η and N (both powers of two) such that Algorithm 1 works. Let us briefly recall its structure: we write both n-bit integer inputs a and b in radix η, or equivalently as the evaluations at η of two polynomials of degree less than $N/2$. We multiply these polynomials in $\mathcal{R}_1[x]$. For this, we compute full N-point DFTs, then a pointwise product, and finally an inverse DFT. Arithmetic in $\mathcal{R}_1$, as in §5, uses representation in radix r_1. For this procedure to correctly compute the integer product $a \cdot b$, the following conditions must hold:

$$\begin{cases} N\eta^2 \leq p_1 & \text{(no overflow occurs in } \mathcal{R}_1\text{)}, \\ 2^{2n} \leq \eta^N - 1 & \text{(correct computation of the product of two } n\text{-bit integers)}, \\ N \mid p_1 - 1 & \text{(a principal } N\text{-th root of unity exists in } \mathcal{R}_1\text{)}. \end{cases}$$

In particular, N is the smallest power of two above $2n/\log_2 \eta$. When choosing η and N subject to the conditions above, we have some freedom. Ultimately, we wish to minimize the number of multiplications in $\mathcal{R}_1$, because we expect those to form the largest part of the computation time. More precisely, we wish to minimize the overall cost $(3E(N)+N)\mathsf{M}_{\mathcal{R}_1}$ of *expensive* multiplications as introduced in §3 ($\mathsf{M}_{\mathcal{R}_1}$ denotes the cost of one expensive multiplication in $\mathcal{R}_1$; we add N because of the pointwise products, and not $4N$ since here we do not use a half-DFT). Using the expression of $E(N)$, a rough estimate of the quantity to minimize is $\frac{n}{\log_2 \eta} \frac{\log_2 n}{\lambda_1 + 1} \mathsf{M}_{\mathcal{R}_1}$,

bit length of both operands: 2^{30}					
p_1	η	N	$3E(N)+N$	bit length of K.S.	lower bound
$\mathbf{984^{16}+1}$	$\mathbf{2^{64}}$	$\mathbf{2^{25}}$	$\mathbf{2^{25}\cdot(16=3\cdot 5+1)}$	$\mathbf{(2\cdot 10+4)\cdot 16=384}$	$\mathbf{2.68\cdot 10^1}$ s
$1984^{16}+1$	2^{64}	2^{25}	$2^{25}\cdot(16=3\cdot 5+1)$	$(2\cdot 11+4)\cdot 16=416$	$3.44\cdot 10^1$ s
$4016^{16}+1$	2^{64}	2^{25}	$2^{25}\cdot(16=3\cdot 5+1)$	$(2\cdot 12+4)\cdot 16=448$	$3.44\cdot 10^1$ s
$448^{32}+1$	2^{128}	2^{24}	$2^{24}\cdot(13=3\cdot 4+1)$	$(2\cdot 9+5)\cdot 32=736$	$3.82\cdot 10^1$ s
$884^{32}+1$	2^{128}	2^{24}	$2^{24}\cdot(13=3\cdot 4+1)$	$(2\cdot 10+5)\cdot 32=800$	$4.45\cdot 10^1$ s
$412^{64}+1$	2^{256}	2^{23}	$2^{23}\cdot(13=3\cdot 4+1)$	$(2\cdot 9+6)\cdot 64=1536$	$7.57\cdot 10^1$ s
$506^{128}+1$	2^{512}	2^{22}	$2^{22}\cdot(10=3\cdot 3+1)$	$(2\cdot 9+7)\cdot 128=3200$	$9.94\cdot 10^1$ s
bit length of both operands: 2^{40}					
p_1	η	N	$3E(N)+N$	bit length of K.S.	lower bound
$984^{16}+1$	2^{32}	2^{36}	$2^{36}\cdot(25=3\cdot 8+1)$	$(2\cdot 10+4)\cdot 16=384$	$8.57\cdot 10^4$ s
$\mathbf{1984^{16}+1}$	$\mathbf{2^{64}}$	$\mathbf{2^{35}}$	$\mathbf{2^{35}\cdot(22=3\cdot 7+1)}$	$\mathbf{(2\cdot 11+4)\cdot 16=416}$	$\mathbf{4.84\cdot 10^4}$ s
$4016^{16}+1$	2^{64}	2^{35}	$2^{35}\cdot(22=3\cdot 7+1)$	$(2\cdot 12+4)\cdot 16=448$	$4.84\cdot 10^4$ s
$448^{32}+1$	2^{64}	2^{35}	$2^{35}\cdot(19=3\cdot 6+1)$	$(2\cdot 9+5)\cdot 32=736$	$1.14\cdot 10^5$ s
$884^{32}+1$	2^{128}	2^{34}	$2^{34}\cdot(19=3\cdot 6+1)$	$(2\cdot 10+5)\cdot 32=800$	$6.66\cdot 10^4$ s
$412^{64}+1$	2^{256}	2^{33}	$2^{33}\cdot(16=3\cdot 5+1)$	$(2\cdot 9+6)\cdot 64=1536$	$9.54\cdot 10^4$ s
$506^{128}+1$	2^{512}	2^{32}	$2^{32}\cdot(13=3\cdot 4+1)$	$(2\cdot 9+7)\cdot 128=3200$	$1.32\cdot 10^5$ s
bit length of both operands: 2^{50}					
p_1	η	N	$3E(N)+N$	bit length of K.S.	lower bound
$984^{16}+1$	2^{32}	2^{46}	$2^{46}\cdot(31=3\cdot 10+1)$	$(2\cdot 10+4)\cdot 16=384$	$1.09\cdot 10^8$ s
$\mathbf{1984^{16}+1}$	$\mathbf{2^{64}}$	$\mathbf{2^{45}}$	$\mathbf{2^{45}\cdot(28=3\cdot 9+1)}$	$\mathbf{(2\cdot 11+4)\cdot 16=416}$	$\mathbf{6.31\cdot 10^7}$ s
$4016^{16}+1$	2^{64}	2^{45}	$2^{45}\cdot(28=3\cdot 9+1)$	$(2\cdot 12+4)\cdot 16=448$	$6.31\cdot 10^7$ s
$448^{32}+1$	2^{64}	2^{45}	$2^{45}\cdot(25=3\cdot 8+1)$	$(2\cdot 9+5)\cdot 32=736$	$1.54\cdot 10^8$ s
$884^{32}+1$	2^{128}	2^{44}	$2^{44}\cdot(25=3\cdot 8+1)$	$(2\cdot 10+5)\cdot 32=800$	$8.97\cdot 10^7$ s
$412^{64}+1$	2^{256}	2^{43}	$2^{43}\cdot(22=3\cdot 7+1)$	$(2\cdot 9+6)\cdot 64=1536$	$1.34\cdot 10^8$ s
$506^{128}+1$	2^{512}	2^{42}	$2^{42}\cdot(19=3\cdot 6+1)$	$(2\cdot 9+7)\cdot 128=3200$	$1.98\cdot 10^8$ s

TABLE 4. Estimated lower bound for the total cost of expensive multiplications in our algorithm depending on the prime used. Timings are based on the multiplication count and the measured time for the Kronecker-Schönhage bit length in the fifth column, on an Intel Xeon E7-4850v3 CPU (2.20GHz).

therefore for n constant we try to minimize

$$Q \approx \frac{\mathsf{M}_{\mathcal{R}_1}}{\lambda_1 \log_2 \eta}.$$

Thus, there is a trade-off to determine: when η grows, larger primes p_1 have to be used: $\mathsf{M}_{\mathcal{R}_1}$ increases, while $\frac{1}{\log_2 \eta}$ decreases. Since the cost $\mathsf{M}_{\mathcal{R}_1}$ is given by the bit length of the prime p_1, the η that we choose should be the largest η for which p_1 is valid (as per the first of the three conditions above). The number of expensive multiplications for various input sizes n and primes p_1 is reported in Table 4. We added in Table 4 the additional constraint that $\log_2 \eta$ be a multiple of the machine word size, to the extent possible (since N must be a power of two anyway, this constraint has no impact).

7.3. **Cost of multiplications in the underlying ring.** We now turn to the two last columns of Table 4. Our goal is to obtain a coarse lower bound on the time we expect our algorithm to take. Arithmetic in $\mathcal{R}_1$, and in particular multiplication, is our main focus. Elements of $\mathcal{R}_1$ are represented in radix r_1. We avoid the conversion between radix r_1 and binary representation by using Kronecker substitution: an element of $\mathcal{R}_1$, represented as a 2^{λ_1}-uple of integers in $[0, r_1)$, is transformed into

an integer of bit length

$$k = (2\lceil \log_2 r_1 \rceil + \lambda_1) \cdot 2^{\lambda_1}.$$

Multiplication in $\mathcal{R}_1$ is then done by multiplying these integers modulo $2^k + 1$ (we deal with signs in the same way as in Algorithm 5 (MulR)). We ignore the cost of converting this product back to radix r_1. This is likely to be at the very least a significant source of inaccuracy in our lower bounds.

The fifth column of Table 4 reports the bit length k introduced above, for the various generalized Fermat primes chosen. Based on this bit length, we determined experimentally on a target machine (Intel Xeon E7-4850v3 CPU clocked at 2.20GHz) the time taken by the function `mpn_mul` in the GMP library [Gt16], thereby giving a lower bound on the multiplication time in $\mathcal{R}_1$. We multiplied this lower bound by the number of expensive multiplication reported on the fourth column of Table 4, from which we deduced a lower bound on the multiplication time for n-bit integers using our algorithm.

The determination of the bit length above led us to restrict the set of generalized Fermat primes to consider: two such primes that lead to identical bit length lead to an identical time for internal multiplications. Therefore, we favor the largest generalized Fermat prime for each value of the Kronecker-Schönhage bit length k above. In our choice, we also favored primes such that r_1 has largest 2-valuation among the candidate values (e.g. both $984^{16} + 1$ and $1018^{16} + 1$ are primes, but we experimented with the former because the latter only allows a maximum transform length of 2^{16}).

We deduce from Table 4 that for realistic sizes, choosing the prime p_1 appropriately can lead to a speed-up of the order of 2 to 4, with all the necessary words of caution: as mentioned above, we deliberately omitted some conversion costs that are unlikely to be negligible in practice, and also our measurements are done with all operands in cache memory, which is quite probably optimistic.

7.4. **Comparison with Schönhage-Strassen.** Let us compare approximatively the cost of Schönhage-Strassen's algorithm to our algorithm. We can do two things. At least up to some size, we can run GMP's implementation of the Schönhage-Strassen algorithm, and obtain an actual computation time. Or we can do as we did in Table 4: count the number of small multiplications involved, and measure their cost. We did both, because the latter approach, which inherently gives a lower bound, is a fairer comparison given that a lower bound is all that we have in Table 4.

Roughly speaking, a Schönhage-Strassen multiplication of two 2^n-bit integers involves $2^{\lfloor (n+1)/2 \rfloor}$ multiplications of $2^{1+\lceil (n+1)/2 \rceil}$-bit modular integers. In truth, a well-tuned implementation of the Schönhage-Strassen algorithm uses all sorts of optimizations that are well outside the scope of this article (see e.g. [GKZ07]), so that this is a crude estimate.

In Table 5, we report how our lower bound compares to the lower bound that we obtain in this way on the running time of the Schönhage-Strassen algorithm. As we did in Table 4, the fourth column is computed by determining experimentally the individual cost of each of the underlying multiplications. For this, we timed GMP's internal routine `mpn_mul_fft`, as it is called in the implementation. The fifth column of Table 5 indicates the real computation time, measured experimentally (we modified GMP's internal `mp_size_t` type to go beyond 31 bits). Our measurements were limited by core memory, since the product of two 2^{40}-bit integers took 1.3TB of RAM. The comparison with the previous column shows that our lower bound on the Schönhage-Strassen time is within a factor of two of the real computation time, which is acceptable.

	Schönhage-Strassen algorithm				§7.1			
bit length	#internal products	internal bit length	lower bound	real time	#internal products	prime	internal bit length	lower bound
2^{30}	2^{15}	$\approx 2^{17}$	$9.73 \cdot 10^0$ s	$1.50 \cdot 10^1$ s	$2^{25} \cdot 16$	$984^{16}+1$	384	$2.68 \cdot 10^1$ s
2^{35}	2^{18}	$\approx 2^{19}$	$3.70 \cdot 10^2$ s	$6.03 \cdot 10^2$ s	$2^{30} \cdot 19$	$984^{16}+1$	384	$1.02 \cdot 10^3$ s
2^{40}	2^{20}	$\approx 2^{22}$	$1.63 \cdot 10^4$ s	$3.04 \cdot 10^4$ s	$2^{35} \cdot 22$	$1984^{16}+1$	416	$4.84 \cdot 10^4$ s
2^{45}	2^{23}	$\approx 2^{24}$	$7.90 \cdot 10^5$ s	—	$2^{40} \cdot 25$	$1984^{16}+1$	416	$1.76 \cdot 10^6$ s
2^{50}	2^{25}	$\approx 2^{27}$	$2.88 \cdot 10^7$ s	—	$2^{45} \cdot 28$	$1984^{16}+1$	416	$6.31 \cdot 10^7$ s
2^{55}	2^{28}	$\approx 2^{29}$	$1.05 \cdot 10^9$ s	—	$2^{50} \cdot 31$	$4016^{16}+1$	448	$2.23 \cdot 10^9$ s
2^{60}	2^{30}	$\approx 2^{32}$	$3.44 \cdot 10^{10}$ s	—	$2^{55} \cdot 34$	$4016^{16}+1$	448	$7.84 \cdot 10^{10}$ s

TABLE 5. Comparison of lower bounds on the running time of the Schönhage-Strassen algorithm and the practical algorithm described in 7.1. The right half is from Table 4. Timings measured on an Intel Xeon E7-4850v3 CPU (2.20GHz).

We conclude from Table 5 that an implementation of our algorithm will unlikely beat an implementation of the Schönhage-Strassen algorithm for sizes below 2^{40}. Above 2^{40}, the ratio of our lower bounds is only slightly more than two. We may speculate that an optimized implementation could compensate this gap.

One of the arguments in favor of our algorithm is that its memory locality is likely much better, because of the shallow recursion.

A direction to consider for optimization can be to improve on the time needed for internal multiplications. For example, we may represent elements of $\mathcal{R}_1$ in radix r_1^2 instead of r_1. In some cases, it might lead to a smaller bit length, at the expense of some extra conversion costs. For example for $p_1 = 1984^{16} + 1$, working in radix 1984^2 leads to polynomials of length 8, and a bit length of $8 \cdot 48 = 384$ bits, instead of 416 bits (see Table 4). Another possibility is to use the multipoint Kronecker substitution proposed by Harvey in [Har09]. For this same example, evaluating at $+2^{24}$ and -2^{24}, we can compute the product via two multiplications of two 192-bit integers, which might be faster. For both ideas however, we have not taken into account the conversion costs, and it seems difficult to be very confident about the induced benefit.

8. Conclusions

Our algorithm follows Fürer's perspective, and improves on the cost of the multiplications in the underlying ring. Although of similar asymptotic efficiency, it therefore differs from the algorithm in [HvdHL16], which is based on Bluestein's chirp transform, Crandall-Fagin reduction, computations modulo a Mersenne prime, and balances the costs of the "expensive" and "cheap" multiplications.

It is interesting to note that both algorithms rely on hypotheses related to the repartition of two different kinds of primes. It is not clear which version is the most practical, but our algorithm avoids the use of bivariate polynomials and seems easier to plug in a classical radix-2^λ FFT by modifying the arithmetic involved. The only additional cost we have to deal with is the question of the decomposition in radix r, and the computation of the modulo, which can be improved using particular primes. However, we do not expect it to beat Schönhage-Strassen for integers of size below 2^{40} bits.

A natural question arises: can we do better? The factor $4^{\log^* n}$ comes from the direct and the inverse FFT we have to compute at each level of recursion, the fact that we have to use some zero-padding each time, and of course the recursion depth, which is $\log^* n + O(1)$.

Following the same approach, it seems hard to improve on any of the previous points. Indeed, the evaluation-interpolation paradigm suggests a direct and an

inverse FFT, and getting a recursion depth of $\frac{1}{2}\log^* n + O(1)$ would require a reduction from n to $\log^{(2)} n$ at each step.

Acknowledgements

The authors are indebted to the anonymous referee, whose careful reading greatly helped enhance the presentation of this article.

References

[Ber01] Daniel J. Bernstein, *Multidigit multiplication for mathematicians*, 2001, `http://cr.yp.to/papers.html#m3`.

[BGS07] Alin Bostan, Pierrick Gaudry, and Eric Schost, *Linear recurrences with polynomial coefficients and application to integer factorization and Cartier-Manin operator*, SIAM J. Comput. **36** (2007), no. 6, 1777–1806.

[BH62] Paul T. Bateman and Roger A. Horn, *A heuristic asymptotic formula concerning the distribution of prime numbers*, Math. Comp. **16** (1962), no. 79, pp. 363–367 (English).

[Blu70] Leo I. Bluestein, *A linear filtering approach to the computation of discrete Fourier transform*, IEEE Trans. Audio and Electroacoustics **18** (1970), no. 4, 451–455.

[BZ10] Richard P. Brent and Paul Zimmerman, *Modern computer arithmetic*, Cambridge Univ. Press, 2010.

[CT65] James W. Cooley and John W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comp. **19** (1965), 297–301. MR 0178586 (31 #2843)

[DG02] Harvey Dubner and Yves Gallot, *Distribution of generalized Fermat prime numbers*, Math. Comp. **71** (2002), no. 238, 825–832. MR 1885631 (2002j:11156)

[DKSS08] Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi, *Fast integer multiplication using modular arithmetic*, 40th annual ACM symposium on Theory of computing (New York, NY, USA), STOC '08, ACM, 2008, pp. 499–506.

[Ell07] P.D.T.A. Elliott, *Primes in progressions to moduli with a large power factor*, Ramanujan J. **13** (2007), no. 1-3, 241–251 (English).

[Für89] Martin Fürer, *On the complexity of integer multiplication (extended abstract)*, Tech. Report CS-89-17, Pennsylvania State University, 1989.

[Für09] ———, *Faster integer multiplication*, SIAM J. Comput. **39** (2009), no. 3, 979–1005.

[GKZ07] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann, *A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm*, Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation (New York, NY, USA), ISSAC '07, ACM, 2007, pp. 167–174.

[Gt16] Torbjörn Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2016, version 6.1.0, `http://gmplib.org/`.

[Har09] David Harvey, *Faster polynomial multiplication via multipoint kronecker substitution*, J. Symbolic Comput. **44** (2009), no. 10, 1502 – 1510.

[HvdH16] David Harvey and Joris van der Hoeven, *Faster integer multiplication using plain vanilla FFT primes*, Math. Comp. (2016), Accepted for publication.

[HvdHL16] David Harvey, Joris van der Hoeven, and Grégoire Lecerf, *Even faster integer multiplication*, J. Complexity **36** (2016), 1–30.

[KO63] Anatolii A. Karatsuba and Yuri Ofman, *Multiplication of multidigit numbers on automata*, Soviet Physics-Doklady **7** (1963), 595–596, (English translation).

[Pap94] Christos M. Papadimitriou, *Computational complexity*, Addison-Wesley, Reading, Massachusetts, 1994.

[Pom77] Carl Pomerance, *On the distribution of amicable numbers*, J. Reine Angew. Math. (1977), 217–222.

[Sch82] Arnold Schönhage, *Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coeficients*, Computer Algebra, EUROCAM '82, European Computer Algebra Conference, Marseille, France, 5-7 April, 1982, Proceedings (Jacques Calmet, ed.), Lecture Notes in Comput. Sci., vol. 144, Springer, 1982, pp. 3–15.

[SS71] Arnold Schönhage and Volker Strassen, *Schnelle multiplikation großer Zahlen*, Computing **7** (1971), no. 3-4, 281–292 (German).

[Too63] Andrei L. Toom, *The complexity of a scheme of functional elements realizing the multiplication of integers*, Soviet Mathematics Doklady **3** (1963), 714–716, (English translation).

[vzGG99] Joachim von zur Gathen and Jürgen Gerhard, *Modern computer algebra*, Cambridge Univ. Press, New York, NY, USA, 1999.

Appendix A. Proof of Proposition 4.3

Proposition 4.3. *Let C_λ be as in Lemma 4.2. We have $\frac{1}{\lambda} = O(C_\lambda)$.*

Proof. We prove that there exists an absolute constant $C > 0$ such that $C_\lambda \geq \frac{C}{\lambda}$ for any $\lambda \geq 1$, where C_λ is defined as in Lemma 4.2.

The idea is to rely on the proof of the main theorem of [Pom77, §2], and to use the main result of [Ell07] for arithmetic progressions with "powerful moduli", since we consider arithmetic progressions $(q \cdot k + r)_k$ where q is a power of two.

Let $\mathcal{P}(x)$ be the set of primes smaller than x, and extend the notation of Lemma 4.2 to define

$$C_\lambda(x) = \frac{1}{2} \prod_{p \in \mathcal{P}(x)} \frac{1 - \chi_\lambda(p)/p}{1 - 1/p}.$$

Throughout this appendix, we use the shorthand notation $q = 2^{\lambda+1}$. Let $\pi(x, q, r)$ be the number of primes $\leq x$ congruent to r mod q. Let $G(x) = \pi(x, q, 1)$. We will use twice the Brun-Titchmarsh inequality, which says that

$$G(x) = \pi(x, q, 1) \leq 2(x/\phi(q))/\log(x/q) = 4x/(q \log(x/q)).$$

Let now $g(t) = G(t) - G(t-q)$. By construction, G and g are constant on intervals $[1+qi, 1+qi+q)$ for any integer $i \geq 0$, and $g(t)$ is equal to to 1 or 0 on that interval depending on whether $1 + qi$ is prime or not. Hence

$$g(1+qi) = \frac{1}{q} \int_{1+qi}^{1+qi+q} g(t)\,\mathrm{d}t \quad \text{and} \quad G(1+qi) = \frac{1}{q} \int_{0}^{1+qi+q} g(t)\,\mathrm{d}t.$$

Let $F(x) = -\log(1 - 2^\lambda/x) = -\log(1 - q/(2x))$, which is a decreasing, convex, and nonnegative function defined for $x > q/2$. Furthermore, since $\lambda \geq 0$, for $x \geq 1 + q/2$ we have $F(x) \leq \log 2$. Our goal is to find an asymptotic lower bound for the (logarithm of the) numerator of $C_\lambda(x)$ for large x (we impose $x \geq 1 + 2q$ below). Equivalently, we seek an upper bound for $\mathcal{S}(x) = \sum_{i=1}^{N} F(1+qi)g(1+qi)$, where we set $N = \lfloor (x-1)/q \rfloor$. Throughout the proof below, implicit constants $O(1)$ are uniform on λ —possibly for x larger than some bound that depends on λ, but that is not an issue since $C_\lambda = \lim_{x\to\infty} C_\lambda(x)$.

$$\begin{aligned}
\mathcal{S}(x) &= \sum_{i=1}^{N} F(1+qi)g(1+qi) = \sum_{i=1}^{N} F(1+qi)\frac{1}{q}\int_{1+qi}^{1+qi+q} g(t)\,\mathrm{d}t \\
&\leq \frac{1}{q}\sum_{i=1}^{N}\int_{1+qi}^{1+qi+q} F(t-q/2)g(t)\,\mathrm{d}t \quad \text{(because } F \text{ is convex)} \\
&\leq \frac{1}{q}\int_{1+q}^{x+q} F(t-q/2)g(t)\,\mathrm{d}t + \underbrace{\frac{1}{q}\int_{x+q}^{1+qN+q} F(t-q/2)g(t)\,\mathrm{d}t}_{O(1)} \\
&\leq F(x+q/2)G(x) - F(1+q/2)G(1+q) - \int_1^x F'(t+q/2)G(t)\,\mathrm{d}t + O(1).
\end{aligned}$$

Since $F(x+q/2) \leq \frac{q}{2x}$ and $G(x) \leq 4x/(q\log x/q))$, the first summand is bounded by $2/\log 2$ for $x \geq 2q$. Since $G(1+q) \leq 1$ and $G(t) = 0$ for $t < 2$ we have:

$$\mathcal{S}(x) \leq O(1) + \int_2^x \frac{q\pi(t,q,1)}{t(2t+q)}\,\mathrm{d}t \leq O(1) + \int_2^x \frac{q\pi(t,q,1)}{2t^2}\,\mathrm{d}t.$$

Elliott [Ell07] proved a theorem that relates $\pi(x, q, r)$ to its asymptotic estimate We state a very weak form of it, namely that there exists an absolute constant K

such that for any $\lambda \geq 0$ and t such that

$$\min(t^{1/3}\exp(-(\log\log t)^3), t^{1/2}\exp(-8\log\log t)) \geq q,$$

we have

$$\left|\pi(t,q,1) - \frac{2t}{q\log t}\right| < \frac{Kt}{q(\log t)^2}. \tag{A.1}$$

The condition above on t can be simplified. There exists an absolute constant H such that for any $x > 1$

$$\min(x^{1/3}\exp(-(\log\log x)^3), x^{1/2}\exp(-8\log\log x)) \geq (Hx)^{1/4}.$$

Thus, for $t \geq q^4/H$, Equation (A.1) holds. We rewrite the upper bound on $\mathcal{S}(x)$:

$$\mathcal{S}(x) \leq O(1) + \underbrace{\int_2^{q^4/H} \frac{q\pi(t,q,1)}{2t^2}\,\mathrm{d}t}_{I_0(\lambda)} + \underbrace{\int_{q^4/H}^{x} \frac{q\pi(t,q,1)}{2t^2}\,\mathrm{d}t}_{I_1(\lambda,x)}.$$

For $I_0(\lambda)$, by Brun-Titchmarsh we have

$$\begin{aligned} I_0(\lambda) &\leq \int_2^{q^4/H} \frac{2}{t\log(t/q)}\,\mathrm{d}t \leq \int_2^{q^3/H} \frac{2}{u\log(u)}\,\mathrm{d}u \\ &\leq 2\log\log(q^3/H) \leq 2\log\lambda + O(1). \end{aligned}$$

We use Elliott's theorem to bound $I_1(\lambda, x)$ (using the notations of Equation (A.1)):

$$\begin{aligned} \left|I_1(\lambda,x) - \int_{q^4/H}^{x} \frac{1}{t\log t}\mathrm{d}t\right| &\leq \int_{q^4/H}^{x} \frac{K}{t(\log t)^2}\mathrm{d}t \\ &\leq -\frac{K}{\log x} + \frac{K}{\log(q^4/H)} = O(1) \\ \text{so that } I_1(\lambda,x) &\leq \log\log x - \log\log(q^4/H) + O(1) \\ &\leq \log\log x - \log\lambda + O(1). \end{aligned}$$

Combining the bounds on I_0 and I_1, we have obtained:

$$\mathcal{S}(x) \leq \log\log x + \log\lambda + O(1).$$

The lower bound on $C_\lambda(x)$ follows: indeed, we have

$$\begin{aligned} -\log C_\lambda(x) &= \sum_{p\in\mathcal{P}(x)} \log\left(1 - \frac{1}{p}\right) + \mathcal{S}(x), \\ &\leq (-\gamma - \log\log x + o(1)) + (\log\log x + \log\lambda + O(1)), \\ \log C_\lambda(x) &\geq O(1) - \log\lambda. \end{aligned}$$

Hence $C_\lambda(x) \geq A/\lambda$ for some absolute constant A, and x large enough. It follows that $C_\lambda \geq A/\lambda$, as claimed. We notice that the multiplier affecting $\log\lambda$ above, and hence the exponent of λ in our lower bound, can be directly traced to the use of the Brun-Titchmarsh inequality in bounding $I_0(\lambda)$. □

E-mail address: svyatoslav.covanov@loria.fr, emmanuel.thome@inria.fr

Université de Lorraine, CNRS, INRIA, LORIA, F-54000 Nancy, France