HAL
open science

A Simple and Fast Algorithm for Computing the N-th Term of a Linearly Recurrent Sequence

Alin Bostan, Ryuhei Mori

To cite this version:

Alin Bostan, Ryuhei Mori. A Simple and Fast Algorithm for Computing the N-th Term of a Linearly Recurrent Sequence. SOSA'21 (SIAM Symposium on Simplicity in Algorithms), Jan 2021, Alexandria, United States. hal-02917827v2

HAL Id: hal-02917827

https://inria.hal.science/hal-02917827v2

Submitted on 21 Dec 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Simple and Fast Algorithm for Computing the N-th Term of a Linearly Recurrent Sequence

Alin Bostan* and Ryuhei Mori†
*Inria, Palaiseau, France and †Tokyo Institute of Technology, Japan
alin.bostan@inria.fr, mori@c.titech.ac.jp

Abstract

We present a simple and fast algorithm for computing the N-th term of a given linearly recurrent sequence. Our new algorithm uses $O(\mathsf{M}(d)\log N)$ arithmetic operations, where d is the order of the recurrence, and $\mathsf{M}(d)$ denotes the number of arithmetic operations for computing the product of two polynomials of degree d. The state-of-the-art algorithm, due to Fiduccia (1985), has the same arithmetic complexity up to a constant factor. Our algorithm is simpler, faster and obtained by a totally different method. We also discuss several algorithmic applications, notably to polynomial modular exponentiation and powering of matrices.

Keywords: Algebraic Algorithms; Computational Complexity; Linearly Recurrent Sequence; Rational Power Series; Fast Fourier Transform

1 Introduction

1.1 General context Computing efficiently selected terms in sequences is a basic and fundamental algorithmic problem, whose applications are ubiquitous, for instance in theoretical computer science [65, 49], algebraic complexity theory [59, 73], computer algebra [28, 71, 48], cryptography [31, 32, 29, 33], algorithmic number theory [72, 1], effective algebraic geometry [12, 36], numerical analysis [52, 51] and computational biology [56].

In simple terms, the problem can be formulated as follows:

> Given a sequence $(u_n)_{n\geq 0}$ in an effective ring[0] R, and given a positive integer $N \in \mathbb{N}$, compute the term u_N as fast as possible.

Here, the input $(u_n)_{n\geq 0} \in R^{\mathbb{N}}$ is assumed to be a recurrent sequence, specified by a data structure consisting in a *recurrence relation* and sufficiently many *initial terms* that uniquely determine its terms.

Efficiency is measured in terms of *ring operations* (algebraic model), or of *bit operations* (Turing machine model). The cost of an algorithm is respectively estimated in terms of *arithmetic complexity* or of *binary complexity*. Both measures have their own usefulness: the algebraic model is relevant when ring operations have essentially unit cost (typically, if R is a finite ring such as the prime field $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$), while the bit complexity model is relevant when elements of R have a variable bitsize, and thus ring operations in R have variable cost (typically, when R is the ring $\mathbb{Z}$ of integer numbers).

The recurrence relation satisfied by the input sequence $(u_n)_{n\geq 0}$ might be of several types:

(C) *linear with constant coefficients*, that is of the form,

$$u_{n+d} = c_{d-1}u_{n+d-1} + \cdots + c_0 u_n, \qquad n \geq 0,$$

for some given $c_0, \ldots, c_{d-1}$ in R. In this case we simply say that $(u_n)_{n\geq 0}$ is *linearly recurrent* (or, *C-recursive*). The most basic examples are the geometric sequence $(q^n)_{n\geq 0}$, for $q \in R$, and the Fibonacci sequence $(F_n)_n$ with $F_{n+2} = F_{n+1} + F_n$ for $n \geq 0$ and $F_0 = 0, F_1 = 1$.

(P) *linear with polynomial coefficients*, of the form,

$$u_{n+d} = c_{d-1}(n)u_{n+d-1} + \cdots + c_0(n)u_n, \quad n \geq 0,$$

for some given rational functions $c_0(x), \ldots, c_{d-1}(x)$ in $R(x)$. In this case the sequence is called *holonomic* (or, *P-recursive*). Among the most basic examples, other than the C-recursive ones, there is the factorial sequence $(n!)_{n\geq 0} = (1, 1, 2, 6, 24, 120, \ldots)$ and the Motzkin sequence $(u_n)_{n\geq 0} = (1, 1, 2, 4, 9, 21, 51, \ldots)$ specified by the recurrence $u_{n+1} = \frac{2n+3}{n+3} \cdot u_n + \frac{3n}{n+3} \cdot u_{n-1}$ and the initial conditions $u_0 = u_1 = 1$.

(Q) *linear with polynomial coefficients in q and q^n*, that is of the form,

$$u_{n+d} = c_{d-1}(q, q^n)u_{n+d-1} + \cdots + c_0(q, q^n)u_n, \; n \geq 0,$$

[0] The ring R is assumed to be commutative with unity and *effective* in the sense that its elements are represented using some data structure, and there exist algorithms for performing the basic ring operations $(+, -, \times)$ and for testing equality of elements in R.

for some $q \in R$ and some rational functions $c_0(x,y), \ldots, c_{d-1}(x,y)$ in $R(x,y)$. In this case, the sequence is called *q-holonomic*; such a sequence can be seen as a q-deformation of a holonomic sequence (in the sense that when $q \mapsto 1$, the limit sequence tends to be holonomic). A typical example is the q-factorial $[n]_q! := (1+q)\cdots(1+q+\cdots+q^{n-1})$.

In all these classes of examples, the recurrence is *linear*, and the integer d that governs the length of the recurrence relation is called the *order* of the corresponding linear recurrence.

Of course, some interesting sequences satisfy *non-linear* recurrences, as is the case for the so-called Somos-4 sequence $(1,1,1,1,2,3,7,23,\ldots)$ defined by: $u_{n+4} = (u_{n+3}u_{n+1}+u_{n+2}^2)/u_n$ together with $u_0 = \cdots = u_3 = 1$, but we will not consider this larger class in what follows.

To compute the N-th term u_N of a sequence of type (P), resp. (Q), the best known algorithms are presented in [16, 12], resp. in [9]. In the algebraic model, they rely on an algorithmic technique called *baby-step/giant-step*, allowing to compute u_N in a number of operations in R that is almost linear in $\sqrt{N}$, up to logarithmic factors. This should be contrasted with the direct iterative algorithm, of arithmetic complexity linear in N.

In the bit model, the same references provide different algorithms based on a different technique, called *binary splitting*; these algorithms are *quasi-optimal* in the sense that they are able to compute u_N in a number of bit operations almost linear (up to logarithmic factors) in the bitsize of the output value u_N. Once again, this should be contrasted with the direct iterative algorithm, whose binary complexity is larger by at least one order of magnitude (e.g., in case (P) the naive algorithm has bit complexity $O(N^3)$).

1.2 The case of C-recursive sequences In what follows, we will restrict our attention to the case (C) only. This case obviously is a subcase of both cases (P) and (Q). It presents an exceptional feature with respect to the algebraic model: contrary to the general cases (P) and (Q), in case (C) it is possible to compute the term u_N using a number of arithmetic operations in R that is only logarithmic in N.

For the geometric sequence $u_n = q^n$, this is known since Pingala (∼200 BC) who seemingly is the inventor of the algorithmic method of *binary powering*, or *square-and-multiply* [46, §4.6.3]. The corresponding algorithm is recursive and based on the equalities

$$q^N = \begin{cases} (q^{N/2})^2, & \text{if } N \text{ is even,} \\ q \cdot (q^{\frac{N-1}{2}})^2, & \text{else.} \end{cases}$$

The arithmetic complexity of this algorithm is bounded by $2\log N$ multiplications[1] in R, which represents a tremendous improvement compared to the naive iterative algorithm that computes the term q^N in $N-1$ multiplications in R, by simply unrolling the recurrence $u_{n+1} = q \cdot u_n$ with $q_0 = 1$.

In the general case (C), Miller and Spencer Brown showed in 1966 [54] that a similar complexity can be obtained by converting[2] the scalar recurrence of order d

$$u_{n+d} = c_{d-1}u_{n+d-1} + \cdots + c_0 u_n, \qquad n \geq 0, \tag{1.1}$$

into a vector recurrence of order 1

$$\underbrace{\begin{bmatrix} u_n \\ u_{n+1} \\ \vdots \\ u_{n+d-1} \end{bmatrix}}_{v_n} = \underbrace{\begin{bmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ c_0 & c_1 & \cdots & c_{d-1} \end{bmatrix}}_{M} \times \underbrace{\begin{bmatrix} u_{n-1} \\ u_n \\ \vdots \\ u_{n+d-2} \end{bmatrix}}_{v_{n-1}}, \; n \geq 1, \tag{1.2}$$

and by using binary powering in the ring $\mathcal{M}_d(R)$, of $d \times d$ matrices with coefficients in R, in order to compute M^N recursively by

$$M^N = \begin{cases} (M^{N/2})^2, & \text{if } N \text{ is even,} \\ M \cdot (M^{\frac{N-1}{2}})^2, & \text{else.} \end{cases}$$

From there, u_N can be read off the matrix-vector product $v_N = M^N \cdot v_0$. The complexity of this method is $O(d^\theta \log N)$ operations in R, where $\theta \in [2,3]$ is any feasible exponent for matrix multiplication in $\mathcal{M}_d(R)$.

Strangely enough, the paper [54] of Miller and Spencer Brown was largely overlooked in the subsequent literature, and their result has been rediscovered several times in the 1980s. For instance, Shortt [68] proposed a $O(\log N)$ algorithm for computing the N-th Fibonacci number[3,4] and extended it together with Wil-

[1] In this article, notation log refers to the logarithm in base 2.

[2] [15, p. 74] calls this *un truc bien connu* ("a well-known trick").

[3] Shortt's algorithm had actually appeared before, in the 1969 edition of Knuth's book [45, p. 552], as a solution of Ex. 26 (p. 421, §4.6.3). The algorithm is based on the so-called *doubling formulas* $(F_{2n}, F_{2n-1}) = (F_n^2 + 2F_nF_{n-1}, F_n^2 + F_{n-1}^2)$, actually due to Lucas (1876) and Catalan (1886), see e.g. [20, Ch. XVII]. The currently best implementation for computing F_N over $\mathbb{Z}$ (mpz_fib_ui from GMP) uses a variant of this method, requiring just two squares (and a few additions) per binary digit of N.

[4] Already in 1899, G. de Rocquigny asked "for an expeditious procedure to compute a very distant term of the Fibonacci sequence" [19]. In response, several methods (including the one mentioned by Knuth in [45, p. 552]) have been published one year later by Rosace (alias), E.-B. Escott, E. Malo, C.-A. Laisant and G. Picou [66]. This fact does not seem to have been noticed in the modern algorithmic literature before the current paper, although the reference [66] is mentioned in Dickson's formidable encyclopedic book [20, p. 404].

son [76] to the computation of order-d Fibonacci numbers in $O(d^3 \log N)$ arithmetic operations. The same cost has also been obtained by Dijkstra [21] and Urbanek [75]. Pettorossi [60], and independently Gries and Levin [34], improved the algorithm and lowered the cost to $O(d^2 \log N)$, essentially by taking into account the sparse structure of the matrix M. See also [22, 50, 23, 24, 39, 63, 30, 44] for similar algorithms.

1.3 Fiduccia's algorithm The currently best algorithm is due to Fiduccia[5] [26]. It is based on the following observation: the matrix M in (1.2) is the transpose of the companion matrix C which represents the R-linear multiplication-by-x map from the quotient ring $R[x]/(\Gamma)$ to itself, where $\Gamma = x^d - \sum_{i=0}^{d-1} c_i x^i$. Therefore, denoting by e the row vector $e = [1 \quad 0 \quad \cdots \quad 0]$, the N-th term u_N equals

(1.3)
$$u_N = e \cdot v_N = e \cdot M^N \cdot v_0 = \left(C^N \cdot e^T\right)^T \cdot v_0 = \langle x^N \bmod \Gamma, v_0 \rangle,$$

where the inner product takes place between the vector $v_0 = \left[u_0 \cdots u_{d-1}\right]$ of initial terms of $(u_n)_{n \geq 0}$, and the vector whose entries are the coefficients of the remainder $(x^N \bmod \Gamma)$ of the Euclidean division of x^N by Γ.

Thus, computing u_N is reduced to computing the coefficients of $(x^N \bmod \Gamma)$, and this can be performed efficiently by using binary powering in the quotient ring $A := R[x]/(\Gamma)$, at the cost of $O(\log N)$ multiplications in A. Each multiplication in A may be performed using $O(\mathsf{M}(d))$ operations in R [27, Ch. 9, Corollary 9.7], where $\mathsf{M}(d)$ denotes the arithmetic cost of polynomial multiplication in $R[x]$ in degree d. Using Fast Fourier Transform (FFT) methods, one may take $\mathsf{M}(d) = O(d \log d)$ when R contains enough roots of unity, and $\mathsf{M}(d) = O(d \log d \log \log d)$ in general [27, Ch. 8].

In conclusion, Fiduccia's algorithm allows the computation of the N-th term u_N of a linearly recurrent sequence of order d using $O(\mathsf{M}(d) \log N)$ operations in R. Since 1985, this is the state-of-the-art algorithm for this task in case (C).

A closer inspection of the proof of [27, Corollary 9.7] shows that a more precise estimate for the arithmetic cost of Fiduccia's algorithm is

$$\mathsf{F}(N, d) = 3\,\mathsf{M}(d) \lfloor \log N \rfloor + O(d \log N) \tag{1.4}$$

operations in R. This comes from the fact that squaring[6] in $A = R[x]/(\Gamma)$ is based on a polynomial product in degree less than d followed by an Euclidean division by Γ of a polynomial of degree less than $2d$. The Euclidean division is reduced to a power series division by the reversal $Q(x) := x^d \cdot \Gamma(1/x)$ of Γ, followed by a polynomial product in degree less than d. The reciprocal of $Q(x)$ is precomputed modulo x^d once and for all (using a formal Newton iteration) in $3\,\mathsf{M}(d) + O(d)$ operations in R, and then each squaring in A also takes $3\,\mathsf{M}(d) + O(d)$ operations in R. The announced cost from (1.4) follows from the fact that binary powering uses $\lfloor \log N \rfloor$ squarings and at most $\lfloor \log N \rfloor$ multiplications by x.

1.4 Main results We propose in this paper a new and simpler algorithm, with a better cost. More precisely, our first main complexity result is:

Theorem 1. *One can compute the N-th term of a linearly recurrent sequence of order d with coefficients in a ring R using*

$$\mathsf{T}(N, d) = 2\,\mathsf{M}(d) \lceil \log(N+1) \rceil + \mathsf{M}(d)$$

arithmetic operations in R.

The proof of this result is based on a very natural algorithm, which will be presented in Section 2.1. Let us remark that it improves by a factor of 1.5 the complexity of Fiduccia's algorithm. This factor is even higher in the FFT setting, where polynomial multiplication is assumed to be performed using Fast Fourier Transform techniques. In this setting, we obtain the following complexity result, which will be proved in Section 4.

Theorem 2. *One can compute the N-th term of a linearly recurrent sequence of order d with coefficients in a field $\mathbb{K}$ supporting FFT using*[7]

$$\sim \frac{2}{3}\,\mathsf{M}(d) \log(N)$$

arithmetic operations in $\mathbb{K}$.

Algorithms 1 and 2 (underlying Theorem 1) and Algorithm 11 (underlying Theorem 2) are both of LSB-first (least significant bit first) type. This prevents them from computing simultaneously *several* consecutive terms of high indices, such as $u_N, \ldots, u_{N+d-1}$. This makes a notable difference with Fiduccia's algorithm from §1.3. For this reason, we will design a second algorithm, of MSB-first (most significant bit first) type, by "transposing" Algorithm 1.

This leads to the following complexity result.

[5] The idea already appears in the 1982 conference paper [25]. We have discovered that the same algorithm had been sketched by D. Knuth in the corrections/changes to [46] published in 1981 in [47, p. 28], where he attributes the result to R. Brent. Almost surely, C. Fiduccia was not aware about this fact.

[6] Note that multiplying by x in A is much easier and has linear arithmetic cost $O(d)$.

[7] Here, and in what follows, the "equivalent" sign $\sim$ is to be understood for fixed N and d going to infinity.

Theorem 3. *One can compute the terms of indices $N-d+1, \ldots, N$ of a linearly recurrent sequence of order d with coefficients in a ring R using*

$$(2\,\mathsf{M}(d)+d)\lceil \log(N+1)\rceil + O(\mathsf{M}(d))$$

arithmetic operations in R.

The method underlying this complexity result is based on Algorithms 5, 6 and 8, which are presented in Section 3. Along the way, using the MSB-first Algorithm 5, we improve the cost of *polynomial modular exponentiation*, which is a central algorithmic task in computer algebra, with many applications. Since this result has an interest *per se*, we isolate it here as our last complexity result.

Theorem 4. *Given $N \in \mathbb{N}$ and a polynomial $\Gamma(x)$ in $R[x]$ of degree d, one can compute $x^N \bmod \Gamma(x)$ using*

$$(2\,\mathsf{M}(d)+d)\lceil \log(N+1)\rceil + \mathsf{M}(d)$$

arithmetic operations in R.

This complexity of $\sim 2\,\mathsf{M}(d)\log N$ compares favorably with the currently best estimate of $\sim 3\,\mathsf{M}(d)\log N$ obtained by square-and-multiply in the quotient ring $R[x]/(\Gamma(x))$, combined with fast modular multiplications performed either classically [27, Corollary 9.7], or using Montgomery's algorithm [55]. In the FFT setting, the gain is even larger, and our results improve on the best estimates, due to Mihăilescu [53].

1.5 Structure of the paper

In Section 2 we propose our LSB-first (least significant bit first) algorithm for computing the N-th term of a C-recursive sequence. We design in Section 3 a second algorithm, which is an MSB-first (most significant bit first) variant, and discuss several algorithmic applications, including polynomial modular exponentiation and powering of matrices. In Section 4 we specialize and analyze Algorithm 1 in the specific FFT setting, where polynomial multiplication is based on Discrete Fourier Transform techniques, and we compare it with the FFT-based Fiduccia's algorithm. We conclude in Section 5 by a summary of results and plans of future work.

2 The LSB-first algorithm and applications

We will prove Theorem 1 in §2.1, where we propose the first main algorithms (Algorithms 1 and 2), which are faster than Fiduccia's algorithm. Then, in §2.2 we instantiate them in the particular case of the Fibonacci sequence. The resulting algorithm is competitive with state-of-the-art algorithms.

2.1 LSB-first algorithm: Proof of Theorem 1

In this section, we give the proof of Theorem 1. Let us denote by $F(x) \in R[[x]]$ the generating function of the sequence $(u_n)_{n\geq 0}$,

$$F(x) := \sum_{n\geq 0} u_n x^n.$$

We use the following classical characterization of generating functions of linearly recurrent sequences [64]. To be self-contained, we include a proof.

Lemma 1. *A sequence $(u_n)_{n\geq 0}$ is linearly recurrent of order d if and only if its generating function is $F(x) = P(x)/Q(x)$ for some polynomials $P(x)$ of degree at most $d-1$ and $Q(x)$ of degree d satisfying $Q(0) = 1$.*

Proof. Assume $(u_n)_{n\geq 0}$ is a linearly recurrent sequence of order d. Let us define $Q(x) := 1 - c_{d-1}x - \cdots - c_0 x^d$, that is the reversal of the characteristic polynomial $\Gamma(x) = x^d - \sum_{i=0}^{d-1} c_i x^i$ of recurrence (1.1). Then, $P(x) := Q(x)\cdot F(x)$ is a polynomial of degree less than d. This is immediately seen by checking that, for any $n \geq 0$, the coefficient of x^{n+d} in the power series $Q(x)\cdot F(x)$ is equal to $u_{n+d} - c_{d-1}u_{n+d-1} - \cdots - c_0 u_n$, hence it is zero by (1.1).

Conversely, assume that the generating function of a sequence $(u_n)_{n\geq 0}$ is $F(x) = P(x)/Q(x)$ for some polynomials $P(x)$ and $Q(x)$ satisfying the conditions in the lemma. Then, $Q(x)F(x) = P(x)$ implies that the coefficient of x^{n+d} in $Q(x)F(x)$ is zero for all integers $n \geq 0$. Hence, the linear recurrence equation (1.1) must be satisfied, where $1 - c_{d-1}x - \cdots - c_0 x^d := Q(x)$. □

Proof of Theorem 1. From the values $u_0, \ldots, u_{d-1}$ and $c_0, \ldots c_{d-1}$, we obtain $Q(x) := 1 - c_{d-1}x - \cdots - c_0 x^d$ without any operations in R, and can compute $P(x) := (u_0 + \cdots + u_{d-1}x^{d-1})\cdot Q(x) \bmod x^d$ using at most $\mathsf{M}(d)$ operations in R. According to Lemma 1, our goal is to compute

$$u_N = [x^N]\,\frac{P(x)}{Q(x)},$$

where $[x^N]\,F(x)$ denotes the coefficient of x^N in $F(x) \in R[[x]]$. From Lemma 1, we can assume that $Q(0) = 1$. But, for the sake of generality, in the following, we only assume that $Q(0)$ is invertible in R. If $N = 0$, we have $u_N = P(0)/Q(0)$. If $N \geq 1$, we multiply $Q(-x)$ by the numerator $P(x)$ and the denominator $Q(x)$, and obtain

$$u_N = [x^N]\,\frac{P(x)Q(-x)}{Q(x)Q(-x)}.$$

Here, $Q(x)Q(-x)$ is an even polynomial, which means that all coefficients of x^{2n+1} for integers $n \geq 0$ in this

Algorithm 1 (OneCoeff)
Input: $P(x)$, $Q(x)$, N **Output**: $[x^N]\,\frac{P(x)}{Q(x)}$

Assumptions: $Q(0)$ invertible and $\deg(P) < \deg(Q) =: d$

while $N \geq 1$ **do**
  $U(x) \leftarrow P(x)Q(-x)$ ▷ $U = \sum_{i=0}^{2d-1} U_i x^i$
  if N is even **then**
    $P(x) \leftarrow \sum_{i=0}^{d-1} U_{2i} x^i$
  else
    $P(x) \leftarrow \sum_{i=0}^{d-1} U_{2i+1} x^i$
  $A(x) \leftarrow Q(x)Q(-x)$ ▷ $A = \sum_{i=0}^{2d} A_i x^i$
  $Q(x) \leftarrow \sum_{i=0}^{d} A_{2i} x^i$
  $N \leftarrow \lfloor N/2 \rfloor$
return $P(0)/Q(0)$

polynomial are equal to 0. Hence, there exists a unique polynomial $V(x)$ satisfying $V(x^2) = Q(x)Q(-x)$. Here, the degree of $V(x)$ is d and $V(0) = Q(0)^2$ is invertible in R. Now, we obtain

$$u_N = [x^N]\,\frac{U(x)}{V(x^2)}$$

for $U(x) := P(x)Q(-x)$. The numerator $U(x)$ has degree at most $2d-1$. Here, $1/V(x^2)$ is an even formal power series. That implies that if N is even (or odd), we can ignore the odd (or even) part of $U(x)$. Let $U_{\mathrm{e}}(x)$ and $U_{\mathrm{o}}(x)$ be the even and odd parts of $U(x)$, respectively, i.e., $U_{\mathrm{e}}(x) := \sum_{i=0}^{d-1} U_{2i} x^i$ and $U_{\mathrm{o}}(x) := \sum_{i=0}^{d-1} U_{2i+1} x^i$ for $U(x) = \sum_{i=0}^{2d-1} U_i x^i$. Then, we obtain the decomposition $U(x) = U_{\mathrm{e}}(x^2) + xU_{\mathrm{o}}(x^2)$. Hence,

$$u_N = \begin{cases} [x^N]\,\frac{U_{\mathrm{e}}(x^2)}{V(x^2)}, & \text{if } N \text{ is even} \\ [x^N]\,\frac{xU_{\mathrm{o}}(x^2)}{V(x^2)}, & \text{else.} \end{cases}$$
$$= \begin{cases} [x^{N/2}]\,\frac{U_{\mathrm{e}}(x)}{V(x)}, & \text{if } N \text{ is even} \\ [x^{(N-1)/2}]\,\frac{U_{\mathrm{o}}(x)}{V(x)}, & \text{else.} \end{cases}$$

Here, $U_{\mathrm{e}}(x)$ and $U_{\mathrm{o}}(x)$ are polynomials of degree at most $d-1$, while $V(x)$ is a polynomial of degree d satisfying that $V(0)$ is invertible in R. Hence, we can repeat this reduction until $N \geq 1$.

Our algorithm for computing $[x^N]\ P(x)/Q(x)$ is summarized in Algorithm 1, and its immediate consequence for computing the N-th term of the linearly recurrent sequence $(u_n)_{n\geq 0}$ defined by eq. (1.1) is displayed in Algorithm 2. Algorithm 1 has complexity $2\,\mathsf{M}(d)\lceil \log(N+1)\rceil$ and Algorithm 2 has cost $2\,\mathsf{M}(d)\lceil \log(N+1)\rceil + \mathsf{M}(d)$, which proves Theorem 1. □

Note that Algorithms 1 and 2 use an idea similar to the ones in [11, 10] which were dedicated to the larger class of algebraic power series, but restricted to positive characteristic only. Algorithm 1 also shares common features with the technique of *section operators* [2, Lemma 4.1] used by Allouche and Shallit to compute the N-th term of k-regular sequences [2, Corollary 4.5] in $O(\log N)$ ring operations.

Algorithm 2 (OneTerm)
Input: rec. (1.1), $u_0, \ldots, u_{d-1}$, N **Output**: u_N

Assumptions: $\Gamma(x) = x^d - \sum_{i=0}^{d-1} c_i x^i$ with $c_0 \neq 0$

$Q(x) \leftarrow x^d \Gamma(1/x)$
$P(x) \leftarrow (u_0 + \cdots + u_{d-1}x^{d-1}) \cdot Q(x) \bmod x^d$
return $[x^N]P(x)/Q(x)$ ▷ using Algorithm 1

Algorithm 1 can be interpreted at the level of recurrences as computing $\sim \log N$ new recurrences produced by the Graeffe process, which is a classical technique to compute the largest root of a real polynomial [40, 57, 58]. Interestingly, the Graeffe process has been used in a purely algebraic context by Schönhage in [67, §3] for computing the reciprocal of a power series, see also [14, §2]. However, our paper seems to be the first reference where the Graeffe process and the section operators approach are combined together.

2.2 New algorithm for Fibonacci numbers To illustrate the mechanism of Algorithm 1, let us instantiate it in the particular case of the Fibonacci sequence

$$F_0 = 0, F_1 = 1, \quad F_{n+2} = F_{n+1} + F_n,\ n \geq 0.$$

The generating function $\sum_{n\geq 0} F_n x^n$ is $x/(1-x-x^2)$. Therefore, the coefficient $F_N = [x^N]\,\frac{x}{1-x-x^2}$ is equal to

$$[x^N]\,\frac{x(1+x-x^2)}{1-3x^2+x^4} = \begin{cases} [x^{\frac{N}{2}}]\,\frac{x}{1-3x+x^2}, & \text{if } N \text{ is even,} \\ [x^{\frac{N-1}{2}}]\,\frac{1-x}{1-3x+x^2}, & \text{else.} \end{cases}$$

The computation of F_N is reduced to that of a coefficient of the form

$$[x^N]\,\frac{a+bx}{1-cx+x^2} = [x^N]\,\frac{(a+bx)(1+cx+x^2)}{1-(c^2-2)x^2+x^4}$$

which is equal to

$$\begin{cases} [x^{\frac{N}{2}}]\,\frac{a+(bc+a)x}{1-(c^2-2)x+x^2}, & \text{if } N \text{ is even,} \\ [x^{\frac{N-1}{2}}]\,\frac{(ac+b)+bx}{1-(c^2-2)x+x^2}, & \text{else.} \end{cases}$$

This yields Algorithm 3 for the computation[8] of F_N.

[8] Notice that the same algorithm can be used to compute efficiently the N-th Fibonacci polynomial, or the N-th Chebyshev polynomial. Fibonacci polynomials in $R[t]$ are defined by $F_{n+2}(t) = t \cdot F_{n+1}(t) + F_n(t)$ with $F_0(t) = 1$ and $F_1(t) = 1$. It is sufficient to initialize c to t^2+2 (instead of 3) and b to t when N is even (instead of 0). The complexity of this algorithm is $O(\mathsf{M}(N))$ operations in R, which is quasi-optimal.

Algorithm 3 (NewFibo) **Input**: N **Output**: F_N

Assumptions: $N \geq 2$

1: $c \leftarrow 3$
2: **if** N is even **then**
3: $\quad [a, b] \leftarrow [0, 1]$
4: **else**
5: $\quad [a, b] \leftarrow [1, -1]$
6: $N \leftarrow \lfloor N/2 \rfloor$
7: **while** $N > 1$ **do**
8: $\quad$ **if** N is even **then**
9: $\quad\quad b \leftarrow a + b \cdot c$
10: $\quad$ **else**
11: $\quad\quad a \leftarrow b + a \cdot c$
12: $\quad c \leftarrow c^2 - 2$
13: $\quad N \leftarrow \lfloor N/2 \rfloor$
14: **return** $b + a \cdot c$

In the direct application of Algorithm 1 to the Fibonacci sequence, the condition $N > 1$ in line 7 should be $N \geq 1$ and $b + a \cdot c$ in line 14 should be a. But, in that case, there is a redundant product c^2 in line 12 when $N = 1$. This modification saves one multiplication and one subtraction. This saving is crucial over the integers, when we consider bit complexity, since c grows exponentially in the iterations, and the bit complexity of the redundant multiplication c^2 occupies a constant factor of the whole bit complexity.

A close inspection reveals that this algorithm computes F_N by a recursive use of the formula

$$F_N = L_{2^{\lfloor \log N \rfloor}} \cdot F_{N-2^{\lfloor \log N \rfloor}} + (-1)^N \cdot F_{2^{1+\lfloor \log N \rfloor}-N},$$

which is a particular instance of the classical formula

$$F_{n+m} = L_m F_n + (-1)^n F_{m-n}$$

relating the Fibonacci numbers and the Lucas numbers $L_n = F_{n+1} + F_{n-1}$.

When N is a power of 2, then Algorithm 3 degenerates into Algorithm 4. This is equivalent to Algorithm `fib`(n) in [18, Fig. 6][9]. It uses $2\log(N) - 3$ products (of which $\log(N) - 2$ are squarings) and $\log(N) - 2$ subtractions.

When N is arbitrary, Algorithm 3 has essentially the same cost: it uses at most $2\log(N) - 1$ products (of which at most $\log(N) - 1$ are squarings) and $2\lfloor \log(N) \rfloor - 1$ additions/subtractions. In contrast, [18, Fig. 6] uses a more complex algorithm, with higher cost.

A nice feature of Algorithm 3 is not only that it is simple and natural, but also that its arithmetic and bit complexity match the complexities of the state-of-art algorithms for computing Fibonacci numbers [74].

[9] This algorithm had also appeared before, in Knuth's book [45, p. 552, second solution].

Algorithm 4 **Input**: N **Output**: F_N

Assumptions: $N \geq 2$ and N is a power of 2

$[b, c] \leftarrow [1, 3]$
$N \leftarrow \lfloor N/2 \rfloor$
while $N > 2$ **do**
$\quad b \leftarrow b \cdot c$
$\quad c \leftarrow c^2 - 2$
$\quad N \leftarrow \lfloor N/2 \rfloor$
return $b \cdot c$

3 The MSB-first algorithm and applications

We present in §3.1 a "most significant bit" (MSB) variant (Algoritm 5) of Algoritm 1. Then we discuss various applications of Algorithms 1 and 5. In §3.2 we design a faster algorithm for polynomial modular exponentiation, that we use in §3.3 to design a faster Fiduccia-like algorithm for computing a slice of d terms of indices $N-d+1, \ldots, N$ in $\sim 2\,\mathsf{M}(d) \log N$ operations.

3.1 The MSB-first algorithm In Fiduccia's algorithm (§1.3), the N-th coefficient u_N in the power series expansion $\sum_{i \geq 0} u_i x^i$ of P/Q is given by the inner product $\langle x^N \bmod \Gamma(x), v_0 \rangle$, where Γ is the reversal polynomial of Q and v_0 is the vector of initial coefficients $\begin{bmatrix} u_0 & \cdots & u_{d-1} \end{bmatrix}$. Here, $x^N \bmod \Gamma(x)$ depends only on the linear recurrence equation (1.1), and is independent of the initial terms v_0. Hence, if we want to compute the N-th terms of k different linearly recurrent sequences that share the same linear recurrence equation (1.1), we can first determine $\rho(x) := x^N \bmod \Gamma(x)$, and then $\langle \rho, v_0^{(i)} \rangle$ for $i = 1, \ldots, k$, where $v_0^{(i)}$ denotes the vector of d initial terms of the i-th sequence. The total arithmetic complexity of this algorithm is $O(\mathsf{M}(d) \log N + kd)$; this is faster than Fiduccia's algorithm repeated independently k times, with cost $O(k\,\mathsf{M}(d) \log N)$.

On the other hand, in Algorithm 1 we iteratively update both the denominator and numerator, and each new numerator depends on the original numerator $P(x)$ which encodes the initial d terms of the sequence. Hence, it is not a priori clear how to obtain with Algorithm 1 the good feature of Fiduccia's algorithm.

In this section, we present an algorithm that computes u_N with complexity equal to that of Algorithm 1 and which, in addition, achieves the cost $O(\mathsf{M}(d) \log N + kd)$ for the above problem with k sequences.

While Algorithm 1 looks at N from the least significant bit (LSB), the main algorithm presented in this section (Algoritm 5) looks at N from the most

Algorithm 5 (SliceCoeff)
Input: $Q(x)$, N **Output**: $\mathcal{F}_{N,d}(1/Q(x))$

Assumptions: $Q(0)$ invertible and $\deg(Q) =: d$

1: **function** SliceCoeff(N, $Q(x)$)
2: **if** $N = 0$ **then**
3: **return** $x^{d-1}/Q(0)$
4: $A(x) \leftarrow Q(x)Q(-x)$ ▷ $A = \sum_{i=0}^{2d} A_i x^i$
5: $V(x) \leftarrow \sum_{i=0}^{d} A_{2i} x^i$
6: $W(x) \leftarrow$ SliceCoeff($\lfloor N/2 \rfloor, V(x)$)
7: **if** N is even **then**
8: $S(x) \leftarrow xW(x^2)$
9: **else**
10: $S(x) \leftarrow W(x^2)$
11: $B(x) \leftarrow Q(-x)S(x)$ ▷ $B = \sum_{i=0}^{3d-1} B_i x^i$
12: **return** $\sum_{i=0}^{d-1} B_{d+i} x^i$

Algorithm 6 (OneCoeffT)
Input: $P(x)$, $Q(x)$, N **Output**: $[x^N]\,\frac{P(x)}{Q(x)}$

Assumptions: $Q(0)$ invertible, $\deg(P) < \deg(Q) =: d$

1: $U \leftarrow \mathcal{F}_{N,d}(1/Q(x))$ using Algorithm 5 ▷
$U = u_{N-d+1} + \cdots + u_N x^{d-1}$
2: **return** $p_0 u_N + \cdots + p_{d-1} u_{N-d+1}$ ▷ $P = \sum_{i=0}^{d-1} p_i x^i$

Algorithm 7
Input: $P_1, \ldots, P_k, Q, N$ **Output**: $[x^N]\,\frac{P_j}{Q}, j = 1, \ldots, k$

Assumptions: $Q(0)$ invertible, $\deg(P) < \deg(Q) =: d$

1: $U \leftarrow \mathcal{F}_{N,d}(1/Q(x))$ using Algorithm 5 ▷
$U = u_{N-d+1} + \cdots + u_N x^{d-1}$
2: **return** $p_0^{(j)} u_N + \cdots + p_{d-1}^{(j)} u_{N-d+1}, j = 1, \ldots, k$ ▷
$P_j = \sum_{i=0}^{d-1} p_i^{(j)} x^i$

significant bit (MSB). In fact, Algoritm 5 is essentially equivalent to "the transposition" of Algorithm 1 in the sense of [8]. This is the reason why this MSB-first algorithm has the same cost as Algorithm 1. However, in order to keep the presentation self-contained, we are not going to appeal here to the general machinery of algorithmic transposition tools, but rather derive the transposed algorithm "by hand", by a direct reasoning.

In order to compute $[x^N]\,P(x)/Q(x)$, it is sufficient to compute the $(N-d+1)$-th term to the N-th term of $1/Q(x)$ since the degree of $P(x)$ is at most $d-1$. Let $\mathcal{F}_{N,d}(\sum_{i\geq 0} a_i x^i) := \sum_{i=0}^{d-1} a_{N-d+1+i} x^i$. Our goal is to compute $\mathcal{F}_{N,d}(1/Q(x))$. We have the equalities

$$\begin{aligned}\mathcal{F}_{N,d}\left(\frac{1}{Q(x)}\right) &= \mathcal{F}_{N,d}\left(\frac{Q(-x)}{Q(x)Q(-x)}\right)\\ &= \mathcal{F}_{N,d}\left(Q(-x)x^{N-2d+1}\mathcal{F}_{N,2d}\left(\frac{1}{Q(x)Q(-x)}\right)\right)\\ &= \mathcal{F}_{2d-1,d}\left(Q(-x)\mathcal{F}_{N,2d}\left(\frac{1}{V(x^2)}\right)\right),\end{aligned}$$

where $V(x^2) := Q(x)Q(-x)$.

In the second equality, we ignore the terms of $1/V(x^2)$ except for the $(N-2d+1)$-th term to the N-th term. In the third equality, we use the fact that $\mathcal{F}_{N,d}(xA(x)) = \mathcal{F}_{N-1,d}(A(x))$. Let now $W(x) := \mathcal{F}_{\lfloor N/2 \rfloor,d}(1/V(x))$. Then, it is easy to see that

$$\mathcal{F}_{N,d}\left(\frac{1}{Q(x)}\right) = \mathcal{F}_{2d-1,d}\left(Q(-x)S(x)\right),$$

where

$$S(x) := \begin{cases} xW(x^2), & \text{if } N \text{ is even}\\ W(x^2), & \text{else.}\end{cases}$$

The resulting method for computing $\mathcal{F}_{N,d}(1/Q(x))$ is summarized in Algorithm 5, and its applications to the computation of $[x^N]P/Q$, and to $[x^N]P^{(i)}/Q$ for several $i = 1, \ldots, k$, are displayed in Algorithms 6 and 7.

Let us analyze the complexity of Algorithms 5 and 6 more carefully. At each step, Algorithm 5 computes $Q(x)Q(-x)$ and $Q(-x)S(x)$, where the degrees of $Q(x)$ and $S(x)$ are d and at most $2d-1$, respectively. Hence a direct analysis concludes that its complexity is $3\,\mathsf{M}(d)\log N$ operations in R. However, an improvement comes from the remark that not all coefficients of $Q(-x)S(x)$ are needed: it is sufficient to compute the d-th coefficient to the $(2d-1)$-th coefficient of $Q(-x)S(x)$. This operation is known as "the middle product", and can be performed with the same arithmetic complexity as the standard product of two polynomials of degree d, plus a few d additional operations [35, 8]. Therefore, if steps 11 and 12 of Algorithm 5 are performed "at once" using a middle product, then the arithmetic complexity drops to $(2\,\mathsf{M}(d)+d)\log N$. This complexity is also inherited by Algorithm 6, which uses at most $2d$ additional operations in the last step.

It should be obvious at this point that the slight variant Algorithm 7 of Algorithm 6 achieves arithmetic complexity $O(\mathsf{M}(d)\log N + kd)$ for the aforementioned problem with k sequences, and more precisely its cost is of at most $(2\,\mathsf{M}(d)+d)\log N + 2kd$ operations in R.

In conclusion, Algorithm 5 achieves the same arithmetic complexity as Algorithm 1 and it extends to Algorithms 6 and 7. All algorithmic techniques specific to the FFT setting, that we will describe in Section 4, can also be applied to Algorithms 5, 6 and 7, yielding the same complexity gains.

Algorithm 8 (NewModExp)
Input: $\Gamma(x)$, N **Output**: $x^N \bmod \Gamma(x)$

Assumptions: $\mathsf{lcoeff}(\Gamma)$ invertible, $\Gamma(0) \neq 0$ and $\deg(\Gamma) =: d$

1: $Q(x) \leftarrow x^d \Gamma(1/x)$
2: $u(x) \leftarrow \mathcal{F}_{N,d}(1/Q(x))$ ▷ using Algorithm 5
3: $v(x) \leftarrow u(x)Q(x) \bmod x^d$
4: **return** $v(1/x)x^{d-1}$

3.2 Faster modular exponentiation The algorithms of §3.1 are not only well-suited to compute the N-th terms of several sequences satisfying the same recurrence. In this section, we show that they also permit a surprising application to the computation of *polynomial modular exponentiations*. This fact has many consequences, since modular exponentiation is a central algorithmic task in algebraic computations. In §3.3, we will discuss a first application in relation with the main topic of our article. Namely, we will design a new Fiduccia-style algorithm for the computation of the N-th term, and actually of a whole slice of $k \geq d$ terms, in $(2\,\mathsf{M}(d) + d)\log N + O((k+d)\mathsf{M}(d)/d)$ ring operations. More consequences will be separately discussed in §3.4.

Assume we are given a polynomial $\Gamma(x) \in R[x]$ of degree d, an integer N, and that we want to compute $\rho(x) := x^N \bmod \Gamma(x)$. Without loss of generality, we may assume $\Gamma(0) \neq 0$. Let $Q(x) \in R[x]$ be the reversal of $\Gamma(x)$, that is $Q(x) := x^d\Gamma(1/x)$. Let us denote the power series expansion of $1/Q$ by $\sum_{i\geq 0} a_i x^i$. Then, equation (1.3) implies that

$$\begin{bmatrix} a_N & \cdots & a_{N+d-1} \end{bmatrix} = \mathbf{r} \times \mathbf{H}, \tag{3.5}$$

where $\mathbf{r} = \begin{bmatrix} r_0 & \cdots & r_{d-1} \end{bmatrix}$ with $\rho = \sum_{i=0}^{d-1} r_i x^i$ and $\mathbf{H}$ is the Hankel matrix

$$\mathbf{H} := \begin{bmatrix} a_0 & \cdots & a_{d-1} \\ a_1 & \cdots & a_d \\ & \vdots & \\ a_{d-1} & \cdots & a_{2d-2} \end{bmatrix}.$$

Note that the matrix $\mathbf{H}$ is invertible, as its determinant is equal (up to a sign) to $([x^d]Q)^{d-1} = \Gamma(0)^{d-1}$. Therefore, $\mathbf{r}$ (and thus ρ) can be found by

(1) computing $\begin{bmatrix} a_N & \cdots & a_{N+d-1} \end{bmatrix}$ using Algorithm 5;

(2) solving the Hankel linear system (3.5).

Step (1) has arithmetic cost $(2\,\mathsf{M}(d) + d)\log(N+d)$, while step (2) has negligible cost $O(\mathsf{M}(d)\log d)$ using [13], see also [6, Ch. 2, §5].

It is actually possible to improve a bit more on this algorithm, by using the next lemma.

Lemma 2. *Let $N \in \mathbb{N}$ and let $\Gamma(x) \in R[x]$ be of degree d with $\Gamma(0) \neq 0$. Let $Q(x) \in R[x]$ be its reversal, $Q(x) := x^d\Gamma(1/x)$. Denote its reciprocal $1/Q$ by $\sum_{i\geq 0} a_i x^i$, and let $u(x)$ be $\mathcal{F}_{N,d}(1/Q(x)) = a_{N-d+1} + \cdots + a_N x^{d-1}$. Define $v(x)$ to be $u(x)Q(x) \bmod x^d$. Then $x^N \bmod \Gamma(x) = v(1/x)x^{d-1}$.*

Proof. Write the Euclidean division $x^N = L(x)\cdot\Gamma(x) + \rho(x)$, where $\deg(L) = N - d$ and $\rho(x) = r_0 + \cdots + r_{d-1}x^{d-1}$. Replacing x by $1/x$ on both sides, and then multiplying by x^N yields $1 = L_{\mathrm{rev}}(x)\cdot Q(x) + x^{N-d+1}\cdot\tilde{\rho}(x)$, where $L_{\mathrm{rev}}(x) = x^{N-d}\cdot L(1/x)$ and $\tilde{\rho}(x) = x^{d-1}\cdot\rho(1/x)$. In other words

$$\frac{1}{Q(x)} = L_{\mathrm{rev}}(x) + x^{N-d+1}\cdot\frac{\tilde{\rho}(x)}{Q(x)}.$$

Since $L_{\mathrm{rev}}(x)$ has degree at most $N-d$, it follows that $u(x) = \frac{\tilde{\rho}(x)}{Q(x)} \bmod x^d$. Therefore, $\tilde{\rho}(x)$ is equal to $v(x)$, and the conclusion follows. □

The merit of Lemma 2 is that it shows that computing $x^N \bmod \Gamma(x)$ can be reduced to computing $\mathcal{F}_{N,d}(1/Q(x))$, plus a few additional operations with negligible cost $\mathsf{M}(d)$. The resulting method is presented in Algorithm 8, whose complexity is $(2\,\mathsf{M}(d) + d)\log N + \mathsf{M}(d)$. This proves Theorem 4.

Note that Algorithm 8 is simpler, and faster by a factor of 1.5, than the classical algorithm based on binary powering in the quotient ring $R[x]/(\Gamma(x))$. Algorithm 5, and hence also Algorithm 8, admits a specialization into the FFT setting (Algorithm 13 in §4) with complexity $\sim \frac{2}{3}\,\mathsf{M}(d)\log N$. Similarly to the case of Algorithm 11 in §4, this FFT variant of Algorithm 8 is faster by a factor of 2.5 than Shoup's (comparatively simple) algorithm [70, §7.3], and by a factor of 1.625 than the (much more complex) algorithm of Mihăilescu [53].

This speed-up might be beneficial for instance in applications to polynomial factoring in $\mathbb{F}_p[x]$, where one time-consuming step to factor $f \in \mathbb{F}_p[x]$ is the computation of $x^p \bmod f$, see [27, Algorithms 14.3, 14.8, 14.13, 14.15, 14.31, 14.33 and 14.36], and also [70, 48].

It is also so in point-counting methods such as Schoof's algorithm and the Schoof-Elkies-Atkin (SEA) algorithm [7, Ch. VII], the second one being the best known method for counting the number of points of elliptic curves defined over finite fields of large characteristic. Indeed, the bulks of these algorithms are computations of x^q modulo the "division polynomial" $f_\ell(x)$ and of x^q modulo the "modular polynomial" $\Phi_\ell(x)$, where $\ell = O(\log(q))$ and $\deg(f_\ell) = O(\ell^2)$, $\deg(\Phi_\ell) = O(\ell)$.

As a final remark, note that while Fiduccia's algorithm shows that computing the terms of indices

Algorithm 9 **Input**: rec. (1.1), $u_0, \dots, u_{d-1}$, N
Output: $u_N, \dots, u_{N+d-1}$

Assumptions: $\Gamma(x) = x^d - \sum_{i=0}^{d-1} c_i x^i$ with $c_0 \neq 0$

1: $\rho(x) \leftarrow x^N \bmod \Gamma(x)$ ▷ using Algorithm 8
2: $U(x) \leftarrow u_0 + \cdots + u_{2d-2}x^{2d-2}$ ▷ using Algorithm in [69, p. 18]
3: $V(x) \leftarrow U(x) \cdot (x^d \cdot \rho(1/x))$ ▷ $V = \sum_{i=0}^{d-1} v_i x^i$
4: **return** $[v_d, \dots, v_{2d-1}]$

$N, \dots, N+d-1$ of a linearly recurrent sequence of order d can be reduced to polynomial modular exponentiation ($x^N \bmod \Gamma(x)$), Algorithm 8 shows that the converse is also true: polynomial modular exponentiation can be reduced to computing the terms of indices $N, \dots, N+d-1$ of a linearly recurrent sequence of order d. Therefore, *these two problems are computationally equivalent*. To our knowledge, this important fact seems not to have been noticed before.

3.3 A new Fiduccia-style algorithm We conclude this section by discussing a straightforward application of Algorithm 8. This is based on the next equality, generalizing (3.5) to any $k \geq 1$:

$$\begin{bmatrix} u_N & \cdots & u_{N+k-1} \end{bmatrix} = \mathbf{r} \times \mathbf{H}_k, \tag{3.6}$$

where as before $\mathbf{r} = \begin{bmatrix} r_0 & \cdots & r_{d-1} \end{bmatrix}$ is the coefficients vector of $\rho = \sum_{i=0}^{d-1} r_i x^i$, with $\rho = x^N \bmod \Gamma(x)$, and $\mathbf{H}_k$ is the Hankel matrix

$$\mathbf{H}_k := \begin{bmatrix} u_0 & \cdots & u_{d-1} & \cdots & \cdots & u_{k-1} \\ u_1 & \cdots & u_d & \cdots & \cdots & u_k \\ & \vdots & \vdots & \vdots & \vdots & \\ u_{d-1} & \cdots & u_{2d-2} \cdots & \cdots & \cdots & u_{k+d-2} \end{bmatrix}.$$

The matrix $\mathbf{H}_k$ is built upon the first terms of the sequence $(u_n)_{n \geq 0}$ satisfying recurrence (1.1) with characteristic polynomial $\Gamma = x^d - \sum_{i=0}^{d-1} c_i x^i$, or equivalently, from the power series expansion of the rational function P/Q with $Q(x) = x^d \Gamma(1/x)$.

Note that the entries of $\mathbf{H}_k$ can be computed either from P and Q, or from the recurrence (1.1) together with the initial terms $u_0, \dots, u_{d-1}$, using $O((k+d)\,\mathsf{M}(d)/d)$ arithmetic operations, by the algorithm in [69, Thm. 3.1], see also [8, §5]. To compute $u_N, \dots, u_{N+k-1}$ it thus only remains to perform the vector-matrix product (3.6).

When $k = 1$, the product $\mathbf{r} \times \mathbf{H}_1$ costs $2d$ operations and it yields the term u_N.

When $k \geq d$, the product $\mathbf{r} \times \mathbf{H}_k$ can be reduced to the polynomial multiplication of $r_{d-1} + \cdots + r_0 x^{d-1}$ by $\sum_{i=0}^{k+d-2} u_i x^i$, and this can be performed using $\lceil \frac{k+d}{d} \rceil \mathsf{M}(d)$ arithmetic operations. As a consequence, the whole slice of coefficients $u_{N+i} = [x^{N+i}]P/Q$ for $i = 0, \dots, k-1$, can be computed using Algorithm 8 and eq. (3.6) for a total cost of arithmetic operations of

$$(2\,\mathsf{M}(d) + d) \log N + O\left(\frac{k+d}{d}\,\mathsf{M}(d)\right).$$

When $k = d$, this proves Theorem 3. The corresponding algorithm is given as Algorithm 9.

We emphasize that this variant of Fiduccia's algorithm is different from Algorithm 1. It is actually a bit slower than Algorithm 1 when $k = 1$. However, when $k > 1$ terms are to be computed, it should be preferred to repeating k times Algorithm 1. It also compares favorably with Fiduccia's original algorithm, whose adaption to k terms has arithmetic complexity

$$3\,\mathsf{M}(d) \log N + O\left(d \log(N) + \frac{k+d}{d}\,\mathsf{M}(d)\right).$$

3.4 Applications In this section, we discuss more applications of the MSB-first algorithms (Algorithm 5 and 8) presented in §3.1 and §3.2. We deal with the case of multiplicities (§3.4.1), and explain a new way to speed up computations in that case. Then, we address another application, to faster powering of matrices (§3.4.2).

To simplify matters, we assume in this section that $R = \mathbb{K}$ is a field.

3.4.1 The case with multiplicities Hyun and his co-authors [42, 41] addressed the following question: is it possible to compute faster the N-th term of a linearly recurrent sequence when the characteristic polynomial of the recurrence has multiple roots? By the Chinese Remainder Theorem, it is sufficient to focus on the case where the characteristic polynomial is a pure power of a squarefree polynomial. In other words, the main step of [42, Algorithm 1] is to compute $x^N \bmod Q$, where $Q = (Q^\star)^m$ and $Q^\star$ is the squarefree part of Q. Under suitable invertibility conditions, the problem is solved in [42, 41] in $O(\mathsf{M}(d^\star) \log N + \mathsf{M}(d) \log d)$ operations in $\mathbb{K}$, where $d^\star = \deg(Q^\star)$ and $d = \deg(Q) = m \cdot d^\star$. This cost is obtained using an algorithm based on bivariate computations, using the isomorphisms between $\mathbb{K}[x]/(Q)$ and $\mathbb{K}[y,x]/(Q^\star(y), (x-y)^m)$ made effective by the so-called *tangling* / *untangling* operations. We now propose an alternatively fast, but simpler, algorithm with the same cost.

Let us explain this on an example, for "multiple-Fibonacci numbers", that is when Q has the form $(Q^\star)^m$, with $Q^\star = 1 - x - x^2$ and $d^\star = 2, d = 2m$.

Assume we want to compute the N-th coefficient u_N in the power series expansion of $(x/(1-x-x^2))^m$.

The cost of Fiduccia's algorithm, and also of our new algorithms, is $O(\mathsf{M}(m) \cdot \log N)$.

Let us explain how we can lower this to $O(\log N + \mathsf{M}(m) \log m)$. The starting point is the observation that, by the structure theorem of linearly recurrent sequences [15, §A.(I)] (see also [62, §2]), u_N is of the form $w_m(N)\phi^N + v_m(N)\psi^N$, where ϕ and ψ are the two roots of $1+x = x^2$ and w_m, v_m are polynomials in $\overline{\mathbb{K}}[x]$ of degree less than m. By an easy liner algebra argument, u_N is thus equal to $W_m(N)F_N + V_m(N)F_{N+1}$, where $W_m(x)$ and $V_m(x)$ are polynomials in $\mathbb{K}[x]$ of degree less than m. These polynomials can be computed by (structured) linear algebra from the first $2m$ values of the sequence (u_n), in complexity $O(\mathsf{M}(m) \log m)$.

For instance, when $d = 2$, we have $U_2(x) = -(x+1)/5$ and $V_2(x) = 2x/5$. Once U_m and V_m are determined, it remains to compute F_N and F_{N+1} using Algorithm 9 in $O(\log N)$ operations in $\mathbb{K}$, then to return the value $W_m(N) \cdot F_N + V_m(N) \cdot F_{N+1}$.

The arguments extend to the general case and yields an algorithm of arithmetic complexity $(2\,\mathsf{M}(d^\star) + d^\star) \log N + O(\mathsf{M}(d) \log d)$.

3.4.2 Faster powering of matrices

Assume we are given a matrix $M \in \mathcal{M}_d(\mathbb{K})$, an integer N, and that we want to compute the N-th power M^N of M.

The arithmetic complexity of binary powering in $\mathcal{M}_d(\mathbb{K})$ is $O(d^\theta \log N)$ operations in $\mathbb{K}$, where as before $\theta \in [2, 3]$ is any feasible exponent for matrix multiplication in $\mathcal{M}_d(\mathbb{K})$.

A better algorithm consists in first computing the characteristic polynomial $\Gamma(x)$ of the matrix M, then the remainder $\rho(x) := x^N \bmod \Gamma(x)$, and finally evaluating the polynomial $\rho(x)$ at M. By the Cayley-Hamilton theorem, $\rho(M) = M^N$. The most costly step is the computation of ρ, which can be done as explained in §3.2 using $\sim 2\,\mathsf{M}(d) \log(N)$ operations in $\mathbb{K}$. The cost of the other two steps is independent of N, and it is respectively $O(d^\theta \log d)$ [43] and $O(d^{\theta+\frac{1}{2}})$, this last cost being achieved using the Paterson-Stockmeyer *baby-step/giant-step* algorithm [59]. The total cost of this algorithm is $(2\,\mathsf{M}(d) + d) \log(N) + O(d^{\theta+\frac{1}{2}})$.

Note that a faster variant (w.r.t. d), of cost $(2\,\mathsf{M}(d) + d) \log(N) + O(d^\theta \log d)$, can be obtained using [28, Corollary 7.4]. The corresponding algorithm is based on the computation of the Frobenius (block-companion) form of the matrix M, followed by the powering of companion matrices, which again reduces to modular exponentiation.

4 Analysis under the FFT multiplication model

In this section, we specialize, optimize and analyze the generic Algorithm 1 to the FFT setting, in which polynomial products are assumed to be performed using the discrete Fourier transform (DFT), and its inverse.

To do this, we will assume that the base ring R possesses roots of unity of sufficiently high order. To simplify the exposition, R will be supposed to be a field, but the arguments also apply without this assumption, modulo some technical complications, see [27, §8.2].

4.1 Discrete Fourier Transform for polynomial products

Let $\mathbb{K}$ be a field with a primitive n-th root ω_n of unity. Let $A \in \mathbb{K}[x]$ be a polynomial of degree at most $d \le n-1$. The DFT $\widehat{A}$ of A is defined by

$$\widehat{A}_y := A(\omega_n^{-y}) = \sum_{i=0}^{n-1} A_i \omega_n^{-yi} \qquad \text{for } y = 0, 1, \ldots, n-1.$$

Here, $A_y = 0$ for $y > d$. It is classical that the DFT map is an invertible $\mathbb{K}$-linear transform from $\mathbb{K}^n$ to itself, and that the polynomial A can be retrieved from its DFT $\widehat{A}$ using the formulas

$$A_i = \frac{1}{n} \sum_{y=0}^{n-1} \widehat{A}_y \omega_n^{yi} \qquad \text{for } i = 0, 1, \ldots, n-1.$$

For computing the polynomial multiplication $C(x) = A(x)B(x)$ for given $A(x), B(x) \in \mathbb{K}[x]$ of degree at most d, it is sufficient to compute the DFT of $C(x)$ for $n \ge 2d+1$. Since $\widehat{C}_y = C(\omega_n^{-y}) = A(\omega_n^{-y})B(\omega_n^{-y}) = \widehat{A}_y \widehat{B}_y$, the polynomial $C(x)$ can be computed using two DFTs and one inverse DFT.

Let $\mathsf{E}(n)$ be an arithmetic complexity for computing a DFT of length n. Then the cost of polynomial multiplication in $\mathbb{K}[x]$ is governed by

$$\mathsf{M}(d) = 3\,\mathsf{E}(2d) + O(d).$$

4.2 Fast Fourier Transform

In this subsection, we briefly recall the Fast Fourier Transform (FFT), which gives the quasi-linear estimate $\mathsf{E}(n) = O(n \log n)$.

Assume n is even. Then, for $y = 0, 1, \ldots, n/2-1$ we have

$$\begin{aligned}
\widehat{A}_y &= \sum_{i=0}^{n/2-1} A_{2i} \omega_n^{-y(2i)} + \sum_{i=0}^{n/2-1} A_{2i+1} \omega_n^{-y(2i+1)} \\
&= \sum_{i=0}^{n/2-1} A_{2i} \omega_{n/2}^{-yi} + \omega_n^{-y} \cdot \sum_{i=0}^{n/2-1} A_{2i+1} \omega_{n/2}^{-yi} \\
&= \widehat{A^{\mathrm{e}}}_y + \omega_n^{-y} \widehat{A^{\mathrm{o}}}_y
\end{aligned}$$

for $A^{\mathrm{e}}(x) := \sum_{i=0}^{n/2} A_{2i} x^i$ and $A^{\mathrm{o}}(x) := \sum_{i=0}^{n/2} A_{2i+1} x^i$.

Similarly, we have $\widehat{A}_{n/2+y} = \widehat{A}^e_y - \omega_n^{-y} \widehat{A}^o_y$. We

therefore obtain the following matrix equation

$$\begin{bmatrix} \widehat{A}_y \\ \widehat{A}_{n/2+y} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \omega_n^{-y} \end{bmatrix} \begin{bmatrix} \widehat{A}_y^{\mathrm{e}} \\ \widehat{A}_y^{\mathrm{o}} \end{bmatrix} \quad \text{for } y = 0, 1, \ldots, n/2 - 1. \tag{4.7}$$

Thus, computing a DFT in size n reduces to two DFTs in size $n/2$. More precisely, $\mathsf{E}(n) \leq 2\,\mathsf{E}(n/2) + (3/2)n$. If n is a power of two, $n = 2^k$, and if the field $\mathbb{K}$ contains a primitive 2^k-th root of unity (as is the case for instance when $\mathbb{K} = \mathbb{C}$, $\mathbb{K} = \mathbb{F}_p$ for a prime number p satisfying $2^k \mid p - 1$), this reduction can be repeated $k = \log n$ times, and it yields the estimate $\mathsf{E}(n) = \frac{3}{2} n \log n$. The corresponding algorithm is called the *decimation-in-time* Cooley–Tukey fast Fourier transform [17], see also [5, §2].

By the arguments of §4.1, we conclude that polynomial multiplication in $\mathbb{K}[x]$ can be performed in arithmetic complexity

$$\mathsf{M}(d) = 9\, d \log d + O(d).$$

4.3 Efficiently doubling the length of a DFT In the FFT setting, it is useful for many applications to compute efficiently a DFT of length $2n$ starting from DFT of length n.

Assume $n \geq d + 1$ and we have at our disposal the DFT $\widehat{A}$ of A, of length n. Assume that we want to compute the DFT $\widehat{A}^{(2n)}$ of length $2n$.

The simplest algorithm is to apply the inverse DFT of length n to obtain A, and then to apply the DFT of length $2n$ to A. This costs $\mathsf{E}(n) + \mathsf{E}(2n)$ arithmetic operations, that is $\frac{9}{2} n \log n + 3n$ operations in $\mathbb{K}$.

This algorithm can be improved using the following formulas

$$\widehat{A}_{2y}^{(2n)} = \sum_{i=0}^{2n-1} A_i \omega_{2n}^{-2yi} = \sum_{i=0}^{n-1} A_i \omega_n^{-yi} = \widehat{A}_y,$$
$$\widehat{A}_{2y+1}^{(2n)} = \sum_{i=0}^{2n-1} A_i \omega_{2n}^{-(2y+1)i} = \sum_{i=0}^{n-1} \omega_{2n}^{-i} A_i \omega_n^{-yi} = \widehat{B}_y,$$

where $B_i := \omega_{2n}^{-i} A_i$ for $i = 0, 1, \ldots, n-1$. We obtain Algorithm 10 with arithmetic complexity $2\,\mathsf{E}(n) + n$, *i.e.* $3n \log n + n$, [5, §12.8], see also [4, 52][10]. Compared with the direct algorithm, the gain is roughly a factor of 3/2.

4.4 Algorithm 1 in the FFT setting Recall that our main objective is, given P, Q in $\mathbb{K}[x]$ with $d = \deg(Q) > \deg(P)$, to compute the N-th coefficient u_N in the series expansion of P/Q.

Algorithm 10 **Input**: $\mathrm{DFT}_n(A)$ **Output**: $\mathrm{DFT}_{2n}(A)$

function $\mathsf{DoubleDFT}(\widehat{A})$
  $A \leftarrow \mathrm{IDFT}_n(\widehat{A})$
  $B_i \leftarrow \omega_{2n}^{-i} A_i$ for $i = 0, 1, \ldots, n-1$
  $\widehat{B} \leftarrow \mathrm{DFT}_n(B)$
  $\widehat{A}_{2y}^{(2n)} \leftarrow \widehat{A}_y$ for $y = 0, 1, \ldots, n-1$
  $\widehat{A}_{2y+1}^{(2n)} \leftarrow \widehat{B}_y$ for $y = 0, 1, \ldots, n-1$
  return $\widehat{A}^{(2n)}$

Let k be the minimum integer satisfying $2^k \geq 2d+1$. Assume that there exists a primitive 2^k-th root of unity in $\mathbb{K}$. In this case, we can employ an FFT-based polynomial multiplication in $\mathbb{K}[x]$. In each iteration of Algorithm 1, it is sufficient to compute $P(x)Q(-x)$ and $Q(x)Q(-x)$. Here, only two FFTs and two inverse FFT of length 2^k are needed since $\widehat{Q}_y^- = \widehat{Q}_{\bar{y}}$ for $Q^-(x) := Q(-x)$ where $\bar{y} := y + 2^{k-1}$ if $y < 2^{k-1}$ and $\bar{y} := y - 2^{k-1}$ if $y \geq 2^{k-1}$. Hence, the arithmetic complexity $\mathsf{S}(d)$ for a single step in Algorithm 1 satisfies $\mathsf{S}(d) \leq 4\,\mathsf{E}(2^k) + O(2^k)$.

In the following we will show the improved estimate

$$\mathsf{S}(d) \leq 4\,\mathsf{E}(2^{k-1}) + O(2^k).$$

Before entering the **while** loop in Algorithm 1, the DFTs $\widehat{P}$ and $\widehat{Q}$ of $P(x)$ and $Q(x)$ of length 2^k are computed, respectively. Inside the **while** loop, $\widehat{P}$ and $\widehat{Q}$ are updated. The recursive formula (4.7) for the decimation-in-time Cooley–Tukey FFT is equivalent to

$$\begin{bmatrix} \widehat{A}_y^{\mathrm{e}} \\ \widehat{A}_y^{\mathrm{o}} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 0 \\ 0 & \omega_n^{y} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \widehat{A}_y \\ \widehat{A}_{2^{k-1}+y} \end{bmatrix} \quad \text{for } y = 0, 1, \ldots, 2^{k-1}.$$

By using this formula, $\widehat{A}^{\mathrm{e}}$ (or $\widehat{A}^{\mathrm{o}}$) can be computed with $O(2^k)$ operations from $\widehat{A}$. By using Algorithm 10, we obtain the updated $\widehat{P}$ from $\widehat{A}^{\mathrm{e}}$ or $\widehat{A}^{\mathrm{o}}$. The algorithm is summarized in Algorithm 11. In each step, $\mathsf{DoubleDFT}$ is called twice. Hence, the total arithmetic complexity of Algorithm 11 is

$$(4\,\mathsf{E}(2^{k-1}) + O(2^k)) \cdot \log N.$$

When d is of the form $2^\ell - 1$, then[11] one can take $k = \ell + 1$ and the cost simplifies to

$$\mathsf{T}(N, d) = 4\,\mathsf{E}(d) \log N + O(d \log N),$$

[10] This "FFT doubling" trick is sometimes attributed to R. Kramer (2004), but we were not able to locate Kramer's paper.

[11] In the general case, it might be useful to use the Truncated Fourier Transform (TFT), which smoothes the "jumps" in complexity exhibited by FFT algorithms [38, 37, 3].

Algorithm 11 (OneCoeff-FFT)
Input: $P(x)$, $Q(x)$, N **Output**: $[x^N]\,\frac{P(x)}{Q(x)}$

$\widehat{P} \leftarrow \mathrm{DFT}_{2^k}(P)$
$\widehat{Q} \leftarrow \mathrm{DFT}_{2^k}(Q)$
while $N \geq 1$ **do**
  $\widehat{U}_y \leftarrow \widehat{P}_y\,\widehat{Q}_{\bar{y}}$ for $y = 0, 1, \ldots, 2^k - 1$
  if N is even **then**
    $\widehat{U}^{\mathrm{e}}_y \leftarrow (\widehat{U}_y + \widehat{U}_{y+2^{k-1}})/2$ for $y = 0, 1, \ldots, 2^{k-1} - 1$
    $\widehat{P} \leftarrow \mathsf{DoubleDFT}(\widehat{U}^{\mathrm{e}}_y)$
  else
    $\widehat{U}^{\mathrm{o}}_y \leftarrow \omega^y_{2^k}(\widehat{U}_y - \widehat{U}_{y+2^{k-1}})/2$ for $y = 0, 1, \ldots, 2^{k-1} - 1$
    $\widehat{P} \leftarrow \mathsf{DoubleDFT}(\widehat{U}^{\mathrm{o}}_y)$
  $\widehat{A}_y \leftarrow \widehat{Q}_y\,\widehat{Q}_{\bar{y}}$ for $y = 0, 1, \ldots, 2^{k-1} - 1$
  $\widehat{Q} \leftarrow \mathsf{DoubleDFT}(\widehat{A})$
  $N \leftarrow \lfloor N/2 \rfloor$
$P(0) \leftarrow \sum_{y=0}^{2^k-1} \widehat{P}_y$
$Q(0) \leftarrow \sum_{y=0}^{2^k-1} \widehat{Q}_y$
return $P(0)/Q(0)$

or, equivalently

$$\mathsf{T}(N, d) = 6\, d \log d \log N + O(d \log N).$$

The (striking) conclusion of this analysis is that, in the FFT setting, our (variant of the) algorithm for computing the N-th term of P/Q uses much less operations than in the general case, namely

$$\mathsf{T}(N, d) \sim \frac{2}{3}\,\mathsf{M}(d) \log N, \tag{4.8}$$

while for a generic multiplication algorithm the cost is $\sim 2\,\mathsf{M}(d) \log N$. This proves Theorem 2.

Note that the complexity bound (4.8) compares favorably with Fidducia's algorithm combined with the best algorithms for modular squaring. For instance, Shoup's algorithm [70, §7.3] computes one modular squaring in the FFT setting using $\sim \frac{5}{3}\,\mathsf{M}(d)$ arithmetic operations, while Mihăilescu's algorithm [53, Table 1] (based on Montgomery's algorithm [55]) uses roughly $\sim \frac{13}{12}\,\mathsf{M}(d)$ arithmetic operations. Our bound (4.8) is better by a factor of 2.5 than Shoup's (comparatively simple) algorithm, and by a factor of 1.625 than the (much more complex) algorithm by Mihăilescu.

Let us point out that all the other algorithms admit similarly fast versions in the FFT setting. We will however not give them in full detail here, mainly for space reasons.

4.5 Algorithm 5 in the FFT setting In this section, we present Algorithm 5 in the FFT setting with arithmetic complexity $((2/3)\mathsf{M}(d) + O(d)) \log N$. Algorithm 5 includes one standard product $Q(x)Q(-x)$ and one middle-product $Q(-x)S(x)$. The standard product can be performed by two FFTs of length 2^{k-1} as in Algorithm 11. For the middle-product, we first compute the cyclic product. i.e., the product in $\mathbb{K}[x]/(x^{2^k} - 1)$, between $Q(-x)$ and $S(x)$, and then extract the second half of the cyclic product. For computing the DFT of the second half $A_{\mathrm{sec}}(x) := \sum_{i=0}^{2^{k-1}-1} A_{i+2^{k-1}} x^i$, we use the formula for the decimation-in-frequency FFT.

Algorithm 12
Input: $\mathrm{DFT}_{2^k}(A)$ **Output**: $\mathrm{DFT}_{2^{k-1}}(A_{\mathsf{sec}})$

function $\mathsf{SecondHalfDFT}(\widehat{A})$
  $\widehat{B}_y \leftarrow \widehat{A}_{2y+1}$ for $y = 0, 1, \ldots, 2^{k-1} - 1$
  $B \leftarrow \mathrm{IDFT}_{2^{k-1}}(\widehat{B})$
  $C_i \leftarrow \omega^i_{2^k} B_i$ for $i = 0, 1, \ldots, 2^{k-1} - 1$
  $\widehat{C} \leftarrow \mathrm{DFT}_{2^{k-1}}(C)$
  $\widehat{D}_y \leftarrow (\widehat{A}_{2y} - \widehat{C}_y)/2$ for $y = 0, 1, \ldots, 2^{k-1} - 1$
  return $\widehat{D}$

$$\widehat{A}_{2y} = \sum_{i=0}^{2^k-1} \omega^{-2iy}_{2^k} A_i = \sum_{i=0}^{2^{k-1}-1} \omega^{-iy}_{2^{k-1}} (A_i + A_{i+2^{k-1}})$$

$$\widehat{A}_{2y+1} = \sum_{i=0}^{2^k-1} \omega^{-i(2y+1)}_{2^k} A_i = \sum_{i=0}^{2^{k-1}-1} \omega^{-iy}_{2^{k-1}} \omega^{-i}_{2^k} (A_i - A_{i+2^{k-1}}).$$

This formula gives Algorithm 12 which transforms $\widehat{A}$ to $\widehat{A_{\mathrm{sec}}}$. The arithmetic complexity of Algorithm 12 is $2\mathsf{E}(2^{k-1}) + O(2^k)$. The whole algorithm is shown in Algorithm 13. The input of the algorithm is N and $\widehat{Q}$ that is the DFT of Q of length 2^k. The output of the algorithm is the DFT of $\mathcal{F}_{N,2^{k-1}}(1/Q(x))$ of length 2^{k-1} rather than length 2^k since the DFT of $W(x^2)$ of length 2^k can be obtained efficiently from the DFT of $W(x)$ of length 2^{k-1}. Algorithm 13 calls one $\mathsf{DoubleDFT}$ and one $\mathsf{SecondHalfDFT}$. Hence, similarly to Algorithm 11, the arithmetic complexity of Algorithm 13 is $(4\mathsf{E}(2^{k-1}) + O(2^k)) \log N$.

5 Conclusion

We have proposed several algorithmic contributions to the classical field of linearly recurrent sequences.

Firstly, we have designed a simple and fast algorithm for computing the N-th term of a linearly recurrent sequence of order d, using $\sim 2\,\mathsf{M}(d) \log N$ arithmetic operations, which is faster by a factor of 1.5 than the state-of-the-art 1985 algorithm due to Fiduccia [26]. When combined with FFT techniques, the algorithm has even better arithmetic complexity $\sim \frac{2}{3}\,\mathsf{M}(d) \log N$

Algorithm 13 (SliceCoeff-FFT) **Input**: $\mathrm{DFT}_{2^k}(Q)$, N
Output: $\mathrm{DFT}_{2^{k-1}}(\mathcal{F}_{N,2^{k-1}}(1/Q(x)))$

function SliceCoeff-FFT$(N, \widehat{Q})$
  if $N = 0$ **then**
    $Q(0) \leftarrow \sum_{y=0}^{2^k-1} \widehat{Q}_y$
    return $[1\ \omega_{2^k}\ \omega_{2^k}^2\ \cdots \omega_{2^k}^{2^k-1}]/Q(0)$
  $\widehat{A}_y \leftarrow \widehat{Q}_y\, \widehat{Q}_{\bar{y}}$ for $y = 0, 1, \ldots, 2^{k-1} - 1$
  $\widehat{V} \leftarrow$ DoubleDFT$(\widehat{A})$
  $\widehat{W} \leftarrow$ SliceCoeff-FFT$(\lfloor N/2 \rfloor, \widehat{V})$
  if N is even **then**
    $\widehat{S}_y \leftarrow \omega_{2^k}^{-y} \widehat{W}_{y \bmod 2^{k-1}}$ for $y = 0, 1, \ldots, 2^k - 1$
  else
    $\widehat{S}_y \leftarrow \widehat{W}_{y \bmod 2^{k-1}}$ for $y = 0, 1, \ldots, 2^k - 1$
  $\widehat{B}_y \leftarrow \widehat{S}_y\, \widehat{Q}_{\bar{y}}$ for $y = 0, 1, \ldots, 2^k - 1$
  return SecondHalfDFT$(\widehat{B})$

which is faster than the fastest variant of Fiduccia's algorithm in the FFT setting by a factor of 1.625. The new algorithms are based on a new method (Algorithm 1) for computing the N-th coefficient of a rational power series.

Secondly, using algorithmic transposition techniques, we have derived from Algorithm 1 a new method (Algorithm 5) for computing simultaneously the coefficients of indices $N - d + 1, \ldots, N$ in the power series expansion of the reciprocal of a degree-d polynomial, using again $\sim 2\,\mathsf{M}(d) \log N$ arithmetic operations. Using Algorithm 5, we have designed a new algorithm for computing the remainder of x^N modulo a given polynomial of degree d, using $\sim 2\,\mathsf{M}(d) \log N$ arithmetic operations as well. This is better by a factor of 1.5 than the previous best algorithm for modular exponentiation, with an even better speed-up in the FFT setting, as for Algorithm 1. Combined with the basic idea of Fiduccia's algorithm, our new algorithm for modular exponentiation yields a faster Fiduccia-like algorithm (by the aforementioned constant factors) that computes a slice of d consecutive terms (of indices $N - d + 1, \ldots, N$) of a linearly recurrent sequence of order d using $\sim 2\,\mathsf{M}(d) \log N$ arithmetic operations.

Thirdly, we have discussed applications of the new algorithms to a few other algorithmic problems, including powering of matrices and the computation of terms of linearly recurrent sequences when the recurrence has roots with (high) multiplicities.

As future work, we plan to investigate further the full power of our technique. To which extent can it be generalized to larger classes of power series? For instance, although it perfectly works for bivariate rational power series $U(x, y)$, the corresponding method does not directly provide a $O(\log N)$-algorithm for computing the (N, N)-th coefficient $u_{N,N}$, the reason being that the $\log N$ new bivariate recurrences produced by the Graeffe process do not have constant orders, as in the univariate case. This is disappointing, but after all not surprising, because the generating function of the sequence $(u_{n,n})_n$ is known to be algebraic, but not rational anymore [61]. As of today, no algorithm is known for computing the N-th coefficient of an algebraic power series faster than in the P-recursive case (P), namely in a number of ring operations almost linear in $\sqrt{N}$.

Acknowledgements. We are grateful to the three reviewers for their kind and constructive remarks. Our special thanks go to Kevin Atienza, whose editorial on https://discuss.codechef.com was our initial source of inspiration, and to Bruno Salvy and Sergey Yurkevich, for their careful reading of a first draft of this work. A. Bostan was supported in part by DeRerumNatura ANR-19-CE40-0018. R. Mori was supported in part by JST PRESTO Grant #JPMJPR1867 and JSPS KAKENHI Grant #JP17K17711, #JP18H04090 and #JP20H04138.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Ann. of Math. (2)*, 160(2):781–793, 2004.

[2] J.-P. Allouche and J. Shallit. The ring of k-regular sequences. *Theoret. Comput. Sci.*, 98(2):163–197, 1992.

[3] A. Arnold. A new truncated Fourier transform algorithm. In *Proc. ISSAC'13*, pages 15–22. ACM, 2013.

[4] D. J. Bernstein. Removing redundancy in high-precision Newton iteration, 2004. Preprint, http://cr.yp.to/fastnewton.html.

[5] D. J. Bernstein. Fast multiplication and its applications. In *Algorithmic number theory: lattices, number fields, curves and cryptography*, volume 44 of *Math. Sci. Res. Inst. Publ.*, pages 325–384. Cambridge Univ. Press, 2008.

[6] D. Bini and V. Y. Pan. *Polynomial and matrix computations. Vol. 1.* Progress in Theoretical Computer Science. Birkhäuser, 1994. Fundamental algorithms.

[7] I. Blake, G. Seroussi, and N. Smart. *Elliptic curves in cryptography*, volume 265 of *London Math. Soc. Lecture Note Ser.* Cambridge University Press, 1999.

[8] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *Proc. ISSAC'03*, pages 37–44. ACM, 2003.

[9] A. Bostan. Computing the N-th Term of a q-Holonomic Sequence. In *Proc. ISSAC'20*, pages 46–53. ACM, 2020.

[10] A. Bostan, X. Caruso, G. Christol, and Ph. Dumas. Fast coefficient computation for algebraic power series in positive characteristic. In *Proceedings of the Thirteenth Algorithmic Number Theory Symposium*, vol-

ume 2 of *Open Book Ser.*, pages 119–135. Math. Sci. Publ., Berkeley, CA, 2019.

[11] A. Bostan, G. Christol, and Ph. Dumas. Fast computation of the Nth term of an algebraic series over a finite prime field. In *Proc. ISSAC'16*, pages 119–126. ACM, 2016.

[12] A. Bostan, P. Gaudry, and É. Schost. Linear recurrences with polynomial coefficients and application to integer factorization and Cartier-Manin operator. *SIAM J. Comput.*, 36(6):1777–1806, 2007.

[13] R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun. Fast solution of Toeplitz systems of equations and computation of Padé approximants. *J. Algorithms*, 1(3):259–295, 1980.

[14] N. Calkin, J. Davis, K. James, E. Perez, and C. Swannack. Computing the integer partition function. *Math. Comp.*, 76(259):1619–1638, 2007.

[15] L. Cerlienco, M. Mignotte, and F. Piras. Suites récurrentes linéaires. Propriétés algébriques et arithmétiques. *L'Enseign. Mathématique*, 33:67–108, 1987.

[16] D. V. Chudnovsky and G. V. Chudnovsky. Approximations and complex multiplication according to Ramanujan. In *Ramanujan revisited (Urbana-Champaign, Ill., 1987)*, pages 375–472. Academic Press, Boston, MA, 1988.

[17] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[18] P. Cull and J. L. Holloway. Computing Fibonacci numbers quickly. *Inform. Process. Lett.*, 32(3):143–149, 1989.

[19] G. de Rocquigny. Question 1541. [I25a]. *L'Intermédiaire des mathématiciens*, 6:148, 1899.

[20] L. E. Dickson. *History of the theory of numbers. Vol. I: Divisibility and primality.* Publication No. 256. Carnegie Institution of Washington, Vol. 1, 1919.

[21] E. W. Dijkstra. In honour of Fibonacci. In *Program Construction. Lecture Notes in Comput. Sci. 69,*, pages 49–50. Springer, Berlin, Heidelberg, 1979.

[22] M. C. Er. Computing sums of order-k Fibonacci numbers in log time. *Inform. Process. Lett.*, 17(1):1–5, 1983.

[23] M. C. Er. A formal derivation of an $O(\log n)$ algorithm for computing Fibonacci numbers. *J. Inform. Optim. Sci.*, 7(1):9–15, 1986.

[24] M. C. Er. An $O(k^2 \log(n/k))$ algorithm for computing generalized order-k Fibonacci numbers with linear space. *J. Inform. Optim. Sci.*, 9(3):343–353, 1988.

[25] C. M. Fiduccia. The n-th power of a companion matrix: Fast solutions to linear recurrences. 20th Proc. of the Annual Allerton Conference on Communication, Control and Computing, pages 934–940, 1982.

[26] C. M. Fiduccia. An efficient formula for linear recurrences. *SIAM J. Comput.*, 14(1):106–112, 1985.

[27] J. von zur Gathen and J. Gerhard. *Modern computer algebra.* Cambridge Univ. Press, third edition, 2013.

[28] M. Giesbrecht. Nearly optimal algorithms for canonical matrix forms. *SIAM J. Comput.*, 24(5):948–969, 1995.

[29] K. J. Giuliani and G. Gong. New LFSR-Based Cryptosystems and the Trace Discrete Log Problem (Trace-DLP). In *Sequences and Their Applications - SETA 2004*, pages 298–312, Berlin, Heidelberg, 2005.

[30] K. J. Giuliani and G. Gong. A new algorithm to compute remote terms in special types of characteristic sequences. In *Sequences and their applications—SETA 2006*, volume 4086 of *Lecture Notes in Comput. Sci.*, pages 237–247. Springer, Berlin, 2006.

[31] G. Gong and L. Harn. Public-key cryptosystems based on cubic finite field extensions. *IEEE Trans. Inform. Theory*, 45(7):2601–2605, 1999.

[32] G. Gong, L. Harn, and H. Wu. The GH public-key cryptosystem. In *Selected areas in cryptography*, volume 2259 of *Lecture Notes in Comput. Sci.*, pages 284–300. Springer, Berlin, 2001.

[33] S. González, L. Huguet, C. Martínez, and H. Villafañe. Discrete logarithm like problems and linear recurring sequences. *Adv. Math. Commun.*, 7(2):187–195, 2013.

[34] D. Gries and G. Levin. Computing Fibonacci numbers (and similarly defined functions) in log time. *Inform. Process. Lett.*, 11(2):68–69, 1980.

[35] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm. I. *Appl. Algebra Engrg. Comm. Comput.*, 14(6):415–438, 2004.

[36] D. Harvey. Counting points on hyperelliptic curves in average polynomial time. *Ann. of Math. (2)*, 179(2):783–803, 2014.

[37] D. Harvey and D. S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In *Proc. ISSAC'10*, pages 325–329. ACM, 2010.

[38] J. van der Hoeven. The truncated Fourier transform and applications. In *Proc. ISSAC'04*, pages 290–296. ACM, 2004.

[39] J. L. Holloway. Algorithms for computing Fibonacci numbers quickly. MSc Thesis. Oregon State Univ. 1988.

[40] A. S. Householder. Dandelin, Lobačevskiĭ, or Graeffe? *Amer. Math. Monthly*, 66:464–466, 1959.

[41] S. G. Hyun, S. Melczer, É. Schost, and C. St-Pierre. Change of basis for $\mathfrak{m}$-primary ideals in one and two variables. In *Proc. ISSAC'19*, pages 227–234. ACM, 2019.

[42] S. G. Hyun, S. Melczer, and C. St-Pierre. A fast algorithm for solving linearly recurrent sequences. *ACM Commun. Comput. Algebra* , page 100–103, 2019.

[43] W. Keller-Gehrig. Fast algorithms for the characteristic polynomial. *Theoret. Comput. Sci.*, 36(2-3):309–317, 1985.

[44] D. I. Khomovsky. Efficient computation of terms of linear recurrence sequences of any order. *Integers*, 18:Paper No. A39, 12, 2018.

[45] D. E. Knuth. *The art of computer programming. Vol. 2: Seminumerical algorithms.* Addison-Wesley Publishing Co., Reading, Mass.-London-Don Mills, Ont, first edition, 1969.

[46] D. E. Knuth. *The art of computer programming.*

Vol. 2. Addison-Wesley Publishing Co., second edition, 1981. Seminumerical algorithms.

[47] D. E. Knuth. The Last Whole Errata Catalog, 1981. Department of Computer Science, Stanford University, Report. No. STAN-CS-81-868, http://infolab.stanford.edu/pub/cstr/reports/cs/tr/81/868/CS-TR-81-868.pdf.

[48] G. Lecerf. New recombination algorithms for bivariate polynomial factorization based on Hensel lifting. *Appl. Algebra Engrg. Comm. Comput.*, 21(2):151–176, 2010.

[49] K. Lürwer-Brüggemeier and M. Ziegler. On faster integer calculations using non-arithmetic primitives. In *Unconventional computation*, volume 5204 of *Lecture Notes in Comput. Sci.*, pages 111–128. Springer, 2008.

[50] A. J. Martin and M. Rem. A presentation of the Fibonacci algorithm. *Inform. Process. Lett.*, 19(2):67–68, 1984.

[51] J. M. McNamee and V. Y. Pan. *Numerical methods for roots of polynomials. Part II*, volume 16 of *Studies in Computational Mathematics*. Elsevier/Academic Press, Amsterdam, 2013.

[52] M. Mezzarobba. NumGfun: a package for numerical and analytic computation and D-finite functions. In *Proc. ISSAC'10*, pages 139–146. ACM, 2010.

[53] P. Mihăilescu. Fast convolutions meet Montgomery. *Math. Comp.*, 77(262):1199–1221, 2008.

[54] J. C. P. Miller and D. J. Spencer Brown. An algorithm for evaluation of remote terms in a linear recurrence sequence. *Computer Journal*, 9:188–190, 1966.

[55] P. L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44(170):519–521, 1985.

[56] G. Nuel and J.-G. Dumas. Sparse approaches for the exact distribution of patterns in long state sequences generated by a Markov source. *Theoret. Comput. Sci.*, 479:22–42, 2013.

[57] V. Pan. Algebraic complexity of computing polynomial zeros. *Comput. Math. Appl.*, 14(4):285–304, 1987.

[58] V. Y. Pan. Solving a polynomial equation: some history and recent progress. *SIAM Rev.*, 39(2):187–220, 1997.

[59] M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2:60–66, 1973.

[60] A. Pettorossi. Derivation of an $O(k^2 \log n)$ algorithm for computing order-k Fibonacci numbers from the $O(k^3 \log n)$ matrix multiplication method. *Inform. Process. Lett.*, 11(4-5):172–179, 1980.

[61] G. Pólya. Sur les séries entières, dont la somme est une fonction algébrique. *Enseign. Math.*, 22:38–47, 1921/1922.

[62] A. J. van der Poorten. Some facts that should be better known, especially about rational functions. In *Number theory and applications (Banff, AB, 1988)*, volume 265 of *NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci.*, pages 497–528. Kluwer Acad. Publ., Dordrecht, 1989.

[63] M. Protasi and M. Talamo. On the number of arithmetical operations for finding Fibonacci numbers. *Theoret. Comput. Sci.*, 64(1):119–124, 1989.

[64] A. Ranum. The general term of a recurring series. *Bull. Amer. Math. Soc.*, 17(9):457–461, 1911.

[65] B. Ravikumar and G. Eisman. Weak minimization of DFA—an algorithm and applications. *Theoret. Comput. Sci.*, 328(1-2):113–133, 2004.

[66] Rosace, E.-B. Escott, E. Malo, C.-A. Laisant, and G. Picou. Answers to Question 1541. [I25a] asked by G. de Rocquigny. *L'Intermédiaire des mathématiciens*, 7:172–177, 1900.

[67] A. Schönhage. Variations on computing reciprocals of power series. *Inform. Process. Lett.*, 74(1-2):41–46, 2000.

[68] J. Shortt. An iterative program to calculate Fibonacci numbers in $O(\log n)$ arithmetic operations. *Inform. Process. Lett.*, 7(6):299–303, 1978.

[69] V. Shoup. A fast deterministic algorithm for factoring polynomials over finite fields of small characteristic. In *Proc. ISSAC'91*, pages 14–21. ACM, 1991.

[70] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symbolic Comput.*, 20(4):363–397, 1995.

[71] A. Storjohann. High-order lifting. In *Proc. ISSAC'02*, pages 246–254. ACM, 2002.

[72] V. Strassen. Einige Resultate über Berechnungskomplexität. *Jber. Deutsch. Math.-Verein.*, 78(1):1–8, 1976/77.

[73] V. Strassen. Polynomials with rational coefficients which are hard to compute. *SIAM J. Comput.*, 3:128–149, 1974.

[74] D. Takahashi. A fast algorithm for computing large Fibonacci numbers. *Inform. Process. Lett.*, 75(6):243–246, 2000.

[75] F. J. Urbanek. An $O(\log n)$ algorithm for computing the nth element of the solution of a difference equation. *Inform. Process. Lett.*, 11(2):66–67, 1980.

[76] T. C. Wilson and J. Shortt. An $O(\log n)$ algorithm for computing general order-k Fibonacci numbers. *Inform. Process. Lett.*, 10(2):68–75, 1980.