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 $O(n \cdot \log n \cdot K^{\log^\star n})$. 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, 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 $O(n^{\log_2 3})$. 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]. 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 $O(n \cdot \log n \cdot \log^{(2)} n)$, 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 $O(n \cdot \log n \cdot 2^{O(\log^\star n)})$ (in contrast, radix-2 FFT is sufficient for the Schönhage-Strassen algorithm). Fürer's result was an acclaimed improvement over 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 over 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 sharper complexity analysis that allows one to make the complexity more explicit, namely $O(n \cdot \log n \cdot 8^{\log^\star n})$ and even $O(n \cdot \log n \cdot 4^{\log^\star n})$ conjecturally.

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

This article presents another variant of Fürer's algorithm. Our algorithm reaches the complexity $O(n \cdot \log n \cdot 4^{\log^\star n})$ 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 4.3. *Let $\lambda \geq 1$ 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 $R \leq r < R(1 + 2\lambda^2)$.*

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 $4^{\log^\star n}$ and not $8^{\log^\star n}$ is original, and made possible only based on the fact that we use generalized Fermat primes.

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 bit sizes above a certain threshold. Section 5 uses material developed in the previous sections and presents our new algorithm (in fact three 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. Section 8 discussed briefly how our heuristic could be changed into a weaker one. Appendix A gives numerical verification code for various inequalities used in the proofs, and Appendix B details our calculations supporting Hypothesis 4.3.

Throughout the article, $\log_2 x$ denotes the logarithm in base 2, and $\log x$ denotes the classical 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 $\log_2$).

2. Background

2.1. **Integers to polynomials.** Let a and b be integers to be multiplied. Let $n = |ab|$ be the bit size of the product $c = ab$. 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.)

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.

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

Input: a, b two integers to be multiplied; we let $n = \log_2 |ab|$
η a power of two; we let $N = \lceil n/\log_2 \eta \rceil$
$\mathcal{R}$ a ring where integers below $N\eta^2$ are unambiguously represented
$\mathcal{S} \subset \mathcal{R}$ a set of N evaluation points
Output: $c = a \cdot b$
function MultiplyIntegersViaMultipointEvaluation$(a, b, \eta, \mathcal{R})$
Let $A(x) \in \mathbb{Z}[x]$ 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

2.2. **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 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 $\mathrm{DFT}_{N,w}$ 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 $N-1$ 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 $N = N_1 N_2$ 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.

The notation $\mathrm{DFT}_{N,\omega}$ denotes a mathematical object rather than an algorithm. Therefore, we need to detail how recursive computations of $\mathrm{DFT}_{N_k,\omega_K}$ 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 2^{k-1} transforms of length two (often called "butterflies"), then recurse with two transforms of length 2^{k-1}. We use the notation Radix2FFT(N, ω, A) for this algorithm.

Algorithm 2 General Cooley-Tukey FFT or order $N = N_1 N_2$

Input: $A = \sum_{i=0}^{N-1} a_i X^i \in \mathcal{R}[X]/X^N - 1$ and ω a principal N-th root of unity. Let $\omega_1 = \omega^{N_2}$ and $\omega_2 = \omega^{N_1}$.
Output: $\hat{A} = \mathrm{DFT}_{N,\omega}(A) = A(1) + A(\omega)X + \cdots + A(\omega^{N-1})X^{N-1}$
function CooleyTukeyFFT(N_1,N_2,ω,A)
  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**
    $\hat{B}_j \leftarrow \mathrm{DFT}_{N_1,\omega_1}(B_j)$ ▷ $w_1 = w^{N_2}$ is a principal N_1-th root
    $\hat{B}_j \leftarrow \hat{B}_j(\omega^j X)$
  end for
  Let $(S_i(Y))_i \in \mathcal{R}[Y]^{N_1}$ be such that $\sum_{j<N_2} \hat{B}_j(X)Y^j = \sum_{i<N_1} S_i(Y)X^i$
  for $i \in [\![0, N_1 - 1]\!]$ **do**
    $\hat{S}_i(Y) \leftarrow \mathrm{DFT}_{N_2,\omega_2}(S_i)$ ▷ $w_2 = w^{N_1}$ is a principal N_2-th root
  end for
  return $\sum_{j<N_1} \hat{S}_i(X^{N_1})X^i$
end function

- "large-radix FFT": More generally, for a length $N = 2^{uq+r}$ with $r < u$, and $q > 0$, compute $N/2^u$ transforms of length 2^u, then recurse with transforms of length $N/2^u$. When all recursive calls are unrolled, we see that the computation is based on transforms of length 2^u (or 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 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 denote by Half-DFT$_{N,\omega}$, the multi-evaluation at odd powers of a $2N$-th root of unity ω. A half-DFT of length N can be computed at essentially the same cost as a DFT of length N by applying proper scaling. More precisely, to multi-evaluate $A(X)$ at $\omega, \omega^3, \ldots, \omega^{2N-1}$, we compute Half-DFT$_{N,\omega}(A(X)) = \mathrm{DFT}_{N,\omega^2}(A(\omega X))$. 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

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

We shall not discuss this point further.

2.3. **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 §2.1 with FFT-based multi-evaluation and interpolation as in §2.2, we obtain quasi-linear integer multiplication algorithms. We identify several tasks whose cost contribute 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);
- point-wise products of elements of $\mathcal{R}$. Recursive calls to the integer multiplication algorithm are of course possible;
- 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 $\mathsf{M}(n) = O(C(N) \cdot K_{\mathrm{FFT}}(\mathcal{R})) + O(N \cdot K_{\mathrm{PW}}(\mathcal{R}))$, with the following notations.

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

The costs $K_{\mathrm{PW}}(\mathcal{R})$ and $K_{\mathrm{FFT}}(\mathcal{R})$ are not necessarily equal.

2.4. **Choice of the base ring.** Depending on $\mathcal{R}$, the bit complexity estimates of §2.3 can be made more precise. Some rings have special roots of unity that allow faster operations (multiplication, addition, subtraction in $\mathcal{R}$) 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 $N = \Theta(\frac{n}{\log_2 n})$ (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

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

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

Schönhage and Strassen proposed the alternative $\mathcal{R} = \mathbb{Z}/(2^t + 1)\mathbb{Z}$, in which 2 is a principal $2t$-th root of unity.

- We pick $t = \Theta(\sqrt{n})$ and the transform length is $N = 2t = \Theta(\sqrt{n})$.
- 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 call to the Schönhage-Strassen's algorithm, so $K_{\mathrm{PW}}(\mathcal{R}) = \mathsf{M}(O(\sqrt{n}))$.

This leads to the complexity equation $\mathsf{M}(n) = O(n \log_2 n) + 2\sqrt{n}\mathsf{M}(\sqrt{n})$, and eventually to the complexity $\mathsf{M}(n) = O(n \cdot \log_2 n \cdot \log_2^{(2)} n)$.

3. Fürer-type bounds

The choices mentioned in §2.4 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 $\log_2 n \cdot \log_2^{(2)} n \cdots \log_2^{(\log^\star n)} n$ 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 w_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 w_N explicitly. We verify easily that:

- w_N is a principal N-th root of unity.
- $w_N^{N/2^{\lambda+1}}$ maps to $x = \exp((2j+1)i\pi/2^\lambda)$ in $\mathcal{R}_j$, so that $w_N^{N/2^{\lambda+1}} = x$ 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 $q > 1$, 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 few remaining *expensive* multiplications that occur within the recursion. Those correspond to the scaling operation $\hat{B}_j \leftarrow \hat{B}_j(\omega^j X)$ in Algorithm 2. Their count $E(N)$ satisfies $E(N) = 2^{\lambda+1} E(\frac{N}{2^{\lambda+1}}) + N$, 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 n-bit integers, Fürer selects $2^\lambda = 2^{\lceil \log^{(2)} n \rceil}$ 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 $N \leq \lceil 2n/\log^2 n \rceil$. This decomposition is described in Algorithm 3 (FurerComplexMul).

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 and point-wise products. For these, we use Kronecker substitution: we encode elements of $\mathcal{R}$ as integers of bit size $O((\log_2 n)^2)$, 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 size of elements of $\mathcal{R}$, that is $O(2^\lambda \log n)$. 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 $\mathsf{M}(n) = n \log n (2^{d \log^\star \sqrt[4]{n}} - d')$ for some constants $d, d' > 0$, so that $\mathsf{M}(n) = n \cdot \log n \cdot 2^{O(\log^\star n)}$.

Algorithm 3 Multiplication of integers with Fürer's algorithm

Input: a and b two n-bit integers
Output: $a \cdot b \mod 2^n + 1$
function FurerComplexMul(a,b,n)
  Let $\lambda = \lceil \log_2^{(2)} n \rceil$, $\eta = 2^{2^{2\lambda-1}}$, $N = \lceil n/\log_2 \eta \rceil = \lceil n/2^{2\lambda-1} \rceil$,
  Let $\mathcal{R} = \mathbb{C}[x]/(x^{2^\lambda} + 1)$, and $\omega = \omega_{2N}$ as in §3.1.
  Let $A_0(X) \in \mathbb{Z}[X]$ be such that $A_0(\eta) = a$; define $B_0(X)$ likewise.
  Let $\tilde{A}(X, x) \in \mathbb{C}[X, x]$ be such that $\tilde{A}(X, 2^\lambda) = A_0(X)$. Define $\tilde{B}$ likewise.
  Map $\tilde{A}$ and $\tilde{B}$ to polynomials A and B in $\mathcal{R}[X]/X^N + 1$
  $\hat{A} \leftarrow \mathsf{LargeRadixFFT}(N, \omega^2, 2^{\lambda+1}, A(\omega X))$ ▷ computes Half-DFT$_{N,\omega}(A)$
  $\hat{B} \leftarrow \mathsf{LargeRadixFFT}(N, \omega^2, 2^{\lambda+1}, B(\omega X))$ ▷ computes Half-DFT$_{N,\omega}(B)$
  $\hat{C} \leftarrow \mathsf{PointwiseProduct}(\hat{A}, \hat{B})$
  $C \leftarrow \frac{1}{N}\mathsf{LargeRadixFFT}(N, \omega^{-2}, 2^{\lambda+1}, \hat{C})(w^{-1}X)$ ▷ computes Half-IFT$_{N,\omega}(\hat{C})$
  Lift C to $\tilde{C} \in \mathbb{C}[X, x]$
  return $\tilde{C} \circ (\eta, 2^\lambda)$
end function

As mentioned in the introduction, Harvey, van der Hoeven and Lecerf in [HvdHL16], and Harvey and van der Hoeven in [HvdH16] propose new algorithms achieving 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.

3.3. **Adjusting the complexity equation.** Equation (3.1), and in fact all recursive complexity equations in this article, involve a constant 3. This corresponds to the cost of two direct transforms, and one inverse transform.

In fact, we can identify separately the cost of the transforms and the cost of multiplications, although we do not systematically do so for brevity. This matters, because the computation of transforms involves multiplications by roots of unity, which are constant data: transforms for those may be precomputed. The consequence is that when we take this into account, complexity equations such as Equation (3.1) can be restated. If the transform of one of the operands of each bilinear operation that occurs is precomputed (e.g. $\hat{B}$ in Algorithm 3), the multiplicative constant 3 is replaced by 2, and N multiplication by scaling factors are spared. We use the notation M' do reflect this "with precomputations" variant. In the case of Fürer's algorithm, we obtain:

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

Such refinements imply that we need to separately count the cost of precomputations, e.g. together with the first call at the top level. Since the same roots of unity are used multiple times throughout the recursive calls, we obtain an overall improvement. This idea is already present in [HvdHL16, §6].

4. Admissible generalized Fermat primes

This section defines admissible generalized Fermat primes, and a descending chain of such primes. This section is independent of the previous sections.

Definition 4.1 (Admissible generalized Fermat primes)**.** A generalized Fermat prime $p = r^{2^\lambda} + 1$, also denoted $p = P(r, \lambda)$, is called *admissible* whenever $\lambda \geq 3$ and r is such that $\lambda \leq \log_2 r < (\lambda + \log_2 \lambda) \log_2(\lambda + \log_2 \lambda) - \lambda/2$.

For notational ease, in Definition 4.1 and throughout this article, whenever we mention a generalized Fermat prime p, we actually consider the pair (r, λ) rather than the prime 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 prime".

The "admissibility" criterion in Definition 4.1 relates naturally to the constraints that are set on r. It will be used to prove the correctness of the algorithms presented in Section 5.

4.1. **Abundance of generalized Fermat primes.** The existence of generalized Fermat primes (admissible or not) in integer intervals can be obtained via the Bateman-Horn conjecture [BH62]. We use it to state the following lemma.

Lemma 4.2. *Fix an integer $\lambda > 0$. Let $\alpha > 1$ be a real number (possibly depending on λ). If the Bateman-Horn conjecture holds for the polynomial $f(x) = x^{2^\lambda} + 1$, then for any large enough real number R, there exists an integer $r \in [R, \alpha R[$ such that $p = f(r) = P(r, \lambda) = r^{2^\lambda} + 1$ is a generalized Fermat prime.*

Proof. For a real number R, let $E(R, \lambda)$ be the number of integers $r \in [1, R[$ such that $f(r)$ is prime. The Bateman-Horn conjecture [BH62] states that there exists a positive constant C (depending on λ) such that as R grows, we have

$$E(R, \lambda) \sim C \int_2^R \frac{\mathrm{d}t}{\log t} \sim C \frac{R}{\log R}.$$

Let now $\epsilon > 0$ be such that $\frac{1+\epsilon}{1-\epsilon} \leq \frac{2\alpha}{1+\alpha}$. Fix B large enough, and at least $B > \alpha^{2/(\alpha-1)}$, such that $\left| \frac{E(R,\lambda)}{CR/\log R} - 1 \right| < \epsilon$ whenever $R > B$. We then have

$$\begin{aligned} E(\alpha R, \lambda) - E(R, \lambda) &\geq C(1-\epsilon)\frac{\alpha R}{\log(\alpha R)} - C(1+\epsilon)\frac{R}{\log R}, \\ &\geq C\frac{R}{\log R}\left((1-\epsilon)\alpha\frac{\log R}{\log \alpha R} - (1+\epsilon)\right), \\ &\geq C\frac{R}{\log R}\left((1-\epsilon)\frac{2\alpha}{\alpha+1} - (1+\epsilon)\right), \\ &= \Omega(R/\log R). \end{aligned}$$

The third inequality above stems from $\frac{\log R}{\log \alpha R} = \frac{1}{1+\frac{\log \alpha}{\log R}} \geq \frac{2}{1+\alpha}$, which is an easy consequence of $R > B > \alpha^{2/(\alpha-1)}$. It follows immediately that $E(\alpha R, \lambda) \geq 1 + E(R, \lambda)$ for R large enough, which is equivalent to our claim. □

One bit of quantitative information that is lacking in the previous statement is how the minimal valid R in Lemma 4.2 evolves as a function of λ. We formulate the following hypothesis.

Hypothesis 4.3. *Let $\lambda \geq 1$ be an integer. For any real number R such that $2^\lambda \leq R \leq 2^{2\lambda}$, there exists a generalized Fermat prime $p = P(r, \lambda)$ such that $R \leq r < R(1 + 2\lambda^2)$.*

Hypothesis 4.3 serves in particular to assert that *admissible* generalized Fermat primes are abundant enough, by the following trivial property.

Lemma 4.4. *Let $\lambda \geq 4$ be an integer. If r is an integer in the interval $[2^\lambda, 2^{2\lambda}(1 + 2\lambda^2)[$ and $p = P(r, \lambda)$ is a generalized Fermat prime, then p is admissible.*

λ	Actual number	Estimate	λ	Actual number	Estimate
1	2	2	7	31	36
2	7	8	8	89	102
3	3	9	9	140	118
4	24	22	10	160	144
5	28	28	11	151	145
6	49	38	12	195	188

TABLE 1. Number of generalized Fermat primes $r^{2^\lambda}+1$ with r in $[R, R\cdot(1+2\lambda^2)[$ with $R=2^\lambda$, compared with the asymptotic estimate which follows from the Bateman-Horn conjecture. Hypothesis 4.3 asserts that the second column is never zero.

Hypothesis 4.3 is in fact stronger than what would be strictly necessary for the complexity analysis presented in this paper to reach the asymptotic complexity we claim in this article. This aspect is briefly discussed in Section 8. Table 1 provides empirical evidence for Hypothesis 4.3, for $R=2^\lambda$, which is empirically the hardest case. On the one hand, following the notations of Lemma 4.2, the Bateman-Horn conjecture gives an explicit *asymptotic* estimate of $E((1+2\lambda^2)R,\lambda)-E(R,\lambda)$ (this estimate is also studied by [DG02]). On the other hand the actual number of generalized Fermat primes with r in the interval $[R, R(1+2\lambda^2)[$ can be computed, as long as the interval width remain manageable. The comparison of these two values supports the heuristic (see also Appendix B).

Throughout the rest of the article, Hypothesis 4.3 is tacitly assumed.

4.2. **Chains of generalized Fermat primes.** The following proposition shows how from admissible generalized Fermat primes, we can build smaller generalized Fermat primes. For large enough inputs, these smaller primes are in turn admissible.

Proposition 4.5. *Let $\lambda\geq 6$, and let $p=P(r,\lambda)=r^{2^\lambda}+1$ be an admissible generalized Fermat prime. 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:

(i) $0\leq \log_2\beta<\lambda'$,

(ii) $\phi(\log_2\beta)\geq \lambda' 2^{\lambda'}$,

(iii) Given $R'=2^{\phi(\log_2\beta)/2^{\lambda'}}$, there exists an integer $r'\in[R', R'(1+2\lambda'^2)[$ 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, p' is admissible.*

In anticipation for the proof of Proposition 4.5, we prove the following bounds.

Lemma 4.6. *Let λ and λ' be as in Proposition 4.5. We have*

$$\log_2(\lambda+2)<\lambda'<3\log_2\lambda-1<\lambda.$$

Proof. Since p is admissible, we have

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

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

$$\lambda'\leq 1+\log_2^{(2)}(1+2^\lambda((\lambda+\log_2\lambda)\log_2(\lambda+\log_2\lambda)-\lambda/2)).$$

λ	λ'
$3 \le \lambda \le 4$	3
$\lambda = 5$	$3, 4$
$6 \le \lambda \le 10$	4

λ	λ'
$11 \le \lambda \le 12$	$4, 5$
$13 \le \lambda \le 25$	5
$26 \le \lambda \le 27$	$5, 6$

λ	λ'
$28 \le \lambda \le 55$	6
$56 \le \lambda \le 58$	$6, 7$
$59 \le \lambda$	≥ 7

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

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

The lower bound above 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 Section 6. Possible values for λ' are given in Table 2. In particular, $\lambda \ge 6$ implies $\lambda' \ge 4$.

Proof of Proposition 4.5. 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 the left and right hand sides are

$$\begin{aligned} 2\lambda' 2^{\lambda'} &\ge 2(\lambda + \log_2 \lambda) \log_2(\lambda + \log_2 \lambda), \\ \phi(0) &\le \lambda + 2((\lambda + \log_2 \lambda) \log_2(\lambda + \log_2 \lambda) - \lambda/2). \end{aligned}$$

Right-hand sides above are equal (this is how the bound on $\log_2 r$ in Definition 4.1 is chosen in the first place).

We have proved that there exists an integer k such that $0 \le k < \lambda'$, and that $\phi(k) \le 2\lambda' 2^{\lambda'} \le 2\phi(k)$. Let $\beta = 2^k$, so that $\log_2 \beta = k$, and hence (i) holds. We have that

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

This implies (ii). Finally, $R' = 2^{\frac{\phi(\log_2 \beta)}{2^{\lambda'}}}$ is such that $2^{\lambda'} \le R' \le 2^{2\lambda'}$. Hypothesis 4.3 then implies (iii), and concludes the proof. Admissibility of p' follows from Lemma 4.4. □

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

Lemma 4.7. *Let $p = P(r, \lambda)$ as in Proposition 4.5. Let $\beta = \text{batchsize}(p)$ and $p' = \text{smallerprime}(p)$. We have*

(i) $1 \le \frac{\log_2 p'}{2\beta \log_2 r} \le 1 + \frac{3 \log_2 \lambda'}{\lambda' - 1} \le 3$. *In particular,* $\frac{\log_2 p'}{\beta \log_2 r} = 2 + o(1)$.

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

Proof. We follow the notations of Proposition 4.5. The lower bound in (i) is easy:

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

The upper bound requires more work. On the one hand, we have

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

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

$$\begin{aligned}
\log_2 p' &= \log_2((p'-1)+1) = \log_2(p'-1) + \log_2(1 + 1/(p'-1)) \\
&\leq 2^{\lambda'} \log_2 r' + 1 \leq 2^{\lambda'} \log_2 R' + 2^{\lambda'} \log_2(1 + 2\lambda'^2) + 1 \\
&\leq \phi(\log_2 \beta) + 2^{\lambda'} \log_2(1 + 2\lambda'^2) + 1 \qquad (4.1) \\
&\leq 2\beta \log_2 r + (2^{\lambda'} - 2) + 2^{\lambda'} \log_2(1 + 2\lambda'^2) + 1 \\
&\leq 2\beta \log_2 r + 2^{\lambda'} \cdot 3 \log_2 \lambda' \quad \text{since } \lambda' \geq 4.
\end{aligned}$$

This proves (i), since $\lambda' \geq 4$ implies $3 \log_2 \lambda' \leq 2(\lambda' - 1)$.

The lower bound in statement (ii) is trivial. The upper bound is derived from inequality (4.1) above. We use $\frac{\phi(\log_2 \beta)}{2^{\lambda'}} \leq 2\lambda'$ (see Proposition 4.5). This gives

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

□

5. Three new algorithms

We now see how we can design an asymptotically fast integer multiplication algorithm that uses rings of integers modulo generalized Fermat primes. First, we show how generalized Fermat primes provide a context that is well adapted to the multiplication of polynomials, with complexity benefits similar to Fürer's algorithm. We then see, using Proposition 4.5, how we can multiply integers modulo generalized Fermat primes in a recursive manner. We finally combine these results to obtain an integer multiplication algorithm.

In the presentation of the algorithms in this section, we deal with roots of unity modulo generalized Fermat primes. Those are computed in the same way as for Fürer's algorithm in Section 3, using Lagrange interpolation. We assume that these roots of unity have been *precomputed*, and the cost of this precomputation is omitted from the complexity estimates. We do take it briefly into account when we present the final adjustments to the complexity estimates in 6.

5.1. **Multiplication of polynomials modulo generalized Fermat primes.** Let $\lambda \geq 4$. Let $p = P(r, \lambda) = r^{2^\lambda} + 1$ be a generalized Fermat prime, and let $\mathcal{R} = \mathbb{Z}/p\mathbb{Z}$. Algorithm 4 (MulRX) is pretty straightforward and insists on the importance of radix conversions to draw profit from the easy multiplications by roots of unity.

The complexity analysis of Algorithm 4 depends crucially on the assumption that $2N$ divides $p - 1$, so that a negacyclic convolution can be used on line 5. Furthermore, because of the special form of generalized Fermat primes, many multiplications by roots of unity are trivial in $\mathcal{R}$, provided that elements are represented in radix r. We can therefore use the same arguments as in Section 3.

On lines 4 and 6, we convert between radix two and radix r, with at most 2^λ terms. Using recursive base conversion algorithms (see [BZ10, §1.7.2]), we can do this with complexity $O(\lambda \mathsf{M}(\log p))$ per coefficient.

Algorithm 4 Multiplication in $\mathcal{R}[X]$ with $\mathcal{R} = \mathbb{Z}/p\mathbb{Z}$ and $p = r^{2^\lambda} + 1$, $\lambda \geq 4$.

1: **Input:** $A, B \in \mathcal{R}[X]/(X^N + 1)$, coefficients represented as binary integers.
   $\mathcal{R}$ must have a principal $2N$-th root of unity.
2: **Output:** $A \cdot B$
3: **function** MulRX(A,B)
4: Rewrite coefficients of A and B in radix r.
5: $C \leftarrow A \cdot B \mod X^N + 1$.
6: Rewrite coefficients of C as binary integers.
7: **return** C
8: **end function**

These arguments are combined to express the complexity of multiplication in $\mathcal{R}[X]/X^N + 1$, which we denote by $\mathsf{M}_{\mathcal{R}[X],N}$. We use the notation $\mathsf{M}_{\mathcal{R}}$ to represent the cost of the remaining expensive multiplications in $\mathcal{R}$ (for elements represented in radix r, as in line 5 of Algorithm 4). The equation for $\mathsf{M}_{\mathcal{R}}$ is covered by the forthcoming analysis in §5.2.

$$\mathsf{M}_{\mathcal{R}[X],N} \leq N(3\lceil \log_{2^{\lambda+1}} N \rceil + 1)\mathsf{M}_{\mathcal{R}} + O(N \cdot \lambda \cdot \mathsf{M}(\log p)) + O(N \cdot \log N \cdot \log p).$$

5.2. **Multiplication modulo generalized Fermat primes.** For this section, let $\lambda \geq 6$. Let $p = P(r, \lambda) = r^{2^\lambda} + 1$ be an admissible generalized Fermat prime, and let $\mathcal{R} = \mathbb{Z}/p\mathbb{Z}$. Algorithm 5 (MulR) sketches our procedure to multiply elements in $\mathcal{R}$, provided they are represented in radix r, that is, $a = \sum_{j<2^\lambda} a_j r^j$.

Algorithm 5 Multiplication in $\mathcal{R} = \mathbb{Z}/p\mathbb{Z}$, with $p = r^{2^\lambda} + 1$ admissible, $\lambda \geq 6$.

1: **Input:** a and b two integers modulo p.
   Both a and b must be represented in radix r: $a = \sum_{j<2^\lambda} a_j r^j$.
2: **Output:** $a \cdot b \mod p$, represented in radix r
3: **function** MulR(a,b)
4: Let $\beta = \text{batchsize}(p)$, $p' = P(r', \lambda') = \text{smallerprime}(p)$, and $\mathcal{R}' = \mathbb{Z}/p'\mathbb{Z}$.
5: Let $N' = 2^\lambda/\beta$. ▷ N' is a power of two.
6: Let $\tilde{A}(X) \in \mathbb{Z}[X]$ be such that $\tilde{A}(r^\beta) = a$; define $\tilde{B}(X)$ likewise.
7: Map $\tilde{A}, \tilde{B}$ to $A, B \in \mathcal{R}'[X]/(X^{N'} + 1)$.
8: $C \leftarrow A \cdot B \mod X^{N'} + 1$ using Algorithm 4 (MulRX).
9: Lift C to $\tilde{C} \in \mathbb{Z}[X]$.
10: Rewrite coefficients in radix r.
11: Compute $\tilde{C}(r^\beta) = c$. ▷ The result is defined modulo $(r^\beta)^{N'} + 1 = p$.
12: **return** c
13: **end function**

The conditions $\lambda \geq 6$ and p admissible allow Proposition 4.5 to be used in defining β and p' in Algorithm 5. Several additional verifications are required to confirm that Algorithm 5 is valid. We summarize the facts to be verified in the following lemma.

Lemma 5.1. *In Algorithm 5, we have the following:*

(*i*) *$\mathcal{R}'$ has a principal $2N'$-th root of unity;*
(*ii*) *Coefficients of $C = A \cdot B$ do not wrap around in $\mathcal{R}'$;*
(*iii*) *c is equal to $ab \bmod p$.*

Proof. To prove (*i*), notice first that $\lambda' \geq 4$ so that $p' \geq 2^{4 \cdot 2^4}$, 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 $\lambda \geq 2$.

To verify (ii), we compute a bound on the coefficients of the product $\tilde{A} \cdot \tilde{B}$. Both polynomials have at most N' coefficients, by construction. Coefficients of their product are thus bounded by $N'(r^\beta)^2$, whose logarithm is $2\beta \log_2 r + \lambda - \log_2 \beta = \phi(\log_2 \beta)$, following the notation of Proposition 4.5. Now again following notations of Proposition 4.5, we have $p' \geq R'^{2^{\lambda'}} \geq 2^{\phi(\log_2 \beta)} \geq N'(r^\beta)^2$.

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 $(r^\beta)^{N'} + 1 = p$. □

Statement (i) in Lemma 5.1 guarantees that line 8 of Algorithm 5 can indeed use Algorithm 4 (MulRX).

Algorithm 5 uses radix conversions (in particular, coefficients of $\tilde{A}$, $\tilde{B}$ and $\tilde{C}$ on lines 6 and 9 are represented as binary integers). We need to determine their cost. On line 6, we convert a β-term expansion in radix r to a binary integer bounded by r^β. On line 10, we convert a binary integer bounded by p' to an expansion in radix r, which takes $\frac{\log_2 p'}{\log_2 r} \leq 6\beta$ terms by Lemma 4.7. Both operations can be done at a similar cost using recursive base conversion algorithms, for a cost of $O(\log \beta \cdot \mathsf{M}(\beta \log r))$. The latter can be expressed as $O(\lambda' \cdot \mathsf{M}(\log p'))$ by (i) in Proposition 4.5, and Lemma 4.7.

Using these arguments and the complexity equation for Algorithm 4 (MulRX), we obtain the following formula for the cost $\mathsf{M}_\mathcal{R}$ of the multiplication in $\mathcal{R}$:

$$\begin{aligned}\mathsf{M}_\mathcal{R} &= \mathsf{M}_{\mathcal{R}'[X],N'} + O(N'\lambda'\mathsf{M}(\log p')) + O(\log p) \\ &= N'(3\lceil \log_{2^{\lambda'+1}} N' \rceil + 1)\mathsf{M}_{\mathcal{R}'} + O(N'\lambda'\mathsf{M}(\log p')) + O(N' \cdot \log N' \cdot \log p').\end{aligned}$$

The following lemma bounds the transform length N'. A complete asymptotic analysis of all terms in the complexity equation is postponed to Section 6.

Lemma 5.2. *Using the notations as above, we have*

$$N' \leq 2\frac{\log_2 p}{\log_2 p'}(1 + \frac{3\log_2 \lambda'}{\lambda' - 1}) < 6\frac{\log_2 p}{\log_2 p'}.$$

Proof. This follows from Lemma 4.7. We have $2^\lambda \log_2 r \leq \log_2 p$, therefore

$$N' = \frac{2^\lambda}{\beta} \leq \frac{\log_2 p}{\beta \log_2 r} \leq 2\frac{\log_2 p}{\log_2 p'}\frac{\log_2 p'}{2\beta \log_2 r} \leq 2\frac{\log_2 p}{\log_2 p'}(1 + \frac{3\log_2 \lambda'}{\lambda' - 1}).$$

□

5.3. **Multiplication in $\mathbb{Z}$ using multiplication in $\mathcal{R}$.** We can build on Algorithms 4 and 5 to obtain an integer multiplication algorithm.

Note however that we avoid the following simple approach because it does not work complexity-wise: we do not multiply integers a and b by considering them as elements of $\mathbb{Z}/p\mathbb{Z}$ for p an admissible generalized Fermat prime whose bit length exceeds that of ab. The reason for that is that unless we consider that p is given beforehand, computing it is likely to be more expensive than computing a product of bit size $\log_2 p$, and would therefore appear dominant, maybe prohibitive even for a precomputation. Moreover, as Algorithm 5 (MulR) uses base conversion algorithms that are slightly more expensive than multiplication, it cannot be used at the top level.

Instead, we perform the top level computation with Algorithm 3 (FurerComplexMul). Multiplication in the ring $\mathbb{C}[X]/X^{2^\lambda}+1$ that is used by Algorithm 3 reduces to multiplication of integers of bit size $n_0 = O((\log n)^2)$, with n denoting the bit size of the product ab (see Equation (3.1)). These integers are represented as polynomials of $\mathcal{R}[X]/(X^N+1)$, and multiplied with Algorithm 4 (MulRX).

The key question is that of the choice of parameters $\mathcal{R}$ and N for Algorithm 4. We wish here to imitate the setup of Algorithm 5 (MulR), as defined by Proposition 4.5. The difficulty is that we do not have the same input parameters. The following proposition provides a satisfactory answer, drawing an intentional parallel with Proposition 4.5.

Proposition 5.3. *Let $n_0 > 160$ be an integer. A generalized Fermat prime denoted* initprime(n_0) *and an integer* initbatch(n_0) *are defined as follows. Let H be the smallest integer such that $n_0 \le H2^H$. Let $\lambda_0 = \lceil \log_2^{(2)}(H2^H) \rceil$. Let $\psi(k) = 2^{k+1}H + H + 2 - k$. There exists a power of two μ such that the following conditions hold:*

(i) $0 \le \log_2 \mu < \lambda_0$,
(ii) $\psi(\log_2 \mu) \ge \lambda_0 2^{\lambda_0}$,
(iii) *Given $R_0 = 2^{\psi(\log_2 \mu)/2^{\lambda_0}}$, there exists an integer $r_0 \in [R_0, R_0(1+2{\lambda_0}^2)[$ such that $p_0 = P(r_0, \lambda_0) = {r_0}^{2^{\lambda_0}} + 1$ is a generalized Fermat prime.*

Given μ and p_0 as above, we let initprime$(n_0) = p_0$ *and* initbatch$(n_0) = \mu$.

Proof. Unsurprisingly, the proof is done exactly along the lines of the proof of Proposition 4.5. The condition $n_0 > 160$ implies $H \ge 6$. Along with the obvious $\mu \leftrightarrow \beta$ and $\psi \leftrightarrow \phi$ translations, H plays the role of both λ and $\log_2 r$. We could informally write these as λ_{-1} and $\log_2 r_{-1}$, and $2^{H2^H}+1$ would play the role of p (although this p has little chance to be prime). The only difference is that ψ is shifted compared to ϕ. This does not hinder the proof. To prove $\psi(0) \le 2\lambda_0 2^{\lambda_0}$, notice that $H \ge 6$, so that

$$\psi(0) = 3H + 2 \le 4H \le 2 \cdot \log_2 H \cdot H \le 2 \log_2^{(2)}(H2^H) \log_2(H2^H) \le 2\lambda_0 2^{\lambda_0}.$$

Beyond that, this correspondence suffices to base our claims, as no argument in the proof of Proposition 4.5 uses the fact that p is assumed to be a prime number (only the "generalized Fermat" form matters). □

The most important intermediary results that we must adapt are Lemma 4.7 and Lemma 5.2.

Lemma 5.4. *Let n_0 and H be as in Proposition 5.3. Let $\mu =$* initbatch(n_0) *and $p_0 =$* initprime(n_0). *We have*

$$\frac{n_0}{\mu H} = \frac{n_0}{\log_2 p_0} \cdot (2 + o(1)) \quad \textit{and} \quad \log_2^{(2)} p_0 = O(\lambda_0).$$

Proof. Consider the quotient $\frac{\log_2 p_0}{2\mu H}$ as in the proof of Lemma 4.7. We have

$$2\mu H + 2^{\lambda_0} \ge 2\mu H + H + \log_2 H \ge \psi(\log_2 \mu) \ge \lambda_0 2^{\lambda_0},$$

so that $2\mu H \ge (\lambda_0 - 1)2^{\lambda_0}$. In the other direction, we have $\log_2 p_0 \le 2\mu H + 2^{\lambda_0} \cdot 3\log_2 \lambda_0$ (exactly as in Lemma 4.7). Together, these bounds show that $\left| \frac{\log_2 p_0}{2\mu H} - 1 \right| \le \frac{3\log_2 \lambda_0}{\lambda_0 - 1} \le 2$. The first bound is a direct consequence (see Lemma 5.2). The bound on $\log_2^{(2)} p_0$ is obtained with similar arguments (see Lemma 4.7). □

Algorithm 6 (MulZ) uses the choices detailed above. The following lemma proves the correctness of the algorithm.

Lemma 5.5. *In Algorithm 6, we have the following for $n_0 \ge 12$:*

Algorithm 6 Multiplication of integers in $\mathbb{Z}$ below the first recursive level

Input: a and b two n_0-bit integers.
Output: $a \cdot b$
Remark: To be used as a sub-algorithm below Algorithm 3 (FurerComplexMul).
function MulZ(a,b)
    Let H be the smallest integer such that $n_0 \leq H2^H$.
    Let $p_0 = P(r_0, \lambda_0) = \text{initprime}(n_0)$ and $\mu = \text{initbatch}(n_0)$.
    Let $\mathcal{R}_0 = \mathbb{Z}/p_0\mathbb{Z}$ and $\eta_0 = 2^{\mu H}$.
    Let $N_0 =$ be the smallest power of two above $2\lceil n_0/(\mu H)\rceil$.
    Let $\tilde{A}(X) \in \mathbb{Z}[X]$ be such that $\tilde{A}(\eta_0) = a$; define $\tilde{B}(X)$ likewise.
    Map $\tilde{A}, \tilde{B}$ to $A, B \in \mathcal{R}_0[X]/(X^{N_0}+1)$.
    $C \leftarrow A \cdot B \mod X^{N_0}+1$ using Algorithm 4 (MulRX).
    Lift C to $\tilde{C} \in \mathbb{Z}[X]$.
    return $c = \tilde{C}(\eta_0) = c$.
end function

(*i*) $\deg(\tilde{A} \cdot \tilde{B}) < N_0$.
(*ii*) *$\mathcal{R}_0$ has a principal $2N_0$-th root of unity;*
(*iii*) *Coefficients of $C = A \cdot B$ do not wrap around in $\mathcal{R}_0$.*

Proof. Fact (*i*) follows trivially from that $\tilde{A}$ and $\tilde{B}$ both have $\lceil n_0/(\mu H)\rceil$ coefficients. To prove (*ii*), following the idea in the proof of (*i*) in Lemma 5.1, we need to prove $\log_2(2N_0) \leq 2^{\lambda_0}$. We can bound $\log_2 N_0$ by $\lceil \log_2(2n_0/(\mu H))\rceil$ (the outer integer part suffices), so that

$$\log_2 N_0 \leq 1 + \log_2(2n_0/(\mu H)) = 2 + \log_2(n_0/(\mu H)).$$
$$\log_2 2N_0 \leq 3 + \log_2(H2^H/(\mu H)) \leq 3 + H - \log_2 \mu.$$

Furthermore, we have $2^{\lambda_0} \geq H + \log_2 H$, but even for $H = 6$ or $H = 7$ we can check that $\lambda_0 \geq 4$, so that the inequality still holds.

To prove (*iii*), we need $N_0\eta_0^2 \leq p_0$. We have

$$\log_2(N_0\eta_0^2) \leq 2 + \log_2(H2^H/(\mu H)) + 2\mu H = 2\mu H + H + 2 - \log_2 \mu = \psi(\log_2 \mu).$$

We have also $p_0 \geq {R_0}^{2^{\lambda_0}} \geq 2^{\psi(\log_2 \mu)}$, which concludes the proof. We note that the above upper bound $\log_2(N_0\eta_0^2) \leq \psi(\log_2 \mu)$ follows from the very definition of ψ, and the adjustment made here in comparison to the function ϕ is essential. □

The prime p_0 used in Algorithm 6 is also easily seen to satisfy $p_0 = 2^{O(\lambda_0 2^{\lambda_0})} = 2^{O((\log n_0)^2)}$. Using $n_0 = O(\log n)^2$, we have in particular $p_0 = o(n)$. As a consequence, the choice of the prime p_0 in Algorithm 6, below the first recursion level, successfully sidesteps the issue of finding generalized Fermat primes of appropriate size. With $p_0 = o(n)$, even a straightforward Eratosthenes sieve is cheap enough to find the prime p_0 (the complexity would be $o(n \log^{(2)} n)$).

Algorithm 6 (MulZ) allows us to express a modified complexity for the multiplication of n-bit integers, where the level of recursion below Algorithm 3 (FurerComplexMul) is done with Algorithm 6. We let $\mathsf{M}_{\text{new}}(n)$ denote this complexity. We have the following, using several notations from §5.3:

$$\mathsf{M}_{\text{new}}(n) = N(3\lceil \log_{2^{\lambda+1}} N\rceil + 1) \cdot \mathsf{M}_{\mathsf{MulZ},\mathcal{R}_0} + O(N \log N \cdot 2^{\lambda} \log n);$$
$$\mathsf{M}_{\mathsf{MulZ},\mathcal{R}_0} = \mathsf{M}_{\mathcal{R}_0[X],N_0} + O(n_0).$$

6. Solution of the recursive complexity equations

6.1. **Summary of the recursive complexity equations.** We now combine the recursive complexity equations of Section 5 into an overall complexity estimate. Two aspects are of particular importance here. First, the recursive nature of the complexity stems from the fact that Algorithm 5 (MulR) calls Algorithm 4 (MulRX), which in turn calls Algorithm 5. For this, we change notations slightly. Within the complexity expression of multiplication in the ring $\mathcal{R}_i$ at some recursive level i (starting from $i = 0$ as in Algorithm 6), we use the subscript $_{i+1}$ to denote all quantities that pertain to the next recursive level (denoted p', $\mathcal{R}'$, N' in §5.2).

The second important aspect, mentioned in §3.3, is precomputations. In accordance with previous notations, $\mathsf{M}'_{\mathcal{R}_i[X],N_i}$ and $\mathsf{M}'_{\mathcal{R}_i}$ respectively denote the complexities of algorithms 4 and 5 at recursive level i, with precomputations (that is, assuming transforms for one of the operand are computed for each bilinear operation that occurs). In the expression below, $\mathcal{P}(n)$ denotes the overall cost of precomputations. We obtain the following equations:

$$\begin{aligned}
\mathsf{M}_{\text{new}}(n) &= N(3\lceil \log_{2^{\lambda+1}} N\rceil + 1)\cdot \mathsf{M}_{\mathsf{MulZ},\mathcal{R}_0} + O(N\log N\cdot 2^{\lambda}\log n) + \mathcal{P}(n);\\
\mathsf{M}_{\mathsf{MulZ},\mathcal{R}_0} &= \mathsf{M}_{\mathcal{R}_0[X],N_0} + O(n_0) \le (3/2)\mathsf{M}'_{\mathcal{R}_0[X],N_0} + O(n_0);\\
\mathsf{M}'_{\mathcal{R}_i[X],N_i} &= N_i(2\lceil \log_{2^{\lambda_i+1}} N_i\rceil)\cdot \mathsf{M}'_{\mathcal{R}_i} + O(N_i\log N_i\cdot \log p_i);\\
\mathsf{M}'_{\mathcal{R}_i} &= N_{i+1}(2\lceil \log_{2^{\lambda_{i+1}+1}} N_{i+1}\rceil)\mathsf{M}'_{\mathcal{R}_{i+1}} + O(N_{i+1}\lambda_{i+1}\mathsf{M}(\log p_{i+1})) + \cdots\\
&\qquad \cdots + O(N_{i+1}\cdot \log N_{i+1}\cdot \log p_{i+1}).
\end{aligned}$$

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

Proposition 6.1. *We keep the above notations. Let $i \ge 0$. Let $\epsilon_{0,i} = \frac{3\log_2 \lambda_{i+1}}{\lambda_{i+1}}$, $\epsilon_{1,i} = \frac{2\log_2 \lambda_i}{\lambda_i}$, 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*

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

Proof. We first bound the second and third term in the equation for $\mathsf{M}'_{\mathcal{R}_i}$, and compare them to $\log p_i\cdot \log^{(2)} p_i$. The third term uses Lemma 5.2. We have

$$\frac{N_{i+1}\log_2 N_{i+1}\log_2 p_{i+1}}{\log_2 p_i \log_2^{(2)} p_i} \le 6\frac{\log_2 N_{i+1}}{\log_2^{(2)} p_{i+1}} = O(1).$$

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

$$\frac{N_{i+1}\lambda_{i+1}\mathsf{M}(\log_2 p_{i+1})}{\log_2 p_i \log_2^{(2)} p_i} \le 6\frac{\lambda_{i+1}\log_2^{(2)} p_{i+1}\log_2^{(3)} p_{i+1}}{\log_2^{(2)} p_i} = O(1).$$

In the expression above, we obtain the upper bound by bounding the numerator by a polynomial in λ_{i+1}.

The most important calculation for the analysis is the comparison of the first term of $\mathsf{M}'_{\mathcal{R}_i}$ with $\log p_i\cdot \log^{(2)} p_i$. Using Lemma 5.2, and also the coarse bound

$\log_2 N_{i+1} = \log_2(2^{\lambda_i}/\beta_i) \le \lambda_i$, we have

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

where the last two lines rearrange some terms, and use the upper bound on λ' from Lemma 4.6 as well as statement (*ii*) from Lemma 4.7. This corresponds to 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 will need the following stronger result, however.

Lemma 6.2. *Let λ_0 be an arbitrarily large integer. Let K be the first integer such that $\lambda_K \le 6$. We have $K = \log^\star \lambda_0 + O(1)$. Furthermore, for $j = 0, 1, 2$:*

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

Proof. The expression of K follows from Lemma 4.6: it suffices to observe that $\lambda \ge \sqrt[3]{2}^{\lambda'}$.

To bound the product, it suffices to bound $\sum_i |\epsilon_{j,i}|$. Let $f_0(x) = \frac{3\log_2 x}{x}$, $f_1(x) = \frac{2\log_2 x}{x}$, 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 $x \ge 4$. Consider the sequence of real numbers defined by $u_0 = 4$, and $u_{k+1} = 2^{u_k/3}$. Independently of the starting value λ_0, we have

$$\begin{aligned} \lambda_K &\ge 4 = u_0, \\ \lambda_{K-1} &\ge 2^{\lambda_K/3} \ge u_1 \quad \text{by Lemma 4.6,} \\ \lambda_{K-k} &\ge u_k \quad \text{for all } k \le K. \end{aligned}$$

This yields $\sum_i |\epsilon_{j,i}| \le \sum_{k=0}^{\infty}\sum_{j=0}^{2} f_j(u_k)$. The latter sum converges. □

6.2. **Precomputations are negligible.** Before we state the final complexity, we examine the cost of precomputations.

- Given an input of bit size n, compute the sequence of generalized Fermat primes that will be used by Algorithm 5 (MulR).
- Compute multiplicative generators in $\mathcal{R}_i$.
- Compute principal roots of unity in the rings $\mathcal{R}_i$.
- Compute the DFT of the roots of unity.

We already discussed in §5.3 how the strategy of doing a preliminary step with Algorithm 3 (FurerComplexMul) was an effective way to eliminate the cost of finding appropriate generalized Fermat primes. In fact, everything that relates to the rings $\mathcal{R}_i$ is cheap to precompute. The reason for this is the same argument as the one we mentioned in §5.3: we have $p_0 = o(n)$, so even naive algorithms are cheap enough. More precisely, we have $\mathcal{P}(n) = o(n \log n)$.

6.3. **Complexity of integer multiplication.**

Theorem 6.3. *The complexity* $\mathsf{M}_{new}(n)$ *of the algorithm presented in §5.3 to multiply* n*-bit integers is*

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

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 $\lambda \sim \log_2^{(2)} n$. The input bit size of Algorithm 6 (MulZ) is $n_0 = \Theta((\log_2 n)^2)$. We have

$$O(N \log N \cdot 2^\lambda \log n) = O((n/(\log_2 n)^2)(\log_2 n)^3) = O(n \log n).$$

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

We thus have

$$\frac{\mathsf{M}_{\text{new}}(n)}{n \log n} = O(1) + O\left(\frac{\mathsf{M}_{\mathsf{MulZ},\mathcal{R}_0}}{n_0(\log_2 n_0)}\right) = O(1) + O\left(\frac{\mathsf{M}'_{\mathcal{R}_0[X],N_0}}{n_0(\log_2 n_0)}\right).$$

We study the last term of the expression above. In Algorithm 6, the length N_0 is $O(n_0/\log_2 n_0)$. Using Lemma 5.4, we have

$$\begin{aligned} O(N_0 \log N_0 \cdot \log p_0) &= O(n_0/\log_2 p_0) \cdot O(\log_2 n_0) \cdot O(\log_2 p_0) \\ &= O(n_0(\log n_0)). \\ \frac{\mathsf{M}'_{\mathcal{R}_0[X],N_0}}{n_0 \log_2 n_0} &= O\left(\frac{n_0/\log_2 p_0}{n_0 \log_2 n_0}\right) \cdot O\left(\frac{\log_2 n_0}{\lambda_0 + 1}\right) \cdot \mathsf{M}'_{\mathcal{R}_0} + O(1) \\ &= O\left(\frac{\mathsf{M}'_{\mathcal{R}_0}}{\log_2 p_0 \log_2^{(2)} p_0}\right) + O(1). \end{aligned}$$

Using now Proposition 6.1 and Lemma 6.2, we can conclude: there exists a constant $C > 0$ such that $(m_i + C) \le 4 \cdot \prod_{0 \le j \le 2}(1 + \epsilon_{j,i}) \cdot (m_{i+1} + C)$, so that we get $m_0 = O(4^K)$. Finally, this gives

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

□

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.

In practice, an implementation of our algorithm does not have to follow the various tricks that make the complexity analysis work. First, the strategy of using Algorithm 3 (FurerComplexMul) for the top level of the tree of recursive calls is not needed if one considers the determination of the first prime p_0 as a precomputation. In light of Table 3, we may consider such primes as given in the context of an implementation. Moreover, the $3N$ additional multiplications due to Half-FFT can be avoided at the top level, for the same reason explained in [GKZ07, §2.3], applied to the Schönhage-Strassen algorithm.

bit size range	p
$2^{16} \leq n < 2^{32}$	$74^{16}+1$
$2^{32} \leq n < 2^{64}$	$884^{32}+1$
$2^{64} \leq n < 2^{128}$	$1084^{64}+1$
$2^{128} \leq n < 2^{256}$	$1738^{128}+1$
$2^{256} \leq n < 2^{512}$	$1348^{256}+1$

TABLE 3. Examples of primes that can be used in the algorithm

Second, the highest priority criterion one should use to choose the primes is not necessarily the one given by Proposition 4.5. For a given parameter η fixing the size of coefficients of polynomials A and B of degree N, the generalized Fermat prime p should simply not be too large compared to $N\eta^2$. Thus, in practice, we choose the smallest prime $p = r^{2^\lambda} + 1$ larger than $N\eta^2$.

For instance, for $n = 2^{31}$, if $\log_2 \eta = 2^5$, then $2\log_2 \eta + \log_2 N = 2^6 + 26 \leq \log_2(74^{16}+1)$. The prime $p = 74^{16}+1$ is the smallest generalized Fermat prime with exponent 2^4 and a bit size above 90. Let us express the cost C of the point-wise product:

$$C = \frac{n}{\log_2 \eta} S(\log_2 p) \approx \frac{n}{\log_2 \eta} S(2\log_2 \eta + \log_2 N).$$

The term S denotes the complexity of Schönhage-Strassen's algorithm. It is clear from this equation that if $\log_2 N \approx \log_2 \eta$, then

$$C \approx 3n \cdot \log_2^{(2)} \eta \log_2^{(3)} \eta$$

whereas if $\log_2 N \ll \log_2 \eta$, then

$$C \approx 2n \cdot \log_2^{(2)} \eta \log_2^{(3)} \eta.$$

In conclusion, the previous analysis shows that, in the context of a practical implementation, we should choose $\log_2 \eta$ such that $\log_2 N \ll \log_2 \eta$ and p such that p is the smallest prime larger than $N\eta^2$.

Moreover, it is possible to use an intermediate strategy between the Kronecker substitution and the strategy corresponding to switching between radix r and radix r^β. For instance, multiplying elements represented in radix 74 in the ring $\mathcal{R} = \mathbb{Z}/p\mathbb{Z} = \mathbb{Z}/(74^{16}+1)\mathbb{Z}$ using Kronecker substitution would require a multiplication of two integers of bit size greater than $2 \cdot 8 \cdot 16 = 256$. This requires, on a 64-bit architecture, 9 machine-word multiplications using Karatsuba on two levels of recursion.

Considering elements of $\mathcal{R}$ in radix 74^2, we can use the multipoint Kronecker substitution proposed by Harvey in [Har09]. This way, we perform 2 multiplications of 128-bit integers. This needs 6 machine-word multiplications, to which we should add the cost of the recomposition in radix 74^2 and the decomposition in radix 74.

Let us compare approximatively the cost of Schönhage-Strassen's algorithm to our algorithm. Roughly speaking, for 2^{30}-bit integers, a Schönhage-Strassen run would involve $2 \cdot 2^{15}$ multiplications of integers of approximate size 2^{16}. In our case, we cut this 2^{30}-bit integer into pieces of size less than 2^5-bit. Thus, $N = 2 \cdot 2^{25}$ (the size of the product is twice the size of the input) and we have

$$N \cdot (3\lceil \log_{2\cdot 16} N \rceil + 1) = 2^{26} \cdot 19$$

multiplications of 288-bit integers using Kronecker substitution if $p = 74^{16}+1$. Thus, we need to know if there is a chance that the cost induced by $2^{26} \cdot 19$ multiplications of 288-bit integers is cheaper than the cost induced by 2^{16} multiplications of 2^{16}-bit integers.

bit size 2^{30}			
prime	expensive multiplications	bit size of K.S.	estimated time (s)
2097208^8+1	$2^{25}\cdot 22$	376	$5.62\cdot 10$
$74^{16}+1$	$2^{26}\cdot 19$	288	$8.61\cdot 10$
$54^{32}+1$	$2^{25}\cdot(3\cdot\lceil\frac{25}{6}\rceil+1)=2^{25}\cdot 16$	$(6\cdot 2+5)\cdot 32=544$	$6.12\cdot 10$
$\mathbf{562^{32}+1}$	$\mathbf{2^{24}\cdot 13}$	$\mathbf{(10\cdot 2+5)\cdot 32=800}$	$\mathbf{3.57\cdot 10}$
$131090^{32}+1$	$2^{23}\cdot 13$	$(18\cdot 2+5)\cdot 32=1312$	$4.82\cdot 10$
bit size 2^{36}			
prime	expensive multiplications	bit size of K.S.	estimated time (s)
2097208^8+1	$2^{31}\cdot 25$	376	$4.08\cdot 10^3$
$2072^{16}+1$	$2^{31}\cdot 22$	448	$4.26\cdot 10^3$
$54^{32}+1$	$2^{31}\cdot 19$	$(6\cdot 2+5)\cdot 32=544$	$4.61\cdot 10^3$
$\mathbf{562^{32}+1}$	$\mathbf{2^{30}\cdot 16}$	$\mathbf{(10\cdot 2+5)\cdot 32=800}$	$\mathbf{3.35\cdot 10^3}$
$131090^{32}+1$	$2^{29}\cdot 16$	$(18\cdot 2+5)\cdot 32=1312$	$3.75\cdot 10^3$
$102^{64}+1$	$2^{30}\cdot 16$	$(7\cdot 2+6)\cdot 64=1280$	$7.00\cdot 10^3$
$562^{64}+1$	$2^{29}\cdot 16$	$(10\cdot 2+6)\cdot 64=1664$	$5.65\cdot 10^3$
bit size 2^{40}			
prime	expensive multiplications	bit size of K.S.	estimated time (s)
2097208^8+1	$2^{35}\cdot 28$	376	$7.32\cdot 10^4$
$2072^{16}+1$	$2^{35}\cdot 22$	448	$6.82\cdot 10^4$
$54^{32}+1$	$2^{35}\cdot 19$	$(6\cdot 2+5)\cdot 32=544$	$7.52\cdot 10^4$
$\mathbf{562^{32}+1}$	$\mathbf{2^{34}\cdot 19}$	$\mathbf{(10\cdot 2+5)\cdot 32=800}$	$\mathbf{6.26\cdot 10^4}$
$131090^{32}+1$	$2^{33}\cdot 19$	$(18\cdot 2+5)\cdot 32=1312$	$7.09\cdot 10^4$
$102^{64}+1$	$2^{34}\cdot 16$	$(7\cdot 2+6)\cdot 64=1280$	$11.28\cdot 10^4$
$562^{64}+1$	$2^{33}\cdot 16$	$(10\cdot 2+6)\cdot 64=1664$	$9.03\cdot 10^4$
bit size 2^{46}			
prime	expensive multiplications	bit size of K.S.	estimated time (s)
$2072^{16}+1$	$2^{41}\cdot 28$	448	$5.55\cdot 10^6$
$54^{32}+1$	$2^{41}\cdot 22$	$(6\cdot 2+5)\cdot 32=544$	$5.47\cdot 10^6$
$\mathbf{884^{32}+1}$	$\mathbf{2^{40}\cdot 22}$	$\mathbf{(10\cdot 2+5)\cdot 32=800}$	$\mathbf{4.64\cdot 10^6}$
$131090^{32}+1$	$2^{39}\cdot 22$	$(18\cdot 2+5)\cdot 32=1312$	$5.25\cdot 10^6$
$562^{64}+1$	$2^{39}\cdot 19$	$(10\cdot 2+6)\cdot 64=1664$	$7.68\cdot 10^6$

TABLE 4. Estimated time for computing the expensive multiplications in our algorithm depending on the prime used.

For 2^{40}-bit integers, we use $p=884^{32}+1$. Thus, we cut our integers in pieces of size 2^7 and $N=2\cdot 2^{33}$. Thus, we multiply $2^{34}(3\cdot\lceil\frac{34}{6}\rceil+1)\approx 19\cdot 2^{34}$ integers of 800-bit using Kronecker substitution. The Schönhage-Strassen involves 2^{21} multiplications of approximately 2^{21}-bit integers.

We investigate in Table 4 how changing the prime used in our algorithm for the multiplication of two n-bit integers, where n takes the values 2^{30}, 2^{36}, 2^{40} or 2^{46}, may impact the estimated time spent to compute expensive multiplications. We computed this estimated time by measuring the average time spent by the routine `mpz_mul` of GMP [Gt16] for different bit sizes on an Intel Core i5-4590 CPU clocked at 3.30GHz. We obtain the expected time spent in expensive multiplications by estimating the bit size of the integers that we get with Kronecker substitution and we multiply the number of expensive multiplications by the average time of `mpz_mul` for this bit size. This approximation does not take into account the fact that expensive multiplications of integers of bit size m are done modulo an integer 2^m+1, which may save a factor 2 in practice. We deduce from Table 4 that changing the prime may improve on the cost induced by the expensive multiplications, but not significantly.

	Schönhage-Strassen algorithm			Section 5			
bit size	nb. mult.	mult. bit size	time (s)	nb. mult.	prime	KS. bit size	time (s)
2^{30}	2^{16}	$\approx 2^{16}$	9.96	$2^{24} \cdot 13$	$562^{32}+1$	800	$3.57 \cdot 10$
2^{36}	2^{18}	$\approx 2^{18}$	$2.60 \cdot 10^2$	$2^{30} \cdot 16$	$562^{32}+1$	800	$3.35 \cdot 10^3$
2^{40}	2^{21}	$\approx 2^{21}$	$2.36 \cdot 10^4$	$2^{34} \cdot 19$	$562^{32}+1$	800	$6.26 \cdot 10^4$
2^{46}	2^{24}	$\approx 2^{24}$	$2.17 \cdot 10^6$	$2^{40} \cdot 22$	$884^{32}+1$	800	$4.64 \cdot 10^6$
2^{50}	2^{26}	$\approx 2^{26}$	$4.10 \cdot 10^7$	$2^{44} \cdot 25$	$884^{32}+1$	800	$7.91 \cdot 10^7$
2^{56}	2^{29}	$\approx 2^{29}$	$2.94 \cdot 10^9$	$2^{50} \cdot 28$	$884^{32}+1$	800	$5.67 \cdot 10^9$

TABLE 5. Comparison of multiplications within the Schönhage-Strassen algorithm and the algorithm of Section 5, relying on measured timings of `mpz_mul` of GMP [Gt16]. The third column "KS. bit size" is the bit size of the integers obtained after Kronecker substitution.

Table 5 gives an estimation of the time required to compute a multiplication of two integers using our algorithm compared to Schönhage-Strassen. This estimation does not take into account the cost of linear operations such as additions, subtractions, shifts, which may be not negligible in practice. We use the best trade-off obtained in Table 4 and the primes proposed in Table 3 by default.

Thus, Table 5 allows one to conclude that an implementation of our algorithm will unlikely beat an implementation of Schönhage-Strassen algorithm for sizes below 2^{40}. Above 2^{40}, it seems that Schönhage-Strassen algorithm is no longer unreachable. Thus, provided that there is an improvement on Kronecker substitution allowing one to spare a factor 2 in the estimation of the cost of the expensive multiplications, it does not seem hopeless to have a competitive implementation.

8. Discussion on the bounds used

The bound on r in Definition 4.1 is sharp, since it corresponds exactly to what is needed for the proof of Proposition 4.5.

The bound in Hypothesis 4.3 uses a somewhat arbitrary factor $1+2\lambda^2$. That factor could be replaced by any polynomial in λ, this would not affect the fact that the bounds used in Proposition 6.1 have a numerator in $O(\log_2 \lambda)$. We note, however, that using a polynomial with a more rapid growth would invalidate some inequalities for small values of λ, so that we would only be able to state our algorithm for larger values of λ.

9. 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 hypothesis 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 $4^{\log^\star n}$ 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 $\log^\star n + O(1)$.

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 $\frac{1}{2}\log^\star n + O(1)$ would require a reduction from n to $\log^{(2)} n$ at each step.

References

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

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

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

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

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

[DG02] Harvey Dubner and Yves Gallot, *Distribution of generalized Fermat prime numbers*, Math. Comput. **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.

[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*, Journal of Symbolic Computation **44** (2009), no. 10, 1502 – 1510.

[HvdH16] David Harvey and Joris van der Hoeven, *Faster integer multiplication using plain vanilla FFT primes*, 2016.

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

[KO63] Anatolii 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.

[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 University Press, New York, NY, USA, 1999.

Appendix A. Notes on various inequalities

The choices of bounds in the article, and in particular the upper bounds in Definition 4.1 and Hypothesis 4.3, might seem arbitrary. Both are mostly related to Proposition 4.5. We discuss here in further detail the reason why these upper bounds were chosen. Furthermore, several inequalities are considered in the text, and justified with very terse arguments. We give more detail here. However, instead of complete calculus-based proofs, we choose to provide numerical verification code.

Range for r in the notion of admissible primes. Definition 4.1 calls for the verification that the allowed range for r is not empty. Let $L(\lambda) = (\lambda + \log_2 \lambda)\log_2(\lambda + \log_2 \lambda) - \lambda/2$. We prove more than non-emptiness of the range for r (which is equivalent to $L(\lambda) \geq \lambda$). We have:

$$\begin{aligned} &\forall \lambda \geq 2, \quad L(\lambda) \geq \lambda, \\ &\forall \lambda \geq 3, \quad L(\lambda) \geq 2\lambda, \\ &\forall \lambda \geq 4, \quad L(\lambda) \geq 2\lambda + \log_2(1 + 2\lambda^2). \end{aligned}$$

Inequalities above can be checked numerically with the following code.

gnuplot code

```
l2(x)=log(x)/log(2)
L(x)=(x+l2(x))*l2(x+l2(x))-x/2
s(x)=1+2*x**2
plot [1:] L(x),x,2*x,2*x+l2(s(x))
```

Upper bound in Hypothesis 4.3: when is smallerprime(p) *admissible ?* The definition of smallerprime(p) uses the same upper bound for the choice of r' as the one that appears in Hypothesis 4.3. Proposition 4.5 shows that $\lambda \geq 6$ suffices to guarantee that $\lambda' \geq 4$, and that $p' = \text{smallerprime}(p)$ is admissible.

This lower bound $\lambda' \geq 4$ is essentially tight. We briefly discuss what would be the implications of having a replacement $s(\lambda)$ playing the role of $1 + 2\lambda^2$, with the goal of making $p' = \text{smallerprime}(p)$ admissible even in the case $\lambda' = 3$. On the one hand, we would need $s(3) \geq 15$, because if $R' = 8$, the smallest $r' \geq R'$ such that $P(r', \lambda')$ is prime is $r' = 118 = 14.75 \times 8$. On the other hand, if $R' = 64$, the admissibility condition of Definition 4.1 would require $s(3) \leq 2^{L(3)}/64 < 6$.

Proposition 4.5 could conceivably be stated for $\lambda \geq 3$ only, with $p' = \text{smallerprime}(p)$ admissible only when $\lambda' \geq 4$. This would not make the result more useful, since Algorithm 5 (MulR) is very likely not competitive modulo primes so small.

Bounds in Lemmas 4.6 and 4.7. In Lemma 4.6, we write

$$\lambda' \leq 1 + \log_2^{(3)} p \leq 1 + \log_2^{(3)}(2r^{2^\lambda}) \leq 1 + \log_2^{(2)}(1 + 2^\lambda L(\lambda))$$

using the notation $L(\lambda)$ that was introduced earlier in this appendix. Lemma 4.6 claims that the right-hand side does not exceed $3\log_2 \lambda - 1$ for $\lambda \geq 4$. Lemma 4.7 includes two similar claims, namely

$$\begin{aligned} &\forall \lambda' \geq 4, && (2^{\lambda'} - 2) + 1 + 2^{\lambda'} \log_2(1 + 2\lambda'^2) && \leq 2^{\lambda'} \cdot 3\log_2 \lambda'. \\ &\forall \lambda' \geq 3, && \log_2\left(1 + 2^{\lambda'}(2\lambda' + \log_2(1 + 2\lambda'^2))\right) && \leq \lambda' + \log_2 \lambda' + 2. \end{aligned}$$

We can check all these claims numerically as follows:

gnuplot code

```
plot [1:] 1+l2(l2(1+2**x*L(x))), 3*l2(x)-1
plot [1:] l2(s(x))+1-1/2**x,3*l2(x)
plot [1:] l2(1+2**x*(2*x+l2(s(x)))), x+l2(x)+2
```

λ	c	λ	c	λ	c	λ	c
1	1.3728	6	3.9427	11	7.2250	16	11.1918
2	2.6789	7	3.1089	12	8.4263	17	10.9965
3	2.0928	8	7.4348	13	8.4693	18	13.0292
4	3.6714	9	7.4889	14	8.0100	19	13.0857
5	3.6130	10	8.0192	15	5.7965	20	14.4516

TABLE 6. Approximations of the infinite product $c = 2^\lambda C$, as defined by Equation (B.1) (product over primes below 10^9).

APPENDIX B. EMPIRICAL VALIDATION OF HYPOTHESIS 4.3 IN TABLE 1

While discussing Lemma 4.2, we introduced the quantities $E(R, \lambda)$ and C. The latter constant, which depends on λ, is actually $C = c/2^\lambda$, with c the following infinite product defined by [BH62].

$$c = \frac{1}{2} \prod_{p \text{ prime}} \frac{1 - \chi_\lambda(p)/p}{1 - 1/p} \text{ where } \chi_\lambda(p) = \begin{cases} 2^\lambda & \text{if } 2^{\lambda+1} \mid p-1, \\ 0 & \text{otherwise.} \end{cases} \tag{B.1}$$

The infinite product above can be computed easily (see also [DG02, §3] for the same computation). We get the approximations reported in Table 6. From there, to estimate the number of generalized Fermat primes with $R_0 \le r < R_1$, we compute $\frac{c}{2^\lambda}(\mathrm{li}(R_1) - \mathrm{li}(R_0))$ where li is the logarithm integral function $\int^x \frac{\mathrm{d}t}{\log t}$. This is done by the following `Magma` code, which one can use to reproduce the first rows of Table 1. Data for larger values of λ in Table 1 was obtained with an improved prime search in C++, parallelized with MPI.

`Magma` code

```
c_constant:=[ 1.372810, 2.678946, 2.092785, 3.671417, 3.612985, 3.942738,
              3.108904, 7.434776, 7.488888, 8.019167, 7.224961, 8.426297,
              8.469321, 8.010046, 5.796499, 11.191794, 10.996544, 13.029243,
              13.085653, 14.451572 ];
for lambda in [1..maxlambda] do
    R0:=2^lambda; // or any choice within [2^lambda..2^(2*lambda)]
    R1:=R0*(1+2*lambda^2);
    est:=c_constant[lambda]/2^lambda*(LogIntegral(R1)-LogIntegral(R0));
    act:=#[r:r in [R0..R1-1] | IsProbablePrime(r^(2^lambda)+1)];
    print lambda, act, Round(est);
end for;
```

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

UNIVERSITÉ DE LORRAINE, CNRS, INRIA, LORIA, F-54000 NANCY, FRANCE