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 $K = 4$. We obtain here the same result $K = 4$ using simple modular arithmetic as a building block, and a careful complexity analysis. We obtain a similar result $K = 4$ using an alternative somewhat simpler algorithm, which relies on arithmetic modulo generalized Fermat primes.

1. Introduction

Beyond the schoolbook algorithm, the first nontrivial algorithm improving the complexity for multiplying n-bit integers is Karatsuba's algorithm in the 1960's [17], which reaches the complexity $O(n^{\log_2 3})$, using a divide-and-conquer method. The Karatsuba algorithm can be viewed as a simple case of a more general evaluation-interpolation paradigm. Indeed, it consists in the evaluation of two polynomials in 0, 1 and ∞, followed by a component-wise multiplication, and an interpolation phase involving a multiplication by a 3×3 matrix. By generalizing this evaluation-interpolation approach, it is possible to improve the complexity to $O(n^{1+\epsilon})$ for any $\epsilon > 0$. This result is known as the Toom-Cook algorithm [23]. In [21], Schönhage and Strassen reached the $O(n \cdot \log n \cdot \log\log n)$ complexity using the fast Fourier transform (FFT), which is a divide-and-conquer algorithm allowing one to quickly evaluate a polynomial at the powers of a primitive root of unity [25]. The key to achieve this complexity result is the appropriate choice of a base ring $\mathbf{R}$ in which evaluation is to be carried out, and in which recursive calls to the multiplication algorithms are also done. The most popular variant of the Schönhage-Strassen algorithm uses $\mathbf{R} = \mathbb{Z}/(2^t + 1)\mathbb{Z}$, where 2 is a primitive $2t$-th root of unity, t being chosen close to $\sqrt{n}$.

For almost 35 years, this complexity estimate remained unbeaten, until Fürer proposed in [10] a new algorithm also relying on the FFT, but using a different ring, namely $\mathbf{R} = \mathbb{C}[x]/(x^P+1)$ for P a suitably chosen power of 2. This combines the benefits of the Schönhage-Strassen algorithm with the possibility to use larger transform length, thereby allowing recursive calls on shorter data. This eventually yields complexity $O(n \cdot \log n \cdot K^{\log^\star n})$ for some constant K.

A variant was subsequently proposed by De, Kurur, Saha and Saptharishi in [6], using, roughly speaking, $\mathbf{R} = \mathbb{Q}_p[x]/(x^P + 1)$ with finite precision (which means working in the ring $\mathbb{Z}/p^\lambda\mathbb{Z}[x]/(x^P + 1)$), and with similar complexity. Be it either in Fürer's original context with complex coefficients, or in this p-adic case, some work is needed to properly estimate the required complexity. However the p-adic case makes the analysis easier.

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

In [16], Harvey, van der Hoeven and Lecerf proposed a new algorithm and a sharper complexity analysis that allows one to make the complexity more explicit, namely $O(n \cdot \log n \cdot 8^{\log^* n})$ and even $O(n \cdot \log n \cdot 4^{\log^* n})$ using a conjecture on Mersenne primes.

We will show in this article a different strategy to reach the $O(n \cdot \log n \cdot 4^{\log^* n})$ by a few variations to the original algorithm proposed by Fürer. Our approach adapts an idea from [9] relying on the (unfortunately unlikely) assumption that there exists an infinite sequence of Fermat primes. Indeed, it appears that we can use the same idea for generalized Fermat primes, which are primes of the form $r^{2^\lambda}+1$. Moreover, the assumption that we use is heuristically valid. This idea, combined with a sharp complexity analysis, permits to reach the complexity $O(n \cdot \log n \cdot 4^{\log^* n})$. Although we obtain a conditional and identical complexity to the one proposed in [16], the algorithms involved are quite distinct. Thus, we present a new algorithm and some theoretical and numerical results about generalized Fermat primes.

This article is organized as follows. Section 2 describes the essential building blocks we use for our algorithm. We describe in Section 3 Fürer's algorithms and give a sketch of Harvey, van der Hoeven and Lecerf contribution. The Section 4 gives a lower bound on the density of generalized Fermat primes defined by the Bateman-Horn conjecture [2] and numerical evidences for a hypothesis on which the main result of this paper relies. Section 5 describes the general course of our algorithm and, in particular, how to use the generalized Fermat primes in order to speed-up the multiplication of two n-bit integers. In Section 6 a sharp complexity analysis is given, leading to the announced conjectural and deterministic binary complexity. Section 7 develops some practical aspects of the algorithm described in this work.

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

2. FFT-based multiplication

2.1. **Integers to polynomials.** Let a and b be n-bit integers. We intend to compute the integer product $c = ab$. The Kronecker substitution technique associates to a and b two univariate polynomials A and B such that $a = A(\eta)$ and $b = B(\eta)$, for η a suitable power of 2. The coefficients of A and B can be read off from the base η expansion of the integers a and b, and are therefore bounded by η. These polynomials are then interpreted in some ring $\mathbf{R}$, and multiplied modulo a polynomial, so that the result can be recovered as $c = C(\eta)$, where the coefficients of C are interpreted as integers. This expression is valid when η is suitably chosen, so that no overflow happens in the computation of the polynomial product.

The core of this procedure is the modular multiplication in $\mathbf{R}$, which is done with Algorithm 1 which multiplies modulo the minimal polynomial of the set of evaluation points $\mathcal{S}$.

Algorithm 1 Polynomial Multiplication

Input: A, B two polynomials in $\mathbf{R}[x]$ whose product has degree less than N, and $\mathcal{S}$ a set of N evaluation points
Output: $C = A \cdot B$
  function PolynomialMultiply(A,B)
    $A \leftarrow \mathsf{MultiEvaluation}(A, \mathcal{S})$
    $B \leftarrow \mathsf{MultiEvaluation}(B, \mathcal{S})$
    $C \leftarrow \mathsf{PointwiseProduct}(A, B)$
    $C \leftarrow \mathsf{Interpolation}(C, \mathcal{S})$
    return C
  end function

Besides the cost of the MultiEvaluation and Interpolation routines, which will be discussed further, the cost of the PointwiseProduct step in Algorithm 1 is easily seen to be exactly N products in the ring $\mathbf{R}$.

Throughout the article, we use the notations above. Namely, the integer n denotes the bit size of the integers whose product we intend to compute. Integers are represented by polynomials evaluated at some η as above. We use an evaluation-interpolation approach, using evaluation at $N = 2^k$ points, which are successive powers of an appropriately chosen N-th root of unity in a ring $\mathbf{R}$. The bit size used for representing elements in $\mathbf{R}$ is denoted by t.

2.2. **Cooley-Tukey FFT.** Let N be a power of 2 and $\mathbf{R}$ be a ring containing an N-th principal root of unity ω, whose definition is given in definition 2.

Definition 1. *Let $\mathbf{R}$ be a ring containing an N-th root of unity ω. ω is said to be an N-th primitive root of unity if*

$$\forall i \in [\![1, N-1]\!], \omega^i \neq 1.$$

Definition 2. *Let $\mathbf{R}$ be a ring containing an N-th root of unity ω. ω is said to be an N-th principal root of unity if*

$$\forall i \in [\![1, N-1]\!], \sum_{j=0}^{N-1} \omega^{ij} = 0.$$

One can notice that a principal root of unity is necessarily a primitive root of unity, but the opposite is wrong.

Example 3. *In $\mathbb{C} \times \mathbb{C}$, the element $(1, i)$ is a 4-th primitive root of unity but not a 4-th principal root of unity.*

Definition 4 (Discrete Fourier Transform (DFT))**.** *Let $\mathbf{R}$ be a ring with ω an N-th principal root of unity.*

The N-point DFT over $\mathbf{R}$ is the isomorphism mapping an element of $\mathbf{R}[x]/(x^N - 1)$ to

$$\mathbf{R}[x]/(x-1) \times \mathbf{R}[x]/(x-\omega) \times \ldots \times \mathbf{R}[x]/(x-\omega^{N-1}).$$

Definition 5 (Half Discrete Fourier Transform (half-DFT))**.** *Let $\mathbf{R}$ be a ring with ω an $2N$-th principal root of unity. The N-point half-DFT over $\mathbf{R}$ is the isomorphism mapping an element of $\mathbf{R}[x]/(x^N + 1)$ to*

$$\mathbf{R}[x]/(x-\omega) \times \mathbf{R}[x]/(x-\omega^3) \times \ldots \times \mathbf{R}[x]/(x-\omega^{2N-1}).$$

The DFT evaluates polynomials at all the powers of ω: given $P \in \mathbf{R}[X]$, the DFT returns

$$P(1), P(\omega), P(\omega^2), \ldots, P(\omega^{N-1})$$

(equivalently, we will identify this N-uple with the polynomial having these coefficients). The set of powers of ω will play the role of the set of evaluation points $\mathcal{S}$ mentioned in Algorithm 1.

We can describe an algorithm computing the DFT of N points in $\mathbf{R}$ using a divide-and-conquer strategy. This algorithm is the Cooley-Tukey FFT [5] (Fast Fourier Transform), which corresponds to Algorithm 2.

One notices that Algorithm 2 takes as a parameter a composition rule $*$ such that:

$$T : (P \in \mathbf{R}[X], \alpha) \to P(\alpha \cdot X).$$

This map corresponds by default to a procedure computing the multiplications of the coefficients P_i of P by α^i by using for instance a recursive call or the school-book multiplication. Thus, the complexity of Algorithm 2 can be expressed recursively. Each call to Radix2FFT involves 2

Algorithm 2 Radix-2 FFT algorithm

Input: $P = \sum_{i=0}^{N-1} p_i X^i$ a polynomial in $R[X]$ of degree $N-1 = 2^k - 1$, ω an N-th root of unity, T a map computing twiddle factors
Output: $P(1) + P(\omega)X + \cdots + P(\omega^{N-1})X^{N-1}$

```
function RADIX2FFT(P,ω,N,*)
    if N = 1 then
        return P
    else
        Q_0 ← Σ_{j=0}^{N/2-1} p_{2j} X^i
        Q_1 ← Σ_{j=0}^{N/2-1} p_{2j+1} X^i
        Q_0 ← Radix2FFT(Q_0, ω^2)
        Q_1 ← Radix2FFT(Q_1, ω^2)
        P ← Q_0(X) + T(Q_1, ω) + X^{N/2}(Q_0(X) − T(Q_1, ω))
        return P
    end if
end function
```

recursive calls on half-size inputs as well as $O(N)$ multiplications (the composition of Q_1 and ω) and additions in $\mathbf{R}$. We thus have

$$C(N) = 2C(N/2) + O(N)$$

from which it follows that $O(Nk) = O(N \log_2 N)$ operations in the ring $\mathbf{R}$ are required.

Given a polynomial $P(x)$ over a ring $\mathbf{R}$ containing a $2N$-th principal root of unity ω, we get $Q(x) = P(x) \mod (x^N - 1)$ by computing the FFT of P using ω^2 as an N-th root of unity. In order to compute $R(x) = P(x) \mod (x^N + 1)$, one can compute $Q'(x) = P(\omega \cdot x) \mod ((\omega \cdot x)^N - 1) = P(\omega \cdot x) \mod (x^N + 1)$ and then $R(x) = Q'(\omega^{-1} \cdot x)$, which means that the FFT can compute the N-point half-DFT with $2N$ additional multiplications due to the composition with $(\omega \cdot x)$ and $(\omega^{-1} \cdot x)$.

A more general form of the Cooley-Tukey FFT recursion exists. This starts by writing the transform length $N = 2^k$ as $N = N_1 N_2 = 2^{k_1+k_2}$, with $N_i = 2^{k_i}$. We organize the coefficients of the input polynomial P as the columns of an $N_1 \times N_2$ matrix, and then perform N_1 "row" transforms of length N_2, followed by N_2 "column" transforms of length N_1. This is Algorithm 3.

One easily sees that Algorithm 3 specializes to Algorithm 2 when $k_1 = 1$. Algorithm 3 leaves unspecified which algorithm is used for the recursive calls denoted by FFT, or more precisely nothing is prescribed regarding how transform length are to be factored as $N = N_1 N_2$ in general.

This "matrix" form of the Cooley-Tukey FFT appeared several times in literature. It is often observed that effectively doing the transposition of the polynomial coefficients, and use Algorithm 3 for balanced transform lengths N_1, N_2 leads to a better performing implementation. As we observe below, this has a stronger impact in the context of Fürer's algorithm.

2.3. **Complexity of integer multiplication.**

Notation 6. *Let $M(n)$ denote the binary complexity of the multiplications of two n-bit integers.*

By combining the evaluation-interpolation scheme of §2.1 with FFT-based multi-evaluation (and interpolation, which is essentially identical and not discussed further), we obtain quasi-linear integer multiplication algorithms. We identify several tasks whose cost contribute to the bit complexity of such algorithms.

- converting the input integers to the polynomials in $\mathbf{R}[X]$;

Algorithm 3 General Cooley-Tukey FFT

Input: $P = \sum_{i=0}^{N-1} p_i X^i \in \mathbf{R}[X]$ of degree $N-1 = N_1 N_2 - 1$ with $N = 2^k$ and $N_i = 2^{k_i}$, and ω an N-th root of unity
Output: $P(1) + P(\omega)X + \cdots + P(\omega^{N-1})X^{N-1}$
function LargeRadixFFT(P,ω,N)
Let $(Q_i(X))_i$ be a sequence in $\mathbf{R}[X]$ such that $P(X) = \sum_{i=0}^{N_1-1} Q_i(X^{N_1})X^i$
for $i \in [\![0, N_1 - 1]\!]$ **do**
$Q_i \leftarrow \mathsf{FFT}(Q_i, \omega^{N_1})$
$Q_i \leftarrow Q_i(\omega^i X)$
end for
Let $(S_j(Y))_j$ be a sequence in $\mathbf{R}[Y]$ such that $\sum_i Q_i(X)Y^i = \sum_j S_j(Y)X^j$
for $j \in [\![0, N_2 - 1]\!]$ **do**
$S_j(Y) \leftarrow \mathsf{FFT}(S_j, \omega^{N_2})$
end for
return $\sum_{j=0}^{N_2} S_j(X^{N_1})X^j$
end function

- multiplications by roots of unity in the FFT computation;
- linear operations in the FFT computation (additions, etc);
- point-wise products involving elements of $\mathbf{R}$. Recursive calls to the integer multiplication algorithm are of course possible;
- recovering an integer from the computed polynomial.

The first and last items have linear complexity whenever the basis η from §2.1 is chosen as a power of 2 and the representation of elements of $\mathbf{R}$ is straightforward. Using notations described in §2.1, we have a bit complexity: $M(n) = O(C(N) \cdot K_{\mathsf{FFT}}(\mathbf{R})) + O(N \cdot K_{\mathsf{PW}}(\mathbf{R}))$ where $K_{\mathsf{PW}}(\mathbf{R})$ denotes the binary cost for the point-wise products in $\mathbf{R}$, while $K_{\mathsf{FFT}}(\mathbf{R})$ denotes the cost for the multiplication by elements of $\mathbf{R}$ which occur within the FFT computation. The costs $K_{\mathsf{PW}}(\mathbf{R})$ and $K_{\mathsf{FFT}}(\mathbf{R})$ may differ slightly.

More precise bit complexity estimates depend of course on the choice of the base ring $\mathbf{R}$. Some rings have special roots of unity allowing faster operations (multiplication, addition, subtraction in $\mathbf{R}$) than others. We now discuss several possible choices.

2.4. **Choice of the base ring.** There are several popular options for choosing the ring $\mathbf{R}$, which were given in [21]. We describe their important characteristics.

When it comes to roots of unity, choosing $\mathbf{R} = \mathbb{C}$ might seem natural. This needs careful analysis of the required precision.

- A precision of $t = \Theta(\log_2 n)$ bits is compatible with a transform length $N = O(\frac{n}{\log_2 n})$.
- The cost $K_{\mathsf{FFT}}(\mathbf{R})$ verifies $K_{\mathsf{FFT}}(\mathbf{R}) = K_{\mathsf{PW}}(\mathbf{R}) = M(\log_2 n)$.

Whence we obtain $M(n) = O(N \log_2 N \cdot M(\log_2 n)) = O(n \cdot M(\log_2 n)$. This leads to

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

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

Schönhage and Strassen proposed an alternative, namely to use the ring $\mathbf{R} = \mathbb{Z}/(2^t + 1)\mathbb{Z}$.

- In this case, t verifies $t = \Theta(\sqrt{n})$ and $N = \Theta(\sqrt{n})$.
- The cost $K_{\mathsf{FFT}}(\mathbf{R})$ corresponds to $N \log_2 N$ multiplications by powers of two in $\mathbf{R}$, which has a linear cost in the bitsize of elements of $\mathbf{R}$, so $K_{\mathsf{FFT}}(\mathbf{R}) = O(\sqrt{n})$.
- The cost $K_{\mathsf{PW}}(\mathbf{R})$ is the cost of a recursive call to the Schönhage-Strassen's algorithm, so $K_{\mathsf{PW}}(\mathbf{R}) = M(O(\sqrt{n}))$.

This leads to the complexity equation $M(n) \leq O(n \log_2 n) + 2\sqrt{n}M(\sqrt{n})$, which leads to the complexity $O(n \log_2 n \log_2^{(2)} n)$.

Pollard described in [19] how to compute the FFT when $\mathbf{R}$ is a field $\mathbb{F}_p$. This method supposes to know for each bitsize n a good composite number $P_n = p_{1,n}...p_{r,n}$ where the $p_{i,n}$ are primes and to use the chinese remainder theorem after having computed r FFT in $\mathbf{R} = \mathbb{Z}/p_{i,n}\mathbb{Z}$. We may suppose that r is "small" ($r = 1$ or $r = 2$). Thus, there should exist an algorithm computing the function mapping a size n to a prime p_n such that $p_n - 1$ is highly composite.

- The primes $p_{i,n}$ have to be of the order of magnitude of $O(\frac{\log_2 n}{r})$, which means that $N = O(\frac{n}{\log n})$.
- In this case, the costs $K_{\mathsf{PW}}(\mathbf{R})$ and $K_{\mathsf{FFT}}(\mathbf{R})$ verify the same property as in the case $\mathbf{R} = \mathbb{C}$.

We have to suppose that the sum of the costs of computing these primes is negligible compared to the computation of the FFT. The computation of an N-th principal root of unity in $\mathbb{Z}/p_n\mathbb{Z}$ for all the p_n should be negligible as well. Thus, we get the same complexity as in the complex field.

3. Fürer-type bounds

The two first 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 $K_{\mathsf{FFT}}(\mathbb{C})$, from expensive roots of unity which leads to the product $\log_2 n \cdot \log_2^{(2)} n \cdots \log_2^{(\log^\star n)} n$.

Fürer proposed two distinct algorithms in [9] and [11]. In [9], the proposed scheme relies on the assumption that there are infinitely many Fermat primes, which seems unlikely. Since the algorithm brought by the current work builds upon such a strategy, and can be seen as a fix to the fact that there are not enough Fermat primes, we briefly review here [9] and [11].

In [16], Harvey, van der Hoeven and Lecerf propose new algorithms achieving a bound similar to the one that Fürer gets. These algorithms rely on Bluestein's chirp transform [3] and they will not be detailed. However, a whole framework has been developped in the same paper, and this framework will be useful to the analysis of the algorithm proposed in the current work.

3.1. The Fermat prime multiplication.

Notation 7. *Let F_λ denote the λ-th Fermat number $2^{2^\lambda} + 1$.*

In [9], Fürer suggests an algorithm relying on the following assumption : there exists $k > 0$ such that for every m, there is a Fermat prime in the sequence $F_{\lambda+1}, F_{\lambda+2}, \ldots, F_{2^{\lambda+k}}$. Thus, it is tempting to compute the product of two integers of n-bit by using the FFT algorithm in a finite field $\mathbb{Z}/p\mathbb{Z}$ where p is a Fermat prime F_λ such that $2^\lambda < n < 2^{2^{\lambda-1}} 2^{\lambda-2}$.

Indeed, since 2 is a 2^λ-th root of unity in $\mathbb{Z}/F_\lambda\mathbb{Z}$, a fraction of the multiplications by a root of unity during the FFT are equivalent to a negacyclic permutation. The presence of the factor k in the assumption related to Fermat primes imply that a few calls to Schönhage-Strassen algorithm might have to be done.

The algorithm 4 follows the same scheme as PolynomialMultiply. In $\mathbb{Z}/F_\lambda\mathbb{Z}$, 3 is a principal $(F_\lambda - 1)$-th root of unity, which explains how it is possible to obtain an N-th principal root of unity.

FFTModFermat is a modification of LargeRadixFFT in the particular case where N is a power of 2. We decompose N as $N = 2^\lambda \cdot \frac{N}{2^\lambda}$ and we call recursively FFTModFermat for the $\frac{N}{2^\lambda}$-points FFT whereas we call Radix2FFT for the 2^λ-points FFT.

Algorithm 4 Multiplication of integers modulo a Fermat Prime

Input: a and b two n-bit integers, F_λ a Fermat prime such that $2^\lambda < n < 2^{2^{\lambda-1}}2^{\lambda-2}$
Output: $a \cdot b \mod 2^n - 1$
  function IntegerMultiplyModFermat(a,b,n,λ)
    Let $A \in \mathbb{Z}/F_\lambda\mathbb{Z}[X]$ be the polynomial such that $A(2^{\lambda-2}) = a$
    Let $B \in \mathbb{Z}/F_\lambda\mathbb{Z}[X]$ be the polynomial such that $B(2^{\lambda-2}) = b$
    $A' \leftarrow \mathsf{FFTModFermat}(A, 3^{(F_\lambda-1)/N}, \lambda, 2^{\lambda-1})$
    $B' \leftarrow \mathsf{FFTModFermat}(B, 3^{(F_\lambda-1)/N}, \lambda, 2^{\lambda-1})$
    $C \leftarrow \mathsf{PointwiseProduct}(A', B')$
    return $\mathsf{FFTModFermat}(C, 3^{-(F_\lambda-1)/N} \mod F_\lambda, \lambda)$
  end function

Algorithm 5 FFT modulo a Fermat Prime

Input: N a power of two, F_λ a Fermat prime, P a polynomial in $\mathbf{R}[X] = \mathbb{Z}/F_\lambda\mathbb{Z}$ of degree $N-1$, ω a N-th principal root of unity
Output: $P(1) + P(\omega)X + \cdots + P(\omega^{N-1})X^{N-1}$
  function FFTModFermat(P,ω,λ,N)
    if $N \leq 2^\lambda$ **then**
      return $\mathsf{Radix2FFT}(P, \omega, N, T_{F_\lambda})$
    end if
    $N_1 \leftarrow 2^\lambda$
    $N_2 \leftarrow N/N_1$
    Let $(Q_i(X))_i$ be a sequence in $\mathbf{R}[X]$ such that $P(X) = \sum_{i=0}^{N_1-1} Q_i(X^{N_1})X^i$
    for $i \in [\![0, N_1 - 1]\!]$ **do**
      $Q_i \leftarrow \mathsf{FFTModFermat}(Q_i, \omega^{N_1}, \lambda, N_2)$
      $Q_i \leftarrow Q_i(\omega^i X)$
    end for
    Let $(S_j(Y))_j$ be a sequence in $\mathbf{R}[Y]$ such that $\sum_i Q_i(X)Y^i = \sum_j S_j(Y)X^j$
    for $j \in [\![0, N_2 - 1]\!]$ **do**
      $S_j(Y) \leftarrow \mathsf{Radix2FFT}(S_j, \omega^{N_2}, N_1, T_{F_\lambda})$
    end for
    return $\sum_{j=0}^{N_2} S_j(X^{N_2})X^j$
  end function

Since the 2^λ-th principal roots of $\mathbb{Z}/F_\lambda\mathbb{Z}$ are powers of 2, the multiplications involved in the 2^λ-points FFT are negacyclic shifts of 2^λ-bit integers, whose binary complexity can be estimated by $O(2^\lambda)$. Thus, the algorithm 6 describes how to compute the composition involved in Radix2FFT negacyclic shifts.

We wish to count how many expensive multiplications by roots of unity are involved in the FFT computation, taking into account the recursive calls to FFTModFermat. This number is easily written as

$$E(N) = 2^{\lambda+1}E(\frac{N}{2^\lambda}) + N, \quad \text{whence } E(N) = N(\lceil \log_{2^\lambda} N \rceil - 1). \tag{1}$$

Algorithm 6 Twiddle factors obtained with the map T_{F_λ}

Input: F_λ a Fermat prime, P a polynomial of $\mathbb{Z}/F_\lambda\mathbb{Z}$ of degree $N < 2^\lambda$ such that $N+1$ is a power of two and the coefficients are represented in radix 2, ω a power of 2 such that $\omega = 2^j$ and $j = 2^{\lambda+1}/N$
Output: $P(\omega \cdot X)$

```
function T_{F_λ}(P,ω)
    if N = 1 then
        return P
    end if
    for i ∈ ⟦0, N − 1⟧ do
        l ← i · j mod 2^λ
        ϵ ← (−1)^{⌊i·j/2^λ⌋}
        Q_i ← ϵ · NegacyclicShift(P_i, 2^λ, l)        ▷ Negacyclic l-bit shift on a 2^λ-bit integer
    end for
    return Q(X) = Σ_i Q_i X^i
end function
```

Let us remember that $N = \frac{n}{2^{\lambda-2}}$ by hypothesis. The cost of the linear operations (for the binary complexity) during the computation of FFTModFermat, including the additions, the subtractions and the negacyclic shifts, is equal to $O(N \log N)$ operations in $\mathbb{Z}/F_\lambda\mathbb{Z}$, which means a binary cost equal to $O(N \log N \cdot 2^\lambda) = O(n \log n)$.

We get the following recursive formula for the binary complexity $M(n)$:

$$M(n) \leq N(3\lceil \log_{2^{\lambda+1}} N \rceil + 1) \cdot M(O(2^\lambda)) + O(n \log n)$$

The first product in the previous formula comes from the expensive multiplications involved in Fürer's algorithm and the second product describes the linear operations such as additions, subtractions, cheap multiplications. The integer 3 corresponds to the accumulation of two direct transforms, and one inverse transform for interpolation.

3.2. **An algorithm relying on multiplications modulo a polynomial.** Since the conjecture stating the existence of an infinite sequence of Fermat primes seems unlikely to hold, a new ring has to be found in order to improve the asymptotical complexity of the integer multiplication. Fürer proposed in [11] an algorithm FurerComplexMul using a ring with cheap roots of unity, yet allowing significantly larger transform length. We provide in this paragraph a description of this algorithm.

The ring used by Fürer is $\mathbf{R} = \mathbb{C}[x]/(x^{2^\lambda} + 1)$. The polynomial x is a natural $2^{\lambda+1}$-th principal root of unity in $\mathbf{R}$. However in this ring, we can also find roots of unity of larger order. For example an N-th principal root of unity may be obtained as the polynomial $\omega(x)$ meeting the conditions

$$\forall j \in [\![0, N[\![,\ \omega(e^{2ij\pi/N})^{N/2^{\lambda+1}} = x \circ (e^{2ij\pi/2^{\lambda+1}}) = e^{2ij\pi/2^{\lambda+1}}.$$

The actual computation of $\omega(x) \in \mathbf{R}$ can be done with Lagrange interpolation. A crucial observation is that $\omega^{\frac{N}{2^{\lambda+1}}} = x \in \mathbf{R}$, which means that a fraction $\frac{2^{\lambda+1}}{N}$ of the roots of unity reduces to a negacyclic permutation in $\mathbf{R}$.

Let us now consider how an FFT of length N in $\mathbf{R}[X]$ can be computed with Algorithm 3, with $N_1 = 2^{\lambda+1}$ and $N_2 = \frac{N}{2^{\lambda+1}}$. The N_1 transforms of length N_1 will be performed recursively with LargeRadixFFT. As for the N_2 transforms of length $N_1 = 2^{\lambda+1}$, since $\omega^{N_2} = x$, all multiplication by roots of unity within these transforms are cheap. The amount of expensive multiplications

is expressed in a similar fashion as in Equation (1), which allows one to state that $E(N) = N(\lceil \log_{2^{\lambda+1}} N \rceil - 1)$.

Fürer defined $\mathbf{R}$ with $2^\lambda = 2^{\lfloor \log^{(2)} n \rfloor}$ and proves that precision $O(\log n)$ is sufficient for the coefficients of the elements of $\mathbf{R}$ occuring in the computation. The integers to be multiplied are split into pieces of r bits, and blocks of $2^\lambda/2$ such pieces are grouped together to form the coefficients of an element of $\mathbf{R}$. In other terms, we have $N \leq 2n/\log^2 n$. Finally, using Kronecker substitution, we embed elements of $\mathbf{R}$ in $\mathbb{Z}$ and we call recursively the algorithm to multiply integers. We get the following recursive formula for the binary complexity $M(n)$:

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

Fürer proves that this recurrence leads to

$$M(n) \leq n \log n (2^{d \log^* \sqrt[4]{n}} - d')$$

for some $d, d' > 0$.

3.3. **Analysis framework of Harvey, van der Hoeven and Lecerf.** In [16], Harvey, van der Hoeven and Lecerf give a whole new algorithm allowing one to compute integer multiplication with a new repartition of cheap and expensive multiplications during the FFT and the use of the Bluestein's chirp transform [3]. Basically, they transform the computation of the Radix2FFT into the computation of the multiplication of two integers, which changes the balance of the cost of different multiplications in Fürer's algorithm.

We focus in this part on the framework developed in [16] simplifying the analysis of the algorithm proposed in the current paper and we reuse some tricks in the algorithm itself.

Definition 8. *Let $\mathbf{\Phi} : (x_0, \infty) \to \mathbb{R}$ be a smooth increasing function, for some $x_0 \in \mathbb{R}$. We say that $\mathbf{\Phi}^* : (x_0, \infty) \to \mathbb{R}^{\geq}$ is an iterator of $\mathbf{\Phi}$ if $\mathbf{\Phi}^*$ is increasing and if*

$$\mathbf{\Phi}^*(x) = \mathbf{\Phi}^*(\mathbf{\Phi}(x)) + 1$$

for all sufficiently large x.

An iterator $\mathbf{\Phi}^*$ can be thought as a generalization of the map $x \to \log^\star(x)$ for $\mathbf{\Phi} = \log$.

Definition 9. *A function $\mathbf{\Phi}$ is logarithmically slow if there exists $2^\lambda \in \mathbb{N}$ such that*

$$(\log^{(2^\lambda)} \circ\, \mathbf{\Phi} \circ \exp^{(2^\lambda)})(x) = \log x + O(1)$$

for $x \to \infty$.

Proposition 10. *For any iterator $\mathbf{\Phi}^*$ of a logarithmically slow function $\mathbf{\Phi}$, we have*

$$\mathbf{\Phi}^*(x) = \log^* x + O(1).$$

Proposition 10 allows us to count the number of recursive calls performed by the algorithm described in Section 4.

Moreover, we have to reuse 2 ideas proposed in [16] improving the complexity analysis.

- It appears that during the execution of LargeRadixFFT, we have to multiply elements of $\mathbf{R}$ by the same root of unity ω several times. Since the multiplication by ω involves the computation of the Fourier transform of the integer associated to ω given by the Kronecker substitution, we can compute it once, and reuse it for all the multiplications requiring ω.
- Finding primes may be an expensive operation. This is why it is proposed to use FurerComplexMul (described in 3.2) at first, and, for deeper levels of recursion, we switch to another algorithm. Consequently, we spare the computation of the most expensive prime. Moreover, the remaining primes that we use are small enough to make their computation negligible.

Those ideas are discussed again in Section 5.

4. Generalized Fermat Primes

This section states the main hypothesis on which the forecoming complexity analysis relies and proves some properties of the density of generalized Fermat primes.

Notation 11. *Let $L(R,\lambda)$ denote the number of generalized Fermat primes $r^{2^\lambda}+1$ with $r \leq R$.*

A conjecture about generalized Fermat primes is needed in order to fix the conjecture of [9]. Such a conjecture requires supportive arguments from results (effective, if possible) on generalized Fermat primes. Those arguments differ from the arguments supporting the conjecture made in [16, §9.1] on Mersenne primes, which we mention here.

Conjecture 12 ([16, §9.1])**.** *Let $\pi_m(x)$ be the number of Mersenne primes less than x. Then there exist constants $0 < a < b$ such that for all $x > 3$*

$$a \log^{(2)} x < \pi_m(x) < b \log^{(2)} x.$$

This conjecture is a weaker version of a conjecture stated by Lenstra, Pomerance, Wagstaff, who conjectured that

$$\pi_m(x) \sim \frac{e^\gamma}{\log 2} \log^{(2)} x$$

as $x \to \infty$. This conjecture relies on probabilistic arguments [13], investigated by Wagstaff in [26] and supported by numerical evidence.

Dubner and Gallot proposed in [7] a study of generalized Fermat primes, supported by numerical evidence and heuristic arguments quite similar to the ones exhibited by Wagstaff. Actually, the fact that there is probably an infinite amount of generalized Fermat primes is a particular case of the "Hypothesis H" stated in 1958 by Sierpiński and Schinzel, and given through a quantitative form by Bateman and Horn in 1962 [2]. Dubner and Gallot [7] give an estimation of the number of generalized Fermat primes $r^{2^\lambda}+1$ for each λ, from $r=2$ to R, assuming the "Hypothesis H". We need a notation to introduce this estimation.

Notation 13. *Let $E(R,\lambda)$ be the expectancy of the number of generalized Fermat primes $r^{2^\lambda}+1$ for r in $[2,R]$, defined as:*

$$E(R,\lambda) = \frac{C_\lambda}{2^\lambda} \sum_{r=2}^{R} \frac{1}{\log r} \sim \frac{C_\lambda}{2^\lambda} \int_2^R \frac{dt}{\log t}. \tag{3}$$

The term C_λ in Equation (3) denotes the quantity $\lim\limits_{K\to\infty} \dfrac{t(K,\lambda)}{u(K,\lambda)}$ where

$$t(K,\lambda) = \prod_{\substack{k \in [\![1,K]\!] \\ k\cdot 2^{\lambda+1}+1 \text{ prime}}} \left(1 - \frac{2^\lambda}{k\cdot 2^{\lambda+1}+1}\right),$$

which is a factor due to the fact that the possible factors of $r^{2^\lambda}+1$ must be of the form $k\cdot 2^{\lambda+1}+1$, and

$$u(K,\lambda) = \prod_{p \text{ prime}}^{K\cdot 2^{\lambda+1}+1} \left(1 - \frac{1}{p}\right)$$

which describes the product of all primes up to the largest one considered in $t(K,\lambda)$. The proof of the fact that $\lim\limits_{K\to\infty} \dfrac{t(K,\lambda)}{u(K,\lambda)}$ exists is given in [2].

When specialized to the case of generalized Fermat primes, the quantitative form of the "Hypothesis H" can be stated as follow.

Conjecture 14 (Hypothesis H applied to generalized Fermat primes)**.** *For any λ, the actual number of generalized Fermat $r^{2^\lambda}+1$ is equivalent to the expectancy $E(R,\lambda)$:*

$$L(R,\lambda) \sim_R E(R,\lambda).$$

Considerations in this article lead us to need the following statement.

Proposition 15. *There exists an absolute constant $C>0$ such that $C_\lambda \geq \frac{C}{\lambda}$ for any λ.*

Proof. See Appendix A. □

Given the numerical values brought by [7], we may consider that this lower bound is quite pessimistic.

Proposition 16. *Assuming Hypothesis H, for any $\lambda>0$, there exists a bound $B_\lambda \geq 2^\lambda$ such that for any $R>B_\lambda$, there exists a generalized Fermat prime $r^{2^\lambda}+1$ such that $r \in [\![R, R\cdot(1+\lambda^2)]\!]$.*

Proof. We come back to the expectancy

$$E(R,\lambda) = \frac{C_\lambda}{2^\lambda}\sum_{r=2}^{R}\frac{1}{\log r}.$$

We have to consider in our case the quantity $\Delta(R,\lambda) = E(R\cdot(1+\lambda^2),\lambda) - E(R,\lambda)$. Assuming the Hypothesis H, there exists B'_λ such that for $R>B'_\lambda$,

$$L(R\cdot(1+\lambda^2),\lambda) - L(R,\lambda) > \frac{1}{2}\Delta(R,\lambda).$$

Thus, one needs to prove that for $B_\lambda > B'_\lambda$ large enough and $R>B_\lambda$, $\Delta(R,\lambda)\geq 2$. In fact, we even prove that $\min\limits_{R>B_\lambda}|\Delta(R,\lambda)|\to\infty$ as λ goes to infinity.

It is known that $\int_2^x \frac{dt}{\log t} \sim \frac{x}{\log x}$. Then, for any $\mu>0$ and x large enough,

$$(1-\mu)\cdot\frac{x}{\log x} \leq \int_2^x \frac{dt}{\log t} \leq (1+\mu)\cdot\frac{x}{\log x}.$$

Since $\sum_{r=2}^R \frac{1}{\log r} \sim \int_2^R \frac{dt}{\log t}$, for any $\nu>0$ and R large enough,

$$(1-\nu)\cdot\int_2^R \frac{dt}{\log t} \leq \sum_{r=2}^{R}\frac{1}{\log r} \leq (1+\nu)\cdot\int_2^R \frac{dt}{\log t}.$$

It is possible now to get a lower bound for $\Delta(R,\lambda)$ assuming that λ is large enough:

$$\Delta(R,\lambda) \geq \frac{C_\lambda}{2^\lambda}\cdot\left((1-\nu)(1-\mu)\cdot\frac{R\cdot(1+\lambda^2)}{\log(R\cdot(1+\lambda^2))} - (1+\nu)(1+\mu)\cdot\frac{R}{\log R}\right).$$

Using the constant C given by Proposition 15 and $2^\lambda \leq R$:

$$\Delta(R,\lambda) \geq \frac{C}{\lambda}\cdot\left((1-\nu)(1-\mu)\cdot\frac{R\cdot(1+\lambda^2)}{\log(R\cdot(1+\lambda^2))} - (1+\nu)(1+\mu)\cdot\frac{R}{\log R}\right).$$

In conclusion,

$$\Delta(R,\lambda) \geq \frac{C}{\lambda}\cdot\left((1-\nu)(1-\mu)\cdot\frac{1+\lambda^2}{2\log(2^\lambda)+\log(1+\lambda^2)} - (1+\nu)(1+\mu)\cdot\frac{1}{\log R}\right),$$

which verifies $\min\limits_{R>B_\lambda}|\Delta(R,\lambda)|\to\infty$ as $\lambda\to\infty$ for μ and ν sufficiently small. □

Assuming the Hypothesis H, for any constant $K > 1$ and any integer λ, there exists a bound B_λ such that for $R \geq B_\lambda$,

$$L(R, \lambda) \geq \frac{1}{K} \cdot E(R, \lambda).$$

However, we are unable to predict how large this bound B_λ can be. In [1], Adleman and Odlyzko propose a conjecture based on generalized Riemann hypothesis suggesting that B_λ is smaller than $\exp(O(\lambda 2^\lambda))$ in the context of generalized Fermat primes. In [18], the authors prove that there exist infinitely many irreducible polynomials f that do not take prime values below $\exp\left(\exp(O(\frac{\log \ell(f)}{\log^{(2)} \ell(f)}))\right)$, with

$$\ell\left(\sum a_i x^i\right) = \sum \text{bitsize}(1 + |a_i|).$$

In this work, we establish a complexity depending on how small B_λ can be. The following hypothesis gives a sufficient condition in order to get the best asymptotic bound that we can reach using the algorithm described in Section 5.

Hypothesis 17. *There exists an absolute constant $K > 1$ and a sequence $\gamma(\lambda)$ verifying*

$$\lambda \leq \gamma(\lambda) \leq \frac{1}{2}\lambda \log_2 \lambda - \frac{1}{4}\lambda$$

such that for λ large enough and any $X \geq 2^{\gamma(\lambda)}$,

$$L(X, \lambda) \geq \frac{1}{K} \cdot E(X, \lambda).$$

However, for our purpose, Hypothesis 17 is too strong. Thus, the complexity analysis of the algorithm described in this work is unchanged if the weaker hypothesis 18 is used.

Hypothesis 18. *There exists a sequence $\gamma(\lambda)$ verifying*

$$\lambda \leq \gamma(\lambda) \leq \frac{1}{2}\lambda \cdot \log_2 \lambda - \frac{1}{4}\lambda$$

such that for λ large enough and for any X such that $2^{\gamma(\lambda)} \leq X \leq 2^{2\cdot\gamma(\lambda)}$, there exists a generalized Fermat prime $r^{2^\lambda} + 1$ such that $r \in [\![X, X(1 + \lambda^2)]\!]$.

Hypothesis 18 allows one to reach the best complexity bound using generalized Fermat primes for the multiplication of two n-bit integers, which means $O(n \log n \cdot 4^{\log^* n})$.

Weaker hypotheses can also be formulated:

Hypothesis 19. *There exists a sequence $\gamma(\lambda) \geq \lambda$ verifying $\gamma(\lambda) = 2^{o(\lambda)}$ such that for λ large enough and for any X such that $2^{\gamma(\lambda)} \leq X \leq 2^{2\cdot\gamma(\lambda)}$, there exists a generalized Fermat prime $r^{2^\lambda} + 1$ such that $r \in [\![X, X(1 + \lambda^2)]\!]$.*

Hypothesis 20. *There exists a sequence $\gamma(\lambda) \geq \lambda$ verifying $\gamma(\lambda) = 2^{\lambda + o(\lambda)}$ such that for λ large enough and for any X such that $2^{\gamma(\lambda)} \leq X \leq 2^{2\cdot\gamma(\lambda)}$, there exists a generalized Fermat prime $r^{2^\lambda} + 1$ such that $r \in [\![X, X(1 + \lambda^2)]\!]$.*

Hypothesis 19 and 20 would give respectively $O(n \log n \cdot 8^{\log^* n})$ and $O(n \log n \cdot 16^{\log^* n})$ for the complexity analysis of the multiplication of two n-bit integers. We might have considered even weaker hypothesis, but the complexity would not be interesting.

In table 1, we compare

$$\Delta(R, \lambda) \sim C_\lambda \cdot \frac{R}{2^\lambda} \cdot \left(\frac{1 + \lambda^2}{\log(R \cdot (1 + \lambda^2))} - \frac{1}{\log R}\right),$$

used in the proof of Proposition 16, to the actual number of generalized Fermat primes $r^{2^\lambda}+1$, taking $R=2^\lambda$ and $R=2^{2\lambda}$. We observe that in practice the estimates given by the conjecture H are close to the reality and that it is not hard to find a suitable generalized Fermat prime for Hypothesis 18.

Interval	$[2^\lambda, 2^\lambda\cdot(1+\lambda^2)]$		$[2^{2\lambda}, 2^{2\lambda}\cdot(1+\lambda^2)]$	
λ	Actual number	Estimate	Actual number	Estimate
2	4	2.54	11	8.36
3	0	3.77	22	21.88
4	10	9.81	139	108.59
5	19	12.93	310	278.32
6	23	17.83	824	752.28
7	16	17.09	1553	1420.09
8	42	48.38	8614	7932.90
9	75	56.49	19707	18182.83
10	82	68.98	—	44289.34

TABLE 1. Number of generalized Fermat primes $r^{2^\lambda}+1$ with r in $[2^\lambda, 2^\lambda\cdot(1+\lambda^2)]$ and $[2^{2\lambda}, 2^{2\lambda}\cdot(1+\lambda^2)]$.
Taking $\gamma(\lambda)=\lambda$, if the second and the fourth column are never 0 for $\lambda\geq 4$ the extreme cases $X=2^{\gamma(\lambda)}$ and $X=2^{2\gamma(\lambda)}$ in Hypothesis 18 are verified.

5. A new algorithm

This section describes a new algorithm to multiply integers. Since it relies on some assumption about the repartition of particular primes, its complexity is conjectural. We prove in the next section that this conjectural complexity is the same as the one that Harvey, van der Hoeven and Lecerf get from the use of Mersenne primes with the Bluestein's chirp transform [16]. However, this new algorithm seems to be simpler to implement.

5.1. A new ring.

The main contribution of this article is to propose a ring $\mathbf{R}$ allowing one to get rid of the disadvantages of the polynomial ring proposed by Fürer [11] and yet benefit from the advantages of the construction proposed in [9]. Instead of working in $\mathbb{C}[x]/(x^{2^\lambda}+1)$, we work in $\mathbf{R}=\mathbb{Z}/p\mathbb{Z}$, where p is a prime chosen as

$$p=r^{2^\lambda}+1.$$

We call GenFermatMul the algorithm computing the multiplication of two n-bit integers with the Cooley-Tukey FFT and using $\mathbf{R}$ as the base ring.

The n-bit integers a and b to be multiplied are decomposed in some base η, as in §2.1. Consequently, we multiply two polynomials

$$A=\sum_{i=0}^{N-1} a_i x^i$$

and

$$B=\sum_{i=0}^{N-1} b_i x^i$$

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 2. Primes used in the algorithm taking $\gamma(\lambda) = \lambda$ in Hypothesis 18

whose coefficients are bounded by 2^η. Thus, the coefficients c_i of $C = A \cdot B \mod x^N - 1$ have the following form:

$$|c_i| = |\sum_{j=0}^{i} a_j \cdot b_{i-j} + \sum_{j=i+1}^{N-1} a_j \cdot b_{N-1-j}| < N \cdot 2^{2\cdot\eta}$$

Thus, for the evaluation-interpolation scheme to be valid, the parameter η and the transform length N must be such that $\log_2 N + 2\log_2 \eta \leq \log_2 p$.

The integer 2^λ plays a role similar to 2^λ in Fürer's construction. We therefore define it likewise, as the smallest power of 2 above $\log_2 n$. We denote by μ the power of 2 such that $\frac{1}{2}\gamma(\lambda) \leq \mu \leq \gamma(\lambda) - 1$, where $\gamma(\lambda)$ is the sequence defined in Hypothesis 18, 19 and 20.

Let then $\eta = \mu 2^\lambda$ and the integer r is chosen subject to the condition that

$$\log_2(r^{2^\lambda}) \geq 2\eta + \log_2 \frac{n}{2^\lambda \mu} \tag{4}$$

which is equivalent to

$$r > R, \text{ with } R = 2^{2\mu + \frac{\log_2 n}{2^\lambda} - \frac{\log_2(2^\lambda \mu)}{2^\lambda}}.$$

By definition of μ, we have

$$\log_2(2^\lambda \mu) \leq \lambda + \log_2(\gamma(\lambda) - 1).$$

So, assuming Hypothesis 18, 19 or 20, there exists n_0 such that for $n \geq n_0$,

$$\log_2(2^\lambda \mu) \leq \log_2 n. \tag{5}$$

Assuming in particular Hypothesis 18, we have $\lambda + \log_2(\gamma(\lambda) - 1) \leq \lambda + \log_2 \lambda + \log_2^{(2)} \lambda - 1$. Since $\lambda + \log_2 \lambda + \log_2^{(2)} \lambda - 1 \leq 3\log_2^{(2)} n + 2$, we observe that the inequality (5) is met for $n \geq 2^{16}$. Thus, we expect that the constant n_0 is not too large.

For $n \geq n_0$, the bound $R = 2^{2\mu + \frac{\log_2 n}{2^\lambda} - \frac{\log_2(2^\lambda \mu)}{2^\lambda}}$ verifies the conditions of Hypothesis 18, 19 and 20:

$$2^{\gamma(n)} \leq R \leq 2^{2\gamma(n)}.$$

Indeed, we have

$$\gamma(\log_2 2^\lambda) + \frac{\log_2 n}{2^\lambda} - \frac{\log_2 n}{2^\lambda} \leq 2\mu + \frac{\log_2 n}{2^\lambda} - \frac{\log_2(2^\lambda \mu)}{2^\lambda} \leq 2\gamma(\log_2 2^\lambda) - 2 + 1.$$

In conclusion, there exists a size n_0 such that, for any $n \geq n_0$, we are able to find a prime verifying the constraint (4). For bit sizes in the foreseeable relevant practical range and well beyond, the primes p may be chosen as given by Table 2.

Notation 21. *Let $s(2^\lambda)$ be the value $2^{\lambda+1}(\gamma(\lambda)+2\log_2\lambda+1)$ for $\gamma(\lambda)$ chosen as in Hypothesis 18, 19 or 20. It is an upper bound of $\log_2 p$, since we have $p=r^{2^\lambda}+1$ and*

$$\log_2 r \leq \log_2\left(2^{2\gamma(\lambda)}\cdot(1+\lambda^2)\right) \leq 2\gamma(\lambda)+\log_2(1+\lambda^2) \leq 2\cdot\gamma(\lambda)+2\log_2\lambda+1.$$

5.2. **Arithmetic properites of the ring chosen.** We now discuss the advantages of the ring $\mathbf{R}=\mathbb{Z}/p\mathbb{Z}$ in our context. The notation $s(2^\lambda)$ denoting a bound on the bit size of p used for multiplying two n-bit integers will be retained throughout this section. We remind that the fact that $p=r^{2^\lambda}+1$ is prime implies that there exists an element of order $p-1$ in $\mathbf{R}$. Since r is even, we have a 2^{2^λ}-th principal root of unity and there exists, consequently, for any power two N such that $N<2^{2^\lambda}$, an N-th principal root of unity. This allows large transform lengths.

Using the base ring $\mathbf{R}$ allows a more compact encoding of the integers to be multiplied compared to Fürer's algorithm presented in [11]. Indeed, the embedding of the coefficients a_i of the polynomial A described in §2.1 leads to elements in $\mathbb{C}[x]/(x^{2^\lambda}+1)$ whose representation $\sum_{i=0}^{2^\lambda-1} a_{ij}x^j$ has $a_{ij}=0$ for $j\in[\![2^\lambda/2,2^\lambda[\![$ and the number of bits required to store the a_{ij} is approximatively twice smaller than the number of bits required to store the coefficients of the product. The elements of the ring $\mathbf{R}$ require approximatively $2\eta=2\mu\cdot 2^\lambda\approx\log_2 r\cdot 2^\lambda$ bits to be represented and, initially, the coefficients of A fit on $\eta=\mu\cdot 2^\lambda$ bits by construction. In other terms, the coefficients of A occupy one fourth of the bitsize of the ring $\mathbb{C}[x]/(x^{2^\lambda}+1)$, whereas in $\mathbf{R}$, they occupy one half of the bitsize.

Given the form of p, elements x of $\mathbb{Z}/p\mathbb{Z}$ can be represented as $x=\sum_{i=0}^{2^\lambda-1}x_i r^i$, with $0\leq x_i<r$ (since the element $r^{2^\lambda}=-1$ cannot be decomposed like the other elements, it has to be treated separately). In other terms, we write down the expansion of x in radix r.

During the Fast Fourier Transform, we perform various operations in the ring $\mathbf{R}$, among which additions, subtractions, multiplications of two ordinary elements of $\mathbf{R}$ and multiplications by powers of r.

- Additions and subtractions are computed within a time linear in 2^λ: computing the sum of $\sum_i x_i r^i$ and $\sum_i y_i r^i$ is basically computing all the sums x_i+y_i and handling the propagation of carries.
- Multiplications of two ordinary elements $\sum_i x_i r^i$ and $\sum_i y_i r^i$ of $\mathbf{R}$ is done by considering these elements as polynomials in T: $\sum x_i T^i$ and $\sum y_i T^i$ in $\mathbb{Z}[T]/(T^{2^\lambda}+1)$.
- Multiplications by powers of r are as hard as negacyclic shifts in $\mathbb{Z}[T]/(T^{2^\lambda}+1)$.

Let us explain why multiplications by powers of r are negacyclic shifts. As an example, the product of $x=\sum_{i\in[\![0,2^\lambda-1]\!]}x_i r^i$ with r writes as:

$$\begin{aligned} x\cdot r &= \sum_{i\in[\![1,2^\lambda]\!]} x_{i-1}r^i = -x_{2^\lambda-1}+\sum_{i\in[\![1,2^\lambda-1]\!]} x_{i-1}r^i, \\ &= (-x_{2^\lambda-1}+r)+(x_0-1)\cdot r+\sum_{i\in[\![2,2^\lambda-1]\!]} x_{i-1}r^i. \end{aligned}$$

We see that in contrast to Fürer's choice, operations in $\mathbf{R}$ must take into account the propagation of carries (possibly further than in the example above, as x_0 may be zero). However the induced cost remains linear.

Moreover, there are a few more operations to take into account in the formula representing the complexity estimate of the algorithm:

- The decomposition of the elements of $\mathbf{R}$ in base r. According to [4, §1.7], this can be done in $c\cdot M(s(2^\lambda))\log 2^\lambda$ for some constant c.

- The modulo r operations after a multiplication in radix-r representation. This requires 2^λ successive reductions modulo r, hence a total cost $c'2^\lambda M(s(2^\lambda)/2^\lambda)$ for some constant c'.
- The linear operations: additions, subtractions, multiplication by powers of r and propagation of carries.

Finally, the fact that we multiply polynomials over $\mathbb{Z}[T]/(T^{2^\lambda}+1)$ implies that the integers associated to those polynomials through Kronecker substitution can be multiplied modulo 2^m+1 for some m. Since multiplying two m-bit integers modulo 2^m+1 implies to multiply polynomials modulo $X^{m/\eta}+1$, we have to use the half-DFT detailed in 2.2, as it is done in Schönhage-Strassen algorithm.

5.3. **Notations and overall structure of the algorithm.** In [16, §6], the authors use the idea that some roots of unity are involved in several multiplications. Let us say that in the ring $\mathbf{R}$ we consider the multiplications by ω, ω being an N-th principal root of unity involved in GenFermatMul. Then let $(a_i)_{1\leq i\leq m}$ be the sequence of elements of $\mathbf{R}$ multiplied by ω in LargeRadixFFT. These m multiplications involve m recursive calls to GenFermatMul, and computing m times a Fourier transform for the integer corresponding to the Kronecker substitution of ω, which can be optimized. Thus, we have to precompute, before the Fast Fourier Transform, the transforms of the N roots of unity by which we multiply elements of $\mathbf{R}$.
Description of the rings. Algorithm GenFermatMul is a recursive algorithm. We assume that we use Kronecker substituion for recursive calls to GenFermatMul. Let i denote the denote the depth in the tree describing all the recursive calls. The quantity $\mathcal{I}$ describes the number of levels of recursion.

- Let p_{i+1} be the prime used by every instance of GenFermatMul at the depth i.
- Let r_{i+1} and λ_{i+1} be the quantities such that $p_{i+1}=r_{i+1}^{2^{\lambda_{i+1}}}+1$.
- Let n_i be the bitsize of the input used by GenFermatMul at the depth i. Since we use Kronecker substitution, $n_i\approx 2\log_2 p_i$.
- Let the sequence (N_i) corresponds to the successive degrees implied at the depth i.
- Let $\mathbf{R}_{i+1}$ denote the ring $\mathbb{Z}/p_{i+1}\mathbb{Z}$.
- Let ω_{i+1} be a $2N_i$-th principal root of unity of $\mathbf{R}_{i+1}$.

The parameters defined above can be precomputed. Thus, we store in a list $\mathcal{L}$ the set of all the primes p_i. All the roots ω_i are computed and represented in radix r_i. The powers of these principal roots of unity ω_i^j for $j\in[\![0,2N_{i-1}-1]\!]$ are computed and we store in a list $\mathcal{G}_i$ their $(2N_{i-1}+1)$-Fourier transforms over $\mathbb{Z}/p_{i+1}\mathbb{Z}$.

In conclusion, we completely describe the rings used by our algorithm with

- a list $\mathcal{L}$ of primes $F(r_i,\lambda_i)$ used at each level of recursion,
- a list $\mathcal{G}_i$ of the Fourier transforms of the $2N_{i-1}$-th roots of unity in radix r_i for each level i.

Algorithm Precomputations describes how the lists $\mathcal{L}$ and $\mathcal{G}_i$ are computed.

Algorithm 8 is a description of GenFermatMul. In lines 12, 13 and 15 of Algorithm 8, we need to define which procedure we use to compute the Fourier transforms. Because the ring $\mathbf{R}_i$ has "cheap" roots of unity, we achieve this with Algorithm NewLargeRadixFFT which is a variation around algorithm LargeRadixFFT presented in Section §2. We do not describe the algorithm computing the inverse Fourier transform, since its pseudocode would be essentially the same as NewLargeRadixFFT.

5.4. **Expressing the complexity from the sequences $(n_i)_i$ and $(\lambda_i)_i$.** Since the algorithm proposed in the current work relies on finding some primes sufficiently quickly, the remarks of [16, §8.2] are useful as well, which has already been discussed in §3.3. It is suggested in particular

Algorithm 7 Precomputations

Input: n a bitsize
Output: $\mathcal{L}$ and $\mathcal{G}_i$

function GenFermatMulPrecomp(n)
  $\mathcal{L} \leftarrow \{\ \}$
  $s \leftarrow n$
  while s is large enough **do**
    2^λ is the smallest integer above $\log_2 s$
    μ is the smallest power of two above $\frac{1}{2}\gamma(\lambda)$
    $\eta \leftarrow 2^\lambda \cdot \mu$
    $N \leftarrow s/\eta$ is the degree of the polynomials that will be multiplied
    p is the smallest prime verifying the constraint (4)
    m is the size given by the Kronecker substitution for multiplying elements of $\mathbb{Z}/p\mathbb{Z}$
    $s \leftarrow m$
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{p\}$
  end while
  $(\mathcal{G}_i)_i \leftarrow (\{\ \})_i$
  for $i \in [\![0, \#\mathcal{L} - 1]\!]$ **do**
    i is the index of the prime $p_i = r_i^{2^{\lambda_i}} + 1$ in $\mathcal{L}$
    g_i is a generator of $\mathbf{R_i} = \mathbb{Z}/p_i\mathbb{Z}^\times$
    N_i is the degree of the polynomials representing elements of $\mathbf{R_i}$
    ω_i is an $2N_{i-1}$-th root of unity in $\mathbf{R_i}$ represented in radix r_i
    for $j \in [\![0, 2N_{i-1} - 1]\!]$ **do**
      $\mathcal{G}_i \leftarrow \mathcal{G}_i \cup \{\mathsf{FFT}(\omega_i^j, \mathbf{R_{i+1}})\}$ ▷ We store the Fourier Transforms of ω_i^j in $\mathbf{R_{i+1}} = \mathbb{Z}/p_{i+1}\mathbb{Z}$
    end for
  end for
  return $\mathcal{L}$,$(\mathcal{G}_i)_i$
end function

that, since there is essentially one prime that is expensive to compute, and which corresponds to the first encountered prime, FurerComplexMul is called for the top level and for deeper recursion levels we switch to another algorithm.

Indeed, assuming that for an input of size n we compute an FFT modulo a prime p verifying $p = 2^{O(\log_2 n)^2}$ ($O(\log_2 n)^2$ corresponds to the size of integers that we multiply on the second level of recursion in FurerComplexMul), the bitsize n' on deeper levels of recursion verifies $n' = O(\log_2^2 n)$. The prime p' corresponding to n' verifies

$$p' = 2^{O(\log_2(\log_2^2 n))^2} = 2^{O(\log_2 \log_2 n)^2} = \log_2 n^{O(\log_2^{(2)} n)} = o(n).$$

Using the Eratosthenes sieve (which is even not the fastest known algorithm) to find all primes below n, we get a complexity estimated to $O(n \log_2^{(2)} n)$.

In conclusion, at the toplevel it is required to use the algorithm FurerComplexMul, and to call GenFermatMul on deeper levels of recursion, using Kronecker substitution to transform polynomials into integers, or any other algorithm which is more efficient for the multiplication of polynomials in $\mathbb{Z}[T]/(T^{2^\lambda} + 1)$. Thus, at the recursive depth $i = 0$ we use FurerComplexMul and n_0 corresponds to the bitsize of elements of $\mathbf{R}_0 = \mathbb{Z}/p_0\mathbb{Z}$ passed to GenFermatMul during the multiplications involved in FurerComplexMul.

Algorithm 8 GenFermatMul

Input: a, b two integers of $\lceil n_i \rceil$ bits
Output: $c = a \times b \mod 2^{n_i} + 1$

function GenFermatMul(a,b,i,$\mathcal{L}$,$\mathcal{G}$)
  if $i < \mathcal{I}$ **then**
    return $a * b$ ▷ Basecase multiplication
  else
    μ is the smallest power of two above $\frac{1}{2}\gamma(\lambda_{i+1})$
    $\eta \leftarrow 2^{\lambda_{i+1}} \cdot \mu$
    A is the polynomial such that $A(\eta) = a$
    B is the polynomial such that $B(\eta) = b$
    A' is the polynomial $A(\omega_{i+1} \cdot X)$ ▷ Due to Half-FFT
    B' is the polynomial $B(\omega_{i+1} \cdot X)$ ▷ Due to Half-FFT
    Decompose the coefficients of A' and B' in radix r_{i+1}
    $P \leftarrow \sum_j A'(\omega_{i+1}^{2j})X^j$
    $Q \leftarrow \sum_j B'(\omega_{i+1}^{2j})X^j$ ▷ Multiplications modulo p_{i+1} are done with a Kronecker substitution and a recursive call to GenFermatMul.
    $R \leftarrow \mathsf{ComponentwiseProduct}(P, Q)$
    $S \leftarrow \sum_j R(\omega_{i+1}^{-2j})X^j$
    $S \leftarrow \frac{1}{N_i} \cdot S$
    Recompose the coefficients of S from radix r_{i+1} to radix 2
    $S \leftarrow S(\omega_{i+1}^{-1} \cdot X)$ ▷ Composition due to Half-FFT
    return $S(2^\eta)$
  end if
end function

Algorithm 9 Optimized Large Radix FFT

Input: $P = \sum_{j=0}^{N_i-1} p_j X^j \in \mathbf{R}_{i+1}[X]$ and $\mathcal{G}_{i+1}$
Output: $P(1) + P(\omega_{i+1}^2)X + \cdots + P(\omega_{i+1}^{2(N_i-1)})X^{N_i-1}$

function NewLargeRadixFFT(P,$\mathcal{G}_{i+1}$)
  Let $(Q_j(X))_j$ be a sequence in $\mathbf{R}_{i+1}[X]$ such that $P(X) = \sum_{j=0}^{2\cdot 2^{\lambda_{i+1}}-1} Q_j(X^{2\cdot 2^{\lambda_{i+1}}})X^j$
  for $j \in [\![0, 2^{\lambda_{i+1}+1} - 1]\!]$ **do**
    $Q_j \leftarrow \mathsf{NewLargeRadixFFT}(Q_j, \mathcal{G}_{i+1})$
    $Q_j \leftarrow Q_j(\mathcal{G}_{i+1}[2j] \cdot X)$ ▷ The factor 2 is due to Half-FFT
  end for
  Let $(S_j(Y))_j$ be a sequence in $\mathbf{R}_{i+1}[Y]$ such that $\sum_k Q_k(X)Y^k = \sum_j S_j(Y)X^j$
  for $j \in [\![0, N_i/2^{\lambda_{i+1}+1} - 1]\!]$ **do**
    $S_j(Y) \leftarrow \mathsf{CheapFFT}(S_j, r_{i+1})$ ▷ The multiplications by r_{i+1}^j are negacyclic shifts
  end for
  return $\sum_{j=0}^{N_i/2^{\lambda_{i+1}+1}} S_j(X^{N_i/(2\cdot 2^{\lambda_{i+1}})})X^j$
end function

Let us denote by $M(n)$ the binary complexity of the algorithm described in 5.3 computing the product of two n-bit integers a and b using FurerComplexMul on toplevel and GenFermatMul on deeper levels.

Let us denote by $U^{\mathcal{G},\mathcal{L}}(i)$ the binary complexity of the algorithm multiplying two elements of $\mathbf{R}_i$ using GenFermatMul, after having precomputed the lists $\mathcal{G}_i$ and $\mathcal{L}$. We write it $U(i)$ if these lists are implicitly specified.

Let $U_\omega(i)$ correspond to the complexity of the algorithm multiplying any element of $\mathbf{R}_i$ and an element of $\mathcal{G}_i$.

Let S denote the binary complexity of the multiplication of two n-bit integers modulo 2^n+1 using the Schönhage-Strassen algorithm: $S(n) \leq dn \log_2 n \log_2 \log_2 n$ for some $d > 0$.

Since the recursive equation associated to FurerComplexMul has already been described in 3.2, we can reuse it and injecting into this equation the complexity U of GenFermatMul:

$$M(n) \leq \underbrace{N(3\lceil \log_{2^{\lambda+1}} N\rceil + 1)\cdot U(1)}_{\text{calls to GenFermatMul}} + \underbrace{O(N\log N \cdot 2^\lambda)}_{\text{linear operations}} + \underbrace{\mathcal{P}(n)}_{\text{precomputations}} \tag{6}$$

where $\mathcal{P}(n)$ denotes the cost for the precomputation of the lists $\mathcal{L}$ and $\mathcal{G}_i$.

Now, one can express the complexity U depending on the complexity U_ω of the multiplication by an element of $\mathcal{G}_i$ like this:

$$U(i) \leq \underbrace{N_i(3\lceil \log_{2^{\lambda_{i+1}+1}} N_i\rceil)(U_\omega(i+1) + \underbrace{c2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right)}_{\text{modulo operations}}) + 4N_i(S(n_{i+1}) + \underbrace{c2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right)}_{\text{modulo operations}})}_{\text{calls to GenFermatMul} \rightarrow \text{DFT of roots of unity precomputed}} + \underbrace{\text{componentwise multiplications and } 3N_i \text{ multiplications due to Half-FFT}}$$

$$\underbrace{c' N_i \lambda_{i+1} S(n_{i+1})}_{\substack{\text{decomposition in radix } r_i \\ \text{and recomposition}}} + \underbrace{c'' n_i \log n_i}_{\text{linear operations}}$$

where c is a constant given by the modulo operation after each multiplication in radix r_i and c' is a constant associated to the decomposition and recomposition in radix r_i (line 9 and 13 of GenFermatMul). The constant c'' corresponds to the cost of linear operations such as negacyclic shifts in radix r_i, addition, subtractions, etc. One notices that the complexity estimate S is used for the modulo operation in radix r_i or for the decomposition in radix r_i: a similar injection has been used in [16] in order to simplify the complexity analysis.

The complexity U_ω can be expressed in a similar way

$$\begin{aligned} U_\omega(i) \leq & N_i(2\lceil \log_{2^{\lambda_{i+1}+1}} N_i\rceil)(U_\omega(i+1) + c2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right)) + 4N_i(S(n_{i+1}) + c2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right)) + \\ & c' N_i \lambda_1 S(n_{i+1}) + c'' n_i \log n_i \end{aligned} \tag{7}$$

but the factor 3 of the multiplications involved during the FFT becomes a factor 2.

Theoretically, we might already compute the binary complexity of our algorithm, assuming that we use Kronecker substitution to multiply elements of $\mathbf{R}_i$. But then, the analysis would be suboptimal, since this substitution involves some zero-padding and thus a useless increase of the size of the coefficients. We would have $O(n \log n \cdot 8^{\log^* n})$ for the binary complexity of integer multiplication.

The next part introduces another strategy, which borrows from the Schönhage-Strassen algorithm and which is very similar to the technique speeding up the multiplication of r-adic numbers used in [24], avoiding the padding due to the Kronecker substitution. The Kronecker substitution, however, is retained for the last level of recursion corresponding to $i = \mathcal{I}$.

5.5. **Avoiding the Kronecker Substitution.** Instead of embedding an element of $\mathbf{R}_i = \mathbb{Z}/\left(r_i^{2^{\lambda_i}} + 1\right)\mathbb{Z}$ in radix-r_i representation into an integer, it might be profitable to stay in radix-r representation and to consider the same element in radix-$r_i^{\beta_{i+1}}$ representation, for some β_{i+1}

dividing 2^{λ_i}. In other terms, some coefficients may be grouped together. We may then perform the multiplication in $\mathbf{R}_i$ via the multiplication of two polynomials modulo $X^{2^{\lambda_i}/\beta_{i+1}}+1$, as it is done in the Schönhage-Strassen algorithm, using the half-DFT detailed in 2.2. Therefore, for computing the product of two polynomials of degree $2^{\lambda_i}/\beta_{i+1}$ modulo $X^{2^{\lambda_i}/\beta_{i+1}}+1$, a ring $\mathbf{R}_{i+1}$ containing a $2\cdot 2^{\lambda_i}/\beta_{i+1}$-th principal root of unity ω_{i+1} is needed and $3\cdot 2^{\lambda_i}/\beta_{i+1}$ additional multiplications due to the half-DFT in $\mathbf{R}_{i+1}$ have to be performed: one composition with $\omega_{i+1}X$ per polynomial given in the input and one composition with $\omega_{i+1}^{-1}X$ after the final inverse DFT.

Grouping coefficients together is only possible in the case of Hypothesis 18, because, otherwise, the integer r_i would not be small enough. Thus, in the following, we work under this hypothesis. We have to use the same sequence $\gamma(\lambda)$.

Proposition 22. *Let $r^\lambda+1$ be a prime such that $r\in\left[2^{\gamma(\lambda)},2^{2\gamma(\lambda)}(1+\lambda^2)\right]$. Let s be the quantity $\log_2(r^\lambda+1)$ (s is approximatively the bitsize of the prime $r^\lambda+1$). Let λ' be the smallest integer such that $2^{\lambda'}\geq\log_2 s$.*

For $\lambda\geq 6$ there exists $\beta\in\mathbb{N}$ a power of two such that

$$2\beta\log_2 r+\lambda-\log_2\beta\leq 2\gamma(\lambda')2^{\lambda'}. \tag{8}$$

Proof. A sufficient condition for the map $k\to 2\cdot 2^k\cdot\log_2 r+\lambda-k$ to be increasing for $k\geq 0$ is $\log_2 r\geq 1$, which is met for our applications. Thus, it is enough to prove that for λ big enough, $2\log_2 r+\lambda\leq 2\gamma(\lambda')2^{\lambda'}$.

By hypothesis, $2\log_2 r+\lambda\leq 4\gamma(\lambda)+2\log_2(1+\lambda^2)+\lambda$. Assuming Hypothesis 18, $4\gamma(\lambda)\leq 2\lambda\cdot\log_2\lambda-\lambda$, which means that $4\gamma(\lambda)+2\log_2(1+\lambda^2)+\lambda\leq 2\lambda\cdot\log_2\lambda+2\log_2(1+\lambda^2)$.

By definition, $2^{\lambda'}\geq\log_2 s>\log_2^{(2)}(r^\lambda)$. Thus, $2^{\lambda'}>\lambda+\log_2^{(2)}r\geq\lambda+\log_2\gamma(\lambda)$ since $r\geq 2^{\gamma(\lambda)}$ by hypothesis. We deduce from that the following:

$$2^{\lambda'+1}\gamma(\lambda')\geq 2^{\lambda'+1}\lambda'\geq 2(\lambda+\log_2\gamma(\lambda))(\log_2(\lambda+\log_2\gamma(\lambda)))\geq 2(\lambda+\log_2\lambda)(\log_2(\lambda+\log_2\lambda)).$$

Then, it is enough to prove that there exists λ_0 such that, for any $\lambda\geq\lambda_0$, we have

$$2\log_2 r+\lambda\leq 2\lambda\cdot\log_2\lambda+2\log_2(1+\lambda^2)\leq 2(\lambda+\log_2\lambda)(\log_2(\lambda+\log_2\lambda))\leq 2^{\lambda'+1}\gamma(\lambda').$$

Thus, we prove that

$$0\leq\underbrace{2(\lambda+\log_2\lambda)(\log_2(\lambda+\log_2\lambda))}_{E}-\underbrace{(2\lambda\cdot\log_2\lambda+2\log_2(1+\lambda^2))}_{F}.$$

We have

$$E=2\lambda\log_2\lambda+2\lambda\log_2\left(1+\frac{\log_2\lambda}{\lambda}\right)+2\log_2\lambda\cdot(\log_2(\lambda+\log_2\lambda))$$

and

$$F=2\lambda\cdot\log_2\lambda+4\log_2\lambda+2\log_2\left(1+\frac{1}{\lambda^2}\right)\leq 2\lambda\cdot\log_2\lambda+4\log_2\lambda+2.$$

Thus,

$$E-F\geq 2\log_2\lambda\cdot(\log_2(\lambda+\log_2\lambda))-4\log_2\lambda-2\geq 2(\log_2\lambda)^2-4\log_2\lambda-2.$$

For $\lambda\geq\lambda_0=6$, the inequality (8) holds, since $2(\log_2\lambda)^2-4\log_2\lambda-2\geq 0$. □

Definition 23 redefines the sequence (λ_i) in the context of the current strategy to multiply elements of $\mathbf{R}_i$.

Definition 23 (How to choose λ_{i+1} knowing λ_i)**.** *Let p_1 be the prime used on the recursion level $i=1$ of GenFermatMul. Then $r_1^{2^{\lambda_1}}+1=p_1$. Let $\mathcal{I}$ be the number of recursion levels.*

Then, for $1\leq i\leq\mathcal{I}$, $2^{\lambda_{i+1}}$ is the smallest power of two above $\log_2^{(2)}p_i$ and $r_{i+1}^{2^{\lambda_{i+1}}}+1$ is the prime used on the i-th level of recursion.

The redefinition of the sequence of (λ_i) for a given i corresponds to the definition of λ' in Proposition 22 for $r^{2^\lambda}+1=p_i$. Thus, given an $i\geq 1$, there exists β such that $2\beta\log_2 r_i+\lambda_i-\log_2\beta\leq 2\gamma(\lambda_{i+1})\cdot 2^{\lambda_{i+1}}$. For our purpose, we will choose the largest power of two β_{i+1} such that $2\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}\leq 2\gamma(\lambda_{i+1})2^{\lambda_{i+1}}$. Thus, this power of two verifies by definition:

$$4\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}-1>2\gamma(\lambda_{i+1})2^{\lambda_{i+1}}.$$

Let us name R_{i+1} the factor such that $\log_2 R_{i+1}=\frac{2\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}}{2^{\lambda_{i+1}}}$. For our purpose, we need to find a prime p_{i+1} such that $\log_2 p_{i+1}\geq 2\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}$. By definition of β_{i+1}, p_{i+1} would be sufficiently large to contain the result of a multiplication of two elements of $\mathbf{R}_i$ for which the coefficients have been grouped in chunks of size β_{i+1}. In order to use Hypothesis 18, one has to prove that $2^{\gamma(\lambda_{i+1})}\leq R_{i+1}\leq 2^{2\gamma(\lambda_{i+1})}$.

Proposition 24. *Let $i\geq 1$ and R_{i+1} be the number defined as $\log_2 R_{i+1}=\frac{2\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}}{2^{\lambda_{i+1}}}$. Then*

$$2^{\lambda_{i+1}}\leq R_{i+1}\leq 2^{2\lambda_{i+1}}.$$

Proof. By construction of β_{i+1}, $2^{\lambda_{i+1}}\log_2 R_{i+1}\leq 2\gamma(\lambda_{i+1})2^{\lambda_{i+1}}$, which means that $\log_2 R_{i+1}\leq 2\gamma(\lambda_{i+1})$ and $R_{i+1}\leq 2^{2\gamma(\lambda_{i+1})}$.

Now, let us use the fact that

$$4\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}-1>2\gamma(\lambda_{i+1})2^{\lambda_{i+1}}.$$

Using the rewriting

$$2\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}=2\beta_{i+1}\log_2 r_i+\frac{1}{2}(\lambda_i-\log_2\beta_{i+1})+\frac{1}{2}(\lambda_i-\log_2\beta_{i+1})+\frac{1}{2}-\frac{1}{2},$$

one gets:

$$2\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}>\gamma(\lambda_{i+1})2^{\lambda_{i+1}}+\frac{1}{2}(\lambda_i-\log_2\beta_{i+1})+\frac{1}{2}>\gamma(\lambda_{i+1})2^{\lambda_{i+1}}.$$

Thus, $P_{i+1}\geq 2^{\lambda_{i+1}}$. □

Proposition 24 allows one to find a prime p_{i+1} not too large compared to the chunks of β_{i+1} coefficients of elements of $\mathbf{R}_i$ in terms of bitsize. Thus, instead of performing the multiplication of two integers through Kronecker substitution, we multiply two polynomials modulo $X^{2^{\lambda_i}/\beta_{i+1}}+1$ over the ring $\mathbf{R}_{i+1}=\mathbb{Z}/p_{i+1}\mathbb{Z}$.

Definition 25 (Definition of p_{i+1})**.** *Given a prime $p_i=r_i^{2^{\lambda_i}}+1$ with $\lambda_i\geq 4$, let β_{i+1} be the largest power of two such that*

$$2\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}\leq 2\gamma(\lambda_{i+1})2^{\lambda_{i+1}}.$$

Let $R_{i+1}=\frac{2\beta_{i+1}\log_2 r_i+\lambda_i-\log_2\beta_{i+1}}{2^{\lambda_{i+1}}}$. Then p_{i+1} is the smallest prime given by Hypothesis 18 taking $R=R_{i+1}$.

Definition 26 (Definition of n_i)**.** *For any $i\geq 0$, n_i denotes the quantity*

$$n_i=\log_2 p_i.$$

In order to be able to use the half-DFT, one needs to verify that there exists in $\mathbf{R}_{i+1}$ a $2N_i$-th principal root of unity, where $N_i=2^{\lambda_i}/\beta_{i+1}$. Therefore, $2N_i$ must divide $p_{i+1}-1$.

Proposition 27. *If $\lambda_i\geq 2$ then $2N_i$ divides $p_{i+1}-1$.*

Proof. Since $p_{i+1} = r_{i+1}^{2^{\lambda_{i+1}}} + 1$ is prime, r_{i+1} is necessarily even (except if $p_{i+1} = 2$). A sufficient condition for $2N_i$ dividing p_{i+1} is $2N_i$ dividing $2^{2^{\lambda_{i+1}}}$. Since $2^{\lambda_{i+1}} \geq \log_2^{(2)} p_i > \lambda_i + \log_2^{(2)} r_i$, if

$$\log_2(2N_i) = \lambda_i + 1 - \log_2 \beta_{i+1} \leq \lambda_i + \log_2^{(2)} r_i$$

then $2N_{i+1}$ divides $2^{2^{\lambda_{i+1}}}$. Therefore, it would be enough to prove that $\log_2^{(2)} r_i - 1 \geq 0$. Since $r_i \geq 2^{\lambda_i}$, this is true for $\log_2 \lambda_i \geq 1$ and thus $\lambda_i \geq 2$. □

Algorithm 10 is a rewriting of Algorithm GenFermatMul taking into account this new strategy. In particular, the input is not anymore integers: it is polynomials in $\mathbb{Z}[T]/(T^{2^{\lambda_i}} + 1)$.

Algorithm 10 New version of GenFermatMul

Input: A, B two polynomials of degree 2^{λ_i} representing elements of $\mathbb{Z}/(r_i^{2^{\lambda_i}} + 1)\mathbb{Z}$
Output: $C = A \times B \mod T^{2^{\lambda_i}} + 1$

function GenFermatMul(A,B,i,$\mathcal{L}$,$\mathcal{G}$)
  if $i < \mathcal{I}$ **then**
    return $A * B$ ▷ Basecase multiplication
  else
    A' is the polynomial obtained from A by grouping β_{i+1} coefficients
    B' is the polynomial obtained from B by grouping β_{i+1} coefficients
    A'' is the polynomial $A'(\omega_{i+1} \cdot X)$ ▷ Due to Half-FFT
    B'' is the polynomial $B'(\omega_{i+1} \cdot X)$ ▷ Due to Half-FFT
    Decompose the coefficients of A'' and B'' in radix r_{i+1}
    $P \leftarrow$ NewLargeRadixFFT(A'', i) ▷ Algorithm 9 called
    $Q \leftarrow$ NewLargeRadixFFT(B'', i) ▷ Algorithm 9 called
    $R \leftarrow$ ComponentwiseProduct(P, Q)
    $S \leftarrow$ NewLargeRadixInverseFFT(R, i) ▷ Algorithm 9 called
    S' is the polynomial $S(\omega_{i+1}^{-1} \cdot X)$ ▷ Due to Half-FFT
    Change the representation of coefficients of S' from radix r_{i+1} to radix r_i
    return S'
  end if
end function

6. A sharper complexity

This section establishes the bound announced in the introduction, reusing the notations introduced in Section 5. First of all, we prove that the quantity $\mathcal{P}(n)$ in Equation (6) is negligible compared to $n \log n$. Secondly, we cut Equation (7) such that negligible parts are identified. Finally, using an induction argument, we get the complexity of GenFermatMul and, thus, a bound on the cost $M(n)$.

Using the bound $s(2^\lambda)$ defined in Section 5 and Proposition 10, we prove the following result.

Proposition 28. *There exists $K \in \mathbb{Z}$ such that $\mathcal{I} = \log_2^* n + K$ for any n.*

Proof. Let us consider the function

$$\Phi : n \to s(2 \log_2 n).$$

Without loss of generality, since Hypothesis 20 is the worst case, we can assume that $\gamma(\lambda) = 2^{\lambda + o(\lambda)}$. Then,

$$\Phi(n) = 2 \log_2 n \cdot (2 \log_2 n + o(\log_2 n)) = 4 \log_2^2 n \cdot (1 + o(1)).$$

We have $\Phi(2^{2^x}) = 4 \cdot 2^{2x} \cdot (1 + O(1))$ and

$$\log_2 \log_2(\Phi(2^{2^x})) = \log_2 x + O(1).$$

In conclusion, Φ is a logarithmically slow function and, therefore, using Proposition 10, $\Phi^*(n) = \log_2^* n + O(1)$, which allows one to conclude. □

6.1. **Precomputations are negligible.** We prove in this part that the precomputations in Equation (6) are negligible, and we do not need to choose from the hypothesis 18, 19 or 20 to get this result. Those precomputations consist in the following computations.

- Given an input of bitsize n, compute the sequence of generalized Fermat primes $r_i^{2^{\lambda_i}} + 1$ for $1 \leq i \leq \mathcal{I}$ that will be used by GenFermatMul.
- Compute generators in $\mathbf{R}_i = \mathbb{Z}/p_i\mathbb{Z}$.
- Compute $2N_{i-1}$-th principal root of unity ω_i such that $\omega_i^{N_{i-1}/2^{\lambda_i}} = r_i$.
- Compute the DFT of the roots of unity.

In §3.3, we discussed how to compute the primes for a bitsize $n' = O(\log n)^2$. Those primes p verify $p = o(n)$. We compute all primes below n using Eratosthenes sieve in $O(n \log^{(2)} n)$, which is negligible compared to $n \log n$.

We give now the different arguments proving that precomputations are negligible. All these arguments rely mainly on the fact that the primes involved in the precomputations are exponenantially smaller than the input size n given to our algorithm multiplying two integers. This fact allows one to use naive algorithms like schoolbook multiplication in the complexity estimates, and simplifies the analysis.

At first, we should note that $n_0 = O(\log n)^2$ where n is the bitsize of the integers that we multiply and n_0 is the bitsize of the input given at the first call to GenFermatMul. Thus, for all $i \geq 1$, λ_i verifies

$$\lambda_i \leq 2 \log_2^{(2)} n_0 = 2 \log_2^{(3)} n + O(1).$$

Each λ_i verifies $2^{\lambda_i} = O(\log_2^{(2)} n)$. As usual we consider the worst case given by Hypothesis 20. Thus, finding primes takes less than $2^{2 \cdot \gamma(\lambda_i)}(1 + \lambda_i^2) = 2^{2^{\lambda_i + o(\lambda_i)}} = 2^{2^{2\lambda_i}}$ tests of primality (we check the primality with the array given by the Eratosthenes sieve). Each test is an exponentiation of an integer of less than $3 \cdot 2^{\lambda_i + o(\lambda_i)}$ bits to the power 2^{λ_i}: we roughly approximate this to $2 \log_2^{(3)} n + O(1)$ multiplications of $O(\log_2^{(2)} n)^2$-bit integers. Since there are approximatively $\log_2^* n$ primes to find, the whole complexity estimate is $2^{O(\log_2^{(2)} n)^2} \log_2^* n \log_2^{(3)} n \cdot M(O(\log_2^{(2)} n)^2) = o(n)$.

The computation of a generator g_i' of $\mathbb{Z}/p_i\mathbb{Z}$ can be done with a deterministic algorithm [22] in $O(p_i^{\frac{1}{4}+\epsilon})$ for any $\epsilon > 0$ and any i. Once one gets this generator g_i', we raise it to the power $\frac{p_i - 1}{2 \cdot N_i}$ in order to get a $2N_{i-1}$-th principal root of $\mathbb{Z}/p_i\mathbb{Z}$ that we call g_i: the cost of this operation is less than $O(\log_2 p_i)$ multiplications in $\mathbb{Z}/p_i\mathbb{Z}$ using fast exponentiation. Since $p_i \leq 2^{s(2 \log_2 n)} \leq 2^{O(\log_2^{(2)} n)^2}$, the cost of an algorithm computing g_i knowing p_i is negligible compared to $n \log n$.

For all p_i, we need now to find a principal root of unity such that raised at the power $N_{i-1}/2^{\lambda_i}$ it is equal to r_i. For a specific p_i, $g_i^{N_{i-1}/2^{\lambda_i}}$ is a $2^{\lambda_i + 1}$-th principal root of unity, thus it is included in $\left\{ r_i^{2j+1} \mid j \in \left[0, 2^{\lambda_i} - 1\right] \right\}$. Let us say that $g_i^{N_{i-1}/2^{\lambda_i}} = r_i^{2j_0+1}$. The quantity $2j_0+1$ is invertible modulo 2^{λ_i+1}, which means that there exists k such that

$$g_i^{kN_{i-1}/2^{\lambda_i}} = r_i^{k \cdot (2j_0+1)} = r_i.$$

Thus, we need to compute $g_i^{N_{i-1}/2^{\lambda_i}}$, which corresponds to the exponentiation of an integer modulo p_i and costs $O(\log_2 p_i)^3 = O(\log_2 n)^3$. Computing the 2^{λ_i} powers of r_i costs $\lambda_i O(\log_2^{(3)} n)$ multiplications modulo p_i, and their complexity is therefore equal to $O(\log_2^{(3)} n(\log_2 n)^2)$. Finding j_0 can be done in $2^{\lambda_i} = O(\log_2^{(2)} n)$ comparisons modulo p_i: $O(\log_2^{(2)} n \cdot \log_2 n)$. Finding k can be done using the extended Euclidean algorithm: $O(\lambda_i M(\lambda_i)) = O(\lambda_i)^3$. In conclusion, computing our principal root of unity ω_i modulo p_i requires $o(n)$ operations. The cost for all p_i will be $o(n \log_2^* n)$ then.

Given $i \geq 1$, we need to compute all the powers of ω_i: the cost is p_i multiplications modulo p_i and is thus

$$O(p_i(\log_2 p_i)^2) = O(2^{O(\log_2 \log_2 n)^2}(\log_2 \log_2 n)^4) = o(n).$$

For each of these powers ω_i^j we compute their representation in radix r_i, which costs

$$p_i \cdot O(\lambda_i M(\log_2 p_i)) = o(n),$$

and a DFT modulo p_{i+1}, which costs $N_i \log_2 N_i M(p_{i+1})$ bit operations and since there at most p_i powers and $N_i \leq 2^{\lambda_i}$ by construction of N_i, the cost is equal to

$$\underbrace{p_i M(p_{i+1})}_{o(n)} \underbrace{2^{\lambda_i}\lambda_i}_{=O(\log_2^{(2)} n \cdot \log_2^{(3)} n)} = o(n \log_2^{(2)} n \log_2^{(3)} n).$$

The global complexity estimate is then equal to $o(n \log_2^{(2)} n \log_2^{(3)} n \log_2^* n)$ which is negligible. In conclusion, $\mathcal{P}(n) = o(n \log n)$.

6.2. **Merging some terms in the recursive formula.** In the following, we assume Hypothesis 18. However, we have a similar analysis under Hypothesis 19 or 20. The only change appears in the constants.

Let us come back to Equation (7):

$$\begin{aligned} U_\omega(i) \leq & N_i(2\lceil \log_{2^{\lambda_{i+1}+1}} N_i \rceil)(U_\omega(i+1) + c2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right)) + 4N_i(S(n_{i+1}) + c2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right)) + \\ & c' N_i \lambda_{i+1} S(n_{i+1}) + c'' n_i \log n_i. \end{aligned}$$

We prove in this part that there exists a constant d_0 such that

$$\begin{aligned} & cN_i(2\lceil \log_{2^{\lambda_{i+1}+1}} N_i \rceil)2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right) + 4N_i(S(n_{i+1}) + \\ & c2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right)) + c' N_i \lambda_{i+1} S(n_{i+1}) + c'' n_i \log n_i \leq d_0 n_i \log_2 n_i. \end{aligned}$$

The two following lemmas prove useful bounds on the size of p_i and r_i.

Lemma 29. *For any $i \geq 1$ such that $\lambda_{i+1} \geq 4$,*

$$\log_2 p_{i+1} \leq 2\beta_{i+1} \log_2 r_i + \lambda_i - \log_2 \beta_{i+1} + 2^{\lambda_{i+1}+1} \log_2 \lambda_{i+1} \leq 2^{\lambda_{i+1}+1}\gamma(\lambda_{i+1}) + 2^{\lambda_{i+1}+2} \log_2 \lambda_{i+1}.$$

Proof. By construction, $p_{i+1} \leq (R_{i+1}(1+\lambda_{i+1}^2))^{2^{\lambda_{i+1}}} + 1$. Let us consider $\log_2 p_{i+1}$. We have the following bound for $\lambda_{i+1} \geq 4$:

$$2^{\lambda_{i+1}}(\log_2(1+\lambda_{i+1}^2)) + 1 \leq 2^{\lambda_{i+1}+2} \log_2 \lambda_{i+1}.$$

Remembering that $\log_2 R_{i+1} = \frac{2\beta_{i+1} \log_2 r_i + \lambda_i - \log_2 \beta_{i+1}}{2^{\lambda_{i+1}}}$, we get the announced result. □

Lemma 30. *For any $i \geq i$, $5\beta_{i+1} \log_2 r_i \geq 2^{\lambda_{i+1}+1}\gamma(\lambda_{i+1})$.*

Proof. By definition of β_{i+1}, $4\beta_{i+1}\log_2 r_i + \lambda_i - \log_2 \beta_{i+1} - 1 > 2^{\lambda_{i+1}+1}\gamma(\lambda_{i+1})$. By construction of r_i, $\log_2 r_i \geq \lambda_i$ and, thus,

$$(4\beta_{i+1}+1)\log_2 r_i - \log_2 \beta_{i+1} - 1 > 2^{\lambda_{i+1}+1}\gamma(\lambda_{i+1}).$$

Since, $5\beta_{i+1}\log_2 r_i \geq (4\beta_{i+1}+1)\log_2 r_i - \log_2 \beta_{i+1} - 1$, we can conclude. □

Lemma 31. *There exists a constant $d_1 > 0$ such that for any $i > 0$*

$$N_i(2\lceil \log_{2^{\lambda_{i+1}+1}} N_i\rceil)2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right) \leq d_1 n_i \log_2 n_i.$$

Proof. By definition of N_i, $N_i = \frac{2^{\lambda_i}}{\beta_{i+1}}$ and thus

$$N_i = \frac{2^{\lambda_i}\log_2 r_i}{\beta_{i+1}\log_2 r_i} \leq \frac{n_i}{\beta_{i+1}\log_2 r_i}.$$

One needs to estimate now the quantity $\lceil \log_{2^{\lambda_{i+1}+1}} N_i\rceil$. We can clearly get the following upper bound:

$$\lceil \log_{2^{\lambda_{i+1}+1}} N_i\rceil \leq \log_{2^{\lambda_{i+1}+1}} N_i + 1.$$

Reusing the definition of N_i, we get

$$\log_{2^{\lambda_{i+1}+1}} N_i + 1 \leq \frac{\lambda_i}{\lambda_{i+1}+1} + 1 \leq \frac{\lambda_i}{\lambda_{i+1}} + 1 \leq 2\frac{\lambda_i}{\lambda_{i+1}}.$$

It remains to estimate $S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right) = S\left(\log_2 r_{i+1}\right)$: there exists a constant d such that $S(n) \leq dn\log_2 n \log_2\log_2 n$ for any n. Then,

$$S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right) \leq d\frac{n_{i+1}}{2^{\lambda_{i+1}}}(\log_2^{(2)} r_{i+1})\log_2^{(3)} r_{i+1}.$$

Remembering that $\log_2 r_{i+1} \leq 2\gamma(\lambda_{i+1}) + \log_2(1+\lambda_{i+1}^2)$ by the bound of Hypothesis 18, we can state the rough upper bound $\log_2 r_{i+1} \leq 3\gamma(\lambda_{i+1}) \leq \frac{3}{2}\lambda_{i+1}\log_2\lambda_{i+1} < 2\lambda_{i+1}\log_2\lambda_{i+1}$.

Combining those upper bounds and Lemma 30, we get

$$N_i(2\lceil \log_{2^{\lambda_{i+1}+1}} N_i\rceil)2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right) \leq 5\frac{n_i}{\gamma(\lambda_{i+1})2^{\lambda_{i+1}+1}}\left(4\cdot\frac{\lambda_i}{\lambda_{i+1}}\right) dn_{i+1}(2\log_2\lambda_{i+1}+2)\log_2(2\log_2\lambda_{i+1}+2).$$

Lemma 29 gives an upper bound on n_{i+1} which can be estimated to

$$2^{\lambda_{i+1}+1}\gamma(\lambda_{i+1}) + 2^{\lambda_{i+1}+2}\log_2\lambda_{i+1} \leq 2^{\lambda_{i+1}+2}\gamma(\lambda_{i+1}).$$

Thus, one gets

$$N_i(2\lceil \log_{2^{\lambda_{i+1}+1}} N_i\rceil)2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right) \leq 5n_i\left(4\frac{\lambda_i}{\lambda_{i+1}}\right) 2d(2\log_2\lambda_{i+1}+1)\log_2(2\log_2\lambda_{i+1}+1).$$

Since $\frac{(2\log_2 x+1)\log_2(2\log_2 x+1)}{x} = o(1)$ when $x \to \infty$ and $\lambda_i \leq \log_2 n_i$, one can conclude. □

Lemma 32. *There exists $d_2 > 0$ such that for any $i > 0$*

$$N_i\lambda_{i+1}S(n_{i+1}) \leq d_2 n_i \log_2 n_i.$$

Proof. Reusing the bounds of the proof of Lemma 31, one gets

$$N_i\lambda_{i+1}S(n_{i+1}) \leq 5n_i\lambda_{i+1}2d\log_2 n_{i+1}\log_2\log_2 n_{i+1}.$$

By definition, $2^{\lambda_{i+1}} \approx \log_2 n_i$ and n_{i+1} is bounded by $2^{\lambda_{i+1}+2}\lambda_{i+1}$. In conclusion,

$$\lambda_{i+1}\log_2 n_{i+1}\log_2\log_2 n_{i+1} = o(\log_2 n_i),$$

from which we deduce the result of Lemma. □

Since $N_i S(n_{i+1}) \leq N_i \lambda_{i+1} S(n_{i+1})$ and

$$N_i 2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right) \leq N_i (2\lceil \log_{2^{\lambda_{i+1}+1}} N_i \rceil) 2^{\lambda_{i+1}} S\left(\frac{n_{i+1}}{2^{\lambda_{i+1}}}\right),$$

Lemma 31 and 32 prove the existence of the announced constant d_0 and we can state Proposition 33.

Proposition 33. *There exists d_0 such that for any $i > 0$*

$$U_\omega(i) \leq N_i (2\lceil \log_{2^{\lambda_{i+1}+1}} N_i \rceil) U_\omega(i+1) + d_0 n_i \log_2 n_i. \tag{9}$$

6.3. **Inductive argument.** Coming back to Equation (9), it is possible to prove the announced complexity estimate for $U_\omega(i)$ through an inductive argument similar to the one used by Fürer in [11].

Theorem 34. *There exist $L > 0$ and $1 > d' > 0$ such that for any $i > 0$*

$$U_\omega(i) \leq L n_i \log_2 n_i (4^{\mathcal{I}-i} - d'),$$

$\mathcal{I}$ being the number of recursive levels.

Proof. The constants L and d' will be determined later. At first, let us assume that the theorem is true for $i+1$ and let us prove it for i. It is thus possible to inject the formula of $U_\omega(i+1)$ into $U_\omega(i)$:

$$U_\omega(i) \leq N_i (2\lceil \log_{2^{\lambda_{i+1}+1}} N_i \rceil) L n_{i+1} \log_2 n_{i+1} (4^{\mathcal{I}-i-1} - d') + d_0 n_i \log_2 n_i.$$

Let us rewrite N_i:

$$N_i = \frac{n_i}{\beta_{i+1} \log_2 r_i}.$$

Moreover, according to Lemma 29,

$$n_{i+1} \leq 2\beta_{i+1} \log_2 r_i + \lambda_i - \log_2 \beta_{i+1} + 2^{\lambda_{i+1}+2} \log_2 \lambda_{i+1}.$$

For $\log_2 n_{i+1}$, we use the following bound:

$$\log_2 n_{i+1} \leq \log_2 (2^{\lambda_{i+1}+1} \gamma(\lambda_{i+1}) + 2^{\lambda_{i+1}+2} \log_2 \lambda_{i+1}) \leq \lambda_{i+1} + 1 + \log_2 \gamma(\lambda_{i+1}) + \log_2 \left(1 + 2\frac{\log_2 \lambda_{i+1}}{\gamma(\lambda_{i+1})}\right).$$

We already proved that

$$\lceil \log_{2^{\lambda_{i+1}+1}} N_i \rceil \leq \frac{\lambda_i}{\lambda_{i+1}} + 1.$$

Thus, we can rewrite the formula $U_\omega(i)$

$$\begin{aligned}
U_\omega(i) \leq & L \frac{n_i}{\beta_{i+1} \log_2 r_i} \cdot 2 \left(\frac{\lambda_i}{\lambda_{i+1}} + 1\right) \left(2\beta_{i+1} \log_2 r_i + \lambda_i - \log_2 \beta_{i+1} + 2^{\lambda_{i+1}+1} \log_2 \lambda_{i+1}\right) \\
& \left(\lambda_{i+1} + 1 + \log_2 \gamma(\lambda_{i+1}) + \log_2 \left(1 + 2\frac{\log_2 \lambda_{i+1}}{\gamma(\lambda_{i+1})}\right)\right) (4^{\mathcal{I}-i-1} - d') + d_0 n_i \log_2 n_i \\
= & 2L \frac{n_i}{\beta_{i+1} \log_2 r_i} \left(\frac{\lambda_i}{\lambda_{i+1}} + 1\right) \left(2\beta_{i+1} \log_2 r_i + O(2^{\lambda_{i+1}+1} \log_2 \lambda_{i+1})\right) (\lambda_{i+1} + O(\log_2 \lambda_{i+1})) \\
& (4^{\mathcal{I}-i-1} - d') + d_0 n_i \log_2 n_i \\
= & 2L n_i \left(\frac{\lambda_i}{\lambda_{i+1}} + 1\right) \left(2 + O\left(\frac{2^{\lambda_{i+1}+1} \log_2 \lambda_{i+1}}{\beta_{i+1} \log_2 r_i}\right)\right) (\lambda_{i+1} + O(\log_2 \lambda_{i+1})) (4^{\mathcal{I}-i-1} - d') \\
& + d_0 n_i \log_2 n_i.
\end{aligned}$$

Since $p_i = r^{2^{\lambda_i}} + 1$, $\lambda_i \leq \log_2 n_i \leq 2^{\lambda_{i+1}}$. Thus, combining this bound on λ_i with the bound of Lemma 30 and the fact that there exists $K \in \mathbb{Z}$ such that $\mathcal{I} = \log_2^* n_i + K$ according to Proposition 28, one can neglect most of the terms of the previous equation, and proves the existence of a constant $d_4 > 0$ such that

$$U_\omega(i) \leq Ln_i\lambda_i(4^{\mathcal{I}-i} - 4d') + d_4 n_i \log_2 n_i + d_0 n_i \log_2 n_i \leq Ln_i \log_2 n_i(4^{\mathcal{I}-i} - 4d') + d_4 n_i \log_2 n_i + d_0 n_i \log_2 n_i.$$

This is due to the fact that $5\beta_{i+1} \log_2 r_i > 2^{\lambda_{i+1}+1}\lambda_{i+1}$ and that

$$\frac{\lambda_i}{\lambda_{i+1}} \cdot \frac{2^{\lambda_{i+1}+1} \log_2 \lambda_{i+1}}{\beta_{i+1} \log_2 r_i} \cdot \lambda_{i+1} \cdot 4^{\log_2^* n_i} \leq 5\frac{\lambda_i}{\lambda_{i+1}} \log_2 \lambda_{i+1} 4^{\log_2^* n_i} = o(\lambda_i \cdot \frac{(\log_2 \lambda_{i+1})^2}{\lambda_{i+1}}) = o(\log_2 n_i).$$

Choosing d' and L such that $\frac{d_4+d_0}{L} \leq 3d'$ and $U_\omega(\mathcal{I})$ is greater than the cost at the deeper level, we prove recursively the result of the theorem.

□

Remark 35. *Assuming Hypothesis 18, at the depth $i+1$, the initial coefficients of the polynomials given to the FFT have a bitsize equal to $\eta_{i+1} = \beta_{i+1} \log r_i$. We reach $4^{\log^* n}$ in the complexity analysis thanks to the fact that $n_{i+1}/\eta_{i+1} = 2 + o(1)$.*

If Hypothesis 18 does not hold and Hypothesis 19 holds, we would have to use Kronecker substitution, which would give $n_{i+1}/\eta_{i+1} = 4 + o(1)$ since the zero-padding doubles the size of the ring $\mathbf{R_{i+1}}$. This explains in particular why we get $8^{\log^* n}$.

If Hypothesis 20 holds, then

$$\log_2 p_{i+1} = 2_{i+1}^{\lambda} \cdot 2^{\lambda_{i+1}+o(\lambda_{i+1})}$$

and

$$\log_2^{(2)} p_{i+1} = 2\lambda_{i+1} + o(\lambda_{i+1}).$$

Thus, we get another factor 2 due to $\log_2^{(2)} p_{i+1} = \log_2 n_{i+1}$ since we would have a factor $2\lambda_{i+1} + O(\log_2 \lambda_{i+1})$ instead of $\lambda_{i+1} + O(\log_2 \lambda_{i+1})$ that contributes to $U_\omega(i)$ in the proof of Theorem 34, which gives $16^{\log^* n}$.

It is not hard to deduce from Theorem 34 and Proposition 28 that, assuming Hypothesis 18,

$$U_\omega(1) = O(n_1 \log_2 n_1 4^{\log_2^* n_1}).$$

Injecting $U_\omega(i+1)$ into $U(i)$, and then $M(n)$, one gets the estimated conditional complexity, which concludes this section.

7. Practical considerations

In practice, the algorithm that has been described in this section should not be implemented following the various tricks that make the complexity analysis work. Firstly, the strategy of using GenFermatMul for the toplevel of the tree of recursive calls is no longer mandatory if one considers the determination of p_0 as a precomputation. In light of Table 2, 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 toplevel, for the same reason explained in [12, §2.3], applied to the Schönhage-Strassen algorithm.

Secondly, the bounds of Hypothesis 18, 19 or 20 are no longer relevant for a competitive implementation. Indeed, Hypothesis 18 is meant to improve the complexity analysis. Thus, we are not forced to use the primes suggested in Section 5.1. However, we retain that given a parameter η fixing the size of coefficients of polynomials A and B of degree N, the bitsize of the generalized Fermat prime p should not be too large compared to $2\eta + \log_2 N$. Thus, in practice, we choose the smallest prime $p = r^{2^\lambda} + 1$ larger than $2\eta + \log_2 N$.

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

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

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

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

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

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

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

Moreover, it is possible to use an intermediate strategy between the Kronecker substitution and the strategy proposed in §5.5. For instance, multiplying elements represented in radix 74 in the ring $\mathbf{R} = \mathbb{Z}/p\mathbb{Z} = \mathbb{Z}/74^{16} + 1\mathbb{Z}$ using Kronecker substitution would require a multiplication of two integers of bitsize greater than $2 \cdot 8 \cdot 16 = 256$. This requires, on a 64-bit architecture, 9 machine-word multiplications using Karastuba on two levels of recursion.

Considering elements of $\mathbf{R}$ in radix 74^2, we can use the multipoint Kronecker substitution proposed by Harvey in [15]. This way, we perform 2 multiplications of 128-bit integers, which corresponds to 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 the new algorithm. Roughly speaking, for 2^{30}-bit integers, a Schönhage-Strassen 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 integers of 288-bit is cheaper than the cost induced by 2^{16} multiplications of size 2^{16}-bit.

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 3 how changing the prime used in GenFermatMul for the multiplication of two n-bit integers, where n equal to 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 [14] for different bitsizes on an architecture Intel Core i5-4590 (3.30GHz and Haswell product). We obtain the expected time spent in expensive multiplications by estimating the bitsize 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 bitsize. This approximation does not take into account the fact that expensive multiplications of integers of bitsize m are done modulo an integer 2^m+1, which may save a factor 2 in practice. We deduce from Table 3 that changing the prime may improve on the cost induced by the expensive multiplications, but not significantly.

bitsize 2^{30}			
prime	expensive multiplications	bitsize 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	$8.61\cdot 10$
$54^{32}+1$	$2^{25}\cdot(3\cdot\lceil\frac{25}{6}\rceil+1)=2^{25}\cdot 16$	$(6\cdot 2+5)\cdot 32=544$	$6.12\cdot 10$
$\mathbf{562^{32}+1}$	$\mathbf{2^{24}\cdot 13}$	$\mathbf{(10\cdot 2+5)\cdot 32=800}$	$\mathbf{3.57\cdot 10}$
$131090^{32}+1$	$2^{23}\cdot 13$	$(18\cdot 2+5)\cdot 32=1312$	$4.82\cdot 10$
bitsize 2^{36}			
prime	expensive multiplications	bitsize 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	$4.26\cdot 10^3$
$54^{32}+1$	$2^{31}\cdot 19$	$(6\cdot 2+5)\cdot 32=544$	$4.61\cdot 10^3$
$\mathbf{562^{32}+1}$	$\mathbf{2^{30}\cdot 16}$	$\mathbf{(10\cdot 2+5)\cdot 32=800}$	$\mathbf{3.35\cdot 10^3}$
$131090^{32}+1$	$2^{29}\cdot 16$	$(18\cdot 2+5)\cdot 32=1312$	$3.75\cdot 10^3$
$102^{64}+1$	$2^{30}\cdot 16$	$(7\cdot 2+6)\cdot 64=1280$	$7.00\cdot 10^3$
$562^{64}+1$	$2^{29}\cdot 16$	$(10\cdot 2+6)\cdot 64=1664$	$5.65\cdot 10^3$
bitsize 2^{40}			
prime	expensive multiplications	bitsize 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	$6.82\cdot 10^4$
$54^{32}+1$	$2^{35}\cdot 19$	$(6\cdot 2+5)\cdot 32=544$	$7.52\cdot 10^4$
$\mathbf{562^{32}+1}$	$\mathbf{2^{34}\cdot 19}$	$\mathbf{(10\cdot 2+5)\cdot 32=800}$	$\mathbf{6.26\cdot 10^4}$
$131090^{32}+1$	$2^{33}\cdot 19$	$(18\cdot 2+5)\cdot 32=1312$	$7.09\cdot 10^4$
$102^{64}+1$	$2^{34}\cdot 16$	$(7\cdot 2+6)\cdot 64=1280$	$11.28\cdot 10^4$
$562^{64}+1$	$2^{33}\cdot 16$	$(10\cdot 2+6)\cdot 64=1664$	$9.03\cdot 10^4$
bitsize 2^{46}			
prime	expensive multiplications	bitsize 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 3. Estimated time for computing the expensive multiplications in GenFermatMul depending on the prime used.

The table 4 gives an estimation of the time required to compute a multiplication of two integers using GenFermatMul 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 tradeoff obtained in Table 3 and the primes proposed in Table 2 by default.

Thus, the table 4 allows one to conclude that an implementation of GenFermatMul will unlikely beat an implementation of Schönhage-Strassen algorithm for sizes below 2^{40}. For sizes above 2^{40}, it seems that Schönhage-Strassen algorithm is not anymore 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 of GenFermatMul.

	Schönhage-Strassen algorithm			GenFermatMul			
bitsize	nb. mult.	mult. bitsize	time (s)	nb. mult.	prime	KS. bitsize	time (s)
2^{30}	2^{16}	$\approx 2^{16}$	9.96	$2^{26} \cdot 19$	$562^{32}+1$	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 4. Comparison of multiplications realized by Schönhage-Strassen algorithm
and GenFermatMul relying on measured times of `mpz_mul` of GMP [14].
The third column in GenFermatMul corresponds to the bitsize of the integers obtained after Kronecker substitution.

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 [16], 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 $4^{\log^* n}$ 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.

We can also question the practicality of our approach. Is it possible to make a competitive implementation of those algorithms which would beat the current implementations of Schönhage-Strassen's algorithm ?

References

[1] L. Adleman and A. Odlyzko. Irreducibility testing and factorization of polynomials. *Math. Comput.*, 41(164):699–709, 1983. doi:10.2307/2007706.

[2] P. T. Bateman and R. A. Horn. A heuristic asymptotic formula concerning the distribution of prime numbers. *Math. Comput.*, 16(79):pp. 363–367, 1962. doi:10.1090/S0025-5718-1962-0148632-7.

[3] L. I. Bluestein. A linear filtering approach to the computation of discrete fourier transform. *Audio and Electroacoustics, IEEE Transactions on*, 18(4):451–455, Dec 1970. doi:10.1109/TAU.1970.1162132.

[4] R. Brent and P. Zimmerman. *Modern computer algebra*. Cambridge University Press, 2011.

[5] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19:297–301, 1965. doi:10.1090/S0025-5718-1965-0178586-1.

[6] A. De, P. P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. In *40th annual ACM symposium on Theory of computing*, STOC '08, pages 499–506, New York, NY, USA, 2008. ACM. doi:10.1145/1374376.1374447.

[7] H. Dubner and Y. Gallot. Distribution of generalized Fermat prime numbers. *Math. Comput.*, 71(238):825–832, 2002. doi:10.1090/S0025-5718-01-01350-3.

[8] P. Elliott. Primes in progressions to moduli with a large power factor. *The Ramanujan Journal*, 13(1-3):241–251, 2007. doi:10.1007/s11139-006-0250-4.

[9] M. Fürer. On the complexity of integer multiplication (extended abstract). Technical Report CS-89-17, Pennsylvania State University, 1989.

[10] M. Fürer. Faster integer multiplication. In D. S. Johnson and U. Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 57–66. ACM, 2007. doi:10.1145/1250790.1250800.

[11] M. Fürer. Faster integer multiplication. *SIAM J. Comput.*, 39(3):979–1005, 2009. doi:10.1137/070711761.

[12] P. Gaudry, A. Kruppa, and P. Zimmermann. A gmp-based implementation of schönhage-strassen's large integer multiplication algorithm. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ISSAC '07, pages 167–174, New York, NY, USA, 2007. ACM. doi:10.1145/1277548.1277572.

[13] D. B. Gillies. Three new Mersenne primes and a statistical theory. *Math. Comput.*, 18(85):93–97, jan 1964. doi:10.1090/S0025-5718-1964-0159774-6.

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

[15] D. Harvey. Faster polynomial multiplication via multipoint kronecker substitution. *Journal of Symbolic Computation*, 44(10):1502 – 1510, 2009. doi:http://dx.doi.org/10.1016/j.jsc.2009.05.004.

[16] D. Harvey, J. van der Hoeven, and G. Lecerf. Even faster integer multiplication. Technical report, ArXiv, 2014. arXiv:1407.3360.

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

[18] K. S. McCurley. Prime values of polynomials and irreducibility testing. *Bull. Amer. Math. Soc. (N.S.)*, 11(1):155–158, 07 1984. doi:10.1090/S0273-0979-1984-15247-9.

[19] J. M. Pollard. The Fast Fourier Transform in a finite field. *Math. Comput.*, 25(114):365–374, apr 1971. doi:10.1090/S0025-5718-1971-0301966-0.

[20] C. Pomerance. On the distribution of amicable numbers. *J. Reine Angew. Math.*, pages 217–222, 1977.

[21] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971. doi:10.1007/BF02242355.

[22] I. Shparlinski. On finding primitive roots in finite fields. *Theor. Comput. Sci.*, 157(2):273–275, May 1996. doi:10.1016/0304-3975(95)00164-6.

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

[24] J. van der Hoeven. Faster relaxed multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 405–412, New York, NY, USA, 2014. ACM. doi:10.1145/2608628.2608657.

[25] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.

[26] S. S. Wagstaff, Jr. Divisors of mersenne numbers. *Math. Comput.*, 40(161):385–397, 1983. doi:10.2307/2007383.

Appendix A. Proof of Proposition 15

Let us 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 the quantity $\lim\limits_{K\to\infty} \dfrac{t(K,\lambda)}{u(K,\lambda)}$ with

$$t(K,\lambda) = \prod_{\substack{k\in[\![1,K]\!] \\ k\cdot 2^{\lambda+1}+1 \text{ prime}}} \left(1 - \frac{2^\lambda}{k\cdot 2^{\lambda+1}+1}\right)$$

and

$$u(K,\lambda)=\prod_{p\text{ prime}}^{K\cdot 2^{\lambda+1}+1}\left(1-\frac{1}{p}\right).$$

The idea is to rely on the proof of the main theorem of [20, §2], and to use the main result of [8] 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.

Let us consider $C_\lambda(x)$:

$$C_\lambda(x)=\frac{\prod\limits_{\substack{p\in\mathcal{P}(x)\\ p\equiv 1 \mod 2^{\lambda+1}}}\left(1-\frac{2^\lambda}{p}\right)}{\prod\limits_{p\in\mathcal{P}(x)}\left(1-\frac{1}{p}\right)}.$$

We prove that there exists $C>0$ such that $C_\lambda\geq C$ for any λ. Thus, let us take a look at the logarithm:

$$-\log C_\lambda(x)=\underbrace{\sum_{p\in\mathcal{P}(x)}\log\left(1-\frac{1}{p}\right)}_{F}-\underbrace{\sum_{\substack{p\in\mathcal{P}(x)\\ p\equiv 1 \mod 2^{\lambda+1}}}\log\left(1-\frac{2^\lambda}{p}\right)}_{G}.$$

By Abel's summation, we have

$$-\log C_\lambda(x)=\underbrace{\sum_{p\in\mathcal{P}(x)}\log\left(1-\frac{1}{p}\right)}_{F}-\underbrace{\sum_{\substack{p\in\mathcal{P}(x)\\ p\equiv 1 \mod 2^{\lambda+1}\\ p\geq 2\cdot 2^{\lambda+1}+1}}\log\left(1-\frac{2^\lambda}{p}\right)}_{G'}-\underbrace{\log\left(1-\frac{2^\lambda}{2^{\lambda+1}+1}\right)\cdot\pi(2^{\lambda+1}+1,2^{\lambda+1},1)}_{\text{First term of } G \text{ if } 2^{\lambda+1}\text{ is prime}}$$

$$=\underbrace{-\gamma-\log^{(2)}x+o(1)}_{F}-\underbrace{\log(1-\frac{2^\lambda}{2^{\lambda+1}+1})\cdot\pi(2^{\lambda+1}+1,2^{\lambda+1},1)}_{\text{First term of } G \text{ if } 2^{\lambda+1}\text{ is prime}}+\log\left(1-\frac{2^\lambda}{2\cdot 2^{\lambda+1}+1}\right)\cdot$$

$$\pi(2\cdot 2^{\lambda+1}+1,2^{\lambda+1},1)-\log\left(1-\frac{2^\lambda}{x}\right)\pi(x,2^{\lambda+1},1)-\int_{2\cdot 2^{\lambda+1}+1}^{x}2^\lambda\frac{\pi(t,2^{\lambda+1},1)}{t^2-2^\lambda t}dt$$

Since $\log(1-\frac{2^\lambda}{x})\pi(x,2^{\lambda+1},1)\sim-\frac{2^\lambda}{x}\pi(x,2^{\lambda+1},1)=o(1)$, we have

$$-\log C_\lambda(x)=\int_{2\cdot 2^{\lambda+1}+1}^{x}2^\lambda\frac{\pi(t,2^{\lambda+1},1)}{t^2-2^\lambda t}dt-\log^{(2)}x-\gamma+O(1). \tag{10}$$

Now, we can use the result of [8]: there exists an absolute constant K such that for any $\lambda\geq 0$, $q=2^{\lambda+1}$, and x such that

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

taking $A=2$ and $B=A+6=8$ in the theorem,

$$\left|\pi(x,2^{\lambda+1},1)-\frac{x}{\phi(2^{\lambda+1})\log x}\right|<\frac{K\cdot x}{\phi(2^{\lambda+1})(\log x)^2}. \tag{11}$$

There exists an absolute constant H such that for any $x>1$

$$\min(x^{1/3}\exp(-(\log^{(2)}x)^3),x^{1/2}\exp(-8\log^{(2)}x))\geq(Hx)^{1/6}.$$

Thus, for $x > \frac{2^{6\cdot(\lambda+1)}}{H}$, the equation (11) is verified.

Coming back to the equality 10, we cut the integral as in [20, §2]:

$$-\log C_\lambda(x) = \int_{2\cdot 2^{\lambda+1}+1}^{2^{6\cdot(\lambda+1)}/H} 2^\lambda \frac{\pi(t,2^{\lambda+1},1)}{t^2-2^\lambda t}dt + \int_{2^{6\cdot(\lambda+1)}/H}^{x} 2^\lambda \frac{\pi(t,2^{\lambda+1},1)}{t^2-2^\lambda t}dt - \log^{(2)} x - \gamma + O(1).$$

We use a Brun-Titchmarsh estimate for the left integral:

$$\int_{2\cdot 2^{\lambda+1}+1}^{2^{6\cdot(\lambda+1)}/H} 2^\lambda \frac{\pi(t,2^{\lambda+1},1)}{t^2-2^\lambda t}dt < \int_{2\cdot 2^{\lambda+1}+1}^{2^{6\cdot(\lambda+1)}/H} \frac{2}{(t-2^\lambda)\log(t/2^{\lambda+1})}dt.$$

Let us use a change of variable $t = u\cdot 2^{\lambda+1}$. Then, $dt = 2^{\lambda+1}du$ and

$$\int_{2\cdot 2^{\lambda+1}+1}^{2^{6\cdot(\lambda+1)}/H} \frac{2}{(t-2^\lambda)\log(t/2^{\lambda+1})}dt < \int_{2}^{2^{5\cdot(\lambda+1)}/H} \frac{2^{\lambda+1}}{(u\cdot 2^{\lambda+1}-2^\lambda)\log(u)}du.$$

We have

$$\int_{2}^{2^{5\cdot(\lambda+1)}/H} \frac{2^{\lambda+1}}{(u\cdot 2^{\lambda+1}-2^\lambda)\log(u)}du = \int_{2}^{2^{5\cdot(\lambda+1)}/H} \frac{1}{(u-\frac{1}{2})\log(u)}du.$$

Since

$$\int_{2}^{2^{5\cdot(\lambda+1)}/H} \frac{1}{(u-\frac{1}{2})\log(u)}du - \int_{2}^{2^{5\cdot(\lambda+1)}/H} \frac{1}{u\log u}du = \int_{2}^{2^{5\cdot(\lambda+1)}/H} \frac{1}{2}\frac{1}{u(u-\frac{1}{2})\log u}$$

and

$$\begin{aligned}\int_{2}^{2^{5\cdot(\lambda+1)}/H} \frac{1}{2}\frac{1}{u(u-\frac{1}{2})\log u} &< \int_{2}^{2^{5\cdot(\lambda+1)}/H} \frac{1}{2\log 2}\frac{1}{(u-\frac{1}{2})^2} \\ &< \int_{2}^{\infty} \frac{1}{2\log 2}\frac{1}{(u-\frac{1}{2})^2}\end{aligned}$$

there exists an absolute constant J such that

$$-\log C_\lambda(x) < \int_{2}^{2^{5\cdot(\lambda+1)}/H} \frac{2}{u\log u}du + \int_{2^{6\cdot(\lambda+1)}/H}^{x} 2^\lambda \frac{\pi(t,2^{\lambda+1},1)}{t^2-2^\lambda t}dt - \log^{(2)} x - \gamma + J.$$

We use the result of [8] to bound

$$\int_{2^{6\cdot(\lambda+1)}/H}^{x} 2^\lambda \frac{\pi(t,2^{\lambda+1},1)}{t^2-2^\lambda t}dt.$$

Since

$$\begin{aligned}\left|\int_{2^{6\cdot(\lambda+1)}/H}^{x} 2^\lambda \frac{\pi(t,2^{\lambda+1},1)}{t^2-2^\lambda t} - \frac{1}{(t-2^\lambda)\log t}\right| dt &\le \int_{2^{6\cdot(\lambda+1)}/H}^{x} \left|2^\lambda \frac{\pi(t,2^{\lambda+1},1)}{t^2-2^\lambda t} - \frac{1}{(t-2^\lambda)\log t}\right| dt \\ &\le \int_{2^{6\cdot(\lambda+1)}/H}^{x} \frac{K}{(t-2^\lambda)(\log t)^2}dt \\ &\le \int_{2^{6\cdot(\lambda+1)}/H}^{x} \frac{K}{(t-2^\lambda)(\log(t-2^\lambda))^2}dt \\ &\le K\cdot\left[-\frac{1}{\log(t-2^\lambda)}\right]_{2^{6\cdot(\lambda+1)}/H}^{x} \\ &\le K\cdot\frac{1}{2^{6\cdot(\lambda+1)}/H-2^\lambda}\end{aligned}$$

we have

$$\int_{2^{6\cdot(\lambda+1)}/H}^{x} 2^{\lambda}\frac{\pi(t,2^{\lambda+1},1)}{t^2-2^{\lambda}t}dt = \int_{2^{6\cdot(\lambda+1)}/H}^{x}\frac{1}{(t-2^{\lambda})\log t}dt + O(1).$$

Thus,

$$\int_{2^{6\cdot(\lambda+1)}/H}^{x} 2^{\lambda}\frac{\pi(t,2^{\lambda+1},1)}{t^2-2^{\lambda}t}dt = \log^{(2)}(x-2^{\lambda}) - \log^{(2)}(2^{6\cdot(\lambda+1)}/H-2^{\lambda}) + O(1).$$

In conclusion, there exists an absolute constant J' such that

$$-\log C_{\lambda}(x) < \int_{2}^{2^{5\cdot(\lambda+1)}/H}\frac{2}{u\log u}du + \log^{(2)} x - \log(\lambda) - \log^{(2)} x - \gamma + J' = \log\lambda + O(1).$$

This implies in particular that there exists an absolute constant $C>0$ such that $C_{\lambda}\geq\frac{C}{\lambda}$. □

Université de Lorraine, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, F-54506, France

Inria, Villers-lès-Nancy, F-54600, France

CNRS, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, F-54506, France
E-mail address: svyatoslav.covanov@inria.fr

E-mail address: emmanuel.thome@inria.fr