Cunningham numbers in accelerated modular arithmetic and applications

Eugene Zima

Physics and Computer Science Department

Wilfrid Laurier University, Waterloo

e-mail: ezima@wlu.ca

ISSAC 2016

The 41st International Symposium on Symbolic and Algebraic Computation
Wilfrid Laurier University, Waterloo, Ontario, Canada, July 19-22, 2016

ISSAC 2016

The International Symposium on Symbolic and Algebraic Computation (ISSAC) is the premier conference for research in symbolic computation and computer algebra. ISSAC 2016 will be the 41st meeting in the series, which started in 1966 and has been held annually since 1981. The conference presents a range of invited speakers, tutorials, poster sessions, software demonstrations and vendor exhibits with a center-piece of contributed research papers.

ISSAC 2016 was held on July 20-22, 2016, at **Wilfrid Laurier University**, Waterloo, Ontario, Canada.

Sponsored by:

Wilfrid Laurier University
LAURIER
Inspiring Lives.

Maplesoft
Mathematics • Modeling • Simulation

MapleDiscovery

FIELDS

University of Waterloo | IQC

“Mersenne” and “Fermat” numbers in accelerated modular arithmetic and applications

Eugene Zima

Physics and Computer Science Department

Wilfrid Laurier University, Waterloo

e-mail: ezima@wlu.ca

Modular arithmetic;
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)
Application (one large modulus)

Given input A, B of size n, compute

$$A \odot B \to C$$

Given input A, B of size n, compute

$$A \odot B \to C$$

Select $m_1, m_2, \ldots m_k$

Map the input:

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

Given input A, B of size n, compute

$$A \odot B \to C$$

Select $m_1, m_2, \ldots m_k$

Map the input:

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

Compute

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

Given input A, B of size n, compute

$$A \odot B \rightarrow C$$

Select $m_1, m_2, \ldots m_k$

Map the input:

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

Compute

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

Reconstruct C from modular images:

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

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

Eliminating Divisions

Eliminating Divisions

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

[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: $(B \Rightarrow 1;0)$.

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

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

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

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.

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 $e_1, e_2, \ldots, e_b$ 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.

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

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

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.

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

Initial reduction and tools

Algorithm 2 BITS(a, m, n) — Bit-range extraction

Require: $0 \leq a,\ m \leq n$

Ensure: The number returned r satisfies $0 \leq r < 2^{n-m}$

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 SIMPLE-REDUCE-MINUS(a, n) — Reduce a by $2^n - 1$

Require: $0 \leq a < 2^{2n}$

Ensure: $0 \leq a \leq 2^n - 1$

1: $t \leftarrow \text{BITS}(a, n, 2n - 1)$
2: $a \leftarrow \text{BITS}(a, 0, n - 1)$
3: $a \leftarrow a + t$
4: $t \leftarrow \text{BITS}(a, n, n)$
5: $a \leftarrow \text{BITS}(a, 0, n - 1) + t$

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

Require: $0 \leq a$

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

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: $\quad r \leftarrow r +$ BITS$(a, i \cdot n, (i + 1) \cdot n - 1)$
5: **end for**
6: **if** $r \geq 2^n$ **then**
7: $\quad$ **return** THEORETIC-REDUCE-MINUS(r, n)
8: **else**
9: $\quad$ **return** r
10: **end if**

Algorithm 5 DC-REDUCE-MINUS(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: **while** $a \geq 2^n$ **do**
3: $\quad b \leftarrow (|a| - 1)/n + 1$ $\{|a|$ measures the bit length of $a\}$
4: $\quad b \leftarrow b/2$
5: $\quad$ mpz_tdiv_q_2exp$(t, a, b \cdot n)$
6: $\quad$ mpz_tdiv_r_2exp$(a, a, b \cdot n)$
7: $\quad a \leftarrow a + t$
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: $\quad a \leftarrow a + 2^n + 1$
8: **end if**
9: mpz_clear(t)

Reconstruction

Algorithm 5 Garner(r, m)

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

1: $M \leftarrow 1$
2: **for** $i = 1$ **to** $N-1$ **do**
3: $\quad M \leftarrow Mm_{i-1}$
4: $\quad M_i \leftarrow M^{-1} \bmod m_i$
5: **end for**
6: $a_0 \leftarrow r_0$
7: **for** $i = 1$ **to** $N-1$ **do**
8: $\quad t \leftarrow a_{i-1}$
9: $\quad$ **for** $j = i-2$ **downto** 0 **do**
10: $\quad\quad t \leftarrow tm_j$
11: $\quad\quad t \leftarrow t + a_j$
12: $\quad$ **end for**
13: $\quad t \leftarrow r_i - t$
14: $\quad a_i \leftarrow tM_i \bmod m_i$
15: **end for**
16: $a \leftarrow a_{N-1}$
17: **for** $i = N-1$ **downto** 0
18: $\quad a \leftarrow am_i$
19: $\quad a \leftarrow a + a_i$
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: $a \equiv r_i \bmod m_i$

```
 1: M ← 1
 2: for i = 1 to N − 1 do
 3:     M ← M m_{i−1}
 4:     M_i ← M^{−1} mod m_i
 5: end for
 6: a_0 ← r_0
 7: for i = 1 to N − 1 do
 8:     t ← a_{i−1}
 9:     for j = i − 2 downto 0 do
10:         t ← t m_j
11:         t ← t + a_j
12:     end for
13:     t ← r_i − t
14:     a_i ← t M_i mod m_i
15: end for
16: a ← a_{N−1}
17: for i = N − 1 downto 0
18:     a ← a m_i
19:     a ← 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.

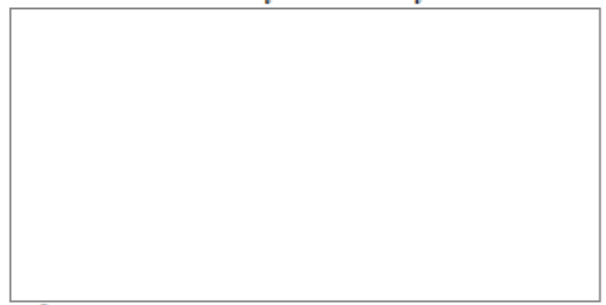
Algorithm 5 Garner(r, m)

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

1: $M \leftarrow 1$
2: **for** $i = 1$ **to** $N-1$ **do**
3: $\quad M \leftarrow Mm_{i-1}$
4: $\quad M_i \leftarrow M^{-1} \bmod m_i$
5: **end for**
6: $a_0 \leftarrow r_0$
7: **for** $i = 1$ **to** $N-1$ **do**
8: $\quad t \leftarrow a_{i-1}$
9: $\quad$ **for** $j = i-2$ **downto** 0 **do**
10: $\quad\quad t \leftarrow tm_j$
11: $\quad\quad t \leftarrow t + a_j$
12: $\quad$ **end for**
13: $\quad t \leftarrow r_i - t$
14: $\quad a_i \leftarrow tM_i \bmod m_i$
15: **end for**
16: $a \leftarrow a_{N-1}$
17: **for** $i = N-1$ **downto** 0
18: $\quad a \leftarrow am_i$
19: $\quad a \leftarrow a + a_i$
20: **end for**
21: **return** a

Algorithm 5 Garner(r, m)

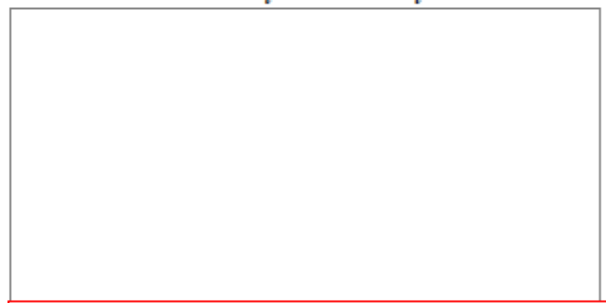
Require: $\forall i \in [0, N-1]: 0 \leq r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$



6: $a_0 \leftarrow r_0$
7: **for** $i = 1$ **to** $N - 1$ **do**
8: $\quad t \leftarrow a_{i-1}$
9: $\quad$ **for** $j = i - 2$ **downto** 0 **do**
10: $\quad\quad t \leftarrow tm_j$
11: $\quad\quad t \leftarrow t + a_j$
12: $\quad$ **end for**
13: $\quad t \leftarrow r_i - t$
14: $\quad a_i \leftarrow tM_i \bmod m_i$
15: **end for**
16: $a \leftarrow a_{N-1}$
17: **for** $i = N - 1$ **downto** 0
18: $\quad a \leftarrow am_i$
19: $\quad a \leftarrow a + a_i$
20: **end for**
21: **return** a

Algorithm 5 Garner(r, m)

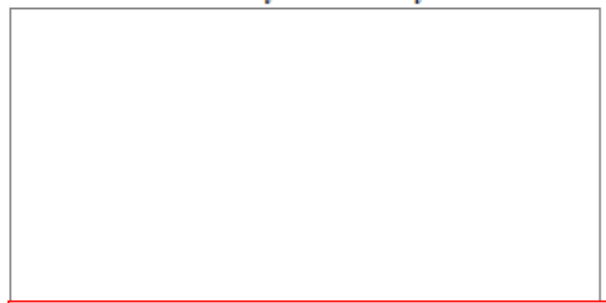
Require: $\forall i \in [0, N-1]: 0 \leq r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$



6: $a_0 \leftarrow r_0$
7: **for** $i = 1$ **to** $N - 1$ **do**
8: $\quad t \leftarrow a_{i-1}$
9: $\quad$ **for** $j = i - 2$ **downto** 0 **do**
10: $\quad\quad t \leftarrow tm_j$
11: $\quad\quad t \leftarrow t + a_j$
12: $\quad$ **end for**
13: $\quad t \leftarrow r_i - t$
14: $\quad a_i \leftarrow tM_i \bmod m_i$
15: **end for**
16: $a \leftarrow a_{N-1}$
17: **for** $i = N - 1$ **downto** 0
18: $\quad a \leftarrow am_i$
19: $\quad a \leftarrow a + a_i$
20: **end for**
21: **return** a

Algorithm 5 Garner(r, m)

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



6: $a_0 \leftarrow r_0$
7: **for** $i = 1$ **to** $N - 1$ **do**
8: $\quad t \leftarrow a_{i-1}$
9: $\quad$ **for** $j = i - 2$ **downto** 0 **do**
10: $\quad\quad t \leftarrow tm_j$
11: $\quad\quad t \leftarrow t + a_j$
12: $\quad$ **end for**
13: $\quad t \leftarrow r_i - t$
14: $\quad a_i \leftarrow tM_i \bmod m_i$
15: **end for**
16: $a \leftarrow a_{N-1}$
17: **for** $i = N - 1$ **downto** 0
18: $\quad a \leftarrow am_i$
19: $\quad a \leftarrow a + a_i$
20: **end for**
21: **return** a

In lines 1–5 of the algorithm, intermediate values $(\prod_{j=0}^{i-1} m_j)^{-1} \bmod m_i$ 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 $X \equiv r_i \pmod{m_i}$, $i = 0, 1, \ldots, N-1$. In lines 16–20, the value of X is calculated using Horner's method.

Algorithm 6 Simple–Reconstruction(r, m)

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

1: $M \leftarrow 1$
2: **for** $i = 1$ **to** $N-1$ **do**
3: $\quad M \leftarrow Mm_{i-1}$
4: $\quad M_i \leftarrow M^{-1} \bmod m_i$
5: **end for**
6: $a \leftarrow r_0$
7: $M \leftarrow m_0$
8: **for** $i = 1$ **to** $N-1$ **do**
9: $\quad t \leftarrow r_i - a$
10: $\quad t \leftarrow tM_i \bmod m_i$
11: $\quad a \leftarrow a + tM$
12: $\quad M \leftarrow Mm_i$
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: $a \equiv r_i \bmod m_i$

1: $M \leftarrow 1$
2: **for** $i = 1$ **to** $N - 1$ **do**
3: $\quad M \leftarrow Mm_{i-1}$
4: $\quad M_i \leftarrow M^{-1} \bmod m_i$
5: **end for**
6: $a \leftarrow r_0$
7: $M \leftarrow m_0$
8: **for** $i = 1$ **to** $N - 1$ **do**
9: $\quad t \leftarrow r_i - a$
10: $\quad t \leftarrow tM_i \bmod m_i$
11: $\quad a \leftarrow a + tM$
12: $\quad M \leftarrow Mm_i$
13: **end for**
14: **return** a

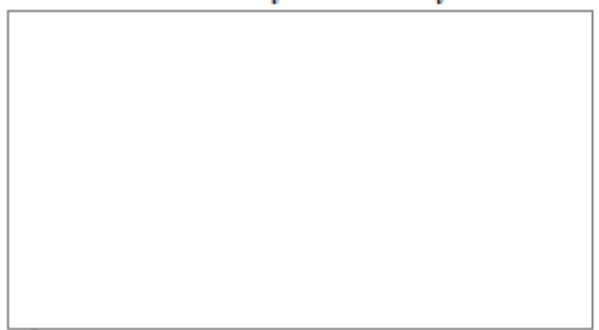
Algorithm 6 Simple–Reconstruction(r, m)

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

1: $M \leftarrow 1$
2: **for** $i = 1$ **to** $N - 1$ **do**
3: $\quad M \leftarrow Mm_{i-1}$
4: $\quad M_i \leftarrow M^{-1} \bmod m_i$
5: **end for**
6: $a \leftarrow r_0$
7: $M \leftarrow m_0$
8: **for** $i = 1$ **to** $N - 1$ **do**
9: $\quad t \leftarrow r_i - a$
10: $\quad t \leftarrow tM_i \bmod m_i$
11: $\quad a \leftarrow a + tM$
12: $\quad M \leftarrow Mm_i$
13: **end for**
14: **return** a

Algorithm 6 Simple–Reconstruction(r, m)

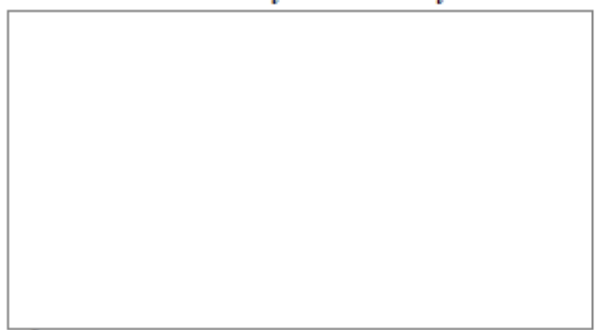
Require: $\forall i \in [0, N-1]: 0 \leq r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$



6: $a \leftarrow r_0$
7: $M \leftarrow m_0$
8: **for** $i = 1$ **to** $N - 1$ **do**
9: $\quad t \leftarrow r_i - a$
10: $\quad t \leftarrow tM_i \bmod m_i$
11: $\quad a \leftarrow a + tM$
12: $\quad M \leftarrow Mm_i$
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: $a \leftarrow r_0$
7: $M \leftarrow m_0$
8: **for** $i = 1$ **to** $N - 1$ **do**
9: $\quad t \leftarrow r_i - a$
10: $\quad t \leftarrow tM_i \bmod m_i$
11: $\quad a \leftarrow a + tM$
12: $\quad M \leftarrow Mm_i$
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: $a \leftarrow r_0$
7: $M \leftarrow m_0$
8: **for** $i = 1$ **to** $N - 1$ **do**
9: $t \leftarrow r_i - a$
10: $t \leftarrow tM_i \bmod m_i$
11: $a \leftarrow a + tM$
12: $M \leftarrow Mm_i$
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.

Application with one modulus – Fermat number

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

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

(2)

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

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

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 $0 < b \leq \sqrt{m}$ [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.