An Introduction to Elliptic Curves and Applications in Cryptography

A Thesis
Presented to
The Established Interdisciplinary Committee for
Mathematics and Computer Science
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Josh Klein Valente

May 2022

Approved for the Committee
(Mathematics and Computer Science)

Jerry Shurman

Adam Groce

Acknowledgments

First I would like to thank my family for all of their love and support. I am extremely grateful for all of the opportunities and encouragement that you have always given me.

I would also like to thank my advisors Jerry Shurman and Adam Groce. Jerry has been a constant force, guiding me through this process day in and day out, and especially all of the time in between. I have had the opportunity to enjoy many of Adam and Jerry's well thought out classes, the best I have ever taken.

My time at Reed would not have been the same without my wonderful friends. From all of our rock climbing adventures, to the late nights flopping quad sevens over your pocket jiggities after cooking dinner together, I am extremely grateful for all of you and the memories we share. I look forward to many more.

Last, but certainly not least, I would like to thank my amazing partner Vilma for all of her love and support throughout this process.

Preface

I began this thesis after a summer where I spent the little time that I wasn't shooting pool, working on the old linux machines that the people in the pool hall know as *"The Jukebox"*. During that time, a friend named Tucker Twomey introduced me to many new ideas in computer science, one of which was the marvel of elliptic curve cryptography. Tucker enthusiastically explained that by using elliptic curves, we achieve a similar level of security as RSA, but with a much shorter key length. At the time, I was just beginning my exploration into cryptography and would soon learn all about the formal theories of modern cryptography in a class here at Reed.

While thinking about ideas for my thesis, I pondered many topics, but cryptography, though I had not done much formal study yet, was always the most interesting to me. It is just one of those things that we depend upon so often in our daily lives, yet when asked what makes our online payments secure, few could say more than magic.

For these reasons, this thesis is a journey into the world of cryptography, specifically exploring elliptic curves which are a major part of modern cryptography. This thesis is written for an undergraduate student of mathematics and computer science who wants to begin exploring cryptography and may use this work as a jumping off point to access higher level concepts and read current literature on the subject.

In the first chapter, we will review concepts from group theory, introduce elliptic curves, and explain the creation of the elliptic curve group, with special care for the associativity condition.

In the second chapter we introduce some foundational ideas from cryptography, and then apply them to the elliptic curve group that we have constructed.

In the third and final chapter we introduce two fundamental algorithms for computing discrete logarithms on elliptic curves, touching on a few modern improvements and current ideas in cryptography research.

List of Abbreviations

DLP	Discrete Logarithm Problem
ECDLP	Elliptic Curve Discrete Logarithm Problem
BSGS	Baby Step Giant Step
ECDH	Elliptic Curve Diffie-Hellman
ECEG	Elliptic Curve El Gamal

Table of Contents

Abstract

In this thesis we explain the fundamentals of modern cryptography with a focus on elliptic curves, detailing the elliptic curve group that is extremely important to modern cryptography. We build a foundation in modern cryptography which we use to construct and implement elliptic curve versions of common applications such as Elliptic Curve Diffie-Hellman and Elliptic Curve El Gamal. We also analyze and implement the Baby Step Giant Step and Pollard's Rho algorithms which compute discrete logarithms on elliptic curves. We explain how these algorithms are better than a brute force search and explore ways to improve them further with some ideas from modern cryptography research.

Chapter 1

Elliptic Curve Group Law

1.1 Group Theory

The following section is intended to briefly summarize the prerequisite group theory needed to understand the Elliptic Curve Group Law and subsequent sections detailing the discrete logarithm problem on elliptic curves. As such, readers comfortable with basic group theory may wish to proceed to **Section 1.2**.

Definition 1.1.1. A **group** is a set G that together with binary operation "$\circ$" satisfies the following group axioms:

1. *(Existence of Identity)* There exists an element e in G such that for every element a in G, $e \circ a = a \circ e = a$. This element e is called the identity for $\circ$ on G, and is unique.

2. *(Existence of Inverses)* For every element a in G, there exists an element b in G such that $a \circ b = b \circ a = e$.

3. *(Associativity)* For any a, b, c in G, $(a \circ b) \circ c = a \circ (b \circ c)$.

Two examples of familiar groups include the set of integers with the binary operation addition and the set of positive real numbers with the binary operation of multiplication.

Definition 1.1.2. An **abelian group** or **commutative group** is a group which satisfies the additional property of commutativity. That is, for any elements a, b in G, $a \circ b = b \circ a$. For example, the set of integers with the binary operation addition satisfies this requirement.

Definition 1.1.3. The **order** of a *group* G is the number of elements in the group. This can be denoted $|G|$ or $\text{ord}(G)$

Definition 1.1.4. A given group G is called **finite** if the order of the group is finite.

Definition 1.1.5. The **order** of an element a is the smallest positive integer n such that $a^n = e$, where a^n indicates $a \circ a \circ a \circ \cdots \circ a$ with n copies of a. For example, an element a for which $a^4 = a \circ a \circ a \circ a = e$ is said to have order 4, or $\text{ord}(a) = 4$.

1.2 Elliptic Curves

The set of equations whose solutions define what are known as *elliptic curves* have a multitude of applications in mathematics. For the purposes of this work, we will only examine the finer details when they are relevant to the application of elliptic curves in Elliptic Curve Cryptography and solving the Elliptic Curve Discrete Logarithm Problem.

Definition 1.2.1. An **elliptic curve**, which we will often denote E, is the set of solutions over a given field K to a cubic equation also known as a degree three polynomial. This cubic equation must have distinct roots to be called an elliptic curve. The requirement of distinct roots is discussed further below. One way of expressing such an equation is as a **Weierstrass equation** given by

$$y^2 = x^3 + Ax + B$$

where A and B are elements of the given field K.

Distinct Roots

Here we briefly discuss the roots of cubic equations and touch on cases of repeated and distinct roots. When studying roots, we can use the discriminant of a cubic curve to determine whether the curve has repeated or distinct roots. For a Weierstrass equation, the discriminant is denoted Δ and can be expressed in terms of the roots of the equation r_1, r_2, and r_3 as

$$\Delta = ((r_1 - r_2)(r_1 - r_3)(r_2 - r_3))^2.$$

For a cubic equation, a discriminant equal to zero indicates that the equation has repeated roots. A cubic curve with repeated roots is known as a singular curve and either contains a double or triple root. A double root is called a node and can be seen in Figure 1.1 below.

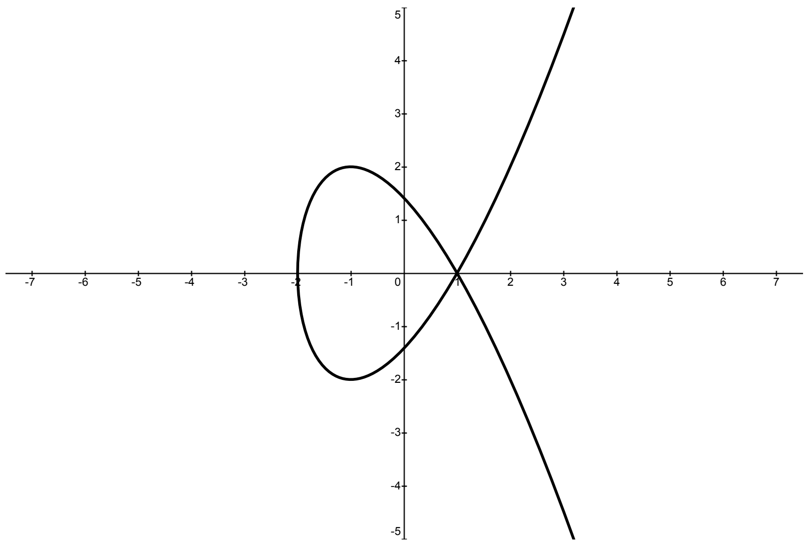

Figure 1.1: A singular cubic with a node, $y^2 = x^3 - 3x + 2$

A triple root is called a cusp and can be seen in Figure 1.2 below.

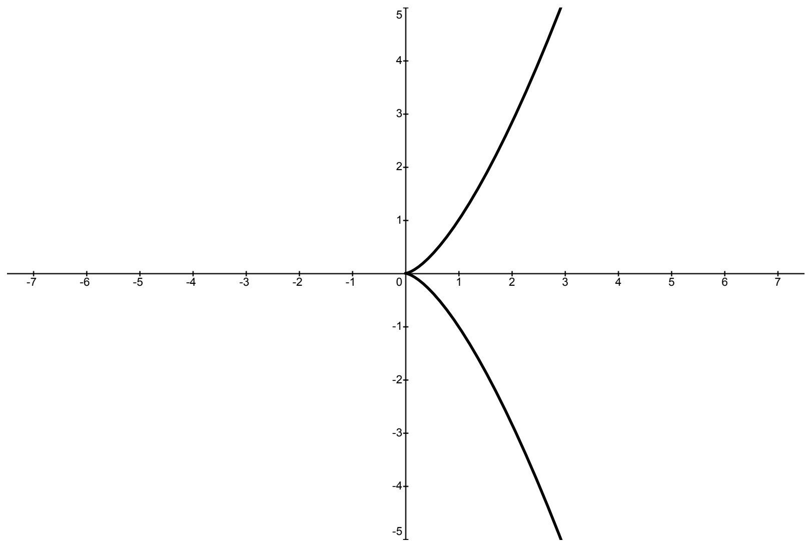

Figure 1.2: A singular cubic with a cusp, $y^2 = x^3$

However, these singular curves do not work well with the addition operation we will define and so we restrict our focus to non-singular curves. Thus we must ensure that the discriminant of a given curve is non-zero. The general discriminant for a cubic of the form $P(X) = ax^3+bx^2+cx+d$ is given by $\Delta = b^2c^2-4ac^3-4b^3d-27a^2d^2+18abcd$. Thus, for a Weierstrass cubic equation where we have $a = 1$, $b = 0$, $c = A$, and $d = B$, the discriminant is given by

$$\begin{aligned}\Delta &= 0^2A^2 - 4(1)(A)^3 - 4(0)^3(B) - 27(1)^2(B)^2 + 18(1)(0)(A)(B)\\ &= -4A^3 - 27B^2\end{aligned}$$

So, the condition that the discriminant of a Weierstrass curve is non-zero is equivalent to requiring that A and B are chosen such that $4A^3 + 27B^2 \neq 0$.

Restricting our study to non-singular curves, or elliptic curves, we will see curves with one or three real roots. A curve with one real root will look similar to Figure 1.3 below.

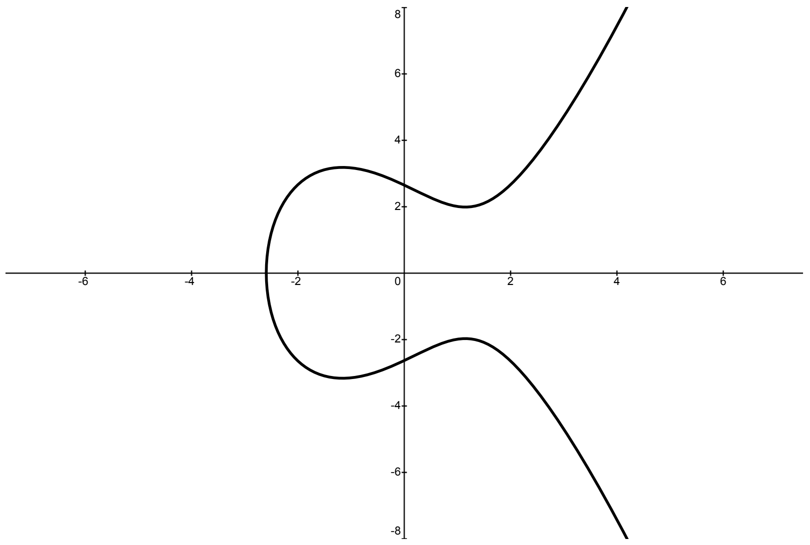

Figure 1.3: A cubic curve with one real root, $y^2 = x^3 - 4x + 7$

A curve with three real roots will look similar to Figure 1.4 below.

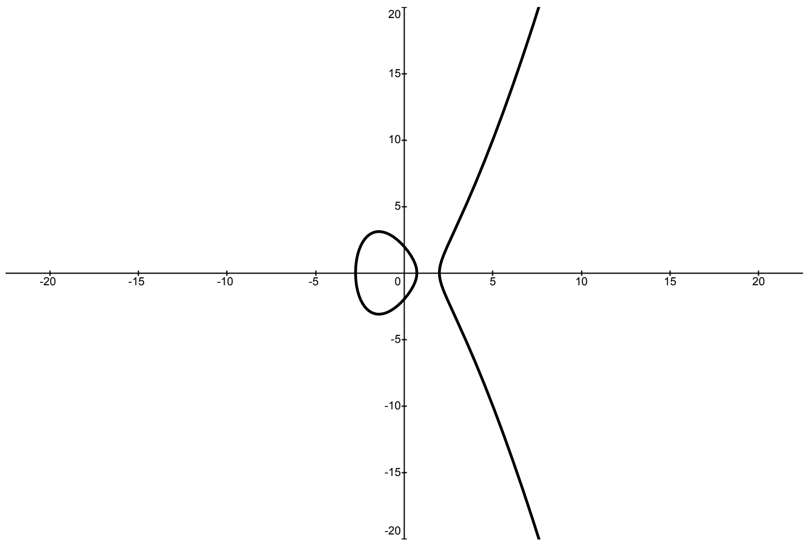

Figure 1.4: A cubic curve with three real roots, $y^2 = x^3 - 6x + 4$

1.3 Elliptic Curve Group Law

To begin the construction of the **elliptic curve group** we demonstrate a binary operation which we will call addition for the set of rational points on a given curve. By Bézout's theorem, a powerful result from algebraic geometry, we know that the number of times a line will intersect a cubic curve is three.[1] Thus, given any two

[1]It is important to note that the use of Bézout's theorem assumes that we are using the projective plane which includes points at infinity, that we allow points of multiplicity such as counting a point of tangency as multiple intersections, and lastly that we permit complex numbers for coordinates [1].

rational points on an elliptic curve E, we can discover a third point on the curve by extending a rational line through the two given points. The rationality of this third point becomes clear by searching for the intersection of the three points with the curve, which yields a cubic equation that has rational coefficients. Since we know that two of the roots of this equation are rational, the third is necessarily rational too.

We formalize the process of composing points and denote such a composition with a "$*$". Thus, given the points P and Q, we draw the line that intersects them, and call this line's third intersection with the cubic $P * Q$. This is demonstrated in Figure 1.5 below.

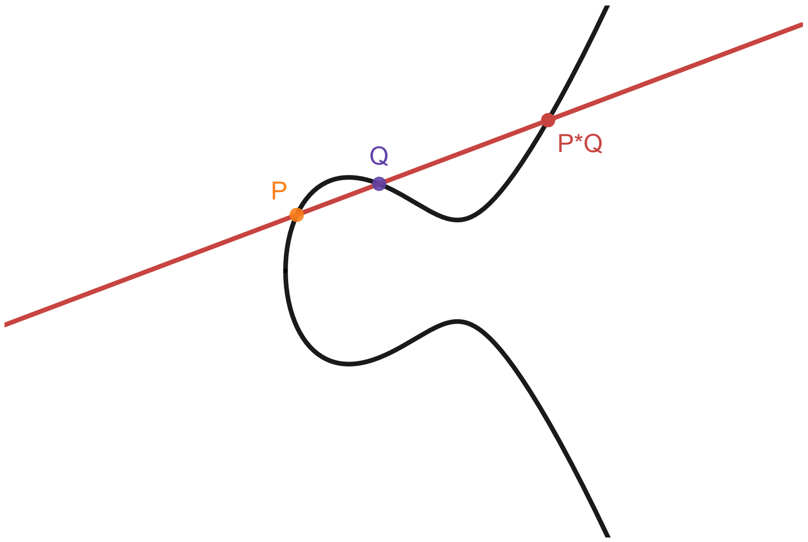

Figure 1.5: Composition of 2 points on an elliptic curve

To generalize this composition law to all elements in the set of points on an elliptic curve we must examine two cases which require further explanation. First, how can we compose a point P with itself?

The solution is discovered by considering the series of compositions of a point P with different points Q which get successively closer to the point P. As the point Q approaches the point P, the line intersecting both points approaches the line tangent to the point P. Then, it is naturally defined that the composition of a point P with itself is given by the point $P * P$ which is the point where the cubic intersects the line tangent to curve at the point P. It may help to continue thinking of the tangent line to P as a line that intersects the curve twice. The first intersection being through P as it arrives, and the second being through P again as it leaves. This is similar to how we compute $P * Q$ in that we draw the line that goes through P and Q. However, this time, P counts as both the first *and* second intersection, leaving the third intersection as the new point on our curve.

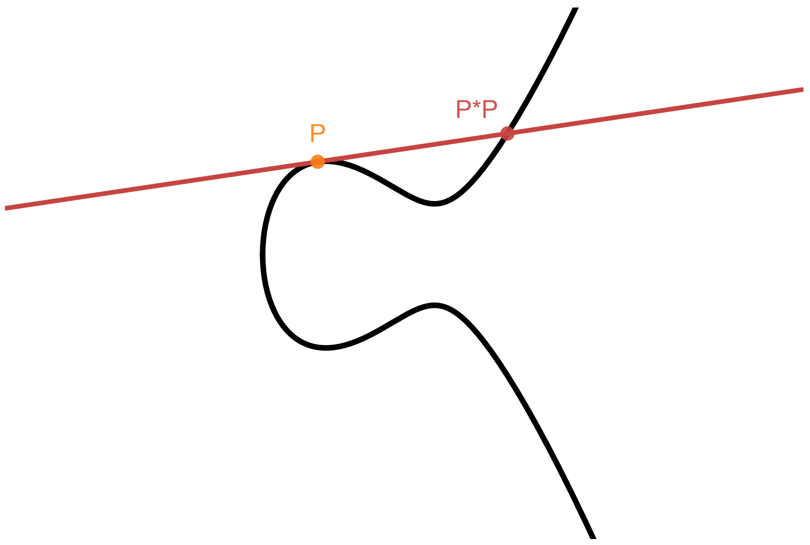

Figure 1.6: Composition of a point with itself on an elliptic curve

Next we must consider a more subtle case in which the line intersecting our two points is vertical as in Figure 1.7 below.

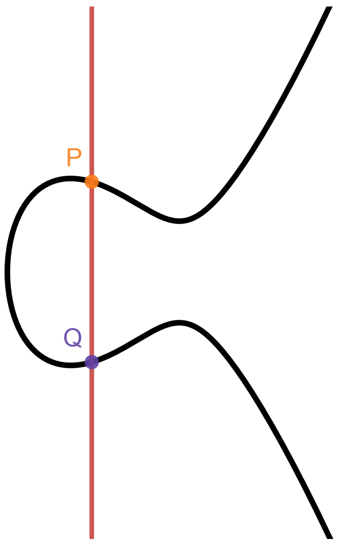

Figure 1.7: Composition case of vertical line

To reconcile this problem we can again consider what happens when we anchor a point P, and shift the point Q closer to the same x coordinate as P. As Q approaches from either side, we see that the line that we draw through P and Q intersects the curve a third time at an increasingly great magnitude in the y coordinate. With this guide we assert that to fix this apparent problem, we must append the set of rational points on a curve with a special **point at infinity**, denoted ∞. This point can be thought of as existing at the "top" and "bottom" of the y-axis. However, for fields other than the real numbers, such as finite fields which are often used in cryptography, this notion of top and bottom is absent, and so it is best to think about the point at infinity as a symbolic point with the property that it exists on a vertical line. This is in fact the way it is treated in cryptographic implementations. Thus we have solved the problem of finding a third intersection for the vertical line by creating it!

1.3.1 Addition

Now, we define the group law which is denoted by a $+$ and called the addition operation since the group will be commutative.

Definition 1.3.1 (Group law,+)**.** Given points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on an elliptic curve E, $P + Q$ is the third intersection with E of the line through P and Q, which is then reflected over the horizontal axis for $P \neq Q$, $x_1 \neq x_2$, and $P, Q \neq \infty$. In the case where $x_1 = x_2$ and $y_1 \neq y_2$, $P + Q = \infty$.

This process is made clearer by Figure 1.8 in which the solid red line passes through P and Q to find the third intersection with E called $P * Q$. Then the point $P * Q$ is reflected over the horizontal axis which is shown using the green dashed line to get the point $P + Q$ on E as desired.

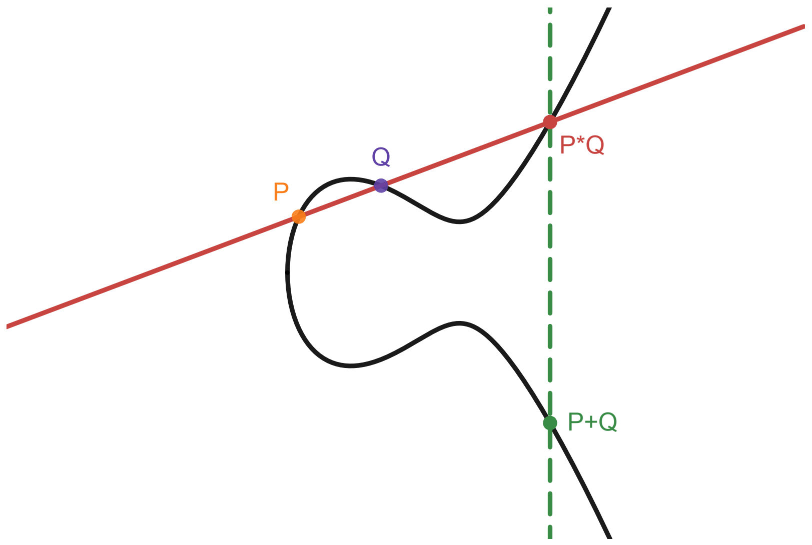

Figure 1.8: The Group Law on an elliptic curve.

If $P = Q$ and $y_1 \neq 0$, $P + Q$ is found by reflecting the intersection of E and the tangent line to the point $P = Q$ over the horizontal axis. This is demonstrated in Figure 1.9 below.

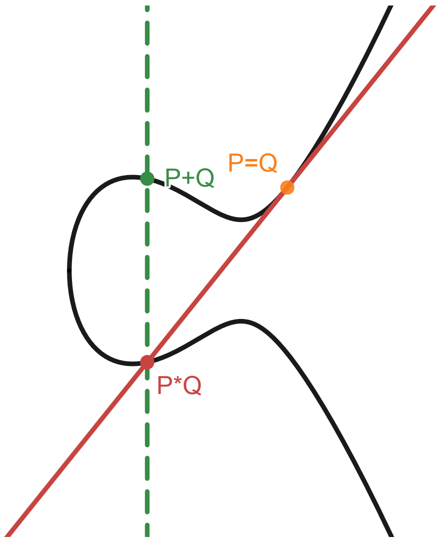

Figure 1.9: Addition of $P + Q$ where $P = Q$ and $y_1 \neq 0$.

If $P = Q$ and $y_1 = 0$, then $P + Q = \infty$ since the tangent line to $P = Q$ intersects the point at infinity, which is still the point at infinity after reflecting over the horizontal axis.

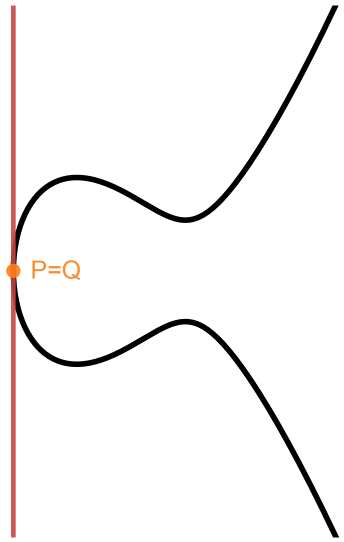

Figure 1.10: Addition of $P + Q$ where $P = Q$ and $y_1 = 0$.

More formally, the 4 cases are as follows:
Given an elliptic curve $E : y^2 = x^3 + Ax + B$ with points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ where $P_1, P_2 \neq \infty$, we find $P_3 = (x_3, y_3) = P_1 + P_2$.

1. If $x_1 \neq x_2$ we have $P_3 = (m^2 - x_1 - x_2, m(x_1 - x_3) - y_1)$ for $m = \frac{y_2 - y_1}{x_2 - x_1}$
2. If $x_1 = x_2$ and $y_1 \neq y_2$, then $P_3 = \infty$
3. If $P_1 = P_2$ and $y_1 \neq 0$, then $P_3 = (m^2 - 2x_1, m(x_1 - x_3) - y_1)$ where $m = \frac{3x_1^2 + A}{2y_1}$

4. If $P_1 = P_2$ and $y_1 = 0$, then $P_3 = \infty$

1.3.2 Elliptic Curve Group Axioms

Now we will show that with the defined group operation of addition, the set of rational points on an elliptic curve including the point at infinity satisfy the group axioms. Further, we will show that the elliptic curve group is abelian.

First, the addition operation is clearly commutative. That is, for any points $P, Q \in E$, $P + Q = Q + P$. This is necessarily the case since the line intersecting P and Q is the same line that intersects Q and P which means that $P * Q = Q * P$. Then, since this same point is reflected across the horizontal axis, we have that $P + Q = Q + P$ as desired.

Next we claim that the identity of the group is the element ∞, the "point at infinity". That is, for any point P in E, $\infty + P = P + \infty = P$. This is shown in Figure 1.11 below.

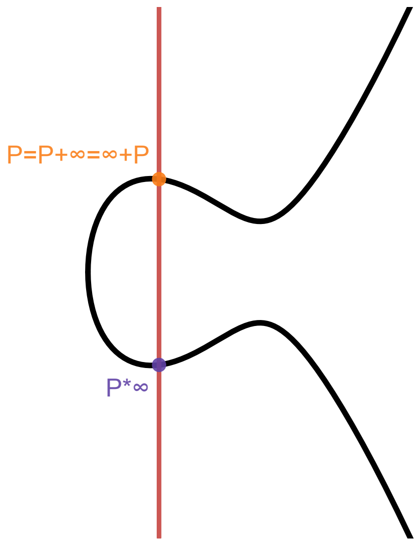

Figure 1.11: Identity element example

Next we show the existence of inverses. Given P on elliptic curve E, $-P$ is the point P reflected over the horizontal axis. This is easily verified by taking the sum of any P and its inverse $-P$ which will result in the identity element ∞. That is, $-P + P = \infty$. This is shown in Figure 1.12 below.

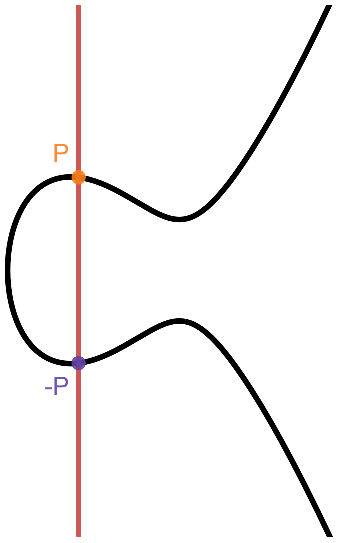

Figure 1.12: Inverse element example

Proving Associativity

Next we will show associativity of the group law, the formal details of which will be discussed in a later section. The following explanation will be done in the geometric style of the previous axioms, but it is especially important to point out here that there is not a simple yet rigorous algebraic analog to this visually intuitive geometric proof. Some methods of proving associativity turn out to be far beyond the scope of this thesis, but we will discuss other rigorous methods after engaging with a geometric argument below.

For associativity, we want that for all $P, Q, R \in E$,

$$(P + Q) + R = P + (Q + R)$$

By definition of the $+$ operator this is equivalently

$$((P + Q) * R) * \infty = \infty * (P * (Q + R))$$

Then, since composing a point with the point at infinity simply reflects the point over the horizontal axis, it is sufficient to show that

$$(P + Q) * R = P * (Q + R)$$

Geometric Proof

We begin with a visual demonstration of the left-hand side in which we add $P + Q$ and compose the resulting point with R. This is shown in Figure 1.13 below.

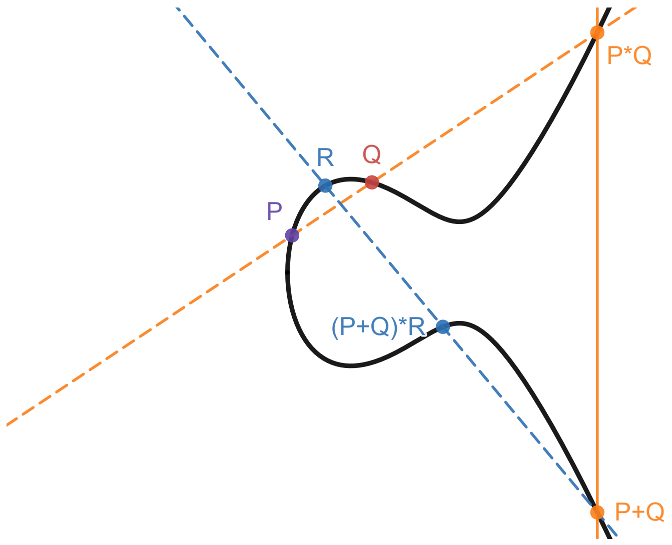

Figure 1.13: $(P+Q)*R$

Next we proceed with the right-hand side for which we add $Q+R$ and compose the resulting point with P. This is shown in Figure 1.14 below. Note that the construction from Figure 1.13 is shown in the background. For associativity, this point $P*(Q+R)$ is supposed to be the same as $(P+Q)*R$ but we have yet to show that these points are not distinct.

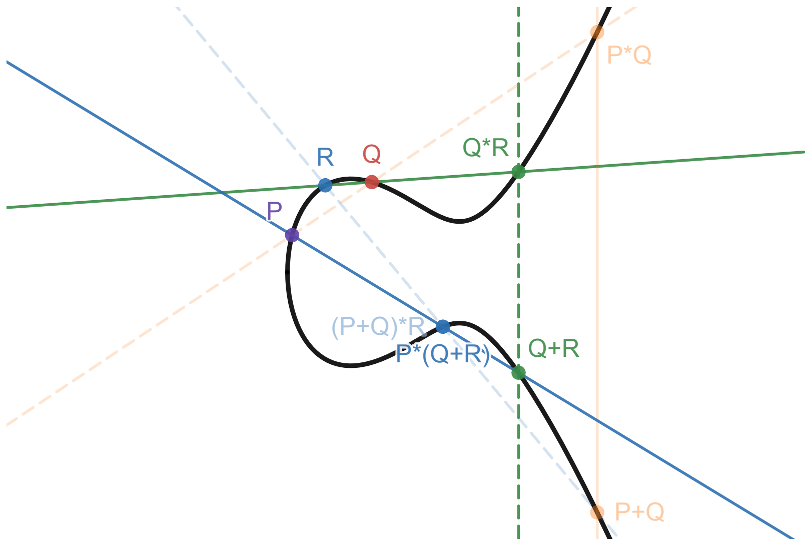

Figure 1.14: $P*(Q+R)$

Now using Figure 1.15 below, consider the point at which the dashed blue line through $P+Q$ and R intersects the solid blue line through P and $Q+R$. If this point of intersection lies on the curve, then this point is both $(P+Q)*R$ and $P*(Q+R)$ which means that $(P+Q)*R = P*(Q+R)$ as desired.

In Figure 1.15 below, we know that each of the eight points

$$P, Q, R, \infty, P * Q, P + Q, Q * R, \text{ and } Q + R$$

lies on both a dashed and solid line in addition to being a point on the curve E.

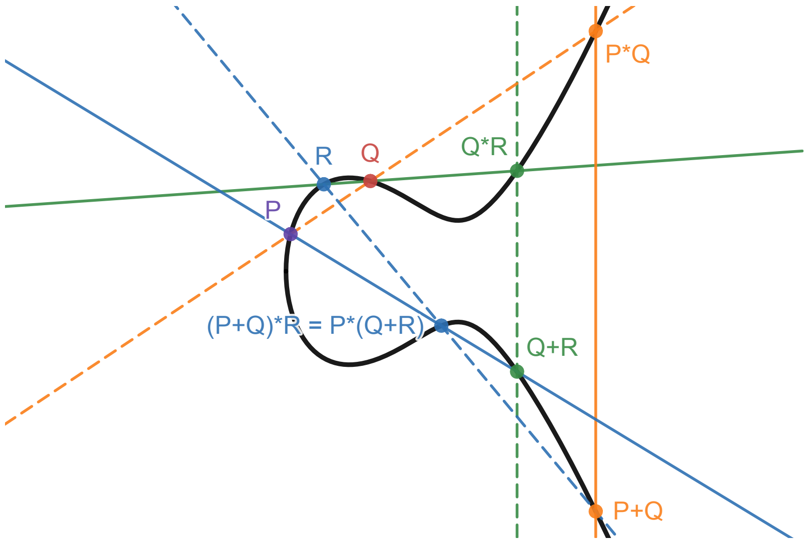

Figure 1.15: Full associativity construction

A ninth point, which we have not yet shown to be on E, is given by the intersection of the dashed blue line through $P + Q$ and R with the solid blue line through P and $Q + R$.

Looking at the two figures below in Figure 1.16, we see that the two sets of three lines which these nine points lie upon form two degenerate cubics. This is to say that we can multiply the equations in each respective set of three lines, solid and dashed, to get two cubic equations.

Further we assert that since we know that the curve E goes through eight of the nine points that the two degenerate cubics go through, the curve E must also go through the ninth point.

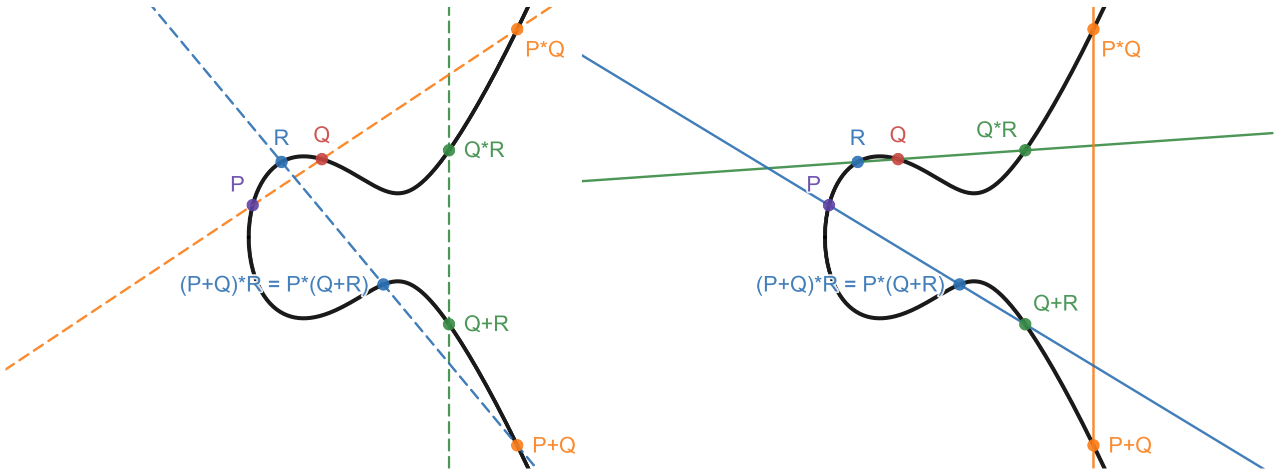

Figure 1.16: Sets of dashed and solid equations

To explain why this assertion is true we must explore the properties of a cubic curve. A general cubic curve is defined by ten coefficients. Such a general equation looks like

$$ax^3 + bx^2y + cxy^2 + dy^3 + ex^2 + fxy + gy^2 + hx + iy + j = 0.$$

Since multiplying these coefficients by a constant results in the same cubic, there are really only nine dimensions in which we can modify a possible cubic. When a point of some cubic is specified, one linear constraint is put on the cubic equation's coefficients such that the dimension that one can modify the cubic in reduces by one. Thus, if you specify one point, the resulting family of possible cubic equations are eight dimensional, and for two points, seven dimensional. Similarly, if eight points are specified, there is only one dimension in which the cubic can be be modified and so these cubic equations are called a one dimensional family.

The one dimensional family of cubic equations passing through eight specified points on the two degenerate cubic curves can be represented by $\lambda_1 F_1 + \lambda_2 F_2$ where $F_1 = 0$ and $F_2 = 0$ are cubic equations for the two degenerate cubic curves, and λ_1, λ_2 are some scalars which we use to make linear combinations of the two curves. Thus the curve E must be of the form $\lambda_1 F_1 + \lambda_2 F_2 = 0$ for some λ_1, λ_2.

Recall that the equations for the two degenerate cubics have a solution at the point where the dashed blue line through $P + Q$ and R intersects the solid blue line through P and $Q + R$. Thus, F_1 and F_2 are zero at that point. Next, recall that the one dimensional family of curves are of the form $\lambda_1 F_1 + \lambda_2 F_2 = 0$. Thus at the ninth point, since F_1 and F_2 are zero, our one dimensional family of curves must also be zero which implies that any curve in this family, namely E, must contain that ninth point, given that it contains the other eight. Thus the geometric proof of associativity is complete and we have shown that the set of rational points on an elliptic curve including the point at infinity, form a group with the addition operation as desired.

Alternate Methods

Now, while intuitive, this is certainly not a complete proof and we have avoided many more subtle issues. Some of these can still be intuitively explained such as a case of P and Q in which you must count P twice when a composition of P and Q results in a line tangent to the elliptic curve. However, a rigorous formal proof of associativity is much more difficult than one may imagine. In our introductory study of elliptic curve cryptography, one may wish to simply believe the geometric intuition and continue forward to more applied topics that build on the nice properties of the abelian elliptic curve group. Many of the high level methods for proving associativity that may be considered more elegant or less tedious unfortunately depend on complex machinery and concepts from algebraic geometry that are beyond the scope of this thesis, and so after a brief discussion of two more accessible methods we will proceed to further topics.

First we explore one of the more standard but tedious proofs of associativity in Lawrence C. Washington's *Elliptic Curves, Number Theory and Cryptography* spanning over twelve pages [2]. The proof idea is similar to the geometric proof idea we explored above in that we start with points P, Q, and R on a Weierstrass elliptic curve E and compute $-((P+Q)+R)$ and $-(P+(Q+R))$. However, Washington computes these using explicit formulas for the group law. Washington must also prove that the point at which $-((P+Q)+R) = -(P+(Q+R))$ lies on E, but he uses paramatarized curves rather than our geometric properties argument above. The majority of the proof is spent dealing with three special cases. First we encounter the possibility that any of the nine points involved are points at infinity, for which the solution is to use projective coordinates. Next, some of the six lines being constructed might be tangent to the elliptic curve, which could make pairs of points equal. For this problem Washington elects to construct definitions which carefully quantify the order to which a line intersects a curve at some given point. Lastly, two lines in the construction may be equal, which falls outside the scope of the cases proven to solve the previous issue. To remedy potential problems this would cause, Washington breaks the issue down into nine distinct cases. Some of these require only intuitive reasoning, while others rely on more complicated lemmas cited from earlier in the proof [2]. With this, the associativity of elliptic curve addition is proven.

Next we explore a more novel approach to the associativity proof developed by Thomas Hales and Rodrigo Raya. However, Hales and Raya's proof of associativity of the group law uses Edwards curves rather than the Weierstrass equations we have seen so far. Weierstrass curves, as explained by Hales and Raya, are the most commonly presented form of elliptic curves, but they are unsatisfactory in the necessary treatment of their various exceptional cases [3]. That is, the lack of a single encompassing formula for the group law on Weierstrass curves make them less than ideal for proving associativity of the group law. Hales and Raya also importantly note that besides complicating the work of the theorist, this shortcoming of necessary exceptional cases can lead to unforeseen side-channel attacks by adversaries as well as more complicated implementations that may lead to mistakes [3]. Edwards curves on the other hand are a way of expressing an elliptic curve which according to Edwards himself "simplifies formulas in the theory of elliptic curves and functions" [4]. In our case this means we get to avoid the need for case-wise treatment of the addition law on elliptic curves since Edwards expresses the curves in the form

$$x^2 + y^2 = a^2 + a^2x^2y^2$$

which allows for an explicit definition of the group law:

$$X = \frac{1}{a} \cdot \frac{xy' + x'y}{1 + xyx'y'}, \quad Y = \frac{1}{a} \cdot \frac{yy' - xx'}{1 - xyx'y'} \text{ [4]}$$

However, instead of proceeding from here by hand, Hales and Raya give a computer assisted proof which allows for large complicated algebraic calculations to be verified electronically. This methodology, while scrutinized in the community at first, allows Hales and Raya to complete an extremely elementary proof without needing to perform the impractical task of verifying all of the relevant calculations needed for such a proof. The proof assistant they used is called the Isabelle/HOL proof assistant and with it they show that "every Edwards elliptic curve...satisfies the axioms of an abelian group". Since any elliptic curve with a characteristic other than two is isomorphic to a curve in Edwards form, this result generalizes for most applications of elliptic curves [3].

Chapter 2

Cryptography and Elliptic Curves

The following chapter will introduce and discuss some basic ideas from modern mathematical cryptography. The reader who is familiar with basic modern mathematical cryptography may wish to proceed to **Section 2.2**.

2.1 Modern Cryptography and Number Theory

Modern cryptography is one of the most important foundations of our modern society. Cryptography is not only important for governments and militaries, but it is also found in almost all of the technology we rely on everyday. For example, sending an email, downloading a file, or making a payment online are all actions that heavily rely on the security and integrity provided by modern cryptography. Despite this presence in many facets of our lives, knowledge of cryptographic theory and fundamental theories on security are possessed by an alarmingly small number of people. This is true even in the field of computer science where it is increasingly common for programmers to be tasked with creating secure applications despite having little to no understanding of fundamental ideas in modern cryptography [5]. This is a worrying idea since the technology being developed by more and more people is becoming more and more integrated with our lives and personal information.

Before the formalization of modern cryptography, which we might now understand as *"the study of mathematical techniques for securing digital information, systems, and distributed computations against adversarial attacks"*, cryptography was thought of as an art, based on creating novel systems which relied on intuitive ideas about security [5]. An example of such a system is the famous Caesar cipher. The Caesar cipher is a type of substitution cipher[1] in which a message can be either encoded or decoded by shifting its letters three letters forward or backward respectively. That is, the letter e would be encoded as the letter h, and the encoded letter f would be decoded to c. This process is easier to conceptualize when viewing a map from

[1]A substitution cipher is a method of encryption in which elements of a plaintext message are substituted for something else according to some scheme to create a ciphertext. This is in contrast to transposition ciphers in which the elements of a plaintext message are rearranged, but not substituted out of the encoded ciphertext.

the plain-text alphabet, or decoded message alphabet, to the cipher-text alphabet, or encoded message alphabet shown below.

$$\begin{array}{rl} \text{plaintext: } & ABCDEFGHIJKLMNOPQRSTUVWXYZ \\ & \updownarrow \\ \text{ciphertext: } & DEFGHIJKLMNOPQRSTUVWXYZABC \end{array}$$

One subtlety that is avoided by showing the complete map is that when shifting "off the end" of the alphabet, we wrap back around to the other side. Try decoding the following ciphertext using the mapping above or just following the "shift back three" rule.[2]

ciphertextExample: WKHJUHDWJLJLQWKHVNB

However, this system and many variations on it are not only extremely vulnerable to attack, but rely on the communicating parties, often called Alice and Bob, to already have a shared secret key which their adversary, Eve, does not know. In the case of the Caesar cipher, the secret key is $k = 3$. The idea of using a secret, or private key, to encode messages is called private-key cryptography, and modern private-key cryptosystems boast secure efficient communication. On the other hand, modern systems also benefit from the ideas of public-key cryptography. This is a scheme in which two parties need not have any shared secret before their first communication. That is, Alice and Bob who do not share any secret key and may not have ever communicated before, can communicate securely over a public, or insecure channel which Eve can eavesdrop on. This is no small accomplishment. The ability to communicate securely over an insecure channel is what makes the modern internet and communication services we depend on possible. That is not to say that private-key cryptography is unimportant. In fact, private-key cryptography systems are often much more efficient than their public counterparts. This means that in practice the public-key system is often only used to first exchange or obtain a shared key, which can then be used for further communication using a private-key system.

To better appreciate the miracle of a public-key exchange, imagine that Alice and Bob are siting in a room on either side of Eve. Then, by only speaking aloud to each other and performing some private computations, Alice and Bob can create a shared secret that only they know, despite Eve listening to everything that has been said. This is what Whitfield Diffie and Martin E. Hellman proposed in their landmark paper *New Directions in Cryptography* [6] published in 1976. At the time, Diffie and Hellman realized that small electronics with capacity for cryptographic security were beginning to become extremely common and that the old standard of securely distributing private keys to each of these devices, or even just sharing a private key with someone you would like to speak with securely, would become quite impractical [6]. Thus, Diffie and Hellman developed a public-key exchange system that reaps all of the benefits discussed above and had a profound impact on the world.

[2] plaintextExample: THEGREATGIGINTHESKY

The impact of such a system was no secret to Diffie and Hellman who began their paper with the famous line,

"We stand today on the brink of a revolution in cryptography. " [6]

2.1.1 The Discrete Logarithm Problem

Before proceeding to the Diffie-Hellman key exchange, we must first understand the mathematical problem that enables such a system. There are many variations of the discrete logarithm problem, but the one most relevant for our study is based upon the finite field of integers modulo p where p is a prime number, which we denote $\mathbb{F}_p$. Also, let g be a primitive element of the field $\mathbb{F}_p$ which is to say that all non-zero elements of $\mathbb{F}_p$ can be generated by taking powers of g. That is to say the multiplicative cyclic subgroup $\mathbb{Z}_p^*$ can be expressed as $\mathbb{Z}_p^* = \{g^0, g^1, g^2, \ldots, g^{p-2}\}$. Then we present the discrete logarithm problem as, given primitive element $g \in \mathbb{F}_p$ and some non-zero $h \in \mathbb{F}_p$, find the smallest integer x such that the following relation holds:

$$g^x \equiv h \pmod{p}$$

The solution to the problem, x, is called the discrete logarithm and is given by $\log_g(h)$.

For certain finite fields and their respective multiplicative subgroups, especially smaller ones, the discrete logarithm problem is not particularly difficult to solve. For example, consider the multiplicative cyclic subgroup of $\mathbb{F}_5$ given by $\mathbb{Z}_5^*$ with $g = 2$ and $h = 3$. Then, solving $2^x \equiv 3 \pmod{5}$ can be done simply by testing the few possible values for x until we find the solution that $2^3 \equiv 3 \pmod{5}$ and so $x = 3$. However, this problem becomes significantly difficult as the finite field becomes larger with larger choices of p. Often primes that are thousands of bits long are used for security in modern cryptographic applications. We will discuss methods for solving difficult instances of the discrete logarithm problem in Chapter 3. Before discussing the way in which the difficulty of cryptographic problems are formally described, we give a more general definition of the discrete logarithm problem which will be better adapted to use with elliptic curves in **Section 2.2.1**.

Definition 2.1.1. Let G be a group with operation $\circ$, elements $g, h \in G$, and $x \in \mathbb{Z}$. The **Discrete Logarithm Problem** is given by $\underbrace{g \circ g \circ \cdots \circ g}_{x \text{ copies of } g} = h$, where the solution is x, the number of copies of g needed to obtain h using the group law.

As prophesied in the 1976 Diffie-Hellman paper *New Directions in Cryptography* [6], the change from cryptography as an ancient art into a modern science in which cryptographic systems could be provably secure means that all modern systems and ideas in cryptography can be grounded in rigorous formal definitions which give a tangible meaning to security. For our purpose of introducing some basic ideas in cryptography which will allow for a study of applications using elliptic curves, we will proceed without formal security definitions. However, the interested reader is encouraged to explore the security experiments, definitions, and proofs in Katz and Lindell's *Introduction to Modern Cryptography* [5] that can be applied to the schemes we define in this chapter.

2.1.2 Diffie-Hellman Key Exchange

Now we may proceed to the Diffie Hellman key exchange. Recall that in this exchange, Alice and Bob have not necessarily communicated with each other before. Despite this, they will communicate publicly, or over an insecure channel, and manage to create a shared secret. This secret, which we will call a *key*, is then often used to communicate securely using some chosen method of encryption. The version of the exchange presented in the original Diffie-Hellman paper [6] uses a the multiplicative subgroup of $\mathbb{F}_p$ given by $\mathbb{Z}_p^*$ where p is prime. We will proceed similarly here.

Alice and Bob begin by establishing some public parameters. They agree on a prime p and an element $g \in \mathbb{Z}_p$, and publish these values for anyone to see.

Next, Alice chooses a secret integer a that she does not ever share with anyone.
Bob follows suit choosing a secret integer b which he does not ever share with anyone.

Now, Alice computes $A \equiv g^a \pmod{p}$ and sends this publicly to Bob.
Bob similarly computes $B \equiv g^b \pmod{p}$ and sends this publicly to Alice.

Lastly, Alice computes $s_A \equiv B^a \pmod{p}$ and Bob similarly computes $s_B \equiv A^b \pmod{p}$. The values that they have computed are the same value since

$$s_A \equiv B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \equiv s_B \pmod{p}$$

Alice and Bob can now use this shared secret $s_A \equiv s_B \pmod{p}$ to encrypt and send messages over an insecure channel using an encryption scheme of their choice.

The security of this exchange seems to depend on the hardness of the discrete logarithm problem for the chosen group. We will examine this in detail below, but first consider what information an adversary, say Eve, has.

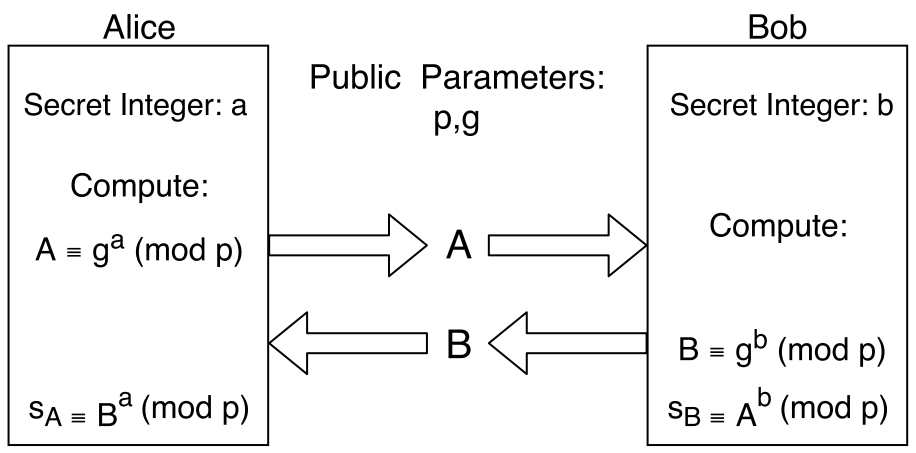

Figure 2.1: Diffie-Hellman key exchange

As seen in Figure 2.1 above, Eve knows the values of p, g, A, and B. However, Eve does not know the secret numbers of Alice or Bob. These values, a and b are only known by those individual parties and should not be easily computable. Thus, for an eavesdropper adversary Eve to discover the shared secret $s_A \equiv s_B \pmod{p}$ she must solve for either a in $A \equiv g^a \pmod{p}$ or b in $B \equiv g^b \pmod{p}$ and then use one of these secret values to compute the shared key.

As mentioned earlier, it may seem that Alice and Bob's shared secret will be secure if Eve is unable to solve the discrete logarithm problem. However, this is not exactly true. If Eve finds a solution to the relevant discrete logarithm problem, she will certainly also be able to discover the shared secret of the Diffie-Hellman key exchange. However, the problem of finding the shared secret of the Diffie-Hellman exchange is a slightly different problem which we will define as follows.

Definition 2.1.2. Given a cyclic group $\mathbb{G}$, generator $g \in \mathbb{G}$, as well as public keys A and B where $A \equiv g^a \pmod{p}$ and $B \equiv g^b \pmod{p}$, the **Diffie-Hellman Problem** is to compute the value $g^{ab} \pmod{p}$.

We can see that the Diffie-Hellman problem is not any harder than the discrete logarithm problem, but this does not necessarily mean that the inverse is true. It is unknown in general whether an efficient algorithm to solve the Diffie-Hellman problem could be reduced to efficiently solve the discrete logarithm problem. However, there have been reductions made to show the two problems to be of equivalent difficulty for certain groups and conditions, especially those important to cryptography-related applications [7].

2.1.3 El Gamal

The next natural step after the Diffie-Hellman public-key exchange is to create a complete public-key cryptosystem which allows Alice and Bob to share private messages over an insecure or public channel. The El Gamal public-key cryptosystem is similarly based upon the hardness of the discrete logarithm problem and was developed by Taher Elgamal in the 1986 paper titled *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms* [8]. We will now explain in general how Bob may send a message m to Alice over an insecure channel using the El Gamal cryptosystem adapted from the Diffie-Hellman public-key exchange, using the multiplicative subgroup of $\mathbb{F}_p$ given by $\mathbb{Z}_p^*$.

Without loss of generality, Alice begins by choosing a large random prime p and an element $g \in \mathbb{F}_p$ preferably of large prime order.

Alice then chooses a secret integer $a \in \mathbb{Z}_p$ which is called her *private key.*

Next, Alice computes $A \equiv g^a \pmod{p}$ and publishes $p, g,$ and A. The value A is called her *public key.*

Bob, the sender, has a message $m \in \mathbb{Z}_p$.

Bob chooses a random element $r \in \mathbb{Z}_p$. It is important to note that he can only use this r for a single message or the scheme is vulnerable to adversarial attack.

Next, Bob computes and sends C_1 and C_2 publicly to Alice where $C_1 \equiv g^r \pmod{p}$ and $C_2 \equiv mA^r \pmod{p}$.

Alice receives C_1, C_2 and begins decryption by finding that $C_1^a = (g^r)^a = (g^a)^r = A^r$.

Alice then finds the inverse of A^r, $(A^r)^{-1}$ and uses this to solve for Bob's message m since $C_2 A^{-r} = mA^r A^{-r} = m$.

Thus Bob has successfully communicated message m to Alice. The idea of this process is that C_2 masks Bob's message and C_1 gives Alice enough information to find m. An adversary Eve, eavesdropping on Alice and Bob's insecure channel will know the values p, g, and A. With these values, one way of attacking the system would be to solve the discrete logarithm problem in $\mathbb{F}_p$, which in this case is to solve for Alice's private key $a = \log_g(A)$.

2.2 Elliptic Curve Cryptography

In the remaining sections of this chapter we will define and discuss the elliptic curve group variations for the cryptographic schemes we have covered thus far. Elliptic curve cryptography (ECC) is considered especially valuable in our world of developing technology. This is because the length of the keys required to achieve the same level of security as competing cryptographic systems such as RSA are significantly shorter. For example, an ECC key that is 512 bits provides the level of security equivalent to an RSA key that is $15,360$ bits long [9]. Smaller keys mean that the CPU can work less, use less memory, and perform computations faster, all of which are extremely important for increasingly common devices with low computation power that are expected to maintain cryptographic security. For these reasons, the NSA includes elliptic curve algorithms including elliptic curve Diffie-Hellman in its recommended cryptographic algorithms list called *Suite B* provided by the National Institute of Standards and Technology (NIST) [10].

Double and Add

As we will discover below, calculating the multiple of a point on an elliptic curve is extremely important for cryptographic systems. That is, given a point P on an elliptic curve, we must find nP for some integer n. Since the group law is addition, this multiplication is simply repeated addition in which we add n copies of P to each other. However, this process becomes computationally intensive as higher and higher values of n are required. Thus, an algorithm to more efficiently compute multiples of points on an elliptic curve has been developed. This algorithm is known as the *double and add* algorithm, which is analogous to the method used for computing large powers in other groups known as *exponentiation by squaring*, or the *square and multiply* method. Before we begin with the algorithm, recall that to double a point, say P, we perform the group operation on P with itself.

Now, given the problem of finding nP given $n \in \mathbb{Z}$ and a point P on an elliptic curve, we can efficiently compute nP using the **double and add** algorithm as follows:

Begin by writing n as a sum of powers of two so that

$$n = n_0 \cdot 2^0 + n_1 \cdot 2^1 + n_2 \cdot 2^2 + n_3 \cdot 2^0 + \cdots + n_r \cdot 2^r$$

where each n_i is either 0 or 1. For example, if $n = 23$, we write

$$n = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^0 + 1 \cdot 2^4.$$

Next we proceed with the algorithm by doubling P and multiplying by the corresponding n_i for each ith doubling. The sum of all of the iterations will total nP.

Following the example from above where $n = 23$, we start with the point P and will end with a sum of $2^0P + 2^1P + 2^2P + 2^4P = 23P$.

Start with the point P.
Double P to get $2P$.
Then add the point $2P$ to P to get $2^0P + 2^1P$.
Next double $2P$ to get 2^2P.
Add 2^2P to the sum to get $2^0P + 2^1P + 2^2P$.
Next double 2^2P to get 2^3P.
This time we do not add 2^3P to the sum because $n_3 = 0$ rather than 1.
Next double 2^3P to get 2^4P.
Finally, add 2^4P to the sum to get $2^0P + 2^1P + 2^2P + 2^4P = 23P$.

Thus we have found $23P$ in only four doublings and three additions. In general, this alogrithm allows us to compute nP in less than $2\log_2(n)$ doubling and adding steps. There are some issues to worry about as n gets larger since the rational coordinates of nP will quickly become unmanageably long, though there are methods to remedy this such as performing all computations modulo n as discussed in Silverman and Tate's *Rational Points on Elliptic Curves* [1].

2.2.1 Elliptic Curve Discrete Logarithm Problem

Now we define the Elliptic Curve Discrete Logarithm Problem (ECDLP), which is like the general discrete logarithm problem discussed above in Section 2.1.1, but due to the more complicated group operation and efficient doubling method associated with computing multiples, can be used to create even more difficult problems utilized by the cryptographic schemes we cover later in the chapter.

Definition 2.2.1. The **Elliptic Curve Discrete Logarithm Problem** is a case of the **Discrete Logarithm Problem** from Definition 2.1.1 in which we will use the elliptic curve group $E(\mathbb{F}_p)$ with the group operation of elliptic curve addition. The notation $E(\mathbb{F}_p)$ refers to the group formed by the set of points on elliptic curve E, including the point at infinity, over the field $\mathbb{F}_p$. Then, given $Q, P \in E(\mathbb{F}_p)$, the **Elliptic Curve Discrete Logarithm Problem** is to find $n \in \mathbb{Z}$ such that

$$nP = Q$$

where $nP = \underbrace{P + P + \cdots + P}_{n \text{ copies of } P}$ and $+$ denotes the group law.

2.2.2 Elliptic Curve Diffie-Hellman

Next we describe the Diffie-Hellman key exchange for the elliptic curve group. This exchange is one of the cryptographic algorithms recommended for use at the TOP SECRET level by the NSA [10]. This algorithm is advised for use with the curve named P-384 which is an elliptic curve given by $y^2 = x^3 - 3x + b$ where

$$\begin{aligned} b =& 27580193559959705877849011840389048093056905856361568521428707301 9\\ & 886892413098608651362607648837451077654397612305 75. \end{aligned}$$

The elliptic curve Diffie-Hellman key exchange proceeds as follows:

Alice and Bob again begin the Diffie-Hellman exchange by establishing some public parameters. This time they are working over the group $E(\mathbb{F}_p)$ and agree on a prime p and an initial point $P \in E(\mathbb{F}_p)$, publishing these values for anyone to see.

Next, Alice chooses an integer a for her private key.
Bob also chooses an integer b for his private key.

Now, Alice computes her public key $A = aP$, and she sends this publicly to Bob. Bob similarly computes his public key $B = bP$, and sends this publicly to Alice.

Lastly, Alice computes $s_A = aB$ and Bob similarly computes $s_B = bA$. The values that they have computed are the same value since

$$s_A = aB = abP = baP = bA = s_B$$

Thus the exchange concludes and Alice and Bob can again use their shared secret $s_A \equiv s_B$ to encrypt and send messages over an insecure channel using an encryption scheme of their choice.

Using the open-source mathematical software system known as **SageMath**, or just Sage, we compute values for the following example.

Example 2.2.1 (Elliptic Curve Diffie-Hellman Key Exchange).
We will use the elliptic curve named "secp256k1" which is the curve used by Bitcoin for its public-key cryptography. The curve is given by $y^2 = x^3 + 7$ and the group is $E(\mathbb{F}_p)$ with $p = 2^{256} - 2^{32} - 977$. Now we create the curve with the following SageMath command using the given parameters:

```
E=EllipticCurve(GF(p),[0,0,0,0,7])
```

Which SageMath recognizes as

```
Elliptic Curve defined by y^2 = x^3 + 7 over Finite Field of size
11579208923731619542357098500868790785326998466564056403945758400790 88
34671663
```

and informs us of the number of points on the curve:

```
115792089237316195423570985008687907852837564279074904382605163141518 1
61494337
```

Now, we choose a public point P by choosing a generator of the group. Note that both the curve and the point P are published for anyone to see. The SageMath for this step is

```
P = E.gens()[0]
```

which provides the following output for the point P:

```
P=(654851700490331417555725529320915554403957221429841935940728734832 2
8125624049:731633777630311410325012597797384410942478878349412111874 27
503803434828368457)
```

Now, Alice chooses a secret integer a and Bob chooses a secret integer b. These values are `a = 57; b = 33718`.

Alice and Bob then create their public keys by "multiplying" the public point P by their secret integers respectively. Then both Alice and Bob share their public keys publicly so that the other can see. We do this in SageMath as `alicePubKey = a*P` which yields

```
alicePubKey=(956018876796720697205016272681719696628735793618584465317
09824014709437415644:9694035418083871844025893977033867926417915351315
3432672507632290838185310161)
```

and `bobPubKey = b*P` which yields

```
bobPubKey=(29287130335063145902497261824772202811748938160521455195734
67482840591354826:6737294024947928605828341288426096534083916742403895
8198266947688164317897554).
```

Last, Alice and Bob compute the shared key. Alice computes abP by doing `a*bobPubKey`, and Bob computes $baP = abP$ by doing `b*alicePubKey`.

Both of these calculations result in the same value, the shared key, given by

```
(770647631054687155719231133805344383080843284143619393208789466809849023115
34:1052630273793398531496747559793039805214084903619284672530723398113363037
10059)
```

Thus we successfully conclude the elliptic curve Diffie-Hellman key exchange example.

2.2.3 Elliptic Curve El Gamal

Next we proceed with the El Gamal public-key encryption scheme using the elliptic curve group.

First Alice and Bob agree on a prime p, elliptic curve group $E(\mathbb{F}_p)$, and a point $P \in E(\mathbb{F}_p)$, all of which are public information.

Now Alice chooses a private key, some integer a.

Alice then computes and publishes her public key, the point $A = aP$.

Next Bob, who has the message $M \in E(\mathbb{F}_p)$, chooses a random integer r.

Then Bob computes $C_1 = rP$ and $C_2 = M + rA$, which he sends publicly to Alice.

This allows Alice to compute M as follows:

$$C_2 - aC_1 = (M + rA) - a(rP) = M + r(aP) - a(rP) = M$$

Thus Bob has sent a message $M \in E(\mathbb{F}_p)$ to Alice using the elliptic curve El Gamal public-key cryptosystem. Eve is again aware of all public information, p, $E(\mathbb{F}_p)$, and P. To decrypt M, Eve can attempt to solve the elliptic curve discrete logarithm problem by finding Alice's private key a.

Also, it is important to note that a plaintext message is not trivially converted to a point in $E(\mathbb{F}_p)$. However, there are methods which can be used to do this.

For example, Bob can choose a random M as a mask for encoding his plaintext message. This idea, as well as an improved message expansion factor is achieved in the Menezes–Vanstone variation of the El Gamal public-key cryptosystem [11].

Next we again use SageMath to compute the following example.

Example 2.2.2 (Elliptic Curve El Gamal Demonstration)**.**
Alice and Bob begin by choosing a b-bit prime p for the group $E(\mathbb{F}_p)$ which we choose using the SageMath commands:

```
b = 9
x = randrange(2^(b-1),2^b);
p = next_prime(x)
```

which results in a prime `p = 431`

Then they choose an elliptic curve group, which we will choose in SageMath by generating coefficients A, B for a random elliptic curve over $\mathbb{F}_p$ as shown below. We include the condition that the generated curve is not singular for reasons explained in the subsection of Section 1.2 on Distinct Roots.

```
A,B = randrange(p), randrange(p)
while (4*A^3 + 27*B^2) % p == 0:
    A, B = randrange(p), randrange(p)
```

This yields the following curve which we create with the command

```
E = EllipticCurve(GF(p), [A,B])
```

which outputs

```
Elliptic Curve defined by y^2 = x^3 + 93*x + 207 over Finite Field of size 431
```

The last set-up step is to choose a public point $P \in E$ which we do by choosing P with `P = E.gens()[0]` in SageMath which yields `P=(393:343)`.

Alice then chooses her private key a with `a = randrange(P.additive_order())` which chooses a random integer less than the order of additive order of P, since larger integers will not contribute more randomness to Alice's public key. In this case $a = 357$, and Alice now uses this to create her public key.

Alice's public key A is computed with `A = a*P` which yields the point `A=(88:302)` which Alice publishes for Bob to see.

Now, Bob has the message `msg = 'Hello Alice.  -Bob'` and knows Alice's public key. However, Bob must first encode his message to an integer. Here we choose to do this by encoding the characters of our message into the character encoding known as UTF-8 which are bytes of base-256 since this is the number of characters that UTF-8 can represent. We do this in SageMath as follows:

```
order = []
for char in msg:
    order.append(ord(char))

encodedMsg = ZZ(order,256)
```

In the code snippet above, ord maps chars to a UTF-8 value in $[0, \dots, 256]$, and the ZZ function creates encodedMsg = order[0] $* 256^0 +$ order[1] $* 256^1 + \dots$ In our example, the resulting encoded message is

```
encodedMsg=33495559872634850626126698439616725017928.
```

The next step in this encoding is to ensure that the digits of `encodedMsg` have a base that is the order of the point P. This is done with the command

```
encodedMsg.digits(P.additive_order())
```

which we assign to the variable `mRep`. This yields the following digits:

```
mRep=[408, 52, 382, 392, 25, 85, 305, 99, 348, 199, 202, 265, 13, 28,
                                221, 21]
```

The last step of encoding is to represent each n-th digit of the encoded message as the n-th point in the list of points in E. At this point we have an encoded message compatible with the encryption scheme and compute the pair $C = (C_1, C_2)$ which Bob sends publicly to Alice. This is demonstrated for our example in the SageMath code below. In this code, M is Bob's encoded message and r is Bob's random integer.

```
C = []
for i in range(len(mRep)):
    r = randrange(P.order())
    m = mRep[i]
    M = E.points()[m]
    C.append([k*P, M + k*Q])  # (C_1,C_2) = (kP, M+kQ)
```

The resulting list of C_1, C_2 pairs is shown below:

```
C=[[(134 : 430), (383 : 392)],
   [(147 : 237), (102 : 254)],
   [(110 : 335), (112 : 252)],
   [(186 : 307), (205 : 267)],
   [(300 : 195), (409 : 72 )],
   [(317 : 238), (358 : 147)],
   [(289 : 79 ), (373 : 245)],
   [(284 : 339), (366 : 379)],
   [(197 : 202), (67  : 85 )],
   [(142 : 79 ), (269 : 157)],
                    [(369 : 284), (147 : 237)],
                    [(127 : 178), (63  : 31 )],
                    [(105 : 133), (61  : 11 )],
                    [(287 : 84 ), (157 : 408)],
                    [(42  : 173), (197 : 229)],
                    [(396 : 25 ), (311 : 118)]]
```

Alice receives this and can obtain the decrypted message `Mdec` by performing

$$C_2 - aC_1 = \text{Mdec}$$

for each C_1, C_2 pair. We compute this in SageMath with

```
Mdec = []
for i in range(len(C)):
    Mdec.append(C[i][1] - a*C[i][0])
```

for which we look up points to get base-n digits back from encoding using

```
[ECurvepoints.index(Mdec[i]) for i in range(len(Mdec))]
```

which we asign to `mDec`. This gives Alice the same digits we saw in Bob's mRep, which Alice calls mDec. These digits that have been decrypted are `mDec = [408, 52, 382, 392, 25, 85, 305, 99, 348, 199, 202, 265, 13, 28, 221, 21]` which are indeed the same as Bob created earlier. Alice can further decode this message to get the plaintext back using the following code which reverses Bob's encoding steps.

```
n = ZZ(mDec,P.additive_order())
nl = n.digits(256)
decoded = ‘’
for digit in nl:
    decoded = decoded+chr(digit)
```

The variable `decoded` then holds Bob's message string

`‘Hello Alice.  -Bob’`

as desired. Thus we conclude the elliptic curve El Gamal SageMath example.

Chapter 3

Generic Algorithms to solve the ECDLP

In the previous chapter we explored ways in which the difficulty of the discrete log problem, especially for an elliptic curve group, could be used to create cryptographic algorithms including the complete public-key cryptosystem El Gamal. The security of these cryptographic algorithms, which are relied upon for everything from sending electronic payments to "TOP SECRET" government communications, are at the very least vulnerable to adversaries who can efficiently solve the discrete logarithm problem. For this reason, it is valuable to spend time exploring the various methods of solving discrete logarithms, specifically on elliptic curves. Finding an efficient method however, is believed to be quite a difficult task. Formally, the idea that the problem of solving discrete logarithms is hard means that we do not know of any algorithm that solves the problem in *polynomial time*. A polynomial time algorithm is considered reasonably practical using modern technology since such an algorithm can solve a problem in some multiple of n^k steps, where k is some non-negative integer and n is a measure of complexity. In the case of the elliptic curve discrete logarithm problem, or ECDLP, the value n is given by the bit-length of p, the size of the field. We will first examine the naive method of solving discrete logarithms on elliptic curves, brute force.

3.1 Brute Force

Brute force is the simplest method for solving discrete logarithms, but as will soon become evident, it is quite insufficient. In the study of these algorithms below, we assume that the discrete logarithm that we are solving for exists since discrete logarithms that exist are the ones relevant to the cryptographic algorithms we have examined. Before moving to the ECDLP, consider the discrete logarithm problem over $\mathbb{Z}_p^*$ for a relatively small prime p. Recall that the discrete logarithm problem is to find x such that $g^x \equiv h \pmod{p}$. The brute force method simply computes increasing powers of g modulo p and compares the values with h. If it finds a match, we have a solution and the algorithm is completed. We implement the brute force method using SageMath to show the following examples for both versions of the discrete logarithm problem discussed. In a small group such as $\mathbb{Z}_{89}^*$ with primitive root $g = 19$, we can solve the discrete logarithm given by

$$19^x \equiv 77 \pmod{89}$$

using the brute force algorithm to get the solution $x = 55$ on a standard personal computer in an average of 320μs. Similarly, in the small elliptic curve group $E(\mathbb{F}_{89})$ where E is given by $y^2 = x^3 + 4x + 27$ with points $P = (9, 13)$ and $Q = (23, 2)$ on E, the ECDLP is to find n in

$$n(9, 13) = (23, 2).$$

Using a standard personal computer, finding the solution $n = 78$ with the brute force method takes an average of 9420μs. While both of these calculations are relatively fast, the brute force method becomes incredibly inefficient as p increases in length, especially for the ECDLP. More formally, the time complexity of our brute force method is based upon the number of bits of p, and so the brute force algorithm has an exponential asymptotic running time complexity. There are no known sub-exponential, or polynomial time algorithms for the general discrete logarithm problem, but there are much more efficient methods than brute force, and we will explore some of these below, focusing only on solving the more difficult elliptic curve discrete logarithm problem.

Before exploring other algorithms, let us first assert the brute force method as the trivial complexity for the algorithms we will compare it to. That is, for an element P of order N, the ECDLP is trivially solved in $\mathcal{O}(N)$ steps using $\mathcal{O}(1)$ space since we may need to check all values of nP while $n < N$, and only need to store two values at once.

3.2 Baby Step Giant Step (BSGS)

The first ECDLP solving algorithm that we will explore is called *Baby Step Giant Step*. This algorithm is credited to Daniel Shanks who published a paper [12] containing the algorithm in 1971. The BSGS algorithm works on any finite cyclic group, but is typically used for prime order groups since the Pohlig-Hellman algorithm [13] is more efficient for composite order groups. Baby Step Giant Step, or BSGS runs in $\mathcal{O}(\sqrt{p})$ time where p is the size of the group, which is much more efficient than the naive brute force method. The algorithm is not as efficient with space however, and runs in $\mathcal{O}(p)$ space. We will explore methods of improving the efficiency of BSGS later, but first we explain how the algorithm is used to solve the ECDLP as follows.

Baby Step Giant Step Algorithm

We are given the ECDLP of finding $n \in \mathbb{Z}$ for $nP = Q$ with $P, Q \in E(\mathbb{F}_p)$.

First, for any $n \in \mathbb{Z}$ we can write $n = am + b$ for some $a, b, m \in \mathbb{Z}$. Thus, we can re-express the ECDLP $nP = Q$ as $bP = Q - amP$ since

$$(am + b)P = Q \implies amP + bP = Q \implies bP = Q - amP$$

Then, the Baby Step Giant Step algorithm below will compute values for bP and values for $Q - amP$. The calculations of bP are considered "baby steps" since we compute $1P, 2P, 3P, \ldots,$ while the calculations of $Q - amP$ are considered "giant steps" since m is very large and we compute $1mP, 2mP, 3mP, \ldots$ during the giant steps. The values of bP are stored in a table, and by checking each giant step against the table of baby steps we will to find a solution to the ECDLP since this can check all points from $0P$ to nP. The algorithm is shown explicitly below.

Algorithm : Baby Step Giant Step

```
1: Compute $m = \lceil \sqrt{p} \rceil$
2: for each b in $(0, \ldots, m)$ do
3:     Compute bP and store value in table
4: for each a in $(0, \ldots, m)$ do
5:     Compute $amP$ and then $Q - amP$
6:     Check table for $bP$ such that $bP = Q - amP$
7:     if point $bP$ exists such that $bP = Q - amP$ then
8:         Return $n = am + b$
```

While this algorithm is substantially faster than brute force, it still runs in exponential time and the storage required to implement this most basic version of the algorithm becomes a limiting factor as the size of the group becomes large. To put things in perspective, consider the elliptic curve group used for bitcoin, named *secp256k1*. This curve and group are given by $y^2 = x^3 + 7$ and $E(\mathbb{F}_p)$ with $p = 2^{256} - 2^{32} - 977$. The maximum amount of space needed for the BSGS algorithm is $\sqrt{p}$. Assuming that we can store an elliptic curve point using roughly 32 bytes, the BSGS algorithm would require $\sqrt{p} \cdot 32$ bytes $\approx 1.0889 \times 10^{40}$ bytes of storage! This is almost 20 orders of magnitude greater than the total amount of data estimated to be stored in the entire world by the year 2025 [14]. Thus, after a demonstration of the BSGS algorithm on the ECDLP implemented in SageMath, we will discuss ways in which the algorithm can be improved, though we will never reach a reasonable storage requirement using BSGS for a group as large as the group based upon the *secp256k1* curve.

Example 3.2.1. (BSGS SageMath)
We begin by generating a random finite field and subsequent random elliptic curve which we will use as our group.

The finite field is generated using a random prime in the range given by `p=random_prime(10**9,10**8)` which we use to create the finite field `F = GF(p)`.

This yields `The finite field F_p where p = 171143057`

Next we generate an elliptic curve using parameters `A=F.random_element()` and `B=F.random_element()` to create elliptic curve `E=EllipticCurve(F,[A,B])` which yields `Elliptic Curve:  y^2 = x^3 + 160611062x + 32469798`.

We then create the points P and Q to be used in the ECDLP with `P=E.gens()[0]` and `Q=E.random_element()`.

Thus we have following elliptic curve discrete logarithm problem.

```
ECDLP: n (30801773,48491602) = (19974888,155634166)
```

To begin the BSGS algorithm we compute m with `m=ceil(sqrt(p))` which yields `m = 13083`.

Now we begin computing the baby steps and giant steps. In this example we will compute both sets before comparing since we are randomly generating small elliptic curve groups which are often quite weak. By computing all of the steps first, we see the worst case running time on these curves.

The baby steps are calculated with `babySteps=[b*P for b in [0..m-1]]` and the giant steps are calculated with `giantSteps=[Q+(-a*m)*P for a in [0..m-1]]`.

We then find a baby step that matches a giant step using `Set(babySteps).intersection(Set(giantSteps))[0]` which is assigned to the variable `matching`. In this example, `matching = (131535049, 60176572)`.

Next we find the index of a and b for the matching element which allows us to find that `4147 * P = Q - 7185 * N * P` since `index_b = babySteps.index(matching) = 4147` and `index_a = giantSteps.index(matching) = 7185`.

Thus we have found `n=index_b+index_a*m` to be `n = 94005502`.

We verify that this is the solution to the ECDLP presented by comparing the values `nP = (19974888, 155634166)` and `Q = (19974888, 155634166)` which indeed match and so we are done. On a standard personal computer, the computation time for the baby steps in this example averaged 4.89 seconds, while the list of giant step computations took an average of 13.9 seconds to complete.

3.2.1 Improving the Efficiency of BSGS

The Baby Step Giant Step algorithm is one of the most straightforward ECDLP solving algorithms to implement. However, the most basic approach is not practical for use when working in very large groups due to the storage problem discussed earlier. Attempts at improving the BSGS algorithm often target the storage issue and look-up time.

For example, a recent paper [15] examines the decryption process of an applied El Gamal scheme, for which BSGS is considered a common method used to solve discrete logarithms. The work focuses on storage reduction and is able to achieve 7 to 14 times storage reduction on the popular *secp* family of curves by employing a truncated lookup table which aims to reduce any redundant information without increasing the cost of decryption [15].

Another group of researchers released a paper [16] that improved the general BSGS algorithm by speeding up the most costly operation in the algorithm, point addition on elliptic curves. Their method for improving the speed of the algorithm was to "reduce the cost of each individual group operation" by using the fact that given points P and Q on a curve, one can compute $P + Q$ and $P - Q$ in less than the cost of performing two group operations by exploiting inversions [16].

Improvements like these are significant, especially when combined with precomputation time or special cases like a discrete logarithm on an interval. However, the finer details of these improvements are beyond the scope of this work, and so we will next examine another method, Pollard's Rho.

3.3 Pollard's Rho

The Pollard's Rho algorithm is another algorithm that is used to attack elliptic curve discrete logarithms. It was first published by John Pollard in 1975, in the paper *A Monte Carlo Method for Factorization* [17]. The Pollard's Rho method runs in roughly the same time as Baby Step Giant Step, but only requires $\mathcal{O}(1)$ space which makes it much more practical for use on large groups, though it may still take an incredibly long time to find a solution.

Pollard's Rho Algorithm

The Pollard's Rho method begins by solving a different problem whose solution can be used to compute n in the ECDLP given by $nP = Q$ for some $P, Q \in E(\mathbb{F}_p)$. The problem that Pollard's Rho solves is to find $a, b, A, B \in \mathbb{Z}$ such that $aP + bQ = AP + BQ$. Given a solution to this problem we find n as desired since

$$\begin{aligned} aP + bQ &= AP + BQ \\ \implies aP + bnP &= AP + BnP \\ \implies (a - A)P &= (B - b)nP \\ \implies n &= (a - A)(B - b)^{-1} \pmod{p} \end{aligned}$$

Now, the main idea for solving the problem presented is to generate a pseudorandom sequence of points X_i where $X_i = a_iP + b_iQ$. Such a sequence can be generated by a pseudorandom function f used to compute the next point in the sequence with $f(X_i) = a_{i+1}P + b_{i+1}Q$. That is, we use the pseudorandom function to generate the sequence by plugging in an X_i and receiving the next term in the sequence, X_{i+1}, since $f(X_i) = a_{i+1}P + b_{i+1}Q = X_{i+1}$. We will not explore the details of pseudorandom functions here, but refer interested readers to Katz and Lindell's book [5], which is a good place to start. Here, we will use a pseudorandom function with the assumption that given an X_i, the function will return the next point in the sequence with coefficients a_{i+1} and b_{i+1} which appear truly random to any adversary limited by polynomial time computation ability.

As we generate this sequence, we will eventually begin cycling through repeated points. This is necessarily the case since the number of points in our group is finite. Once a cycle is found, that is we find a matching point $X_i = X_j$, the solution to the ECDLP is found using the calculation above. The name of the algorithm, Pollard's Rho, comes from the shape of the letter ρ which is the shape that the sequence we generate creates when it cycles. This is shown in Figure 3.1 below for a sequence with a match at $i = 5$ and $j = 58$.

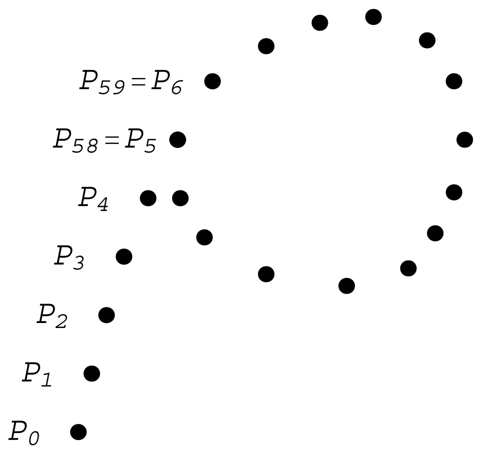

Figure 3.1: The ρ of Pollard's Rho [2]

Now, it seems that this algorithm does not use any less storage than Baby Step Giant Step since we are currently storing all of the points X_i until we find a match. However, by incorporating R.W. Floyd's cycle-finding algorithm as in Washington [2], we can improve our algorithm to only store two points. The idea is that once two points match, all subsequent pairs of points that are the same index distance from each other will match as well. This is simply because the sequence of X_i's is periodic with a period given by the difference, say d, in the indices of a matching pair of points. Thus we only need to store the pair of points (X_i, X_{2i}) for the current index $i = 1, 2, 3, \ldots,$ discarding all previous pairs. This gives us the much more practical space complexity of $\mathcal{O}(1)$. The time complexity of Pollard's Rho is more complicated, but if we assume that the pseudorandom function is truly random, the algorithm can be shown to have a worst case running time of $\sqrt{p}$ steps using the birthday paradox [18].

3.3.1 Improving the Efficiency of Pollard's Rho

The Pollard's Rho method of solving discrete logarithms has seen many improvements since Pollard's initial version in 1975. One way that the algorithm has been improved is by using a more efficient cycle finding algorithm proposed by Richard Brent. Brent's cycle finding algorithm is claimed to by 36% faster than Floyd's and results in a Pollard's Rho method that is 24% faster than the original [19].

Another method of improving Pollard's Rho is to run the algorithm on multiple processors, starting each at a randomly chosen point. This can be further enhanced by allowing the sequences generated on different processors to collide by using the same generating function as proposed by Oorschot and Wiener in a 1996 paper [20] who achieve a n-times speed increase for n processors running in parallel.

Last, a more recent paper [21] examines the Pollard's Rho assumption of a random walk simulated by the generating function. In this paper, Teske shows that this assumption does not hold for the original proposed algorithm, but by using a different class of random walks, one is able to achieve the performance claimed by the random walk assumption on arbitrarily large groups of prime order, a speed increase of roughly 20% [21].

References

[1] Joseph H. Silverman and John T. Tate. *Rational Points on Elliptic Curves*. Springer Publishing Company, Incorporated, 2nd edition, 2015.

[2] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman & Hall/CRC, 2008.

[3] Thomas Hales and Rodrigo Raya. Formal proof of the group law for edwards elliptic curves, 2020.

[4] Harold Edwards. A normal form for elliptic curves. *Bulletin of The American Mathematical Society - BULL AMER MATH SOC*, 44:393–423, 07 2007.

[5] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, third edition, 2021.

[6] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[7] K. Bentahar. The equivalence between the dhp and dlp for elliptic curves used in practical applications, revisited. pages 376–391, 12 2005.

[8] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[9] Yacine Rebahi, Jordi Jaen Pallares, Nguyen Tuan Minh, Sven Ehlert, Gergely Kovacs, and Dorgham Sisalem. Performance analysis of identity management in the session initiation protocol (sip). In *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pages 711–717, 2008.

[10] Suite b cryptography, Jul 2014.

[11] Alfred J. Menezes and Scott A. Vanstone. Elliptic curve cryptosystems and their implementation. *J. Cryptol.*, 6(4):209–224, sep 1993.

[12] Daniel Shanks. Class number, a theory of factorization, and genera. 1971.

[13] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms overgf(p)and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[14] David Reinsel, John Gantz, and John Rydning. Nov 2018.

[15] Panagiotis Chatzigiannis, Konstantinos Chalkias, and Valeria Nikolaenko. Homomorphic decryption in blockchains via compressed discrete-log lookup tables. Cryptology ePrint Archive, Report 2021/899, 2021. `https://ia.cr/2021/899`.

[16] Steven D. Galbraith, Ping Wang, and Fangguo Zhang. Computing elliptic curve discrete logarithms with improved baby-step giant-step algorithm. Cryptology ePrint Archive, Report 2015/605, 2015. `https://ia.cr/2015/605`.

[17] J. M. Pollard. A monte carlo method for factorization. *BIT*, 15(3):331–334, sep 1975.

[18] Jeong Han Kim, Ravi Montenegro, Yuval Peres, and Prasad Tetali. A Birthday Paradox for Markov chains with an optimal bound for collision in the Pollard Rho algorithm for discrete logarithm. *The Annals of Applied Probability*, 20(2):495 – 521, 2010.

[19] Richard P. Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20:176–184, 1980.

[20] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12:1–28, 1999.

[21] Edlyn Teske. On random walks for pollard's rho method. *Math. Comput.*, 70:809–825, 04 2001.