Discrete Logarithm Problem in Finite Fields and Applications to Cryptography:

Protocols, Algorithms and Source Codes in C++

by

Divyesh Bhagwanji Chudasama

A thesis submitted in fulfilment of the degree of Master Of Science in Internet Computing and Network Security

Loughborough University

14th September 2012

Copyright 2012 Divyesh Bhagwanji Chudasama

Preface

The word *cryptography* originates from the Greek language from the words *kryptos* meaning "hidden" and *graphein* meaning "to write" and is the study of techniques for safe communication between two parties in the presence of adversaries. *Cryptography* is a field intersecting the disciplines of mathematics, computer science and electrical engineering and deals with issues related to information security, such as: data confidentiality, data integrity and data authentication. More generally, it ensures and overcomes the influence of third parties upon this information, so that they do not understand what information is being transmitted over a public channel, verifies that the information sent is received from the rightful sender and makes sure that during transit, the information has not been tampered with by any unauthorised party. Prior to the current times, cryptography was commendably identical to *encryption*; converting information from a comprehensible form into incomprehensible form and back again at the other end, making it unreadable by adversaries without possessing secret knowledge. History recalls that such techniques were solely used by spies, military leaders, and diplomats, however the earliest forms of cryptography can be seen through the"shift cipher" of Julius Caesar. In the early 20$^{\text{th}}$ century, many mechanical encryption/decryption devices were invented amongst which the most famous is the "Enigma" machine that was used by the German government and military from the late 20's and World War II.

The application of cryptology methods have become gradually extensive since World War I and the introduction of the computer. Modern cryptography deeply relies on mathematical theory and computer science practice; designing cryptographic algorithms based on computational hardness assumptions, so that such algorithms are hard to break in practice by any third party, making them computationally secure. Today cryptography is applied to help protect a variety of everyday tasks, some of which are: transferring money via ATMs, internet banking, transmitting voice, photo and video data over mobile networks, online shopping or electroinc commerce and the use of email, instant messaging and communication techniques via computer for both business and personal discussions.

It wasn't until 1976 that cryptography underwent a revolutionary change with the discovery of *public-key cryptography* by Whitfield Diffie and Martin Hellman, providing a method of public-key agreement [11]. This key exchange technique uses exponentiation in a *finite field* which then was named as Diffie-Hellman key exchange. This method was the first of it's kind, allowing establishment of a shared secret-key via a legitimate communications channel without the need of a prior shared secret. Prior to this discovery, all methods of cryptographic communications required both parties to approve on a secret key beforehand in privacy and to make sure this key stayed hidden from any third parties or else the information would become vulnerable to tampering; this caused huge obstacles when wanting to communicate securely over long distances or when there was no need for preceding understanding. However, public-key cryptography eliminated this need, and let secure communication take place without the need to secretly exchange keys amongst the two parties involved. Thanks to this remarkable discovery, a comprehensive use of application of cryptography is possible today.

Nevertheless, every public-key cryptography case faces a computational problem in relation to it's security, that is, that security is guaranteed until solving this problem is infeasible. Putting aside individual problems, in the general case, in this modern age there are only two key problems behind the security of all widely used public-key cryptography, one belonging to the branch of number theory and the second belonging to the branch of group theory, both from the field of pure mathematics. The first problem is the *integer factorisation problem* and the second is the *discrete logarithm problem* respectively. The focus of this thesis is on public-key systems based on the *discrete logarithm problem.*

In particular, this text will take into account, the current understanding on the discrete logarithm problem over *finite fields*, with knowledge relating also to the more general problem and cryptography as a whole. It is to be noted that the subject will not be presented under full scope due to the limitation on time, but the most imperative examples of discrete logarithms used within public-key cryptography, especially considering the extensive use and prominence of applications, are central to the subject of this text. Further, also presented in this thesis are the problems faced for efficient computation in finite fields, as a result of which effectual implementations of public-key cryptography will be reviewed. Lastly, what follows are the various algorithms for computing discrete logarithms, alongside related issues in finite fields that could hold as potential attack mechanisms towards the cryptosystems that will be discussed.

This thesis is organised into a chapter by chapter structure, the details of which are outlined below:

Chapter 1 is based on an overview of the relevant topics that form the foundation of the subject of this thesis, to aid readers in understanding the arguments that will later be discussed in this text. It explains the concept and fundamentals of group theory and finite fields and outlines their vital importance and basis for the discrete logarithm problem, however, prior acquaintance with the theory of finite fields and related algebraic concepts including groups, rings, polynomials and basic number theory would be of benefit. Further, If undergraduate level knowledge is possessed in the discipline of pure mathematics then one should not face any perplexity in understanding the material discussed in this text.

Chapter 2 is a literature review on the discrete logarithm problem and presents a historical outline of cryptography, its developments, public-key distribution and how the discrete logarithm came about.

The third chapter brings forward the concept of the public key paradigm and explains what discrete logarithms are and the discrete logarithm problem. The chapter also presents: the Diffie-Hellman key exchange protocol and the Elgamal cryptosystem. These two are“cryptographic primitives” that are based on the difficulty of computing discrete logarithms for their security.

Chapter 4 covers information on efficient computation in finite fields. It discusses techniques for expressing finite fields of both prime and prime power order, which will appear in the chapters to follow. Further, the chapter studies a few non-trivial algorithms for carrying out straight forward computational tasks with multiplication and exponentiation. This chapter also looks into the issue of creating irreducible polynomials over finite fields but only briefly as the problem is wholly related to one of the given field representations. The information reviewed in this chapter links to the material from its previous chapter on the implementation of both the cryptographic primitives and the following chapter analyses the different attacks on these primitives.

Chapter 5 reviews and justifies various discrete logarithm algorithms that relate to subjective finite cycle groups and can be used to attack the primitives outlined in Chapter 3 of this text. The chapter is split into three sections; the first section presents the various different procedures used by the algorithms discussed, the second is based on “generic discrete logarithm algorithms” whilst the final section explores “index-calculus” algorithms, also known as non-generic discrete logarithm algorithms.

Chapter 6 explores the implementation of algorithms discussed in Chapter 5. It discusses the programming language used, the sources of the pseudocode used, explains the functionality of the code, looks into the robustness and efficiency of the implementations and discusses the experimental outcomes.

The final chapter summarises the text into a conclusion, and reviews improvements and additional work that could be a possible scope of future work, were the project to be done again.

Declaration of Authorship

I hereby declare that I wrote this Master Thesis solely by myself and used no resources other than those cited.

Friday 14th September 2012

Divyesh B. Chudasama

Acknowledgements

To begin with, I would like to sincerely and whole-heartedly thank Loughborough University for accepting me onto their Masters Programme, for which reason this thesis exists today.

I would secondly like to express my greatest gratitude to Dr. Ana.M.Salagean, my project supervisor whose repeated advise, guidance, patience and exceeding insights are the purpose for this thesis being possible.

Next, a huge appreciation to my friends, Nasreen Jabbar, Abhishek Sunil, Abbas Camp and Samuel Simpson for their ever grateful support throughout this thesis. Nasreen proofread a draft copy of this thesis for mathematical, spelling and grammatical errors for which I am obliged to her. Abhishek confirmed many of the mathematical calculations presented, offered beneficial propositions for improvement and assisted in the testing stage of the algorithms by verifying the different outputs obtained. I thank Samuel and Abbas for providing ever helpful advise and tips on program design and efficiency without which my implementations would have lacked their robustness. Their feedback and assitance was greatly helpful and is highly valued.

Last but not least, I am forever indebted to my parents, without who's unconditional love, care, upbringing and support I would not today be where I am. They inspired me with the passion to learn. It is to them and my lovely sister Bhavini that I dedicate this thesis.

Abstract

Discrete Logarithms in finite fields have proven to provide grave importance and significance within the cryptographic field. It is such that given a primitive element α of a finite field $\mathbb{F}_n$, the discrete logarithm of a nonzero element $\beta \in \mathbb{F}_n$ is the integer x, for which *1* $\leq$ x $\leq$ *n-1*, where $\beta = \alpha^x$. Had an efficient discrete logarithm been discovered, then many of today's cryptographic systems would have been insecure for making this problem central to the scope of research over recent years. This text critically surveys the known algorithms within this area, as well as reviewing past works, cryptographical developments and related knowledge to the *discrete logarithm problem.* It has been discovered from past reasearch that discrete logarithms in the fields $\mathbb{F}_p$ (where p is a prime) carry a much easier implementation level than that of $\mathbb{F}_{2^n}$. Therefore it can be said that the fields $\mathbb{F}_{2^n}$ are to be avoided in all cryptographic applications as fields $\mathbb{F}_p$ appear to provide a higher and increased level of security. Nevertheless, the focus of this thesis will be on the fields $\mathbb{F}_{2^n}$.

Notation

Symbol	Interpretation
$\mathbb{N}$	Set of *natural numbers*, $\{1,2,3,...\}$.
$\mathbb{Z}$	Set of *integers*, $\{0, \pm1, \pm2, \pm3, ...\}$.
$\mathbb{Z}_n$	Ring of *integers* modulo n, $\{0,1,2,...,n\text{-}1\}$.
$\mathbb{Z}_p^*/\mathbb{F}_p^*$	Multiplicative group of integers of units modulo p.
R^*	The multiplicative group of the ring R.
$R[x]$	The ring of polynomials over the ring R, in the indeterminate x.
$\gcd(f(x),g(x))$	Unique monic polynomial of greatest degree dividing $f(x)$ and $g(x)$.
$\langle\alpha\rangle$	Ideal generated by the ring element α.
$\square$	End of proof.

Pseudocode

Throughout this text, various algorithms will be presented in a ficticious programming manner, meaning that the operations of the algorithms are clearly evident and does not include the detailed and complicated syntax that would otherwise be seen in a real programming language. Nevertheless, we would like to mention here, that the pseudocode will be provided for those algorithms which will be both researched and implemented as one of the key tasks of this project requires the implementation of either one algorithm in great depth or program a set quantity of the discrete logarithm algorithms.

An attempt has been made herewith, such that even an individual not possessing any knowledge or familiar with programming languages should be able to understand the pseudocode presented without any difficulty. Before proceeding further, we would like to note the following; for the pseudocode:

- The Algorithm arguments are presented as **Require**, clearly seperated by commas.
- $\leftarrow$ is used to assign variables.
- Bold font is used to outline control statements and these act equivalent to as they do in a real programming language. Some of the control statements seen will be: **Do, For, If, Return, Unconditionally** and **While**.

Contents

Chapter 1

Overview of Relevant Topics

As of the discrete nature of the problem presented in this thesis, nearly all the work will be restrained to the mathematics of group theory. For this reason, it is required that readers of this paper possess knowledge and are common to the concepts of group theory and the fundamentals on which it is built. Though it would be of benefit, it is not required that one is also well versed with an understanding of field theory in order to follow the arguments presented in this text.

1.1 Groups

This section outlines an overview of the basic algebraic objects and their properties that are required to be known for full understanding of this text.

1.1.1 Definition

A *group* is an ordered pair *(G, $*$)* such that G is a set, $*$ is an associative binary operation on G and $\exists e \in G$ such that:

1. if a$\in$G, then a$*$e=a.
2. if a$\in$G, then $\exists$a$^{-1}\in$G such that a$*$a^{-1}=e.

If not misunderstood, then it is sufficient to denote the group *(G, $*$)* by G and the element $a*b$ by ab. At times the operation $*$ is denoted by $+$ and called addition.[4]

This thesis will not look closely into the properties with depth as the text is not a study of group theory, however the information provided in the chapter will suffice for readers to follow and understand the remaining arguments of this text.

Note 1: The element e is known as the identity of G, and the element a^{-1} is called the inverse of a.

1.2 Finite Fields

Finite fields are determined from the fundamental theorem of Galois Theory and are also known by the name Galois Fields. Galois Theory is amongst one of the most central and vital subjects in mathematics.

A brief summary; there are two types of finite fields, $\mathbb{F}_p$ and $\mathbb{F}_{p^n}$, the former dealing with integer values with operation modulo p whilst the latter deals with irreducible (primitive) polynomials of degree n, our focus being on the latter type. We begin by stating basic properties and facts about finite fields.

1.2.1 Basic Properties

Definition 1. *A finite field is a field* $\mathbb{F}$ *which contains a finite number of elements. The order of* $\mathbb{F}$ *is the number of elements in* $\mathbb{F}$.

Fact 1 (*uniqueness and existence of finite fields*)

1. Consider $\mathbb{F}$ to be a finite field, then $\mathbb{F}$ contains p^n elements for some prime p and integer $n \geq 1$.

2. Every prime power order p^m, has a unique (up to isomorphism) finite field of order p^n. This field is denoted as $\mathbb{F}_{p^n}$.

Speaking informally, *isomorphic* means two finite fields are the same in structure, however their field elements may be represented differently. We here note that if p is a prime then $\mathbb{F}_p$ is a field, and therefore every field of order p is isomorphic to $\mathbb{F}_p$. Only if otherwise stated, the finite field $\mathbb{F}_p$ will be identified with $\mathbb{Z}_p$.

Fact 2 If $\mathbb{F}_q$ is a finite field of order q where $q = p^n$, p a prime, then the characteristic of $\mathbb{F}_q$ is p. Further, $\mathbb{F}_p$ contains a copy of $\mathbb{Z}_p$ as a subfield. Therefore, $\mathbb{F}_q$ can be seen as an extension field of $\mathbb{Z}_p$ of degree n.

Fact 3 (*subfields of a finite field*) Let $\mathbb{F}_q$ be a finite field of order $q = p^n$. Then every subfield of $\mathbb{F}_q$ has order p^m, for some m which has a positive divisor n. Equally , if m is a positive divisor of n, then only one subfield of $\mathbb{F}_q$ of order p^m; an element $a \in \mathbb{F}_q$ belongs to the subfield $\mathbb{F}_{p^m}$ iff $a^{pm} = a$.

Definition 2. *All non-zero elements of $\mathbb{F}_q$ form a group called the multiplicative group of $\mathbb{F}_q$, denoted $\mathbb{Z}_q^*$, in which the only operation is multiplication.*

Fact 4 $\mathbb{Z}_q^*$ is a cyclic group of order q -1. Hence $a^q = a$ for all $a \in \mathbb{F}_q$.

Definition 3. *A generator of the cyclic group $\mathbb{Z}_q^*$ is also known as a primitive element or generator of $\mathbb{F}_q$*

Fact 5 If $a, b \in \mathbb{F}_q$, a finite field of characteristic p, then

$$(a+b)^{p^t} = a^{p^t} + b^{p^t} \text{ for all } t \geq 0.$$

1.3 Construction of $\mathbb{F}_{p^n}$

Definition 4. *The field $\mathbb{F}_{p^n}$ is constructed as $\mathbb{Z}_p[x]/<f>$, where f is an irreducible (primitive) polynomial of degree n in $\mathbb{Z}_p[x]$.*

Alternatively $\mathbb{F}_{p^n}$ is the field $\mathbb{Z}_p[\alpha]$, where α is a root of an irreducible (primitive) polynomial of degree n.

Theorem 1. *If f is primitive then α generates the cyclic group $\mathbb{F}_q^*$.*

We can reperesent the elements of $\mathbb{F}_{p^n}$ in two ways:

1. $\{a_0 + a_1\alpha + ... + a_{n-1}\alpha \mid \mathrm{a}_i \in \mathbb{Z}_p\}$
2. $\{0, 1, \alpha, \alpha^2, .. , \alpha^{p^n-2}\}$

The Discrete Logarithm Problem converts from the representation shown in number 1 to the representation shown in number 2.

Chapter 2

Literature Review

2.1 Cryptographic Development

Cryptography has passed through a revolution over the past three decades. It wasn't until the mid 1970's that cryptography faced the need to develop cryptographic systems, which reduced the need of secure key distribution channels and provide an alternative for a written signature due to the development in hardware technology. Juxtaposition, growth in information theory and computer science lead to probabilistic aspirations of secure cryptosystems that transformed this art into a science.

Following this transformation, telecommunications soon overtook most mail with the expansion of computer controlled communication networks, bridging the communication gap across countries. This however, posed many such applications to become vulnerable to eavesdropping and injection of illegitimate messages, opening up the need for a security solution within the cryptographic field.

Nevertheless, since it's ancient roots, until recently, the most well known problem in cryptography was that of privacy: stopping the retrieval of information from an unauthorised entity over an insecure channel. For cryptographic security, it was required however, a key be shared amongst both communication parties inclusively, without any third party sharing knowledge of it. This used to be achieved by exchange of the key over the means of a secure channel like a private courier or registered mail, in the case of two individual entities such as Alice and Bob involved in one-to-one interaction. For business however, it was common practice to communicate without prior acquaintance as holding back business contact till the physical transfer of the key was simply impractical, unprofessional and unrealistic. The key distribution problem posed a great barrier in the transfer of business communications to large teleprocessing networks due to the cost and delay it imposed.

2.2 Public-Key Distribution

It wasn't until later, with the proposition of public key distribution systems that the requirement for a secure key distribution channel overcome. This proposition asks of a system and two users wishing to communicate. The system works as such that the users interact back and forth till they both reach a common key; eavesdropping by a third party would make computing the key computationally infeasible for the subject. Not only key distribution, but business communications also raises a question on authentication, the validation of which was given via signatures on a written agreement as proof of the acceptance of the contract. A *public-key distribution* works in the same manner, however the use of signatures for such a system involves the transfer and storage of written contracts. For a digital replacement for the paper system, every user had to produce a message the authenticity of which can be verified by any other person, however not produced by anyone else, not even the recipient. As the messages can only be originated by one person but likewise received by many, it is classified as a broadcast cipher.

Juxtaposed the increase in new cryptographic problems through the development in communications and computation, it's child divisions, information theory and theory of computation began to provide tools for the answer to the crucial classical crypto graphical problem.

Since the ancient times, one of the oldest themes of cryptographic research has been one of unbreakable codes. This wasn't until recently, as all such proposed systems have been broken, but earlier this century, back in the nineteen twenties, the "one time pad" scheme was invented and shown and still is mathematically unbreakable. Even having considered the current and future computation powers, there is no way to break it, purely because it is mathematically impossible. We note that one-time pads are not of central interest to this text, but if the reader wishes to gain insight knowledge upon this subject then please refer to [11].

As opposed to crypto graphical developments, the security of the majority of current cryptographic systems is based on the level of computational difficulty a cryptanalyst faces of being able to find the plaintext without any knowledge of the key. This particular challenge remained within the domains of computational complexity and algorithmic analysis; two such disciplines that explore the difficulty levels of solving computational problems.

2.3 Computing Logarithms

In light of what has been discussed so far in this chapter, Diffie and Hellman introduced a technique that uses the concept of the difficulty of computing logarithms over a finite field $\mathbb{F}_p$ [5] and is known as Galios' Field. Despite their proposal involving the finite field $\mathbb{F}_p$, it has since been adapted and used for the finite field $\mathbb{F}_{2^n}$ because of its ease of implementation. We take interest in the finite filed $\mathbb{F}_{2^n}$ for this text and subsequent arguments will follow, but first let us here explain the logarithmic technique presented by Diffie and Hellman in their invited paper in 1976 [11].

The computation works as such; for the finite field $\mathbb{F}_p$ with a prime number p of elements, allow

$$Y = \alpha^X \text{ mod } p, \text{ for } 1 \leq X \leq p\text{-}1$$

in which α is a primitive element of $\mathbb{F}_p$ that is fixed, X is known to be the logarithm of Y to the base α mod p, mathematically denoted:

$$X = \log_\alpha Y \text{mod} p, \text{ for } 1 \leq Y \leq p\text{-}1$$

It is easy and straightforward to calculate Y from X, considering $2{\times}\log_2 p$ multiplications to be the calculation. To state and example, like $X = 18$,

$$\text{Y} = \alpha^{18} = (((\alpha^2)^2)^2)^2 \times \alpha^2$$

Though this is straightforward, doing the opposite operation of finding X from Y is of higher difficulty, especially for certain values of p, for which case the most appropriate algorithm is to be determined.

Yet as difficult a method this may prove to be it's security very heavily depends on the hardness of computing logarithms mod p. Were an algortihm with growth $\log_2$q to be found then this system would be broken. The Diffie-Hellman key exchange method is discussed later in Chapter 3 of this thesis for now we are to continue reviewing related literature. The best p chosen is $p^{1/2}$ being the best measure of the difficulty of the problem and a good common algorithm for using logs and mod$p$, alongside being quite near optimal. A number is chosen uniformly, which users have produced from integers {1,2, ...}, keeps X_i secret, but places

$$Y_i = \alpha^{X_i} \text{mod } p$$

To cite an example of breaking such a system with a known log mod q, consider a public file holding names and addresses of individuals from one particular neighbourhood, when private communication is then used by i users, the following therefore becomes their key,

$$K_{ij} = \alpha^{X_i X_j} \text{mod } p$$

K_{ij} is acquired by user i and this is done by acquiring Y_j as of the public file. Let

$$\begin{aligned} K_{ij} &= Y_j^{X_i} \text{mod } p \\ &= (\alpha^{X_j})^{X_i} \text{mod } p \\ &= \alpha^{X_j X_i} \\ &= \alpha^{X_j X_i} \text{mod } p \end{aligned}$$

User j obtains K_{ij} in the similar fashion

$$K_{ij} = Y_i^{X_j} \text{mod } p$$

Another user must compute K_{ij} from Y_i and Y_j, for example, by computing

$$K_{ij} = Y_i^{(log_\alpha Y_j)} \text{mod } p$$

Hence, It is evident that the system can break without difficulty if the log mod p is known. Although, there is no evidence that the system is secure if logs mod p are not easy to compute, and X_i or X_j always have to be computed before K_{ij} can be computed from Y_i and Y_j.

If 2^b is a little more than p (which is a prime), thereafter all quantities are seen as b bit numbers. Exponentiation then takes at most $2b$ multiplications mod p, while by hypothesis, taking logs requires $p^{1/2} = 2^1/2$ operations. As a result, the crypt analytical effort exponentially grows with rightful efforts; to obtain X_i to Y_i, or Y_i and X_j to K_{ij} when b=200, then a maximum of 400 multiplications are needed, as well as logs mod$p$, which need 2^{100} or roughly 10^{30} operations.

Continuing with computation we now consider our field of interest $\mathbb{F}_{2^n}$ in which computations are understood under the following context:

- A specific irreducible polynomial is $f(x)$ is chosen which is of order n over the field $\mathbb{F}_2$. $C\in$ $\mathbb{F}_2$ is polynomial C(x) over $\mathbb{F}_2$, understood as mod $p(x)$, and the field contains a maximum of 2^n elements.

- Addition and multiplication are understood as polynomial addition and multiplication over $\mathbb{F}_2$mod $(p(x))$.

It is required however, that the polynomial $f(x)$ is *primitive*; as m lies in the integer range of 0,1,2,...,2^n - 2, the elements A(x)$\equiv$ x^{m}mod*f(x)* of the field take each nonzero element of $\mathbb{F}$ once and only once. Further, the nonzero, element A(x) and integer m are related so that m is referred to as the *logarithm* of *A(x)*. As x^{2^n-1} $\equiv$1mod *p(x)*, this logarithm can therefore be defined as 2^n - 1 meaning the logarithm m is considered to be in the scope of integers mod 2^n - 1, (also called the ring $Z/(2^n$ - 1)). It was this calculation of m that gave birth to the subject of this text as we know it today, and was named the *discrete logarithm problem.* It is advised, that if one so desires to gain a deeper insight into logarithms in fields of characteristic two then please refer to [12].

2.3.1 Computational Complexity Issues

The main question of security, 'What is a possible computation?' derives from the consideration of security of cryptosystems against computational attacks. In mathematics, this is called computational complexity theory and this section consists of the complexity issues along with the discrete logarithm problem.

Though chapter 5 of this thesis explores the problem in detail we here consider the discrete logarithm problem under the context of the *Compiutational Complexity Issues.* The discrete logarithm problem for the group G may be stated as:

"Given $\alpha \in G$ and $\beta \in g$, find an integer x such that $\alpha^x = \beta$.

The statement of the problem above is the primary aspect that needs to be considered, yet, firstly, there is logic to discuss the discrete logarithm problem in an arbitrary semigroup; however, in many cases the curiosity is in precise examples of finite cyclic groups; the above statement has precisely eliminated that β may not possibly be within the cyclic group generated by α. There is another way to state the problem:

"Given that $\beta, \alpha \in G$, determine if there exists an integer x such that $\alpha^x = \beta$, and if so, find such an x.

Nevertheless, more difficulty could be faced in solving this version of the problem, so, in order to resolve the discrete logarithm problem, analysing an algorithm could be utilised and as to go ahead with the analysis, we should be aware that $\beta \in g$. From now on, the assumption will be made that the group G is cyclic, with $G = g$, so the initial statement of the problem can be used.

The statement "given the group G should be defined in an unambiguous way. There could be many abstract ways that groups could occur; the following gives a few examples:

- The imaginary quadratic fields class group, which is received by the discriminant
- An elliptic curve over a finite field or rational numbers, where a group of points occur, received by the equation of the curve and the specification of the field

- Galois group of a polynomial
- Finite Albelian group which is received by invarian factors or generators and relations.

The following assumptions need to be made for the group G:

- Efficient algorithms for testing equality of group elements are known
- An efficient algorithm for multipling any two elements of the group is known
- The group is finite, generated by α, with known order n.

The classification of computational problems is the aim of the field of computational complexity whilst corresponding the level of difficulty. An example of the discrete logarithm algorithm is a random polynomial time reduction between two computational issues, which could be used to determine the order of the base for the logarithm. This type of reduction can be seen as a partial ordering which is induced from the computational issues depending on the complexity dealt with, and valuable data could be given on the intrinsic complexity of the issue. In this particular situation, it could be hard to determine the discrete logarithms because it is initially used to determine the order of that base in the group. We will not consider the analysis and evaluation of the various discrete logarithm algorithms and their applicable fields in the literature review as Chapter 5 of this text is dedicated to this very topic.

2.3.2 Reduction

Firstly we have to consider a few simple methods, which will decrease a discrete logarithm problem in a cyclic group G to discrete logarithm problems in different subgroups. In order to start, it is essential to mention all the rules involved, yet, it is not too difficult to multiply and take inverses in G, which is required in this case. There could be a problem in some cases where there lies a possibility that two elements could be equal, for example, suppose the group is viewed as a structure with proportions, or there maybe a group of binary quadratic forms. However, the following two assumptions should always be made while carrying out a group operation; the cost of deciding if the two elements of G are equal is of the identical magnitude and the group elements could have allocated symbols so that they could be kept.

2.3.2.1 The first reduction

Suppose the cyclic group G has the order n and n is factored significantly as $n = u \times v$, where u and v are coprime, and thus gcd (u, v) =1. So now the issue of solving a discrete logarithm in G is decreased into solving for order u and v of the discrete logarithm in the subgroup of G. If we consider the case where $G =$ g, then g^u generates the subgroup of u^{th} powers in G, of order v, and in the same way g^v generates the subgroup of v^{th} powers, with order u. Solving the discrete logarithm l_u, l_v, where

$$(g^u)^{l_u} = t^u$$
$$(g^v)^{l_v} = t^v$$

By performing repeated squaring one can solve the powers g^u, g^u, t^u, t^v. (Buhler and Wagon 2008)

Adopting the extended Euclidean algorithm may discover the integers a, b, by doing $a \times u + b \times v = 1$. So:

$$t = t^{a_u + b_v} = (t^u)^a (t^v)^b = g^{ul_u a} g^{vl_v b} = g^{aul_u + bvl_v}$$

and we come to the conclusion that the discrete logarithm of t is $aul_u + bvl_v$.

2.3.2.2 The next reduction

This reduction assumes that the order is a prime power of G, so for example, p^a, where $a > 1$. The discrete logarithm issue of order p in the cyclic subgroup of G could have been reduced to a from the discrete logarithm in the group. Suppose we need to discover the l from $g^l = t$, in the base p, if we add in the l, where l is satisfied for a smallest positive value, then

$$l = b_0 + b_1 p + ... + b_{a-1} p^{a-1},$$

For each b_j being an integer in [0, p-1], b_0, b_1, b_{p-1} should be discovered in order. Consider the following:

$$t^{p^{a-1}} = (g^l)^{p^{a-1}} = g^{lp^{a-1}} = g^{b_0 p^{a-1}} = (g^{p^{a-1}})^{b_0},$$

The p^{a-1} powers which are generated by $g^{p^{a-1}}$ in the cyclic subgroup, b_0 is the resolution of the discrete logarithm problem. Assume the calculation for $b_0 b_{j-1}$, has already been done. Note that:

$$t_j = tg^{-b_0 - b_1 p - ... - b_{j-1} p^{j-1}}.$$

It can be observed that t^j is a power of p^j, and therefore $t_j^{p^{a-j-1}}$ is in the group with powers of p^{a-1}. The discrete logarithm issue for b_j needs to be solved.

$$t_j^{p^{a-j-1}} = (g^{p^{a-1}})_j^b$$

we now need to look for the base p digit of l.

To demonstrate such reduction, we need a help of an example. Consider an example where $g = 11$, $t = 17$ in $(\mathbb{Z}/101\mathbb{Z})$*. It is sufficient to solve two discrete logarithm issues for every groups of order 2 and 5 as 100 is the order of the group. So $17^{25} \equiv$ -1(mod 101), order 4 in the subgroup, the element 17^{25} should have 2 as their discrete logarithm i.e. $\mathrm{l}_{25} = 2$. The two problems have now been solved for the order 2, the former being 0 and the latter therefore being $\mathrm{l}_{25} = 0 + 1.2 = 2$).

We should now find l_4 for $11^{l_4} = 17^4 = 95$(mod 101), the order of 25 is being solved for the discrete logarithm in a cyclic group. The discrete logarithm calculation had been decreased to two in a group of order 5. Firstly we have to calculate

$$17^20 = 95^5 = 1(\text{mod } 101),$$

this concludes that l_4 is a multiple of 5. In addition, $11^{20} = 87$(mod 101) and for the 95 modulo 101, we need to look for a power which corresponds to 87, so the solution is 0,1,2,3 or 4. It is obviously not 0 or 1 hence experiment with the number 2 yields the correct answer. So

$$\mathrm{l}_4 = 0 + 2 \times 5 = 10.$$

Next we consider

$$(-6) \times 4 + 1 \times 25 = 25,$$

therefore

$$(-6) \times 4 \times 10 + 1 \times 25 \times 2 = -190,$$

this is the smallest positive discrete logarithm for 17 $\times$ 10.

We can now control the group order, however, this may not be the situation most of the time, so we should decrease the problem to a smaller situation as illustrated just now in the above example. This could be done by running a factorisation algorithm on the order because some discrete algorithms are in general difficult in practice than factoring. Smaller situations ease the level of calculation, unless in some cases where one has to work on subgroups equally as hard as working within the full group.

2.4 Historical Perspective

In the past the growth of cryptography has not indicated public key systems and a one-way authentication system, but it can be seen as a natural growth of development in cryptography going back to hundreds of years.

As already mentioned, an important aspect of cryptography is privacy; an aspect that was uncertain during the first stages of cryptography. An example of this could include the Caesar Cipher; when a letter is substituted by the third place, for example, A is carried to D and B is carried to E and so on, the Caesar cipher cryptosystem relies on the security by ensuring the whole process of encryption is kept secret.

The general system had to negotiate when the telegraph was generated, this happened because of the difference between the general system and a precise key. An example of this is theft of crypto graphical devices, with no negotiation of upcoming encrypted messages in new keys. Kerchoffs codified this theory, where in 1881 it was stated that any correspondents should hesitate with the negotiation of a cryptography system. Around 1960 however, keeping messages a secret was out of place as cryptosystems came across and therefore it was quite strong to tolerate a common plaintext cryptanalytic attack. Such enlargement reduced the part of the system that had to be kept secret from the public, by removing dreary convenients rephrasing diplomatic dispatches. In order to decrease secrecy, public key systems are sustained and there were restrictions to computations in cryptography systems before this century. These computations were done by either hand or normal slide rule like devices.

The actual revolutionary commenced just after World War 1 in which extraordinary machines were built for encrypting. Cryptography was restricted to operations that could be done with easy electromechanical systems, till there was expansion of normal purpose digital hardware and the expansion of digital computers has untied it from the restrictions of computing with gears and the search is authorized for better types of encryption to only cryptographic principle.

Chapter 3

Discrete Logarithms and Cryptosystems

This chapter will begin by discussing discrete logarithms in finite fields, stating associated properties and specifying mathematical characteristics. The chapter then follows by the introduction of public-key cryptography and outlines two current algorithms established on the discrete logarithm problem.

3.1 Discrete Logarithms in Finite Fields

The key interest of this text dwells around the subject of discrete logarithms in the finite field q, where q=2^n. The algorithms presented, are defined in the generic setting of a finite cyclic group G (multiplicatively written) of order n, with generator α, also known as the primitive element α of the multiplicative group $\mathbb{F}_p$. To take a more tangible approach, it can be advised that it is suitable for the reader to think G to be the multiplicative group $\mathbb{F}_p^*$ of order (p-1), the group operation of which is multiplication modulo p.

Definition 5. *Let G be a finite cyclic group of order n, α is a generator of G, and $\beta \in G$. The discrete logartihm problem of β to the base α, written $log_\alpha \beta$, is the distinct integer x, $0 \leq x \leq n\text{-}1$, such that β=α^x.*

As mentioned above it may help the reader to understand the multiplicative group $\mathbb{Z}_p^*$ = [1,...,p-1] as a concrete instance of the finite cyclic group G. So:

$$\mathbf{log}_\alpha\mathbf{:}\ \mathbb{Z}_p^* \longrightarrow \mathbb{Z}_{p-1}$$

such that the discrete logarithm problem converts integer multiplication of modulo a prime p into integer addition of modulo p-1. To aid understanding, an example is presented below.

Example 1. *Let p=29, then G:=$\mathbb{Z}_{29}^*$ is a finite cyclic group of order n=28. A generator of $\mathbb{Z}_{29}^*$ is α= 2 and β = 17. The discrete logarithm holds as a unique integer x such that 0 $\leq x \leq$ 28 and β=α^x.*

This gives $17 = 2^x$ so the discrete logarithm is:

$$x = \log_\alpha 17 = \log_2 17$$

Now

$$\alpha^x = 17 \text{mod } p$$
$$\Rightarrow \alpha^x = 17 \text{mod } 29$$
$$\Rightarrow 2^x = 17 \text{mod } 29$$

Since $2^{21} = 2097152$, element β will take the value **17** as:

$$17(\text{mod}29) = \text{x} = \log_2(17) = 21$$

Meaning that the **discrete logarithm is**:

$$\text{x} = \log_2(17) = 21$$

Now when $x = 21$ it holds that:

$$2^{21} = 17\text{mod } 29$$

The following steps outline why β takes the value of 17:

1. $2^{21} = 2097152$
2. 2097152mod 29 = 72315.58621.
3. The integer value 72315 is then taken as the quotient.
4. Now 72315 x 29 = 2097135.
5. Finally, we now calculate the difference between the answers from step 1 and step 4, 2097152 - 2097135 = 17
6. Hence, $\beta = 17$.

This proves that $\beta = 17$ is a sufficient element of G and that $x = 21$ is the unique integer for $0 \leq x \leq n$-1, such that $\beta = \alpha^x$. Before progressing further, it is important to state some elementary facts about logarithms.

Fact 6 Let α be a generator of a cyclic group G of order n and let $\beta, \alpha \in G$. Let s be an integer. Then:

- $\log_\alpha(\beta\gamma) = (\log_\alpha\beta + \log_\alpha\gamma)\bmod n$
- $\log_\alpha(\beta^s) = s\log_\alpha\beta \bmod n$
- $\log_\alpha(\frac{\beta}{\gamma}) = (\log_\alpha\beta - \log_\alpha\gamma)\bmod n$

In cryptography, the groups of most interest are multiplicative group $\mathbb{Z}_p^*$ of the finite field $\mathbb{F}_p$, especially the cases of the multiplicative group $\mathbb{Z}_p^*$ of the integers modulo a prime p, and the multiplicative group $\mathbb{Z}_{2^n}^*$ of the finite field $\mathbb{F}_{2^n}$ of characteristic two. Now we continue on to defining the discrete logarithm problem for both of these fields.

3.1.1 The Discrete Logarithm Problem in $\mathbb{F}_p$

Definition 6. *The discrete logarithm problem (DLP) in $\mathbb{F}_p$ is the following: given a cyclic group G, a prime p, a generator α of G and an element $\beta \in G$, find the integer x, $0 \leq x \leq p$-1, such that α^x=β(mod)p.*

3.1.2 The Discrete Logarithm Problem in $\mathbb{F}_{2^n}$

Definition 7. *The discrete logarithm problem (DLP) in $\mathbb{F}_{2^n}$ is the following: given an irreducible (primitive) polynomial $f(x)$ of degree n, a cyclic group $G = \mathbb{Z}_{2^n}^*$, a generator α, and a primitive element $\beta \in G$, find the integer x, $0 \leq x \leq 2^n$-1, such that α^x=β.*

Note 2: *Solving the discrete logarithm problem in a finite cyclic group G of order n is actually computing an isomorphism between G and $\mathbb{Z}_n$.* Any two cyclic groups that are isomorphic means that they have the same structure though the elements within them can be written in differing representations. Nevertheless, having kept this property in mind it does not follow that having found an efficient algorithm for computing logarithms in one group will also lead to an efficient algorithm for the second group. To understand this, take it that every cyclic group of order n is isomorphic to the additive group of order $\mathbb{Z}_n$, i.e. the integers {0,1,2,...,n-1} with group operation addition modulo n. Moreover, in the latter group, the problem of finding a distinctive integer x satisfying $ax \equiv$ *b(mod n)* such that *a,b*$\in\mathbb{Z}_n$, is as straight forward as demonstrated in the following. Firstly, it is important to note that there is no solution x if *d=(a,n)* is not divisible by b. Or else, if d is divisible by b, then it is possible to use the extended Euclidean algorithm to determine integers s and t, such that *as+nt=d.* Now, multiplying both sides of this equation by integer $(\frac{b}{d})$ will give: $a(\frac{sb}{d}) + n(\frac{tb}{d})$ = b and reducing by modulo n holds $a(\frac{sb}{d})\equiv b \bmod n$. Therefore $x = (\frac{sb}{d}) \bmod n$ is the easily obtainable and needed solution.

The algorithms for the DLP are outlined in detail in Chapter 5 of this text, however in brief can be categorised in the following way:

1. Algorithms that work in arbitrary groups such as: exhaustive search, Shanks' "Baby-Step Giant-Step" algorithm and Pollard's Rho algorithm.
2. Algorithms working in arbitrary groups are efficient, particularly if the group has only small prime factors e.g. The Pohlig-Hellman algorithm and
3. The Index-Calculus algorithms, that are effective in certain groups only.

3.2 Properties of the Discrete Logarithm

The properties of the discrete logarithm problem are the same to those of the familiar logarithm [**Fact 6**]. We here define an additional property, once again common to both and readers are to understand G to be a finite cyclic group of order n written multiplicatively, where α is a generator of G.

Definition 8. *Change of Base Property*

$\widetilde{\alpha}$ is an element, a second generator of G and h is an element of G, then

$$\log_{\alpha}(\tfrac{h}{\widetilde{\alpha}}) \equiv (\log_{\alpha}(h) \text{ - } \log_{\alpha}(\widetilde{\alpha})(\text{mod } n))$$

Proof. The proofs of these properties will not be considered in this text, but if the reader wishes to understand the proofs then this is left as an exercise. □

Remark: Division may not always be possible because not all the elements of the additive group $\mathbb{Z}_n$, then r will be invertible iff $\gcd(r,n) = 1$ (greatest common divisor), where both r and n are relatively prime. Nevertheless provided that $\widetilde{\alpha}$ is a generator of G and that $\widetilde{\alpha}=\alpha^k$ for some k in $\mathbb{Z}_n$ if and only if $\gcd(k,n) = 1$, meaning it is now clear that $\log_{\alpha}(\widetilde{\alpha}) = k$ is invertible in $\mathbb{Z}_n$.

In relation to the Change-of-Base property, it does not matter what generator is chosen as this does not impact the difficulty level of the discrete logarithm problem. This is evident in the following: assume a method exists such that the discrete logarithm problem can be solved for base α in G. A second generator $\widetilde{\alpha}$ of G is then given with $h \in G$, so that the problem of finding $\log_{\alpha}(h)$ can be efficiently converted into a problem with the discrete logarithm base α by the Change-of-Base property. Hence, making all the generators effectively the same.

Theorem 2. *Discrete Logarithms of Products*

It is true for any finite field F_q, primitive element α and elements $\beta_0, \beta_1, ..., \beta_{k-1} \in \mathbb{Z}_q^*$ that:

$$\log_\alpha(\prod_{i=0}^{k-1} \beta_i) \equiv \sum_{i=0}^{k-1} \log_\alpha(\beta_i) \bmod \textit{(q-1)}$$

With consideration to the above, the discrete logarithms of $\mathbb{F}_q^*$ are thought to be elements of the additive group $\mathbb{Z}_{q-1}$. Further, it will suffice to think the discrete logarithm function as an isomorphism $\log_\alpha$:$\mathbb{F}_q^* \longrightarrow \mathbb{Z}_{q-1}$, and the key interest which we are focusing on is the analysis of this isomorphism.

3.3 Complexity

Definition 9. *Time and Space Complexity*

Time complexity is a term used to indicate the number of steps an algorithm must go through to reach a solution of a problem, expressed as a function of input size normally denoted *n*.

Likewise, *space complexity* as from it's name, relates to the amount of elementary units of "memory" required by an algorithm. Space complexity is also understood under the same input size and in the situation of an algorithm defined probabilistic, both time and space complexities involve the *predicted* numbers of steps and units of storage.

Algorithms involving finite fields, take for their elementary computation, field addition and/or field multiplication. This thesis will take an elementary computation as a multiplication between two field elements. It is to be made clear therefore that it follows that the algorithms computational complexity is not affected by the number of field additions that a particular algorithm is in need of. This choice of elementary computation is sufficient as we feel additions in the general case are performed at a quicker rate than compared to multiplications on computers and in the majority of algorithms, multiplications are not normally outnumbered by additions by an excessive quantity. The elementary storage unit is nearly always understood as a bit, and this the element size is $\log_q$ for an element of $\mathbb{F}_q$.

Two algorithms will now be defined in terms of time complexity - the *Polynomial Time Algorithm* and the *Exponential Time Algorithm.*

Definition 10. *Polynomial Time Algorithm*

An algorithm whose worst-case running time function is of the form *O(n^k)*, where n is the input size and k is a constant, is known to be a *polynomial time* *algorithm.* Any algorithm whose time complexity is not so bounded is an *exponential time algorithm.*

It is understood that polynomial time algorithms are *efficient* whilst exponential-time algorithms are not such and considered *inefficient.* Having stated this, these distinction rely on independent scenario's as for some practical situations they do not apply. The degree of polynomial is of great importance when thinking about polynomial-time complexity. If we consider an algorithm with running time $O(n^{ln(lnn)})$, where n is the input size, then it is asymptotically slower if compared to an algorithm with running time $O(n^{100})$. However, for smaller values of n, particularly if the constants hidden by the *big-O* notation are smaller, then the former algorithm may prove practically faster. But, the present discussion is in the light of worse-case complexities, we would like to remind the reader that in the field of cryptography, it is average-case complexity that is of more relevance rather than worst-case complexity. Moreover, a cryptanalysis problem that is on average difficult and not only for isolated cases is one that is understood as a secure combination for encryption schemes.

The "big-O" notation was mentioned above when explaining running time for both time and space complexities. This notation is one that is commonly seen when discussing such complexities and the definiton to follow should suffice in understanding this important notation.

Let us remind the reader, despite having already explained the fundamental theorems and terms in group theory in this thesis, a well versed mathematical background would prove beneficial.

Definition 11. *"Big-O" Notation*

A function $f(n) = Og(n)$ if for a positive constant $c>0$ there exists a constant n_0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.

May we take this opportunity to also define the following:

1. *(Asymptotic Lower Bound)* $f(n) = \Omega(g(n))$ if constant $c>0$ and integer $n_0>0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

2. *(Asymptotic Tight Bound)* $f(n) = \Theta(g(n))$ if constants $c_1,\ c_2 >0$ and integer $n_0 >0$ so that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n > n_0$.

$f(n) = O(g(n))$ has the meaning that, compared to $g(n)$, asymptotically *f* grows no faster within a constant multiple whereas, $f(m) = \Omega(g(n))$ tells one that $f(n)$ at the minimum, grows at an equivalent rate asymptotically as $g(n))$ under the same context. There also exists a small o-notation, the definition of which is ditto to"Big-O" notation:

3. A function $f(n) = o(g(n))$ if for any positive constant n_0>0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Furthermore, $f(n) = o(g(n))$ has the meaning that $g(n)$ is not an asymptotically tight upper bound for function $f(n)$. Alternatively this can be understood as $f(n)$ being insignificant relative to $g(n)$ as n increases. The particular expression $o(1)$ most generally means:

$$\lim_{n\to\infty} |\mathbf{f(n)}| = \mathbf{0}$$

3.4 Public Key Cryptography with Discrete Logarithms

Having covered the historic perspective and cryptographic developments in the literature review, it can justly be summarised that the key concept behind a public key cryptosystem is to allow one party A to send an encrypted message m to another party B via the use of a publicly known encryption key e_A. However, it is such that the encrypted message can only be decrypted by party B by it's secret private decryption key denoted d_A.

Computationally, d_A must be infeasible to derive from the public encryption key e_A. This certain computational property acts as a security measure against opponents/unauthorised parties, were they to attempt to eavesdrop during the transmission process of m from A to B, and try to decrypt the message with knowledge of e_A.

It may have already been understood from previous reading that the public key cryptosystem was first proposed by Diffie and Hellman in [11] after which many further developments followed; the RSA public cryptosystem was invented by Rivest, Shamir and Aldeman. This system relied on the difficulty of the integer factorisation problem for it's security, whilst ElGamal published a public key system based on the discrete logarithm problem in the multiplicative group of a prime field in 1985[]. To cite an example of how a public key cryptosystem functions:

- Take p to be a prime such that the discrete logarithm problem in $(\mathbb{Z}/\mathbb{Z}_p)^*$ is instractible, and a is a generator of $\mathbb{Z}/\mathbb{Z}_p^*$.

- Party A decides on a secret key d_A, (belonging to $\mathbb{Z}$) whilst alongside publishing it's public encryption key e_A - the integer $a^{d_A}\text{mod } p$.

- On the other hand, party B choose a random $k \in \mathbb{Z}/(p-1)\mathbb{Z}$ for the encrypted message m that it intends to send to A to read. Once, k is chosen, B then computes the pair (y_1, y_2), where

  $$y_1 = a^k \text{mod } p,\ y_2 = me_A^k \text{mod } p.$$

  Now as:

  $$\frac{y_2}{y_1^{d_A}} \equiv \frac{me_A^k}{a^{kd_A}} \equiv \frac{ma^{kd_A}}{a^{kd_A}} \equiv m\text{mod } p,$$

  it is now possible for party A to decrypt m by calculating the inverse of $y_1\text{mod } p$ followed by the computation of $y_2(y_1^{-1})^{d_A}\text{mod } p$.

  Now we describe two cryptographic protocols that are used in practical scenario's to achieve secret communication. They are of particular interest because the security of both these cryptosystems depends on the difficulty of computing discrete logarithms, outlining their importance and application to the field of cryptography. Both protocols see the use of parties A and B.

3.4.1 The Diffie-Hellman Key Exchange

To begin with we review the Diffie-Hellman key exchange protocol. This key agreement was the first solution to the key distribution problem (explained above) that enables two parties without prior acquaintance to establish a shared secret over an insecure connection/open channel.

This procedure keeps the key protected from passive adversaries, however, fails to do so for active adversaries, which can intercept, modify or inject messages. Both parties don't have key authentication or entity authentication, hence the source identity of the incoming message or the identity of the party which could know the resulting key are known by either party. The Diffie-Hellman key agreement procedure is as follows:

Diffie-Hellman Key Exchange

1. *One-time Setup.* A and B both select an appropriate prime p and generator α of $\mathbb{Z}_{|}^*$ over the insecure channel.

2. *Protocol messages:*

$$A \longrightarrow B\text{: } \alpha^x \text{mod } p$$
$$A \longleftarrow B\text{: } \alpha^y \text{mod } p$$

3. *Protocol actions.* Every time a shared key is needed then perform the following steps:

   a) A chooses a random secret x, $1 \leq x \leq p$ - 2 and sends B the message $A \longrightarrow B$: α^xmod p .
   b) B chooses a random secret y, $1 \leq y \leq p$ - 2 and sends A the message $A \longleftarrow B$: α^ymod p.
   c) B receives α^x and computes the shared key as $K = (\alpha^x)^y$mod p
   d) A receives α^y and computes the shared key as $K = (\alpha^y)^x$mod p

Though it may seem the Diffie-Hellman key agreement lets each party ensure key freshness and prevent key control, by using exponentiation by squaring (explained in next chapter) steps 2 and 3 can be computed efficiently. However, a crucial fact is that both A amd B arrive at equivalent random keys in step 3 because:

$$K \equiv B^x \equiv (\alpha^y)^x \equiv (\alpha^x)^y \equiv A^y (\text{mod } p)$$

This, therefore enables both parties to finalise a random secret key in $\mathbb{Z}_{\mathrm{l}}^*$ without prior communication over their insecure channel.

It further follows that for passive adversaries, computing $K = \alpha^{xy}(\text{mod } p)$ from $A{:=}\alpha^x(\text{mod } p)$ and $B{:=}\alpha^y(\text{mod } p)$ directly would hold to be a challenging task, and is named the computational Diffie-Hellman (CDH) assumption.

This, therefore further leads to the summarisation that the greatest chance for one to obtain the secret key is by using the discrete logarithm to find $a = \log_\alpha(A)$ or $b = \log_\alpha(B)$, then compute $K = B^a$mod p, or $K = A^b$mod p. But, for suitable primes p, the complexity assumption on the discrete logarithm problem indicates the task to be one of great difficulty thereby making the Diffie-Hellman key exchange quite secure.

3.4.2 The ElGamal Cryptosystem

The ElGamal Cryptosystem was first proposed by E.ElGamal in 1985, the security of this too relied on the complexity of computing discrete logarithms. In current times the default public key cipher to be used in cryptographic software such as GNU Privacy Guard is in fact the *ElGamal Cryptosystem.* Below we outline briefly the cryptosystem:

Elgamal Cryptosystem

1. *One-time setup (key generation and publication).* Every user B does the following:

   - Pick an appropriate prime p and generator α of $\mathbb{Z}_{\text{I}}^{*}$.
   - Select a random integer b, $1 \leq b \leq p$ - 2 and compute α^{b}mod p. This is the private key. B then publishes it's public key (p,α, α^{b}), whilst private key b is kept secret.

2. *Protocol messages:*

$$A \longrightarrow B\text{: } \alpha^{x}\text{mod } p$$

3. *Protocol actions.* Every time a shared key is needed, the following steps are performed:

   a) An authentic copy of *B's* public key (p,α, α^{b}) is recieved by A. A now chooses a random integer x, $1 \leq x \leq p$ - 2, and sends B the message $A \longrightarrow B$: α^{x}mod p. The key $K = (\alpha^{b})^{x}$mod p is computed by A.
   b) Party B computes the same key on reciept of the message $A \longrightarrow B$: α^{x}mod p as $K = (\alpha^{x})^{b}$mod p.

From the above it is evident that in order to decrypt the message, an eavesdropper would need to share knowledge of the private key. In order to extract b from the information collected, the adversary would need to compute the discrete logarithm, but if this is infeasible, then secure and private communication amongst both parties A and B is guaranteed.

Chapter 4

Efficient Computations in Finite Fields

This chapter is fully devoted to the concept of finite fields and their computational representations to allow the performance of different operations on them. Polynomials and ring integers are two such representations that we take high interest in this text, juxtaposition efficient arithmetic techniques and the computation of greatest common divisors (GCD's) will also be reviewed. Readers are to be made aware that all content discussed in this chapter is considered 'preliminary', however, hold no direct connection to the coming chapters and so can be skipped if desired. This chapter considers ideas more known to be "behind the scenes" than compared to those visible to the outside world or considered "higher level" content, which is left to discuss in later chapters.

4.1 Finite Field Representations

It is of grave importance to understand that finite fields are abstract structures and so far any computation to be performed within them, they need a certain representation. The particular word representation can be described as such under this particular context - *a solid description of all field elements involved and their interaction under binary operations of addition and multiplication.*

Depending on the representation adapted, each has it's advantages and disadvantages in relation to the required CPU memory needed to represent an element of a field on a computer and the number of processor cycles needed to perform operations.

In particular, focus shall remain on one representation for *prime order fields* and one representation for *prime power order fields* throughout, mainly because of the two being spontaneous, are likely to be familiar to the reader and are rather straight-forward to implement on a computer.

4.1.1 Fields of Prime Order

To begin with, we shall deal with fields of prime order more commonly understood as *integer representation*. When dealing with such a representation the operations are of modular nature, i.e. modular arithmetic, including modular addition, subtraction, multiplication and division.

As this thesis is not a study on, modular arithmetic, we shall not discuss it in detail, but instead briefly mention and explain related properties.

4.1.1.1 Properties of Modular Arithmetic

Consider the set $\mathbb{Z}_n$ as the non-negative integers less than n:

$$\mathbb{Z}_n = \{0,1,...,(n \text{ - } 1)\}$$

This particular set is called the **set of residues**, in which every integer in $\mathbb{Z}_n$ represents a residue class. From all these, the smallest non-negative integer is mostly taken to be the one representing the residue class. Residue class (mod$n$) can be labelled as $\{0\}$, $\{1\}$, $\{2\}$,...,$\{n$ - 1$\}$ where:

$$\{r\} = \{a{:}a \text{ is an integer, } \{a\} \equiv r(\text{mod})\ n\}$$

To cite an example:

$$\begin{gathered}\text{The residue classes (mod4) are:}\\ \{0\} = \{...,\text{-}16, \text{-}12, \text{-}8, 0, 4, 0, 8, 12, 16,...\}\\ \{1\} = \{...,\text{-}15, \text{-}11, \text{-}7, \text{-}3, 1, 5, 9, 13, 17,...\}\\ \{2\} = \{...,\text{-}14, \text{-}10, \text{-}6, \text{-}2, 2, 6, 10, 14, 18,..\}\\ \{3\} = \{...,\text{-}13, \text{-}9, \text{-}5, \text{-}1, 3, 7, 11, 15, 19,...\}\end{gathered}$$

The properties of Modular Arithmetic for integers in $\mathbb{Z}_n$ can be summarised as such:

1. **Commutative Laws**

   - $(w + x)\text{mod } n = (x + w)\text{mod } n$
   - $(w \times x)\text{mod } n = (x \times w)\text{mod } n$

2. **Associative Laws**

   - $((w{+}x) + y)\text{mod } n = (w + (x{+}y))\text{mod } n$

- $((w \times x) \times y) \bmod n = (w \times (x \times y)) \bmod n$

3. **Distributive Law**

   - $(w \times (x+y)) \bmod n = ((w \times x) + (w \times y)) \bmod n$

4. **Identities**

   - $(0+w) \bmod n = w \bmod n$
   - $(1 \times w) \bmod n = w \bmod n$

5. **Additive Inverse (-w)**

   - For each $w \in \mathbb{Z}_n$ there exists a z such that $w+z \equiv 0 \bmod n$.

Division in modular arithmetic is carried out by using the Extended Euclidean algorithm, which is also used to calculate multiplicative inverses and then perform a multiplication. The predecessor, Euclidean algorithm can be used to calculate greatest common divisors. Both Euclidean algorithms will not be considered in detail but the following theorem summarises the former:

Theorem 3. *For any non-negative integer a and any positive integer b,*

$$\gcd(a, b) = \gcd(b,\, a \bmod b)$$

Example 2. *gcd(55,22) = gcd(22, 55mod22) = gcd(22,11) = 11.*

Meletiou and Mullen discuss fields of prime order and fields of prime power order in a very theoretical and mathematical perspective in [18] to which readers are referred to if they desire to gain insight knowledge.

The next section discusses prime power order fields that are represented via polynomials.

4.1.2 Finite Fields of Prime Power Order

This representation involves the use of polynomials, in particular each polynomial coefficient. Nevertheless it is required that the polynomial be **irreducible**. When we say irreducible we mean "a polynomial $f(x)$ over a field $\mathbb{F}$ is **irreducible** if and only if $f(x)$ can't be written as a product of two polynomials, both over $\mathbb{F}$, and both with a degree lower than that of $f(x)$. By analogy to integers, an irreducible polynomial is also called a **prime polynomial**".

Note 3: We refer to the finite field $\mathbb{F}$ above to be of order p^m.

Fact 7 Let $f(x) \in \mathbb{Z}_p[x]$ be an irreducible polynomial of degree m. Then $\mathbb{Z}_p[x]/f(x)$ is a finite field of order p^m. Addition and multiplication of polynomials is performed modulo $f(x)$.

4.1.2.1 Polynomial Arithmetic

The fact to follow ensures that all finite fields can be represented through polynomials.

Fact 8 For each $m \geq 1$, there exists a monic irreducible polynomial of degree m over $\mathbb{Z}_p$. Therefore, every finite field has a polynomial basis representation.

Determining polynomials that are irreducible over finite fields will be explained in a later section in this chapter.

As for elements of the finite field $\mathbb{F}_{p^m}$, these can be represented by polynomials in $\mathbb{Z}_p[x]$ of degree less than m. Say the two polynomials $g(x)$ and $h(x)$ belong to $\mathbb{F}_{p^m}$ it then follows that addition is the same as normal polynomial addition in $\mathbb{Z}_p[x]$. The product $g(x)h(x)$ can be obtained by firstly multiplying $g(x)$ and $h(x)$ as polynomials via the ordinary method, and then dividing by $f(x)$ and taking the remainder, (where $f(x)$ denotes an irreducible polynomial$\in \mathbb{Z}_p[x]$). As with integers, multiplicative inverses in $\mathbb{F}_{p^m}$ can also be calculated via the extended Euclidean algorithm for the polynomial ring $\mathbb{Z}_p[x]$.

To cite examples of polynomial arithmetic please consider the following:

Example 3. *Consider the finite field $\mathbb{F}_{2^4}$ (of order 16) and the irreducible polynomial f(x) = $x^4 + x + 1$ over $\mathbb{Z}_2$ (The irreducibility of f(x) can be verified with the algorithm presented in the next section). This then yields that the finite field $\mathbb{F}_{2^4}$ can be represented as the set of all polynomials over $\mathbb{F}_2$ of degree less than 4. So,*

$$\mathbb{F}_{2^4} = \{a_3x^3 + a_2x^2 + a_1x + a_0 \mid a_i \in \{0,1\}\}.$$

For simplicity, the polynomial $a_3x^3 + a_2x^2 + a_1x + a_0$ can be seen as the vector $(a_3 a_2 a_1 a_0)$ with length 4, and

$$\mathbb{F}_{2^4} = \{(a_3 a_2 a_1 a_0) \mid a_i \in \{0,1\}\}.$$

The following present examples of field arithmetic:

1. Addition of field elements is done component wise: e.g. (1011) + (1001) $\equiv$ (0010).

2. The field elements (1101) and (1001) can be multiplied as polynomials and then the remainder taken after this product is divided by $f(x)$:

   $(x^3 + x^2 + 1)(x^3 + 1) = x^6 + x^5 + x^2 + 1 \equiv x^3 + x^2 + x + 1 \pmod{f(x)}$,

   this implies that (1101)(1011) = (1111).

3. The multiplicative identity is $\mathbb{F}_{2^4}$ is (0001).

4. The inverse of (1011) is (0101). This can be verified by considering:

   $(x^3 + x + 1)(x^2 + 1) = x^5 + x^2 + x + 1 \equiv 1\text{mod } (f(x))$, implying

   (1011)(0101) = (0001).

$f(x)$ is a primitive polynomial, or, equally, the field element $x = (0010)$ is a generator of $\mathbb{F}_2^*$. This can be proven by checking that all the non-zero elements in $\mathbb{F}_{2^4}$ can be obtained as powers of x. The below table summarises computations of x *modulo* $f(x)$.

The powers of x modulo f(x) $= x^4 + x + 1$		
i	x^i mod $x^4 + x + 1$	Vector Notation
0	1	(0001)
1	x	(0010)
2	x^2	(0100)
3	x^3	(1000)
4	$x + 1$	(0011)
5	$x^2 + x$	(0110)
6	$x^3 + x^2$	(1100)
7	$x^3 + x + 1$	(1011)
8	$x^2 + 1$	(0101)
9	$x^3 + x$	(1010)
10	$x^2 + x + 1$	(0111)
11	$x^3 + x^2 + x$	(1110)
12	$x^3 + x^2 + x + 1$	(1111)
13	$x^3 + x^2 + 1$	(1101)
14	$x^3 + 1$	(1001)

Considering the above from a computational perspective, it follows that a sufficient irreducible polynomial $f(x)$ is first to be found to use as a modulus, after which implementation can follow accordingly.

4.1.3 Further Reading

Despite having reviewed both integer and polynomial representation and arithmetic in this chapter, we would consider to mention [18] again in which Meletiou and Mullen discuss both prime order and prime power order fields in grave mathematical detail. For deeper understanding on polynomial arithmetic please refer to [2], which gives a breakdown of polynomial arithmetic in a step-by-step manner.

4.2 Discrete Exponentiation

In continuation to the above subject of finite field arithmetic, this section considers the topic of exponentiation. Exponentiation is closely linked to the operation of multiplication in finite fields and is one of the most important arithmetic operations for public-key cryptography. Just to emphasise its important, the famous RSA scheme requires exponentiation in $\mathbb{Z}_m$ for some positive integer m, similarly both the Diffie-Hellman key agreement and the ElGamal encryption scheme use exponentiation in $\mathbb{Z}_p$ for some large prime p.

Coming back to the discrete logarithm problem, as difficult as it's solution may be believed to be, it is also believed to be an example of a one-way function and it turns out that it is easier to define it as it's inverse - the discrete exponentiation function. We refer to the discrete exponentiation as the analogue of the well known exponentiation, but occurring in a finite cyclic group.

Definition 12. *Discrete Exponentiation*

G is a finite cyclic group of order n, written multiplicatively, and let α be a generator of G. The discrete exponentiation function is then defined as would be expected:

$$\exp_\alpha \colon \mathbb{Z}_n \longrightarrow G, x \longmapsto \alpha^x$$

It follows that the elements α^i for i in $\mathbb{Z}_n$ are unique from one another as α is a generator of G with order n. The map $\exp_\alpha$ can therefore be called injective, yet the domain $\mathbb{Z}_n$ and the range G of this map both contain n elements meaning $\exp_\alpha$ is actually bijective. Now the simple method of computing this function involves repeatedly multiplying α by itself x-1 times $\alpha^x = \overbrace{\alpha ... \alpha}^{x}$ needing $O(x)$ group operations. Nevertheless, there exists a much more superior and efficient technique and the next section is dedicated to this very technique.

4.3 Exponentiation by Squaring

Exponentiation by squaring allows a faster exponentiation method by exploiting the binary representation of the exponent $x = (x_{k-1},...,x_1x_0)_2$. Before we continue let us note that,

$$x = \sum_{j=0}^{k-1} x_j 2^j = \sum_{x_j=1} 2^j$$

It follows that α^x can then be written as:

$$\alpha^x = \alpha^{\sum_{x_i=1} 2^i} = \prod_{x_i=1} \alpha^{2^i}$$

Further,

$$\alpha^{2^i} = \left(\alpha^{2^{i-1}}\right)^2$$

α^{2^i} can be found by successive use of single squaring. It can then be said that the number of required group operations to compute α^x is dependent on the binary length of x!

Exponentiation by squaring can be recursively expressed as such:

$$\exp_\alpha(x) := \begin{cases} 1, & \text{if } x = 0 \\ exp_\alpha^2(x/2), & \text{if } x \text{ is even} \\ \alpha.exp_\alpha^2((x-1)/2), & \text{if } x \text{ is odd} \end{cases}$$

The key advantage of exponentiation by squaring is that it is extremely fast in it's algorithmic calculation, no matter how large the value of x is. This is because the maximum number of multiplication and squaring operations it requires is equal to the length of the binary version of the input. Therefore, this method has a runtime of only $O(\log_2(x))$. We can continue that the properties of groups ensures $\alpha^{-1} = (\alpha^x)^{-1}$ adapting this method to also be usable when involving negative exponents $\exp_\alpha$(-x), given that the elements can be inverted.

The following presents an algorithm for computing exponentiation by squaring:

4.4 Irreducible Polynomials

If we recall, a polynomial $f(x) \in \mathbb{Z}_p[x]$ of $m \geq 1$ is *irreducible over* $\mathbb{Z}_p$ if it can't be written as a product of two polynomials in $\mathbb{Z}_p[x]$. each with degree less than m. $f(x)$ allows elements of the finite field $\mathbb{F}_{p^m}$ as $\mathbb{F}_{p^m} = \mathbb{Z}_p[x]/f(x)$, with all polynomials in $\mathbb{Z}_p[x]$ of degree $<m$ and polynomials is carried out modulo $f(x)$.

Algorithm 1 Exponentiation by Squaring

```
Require: $\alpha$ in $G$ and $x$ in $\mathbb{Z}$ with a multiplicative cyclic group $G$
Ensure: $\beta = \alpha^x$

1. $\beta \leftarrow 1$ /* 1 is the identity element of $G$ */
2. while $x > 0$ do
     if $x$ is odd then
       $\beta \leftarrow \beta \times \alpha$
       $x \leftarrow x - 1$
     end if
     $\beta \leftarrow \beta^2$
     $x \leftarrow x/2$
3. end while
```

We present in this section a method through how irreducible polynomials can be constructed over $\mathbb{Z}_p$ in which p is a prime. For crytpographic applications however, finite fields of characterstic two $\mathbb{F}_{2^m}$ are of particular interest as these fields allow efficient arithmetic performance in both software and hardware. Hence, irreducible polynomials over $\mathbb{Z}_2$ are of close attention.

If the chosen irreducible polynomial has few non-zero terms then finite field arithmetic is made more efficient to implement. Consider $f(x)$ to be an irreducible polynomial over $\mathbb{Z}_p$ a non-zero element in $\mathbb{Z}_p$, it then follows that a $f(x)$ is also irreducible over $\mathbb{Z}_p$. It can therefore be said that we can limit our focus down to monic polynomials in $\mathbb{Z}_p$ polynomials which have a leading co-efficient of 1. This therefore means if $f(x)$ is an irreducible polynomial, it's constant term has to then be non-zero, especially in the case when $f(x)\in \mathbb{Z}_2[x]$, for which it's constant term must be 1.

There exists a formula for calculating the exact number of monic irreducible polynomials in $Z_p[x]$ with fixed degree, via the aid of a function called the Möbius function.

Definition 13. *Let m be a particular integer. The Möbius function μ is defined by,*

$$\mu(m) = \begin{cases} 1, & \text{if } m = 1 \\ 0, & \text{if } m \text{ is divisible by the square of a prime} \\ -1^k, & \text{if } m \text{ is the product of } k \text{ distinct primes} \end{cases}$$

Example 4. *(Möbius function)*

The table below shows the first 10 values of m for the Mobiöus function $\mu(m)$:

□

Table 4.1: First 10 values of Mobiöus function

m	1	2	3	4	5	6	7	8	9	10
$\mu(m)$	1	-1	-1	0	-1	1	-1	0	0	1

4.4.1 Testing for irreducibility

We now present an algorithm which allows for testing a polynomial for irreducibility. In comparison to testing primality of integers, testing polynomials for irreducibility in $\mathbb{Z}_p[x]$ is much simpler. All that has to be done is test whether a polynomial has no irreducible factors with degree $\leq [m/2]$.

Fact 9 Consider p a prime and k a positive integer:

1. The product of all monic irreducible polynomials in $\mathbb{Z}_p[x]$ of degree dividing k is equivalent to x^{p^k} - x.
2. Allow $f(x)$ to be a polynomial with degree m in $\mathbb{Z}_p[x]$. $f(x)$ is then irreducible over $\mathbb{Z}_p$ iff gcd($f(x)$, x^{p^k} - x) = 1 for every i, 1 $\leq i \leq [m/2]$.

Algorithm 2 Testing a polynomial for irreducibility

Require: a prime p and a monic polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$
Ensure: an answer to the question "Is $f(x)$ irreducible over $\mathbb{Z}_p$?"

```
1. $u(x) \leftarrow x$
2. for $i = 1$ from 1 to [m/2] do
      - Compute $u(x) \leftarrow u(x)^p$ mod $f(x)$ using the Repeated square-and-
      multiply algorithm for exponentiation in ($F_{p^m}$). (Note that $u(x)$ is a
      polynomial in $\mathbb{Z}_p[x]$ of degree less than $m$)
      - Compute $d(x) = \gcd(f(x), u(x) - x)$(Using the Euclidean Algorithm
      for $\mathbb{Z}_p[x]$
      - If $d(x) \neq 1$ then return("irreducible")
3. end for
4. return("irreducible")
```

4.4.2 Generating irreducible polynomials

We now present an algorithm for generating a random monic polynomial over $\mathbb{Z}_p$.

Algorithm 3 Generating a random monic irreducible polynomial over $\mathbb{Z}_p[x]$

```
Require: a prime $p$ and a positive integer $m$
Ensure: a monic irreducible polynomial $f(x)$ of degree $m$ in $\mathbb{Z}_p[x]$
    1. Repeat the following:
    2. (Generate a random monic polynomial of degree m in $\mathbb{Z}_p[x]$). Randomly
       select integers $a_0$, $a_1$, $a_2$,...,$a_{m-1}$ between 0 and $p$ - 1 with $a_0 \neq 0$. Let
       $f(x)$ be the polynomial $f(x) = x^m + a_{m-1}x^{m-1}$ +...+$a_2x^2$+$a_1x$+$a_0$
    3. Use Algorithm 2 to test whether $f(x)$ is irreducible over $\mathbb{Z}_p$
    4. Until $f(x)$ is irreducible
    5. return($f(x)$)
```

4.5 Primitive Polynomials

Definition 14. *An irreducible polynomial f(x) $\in \mathbb{Z}_p$[x] of degree m is called a primitive polynomial if x is a generator of $\mathbb{Z}^*_{p^m}$, the multiplicative group of all the non-zero elements in $\mathbb{F}_{p^m} = \mathbb{Z}_p$[x]/(f(x)).*

If the factorisation of the integer p^m - 1 is known, Fact 5 then holds true an effective algorithm (Algorithm 4) to assess if $f(x)$ is a primitive polynomial or not. However, if the factorisation of p^m - 1 is not known then there is no algorithm known for this test.

Fact 10 Consider p to be prime and let the unique prime factors of P^m-1 be $r_1, r_2, ..., r_t$. It then follows that an irreducible polynomial $f(x) \in \mathbb{Z}_p[x]$ is primitive iff for each i, $1 \leq i \leq t$:

$$x^{(p^m-1)/r_i} \not\equiv 1 \pmod{f(x)}.$$

(Such that, x is an element of order $p^m - 1$ in the field $\mathbb{Z}_p[x]/(f(x))$.)

The next page outlines an algorithm for testing if an irreducible polynomial is primitive or not.

Algorithm 4 Testing whether an irreducible polynomial is primitive

Require: a prime p, a positive integer m, the unique prime factors $r_1, r_2, ..., r_t$ of p^m -1, and a monic irreducible polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$.

Ensure: An answer to the question: "Is $f(x)$ a primitive polynomial?"

```
1. for $i$ from 1 $t$ do
    1.1 Compute $l(x) = x^{(p^m-1)/r_i}$mod $f(x)$ (Using Algorithm 8).
    1.2 If $l(x) = 1$ thn return("not primitive").
2. end for
3. Return("primitive").
```

Precisely there exists $\phi(p^m - 1)/m$ monic primitive polynomials with degree m in $\mathbb{Z}_p[x]$, i which the Euler phi function is ϕ. As the quantity of monic irreducible polynomials with degree m is approximately p^m/m, it holds true that the likelihood of a random monic irreducible polynomial with degree m in $\mathbb{Z}_p[x]$ is primitive is roughly $\phi(p^m - 1)/p^m$. Considering the lower bound for the Euler phi function, there is at minimum $1/(6\text{lnln}p^m)$ possibility that this can be seen. This then yields the algorithm outlined below for generating primitive polynomials.

Algorithm 5 Generating a random monic primitive polynomial in $\mathbb{Z}_p$

Require: a prime p, integer $m \geq 1$, and the unique prime factors $r_1, r_2, ..., r_t$ of p^m -1.

Ensure: A monic primitive polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$.

```
1. Repeat the following until $f(x)$ is primitive:
2. Use Algorithm 3 to generate a random monic irreducible polynomial $f(x)$
   of degree $m$ in $\mathbb{Z}_p[x]$
3. Use Algorithm 4 to test whether $f(x)$ is primitive
4. Return($f(x)$).
```

4.5.1 Further Reading

We can understand the level of relevance and importance irreducible polynomials carry for finite fields of prime power order. It is through this property that polynomials can be used to represent finite fields GF(p^m). Integer representation in $\mathbb{Z}_p$ allows for modular arithmetic whilst representation in GF(p^m) asks for polynomial modular arithmetic also known as finite filed arithmetic. In addition to the material discussed in this chapter, finite field representation also touches the topic of irredicuble trinomials, which have not been discussed in this chapter as they pose to be extra knowledge on the subject of polynomials. For this reason we recommend readers refer to [1] Handbook of Applied Cryptography for further knowledge on this subject and in depth detail on monic polynomials, which have been very briefly considered in review of this chapter.

Chapter 5

Determining the Discrete Logarithm

We now consider the different algorithms that exist for solving the discrete logarithm problem. Through presenting these algorithms we shall adapt an understanding on why it is believed to be computationally difficult and study the time and space complexities of the different algorithms.

The algorithms can be grouped into two categories *generic* and *non-generic.* We first present algorithms belonging to the generic class in the following order:

1. Brute Force Search
2. Shanks' "Baby-Step Giant-Step" Method
3. Pollard's ρ Algorithm for logarithms (1978)
4. Pohlig-Hellman Algorithm (1978)

The chapter then reviews an algorithm which is of "*non-generic*" nature called the Index-Calculus Algorithm. Till date this algorithm is said to accomplish the lowest complexities known. The Index-Calculus Algorithm is specific only to finite fields and $\mathbb{Z}_p^*$ which are the key interest of this text, whilst the generic group algorithms are applicable over any type of cyclic group, including elliptic curves groups and subgroups of $\mathbb{Z}_p^*$, where more efficient methods are not known to apply.

Any of the techniques discussed in the previous chapter for efficient computations in finite fields can be used in implementing these algorithms, and readers are to understand the pseudocode's and examples illustrated in this chapter are under the context of the finite field $\mathbb{F}_p$ for ease of understanding and explanatory purposes. For finite field $\mathbb{F}_{2^n}$ we are to take the input as irreducible polynomials of the form $f(x)$ and $g(x)$ and the mathematical operations as polynomial arithmetic, unless stated.

5.1 Procedures in Generic Discrete Logarithm Algorithms

Before moving on to present and discuss the generic Discrete Logarithm Algorithms we would here like to first briefly present the procedures that the algorithms individually use to obtain $x = \log_\alpha \beta$.

5.1.1 Extended Euclidean Algorithm

The *Extended Euclidean Algorithm* is an extension of its predecessor the Euclidean Algorithm and is used to compute the greatest common divisor d of two integers a and b alongside integers x and y that satisfy the property $ax + by = d$.

Algorithm 6 Extended Euclidean Algorithm

```
Require: Two non-negative integers $a$ and $b$ with $a \geq b$
Ensure: $d = \gcd(a, b)$ and integers $x$, $y$ satisfying $ax + by = d$

1. if $b = 0$ then
       $d \leftarrow a, x \leftarrow 1, y \leftarrow 0$, and return$(d, x, y)$
2. end if
3. Set $x_2 \leftarrow 1, x_1 \leftarrow 0, y_2 \leftarrow 0, y_1 \leftarrow 1$
4. while $b > 0$ do
       $q \leftarrow [a/b], r \leftarrow a - qb, x \leftarrow x_2 - qx_1, y \leftarrow y_2 - qy_1$
       $a \leftarrow b, b \leftarrow r, x_2 \leftarrow x_1, x_1 \leftarrow x, y_2 \leftarrow y_1$, and $y_1 \leftarrow y$
5. end while
6. Set $d \leftarrow a, x \leftarrow x_2, y \leftarrow y_2$, and return$(d, x, y)$
```

The above procedure has a running time of $O((\log n)^2)$ bit operations.

Example 5. *(Extended Euclidean Algorithm)*

Extended Euclidean Algorithm with inputs $a = 4864$, $b = 3458$

q	r	x	y	a	b	x_2	x_1	y_2	y_1
-	-	-	-	4864	3458	1	0	0	1
1	1406	1	-1	3458	1406	0	1	1	-1
2	646	-2	3	1406	646	1	-2	-1	3
2	114	5	-7	646	114	-2	5	3	-7
5	76	-27	38	114	76	5	-27	-7	38
1	38	32	-45	76	38	-27	32	38	-45
2	0	-91	128	38	0	32	-91	-45	128

The table overleaf outlines the steps of Algorithm 6 with inputs $a = 4864$ and $b = 3458$. Therefore gcd(4864,3458) = 38 and (4864)(32) + (3458)(-45) = 38. □

5.1.2 Multiplicative Inverse

Algorithm 7 Computing multiplicative inverses in $\mathbb{Z}_n$

```
Require: $a \in \mathbb{Z}_n$
Ensure: $a^{-1}$mod $n$, provided that it exists
    1. Use the extended euclidean algorithm (Algorithm 6) to find integers $x$ and
       $y$ such that $ax + ny = d$, where $d = \gcd(a, n)$
    2. if $d > 1$ then
           $a^{-1}$ mod $n$ does not exist. Otherwise, return($x$)
    3. end if
```

5.1.3 Square-and-Multiply

Algorithm 8 Repeated square-and-multiply algorithm for exponentiation in $\mathbb{Z}_n$

```
Require: $a \in \mathbb{Z}_n$, and integer $0 \leq k < n$ where $k = \sum_{i=0}^{t} k_i 2^i$
Ensure: $a^k$mod$n$
    1. $b \leftarrow 1$.
    2. if $k = 0$ then return($b$)
    3. end if
    4. Set $A \leftarrow a$
    5. if $k_0 = 1$ then set $b \leftarrow a$
    6. end if
    7. for $i$ from 1 to $t$ do
       7.1 Set $A \leftarrow A^2$ mod $n$
       7.2 if $k_i = 1$ then set $b \leftarrow A \times (b$mod $n)$
    8. end if
    9. end for
   10. return($b$)
```

5.1.4 Chinese Remainder Theorem, CRT

If the integers $n_1, n_2, ..., n_k$ are pairwise relatively prime, then the system of simultaneous congruences

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

has a unique solution modulo $n = n_1 n_2 ... n_k$.

5.1.5 Gauss's Algorithm

The solution x to the simultaneous congruences in the Chinese Remainder Theorem can be computed as $x = \sum_{i=1}^{k} a_i N_i M_i \bmod n$, where $N_i = n/n_i$ and $M_i = N_i^{-1} \bmod n_i$. These computations require $O((\log n)^2)$ bit operations.

5.1.6 Floyd's Cycle-finding algorithm

Floyd's Cycle-finding algorithm begins with the pair (x_1, x_2), and iteratively computes (x_i, x_{2i}) from a pre-computed pair (x_{i-1}, x_{2i-2}), until $x_m = x_{2m}$ for some m. If the tail of the sequence has length λ and the cycle has length μ, then the first occurance of $x_m = x_{2m}$ is at the point when $m = \mu(1 + [\lambda/\mu])$. We note here that $\lambda < m \leq \lambda + \mu$, and as a result the anticipated running time of Floyd's cycle-finding algorithm is $O(\sqrt{n})$.

5.2 Generic Discrete Logarithm Algorithms

We now present the generic Discrete Logarithm Algorithms. Alongside analyses- and description we will also present pseudocode for each and the variable structure will be as such:

- α = Generator of the cyclic group G
- β = An element of G
- n = The order of the group G

5.2.1 Brute-Force Search

The first and foremost approach in solving the discrete logarithm problem would be to begin with the simplest method, Brute-Force (This method can be referred to as *Exhaustive Search* or *Trial Exponentiation.*) This approach simply involves trying every possible exponent $(\alpha^0, \alpha^1, \alpha^2 ...)$ until a match is obtainable i.e. until $\alpha^x = \beta$.

Algorithm 9 Brute-Force Search

Require: A generator α, group element β and cyclic group G of order n
Ensure: The discrete logarithm $x = \log_\alpha \beta$

```
1. $b \Leftarrow 1$
2. $x \Leftarrow 0$
3. **while** $\beta \neq b$ **do**
       $b \Leftarrow b \times \alpha$
       $x \Leftarrow x + 1$
4. **end while**
5. **return** $x$
```

The point at which $\alpha^x = \beta$, determines the solution x (the discrete logarithm). Please refer back to Chapter 3 of this text to review the Discrete Logarithm problem in detail. Yet, this method is inefficient when dealing with too large a number n i.e. the group order is exceedingly large (particularly for cryptographic interest.) This is becasue, as with the general case, the brute-force requires $O(n)$ group operations to run.

5.2.2 Shanks' "Baby-Step Giant-Step" Algorithm

The next algorithm presented to solve the discrete logarithm problem is known as Shanks' "Baby-Step Giant-Step" Algorithm, which was proposed by D.Shanks and is a time memory trade-off of the *brute-force search*. As from it's name, it consists of two stages: the first stage involves consecutively stepping through the first j powers of $\alpha^j : \alpha^1, \alpha^2, ..., \alpha^{j-1}$ known as the **"Baby-Steps"** of the algorithm. Each exponent value j is stored in a hash table/list, which is indexed by α^j. After completing j steps, we have a discrete logarithm table coinciding however, to only the first j elements of the cyclic group.

The **"Giant-Steps"** are based on the following observation, if $\beta = \alpha^x$, then it is possible to write $x = im + j$ where $m = \sqrt{n}$, $j < m$ and $0 \leq i$. It follows that:

$$\alpha^x = \alpha^{im}\alpha^j$$

$$\Rightarrow \beta(\alpha^{-m})^i = \alpha^j$$

meaning that in this second stage the input $\beta = \alpha^x$ is transformed into a value which lies in the precomputed discrete logarithm range. We begin at α^x and check through the cyclic group at x elements at a time until we reach the beginning of the cycle of the precomputed logarithms. This then leads to the algorithm outlined below for computing x:

Algorithm 10 Shanks' "Baby-Step Giant-Step" Algorithm

```
Require: A generator $\alpha$, group element $\beta$ and cyclic group $G$ or order $n$
Ensure: The discrete logarithm $x = \log_\alpha\beta$

  1. Set $m \leftarrow \sqrt{n}$
  2. Construct a table with entries $(j, \alpha^j)$ for $0 \leq j \leq m$. Sort this table by
     second component
  3. Compute $\alpha^{-m}$ and set $\gamma \leftarrow \beta$
  4. for $i$ from 0 to $m$-1 do
       4.1 Check if $\gamma$ is the second component of some entry in the table
       4.2 If $\gamma = \alpha^j$ then return($x = im + j$)
       4.3 Set $\gamma \leftarrow \gamma \times \alpha^{-m}$
  5. end for
```

Algorithm 10 needs a storage space for holding $O(\sqrt{n})$ number of group elements. The table in step 2 requires $O(\sqrt{n})$ multiplications to make and $O(\sqrt{n}\log n)$ comparisons to sort. Once the table has been constructed, step 4 then takes $O(\sqrt{n})$ table look-ups. We here assume that a group multiplication takes longer than $\log n$ comparisons, the running time of Shanks' Algorithm is as such.

Fact 11 The running time of Algortihm 5 is $O(\sqrt{n})$ group multiplications.

Example 6. *Shanks' "Baby-Step Giant-Step" Algorithm for logarithms in a cyclic group G.*

Let $p = 113$. The element $\alpha = 3$ is a generator of cyclic group G of order $n = 112$. Consider $\beta = 57$. Then $\log_3 57$ is computed as follows.

1. Set $m \leftarrow 112 = 11$.
2. Construct a table whose entries are $(j,\ \alpha^j \bmod p)$ for $0 \leq j < 11$:

   and sort the table by second component:
3. Using the multiplicative inverse method, compute $\alpha^{-1} = 3^{-1} \bmod 113 = 38$ and then compute $\alpha^{-m} = 38^{-11} \bmod 113 = 58$.

j	0	1	2	3	4	5	6	7	8	9	10
3^j mod113	1	3	9	27	81	17	51	40	7	21	63

j	0	1	8	2	5	9	3	7	6	10	4
3^j mod113	1	3	7	9	17	21	27	40	51	63	81

4. Next $\gamma = \beta\alpha^{-mi}$mod 113 for i = 0,1,2,... is computed until a value in the second row of the table is obtained. This yields:

i	0	1	2	3	4	5	6	7	8	9
$\gamma = 57 \times 58^i$mod 113	57	29	100	37	112	55	26	39	2	3

Finally, since $\beta\alpha^{-9m} = 3 = \alpha^1$, $\beta = \alpha^{100}$ and, therefore, $\log_3 57 = 100$. □

Note 4: Step 3 of Algorithm 10 uses the *multiplicative inverse*, (Algorithm 7) whilst step 4 requires the use of the *square-and-multiply* method (Algorithm 8) for $\gamma = \beta\alpha^{-mi}$ especially when dealing with large power values.

5.2.3 Pollard's ρ Algorithm for Discrete Logarithms

The next algorithm we present is known as Pollard's ρ (pronounced "Rho") Algorithm for logarithms. We would like to mention here that another algorithm with the same name i.e. Pollard's ρ method exists, however is of a different nature to the current topic of discussion. Both algorithms however, have been proposed by J.Pollard in 1978, the former being for discrete logarithms whilst the latter deals with prime factorisation.

This third algorithm shares an equivalent running time to that of Algorithm 10 - Shanks' "Baby-Step Giant-Step" Algorithm, however is practically more efficient and prefferabe due to it's negligable storage requirement.

It is currently the most generic algorithm known for computing discrete logarithms and is a randomised algorithm because it's time complexity is derived from an assumption based on a pseudorandom sequence as outlined below:

Take S_1, S_2 and S_3 to be three equally sized sets of a partitioned group G. The set size of each is to be based on a trivially testable property and when selecting the partition, care must be taken, e.g. $1 \notin S_2$. A sequence of group elements $x_0, x_1, x_2, ...$ with $x_0 = 1$ can then be defined and

$$x_{i+1} = f(x_i) = \begin{cases} \beta \times x_i, & \text{if} x_i \in S_1 \\ x_i^2, & \text{if} x_i \in S_2 \\ \alpha \times x_i, & \text{if} x_i \in S_3 \end{cases} \tag{5.1}$$

for $i \geq 0$. Following (5.1), two further integer sequences (5.2) and (5.3) can be derived for $a_0, a_1, a_2, ...$ and $b_0, b_1, b_2, ...$ which satisfies $x_i = \alpha^{a_i}\beta^{b_i}$ for $i \geq 0 : a_0 = 0, b_0 = 0$ and for $i \geq 0$,

$$a_{i+1} = \begin{cases} a_i, & \text{if} x_i \in S_1 \\ (2a_i) mod n, & \text{if} x_i \in S_2 \\ (a_i + 1) mod n, & \text{if} x_i \in S_3 \end{cases} \tag{5.2}$$

and

$$b_{i+1} = \begin{cases} (b_i + 1) mod n, & \text{if} x_i \in S_1 \\ (2b_i) mod n, & \text{if} x_i \in S_2 \\ b_i, & \text{if} x_i \in S_3 \end{cases} \tag{5.3}$$

Pollard's ρ algorithm then uses Floyd's cycle finding algorithm (§5.1.6) to determine two group elements x_i and x_{2i} such that $x_i = x_{2i}$. This, therefore leads to:

$$\alpha^{a_i}\beta b_i = \alpha^{a_{2i}}\beta^{b_{2i}}$$
$$\Rightarrow \beta^{b_i - b_{2_i}} = \alpha^{a_{2i} - a_i}$$

If logarithms to the base α are then taken from both sides, the following is obtained:

$$(b_i - b_{2i}) \times \log_\alpha \beta \equiv (a_{2i} - a_i)(\text{mod } n)$$

The discrete logarithm $\log_\alpha \beta$ can be obtained by effectively solving the equation $(b_i - b_{2i}) \times \log_\alpha \beta \equiv (a_{2i} - a_i)(\text{mod } n)$, considering $b_i \not\equiv b_{2i}$ (mod n).

Note 5: $b_i \equiv b_{2i}$ takes place with negligible probability.

The next page illustrates pseudocode for Pollard's ρ algorithm followed by an example.

Algorithm 11 Pollard's ρ algorithm for computing discrete logarithms

Require: A generator α, group element β and cyclic group G or order n

Ensure: The discrete logarithm $x = \log_\alpha \beta$

```
1. Set $x_0 \leftarrow 1, a_0 \leftarrow 0, b_0 \leftarrow 0$

2. for $i$ = 1,2,... do
     - Using the quantities $x_{i-1}, a_{i-1}, b_{i-1}$, and $x_{2i-2}, a_{2i-2}, b_{2i-2}$ computed
     previously, compute $x_i, a_i, b_i$ and $x_{2i}, a_{2i}, b_{2i}$ using the above three equa-
     tions
     if $x_i = x_{2i}$ then
        Set $r \leftarrow b_i - b_{2i}$ mod$n$
        if $r = 0$ then
           then terminate the algorithm with failure;
        end if
        otherwise, compute $x = r^{-1}(a_{2i} - a_i)$ mod$n$ and return($x$)
     end if

3. end for
```

We note here if Algorithm 11 fails (in the rare occasion) then the method can be repeated with random integers a_0, b_0 in the range [1, n - 1] and using $x_0 = \alpha^{a_0}\beta^{b_0}$ as the starting value.

Example 7. *Pollard's rho algorithm for logarithms of the subgroup G.*

The element $\alpha = 2$ is a generator of the subgroup G of order $n = 191$. Suppose $\beta = 228$. Partition the elements of G into three subsets according to the rule $x \in S_1$ if $x \equiv 1 \pmod 3$, $x \in S_2$ if $x \equiv 0 \pmod 3$, and $x \in S_3$ if $x \equiv 2$ (mod 3). The below table shows values of $x_i, a_i, b_i, x_{2i}, a_{2i}$ and b_{2i} at the end of each oteration of step 2 of Algorithm 11. Note that $x_{14} = x_{28} = 144$. Finally, compute $r = b_{14} - b_{28}$ mod 191 = 125, $r^{-1} = 125^{-1}$ mod191 = 136, and $r^{-1}(a_{28} - a_{14})$ mod191 = 110. Therefore, $\log_2 228 = 110$. □

Intermediate steps of Pollard's ρ algorithm in Example 7

i	x_i	a_i	b_i	x_{2i}	a_{2i}	b_{2i}
1	228	0	1	279	0	2
2	279	0	2	184	1	4
3	92	0	4	14	1	6
4	184	1	4	256	2	7
5	205	1	5	304	3	8
6	14	1	6	121	6	18
7	28	2	6	144	12	38
8	256	2	7	235	48	152
9	152	2	8	72	48	154
10	304	3	8	14	96	118
11	372	3	9	256	97	119
12	121	6	18	304	98	120
13	12	6	19	121	5	51
14	144	12	38	144	10	104

Fact 12 Take G to be a group of order of a prime number n. Assume a function $f : G \longrightarrow G$ behaves like a random function, it can then be anticipated that the running time of Pollard's ρ algorithm for discrete logarithms in G is $O(\sqrt{n})$ number of group operations. Further, the algorithm uses negligible space.

5.2.4 Pohlig-Hellman Algorithm

The last generic discrete logarithm algorithm known is called the Pohlig-Hellman algorithm and was initially discovered by R.Silver however was first published by S.Pohlig and M.Hellman, hence getting its name. It is also sometimes referred to as the "Silver Pohlig-Hellman" Algorithm.

It begins by using prime factorization to factorise the order n of the cyclic group G, where the prime factorisation of n is $n = p_1^{e_1} p_2^{e_2} ... p_r^{e_r}$. This is also denoted as $n = \prod_{i=1}^{k} p_i^{e_i}$ in which p_i corresponds to the distinct primes whilst e_i represents the natural exponents. The algorithm then uses the Chinese Remainder Therorem (§5.1.4) so that $\log_\alpha \beta$ can be recovered from $x_i, x_2, ..., x_i$, $\log_\alpha \beta$ modulo n where:

$$\begin{gathered} x_1 \equiv \log_\alpha \beta \pmod{p_1^{e_1}} \\ x_2 \equiv \log_\alpha \beta \pmod{p_2^{e_2}} \\ \vdots \\ x_i \equiv \log_\alpha \beta \pmod{p_i^{e_i}} \end{gathered}$$

Every x_i integer value is found from computing the digits $l_(e_i - 1)$ respectively of it's p_i-ary representation: $x_i = l_0 + l_1 p_i + ... + l_{e_i} - l_{p_1}^{e_i - 1}$ for which $0 \leq l_j \leq p_i - 1$.

To verify the correctness of Algorithm 12, first observe that in step 2.3 $\overline{\alpha}$ is of order q. It then yields that at iteration j of the next step, $\gamma = \alpha^{l_0 + l_{1q} + ... + l_{j-1} q^{j-1}}$. It therefore follows:

$$\begin{aligned}\overline{\beta} = (\beta\gamma)^{n/q^{j+1}} &= (\alpha^{x - l_0 - l_i q - ... - l_{j-1} q^{j-1}})^{n/q^{j+1}} \\ &= (\alpha^{n/q^{j+1}})^{x_1 - l_0 - l_1 q - ... - l_{j-1} q^{j-1}} \\ &= (\alpha^{n/q^{j+1}})^{l_j q^j + ... + l_{e-1} q^{e-1}} \\ &= (\alpha^{n/q})^{l_j + ... + l_{e-1} q^{e-1-j}} \\ &= (\overline{\alpha})^{l_j}\end{aligned}$$

which holds true the last equality as $\overline{\alpha}$ has order q. It therefore follows that $\log_{\overline{\alpha}} \overline{\beta} \equiv l_j$.

Algorithm 12 Pohlig-Hellman algorithm for computing discrete logarithms

```
Require: A generator $\alpha$, group element $\beta$ and cyclic group $G$ or order $n$
Ensure: The discrete logarithm $x = \log_\alpha \beta$

  1. Find the prime factorization of $n : n = p_1^{e_1} p_2^{e_2} ... p_r^{e_r}$, where $e_i \geq 1$

  2. for $i$ from 1 to $r$ do
       (Compute $x_i = l_0 + l_i p_i + ... + l_{e_1 - 1} p_i^{e_i - 1}$, where $x_i = x \bmod p_i^{e_i}$)
       2.1 (Simplify the notation) Set $q \leftarrow p_i$ and $e \leftarrow e_i$
       2.2 Set $\gamma \leftarrow 1$ and $l_{-1} \leftarrow 0$
       2.3 Compute $\overline{\alpha} \leftarrow \alpha^{n/q}$
       2.4 Now compute the $l_j$:
       for $j$ from 0 to $e$-1 do
         Compute $\gamma \leftarrow \gamma \alpha^{l_{j-1} q^{j-1}}$ and $\overline{\beta} \leftarrow (\beta\gamma^{-1})^{n/q^{j+1}}$
         Compute $l_j \leftarrow \log_{\overline{\alpha}} \overline{\beta}$ (e.g. using Algorithm 10; see (3) in the Note 7)
       end for
       2.5 Set $x_i \leftarrow l_0 + l_1 q + ... + l_{e-1} q^{e-1}$

  3. end for

  4. Use Gauss's Algorithm () to compute the integer $x$, $0 \leq x \leq n - 1$, such
     that $x \equiv x_i \pmod{p_i^{e_i}}$ for $1 \leq i \leq r$

  5. return($x$)
```

The below example demonstrates Algorithm 12 with artificially small parameters.

Example 8. *Pohlig-Hellman algorithm for logarithms in G*

Let $p = 251$. The element $\alpha = 71$ is a generator of the group G of order $n = 250$ ($n = p$ -1). Consider $\beta = 210$, then $x = \log_{71} 210$ is computed as follows.

1. The prime factorisation of n is $250 = 2 \times 5^3$.

2. (*Compute* $x_1 = x \bmod 2$) Compute $\overline{\alpha} = \alpha^{n/2} \bmod p = 250$ and $\overline{\beta} = \beta^{n/2} \bmod p = 250$. Then $x_1 = \log_{250} 250 = 1$.

3. (*Compute* $x_2 = x \bmod 5^3 = l_0 + l_1 5 + l_2 5^2$)

   - Compute $\overline{\alpha} = \alpha^{n/5} \bmod p = 20$.
   - Compute $\gamma = 1$ and $\overline{\beta} = (\beta\gamma^{-1})^{n/5} \bmod p = 149$. Using brute-force search (Algorithm 7), compute $l_0 = \log_{20} 149 = 2$.
   - Compute $\gamma = \gamma\alpha^2 \bmod p = 21$ and $\overline{\beta} = (\beta\gamma^{-1})^{n/25} \bmod p = 113$. Using brute-force search, compute $l_1 = \log_{20} 113 = 4$.
   - Compute $\gamma = \gamma\alpha^{4\times5} \bmod p = 115$ and $\overline{\beta} = (\beta\gamma^{-1})^{(p-1)/125} \bmod p = 149$. Using brute-force search, compute $l_2 = \log_{20} 149 = 2$.
   - Hence, $x_2 = 2 + 4 \times 5 + 2 \times 5^2 = 72$.

4. Finally, solve the pair of congruences $x \equiv 1 \pmod 2$, $x \equiv 72 \pmod{125}$ to get $x = \log_{71} 210 = 197$.

$\square$

Fact 13 Having factorised n, the Pohlig-Hellman Algorithm has a running time of $O(\sum_{i=1}^{r} e_i(\log n + \sqrt{p_i}))$ group multiplications.

Definition 15. *Smooth Integer*

Let B be a positive integer. An integer n is said to be $B-$*smooth*, or *smooth with respect to a bound* B, if all it's prime factors are $\leq B$.

Note 6: (*Efficiency of Pohlig-Hellman*) The Pohlig-Hellman algorithm is only effective if every prime divisor p_i of n is relatively small; if n is a smooth integer (Definition 15). The below example illustrates a group in which the Pohlig-Hellman algorithm is effective. Consider p a 107-digit prime, of the group G:

$$p = 227088231986781039743145181950291021585250524967592855\\ 96453269189798311427475159776411276642277139650833937.$$

The order of G is $n = p$ -1 $= 2^4 \times 104729^8 \times 224737^8 \times 350377^4$. As the largest prime divisor of p - 1 is only 350377, it is fairly straight-forward to calculate logarithms in this group witht the Pohlig-Hellman algorithm.

Note 7: (*miscellaneous*)

1. The Pohlig-Hellman Algorithm is the same as Shanks' Algorithm if the order n of group G is a prime.

2. It is necessary for the order n to be a smooth integer, or else the Pohlig-Hellman Algorithm becomes inefficient. Further, in step 1 of Algorithm 12 a factorising algorithm that finds small factors should first be employed. An example of such an algorithm is Pollard's ρ algorithm for prime factorisation which was mentioned earlier in this chapter when discussing Algorithm 11.

3. The storage needed for Algorithm 10 in step 2.4 can be overcome by using Pollard's ρ Algorithm (Algorithm 11).

4. Shanks' "Baby-Step Giant-Step" algorithm is viewed as hypothetical interest, as it is practical. On the other hand, Pollard has an equivalent time and the space is smaller and so there are two ways in which this can progress. The first is regarding the upper bound that the algorithm needs, to say, the order of the group, and the other is regarding the bound running-time and a validity is required. Yet, some arguments are still made about the assumption of approximate running time.

5.2.5 Further Reading

The previous section reviews the "generic" algorithms available today to solve the discrete logarithm problem. The information provided in this chapter is sufficient enough for readers to understand these algorithms, yet algorithm specific papers exist for Pollard's ρ algorithm for logarithms and the Pohlig-Hellman Algorithm, which study both in grave mathematical detail. If the reader so wishes to visit these texts and study these algorithms to a deeper understanding then please refer to [24] and [19].

5.3 Non-generic Discrete Logarithm Algorithm

This section outlines the Index-Calculus algorithm and illustrates how it works in two kinds of groups used in practical applications.

5.3.1 Index-Calculus Algorithm

This is known to be the most powerful method for computing discrete logarithms and is understood to give a subexponential-time algorithm. We cover the

Index-Calculus algorithm under two contexts; the first describes the method in the general setting of a cyclic group G, for whihc an algorithm follows, and the second illustrates the Index-Calculus Method in $\mathbb{F}_{2^n}^*$.

This algorithm begins by having to choose a relatively small subset of S of elements of G, known as the *factor base.* This selection is done in such a way that a large fraction of elements of G can be efficiently expressed as element products from S. The following algorithm proceeds to precompute a database of logarithms consisting of all elements in S, and then repeatedly use this database every time the logarithm of a certain group element is desired. Yet, we mention here that the next algorithm is incomplete for two reasons. The first being that no method is specified for choosing the factor base S. Secondly, a technique for efficiently generating relations outlined in Algorithm 13 is neither specified. For ease and reduction in size for the system of equations to solve in Step 3, it is necessary that the factor base S must be a subset of G that is small but not too small either (so that the anticipated trial number to generate a relation (5.5) is not exceedingly large). Adequate factor bases and techniques however, do exist for generating relations for some cyclic groups G and $\mathbb{F}_{2^n}^*$, in particular the multiplicative group G of a general finite field $\mathbb{F}_p$. These will be discussed in the next two sections.

The next page illustrates pseudocode for the Index-Calculus Algorithm in cyclic groups.

Algorithm 13 Index-Calculus algorithm for computing discrete logarithms in cyclic groups

Require: A generator α, group element β and cyclic group G or order n
Ensure: The discrete logarithm $y = \log_{\alpha}\beta$

1. *(Select a factor base S)* Choose a subset $S = [p_1, p_2, ..., p_t]$ of G such that a "significant proportion" of all elements in G can be efficiently expressed as a product of element form S.

2. *Collect linear relations involving logarithms of elements in S)*

   2.1 Select a random ingeter k, $0 \leq k \leq n$ -1, and compute α^k.

   2.2 Try to write α^k as a product of elements in S:

$$\alpha^k = \prod_{i=1}^{t} p_i^{c_i}, \text{ where } c_i \geq 0. \tag{5.4}$$

   If successful, take logarithms of both sides of the above equation to obtain a linear relation

$$k \equiv \sum_{i=0}^{t} c_i \log_{\alpha} p_i \text{ (mod} n). \tag{5.5}$$

   Repeat steps 2.1 and 2.2 until $t + c$ relations of the above form are obtained (c is a small positive integer, e.g. $c = 10$, such that te system of equations given by the $t + c$ relations has a unique solution with high probability).

3. (*Find the logarithms of elements in S*) Working modulo n, solve the linear system of $t + C$ equations (in T unknows) of the form (5.5) collected in step 2 to obtain the values of $\log_{\alpha} p_i, 1 \leq i \leq t$.

4. (*Compute y*)

   4.1 Select a random integer $k, 0 \leq k \leq n$ -1, and compute $\beta \times \alpha^k$.

   4.2 Try to write $\beta \times \alpha^k$ as a product of elements in S:

$$\beta \times \alpha^k \equiv \sum_{i=1}^{t} p_i^{d_i}, \text{ where } d_i \geq 0.$$

   If the attempt is unsuccesful then repeat 4.1. Otherwise, taking logarithms of both sides of the above equation yields $\log_{\alpha}\beta = (\sum_{i=1}^{t} d_i(\log_{\alpha} p_i - k)$ mod$n$; thus compute $y = (\sum_{i=1}^{t} d_i(\log_{\alpha} p_i - k)$ mod$n$ and **return**(y).

5.3.2 Index-Calculus Algorithm in $\mathbb{F}_p^*$

For finite fields $\mathbb{F}_p$ in which p is a prime, the first t prime numbers are used to select the factor base S. To generate a relation (5.4) α^k mod$p$ is first computed after which tiral division is used to verify if this intger is a product of primes in S. The example below outlines Algorithm 13 in $\mathbb{F}_p^*$ in a problem with artificially small parameters.

Example 9. *(Index-Calculus algorithm for logarithms in $\mathbb{F}_{229}^*$)*

Let $p = 229$. The element $\alpha = 6$ is a generator of $\mathbb{F}_{229}^*$ of order $n = 228$. Consider $\beta = 13$. The $\log_6 13$ is computed as follows, using the index-calculus technique.

1. The factor base is chosen to be the first 5 primes: S = [2,3,5,7,11].

2. The follwoing six relations consisting of the elements of the factor base are obtained (unsuccessful attempts are note shown):

$$\begin{gathered}
6^{100} \bmod 229 = 180 = 2^2 \times 3^2 \times 5 \\
6^{18} \bmod 229 = 176 = 2^4 \times 11 \\
6^{12} \bmod 229 = 165 = 3 \times 5 \times 11 \\
6^{62} \bmod 229 = 154 = 2 \times 7 \times 11 \\
6^{143} \bmod 229 = 198 = 2 \times 3^2 \times 11 \\
6^{206} \bmod 229 = 210 = 2 \times 3 \times 5 \times 7.
\end{gathered}$$

   The next six equations involving the logarithms of elements in the factor base, follow from the above relations:

$$\begin{gathered}
100 \equiv 2\log_6 2 + 2\log_6 3 + \log_6 5 \pmod{228} \\
18 \equiv 4\log_6 2 + \log_6 11 \pmod{228} \\
12 \equiv \log_6 3 + \log_6 5 + \log_6 11 \pmod{228} \\
62 \equiv \log_6 2 + \log_6 7 + \log_6 11 \pmod{228} \\
143 \equiv \log_6 2 + 2\log_6 3 + \log_6 11 \pmod{228} \\
206 \equiv \log_6 2 + \log_6 3 + \log_6 5 + \log_6 7 \pmod{228}.
\end{gathered}$$

3. Solving the linear system of six equations in five unknown (the logarithm x_i = $\log_6 p_i$) yields the solutions $\log_6 2 = 21$, $\log_6 3 = 208$, $\log_6 5 = 98$, $\log_6 7 = 107$, and $\log_6 11 = 162$.

4. Suppose that the integer $k = 77$ is selected. Since $\beta \times \alpha^k = 13 \times 6^{77}$ mod229 = $147 = 3 \times 7^2$, it follows that

$$\log_6 13 = (\log_6 3 + 2\log_6 7 \text{ - } 77) \bmod 228 = 117.$$

$\square$

5.3.3 Index-Calculus Algorithm in $\mathbb{F}_{2^n}^*$

For finite fields $\mathbb{F}_{2^n}^*$, the factor base S is selected as the set of all irreducible polynomials in $\mathbb{F}_2[x]$ where some prescribed bound b is at most taken as it's degree. a relation (5.4) is found by computing $\alpha^x \bmod f(x)$ then trial division is used to make sure this polynomial is a product of polynomials in S. Example 10 illustrates the index-calculus algorithm in $\mathbb{F}_{2^n}^*$ on a problem with once again artificially small parameters.

Example 10. *(Algorithm 13 for logarithms in $\mathbb{F}_{2^n}^*$)*

The polynomial $f(x) = x^7 + x + 1$ is irreducible over $\mathbb{F}_2$. Hence, the elements of the finite field $\mathbb{F}_{2^7}$ of order 128 can be represented as the set of all polynomials in $\mathbb{F}_2[x]$ of degree at most 6, where multiplication is performed modulo $f(x)$. The order of $\mathbb{F}_{2^7}^*$ is $n = 2^7$-1 $= 127$, and $\alpha = x$ is a generator of $\mathbb{F}_{2^7}^*$. Suppose $\beta = x^4 + x^3 + x^2 + x + 1$. Then $y = \log_x \beta$ can be computed as follows, using the index-calculus method.

1. The factor base is selected to be the set of all irreducible polynomials in $\mathbb{F}_2[x]$ of degree at most 3: $S = \{x, x+1, x^2+x+1, x^3+x+1, x^3+x^2+1\}$.

2. The following five relations involving elements of the factor base are obtained (unsuccessful attempts are not shown):

$$\begin{aligned} x^{18} \bmod f(x) &= x^6 + x^4 = x^4(x+1)^2 \\ x^{105} \bmod f(x) &= x^6 + x^5 + x^4 + x = x(x+1)^2(x^3+x^2+1) \\ x^{72} \bmod f(x) &= x^6 + x^5 + x^3 + x^2 = x^2(x+1)^2(x^2+x+1) \\ x^{45} \bmod f(x) &= x^5 + x^2 + x + 1 = x(x+1)^2(x^3+x+1) \\ x^{121} \bmod f(x) &= x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 = (x^3+x+1)(x^3+x^2+1). \end{aligned}$$

   From these relations, the following five equations involving the logarithms of elements in factor base (for notation efficiency, let $p_1 = \log_x x$, $p_2 = \log_x(x+1)$, $p_3 = \log_x(x^2+x+1)$, $p_4 = \log_x(x^3+x+1)$, and $p_5 = \log_x(x^3+x^2+1)$:

$$\begin{aligned} 18 &\equiv 4p_1 + 2p_2 \pmod{127} \\ 105 &\equiv p_1 + 2p_2 + p_5 \pmod{127} \\ 72 &\equiv 2p_1 + 2p_2 + p_3 \pmod{127} \\ 45 &\equiv 2p_2 + p_4 \pmod{127} \\ 121 &\equiv p_4 + p_5 \pmod{127}. \end{aligned}$$

3. Solving the linear system of five equations in five unknowns yields the values $p_1 = 1, p_2 = 7, p_3 = 56, p_4 = 31$, and $p_5 = 90$.

4. Suppose $k = 66$ is selected. Since:

   $$\beta\alpha^k = (x^4 + x^3 + x^2 + x + 1)x^{66} \bmod f(x) = x^5 + x^3 + x = x(x^2 + x + 1)^2,$$

   it follows that

   $$\log_x(x^4 + x^3 + x^2 + x + 1) = (p_1 + 2p_3 - 66) \bmod 127 = 47.$$

□

Note 8: *(Running-time of the Index-Calculus method)* The size t of the factor base should be selected thoughtfully in order to enhance the running time of the Index-Calculus method. Ideally, this choice is based in acquaintance regarding the allocation of smooth integer in the interval [1, p-1] for $\mathbb{F}_p^*$ and on the distribution of *smooth polynomials* (polynomials which all the have irreducibe factors of small degrees) amid polynomials in $\mathbb{F}_2[x]$ with degree less than m. Having an ideal t value the index-calculus algorithm as earlier explained for $\mathbb{F}_p^*$ and $\mathbb{F}_{2^n}^*$ has an anticipated running time of $\mathrm{L}_1[\frac{1}{2}, c]$ where $q = p$, q=2^n, and $c > 0$ is a constant.

Note 9: *(Fastest algorithms known for discrete logarithms in $\mathbb{F}_p^*$ and $\mathbb{F}_{2^n}^*$)* At present for $\mathbb{F}_p^*$, the best algorithm known for computing discrete logarithms is a combination of the index-calculus algorithm called the *Coppersmith algorithm* and has an anticipated running-time of $L_{2^n}[\frac{1}{3}, c]$ for some constant $c < 1.587$. For $\mathbb{F}_p^*$, the best known algorithm is the *number field sieve*, with an expected running time of $L_p[\frac{1}{3}, 1.923]$. Both of these techniques however, will not be considered in detail in this text, so readers are advised to refer to [33] for further detailed knowledge on the *Coppersmith algorithm* and the *number field sieve.*

Note 10: *(Parallelisation of the Index-Calculus algorithm)*

1. In the case of ideal choice of parameters, the phase that takes up the most time in the Index-Calculus algorithm is generally step 2 in algorithm 11, in which relations involving factor base logarithms are determined. This task is trivial to split between a processor network by allowing the processors to search for relations independently of each other. All the relations that are generated are gathered bu one central processor. Once a sufficient member of relations have been generated, then relations in step 3 of algorithm 11 can be solved on one single (even possibly parallel) computer.

2. For a given finite field the database of factor base logarithms is only required to be computed once. Relative to this, step 4 of algorithm 11 in which individual logarithms are calculated is exceptionally quicker.

Chapter 6

Implementation in C++

This chapter discusses the implmentation of the algorithm's discussed in the previous chapter. The programming language used was C++ and we illustrate what representations of finite fields have been considered for the implementations (Chapter 4 §4.5.1 and §4.5.2), what psuedocode was used, the testing procedure, the functionality of each preogram and their robustness and efficiency. Initially programming started with the Dev C++ Integrated Development Environment (IDE) but was later changed to Visual Studio 2010 and Eclipse IDE, the reasons for which will follow in this chapter. We begin by noting that all the "generic" discrete logarithm algorithms have been implemented in the chosen language, that is:

1. Brute-Force Search
2. Shanks' "Baby-Step Giant-Step" Method
3. Pollard's ρ Algorithm for logarithms (1978)
4. Pohlig-Hellman Algorithm (1978)

The pseudocode provided for each algorithm (including the Index-Calculus algorithm) in the previous chapter are all from [1], however various different sources have been referred to during implementation. For the Brute-Force search pseudocode from [32] was used, [1] was referred to for Shanks' "Baby-Step Giant-Step" Method, and the Pohlig-Hellman algorithm whilst `http://www-ti.informatik.uni-tuebingen.de/~reinhard/krypto/English/pollardrho_e.html` was used for Pollard's ρ algorithm for logarithms.

6.1 Implementation for $\mathbb{F}_p$

For the finite field $\mathbb{F}_p$, four algorithms have been implemented (as already mentioned) of *integer* representation for which no external libraries were used.

6.1.1 Functionality

The Brute-Force Search, Shanks' "Baby-Step Giant Step" Method, Pollard ρ algorithm for logarithms and Pohlig-Hellman algorithm all funtion in the same way with input of $\alpha \in G$, $\beta \in G$ and the order n of the cyclic group G. (To quickly recap, for this field α is a generator of the group G and β an element of the group G). Each algorithm solves the Discrete Logarithm Problem by finding the unique integer x such that $\alpha^x = \beta$.

Each implementation is an exectuable program, using command prompt to take in input arguments. The algorithms function such that upon execution a command line window is opened that:

1. Firstly asks the user to input the value of group element α (remember input in $\mathbb{F}_p$ is integer value):

   "Please enter a value for the generator α: ..."

2. Once having entered the value for α, the program then displays a message asking the user to input the value of β:

   "Please enter a value for the element β: ..."

3. After this, the last argument that needs to be inputted is the value of the gorup n.

   "Please enter a value for the order 'n' of the group:"

4. The program then displays the exponent value 'x' in the form of $\alpha^x = \beta$ or x=$\log_\alpha \beta$:

   "The discrete logarithm x is ..."

In the case when no x values exist, depending on the algorithm being used, according outputs will be displayed:

- Brute-Force displays 0.

- Shanks' "Baby-Step Giant Step" Method returns the following message if the multiplicative inverse is incomputable '*Multiplicative inverse cannot be computed, please check input!*', or if the value of α is not primitive, then the program will not display anything further information and terminates. For some strange cases where the multiplicative inverse cannot be calculated, the program will display the above message as well as a value for the discrete logarithm x. In this case, the value x is to be ignored as this value is incorrect.

- Pollards ρ algorithm for logarithms displays a '*no unique value x exists*' message and 0. When (in the rare case) the discrete logarithm 'x' is displayed as 0 alone without the message, it means that the unique integer x is 0.

- Pohlig-Hellman algorithm doesn't display any answer if values are not relatively prime or terminates if p has no prime factors.

Note 11: In all functions, the intermediate steps are NOT displayed, only the final answer is given, as we believe these were not neccessary to be known by the user. For intermediate steps please refer to the hyperlinks used during testing, or please see the examples provided for each algorithm in this text as they clearly illustrate the procedures step by step.

6.1.2 Efficiency

To make the programs more robust and efficient, the elapsed processor times have also been calculated. This is robust as the efficiency of software is judged in context of the processor time than compared to real time, giving a realistic time measure on the speed of each algorithm to a sufficient level of accuracy. The processor times are displayed in seconds as such:

"The processor time elapsed is: '...' seconds"

6.1.3 Testing and Debugging Procedures

The same testing procedure was carried out for all four algorithms:

- **Brute-Force Search** - This was the most trivial algorithm to implement from the four and due to this fact it was possible to calculate values using mental arithmetic (on paper), to which the program output values were then compared. It is yet evident that mentally, calculations would only be possible for restricted values, because for too large values the algorithm and mental calculation both become inefficient. As for wirtten examples in books, no such examples exist due to the trivial nature of the algorithm. Debugging consisted in the form of printing each line of caclulation until the final answer. This posed as a mechanism for backtracking and verifying mathematical error in the code were any to occur.

- **Shanks' "Baby-Step Giant-Step" Method** - The same technique was adapted for Shanks' "Baby-Step Giant-Step" Method as to the Brute-Force Search, in which an example from [1] was used to verify with during the debugging process. For fairness and correctness of the algorithm, various online examples using `http://www-ti.informatik.uni-tuebingen.de/~reinhard/krypto/English/shank_e.html`, an online Discrete Logarithm applet, were tested. The usefulness and advantage of the online applet will follow as the same resource hase been used for the next two algorithms too, the benefits of which are ditto for the former and latter. For further verification, other text book examples were used.

- **Pollard's ρ algorithm for logarithms and Pohlig-Hellman Algorithm** - For both the final two algorithms the hyperlinks provided at the start of this chapter were used, to recite `http://www-ti.informatik.uni-tuebingen.de/~reinhard/krypto/English/pollardrho_e.html` and `http://www-ti.informatik.uni-tuebingen.de/~reinhard/krypto/PohligHellman/english.html`. Both these pages provided online calculators for the Discrete Logarithm Problem for the Pollards ρ and Pohlig-Hellman algorithms and proved to be excellent resources during the testing stage as experiments could be carried out for all values and directly compared. Values were first tested on the online calculator then on the executable programs implemented. The bigger advantage gained by using the hyperlinks was that each line of computation was printed on screen along with the final answer, which proved extremely useful during the debugging stage. Yet, one thing was spotted during testing with the Pohlig-Hellman algorithm; most values obtained by both the online calculator and written program were the same, however for some examples this was not the case. Particular examples in different answers between both sources; we believe the only reason for this being the source of pseudocode on which the programs are based. The applet having provided it's own, yet as mentioned earlier, our implemented version using the pseudocode from [1], distinguishing the slight variation amongst both. It is was also seen that for a certain minority of cases with the Pohlig-Hellman algorithm, Eclipse IDE threw **'Range 1 error'**; 95% of the time a range 1 error is thrown due to a vector problem using g++.exe. This is a compiler issue beyond our control to resolve, however, is only thrown for a very few examples (as already mentioned) hence, can be passed without too much of a concern.

6.1.4 Elapsed Time

When testing, the elapsed times for each of the algorithms were taken in consideration. In the general case, the Pohlig-Hellman algorithm proved to take the longest, which was anticipated, yet their were cases where the elapsed time was computationaly quick. Conversely, at times Pollard's ρ algorithm for discrete logarithms and and Shank's method proved to take longer than the latter. An observation that was collected from the testing procedure was that the input values would reflect the elapsed time and for this reason, it wouldn't be appropriate to come to a concluded decision upon the time efficiency regarding which procedure from those stated would be best suited to solve the discrete logarithm problem.

Ofcourse, a further consideration and observation taken into account during this stage was on the various conditions posed by the different algorithms. To cite an example, in order for the Pohlig-Hellman algorithm to function, it is required that the value of the input n has prime factors, i.e. is factorisable, and that both α and β are relatively prime. Also for Shank's "Baby-Step Giant-Step" Method it is required that α is primitive and that the multiplicative inverse is computable, without which the entire algorithm is of no use. To further explain, for Pollard's ρ algorithm if the value of r is equivalent to 0 then the algorithm will terminate from this point; in all such situations, the elapsed time was disregarded as the final answer due to the procedure terminating and as a result the problem not being solved.

6.1.5 Implementation Issues

Two key issues were faced during the implementation stage for finite fields $\mathbb{F}_p$. In particular for Pollards ρ algortihm for logarithms for which a challenge was faced on what implementation to follow, as various different versions of pseudocode was seen. However, this was overcome once having found the online calculator for this algorithm as pseudocode was also provided with the working solution. Readers are to be reminded here that for this reason a different answer will be obtained to that shown in example 7 of this text if tested with the program written, the former being from [1] a different slightly different version of the algorithm to the actual source used. Note, example 7 was used in particular for fairness i.e. keeping to a single source for examples, further it illustrates the use of Pollars ρ algorithm to a high standard of understanding. As for Brute-Force Search, Shanks' "Baby-Step Giant Step" Method the only issue faced (and remains unresolved) was the unexpected appearance of a discrete logarithm value in the case where the multiplicative inverse in uncomputable. This probelm is rather strange and it is strongly believed that this error should not be occuring as the source code was repeatedly tested and varied, yet this remains to be a minor 'bug' in the program.

The second issue that was faced, was one beyond one's control, to say a compiler issue. Whilst implementing the Pohlig-Hellman algorithm, a problem regarding array size was confronted, in which an integer value k was assigned to a vector of size x in the following way:

int k = x.size()

Two arrays were then required, both the size of vector x, which would be initialised in C++ in the follwoing way:

m[k], a[k]

Yet, Visual Studio 2010 threw an error on this line stating, *"array size has to be constant value"*. However, when the same line of code was attempted in Eclipse IDE, no such errors were thrown. This clearly illustrates the difference in compiler set-up. Eclipse IDE uses *mingw* and *cygwin*, whilst Visual Studio 2010 makes use of the *ms compiler*. Initially, this was thought to be bad programming practice, however it was clarified to be a compiler issue after professional consultation.

6.2 Implementation for $\mathbb{F}_{2^n}$

For the finite field $\mathbb{F}_{2^n}$ initially a new library with a new class was to be constructed as C++ does not have a predefined type *Galois Field Element*. However, this idea was later dropped as an already Galois Field Arithmetic library had been written by an individual named Arash Partow and was available online to download and use. This was considerably helpful because it saved a substantial amount of time compared to wiritng one up from scratch. The external library was downloaded from `http://www.partow.net/projects/galois/` and comes with a free use permission licence under the guidelines and in accordance with the most current version of the "Common Public Licence".

For this representation the Brute-Force Search, Shanks' "Baby-Step Giant-Step" Method and Pollard's ρ algorithm for logarithms were implemented, however, a different version of pseudocode for the Brute-Force Search was used i.e. a precomputed table version in which a list/table is used to compare values as demonstrated in Shanks' "Baby-Step Giant-Step" Method. This version was used as it was much more efficient and accurate for the context $\mathbb{F}_{2^n}$ and the polynomial representation and the main advantage of this method is the instant solutions of subsequent discrete logarithms in the same group; only a single table lookup is needed. This pseudocode (Algorithm 14) is provided on the next page and the source was once again [32]:

Algorithm 14 Brute-Force Search - Precomputed Table Algorithm

```
Require: A generator $\alpha$, primitive element $\beta$ and a primitive polynomial $f(x)$
  of degree $n$
Ensure: The discrete logarithm $\alpha^x = \beta$
    1. First build the table such that $hash[\alpha^x] = x$ for $0 \leq x < 2^n - 1$
    2. $b \Leftarrow 1$
    3. for $x = 0$ to [table size] do
         hash$[b] \Leftarrow x$
         $b \Leftarrow b \times \alpha$
    4. end for
    5. Now perform the table lookup
    6. $x \Leftarrow$ hash$[\beta]$
    7. return $x$
```

This algorithm is exponential in both time and space complexities and the table holds 2^n - 1 values of size n which gives an asymptotic size of $O(n2^n)$.

6.2.1 Functionality

Functionality for the three algorithms were not exactly as desired due to the limitations posed by the external library, as implementation had to be done according to the library. This is one key disadvantage (under all contexts, not just for this project) of using external libraries.

The same implementation of Brute-Force Search, Shanks' "Baby-Step Giant-Step" Method and Pollard's ρ algorithm for logarithms for $\mathbb{F}_p$ were used, but just modified for use with the library. As implementations already existed, there was no neccessity to re-implement, as the same procedures applied to $\mathbb{F}_{2^n}$, however just under the context of a Galois Field.

Continuing with functionality, all three algorithms work in a different way to their twins. Due to the restricted functionality and simplicity of the library, all input arguments had to be hardcoded into the program, and modified accordingly. A new namespace **'galois'** had to be used to enable the functionality of library in addition to including the required header files. More on this will be explained in the section to follow.

In order to use the algorithms a primitive polynomial has to be initialised in the form of a vector in which the polynomial coefficients are to coded in ascending order. Hence the polynomial $p(x) = x^4 + x^3 + 0x^2 + x + 1$ is defined as such:

unsigned int poly[5] = {1, 1, 0, 1, 1}

The Galois Field is then setup, what is important in this is the exponent value e.g. 2^n. So, for a Galois Field of type 2^4 we use:

galois::GaloisField gf(4, poly)

Once the field has been set-up, the Galois Field Element needs to be initialised. The way in which this is done, is to reference an already initialised Galois field to pass to the field element and the field element's initial vector form value within the particular Galois field also has to be passed as such:

galois::GaloisFieldElement element1(&gf, 1)
galois::GaloisFieldElement element2(&gf, 2)

Note 12: In the case of this project, element 1 = α and element 2 = β. This therefore yeids:

galois::GaloisFieldElement α**(&gf, 2)**
galois::GaloisFieldElement β**(&gf, 4)**

where the value of α is always 2, and β is subject to change, however for fairness in testing β was fixed to 4. Variable n is replaced by o, still representing the order of the group, i.e. the number of elements. For $\mathbb{F}_p$ n denoted $n = p - 1$, likewise for F_{2^n} o denotes $o = (2^n) - 1$.

The remaining algorithms are the same, however, adapted where needed for convenience, execution and for Finite Field Arithmetic. The source codes **'ex_search_2n.cpp'**, **'BSGS_2n.cpp'** and **'Pollard_Rho_2n.cpp'** illustrate the use of the Galois Field Arithmetic library.

Note 13: A keen eye has to be kept when editing and changing the primitive polynomials and field size, in particualr for Shanks' "Baby-Step Giant-Step" Method and Pollard's ρ algorithm for logarithms. Care has to be taken these two algorithms because they both require various areas at which changes need to me made according to the value of the field size, and therefore when the field size is changed, these areas will also have to be modified (these lines have cleary been highlighted in the source code through commenting). Further, a change in field size would therefore mean a change in polynomial size i.e. the number of polynomial coefficients, and in the case this is forgotten to do, an 'unhandled win32 exception' occurs causing the program to crash. This exception deems extremely useful when such an error has occured and proves to be a beneficial debugging tool.

Note 14: For all three algorithms is it essential to use ONLY irreducible primitive polynomial coefficients or else they will crash displaying the already mentioned 'unhandled win32 exception'. If a mistake is made with entering the coefficient values and an exception is to occur, it may seem to the user that the program is not functioning when in reality this is not the case, hence, the use of irreducible primitive polynomials is to be ensured.

6.2.2 Efficiency

Once again, for the same reasons stated in §6.1.2 the elapsed processor time in calculating x for the algorithms are measured and displayed in the same format.

6.2.3 Testing and Debugging Procedures

For the testing procedure, irreducible (primitive) polynomials were used. For fairness, for each degree n, the primitive polynomials used were selected at random, the Galois Field Element α fixed at value 2 and β fixed at value 4. The primitive polynomials used were selected from 'test data', to say, a *table of irreducible polynomials for the modulus 2*, where n ranged from 1 to 10. The same debugging process to §6.1.3 was followed.

6.2.4 Elapsed Time

When testing with the Brute-Force Search for $\mathbb{F}_{2^n}$ rather interesting yet, anticipated results were obtained. The table on the next page outlines the test data used; the degree of the poynomial, the polyomial coefficients, and elapsed time in seconds, $n = 1$, 2 and 3 have been disregarded for accuracy and efficiency.

Brute-Force Search Results

Irreducible Polynomials for the Modulus 2		
n	**Polynomial coefficients**	**Elapsed processor time in secs**
4	10011	0.031s
	11001	0.031s
5	100101	0.032s
	101001	0.032s
	101111	0.015s
	110111	0.031s
	111011	0.015s
	111101	0.031s
6	1000011	0.031s
	1011011	0.016s
	1100001	0.032s
	1100111	0.016s
	1101101	0.031s
	1110011	0.016s
7	10000011	0.031s
	10001001	0.032s
	10011101	0.031s
	10100111	0.031s
	10111111	0.031s
	11000001	0.031s
	11101111	0.031s
	11110001	0.031s
8	100011101	0.031s
	101001101	0.032s
	110101001	0.609s
	111001111	0.610s
	111100111	0.610s
	111110101	0.594s
9	1000010001	1.125s
	1000100001	1.093s
	1010100101	1.093s
	1101111111	1.110s
	1100011111	1.094s
	1111001011	1.109s
10	10000001001	2.015s

The results shown in the table above are only a sample of the tested data used with the intention to illustrate the scope, efficieny and time difference amongst the calculations. An interesting observation was gathered from the testing procedure; the results illustrated show a genuine increase in time from $n = 8$, meaning, as there is an increase in polynomial size, the elapsed processor time will also accordingly increase. Further, the results gathered were tested one after another without any intermediate interference in ascending order, i.e. from $n = 4$ to $n = 10$, yet when the same test was carried out again reversely then a different result base was obtained.

For accuracy, the same test was again carried out on the following day to the above, however, in reverse order and it was discovered that the running processor times were different to those stated in the table. In fact, the elapsed time for larger polynomials i.e. when $n = 8$, 9 and 10, was 0.31 seconds, clearly meaning that if tests are repeatedly carried out in ascending order of polynomial degree then this will effect the elapsed calculation time for larger degree powers. A definite reason for this could not be confirmed, yet one of two assumptions could be made. Computationally, it was seen that for the former experiment the program took longer than usual time to execute for larger polynomials before an output was displayed on command line; a waiting time of 20 - 30 *real-time* secs. It could be understood that this waiting time is considered within the elapsed time, hence resulting in an overall longer time frame for calculation. A second assumption could be related to the amount of storage memory required by the program, hence reflecting the time and speed difference, the worst case scenario being an overload of memory. This was the key reason for why the table on the previous page contains only one polynomial for $n = 10$, it wasn't that no further data for this value was available but in fact, computation was taking a considerable amount of time. Besides, the maximum value taken by n in the test data was 10 and so it can be fair to say that the algorithm functions for all tested polynomial sizes. But in addition to this statement, the latter experiment too saw an equivalent waiting time before an output was displayed yet the elapsed time was revealed as 0.31 seconds, making the the former argument carry no weight.

Therefore, taking into account the observations, analysis and results from both tests, it can be accomplished that the average processor time taken by the Brute-Force Search under the finite field $\mathbb{F}_{2^n}$ is 0.31 seconds.

The results for Shanks' "Baby-Step Giant-Step" Method and Pollard's Rho algorithm for logarithms will now also be illustrated in the following pages.

Shanks' "Baby-Step Giant-Step" Method Results

Irreducible Polynomials for the Modulus 2		
n	**Polynomial coefficients**	**Elapsed processor time in secs**
4	10011	0.016s
	11001	0.031s
5	100101	0.031s
	101001	0.031s
	101111	0.015s
	110111	0.031s
	111011	0.031s
	111101	0.032s
6	1000011	0.015s
	1011011	0.031s
	1100001	0.015s
	1100111	0.031s
	1101101	0.031s
	1110011	0.031s
7	10000011	0.031s
	10001001	0.031s
	10011101	0.031s
	10100111	0.031s
	10111111	0.031s
	11000001	0.032s
	11101111	0.032s
	11110001	0.032s
8	100011101	0.016s
	101001101	0.031s
	110101001	0.031s
	111001111	0.015s
	111100111	0.032s
	111110101	0.031s
9	1000010001	0.031s
	1000100001	0.031s
	1010100101	0.015s
	1101111111	0.031s
	1100011111	0.031s
	1111001011	0.031s
10	10000001001	0.016s

Pollard's ρ algorithm for logarithms Results

Irreducible Polynomials for the Modulus 2		
n	**Polynomial coefficients**	**Elapsed processor time in secs**
4	10011	0.031s
	11001	0.031s
5	100101	0.031s
	101001	0.031s
	101111	0.047s
	110111	0.031s
	111011	0.046s
	111101	0.031s
6	1000011	0.031s
	1011011	0.032s
	1100001	0.031s
	1100111	0.046s
	1101101	0.031s
	1110011	0.031s
7	10000011	0.031s
	10001001	0.031s
	10011101	0.031s
	10100111	0.031s
	10111111	0.046s
	11000001	0.047s
	11101111	0.031s
	11110001	0.031s
8	100011101	0.032s
	101001101	0.031s
	110101001	0.047s
	111001111	0.047s
	111100111	0.047s
	111110101	0.047s
9	1000010001	0.031s
	1000100001	0.047s
	1010100101	0.047s
	1101111111	0.047s
	1100011111	0.047s
	1111001011	0.047s
10	10000001001	0.046s

Once again the results for the final two tables illustrate that the average running times for both Shanks' "Baby-Step Giant-Step" Method and Pollard's ρ algorithm for logarithms is also 0.031 processor seconds. From this data it can be concluded that all three algorithms in $\mathbb{F}_{2^n}$ have the same running time, providing the same efficiency level for each.

Yet, as for robustness, two particular analysis' were made. Firstly, when using Shanks' Method, in addition to the testing stage, it was seen that when the value of β is assigned to the value 9 or above, regardless of the field size or primitive polynomial, the program will execute, however, will not display an answer; only a description of the program is displayed on screen. This is the case when the multiplicative inverse cannot be computed for the algorithm meaning the program will no longer continue and terminate. As with it's sister implementation for $\mathbb{F}_p$, a message was conditioned to be displayed, however the same program was faced, this time to a much higher degree. Depsite the **if** and **else** statements, the message displayed for every answer and it deemed fit to therefore, remove the message entirely as this is not at all professional or efficient.

Secondly, regarding Pollard's ρ algorithm, this was also tested using the results illustrated in the corresponding table and strangley enough, for all primitive polynomials, field sizes and values for β, no discrete logarithm x existed due to the value of r equating 0. For this reason, one may argue that the algorithm does not function, however in defence it can be clearly said, the algorithm does indeed function correctly as different values have different elapsed times, and further, as an increase in field size is seen, a real-time wait of 20-30 secs is also experienced as explained already for the Brute-Force Search. This clearly identifies that the program is in fact running and functioning as required, however as the algorithm states, certain conditions allow for the answer specified. Another assumption that could be made from this result is that Pollard's ρ algorithm for logarithms is not applicable to the finite field $\mathbb{F}_{2^n}$. We say this as till date, all uses of this algorithm have been illustrated under $\mathbb{F}_p$, and so it cannot be said for sure how accurate this assumption could be. Yet, to determine this, a considerable amount of time would be needed, and hence this could become a foundation for future work.

In summary, all three algorithms function as required and are of equivalent efficiency, however, from the three the Brute-Force Search can be believed to be the most reliable in providing an answer/output, due to the conditions and reasons discussed above and put forward by the latter procedures. Nevertheless, this does not mean that only Brute-Force is to be used, as the other algorithms may prove to be more suited based on what output/result is required.

6.2.5 Implementation Issues

Once having downloaded the external library, it was first tested for use with only the header files alone. This trial was first carried out in Dev C++ IDE, however a link error was faced as no library could be found, or connected to.

The same trial was then carried out in Visual Studio 2010 and yet again a link error was faced. It was then discovered that despite having downloaded the external library, the *.lib* file that makes the library was missing.

A downside to the library was that poor documentation was provided for it, which at times posed to be very inefficient. This meant that the static library had to be built using the downloaded files with an IDE.

Initially it was believed that the library was corrupt however after thorough research and professional consultation, it came forward that no *.lib* file existed.

Visual Studio 2010 was then used to create a static Galois field library and then referenced and linked to the associated source files after which the source successfully compiled and ran.

This experiment was also tried in Eclipse IDE by creating the static library again, linking, building and running **'ex_search_2n.cpp'**, **'BSGS_2n.cpp'** and **'Pollard_Rho_2n.cpp'**. Yet, once again, the difference in compilers was evident as the programs terminated without execution in Eclipse IDE, whilst executing smoothly in Visual Studio 2010.

Chapter 7

Conclusion

In conclusion, it can be summarised that there is room for further research and questioning upon the level of security for fields $\mathbb{F}_{2^n}$, in particular for cryptographic applications. The discrete logarithm problem will therefore continue to be a subject of research and this text reflects and surveys it's importance.

Through the study of known algorithms, this text confronts the discrete logarithm problem under two types of finite fields $\mathbb{F}_p$ and $\mathbb{F}_{2^n}$. It is evident why discrete logarithms in $\mathbb{F}_p$ carry a much easier implementation level and provide a greater and increased level of security in cryptographic applications than $\mathbb{F}_{2^n}$.

As a whole, the project has progressed smoothly. Initially, focus began on the field $\mathbb{F}_{2^n}$, yet as time progressed $\mathbb{F}_p$ was also bought into consideration. Having programmed in both fields raised practical awareness about both implementation issues and compiler differences, all of which have clearly been discussed and presented in this thesis. Other than those stated, no other real challenges were faced, and development was checked on a fortnightly basis to ensure consistency was maintained. Nevertheless, it can rightfully be said that there is still scope of future improvement for this project.

Theoretical research has stated the Index-Calculus method to be the best and most efficient algorithm known till date for $\mathbb{F}_{2^n}$, yet time did not allow this statement to be proven through implementation. Therefore, a future improvement to the project could involve implementing and investigating into solving the discrete logarithm problem using the Index-Calculus method, firstly for $\mathbb{F}_p$ and if possible $\mathbb{F}_{2^n}$. It would also have been of benefit to implement the 'Pohlig-Hellman' algorithm for the field $\mathbb{F}_{2^n}$ for this project; this was a thought considered and attempted but due to the difference in compilers this could not be completed. As for the mathematical, analytical and written content provided, this can be passed as sufficient; to summarise - for future improvements, effort and endeavor is needed practically rather than theoretically.

To say the least, generally speaking, the results produced in support and completion of this project, in the timescale allocated, can be taken as fruitful and understood to have fulfilled the purpose of the **'Discrete Logarithm Problem in finite fields and applications to cryptography'**.

References

[1] A.J.Menezes, P.C.Van Oorschot, S.A.Vanstone "Handbook of Applied Cryptography" *CRC Press LLC*, 1997

[2] W.Stallings "Cryptography and Network Security - Principles and Practice (Fifth Edition)" *Pearson Education Inc.*, 2011

[3] D.R.Stinson "Cryptography - Theory and Practice" *CRC Press LLC*, 2000

[4] W.R.Scott "Group Theory" *Dover Publications, Inc. New York*, 1987

[5] S.H.Weintraub "Galios Theory" *Springer Science+Business Media, Inc. USA*, 2006

[6] G.Smith and O.Tabachnikova "Topics in Group Theory" *Springer-Verlag London Limited*, 2000

[7] R.J.Mceliece "Finite Fields for Computer Scientists and Engineers" *Kluwer Academic Publishers. USA*, 1987

[8] B.Schneier "Applied Cryptography, SECOND EDITION: Protocols, Algorithms and Source Code in C" *John Wiley and Sons, Inc.*, 1996

[9] I.F.Blake, R.C.Mullin and S.A.Vanstone "Computing Logarithms in GF(2^n)" *Advances in Cryptology-BERLIN: SPRINGER, (Vol 196)*, 1985

[10] O.Schirokauer, D.Weber and T.Denny "Discrete Logarithms and the Effectiveness of the Index Calculus Method" *Department of Mathematics - Oberlin College, USA, Universität des Saarlandes, Germany*

[11] W.Diffie and M.E.Hellman "New Directions in Cryptography" *IEEE Transactions on Information Theory (Vol-IT 22/6)*, 1976

[12] D.Coppersmith "Fast Evaluation of Logarithms in Fields of Characteristic Two" *IEEE Transactions on Information Theory (Vol-IT 30/4)*, 1984

[13] E.Taher "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms" *IEEE Transactions on Information Theory, (Vol-IT 31/4)*, 1985

[14] E.Taher "A Subexponential-Time Algorithm for Computing Discrete Logarithms over GF(p^2)" *IEEE Transactions on Information Theory (Vol-IT 31/4)*, 1985

[15] H.Niederreiter "A Short Proof for Explicit Formulas for Discrete Logarithms in Finite Fields" *AAECC (Vol 1), Springer-Verlag*, 1990

[16] L.Rudolf "Computational Problems in the Theory of Finite Fields" *AAECC (Vol 2), Springer-Verlag*, 1991

[17] A.J.Menezes and S.A.Vanstone "A Note on Cyclic Groups, Finite Fields, and the Discrete Logarithm Problem" *AAECC (Vol 3), Springer-Verlag*, 1992

[18] G.Meletiou and G.L.Mullen "A Note on Discrete Logarithms in Finite Fields" *AAECC (Vol 3), Springer-Verlag*, 1992

[19] A.J.Ly Thiong "A Serial Version of the Pohlig-Hellman Algorithm for Computing Discrete Logarithms" *AAECC (Vol 4), Springer-Verlag*, 1993

[20] A.M.Odlyzko "Discrete Logarithms: The Past and the Future" *Designs, Codes and Cryptography (Vol 19), Kluwer Academic Publishers, Boston*, 2000

[21] B.A.Lamacchia and A.M.Odlyzko "Computation of Discrete Logarithms in Prime Fields" *Designs, Codes and Cryptography (Vol 1), Academic Publishers*, 1991

[22] R.L.Bender and C.Pomerance "Rigorous discrete logarithm computations in finite fields via smooth polynomials" *AMS/IP Studies in Advanced Mathematics (Vol 7)*, 1998

[23] L.M.Adleman and J.Demarrais "A Subexponential Algorithm for Discrete Logarithms over all Finite Fields" *Mathematics of Computation (Vol 61/203)* , 1993

[24] E.Teske "Speeding up Pollards Rho Method for computing Discrete Logarithms" *Technische Universität Darmstadt, Institut für Theoretische Informatik, Alexander-Strasse 10, 64283, Darmstadt, Germany*

[25] H.Davenprot "Bases for Finite Fields" *J.London Math Society, (Vol 43)*, 1968

[26] K.S.McCurley "The Discrete Logarithm Problem" *American Mathematical Society - Proceedings of Symposis in Applied Mathematics, (Vol 42)*, 1990

[27] A.M.Odlyzko "Discrete Logarithms in Finite Fields and their Cryptographic Significance" *AT and T Laboratories, New Jersey*

[28] C.Pomerance "Elementary thoughts on discrete logarithms" *Algorithmic Number Theory, MSRI Publications, (Vol 44)*, 2008

[29] E.D.Win, A.Bosselaers, S.Vandenberghe, P.D.Gersem, J.Vandewalle "A Fast Software Implementation for Arithmetic Operations in GF(2^n) (PRE-PRINT)" *Katholieke Universiteit, Leuven, ESAT-COSIC, K. Mercieriaan 94, B-3001 Henverlee, Belgium*

[30] J.Hoffstein, J.Pipher and J.H.Silverman "An Introduction to Mathematical Cryptography" *Mathematics 158, Brown University - Fall 2004*, Preliminary Version

[31] R.Knight "Individual Bit Security of the Discrete Logarithm: Theory and Implementation Using Oracles" *Master Thesis*, 2007

[32] J.Mihalcik "An analysis of Algorithms for solving Discrete Logarithms in fixed groups" *Thesis*, Naval Postgraduate School, Monterey, California, 2010

[33] L.Maurits "Public Key Cryptography using Discrete Logarithms in Finite Fields: Algorithms, Efficient Implementation and Attacks" *BSc Thesis*, The University of Adelaide, Australia

Appendix A

Source Codes

A.1 Algorithms for $\mathbb{F}_p$

A.1.1 Brute-Force Search

```
/*------------------------ex_search.cpp---------------------

Implementation of the Exhaustive Search/Brute force method for
computing the unique integer x such that alpha^x=beta for the
Discrete Logarithm Problem in GF(p), with cyclic group G, order
n, generator alpha and group element beta.

Code written by Divyesh B Chudasama on 07/07/2012.
Copyright (c) 2012

--------------------------ex_search.cpp--------------------*/

/*----------------Exhaustive Search Method-------------------*/

#include <stdio.h>
#include <time.h>
#include "std_lib_facilities.h"

int ex_search(int alpha, int beta, int n){ // Function Exhaustive Search

int b = 1;
int x = 0;

while (beta != b){
      b = b*alpha;
      x = x + 1;
      }

      return x;
}

/*----------------End of Exhaustive Search Method----------------*/

/*------------------------Main Function------------------------*/

int main(){

int alpha;
int beta;
int n;
int ans =0;

    double time, timedif; // Double is used here to show small values

    time = (double) clock(); // Get the intial time
    time = time/CLOCKS_PER_SEC;  // Time in seconds

    cout<<"\n Divyesh B Chudasama"
          "\n Copyright (c) 2012";

    cout<<"\n \n Implementation of the Exhaustive Search Method for"
          "\n computing the unique integer x such that"
          "\n x = log_{alpha}beta for the Discrete Logarithm"
    "\n Problem in GF(p), with cyclic group G,"
    "\n order n, generator alpha and group element beta. ";

    cout<<"\n \n Please enter a value for the generator 'alpha': ";
    cin>>alpha;
    cout<<"\n Please enter a value for the element 'beta': ";
    cin>>beta;
    cout<<"\n Please enter a value for the order 'n' of the group:";
    cin>>n;
    ans = ex_search(alpha, beta, n); // Calling Exhaustive Search

    cout<<endl<<"\n The discrete logarithm x is: "<<ans<<endl<<endl;

    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time;
    // Calling clock() function from start of program, subtracting
    // its return value, then to obtain the final processor time
    // in seconds, divide the value returned by clock() by
    // CLOCK_PER_SEC

    cout<<"\n The processor time elapsed is: "<<timedif<<" seconds"
    <<endl<<endl;

    keep_window_open();

    return 0;
}

/*------------------------End of Main Function--------------------*/
```

A.1.2 Shanks' "Baby-Step Giant-Step" Method

```
/*----------------------------BSGS.cpp-----------------------------

Implementation of Shanks' Baby-Step Giant Step Method for computing
the unique integer x such that x = log_{alpha}(beta) for the Discrete
Logarithm Problem, in GF(p), with cyclic group G, order n, generator
alpha and group element beta.

Code written by Divyesh B Chudasama on 17/06/2012.
Copyright (c) 2012

----------------------------BSGS.cpp----------------------------*/

/*----------------------Variable Declarations---------------------*/

#include <stdio.h>
#include "std_lib_facilities.h"
```

```
#include <iostream>
#include <math.h>
#include <list>
#include <time.h>

using namespace std;

/**************************Functions****************************/

/*-----------------Multiplicative inverse in Z[n]----------------*/

struct  euclidean { // Creating a structure called euclidean

        int a; // First element in struct
        int b; // Second element in struct
        int c; // Third element in struct

};

euclidean extended_euclidean(int a, int n) // Function
//extended_euclidean that will return type euclidean

{
    // Using the extended Euclidean algorithm to find integers x
    //such and y such that ax+ny = d, where d=gcd(a,n)

    int x,y,d,q,r;

    //Step 1.
    if(n==0){
    d=a; // d<-a
    x=1; // x<-1
    y=0; // y<-0

    euclidean result={d,x,y}; // In the struct euclidean,
    //elements 1,2 and 3 are d,x and y respectively

    return result; // return result
}
```

```
    //Step 2.
    int x2=1; // x2<-1
    int x1=0; // x1<-0
    int y2=0; // y2<-0
    int y1=1; // y1<-1


    //Step 3.
    while (n>0) { // While n>0 do the following:

          q=a/n; // q<-a/n
          r=a-(q*n); // r<-a-(q*n)
          x =(x2-(q*x1)); // x<-(x2-(q*x1))
          y=y2-(q*y1); // y<-y2-(q*y1)
          a=n; // a<-n
          n=r; // n<-r
          x2=x1; // x2<-x1
          x1=x; // x1<-x
          y2=y1; // y2<-y1
          y1=y; // y1<-y
    }

    //Step 4.
    d=a; // d<-a
    x=x2; // x<-x2
    y=y2; // y<-y2

    euclidean result={d,x,y}; // In the struct euclidean,
    //elements 1,2 and 3 are d, x and y respectively

    return result; // return result

}   // This ends the extended Euclidean's algorithm

int mult_inv(int alpha,int n){

    // Making a new structure
    euclidean result_new = extended_euclidean(alpha,n); //Calling
    // the function extended_euclidean and assigning it as type
    //euclidean

    if(!(result_new.a>1)){ // If d (the first element in the struct)
    //is greater 1 then a^(-1) mod n does not exist

    return result_new.b;
    // Return x (the second element in the struct)
    }

    else{

    cout<<"\n \n Multiplicative inverse cannot be computed"
          "\n please check input!"<<endl;

    return result_new.a;
}
}

/*--------------End of Multiplicative inverse in Z[n]-------------*/

/*----------------Square and Multiply algorithm-------------------*/

int s_and_m(int alpha, int A, int n){

    int b=1;

    while(A>0) // While A>0 do:
    {
              if ((A-1)%2==0) // If A=1{mod2}
              {
                              b=b*alpha; //b=b*alpha
                              b=b%n; // b=(b*alpha)modn
              }
              alpha=alpha*alpha; // alpha=alpha^2
              alpha=alpha%n; // alpha=alpha^2(modn)
              A=int(A/2); // A=[A/2]
    }
    return b; // Return b

}

/*-----------------End of Square and Multiply--------------------*/

/*----------------Big-Step Giant-Step function-------------------*/

int bsgs(int alpha, int beta, int n){
    int m;
    int j;
    int p=n+1;
    int i;
    int x;

//Step 1.
m=int(sqrt(n))+1; //  Setting variable m to the square root of n
//then add 1

//Step 2.
vector<int> B1; // B1 is a list of entries of values of size m
vector<int> B2; // Second list for sorting, this is the same as B1

/*-----------Baby Step: B=(j, (alpha^j)modp) for 0<=j<m-----------*/

double a;
a = double(alpha);
for(j=0;j<m;j++)
{

      B1.push_back((int(pow(a, j)))%p); // Adds a new element at each
      // iteration to list B1, where the second element from each
      // pair is (alpha^j)modp.
      B2.push_back((int(pow(a, j)))%p); // Adding elements to list B2

                 }

sort(B2.begin(), B2.end()); // Sorting the second list
```

```
//Step 3.
int c=mult_inv(alpha,p); // Assigning integer c to multiplicative
//inverse of (alpha^-1)modp

double e=double(s_and_m(c,m,p)); // Using square and multiply method
//to calculate (alpha^-m)modp.

/*--------------------End of Baby Step--------------------------*/

/*------Giant Steps: gamma=(beta)(alpha^(-m*i))modp, 2)gamma^i where
i=1,2,3..... and then comparing the values with the BabyStep results
to find a common value---------*/

//Step 4.
int gamma;

bool endwhile=true;
int z;
i=0;
while(endwhile){

              gamma=s_and_m(e,i,p); // square and multiply for values
              //(e^i)modp
              gamma=gamma*beta; // gamma = gamma*beta
              gamma=gamma%p; // gamma = (gamma)modp

              for(int l=0; l<m; l++){

              if(B2[l]==gamma){ // Comparing values from B2 to gamma
                            endwhile=false;
                            z=i;

              for (int delta=0; delta<m; delta++){

                  if (B2[l]==B1[delta]){ // Comparing lists B2
                  //and B1 to find the common component
                  //in both

                  j=delta;
                  //cout << j << endl;
                  }
                            }
              }
              }
              gamma=0;
              i++;
              }
              //return z;

              x=(z*m)+j;  // x=(z*m)+j

                         return x;

                         }

/*---------------------End of Giant Step----------------------*/

/*------------End of Baby-Step Giant-Step Function-------------*/

/***********************End of Functions***********************/

/*-----------------------Main Function-------------------------*/

int main(){

int alpha;
int beta;
int n;

    double time, timedif; // Double is used here to show small values

    time = (double) clock(); // Get the intial time
    time = time/CLOCKS_PER_SEC;  // Time in seconds

    cout<<"\n Divyesh B Chudasama"
          "\n Copyright (c) 2012";

    cout<<"\n \n Implementation of Shanks' Baby-Step Giant-Step Method"
```

```
          "\n for computing the unique integer x such that"
          "\n x = log_{alpha}beta for the Discrete Logarithm Problem"
          "\n in GF(p), with cyclic group G, order n, generator alpha"
          "\n and group element beta. ";

    cout<<"\n \n Please enter a value for the generator ‘alpha’: ";
    cin>>alpha;
    cout<<"\n Please enter a value for the element ‘beta’: ";
    cin>>beta;
    cout<<"\n Please enter a value for the order ‘n’ of the group:";
    cin>>n;

    int x=bsgs(alpha,beta,n); // Calling Baby-Step Giant-Step

    cout<<endl<<"\n The discrete logarithm x is: "<<x<<endl<<endl;

    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time;
    // Calling clock() function from start of program, subtracting
    // its return value, then to obtain the final processor time
    //in seconds, divide the value returned by clock()
    //by CLOCK_PER_SEC

    cout<<"\n The processor time elapsed is: "<<timedif<<" seconds"
    <<endl<<endl;

    keep_window_open();

    return 0;
}

/*-------------------End of Main Function----------------------*/
```

A.1.3 Pollard’s ρ algorithm for logarithms

```
/*---------------------pollard_rho.cpp--------------------------

Implementation of Pollard’s Rho Method for computing the unique
integer x such that x = log_{alpha}(beta) for the Discrete Logarithm
```

```
Problem in GF(p), with cyclic group G, order n, generator alpha
and group element beta.

Code written by Divyesh B Chudasama on 09/07/2012.
Copyright (c) 2012

----------------------pollard_rho.cpp---------------------------*/

#include <stdio.h>
#include <time.h>
#include "std_lib_facilities.h"

/*************************Functions***************************/

/*---------------Set functions for S1, S2 and S3----------------*/

int alpha;
int beta;
int n;

int f(int x){// Function f(x_i) for sequence of group elements
             //x_(i+1) where, x=(x_i)

    if(x%3==0){ // If x_i is an element of S2
                x=(x*x)%n;// x_(i+1)=((x_i)^2)modn
                }

                else if(x%3==1){ // If x_i is an element of S3
                     x=(alpha*x)%n; // x_(i+1)=(alpha*(x_i))modn
                     }

                else if(x%3==2){ // If x_i is an element of S1
                     x=(beta*x)%n; // x_(i+1)=(beta*(x_i))modn
                     }
                     return x;
                     }

int g(int x, int a){ // Function g(x_i,a_i) for sequence of
                     //integers a_(i+1) where x=(x_i) and
                     //a=(a_i)/a_(i+1)

    if(x%3==0){ // If x_i is an element of S2
               a=(2*a); // a_(i+1)=(2*(a_i))
               }

               else if(x%3==1){ // If x_i is an element of S3
                    a=(a+1); // a_(i+1)=(a_i)+1
                    }

               else if(x%3==2){ // If x_i is an element of S1
                    a=a; // a_(i+1)=a_i
                    }
                    return a;
                    }

int h(int x, int b){ // Function h(x_i,b_i) for sequence of
                     //integers b_(i+1)
                     // where x=(x_i) and b=(b_i)/b_(b+1)

    if(x%3==0){ // If x_i is an element of S2
               b=(2*b); // b_(i+1)=(2*(b_i))
               }

               else if(x%3==1){ // If x_i is an element of S3
                    b=b; // b_(i+1)=(b_i)
                    }

               else if(x%3==2){ // If x_i is an element of S1
                    b=(b+1); // b_(i+1)=(b_i)+1
                    }
                    return b;
                    }

/*------------End of Set functions for S1, S2 and S3-----------*/

/*--------------------Pollard's Rho function-------------------*/

int pollard_rho(int alpha, int beta, int n){
```

```
int i;

vector<int> x; // Vector list x
vector<int> a; // Vector list a
vector<int> b; // Vector list b

x.push_back(1); // Adding entry x_0=1 to the vector list x
a.push_back(0); // Adding entry a_0=0 to the vector list a
b.push_back(0); // Adding entry b_0=0 to the vector list b

int X;
int A;
int B;
int r;
int c;
int d;
int e;
int z = 0;
int p=n-1;


for(i=1; i<n; i++){
// Computing the values x_i, a_i and b_i using the quantities
// of x_(i-1), a_(i-1) and b(i-1) computed previously from
// functions f, g and h

         c=f(x[i-1]); // c=f(x_(i-1))

         x.push_back(c); // Each iterative value of c is added
         // to vector list x

         d=(g(x[i-1], a[i-1]))%p; // d=g(x_(i-1),a_(i-1))
         a.push_back(d); // Each iterative value of d is
         //added to vector list a

         e=(h(x[i-1], b[i-1]))%p; // e=h(x_(i-1),b(i-1))
         b.push_back(e); // Each iterative value of e is
         //added to vector list b
         }

    for(i=1; i<n; i++){
    // Now computing x_(2i), a_(2i) and b_(2i)

         if(x[i]==x[2*i]){

         X=f(f(x[(2*i)-2])); // X=x_(2i)

         A=(g(f(x[(2*i)-2]), g(x[(2*i)-2], a[(2*i)-2])))%p;
         // A=a_(2i)

         B=(h(f(x[(2*i)-2]), h(x[(2*i)-2], b[(2*i)-2])))%p;
         // B=b_(2i)

         break;
         }
         }
    for(i=1; i<n; i++){
       if( x[i] == X ){ // If x_i = x_(2i)

            r=(b[i]-B); // r=((b_i)-b_(2i))

       if(r==0){ // If r=0

            cout<<"\n 'Pollard's rho method failed to find unique
                  "\n integer x as r=0!"
                  "\n No Discrete Logarithm x for the above values!'"
                  <<endl;

            return r;
            break; // Terminate algorithm with failure
            }

            else{

                 z=(a[i]-A)/(B-b[i]);
                 // z=(((a_i)-a_(2i))/(b_(2i)-b_i))
                 z= z%p;
                 if(z<0){ // If z is -ve
                         z=(p+z); // then mod(n-1)+z
                         }
                         }
                 return z;
                 break;
                 }

}
}

/*------------------End of Pollard's Rho function-----------------*/

/**********************End of Functions*************************/

/*-------------------------Main function-------------------------*/

int main(){

    double time, timedif; // Double is used here to show small values

    time = (double) clock(); // Get the intial time
    time = time/CLOCKS_PER_SEC;  // Time in seconds

    cout<<"\n Divyesh B Chudasama"
          "\n Copyright (c) 2012";

    cout<<"\n \n Implementation of Pollard's Rho Method for computing"
          "\n the unique integer x such that x = log_{alpha}beta"
          "\n for the Discrete Logarithm Problem in GF(p), with cyclic"
          "\n group G, order n, generator alpha and group element"
          "\n beta. ";

    cout<<"\n \n Please enter a value for the generator 'alpha': ";
    cin>>alpha;
    cout<<"\n Please enter a value for the element 'beta': ";
    cin>>beta;
    cout<<"\n Please enter a value for the order 'n' of the group:";
    cin>>n;

    int y=pollard_rho(alpha, beta, n); //Calling Pollard Rho

    cout<<endl<<"\n The discrete logarithm x is: "<<y<<endl<<endl;

    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time;
    // Calling clock() function from start of program, subtracting
    // its return value, then to obtain the final processor time in
    // seconds, divide the value returned by clock() by CLOCK_PER_SEC

    cout<<"\n The processor time elapsed is: "<<timedif<<" seconds"
    <<endl<<endl;

    keep_window_open();

    return 0;
}

/*---------------------End of Main function---------------------*/
```

A.1.4 Pohlig-Hellman Algorithm

```
/*---------------------Pohlig_Hellman.cpp-------------------------

Implementation of the Pohlig-Hellman Algorithm for computing the
unique integer x such that x = log_{alpha}(beta) for the Discrete
Logarithm Problem,in GF(p), with cyclic group G, order n, generator
alpha and group element beta.

Code written by Divyesh B Chudasama on 12/08/2012.
Copyright (c) 2012

-----------------------Pohlig_Hellman.cpp-----------------------*/

#include <stdio.h>
#include <iostream>
```

```
#include <cmath>
#include <stdlib.h>
#include <time.h>
#include "std_lib_facilities.h"

/**************************Functions****************************/

struct ph{ // Structure ph
       vector<int> a; // First element in structure
       vector<int> b; // Second element in structure
       };

struct  euclidean { // Creating another structure called euclidean

        int a; // First element in struct
        int b; // Second element in struct
        int c; // Third element in struct

};

euclidean extended_euclidean(int a, int n)
// Function extended_euclidean that
// will return type euclidean

{
    // Using the extended Euclidean algorithm to find integers x
    // such and y such that ax+ny = d, where d=gcd(a,n)

    int x,y,d,q,r;

    //Step 1.
    if(n==0){
    d=a; // d<-a
    x=1; // x<-1
    y=0; // y<-0

    euclidean result={d,x,y}; // In the struct euclidean,
    //elements 1,2 and 3 are d,x and y respectively
    return result; // return result
}

    //Step 2.
    int x2=1; // x2<-1
    int x1=0; // x1<-0
    int y2=0; // y2<-0
    int y1=1; // y1<-1


    //Step 3.
    while (n>0) { // While n>0 do the following:

          q=a/n; // q<-a/n
          r=a-(q*n); // r<-a-(q*n)
          x =(x2-(q*x1)); // x<-(x2-(q*x1))
          y=y2-(q*y1); // y<-y2-(q*y1)
          a=n; // a<-n
          n=r; // n<-r
          x2=x1; // x2<-x1
          x1=x; // x1<-x
          y2=y1; // y2<-y1
          y1=y; // y1<-y
    }

    //Step 4.
    d=a; // d<-a
    x=x2; // x<-x2
    y=y2; // y<-y2

    euclidean result={d,x,y}; // In the struct euclidean,
    //elements 1,2 and 3 are d, x and y respectively

    return result; // return result

}   // This ends the extended Euclidean's algorithm

int mult_inv(int alpha,int n){
    // Making a new structure

    euclidean result_new = extended_euclidean(alpha,n);
    //Calling the function extended_euclidean and
    //assigning it as type euclidean

    if(!(result_new.a>1)) // If d (the first element in the struct)
    // is greater 1 then a^(-1) mod n does not exist

    return result_new.b; // Return x (the second element in the struct)

    else

    return result_new.a;
}

/*-------------------Greatest Common Divisor---------------------*/

int gcd(int a,int b) { // Function gcd for use in Chinese Remainder
                       // Theorem
    if(a==0)
    return b;
    if(b==0)
    return a;
    if(a>b)
    return(gcd(a%b,b)); // Greatest Common Divisor (a mod(b), b)
    if(b>a)
    return(gcd(b%a,a)); // Greatest Common Divisor (b mod(a), a)
    }

/*----------------End of Greatest Common Divisor------------------*/

/*----------------------Inverse Function------------------------*/

int inverse(int a,int b) {// Function inverse for use in CRT
                          // here a-->modulus,b-->element
    int gcd,x=0,y=1,k=a;
    int u=1, v=0, m, n, q, r;
    gcd = b;
    while(a!=0){ // a is NOT equal to 0
           q=gcd/a; // q<-gcd/a
           r=gcd%a; // r<-gcd mod a
           m=x-u*q; // m<-x-(u*q)
           n=y-v*q; // n<-y-(v*q)
           gcd=a; // gcd<-a
           a=r; // a<-r
           x=u; // x<-u
           y=v; // y<-u
           u=m; // u<-m
           v=n; // v<-n
           }
           while(y<0){
                 y=k+y;
           }return y; // Give back value of y
}

/*---------------------End of Inverse function---------------------*/

/*--------------------Chinese Remainder Theorem--------------------*/

int ChineseTheorem(vector<int> x, vector<int> Z){

int k = x.size(); // k is the size of vector x
int m[k],a[k],i,n=1; // Initialising array's m and a of size k i.e
                              // equivalent to the size of vector x
                              // and integers i and n. Assigning n
                              // to value 1.

for(i=0;i<k;i++){ // i iterates from 0 to (vector size x)-1
                 a[i] = Z[i]; // Assigning array a to vector Z
                 m[i] = x[i]; // Assigning array m to vector x
       if(i>0){

                if(gcd(m[i-1],m[i])!=1){ // If the Greatest Common
                //Divisor of (m[i-1],m[i]) is NOT equal to 1

       return 1;
                   }
                   }
                   n=n*m[i]; // Every iterative value of array m is
                   // multiplied by n then assigned to n
                   }

       int sol=0,B=1
       for(i=0;i<k;i++){

       B=a[i]*(n/m[i])*inverse(m[i],n/m[i]); // Integer B is
       // equivalent to array a multiplied by (n divided by
       //array m) multiplied by the inverse of (array m and n
       //divided by array m), for which i iterates from 0 to
       //vector size x

                         sol=sol+B; // sol = sol add the value of B
                         }sol=sol%n; // sol = sol mod n
return sol;
}

/*----------------End of Chinese Remainder Theorem-----------------*/

/*-------------------Square and Multiply algorithm-----------------*/

int s_and_m(int alpha, int A, int n){

    int b=1;

    while(A>0) // While A>0 do:
    {
              if ((A-1)%2==0) // If A=1{mod2}
              {
                             b=b*alpha; //b=b*alpha
                             b=b%n; // b=(b*alpha)modn
              }
              alpha=alpha*alpha; // alpha=alpha^2
              alpha=alpha%n; // alpha=alpha^2(modn)
              A=int(A/2); // A=[A/2]
    }
    return b; // Return b
```

```
}

/*---------------End of Square and Multiply algorithm--------------*/

/*--------------------Prime Factorisation-------------------------*/

ph prime_fact(int n) {// Function prime factorisation of structure ph

int i = 2;
int count;
vector<int> q, c;

while (n > 1 && i <= n) { // n has to be both greater then 1 AND less
                          // than and equal to i. && Boolean operator
                          // AND, hence will only work if both
                          // conditions on left and right hand sides
                          //are satisfied

  if (n % i == 0){ // If n mod i is NOT equal to 0

        q.push_back(i); // Adding i values to vector q
        count = 1;
        n = n / i; // n = n divided by i
        while (n % i == 0) { // n mod i is NOT equal to 0
              count = count + 1;
              n = n / i;
              }

        c.push_back(count); // Adding count values to vector c
        }
  i = i + 1;
  }
  ph pH = {q,c}; // Storing vectors q and c in pH
  return pH; // Give back pH
}

/*------------------End of Prime Factorisation------------------*/
```

```
/*--------------------Pohlig-Hellman algorithm--------------------*/

int PHellman(int a, int b, int n){

    vector<int> g, l, alp, A, B;
    int z = 0, q = 0, e = 0, delta, gamma, alpha, beta, r;

    int p=n+1;
    ph x=prime_fact(n); // Calling function prime factorisation
                        // and factorising for prime factors
                        // of the value of order n

    for(int k = 0; k<x.a.size();k++){ // k goes from 0 to value of
                                      //(prime facotorisation - 1)

    q = x.a[k]; // q = values of primes factors a[k]
    e= x.b[k]; // e = exponential values of prime factors b[k]

    B.push_back(int(pow((double)x.a[k], x.b[k])));
    // Calculating q^e and adding to vector B

    l.push_back(0); // l_[-1]=0
    gamma=1;

    for(int j =0; j<e;j++){ // j goes from 0 to
                            // (maximum exponent value - 1)

    alpha=s_and_m(a, n/q, p); // Square and multiply method for
                              // (a^(n/q)) then equating to alpha

    gamma=gamma*(s_and_m(a, (l[j]*((int)(pow((double) q, j-1)))), p));

    // Square and multiply method for gamma*a^((l_[j-1])*(q^[j-1]))
    // then mod p and assigning to gamma

    gamma=gamma%p; // Gamma = gamma mod(p)

    for(int i=0;i<=x.a[k]-1;i++){ // i goes from 0 to q-1
    z=s_and_m(a, ((n*i)/x.a[k]), p);
    // Square and multiply method for a^((n*i)/q) then mod p
    // and assigning to z

    g.push_back(z); // Adding z value to vector g
}


    beta=mult_inv(gamma, p); // Multiplicative inverse of gamma
                             // then mod p and assigning to beta
    beta=beta*b;
    while(beta<0){
    beta=beta+p;
}
    beta = beta%p; // beta mod(p)

    beta=s_and_m(beta, (n/(int(pow((double)x.a[k], j+1)))),p);
    // Square and multiply for beta^(n/q^(j+1)) then mod p.
    // Assigning value to beta

    for(int y=0; y<g.size(); y++){ // y goes from 0 to
                                   // (size of vector g) - 1

    if(beta==g[y]){ // Comparing values of beta and y^th
                    // value in vector g
    l.push_back(y); // Adding y values to vector l

    g.clear(); // Clearing vector g
    break;
}
}
}
int t =0;
    for(int q=1;q<l.size();q++){// q goes from 1 to (size of l) - 1
        t = t + (l[q]*((int)pow((double) x.a[k], q-1)));
        // t + (l[q]*(prime factors^(q-1))) then assign to t
        }

        A.push_back(t); // Adding t values to the vector A
l.clear(); // Clearing vector l
```

```
}

int m = ChineseTheorem(B, A); // Calling the function Chinese Remainder
                              // Theorem to solve pair of congruences
                              // A and B. Equate to integer m
return m; // Give back value of m
}

/*-----------------End of Pohlig-Hellman algorithm-----------------*/

/***********************End of Functions**************************/

/*-------------------------Main Function--------------------------*/

int main(){

    int alpha, beta ,n;

     double time, timedif; // Double is used here to show small values

    time = (double) clock(); // Get the intial time
    time = time/CLOCKS_PER_SEC;  // Time in seconds

    cout<<"\n Divyesh B Chudasama"
          "\n Copyright (c) 2012";

     cout<<"\n \n Implementation of the Pohlig Hellman algorithm for"
           "\n computing the unique integer x such that"
           "\n x = log_{alpha}beta for the Discrete Logarithm"
           "\n Problem in GF(p), with cyclic group G, order n,"
           "\n generator alpha and group element beta. ";

    cout<<"\n \n Please enter a value for 'alpha': ";
    cin>>alpha;
    cout<<"\n Please enter a value for 'beta': ";
    cin>>beta;
    cout<<"\n Please enter a value for 'n': ";
    cin>>n;
```

```
   int r = PHellman(alpha, beta, n); // Calling Pohlig Hellman

   if(r==1)

       cout<<"\n There is no unique solution to this problem.\n\n";

   else if(r > 1)

    cout<<endl<<"\n The discrete logarithm x is: "<<r<<endl<<endl;

    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time;
    // Calling clock() function from start of program, subtracting
    // its return value, then to obtain the final processor time in
    // seconds, divide the value returned by clock() by
    // CLOCK_PER_SEC

    cout<<"\n The processor time elapsed is: "<<timedif<<" seconds"
    <<endl<<endl;

    keep_window_open();

}

/*---------------------End of Main Function---------------------*/
```

A.2 Algorithms for $\mathbb{F}_{p^n}$

A.2.1 Brute-Force Search

```
/*----------------------ex_search_2n.cpp--------------------------

Implementation of the Exhaustive Search/Brute Force method for
computing the unique integer x such beta=alpha^x for the Discrete
Logarithm Problem in the finite field GF(2^n), where n is the value
of the field exponent, alpha and beta are Galois Field Elements and
prim_poly is the primitive polynomial of the field.

Code written by Divyesh B Chudasama on 14/08/2012.
Copyright (c) 2012

------------------------ex_search_2n.cpp--------------------------*/

/*--------------------Exhaustive Search Method---------------------*/

//#include "stdafx.h" // Only include for VS 2010
#include <stdio.h>
#include <time.h>
#include "std_lib_facilities.h"
#include "GaloisFieldElement.h"

using namespace galois;

// Initialising primitive polynomial
unsigned int prim_poly[4] = {1, 0, 1, 1};

// Creating Galois Field, where the integer value has to be changed
//according to the power
GaloisField gf(3, prim_poly);

GaloisFieldElement alpha(&gf, 2);
//Galois Field Element alpha...change value accordingly

GaloisFieldElement beta(&gf, 4);
// Galois Field Element beta...change value accordingly
```

```
unsigned int o = (2^3)-1;
// Exponent power corresponds to value assigned above when
//initialising Galois Field.

int ex_search(GaloisFieldElement alpha, GaloisFieldElement beta,
unsigned int o){ // Function Exhaustive Search

vector<GaloisFieldElement> B1; // List

GaloisFieldElement b(&gf, 1);

unsigned int x;

for(x=0; x<o; x++){

        B1.push_back(b); // Adding entries to list B1
        b=(b*alpha);

        }

for(unsigned int y=0; y<B1.size(); y++){ //

    if(B1[y]==beta) // Comparing list value to find a match with beta

        return y;
}

}

/*------------------End of Exhaustive Search Method----------------*/

/*---------------------------Main Function-------------------------*/

int main(){

int ans;

double time, timedif; // Double is used here to show small values
```

```
    time = (double) clock(); // Get the initial time
    time = time/CLOCKS_PER_SEC;  // Time in seconds

    cout<<"\n Divyesh B Chudasama"
          "\n Copyright (c) 2012";

    cout<<"\n \n Implementation of the Exhaustive Search/Brute-Force"
          "\n method for computing the unique integer x for the"
          "\n Discrete Logarithm Problem in the finite field"
          "\n GF(2^n), where n is the value of the field exponent"
          "\n alpha and beta are Galois Field Elements and prim_poly
          "\n is the primitive polynomial of the field. ";

    ans = ex_search(alpha, beta, 7); // Calling Exhaustive Search

    cout<<endl<<"\n \n The discrete logarithm x is: "<<ans<<endl<<endl;

    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time;
    // Calling clock() function from start of program, subtracting
    // its return value, then to obtain the final processor time
    // in seconds, divide the value returned by clock()
    // by CLOCK_PER_SEC

    cout<<"\n The processor time elapsed is: "<<timedif<<" seconds"
    <<endl<<endl;

keep_window_open();

    return 0;
}

/*---------------------End of Main Function---------------------*/
```

A.2.2 Shanks' "Baby-Step Giant-Step" Method

```
/*---------------------------BSGS_2n.cpp------------------------

Implementation of Shanks' Baby-Step Giant Step Method for computing
the unique integer x such that beta=alpha^x for the Discrete
Logarithm Problem, in the finite field GF(2^n), where n is the value
of the field exponent, alpha and beta are Galois Field Elements and
prim_poly is the primitive polynomial of the field.

Code written by Divyesh B Chudasama on 25/08/2012.
Copyright (c) 2012

--------------------------BSGS_2n.cpp-------------------------*/

/*-------------------Variable Declarations--------------------*/

#include "stdafx.h"
#include <stdio.h>
#include "std_lib_facilities.h"
#include <iostream>
#include <math.h>
#include <list>
#include <time.h>
#include "GaloisFieldElement.h"

using namespace galois;

// Initialising primitive polynomial
unsigned int prim_poly[11] = {1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1};


GaloisField gf(10, prim_poly);
// Creating Galois Field, where the integer value represents
// the exponent n, i.e. 2^n. This value has to be changed
// according to the power

GaloisFieldElement alpha(&gf, 2);
// Galois Field Element alpha is fixed at value 2
```

```
GaloisFieldElement beta(&gf, 4);
// Galois Field element beta...change value accordingly

/************************Functions*************************/

/*---------------Multiplicative inverse in Z[n]---------------*/

struct  euclidean { // Creating a structure called euclidean

        GaloisFieldElement a; // First element in struct
        GaloisFieldElement b; // Second element in struct
        GaloisFieldElement c; // Third element in struct

};

euclidean extended_euclidean(GaloisFieldElement g,
GaloisFieldElement h)
// Function extended_euclidean that will return type euclidean

{
    // Using the extended Euclidean algorithm to find integers x
    // such and y such that ax+ny = d, where d=gcd(g,h)

    GaloisFieldElement d,q,r;

    //Step 1.
    if(h==0){
    d=g; // d<-g
    GaloisFieldElement x(&gf, 1); // x<-1
    GaloisFieldElement y(&gf, 0); // y<-0

euclidean result= {d,x,y}; // In the struct euclidean, elements 1,2
                           // and 3 are d,x and y respectively

    return result; // return result
}

    //Step 2.
```

```
GaloisFieldElement x(&gf, 1); // x<-1
    GaloisFieldElement y(&gf, 0); // y<-0
    GaloisFieldElement x2(&gf, 1);// x2<-1
    GaloisFieldElement x1(&gf, 0); // x1<-0
    GaloisFieldElement y2(&gf, 0); // y2<-0
    GaloisFieldElement y1(&gf, 1); // y1<-1


    //Step 3.
    while (h>0) { // While h is NOT equal to 0 do the following:

          q=g/h; // q<-a/n
          r=g-(q*h); // r<-a-(q*n)
          x=x2-(q*x1); // x<-(x2-(q*x1))
          y=y2-(q*y1); // y<-y2-(q*y1)
          g=h; // a<-n
          h=r; // n<-r
          x2=x1; // x2<-x1
          x1=x; // x1<-x
          y2=y1; // y2<-y1
          y1=y; // y1<-y
    }

    //Step 4.
    d=g; // d<-g
    x=x2; // x<-x2
    y=y2; // y<-y2

    euclidean result={d,x,y}; // In the struct euclidean, elements 1,
                              // 2 and 3 are d,x and y respectively

    return result; // return result

}   // This ends the extended Euclidean's algorithm

GaloisFieldElement mult_inv(GaloisFieldElement alpha,
GaloisFieldElement n){

    // Making a new structure
    euclidean result_new = extended_euclidean(alpha,n);
    //Calling the function extended_euclidean and
    //assigning it as type euclidean

if(!(result_new.a>1)) // If d (the first element in the struct)
                      // is greater 1 then a^(-1) mod n does
                      // not exist

return result_new.b; // Return x (the second element in the struct)

else

return result_new.a;

}

/*-------------End of Multiplicative inverse in Z[n]------------*/

/*----------------Square and Multiply algorithm-----------------*/

GaloisFieldElement s_and_m(GaloisFieldElement alpha, int A){

    GaloisFieldElement b(&gf, 1);

    while(A>0) // While A>0 do:
    {
              if ((A-1)==0) // If A-1=0
              {
                              b=b*alpha; //b=b*alpha
              }
              alpha=alpha*alpha; // alpha=alpha^2

              A=A/2; // A=[A/2]
    }
    return b; // Return b

}

/*-----------------End of Square and Multiply-------------------*/
```

```
/*-------------------Big-Step Giant-Step function--------------*/

int bsgs(GaloisFieldElement alpha, GaloisFieldElement beta,
unsigned int o){

    unsigned int m;
    unsigned int j;
    double f;

GaloisFieldElement s(&gf, 1); // s is fixed to 1,
  // GaloisFieldElement for conveniency

 GaloisFieldElement p = ((alpha^10) - s) + s;
// p = ((2^n)-1) + 1 = (2^n), the value of the
// power is to be changed according to n

    unsigned int i;
    unsigned int x;

//Step 1.
f = pow(2.0, 10.0) - 1; // For conveniency values of type double
                        //have been used the second input is to
                        //be changed based on the value of n

m=int(sqrt(f))+1;
//  Setting variable m to the square root of o then add 1


//Step 2.
vector<GaloisFieldElement> B1;
// B1 is a list of entries of values of size m

vector<GaloisFieldElement> B2;
// Second list for sorting, this is the same as B1

/*---------Baby Step: B=(j, (alpha^j)modp) for 0<=j<m----------*/

//double a;
//a = double(alpha);
for(j=0;j<m;j++)
{

    B1.push_back((alpha^j)); // Adds a new element at each
    // iteration to list B1, where the second element from each
    // pair is (alpha^j).

    B2.push_back((alpha^j)); // Adding elements to list B2

                 }

sort(B2.begin(), B2.end()); // Sorting the second list

//Step 3.
GaloisFieldElement c;

c = mult_inv(alpha,p);
// Assigning integer c to multiplicative inverse of
// (alpha^-1)modp


GaloisFieldElement e;

e = s_and_m(c,m); // Using square and multiply method to
                  //calculate (alpha^-m)

/*---------------------End of Baby Step----------------------*/

/*------Giant Steps: gamma=(beta)(alpha^(-m*i))modp, 2)gamma^i
where i=1,2,3..... and then comparing the values with the BabyStep
results to find a common value---------*/

//Step 4.
GaloisFieldElement gamma;

bool endwhile=true;
int z;
i=0;
```

```
while(endwhile){

gamma=s_and_m(e,i); // square and multiply for values (e^i)modp
              gamma=gamma*beta; // gamma = gamma*beta

              for(int l=0; l<m; l++){

              if(B2[l]==gamma){ // Comparing values from B2 to gamma
                            endwhile=false;
                            z=i;

              for (int delta=0; delta<m; delta++){

if (B2[l]==B1[delta]){ // Comparing lists B2 and B1 to find
                       // the common component in both
j=delta;

                  }
                            }
              }
              }
              gamma=0;
              i++;
              }
              //return z;

              x=(z*m)+j;  // x=(z*m)+j

                         return x;

                         }

/*---------------------End of Giant Step---------------------*/

/*------------End of Baby-Step Giant-Step Function------------*/

/********************End of Functions***********************/

/*----------------------Main Function-----------------------*/
```

```
int main(){

    double time, timedif;
    // Double is used here to show small values

    time = (double) clock(); // Get the intial time
    time = time/CLOCKS_PER_SEC;  // Time in seconds

    cout<<"\n Divyesh B Chudasama"
          "\n Copyright (c) 2012";

    cout<<"\n \n Implementation of Shanks' Baby-Step Giant-Step"
          "\n Method for computing the unique integer x such"
          "\n that beta = alpha^x, for the Discrete Logarithm"
          "\n Problem in the finite field GF(2^n), where n is"
          "\n the value of the field exponent, alpha and beta are"
          "\n Galois Field Elements and prim_poly is the primitive"
          "\n polynomial of the field. ";

    int x=bsgs(alpha,beta,1023); // Calling Baby-Step Giant-Step
   // the last input is to be changed based on (2^n)-1

    cout<<endl<<"\n The discrete logarithm x is: "<<x<<endl<<endl;

    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time;
    // Calling clock() function from start of program,
    // subtracting its return value, then to obtain the final
    // processor time in seconds, divide the value returned
    // by clock() by CLOCK_PER_SEC

    cout<<"\n The processor time elapsed is: "<<timedif<<" seconds"
    <<endl<<endl;

    keep_window_open();

    return 0;
}
```

```
/*--------------------End of Main Function--------------------*/
```

A.2.3 Pollard's ρ algorithm for logarithms

```
/*-------------------pollard_rho.cpp-------------------------

Implementation of Pollard's Rho Method for computing the unique
integer x such that beta=alpha^x for the Discrete Logarithm
Problem, in the finite field GF(2^n), where n is the value of the
field exponent, alpha and beta are Galois Field Elements and
prim_poly is the primitive polynomial of the field.

Code written by Divyesh B Chudasama on 26/08/2012.
Copyright (c) 2012

-------------------pollard_rho.cpp---------------------------*/

#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#include "std_lib_facilities.h"
#include "GaloisFieldElement.h"

using namespace galois;

unsigned int prim_poly[7] = {1, 0, 0, 0, 0, 1, 1};
// Initialising primitive polynomial

GaloisField gf(6, prim_poly);
// Creating Galois Field, where the integer value
// represents the exponent n, i.e. 2^n. This value
// has to be changed according to the power

GaloisFieldElement alpha(&gf, 2);
// Galois Field Element alpha is fixed at value 2

GaloisFieldElement beta(&gf, 4);
// Galois Field element beta...change value accordingly
```

```
unsigned int O = int(pow(2.0, 6.0))-1;
// Exponent power corresponds to value assigned above
// when initialising Galois Field.

/***********************Functions**************************/

/*-------------Set functions for S1, S2 and S3--------------*/

GaloisFieldElement f(GaloisFieldElement x){// Function f(x_i)
//for sequence of group elements x_(i+1) where x=(x_i)

    if(x==0){ // If x_i is an element of S2
                x=(x*x);// x_(i+1)=((x_i)^2)
                }

                else if(x==1){ // If x_i is an element of S3
                     x=(alpha*x); // x_(i+1)=(alpha*(x_i))
                     }

                else if(x==2){ // If x_i is an element of S1
                     x=(beta*x); // x_(i+1)=(beta*(x_i))
                     }
                     return x;
                     }

GaloisFieldElement g(GaloisFieldElement x, GaloisFieldElement a){
// Function g(x_i,a_i) for sequence of integers a_(i+1)
// where x=(x_i) and a=(a_i)/a_(i+1)

GaloisFieldElement c(&gf, 1); // c=1, for use with x==1 condition
GaloisFieldElement f(&gf, 2); // f=2, for use with x==0 condition

    if(x==0){ // If x_i is an element of S2
               a=(f*a); // a_(i+1)=(2*(a_i))
               }

               else if(x==1){ // If x_i is an element of S3
                    a=(a+c); // a_(i+1)=(a_i)+1
                    }

                else if(x==2){ // If x_i is an element of S1
                    a=a; // a_(i+1)=a_i
                    }
                    return a;
                    }

GaloisFieldElement h(GaloisFieldElement x, GaloisFieldElement b){
// Function h(x_i,b_i) for sequence of integers b_(i+1)
// where x=(x_i) and b=(b_i)/b_(b+1)

GaloisFieldElement e(&gf, 1); // e=1, for use with x==2 condition
GaloisFieldElement q(&gf, 2); // q=2, for use with x==0 condition

    if(x==0){ // If x_i is an element of S2
                b=(q*b); // b_(i+1)=(2*(b_i))
                }

                else if(x==1){ // If x_i is an element of S3
                    b=b; // b_(i+1)=(b_i)
                    }

                else if(x==2){ // If x_i is an element of S1
                    b=(b+e); // b_(i+1)=(b_i)+1
                    }
                    return b;
                    }

/*-----------End of Set functions for S1, S2 and S3---------*/

/*------------------Pollard's Rho function-----------------*/

GaloisFieldElement pollard_rho(GaloisFieldElement alpha,
GaloisFieldElement beta, unsigned int O){
// Pollard's rho method function

    unsigned int i;
    vector<GaloisFieldElement> x; // Vector list x
    vector<GaloisFieldElement> a; // Vector list a
    vector<GaloisFieldElement> b; // Vector list b

GaloisFieldElement w(&gf, 1); // w=1, w is equivalent to x_0
GaloisFieldElement u(&gf, 0); // u=0, u is equivalent to a_0
GaloisFieldElement p(&gf, 0); // p=0, p is equivalent to b_0

    x.push_back(w); // Adding entry x_0=1 to the vector list x
    a.push_back(u); // Adding entry a_0=0 to the vector list a
    b.push_back(p); // Adding entry b_0=0 to the vector list b

    GaloisFieldElement X;
    GaloisFieldElement A;
    GaloisFieldElement B;
    GaloisFieldElement r;
    GaloisFieldElement c;
    GaloisFieldElement d;
    GaloisFieldElement e;

GaloisFieldElement z(&gf, 0); // z=0

GaloisFieldElement t;
GaloisFieldElement T(&gf, 1); // T=1

t = (alpha^6) - T; // t = (2^n) - 1. The exponent will change based
    // on the value of n. Alpha, and T have been
    // used for efficieny and convenience.

    for(i=1; i<O; i++){
    // Computing the values x_i, a_i and b_i using the quantities
    // of x_(i-1), a_(i-1) and b(i-1) computed previously from
    // functions f, g and h

             c=f(x[i-1]); // c=f(x_(i-1))

             x.push_back(c); // Each iterative value of c is added
                             // to vector list x
```

```
         d=(g(x[i-1], a[i-1])); // d=g(x_(i-1),a_(i-1))

         a.push_back(d); // Each iterative value of d is added
                         // to vector list a

         e=(h(x[i-1], b[i-1])); // e=h(x_(i-1),b(i-1))

         b.push_back(e); // Each iterative value of e is
                        // added to vector list b

         }

   for(i=1; i<O; i++){
   // Now computing x_(2i), a_(2i) and b_(2i)

         if(x[i]==x[2*i]){

         X=f(f(x[(2*i)-2])); // X=x_(2i)

         A=(g(f(x[(2*i)-2]), g(x[(2*i)-2], a[(2*i)-2])));
         // A=a_(2i)

         B=(h(f(x[(2*i)-2]), h(x[(2*i)-2], b[(2*i)-2])));
         // B=b_(2i)

         break;
         }
         }
   for(i=1; i<O; i++){
      if( x[i] == X ){ // If x_i = x_(2i)

         r=(b[i]-B); // r=((b_i)-b_(2i))

      if(r==0){ // If r=0
           cout<<"\n \n 'Pollard's rho method failed to find"
                 "\n unique integer x as r=0!"

                 "\n No Discrete Logarithm x exists for the"
                 "\n primitive polynomial!'"<<endl;
```

```
            return r;
            break; // Terminate algorithm with failure
            }

            else{

     z=(a[i]-A)/(B-b[i]); // z=(((a_i)-a_(2i))/(b_(2i)-b_i))

                 if(z<0){ // If z is -ve
                        z=(t+z); // then ((2^n)-1)+z
                        }
                        }
                 return z;
                 break;
                 }

}
}

/*-----------------End of Pollard's Rho function----------------*/

/*********************End of Functions************************/

/*-----------------------Main function------------------------*/

int main(){

    double time, timedif; // Double is used here to show small values

    time = (double) clock(); // Get the intial time
    time = time/CLOCKS_PER_SEC;  // Time in seconds

    cout<<"\n Divyesh B Chudasama"
          "\n Copyright (c) 2012";

    cout<<"\n \n Implementation of Pollard's Rho Method for"
          "\n computing the unique integer x such that"
          "\n beta = alpha^x, for the Discrete Logarithm"
          "\n Problem in the finite field GF(2^n), where n is"
          "\n the value of the field exponent, alpha and beta"
          "\n are Galois Field Elements and prim_poly is the
          " \n primitive polynomial of the field. ";

    GaloisFieldElement y;

    y = pollard_rho(alpha, beta, 63); //Calling Pollard Rho
    // the last input is to be changed based on (2^n)-1

    cout<<endl<<"\n The discrete logarithm x is: "<<y<<endl<<endl;

    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time;
    // Calling clock() function from start of program,
    // subtracting its return value, then to obtain the final
    // processor time in seconds, divide the value returned
    //by clock() by CLOCK_PER_SEC

    cout<<"\n The processor time elapsed is: "<<timedif<<" seconds"
    <<endl<<endl;

    keep_window_open();

    return 0;
}

/*-------------------End of Main function--------------------*/
```