Accelerating modular arithmetic with special choice of moduli

Eugene Zima

Physics and Computer Science Department

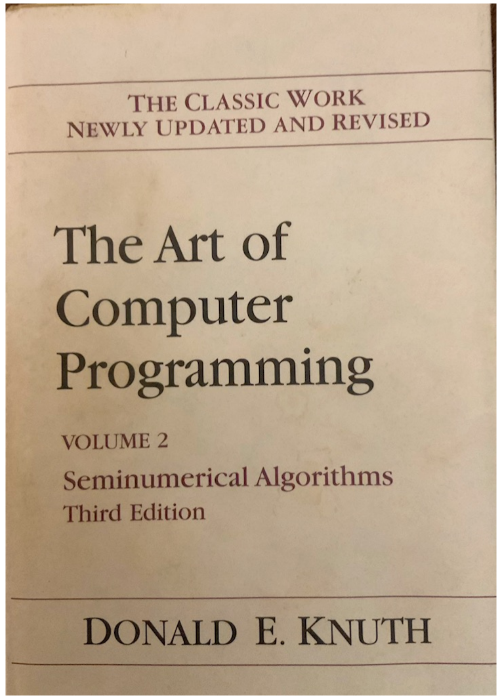
Wilfrid Laurier University, Waterloo

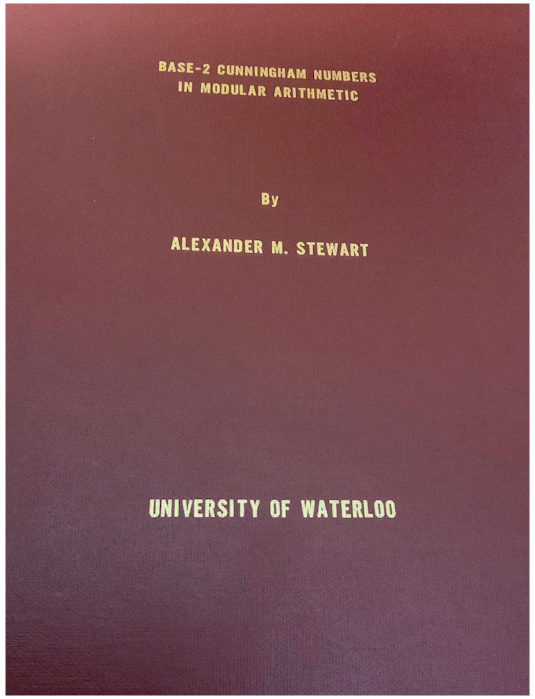
e-mail: ezima@wlu.ca

Joint work with:

Alex Stewart (2005-2007), Yu Li (2018), Benjamin Chen (2022-2023)

Modular arithmetic;
Approaches to acceleration;
Choice of moduli;
Eliminating divisions (Knuth – Mersenne; Fermat)
Reduction of large numbers (Mersenne trick)
Reconstruction
Eliminating computing inverses
Eliminating multiplications (do we really need Horner scheme?)
Experimental results (matrix multiplication)





Given input A, B of size n, compute

$$A \odot B \to C$$

Given input A, B of size n, compute

$$A \odot B \rightarrow C$$

Select $m_1, m_2, \ldots m_k$

Map the input:

$$a_i = A \mod m_i, \quad b_i = B \mod m_i, i = 1, \ldots, k$$

assuming $0 \leq a_i < m_i$, $0 \leq b_i < m_i$.

Given input A, B of size n, compute

$$A \odot B \rightarrow C$$

Select $m_1, m_2, \ldots m_k$

Map the input:

$$a_i = A \mod m_i, \quad b_i = B \mod m_i, i = 1, \ldots, k$$

assuming $0 \leq a_i < m_i$, $0 \leq b_i < m_i$.

Compute

$$a_i \odot b_i \mod m_i \rightarrow c_i, i = 1, \ldots, k$$

Given input A, B of size n, compute

$$A \odot B \to C$$

Select $m_1, m_2, \ldots m_k$

Map the input:

$$a_i = A \mod m_i, \quad b_i = B \mod m_i, i = 1, \ldots, k$$

assuming $0 \leq a_i < m_i$, $0 \leq b_i < m_i$.

Compute

$$a_i \odot b_i \mod m_i \to c_i, i = 1, \ldots, k$$

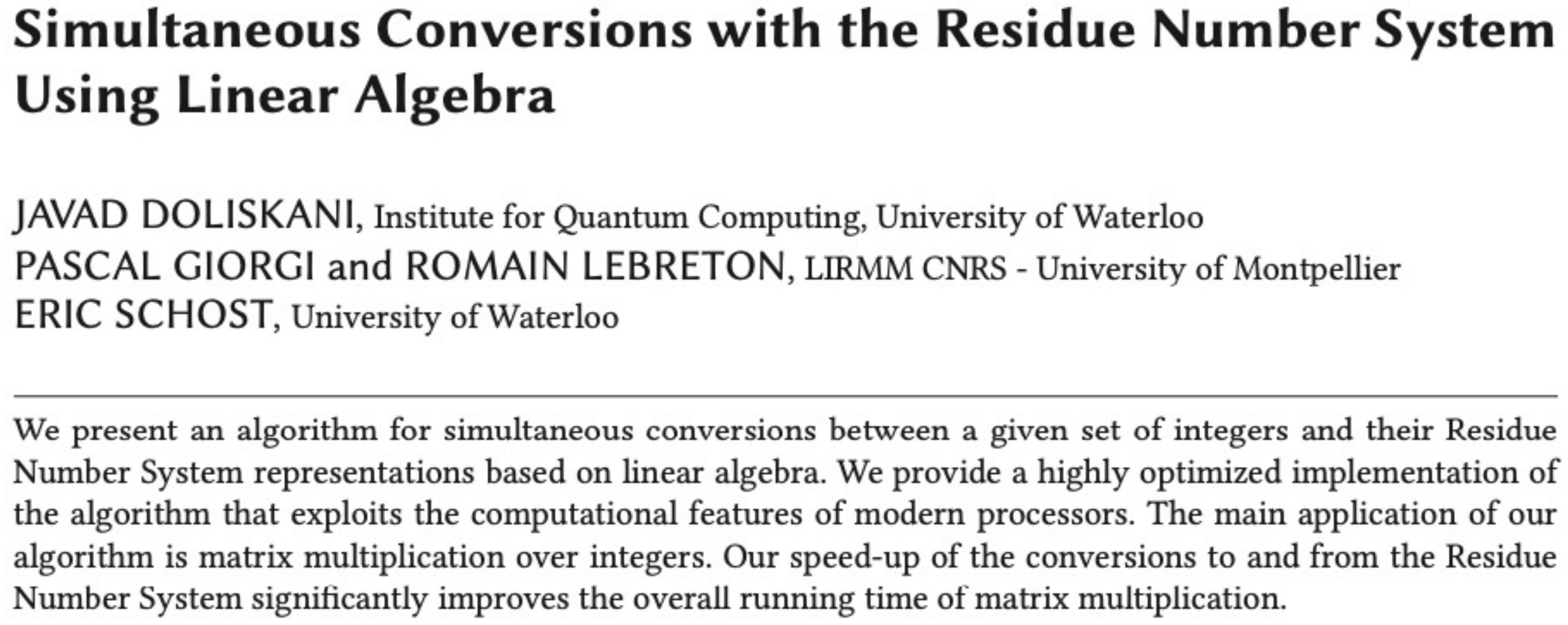
Reconstruct C from modular images, i.e. find C such that

$$C \mod m_i = c_i, i = 1, \ldots, k$$

If the complexity of $\odot$ is $M(n)$ this approach reduces it to $kM(\frac{n}{k})$ plus some "overhead".

Overhead related to the modular arithmetic involves overhead for conversion to the modular representation (division with remainders) and overhead for reconstruction.

Approaches to acceleration of modular arithmetic:



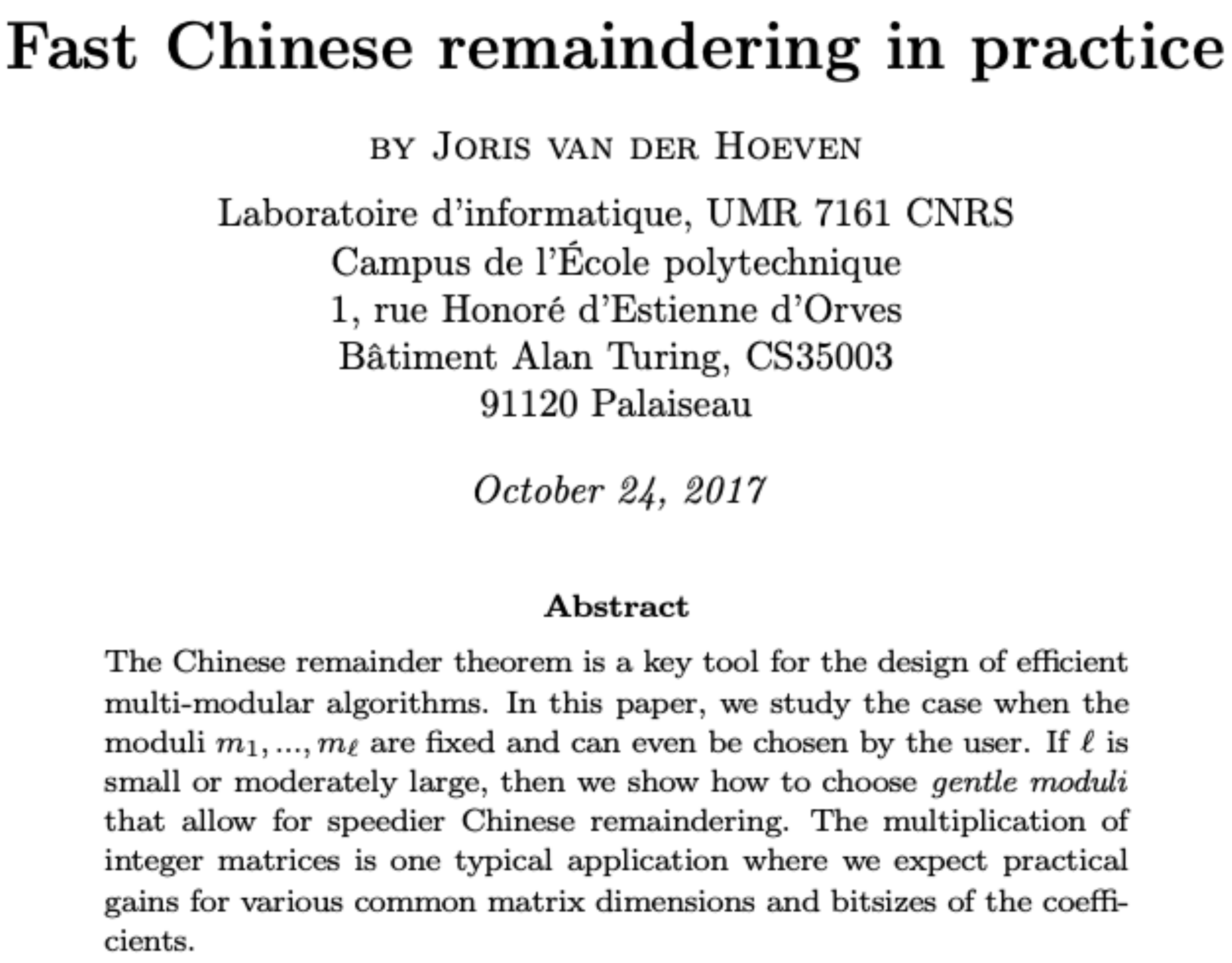
Simultaneous Conversions with the Residue Number System Using Linear Algebra

JAVAD DOLISKANI, Institute for Quantum Computing, University of Waterloo
PASCAL GIORGI and ROMAIN LEBRETON, LIRMM CNRS - University of Montpellier
ERIC SCHOST, University of Waterloo

We present an algorithm for simultaneous conversions between a given set of integers and their Residue Number System representations based on linear algebra. We provide a highly optimized implementation of the algorithm that exploits the computational features of modern processors. The main application of our algorithm is matrix multiplication over integers. Our speed-up of the conversions to and from the Residue Number System significantly improves the overall running time of matrix multiplication.

J. Doliskani, P. Giorgi, R. Lebreton, E. Schost Simultaneous Conversions with the Residue Number System Using Linear Algebra. *ACM Transactions on Mathematical Software*, Volume 44, Issue 3, Article No.: 27, pp. 1–21, 2018.

Approaches to acceleration of modular arithmetic:



Fast Chinese remaindering in practice

BY JORIS VAN DER HOEVEN

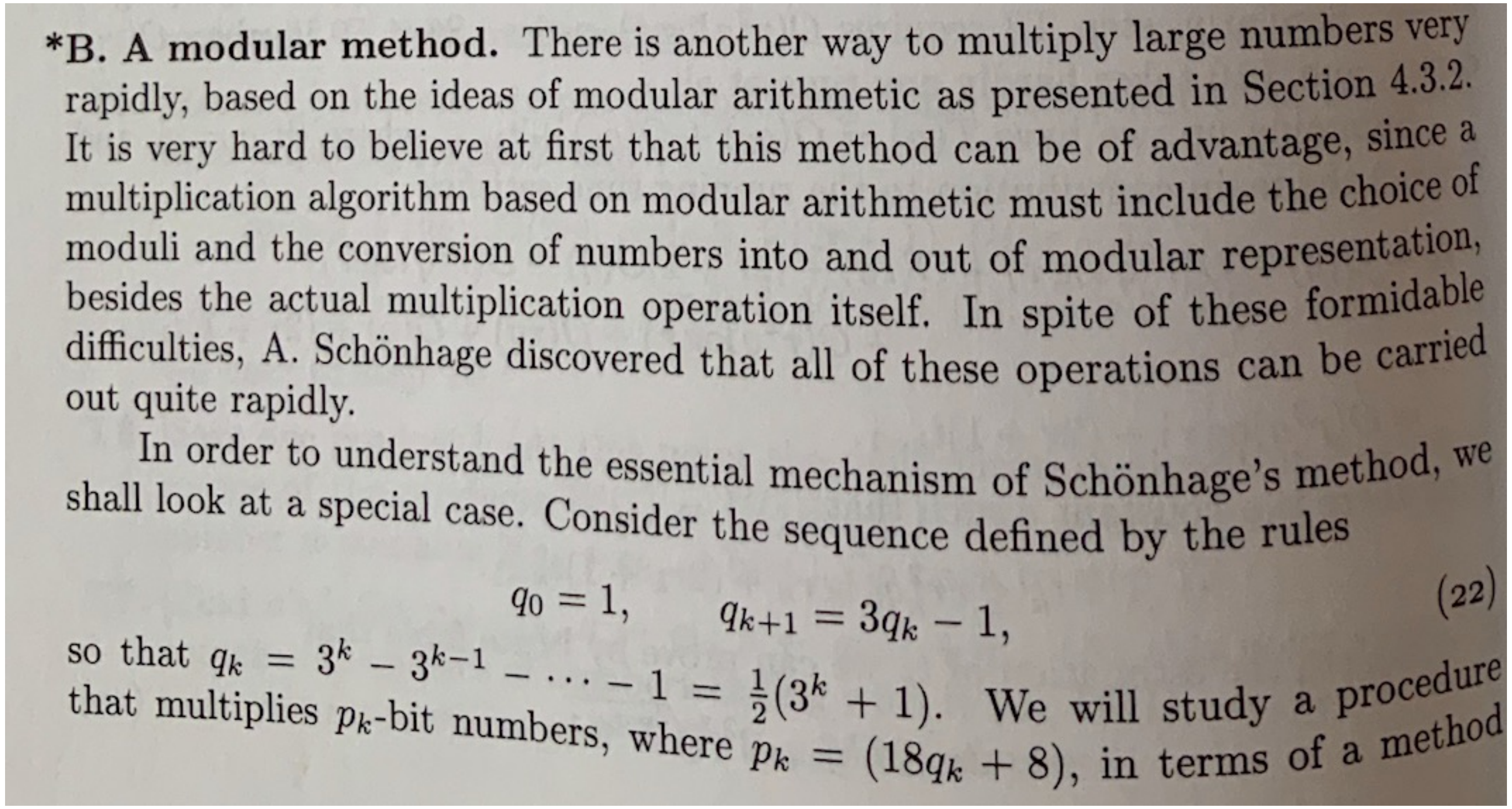
Laboratoire d'informatique, UMR 7161 CNRS
Campus de l'École polytechnique
1, rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing, CS35003
91120 Palaiseau

October 24, 2017

Abstract

The Chinese remainder theorem is a key tool for the design of efficient multi-modular algorithms. In this paper, we study the case when the moduli $m_1, ..., m_\ell$ are fixed and can even be chosen by the user. If ℓ is small or moderately large, then we show how to choose *gentle moduli* that allow for speedier Chinese remaindering. The multiplication of integer matrices is one typical application where we expect practical gains for various common matrix dimensions and bitsizes of the coefficients.

Another approach:



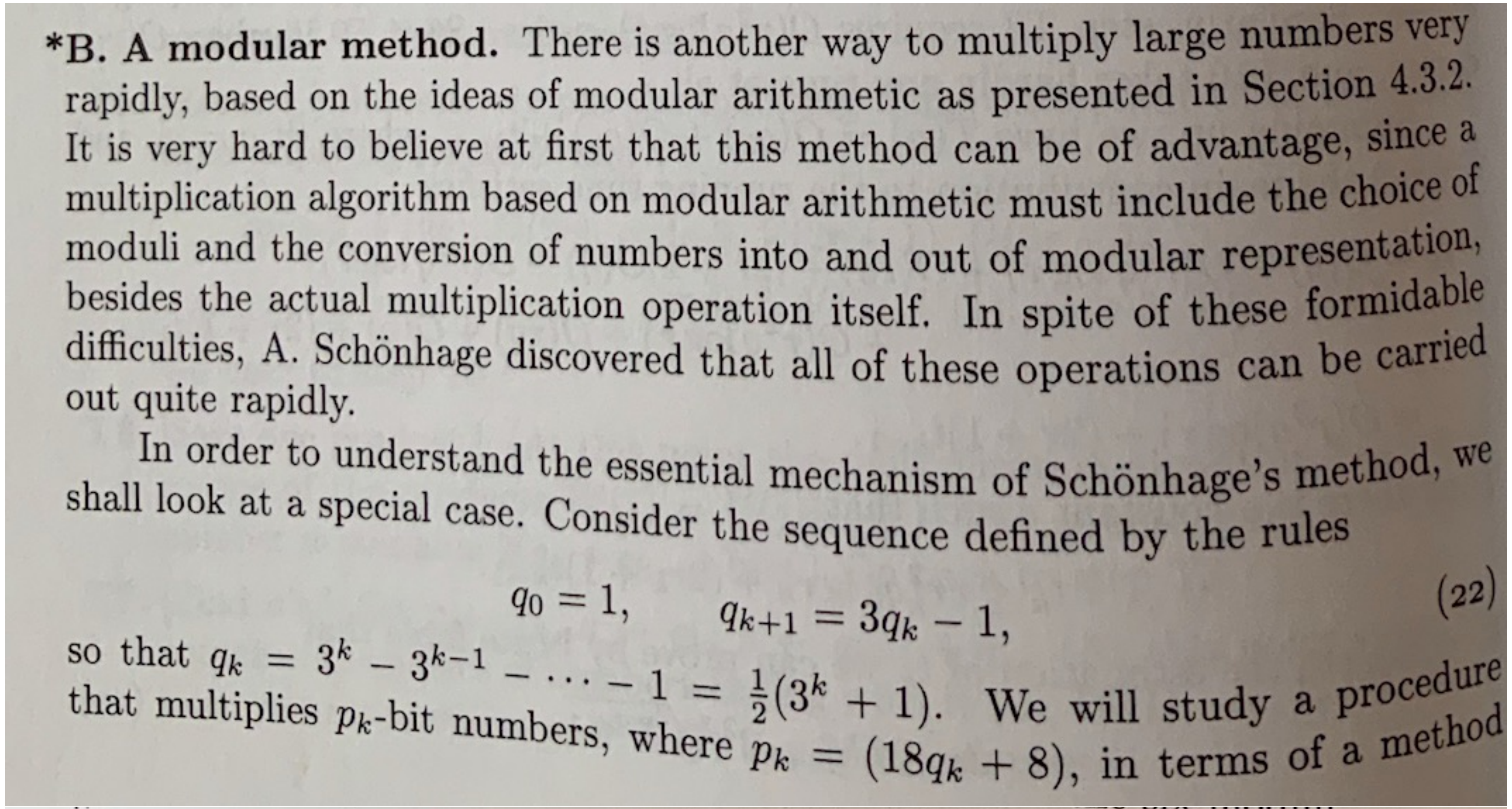
***B. A modular method.** There is another way to multiply large numbers very rapidly, based on the ideas of modular arithmetic as presented in Section 4.3.2. It is very hard to believe at first that this method can be of advantage, since a multiplication algorithm based on modular arithmetic must include the choice of moduli and the conversion of numbers into and out of modular representation, besides the actual multiplication operation itself. In spite of these formidable difficulties, A. Schönhage discovered that all of these operations can be carried out quite rapidly.

In order to understand the essential mechanism of Schönhage's method, we shall look at a special case. Consider the sequence defined by the rules

$$q_0 = 1, \qquad q_{k+1} = 3q_k - 1, \tag{22}$$

so that $q_k = 3^k - 3^{k-1} - \cdots - 1 = \frac{1}{2}(3^k + 1)$. We will study a procedure that multiplies p_k-bit numbers, where $p_k = (18q_k + 8)$, in terms of a method

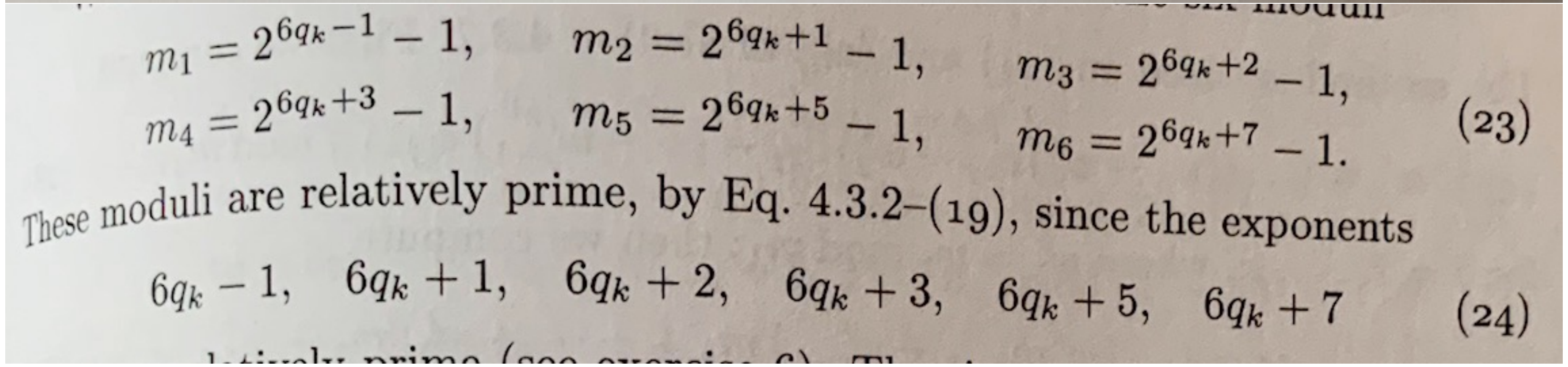
Another approach:


***B. A modular method.** There is another way to multiply large numbers very rapidly, based on the ideas of modular arithmetic as presented in Section 4.3.2. It is very hard to believe at first that this method can be of advantage, since a multiplication algorithm based on modular arithmetic must include the choice of moduli and the conversion of numbers into and out of modular representation, besides the actual multiplication operation itself. In spite of these formidable difficulties, A. Schönhage discovered that all of these operations can be carried out quite rapidly.

In order to understand the essential mechanism of Schönhage's method, we shall look at a special case. Consider the sequence defined by the rules

$$q_0 = 1, \qquad q_{k+1} = 3q_k - 1, \tag{22}$$

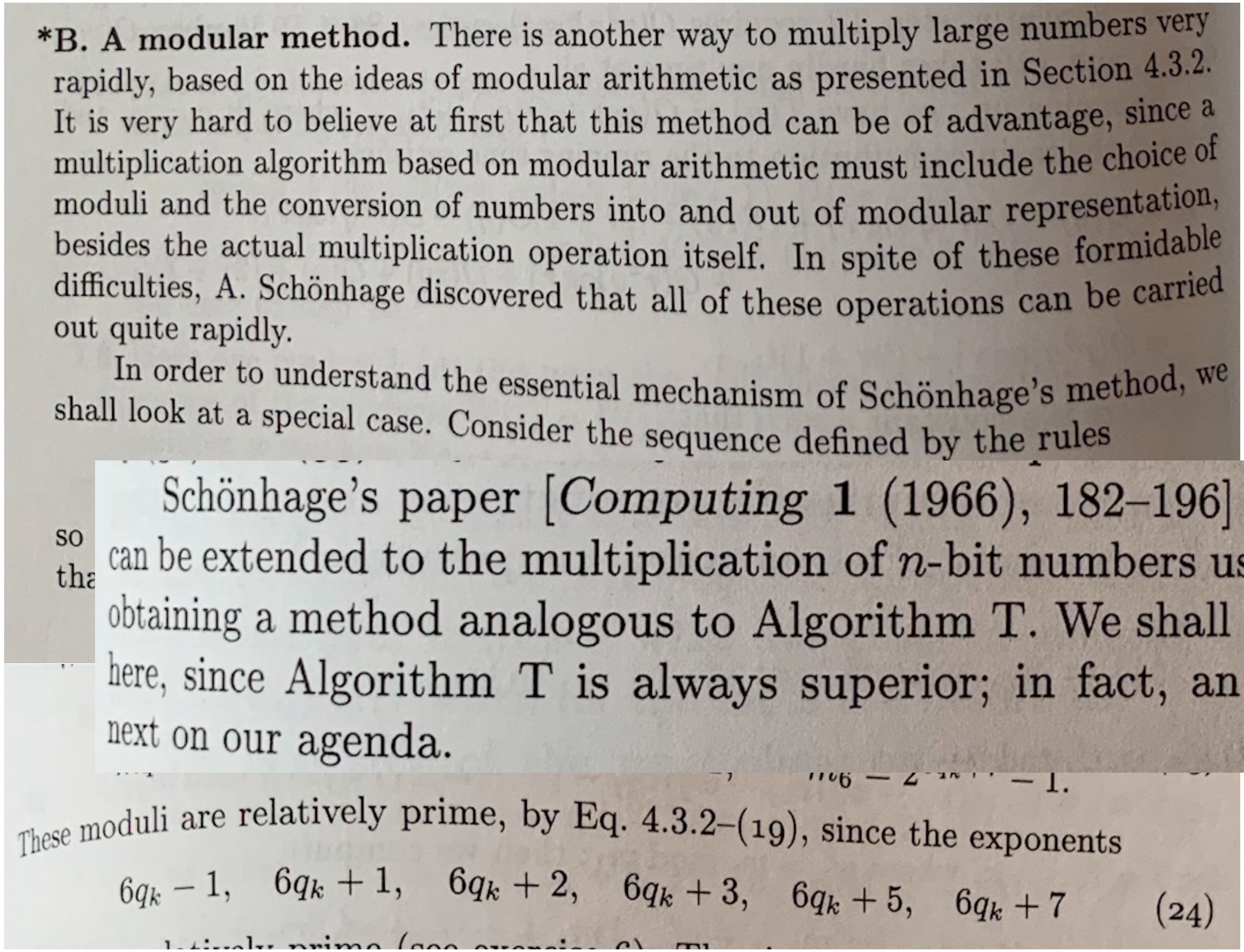
so that $q_k = 3^k - 3^{k-1} - \cdots - 1 = \frac{1}{2}(3^k + 1)$. We will study a procedure that multiplies p_k-bit numbers, where $p_k = (18q_k + 8)$, in terms of a method


$$\begin{aligned} m_1 &= 2^{6q_k-1} - 1, & m_2 &= 2^{6q_k+1} - 1, & m_3 &= 2^{6q_k+2} - 1,\\ m_4 &= 2^{6q_k+3} - 1, & m_5 &= 2^{6q_k+5} - 1, & m_6 &= 2^{6q_k+7} - 1. \end{aligned} \tag{23}$$

These moduli are relatively prime, by Eq. 4.3.2–(19), since the exponents

$$6q_k - 1, \quad 6q_k + 1, \quad 6q_k + 2, \quad 6q_k + 3, \quad 6q_k + 5, \quad 6q_k + 7 \tag{24}$$

Another approach:



***B. A modular method.** There is another way to multiply large numbers very rapidly, based on the ideas of modular arithmetic as presented in Section 4.3.2. It is very hard to believe at first that this method can be of advantage, since a multiplication algorithm based on modular arithmetic must include the choice of moduli and the conversion of numbers into and out of modular representation, besides the actual multiplication operation itself. In spite of these formidable difficulties, A. Schönhage discovered that all of these operations can be carried out quite rapidly.

In order to understand the essential mechanism of Schönhage's method, we shall look at a special case. Consider the sequence defined by the rules

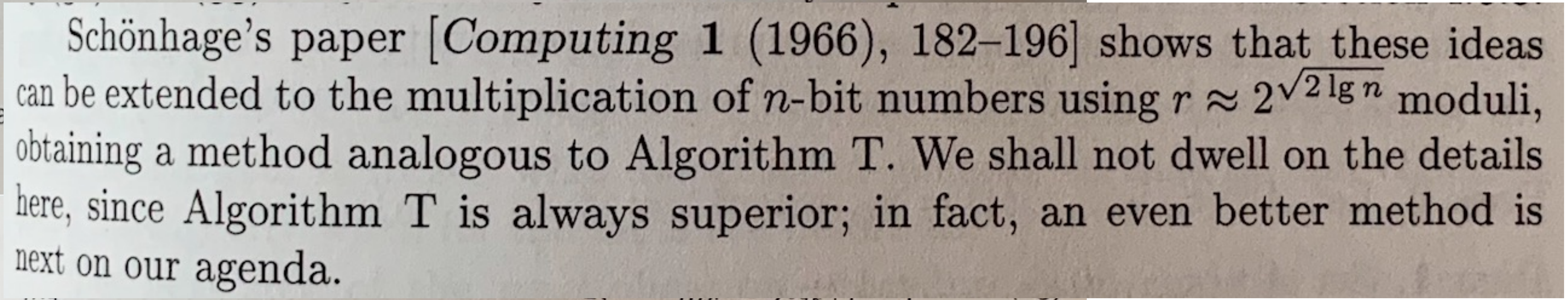
so

tha

$m_6 = 2^{\dots} - 1.$

These moduli are relatively prime, by Eq. 4.3.2–(19), since the exponents

$$6q_k - 1, \quad 6q_k + 1, \quad 6q_k + 2, \quad 6q_k + 3, \quad 6q_k + 5, \quad 6q_k + 7 \qquad (24)$$



Schönhage's paper [*Computing* **1** (1966), 182–196] shows that these ideas can be extended to the multiplication of n-bit numbers using $r \approx 2^{\sqrt{2 \lg n}}$ moduli, obtaining a method analogous to Algorithm T. We shall not dwell on the details here, since Algorithm T is always superior; in fact, an even better method is next on our agenda.

Overhead related to the modular arithmetic involves overhead for conversion to the modular representation (division with remainders) and overhead for reconstruction.

Algorithm 5 Garner(r, m)

```
Require: $\forall i \in [0, N - 1]: 0 \leq r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 1: $M \longleftarrow 1$
 2: for $i = 1$ to $N - 1$ do
 3:     $M \longleftarrow Mm_{i-1}$
 4:     $M_i \longleftarrow M^{-1} \bmod m_i$
 5: end for
 6: $a_0 \longleftarrow r_0$
 7: for $i = 1$ to $N - 1$ do
 8:     $t \longleftarrow a_{i-1}$
 9:     for $j = i - 2$ downto $0$ do
10:         $t \longleftarrow tm_j$
11:         $t \longleftarrow t + a_j$
12:     end for
13:     $t \longleftarrow r_i - t$
14:     $a_i \longleftarrow tM_i \bmod m_i$
15: end for
16: $a \longleftarrow a_{N-1}$
17: for $i = N - 1$ downto $0$
18:     $a \longleftarrow am_i$
19:     $a \longleftarrow a + a_i$
20: end for
21: return $a$
```

Algorithm 5 Garner(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 1: $M \longleftarrow 1$
 2: for $i = 1$ to $N - 1$ do
 3:     $M \longleftarrow M m_{i-1}$
 4:     $M_i \longleftarrow M^{-1} \bmod m_i$
 5: end for
 6: $a_0 \longleftarrow r_0$
 7: for $i = 1$ to $N - 1$ do
 8:     $t \longleftarrow a_{i-1}$
 9:     for $j = i - 2$ downto 0 do
10:         $t \longleftarrow t m_j$
11:         $t \longleftarrow t + a_j$
12:     end for
13:     $t \longleftarrow r_i - t$
14:     $a_i \longleftarrow t M_i \bmod m_i$
15: end for
16: $a \longleftarrow a_{N-1}$
17: for $i = N - 1$ downto 0
18:     $a \longleftarrow a m_i$
19:     $a \longleftarrow a + a_i$
20: end for
21: return $a$
```

In lines 1–5 of the algorithm, intermediate values $\left(\prod_{j=0}^{i-1} m_j\right)^{-1} \bmod m_i$ are calculated. Note that these values can be calculated once and then be used every time we need to reconstruct the result obtained in the same system of moduli. In lines 6–15, coefficients of the mixed radix representation are calculated, i.e., numbers $a_0, a_1, \ldots, a_{N-1}$ such that

$$X = a_0 + a_1 m_0 + a_2 m_0 m_1 + \ldots$$

$$+ a_{N-1} m_0 m_1 \ldots m_{N-2},$$

$$0 \le a_i < m_i, \quad 0 \le X < m_0 m_1 \ldots m_{N-1},$$

and $X \equiv r_i (\bmod m_i)$, $i = 0, 1, \ldots, N-1$. In lines 16–20, the value of X is calculated using Horner's method.

Algorithm 6 Simple–Reconstruction(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 1: $M \longleftarrow 1$
 2: for $i = 1$ to $N - 1$ do
 3:     $M \longleftarrow Mm_{i-1}$
 4:     $M_i \longleftarrow M^{-1} \bmod m_i$
 5: end for
 6: $a \longleftarrow r_0$
 7: $M \longleftarrow m_0$
 8: for $i = 1$ to $N - 1$ do
 9:     $t \longleftarrow r_i - a$
10:     $t \longleftarrow tM_i \bmod m_i$
11:     $a \longleftarrow a + tM$
12:     $M \longleftarrow Mm_i$
13: end for
14: return $a$
```

Eliminating Divisions

Eliminating Divisions

[8] D. E. Knuth. *The Art of Computer Programming*, Vol 2.

Eliminating Divisions

[8] D. E. Knuth. *The Art of Computer Programming*, Vol 2.

Consider the modulus 2^n-1 for natural $n > 1$. Than addition, subtraction and multiplication mod $2^n - 1$ (denoted respectively by $\oplus$, $\ominus$ and $\otimes$) can be defined as follows [8]. For $0 \le u < 2^n$ and $0 \le v < 2^n$ define

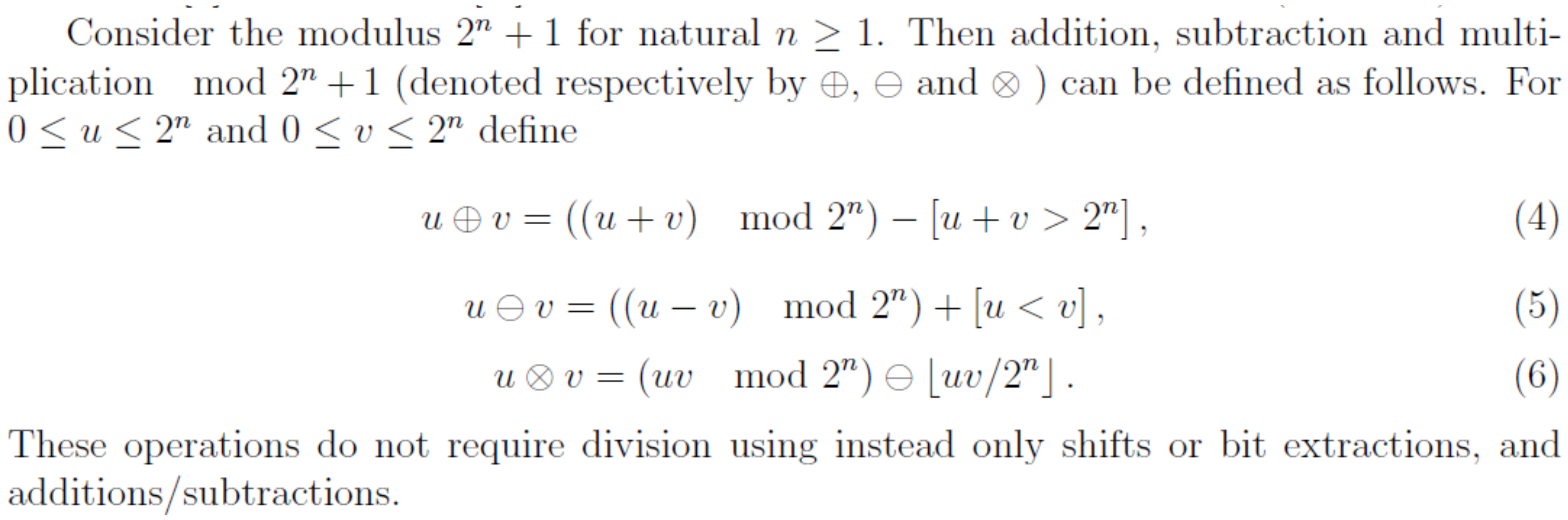
$$u \oplus v = ((u+v) \mod 2^n) + [u+v \ge 2^n]\,, \tag{1}$$

$$u \ominus v = ((u-v) \mod 2^n) - [u < v]\,, \tag{2}$$

$$u \otimes v = (uv \mod 2^n) \oplus \lfloor uv/2^n \rfloor\,. \tag{3}$$

Following [8] we denote by $[B]$ the characteristic function of condition B: $(B \Rightarrow 1; 0)$.

Why do not we consider + instead of -?



Consider the modulus 2^n+1 for natural $n \geq 1$. Then addition, subtraction and multiplication $\mod 2^n+1$ (denoted respectively by $\oplus$, $\ominus$ and $\otimes$) can be defined as follows. For $0 \leq u \leq 2^n$ and $0 \leq v \leq 2^n$ define

$$u \oplus v = ((u+v) \mod 2^n) - [u+v > 2^n] , \tag{4}$$

$$u \ominus v = ((u-v) \mod 2^n) + [u < v] , \tag{5}$$

$$u \otimes v = (uv \mod 2^n) \ominus \lfloor uv/2^n \rfloor . \tag{6}$$

These operations do not require division using instead only shifts or bit extractions, and additions/subtractions.

Choosing moduli:

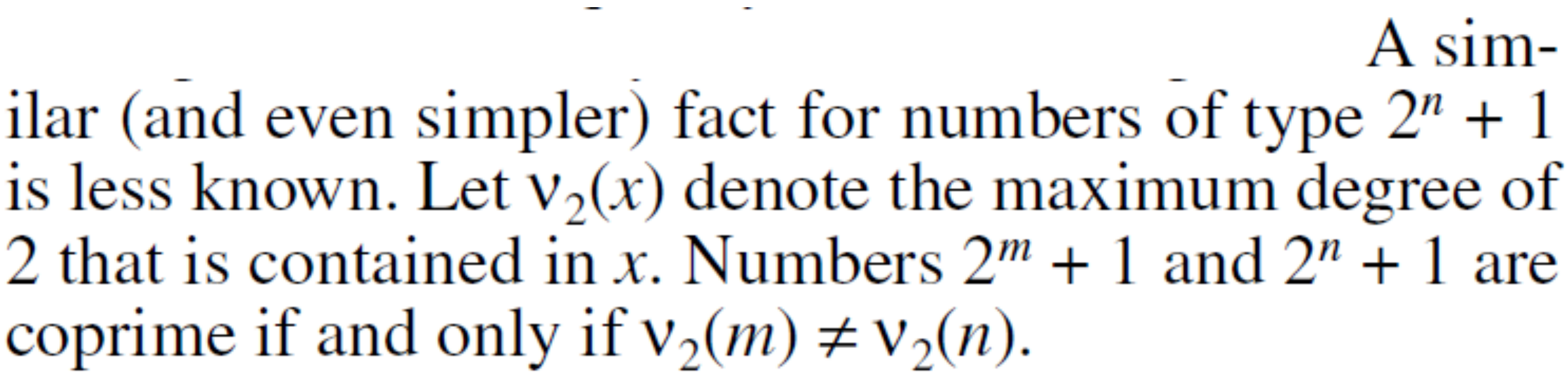
What is required?

1) **Relative primality**
2) **Ease of pairwise inversion**

$$\gcd(2^n - 1, 2^m - 1) = 2^{\gcd(n,m)} - 1$$

$$2^n - 1 \perp 2^m - 1 \iff n \perp m$$

1st surprise …



A similar (and even simpler) fact for numbers of type $2^n + 1$ is less known. Let $\nu_2(x)$ denote the maximum degree of 2 that is contained in x. Numbers $2^m + 1$ and $2^n + 1$ are coprime if and only if $\nu_2(m) \neq \nu_2(n)$.

This is a corollary of a more general result: for any positive integers a, n, m, $a > 1$ [4],

$$\mathrm{GCD}(a^m + 1, a^n + 1) = \begin{cases} a^{\mathrm{GCD}(m,n)} + 1, & \text{if } \nu_2(m) = \nu_2(n) \\ 1, & \text{if } \nu_2(m) \neq \nu_2(n) \\ & \text{and } a \text{ is even} \\ 2, & \text{if } \nu_2(m) \neq \nu_2(n) \\ & \text{and } a \text{ is odd.} \end{cases}$$

It is interesting that the latter result is not usually mentioned in monographs, such as [1], and textbooks on number theory.

4. Cade, J.J., Kee-Wai, Lau, Pedersen, A., and Lossers, O.P., Problem E3288. Problems and Solutions, *The Am. Math. Monthly*, 1990, vol. 97, no. 4, pp. 344–345.

In the book [5] devoted to Fermat numbers [5] (which appeared 11 years later after publication [4]), only a particular case of this result is proven:

$$\mathrm{GCD}(2^m + 1, 2^{mn} + 1) = \begin{cases} 1, & \text{if } n \text{ is even} \\ 2^m + 1, & \text{if } n \text{ is odd.} \end{cases}$$

5. Křížek, M., Luca, F., and Somer, L., *17 Lectures on Fermat Numbers: From Number Theory to Geometry.* New York: Springer, 2001.

2nd surprise …

Proposition 1. *Consider moduli of the form*

$$m_i = 2^{a2^i} + 1, \quad i = 0, 2, \ldots, n; \tag{7}$$

where a is an arbitrary positive integer, and products

$$M_i = \prod_{j=0}^{i-1} m_j = \prod_{j=0}^{i-1} \left(2^{a2^j} + 1\right), \quad i = 0, 1, \ldots, n-1. \tag{8}$$

Then

$$M_i^{-1} \quad \text{mod } m_i = 2^{a2^i - 1} - 2^{a-1} + 1, \quad i = 1, 2, ..., n. \tag{9}$$

Proposition 1. *Consider moduli of the form*

$$m_i = 2^{a2^i} + 1, \quad i = 0, 2, \ldots, n; \tag{7}$$

where a is an arbitrary positive integer, and products

$$M_i = \prod_{j=0}^{i-1} m_j = \prod_{j=0}^{i-1} \left(2^{a2^j} + 1\right), \quad i = 0, 1, \ldots, n-1. \tag{8}$$

Then

$$M_i^{-1} \mod m_i = 2^{a2^i - 1} - 2^{a-1} + 1, \quad i = 1, 2, ..., n. \tag{9}$$

multiplication by the sparse inverse is just 2 shifts, 1 addition, and 1 subtraction. If $a = 1$ (i.e., the moduli are Fermat numbers),

$$M_i^{-1} \mod m_i = 2^{2^i - 1}, \quad i = 1, 2, ...$$

and multiplication by the inverse requires shifts only.

A more general result is valid for an arbitrary numerical system with an even base B. Consider moduli of type $m_i = B^{2^i} + 1$ $(i = 0, 1, \ldots, k)$ and products $M_i = \prod_{j=0}^{i-1} m_j = \prod_{j=0}^{i-1} (B^{2^j} + 1)$ $(i = 1, 2, \ldots, k)$.

Proposition 2.

$$M_i^{-1} \bmod m_i = \frac{B^{2^i} - B + 2}{2}, \quad i = 1, 2, \ldots, k.$$

For numbers of type 2+, the simplest scheme of modulus choice is based on “shift”: the first exponent a is chosen in an arbitrary way; every consecutive exponent is obtained from the previous one by multiplying by 2 (which corresponds to the one bit left shift in binary notation). This scheme generates moduli $2^a + 1$, $2^{2a} + 1$, $2^{4a} + 1$, …, $2^{2^i a} + 1$. If a is chosen to be equal to 1, then the moduli are consecutive Fermat numbers $2^1 + 1$, $2^2 + 1$, $2^4 + 1$, …, $2^{2^i} + 1$, ….

Another ("block") strategy consists in generating exponents of the same bit length, which gives a series of moduli that is better balanced in lengths. First, the number of moduli b is chosen, and the exponents e_1, e_2, …, e_b are generated using recurrence $e_b = 2^{b+1} - 1$, $e_k = e_{k+1} - 2^{b-k-1}$, $k = b - 1, b - 2, \ldots, 1$. Thus, binary notation of the exponent e_i ($i = 1, \ldots, b$) ends with $(b - i)$ zeros following 1, which ensures coprimality of the corresponding moduli. For example, for $b = 4$, the following four moduli with 4-bit exponents will be generated: 8, 12, 14, and 15.

Initial reduction and tools

Algorithm 2 $\textsc{Bits}(a, m, n)$ — Bit-range extraction

```
Require: $0 \le a,\ m \le n$
Ensure: The number returned $r$ satisfies $0 \le r < 2^{n-m}$
1: mpz_init($r$) {If not already done}
2: mpz_tdiv_r_2exp($r, a, n$)
3: mpz_tdiv_q_2exp($r, r, m$)
4: return $r$
```

Algorithm 3 $\textsc{Simple-Reduce-Minus}(a, n)$ — Reduce a by $2^n - 1$

```
Require: $0 \le a < 2^{2n}$
Ensure: $0 \le a \le 2^n - 1$
1: $t \leftarrow \textsc{Bits}(a, n, 2n - 1)$
2: $a \leftarrow \textsc{Bits}(a, 0, n - 1)$
3: $a \leftarrow a + t$
4: $t \leftarrow \textsc{Bits}(a, n, n)$
5: $a \leftarrow \textsc{Bits}(a, 0, n - 1) + t$
```

Algorithm 4 THEORETIC-REDUCE-MINUS(a, n) — Reduce a by $2^n - 1$

```
Require: $0 \le a$
Ensure: The number returned $r$ satisfies $0 \le r \le 2^n - 1$
 1: $r \leftarrow 0$
 2: $b \leftarrow (|a| - 1)/n + 1$ {$|a|$ measures the bit length of $a$}
 3: for $i = 0$ to $b - 1$ do
 4:     $r \leftarrow r + \textsc{Bits}(a, i \cdot n, (i + 1) \cdot n - 1)$
 5: end for
 6: if $r \ge 2^n$ then
 7:     return THEORETIC-REDUCE-MINUS$(r, n)$
 8: else
 9:     return $r$
10: end if
```

Algorithm 5 DC-Reduce-Minus(a, n) — Reduce a by $2^n - 1$

```
Require: $0 \le a$
Ensure: $0 \le a \le 2^n - 1$
1: mpz_init($t$)
2: while $a \ge 2^n$ do
3:     $b \leftarrow (|a| - 1)/n + 1$ {$|a|$ measures the bit length of $a$}
4:     $b \leftarrow b/2$
5:     mpz_tdiv_q_2exp($t, a, b \cdot n$)
6:     mpz_tdiv_r_2exp($a, a, b \cdot n$)
7:     $a \leftarrow a + t$
8: end while
9: mpz_clear($t$)
```

Algorithms 3 – 4 can be modified providing implementation of similar procedures for reductions by $2^n + 1$ in the same way as (4)–(6) are modifications of (1)–(3) that use the congruence $2^n \equiv -1 (\mod 2^n + 1)$ instead of $2^n \equiv 1 (\mod 2^n - 1)$. For reductions of large input we can use the fact that $2^n + 1 \mid 2^{2n} - 1$. Thus, to reduce by $2^n + 1$, first reduce by $2^{2n} - 1$ and than reduce the result by $2^n + 1$ using simple reduction. Together, these steps form DC-REDUCE-PLUS.

Algorithm 6 DC-REDUCE-PLUS(a, n) — Reduce a by $2^n + 1$

```
Require: $0 \leq a$
Ensure: $0 \leq a \leq 2^n + 1$
1: mpz_init($t$)
2: DC-REDUCE-MINUS($a, 2n$)
3: mpz_tdiv_q_2exp($t, a, n$)
4: mpz_tdiv_r_2exp($a, a, n$)
5: $a \leftarrow a - t$
6: if $a < 0$ then
7:     $a \leftarrow a + 2^n + 1$
8: end if
9: mpz_clear($t$)
```

Reconstruction

Algorithm 5 Garner(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 1: $M \longleftarrow 1$
 2: for $i = 1$ to $N-1$ do
 3:     $M \longleftarrow Mm_{i-1}$
 4:     $M_i \longleftarrow M^{-1} \bmod m_i$
 5: end for
 6: $a_0 \longleftarrow r_0$
 7: for $i = 1$ to $N-1$ do
 8:     $t \longleftarrow a_{i-1}$
 9:     for $j = i-2$ downto $0$ do
10:         $t \longleftarrow tm_j$
11:         $t \longleftarrow t + a_j$
12:     end for
13:     $t \longleftarrow r_i - t$
14:     $a_i \longleftarrow tM_i \bmod m_i$
15: end for
16: $a \longleftarrow a_{N-1}$
17: for $i = N-1$ downto $0$
18:     $a \longleftarrow am_i$
19:     $a \longleftarrow a + a_i$
20: end for
21: return $a$
```

Algorithm 5 Garner(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 1: $M \longleftarrow 1$
 2: for $i = 1$ to $N-1$ do
 3:     $M \longleftarrow Mm_{i-1}$
 4:     $M_i \longleftarrow M^{-1} \bmod m_i$
 5: end for
 6: $a_0 \longleftarrow r_0$
 7: for $i = 1$ to $N-1$ do
 8:     $t \longleftarrow a_{i-1}$
 9:     for $j = i-2$ downto $0$ do
10:         $t \longleftarrow tm_j$
11:         $t \longleftarrow t + a_j$
12:     end for
13:     $t \longleftarrow r_i - t$
14:     $a_i \longleftarrow tM_i \bmod m_i$
15: end for
16: $a \longleftarrow a_{N-1}$
17: for $i = N-1$ downto $0$
18:     $a \longleftarrow am_i$
19:     $a \longleftarrow a + a_i$
20: end for
21: return $a$
```

In lines 1–5 of the algorithm, intermediate values $\left(\prod_{j=0}^{i-1} m_j\right)^{-1} \bmod m_i$ are calculated. Note that these values can be calculated once and then be used every time we need to reconstruct the result obtained in the same system of moduli. In lines 6–15, coefficients of the mixed radix representation are calculated, i.e., numbers $a_0, a_1, \ldots, a_{N-1}$ such that

$$X = a_0 + a_1 m_0 + a_2 m_0 m_1 + \ldots$$
$$+ a_{N-1} m_0 m_1 \ldots m_{N-2},$$
$$0 \le a_i < m_i, \quad 0 \le X < m_0 m_1 \ldots m_{N-1},$$

and $X \equiv r_i (\bmod m_i)$, $i = 0, 1, \ldots, N-1$. In lines 16–20, the value of X is calculated using Horner's method.

Algorithm 5 Garner(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 1: $M \longleftarrow 1$
 2: for $i = 1$ to $N-1$ do
 3:     $M \longleftarrow Mm_{i-1}$
 4:     $M_i \longleftarrow M^{-1} \bmod m_i$
 5: end for
 6: $a_0 \longleftarrow r_0$
 7: for $i = 1$ to $N-1$ do
 8:     $t \longleftarrow a_{i-1}$
 9:     for $j = i-2$ downto $0$ do
10:         $t \longleftarrow tm_j$
11:         $t \longleftarrow t + a_j$
12:     end for
13:     $t \longleftarrow r_i - t$
14:     $a_i \longleftarrow tM_i \bmod m_i$
15: end for
16: $a \longleftarrow a_{N-1}$
17: for $i = N-1$ downto $0$
18:     $a \longleftarrow am_i$
19:     $a \longleftarrow a + a_i$
20: end for
21: return $a$
```

Algorithm 5 Garner(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 6: $a_0 \leftarrow r_0$
 7: for $i = 1$ to $N-1$ do
 8:     $t \leftarrow a_{i-1}$
 9:     for $j = i-2$ downto $0$ do
10:         $t \leftarrow tm_j$
11:         $t \leftarrow t + a_j$
12:     end for
13:     $t \leftarrow r_i - t$
14:     $a_i \leftarrow tM_i \bmod m_i$
15: end for
16: $a \leftarrow a_{N-1}$
17: for $i = N-1$ downto $0$
18:     $a \leftarrow am_i$
19:     $a \leftarrow a + a_i$
20: end for
21: return $a$
```

Algorithm 5 Garner(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
6: $a_0 \longleftarrow r_0$
7: for $i = 1$ to $N-1$ do
8:     $t \longleftarrow a_{i-1}$
9:     for $j = i-2$ downto $0$ do
10:        $t \longleftarrow tm_j$
11:        $t \longleftarrow t + a_j$
12:    end for
13:    $t \longleftarrow r_i - t$
14:    $a_i \longleftarrow tM_i \bmod m_i$
15: end for
16: $a \longleftarrow a_{N-1}$
17: for $i = N-1$ downto $0$
18:    $a \longleftarrow am_i$
19:    $a \longleftarrow a + a_i$
20: end for
21: return $a$
```

Algorithm 5 Garner(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
6: $a_0 \longleftarrow r_0$
7: for $i = 1$ to $N-1$ do
8:     $t \longleftarrow a_{i-1}$
9:     for $j = i-2$ downto $0$ do
10:        $t \longleftarrow tm_j$
11:        $t \longleftarrow t + a_j$
12:    end for
13:    $t \longleftarrow r_i - t$
14:    $a_i \longleftarrow tM_i \bmod m_i$
15: end for
16: $a \longleftarrow a_{N-1}$
17: for $i = N-1$ downto $0$
18:    $a \longleftarrow am_i$
19:    $a \longleftarrow a + a_i$
20: end for
21: return $a$
```

Algorithm 6 Simple–Reconstruction(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \leq r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 1: $M \longleftarrow 1$
 2: for $i = 1$ to $N - 1$ do
 3:     $M \longleftarrow Mm_{i-1}$
 4:     $M_i \longleftarrow M^{-1} \bmod m_i$
 5: end for
 6: $a \longleftarrow r_0$
 7: $M \longleftarrow m_0$
 8: for $i = 1$ to $N - 1$ do
 9:     $t \longleftarrow r_i - a$
10:     $t \longleftarrow tM_i \bmod m_i$
11:     $a \longleftarrow a + tM$
12:     $M \longleftarrow Mm_i$
13: end for
14: return $a$
```

Algorithm 6 Simple–Reconstruction(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 1: $M \longleftarrow 1$
 2: for $i = 1$ to $N - 1$ do
 3:     $M \longleftarrow Mm_{i-1}$
 4:     $M_i \longleftarrow M^{-1} \bmod m_i$
 5: end for
 6: $a \longleftarrow r_0$
 7: $M \longleftarrow m_0$
 8: for $i = 1$ to $N - 1$ do
 9:     $t \longleftarrow r_i - a$
10:     $t \longleftarrow tM_i \bmod m_i$
11:     $a \longleftarrow a + tM$
12:     $M \longleftarrow Mm_i$
13: end for
14: return $a$
```

Algorithm 6 Simple–Reconstruction(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 1: $M \longleftarrow 1$
 2: for $i = 1$ to $N - 1$ do
 3:     $M \longleftarrow Mm_{i-1}$
 4:     $M_i \longleftarrow M^{-1} \bmod m_i$
 5: end for
 6: $a \longleftarrow r_0$
 7: $M \longleftarrow m_0$
 8: for $i = 1$ to $N - 1$ do
 9:     $t \longleftarrow r_i - a$
10:     $t \longleftarrow tM_i \bmod m_i$
11:     $a \longleftarrow a + tM$
12:     $M \longleftarrow Mm_i$
13: end for
14: return $a$
```

Algorithm 6 Simple–Reconstruction(r, m)

Require: $\forall i \in [0, N-1]: 0 \leq r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$

```
 6: $a \longleftarrow r_0$
 7: $M \longleftarrow m_0$
 8: for $i = 1$ to $N - 1$ do
 9:     $t \longleftarrow r_i - a$
10:     $t \longleftarrow tM_i \bmod m_i$
11:     $a \longleftarrow a + tM$
12:     $M \longleftarrow Mm_i$
13: end for
14: return $a$
```

Algorithm 6 Simple–Reconstruction(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 6: $a \longleftarrow r_0$
 7: $M \longleftarrow m_0$
 8: for $i = 1$ to $N - 1$ do
 9:     $t \longleftarrow r_i - a$
10:     $\boxed{t \longleftarrow tM_i \bmod m_i}$
11:     $a \longleftarrow a + tM$
12:     $M \longleftarrow Mm_i$
13: end for
14: return $a$
```

Algorithm 6 Simple–Reconstruction(r, m)

```
Require: $\forall i \in [0, N-1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
 6: $a \longleftarrow r_0$
 7: $M \longleftarrow m_0$
 8: for $i = 1$ to $N - 1$ do
 9:     $t \longleftarrow r_i - a$
10:     $\boxed{t \longleftarrow tM_i \bmod m_i}$
11:     $\boxed{a \longleftarrow a + tM}$
12:     $M \longleftarrow Mm_i$
13: end for
14: return $a$
```

Experiments

In order to compare the time costs of different implementations, we used the same problem with the initial data of different bit length, namely, multiplication of two matrices with integer elements. Matrices of sizes 64×64 and 128×128 were generated randomly using tools of the GMP package. The bit length of the matrix elements varied from 16 to $12288 = \frac{3}{2} \cdot 8192$ bits.

Test runs were held on the AMD Duron 750M processor with 512 Mb RAM in the Debian GNU/Linux operating system supplied with the GMP package version 4.1.4–6. We used the GNU g++ compiler version 1:3.3.5–13. For the sake of comparison, each matrix multiplication was also run using the GMP built-in fast integer arithmetic (`mpz_class` methods).

- For comparatively small values of bit length of the matrices elements, the GMP built-in arithmetic is the fastest. This is not surprising, since the GMP package contains implementations of the best algorithms of integer multiplication optimized for the processor architecture and chooses the fastest code for the specific size of the multipliers.

- Modular arithmetic optimized for the Cunningham numbers is always faster than the standard modular arithmetic.

- Modular arithmetic optimized for the Cunningham numbers is always faster than the standard modular arithmetic.
- The time for the result reconstruction is reduced considerably if the multiplication-free variant of the Garner algorithm is used: from 20% of total problem run time to 0.4% for the same input data.

- When the bit length of matrix elements increases, the implementation that uses the shift scheme generated moduli of type 2+ starts to prevail over the fast GMP arithmetic. For example, multiplication of two pseudorandom matrices of size 64×64 with the elements uniformly distributed between 0 and $2^{32768} - 1$ based on the use of the shift scheme with the first modulus $2^{65} + 1$ took 283 seconds. The same calculation by the `mpz_class` methods took 495 seconds.

- Efficiency of the shift scheme depends on the selection of the parameter a (exponent of the smallest modulus). The appropriate selection is possible when the initial data allow obtaining a good estimate of the result size.

Simultaneous Conversions with the Residue Number System Using Linear Algebra

JAVAD DOLISKANI, Institute for Quantum Computing, University of Waterloo
PASCAL GIORGI and ROMAIN LEBRETON, LIRMM CNRS - University of Montpellier
ERIC SCHOST, University of Waterloo

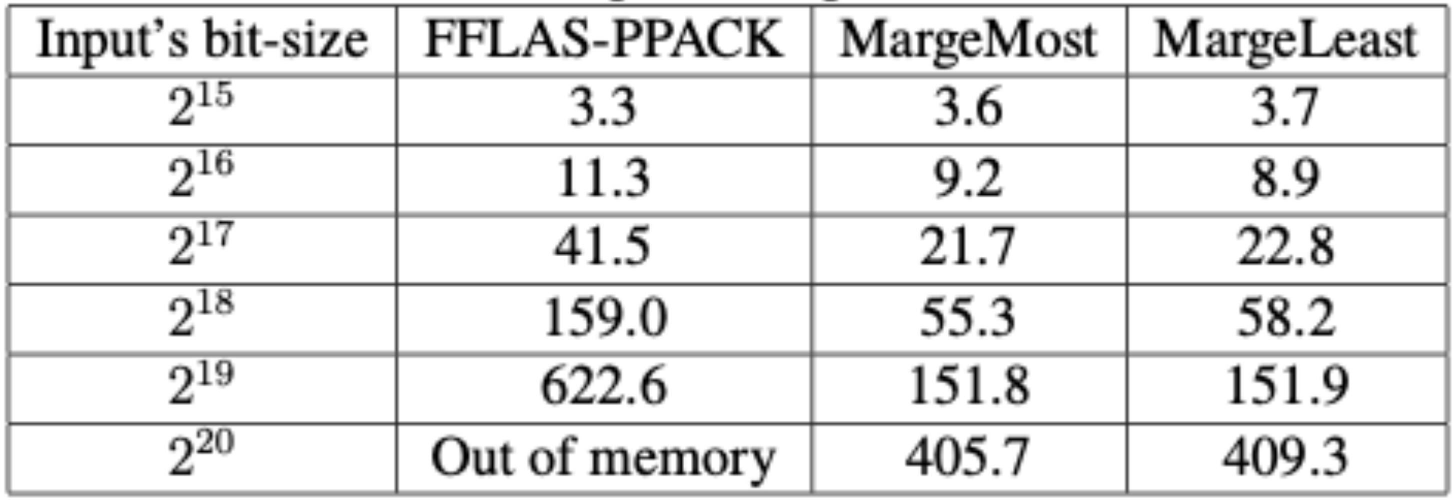
We present an algorithm for simultaneous conversions between a given set of integers and their Residue Number System representations based on linear algebra. We provide a highly optimized implementation of the algorithm that exploits the computational features of modern processors. The main application of our algorithm is matrix multiplication over integers. Our speed-up of the conversions to and from the Residue Number System significantly improves the overall running time of matrix multiplication.

Simultaneous Conversions with the Residue Number System Using Linear Algebra

JAVAD DOLISKANI, Institute for Quantum Computing, University of Waterloo
PASCAL GIORGI and ROMAIN LEBRETON, LIRMM CNRS - University of Montpellier
ERIC SCHOST, University of Waterloo

We prese heir Residue
Number nentation of
the algor ation of our
algorithn the Residue
Number

Table 2.1: 32×32 matrix-product optimal total time in seconds



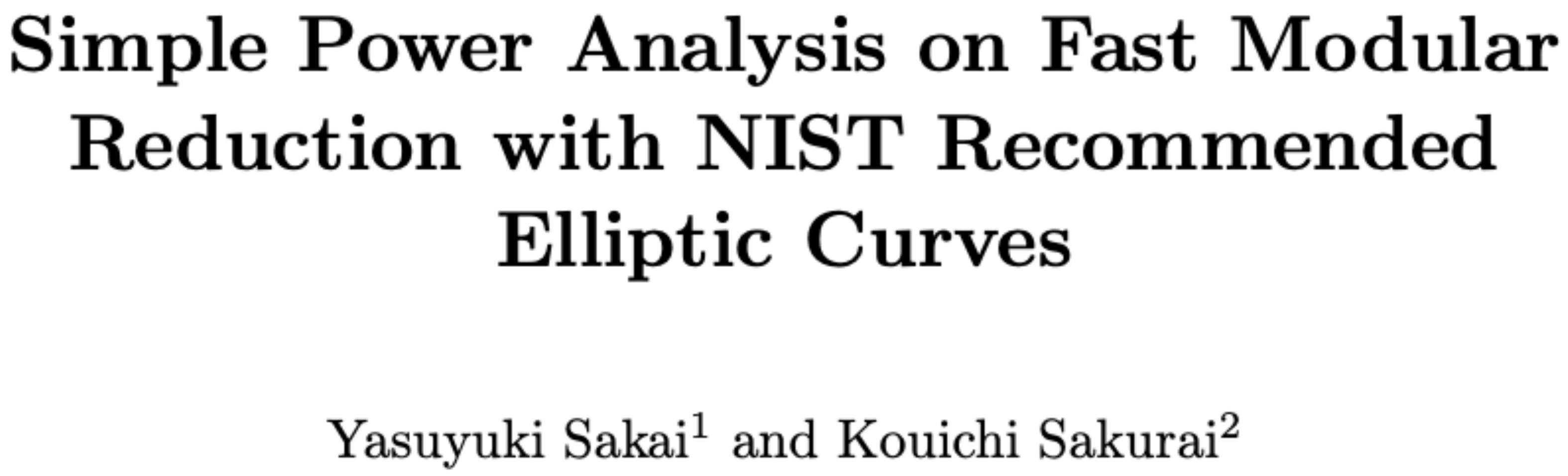
Input's bit-size	FFLAS-PPACK	MargeMost	MargeLeast
2^{15}	3.3	3.6	3.7
2^{16}	11.3	9.2	8.9
2^{17}	41.5	21.7	22.8
2^{18}	159.0	55.3	58.2
2^{19}	622.6	151.8	151.9
2^{20}	Out of memory	405.7	409.3

Three-term moduli: get balance of moduli size back

$$m_i = 2^n \pm 2^{k_i} \pm 1, \quad k_i < n$$

Three-term moduli: get balance of moduli size back

$$m_i = 2^n \pm 2^{k_i} \pm 1, \quad k_i < n$$



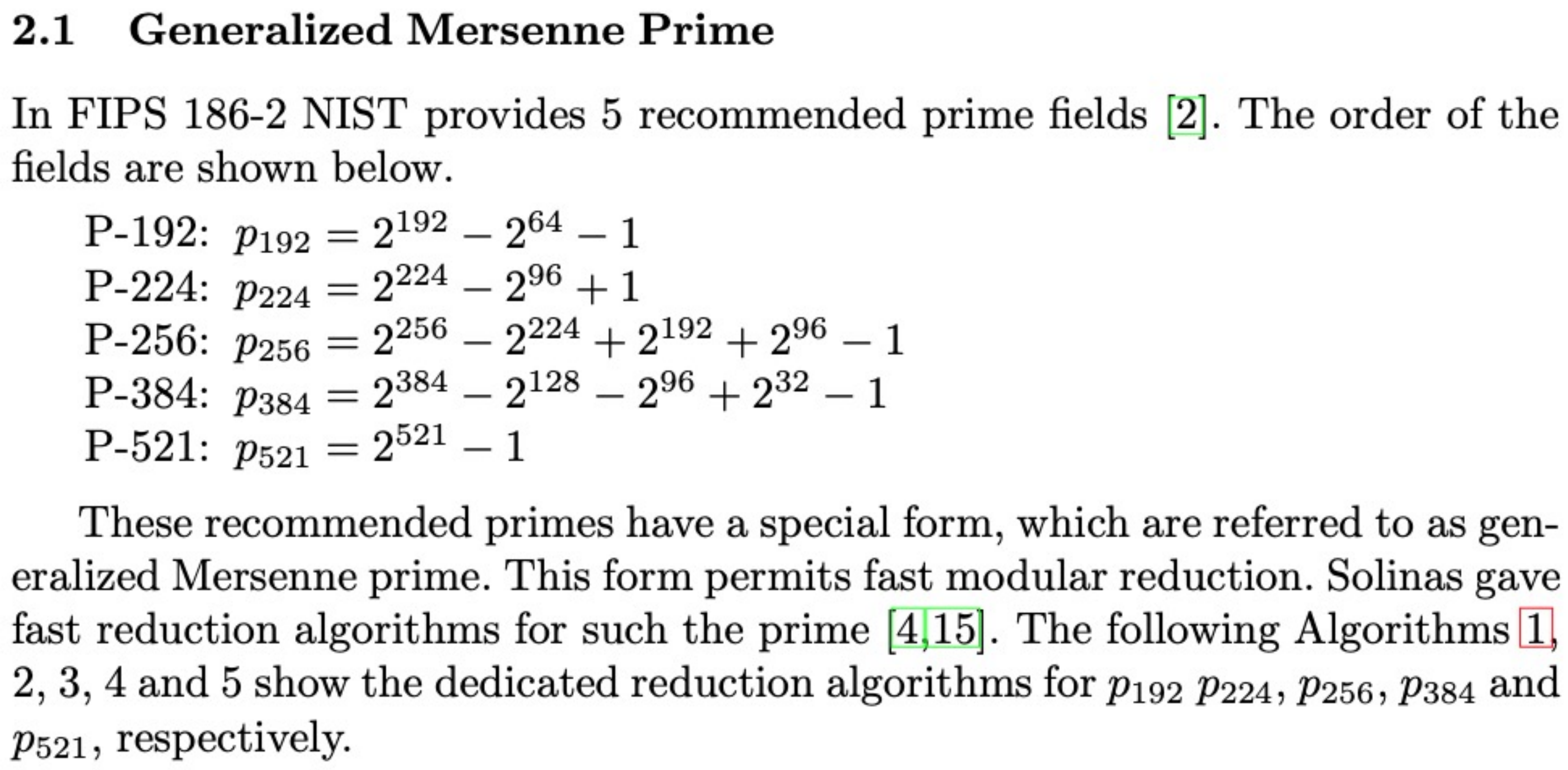
Simple Power Analysis on Fast Modular Reduction with NIST Recommended Elliptic Curves

Yasuyuki Sakai[1] and Kouichi Sakurai[2]

Three-term moduli

$$m_i = 2^n \pm 2^{k_i} \pm 1, \quad k_i < n$$

Simple I
Reduc



2.1 Generalized Mersenne Prime

In FIPS 186-2 NIST provides 5 recommended prime fields [2]. The order of the fields are shown below.

P-192: $p_{192} = 2^{192} - 2^{64} - 1$
P-224: $p_{224} = 2^{224} - 2^{96} + 1$
P-256: $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
P-384: $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
P-521: $p_{521} = 2^{521} - 1$

These recommended primes have a special form, which are referred to as generalized Mersenne prime. This form permits fast modular reduction. Solinas gave fast reduction algorithms for such the prime [4,15]. The following Algorithms 1, 2, 3, 4 and 5 show the dedicated reduction algorithms for p_{192} p_{224}, p_{256}, p_{384} and p_{521}, respectively.

GENERALISED MERSENNE NUMBERS REVISITED

ROBERT GRANGER AND ANDREW MOSS

ABSTRACT. Generalised Mersenne Numbers (GMNs) were defined by Solinas in 1999 and feature in the NIST (FIPS 186-2) and SECG standards for use in elliptic curve cryptography. Their form is such that modular reduction is extremely efficient, thus making them an attractive choice for modular mul-

Low-Weight Polynomial Form Integers for Efficient Modular Multiplication

Jaewook Chung and M. Anwar Hasan

February 9, 2006

Abstract

In 1999, Jerome Solinas introduced families of moduli called the generalized Mersenne numbers (GMNs), which are expressed in low-weight polynomial form, $p = f(t)$, where t is limited to a power of 2. GMNs are very useful in elliptic curve cryptosystems over prime fields, since only integer additions and subtractions are required in modular reductions. However, since there are not many GMNs and each GMN requires a dedicated implementation, GMNs are hardly useful for other cryptosystems. Here we modify GMN by removing restriction on the choice of t and restricting the coefficients of $f(t)$ to 0 and ± 1. We call such families of moduli low-weight polynomial form integers (LWPFIs). We show an efficient modular multiplication method using LWPFI moduli. LWPFIs allow general implementation and there exist many LWPFI moduli. One may consider LWPFIs as a trade-off between general integers and GMNs.

Three-term moduli

$$m_i = 2^n \pm 2^{k_i} \pm 1, \quad k_i < n$$

Consider $m = 2^n - 2^k + 1$

Consider $$m = 2^n - 2^k + 1$$

Algorithm 2 Theoretic-Reduce(a, m) — Reduce a by $m = 2^n - 2^k + 1$

Require: $0 \le a$

Ensure: The number returned r satisfies $0 \le r \le 2^n - 1$

```
$aa \leftarrow a$
$r \leftarrow \text{rem}(aa, 2^n)$
$q \leftarrow \text{quo}(aa, 2^n)$
while $q \neq 0$ do
    $aa \leftarrow r + q \cdot (2^k - 1)$
    $r \leftarrow \text{rem}(aa, 2^n)$
    $q \leftarrow \text{quo}(aa, 2^n)$
end while
if $r \ge 2^n - 2^k + 1$ then $r \leftarrow r - (2^n - 2^k + 1)$
end if
return r
```

If k ~ 3/4 n the while loop has at most 4 iterations; If k ~ 1/2 n the while loop has at most 2 iterations;

The modulus that I am using is 2^2^17 – 2^2^10 + 1.

The first column controls how large each entry is.

rand[0..n]	Regular GMP (s)	Simple Reduce (s)
2^2^18	2.04586	0.0336324
2^2^19	6.96865	0.0816615
2^2^20	13.5465	0.243587
2^2^21	26.9513	0.885439
2^2^22	54.3707	3.504

Choosing moduli:

$$m_1 = 2^n - 2^k + 1, m_2 = 2^n - 2^\ell + 1, k > \ell$$

Choosing moduli:

$$m_1 = 2^n - 2^k + 1, m_2 = 2^n - 2^\ell + 1, k > \ell$$

If $n \mod (k-\ell) = k \mod (k-\ell)$ or $n \mod (k-\ell) = 0$ then $\gcd(m_1, m_2) = 1$.

Similar conditions hold for different choice of +/- signs.

Additional requirement is related to the sparsity of inverses… and scalability. (Maple workbook)

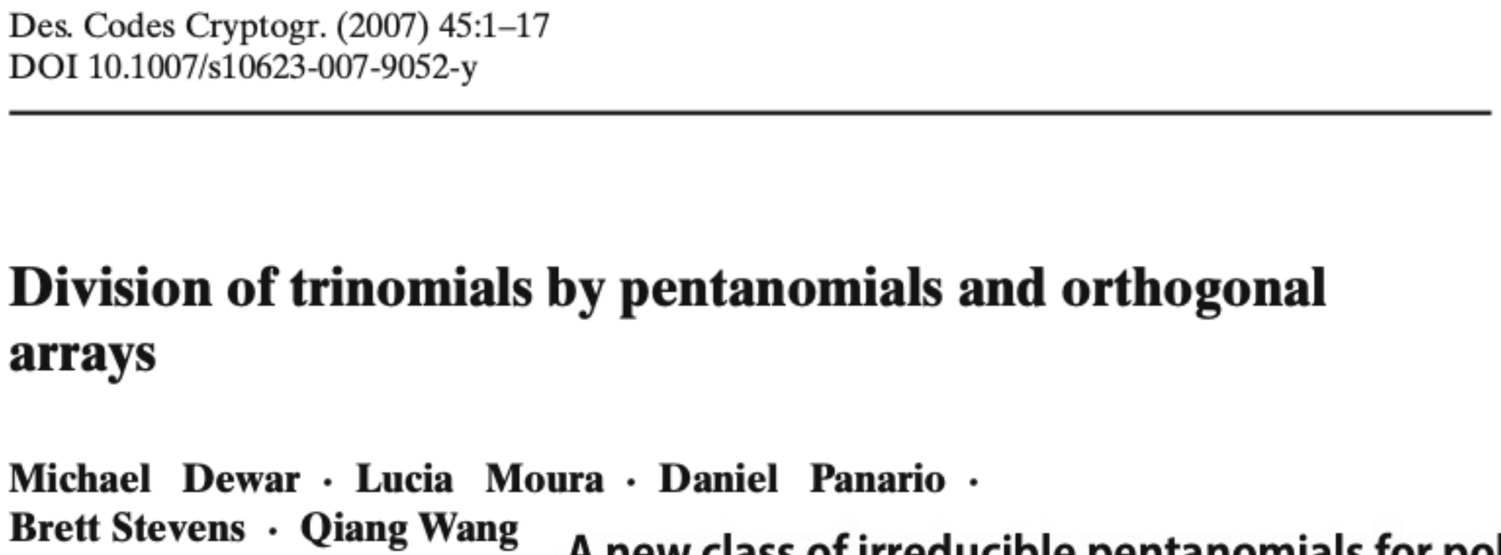
The following moduli are used as the base of the fast reconstruction

$LL := [x^{100} - x^{60} + 1, x^{100} - x^{55} + 1, x^{100} - x^{50} + 1, x^{100} - x^{40} + 1, x^{100}, x^{100} + 1]$

scalingFactor	randBits	garnerTimeUsed	fastReconstructionTimeUsed	Ratio
100	15000	0.00129975	0.000233988	5.55
100	20000	0.0012949	0.000223788	5.79
100	25000	0.00140769	0.00028692	4.91
100	30000	0.00145901	0.000329512	4.43
100	35000	0.00152148	0.000399555	3.81
100	40000	0.00138351	0.000416346	3.32
1000	150000	0.0206575	0.00253134	8.16
1000	200000	0.0192453	0.00200695	9.59
1000	250000	0.0202483	0.00274796	7.37
1000	300000	0.0209492	0.00337775	6.20
1000	350000	0.0218142	0.00371647	5.87
1000	400000	0.0224212	0.00410989	5.46
10000	1500000	0.320023	0.0203086	15.76
10000	2000000	0.336093	0.0311475	10.79
10000	2500000	0.349585	0.0424044	8.24
10000	3000000	0.358395	0.0524323	6.84
10000	3500000	0.373805	0.05753	6.50
10000	4000000	0.379093	0.0625795	6.06

Relation with polynomials (trinomials, pentanomials, fewnomials).

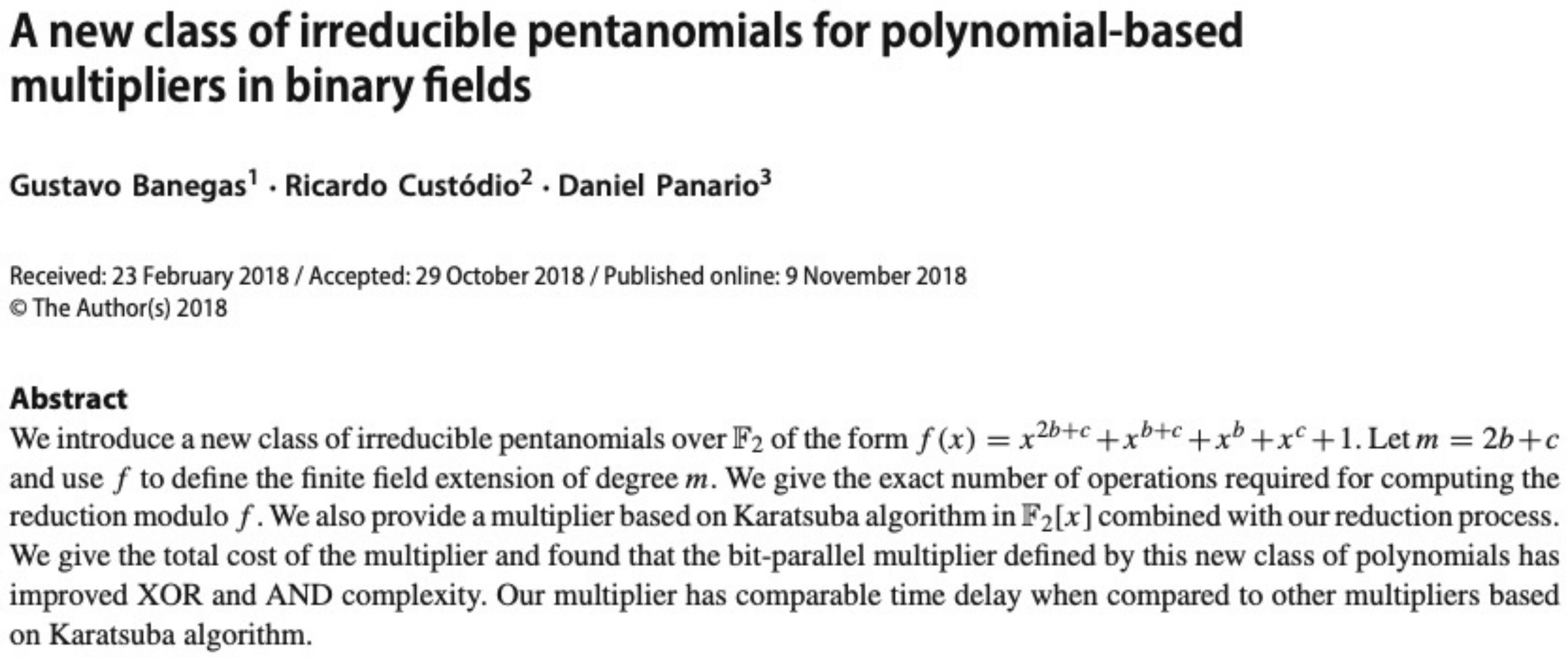
Relation with polynomials (trinomials, pentanomials, fewnomials).



Des. Codes Cryptogr. (2007) 45:1–17
DOI 10.1007/s10623-007-9052-y

Division of trinomials by pentanomials and orthogonal arrays

Michael Dewar · Lucia Moura · Daniel Panario · Brett Stevens · Qiang Wang



A new class of irreducible pentanomials for polynomial-based multipliers in binary fields

Gustavo Banegas[1] · Ricardo Custódio[2] · Daniel Panario[3]

Received: 23 February 2018 / Accepted: 29 October 2018 / Published online: 9 November 2018

Abstract
We introduce a new class of irreducible pentanomials over $\mathbb{F}_2$ of the form $f(x) = x^{2b+c} + x^{b+c} + x^b + x^c + 1$. Let $m = 2b + c$ and use f to define the finite field extension of degree m. We give the exact number of operations required for computing the reduction modulo f. We also provide a multiplier based on Karatsuba algorithm in $\mathbb{F}_2[x]$ combined with our reduction process. We give the total cost of the multiplier and found that the bit-parallel multiplier defined by this new class of polynomials has improved XOR and AND complexity. Our multiplier has comparable time delay when compared to other multipliers based on Karatsuba algorithm.

Thank you !

Application with one modulus – Fermat number

$$\zeta(3) \approx \frac{1}{2} \sum_{n=0}^{N-1} \frac{(-1)^n \left(205\, n^2 + 250\, n + 77\right) \left((n+1)!\right)^5 (n!)^5}{\left((2\, n+2)!\right)^5}.$$

$$\zeta(3) \approx \frac{1}{2} \sum_{n=0}^{N-1} \frac{(-1)^n \left(205\,n^2 + 250\,n + 77\right) \left((n+1)!\right)^5 (n!)^5}{\left((2\,n+2)!\right)^5}.$$

It was first shown that the reduced fraction $S(N)$ has numerator and denominator whose sizes are $O(N)$ instead of the $O(N \log N)$ fraction computed by standard

$$\zeta(3) \approx \frac{1}{2}\sum_{n=0}^{N-1} \frac{(-1)^n\,(205\,n^2+250\,n+77)\,((n+1)!)^5\,(n!)^5}{((2\,n+2)!)^5}.$$

Given positive integers r and m, the rational number reconstruction problem is to find a and b such that $r \equiv ab^{-1} \bmod m$, $\gcd(b,m) = 1$, $|a| < \sqrt{m}/2$, and $0 < b \le \sqrt{m}$ [12]. The recovered fraction is also reduced. Numerous algorithms exist to solve this problem with a time complexity of $O(M(d)\log d)$ if the bit length of m is $O(d)$ [14], [15].

$$\zeta(3) \approx \frac{1}{2} \sum_{n=0}^{N-1} \frac{(-1)^n \left(205\, n^2 + 250\, n + 77\right) \left((n+1)!\right)^5 \left(n!\right)^5}{\left((2\, n + 2)!\right)^5}.$$

Theorem 1: If p is an odd prime, then any divisor of the Mersenne number $M_p = 2^p - 1$ is of the form $2kp + 1$ where k is a positive integer.