Imperial College London

Cryptography and Number Theory

Shivan Parmar

Undergraduate Research Opportunity Programme (UROP)

Supervised by

Dr Carl Wang-Erickson

Department of Mathematics

Imperial College London

September 2019

Contents

1. Introduction

It is almost impossible to overstate the importance of cryptography in the electronic age. Encryption forms the backbone of internet security systems, allowing people to confidently do business online without fear of deceit or deception and protecting the fundamental human right to privacy. Briefly, cryptography's importance is three-fold: it is used in authentication (e.g. logging onto online banking using unique digital ID created by your computer), integrity (think of a digital signature attached to the bottom of an email to prevent forgery), and confidentiality (end-to-end encryption on instant messaging platforms). The widespread adoption of decentralised cloud-based platforms to store data means that network security of information in transit is more crucial than ever.

Cryptographic algorithms fall broadly into two categories - symmetric and asymmetric ciphers. With a symmetric cipher, both parties have access to a secret key beforehand, and this key is used to encrypt information which is then transmitted across a public channel. This is notably different to the asymmetric cipher used in public key cryptography, in which there is a public key which is widely disseminated, and a private key known only to the owner.

Public key cryptography fundamentally relies on hard problems in mathematics, allowing the secure transmission of private information over public communication channels. Good candidates for such hard problems have a trapdoor – a piece of information which renders their solution trivial, but without which the problem is extremely difficult. This means that if both parties (who in this paper we'll refer to as Alice and Bob) have access to the trapdoor information, any middleman eavesdropper (who we'll call Eve) will be unable to decrypt the message in a practical amount of time. This is loosely analogous to a combination lock: with the code for the lock it is very easy to open, without it the prospect of trying every single possibility is enough to put off even the most determined of thieves.

This paper starts with an overview of finite fields and a 'hard problem' known as the Discrete Logarithm Problem. It explores how this is applied in practice with Diffie–Hellman key-exchange; a method by which two parties can securely decide on a mutual private key using only a public communication channel, and the various methods with which such an encryption system can be attacked. To finish, cryptographic methods featuring elliptic curves are explored, and comparisons made between the finite field counterparts. Throughout, Python implementations of different encryption and decryption algorithms are presented.

2. Basics of Finite Fields

We work towards proving that $\mathbb{F}_p^\times$ is cyclic. This is important, since it guarantees the existence of a unique solution to the discrete logarithm problem, which we will see in due course.

Definition 2.0.1. (Characteristic) The characteristic of a ring R is p if p is the smallest number of times the multiplicative identity (1) must be added to itself to get the additive identity (0). A ring is said to have characteristic zero when the additive identity 0 cannot be obtained by inductively adding the multiplicative identity 1 to itself.

We can then state the following lemma:

Lemma 2.0.2. *Any field has characteristic zero or a prime.*

Proof. Consider the sequence of elements of a field $(1, 1+1, 1+1+1, \ldots)$. Define $a_0 = 0$ and $a_n = a_{n-1} + 1$. Then since F is finite, it must be true that $a_l = a_k$ for some $l > k$. So then $a_l - a_k = 0$, and $l - k > 0$. It follows that there exists a least $m > 0$ such that $a_m = 0$. Since $1 \neq 0$, $m \geq 2$. Suppose $m = m_1 m_2$, where $m_1 > 1$ and $m_2 > 0$. Then $a_m = 0 \implies a_{m_1} = 0$ or $a_{m_2} = 0$ since a field has no zero divisors. This is a contradiction, since we defined m to be the least such number with this property. Hence m is prime, and we call it the characteristic of the field. Note that since an integral domain has no zero divisors by definition, this proof also applies to domains. □

Definition 2.0.3. A prime subfield of F is the intersection of all subfields of F.

Note the prime subfield can be finite or infinite, depending on the field F.

Proposition 2.0.4. *The prime subfield can equivalently be defined as the the subfield of F generated by the multiplicative identity 1.*

Proof. Denote as K the intersection of all subfields of F, and denote as L the subfield of F generated by 1. It is obvious that $K \subset L$, since $< 1 >$ is a subfield of F. Next we show that $L \subset K$. Take some $l \in L$. Then in slightly sloppy notation, $l = 1 + \ldots + 1$. Since $1 \in U$ for all subfields U of F by definition, we have that $l \in U$ for all subfields U of F, because fields are closed under addition. Thus $l \in K$, completing the proof. □

Proposition 2.0.5. *Let K be a finite field of characteristic p. The prime subfield of K is isomorphic to $\mathbb{F}_p$, the finite field of p elements.*

Proof. Represent $\mathbb{F}_p$ as $\{0, 1, ..., p-1\}$. Define a map into K as $\phi : \mathbb{F}_p \to K, r \to f \cdot 1$, where 1 is the multiplicative unit. ϕ is clearly additive and multiplicative, so is a field homomorphism. To show $\mathbb{F}_p \subset K$, it remains to show that the map is injective. Assume to the contrary that for some $p > r > s \geq 0, \phi(r) = \phi(s)$. Let $c = r - s > 0$. c can be considered an element of $\mathbb{F}_p^\star$, so has a multiplicative inverse $c^{-1} \in \mathbb{F}_p$.
Then $\phi(1) = \phi(c \cdot c^{-1}) = \phi(c)\phi(c^{-1}) = (\phi(r) - \phi(s))\phi(c^{-1}) = 0$. But by the definition of ϕ, $\phi(1) = 1 \neq 0$, since K is a field, contradiction. So ϕ is an isomorphism between $\mathbb{F}_p$ and the image of the homomorphism $\mathrm{Im}(\phi) \subset K$. This proves $\mathrm{Im}(\phi)$ is a subfield of K. has no non-trivial subfield; it is its own prime subfield, and so the same is a true of $\mathrm{Im}(\phi)$. Thus $\mathrm{Im}(\phi)$ is the prime field of K. □

We present a second, neater proof of the above proposition. It relies on the following lemma.

Lemma 2.0.6. *The only ideals of a field are 0, or the entire field.*

Proof. Clearly $\{0\}$ is an ideal of F. Let I be a non-zero ideal of F. Then it contains at least one non-zero element, denoted i. Since F is a field, $i^{-1} \in F$. Since I is an ideal, $ir \in I \quad \forall r \in R$, and in particular, $ii^{-1} = 1 \in I$. But if $1 \in I$, then $1 \cdot r = r \cdot 1 = r \in I \quad \forall r \in R$, and so $I = R$. □

This allows us to prove using a different construction, the injectivity of the homomorphism from the field to a ring, as re-stated in the following proposition.

Proposition 2.0.7. *If $\phi : F \to R$ is any homomorphism from a field to a non-trivial ring, then ϕ is injective.*

Proof. Let I be the kernel of the homomorphism ϕ (recall that this is an ideal). Then I cannot be R, so $I = \{0\}$, which implies that ϕ is injective. □

We now work towards proving an important theorem about the multiplicative group of a finite field of prime-power order.

Let d be an integer ≥ 1. Let $\phi(d)$ denote Euler's totient function.

Lemma 2.0.8. *If n is an integer ≥ 1, then $n = \sum_{d|n} \phi(d)$.*

Proof. If $d|n$, Let C_d be the unique subgroup of $\mathbb{Z}/n\mathbb{Z}$ of order d and let Φ_d be the set of generators of C_d. Since all elements of $\mathbb{Z}/n\mathbb{Z}$ generate exactly one of the C_d, $\mathbb{Z}/n\mathbb{Z}$ is the disjoint union of the Φ_d, and

$$n = |\mathbb{Z}/n\mathbb{Z}| = \sum_{d|n} |\Phi_d| = \sum_{d|n} \phi(d)$$

□

Lemma 2.0.9. *Let G be a finite group of order n. Suppose that for all divisors d of n, $\{x \in H | x^d = 1\}$ has at most d elements. Then G is cyclic.*

Proof. Let d be a divisor of n, and consider the set G_d consisting of elements of G with order d. Suppose $G_d \neq \emptyset$, then $\exists y \in G_d$. Clearly $\langle y \rangle \subset \{x \in G | x^d = 1\}$. But $|\langle y \rangle| = d$, so $\langle y \rangle = \{x \in G | x^d = 1\}$, by the hypothesis of the lemma. Thus, G_d is the set of generators of $\langle y \rangle$ of order d, so $|G_d| = \phi(d)$. So either G_d is empty, or has cardinality $\phi(d)$ for all d dividing n. But

$$n = |G| = \sum_{d|n} |G_d| = \sum_{d|n} \phi(d) = n$$

so $|G_d| = \phi(d) \quad \forall d|n$. In particular G_n is non-empty. Thus G is cyclic. □

Definition 2.0.10. A unique factorisation domain (UFD) is an integral domain in which every non-zero non-unit element can be written as product of irreducible elements.

Proposition 2.0.11. *A polynomial of degree d has at most d solutions in $\mathbb{F}$, even when counted with multiplicities.*

Proof. It is clear that this is equivalent to proving that a polynomial of degree at most n with more than n roots vanishes identically. This can be proved by induction. The base case $n = 0$ is obvious. Next take a polynomial f of degree at most n and let $x_1, ..., x_{n+1}$ be distinct roots of f. By the factor theorem,

$$f(x) = (x - x_{n+1})g(x)$$

where g has degree at most $n - 1$. Substitute $x = x_i$ for $i = 1, ..., n$. For all these values of x, the left hand side vanishes and $(x_i - x_{n+1})$ is non-zero. Hence all these x_i must be roots of g, and so inductively, g is identically zero. □

This proof works because $F[x]$ is a unique factorisation domain, and thus the number of roots of any polynomial in $F[x]$ is the same as the number of linear factors in its unique prime factorisation.

Theorem 2.0.12. *The multiplicative group $\mathbb{F}_q^\times$ of a finite field $\mathbb{F}_q$ is cyclic of order $q - 1$, where $q = p^n$, p prime, $n \geq 1$.*

Proof. Note that $\mathbb{F}^\times$ has p^{n-1} elements. Let d be any divisor of p^{n-1}. By the previous proposition, it is clear that the polynomial $x^d - 1 = 0$ has at most d roots. And by the lemma, this implies that our group is cyclic. □

Finally we state and prove Fermat's Little Theorem, a useful result for the following section.

Theorem 2.0.13. *Let p be a prime, and a an integer. then*

$$a^{p-1} \equiv 1 \mod p$$

Proof. Consider the set $G = \{1, ..., p-1\}$. It can easily be checked that under multiplication modulo p, this forms a group. Let a be an element of G. Let k be the order of a in the group. Then $< a >$ is a subgroup of order k, and so by Lagrange's theorem, $k|(p-1)$, the order of G. Therefore we can express $p-1$ as km, for some positive integer m, and

$$a^{p-1} \equiv a^{km} \equiv (a^k)^m \equiv 1^m \equiv 1 \mod p$$

□

3. Discrete Logarithms and Diffie–Hellman

In this section, the concept of a discrete logarithm is introduced, and its application in an elementary key exchange procedure is outlined. First we introduce some mathematical preliminaries.

3.1. **The Discrete Logarithm Problem.**

Definition 3.1.1. Let g be a primitive root for $\mathbb{F}_p$ and let h be a non-zero element of $\mathbb{F}_p$. The Discrete Logarithm Problem is the problem of finding an exponent x such that $g^x \equiv h \mod p$.

The existence of a solution to the discrete logarithm problem is guaranteed by the cyclicity of $\mathbb{F}_p^\times$. Note that since the field is finite, if there is one solution then there are infinitely many by the following reasoning. Fermat's little theorem says that $g^{p-1} \equiv 1 \mod p$. Hence if x is a solution to $g^x \equiv h$, then $x + k(p-1)$ is also a solution for every value of k because $g^{x+k(p-1)} = g^x \cdot (g^{p-1})^k \equiv h \cdot 1^k \equiv h \mod p$. Thus our discrete logarithm $\log_g(h)$ is defined only up to adding and subtracting multiples of $p-1$.

It is a 'hard problem' to solve the discrete logarithm problem; we will come back to exactly what this means later but in essence the most efficient way to compute the discrete logarithm is not much better than checking every case. This makes it an ideal candidate for the basis of a cryptographic algorithm.

3.2. **Diffie–Hellman Key Exchange.**

To best demonstrate the principle of the key exchange, consider the following hypothetical scenario. Suppose Alice and Bob want to transmit a private message through a public communication channel, without an eavesdropper Eve being able to decipher the message. They first agree on a large prime and a non-zero integer $g \mod p$, whose order in $\mathbb{F}_p^\star$ is a large prime. Alice picks a secret integer a, and Bob picks a secret integer b. They then each compute $g^a \mod p$ and $g^b \mod p$ respectively, and these computed values are exchanged on a public platform which Eve has access to. Then Alice and Bob use their secret integers to compute $A' \equiv B^a \mod p$ and $B' \equiv A^b \mod p$ respectively. However, note that

$$A' \equiv B^a \equiv (g^b)^a \equiv (g^a)^b \equiv A^b \equiv B' \mod p.$$

Thus they have exchanged a 'secret' key. In order to intercept this key, Eve would have to solve the following 'hard' problem.

Definition 3.2.1. Let p be a prime and g an integer. The Diffie–Hellman Problem is that of computing $g^{ab} \mod p$ from the known values of $g^a \mod p$ and $g^b \mod p$.

It is clear that the Diffie–Hellman problem is no harder than the Discrete Logarithm Problem, since if Eve knows a and b, computing $g^{ab} \mod p$ is trivial. However the converse is unclear, but in practice it is generally acknowledged that the Diffie–Hellman problem is hard enough to be secure (simply because no-one has discovered a way of solving it which does not involve solving the Discrete Logarithm Problem).This key exchange can be developed into a complete crypto-system.

3.3. **ElGamal Cryptosystem.** Alice requires a large prime p and an element $g \mod p$ of large prime order. She then computes $A \equiv g^a \mod p$, where a is a secret number of her choice, and publishes the quantity A. Bob has a message m, suppose it is an integer between 2 and p. He chooses a random 'ephemeral' key $k \mod p$, and uses it to encrypt one message, then discards k. He then computes the quantities $c_1 \equiv g^k \mod p$ and $c_2 \equiv mA^k \mod p$, and sends the ciphertext (c1,c2) to Alice. Alice knows a, so can compute $x \equiv c_1^a \mod p$, and hence also $x^{-1} \mod p$. Thus Alice can compute the quantity

$$\begin{aligned} x^{-1} \cdot c_2 &\equiv (c_1^a)^{-1} \cdot c_2 \mod p \\ &\equiv (g^{ak})^{-1} \cdot (mA^k) \mod p \\ &\equiv (g^{ak})^{-1} \cdot (m(g^a)^k) \mod p \\ &\equiv m \end{aligned}$$

and decrypt Bob's message! It can be shown that ElGamal is as hard to attack as the Diffie–Hellman problem.

Proposition 3.3.1. *Fix a prime p and base g to use for ElGamal encryption. Suppose the eavesdropper Eve can access an oracle decrypting arbitrary ElGamal ciphertexts encrypted using arbitrary ephemeral public keys. Then this oracle can be used to solve the corresponding Diffie–Hellman problem.*

Proof. Recall that in the Diffie–Hellman problem, Eve knows the quantities $A \equiv g^a \mod p$ and $B \equiv g^b \mod p$ and wants to compute the value of $g^{ab} \mod p$. An ElGamal oracle takes a prime p, base g, public key A and public cipher-text (c_1, c_2), and returns the quantity $(c_1^a)^{-1} \cdot c_2 \mod p$. In order to solve the Diffie–Hellman problem, Eve can simply choose $c_1 = B = g^b$ and $c_2 = 1$. Then she will obtain $(g^{ab})^{-1}$, and can easily just take the inverse. □

In comparison with the RSA algorithm, the ElGamal algorithm has the advantage that each time the plaintext is encrypted, it gives a completely different ciphertext. However it has the disadvantage that the ciphertext is twice as long as the plaintext.

4. Algorithms to Solve Discrete Logarithm Problem

In order to better understand the difficulty of breaking Diffie–Hellman and ElGamal, we now shift our attention to the algorithms available at the disposal of Eve, in order to solve the Discrete Logarithm Problem. We begin by considering algorithms which work in all cases, and then some quicker algorithms which work in special cases where Alice and Bob have been careless with their choice of numbers. These have been implemented in Python.

4.1. **Quantifying the Efficiency of an Algorithm.** Before diving into different algorithms to solve the discrete logarithm problem, it is worth first formalising the notion of how efficient an algorithm is, given by how many steps are required to come to a solution. For this we introduce some new notation.

Definition 4.1.1. Let $f : S \to \mathbb{R}_{\geq 0}$, where $S \subset \mathbb{R}_{\geq 0}$, and $g : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. We say $f(x)$ is $\mathcal{O}(g(x))$ if there exist real constants $C > 0$ and $M \geq 0$ such that $f(x) \leq C \cdot g(x)$ for $x \geq M$.

In a cryptography context, an algorithm is considered fast if it works in polynomial time, i.e. is $\mathcal{O}(x^n) : n \in \mathbb{R}$, and slow if it works in exponential time, i.e. is $\mathcal{O}(e^x)$. Many of the most difficult questions in cryptography involve trying to find ways to solve hard problems in sub-exponential times.

4.2. **Initial Algorithms.** We first note that there is a trivial bound on the time order of any algorithm to solve the discrete logarithm, however using more advanced algorithms we hope to improve on this baseline.

Proposition 4.2.1. *Let g be a group and $g \in G$ an element of order N. Then the discrete logarithm problem can be solved in $\mathcal{O}(N)$ steps, where each step consists of a multiplication by g.*

Explicitly, this algorithm is simply raising g to successive powers modulo p until the required power x is found, such that $g^x \equiv h \mod p$. A slightly faster algorithm to solve the Discrete Logarithm Problem is the following.

Proposition 4.2.2. *(Shanks' Babystep-Giantstep algorithm) Let G be a group, and $g \in G$ have order $N \geq 2$. The following algorithm solves the Discrete Logarithm Problem $g^x \equiv h \mod p$ in $\mathcal{O}(\sqrt{N} \cdot \log(N))$ steps, using $\mathcal{O}(\sqrt{N})$ of storage.*

(1) Let $n = 1 + \lfloor \sqrt{N} \rfloor$.

(2) Create two lists:

List 1: $e, g, g^2, ..., g^n$

List 2: $h, hg^{-n}, hg^{-2n}, ..., hg^{-n^2}$.

(3) *Find a match between both lists, of the form* $g^i = hg^{-jn}$. *Then* $x = i + jn$ *is a solution to the discrete logarithm problem* $g^x = h$.

The proof of this result is presented, followed by a Python implementation of the method.

Proof. First we show that the algorithm works in $\mathcal{O}(\sqrt{N} \cdot \log(N))$ steps. Creating the two lists takes 2n multiplications, and assuming a solution exists, finding a match between the two lists takes some multiple of $n \cdot \log n \approx n \cdot \log \sqrt{N}$ steps using standard sorting and searching algorithms, so the run-time is as given.

It remains to prove that lists 1 and 2 always have such a match. Let x be the unknown solution to the discrete logarithm problem, and write x as $x = nq + r$, where $0 \leq r < n$ and $1 \leq x < N$. Then since $n > \sqrt{N}$, $q = \frac{x-r}{n} < \frac{N}{n} < n$ since $n > \sqrt{N}$. Thus $g^x = h$ can be rewritten as $g^r = h \cdot g^{-qn}$ with $0 \leq r < n$ and $0 \leq q < n$. Note that g^r is in list 1, and hg^{-qn} is in list 2. □

```
import numpy as np
def dlp(p,g,h):
    N = 1
    while pow(g,N,p) != 1:
        N+=1
    n = int(1 + np.floor(np.sqrt(N)))
    list_1 = []
    list_2 = []
    for y in range(n):
        list_1.append(pow(g,y,p))
        z = pow(g,y*n,p)
        z_inv = pow(z,p-2,p)
        list_2.append((h*z_inv)%p)
    intersection = list(set(list_1).intersection(list_2))
    position_1 = list_1.index(intersection[0])
    position_2 = list_2.index(intersection[0])
    x = (position_1 + n*position_2) % p
    print('The solution to the Discrete Logarithm Problem is x = ', x)
```

4.3. **Chinese Remainder Theorem.** The next algorithm we consider is much quicker but only works in the case where Alice and Bob have carelessly chosen an element $g \in G$ of order N, where N factors into the

product of small primes. However, it relies on the Chinese Remainder Theorem, which we state and prove in full:

Theorem 4.3.1. *Let $m_1, m_2, ..., m_k$ be a collection of pairwise relatively prime integers. Let $a_1, ..., a_k \in \mathbb{Z}$ be arbitrary integers. Then the system*

$$x \equiv a_1 \mod m_1, x \equiv a_2 \mod m_2, ..., x \equiv a_i \mod m_i$$

has a unique solution modulo $m_1 m_2 ... m_k$.

Proof. **Existence.** Suppose that for some value of i, we have a solution $x = c_i$ to the first i simultaneous congruences, and we want a solution which also satisfies one more congruence, $x \equiv a_{i+1} \mod m_{i+1}$, of the form $x = c_i + m_1 m_2 ... m_i y$. Thus we need to find a value of y satisfying

$$c_i + m_1 m_2 ... m_i y \equiv a_{i+1} \mod m_{i+1} \quad (\star).$$

Well, since

$$gcd(a, m) = 1 \Longleftrightarrow \exists b \in \mathbb{Z} \quad s.t. \quad a \cdot b \equiv 1 \mod m$$

by elementary number theory, we have

$$gcd(m_{i+1}, m_1, m_2, ..., m_i) = 1 \Longleftrightarrow \exists z \quad s.t. \quad m_1 m_2 ... m_i z \equiv 1 \mod m_{i+1}.$$

Letting $\eta = a_{i+1} - c_i \mod m_{i+1}$ and $y = \eta z$, this is exactly the condition $(\star)$ we wish to satisfy.

Uniqueness. We prove this for the case of two congruences, and the argument easily generalises to multiple congruences. Suppose for a contradiction there are two solutions to the congruences modulo mn. So if we have $x \equiv a \mod m$, $x \equiv b \mod n$ with $x \equiv c \mod mn$ and $x \equiv d \mod mn$, then $0 \equiv (c - d) \mod mn$. Thus $c \equiv d \mod mn$, and so $c = d$ since c and d are both non-negative and less than mn. □

The Chinese Remainder Theorem can be implemented as follows

```
import numpy as np


def chinese_remainder(a,m):
    if len(a) != len(m):
        return 'a and m are of different lengths'
    else:
```

```
        for i in range(len(m)):
            for j in range(i+1,len(m)):
                if ehcf(m[i],m[j])[2] != 1:
                    return 'The chosen moduli are not coprime'
        moduli_product = np.prod(m)
        return solve_system(a,m) % moduli_product

def solve_system(a,m):
    # function takes in as arguments two vectors of equal length, a and m
    i=len(a)
    if i==1:
        x=a[0]
    else:
        b,n=a[0:i-1],m[0:i-1]
        c=solve_system(b,n)
        p = np.prod(n)
        y=findinverse(p,m[i-1])*(a[i-1]-c)
        x=c+p*y
    return x

# extended Euclid's algorithm
def ehcf(a, b):
#initialisation
    p1,q1,h1,p2,q2,h2 = 1,0,a,0,1,b
    from math import floor
    while h2 != 0:
        r = floor(h1/h2)
        p3 = p1-r*p2
        q3 = q1-r*q2
        h3 = h1-r*h2
        # reassign variables
        p1,q1,h1,p2,q2,h2=p2,q2,h2,p3,q3,h3
```

```
        # return output as a tuple
    return (p1, q1, h1)

def findinverse(k,p):
    l = ehcf(k,p)[0] % p
    return l
```

4.4. **Pollig–Hellman Algorithm.** The Pollig–Hellman algorithm is a method for rapidly solving the discrete logarithm $g^x \equiv h \mod p$ in the case that $N = |g|$ is a product of small prime powers.

Proposition 4.4.1. *Let G be a group, and suppose that we have an algorithm which can solve $g^x = h$ in $\mathcal{O}(S_{q^e})$ steps, given that $g \in G$ has order q^e. Then if instead N factors into a product of prime powers as $N = q_1^{e_1}...q_t^{e_t}$, then the discrete logarithm problem can be solved in $\mathcal{O}(\sum_{i=1}^{t} S_{q_i^{e_i}} + \log N)$ steps, using the following procedure:*

(1) For each $1 \leq i \leq t$, let $g_i = g^{\frac{N}{q_i^{e_i}}}$ and $h_i = h^{\frac{N}{q_i^{e_i}}}$. Notice g_i has prime power order, so we can use e.g. Shanks' Babystep-Giantstep algorithm to solve the discrete logarithm problem $g_i^y = h_i$. Let $y = y_i$ be a solution to this.

(2) Then put these solutions together, using the Chinese Remainder Theorem to solve

$$x \equiv y_1 \mod q_1^{e_1}, \quad x \equiv y_2 \mod q_2^{e_2}, \quad ... \quad , x \equiv y_t \mod q_t^{e^t} \quad (\star)$$

Proof. The run-time of the algorithm is clear; it takes $\mathcal{O}(\sum_{i=1}^{t} S_{q_i^{e_i}})$ to complete step 1, and $\mathcal{O}(\log N)$ to piece these together using the Chinese Remainder Theorem.

Suppose x is a solution to the system of congruences $(\star)$. Then for each i, we can write $x = y_i + q_i^{e_i} z_i$, for some z_i. Then

$$\begin{aligned}
(g^x)^{\frac{N}{q_i^{e_i}}} &= (g^{y_i + q_i^{e_i} z_i})^{\frac{N}{q_i^{e_i}}} \\
&= (g^{\frac{N}{q_i^{e_i}}})^{y_i} \cdot g^{N z_i} \\
&= (g^{\frac{N}{q_i^{e_i}}})^{y_i} \quad \text{since } g^N = 1 \\
&= g_i^{y_i} \quad \text{by definition of } g_i \\
&= h_i \\
&= h^{\frac{N}{q_i^{e_i}}} \quad \text{by definition of } h_i
\end{aligned}$$

Expressed in terms of discrete logarithms,

$$\frac{N}{q_i^{e_i}} \cdot x \equiv \frac{N}{q_i^{e_i}} \cdot \log_g(h) \mod N \quad (\star)$$

Since $\frac{N}{q_i^{e_i}}, \quad 1 \leq i \leq t$ have no non-trivial common factors, by the extended Euclid's algorithm it is possible to find coefficients c_i such that $\frac{N}{q_1^{e_1}} + \frac{N}{q_2^{e_2}} + ... + \frac{N}{q_3^{e_3}} = 1$. Therefore summing over both sides of $(\star)$,

$$\sum_{i=1}^{t} \frac{N}{q_i^{e_i}} c_i x \equiv \sum_{i=1}^{t} \frac{N}{q_i^{e_i}} c_i \log_g(h) \mod N$$

$$\implies x \equiv \log_g(h) \mod N$$

□

Below is a Python implementation of this

```
import numpy as np
from sympy.ntheory import factorint

# solve discrete logarithm problem for g a product of prime powers
def pohlig_hellman(g,h,p):
    # first need to calculate order of g
    N = 1
    while pow(g,N,p) != 1:
        N+=1
    # split into prime factors
    N_fact = factorint(N)
    a = []
    m = []
    for i in N_fact:
        g_i = pow(g,int(N/(i**N_fact[i])),p)
        h_i = pow(h,int(N/(i**N_fact[i])),p)
        y_i = dlp(g_i,h_i,p)
        a.append(y_i)
        m.append(i**N_fact[i])
        # crt used to piece together the solutions for prime powers
    return chinese_remainder(a,m)
```

Because of the Pohlig–Hellman method, we conclude that $g^x = h$ is easy to solve if the order of g is a product of powers of small primes. Consider the discrete logarithm problem in $\mathbb{F}_p$. Since $p-1$ is always even, the best Alice and Bob can do is to pick some $p = 2q+1$, where q is prime, and use an element g of order q. Then the running time of Shanks' Babystep-Giantstep would be $O(\sqrt{q}) = O(\sqrt{p})$.

The following proposition demonstrates that the runtime does not scale up significantly if g has order prime power, rather than prime order.

Proposition 4.4.2. *Let G be a group and q be a prime. Suppose that we know an algorithm which takes S_q steps to solve the discrete logarithm problem $g^x = h$ in G whenever g has order q. Now suppose $g \in G$ is an element of order $q^e, e \geq 1$. Then the discrete logarithm problem $g^x = h$ can be solved in $\mathcal{O}(eS_q)$ steps.*

Proof. Write the unknown exponent x in the form

$$x = x_0 + x_1 q + x_2 q^2 + ... + x_{e-1} q^{e-1}$$

. Note we've essentially written x in a different base. We aim to successively determine the coefficients x_i. Since $g^{q^{e-1}}$ has order q, we have that

$$\begin{aligned} h^{q^{e-1}} &= (g^x)^{q^{e-1}} \\ &= (g^{x_0 + x_1 q + x_2 q^2 + ... + x_{e-1} q^{e-1}})^{q^{e-1}} \\ &= g^{x_0 q^{e-1}} \cdot (g^{q^e})^{x_1 + x_2 q + ... + x_{e-2}} \\ &= (g^{q^{e-1}})^{x_0} \end{aligned}$$

since $g^{q^e} = 1$. But this final expression just gives us a discrete logarithm problem whose base is an element of order q. By assumption, it can be solved in S_q steps for x_0.
Next we do the same for q^{e-2}:

$$\begin{aligned} h^{q^{e-2}} &= (g^x)^{q^{e-2}} \\ &= (g^{x_0 + x_1 q + x_2 q^2 + ... + x_{e-1} q^{e-1}})^{q^{e-2}} \\ &= g^{x_0 q^{e-2}} \cdot g^{x_0 q^{e-1}} \cdot (g^{q^e})^{x_1 + x_2 q + ... + x_{e-2}} \\ &= g^{x_0 q^{e-2}} \cdot g^{x_0 q^{e-1}} \end{aligned}$$

So the discrete logarithm problem

$$(g^{q^{e-1}})^{x_1} = (h \cdot g^{-x_0})^{q^{e-2}}).$$

Continuing inductively, the entire exponent $x = x_0 + x_1 q + x_2 q^2 + ... + x_{e-1} q^{e-1}$ can be calculated in $\mathcal{O}(eS_q)$ steps, solving the original DLP. □

4.5. **Pollard's ρ algorithm.** So far the most efficient general algorithm introduced to solve the discrete logarithm problem in all cases is Shanks' Babystep-Giantstep algorithm. Pollard's ρ algorithm is slightly more sophisticated, taking the same amount of time but with the advantage that only two numbers have to be stored at any one time for the algorithm to run. We first introduce the method abstractly and then apply it specifically to solving our system $g^x = h$.

Let S be a finite set, and choose $f : S \to S$ to be a function which is good at mixing up the elements of S. Start with some element $x \in S$, and repeatedly apply the function f to create a sequence of elements. Since S is finite, this sequence must eventually loop back on itself. Denote by T the number of elements in the 'tail' (the portion of the sequence before getting to the loop), and by M the number of elements in the loop.

We define a collision as an element of the sequence which is repeated. Using this construction, it is possible to detect a collision in $O(\sqrt{N})$ steps without storing all the values. We compute not only the sequence x_i defined above, but also a second sequence y_i defined by $y_{i+1} = f(f(y_i)), \quad i = 0, 1, ...,$ i.e. $y_i = x_{2i}$.

To prove that a collision always occurs, note that for $j > i$, $x_j = x_i \Longleftrightarrow i \geq T$ and $j \equiv i \mod M$. So, $x_{2i} = x_i \Longleftrightarrow i \geq T$ and $2i \equiv i \mod M \Longleftrightarrow M|i$. Since one of $T, T+1, ..., T+M-1$ is divisible by M, this proves $x_{2i} = x_i$ for some $1 \leq i < T + M$.

We apply this in the specific context of solving the discrete logarithm problem, but first consider a general proposition concerning the efficiency of any algorithm which solves the DLP.

Proposition 4.5.1. *Let G be a group and $h \in G$ an element of order N. Then assuming the discrete logarithm problem $h^x = b$ has a solution, we claim it can be found in $\mathcal{O}(\sqrt{N})$ steps, where each step is an exponentiation in the group G.*

Sketch of Proof. We write $x = y - z$ and look for a solution to $h^y = b \cdot h^z$, by making a list of h^y values and a list of bh^z values, and looking for a match between the lists.

First choose random exponents $y_1, ..., y_n$ between 1 and N, and compute the values of $h^{y_1}, h^{y_2}, ..., h^{y_n}$ in G. Note that these values are all in the set $S = \{1, h, ..., h^{N-1}\}$. Next choose additional random exponents $z_1, ..., z_n$ between 1 and k, and compute $bh^{z_1}, bh^{z_2}, ..., bh^{z_n}$. Since we assume a solution exists, bh^{z_i} is also in the set S for some $i \in \{1, ..., n\}$. Once we get a match of the form $h^y = bh_i^z$, we've solved the discrete logarithm problem, setting $x = y - z_i$.

Each of the lists has n elements, so it takes about $2n$ steps to assemble each list. Each of these 'steps' consists of computing h^i for some i between 1 and N. Using a fast exponentiation algorithm, this takes approximately $2\log_2(i)$ group multiplications. Thus in total it takes about $4n\log_2(N)$ multiplications to assemble both lists. It will then take $n\log_2(n)$ to check that for a common element between the lists. Taking $n \approx 3\sqrt{N}$, which gives us a relatively high chance of a success, the computation is of the time order stated. □

4.6. **Discrete Logarithms via Pollard's method.** We want to solve $g^t = a$ in $\mathbb{F}_p^\star$ where g is a primitive root mod p, by finding a collision between $g^i a^j$ and $g^k a^l$ for known exponents i, j, k, l.

Let

$$f(x) = \begin{cases} g(x) & 0 \le x < \frac{p}{3} \\ x^2 & \frac{p}{3} \le x < \frac{2p}{3} \\ ax & \frac{2p}{3} \le x < p \end{cases} \pmod p$$

After i steps, we have $x_i = (f \circ f \circ ... \circ f)(1) = g^{\alpha_i} \cdot a^{\beta_i}$, where $\alpha_0 = \beta_0 = 0$, and

$$\alpha_{i+1} = \begin{cases} \alpha_{i+1} & 0 \le x < \frac{p}{3} \\ 2\alpha_i & \frac{p}{3} \le x < \frac{2p}{3} \\ \alpha_i & \frac{2p}{3} \le x < p \end{cases} \qquad \beta i + 1 = \begin{cases} \beta_i & 0 \le x < \frac{p}{3} \\ 2\beta_i & \frac{p}{3} \le x < \frac{2p}{3} \\ \beta_{i+1} & \frac{2p}{3} \le x < p \end{cases}$$

It suffices to keep track of these constants $\mod p-1$, since $g^{p-1} = 1$, $a^{p-1} = 1$. We then compute the sequence given by $y_0 = 1, y_{i+1} = f(f(y_i))$. Note that $y_i = x_{2i} = g^{\gamma_i} \cdot a^{\delta_i}$.

Applying the procedure, eventually a collision is found in the x and y sequences, say $y_i = x_i$, meaning $g^{\alpha_i} a^{\beta_i} = g^{\gamma_i} a^{\delta_i}$. Let $u \equiv \alpha_i - \gamma_i \mod p-1$ and $v = \delta_i - \beta_i \mod p-1$, then $g^u = a^v$ in $\mathbb{F}_p$. Equivalently,

$$v \log_g(a) \equiv u \mod p-1 \qquad (\star).$$

If $gcd(v, p-1) = 1$, can simply multiply both sides by inverse of $v \mod p-1$ to solve the discrete logarithm problem. And if $d = gcd(v, p-1) \geq 2$, use extended Euclidean algorithm to find integer s such that $sv \equiv d \mod p-1$. Multiplying both sides of $(\star)$ by s, $d \log_g(a) \equiv w \mod p-1$, where $w \equiv su \mod p-1$. Thus the full set of solutions are

$$\log_g(a) \in \{\frac{w}{d} + k \cdot \frac{p-1}{d} : k = 0, ..., d-1\}.$$

Note that in practice the value of d is small, so it suffices to check each of the d values until the correct one is found.

```
# extended Euclid's algorithm
def ehcf(a, b):
    p1,q1,h1,p2,q2,h2 = 1,0,a,0,1,b  # initialisation
    from math import floor
    while h2 != 0:
        r = floor(h1/h2)
        p3 = p1-r*p2
        q3 = q1-r*q2
        h3 = h1-r*h2
        p1,q1,h1,p2,q2,h2=p2,q2,h2,p3,q3,h3  # reassign variables
    return (p1, q1, h1)  # return output as a tuple

# modular inverse
def findinverse(k,p):
    l = ehcf(k,p)[0] % p
    return l

def mixing_function(x,g,a,p):
    x = x % p
    if x <= p/3:
```

```
        return g*x % p
    elif x < 2*p/3:
        return pow(x,2,p)
    else:
        return a*x % p

def alpha_function(alpha,x,p):
    alpha = alpha % (p-1)
    if x <= p/3:
        return (alpha + 1) % (p-1)
    elif x < 2*p/3:
        return 2*alpha % (p-1)
    else:
        return alpha

def beta_function(beta,x,p):
    beta = beta % (p-1)
    if x <= p/3:
        return beta
    elif x < 2*p/3:
        return 2*beta % (p-1)
    else:
        return (beta + 1) % (p-1)

def pollards_rho(g,a,p):
    i = 1
    x = mixing_function(1,g,a,p)
    y = mixing_function(mixing_function(1,g,a,p),g,a,p)
    alpha = alpha_function(0,x,p)
    beta = beta_function(0,x,p)
    gamma = alpha_function(alpha_function(0,y,p),mixing_function(y,g,a,p),p)
    delta = beta_function(beta_function(0,y,p),mixing_function(y,g,a,p),p)
```

```
    while x != y:
        i += 1
        alpha = alpha_function(alpha,x,p)
        beta = beta_function(beta,x,p)
        gamma = alpha_function(alpha_function(gamma,y,p),mixing_function(y,g,a,p),p)
        delta = beta_function(beta_function(delta,y,p),mixing_function(y,g,a,p),p)
        x = mixing_function(x,g,a,p)
        y = mixing_function(mixing_function(y,g,a,p),g,a,p)
    u = (alpha - gamma) % (p-1)
    v = (delta - beta) % (p-1)
    d = ehcf(v,p-1)[2]
    if d == 1:
        return (findinverse(v,p-1)*u) % p
    else:
        s = ehcf(v,p-1)[0]
        w = (s*u) % (p-1)
        for k in range(d):
            if pow(g,(w//d + k*(p-1)//d)%p,p) == a %p:
                return (w//d + k*(p-1)//d)%p
        return 'Algorithm failed'
```

4.7. **Index Calculus method for solving Discrete Logarithm Problem.** We conclude this section with a discussion of the index calculus method, currently the fastest known method to solve the discrete logarithm problem.

Definition 4.7.1. An integer n is called B-smooth if $p \leq B$, for every prime factor p of n.

Consider the now familiar discrete logarithm problem, in which g is a primitive root mod p, so its powers give all of $\mathbb{F}_p$. We choose a value B and solve $g^x \equiv l \mod p$ for all primes $l \leq B$. Having done this, consider $hg^{-k} \mod p$ for k = 1,2,... until we find a value of k such that hg^{-k} is B-smooth.

For this value of k, we have hg^{-k} for certain exponents e_l. Taking discrete logarithms, we obtain

$$log_g(h) \equiv k + \sum_{l \leq B} e_l \cdot \log_g(l)$$

where discrete logarithms are defined only modulo $p-1$.

This can then be solved, assuming we've already calculated $\log_g(l)$ for all primes l less than B. In order to calculate these, for a random selection of exponents i we compute $g_i \equiv g^i \mod p$, with $0 < g_i < p$. If g_i is not B-smooth, discard it. But if it is, factor it as $g_i = \prod_{l \leq B} l^{u_l^{(i)}}$. This gives us the relation

$$i \equiv \log_g(g_i) \equiv \sum_{l \leq B} u_l(i) \cdot \log_g(l) \mod p-1$$

Consider an equation of the above form. We first solve congruences mod q for each prime q dividing $p-1$. If q appears in the factorisation of $p-1$ to a power q^e, we lift the solution from $\mathbb{Z}/q\mathbb{Z}$ to $\mathbb{Z}/q^e\mathbb{Z}$, before using the Chinese Remainder Theorem to combine solutions modulo prime powers to obtain a solution modulo $p-1$.

Perform this process for each equation of the above form. We introduce the notation $\pi(x)$ to denote the number of primes less than or equal to x. So long as we have $\pi(B)$ equations of the above form, we can use the Chinese Remainder Theorem to knit them together, and thus solve the system of equations.

The algorithms below check that g is a primitive root modulo p, and calculate a list of integers of the form $g_i = g^i \mod p$, all of which are B-smooth.

```
from sympy.ntheory import factorint, isprime
from random import randint

# first we need to check whether g is a primitive root mod p:
def primitive_checker(a,p):
    if isprime(p) == False:
        return 'p is not prime'
    else:
        list1 = []
        s = p-1 # this is Euler's totient function
        N_fact = factorint(s)
        for i in N_fact:
            list1.append(pow(a,int(s/i),p) != 1)
        if all(char==True for char in list1):
            return True
```

```
        else:
            return False

# Returns a list of integers, each of which is a product of primes less than B
def B_smooth_gis(g,p,B):
    primes = []
    for i in range(B):
        if isprime(i)== True:
            primes.append(i)
    print('primes: ',primes)
    gi_list = []
    for _ in range(1000):
        g_i = pow(g,randint(1,1000), p)
        gi_list.append(g_i)
    gi_factors = []
    B_smooth_gis = []
    for g_i in gi_list:
        gi_factors = factorint(g_i)
        if all(factors < B for factors in gi_factors):
            B_smooth_gis.append(g_i)
    B_smooth_gis = B_smooth_gis[0:len(primes)]
    B_smooth_gis = list(set(B_smooth_gis))
    return B_smooth_gis
```

5. Elliptic Curve Cryptography

Instead of presenting the discrete logarithm problem, Diffie–Hellmann key exchange and ElGamal encryption over the $\mathbb{F}_p^\times$, it is possible to instead construct them analogously over $E(\mathbb{F}_p)$, which is also a group. Multiplication in an elliptic curve is a more difficult task than exponentiation in a finite field and so the fundamental hard problem of the elliptic discrete logarithm problem is harder than the classic discrete logarithm problem, making it a very good candidate for cryptography. The mathematical preliminaries of this construction are discussed.

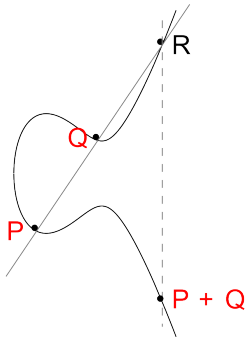
Definition 5.0.1. An elliptic curve E is the set of solutions to a Weierstrass equation

$$E : Y^2 = X^3 + AX + B$$

together with an extra point $\mathcal{O}$, where the constants A and B must satisfy $4A^3 + 27B^2 \neq 0$

We can define an addition law on the elliptic curve. If P and Q are two points on E, and L is the line connecting P to Q (or the tangent line to P, if $P = Q$), then the intersection of E and L consists of P, Q and one other point, call it R. Suppose R has coordinates (x, y). Then $P + Q$ is defined as equal to $R' = (x, -y)$.

Note that the point $\mathcal{O}$ is defined as living on every vertical line, at 'infinity'. Thus the addition of two points lying on the same vertical line is defined as $\mathcal{O}$.



Similarly, subtraction of points $P - Q$ is defined as $P + (-Q)$, where $-Q$ is the reflection of the point Q in the x-axis. The extra condition on A and B is essentially saying that the discriminant of the cubic $x^3 + Ax + B$ is not equal to zero, i.e. the solution set has no singularities.

The elliptic curve, along with addition of points defined as above, forms an abelian group (all conditions except for associativity can be checked easily). An efficient algorithm for calculating the addition of two points on an elliptic curve is implemented in Python as follows:

```
# Elliptic curve addition algorithm
# E: Y^2 = X^3 + AX + B
def elliptic_add(A,B, p_1,p_2):
    if 4*A**3 + 27*B**2 == 0:
        return 'This is not a valid elliptic curve'
    if p_1 == (0,0):
        return p_2
    elif p_2 == (0,0):
        return p_1
    else:
        (x_1,y_1) = (p_1[0],p_1[1])
        (x_2,y_2) = (p_2[0],p_2[1])
        if x_1 == x_2 and y_1 != y_2:
            return (0,0)
        else:
            if p_1 != p_2:
                l = (y_2-y_1)/(x_2-x_1)
            elif p_1 == p_2:
                l = (3*x_1**2 + A)/2*y_1
            x_3 = l**2 - x_1 - x_2
            y_3 = l*(x_1 - x_3) - y_1
            return (x_3,y_3)
```

5.1. **Elliptic Curves over Finite Fields.**

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \text{ such that } y^2 = x^3 + Ax + B\} \cup \{\mathcal{O}\}$$

The elliptic curve addition algorithm applied to P and Q yields a point in $E(\mathbb{F}_p)$, which we denote $P+Q$. This addition law makes $E(\mathbb{F}_p)$ a finite abelian group.

In order to implement addition over $E(F_p)$, we need to remember that a fraction over a finite field is a representation of the solution to a certain algebraic equation. For instance, $\frac{2}{5}$ over the field $\mathbb{F}_{13}$ represents the solution to $5x = 2 \mod 13$.

```
def finite_frac_eval(a,b,p):
    if b%p == 1:
        return a%p
    elif a%p == 1:
        return findinverse(b%p,p)
    else:
        for x in range(p):
            if (b*x) % p == a % p:
                return x
        return 'Modular inverse does not exist'


def elliptic_finite_add(A,B,p,p_1,p_2):
    if 4*A**3 + 27*B**2 == 0:
        return 'This is not a valid elliptic curve'
    if p_1 == (0,0):
        return p_2
    elif p_2 == (0,0):
        return p_1
    else:
        (x_1,y_1) = (p_1[0],p_1[1])
        (x_2,y_2) = (p_2[0],p_2[1])
        if x_1 == x_2 and y_1 != y_2:
            return (0,0)
        else:
            if p_1 != p_2:
                l = finite_frac_eval(y_2-y_1,x_2-x_1,p)
            elif p_1 == p_2:
```

```
            l = finite_frac_eval(3*x_1**2 + A,2*y_1,p)
        x_3 = l**2 - x_1 - x_2
        y_3 = l*(x_1 - x_3) - y_1
        return (x_3%p,y_3%p)
```

The following lists all the points lying in the field $E(\mathbb{F}_p)$ and checks that under addition, it does indeed form an abelian group.

```
from elliptic_finite_add import add
import itertools

def elliptic_finite_points(A,B,p):
    if p <= 3:
        return 'Choose a larger finite field'
    F_p = []
    for y in range(p):
        for x in range(p):
            if pow(y,2,p) == (x**3 + A*x + B) % p or x == y == 0:
                F_p.append((x,y))
    return F_p

def finite_group_checker(A,B,p):
    for pair in itertools.combinations(elliptic_finite_points(A,B,p), 2):
        A_plus_B = (add(A,B,p,*pair)[0] % p, add(A,B,p,*pair)[1] % p)
        if A_plus_B not in elliptic_finite_points(A,B,p):
            return 'Not a group'
    return 'This is a group'
```

Before moving on to the explicit use of elliptic curves in cryptography, in the form of the elliptic curve analogy to the discrete logarithm problem, it is worth pausing to give a bound on the number of points on an elliptic curve over a finite field.

Theorem 5.1.1. *(Hasse) Let E be an elliptic curve over $\mathbb{F}_p$. Then*

$$\#E(\mathbb{F}_p) = p + 1 - t_p$$

where t_p satisfies $|t_p| \leq 2\sqrt{p}$.

5.2. **Elliptic Curve Discrete Logarithm Problem (ECDLP).** We are now in a position to define our discrete logarithm problem. Alice chooses and publishes P and Q, and her secret is an integer n such that $Q = nP$. By analogy with the discrete logarithm problem in $\mathbb{F}_p^\star$, we denote this integer n by $n = \log_P(Q)$. The elliptic discrete logarithm of Q with respect to P.

Note that if there is one value of n satisfying $Q = nP$, then there are many (since $E(\mathbb{F}_p)$ is finite). Thus the value of $\log_p(Q)$ is really an element of $\mathbb{Z}/s\mathbb{Z}$, where s is the least integer such that $kP = (k+s)P$ for some k. Since $\log_P(Q_1 + Q_2) = \log_P(Q_1) + \log_P(Q_2) \quad \forall Q_1, Q_2 \in E(\mathbb{F}_p)$, the elliptic discrete logarithm defines a group homomorphism $E(\mathbb{F}_p) \to \mathbb{Z}/s\mathbb{Z}$.

In general, the fastest way to solve the ECDLP is an algorithm such as a modified form of Pollard's ρ algorithm, which solves it in $\mathcal{O}(\sqrt{p})$. However, there are no sub-exponential algorithms to solve ECDLP, as the index calculus method solved the normal DLP. This is because there is no straightforward notion of 'smoothness' in $E(F_p)$. In prime fields there is an easy mapping from the multiplicative group to the integers, but such a simple mapping does not exist for $E(F_p)$. There are again special cases for Alice and Bob to avoid, where fast algorithms exist to solve ECDLP, and these cases will be explored in more detail later in this paper.

5.3. **Double and Add Algorithm.** In order to take advantage of the cryptographic opportunities the elliptic curve DLP presents, a fast algorithm is needed for scalar multiplication, i.e. to compute the value of nP given both n and P, without having to iterate through the n addition steps. Though faster algorithms do exist, the moderately fast 'double and add' method is described here for its simplicity.

First n is decomposed into its binary representation:

$$n = n_0 + n_1 \cdot 2 + n_2 \cdot 4 + ... + n_r \cdot 2^r \text{ with } n_0, n_1, ..., n_r \in \{0, 1\}$$

Then the following algorithm is implemented:

```
import numpy as np
def multiply(A,B,p,n,P): # using the double and add algorithm
    Q = P
    R = (0,0)
```

```
    print(n,Q,R)
    while n>0:
        if n % 2 == 1:
            R = elliptic_finite_add(A,B,p,R,Q)
        Q = elliptic_finite_add(A,B,p,Q,Q)
        n = np.floor(n/2)
        print(n,Q,R)
    return R
```

Addition of two points in $E(\mathbb{F}_p)$ is referred to as a point operation. Computing nP using this method takes at most $2r$ point operations in $E(\mathbb{F}_p)$.

5.4. **An Improvement to the Double and Add algorithm.** The double and add algorithm is reasonably fast, but relies on doubling in an elliptic curve over a finite field which is an intrinsically involved computation. It can be made more efficient by replacing the doubling map by the Frobenius one:

Definition 5.4.1. The Frobenius map is defined as $\tau : \mathbb{F}_{p^k} \to \mathbb{F}_{p^k}; \quad \alpha \mapsto \alpha^p$

It is much easier to compute $\tau(P)$ than $2P$, since the exponentiation can be applied to each coordinate in turn. Note that the Frobenius map has lots of nice properties - for instance it is an automorphism group of $E(\mathbb{F}_p)$.

The following theorem gives us the key idea that the action of the Frobenius map on $E(\mathbb{F}_{2^k})$ satisfies a certain quadratic equation:

Theorem 5.4.2. *Let E be an elliptic curve over $\mathbb{F}_p$, and denote $t = p + 1 - \#E(\mathbb{F}_p)$. Note that by Hasse's theorem (see previous), $|t| \leq 2\sqrt{p}$. Claim the following*

(1) If α and β are roots of $Z^2 - tZ + p$, then $|\alpha| = |\beta| = \sqrt{p}$ and $\forall k \geq 1$, $\#E(\mathbb{F}_{p^k}) = p^k + 1 - \alpha^k - \beta^k$

(2) Let $\tau : E(\mathbb{F}_p) \to E(\mathbb{F}_p)$, $(x, y) \mapsto (x^p, y^p)$ (i.e. the Frobenius map).

Then for all $Q \in E(\mathbb{F}_p)$, $\tau^2(Q) - t \cdot \tau(Q) + p \cdot Q = 0$.

This is an important theorem, because it enables us to calculate the number of elements in $E(\mathbb{F}_p)$ with relative ease.

Definition 5.4.3. A Koblitz curve is defined as an elliptic curve defined over $\mathbb{F}_2$ by $E_\alpha : Y^2 + XY = X^3 + aX^2 + 1, \quad a \in \{0, 1\}$. The discriminant of the curve is one.

When our curve is a Koblitz curve, there is a very easy algorithm for adding points on the curve. When the curve is not Koblitz, some adjustment is needed but a similar method can be used. See [1,p.334] for details.

5.5. **Elliptic Diffie–Hellman key exchange.**

- Trusted party chooses and publishes large prime p, elliptic curve E over $\mathbb{F}_p$, and point P in $E(\mathbb{F}_p)$.
- Alice chooses secret integer n_A, and computes $Q_A = n_A P$; Bob chooses n_B and computes $Q_B = n_B P$.
- Values of Q_A and Q_B are publicly exchanged
- Alice computes $n_A Q_B$ and Bob computes $n_B Q_A$. But note these are the same, since $n_A Q_B = n_A(n_B P) = n_B(n_A P) = n_B Q_A$.

Note that Alice and Bob should only send each other the x-values of Q_A and Q_B, and should use only the x-value of their final product as their shared secret value. This is because the y-values can be very easily be determined from the corresponding x-values, so are not secure.

5.6. **Lenstra's Factorisation Algorithm.** Elliptic curves were introduced to the cryptographic community by Lenstra's algorithm, an elliptic analogue of Pollard's $\rho - 1$ factorisation algorithm to factorise large integers. It works especially well when the number has a relatively small factor.

Consider an elliptic curve modulo N, where N is not prime (so that the ring $\mathbb{Z}/N\mathbb{Z}$ is not a field). Suppose $P = (a, b)$ is a point on E modulo N, i.e. $b^2 \equiv a^3 + A \cdot a + B \mod N$. Apply the elliptic curve addition algorithm to compute multiples of P, until you get "stuck". This will occur when a certain number x does not have a reciprocal modulo N, and so tells us that x is a divisor of N. Once we've computed $Q = (n-1)! \cdot P$, it is easy to compute $n! \cdot P$, since this is equal to nQ. There are three possibilities for this stage:

(1) nQ can be computed.

(2) We need to find the reciprocal of a number d which is a multiple of N (this case is unlikely so if it occurs we just try again with a different initialisation).

(3) Need to find the reciprocal of a number d satisfying $1 < gcd(d, N) < N$, and so the computation fails, and gcd(d,N) is a nontrivial factor of N.

This is implemented below:

```
def lenstra_factorise(N):
    # initialisation
    A = randint(1,N)
    a = randint(1,N)
```

```
    b = randint(1,N)
    P = (a,b)
    B = (b**2 - a**3 - A*a) % N
    lenstra_iterate(N,A,B,P)


def lenstra_iterate(N,A,B,P):
    for j in range(2,50):
        try:
            Q = dna.multiply(A,B,N,j,P)
            P = Q
        # if this computation fails, have found d>1 s.t. d|N
        except:
            Q = P
            R = (inf,inf)
            d = 1
            while j>0:
                if j % 2 == 1:
                    try:
                        R = add(A,B,N,R,Q)
                        print('R,Q = ',R,Q)
                    except:
                        print('exception 1 raised')
                        d = ehcf((Q[0] - R[0]) % N , N)[2]
                        break
                try:
                    Q = add(A,B,N,Q,Q)
                    j = j//2
                except:
                    print('exception 2 raised')
                    d = ehcf((2*Q[1]) % N, N)[2]
                    break
```

```
            if d < N:
                print( 'd = ', d)
                break
            elif d == N:
                lenstra_factorise(N)
    return 'I can\'t factorise ',N
```

5.7. **"Nothing up my Sleeve" Number.** The one disadvantage of elliptic curve cryptography, in comparison to RSA, is that it requires the use of a specified elliptic curve which is not susceptible to any particular form of special attack. This raises trust issues about the vulnerability of a particular elliptic curve.

In order to convince someone that a particular curve is safe and not susceptible to some form of special attack, an additional domain parameter called the seed is used. This is a random number, the hash of which is used to generate either the coefficients of the elliptic curve, or the base point g, or both. Since hash inversion is a difficult problem, it is hard to construct a seed from the domain parameters. Curves generated from a seed are said to be verifiably random.

However, this trust can still be eroded. The random seeds for NIST curves having been published with no justification by the US government, and they are now used for a wide variety of internet security purposes. It is therefore possible that some weak class of elliptic curves was found and specified:

"I no longer trust the constants. I believe the NSA has manipulated them through their relationships with industry" —Bruce Schneier, The NSA Is Breaking Most Encryption on the Internet (2013)

5.8. **ECDSA.** Digital signature algorithms aim to have the property that only one individual can create a signature, which is then published, but anyone is able to check the validity of this signature. In the context of our protagonists, Alice wants to sign a message with her private key, d_A, and Bob wants to validate the signature using Alice's public key H_A. Denote the hash of the message, truncated to the bit length of the order of the subgroup n, as z.

5.8.1. *Generating signature.*

(1) Choose random integer $k \in \{1, ..., n-1\}$.

(2) Calculate $P = kG$ and $r = x_p \mod n$ (where x_p denotes the x coordinate of P)

(3) If r = 0, choose another k and try again

(4) Calculate $s = k^{-1}(z + rd_a) \mod n$

(5) If $s = 0$, choose another k and try again

The pair (r, s) is the signature. Essentially, the 'secret' is k, which is hidden in r because of point multiplication. r is then bound to the message hash by $s = k^{-1}(z + rd_n)$.

5.8.2. *Verifying signature.*

(1) Calculate $u_1 = s^{-1}z \mod n$ and $u_2 = s^{-1}r \mod n$

(2) Let $P = u_1G + u_2H_A$

(3) The signature is valid if $r = x_p \mod n$

This is because

$$\begin{aligned} P &= u_1G + u_2H_A \\ &= u_1G + u_2d_AG \\ &= (u_1 + u_2d_A)G \\ &= (s^{-1}z + s^{-1}rd_A)G \\ &= s^{-1}(z + rd_A)G \end{aligned}$$

Recall that s was defined as $s = k^{-1}(z + rd_a) \mod n$, and so $k = s^{-1}(z + rd_A) \mod n$. Thus $P = s^{-1}(z + rd_A)G = kG$

5.8.3. *Security precautions.* When generating signatures, k must be kept secret and must be changed for each instance of the generated signature. Sony PlayStation 3 games were found to have a serious security flaw whereby the signatures on all games were generated by a static k [7]. This meant that Sony's private key, call it d_s, could be recovered simply by buying two signed games, and extracting their hashes(z_1 and z_2), signatures((r_1, s_1) and (r_2, s_2)), and domain parameters, as follows.

- Since $r = x_p \mod n$ and $P = kG$ is the same for both signatures, we have $r_1 = r_2$.
- Consider $(s_1 - s_2) \mod n = k^{-1}(z_1 - z_2) \mod n$. Rearranging, $k = \frac{z_1 - z_2}{s_1 - s_2} \mod n$.
- Since $s = k^{-1}(z + rd_s) \mod n$, $d_s = r^{-1}(sk - z) \mod n$. All of these quantities are known, and so the value of Sony's private key could then be calculated.

It is also important that a suitable elliptic curve is chosen, over which to perform these algorithms. All curves which are super-singular must be avoided, since the elliptic curve discrete logarithm problem can be reduced to the discrete logarithm problem in a finite field, which is known to have sub-exponential complexity.

6. Acknowledgement

I'd like to extend my gratitude to Dr Carl Wang-Erickson for all his support and guidance, especially for agreeing to supervise me whilst busy moving to a new teaching post in a different country, and without whom this project would not have been possible.

References

[1] Conrad, K. (2019). Cyclicity of (Z/(p))×. [ebook]
Available at: https://kconrad.math.uconn.edu/blurbs/grouptheory/cyclicmodp.pdf [Accessed 1 Sep. 2019].

[2] Hoffstein, J., Pipher, J. and Silverman, J. (2014). An Introduction to Mathematical Cryptography. New York, NY: Springer New York.

[3] Serre., J. (1973). A Course in Arithmetic. New-York: Springer-Verlag, p.14.

[4] Silverman, J. and Tate, J. (2015). Rational points on elliptic curves. New York: Springer.

[5] Smart, N. (1997). The Discrete Logarithm Problemon Elliptic Curves of Trace One. [ebook] Bristol: HP Laboratories. Available at: `https://www.hpl.hp.com/techreports/97/HPL-97-128.pdf` [Accessed 1 Sep. 2019].

[6] Green, H. (2018). The p-adic numbers. [ebook] London.
Available at: `http://wwwf.imperial.ac.uk/ cwangeri/pdfs/Green--The_p-adic_-numbers--UROP2018.pdf` [Accessed 1 Sep. 2019].

[7] Corbellini, A. (2015). Elliptic Curve Cryptography: a Gentle Introduction - Andrea Corbellini. [online] Corbellini.name. Available at: https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/ [Accessed 17 Sep. 2019].

[8] Ruprai, R. (2007). Improvements in the Index-Calculus Algorithm for Solving the Discrete Logarithm Problem over F_p. [online] Available at: http://www.isg.rhul.ac.uk/ prai175/ISGStudentSem07/IndexCalculus.pdf [Accessed 17 Sep. 2019].