MODULAR EXPONENTIATION METHODS IN CRYPTOGRAPHY

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HASAN BARTU YÜNÜAK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

SEPTEMBER 2017

Approval of the thesis:

MODULAR EXPONENTIATION METHODS IN CRYPTOGRAPHY

submitted by **HASAN BARTU YÜNÜAK** in partial fulfillment of the requirements for the degree of **Master of Science in Department of Cryptography, Middle East Technical University** by,

Prof. Dr. Bülent Karasözen
Director, Graduate School of **Applied Mathematics** __________

Prof. Dr. Ferruh Özbudak
Head of Department, **Cryptography** __________

Assoc. Prof. Dr. Murat Cenk
Supervisor, **Cryptography, METU** __________

Examining Committee Members:

Prof. Dr. Ferruh Özbudak
Mathematics, METU __________

Assoc. Prof. Dr. Murat Cenk
Cryptography, METU __________

Asst. Prof. Dr. Oğuz Yayla
Mathematics, Hacettpe University __________

Date: __________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: HASAN BARTU YÜNÜAK

Signature :

ABSTRACT

MODULAR EXPONENTIATION METHODS IN CRYPTOGRAPHY

Yünüak, Hasan Bartu
M.S., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Murat Cenk

September 2017, 51 pages

Modular exponentiation has an important role in many cryptographic algorithms. These exponentiation methods differ in the bases used and their representations, the repeating aspect, and for which algorithms they are used for: fixed or variable base. Our research aims to compare the efficiencies and implementation timings for some selected algorithms. Also, we look at the options for using a dedicated cubing algorithm, and compare them with the current algorithms.

Keywords : Modular exponentiation, big integer implementations, cubing, fixed exponent, variable exponent

ÖZ

KRİPTOGRAFİDE MODÜLER ÜS ALMA YÖNTEMLERİ

Yünüak, Hasan Bartu
Yüksek Lisans, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Murat Cenk

Eylül 2017, 51 sayfa

Modüler üs alma, kriptografik algoritmalar için büyük önem taşır. Üs alma metotlarının kullanılan taban ve gösterim şekilleri, tekrarlı üst alımlarda kullanılan yöntemler, sabit ve değişken üs gerektiren algoritmalarda kullanılmaları gibi farkları vardır. Araştırmamız bazı seçtiğimiz üs alma algoritmalarının gerçekleştirme zamanlamalarını ve verimliliklerini karşılaştırmayı amaçlar. Ayrıca, özelleşmiş küp alma yöntemlerinin kullanılabilirliğini ve seçeneklerini araştırıp şu anda kullanılan algoritmalarla karşılaştırmaları yapılmıştır.

Anahtar Kelimeler : Modüler üs alma, büyük sayı gerçekleştirmeleri, küp alma, sabit üs, değişken üs

To My Family

ACKNOWLEDGMENTS

I would like to thank my supervisor Assoc. Prof. Dr. Murat Cenk. His guidance and support had a huge part in this thesis. It was a great pleasure to work with him.

Many thanks goes to Murat B. İlter and H. Ali Şahin for their inputs on this thesis. If not for their help, this thesis would have taken much longer.

I would like to thank my family for their support about this thesis, and everything else.

I am deeply grateful to my wife Melis Demir Yünüak. This thesis would not be possible without her. She listened to my every concern and helped me immensely every single day.

This work is partially supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under the grant no 115R289.

TABLE OF CONTENTS

ABSTRACT vii

ÖZ ix

ACKNOWLEDGMENTS xiii

TABLE OF CONTENTS xv

LIST OF FIGURES xix

LIST OF TABLES xxi

LIST OF ABBREVIATIONS xxiii

CHAPTERS

1 INTRODUCTION 1

2 PRELIMINARIES 3

2.1 RSA Cryptosystem 3

2.1.1 RSA Scheme 3

2.2 Elliptic Curve Cryptography 4

2.2.1 Elliptic Curve ElGamal Cryptosystem 6

2.3 Algorithm Complexity and The Big-$\mathcal{O}$ Notation 6

2.4 Modular Reduction 7

2.4.1 Barret Reduction 7

2.4.2 Montogmery Reduction 8

3 MULTIPLICATION ALGORITHMS . . . 11

3.1 Schoolbook Algorithm . . . 11

3.2 Karatsuba Algorithm . . . 13

3.3 Toom-Cook Algorithms . . . 15

3.3.1 Toom-2 . . . 15

3.3.2 Toom-3 . . . 17

3.3.3 Unbalanced Toom-3 . . . 17

4 POWERS 2 AND 3 . . . 21

4.1 Squaring Using Multiplication Methods . . . 21

4.1.1 Schoolbook Squaring . . . 21

4.1.2 Karatsuba Squaring . . . 22

4.2 Asymmetric Squaring . . . 22

4.3 Classical Cubing Method . . . 24

4.4 Zanoni's Cubing Algorithm . . . 24

4.4.1 Modular Approach . . . 26

5 REPRESENTATIONS OF EXPONENTS AND EXPONENTIATION ALGORITHMS . . . 29

5.1 Schoolbook Exponentiation . . . 29

5.2 Base-w Representation . . . 29

5.2.1 Algorithm . . . 30

5.2.2 Binary Representation and Repeated Squaring Method 31

5.2.3 Ternary Representation and Repeated Cubing Method 32

5.2.4 Window Methods . . . 33

5.2.5 Constant Length Nonzero Windows 34
5.2.6 Variable Length Nonzero Windows 35
5.3 Non Adjacent Form 36
5.4 Hybrid Binary-Ternary Number System 37
5.4.1 Algorithm 38
5.4.2 Exponentiation Using HBTNS 39
5.5 Addition Chains 40
5.6 Double-Base Representation 42
5.6.1 Tree Method 43
5.6.2 DAG Method 44
5.7 Implementation Results and Comparisons 45
6 CONCLUSION 47
REFERENCES 49

LIST OF FIGURES

Figure 3.1 The cost of schoolbook algorithm 12
Figure 3.2 The cost of Karatsuba algorithm 14

Figure 5.1 Tree graph for g^{19} 43
Figure 5.2 DAG for g^{19} 45

LIST OF TABLES

Table 2.1 Complexity Scales 7

Table 4.1 Time comparisons of two cubing algorithms 25

Table 4.2 Time comparisons of *gmp* library's reduction function for different bit-sizes 26

Table 4.3 Time comparisons of two modular cubing algorithms 27

Table 5.1 Time comparisons for repeated squaring, cubing and hbtns exponentiation 46

Table 5.2 Time comparisons for repeated cubing and hbtns exponentiation using modular Zanoni's cubing method. 46

LIST OF ABBREVIATIONS

ABBRV	Abbreviation
$\mathbb{R}$	Real Numbers
$\mathbb{Z}$	Integers
$k\mathbf{B}$	k-many bit shifts

CHAPTER 1

INTRODUCTION

Cryptography has an important role in the world. Especially after both world wars, the importance of information security became undenaibly apparent. Today, with many devices connecting to internet, online shopping, banking, communication, and other transmissions over unsecure channels, cryptography only grows more essential. In ancient times, obscuring the information is done via basic systems such as Caesar Cipher [28]. Later during the world wars, mechanical cryptosystems one of which is Enigma [36] is used. With the advances in computation power in the recent years, most cryptographic algorithms use computers and related technologies for secure transmission of information.

Today, instead of simple systems, two major methods are used for cryptography: Symmetric and Public Key Cryptography. Symmetric Key systems use the same key for encryption and decryption, or a key that is easily obtainable from the other. These systems are usually very fast, like AES [11] or now depreciated [3] DES [30] are two examples. The problem with these kind of systems is that the key exchange between the two parties has to be done over a secure channel, and it is hard to do so [30]. On the other hand, public key systems use a different key for encrypition and decryption, named public and private keys. These keys cannot be obtained from one another easily. Public keys are, following their name, are public. This solves the key exchange problem. However, these systems tend to be slower than Symmetric Key systems. RSA and ElGamal cryptosystems are two examples of Public Key cryptosystems.

The low speed of Public Key systems is because of the costly operations in computing the keys, encryption, decryption and other levels of the cryptosystem. For example, RSA uses modular exponentiation [11] with very big numbers. Modular exponentiation is already a slow operation and with the increasing speed of computation we use even larger numbers to be secure against cryptoanalysis [35]. Hence for RSA and other cryptosystems that use similar operations, modular exponentiation has to be studied and improved.

In this thesis, we study modular exponentiation for cryptography. We look at the the exponentiation methods and their use in variable or fixed exponents. Our research is on general algorithms but we also specifically investigate cubing algorithms. In Chapter 2, we introduce some public key cryptography algorithms, cost computation and reduction algorithms as preliminary information. In Chapter 3, we study some

multiplication algorithms that are generally used for exponentiation. In Chapter 4, we investigate both the squaring and cubing methods. Our main endeavour in this section is analyzing Zanoni's cubing algorithm [6] for integers as a modular cubing algorithm. In Chapter 5 we research the repeated use of all these methods for large integer exponentiation. Many algorithms are studied, and their costs compared. We also implemented these algorithms. Implementation results for the ones that showed promise with cubing in theory for variable exponents are given at the end. For fixed exponents, we only give the recent results and their complexities. In Chapter 6, we give the conclusion of the thesis and a note about future work.

CHAPTER 2

PRELIMINARIES

In this chapter we present some basic information about public key cryptosystems. As mentioned before, exponentiation is a main operation on these systems. Then we continue with an introduction to elliptic curve cryptography. The operations used on elliptic curves are closely related to exponentiation. After introducing elliptic curves, we give the definition of big-$\mathcal{O}$ notation and its use in complexity computations. We use this notation throughout the thesis. At the end we give two useful reduction algorithms since modular reduction is of interest to us, the exponentiations used in public key cryptography are usually modular.

2.1 RSA Cryptosystem

RSA is a public key cryptosystem invented by R. L. Rivest, A. Shamir, L. Adleman in 1977 [33]. It is used in many public practical applications such as e-commerce, message authentciation, secure e-mail, key exchange, etc. Its security depends on one of the *hard* problems in mathematics, the integer factorization search problem.

Definition 2.1. *Integer factorization search problem (IFSP):* Given a composite integer n, find a p such that $p \mid n$, and $1 < p < n$.

As the size of n increases, this problem becomes harder and p becomes infeasable to be computed. The RSA Cryptosystem is based on the hardness of this problem. The main operation of RSA is modular exponentiation. In encryption, decryption and digital signing, modular exponentiation is used. In the next sections we give the schema of RSA, assuming B is trying to send a message to A.

2.1.1 RSA Scheme

First, A chooses two big primes p_A and q_A not equal to each other. There are some constraints on the choice of these primes to defend against attacks such as *Pollard's p-1 attack* [31] and *William's p+1 attack* [38]. Primes p_A and q_A must be kept private, since the system's security depends wholly on these two primes. Currently, choosing

these primes with bit sizes at least 1024-bits is thought to be *secure* [23] however, as the computer science progresses, the size will increase as well.

After the primes are chosen, part of A's public key, $n_A = p_A \cdot q_A$, is computed by A. Notice that IFSP relates to this point, since it is hard to acquire p_A and q_A from knowing only n_A. After this, A finds the *Euler phi* [17] of n_A: $\phi_A(n_A) = (p_A - 1) \cdot (q_A - 1)$. A randomly selects an integer e_A satisfying $gcd(\phi(n_A), e_A) = 1$ where $1 < e_A < \phi(n_A)$. Finally, A finds the inverse of e_A for the private key, $e_A \cdot d_A \equiv 1 \bmod \phi(n_A)$. The public key of A is (n_A, e_A) and the private key is d_A. Note that without knowing p_A, q_A, we cannot get $\phi(n_A)$ and hence cannot find d_A from e_A.

In the encryption step, B encodes the message he wants to send into an integer. If the integer is larger than $n-1$, then the message must be separated into blocks. Then, each block is encrypted separately using modes of operation [30] or with protocols such as PKCS#1 [22]. Nevertheless, each block is encrypted similarly, so we only give the schema for one block of message. Assume the integer encoding of the message is P. After encoding, B gets the public key of A, (n_A, e_A). Using this key, B computes

$$C = P^{e_A} \bmod n_A \tag{2.1}$$

to get the ciphertext C. Sending C to A, the encryption exchange is done. Acquiring C, A does the following to decrypt the ciphertext

$$P = C^{d_A} \bmod n_A, \tag{2.2}$$

and finds the plaintext message P.

The RSA system can be used to sign messages as well. Digital signatures allow the signer to prove that she was the one send the message and prevent fake signatures. When signing a message, we don't assume the plaintext is hidden, because encryption and decryption processess can be included almost trivially. Assuming the public and private keys are chosen as above, A signs a message P with

$$S = P^{d_A} \bmod n_A. \tag{2.3}$$

After getting the signed message along with the public key of A, B verifies the correctness if

$$P = S^{e_A} \bmod n_A. \tag{2.4}$$

The signature can be seen as an inversed form of encryption. Since only A knows the private key of A, only he is able to sign the message with d_A.

As it can be seen from the Equations 2.1 to 2.4, every major operation in RSA is modular exponentiation therefore improving the speed of this operation is very important.

2.2 Elliptic Curve Cryptography

Elliptic curve cryptography consists of public key systems that use elliptic curve structures. In these structures we do not have exponentiation in the traditional integer sense,

however, multiple point additions (called scalar multiplication or point multiplication) is analogous to integer exponentiation. Before we continue with those, we give a brief introduction on elliptic curves:

Definition 2.2. An *elliptic curve* E is a curve defined over the finite field $\mathbb{F}_p$ where p is a prime number. Given $a, b \in \mathbb{F}_p$ where $4a^3 + 27b^2 \not\equiv 0$ (mod p), E is defined by an equation with the form

$$y^2 = x^3 + ax + b. \tag{2.5}$$

If $x, y \in \mathbb{F}_p$ satisfies the Equation 2.5, then (x, y) is on the curve. The point at infinity is denoted by ∞, and used as the identity element. We denote the set of points of E by $E(\mathbb{F}_p)$. This definition assumes $char(\mathbb{F}_p) \neq 2, 3$. For simplicity and the sake of introductory examples, we only give these type of curves. Next, we give the group law for these kind of curves.

Let $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ be points on the curve $E(\mathbb{F}_p)$ as it is defined in Definition 2.2. Assume $P_1 \neq \pm P_2$.

- The identity element is denoted by ∞, $P_1 + \infty = \infty + P_1 = P_1$ for all points in $E(\mathbb{F}_p)$.
- The negative of P_1 is $-P_1$ and is a point on $E(F_p)$ defined as $(x, -y)$. $P_1 - P_1 = \infty$ and $\infty = -\infty$.
- Adding two points in an elliptic curve is referred to as point addition. $P_1 + P_2 = P_3$ where $P_3 = (x_3, y_3)$ is found with the following operations:

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1.$$

- The operation $P_1 + P_1 = 2P_1$ is called point doubling. Assume $P_1 \neq -P_1$. We find $2P_1 = (x_3, y_3)$ by

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \quad y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1.$$

As it can be seen addition and doubling is very costly, using a myriad of multiplications if used consecutively. However, negation is almost free. The relation between addition and doubling can be seen as the relation between multiplication and squaring in the integer case, which we investigate in the later chapters. Following the group law, we give the elliptic curve variation of the discrete logarithm problem, which deals with elliptic curve scalar multiplication.

Definition 2.3. Let E be an elliptic curve defined over a finite field F. Assume Q and P are two points on $E(K)$ with $nP = Q$. The *elliptic curve discrete logarithm problem (ECDLP)* is to find n given P and Q.

2.2.1 Elliptic Curve ElGamal Cryptosystem

The elliptic curve version of the ElGamal Cryptosystem[26] is similar to the integer version [18], and is a good example for observing the similarities between elliptic curve scalar multiplication and integer exponentiation.

For key generation, we publicly choose an elliptic curve E defined over the finite field $\mathbb{F}_p$. We take a point P on $E(F_p)$ with prime order n. Then we have a cyclic subgroup with generator P, contained in $E(\mathbb{F}_p)$. We call E, P, n the *domain parameters*. Private key is generated by choosing a random d where $1 \leq d \leq n-1$. The public key Q is found by

$$dP = Q.$$

Finding d from Q is exactly the elliptic curve discrete logarithm problem.

Assume A is trying to send a message to B. For encryption, A takes a plaintext m and encodes it into a point M. This can be done by Koblitz's method [2]. Then A randomly chooses an integer k from $1 \leq k \leq n-1$, which he computes kQ with, where Q is B's public key. The ciphertext is a pair of points (C_0, C_1) in $E(F_p)$ and is computed by

$$C_0 = kP,$$
$$C_1 = M + kQ.$$

After getting (C_0, C_1), B uses the private key d and finds M.

$$kQ = dC_0,$$
$$M = C_1 - kQ.$$

Only B can find kQ, through $dC_0 = d(kP) = k(dP) = kQ$. It can be seen that finding kQ from Q and C_0 is a hard problem and similar to integer Diffie-Hellman problem [19].

Notice that we use point addition kP on an integer k with an elliptic curve point P on key generation, encryption and decryption. Since this operation is very costly, the study of scalar multiplication is very important. This is where the exponentiation algorithms come into place. We can think of kP as $P + P + \cdots + P$, which is similar to the form for integers, $g^k = g \cdot g \cdots g$. The representation and the order we do the multiplications can reduce the cost of computing kP. One important advantage we have here is computing $-Q$ from Q is very fast depending on the chosen curve. This is similar to g^{-1} from the integer case. This allows us to use the NAF and the double base representation methods from Chapter 5 without considering the cost of $-Q$ (analogous to an inversion in integer case, a costly operation).

2.3 Algorithm Complexity and The Big-$\mathcal{O}$ Notation

Time complexity is a way of comparing different algorithms. Generally what happens is we count the operations, find the cost of the smallest operations and apply the costs

to the whole operation. When dealing with algorithms, especially if speed is of concern we use a tool called the *big-$\mathcal{O}$ notation*. What $\mathcal{O}$ does is that using the notation, we only take the costs of the most costly operations, and faster operations are *absorbed* in those. This is because we are thinking asymptotically when we compare the algorithms, and at larges sizes the costly operations are bound to have a larger impact on the overall cost. This leads to an upper bound on the cost. Formally,

$$f(n) \in \mathcal{O}(g(n)) \text{ if } 0 \leq f(n) \leq c \cdot g(n) \text{ for } n_0, c \in \mathbb{Z}^+, \forall n \geq n_0.$$

Table 2.1: Complexity Scales

$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(2^n)$
Constant	Logarithmic	Linear	Quasilinear	Quadratic	Exponential

In Table 2.1, the leftmost side is the fastest, and as we go to the right the complexity is *worse*.

2.4 Modular Reduction

As we have said before, in many cryptographic systems, especially systems like RSA modular operations are used. In modular exponentiation, for some algorithms there is at least one reduction at each step. Considering the basic reduction is just a division, and division is a costly operation [4], many reduction methods are designed. Among them, we give two important ones, Barret and Montgomery reduction methods.

2.4.1 Barret Reduction

Barret reduction [9] uses a reciprocal approximation that is dependent on the modulus but not the integers to be reduced. Let a be a $2n$-digit positive integer that we aim to reduce. Let N be an n-digit integer, chosen as the modulus. Choose an integer base $b = 2^l$ for a suitable l that does not depend on a. Before starting with the algorithm, we first define a useful notation.

Definition 2.4. The *reciprocal* of N is defined as $\mu = \lfloor \frac{b^{2n}}{N} \rfloor$.

Since the result of the reduction of $a \pmod N$ is the remainder r from a/N, we can see that the quotient q is $\lfloor \frac{a}{N} \rfloor$ from the same operation. We use the reciprocal of N here to approximate q:

$$q = \left\lfloor \frac{a}{N} \right\rfloor = \left\lfloor \frac{\frac{a}{b^{n-1}} \frac{b^{2n}}{N}}{b^{n+1}} \right\rfloor,$$

which we approximate to $\hat{q}$ by

$$\hat{q} = \left\lfloor \frac{\lfloor \frac{a}{b^{n-1}} \rfloor \mu}{b^{n+1}} \right\rfloor \leq \left\lfloor \frac{a}{N} \right\rfloor = q.$$

In addition to this, we note that $q - 2 \leq \hat{q} \leq q$ [9]. Using $\hat{q}$, we can find $\hat{a} = a - \hat{q}N$ which is the remainder r if $\hat{q} = q$. Notice that $\hat{a} = a \pmod N$, therefore if we don't get r right away in the algorithm below, we need at most 2 subtractions by N since $q - 2 \leq \hat{q}$. Below is the full algorithm.

Algorithm 1 Barret Reduction

Input: $N, b, n, a, \mu \in \mathbb{Z}; n = \lfloor \log_b N \rfloor + 1, \mu = \lfloor \frac{b^{2n}}{N} \rfloor, 0 \leq a < b^{2n}$
Output: $r = a \pmod N$

1: $\hat{q} \leftarrow \lfloor \lfloor \frac{a}{b^{n-1}} \rfloor \frac{\mu}{b^{n+1}} \rfloor$
2: $r \leftarrow (a \pmod{b^{n+1}}) - (\hat{q}N \pmod{b^{n+1}})$
3: **if** $r < 0$ **then**
4: $\quad r \leftarrow r + b^{n+1}$
5: **end if**
6: **while** $r \geq N$ **do**
7: $\quad r \leftarrow r - N$
8: **end while**
9: **return** r

The important point about this algorithm is that μ does not depend on the integer to be reduced. Therefore, if we have many reductions necessary in the computation, the cost of computing μ only appears once. The other divisions in step 2 are nothing but shifts in base-b. Further optimizations are possible [21]. If we do the operations other than the division by b^{n+1} in step 1, we don't use the rightmost $n + 1$ digits, hence they are unnecessary to compute. Furthermore, since division by a power of b is a shift, we only need the rightmost $n + 1$ digits of $\hat{q}N$ in step 2. This can also be optimized because N is already smaller than b^n.

2.4.2 Montogmery Reduction

Montgomery [9] discovered a way of representing elements from $\mathbb{Z}_N$. This reduction uses that representation with the following definition.

Definition 2.5. Let $N, R \in \mathbb{Z}$ such that $\gcd(N, R) = 1$. For $0 \leq a \leq N - 1$, the *Montgomery representation* is $[a] = (aR) \pmod N$.

With this representation, we define the reduction by $\text{REDC}(a) = (aR^{-1}) \pmod N$. Below we give the algorithm where r_i is the ith digit of r.

Algorithm 2 Montgomery Reduction

Input: $n, a, b, R, N, N' \in \mathbb{Z};\ n = \lfloor \log_b N \rfloor + 1,\ \gcd(N, b) = 1, R = b^n,\ N' = (-N^{-1}) \pmod{b},\ 0 \leq a < b^{2n}$

Output: $r = a \pmod{N}$

1: $r \leftarrow a$
2: **for** i **from** 0 **to** $n-1$ **do**
3: $\quad k \leftarrow r_i N' \pmod{b}$
4: $\quad r \leftarrow r + kNb^i$
5: **end for**
6: $r \leftarrow \frac{r}{R}$
7: **if** $r \geq N$ **then**
8: $\quad r \leftarrow r - N$
9: **end if**
10: **return** r

If we have R and N' as they are in Algorithm 2, and choose $k \equiv aN' \pmod{R}$ such that $0 \leq k \leq N-1$, then we see that $R \mid (a+kN)$. If we select an integer $r = (a+kN)/R$ which is $r = aR^{-1} \pmod{N}$ since $\gcd(R, N) = 1$, we have $0 \leq r < 2N$ [9]. Then at most one subtraction by N gives the result. The algorithm uses these techniques digit by digit in the *for* loop in steps 2 to 5. The division by R, which is a digit shift is in step 6 and step 8 makes sure the result is the smallest positive r that is congruent to a modulo N.

CHAPTER 3

MULTIPLICATION ALGORITHMS

Multiplication is undoubtedly a very important operation in cryptography. Integer and finite field multiplications are used in various cryptosystems [9]. Since we are studying exponentiation, we have to explore what the exponentiation algorithms are based on - multiplication. This chapter aims to explain some of the general use multiplication algorithms. The algorithms we explain here will be useful in the later chapters.

Large integers can be thought of as polynomials. We can represent an integer a with the polynomial f where

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0. \tag{3.1}$$

In Equation 3.1, $a_i < w$ for $0 \leq i < n$ for some chosen positive integer w called the word size. An integer a with a polynomial representation $f(x)$ with degree $n-1$ is said to be an n-word integer. We tend to use this form for arithmetic operations since it allows us to make use of the polynomial structure and recursion.

In all sections below, $\mathbf{M}(n)$ will be used to denote the cost of multiplying two size-n integers, while $\mathbf{S}(n)$ and $\mathbf{C}(n)$ is used for costs of squaring and cubing of a size-n integer.

3.1 Schoolbook Algorithm

The schoolbook method is a basic multiplication method. Using the polynomial representation, it uses straightforward computation. The algorithm is 2-way, since the polynomials are separated into two parts to find the result recursively. This method of separating and multiplying coefficients is called *divide and conquer*.

Assume we have two positive integers a and b, with polynomial representations:

$$\begin{aligned} a(x) &= a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0, \\ b(x) &= b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_1 x + b_0. \end{aligned} \tag{3.2}$$

Instead of multiplying the polynomials this way, we separate them into two parts. Both parts have degree $\frac{n}{2} - 1$.

$$\begin{aligned} a(x) &= A_1 x^{\frac{n}{2}} + A_0 \text{ where} \\ A_1 &= a_{n-1} x^{\frac{n}{2}-1} + \cdots + a_{\frac{n}{2}+1} x + a_{\frac{n}{2}} \\ A_0 &= a_{\frac{n}{2}-1} x^{\frac{n}{2}-1} + \cdots + a_1 x + a_0 \\ &\text{and} \\ b(x) &= B_1 x^{\frac{n}{2}} + B_0 \text{ where} \\ B_1 &= b_{n-1} x^{\frac{n}{2}-1} + \cdots + b_{\frac{n}{2}+1} x + b_{\frac{n}{2}} \\ B_0 &= b_{\frac{n}{2}-1} x^{\frac{n}{2}-1} + \cdots + b_1 x + b_0 . \end{aligned} \tag{3.3}$$

Let $c = a \cdot b$, hence $c(x) = a(x)b(x) = c_{2n_2} x^{2n-2} + \cdots + c_1 x + c_0$. To find $c(x)$, the schoolbook algorithm is used in the following way:

$$a(x)b(x) = A_1 B_1 x^n + (A_1 B_0 + A_0 B_1) x^{\frac{n}{2}} + A_0 B_0 . \tag{3.4}$$

As it can be seen, it is the usual polynomial multiplication. There are 4 multiplications, $A_1B_1, A_1B_0, A_0B_1, A_0B_0$ of sizes-$n/2$, and an addition $(A_1B_0 + A_0B_1)$ of size-n. Since we look at these polynomials as integers, there are the additions between the coefficients of x^n with $x^{\frac{n}{2}}$ and of $x^{\frac{n}{2}}$ with $x^0 = 1$. Therefore we must compute the number of operations that come from those steps as well. The complete operations can be seen in Figure 3.1. The lines denote the polynomials, and the two ends of each line state the first and the last coefficient of that polynomial. The dotted lines show where the overlaps on coefficients are located. Since each A_iB_j is of size $n-1$, in 1 and 3 we have $\frac{n}{2} - 1$ additions. In 2, there are $n-1$ additions. Then in total we have $2n-3$ additions.

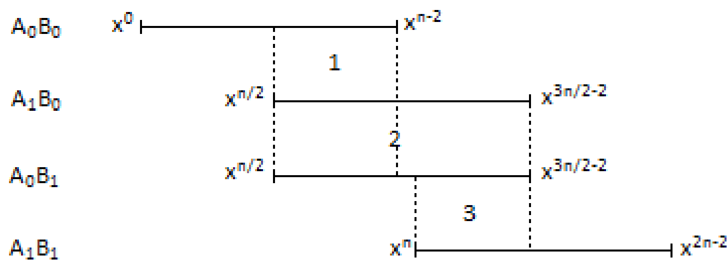

Figure 3.1: The cost of schoolbook algorithm

Since multiplications take the most time in such an algorithm we generally say the

schoolbook algorithm costs $4\mathbf{M}(\frac{n}{2})$. But from the recursive application of the algorithm we find the computational complexity as

$$\begin{aligned}\mathbf{M}(n) &= 4\mathbf{M}\left(\frac{n}{2}\right) + 2\cdot n - 3\\ &= 4(4\mathbf{M}\left(\frac{n}{4}\right) + 2\cdot\frac{n}{2} - 3) + 2\cdot n - 3\\ &= 4^2\mathbf{M}\left(\frac{n}{2^2}\right) + 4\cdot 2\cdot\frac{n}{2} + 2\cdot n - 4\cdot 3 - 3\end{aligned}$$

If we continue this way until we hit 2^l, we get $\mathbf{M}(\frac{n}{n}) = 1$ and group the operands,

$$\begin{aligned} &= 4^l + 2\cdot\frac{1-\left(\frac{4}{2}\right)^l}{1-\frac{4}{2}}\cdot n - 3\cdot\frac{1-4^l}{1-4}\\ &= 4^l - 2\cdot(1-2^l)\cdot n + 1 - 4^l\\ &= n^2 - 2\cdot(1-n)\cdot n + 1 - n^2\\ &= n^2 - 2n + 2n^2 + 1 - n^2\\ &= 2n^2 - 2n + 1 \in \mathcal{O}(n^2)\end{aligned}$$

From these, we see that the computational complexity of the schoolbook multiplication is $\mathcal{O}(n^2)$.

3.2 Karatsuba Algorithm

Up until 1960, it has been thought that the computational cost of multiplication is at best $\mathcal{O}(n^2)$, as it is in the schoolbook multiplication. But a paper published in 1962 about the discoveries of Anatoly Karatsuba [24] shows that Karatsuba multiplication is a faster algorithm after a certain threshold compared to the schoolbook method. This discovery led to other algorithms as well, like Toom-Cook methods, Schöngage-Strassen, Fürer's multiplication etc. [29].

This method is also a 2-way algorithm like schoolbook method. Using the same separation as in Equation 3.3, we multiply two polynomials $a(x) = A_1x^{\frac{n}{2}} + A_0$ and $b(x) = B_1x^{\frac{n}{2}} + B_0$ as

$$a(x)b(x) = A_1B_1x^n + ((A_0+A_1)(B_0+B_1) - A_1B_1 - A_0B_0)x^{\frac{n}{2}} + A_0B_0 \quad (3.5)$$

If we count the multiplications here we see that there are only 3, $A_1B_1, A_0B_0, (A_0 + A_1)(B_0 + B_1)$. Comparing this with the schoolbook method, we have one less multiplication but more additions in the coefficient of $x^{\frac{n}{2}}$. Cost of a single addition is much less than the cost of a single multiplication. Since we use Karatsuba as a recursive algorithm, as the size increases the number of both operations increase as well. Compared to schoolbook where we have $4\mathbf{M}(n)$ but less additions, Karatsuba's $3\mathbf{M}(n)$

and more additions becomes faster after some size. To find the computational cost of Karatsuba's Algorithm, we make a similar figure to the schoolbook algorithm's Figure 3.1.

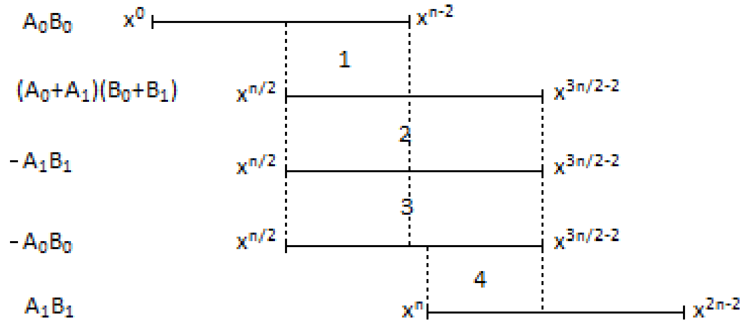

Figure 3.2: The cost of Karatsuba algorithm

To compute the cost, since this algorithm is recursive as well, we use similar operations to the ones in the complexity computations from Section 3.1. We have 3 recursive multiplications. The addition of $(A_0 + A_1)$ and $(B_0 + B_1)$ costs $n/2 - 1$ additions each. Looking at Figure 3.2 we have $n/2 - 1$ additions in 1 and 4, $n - 1$ in 2 an 3 for a total of $4n - 6$ additions. Karatsuba is also a 2-way algorithm, so we assume the sizes are halved at each recursive separation. Assuming bit size of $l = \log_2 n$. Then we have the following equations.

$$\begin{aligned}\mathbf{M}(n) &= 3\mathbf{M}\left(\frac{n}{2}\right) + 4 \cdot n - 6 \\ &= 3\left(3\mathbf{M}\left(\frac{n}{4}\right) + 4 \cdot \frac{n}{2} - 6\right) + 4 \cdot n - 6 \\ &= 3^2\mathbf{M}\left(\frac{n}{2^2}\right) + 3 \cdot 4 \cdot \frac{n}{2} + 4 \cdot n - 3 \cdot 6 - 6\end{aligned}$$

If we continue until we hit 2^l, we get $\mathbf{M}(\frac{n}{n}) = 1$ and group the operands,

$$\begin{aligned}&= 3^l + 4 \cdot \frac{1 - \left(\frac{3}{2}\right)^l}{1 - \frac{3}{2}} \cdot n - 6 \cdot \frac{1 - 3^l}{1 - 3} \\ &= 3^l + 8 \cdot \left(\frac{3^l}{2^l} - 1\right) \cdot n + 3(1 - 3^l) \\ &= n^{\log_2 3} + 8 \cdot \left(\frac{n^{\log_2 3}}{n^{\log_2 n}} \cdot n - 1\right) + 3 - 3n^{\log_2 3} \\ &= n^{\log_2 3} + 8n^{\log_2 3} + 3n^{\log_2 3} - 8n + 3 \\ &= 6n^{\log_2 3} - 8n + 3 \in \mathcal{O}(n^{log_2 3})\end{aligned}$$

Hence, we can see that the cost of the Karatsuba algorithm is $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.58})$. Which is asymptotically better than the schoolbook algorithm's $\mathcal{O}(n^2)$.

3.3 Toom-Cook Algorithms

Toom-Cook algorithms, Toom for short, define a family of multiplication methods that lower the asymptotical complexity of multiplication from $\mathcal{O}(n^2)$. After Karatsuba's work, Toom [37] and Cook [10] generalized these methods for a variety of k-way multiplications.

Toom methods are a form of polynomial interpolation. Given two polynomials and the result of their multiplication with undetermined coefficients, the resultant polynomial is evaluated at previously chosen points. Using these evaluations we find the coefficients of the resultant polynomial.

Toom methods has four steps. In the splitting step, we divide the polynomials to be multiplied into a number of pieces with previously determined degrees. If the original polynomials have the same degree, we use Toom-k methods, if they have different degrees, i.e. if they are unbalanced, we use Toom-$k + 1/2$ methods. Then we evaluate these polynomials at previously chosen points. After the evaluation phase, we have the actual interpolation, which is a matrix inversion. Finally we have the recomposition step where we combine the result back into an integer.

The degrees, the number of coefficients and therefore the matrix to be inverted is selected prior to multiplication so the cost can be asymptotically determined by the number of the coefficient multiplications arise from the interpolation step. The evaluation and the interpolation steps can be combined as well. Similar to the Karatsuba and schoolbook algorithms, we usually refer to the cost of a Toom multiplication with m-many multiplications.

3.3.1 Toom-2

Toom-2 uses 2-way separation.

$$a(x) = a_1x + a_0 \text{ and } b(x) = b_1x + b_0 \tag{3.6}$$

Let $c(x) = a(x)b(x) = c_2x^2 + c_1x + c_0 = a_1b_1x^2 + (a_0b_1 + a_1b_0)x + a_0b_0$. To find the three coefficients $\{c_2, c_1, c_0\}$, we need three interpolation points. These points are chosen such that the operations we need are minimal. The points used here are $\{0, 1, \infty\}$. By the point ∞, we don't literally mean infinity, but we mean the point which gives c_2 in the polynomial evaluation. Since a and b have smaller degrees than c, c_2 cannot be found by inputting points into the two operand polynomials, hence the use of the notation ∞. When we look at the evaluations at these points, we get the

following equations:

$$\begin{aligned} c(0) &= c_0 \\ &= a_0 b_0, \\ c(1) &= c_2 + c_1 + c_0 \\ &= (a_1 + a_0)(b_1 + b_0), \\ c(\infty) &= c_2 \\ &= a_1 b_1. \end{aligned} \tag{3.7}$$

Using these equations, we form the following linear equation system consisting of coefficient multiplications, or equivalently a matrix-vector product:

$$\begin{aligned} S = \begin{bmatrix} S_1 \\ S_2 \\ S_3 \end{bmatrix} &= \begin{bmatrix} a_1 b_1 \\ (a_1 + a_0)(b_1 + b_0) \\ a_0 b_0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_2 \\ c_1 \\ c_0 \end{bmatrix} = WC. \end{aligned} \tag{3.8}$$

To find $\{c_2, c_1, c_0\}$, we invert the 3x3 matrix in the left multiplicative operand in Equation 3.8. An important observation here is that we can either compute S_i as we invert, or compute the matrix multiplication after the inversion but there is no difference between doing either. We invert the S matrix by the following procedure. This is called the *inversion sequence*.

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \xrightarrow[R_2 - R_3]{R_2 - R_1} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right], \tag{3.9}$$

which leads to

$$\begin{aligned} W^{-1}S &= \begin{bmatrix} S_1 \\ S_2 - S_1 - S_3 \\ S_3 \end{bmatrix} \\ &= \begin{bmatrix} a_1 b_1 \\ (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0 \\ a_0 b_0 \end{bmatrix} = \begin{bmatrix} c_2 \\ c_1 \\ c_0 \end{bmatrix} = C. \end{aligned} \tag{3.10}$$

So, to find the product of two integers that can split as in Equation 3.6, we do the operations in Equation 3.10. If we look at the three operations itself, we see that they are the same operations in Equation 3.5, the Karatsuba algorithm. After we find all the c_i, we do the recombination phase to find the resulting integer, which is nothing more than computing $c_2 x^2 + c_1 x + c_0$, where x's are just shifts and memory management. Obviously this is also the same as it is in Karatsuba's algorithm. Its cost is the same as Karatsuba's. This is one of the reasons why Toom-Cook methods are called a generalization of the Karatsuba multiplication.

3.3.2 Toom-3

Toom-3 method is a 3-way method. This method is mentioned here without much investigation, since a version of it, the unbalanced Toom-3 is used in next chapters, albeit with some difference.

$$a(x) = a_2x^2 + a_1x + a_0 \text{ and } b(x) = b_2x^2 + b_1x + b_0 \tag{3.11}$$

The interpolation points we use are $\{0, 1, -1, 2, \infty\}$. Similar to Toom-2, we construct $c(x) = a(x)b(x)$, having the coefficients c_i for $0 \leq i \leq 4$. Then we have $5S_i$ and the matrix to be inverted as given below:

$$\begin{aligned} S = \begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \end{bmatrix} &= \begin{bmatrix} a_0b_0 \\ (a_2+a_1+a_0)(b_2+b_1+b_0) \\ (a_2-a_1+a_0)(b_2-b_1+b_0) \\ (4a_2+2a_1+a_0)(4b_2+2b_1+b_0) \\ a_2b_2 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = WC \end{aligned} \tag{3.12}$$

With these, we have an inversion sequence with the cost $8\mathbf{A}(n)+3\mathbf{B}$ that can be found in [5]. In the evaluation phase, our cost is $5\mathbf{M}(\frac{n}{2})$.

3.3.3 Unbalanced Toom-3

This method is important since it has a use in cubing. Consider the cube of a degree 1 polynomial $p(x) = p_1x + p_0$. We can find the cube with $p(x)^2p(x)$. If we look at $p(x)^2 = p_1^2x^2 + 2p_1p_0x + p_0^2$, we see that we can divide this in another way: $p(x)^2 = q_3x^3 + q_2x^2 + q_1x + q_0$. Since p_1^2, $2p_1p_0$ and p_0^2 is nearly double the size of p_1 and p_0, this way is actually the recommended way of representing the polynomial so that the multiplication with $p(x)$ easier. Then we have a multiplication of two polynomials with degrees 3 and 1, hence we use a method specialized for these kind of polynomials, Unbalanced Toom-3 [39].

Assume the polynomials we want to multiply are

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \text{ and } b(x) = b_1x + b_0. \tag{3.13}$$

Let $c(x) = a(x)b(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$. Unbalanced Toom-3 uses the same 5 interpolation points as Toom-3, $\{0, 1, -1, 2, \infty\}$. In the equation below we put

these points into polynomials c and a, b.

$$\begin{aligned} c(0) &= c_0 \\ &= a_0 b_0, \\ c(1) &= c_4 + c_3 + c_2 + c_1 + c_0 \\ &= (a_3 + a_2 + a_1 + a_0)(b_1 + b_0), \\ c(-1) &= c_4 - c_3 + c_2 - c_1 + c_0 \\ &= (-a_3 + a_2 - a_1 + a_0)(-b_1 + b_0), \\ c(2) &= 16c_4 + 8c_3 + 4c_2 + 2c_1 + c_0 \\ &= (8a_3 + 4a_2 + 2a_1 + a_0)(2b_1 + b_0), \\ c(\infty) &= c_4 \\ &= a_3 b_1. \end{aligned} \tag{3.14}$$

Like in all the other Toom-Cook methods, we use these equations to form a linear equation system on c_i:

$$\begin{aligned} S = \begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \end{bmatrix} &= \begin{bmatrix} a_0 b_0 \\ (a_3 + a_2 + a_1 + a_0)(b_1 + b_0) \\ (-a_3 + a_2 - a_1 + a_0)(-b_1 + b_0) \\ (8a_3 + 4a_2 + 2a_1 + a_0)(2b_1 + b_0) \\ a_3 b_1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = WC. \end{aligned} \tag{3.15}$$

Here we have many coefficients to compute with, unlike Toom-2's 2 coefficients for each polynomial. If we look at S_2 and S_3, we see that $a_2 + a_0$ appear in both cases. Then we can compute it only once, and reuse it later on. This means that there are optimizations for computing these operations. An optimization is given by Zanoni [39] where he evaluates the points $\{1, -1, 2\}$. The points $\{0, \infty\}$ are not included in the *evaluation sequence*, since they are just single coefficient multiplications. In the following equations, the sequence shown by Bodrato is given. Obviously, the equations must be done in the correct order. Assuming each a_i and b_i are $\frac{n}{2}$ words each, the cost for the "a" side is $7\mathbf{A}(\frac{n}{2}) + 3\mathbf{B}$ and the cost for the "b" side is $3\mathbf{A}(\frac{n}{2})$.

1	$u_4 = a_2 + a_0$	1	$v_2 = b_1 + b_0 \rightarrow$ "b" part of S_2
2	$u_3 = a_3 + a_1$	2	$v_3 = b_0 - b_1 \rightarrow$ "b" part of S_3
3	$u_2 = u_3 + u_4 \rightarrow$ "a" part of S_2	3	$v_4 = v_2 + b_1 \rightarrow$ "b" part of S_4
4	$u_3 = u_4 - u_3 \rightarrow$ "a" part of S_3		
5	$u_4 = a_2 + (a_3 \ll 1)$		
6	$u_4 = a_1 + (u_4 \ll 1)$		
7	$u_4 = a_0 + (u_4 \ll 1) \rightarrow$ "a" part of S_4		

(3.16)

After the evaluation, we do the multiplications to compute all S_i where $u_i v_i = S_i$ for $1 < i < 5$; $S_1 = a_0 b_0$ and $S_5 = a_3 b_1$. This takes $5\mathbf{M}(\frac{n}{2})$. An inversion sequence of the matrix W can be found in [5] and it costs $8\mathbf{A}(n) + 3\mathbf{B}$ and a division by 3. Then the recombination step follows like any other Toom-Cook algorithm. There are many different ways to do all the steps with differing number of shifts and additions, but the multiplications stay the same so when we are talking about the cost of Unbalanced Toom-3 for polynomials with n word coefficients we generally say it has a cost of $5\mathbf{M}(\frac{n}{2})$.

CHAPTER 4

POWERS 2 AND 3

In this chapter we investigate some squaring and cubing methods. We first compare the costs of usual multiplication methods when both operands are the same versus when they are different. Then we go into asymmetric squaring which uses interpolation, but with somehow *unorthodox* linear equations. After these we look at two cubing algorithms, and discuss their efficiencies if used as modular operations and non-modular operations.

4.1 Squaring Using Multiplication Methods

Generally, usual multiplication methods are used for squaring [8] like the schoolbook algorithm or the Karatsuba algorithm. But, since we are multiplying the same integer with itself there are some optimizations. For some algorithms, the same multiplications or additions appear more than once. So, instead of doing the same operation twice, we use the result from before, which leads to a less costly algorithm. Below we give both the schoolbook and the Karatsuba squaring versions and their costs. They are very similar to the multiplication cases, so we do not analyze as in depth as their multiplication counterparts.

For both algorithms, we use the same notation in Equation 3.3, ie. we divide the polynomial representation into two. However, since we are interested in squaring, we multiply $a(x)$ with itself.

4.1.1 Schoolbook Squaring

Repeating the method in 3.1 using $a(x)$ only, we get

$$a(x)^2 = A_1^2 x^n + 2A_1A_0x^{\frac{n}{2}} + A_0^2. \tag{4.1}$$

Looking at this equation, we see that we have one less addition in the coefficient of $x^{\frac{n}{2}}$. With this in mind and using the algorithm recursively, we have similar cost computations to the multiplication case. We see that we have $\mathbf{S}(n) = 2\mathbf{S}(\frac{n}{2}) + \mathbf{M}(\frac{n}{2}) + n - 2$ operations, which makes this algorithm faster than schoolbook multiplication.

4.1.2 Karatsuba Squaring

Similar to the schoolbook squaring, we use the method in 3.2 using only $a(x)$:

$$a(x)^2 = A_1^2 x^n + ((A_0 + A_1)^2 - A_1^2 - A_0^2)x^{\frac{n}{2}} + A_0^2. \tag{4.2}$$

Again we save an addition in the same place similar to the last section, and we have only squares and zero multiplications. If we compute its cost recursively we have $\mathbf{S}(n) = 3\mathbf{S}(\frac{n}{2}) + 7\frac{n}{2} - 5$ operations. Even though this is faster than both the squaring and multiplication variants of the schoolbook and Karatsuba multiplication algortihms, because of the schoolbook squaring algorithm's speed, we start using this algorithm at a higher word size [20].

4.2 Asymmetric Squaring

In this section we look at the *asymmetric* squaring methods as they are given in [8] by Chung and Hasan. Their work investigates a new 3-way squaring algorithm that is similar to Toom-3 but without the constant divisions that arise in that algorithm. To achieve that they change the linear system constructions by finding linear equations with c_i's that cannot be found only by evaluating the polynomials, where c_i are the coefficients of $c(x) = a(x)^2$, and $a(x) = a_2x^2 + a_1x + a_0$.

They use four methods to find these linear equations and they are given below:

- **Modulus:** A known method used in finite field squaring; using some small integers u and v, take the modulo of $c(x) = a(x)^2$ by $(x^2 + ux + v^2)$. This leads to the equivalence

  $$c_1' + c_0' \equiv (a_1'x + a_0')^2 \pmod{(x^2 + ux + v^2)},$$

  where each side are the results of the aforementioned modulus operation. If we expand the equation, we can see that we have two new linear equations on c_i':

  $$\begin{aligned} c_1' &= a_1'(2a_0' - ua_1'), \\ c_0' &= (a_0' - va_1')(a_0' + va_1'). \end{aligned}$$

  Note that each only requires two coefficient multiplications.

- **Hermite Interpolation:** This is a known method in which we derive $c(x)$ and evaluate there. Since this is a square,

  $$c'(x) = 2a(x)a'(x)$$

  and we only have one coefficient multiplication.

- **Difference:** This is simply the difference of two squares with disctint x_i and x_j:

  $$a(x_i)^2 - a(x_j)^2 = (a(x_i) + a(x_j))(a(x_i) - a(x_j)).$$

  This is useful if we use $a(x_i)$ and $a(x_j)$ in the evaluation already, then we find another equation with two additions and a multiplication.

- **Duality:** Another very known method where we find c_i and c_{2n-i-2} with the same function using different input. If we have

$$c_i = f(a_0, a_1, \cdots, a_{n-1}),$$

then

$$c_{2n-i-2} = f(a_{n-1}, a_{n-2} \cdots, a_0).$$

So, if we have a new linear equation from other methods, we can get another one using duality method.

Using these methods, they propose three squaring methods in [8]. Here we only give SQR_3 as it is the fastest out of all three.

Squaring $a(x) = a_2x^2 + a_1x + a_0$ with the result $c(x) = a(x)^2 = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$ using SQR_3 [8]:

$$\begin{aligned} c_0 &= S_0 = a_0^2 \\ S_1 &= (a_2 + a_1 + a_0)^2 \\ S_2 &= (a_2 - a_1 + a_0)^2 \\ c_3 &= S_3 = 2a_1a_2 \\ c_4 &= s_4 = a_2^2 \\ T_1 &= (S_1 + S_2)/2 \\ c_1 &= S_1 - T_1 - S_3 \\ c_2 &= T_1 - S_4 - S_0 \end{aligned} \tag{4.3}$$

This computation requires $4\mathbf{S}(n) + \mathbf{M}(n)$ with a little overhead compared to Toom-3's $5\mathbf{S}(n)$ with large overhead. Notice that there are no divisions as well. The test results in [8] show that between 2400-6700 bits long integer squaring, SQR_3 is faster than *gmp* library's implementation of multiplication (*mpn_mul()*). This is a very important result as it shows that taking a square out in favour of a multiplication may lead to a speed up in certain sizes if the overhead is less enough to cover the multiplication. This is one of the reasons we thought Zanoni's cubing algorithm can be used in a modular way, explained in the next sections.

The reason we investigated asymmetric squaring was firstly, it was a fast squaring algorithm and secondly the four methods proposed are interesting. We searched for ways to use them in cubing algorithms, classical and Zanoni's both, but the coefficients do not behave as nicely as they do in the squaring. The reduction of complexity results from using same coefficents many times for squaring. For cubing, this doesn't happen. Modulus and Hermite interpolation methods introduce too many coefficents, difference method is not important without any new linear equations, there is no use trying duality.

4.3 Classical Cubing Method

Take a big integer a, and its polynomial representation $f(x) = a_1x + a_0$, where a_0 is n words and a_1 is $t \in \{n, n-1\}$ words where $n \in \mathbb{Z}^+$. For our intents, let us say $n \cong t$ so that we can say the size of a is $2n$ words long. As it can be seen, this method uses 2-way splitting. To find the cube, what we do is

$$a^3 = (f(x))^3 = (a_1x + a_0)^2(a_1x + a_0) \tag{4.4}$$

It is pretty basic, we first square the polynomial, then multiply it with itself. If we look at this from a non-modular perspective, we see that a^2 is a $4n$ word integer, while a is $2n$ words long. One of the best algorithms for multiplying these integers is to use Unbalanced Toom-3 algorithm, from section 3.3.2. As previously noted, Unbalanced Toom-3 Algorithm uses 5 multiplications of half-size in its interpolation step. Counting the first square of $2n$ word long integer as well, the cost of cubing using this method is $\mathbf{C}(2n) = \mathbf{S}(2n) + 5\mathbf{M}(n)$. $\mathbf{S}(2n)$ can be changed to whichever squaring algorithm is used. For example, if we use the Karatsuba algorithm to compute the square, we have $\mathbf{C}(2n) = 3\mathbf{S}(n) + 5\mathbf{M}(n)$.

On the other hand, if we need reductions we change the algorithms used in this method. After we take the square of a_1x+a_0, we reduce it using one of the appropriate reduction algorithms. Let us denote the reduction of kn word integer using a mn word modulus with $\mathbf{R}(kn \to mn)$. Then this reduction of square costs $\mathbf{R}(4n \to 2n)$. After the reduction we are left with a $2n$ word integer b. To multiply b with a, we don't have to use Unbalanced Toom-3, and we should not since b and a are of same length. Instead we can use Karatsuba's algorithm, which will net us a cost of $3\mathbf{M}(n)$. After that, since the size is again $4n$, we need another reduction of $\mathbf{R}(4n \to 2n)$. In total, assuming we use Karatsuba for both the multiplications and the square, the cost of this modular cubing algorithm is $3\mathbf{S}(n) + 3\mathbf{M}(n) + 2\mathbf{R}(4n \to 2n)$.

4.4 Zanoni's Cubing Algorithm

Zanoni's [6] cubing algorithm is a very different approach then what we have seen before in this thesis. Compared to the other usual multiplication methods, Zanoni's method does not compute the cube of a polynomial directly. Instead, it computes another polynomial that is somewhat similar to the cube of the orignial polynomial, and using fairly non-expensive operations finds the cube from there. Let f, a, n be defined in 4.3. As per Zanoni's definition, the idea of computing a similar polynomial comes from the following.

Definition 4.1. Let $f(x)$ and $g(x)$ be two polynomials in $\mathbb{Z}[x]$ of same degree k with coefficients f_i and g_i respectively. f is a *master* of $g \iff f_i \mid g_i$ for $0 < i < k$, and if for some i the coefficient f_i or g_i is zero then the other polynomial's respective coefficient is also zero.

If we have an ordered vector v that has all the $\dfrac{g_i}{f_i}$ in the correct order then we say

that f is a v-*master* of g. For example, $f(x) = 6x^3 + 4x^2 + 1$ is a $[3, 4, 1]$-master of $g(x) = 2x^3 + x^2 + 1$, but not a master of $h(x) = 2x^3 + 1$.

Using this definition, and the polynomial f from before, we look at g, the $[1, 1, 1, 9]$-master of f.

$$\begin{aligned} g(x) =& a_1^3x^3 + 3a_1^2a_0x^2 + 3a_1a_0^2x + 9a_0^3 \\ =& (a_1^2x^2 + 3a_0^2)(a_1x + 3a_0). \end{aligned} \tag{4.5}$$

As we can see, this polynomial only differs from $(f(x))^3$ in the x^0's coefficient and also can be factorized into different polynomials. This was not the case with $(f(x))^3$. Looking at the factorization, we have $2\mathbf{S}(n)$ in the first part. With this in mind, and considering that x^2 shifts the coefficient $2n$ words, $(a_1^2x^2 + 3a_0^2)$ becomes a $4n$ word polynomial. Hence the multiplication with $(a_1x + 3a_0)$ is a $4n$ by $2n$ word multiplication for which we can use Unbalanced Toom-3. After the multiplication, we can divide the last coefficient by 9 to get back to the original cube.

However, as Zanoni points out in his paper, this is not the case. Unfortunately in the Unbalanced Toom-3 method the term $9a_0^3$ never appears alone. To be able to use Unbalanced Toom-3, he proposes to change the polynomial g into a more suitable form.

$$\begin{aligned} h(x) =& (a_{1,1}a_1)x^4 + (a_{1,0}a_1 + 3a_{1,1}a_0)x^3 + (3a_{0,1}a_0 + 3a_{0,1}a_1)x^2 \\ &+ (27a_{0,0}a_1 + 9a_{0,1}a_0)x + 81a_{0,0}a_0 \\ =& \sum_{i=0}^{4} c_i x^i, \end{aligned} \tag{4.6}$$

where $a_1^2 = a_{1,1}x + a_{1,0}$, $a_0^2 = a_{0,1}x + a_{0,0}$. Now this polynomial works perfectly with Unbalanced Toom-3. After the algorithm, we just divide c_0 by 81 and c_1 by 9 to go back to $(f(x))^3$.

If we count the operations, the cost of this algorithm can be seen as $\mathbf{C}(n) = 2\mathbf{S}(n) + 5\mathbf{M}(n)$. The two constant divisions at the end are fast since the sizes we work on are very large and the divisors are known beforehand so we can optimize them.

Table 4.1: Time comparisons of two cubing algorithms

#-bits	Classical	Zanoni's
512	$\mathbf{0.01266 \cdot 10^{-5}}$	$0.03681 \cdot 10^{-5}$
1024	$\mathbf{0.04582 \cdot 10^{-5}}$	$0.06757 \cdot 10^{-5}$
2048	$\mathbf{0.16623 \cdot 10^{-5}}$	$0.16712 \cdot 10^{-5}$
4096	$0.48600 \cdot 10^{-5}$	$\mathbf{0.47783 \cdot 10^{-5}}$
8192	$1.42976 \cdot 10^{-5}$	$\mathbf{1.38949 \cdot 10^{-5}}$

In Table 4.1, we see that after 2048-bits, Zanoni's algorithm starts to become faster than the classical one. The implementations are done using the *gmp* library on *C++*, using *mpn_* level functions. Further notes about the implementations for this chapter can be found in Section 5.7.

4.4.1 Modular Approach

Since our main purpose is to find a modular cubing algorithm, we looked at where we can put the reduction operations in Zanoni's Algorithm. Assuming the integer to be cubed and the modulus both are $2n$ words, we don't have many places to reduce. Most obvious choice is to reduce at the end of the whole cubing operation, but a cubed $2n$ word integer becomes a $6n$ word integer. This was problematic since we thought that $\mathbf{R}(6n \to 2n)$ costs more than $2\mathbf{R}(4n \to 2n)$. Our first implementations used the general *mpz_* level functions in the *gmp* library. We don't include these timings, but in them it showed that the larger the difference between the modulo and the integer to be reduced, the slower the reduction by a great amount. However, later on when we implemented the same reductions using *mpn_* level functions, we got the results shown in Table 4.2. In the table, #-bits is the bit size of the modulo, which we take to be $2n$-words. The columns are for integers with kn-words. For example, for the first row, $n = 256$, and the integers to be reduced are $768, 1024, 1536, 2048$-bits.

Table 4.2: Time comparisons of *gmp* library's reduction function for different bit-sizes

#-bits	3n	4n	5n	6n
512	$0.00146 \cdot 10^{-5}$	$0.00170 \cdot 10^{-5}$	$0.00249 \cdot 10^{-5}$	$0.00305 \cdot 10^{-5}$
1024	$0.00276 \cdot 10^{-5}$	$0.00391 \cdot 10^{-5}$	$0.00553 \cdot 10^{-5}$	$0.00702 \cdot 10^{-5}$
2048	$0.00622 \cdot 10^{-5}$	$0.01078 \cdot 10^{-5}$	$0.01574 \cdot 10^{-5}$	$0.02069 \cdot 10^{-5}$
4096	$0.01780 \cdot 10^{-5}$	$0.03415 \cdot 10^{-5}$	$0.05103 \cdot 10^{-5}$	$0.06742 \cdot 10^{-5}$

The table shows us that the argument above does not hold, and

$$2\mathbf{R}(4n \to 2n) \approx \mathbf{R}(6n \to 2n).$$

This changed our implementation. Below, we first give the other places which we can reduce at, in case there is an optimized way of reducing for those specific differences in size.

We cannot reduce after the first squaring, which increases the size to $4n$, because then the form would change. We cannot reduce inside the Unbalanced Toom-3 algorithm since multiplications there are between n word integers already. On the other hand, the recombination step is available. Assume that we are done with the evaluation and interpolation steps. We are left with 5 integers $w_0, w_1, w_{-1}, w_2, w_\infty$, corresponding to the interpolation points $\{0, 1, -1, 2, \infty\}$. Then the recombination step is

$$(((((((w_\infty \ll n) + w_2) \ll n) + w_1) \ll n) + w_{-1}) \ll n) + w_0. \tag{4.7}$$

Here, when we shift w_∞ by n and add w_2 to it, we get a $3n$ word integer. We can then reduce it to a $2n$ word integer and continue along the recombination step in this way. Or, we can keep the integers as is until the addition of w_1, then reduce the $4n$ word integer to a $2n$ one. In the first version we have $4\mathbf{R}(3n \to 2n)$, in the second one we have $2\mathbf{R}(4n \to 2n)$. The second is the same as what happens if you square and multiply instead of using Zanoni's algorithm. There you have the same reductions.

On the other hand, looking at 4.2, we see that we can just reduce at the end with a similar if not a lower cost. There is an additional benefit of this as well. When implementing the result of a multiplication operation such as the Toom-Cook methods, some of the shifts and additions are not necessary. Since we know where all the interpolation points will end up in the result, we can just put them at their correct places in the memory. We know that w_0 is always going to start from the first word of the result, let us call it the word 0. Similarly, w_1 starts from the word-$2n$, and w_∞ from the word-$4n$. If we put these points at their correct positions, we are only left with two additions, the addition at the word-n with w_{-1} and at $3n$ with w_2. Note that we do not need shifts here. If we were to use a reduction anywhere else, we would need to do all of the additions and shifts. In the implementation results below we use the implementation without shifts and additions for modular Zanoni's cubing algorithm.

Table 4.3: Time comparisons of two modular cubing algorithms

#-bits	Classical	Zanoni's
512	$\mathbf{0.04161 \cdot 10^{-5}}$	$0.06983 \cdot 10^{-5}$
1024	$\mathbf{0.11333 \cdot 10^{-5}}$	$0.14204 \cdot 10^{-5}$
2048	$\mathbf{0.32512 \cdot 10^{-5}}$	$0.38226 \cdot 10^{-5}$
4096	$\mathbf{1.02509 \cdot 10^{-5}}$	$1.17267 \cdot 10^{-5}$
8192	$\mathbf{3.21581 \cdot 10^{-5}}$	$3.58838 \cdot 10^{-5}$

In Table 4.3, the reduction modulo and the integers to be cubed are of the same bit size. As we can see from the table, compared to the classical method, modular Zanoni's method is worse. The difference between these results and Table 4.1 comes from the fact that the methods we compare with Zanoni's method are different in both cases, as discussed in 4.3. This table shows that using modular Zanoni's method for modular exponentiation is worse than just using the straightforward modular cubing.

CHAPTER 5

REPRESENTATIONS OF EXPONENTS AND EXPONENTIATION ALGORITHMS

In this chapter, we explain some different ways of representing an integer exponent. Some of these representations are for general usage, however, addition chains, hybrid binary-ternary number system and double base representations are very useful in exponentiation and point multiplication for elliptic curves. We also give the exponentiation algorithms that are used with given representations. Complexities of the algorithms are computed as well.

Since our aim is to explore the use of a modular cubing algorithm, we investigate the representations that could be useful for cubing alongside the ones that are useful in general.

5.1 Schoolbook Exponentiation

Like any other integer operation, there is an obvious method for exponentiation as well. Let g and n be positive integers. To find g^n, we multiply g by itself n times,

$$g^n = g^2 \cdot g \cdot g \cdots g. \tag{5.1}$$

Notice that the first multiplication is a square, and all the others are multiplications. So the total cost of schoolbook exponentiation is $\mathrm{S} + (n-2)\mathrm{M}$. If we look at it in a modular way, after each multiplication and square we need to reduce the result, which makes $n-1$ necessary reductions. Reduction is also a costly operation, and this many operations make the algorithm unusable for a cryptosystem taliored for exponentiation, such as RSA. As we will shortly see, this basic method is very slow compared to the other methods.

5.2 Base-w Representation

One of the methods for representing an integer is base-w method, similar to polynomial representation equation 3.1. Base-10, or the decimal representation is used daily, and

bases 2, binary, and 16, hexadecimal, are widely used in computer science. For a given non-negative $n \in \mathbb{Z}$ and a base $w \geq \mathbb{Z}^+$, we have a unique representation of the form

$$n = d_{k-1}w^{k-1} + d_{k-2}w^{k-2} + \cdots + d_1 w + d_0, \tag{5.2}$$

where d_i are non-negative integers such that $d_i < w$ for $0 \leq i \leq k-1$, and $k \in \mathbb{Z}^+$ is the length of n in base-w. In this form, since each $d_i < w$ and the exponent of w is at most $k-1$, it can be seen that $w^k > n \geq w^{k-1}$. Therefore $k = \lfloor log_w n \rfloor + 1$ where $\lfloor n \rfloor$ denotes the largest integer smaller than or equal to n. The representation is most of the time represented with $(d_{k-1}, d_{k-2}, \cdots, d_1, d_0)_w$. Below is the algorithm for finding such d_i's.

5.2.1 Algorithm

Algorithm 3 Base-w Representation

Input: $n, w \in \mathbb{Z}, w \geq 2$
Output: $d_{k-1}, d_{k-2}, \cdots, d_1, d_0$
1: $i \leftarrow 0$
2: **while** $n \geq w$ **do**
3: $\quad m \leftarrow \lfloor n/w \rfloor$
4: $\quad d_i \leftarrow n - mw$
5: $\quad n \leftarrow m$
6: $\quad i \leftarrow i + 1$
7: **end while**
8: $d_i \leftarrow n$
9: **return** $d_{k-1}, d_{k-2}, \cdots, d_1, d_0$

In Algorithm 3, we divide n by w at each iteration to find the quotient q and the remainder d_i. Then we assign q to n and continue until n is zero.

Example 5.1. Using Algorithm 3 to find the base-4 representation of 133.

$$\begin{array}{lll} i = 0 & q = 3 & d_0 = 1 \\ i = 1 & q = 8 & d_1 = 1 \\ i = 2 & q = 2 & d_2 = 0 \\ i = 3 & q = N/A & d_3 = 2 \end{array} \tag{5.3}$$

So, $133 = 2 \cdot 4^3 + 0 \cdot 4^2 + 1 \cdot 4 + 1 = (2011)_4$. Throught the thesis the least significant digit, ie. the digit that is the coefficient of $w^0 = 1$ is on the rightmost side unless otherwise specified.

The cost of the base representation depends heavily on the base itself. As the base gets larger, the number of iterations becomes lower. However, the divisions get more costly.

5.2.2 Binary Representation and Repeated Squaring Method

Binary, bit, or the base-2 representation can be found by setting $w = 2$ in Algorithm 3. However, in most computers base-2 representation is the default. In computer memory, integers are stored in different sizes of bytes, where each byte is 8-bits. Since the integers are already stored as bits, we can use shifts and logical operations to find the i-th bit of the integer. Assuming we have an integer n, getting the i-th bit is done the following way:

$$(n \gg i)\&1, \tag{5.4}$$

where $\gg i$ denotes the right shift by i bits, i.e. division by 2, $\&$ is the logical *and* operation. These operations are basically *free*, hence finding the representation does not require the utilization of Algorithm 3. This also allows us to not allocate memory for storing bit representation.

Bit representation has many applications on computer science, cryptography and many other disciplines, and its application on exponentiation is one of the major methods for computing g^n. Bit representation is used with *repeated squaring algorithm* and it has two versions, *left-to-right* and *right-to-left*. Assuming $n = (d_{k-1}, \cdots, d_1, d_0)_2$, both versions are given below.

Algorithm 4 Repeated Squaring Algorithm, *l-t-r*

Input: $g, n \in Z^+, n = (d_{k-1}, \cdots, d_1, d_0)_2$
Output: g^n

```
 1: if d_{k-1} = 1 then
 2:     r ← g
 3: else
 4:     r ← 1
 5: end if
 6: for i from k − 2 to 0 do
 7:     r ← r^2
 8:     if d_i = 1 then
 9:         r ← r · g
10:     end if
11: end for
12: return r
```

Left-to-right repeated squaring method works in this way: After the initialization steps 1-4, we go bit by bit from the leftmost to the rightmost, as per the algorithm's name. At each bit, we first take the square of the result variable, and check whether that bit is 0 or 1. If it is 0, we continue to the next iteration, if it is 1, we multiply the result with the base.

Example 5.2. Let us find g^{15}. Bit representation of 15 is $(1111)_2$. We follow Algorithm 4:

$$r = g \to g^2 \to g^3 \to g^6 \to g^7 \to g^{14} \to g^{15}. \tag{5.5}$$

We see that we have $3\mathbf{M} + 3\mathbf{S}$. We can generalize this cost computation. A random integer of bit size k has $k/2$ 1-bits and 0-bits on average in its bit representation. Since we only have assignments in the initialization steps, we have $k-1$ iterations. Since we have a square at each step and a multiplication at $k/2$ steps on average, we can say that Algorithm 4 has a cost of $(k-1)\mathbf{S} + (\frac{k}{2}-1)\mathbf{M}$.

Algorithm 5 Repeated Squaring Algorithm, *r-t-l*

Input: $g, n \in Z^+, n = (d_{k-1}, \cdots, d_1, d_0)_2$
Output: g^n

```
1:  s ← g
2:  r ← 1
3:  for i from 1 to k − 1 do
4:      if d_i = 1 then
5:          r ← r · s
6:      end if
7:      s ← s^2
8:  end for
9:  r ← r · s
10: return r
```

Right-to-left repeated squaring starts from the least significant digit instead of the most. This difference does not change the number of operations, but the algorithm separates the squaring and multiplication in the iterations. This results in the need for additional space for the powers of g in the variable s.

Example 5.3. Let us find g^{15} using Algorithm 5. Bit representation of 15 was $(1111)_2$.

$$\begin{aligned} r = &\ 1 \to g \quad \to g^3 \to g^7 \quad \to g^{15}, \\ s = &\ g \to g^2 \quad \to g^4 \to g^8. \end{aligned} \tag{5.6}$$

5.2.3 Ternary Representation and Repeated Cubing Method

The ternary representation is the same as taking $w = 3$ in Algorithm 3. In this representation, we have digit 2 along with digits 1 and 0. Therefore we have to use Algorithm 3, or use additional operations to Equation 5.4. Since base-3 does not naturally appear in computers, we also have to store the representation digits in an array.

Apart from repeated squaring, we have the m-ary method for computing the power of a big integer. It is described in Knuth's book [25]. This method can be seen as the generalised version of the Algorithm 4 for other bases than 2. It incorporates the other digits that come up when we write the integer in base-m. However, we are interested in power 3, so we only give the algorithm for 3-ary, or the repeated cubing method.

Algorithm 6 Repeated Cubing Algorithm

Input: $g, n \in Z^+, n = (d_{k-1}, \cdots, d_1, d_0)_3$
Output: g^n

```
1:  s ← g^2
2:  if d_{k-1} = 0 then
3:      r ← 1
4:  else if d_{k-1} = 1 then
5:      r ← g
6:  else
7:      r ← s
8:  end if
9:  for i from k − 2 to 0 do
10:     r ← r^3
11:     if d_i = 1 then
12:         r ← r · g
13:     else if d_i = 2 then
14:         r ← r · s
15:     end if
16: end for
17: return r
```

The m-ary method can easily be deduced from Algorithm 6. Instead of computing just the square of g, we also compute the other digits arise from the base representation which increases the precomputation phase. And in the iteration starting from step 9 of the algorithm, we check the digit we are on and multiply the result with the according precomputed power.

The cost of this algorithm is computed similar to *repeated squaring*. Since now we have 3 digits in the representation, each can occur with $\frac{1}{3}$ probability in a random integer. Then we have $(k-1)\mathbf{C} + \frac{2}{3}(k-1)\mathbf{M} + \mathbf{S}$ many operations. If the base-3 representation does not have a 2 in it, then the algorithm can be arranged to not to compute the square.

We can compare binary and ternary methods by their relative sizes. Say that an integer n is of size k if represented with base-2, and size k' if represented with base-3. If we compare these lengths, we get the equality $1.58k' = k$.

5.2.4 Window Methods

In the m-ary method, instead of writing the exponent in base-m we can group the bits in a certain way. If we would like to use the m-ary method then we can divide the bit representation of the exponent in a way where each group has size $\log m = k$. This is especially easy if m is a power of two, otherwise we pad after the most significant bit with 0's until it fills a k bit group. As an example, if the exponent is 206, then its bit representation is $(11001110)_2$. And if we would like to use 4-ary method, we divide

the bit representation into $\log 4 = 2$ bits each. Then the exponent becomes

$$11 \mid 00 \mid 11 \mid 10.$$

Each of these parts are called *windows*. If we turn back to the 3-ary method we see that if we get a 0 in its base 3 expansion, we only take a cube. Obviously this is also the case with the general m-ary methods. In the above example, we only take the fourth power at the second window from the left. At all the other windows, there is at least one multiplication which raises the cost. Windows with only zero bits in them are called *zero windows*.

Since raising to the power m is inescapable, we'd rather have less multiplications at each window, which means having more zero windows. It can be seen that as the exponents get larger, the number of zero windows gets lower. Assuming in a random integer zeroes and ones are distributed equally, and if we try to split an exponent into k bits each, the probability that a window is a zero window is 2^{-k}. On the other hand, as k gets smaller, we do not have big jumps of larger exponents, hence we have more multiplications. The *sliding window methods* [25] tries to solve this problem by allowing variable lengths of windows to get more zero windows while also balancing the size of k.

The sliding window exponentiation algorithm is the general m-ary method, however instead of just representing the exponent in base-m we split the bit representation into windows. There are different ways of splitting the windows, here we give two. One of them fixes the sizes of nonzero windows, the other allows them to have different values under a limit.

5.2.5 Constant Length Nonzero Windows

This method is proposed in [25] In *CLNW*, we start from the least significant bit and collect the bits in the representation into a nonzero window or a zero window at each step. As the name suggests, all nonzero windows are the same length. Assuming the length of the windows is k, the algorithm can be explained with the following steps [27]:

- **First Step.** Check the least significant bit of the exponent. If the bit is 0, collect the bit into the first zero window, and go to Zero Step, otherwise go to Nonzero Step.
- **Zero Step.** Check the next bit. If it is 0, collect the bit into the current zero window and continue with Zero Step. Otherwise go to Nonzero Step.
- **Nonzero Step.** Collect the current bit and the next $k-1$ into a nonzero window. Check the next bit. If it is 0, collect it into a new zero window, and go to Zero Step. Otherwise continue with Nonzero Step.

The algorithm ends when there are no more bits to read. As it can be seen, all the successive 0's are collected into a single zero window hence no two zero windows

are allowed consecutively. After this step, we precompute the required exponents and continue with the m-ary algorithm. For example, using $k = 3$ and taking the exponent as $20708 = (101000011100100)_2$, to find g^{20708} for an integer g, we start with

$$101 \mid 0000 \mid 111 \mid 001 \mid 00.$$

Continuing with this example, we need to consider the windows $\{001, 011, 101, 111\}$. The windows $\{010, 100, 110\}$ will not appear, since their last 0's have to be in a zero window. The others appear, therefore we need to precompute the exponentiations for them. Since 001 is 1, we don't have to precompute it. Then we are left with $\{3, 5, 7\}$, so we find $\{g^3, g^5, g^7\}$. As per m-ary algorithm, we find g^{20708} with the following sequence:

$$g^5 \to (g^5)^{16} \to (g^{80})^8 \to g^{640} \cdot g^7 \to (g^{647})^8 \to (g^{5176}) \cdot g \to (g^{5177})^4 \to g^{20708}.$$

Using this method without considering the precomputations we need $2\mathbf{M} + 12\mathbf{S}$. If we take precomputation computations into account as well, we need $5\mathbf{M}+13\mathbf{S}$. If we were to compute the same exponent using the repeated squaring method, we would have needed $5\mathbf{M} + 14\mathbf{S}$. One less square comes from computing $\{g^3, g^5, g^7\}$ consecutively.

5.2.6 Variable Length Nonzero Windows

This method is proposed in [7]. In this version, lengths of nonzero windows can change under a maximum amount. For *VLNW*, we use two parameters. k, the maximum length of nonzero windows, and q, the minimum required zeros to change from nonzero to zero window. We use the following procedure to split the integer into windows:

- **First Step.** Just as it was in *CLNW*, we start by checking the least significant bit of the exponent. If it is 0, collect it into the first zero window, and go to Zero Step, otherwise go to Nonzero Step.
- **Zero Step.** This is also the same with *CLNW*. Check the next bit, if it is 0 collect it into the current zero window and continue with another Zero Step. Otherwise go to Nonzero Step.
- **Nonzero Step.** Collect the current bit into a nonzero window. Chect the next $q - 1$ bits. If all of them are 0, go to Zero Step. If there is a 0 before all $q - 1$ bits are checked, collect the bits before that first 0 into the current nonzero window, collect the 0 into a zero window, then go to Zero Step. If all $q - 1$ bits are nonzero, collect them into the current nonzero window. After all q bits are checked including the bit at the start of the step, check the next bit. If it is 0, collect it into a zero window and go to Zero Step. Otherwise go to Nonzero Step.

Similar to *CLNW*, two zero windows will not be adjacent to each other. Instead they will be assimilated into a single zero window. Two nonzero windows can be adjacent, however this means that the least significant one is full, ie. it has a length of k. We

can start with least or most significant bit with *VLNW* [27]. If we continue with our example from the last section, writing 20708 in *VLNW* with $k = 3$ and $q = 2$ and starting from the least significant bit, we get

$$101 \mid 0000 \mid 111 \mid 00 \mid 1 \mid 00.$$

With this method we can reduce the number of precomputed powers. It is suggested that using a larger window size may lead to a reduced number of multiplications [7].

5.3 Non Adjacent Form

The *non adjacent form* [32] of a positive integer is a unique representation with digit -1 along with 0 and 1. This form has a property that allows us to do less multiplications if used for an exponentiation algorithm: two consecutive digits cannot be nonzero. This form of $n \in \mathbb{Z}^+$ can be computed using Algorithm 7 [21]. We denote the *non adjacent form* of n by NAF(n) $= \sum_{i=0}^{k-1} d_i 2^i = (d_{k-1}, \cdots, d_1, d_0)_{NAF}$ where $d_i \in \{-1, 0, +1\}$.

Algorithm 7 Finding NAF(n)

Input: $n \in Z^+$
Output: NAF(n) $= (d_{k-1}, \cdots, d_1, d_0)_{NAF}$

1: $i \leftarrow 0$
2: **while** $n \geq 1$ **do**
3:     **if** $n \pmod 2 = 1$ **then**
4:         $d_i \leftarrow 2 - (n \bmod 4)$
5:         $n \leftarrow n - d_i$
6:     **else**
7:         $d_i \leftarrow 0$
8:     **end if**
9:     $n \leftarrow \frac{n}{2}$
10:     $i \leftarrow i + 1$
11: **end while**
12: **return** $(d_{k-1}, \cdots, d_1, d_0)_{NAF}$

In Algorithm 7, we consecutively divide n by 2. When k is odd we set the remainder to be one of $\{-1, +1\}$ so that in the next iteration n is even, hence we make sure that the next digit is 0. There are further properties [21] of NAF. Compared to the bit representation, the length of NAF(n) can be at most one more, and in average, there are $1/3$ nonzero digits in a given NAF representation.

Algorithm 8 Computing exponentiation with NAF

Input: $g, n \in Z^+, n = (d_{k-1}, \cdots, d_1, d_0)_{NAF}$
Output: g^n

```
1:  r ← 1
2:  m ← g^{-1}
3:  for i from k − 1 to 0 do
4:      r ← r^2
5:      if d_i = 1 then
6:          r ← r · g
7:      else if d_i = −1 then
8:          r ← r · m
9:      end if
10: end for
11: return r
```

Algorithm 8 is a modified version of Algorithm 4. There are two big differences however. Firstly, the NAF(n) has less nonzero digits, so there are less multiplications. Since in average there are $1/3$ nonzero digits, we can say that there will be $k/3$ multiplications and k squares. The second difference is the division by m or the computation of the inverse in the modular case. The inversion operation can take a considerable time [21], therefore rendering the NAF exponentiation useless for algorithms such as RSA. Nonetheless, it is useful in fixed exponent - variable base exponentiation, because we can precomute and store the inverse for later exponentiations.

Example 5.4. Using Algorithm 7 we find NAF(15) = $(1000\ –1)_{NAF}$. We compute g^{15} by the sequence

$$g^{15} = 1 \rightarrow g \rightarrow g^2 \rightarrow g^4 \rightarrow g^8 \rightarrow g^{16} \rightarrow g^{15} \tag{5.7}$$

The cost is $\mathbf{M} + 4\mathbf{S} + \mathbf{I}$. Comparing this to the sequence using repeated squaring algorithm in Example 5.2, we see that we have one more square but 2 less multiplications. However, if we are not using the same base for many operations, we have the cost of inversion as well.

5.4 Hybrid Binary-Ternary Number System

The Hybrid Binary-Ternary Number System (HBTNS) is introduced as a representation system for exponentiation by Dimitrov and Cooklev in [13]. This system mixes bases 2 and 3 and uses both bases to express the integer. We need two arrays to represent the integer, one for the digits and another for the bases each digit belongs to. Since both bases are used, the digit representation has a shorter length than base 2, but a longer one than base 3. So it can be thought as a system with a base between 2 and 3. Finding the representation is not as trivial as the base 2 case, and we have two arrays of same size with each other to keep in the memory. Nonetheless, it is still faster to find than optimized addition chains and other fixed exponent algorithms.

5.4.1 Algorithm

Algorithm 9 Hybrid Binary-Ternary Form Representation

Input: $n \in \mathbb{Z}^+$
Output: The arrays *digit* and *base*

```
1: i ← 0
2: while n > 0 do
3:     if n ≡ 0 (mod 3) then
4:         digit[i] ← 0
5:         base[i] ← 3
6:     else
7:         if n ≡ 0 (mod 2) then
8:             digit[i] ← 0
9:         else
10:            digit[i] ← 1
11:        end if
12:        base[i] ← 2
13:    end if
14:    n ← ⌊n/base[i]⌋
15:    i ← i + 1
16: end while
17: return digit, base
```

The algorithm is actually very similar to the algorithm for finding bit representation. If n at the current iteration is divisible by 3, then its digit is 1 with base 3. If n is not divisible by 3, then we continue as if we are writing its base 2 representation. It can also be seen that the digits of a given integer can only be 1 in base 2, and *digit* always ends with 1. This will be important in the next section.

Example 5.5. HBT-form of some integers are given below, with the least significant digit on the right:

$$66 = \begin{matrix} [1 & 0 & 1 & 1 & 0 & 0] & \leftarrow & \textit{digit} \\ [2 & 2 & 2 & 2 & 2 & 3] & \leftarrow & \textit{base} \end{matrix} \tag{5.8}$$

$$113 = \begin{matrix} [1 & 0 & 1 & 0 & 0 & 0 & 1] & \leftarrow & \textit{digit} \\ [2 & 3 & 2 & 2 & 2 & 2 & 2] & \leftarrow & \textit{base} \end{matrix} \tag{5.9}$$

$$495 = \begin{matrix} [1 & 0 & 0 & 0 & 1 & 0 & 0] & \leftarrow & \textit{digit} \\ [2 & 3 & 3 & 3 & 2 & 3 & 3] & \leftarrow & \textit{base} \end{matrix} \tag{5.10}$$

Getting back to the integer from a given representation requires some explanation. Starting from the least significant side, each time we get a base we increment the exponent of that base by 1, while the exponent of the other base stays the same. If the digit is k, we multiply the current $2^i 3^j$ with k to get $k \cdot 2^i 3^j$. So if the digit is 1, it is included in the representation. Also, base 3 always starts with 3^1, and 3^0 never appears since 2^0 is already 1.

Example 5.6.

$$\begin{aligned} 66 &= 1 \cdot 2^4 3^1 + 0 \cdot 2^3 3^1 + 1 \cdot 2^2 3^1 + 0 \cdot 2^1 3^1 + 0 \cdot 2^0 3^1 + 0 \cdot 3^1 && (5.11) \\ &= 2^4 3 + 2^2 \\ 113 &= 1 \cdot 2^5 3^1 + 0 \cdot 2^4 3^1 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 && (5.12) \\ &= 2^5 3 + 2^4 + 1 \\ 495 &= 1 \cdot 2^1 3^5 + 0 \cdot 2^0 3^5 + 0 \cdot 2^0 3^4 + 0 \cdot 2^0 3^3 + 1 \cdot 2^0 3^2 + 0 \cdot 3^2 + 0 \cdot 3^1 && (5.13) \\ &= 2 \cdot 3^5 + 3^2 \end{aligned}$$

5.4.2 Exponentiation Using HBTNS

Let n be the integer exponent. Assume n has the Hybrid Binary-Ternary form as such: $digit_n = (d_{m-1}, \cdots, d_1, d_0)$ and $base_n = (b_{m-1}, \cdots, b_1, b_0)$. Then we can compute g^n where $g \in \mathbb{Z}$ using the following algorithm.

Algorithm 10 HBTNS Exponentiation

Input: $g \in Z$, $n \in Z^+$ with the hbt-form $digit_n$ and $base_n$
Output: g^n

1: $r \leftarrow 1$
2: **for** i **from** 0 **to** $m-1$ **do**
3:     **if** $b_i = 3$ **then**
4:         $g \leftarrow g^3$
5:     **else**
6:         **if** $d_i = 1$ **then**
7:             $r \leftarrow r \cdot g$
8:         **end if**
9:         $g \leftarrow g^2$
10:     **end if**
11: **end for**
12: **return** r

The last square is not necessary when $i = m - 1$, since it is not incorporated into r, hence we can omit it.

Using the representations for different n from Section 5.4.1, the following sequence of exponentiations are found for g^n. Chain for r and chain for g are given as well, so that it is easier to follow the algorithm.

Example 5.7.

$$\begin{aligned} n = 66 &: \\ \text{Chain for } g &: g^3 \to g^6 \to g^{12} \to g^{24} \to g^{48} \\ \text{Chain for } r &: g^6 \to g^{18} \to g^{66} \\ \text{Full chain} &: g^3 \to g^6 \to g^{12} \to g^{24} \to g^{48} \to g^{66} \\ \text{Cost} &: 2\mathbf{M} + 4\mathbf{S} + 1\mathbf{C} \end{aligned} \tag{5.14}$$

$$\begin{aligned} n = 113 &: \\ \text{Chain for } g &: g^2 \to g^4 \to g^8 \to g^{16} \to g^{32} \to g^{96} \\ \text{Chain for } r &: g^1 \to g^{17} \to g^{113} \\ \text{Full chain} &: g^1 \to g^2 \to g^4 \to g^8 \to g^{16} \to g^{17} \to g^{32} \to g^{96} \to g^{113} \\ \text{Cost} &: 2\mathbf{M} + 5\mathbf{S} + 1\mathbf{C} \end{aligned} \tag{5.15}$$

$$\begin{aligned} n = 495 &: \\ \text{Chain for } g &: g^3 \to g^9 \to g^{18} \to g^{54} \to g^{162} \to g^{486} \\ \text{Chain for } r &: g^9 \to g^{495} \\ \text{Full chain} &: g^3 \to g^9 \to g^{18} \to g^{54} \to g^{162} \to g^{486} \to g^{495} \\ \text{Cost} &: 1\mathbf{M} + 1\mathbf{S} + 5\mathbf{C} \end{aligned} \tag{5.16}$$

To compare these 3 chains with repeated squaring, we take $\mathbf{C} = \mathbf{S} + \mathbf{M}$. $66 = (1000010)_2$ costs $\mathbf{M} + 6\mathbf{S}$ with the repeated squaring method, which is better than *HBTNS* exponentiation. On the other hand, $113 = (1110001)_2$, and the total cost is $3\mathbf{M} + 6\mathbf{S}$, the same with this method. $495 = (111101111)_2$ is $7\mathbf{M} + 8\mathbf{S}$, much worse than the *HBTNS* method.

The relationship between the binary representation and *HBTNS* is given in [13]. The number of digits an n-bit integer has is $0.8811n$, and the number of multiplications corresponding to the 1's is approximated to $0.3388n$.

5.5 Addition Chains

Introduced by Knuth [25], an addition chain is an integer representation specifically useful for exponentiation. The other algorithms mentioned in this chapter before were general algorithms. For example, the repeated squaring algorithm uses only multiplications and squares. Let us try not to focus on the squares, but also count them as multiplications. Now, finding the binary representation of an integer is very fast. However there is no guarantee that the repeated squaring method is the fastest exponentiation method for a specific integer, with binary representation and using only multiplications (and squares). Using *optimized* addition chains for a certain integer, the exponentiation uses a minimal amount of multiplications.

Definition 5.1. A sequence of integers with $a_0 = 1$ and $a_k = n$

$$a_0, a_1, \cdots, a_k \tag{5.17}$$

such that $a_r = a_i + a_j$ for $0 \leq i \leq j < r$ and $r \leq m$ is called an *addition chain*.

Considering this definition, we can see that the methods we have discussed so far such as the repeated squaring and the window methods are nothing but different algorithms for finding addition chains. However, as said before they obviously do not always find the shortest chain. Moreover, finding the shortest addition chain is an NP-complete [16] problem. What this means is that we have to compute all the chains to find the shortest one, hence it takes considerable time even for small exponents. As the integers get larger, the time it takes to compute the shortest chain is too much, and so using the shortest addition chain is infeasible. Nevertheless, the shortest chain can be found for smaller exponents.

Example 5.8. A short addition chain for 3691.

$$g \to g^2 \to g^3 \to g^5 \to g^7 \to g^{14} \to g^{28} \to g^{56} \to g^{112} \to g^{115} \to \\ g^{230} \to g^{460} \to g^{920} \to g^{1840} \to g^{1845} \to g^{3690} \to g^{3691}.$$

This chain costs $9\mathbf{S} + 7\mathbf{M}$. If we were to use a bit representation and the repeated squaring method, the chain would look like this:

$$g \to g^2 \to g^3 \to g^6 \to g^7 \to g^{14} \to g^{28} \to g^{56} \to g^{57} \to g^{114} \to g^{115} \to \\ g^{230} \to g^{460} \to g^{461} \to g^{922} \to g^{1844} \to g^{1845} \to g^{3690} \to g^{3691}.$$

The bit representation of 3691 is $(111001101011)_2$. Total cost is $11\mathbf{S} + 7\mathbf{M}$, which is worse than the shortest chain.

We have lower and upper bounds for the lengths of shortest addition chains. The upper bound comes from the chain that arises from the repeated squaring method, and the lower bound is found by Schönhage [34].

- An upper bound is $\lfloor \log_2 n \rfloor + H(n) - 1$ where $H(x)$ is the *Hamming weight* function.
- A lower bound is given by $\log_2 n + \log_2 (H(n)) - 2.13$.

In addtion to addition chains, we also have addition-subtraction chains which is very similar. The difference is that we also allow subtractions in the chain, which is equivalent to multiplying with the inverse of an intermediate result. NAF exponentiation can be thought of a way of finding addition-subtraction chains, but it only allows the multiplication with the inverse of the base and not the inverses of intermediate results. Unfortunately, along with the cost of finding the chain, finding the inverses results in an increase in cost. However since we are not using shortest chains with variable exponents, the balance must be on the length of the chain versus the amount of precomputed inverses. Finding shortest chains here is again a problem, but it is not known whether they are NP-hard or not.

5.6 Double-Base Representation

Double-base representation is, in a way, a generalization of most of the representations up to this point. Similar to *HBTNS* we use two bases to represent the digits of an integer, following certain rules. Any two bases can be used in the representation, however we only give the definition for the bases 2 and 3, as they are the focus of our research.

Definition 5.2. A double base representation of an integer n written as

$$n = \sum_{i=1}^{l} d_i 2^{a_i} 3^{b_i},$$

and is called a base-$\{2,3\}$ representation, where $d_i \in S$ for a chosen set S, and $a_1, \cdots, a_l$ and $b_1, \cdots, b_l$ are non-decreasing sequences starting from 0 [1].

If we look at double-bases from an exponentiation viewpoint, we see that the set S designates the multiplications. Relating this with the earlier representations, in base-2 representation, we can only have 0 or 1 as the coefficients so we either multiply or don't. In NAF, we can have multiplications with inverses so the S in NAF is $\{-1, 0, 1\}$.

Representing an exponent in double-base allows an interesting tradeoff between the exponents from respective bases. One base is more costly but *jumps* a larger exponent gap, on the other hand the other is faster but to conquer the same gap it uses more exponents. For example, using the base 2 or base 3 alone would mean either more operations with faster powers or less operations with expensive powers. Using base-$\{2,3\}$ representation we can optimize both of these, if we can find the best representation. Obviously this representation is not unique even for a given S.

To denote the succession of exponentiation, instead of using the representation we use double-base chains. These chains are introduced in [14] by Dimitrov, Imbert, Mishra. Similar to addition chains, they are another way of visualizing the successive exponentiation process. They share the same optimization problem with addition chains as well: finding the best double-base chain for a given integer. There are many parameters factoring into this, such as the chosen bases or the set S. Even for a suitably chosen S and bases, finding the optimized double-base chain takes time. Because of this, they are often researched for elliptic curve cryptography, because the operations there are already costly. Also the speed of computing point multiplication with different bases can be quite similar to finding the chain, allowing for better tradeoffs using this method. Also they are useful in fixed power exponentiation if the cost of computing the exponent warrants the use of finding the best double-base chain.

We use the notation in [1] to denote base-$\{2,3\}$ chains. For an integer g, $(b \cdot x + s)$ appears as $g^b \cdot g^s$ in the operations where b is the base and $s \in S$. The $b \cdot x$ part uses the result from the last operation but s part uses the original g. For example, an exponentiation chain in base-$\{2,3\}$ with $S = \{0,1\}$ for g^{15} can be given by

$$(3x, 2x+1, 2x+1),$$

which is then computed with

$$(x^3) \to (x^6 \to x^7) \to (x^{14} \to x^{15}).$$

Below we give two methods of finding double-base chains. Both of these methods use graphs to find the chains.

5.6.1 Tree Method

The tree method is found by Doche and Habsieger in [15]. As its name, it uses tree graphs. To find a base-$\{2,3\}$ chain with $S = \{+1, -1\}$ for exponent n, they propose the following procedure:

- Choose the root of the tree as n. Start from the root and repeat the next steps at each node.
- Take every 2 and 3 factors out from the node.
- Add $+1$ and -1 to the remaining integer on the node to make two new leaves.
- Repeat until we have a node with value 1.

Using this method, we have two choices. We can either speed up the algorithm with deleting the larger nodes as we continue with the algorithm so that only the small nodes appear, or we can do the algorithm as is. The first approach is faster since you deal with less nodes but it is a greedy approach and may not always give the best chain. The second one however gives an optimal chain [1] but it takes longer to compute the tree and we have to use another algorithm to traverse it. The next section iterates on this.

Example 5.9. Finding a base-$\{2,3\}$ chain with $S = \{+1, -1\}$ of 19, to find g^{19} for some integer g is given in Figure 5.1.

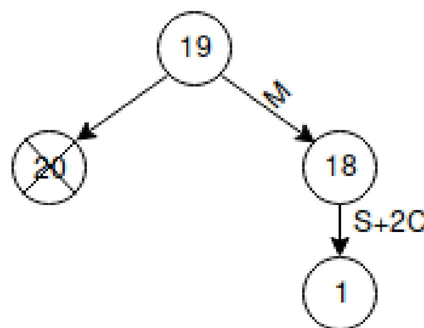

Figure 5.1: Tree graph for g^{19}

The chain can be represented with:

$$g \to g^2 \to g^6 \to g^{18} \to g^{19}.$$

It costs $2\mathbf{C} + \mathbf{S} + \mathbf{M} = 3\mathbf{S} + 3\mathbf{M}$.

5.6.2 DAG Method

In [1] it is proved to that using *directed acyclic graphs* to map out the shortest path between the root node n and the last node 1 is the same as finding an optimized double-base chain for n. They propose three algorithms but we only give the basic one, the one that uses Dijkstra's Algorithm [12] for finding the shortest path in a graph.

The directed acyclic graph D to be used in this algorithm is defined in the following way [1]. Assume the double base we have is base-$\{v, w\}$ and set S with the restriction that $0 \in S$ and representation $(b \cdot x + s)$, $b \in \{v, w\}, s \in S$. First, let there be a vertex for each integer $n \leq N$ for some integer bound N. Join the vertexes for s and n with the edges $vn + s \rightarrow n$ and $wn + s \rightarrow n$, if $vn + s > n$ and $wn + s > n$. Label these edges $(v \cdot x + s)$ and $(w \cdot x + s)$ respectively. Notice that the definition of the graph does not depend on the chosen exponent. It is easy to see that any node n can only reach a finite number of nodes, and it is stated in [1] that this finite number of nodes is at most $\mathcal{O}((logn)^2)$ for any chosen S.

After constructing the graph, we use Dijkstra's Algorithm to find a shortest path between the exponent n to 0 in the graph. Dijkstra's Algorithm searches the *lengths* of every edge and finds the path with the smallest length, and we haven't defined those for the DAG. The way we do that is we first find the cost of each operation in $(b \cdot x + s)$ for all b, s as defined above. For example we assign $S + M$ for $(2 \cdot x + 1)$ and C for $(3 \cdot x)$ and so on. To make the comparisons easier, we use the algorithms from Chapters 3 and 4. If we work with a modulus, the costs at each level depend only on the size of n so that there is no need for additional cost computations for each vertex n.

As we can see, we only make the graph once and for repeated uses of the system its cost can be neglected. However, even though Dijkstra's algorithm is a polynomial time algorithm, the whole algorithm still has a sizeable overhead [1] and makes the DAG method only usable for fixed exponents.

Example 5.10. Finding a base-$\{2, 3\}$ chain with $S = \{+1, -1\}$ of 19, to find g^{19} for some integer g is shown in Figure 5.2. Both the graph starting from 19 and the result of Dijkstra's Algorithm are given. The 0 node is unnecessary hence it is not included.

Assume that $\mathbf{C} = \mathbf{S} + \mathbf{M}$ and $\mathbf{S} < \mathbf{M}$. Then according to the graph, the one of the possible best paths is $(19, 10, 5, 2, 1)$, which costs $4\mathbf{S} + 2\mathbf{M}$. The chain is given by:

$$g \rightarrow g^2 \rightarrow g^4 \rightarrow g^5 \rightarrow g^{10} \rightarrow g^{20} \rightarrow g^{19}$$

Comparing with the tree method, we have given a multiplication in favour of a square. The inversion is precomputed in both cases. Generating the graph and the Dijkstra's Algorithm costs are higher in the DAG case, but the chain is more optimized than the tree case.

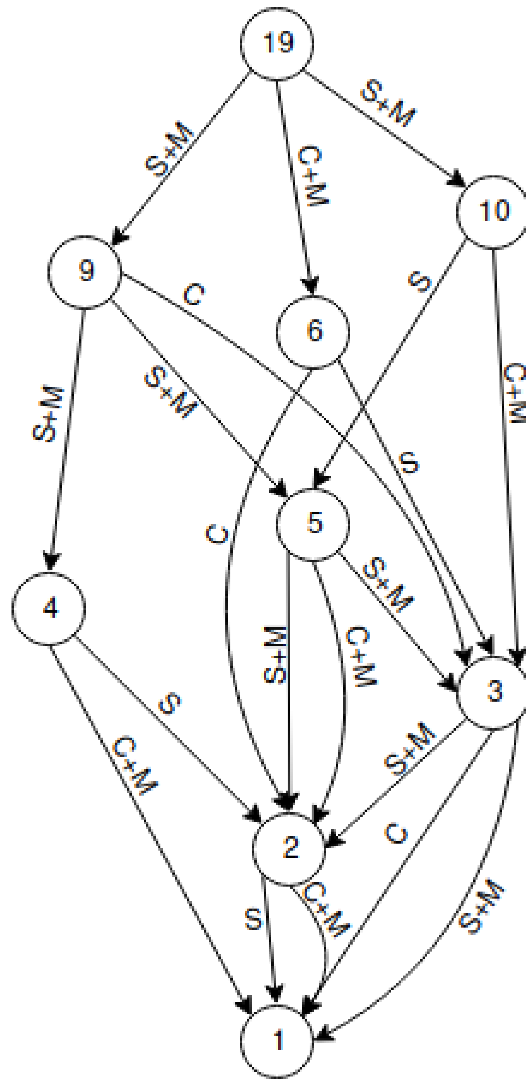

Figure 5.2: DAG for g^{19}

5.7 Implementation Results and Comparisons

We implemented three modular exponentiation algorithms, using both the classical and Zanoni's method for a total of 5 timings. These implementations are done on a machine with Intel i5-6200U, 2.8 GHz and 8GB's RAM. The operating system was Manjaro Linux. Implementations use *C* with the *gmp* big number library version $6.1.2$ for all of the main functions, and *C++* for some auxillary functions and input/output. The compiler used is *gcc* version $7.1.1$. All implementations use only the *mpn_* level *gmp* functions, which are the lowest level and most memory management is done by us and not the library.

Table 5.1: Time comparisons for repeated squaring, cubing and hbtns exponentiation

#-bits	RS	RC	HBTNSE
512	$\mathbf{0.01802 \cdot 10^{-3}}$	$0.01913 \cdot 10^{-3}$	$0.01809 \cdot 10^{-3}$
1024	$0.08719 \cdot 10^{-3}$	$0.09766 \cdot 10^{-3}$	$\mathbf{0.08593 \cdot 10^{-3}}$
2048	$0.50161 \cdot 10^{-3}$	$0.56877 \cdot 10^{-3}$	$\mathbf{0.48400 \cdot 10^{-3}}$
4096	$3.05223 \cdot 10^{-3}$	$3.55163 \cdot 10^{-3}$	$\mathbf{2.97026 \cdot 10^{-3}}$
8192	$18.5382 \cdot 10^{-3}$	$22.0898 \cdot 10^{-3}$	$\mathbf{18.4278 \cdot 10^{-3}}$

The bit sizes are for all of the operands, for example the modulus, the base and the exponent are all 512-bits in the second rows. All operands are generated randomly, and for a single timing, 100 different exponents are used, then each exponentiation is looped 100 times. The times given in every table is in seconds and is a mean of the 100 exponents and loops. The operands' most significant bits are always 1.

Looking at Table 5.1, we see that the repeated cubing algorithm is worse on every bit sizes but *HBTNS* exponentiation quickly catches up to the repeated squaring algorithm, and passes it after around 1024-bits.

Table 5.2: Time comparisons for repeated cubing and hbtns exponentiation using modular Zanoni's cubing method.

#-bits	RC	HBTNSE
512	$0.02838 \cdot 10^{-3}$	$\mathbf{0.02143 \cdot 10^{-3}}$
1024	$0.12103 \cdot 10^{-3}$	$\mathbf{0.09759 \cdot 10^{-3}}$
2048	$0.65238 \cdot 10^{-3}$	$\mathbf{0.51640 \cdot 10^{-3}}$
4096	$3.93786 \cdot 10^{-3}$	$\mathbf{3.12083 \cdot 10^{-3}}$
8192	$24.4370 \cdot 10^{-3}$	$\mathbf{19.0999 \cdot 10^{-3}}$

Table 5.2 shows that hybrid binary-ternary expansion is better than repeated cubing with Zanoni's algorithm as expected. Interestingly, if we compare the hybrid Zanoni's version with the Table 5.1, we can see that starting from 1024-bits, hybrid Zanoni's version is better than the repeated cubing with the classical cubing. This is not enough for using Zanoni's cubing in modular exponentiation, but it shows that the *HBTNS* method is a better method than repeated squaring and cubing both, after some bit size.

CHAPTER 6

CONCLUSION

Public key cryptography algorithms are important and widely used in many areas in both daily life and secure systems. Some of these algorithms heavly use modular exponentiation, such as RSA. As explained in Chapter 2, an analogue of modular exponentiation is used in elliptic curve cryptosystems. The disadvantage of these cryptosystems is that they are slower than their symmetric counterparts. For example, AES [11] algorithm is also used and is faster than RSA. However the key exchange problems and other limitations of symmetric systems make public key systems very attractive. To make them faster, we have to speed up the main operation, which is modular exponentiation.

In this thesis, we investigated methods for exponentiation, both modular and non-modular. Our focus was on the third power, cube, however we studied general methods, squaring and multiplication as well. We aimed to speed up cryptographic algorithms using cubing, hence we studied both variable and fixed exponent methods. We mentioned recent improvements on elliptic curve fixed exponent methods [1].

Before starting with exponentiation, in Chapter 3 we gave some of the multiplication algorithms used both in general exponentiation and one for integer cubing in particular. We computed their costs and compared them.

In Chapter 4 we focused on the squaring and cubing. Squaring comes along with cubing since we cannot compute all exponents with only using power 3, we need squares as well. We also mentioned asymmetric methods that we have came across in our research for cubing. For cubing we focused on Zanoni's algorithm since it was faster than usual methods if computed in a non-modular fashion, and found a way to make it modular. We tried to use asymmetric methods used in squaring. However, asymmetric methods didn't work, and while the modular version worked perfectly, it was slower than usual square and multiply for computing a cube.

The next chapter, we changed our focus to repeated use of the algorithms given in the earlier chapters. We gave the algorithms, complexities, and investigated many of them and their respective representations thoroughly. At the end of the chapter we included our test results as well.

In conclusion, we looked at exponentiation algorithms with cubing in mind, and their repeated use. We took many inspirations from the elliptic curve case, and used hybrid

number systems for integers. We have investigated many algorithms, but our proposed modular Zanoni's cubing algorithm turned out to be slower than the current methods. Hence, it is not useful for cryptography. Nonetheless, the wide research made for this thesis will allow later research on this subject to be more focused. As a future work, using modified versions of hybrid number systems, Zanoni's work and asymmetric squaring, we may research a fifth power algorithm. The flexibility in factorization for fifth power may allow for interesting results.

REFERENCES

[1] D. J. Bernstein, C. Chuengsatiansup, and T. Lange, Double-base scalar multiplication revisited., IACR Cryptology ePrint Archive, 2017, p. 37, 2017.

[2] P. Bh, D. Chandravathi, and P. P. Roja, Encoding and decoding of a message in the implementation of elliptic curve cryptography using koblitz's method, International Journal on Computer Science and Engineering, 2(05), pp. 1904–1907, 2010.

[3] E. Biham and A. Shamir, *Differential cryptanalysis of the data encryption standard*, Springer Science & Business Media, 2012.

[4] D. Bini and V. Pan, Polynomial division and its computational complexity, Journal of Complexity, 2(3), pp. 179–203, 1986.

[5] M. Bodrato and A. Zanoni, Integer and polynomial multiplication: Towards optimal toom-cook matrices, in *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pp. 17–24, ACM, 2007.

[6] M. Bodrato and A. Zanoni, A new algorithm for long integer cube computation with some insight into higher powers, in *Computer Algebra in Scientific Computing*, pp. 34–46, Springer, 2012.

[7] J. Bos and M. Coster, Addition chain heuristics, in *Conference on the Theory and Application of Cryptology*, pp. 400–407, Springer, 1989.

[8] J. Chung and M. A. Hasan, Asymmetric squaring formulae, in *Computer Arithmetic, 2007. ARITH'07. 18th IEEE Symposium on*, pp. 113–122, IEEE, 2007.

[9] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of elliptic and hyperelliptic curve cryptography*, CRC press, 2005.

[10] S. A. Cook and S. O. Aanderaa, On the minimum computation time of functions, Transactions of the American Mathematical Society, 142, pp. 291–314, 1969.

[11] J. Daemen and V. Rijmen, Aes proposal: Rijndael, 1999.

[12] E. W. Dijkstra, A note on two problems in connexion with graphs, Numerische mathematik, 1(1), pp. 269–271, 1959.

[13] V. Dimitrov and T. Cooklev, Two algorithms for modular exponentiation using nonstandard arithmetics, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 78(1), pp. 82–87, 1995.

[14] V. Dimitrov, L. Imbert, and P. K. Mishra, Efficient and secure elliptic curve point multiplication using double-base chains, in *Asiacrypt*, volume 3788, pp. 59–78, Springer, 2005.

[15] C. Doche and L. Habsieger, A tree-based approach for computing double-base chains, Lecture Notes in Computer Science, 5107, pp. 433–446, 2008.

[16] P. Downey, B. Leong, and R. Sethi, Computing sequences with addition chains, SIAM Journal on Computing, 10(3), pp. 638–646, 1981.

[17] J. R. Durbin, *Modern algebra: An introduction*, John Wiley & Sons, 2008.

[18] T. ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, IEEE transactions on information theory, 31(4), pp. 469–472, 1985.

[19] S. D. Galbraith, *Mathematics of public key cryptography*, Cambridge University Press, 2012.

[20] T. Granlund, Gmp 4.1. 2: Gnu multiple precision arithmetic library (2004), URL: http://www. swox. com/gmp.

[21] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*, Springer Science & Business Media, 2006.

[22] J. Jonsson, K. Moriarty, B. Kaliski, and A. Rusch, Pkcs# 1: Rsa cryptography specifications version 2.2, 2016.

[23] B. Kaliski, Twirl and rsa key size, 2003.

[24] A. Karatsuba, Multiplication of multidigit numbers on automata, in *Sov. Phys. Dokl.*, volume 7, pp. 595–596, 1963.

[25] D. E. Knuth, The art of computer programming, 3rd edn. seminumerical algorithms, vol. 2, 1997.

[26] N. Koblitz, Elliptic curve cryptosystems, Mathematics of computation, 48(177), pp. 203–209, 1987.

[27] C. K. Koc, High-speed rsa implementation, Technical report, Technical Report, RSA Laboratories, 1994.

[28] D. Luciano and G. Prichett, Cryptology: From caesar ciphers to public-key cryptosystems, The College Mathematics Journal, 18(1), pp. 2–17, 1987.

[29] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*, CRC press, 1996.

[30] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*, Springer Science & Business Media, 2009.

[31] J. M. Pollard, Theorems on factorization and primality testing, in *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 76, pp. 521–528, Cambridge University Press, 1974.

[32] G. W. Reitwiesner, Binary arithmetic, Advances in computers, 1, pp. 231–308, 1960.

[33] R. L. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Communications of the ACM, 21(2), pp. 120–126, 1978.

[34] A. Schönhage, A lower bound for the length of addition chains, Theoretical Computer Science, 1(1), pp. 1–12, 1975.

[35] R. D. Silverman, A cost-based security analysis of symmetric and asymmetric key length, RSA Laboratories, CryptoBytes, Bulletins, 13, 1999.

[36] S. Singh, *The code book: the science of secrecy from ancient Egypt to quantum cryptography*, Anchor, 2000.

[37] A. L. Toom, The complexity of a scheme of functional elements realizing the multiplication of integers, in *Soviet Mathematics Doklady*, volume 3, pp. 714–716, 1963.

[38] H. C. Williams, A p+ 1 method of factoring, Mathematics of Computation, 39(159), pp. 225–234, 1982.

[39] A. Zanoni, Some toom-cook methods for even long integers, Proceedings of ICTAMI, pp. 807–828, 2009.