FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Complexity analysis of binary algorithms for modular inversion
Student:	Bc. Ivana Trummová
Supervisor:	prof. Ing. Róbert Lórencz, CSc.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2020/2021

Instructions

Study algorithms for modular inversion from publications [1] and [2].
Express their computational complexity and compare them with the complexity of other algorithms found in publications that cite them.
From the acquired knowledge, try to find a suitable recommendation for modifying the binary algorithms [1] and [2] for modular inversion to improve their computational complexity.

_

[1] R. Lórencz, New algorithm for classical modular inverse, International Workshop on Cryptographic Hardware and Embedded Systems, 57-70, Springer, Berlin, Heidelberg, 2002.
[2] R. Lórencz, J. Hlaváč: Subtraction-free Almost Montgomery Inverse algorithm. Information Processing Letters, Volume 94, Issue 1, 2005, Pages 11-14, ISSN 0020-0190.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 8 January 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Complexity analysis of binary algorithms for modular inversion

Bc. Ivana Trummová

Department of Information Security
Supervisor: prof. Ing. Róbert Lórencz, CSc.

May 6, 2021

Acknowledgements

Firstly, I would like to thank my supervisor prof. Ing. Róbert Lórencz, CSc. for his time and safe space to think about new ideas and for many pieces of advice.
I also thank my friends Tomáš and Jan for helping me with algebra and programming, and my partner Lukáš, without whom I would not be able to finish.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 6, 2021 . . . .

Citation of this thesis

Trummová, Ivana. *Complexity analysis of binary algorithms for modular inversion.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Modulární inverze je operace, která se v moderní vědě a technice hojně využívá – zejména v kryptografii. Existuje více způsobů, jak modulární inverzi najít, a hledání ideálního způsobu stále není u konce. V této práci představujeme analýzu složitosti vybraných algoritmů a některé z nápadů z relevantní literatury, jak tyto algoritmy vylepšit.

Klíčová slova Modulární inverze, Montgomeryho modulární inverze, složitost, modulární aritmetika.

Abstract

Modular inverse is a widely used operation in modern science and technology, particularly in cryptography. There are many ways how to find modular inverse of an integer and the research to find the ideal one is still active. In this work, we present a complexity analysis of several chosen algorithms and some of the ideas about improving them drawn from relevant literature.

Keywords Modular inverse, Montgomery modular inverse, complexity, modular arithmetic.

Contents

List of Tables

Introduction

Motivation

In cryptography, most of the current widely used algorithms depend on the same or similar mathematical principals, such as discrete mathematics, in particular finite fields and modular arithmetic. In many applications, the speed of the computation is a crucial parameter of the quality of an algorithm. In other cases, for mobile devices, the power consumption may be the key quality to look for in algorithms. Therefore we aim to simplify the methods that we know and try to come up with solutions that are still proved to ensure a correct output, but spend less time and energy in order to do it.

Use of modular inverse

Multiplicative inverse in a Galois field is an operation that is useful in various cryptographic algorithms. The most known usage is probably RSA encryption, where the inverse is computed during the decipherment phase [1]. Also, it is used in certain digital signature systems [2], in computing point operations on elliptic curves defined over a Galois field [3, 4], or in addition-subtraction chain [5, 6].

Goal of the thesis

There are various methods for computing modular inverse of an element in a finite field. The most naive approach is the use of Extended Euclid's algorithm, where the inverse appears as a by-product of "searching" for the greatest common divisor of two relatively prime numbers (which is 1). Other algorithms, such as Binary Euclid's algorithm, Penk's algorithm, and Montgomery's algorithm, evolved from the basic concept of Euclid.

In 2002, professor Lórencz published a paper [7] called New Algorithm for Classical Modular Inverse. He proposed an algorithm that is based on binary Euclid's algorithm, but instead of using right shifts, additions and subtractions, he focuses on minimizing the number of operations that cost more time and effort of the processing unit. The proposed algorithm uses left shifts (which are used to realize multiplication by two) as a basic idea. The purpose of this work is to revisit these algorithms, research relevant literature, search for an eventual improvements and propose a complexity analysis.

Organization of the chapters

The initial chapter *Theoretical background* (1) covers basic definitions and mathematical concepts which are necessary for the reader to understand the topic. The second part of this chapter called *Algorithm Zoo* (1.5) covers a family of algorithms computing modular inverse – from basic Extended Euclid's algorithm to newer methods.

In the second chapter (2), an idea of Lórencz's Left shift algorithm is described.

In chapter 3 called *Research on the topic*, there is an overview of the existent literature and publications that cite from [7] and [8]. Several papers are mentioned and we talk about optimization ideas.

Chapter 4 – *Tao Wu algorithm* proposes a detailed look into Tao Wu's paper [9] about a simplified version of Left shift algorithm.

The last chapter (5), *Complexity analysis*, contains a statistical analysis of operations used by simulated algorithms.

Chapter **1**

Theoretical background

This section covers basic concepts and mathematical definitions needed for a comfortable reading of the text, and it also defines the terminology used in the work. Most of these concepts are a part of algebra and modular (residual) arithmetic and are paraphrased from [10] and [11].

Let $\mathcal{I}$ denote the set of all integers.

1.1 Basic concepts

Definition 1.1.1. Divisibility
Let $a, b \in \mathcal{I}$, $a < b$. We say that a *divides* b (and write $a \mid b$) if $b = ac$ for some $c \in \mathcal{I}$.

Definition 1.1.2. Greatest common divisor
Let $a, b \in \mathcal{I}$. Then there is at least one integer c, $c > 0$, which divides both a and b (1 has always this property). The greatest integer that divides both a and b is called *greatest common divisor* of a and b, or $\gcd(a, b)$.

Definition 1.1.3. Prime numbers
Integer p is *prime*, if there are no other integers that divide p other than 1 and p itself.

Definition 1.1.4. Co-primality
Two integers a, b are *co-prime* or *relatively prime*, if their $\gcd(a, b) = 1$.

1.2 Modular arithmetic and Galois fields

Definition 1.2.1. Bézout's coefficients
For any $a, b \in \mathcal{I}$, there exists their greatest common divisor $\gcd(a, b) \in \mathcal{I}$, and two coefficients $u, v \in \mathcal{I}$ (*Bézout's coefficients*) that hold

$$gcd(a, b) = a \cdot u + b \cdot v \tag{1.1}$$

In the literature, this claim is called the *Bézout's theorem.* The theorem and identity 1.1 are key elements for computing modular inverse by Extended Euclid's algorithm (shown later).

Definition 1.2.2. Congruence modulo
Let $a, b, m \in \mathcal{I}$, $m > 1$. Then a is *congruent* to b modulo m (we write $a \equiv b \pmod{m}$) if $m \mid (a - b)$. *Congruence modulo* m is an equivalence relation, so it has properties of symmetry, reflexivity and transitivity.

Definition 1.2.3. Ring, commutative ring, ring with identity
Let $\mathcal{R}$ be a set with 0, and operations addition $(+)$ and multiplication $(\cdot)$. Then $(\mathcal{R}, +, \cdot)$ is called a ring if, for all $a, b, c \in \mathcal{R}$ it holds:

Associativity $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

Commutativity of addition $a + b = b + a$

Zero Special element 0 has property $0 + a = a + 0 = a$

Additive inverse For every a there exists element $-a$ such that $a + (-a) = 0$

Distributivity $(a + b) \cdot c = a \cdot b + a \cdot c$

Additionally, if $a \cdot b = b \cdot a$ for any $a, b \in \mathcal{R}$, the ring $\mathcal{R}$ is called a *commutative ring.*
If there is element $1 \in \mathcal{R}$ with the property $a \cdot 1 = 1 \cdot a = a$ for every $a \in \mathcal{R}$, then the ring $\mathcal{R}$ is called a *ring with identity.*

Definition 1.2.4. Multiplicative inverse
Let $\mathcal{R}$ be a commutative ring with identity, $0 \neq 1$, $a \in \mathcal{R}$. A *multiplicative inverse* of a is $a^{-1} \in \mathcal{R}$, which holds $a \cdot a^{-1} = a^{-1} \cdot a = 1$.

Definition 1.2.5. Field, finite field, Galois field
A commutative ring with identity $\mathcal{R}$, where $0 \neq 1$ and for every $a \neq 0 \in \mathcal{R}$ there is a multiplicative inverse, is called a *field*.
If a field has a finite number of elements, it is called a *finite field*.
A field that contains q elements (where q is a power of a prime number) is called *Galois field*. It is denoted GF(q).

We will only focus on GF(p), p is a prime number greater than 2. When we say we compute or search for modular inverse of an element, we mean multiplicative inverse of this element in GF(p), there p is the modulus.

1.3 Montgomery modular inverse

Let $\mathcal{I}_m = \{0, 1, 2, \ldots, m-1\}$, $m \in \mathcal{I}, m > 1$.

Definition 1.3.1. Least non-negative residue
Each integer $a \in \mathcal{I}$ is congruent modulo m to exactly one element of set $\mathcal{I}_m$. This creates a map $|\cdot|_m : \mathcal{I} \mapsto \mathcal{I}_m$ defined as $|a|_m = r$, $0 \leq r < m$ and $a \equiv r$. Integer r is called the *least non-negative residue* of a modulo m. The map has these important properties:
If $a, b, m \in \mathcal{I}$ and $m > 1$:

1. $|a|_m$ is unique.
2. $|a|_m = |b|_m$ if and only if $a \equiv b \pmod{m}$.
3. $|k \cdot m|_m = 0$ for every $k \in \mathcal{I}$.

Definition 1.3.2. Montgomery domain
The Montgomery representation of a residue $|a|_m \in \mathcal{I}$ is defined as integer $|b|_m = |a \cdot R|_m$, where $R \in \mathcal{I}$ is a radix co-prime to m, $R > m$.
In our case, R is a power of 2.

Definition 1.3.3. Addition and subtraction in Montgomery domain
Let us have $|a|_m, |b|_m, |c|_m \in \mathcal{I}_m$ and their Montgomery representations $|a \cdot R|_m, |b \cdot R|_m, |c \cdot R|_m \in \mathcal{I}_m$. Then we define addition and subtraction as follows:

$$\big|\, |a \cdot R|_m + |b \cdot R|_m \,\big|_m = |c \cdot R|_m \iff \big|\, |a|_m + |b|_m \,\big|_m = |c|_m$$

$$\big|\, |a \cdot R|_m - |b \cdot R|_m \,\big|_m = |c \cdot R|_m \iff \big|\, |a|_m - |b|_m \,\big|_m = |c|_m$$

Definition 1.3.4. Montgomery multiplication
As opposed to addition and subtraction, which can be performed in the same way as in integer domain, multiplication in Montgomery domain needs an extra step. Since we multiply two elements, both of which are multiples of R, the product has to be divided by one R to stay in the Montgomery domain.

$$\big|\, |a \cdot R|_m \cdot |b \cdot R|_m \cdot |R^{-1}|_m \,\big|_m = |c \cdot R|_m \Longleftrightarrow \big|\, |a|_m \cdot |b|_m \,\big|_m = |c|_m$$

From now on, suppose $R = 2^n$.

Definition 1.3.5. Montgomery multiplication algorithm
Let $\overline{a} = |a \cdot R|_m$ and $\overline{b} = |b \cdot R|_m$ denote the Montgomery representations. Let $\overline{a} = (\overline{a}_{n-1}\overline{a}_{n-2}\ldots\overline{a}_0)_2$, where $\overline{a}$ is in base 2 with $\overline{a}_0$ =LSB. Let $0 \leq \overline{b} < m$. Basic Montgomery multiplication algorithm goes as follows:

IN: $\overline{a}, \overline{b}, m, n$

OUT: $|\overline{a}\overline{b}R^{-1}|_m$

1. $s := 0, i := 0$
2. while $(i < n)$:
3. $\quad x := x + \overline{a}_i\overline{b}$
4. $\quad x := (x + x_0 m)/2)$
5. $\quad i := i + 1$
6. if $(x \geq m)$:
7. $\quad x := x - m$
8. return $x = |\overline{a}\overline{b}R^{-1}|_m$

Once we have the Montgomery representations of our integers, this algorithm is very efficient. The important feature to notice is that a multiplication and division by 2 is performed, which is a very simple operation.

Definition 1.3.6. Montgomery modular inverse
Let $a, p \in \mathcal{I}$, $a \in [1, p-1]$. *Montgomery modular inverse* of a or MMI(a) is defined as an integer $b \in [1, p-1]$ such that

$$b \equiv a^{-1} 2^n \pmod{p} \tag{1.2}$$

where a is relatively prime to p and $n = \lceil \log_2 p \rceil$.

Definition 1.3.7. Almost Montgomery inverse
In some algorithms, an intermediate result called *Almost Montgomery inverse* of an integer a modulo prime p is calculated:

$$AMI(a) \equiv a^{-1} 2^k \pmod{p},$$

where $a \in [1, p-1], p$ is a prime and $k \in [n, 2n]$ is the number of iterations performed.
As authors of [8] mention in the paper:

$$MMI(a) \equiv AMI(a) \cdot 2^{n-k} \equiv a^{-1} 2^n \pmod{p}.$$

Once we get Almost Montgomery inverse as an intermediate result, the algorithm can either use a second phase to go back to integer domain (which is shown later in 1.5), or by multiplying by 2, we can quickly get the Montgomery representation of the inverse. This might be useful if we want to continue to work with the value – the properties of Montgomery domain allow us to quickly multiply or use different methods.

1.4 Computational complexity

Definition 1.4.1. Big O notation
Let $\mathcal{N}$ denote the set of natural numbers. If f, g are two functions from $\mathcal{N}$ to $\mathcal{N}$, then we say that $f = \mathcal{O}(g)$ if there exists a constant c such that $f(n) \leq c \cdot g(n)$ for sufficiently large n.

1.5 Algorithm zoo

There are plenty of approaches and methods of of computing modular inverse. In mathematics, a problem is often solved by transformation of the original task to another. The first idea of how to efficiently compute modular inverse (better than guessing) was to obtain it as a by-product of computing the greatest common divisor of two integers. Therefore we have to begin with the Euclid's algorithm.

1.5.1 Euclid's Algorithm

Euclid's algorithm is a method for computing the greatest common divisor (gcd) of two integers without having to factorize them. It is one of the most basic algebraic algorithms that we know of, having its origin in ancient Greece (about 300 BC). Most of the modern modular inverse algorithms are based on this method. The algorithm is based on two properties of greatest common divisor. For any integers a, b:

$$gcd(a, 0) = |a|; \tag{1.3}$$

$$gcd(a, b) = gcd(b, a \pmod{b}). \tag{1.4}$$

In order to find the greatest common divisor, we would repeatedly substitute the larger of two values by the remainder of their division, until we reach zero. At that point, the other value is equal to their gcd. The Euclid's algorithm goes as follows:

- IN: integers $a, b, a > b$
- OUT: $gcd(a, b)$
- While $(b \neq 0)$:
  $r = a \pmod{b}$
  $a = b$
  $b = r$
- Return $a = gcd(a, b)$.

Definition 1.5.1. Let a be an integer. Then $L(a)$ indicates the number of digits in particular base (length).

Time complexity of Euclid's algorithm is $\mathcal{O}((n-d+1)m) \leq \mathcal{O}(nm)$, where $n = L(a)$, $m = L(b)$, $d = L(gcd(a,b))$. Full proof is in [12]. The algorithm halts with the correct output because of the equations 1.3 and 1.4, which are relevant in every Euclidean domain (definition and description of Euclidean domain are in [10]).

1.5.2 Extended Euclid's Algorithm

This is the first method by which we actually can find the multiplicative inverse of an element of finite field. The key piece of knowledge is Bézout's theorem. The Extended Euclid's algorithm goes as follows:

- IN: integers $a, b, a > b$
- OUT: $gcd(a,b)$ and coefficients u, v that hold 1.1.
- $a_0 = a, \quad u_0 = 1, \quad v_0 = 0;$
  $a_1 = b, \quad u_1 = 0, \quad v_1 = 1;$
  $a_{i+1} = r, \quad u_{i+1} = u_{i-1} - u_i q, \quad v_{i+1} = v_{i-1} - v_i q,$, where q, r are chosen to satisfy
  $$a_{i-1} = a_i q + r, r < a_i. \tag{1.5}$$
  If $a_{i+1} = 0$, return $a_i = 1, u := u_i, v := v_i$.

Based on equation 1.1, we can use this algorithm for finding the modular inverse in case $gcd(a,b) = 1$, which is always true for two co-prime integers. Then, using the Bézout's theorem, we have

$$a \cdot u = 1 - b \cdot v \equiv 1 \pmod{b} \tag{1.6}$$

Since b is the modulus, $(-b \cdot v) \equiv 0 \pmod{b}$ and this leaves u as an inverse of $a \pmod{b}$.

Time complexity is the same as in the case of Euclid's algorithm. The only difference in every step is several additions and subtractions, and their asymptotic complexity is lower or the same as division.

1.5.3 Binary Algorithm

The binary version of the previous algorithm was designed for implementation – we wanted to avoid integer division, because that is a complex operation. This version uses division by two, that can be realized just by shifting the bits to the right and changing there position by one place. The correctness is ensured by these equations (as Knuth writes in [13]):
Let a, b be positive integers. Then

$$gcd(a, b) = 2gcd(a/2, b/2) \tag{1.7}$$

for a, b both even,

$$gcd(a, b) = gcd(a/2, b) \tag{1.8}$$

for a even and b odd,

$$gcd(a, b) = 2gcd(|a - b|, b), \tag{1.9}$$

$$a - b \text{ is even}, |a - b| < max(a, b) \tag{1.10}$$

for a, b both odd.

Proof. Let a, b be integers. Then we can write

$$a = p_1^{k_1} \cdot p_2^{k_2} \cdot \ldots \cdot p_m^{k_m}, b = p_1^{l_1} \cdot p_2^{l_2} \cdot \ldots \cdot p_n^{l_n},$$

where $k_i, l_i \geq 0$ for $i = 0, \ldots, n$. By definition 1.1.2 we can write

$$gcd(a, b) = p_1^{min(k_1,l_1)} \cdot p_2^{min(k_2,l_2)} \cdot \ldots \cdot p_n^{min(k_n,l_n)}.$$

(1.7) Since a, b are both even, we can write $a = 2 \cdot a', b = 2 \cdot b'$ for integers $a' = a/2, b' = b/2$.

We have to prove that $gcd(2a', 2b')$ divides $2gcd(a', b')$ and vice versa. Let $c = gcd(2a', 2b')$. Since $c \mid 2a'$, there exists an integer x that holds $2a' = cx$, and since $c \mid 2b'$, there exists an integer y that holds $2b' = cy$. 2 is a common divisor of $2a', 2b'$, so there is an integer z with the property $c = 2z$. We can write $2a' = 2zx$ and $2b' = 2zy$, from which we get $a' = zx, b' = zy$, and z is a common divisor of a', b', therefore $z \mid gcd(a', b')$. Hence $c = 2z \mid 2 \cdot gcd(a', b')$.

Reversely, $gcd(a', b') \mid a', b'$, so $2 \cdot gcd(a', b')$ divides both $2a'$ and $2b'$, thus it has to divide their common divisor.

(1.8) Let us consider the factorization of a, b mentioned above. The greatest common divisor of even a and odd b must be odd by the definition and must be the same even if we divide a by two, because in the factorization, the minimal exponent of 2 is 0.

(1.9) Validity of this equation follows the fact that

$$gcd(a, b) = gcd(a, b - qa)$$

for any integer q. This holds because any common divisor of a and b is a divisor of both a and $a - qb$, and, conversely, any common divisor of a and $a - qb$ must divide both a and b. See [13].

□

Given two positive integers a and b, the binary algorithm finds their greatest common divisor. This algorithm is described and analyzed more in depth in [13].

IN positive integers a, b

OUT $gcd(a, b)$

1. **Find the power of two.** Set $k \leftarrow 0, u \leftarrow a, v \leftarrow b$. Repeatedly set $k \leftarrow k + 1$, $u \leftarrow a/2$, $v \leftarrow v/2$, zero or more times until one of the values u, v is odd
2. **Initialize.** If u is odd, set $t \leftarrow -v$ and go to *4*, otherwise set $t \leftarrow u$
3. **Halve t.** Set $t \leftarrow t/2$
4. **Is t even?** If t is even, go back to *3*
5. **Reset $\max(u, v)$.** If $t > 0$, set $u \leftarrow t$, otherwise set $v \leftarrow -t$
6. **Subtract.** Set $t \leftarrow u - v$. If $t \neq 0$, go back to *3*. Otherwise the algorithm terminates with $u \cdot 2^k$, which is the desired gcd.

1.5.4 Penk's Algorithm

As we saw in the binary version of the Euclid's algorithm, many steps of the computation can be transformed and executed by different operations. The general idea is to avoid any operation that would be costly in terms of computational complexity, such as integer division. As a result, many methods rely on a heavier use of cheaper operations such as shifting the bits to the right (division by two) or to the left (multiply by two).

One of the ways to compute the modular inverse efficiently is a *right-shift* approach. This right-shift algorithm 1 is attributed to M. Penk and uses the Euclidean method as a base (see [13]) – a version from [7] is presented.

As Lai writes in [14], there are several main elements that keep the structure the same. The main *while* loop with a conditional test for v resembles the original Euclidean test for b. The three main branches in the loop (line 4, 9 and 14 in 1) represent all the cases which happen for values u, v and the ways this algorithm proceeds with them according to the equations 1.7, 1.8, 1.9 – these relations are used to avoid integer division. The *if* conditions after the main loop (lines 24, 26) are used as corrections to output a result that is in the right interval.

Output consists of r, the modular inverse, and k, the number of halvings of u and v.

1.5.5 Montgomery Algorithm

Montgomery algorithm consists of two phases – the first one uses the Montgomery domain to compute an intermediate result – Almost Montgomery inverse (output y in algorithm 2). The fact that this method uses another algebraic structure gives us the benefit of fewer operations. On the other hand, the downside is that a second phase is needed to convert the output back to the integer domain using multiple shifts (divisions by two) and offsets by module p (additions).

The key difference between Montgomery and the previous Penk's classical algorithm is that the Montgomery computes the Almost Montgomery inverse quickly, but needs extra time for the correction phase.

Algorithm 1: Penk

```
Input: a ∈ [1, p − 1] and p
Output: r ∈ [1, p − 1] and k, where r = a^{-1} (mod p),
        and n ≤ k ≤ 2n
u := p, v := a, r := 0, s := 1
k = 0
while v > 0 do
    if u is even then
        if r is even then
            u := u/2, r := r/2, k := k + 1
        else
            u := u/2, r := (r + p)/2, k := k + 1
    else if v is even then
        if s is even then
            v := v/2, s := s/2, k := k + 1
        else
            v := v/2, s := (s + p)/2, k := k + 1
    else
        x := (u − v)
        if x > 0 then
            u := x, r := r − s
            if r < 0 then
                r := r + p
        else
            v := −x, s := s − r
            if s < 0 then
                s := s + p
if r > p then
    r := r − p
if r < 0 then
    r := r + p
return r, k
```

Algorithm 2: Montgomery – Phase I

Input: $a \in [1, p-1]$ and p
Output: $y \in [1, p-1]$ and k, where $y = a^{-1}2^k \pmod{p}$,
and $n \leq k \leq 2n$

$u := p, v := a, r := 0, s := 1$
$k = 0$
while $v > 0$ **do**
  if u *is even* **then**
    $u := u/2, s := 2s, k := k+1$
  else if v *is even* **then**
    $v := v/2, r := 2r, k := k+1$
  else
    $x := (u - v)$
    if $x > 0$ **then**
      $u := x/2, r := r + s, s := 2s, k = k+1$
    else
      $v := -x/2, s := r + s, r := 2r, k = k+1$
if $r > p$ **then**
  $r := r - p$
return $y = p - r, k$

Algorithm 3: Montgomery – Phase II

Input: $y \in [1, p-1]$, p and k from Phase I
Output: $y \in [1, p-1]$, where $r = a^{-1} \pmod{p}$, and $2k$ from Phase I

for $i = 1$ *to* k **do**
  if r *is even* **then**
    $r := r/2$
  else
    $r := (r + p)/2$
return r *and* $2k$

1.5.6 Almost Montgomery algorithm (Kaliski)

Algorithm 4 is very similar to Montgomery algorithm 2, but the output is slightly different – this method also computes Almost Montgomery inverse, but less amount of loop iterations is used and k is smaller than in algorithm 2. This version is proposed by Kaliski and published in [8] – it doesn't contain the second phase, but we may assume it is the same, since the output here is

$$o \equiv a^{-1}2^k \pmod p, n-1 \leq k \leq 2n.$$

Algorithm 4: AMI with subtractions

Input: $a \in [1, p-1]$ and p

Output: $o \in [1, p-1]$ and k, where $o = a^{-1}2^k \pmod p$, and $n-1 \leq k \leq 2n$

```
u := p, v := a, r := 0, s := 1
k = 0
while 1 do
    if u is even then
        u := u/2, s := 2s
    else if v is even then
        v := v/2, r := 2r
    else
        x := (u − v), y = r + s
        if x = 0 then
            return o = s, k
        if CARRY(x) = 1 then
            u := x/2, r := y, s := 2s
        else
            v := −x/2, s := y, r := 2r
    k = k + 1
```

1.5.7 Subtraction free AMI

In [8], Lórencz and Hlaváč propose a modification of the Kaliski's version of Almost Montgomery algorithm 4. The main difference is that algorithm 5 uses addition instead of subtraction, and as shown later in the statistical analysis, it brings a slight improvement regarding the number of operations – this method avoids the operation of negation completely. Just as the method above, this algorithm's output is $AMI(a)$, and second phase is needed for computation of the classical modular inverse of a.

Algorithm 5: Subtraction-free AMI

Input: $a \in [1, p-1]$ and p
Output: $o \in [1, p-1]$ and k, where $o = a^{-1}2^k \pmod{p}$,
and $n - 1 \leq k \leq 2n$

$u := -p, v := a, r := 0, s := 1$
$k = 0$
while 1 **do**
  if u *is even* **then**
    $u := u/2, s := 2s$
  else if v *is even* **then**
    $v := v/2, r := 2r$
  else
    $x := (u + v), y = r + s$
    if $x = 0$ **then**
      return $o = s, k$
    if $CARRY(x) = 0$ **then**
      $u := x/2, r := y, s := 2s$
    else
      $v := x/2, s := y, r := 2r$
  $k = k + 1$

1.5.8 Left Shift Algorithm

The *Left shift* algorithm is the proposed new method to effectively compute the classical modular inverse in [7] (New Algorithm for Modular Inverse).

The main approach was to avoid the drawbacks of the algorithms that use right shifts (algorithms 1, 2). Both of these algorithms are using operations additions and subtractions and this algorithm is designed with the intention of limited use of addition and subtraction, and rather uses bigger amount of left shifts. Algorithm 6 keeps the variables u, v (that represent the master thread) aligned to the left – so when a subtraction is performed, it clears the leading bit(s) (MSB). Shifting the variables to the left is performed in the main while loop, where the condition checks if u or v still can shift to the left. When both variables are aligned, a subtraction (or addition in case one of different signs) is performed.

Algorithm 6: Left-Shift Algorithm

Input: $a \in [1, p-1]$ and p

Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod p$, c_u, c_v and $0 < c\_v + c\_u \leq 2n$

```
u := p, v := a, r := 0, s := 1
c_u = 0, c_v = 0
while (u ≠ ±2^{c_u} & v ≠ ±2^{c_v}) do
    if (u_n, u_{n-1} = 0) or (u_n, u_{n-1} = 1& OR(u_{n-2}, ..., u_0) = 1) then
        if (c_u ≥ c_v) then
            u := 2u, r := 2r, c_u := c_u + 1
        else
            u := 2u, s := s/2, c_u := c_u + 1
    else if (v_n, v_{n-1} = 0) or (v_n, v_{n-1} = 1& OR(v_{n-2}, ..., v_0) = 1) then
        if (c_v ≥ c_u) then
            v := 2v, s := 2s, c_v := c_v + 1
        else
            v := 2v, r := r/2, c_v := c_v + 1
    else
        if (v_n = u_n) then
            oper = "−"
        else
            oper = "+"
        if (c_u ≤ c_v) then
            u := u oper v, r := r oper s
        else
            v := v oper u, s := s oper r
if (v = ±2^{c_v}) then
    r := s, u_n := v_n
if (u_n = 1) then
    if (r < 0) then
        r := −r
    else
        r := p − r
if (r < 0) then
    r := r + p
return r, c_u, and c_v
```

Chapter 2

Proposal of a proof of Left shift algorithm correctness

In [7], there is a mathematical proof that Left shift algorithm (algorithm 6) halts within a finite number of steps and with correct output. The proof operates with the master part of the computation (variables u, v). In this chapter, I would like to propose an alternative point of view on the correctness and try to describe the way how the algorithm operates – and cover also the slave part of the computation, variables r, s and their relation to the correct output.

There are two key equations that values hold throughout the algorithm and are useful to understand the structure of the method and how the results are computed.

$$\frac{u}{2^d} \equiv r \cdot a \pmod{p} \tag{2.1}$$

$$\frac{v}{2^d} \equiv s \cdot a \pmod{p} \tag{2.2}$$

where $d = min(c\_u, c\_v)$ and a, p is the input pair of integer and prime modulus.

Theorem 1. Algorithm 6 holds equations 2.1 and 2.2 in every step of the main loop.

Proof. The proof is trivial for the initial step. On lines 1 and 2, we initialize values the following way:

$$u_0 := p, v_0 := a, r := 0, s := 1, c\_u := c\_v := 0.$$

The two equations then look like this:

$$\frac{p}{2^0} \equiv 0 \cdot a \pmod p$$

$$\frac{a}{2^0} \equiv 1 \cdot a \pmod p$$

During the main while loop there are three possibilities of what could happen to the values. Either one of the values is shifted to the left (ii), or an operation of addition or subtraction is applied to them (i).

Suppose that after i-th iteration of the loop equations 2.1 and 2.2 hold for values u_i, v_i, r_i, s_i, and d_i.

1. In the case of performing subtraction, value d_i is not changed, and without loss of generality suppose $c\_u \leq c\_v$ (we can suppose that since neither c_u nor c_v is changed in this branch). Then we have

   $$u_{i+1} \equiv u_i - v_i \equiv r_i \cdot a \cdot 2^{d_i} - s_i \cdot a \cdot 2^{d_i} \equiv (r_i - s_i) \cdot 2^{d_i} \equiv (r_{i+1}) \cdot a \cdot 2^{d_i} \pmod p$$

   and therefore

   $$\frac{u_{i+1}}{2^{d_i}} \equiv r_{i+1} \cdot a \pmod p,$$

   and since $d_i = d_{i+1}$, equation 2.1 still holds, and 2.2 remains untouched. We can use the same justification in the case of performing an addition – we just used basic principals of modular arithmetic.

2. In the case of performing shifting branches (suppose u is shifted to the right, in case of shifting v the discussion would be the same), we have

   $$u_{i+1} := 2u_i, c\_u_{i+1} := c\_u_i + 1.$$

   Depending on c_u_i and c_v_i, there are two possibilities. Firstly suppose $c\_u_i \geq c\_v_i$, then we have $r_{i+1} := 2r_i$ and since c_v_i stays the same (this value is not changed in the u-shifting branch), $d_{i+1} = d_i$. Equation 2.2 remains untouched in this scenario as well, and we have

   $$\frac{u_{i+1}}{2^{d_{i+1}}} \equiv \frac{2u_i}{2^{d_i}} \equiv 2r_i \cdot a \equiv r_{i+1} \cdot a \pmod p.$$

   The last case is when $c\_u_i < c\_v_i$. When we shift u in this case, c_u_i is the minimum of two counters and is increased, so $d_{i+1} = d_i + 1$ and $s_{i+1} := s_i/2$. The first equation now looks like this:

   $$\frac{u_{i+1}}{2^{d_{i+1}}} \equiv \frac{2u_i}{2^{d_i+1}} \equiv r_i \cdot a \equiv r_{i+1} \cdot a \pmod p.$$

The second equation also holds:

$$\frac{v_{i+1}}{2^{d_{i+1}}} \equiv \frac{v_i}{2^{d_i+1}} \equiv \frac{s_i}{2} \cdot a \equiv s_{i+1} \cdot a \pmod p.$$

$\square$

2.1 Example on particular values

Here, the idea of the proof is showed on particular values. Let us have $(a, p) = (10, 13)$; the initialization step is:

$$u_0 := 13, v_0 := 10, r := 0, s := 1, c\_u := c\_v := 0.$$

In the table below, we have the example of the calculation (the same as in [7]).

l	operations	values of registers	tests
0		$u^{(0)} = (13)_{10} = (01011.)_2$ $v^{(0)} = (10)_{10} = (01010.)_2$ $r^{(0)} = (0)_{10} = (00000.)_2$ $s^{(0)} = (1)_{10} = (00001.)_2$	$u^{(0)} \neq \pm 2^0$ $v^{(0)} \neq \pm 2^0$
1	$u^{(1)} = u^{(0)} - v^{(0)}$ $r^{(1)} = r^{(0)} - s^{(0)}$	$u^{(1)} = (3)_{10} = (00011.)_2$ $v^{(1)} = (10)_{10} = (01010.)_2$ $r^{(1)} = (-1)_{10} = (11111.)_2$ $s^{(1)} = (1)_{10} = (00001.)_2$	$u^{(1)} \neq \pm 2^0$ $v^{(1)} \neq \pm 2^0$
2	$u^{(2)} = 4u^{(1)}$ $r^{(2)} = 4r^{(1)}$	$u^{(2)} = (12)_{10} = (011.00)_2$ $v^{(2)} = (10)_{10} = (01010.)_2$ $r^{(2)} = (-4)_{10} = (111.00)_2$ $s^{(2)} = (1)_{10} = (00001.)_2$	$u^{(2)} \neq \pm 2^2$ $v^{(2)} \neq \pm 2^0$
3	 $v^{(3)} = v^{(2)} - u^{(2)}$ $s^{(3)} = s^{(2)} - r^{(2)}$	$u^{(3)} = (12)_{10} = (011.00)_2$ $v^{(3)} = (-2)_{10} = (11110.)_2$ $r^{(3)} = (-4)_{10} = (111.00)_2$ $s^{(3)} = (5)_{10} = (00101.)_2$	$u^{(3)} \neq \pm 2^2$ $v^{(3)} \neq \pm 2^0$
4	 $v^{(4)} = 4v^{(3)}$ $r^{(4)} = r^{(3)}/4$	$u^{(4)} = (12)_{10} = (011.00)_2$ $v^{(4)} = (-8)_{10} = (110.00)_2$ $r^{(4)} = (-1)_{10} = (11111.)_2$ $s^{(4)} = (5)_{10} = (00101.)_2$	$u^{(4)} \neq \pm 2^2$ $v^{(4)} \neq \pm 2^2$

5	$u^{(5)} = u^{(4)} + v^{(4)}$ $r^{(5)} = r^{(4)} + s^{(4)}$	$u^{(5)} = (4)_{10} = (001.00)_2$ $r^{(5)} = (4)_{10} = (00100.)_2$	$u^{(5)} = 2^2$

For $l = 0$, equations 2.1 and 2.2 look like this in the beginning:

$$\frac{13}{2^0} \equiv 0 \cdot 0 \pmod{13}$$

$$\frac{10}{2^0} \equiv 1 \cdot 10 \pmod{13}$$

The next iteration $l = 1$ corresponds to the case (i) – performing subtraction. Value $d_1 = 0$ remains unchanged, and we have

$$\frac{u_1}{2^{d_0}} \equiv \frac{3}{2^0} \equiv -1 \cdot 10 \pmod{13} \equiv r_1 \cdot a \pmod{p},$$

whereas the second equation remains untouched. Iteration $l = 2$ shows the case (ii). In this case, two left shifts were performed on u, and because v has not been shifted yet, $d = 0$, so we have

$$\frac{u_2}{2^{d_2}} \equiv \frac{4u_1}{2^{d_1}} \equiv \frac{12}{2^0} \equiv 4 \cdot (-1) \cdot 10 \equiv 4r_1 \cdot a \equiv r_2 \cdot a \pmod{13}.$$

For $l = 3$, we have once again the case (i), where u is subtracted from v.

$$\frac{v_3}{2^{d_2}} \equiv \frac{-2}{2^0} \equiv 5 \cdot 10 \pmod{13} \equiv s_3 \cdot a \pmod{p},$$

and for $l = 4$, we have again two left shifts applied on v – so now $d = 2$.

$$\frac{v_4}{2^{d_4}} \equiv \frac{4v_3}{2^{d_3+2}} \equiv \frac{-8}{2^2} \equiv 5 \cdot 10 \equiv s_4 \cdot a \pmod{13}.$$

$$\frac{u_4}{2^{d_4}} \equiv \frac{u_3}{2^{d_3+2}} \equiv \frac{12}{2^2} \equiv (-1) \cdot 10 \equiv r_4 \cdot a \pmod{13}.$$

Last iteration $l = 5$ is addition:

$$\frac{u_5}{2^{d_5}} \equiv \frac{4}{2^2} \equiv (4) \cdot 10 \equiv r_5 \cdot a \pmod{13}.$$

Then the loop ends, because loop condition is satisfied: $u_5 = 2^2$ and we already have the result from the last equation for $l = 5$ saved in variable r.

Chapter 3

Research on the topic

There are currently 55 articles, theses and papers that cite Lorencz's work [7], and 7 publications that cite paper [8]. The vast majority of them only cite the paper as one of many resources or as a literature for further reading, or usually as an example of previous work in the field. However, some of them take the paper more into consideration and focus more on the analysis of the Left-shift algorithm. These are the papers we will focus on.

3.1 Further work

3.1.1 Lai

In 2004, Gerald Lai published Analysis of Modular Inverse GF(p) Implementations [14]. His paper examines five classical modular (or Montgomery) inverse algorithms in $GF(p)$. In his words, he attempted to study the evolution of modular inversion methods and to trace key areas of improvement efficiency of hardware improvement.

This paper is not an experimental study. Lai does not implement the algorithms or count the operations, but he dives deep into explaining the steps in the methods and their evolution. He suggests a way how to understand individual steps and operations, and describes how the modern algorithms transformed from binary extended Euclidean algorithm in order to be more efficient and cost less.

Regarding Lorencz's Left-shift algorithm, Lai notes this:

'(...) benchmark results that show algorithmic performance do not necessarily reflect hardware implementation performance and costs for the same algorithm. For example, the number of addition and subtraction operations is indicative of how often the arithmetic units are active for the payload of computations. While this may be useful for certain power usage estimations, it does not provide a complete picture of the hardware costs in building the design. If the delay is essential, the critical path of a particular hardware can be the limiting factor and hence, different implementations of the arithmetic units can be applied to offset the effects of carry propagation.'

3.1.2 Hars

In 2006, Laszlo Hars presented improved algorithms for computing classical modular inverse of large integers, without multiplications of any kind [15]. He included Lorencz's algorithm and he also added a comparison between improved algorithms.

Hars proposes a justification, where he explains the way how the Left shift algorithm was created. Then he suggests an improvement (or speedup technique) for the Left shift algorithm in part *3.2.2. Best left shift: algorithm LS3*.

Apart from describing the Left shift algorithm, Hars also writes about possible improvements of right shift algorithm, where he introduces a plus-minus trick that helps decrease the length of operands and speeds up the process.

The plus-minus trick is a modification often used for the right shift algorithm: if u and v are odd, check if $u + v$ has 2 trailing 0 bits, otherwise we know that $u - v$ does. In the former case, if $u + v$ is of the same length as the larger of them, the right shift operation reduces the length by 2 bits from this larger length, otherwise by only one bit. It means that the length reduction is sometimes improved, so the number of iterations decreases.

This plus-minus trick does not work for the Left shift algorithm, because the addition never clears the MS bit (and the shifts are only to the left, so we would like to clear these bits on the left). A subtraction could, on the other hand, clear one or more MS bits (if u and v are close), or we could try $2u - v$ or $2v - u$ if these values are closer. Hars proposes an improved algorithm called LS3 – after 3 possible reductions above. With the knowledge of a few

MS bits one could determine which one of the three reductions will give the largest amount of decrease in length of the operands.

Also, in the conclusion of his paper, he writes about further optimizations of the algorithm – how to speed it up a little further. When u and v become short, a table lookup could speed up the finishing calculations. If only one of them becomes small (short), or there is a large difference between u and v, we could perform a different algorithmic step which would be best for the particular computing platform. However, Hars also notes that all of the ideas would be only quite small improvement of speed.

3.1.3 Shivashankar

Shivashankar [16] discusses the implementation of all three algorithms in [7]. Regarding Left shift algorithm, he describes the implementation in several parts and adds hardware design.

The Left shift algorithm is divided into logic units by variables. There is a part for counters c_v and c_u, for the operands u, v, r, and s and the last bit is for the final computation of the output (after the main while loop). Although there is no new idea of an optimization in Shivashankar's paper, the hardware designs could be of use in further exploration.

3.1.4 Choi

In 2015, Choi published an analysis of three modular inverse algorithms performance [17]. He compares right shift algorithm (RS), Left shift algorithm (LS – 6) and Eucleidean modular inverse (EM). The basic concepts of the three algorithms are as follows. LS aligns variables u and v to the left, subtracts the smaller number of u and v from the bigger, substitutes the bigger number with the result, repeats alignment, subtraction and substitution. RS performs right shift alignment, and EM performs division instead of alignment and subtraction.

The analysis has been done in the following way: The authors implemented all of the three algorithms and stated this for the measuring part:

'(...) To analyze the processing time of each algorithm, one or both of the shift and subtraction operations should be taken into account. Since shift operations are sometimes performed in a row without subtraction for alignment, shift operations are be performed on every clock cycle; while subtraction operations are selectively performed. This means that analysis on the processing speed is done by just counting shift operations, and the number of subtraction operations are not considered for performance analysis unlike software implementation.'

Since the Left shift algorithm is designed in such a way that it minimises the use of additions and subtractions and uses larger amount of shifts instead, in this analysis, RS shows the best performance, and uses less synthesised area then LS and EM. Although by Choi's analysis RS shows to be the best in terms of performance, the analysis apparently doesn't take into consideration that a shift operation applied by itself is much cheaper than a shift followed by a subtraction.

3.1.5 Wu

In a paper from Tao Wu [9], he proposes a new version of Lórencz's algorithm, he calls it "Simplified algorithm". This algorithm was similar to the original, with small tweaks, and it deserved a more complex analysis, therefore a whole chapter is dedicated to this paper.

3.1.6 Liu

In 2017, Liu and collective published a paper [18] that focuses on efficiently computable endomorphisms in elliptic curves. They develop several optimizations of different algorithms, including an optimization of Montgomery Modular inverse algorithm. Their algorithm is optimized for pseudo-Mersenne primes.

Definition 3.1.1. Pseudo-Mersenne prime is a *prime* of the form

$$p = 2^m - k,$$

where k is an integer for which

$$0 < |k| < 2^{\lfloor m/2 \rfloor}.$$

If $k = 1$, then p is a *Mersenne prime* (and m must necessarily be a prime). If $k = -1$, then p is called a *Fermat prime* (and m must necessarily be a power of two).

Algorithm consists of two phases. In the beginning of Phase I, two additions are performed (Algorithms 4 and 5 also work with this step, but inside the *if-else* block). Then, in *if-else* block, according to the sign flag of $x = u+v$, variables $\{u, v, r, s, k\}$ are updated. The new building brick is the operation $DET(x)$ – trailing zero detection. Function $DET(x)$ counts how many bits of x are zeroes (starting from LSB), in other words how many times x can be divided by two (right-shifted) without loss of information. Operations $\gg$ and $\ll$ denote shifting to the right or left by a particular number of bits. Right and left shifts are not executed one per each iteration, but variables are shifted by the number of trailing zeros (lines 11, 13, 15, 17). The core idea is to remove all trailing zeros of $(u + v)$ in every iteration, which keeps u and v always odd so that $(u + v)$ converges to zero very quickly.

Algorithm 7: Optimized Montgomery algorithm for $2^n - c$

Input: $a \in [1, 2^n)$ and is odd, and $p > 2$, n-bit prime, precomputed $T = 2^{-2n} \pmod p$

Output: $r \in [1, 2^n)$, where $r = a^{-1} \pmod p$

```
1  \\ Phase I
2  u := −p, v := a, r := 0, s := 1
3  k = 0
4  while 1 do
5      x := (u + v)
6      y := (r + s)
7      tlz_x := DET(x)
8      if x = 0 then
9          break;
10     else if x < 0 then
11         u := x ≫ tlz_x
12         r := y
13         s := s ≪ tlz_x
14     else
15         v := x ≫ tlz_x
16         s := y
17         r := r ≪ tlz_x
18     k := k + tlz_x
19 \\ Phase II
20 s = s · 2^(2n−k) (mod p)
21 s = s · T (mod p)
```

3.2 Ideas to follow up

One of the main goals of this work is to try to find suitable recommendations for simplification, optimization or decrease of computational complexity of classical or Montgomery modular inverse algorithms.

The first idea appears in the publication [15] from Hars. He introduces a variation of plus-minus trick for LS algorithm. Then he proposes various optimizations that could increase the speed a little bit e.g. using a table lookup when values u, v become short enough.

Tao Wu implemented the idea into the *Simplified* Left shift algorithm. This idea, unfortunately, doesn't bring desired simplification, as explained in the next chapter.

The most interesting idea seems to be the trailing zeros detection proposed by Liu in [18]. The downside of the particular design of Algorithm 7 is that it does not ensure the correct or effective run and result for even integers on the input. However, this fact doesn't mean that the trailing zero detection couldn't be a part of design of another, correct algorithm computing modular inverse.

Chapter **4**

Tao Wu algorithm

In [9] Tao Wu proposes a simplified version of the Left shift algorithm. He divides it into two phases (8, 9). Main difference between the algorithm 6 and this simplified version is that Tao Wu has left out two *if-then-else* blocks and substituted them by only the second of the two branches. The conditions from lines 5–9 and 11–15 in 6 are replaced by lines 5 and 7 in 8. The second main difference is that the first phase 8 doesn't output the modular inverse, but it gives us only values r, c_v that satisfy equation below:

$$r \cdot 2^{c-v} \pmod{p} \equiv a^{-1} \pmod{p} \tag{4.1}$$

Computing of the final modular inverse is implemented in Phase II, which we can see in 9.

Although it appears that this algorithm computes classical modular inverse with less conditions and therefore less operations, it does not ensure correct output if we stick to the exact version proposed in the article. The problematic part of the algorithm is line 5 in algorithm 8. Since an if-then-else block has been omitted, there is no guarantee whether value s is even (In algorithm 6, we use right shift only if s or r is even, and that is achieved by comparing the amount of left shifts, hence the c_u, c_v tests). It may happen that $s = \pm 1$ (or $r = \pm 1$ on line 7), and we end up with undefined value: $s/2 = 1/2$, but we are not in rational numbers. Since the operation "division by two" is realized by right shift, the "desired" output in this case is zero, but that means loss of information and incorrect output of the algorithm. An example with values $a = 4, p = 13$ is presented below in appendix B. Phase 1 ends with results

$[r, c\_v] = [12, 2]$. Phase 2 now computes $y = r \cdot 2^{c-v} \pmod p = 11 \neq 10$, which would in this case be the correct output.

Algorithm 8: Tao Wu – Phase I

Input: $a \in [1, p-1]$ and p
Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod p$, c_u, c_v
and $0 < c\_v + c\_u \leq 2n$

$u := p, v := a, r := 0, s := 1$
$c\_u = 0, c\_v = 0$
while $(u \neq \pm 2^{c-u}$ & $v \neq \pm 2^{c-v})$ **do**
    if $(u_n, u_{n-1} = 0)$ *or* $(u_n, u_{n-1} = 1\& \,\mathrm{OR}\,(u_{n-2}, \ldots, u_0) = 1)$ **then**
        $u := 2u, s := s/2, c\_u := c\_u + 1$
    else if $(v_n, v_{n-1} = 0)$ *or* $(v_n, v_{n-1} = 1\& \,\mathrm{OR}\,(v_{n-2}, \ldots, v_0) = 1)$ **then**
        $v := 2v, r := r/2, c\_v := c\_v + 1$
    else
        if $(v_n = u_n)$ **then**
            oper = " − "
        else
            oper = " + "
        if $(c\_u \leq c\_v)$ **then**
            $u := u$ oper $v, r := r$ oper s
        else
            $v := v$ oper $u, s := s$ oper r
if $(v = \pm 2^{c-v})$ **then**
    $r := s, u_n := v_n, c\_v := c\_u$
if $(u_n = 1)$ **then**
    if $(r < 0)$ **then**
        $r := -r$
    else
        $r := p - r$
if $(r < 0)$ **then**
    $r := r + p$
return $r, c\_u,$ *and* c_v

Algorithm 9: Tao Wu – Phase II

Input: r, c_v, p from Phase 1

Output: $y = r \cdot 2^{c-v} \pmod p = a^{-1} \pmod p$

for $i = 1$ *to* c_v **do**
  r = 2r **if** $r \geq p$ **then**
    $r := r - p$

return $y := r$

4.1 Proposed corrections of Tao Wu's algorithm

The original objective was to compare algorithms 6 and 8, 9 to decide which is less complex and show if any progress has been made, but we can see that in case of algorithm by Tao Wu, there is no guarantee of a correct output (however, for some inputs, it does work). In order to have something relevant to compare, a few corrections had to be made.

4.1.1 Odd value divided by two

There are three parts where one need to patch algorithm 8 to make it work correctly. As stated in [14], checking for evenness or oddness is done very easily by checking the least significant bit. However, on line 5 and 7 there is no such check that the divided number is even. To avoid a division of an odd number or shifting one to the right, we added a check (lines 6–8, 11–13). If the value is even, the division (right shift) is performed. Otherwise, modulus is added to current value and the value is offset, and since modulus is always an odd prime, we can then divide an even number by two. The calculation remains unchanged, because all the operations are performed modulo p, since we compute in $GF(p)$. With such correction we can avoid losing information and preserving the correctness of the computation.

4.1.2 Value out of range in third branch

The other problem can be the operation of addition or subtraction in the third branch. In some cases it may happen that (possibly due to the solution of division by two) the absolute value of the result of the operation is larger than module p, so we added a line there as well that ensures that the result is in the correct bounds – between 1 and p.

4.1.3 Final correction of the output

Finally, a patch has been added to the end of Phase 2. In some cases, we need to add a final correction that puts the output between 1 and p, that leads to another check. We can see the final functioning algorithm (10, 11). Added corrections are green.

Algorithm 10: Tao Wu with corrections – Phase I

Input: $a \in [1, p-1]$ and p
Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod p$, c_u, c_v and $0 < c\_v + c\_u \leq 2n$

```
u := p, v := a, r := 0, s := 1
c_u = 0, c_v = 0
while (u ≠ ±2^{c_u} & v ≠ ±2^{c_v}) do
    if (u_n, u_{n-1} = 0) or (u_n, u_{n-1} = 1 & OR(u_{n-2}, ..., u_0) = 1) then
        u := 2u, c_u := c_u + 1
        if s is odd then
            s := s + p
        s := s/2
    else if (v_n, v_{n-1} = 0) or (v_n, v_{n-1} = 1 & OR(v_{n-2}, ..., v_0) = 1) then
        v := 2v, c_v := c_v + 1
        if r is odd then
            r := r + p
        r := r/2
    else
        if (v_n = u_n) then
            oper = " − "
        else
            oper = " + "
        if (c_u ≤ c_v) then
            u := u oper v, r := r oper s
            if (r > p) or (r < −p) then
                r := r (mod p);
        else
            v := v oper u, s := s oper r
            if (s > p) or (s < −p) then
                s := s (mod p);
if (v = ±2^{c_v}) then
    r := s, u_n := v_n, c_v := c_u
if (u_n = 1) then
    if (r < 0) then
        r := −r
    else
        r := p − r
if (r < 0) then
    r := r + p
return r, c_u, and c_v
```

Algorithm 11: Tao Wu with corrections – Phase II

Input: r, c_v, p from Phase I

Output: $y = r \cdot 2^{c-v} \pmod{p} = a^{-1} \pmod{p}$

```
for i = 1 to c_v do
    r = 2r
    if r ≥ p then
        r := r − p
    if r < 0 then
        r := r + p
return y := r
```

CHAPTER 5

Complexity analysis

The main goal of my thesis was to express the computational complexity of algorithms in publications [7] and [8] and compare them with algorithms found in publications that cite them. In [7], Lórencz compared algorithms 1, 2 and 6. In this chapter, this comparison is extended and new algorithms are added and analyzed.

5.1 Algorithms

Algorithms used in the comparison were implemented in C language and simulated the sequence of performed operations by design in above-mentioned papers. All of the operations applied on the variables and numbers of loop iterations were counted. The simulations were executed repeatedly for all primes $p < 2^{14} = 16384$ and all $a \in (1, p)$ for each p. In total, each algorithm was executed for 14580841 different input pairs. For the algorithm 7 (Optimized Montgomery, A7), the statistics cover only 14393301 correct outputs (which is 98.71% from the whole dataset). In section 5.4, we can see maximal, minimal and average numbers of uses of each operation.

All of the algorithms that were implemented and analyzed are mentioned also below in the short overview. Also, for better orientation, algorithm are labeled A1–A7 in this chapter.

A1 **Penk's algorithm** (1) – classical modular inverse algorithm. To check which operations were counted, see 12.

A2 **Montgomery algorithm** (2, 3) – consists of two phases (**A2P1**, **A2P2**), which were considered both separately (for comparing with Almost Montgomery algorithms and Optimized Montgomery algorithm) and together (for comparing with classical algorithms). See also 13, 14.

A3 **Lórencz's algorithm** (6) – left shift approach, classical modular inverse algorithm. See 15.

A4 **Kaliski's Algorithm** (4) – Almost Montgomery Inverse algorithm. Needs a second phase (5.4.2). See 16.

A5 **Subtraction Free AMI** (5) – Almost Montgomery Inverse algorithm, version without subtractions. Needs a second phase (5.4.2). See 17.

A6 **Tao Wu algorithm** (10, 11) – corrected version of Tao Wu's "Simplified" Left shift algorithm. See also 18, 19.

A7 **Optimized Montgomery algorithm** (7) – algorithm proposed in [18], optimized for pseudo-Mersenne primes. Does not ensure correct outputs for even numbers. Needs a second phase (5.4.2). See 20.

5.2 Operations

In the table 5.2.1, there is an overview of the operations counted in the statistics. Detailed breakdown of counted operations is in the appendix A. There are several classes of operations separated by the complexity.

5.2.1 Shift

Since all the registers hold values in base 2, shifting or moving bits to the left corresponds to multiplying the value by two, and shifting to the right is division by two. This operation is very cost-effective, because it can be implemented very easily.

Let x be a value that consists of n bits where x_0 is LSB. Lower index $()_2$ denotes the base 2.

$$x = (x_{n-1}, x_{n-2}, \dots x_0)_2$$

Value x shifted to the left and to the right:

$$2x = (x_{n-2}, \dots x_0, 0)_2$$

$$x/2 = (0, x_{n-1}, x_{n-2}, \dots x_1)_2$$

Since the LSB or MSB are cleared away during shifts, one has to check whether the operation makes sense and if it will be performed correctly. When right shift is applied to odd value, a bit of information will be lost and the output won't be correct. If left shift is applied to a value that is too big, it can overflow.

Table 5.1: Operations

name	**label**	**operation**	**comment**
addition	add	$a + b$	
subtraction	sub	$a - b$	
greater than / less than test	test	$a > b; a < b$	realized by subtraction
negation	neg	$a = -a$	needs addition (2's complement)
shift	shift	$a \ll 1; a \gg 1;$	left and right shift, usually multiplying or division by 2
zero comparison	zero	$a > 0, a == 0$	checking zero flag
evenness	even	$2\|a$	even or odd – checking LSB
while loop	loop	loop condition	number of loop iterations
counter incrementation	k	$k++$	counter incrementation

5.2.2 Addition, subtraction

The complexity of both addition and subtraction depends on the length of numbers that are added together or subtracted from each other. One has to take into consideration each digit and the operation cannot be easily simplified because of the carry bit. As described in [12], asymptotic complexity of addition is

$$\mathcal{O}(max(m, n)),$$

where m, n are numbers of digits of the two values. Subtraction falls into the same complexity category. The operation *test* (comparison less than, greater than) is considered to be equivalent operation, because it usually is realized by subtraction (and checking the sign flag afterwards). Since all the algorithms work with 2's complement code for the negative value, operation *neg* (negation of a number) is realized by inverting the bits and adding 1, so the complexity is the same as for an addition.

5.2.3 Zero comparisons

Comparing an integer with zero, checking the sign or checking whether a value is even or odd takes only one bit or flag to look for. Therefore these operations are way faster than addition or subtraction.

5.2.4 Operations on counters

Even though throughout the run of the algorithms we use operations such as addition (more precisely incrementation by one) and testing the size of the counters, these operations fall into a different complexity category, because their size is logarithmic compared to the actual values that we work with. Therefore, these operations were not added to the statistics. The only number which is interesting to see, is the k value in Montgomery algorithm – this is the number of iterations in the second phase.

5.3 Methodology

When choosing the best algorithm, the criteria are the following:

1. The algorithm has to output correct results for all inputs.

2. The algorithm executes (on average) the least amount of expensive operations (additions, subtractions, tests and negations).

3. If more than one algorithms meet the first two criteria, the less overall operations the better.

When we compare the classical methods with Montgomery methods, we have to consider the need of a second phase that transforms the intermediate Almost Montgomery inverse into classical one. In the table, we compare means of all above-mentioned algorithms with each other, and algorithms A2, A4, A5 and A7 are including the operations in the second phase.

5.4 Results

In tables shown throughout this section, we can see the results of counting the operations presented above. The most informative row is the average number of use of each operation – *mean*. Then, for a complete overview of how the data look like, there is standard deviation (*std*), minimal (*min*) and maximal (*max*) number of uses within one algorithm run, and rows *25%, 50%, 75%* denote the quartiles, *50%* being the median. We can use these rows (especially median) to check whether it is approximately equal to mean, to know how well mean describes the dataset.

5.4.1 Classical Modular algorithms

Classical Modular algorithm are A1, A3 and A6. By looking into the first table, we see data describing runs of Penk's right shift method. There is rather heavy use of checking evenness of the values, and operations of subtraction and addition together are in average used over 30 times.

Table 5.2: A1 – Penk's algorithm

	add	**sub**	**test**	**neg**	**shift**	**zero**	**even**	**loop**
mean	14.24	20.16	–	5.13	36.16	20.16	65.04	28.16
std	3.34	2.94	–	1.49	4.71	2.94	6.37	2.48
min	2	4	–	1	4	4	11	5
25%	12	18	–	4	32	18	61	27
50%	14	20	–	5	36	20	66	28
75%	16	22	–	6	40	22	69	30
max	38	28	–	13	52	28	91	39

Statistics regarding the Left shift algorithm in A3 confirm the experimental part of Lórencz's study [7]. We see larger amount of shifts (on average over 40 during an instance), whereas a significant decrease of using additions, subtractions (around 18 altogether) and complete avoidance of using other operations, that would be of significant cost.

In A6 we can see results for Tao Wu algorithm. The intended goal of Tao Wu in [9] was to simplify Left Shift algorithm, however, as discussed in chapter about Tao Wu's algorithm, the proposed method wasn't correctly designed. For the sake of comparing the complexity, a series of corrections were made

Table 5.3: A3 – Left shift algorithm

	add	**sub**	**test**	**neg**	**shift**	**zero**	**even**	**loop**
mean	7.72	10.53	–	–	41.12	–	–	29.69
std	3.49	3.56	–	–	6.56	–	–	5.02
min	0	2	–	–	0	–	–	1
25%	6	8	–	–	38	–	–	27
50%	8	10	–	–	42	–	–	31
75%	10	12	–	–	46	–	–	33
max	24	28	–	–	48	–	–	44

Table 5.4: A6 – Tao Wu's algorithm

	add	**sub**	**test**	**neg**	**shift**	**zero**	**even**	**loop**
mean	18.2	15.84	28.88	9.12	51.76	10.63	20.56	29.69
std	4.96	4.23	5.64	2.3	8.08	1.57	3.28	5.02
min	0	2	2	1	0	0	0	1
25%	15	13	25	8	48	10	19	27
50%	18	16	29	9	54	11	21	31
75%	22	19	33	11	58	12	23	33
max	3	36	52	20	60	12	24	44

and statistics of the operations describe the corrected version. The numbers of used operations, each being a lot higher then those in A3, show, that this approach doesn't appear to be simpler – it's the other way around. On one hand, this table doesn't prove that Tao Wu's idea is useless and that it can't lead us to an improvement. On the other hand, the incorrectness of his design indicates that it might be better to go another way.

5.4.2 Montgomery Modular algorithms

In this section, we compare four algorithms (A2, A4, A5, A7) that are computing Almost Montgomery inverse as an intermediate result, all of which need second phase in order to return to classical modular inverse. The second phase is the same for all of them, since it is only division by a particular power of two.

The only difference between A2P1 and A4 is that A2P1 executes one more iteration. When $x = u - v$ is zero, A4 (and also A5) halts right away, but

Table 5.5: A2P1 – Montgomery's algorithm, Phase I

	add	sub	test	neg	shift	zero	even	loop	k
mean	10.08	10.08	–	5.13	38.16	29.16	33.75	19.08	19.08
std	1.47	1.47	–	1.49	4.71	1.47	3.84	2.36	2.36
min	2	2	–	1	6	2	6	3	3
25%	9	9	–	4	34	9	31	17	17
50%	10	10	–	5	38	10	34	19	19
75%	11	11	–	6	42	11	36	21	21
max	14	14	–	13	54	14	53	27	27

A2P2 proceeds to case $x \leq 0$ and executes line 13 in 2:

$$v = -x/2, s = r + s, r = 2r, k = k + 1.$$

This corresponds to slightly different results: By average, A2P1 executes two more shifts, one more negation and k is greater by 1. Also, Montgomery's algorithm compares to zero much more – on average 29 times in contrast to 19 times in A4 and A5.

Table 5.6: A4 – Almost Montgomery Algorithm (with subtractions)

	add	sub	test	neg	shift	zero	even	loop	k
mean	10.08	10.08	–	4.13	36.16	19.16	33.75	19.08	18.08
std	1.47	1.47	–	1.49	4.71	2.94	3.84	2.36	2.36
min	2	2	–	0	4	3	6	3	2
25%	9	9	–	3	32	17	31	17	16
50%	10	10	–	4	36	19	34	19	18
75%	11	11	–	5	40	21	36	21	20
max	14	14	–	12	52	27	53	27	26

Algorithm A5 appears to be slightly better than A4 – this has been already proved in [8] and this analysis confirms that fact. A4 uses, apart of the same amount of additions and subtractions (on average 10.08 + 10.08, opposed to 20.16 additions in A5), over 4 more negations (A5 doesn't need any). The rest of the table is the same – and this corresponds to the speedup described in [8].

When looking at the results from Optimized Montgomery algorithm (A7), we see similar average values as in the other algorithms – a little more additions, testing evenness, less shifts. The biggest drawbacks of algorithm A7

Table 5.7: A5 – Almost Montgomery Algorithm (without subtractions)

	add	sub	test	neg	shift	zero	even	loop	k
mean	20.16	–	–	–	36.16	19.16	33.75	19.08	18.08
std	2.94	–	–	–	4.71	2.94	3.84	2.36	2.36
min	4	–	–	–	4	3	6	3	2
25%	18	–	–	–	32	17	31	17	16
50%	20	–	–	–	36	19	34	19	18
75%	22	–	–	–	40	21	36	21	20
max	28	–	–	–	52	27	53	27	26

are that the method is not applicable to all of the inputs – some of the inputs where integer a is even don't output the correct results. The data also show that some of the even inputs, even though the output was correct, show absurdly high amounts of used operations – we see the maximum of 1024 additions or 1023 zeros (these values were counted when the input pair was $(a, p) = (2, 1021)$). We see from the values of quartiles that only a minority of runs are this ineffective, but it is still a flaw.

Table 5.8: A7 – Optimized Montgomery algorithm

	add	sub	test	neg	shift	zero	even	loop	k
mean	23.83	–	–	–	32.29	22.82	11.91	11.91	16.15
std	8.65	–	–	–	2.08	8.04	7.73	7.73	2.94
min	6	–	–	–	0	5	3	3	0
25%	20	–	–	–	28	19	10	10	14
50%	22	–	–	–	32	21	11	11	16
75%	26	–	–	–	36	25	13	13	18
max	1024	–	–	–	50	1023	512	512	25

5.5 Complexity comparison

In the table below, we can see the comparison of *means* of every above-mentioned algorithm. Lines corresponding to Montgomery algorithms (A2, A4, A5, A7) are increased by the number of operations used in Phase II (table 5.4.2).

Phase II ensures that intermediate value $AMI(a) = a^{-1}2^k$ is transformed back to a^{-1}, which is done by division by two with occasional addition of the

Table 5.9: A2P2 – Montgomery's algorithm, Phase II

	add	**sub**	**test**	**neg**	**shift**	**zero**	**even**	**loop**	**k**
mean	9.56	–	–	–	19.08	–	19.08	19.08	19.08
std	2.54	–	–	–	2.36	–	2.36	2.36	2.36
min	1	–	–	–	3	–	3	3	3
25%	8	–	–	–	17	–	17	17	17
50%	9	–	–	–	19	–	19	19	19
75%	11	–	–	–	21	–	21	21	21
max	25	–	–	–	27	–	27	27	27

prime module. Phase II in A7 is described in [18] as two multiplications:

$$s = s \cdot 2^{2n-k},$$

$$s = s \cdot T,$$

where s is the register with $AMI(a)$ and $T = 2^{-2n}$ is a precomputed number. However, we can also write

$$s = s \cdot (2^{2n-k}T) = s \cdot 2^{2n-k-2n} = s \cdot 2^{-k},$$

which in the end requires the same amount of the operations as Phase II of other algorithms.

Table 5.10: Means: Algorithms A1–A7

	add	**sub**	**test**	**neg**	**sum**	**shift**	**zero**	**even**	**loop**	**k**
A1	14.24	20.16	–	5.13	39.53	36.16	20.16	65.04	28.16	–
A2	20.04	10.08	–	5.13	35.25	57.24	29.16	52.83	19.08	19.08
A3	7.72	10.53	–	–	18.25	41.12	–	–	29.69	–
A4	20.04	10.08	–	4.13	34.25	55.24	19.16	52.83	19.08	18.08
A5	30.12	–	–	–	30.12	55.24	19.16	52.83	19.08	18.08
A6	18.2	15.84	28.88	9.12	72.04	51.76	10.63	20.56	29.69	–
A7	33.39	–	–	–	33.39	51.37	22.82	33.45	14.37	16.03

The results we focus on are in the column *sum* – that is the total number of complex operations (additions, subtractions, test and negations) used in respective algorithms. Firstly, we have to disqualify algorithm A7 from the race for the fact that it doesn't output correct values for all even numbers. Algorithm A6 has the worst numbers, mainly because of the corrections that have been done. On the other side of the charts, algorithm A3 has the best results of executed operation from all of the simulated algorithms.

5.6 Suggestions for future work

As every thesis, this one has its limitations. Firstly, not all algorithms that compute modular inverse are based on Euclid's algorithm. My research didn't cover those which are not – some of them are mentioned in an overview in [19].

Regarding Tao Wu's algorithm and corrections made in order to make it work correctly, the patches were made with the objective of correctness – effectivity is way harder to achieve. However, with more time and resources it may be possible to find a way how to use his idea and make less time-consuming patches.

In chapter 2 a general idea of Left shift algorithm is described. A future work might find a minimal number of operations needed to compute a classical modular inverse and a mathematical proof.

The complexity analysis in chapter 5 show us only a basic summary of properties of studied algorithms. A more complex study could show us whether some of the algorithms is more suitable for a particular type of inputs. Also, some patches could be done to the Optimized Montgomery algorithm to get correct outputs for every input.

Chapter 6

Conclusion

The main goal of this thesis was to study algorithms for modular inversion, mainly those published in [7] and [8], express their computational complexity and compare them with other algorithms found in publications that cite them, then try to find a suitable recommendation for modifying the binary algorithms to improve their complexity.

Overview and description of all the studied algorithms is in 1.5. All publications that cite papers [7] and [8] have been researched and two possible improved algorithm have been found (Tao Wu's algorithm in [9], Liu's Optimized Montgomery algorithm in [18]) and analyzed.

During the analysis of Tao Wu's algorithm I have discovered several critical problems with proposed algorithm and proposed corrections and patches for the algorithm to output the correct result 4.

A comparison and complexity analysis of all the above-mentioned algorithms was carried out and the results were analyzed in chapter 5. Lórencz's Left shift algorithm has been proved to have executed the least amount of expensive operations.

There are several thoughts about where to go next that appeared during during working on the thesis – trailing zero detection from [18], plus-minus trick and lookup table from [15]. These could be the next steps to explore in future research.

Bibliography

[1] Rivest, R. L.; Shamir, A.; et al. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, volume 21, no. 2, 1978: pp. 120–126.

[2] Fips, P. 186-2. digital signature standard (dss). *National Institute of Standards and Technology (NIST)*, volume 20, 2000: p. 13.

[3] Koblitz, N. Elliptic curve cryptosystems. *Mathematics of computation*, volume 48, no. 177, 1987: pp. 203–209.

[4] Menezes, A. J. *Elliptic curve public key cryptosystems*, volume 234. Springer Science & Business Media, 1993.

[5] Koç, C. K. High-radix and bit recoding techniques for modular exponentiation. *International Journal of Computer Mathematics*, volume 40, no. 3-4, 1991: pp. 139–156.

[6] Eğecioğlu, Ö.; Koç, Ç. K. Exponentiation using canonical recoding. *Theoretical Computer Science*, volume 129, no. 2, 1994: pp. 407–417.

[7] Lórencz, R. New algorithm for classical modular inverse. In *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2002, pp. 57–70.

[8] Lórencz, R.; Hlaváč, J. Subtraction-free almost Montgomery inverse algorithm. *Information processing letters*, volume 94, no. 1, 2005: pp. 11–14.

[9] Tao, W. Proof and Improvement of Shantz's Modular Division and Lórencz's Modular Inverse Algorithms. *DEStech Transactions on Computer Science and Engineering*, , no. cmsam, 2018.

[10] Stanovský, D. *Základy algebry*. Matfyzpress, 2010.

[11] Hlaváč, J. *Hardware Implementation of Algorithms for Computations in Finite Fields*. Dissertation thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2010.

[12] Stanovský, D.; Barto, L. *Počítačová algebra*. Matfyzpress, 2011.

[13] Knuth, D. E. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[14] Lai, G. Analysis of modular inverse gf (p) implementations. *IEEE Trans. Inform. School Of Electrical Engineering And Computer Science, Oregon State University, Corvallis, Oregon*, volume 97331, 2004.

[15] Hars, L. Modular inverse algorithms without multiplications for cryptographic applications. *EURASIP Journal on Embedded Systems*, volume 2006, no. 1, 2006: p. 032192.

[16] Shivashankar, N. *Design and Analysis of Modular Architectures for an RNS to Mixed Radix Conversion Multi-processor*. Dissertation thesis, University of Cincinnati, 2014.

[17] Choi, P.; Kong, J.-T.; et al. Analysis of hardware modular inversion modules for elliptic curve cryptography. In *2015 International SoC Design Conference (ISOCC)*, IEEE, 2015, pp. 313–314.

[18] Liu, Z.; Großschädl, J.; et al. Elliptic curve cryptography with efficiently computable endomorphisms and its hardware implementations for the internet of things. *IEEE Transactions on Computers*, volume 66, no. 5, 2016: pp. 773–785.

[19] Hu, Z.; Dychka, I.; et al. The analysis and investigation of multiplicative inverse searching methods in the ring of integers modulo M. *International Journal of Intelligent Systems and Applications*, volume 8, no. 11, 2016: p. 9.

Appendix A

Counting the operations

In this chapter, all algorithms that were used in the experimental part are described and all the operation that were counted are labeled by colour. Only the operations in the main while loop are counted. In the table A, the operations that are held in regard are shown. Operations that were counted in the statistics are distinguished by colour.

Table A.1: Operations

name	**label**	**operation**	**comment**
addition	add	$a + b$	
subtraction	sub	$a - b$	
greater than / less than test	test	$a > b; a < b$	realized by subtraction
negation	neg	$a = -a$	needs addition (2's complement)
shift	shift	$a \ll 1; a \gg 1;$	left and right shift, usually multiplying or division by 2
zero comparison	zero	$a > 0, a == 0$	checking zero flag
evenness	even	$2\|a$	even or odd – checking LSB
while loop	loop	loop condition	number of loop iterations
counter incrementation	k	$k++$	counter incrementation

Algorithm 12: Penk

Input: $a \in [1, p-1]$ and p

Output: $r \in [1, p-1]$ and k, where $r = a^{-1} \pmod{p}$, and $n \leq k \leq 2n$

```
u := p, v := a, r := 0, s := 1
k = 0
while v > 0 do
    if u is even then
        if r is even then
            u := u/2, r := r/2, k := k + 1
        else
            u := u/2, r := (r + p)/2, k := k + 1
    else if v is even then
        if s is even then
            v := v/2, s := s/2, k := k + 1
        else
            v := v/2, s := (s + p)/2, k := k + 1
    else
        x := (u − v)
        if x > 0 then
            u := x, r := r − s
            if r < 0 then
                r := r + p
        else
            v := −x, s := s − r
            if s < 0 then
                s := s + p
if r > p then
    r := r − p
if r < 0 then
    r := r + p
return r, k
```

Algorithm 13: Montgomery – Phase I

Input: $a \in [1, p-1]$ and p

Output: $y \in [1, p-1]$ and k, where $y = a^{-1}2^k \pmod p$, and $n \leq k \leq 2n$

```
u := p, v := a, r := 0, s := 1
k = 0
while v > 0 do
    if u is even then
        u := u/2, s := 2s, k := k + 1
    else if v is even then
        v := v/2, r := 2r, k := k + 1
    else
        x := (u − v)
        if x > 0 then
            u := x/2, r := r + s, s := 2s, k = k + 1
        else
            v := −x/2, s := r + s, r := 2r, k = k + 1
if r > p then
    r := r − p
return y = p − r, k
```

Algorithm 14: Montgomery – Phase II

Input: $y \in [1, p-1]$, p and k from Phase I

Output: $y \in [1, p-1]$, where $r = a^{-1} \pmod p$, and $2k$ from Phase I

```
for i = 1 to k do
    if r is even then
        r := r/2
    else
        r := (r + p)/2
return r and 2k
```

Algorithm 15: Left-Shift Algorithm

Input: $a \in [1, p-1]$ and p

Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod{p}$, c_u, c_v and $0 < c\_v + c\_u \leq 2n$

$u := p, v := a, r := 0, s := 1$
$c\_u = 0, c\_v = 0$
while $(u \neq \pm 2^{c-u}$ & $v \neq \pm 2^{c-v})$ **do**
    if $(u_n, u_{n-1} = 0)$ *or* $(u_n, u_{n-1} = 1\&\,\mathrm{OR}\,(u_{n-2}, \ldots, u_0) = 1)$ **then**
        if $(c\_u \geq c\_v)$ **then**
            $u := 2u, r := 2r, c\_u := c\_u + 1$
        else
            $u := 2u, s := s/2, c\_u := c\_u + 1$
    else if $(v_n, v_{n-1} = 0)$ *or* $(v_n, v_{n-1} = 1\&\,\mathrm{OR}\,(v_{n-2}, \ldots, v_0) = 1)$ **then**
        if $(c\_v \geq c\_u)$ **then**
            $v := 2v, s := 2s, c\_v := c\_v + 1$
        else
            $v := 2v, r := r/2, c\_v := c\_v + 1$
    else
        if $(v_n = u_n)$ **then**
            oper = " − "
        else
            oper = " + "
        if $(c\_u \leq c\_v)$ **then**
            $u := u$ oper $v, r := r$ oper s
        else
            $v := v$ oper $u, s := s$ oper r
if $(v = \pm 2^{c-v})$ **then**
    $r := s, u_n := v_n$
if $(u_n = 1)$ **then**
    if $(r < 0)$ **then**
        $r := -r$
    else
        $r := p - r$
if $(r < 0)$ **then**
    $r := r + p$
return $r, c\_u,$ *and* c_v

Algorithm 16: AMI with subtractions

Input: $a \in [1, p-1]$ and p

Output: $o \in [1, p-1]$ and k, where $o = a^{-1}2^k \pmod p$, and $n - 1 \leq k \leq 2n$

$u := p, v := a, r := 0, s := 1$
$k = 0$
while 1 **do**
    if *u is even* **then**
        $u := u/2, s := 2s$
    else if *v is even* **then**
        $v := v/2, r := 2r$
    else
        $x := (u - v), y = r + s$
        if $x = 0$ **then**
            return $o = s, k$
        if $CARRY(x) = 1$ **then**
            $u := x/2, r := y, s := 2s$
        else
            $v := -x/2, s := y, r := 2r$
    $k = k + 1$

Algorithm 17: Subtraction-free AMI

Input: $a \in [1, p-1]$ and p

Output: $o \in [1, p-1]$ and k, where $o = a^{-1}2^k \pmod{p}$, and $n - 1 \leq k \leq 2n$

```
u := −p, v := a, r := 0, s := 1
k = 0
while 1 do
    if u is even then
        u := u/2, s := 2s
    else if v is even then
        v := v/2, r := 2r
    else
        x := (u + v), y = r + s
        if x = 0 then
            return o = s, k
        if CARRY(x) = 0 then
            u := x/2, r := y, s := 2s
        else
            v := x/2, s := y, r := 2r
    k = k + 1
```

Algorithm 18: Tao Wu with corrections – Phase I

Input: $a \in [1, p-1]$ and p

Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod{p}$, c_u, c_v and $0 < c\_v + c\_u \leq 2n$

$u := p, v := a, r := 0, s := 1$
$c\_u = 0, c\_v = 0$
while $(u \neq \pm 2^{c-u}$ & $v \neq \pm 2^{c-v})$ **do**
    if $(u_n, u_{n-1} = 0)$ *or* $(u_n, u_{n-1} = 1\&\,\mathrm{OR}\,(u_{n-2}, \ldots, u_0) = 1)$ **then**
        $u := 2u, c\_u := c\_u + 1$
        if s *is odd* **then**
            $\mathrm{s} := \mathrm{s} + \mathrm{p}$
        $s := s/2$
    else if $(v_n, v_{n-1} = 0)$ *or* $(v_n, v_{n-1} = 1\&\,\mathrm{OR}\,(v_{n-2}, \ldots, v_0) = 1)$ **then**
        $v := 2v, c\_v := c\_v + 1$
        if r *is odd* **then**
            $\mathrm{r} := \mathrm{r} + \mathrm{p}$
        $r := r/2$
    else
        if $(v_n = u_n)$ **then**
            $\mathrm{oper} = " - "$
        else
            $\mathrm{oper} = " + "$
        if $(c\_u \leq c\_v)$ **then**
            $u := u \;\mathrm{oper}\; v, r := r \;\mathrm{oper}\; s$
            if $r > p$ **then**
                $r := r - p;$
        else
            $v := v \;\mathrm{oper}\; u, s := s \;\mathrm{oper}\; r$
            if $s > p$ **then**
                $s := s - p;$
if $(v = \pm 2^{c-v})$ **then**
    $r := s, u_n := v_n, c\_v := c\_u$
if $(u_n = 1)$ **then**
    if $(r < 0)$ **then**
        $r := -r$
    else
        $r := p - r$
if $(r < 0)$ **then**
    $r := r + p$
return $r, c\_u,$ *and* c_v

Algorithm 19: Tao Wu with corrections – Phase II

Input: r, c_v, p from Phase I

Output: $y = r \cdot 2^{c-v} \pmod{p} = a^{-1} \pmod{p}$

for $i = 1$ *to* c_v **do**
    $r = 2r$
    if $r \geq p$ **then**
        $r := r - p$
    if $r < 0$ **then**
        $r := r + p$
return $y := r$

Algorithm 20: Optimized Montgomery algorithm for $2^n - c$

Input: $a \in [1, 2^n)$ and is odd, and $p > 2$, n-bit prime, precomputed $T = 2^{-2n} \pmod{p}$

Output: $r \in [1, 2^n)$, where $r = a^{-1} \pmod{p}$

\\ Phase I
$u := -p, v := a, r := 0, s := 1$
$k = 0$
while 1 **do**
    $x := (u + v)$
    $y := (r + s)$
    $tlz_x := DET(x)$
    if $x = 0$ **then**
        break;
    else if $x < 0$ **then**
        $u := x \gg tlz_x$
        $r := y$
        $s := s \ll tlz_x$
    else
        $v := x \gg tlz_x$
        $s := y$
        $:= r \ll tlz_x$
    $k := k + tlz_x$
\\ Phase II
$s = s \cdot 2^{2n-k} \pmod{p}$
$s = s \cdot T \pmod{p}$

Appendix B

Corrections of Tao Wu algorithm

B.1 Illustration of the correction I

This is an example of a run of algorithm 8 for input values $(a, p) = (4, 13)$. This example illustrates the problem with no check whether values r, s are divisible by two when they are shifted to the right. As a result, there is a loss of information.

In the table below, we see that the computation runs into a problem in 3rd iteration ($l = 3$), where s is divided by two although it is equal to 1. If we allow the right shift, we will end up with an incorrect step of calculation. The variable l denotes number of the current iteration of *while* loop. $RS(x)$ denotes the operation of right shift of a value x.

$$s^{(3)} = s^{(2)}/2 = RS((1)_{10}) = RS((00001)_2) = (00000)_2 = (0)_{10}$$

In the next iteration ($l = 4$), the error propagates and at the end of the loop we have

$$r = r^{(4)} + p = (-1)_{10} + (13)_{10} = (12)_{10}$$

instead of

$$r = r^{(4)} + p = (-8)_{10} + (13)_{10} = (5)_{10}.$$

Table B.1: Tao Wu's algorithm – correction I

l	operations	values of registers	tests
0		$u^{(0)} = (13)_{10} = (01101)_2$ $v^{(0)} = (4)_{10} = (00100)_2$ $r^{(0)} = (0)_{10} = (00000)_2$ $s^{(0)} = (1)_{10} = (00001)_2$	$u^{(0)} \neq \pm 2^0$ $v^{(0)} \neq \pm 2^0$
1	 $v^{(1)} = 2v^{(0)}$ $r^{(1)} = r^{(0)}/2$	$u^{(1)} = (13)_{10} = (01101)_2$ $v^{(1)} = (8)_{10} = (01000)_2$ $r^{(1)} = (0)_{10} = (00000)_2$ $s^{(1)} = (1)_{10} = (00001)_2$	$u^{(1)} \neq \pm 2^0$ $v^{(1)} \neq \pm 2^1$
2	$u^{(2)} = u^{(1)} - v^{(1)}$ $r^{(2)} = r^{(1)} - s^{(1)}$	$u^{(2)} = (5)_{10} = (00101)_2$ $v^{(2)} = (8)_{10} = (01000)_2$ $r^{(2)} = (-1)_{10} = (11111)_2$ $s^{(2)} = (1)_{10} = (00001)_2$	$u^{(2)} \neq \pm 2^0$ $v^{(2)} \neq \pm 2^1$
3	$u^{(3)} = 2u^{(2)}$ $s^{(3)} = s^{(2)}/2$	$u^{(3)} = (10)_{10} = (01010)_2$ $v^{(3)} = (8)_{10} = (01000)_2$ $r^{(3)} = (-1)_{10} = (11111)_2$ $s^{(3)} = (0)_{10} = (00000)_2$	$u^{(3)} \neq \pm 2^1$ $v^{(3)} \neq \pm 2^1$
4	$u^{(4)} = u^{(3)} - v^{(3)}$ $r^{(4)} = r^{(3)} - s^{(3)}$	$u^{(4)} = (2)_{10} = (00010)_2$ $v^{(4)} = (8)_{10} = (01000)_2$ $r^{(4)} = (-1)_{10} = (11111)_2$ $s^{(4)} = (0)_{10} = (00000)_2$	$u^{(4)} = \pm 2^1$
	$r = r^{(4)} + p$	$r = (12)_{10} = (01100)_2$	

In Phase II (algorithm 9), the computation goes as follows ($c_v = 1$):

$$y^{(0)} = r = 12$$

$$y = y^{(1)} = 2y^{(0)} - p = 24 - 13 = 11$$

but correctly it should be this computation:

$$y^{(0)} = r = 5$$

$$y = y^{(1)} = 2y^{(0)} = 10$$

We see, that 10 is the correct output: $10 * 4 = 1 \pmod{p}$.

B.2 Illustration of the correction II

Tables in this section illustrate the correct and incorrect run of the algorithm (8) – with and without the correction in the second branch. For example, this effects the computation of modular inverse for $(a, p) = (68, 347)$. The first table shows the run before adding the correction. The algorithm starts to work incorrectly at $l = 12$.

Table B.2: Tao Wu's algorithm – incorrect run

l	operations	values of registers	tests
0		$u^{(0)} = (347)_{10} = (0101011011.)_2$	$u^{(0)} \neq \pm 2^0$
		$v^{(0)} = (68)_{10} = (0001000100.)_2$	$v^{(0)} \neq \pm 2^0$
		$r^{(0)} = (0)_{10} = (0000000000.)_2$	
		$s^{(0)} = (1)_{10} = (0000000001.)_2$	
1		$u^{(1)} = (347)_{10} = (0101011011.)_2$	$u^{(1)} \neq \pm 2^0$
	$v^{(1)} = 4v^{(0)}$	$v^{(1)} = (272)_{10} = (01000100.00)_2$	$v^{(1)} \neq \pm 2^2$
	$r^{(1)} = r^{(0)}/4$	$r^{(1)} = (0)_{10} = (0000000000.)_2$	
		$s^{(1)} = (1)_{10} = (0000000001.)_2$	
2	$u^{(2)} = u^{(1)} - v^{(1)}$	$u^{(2)} = (75)_{10} = (0001001011.)_2$	$u^{(2)} \neq \pm 2^2$
		$v^{(2)} = (272)_{10} = (01000100.00)_2$	$v^{(2)} \neq \pm 2^2$
	$r^{(2)} = r^{(1)} - s^{(1)}$	$r^{(2)} = (-1)_{10} = (1111111111.)_2$	
		$s^{(2)} = (1)_{10} = (0000000001.)_2$	
3	$u^{(3)} = 4u^{(2)}$	$u^{(3)} = (300)_{10} = (01001011.00)_2$	$u^{(3)} \neq \pm 2^2$
		$v^{(3)} = (272)_{10} = (01000100.00)_2$	$v^{(3)} \neq \pm 2^2$
		$r^{(3)} = (-1)_{10} = (1111111111.)_2$	
	$s^{(3)} = s^{(2)}/4$	$s^{(3)} = (87)_{10} = (0001010111.)_2$	
4	$u^{(4)} = u^{(3)} - v^{(3)}$	$u^{(4)} = (28)_{10} = (0000011100.)_2$	$u^{(4)} \neq \pm 2^2$
		$v^{(4)} = (272)_{10} = (01000100.00)_2$	$v^{(4)} \neq \pm 2^2$
	$r^{(4)} = r^{(3)} - s^{(3)}$	$r^{(4)} = (-88)_{10} = (1110101000.)_2$	
		$s^{(4)} = (87)_{10} = (0001010111.)_2$	
5	$u^{(5)} = 16u^{(4)}$	$u^{(5)} = (448)_{10} = (011100.0000)_2$	$u^{(5)} \neq \pm 2^2$
		$v^{(5)} = (272)_{10} = (01000100.00)_2$	$v^{(5)} \neq \pm 2^2$
		$r^{(5)} = (-88)_{10} = (1110101000.)_2$	
	$s^{(5)} = s^{(4)}/16$	$s^{(5)} = (244)_{10} = (0011110100.)_2$	

Continued on next page

Table B.2 – continued from previous page

6	$v^{(6)} = v^{(5)} - u^{(5)}$ $s^{(6)} = s^{(5)} - r^{(5)}$	$u^{(6)} = (448)_{10} = (0111000000.)_2$ $v^{(6)} = (-176)_{10} = (1101010000.)_2$ $r^{(6)} = (-88)_{10} = (1110101000.)_2$ $s^{(6)} = (332)_{10} = (0101001100.)_2$	$u^{(6)} \neq \pm 2^6$ $v^{(6)} \neq \pm 2^2$
7	$v^{(7)} = 2v^{(6)}$ $r^{(7)} = r^{(6)}/2$	$u^{(7)} = (448)_{10} = (0101011011.)_2$ $v^{(7)} = (-352)_{10} = (101010000.0)_2$ $r^{(7)} = (-44)_{10} = (1111010100)_2$ $s^{(7)} = (332)_{10} = (0101001100.)_2$	$u^{(7)} \neq \pm 2^6$ $v^{(7)} \neq \pm 2^3$
8	$v^{(8)} = v^{(7)} + u^{(7)}$ $s^{(8)} = s^{(7)} + r^{(7)}$	$u^{(8)} = (448)_{10} = (0101011011.)_2$ $v^{(8)} = (96)_{10} = (0001100000.)_2$ $r^{(8)} = (-44)_{10} = (1111010100)_2$ $s^{(8)} = (288)_{10} = (0101001100.)_2$	$u^{(6)} \neq \pm 2^6$ $v^{(8)} \neq \pm 2^3$
9	$v^{(9)} = 4v^{(8)}$ $r^{(9)} = r^{(8)}/4$	$u^{(9)} = (448)_{10} = (0101011011.)_2$ $v^{(9)} = (384)_{10} = (01100000.00)_2$ $r^{(9)} = (-11)_{10} = (1111010100)_2$ $s^{(9)} = (288)_{10} = (0101001100.)_2$	$u^{(9)} \neq \pm 2^6$ $v^{(9)} \neq \pm 2^5$
10	$v^{(10)} = v^{(9)} - u^{(9)}$ $s^{(10)} = s^{(9)} - r^{(9)}$	$u^{(10)} = (448)_{10} = (0111000000.)_2$ $v^{(10)} = (-64)_{10} = (1111000000.)_2$ $r^{(10)} = (-11)_{10} = (1111010100)_2$ $s^{(10)} = (299)_{10} = (0100101011)_2$	$u^{(10)} \neq \pm 2^6$ $v^{(6)} \neq \pm 2^5$
11	$v^{(11)} = 4v^{10)}$ $r^{(11)} = r^{(10)}/4$	$u^{(11)} = (448)_{10} = (0101011011.)_2$ $v^{(11)} = (-256)_{10} = (11000000.00)_2$ $r^{(11)} = (84)_{10} = (0001010100)_2$ $s^{(11)} = (299)_{10} = (0100101011)_2$	$u^{(11)} \neq \pm 2^6$ $v^{(11)} \neq \pm 2^7$
12	$u^{(12)} = u^{(11)} + v^{(11)}$ $r^{(12)} = r^{(11)} + s^{(11)}$	$u^{(12)} = (192)_{10} = (0011000000)_2$ $v^{(12)} = (-256)_{10} = (11000000.00)_2$ $r^{(12)} = (383)_{10} = (0101111111)_2$ $s^{(4)} = (299)_{10} = (0100101011)_2$	$u^{(12)} \neq \pm 2^6$ $v^{(12)} \neq \pm 2^7$
13	$u^{(13)} = 2u^{(12)}$ $s^{(13)} = s^{(12)}/2$	$u^{(13)} = (384)_{10} = (011000000.0)_2$ $v^{(13)} = (-256)_{10} = (11000000.00)_2$ $r^{(13)} = (383)_{10} = (0101111111)_2$ $s^{(13)} = (323)_{10} = (0101000011)_2$	$u^{(13)} \neq \pm 2^7$ $v^{(13)} \neq \pm 2^7$

Continued on next page

Table B.2 – continued from previous page

14	$u^{(14)} = u^{(13)} + v^{(13)}$	$u^{(14)} = (128)_{10} = (0010000000.)_2$	$u^{(14)} = \pm 2^7$
		$v^{(14)} = (-256)_{10} = (11000000.00)_2$	$v^{(14)} \neq \pm 2^7$
	$r^{(14)} = r^{(13)} + s^{(13)}$	$r^{(12)} = (706)_{10} = (0011000010)_2$	
		$s^{(14)} = (323)_{10} = (0101000011)_2$	

Table B.3: Tao Wu's algorithm – correct run

l	operations	values of registers	tests
12c	$u^{(12)} = u^{(11)} + v^{(11)}$	$u^{(12)} = (192)_{10} = (0011000000)_2$	$u^{(12)} \neq \pm 2^6$
		$v^{(12)} = (-256)_{10} = (11000000.00)_2$	$v^{(12)} \neq \pm 2^7$
	$r^{(12)} \pmod p$	$r^{(12)} = (36)_{10} = (0000100100)_2$	
		$s^{(4)} = (299)_{10} = (0100101011)_2$	
13c	$u^{(13)} = 2u^{(12)}$	$u^{(13)} = (384)_{10} = (011000000.0)_2$	$u^{(13)} \neq \pm 2^7$
		$v^{(13)} = (-256)_{10} = (11000000.00)_2$	$v^{(13)} \neq \pm 2^7$
		$r^{(9)} = (36)_{10} = (0000100100)_2$	
	$s^{(13)} = s^{(12)}/2$	$s^{(13)} = (323)_{10} = (0101000011)_2$	
14c	$u^{(14)} = u^{(13)} + v^{(13)}$	$u^{(14)} = (128)_{10} = (0010000000.)_2$	$u^{(14)} = \pm 2^7$
		$v^{(14)} = (-256)_{10} = (11000000.00)_2$	$v^{(14)} \neq \pm 2^7$
	$r^{(14)} = r^{(13)} + s^{(13)}$	$r^{(12)} = (12)_{10} = (0000001100)_2$	
		$s^{(14)} = (323)_{10} = (0101000011)_2$	

Here, the table illustrates the run of the second phase and the difference between output values.

Table B.4: Tao Wu's algorithm – Phase II (incorrect)

l	operations	values of registers	tests
II	$r^{(0)}$	$r^{(0)} = (706)_{10}$	
	$r^{(1)} = 2r^{(0)} - p$	$r^{(1)} = (1065)_{10}$	
	$r^{(2)} = 2r^{(1)} - p$	$r^{(2)} = (1783)_{10}$	
	$r^{(3)} = 2r^{(2)} - p$	$r^{(3)} = (3219)_{10}$	
	$r^{(4)} = 2r^{(3)} - p$	$r^{(4)} = (6091)_{10}$	
	$r^{(5)} = 2r^{(4)} - p$	$r^{(5)} = (11835)_{10}$	
	$r^{(6)} = 2r^{(5)} - p$	$r^{(6)} = (23323)_{10}$	
	$r^{(7)} = 2r^{(6)} - p$	$r^{(7)} = (-18890)_{10}$	

Table B.5: Tao Wu's algorithm – Phase II (correct)

l	operations	values of registers	tests
II	$r^{(0)}$	$r^{(0)} = (12)_{10}$	
	$r^{(1)} = 16r^{(0)}$	$r^{(1)} = (192)_{10}$	
	$r^{(2)} = 2r^{(1)} - p$	$r^{(2)} = (37)_{10}$	
	$r^{(3)} = 4r^{(2)} - p$	$r^{(3)} = (148)_{10}$	

Appendix **C**

Acronyms

LSB least significant bit

MSB most significant bit

gcd greatest common divisor

MMI Montgomery modular inverse

AMI Almost Montgomery inverse

```
readme.txt ........ the file with CD contents description
src ............... the directory of source codes
text .............. the thesis text directory
  thesis.pdf ...... the thesis text in PDF format
```