Operational notes

Document updated on **September 8, 2022**.

The following colors are **not** part of the final product, but serve as highlights in the editing/review process:

- text that needs attention from the Subject Matter Experts: Mirco, Anna,& Jan
- terms that have not yet been defined in the book
- things that need to be checked only at the very final typesetting stage (and it doesn't make sense to do them before)
- text that needs advice from the communications/marketing team: Aaron & Shane
- text that needs to be completed or otherwise edited (by Sylvia)

NB: This PDF only includes the following chapters: Introduction, Preliminaries, Algebra.

Todo list

Contents

Preface

This book began as a set of lectures and notes the author gave at the Zero Knowledge Summit – ZK0x02 in Berlin. It arose from the desire to collect the scattered information around the topic of zk-SNARKS and present them to an audience that does not have a strong background in cryptography.

After more than a decade of intense and fast-paced research on zk-SNARKs by mathematicians and cryptographers around the globe, the field is now racing towards full maturity, and we believe we can see the first saturation effects appear at the horizon. We hope this book will equip the reader with most of the grounding knowledge they need to tackle the vast literature in this remarkable field.

what does saturation mean here?

As security auditors at Least Authority, we audited quite a few SNARK-based systems. This book includes a distillate of our learnings from the time we spent on various audits, and serves as a polished version of what one might call an auditor's adaptation of classical lab notebooks.

We intend to let illustrative examples drive the discussion and present the key ideas of all basic concepts relevant to the understanding of zk-SNARKS with as little mathematics as possible. For those who are new to this topic, it is our hope that this book might be particularly useful as a first read and prelude to more complete or advanced expositions.

On the other hand, we also hope our beginner-friendly intentions do not leave the more advanced readers dissatisfied by our chosen lack of formality, so in cases where our discussion does sacrifice rigor, we will point the reader to the literature where a more thorough exposition can be found.

Chapter 1

Introduction

Add short description of zk-proofs

1.1 Target audience

How much mathematics do you need to understand zero-knowledge proofs? The answer, of course, depends on the level of understanding you aim for. It is possible to describe zero-knowledge proofs without using any mathematics at all; however, to read a foundational paper like Groth [2016] and to understand the internals of snark implementations, some knowledge of mathematics is needed to be able to follow the discussion.

zero-knowledge proofs

Without a solid grounding in mathematics, someone who is interested in learning the concepts of zero-knowledge proofs, but who has never seen or dealt with, say, a prime field 4.3.1, or an elliptic curve 5, may quickly become overwhelmed. This is not so much due to the complexity of the mathematics needed, but rather because of the vast amount of technical jargon, unknown terms, and obscure symbols that quickly makes a text unreadable, even though the concepts themselves are not actually that complicted. As a result, the reader might either lose interest, or pick up some incoherent bits and pieces of knowledge that, in the worst case scenario, result in immature and unsecure code.

I'd remove references from here

This is why we dedicated large parts of the book to explaining the mathematical foundations needed to understand the basic concepts underlying snark development. We encourage the reader who is not familiar with basic number theory and elliptic curves to take the time and read this and the following chapters, until they are able to solve at least a few exercises in each chapter.

We start our explanations at a very basic level, and only assume pre-existing knowledge of fundamental concepts like high school integer arithmetic. At the same time, we'll attempt to teach you to "think mathematically", and to show you that there are numbers and mathematical structures out there that appear to be very different from the things you learned about in high school, but on a deeper level, they are actually quite similar, as we will see in various examples below in this chapter.

We want to stress, however that this introduction is informal, incomplete and optimized to enable the reader to understand zero-knowledge concepts as efficiently as possible. Our focus and design choices are to include as little theory as necessary, focusing on a wealth of numerical examples. We believe that such an informal, example-driven approach to learning mathematics may make it easier for beginners to digest the material in the initial stages.

For instance, as a beginner, you would probably find it more beneficial to first compute a simple toy **SNARK** with pen and paper all the way through, before actually developing real-world production-ready systems. In addition, it's useful to have a few simple examples in your head before getting started with reading actual academic papers.

However, in order to be able to derive these toy examples, some mathematical groundwork is needed. This book therefore will help the inexperienced reader to focus on what we believe is important, accompanied by exercises that you are encouraged to recompute yourself. Every section contains a list of exercises in increasing order of difficulty, to help the you memorize and apply the concepts.

1.2 How to read this book

Since the MoonMath Manual aims to become a complete introduction to all topics relevant for beginners, there are ways of reading it. The first and most obvious one is linear, following the order of the chapters. We recommend this if you have little pre-existing knowledge of mathematics and cryptography. We start with basic concepts like natural numbers, prime numbers, and how we can perform operations on such sets in different kinds of arithmetic. Then we move on to algebraic constructs like groups, rings, prime fields and projective planes.

Early in the book, we develop examples that we gradually extend with the things we learn in each chapter. This way, we incrementally build a few real-world SNARKs over full-fledged cryptographic systems that are nevertheless simple enough to be computed by pen and paper to illustrate all steps in great detail.

Readers who mainly want to get a better understanding of elliptic curves as they arise in zero-knowledge-based proving systems might want to start with our introduction of the BLS6_6 curve [], which is a pen-and-paper, pairing-friendly elliptic curve. All concepts needed to understand this curve are introduced in chapter []. Programmers who develop real-world SNARKS frequently face the situation where it would be useful do some pen-and-paper computations before implementing the real thing. In this book, we specifically designed the BLS6_6 curve for this purpose. It is a pairing friendly curve that has all the properties needed to do pairing-based computations without any computer assistance, which often helps to understand delicate details of your system.

Readers who are primarily interested in building a simple pen-and-paper zk-SNARK and in filling in their knowledge gaps as they appear throughout the process might want to start with our 3-factorization example []. In [] we introduce the 3-factorization problem as a statement in a formal language. If this is too abstract, the reader might start in [] where we describe the 3-factorization problem as an algebraic circuit. In [] we introduce various levels of zero knowledge, and we show how to compute solutions in []. We then transform the circuit into an associated Rank-1 Constraint System in [], and transform that constraint system into a Quadratic Arithmetic Program in []. In [], we show how solutions are transformed into polynomial divisibility problems. In [], we use the result of those examples to derive a Groth16 zk-SNARK for the 3-factorization problem. In [], we compute the prover and the verifier key. In [] we compute a zk-SNARK and in [] we verify that zk-SNARK. In [] we show how to simulate proofs.

Chapter 2

Preliminaries

2.1 Runtime complexity

Before we deal with the mathematics behind zero-knowledge proof systems, we must first say a few words about the runtime of an algorithm or the time complexity of a mathematical problem. This is particularly important for us when we analyze the security of various SNARK systems.

should this whole section be moved to section 4.1.6.1?

Roughly speaking, the runtime complexity of an algorithm describes the amount of elementary computation steps that this algorithm requires in order to solve a problem, depending on the size of the input data.

Of course, the exact amount of arithmetic operations required depends on many factors, such as the implementation, the operating system used, the CPU and so on. However, this level of accuracy is seldom required for our purposes, so the "magnitude" of the computational effort is used instead.

In other words, instead of specifying the individual calculation steps, we look for an upper limit which approximates the runtime as soon as the input quantity becomes very large.

To talk about the security of cryptographic systems, we distinguish the following levels of complexity:

- $\mathcal{O}(n)$ means that the running time of the algorithm to be considered is linearly dependent on the size of the input set n
- $\mathcal{O}(n^k)$ means that the running time is polynomial
- $\mathcal{O}(2^n)$ stands for an exponential running time.

An algorithm which has a running time that is greater than a polynomial is often simply referred to as **slow**.

A generalization of the runtime complexity of an algorithm is the so-called **time complexity of a mathematical problem**, which is defined as the runtime of the fastest possible algorithm that can still solve this problem.

Since the time complexity of a mathematical problem is concerned with the runtime analysis of all possible (and thus possibly still undiscovered) algorithms, this is often very difficult to determine.

For our topic, the time complexity of the so-called discrete logarithm problem is particularly important (see section 4.1.6.1). This is a problem for which we only know slow algorithms on classical computers at the moment, but for which we cannot rule out that faster algorithms also exist.

2.2 Software Used in This Book

It order to provide an interactive learning experience, and to allow getting hands-on with the concepts described in this book, we give examples for how to program them in the SageMath programming language. Sage is based on the learning-friendly programming language Python, extended and optimized for computations involving algebraic objects. Therefore, we recommend installing Sage before diving into the following chapters.

The installation steps for various system configurations are described on the Sage website. Note that we use Sage version 9, so if you are using Linux and your package manager only contains version 8, you may need to choose a different installation path, such as using prebuilt binaries.

If you are not familiar with SageMath, we recommend you consult the Sage Tutorial.

Chapter 4

Algebra

In the previous chapter, we gave an introduction to the basic computational tools needed for a pen-and-paper approach to SNARKs. In this chapter, we provide a more abstract clarification of relevant mathematical terminology such as **groups**, **rings** and **fields**.

Scientific literature on cryptography frequently contains such terms, and it is necessary to get at least some understanding of these terms to be able to follow the literature.

4.1 Commutative Groups

Commutative groups are abstractions that capture the essence of mathematical phenomena, like addition and subtraction, or multiplication and division.

To understand commutative groups, let us think back to when we learned about the addition and subtraction of integers in school. We have learned that, whenever we add two integers, the result is guaranteed to be an integer as well. We have also learned that adding zero to any integer means that "nothing happens" since the result of the addition is the same integer we started with. Furthermore, we have learned that the order in which we add two (or more) integers does not matter, that brackets have no influence on the result of addition, and that, for every integer, there is always another integer (the negative) such that we get zero when we add them together.

These conditions are the defining properties of a commutative group, and mathematicians have realized that the exact same set of rules can be found in very different mathematical structures. It therefore makes sense to give an abstract, formal definition of what a group should be, detached from any concrete examples such as integers. This lets us handle entities of very different mathematical origins in a flexible way, while retaining essential structural aspects of many objects in abstract algebra and beyond.

Distilling these rules to the smallest independent list of properties and making them abstract, we arrive at the following definition of a commutative group:

revise the counter for definitions, current one too long

revise the counter for definitions, current one too long

Definition 4.1.0.1. A **commutative group** $(\mathbb{G},\cdot)$ consists of a set $\mathbb{G}$ and a **map** $\cdot : \mathbb{G}\times\mathbb{G}\to\mathbb{G}$. The map is called the **group law**, and it combines two elements of the set $\mathbb{G}$ into a third one such that the following properties hold:

- **Commutativity**: For all $g_1, g_2 \in \mathbb{G}$, the equation $g_1 \cdot g_2 = g_2 \cdot g_1$ holds.
- **Associativity**: For every $g_1, g_2, g_3 \in \mathbb{G}$ the equation $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$ holds.
- **Existence of a neutral element**: For every $g \in \mathbb{G}$, there is an $e \in \mathbb{G}$ such that $e \cdot g = g$.

- **Existence of an inverse**: For every $g \in \mathbb{G}$, there is a $g^{-1} \in \mathbb{G}$ such that $g \cdot g^{-1} = e$.

If $(\mathbb{G}, \cdot)$ is a group, and $\mathbb{G}' \subset \mathbb{G}$ is a subset of $\mathbb{G}$ such that the **restriction** of the group law $\cdot : \mathbb{G}' \times \mathbb{G}' \to \mathbb{G}'$ is a group law on $\mathbb{G}'$, then $(\mathbb{G}', \cdot)$ is called a **subgroup** of $(\mathbb{G}, \cdot)$.

Rephrasing the abstract definition in layman's terms, a group is something where we can do computations in a way that resembles the behavior of the addition of integers. Specifically, this means we can combine some element with another element into a new element in a way that is reversible and where the order of combining elements doesn't matter.

Notation and Symbols 3. Since we are exclusively concerned with commutative groups in this book, we often just call them groups, keeping the notation of commutativity implicit.[1]

If there is no risk of ambiguity (about what the group law of a group is), we frequently drop the symbol $\cdot$ and simply write $\mathbb{G}$ as notation for the group, keeping the group law implicit. In this case we also say that $\mathbb{G}$ is of group type, indicating that $\mathbb{G}$ is not simply a set but a set together with a group law.

Notation and Symbols 4 (**Additive notation**). For commutative groups $(\mathbb{G}, \cdot)$, we sometimes use the so-called **additive notation** $(\mathbb{G}, +)$, that is, we write $+$ instead of $\cdot$ for the group law, 0 for the neutral element and $-g := g^{-1}$ for the inverse of an element $g \in \mathbb{G}$.

As we will see in the following chapters, groups are heavily used in cryptography and in SNARKs.[2] But let us look at some more familiar examples fist.

Example 29 (Integer Addition and Subtraction). The set $(\mathbb{Z}, +)$ of integers with integer addition is the archetypical example of a commutative group, where the group law is traditionally written in additive notation (notation 4).

To compare integer addition against the abstract axioms of a commuative group, we first note that integer addition is **commutative and associative**, since $a+b = b+a$ as well as $(a+b)+c = a+(b+c)$ for all integers $a, b, c \in \mathbb{Z}$. The **neutral element** e is the number 0, since $a+0 = a$ for all integers $a \in \mathbb{Z}$. Furthermore, the **inverse** of a number is its negative counterpart, since $a+(-a) = 0$ for all $a \in \mathbb{Z}$. This implies that integers with addition are indeed a commutative group in the abstract sense.

To give an example of a subgroup of the group of integers, consider the set of even numbers, including 0.

$$\mathbb{Z}_{even} := \{\ldots, -4, -2, 0, 2, 4, \ldots\}$$

We can see that this set is a subgroup of $(\mathbb{Z}, +)$, since the sum of two even numbers is always an even number again, since the neutral element 0 is a member of $\mathbb{Z}_{even}$ and sice the negative of an even number is itself an even number.

Example 30 (The trivial group). The most basic example of a commutative group is the group with just one element $\{\bullet\}$ and the group law $\bullet \cdot \bullet = \bullet$. We call it the **trivial group**.

The trivial group is a subgroup of any group. To see that, let $(\mathbb{G}, \cdot)$ be a group with the neutral element $e \in \mathbb{G}$. Then $e \cdot e = e$ as well as $e^{-1} = e$ both hold. Consequently, the set $\{e\}$ is a subgroup of $\mathbb{G}$. In particular, $\{0\}$ is a subgroup of $(\mathbb{Z}, +)$, since $0+0 = 0$.

[1]Commutative groups are also called **Abelian groups**. A set $\mathbb{G}$ with a map $\cdot$ that satisfies all previously mentioned rules except for the commutativity law is called a **non-commutative group.**

[2]A more in-depth introduction to commutative groups can be found for example in chapter 1, section 1 of Lidl and Niederreiter [1986] or in chapter 1 of Fuchs [2015]. An introduction more tailored to the needs in cryptography can be found for example in chapter 3, section 8.1.3 of Katz and Lindell [2007].

Example 31. Consider addition in modulo 6 arithmetics $(\mathbb{Z}_6,+)$, as defined in in example 9. As we see, the remainder 0 is the neutral element in modulo 6 addition, and the inverse of a remainder r is given by $6-r$, because $r+(6-r)=6$. 6 is congruent to 0 since $6 \bmod 6 = 0$. Moreover, $r_1+r_2=r_2+r_1$ as well as $(r_1+r_2)+r_3=r_1+(r_2+r_3)$ are inherited from integer addition. We therefore see that $(\mathbb{Z}_6,+)$ is a group.

The previous example of a commutative group is a very important one for this book. Abstracting from this example and considering residue classes $(\mathbb{Z}_n,+)$ for arbitrary moduli n, it can be shown that $(\mathbb{Z}_n,+)$ is a commutative group with the neutral element 0 and the additive inverse $n-r$ for any element $r\in\mathbb{Z}_n$. We call such a group the **remainder class group** of modulus n.

Exercise 32. Consider example 14 again, and let $\mathbb{Z}_5^*$ be the set of all remainder classes from $\mathbb{Z}_5$ without the class 0. Then $\mathbb{Z}_5^*=\{1,2,3,4\}$. Show that $(\mathbb{Z}_5^*,\cdot)$ is a commutative group.

Exercise 33. Generalizing the previous exercise, consider the general modulus n, and let $\mathbb{Z}_n^*$ be the set of all remainder classes from $\mathbb{Z}_n$ without the class 0. Then $\mathbb{Z}_n^*=\{1,2,\ldots,n-1\}$. Provide a counter-example to show that $(\mathbb{Z}_n^*,\cdot)$ is not a group in general.

Find a condition such that $(\mathbb{Z}_n^*,\cdot)$ is a commutative group, compute the neutral element, give a closed form for the inverse of any element and prove the commutative group axioms.

4.1.1 Finite groups

As we have seen in the previous examples, groups can either contain infinitely many elements (such as integers) or finitely many elements (as for example the remainder class groups $(\mathbb{Z}_n,+)$). To capture this distinction, a group is called a **finite group** if the underlying set of elements is finite. In that case, the number of elements of that group is called its **order**.[3]

Notation and Symbols 5. Let $\mathbb{G}$ be a finite group. We write $ord(\mathbb{G})$ or $|\mathbb{G}|$ for the order of $\mathbb{G}$.

Example 32. Consider the remainder class groups $(\mathbb{Z}_6,+)$ from example 9, the group $(\mathbb{Z}_5,+)$ from example 14, and the group $(\mathbb{Z}_5^*,\cdot)$ from exercise 32. We can easily see that the order of $(\mathbb{Z}_6,+)$ is 6, the order of $(\mathbb{Z}_5,+)$ is 5 and the order of $(\mathbb{Z}_5^*,\cdot)$ is 4.

Exercise 34. Let $n\in\mathbb{N}$ with $n\geq 2$ be some modulus. What is the order of the remainder class group $(\mathbb{Z}_n,+)$?

4.1.2 Generators

Listing the set of elements of a group can be complicated, and it is not always obvious how to actually compute elements of a given group. From a practical point of view, it is therefore desirable to have groups with a **generator set**. This is a small subset of elements from which all other elements can be generated by applying the group law repeatedly to only the elements of the generator set and/or their inverses.

Of course, every group $\mathbb{G}$ has a trivial set of generators, when we just consider every element of the group to be in the generator set. The more interesting question is to find smallest possible generator set for a given group. Of particular interest in this regard are groups that have a generator set that contains a single element only. In this case, there exists a (not necessarily unique) element $g\in\mathbb{G}$ such that every other element from $\mathbb{G}$ can be computed by the repeated combination of g and its inverse g^{-1} only.

[3]An introduction to finite groups is given in chapter 1 of Fuchs [2015]. An introduction from the perspective of cryptography can be found in chapter 3, section 8.3.1 of Katz and Lindell [2007].

Definition 4.1.2.1 (**Cyclic groups**). Groups with single, not necessarily unique, generators are called **cyclic groups** and any element $g \in \mathbb{G}$ that is able to generate $\mathbb{G}$ is called a **generator**.

Example 33. The most basic example of a cyclic group is the group of integers with integer addition $(\mathbb{Z}, +)$. In this case, the number 1 is a generator of $\mathbb{Z}$, since every integer can be obtained by repeatedly adding either 1 or its inverse -1 to itself. For example, -4 is generated by 1, since $-4 = -1 + (-1) + (-1) + (-1)$. Another generator of $\mathbb{Z}$ is the number -1.

Example 34. Consider the group $(\mathbb{Z}_5^*, \cdot)$ from exercise 32. Since $2^1 = 2$, $2^2 = 4$, $2^3 = 3$ and $2^4 = 1$, the element 2 is a generator of $(\mathbb{Z}_5^*, \cdot)$. Moreover, since $3^1 = 3$, $3^2 = 4$, $3^3 = 2$ and $3^4 = 1$, the element 3 is another generator of $(\mathbb{Z}_5^*, \cdot)$. Cyclic groups can therefore have more than one generator. However since $4^1 = 4$, $4^2 = 1$, $4^3 = 4$ and in general $4^k = 4$ for k odd and $4^k = 1$ for k even the element 4 is not a generator of $(\mathbb{Z}_5^*, \cdot)$. It follows that in general not every element of a finite cyclic group is a generator.

Example 35. Consider a modulus n and the remainder class groups $(\mathbb{Z}_n, +)$ from exercise 34. These groups are cyclic, with generator 1, since every other element of that group can be constructed by repeatedly adding the remainder class 1 to itself. Since $\mathbb{Z}_n$ is also finite, we know that $(\mathbb{Z}_n, +)$ is a finite cyclic group of order n.

Exercise 35. Consider the group $(\mathbb{Z}_6, +)$ of modular 6 addition from example 9. Show that $5 \in \mathbb{Z}_6$ is a generator, and then show that $2 \in \mathbb{Z}_6$ is not a generator.

Exercise 36. Let $p \in \mathbb{P}$ be prime number and $(\mathbb{Z}_p^*, \cdot)$ the finite group from exercise 33. Show that $(\mathbb{Z}_p^*, \cdot)$ is cyclic.

4.1.3 The exponential map

Observe that, when $\mathbb{G}$ is a cyclic group of order n and $g \in \mathbb{G}$ is a generator of $\mathbb{G}$, then there exists a so-called **exponential map**, which maps the additive group law of the remainder class group $(\mathbb{Z}_n, +)$ onto the group law of $\mathbb{G}$ in a one-to-one correspondence. The exponential map can be formalized as in (4.1) below (where g^x means "multiply g by itself x times" and $g^0 = e_{\mathbb{G}}$).

$$g^{(\cdot)} : \mathbb{Z}_n \to \mathbb{G}\, x \mapsto g^x \tag{4.1}$$

To see how the exponential map works, first observe that, since $g^0 := e_{\mathbb{G}}$ by definition, the neutral element of $\mathbb{Z}_n$ is mapped to the neutral element of $\mathbb{G}$. Furthermore, since $g^{x+y} = g^x \cdot g^y$, the map respects the group law.

Notation and Symbols 6 (**Scalar multiplication**). If a group $(\mathbb{G}, +)$ is written in additive notation (notation 4), then the exponential map is often called **scalar multiplication**, and written as follows:

$$(\cdot) \cdot g : \mathbb{Z}_n \to \mathbb{G}\, ;\, x \mapsto x \cdot g \tag{4.2}$$

In this notation, the symbol $x \cdot g$ is defined as "add the generator g to itself x times" and the symbol $0 \cdot g$ is defined to be the neutral element in $\mathbb{G}$.

Cryptographic applications often utilize finite cyclic groups of a very large order n, which means that computing the exponential map by repeated multiplication of the generator with itself is infeasible for very large remainder classes.[4] Algorithm 5, called **square and multiply**, solves this problem by computing the exponential map in approximately k steps, where k is the

unify title of Alg with text

[4]However, methods for fast exponentiations have been known for a long time. A detailed introduction can be found, for example, in chapter 1, section 7 of Mignotte [1992].

bit length of the exponent (3.4):SB: I think moving the explanation of bit length here would work better

move 3.4 here

Algorithm 5 Cyclic Group Exponentiation

```
Require: $g$ group generator of order $n$
Require: $x \in \mathbb{Z}_n$
  procedure EXPONENTIATION($g, x$)
      Let $(b_0, \dots, b_k)$ be a binary representation of $x$        ▷ see example XXX
      $h \leftarrow g$
      $y \leftarrow e_{\mathbb{G}}$
      for $0 \leq j < k$ do
          if $b_j = 1$ then
              $y \leftarrow y \cdot h$        ▷ multiply
          end if
          $h \leftarrow h \cdot h$        ▷ square
      end for
      return $y$
  end procedure
Ensure: $y = g^x$
```

check algorithm floating

Because the exponential map respects the group law, it doesn't matter if we do our computation in $\mathbb{Z}_n$ before we write the result into the exponent of g or afterwards: the result will be the same in both cases. The latter mehtod is usually referred to as doing computations "in the exponent". In cryptography in general, and in SNARK development in particular, we often perform computations "in the exponent" of a generator.

Example 36. Consider the multiplicative group $(\mathbb{Z}_5^*, \cdot)$ from exercise 32. We know from 36 that $\mathbb{Z}_5^*$ is a cyclic group of order 4, and that the element $3 \in \mathbb{Z}_5^*$ is a generator. This means that we also know that the following map respects the group law of addition in $\mathbb{Z}_4$ and the group law of multiplication in $\mathbb{Z}_5^*$:

$$3^{(\cdot)} : \mathbb{Z}_4 \to \mathbb{Z}_5^* \; ; \; x \mapsto 3^x$$

To do an example computation "in the exponent" of 3 , let's perform the calculation $1+3+2$ in the exponent of the generator 3:

$$\begin{aligned} 3^{1+3+2} &= 3^2 && (4.3) \\ &= 4 && (4.4) \end{aligned}$$

In (4.3) above, we first performed the computation $1+3+2 = 1$ in the remainder class group $(\mathbb{Z}_4, +)$ and then applied the exponential map $3^{(\cdot)}$ to the result in (4.4).

should be 2?

However, since the exponential map (4.1) respects the group law, we also could map each summand into $(\mathbb{Z}_5^*, \cdot)$ first and then apply the group law of $(\mathbb{Z}_5^*, \cdot)$. The result is guranteed to be the same:

$$\begin{aligned} 3^1 \cdot 3^3 \cdot 3^2 &= 3 \cdot 2 \cdot 4 \\ &= 1 \cdot 4 \\ &= 4 \end{aligned}$$

Since the exponential map (4.1) is a one-to-one correspondence that respects the group law, it can be shown that this map has an inverse with respect to the base g, called the **base g discrete logarithm map**:

$$log_g(\cdot) : \mathbb{G} \to \mathbb{Z}_n \; x \mapsto log_g(x) \tag{4.5}$$

Discrete logarithms are highly important in cryptography, because there are finite cyclic groups where the exponential map and its inverse, the discrete logarithm map, are believed to be one-way functions, which informally means that computing the exponential map is fast, while computing the logarithm map is slow (We will look into a more precise definition in 4.1.6).

Example 37. Consider the exponential map $3^{(\cdot)}$ from example 36. Its inverse is the discrete logarithm to the base 3, given by the map below:

$$log_3(\cdot) : \mathbb{Z}_5^* \to \mathbb{Z}_4 \; x \mapsto log_3(x)$$

In contrast to the exponential map $3^{(\cdot)}$, we have no way to actually compute this map, other than by trying all elements of the group until we find the correct one. For example, in order to compute $log_3(4)$, we have to find some $x \in \mathbb{Z}_4$ such that $3^x = 4$, and all we can do is repeatedly insert elements x into the exponent of 3 until the result is 4. To do this, let's write down all the images of $3^{(\cdot)}$:

images

$$3^0 = 1, \quad 3^1 = 3, \quad 3^2 = 4, \quad 3^3 = 2$$

Since the discrete logarithm $log_3(\cdot)$ is defined as the inverse to this function, we can use those images to compute the discrete logarithm:

$$log_3(1) = 0, \quad log_3(2) = 3, \quad log_3(3) = 1, \quad log_3(4) = 2$$

Note that this computation was only possible because we were able to write down all images of the exponential map. However, in real world applications the groups in consideration are too large to write down the images of the exponential map.

Exercise 37 (Efficient Scalar Multiplication). Let $(\mathbb{G},+)$ be a finite cyclic group of order n. Consider algorithm 5 and define its analog for groups in additive notation.

4.1.4 Factor Groups

As we know from the fundamental theorem of arithmetic (3.7), every natural number n is a product of factors, the most basic of which are prime numbers. This parallels subgroups of finite cyclic groups in an interesting way.

Definition 4.1.4.1 (**The fundamental theorem of finite cyclic groups**). If $\mathbb{G}$ is a finite cyclic group of order n, then every subgroup $\mathbb{G}'$ of $\mathbb{G}$ is finite and cyclic, and the order $\mathbb{G}'$ is a factor of n. Moreover for each factor k of n, $\mathbb{G}$ has exactly one subgroup of order k. This is known as the **fundamental theorem of finite cyclic groups**.

Notation and Symbols 7. If $\mathbb{G}$ is a finite cyclic group of order n and k is a factor of n, then we write $\mathbb{G}[k]$ for the unique finite cyclic group which is the order k subgroup of $\mathbb{G}$, and call it a **factor group** of $\mathbb{G}$.

One particularly interesting situation occurs if the order of a given finite cyclic group is a prime number. As we know from the fundamental theorem of arithmetics (3.7), prime numbers have only two factors: the number 1 and the prime number itself. It then follows from the fundamental theorem of finite cyclic groups (definition 4.1.4.1) that those groups have no subgroups other than the trivial group (example 30) and the group itself.

Cryptographic protocols often assume the existence of finite cyclic groups of prime order. However some real-world implementations of those protocols are not defined on prime order groups, but on groups where the order consist of a (usually large) prime number that has small cofactors (see notation 1). In this case, a method called **cofactor clearing** has to be applied to ensure that the computations are not done in the group itself but in its (large) prime order subgroup.

To understand cofactor clearing in detail, let $\mathbb{G}$ be a finite cyclic group of order n, and let k be a factor of n with associated factor group $\mathbb{G}[k]$. We can project any element $g \in \mathbb{G}[k]$ onto the neutral element e of $\mathbb{G}$ by multiplying g k-times with itself:

$$g^k = e \tag{4.6}$$

Consequently, if $c := n \text{ div } k$ is the cofactor of k in n, then any element from the full group $g \in \mathbb{G}$ can be projected into the factor group $\mathbb{G}[k]$ by multiplying g c-times with itself. This defines the following map, which is often called **cofactor clearing** in cryptographic literature:

$$(\cdot)^c : \mathbb{G} \to \mathbb{G}[k] \;:\; g \mapsto g^c \tag{4.7}$$

Example 38. Consider the finite cyclic group $(\mathbb{Z}_5^*, \cdot)$ from example 34. Since the order of $\mathbb{Z}_5^*$ is 4, and 4 has the factors 1, 2 and 4, it follows from the fundamental theorem of finite cyclic groups (definition 4.1.4.1) that $\mathbb{Z}_5^*$ has 3 unique subgroups. In fact, the unique subgroup $\mathbb{Z}_5^*[1]$ of order 1 is given by the trivial group $\{1\}$ that contains only the multiplicative neutral element 1. The unique subgroup $\mathbb{Z}_5^*[4]$ of order 4 is $\mathbb{Z}_5^*$ itself, since, by definition, every group is trivially a subgroup of itself. The unique subgroup $\mathbb{Z}_5^*[2]$ of order 2 is more interesting, and is given by the set $\mathbb{Z}_5^*[2] = \{1,4\}$.

Since $\mathbb{Z}_5^*$ is not a prime order group, and, since the only prime factor of 4 is 2, the "large" prime order subgroup of $\mathbb{Z}_5^*$ is $\mathbb{Z}_5^*[2]$. Moreover, since the cofactor of 2 in 4 is also 2, we get the cofactor clearing map $(\cdot)^2 : \mathbb{Z}_5^* \to \mathbb{Z}_5^*[2]$. As expected, when we apply this map to all elements of $\mathbb{Z}_5^*$, we see that it maps onto the elements of $\mathbb{Z}_5^*[2]$ only:

$$1^2 = 1 \quad 2^2 = 4 \quad 3^2 = 4 \quad 4^2 = 1 \tag{4.8}$$

We can therefore use this map to "clear the cofactor" of any element from $\mathbb{Z}_5^*$, which means that the element is projected onto the "large" prime order subgroup $\mathbb{Z}_5^*[2]$.

Exercise 38. Consider the previous example 38, and show that $\mathbb{Z}_5^*[2]$ is a commutative group.

Exercise 39. Consider the finite cyclic group $(\mathbb{Z}_6, +)$ of modular 6 addition from example 35. Describe all subgroups of $(\mathbb{Z}_6, +)$. Identify the large prime order subgroup of $\mathbb{Z}_6$, define its cofactor clearing map and apply that map to all elements of $\mathbb{Z}_6$.

Exercise 40. Let $(\mathbb{Z}_p^*, \cdot)$ be the cyclic group from exercise 36. Show that, for $p \geq 5$, not every element $x \in \mathbb{F}_p^*$ is a generator of $\mathbb{F}_p^*$.

4.1.5 Pairings

Of particular importance for the development of SNARKs are so-called **pairing maps** on commutative groups, defined below.

Definition 4.1.5.1 (**Pairing map**). Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ be three commutative groups. Then a **pairing map** is a function

$$e(\cdot,\cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3 \tag{4.9}$$

This function takes pairs (g_1, g_2) of elements from $\mathbb{G}_1$ and $\mathbb{G}_2$, and maps them to elements from $\mathbb{G}_3$ such that the **bilinearity** property holds, which means that for all $g_1, g_1' \in \mathbb{G}_1$ and $g_2, g_2' \in \mathbb{G}_2$ the following two identities are satisfied:

$$e(g_1 \cdot g_1', g_2) = e(g_1, g_2) \cdot e(g_1', g_2) \quad \text{and} \quad e(g_1, g_2 \cdot g_2') = e(g_1, g_2) \cdot e(g_1, g_2') \tag{4.10}$$

Informally speaking, bilinearity means that it doesn't matter if we first execute the group law on one side and then apply the bilinear map, or if we first apply the bilinear map and then apply the group law in $\mathbb{G}_3$.

A pairing map is called **non-degenerate** if, whenever the result of the pairing is the neutral element in $\mathbb{G}_3$, one of the input values is the neutral element of $\mathbb{G}_1$ or $\mathbb{G}_2$. To be more precise, $e(g_1, g_2) = e_{\mathbb{G}_3}$ implies $g_1 = e_{\mathbb{G}_1}$ or $g_2 = e_{\mathbb{G}_2}$.

Example 39. One of the most basic examples of a non-degenerate pairing involves the groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ all being groups of integers with addition $(\mathbb{Z}, +)$. In this case, the following map defines a non-degenerate pairing:

$$e(\cdot, \cdot) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \; (a, b) \mapsto a \cdot b \tag{4.11}$$

Note that bilinearity follows from the distributive law of integers, meaning that, for $a, b, c \in \mathbb{Z}$, the equation $e(a+b, c) = (a+b) \cdot c = a \cdot c + b \cdot c = e(a, c) + e(b, c)$ holds (and the same reasoning is true for the second argument b).

b

To see that $e(\cdot, \cdot)$ is non-degenerate, assume that $e(a, b) = 0$. Then $a \cdot b = 0$ implies that a or b must be zero.

Exercise 41 (Arithmetic laws for pairing maps). Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ be finite cyclic groups of the same order n, and let $e(\cdot, \cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$ be a pairing map. Show that, for given $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$ and all $a, b \in \mathbb{Z}_n$, the following identity holds:

$$e(g_1^a, g_2^b) = e(g_1, g_2)^{a \cdot b} \tag{4.12}$$

Exercise 42. Consider the remainder class groups $(\mathbb{Z}_n, +)$ from example 34 for some modulus n. Show that the following map is a pairing map.

$$e(\cdot, \cdot) : \mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n \; (a, b) \mapsto a \cdot b \tag{4.13}$$

Why is the pairing not non-degenerate in general, and what condition must be imposed on n such that the pairing will be non-degenerate?

4.1.6 Cryptographic Groups

In this section, we look at classes of groups that are believed to satisfy certain **computational hardness assumptions**, meaning that it is not feasible to compute them in polynomial time.[5]

Example 40. To give an example for a well-known computational hardness assumption, consider the problem of factorization, i.e. computing the prime factors of a composite integer (see example 1). If the prime factors are very large, this is infeasible to do, and is expected to remain infeasible. We assume the problem is **computationally hard** or **infeasible**.

[5] A more detailed introduction to computational hardness assumptions and their applications in cryptography can be found in chapter 3, section 8 in Katz and Lindell [2007].

Note that, in example 40, we say that the problem is infeasible to solve **if the prime factors are large enough**. Naturally, this is made more precise in the cryptographic standard model, where we have a security parameter, and we say that "there exists a security parameter such that it is not feasible to compute a solution to the problem". In the following examples, the security parameter roughly correlates with the order of the group in consideration. In this book, we do not include the security parameter in our definitions, since we only aim to provide an intuitive understanding of the cryptographic assumptions, not teach the ability to perform rigorous analysis.

Furthermore, understand that these are **assumptions**. Academics have been looking for efficient prime factorization algorithms for a long time, and they have been getting better and better while computers have become faster and faster – but there always was a higher security parameter for which the problem still was infeasible.

In what follows, we describe a few problems arising in the context of groups in cryptography that are assumed to be infeasible. We will refer to them throughout the book.

4.1.6.1 The Discrete Logarithm Problem

The so-called **Discrete Logarithm Problem (DLP)**, also called the **Discrete Logarithm Assumption**, is one of the most fundamental assumptions in cryptography.

Definition 4.1.6.1. Let $\mathbb{G}$ be a finite cyclic group of order r and let g be a generator of $\mathbb{G}$. We know from (4.1) that there is an exponential map $g^{(\cdot)} : \mathbb{Z}_r \to \mathbb{G}\ ;\ x \mapsto g^x$ that maps the residue classes from modulo r arithmetic onto the group in a $1:1$ correspondence. The **Discrete Logarithm Problem** is the task of finding an inverse to this map, that is, to find a solution $x \in \mathbb{Z}_r$ to the following equation for some given $h, g \in \mathbb{G}$:

$$h = g^x \tag{4.14}$$

There are groups in which the DLP is assumed to be infeasible to solve, and there are groups in which it isn't. We call the former group **DL-secure** groups.

Rephrasing the previous definition, it is believed that, in DL-secure groups, there is a number n such that it is infeasible to compute some number x that solves the equation $h = g^x$ for a given h and g, assuming that the order of the group n is large enough. The number n here corresponds to the security parameter discussed above.

Example 41 (Public key cryptography). One the most basic examples of an application for DL-secure groups is in public key cryptography, where the parties publicly agree on some pair $(\mathbb{G}, g)$ such that $\mathbb{G}$ is a finite cyclic group of appropriate order n, believed to be a DL-secure group, and g is a generator of $\mathbb{G}$.

In this setting, a secret key is some number $sk \in \mathbb{Z}_r$ and the associated public key pk is the group element $pk = g^{sk}$. Since discrete logarithms are assumed to be hard, it is infeasible for an attacker to compute the secret key from the public key, as this would involve finding solutions x to the following equation (which is believed to be infeasible):

$$pk = g^x \tag{4.15}$$

As example 41 shows, identifying DL-secure groups is an important practical problem. Unfortunately, it is easy to see that it does not make sense to assume the hardness of the Discrete Logarithm Problem in all finite cyclic groups: counterexamples are common and easy to construct.

mention a few examples

4.1.6.2 The decisional Diffie–Hellman assumption

Definition 4.1.6.2. Let $\mathbb{G}$ be a finite cyclic group of order n and let g be a generator of $\mathbb{G}$. The decisional Diffie–Hellman (DDH) problem is to distinguish (g^a, g^b, g^{ab}) from the triple (g^a, g^b, g^c) for uniformly random values $a, b, c \in \mathbb{Z}_r$.

If we assume the DDH problem is infeasible to solve in $\mathbb{G}$, we call $\mathbb{G}$ a **DDH-secure** group.

DDH-security is a stronger assumption than DL-security (4.1.6.1), in the sense that if the DDH problem is infeasible, so is the DLP, but not necessarily the other way around.

To see why this is the case, assume that the discrete logarithm assumption does not hold. In that case, given a generator g and a group element h, it is easy to compute some element $x \in \mathbb{Z}_p$ with $h = g^x$. Then the decisional Diffie–Hellman assumption cannot hold, since given some triple (g^a, g^b, z), one could efficiently decide whether $z = g^{ab}$ is true by first computing the discrete logarithm b of g^b, then computing $g^{ab} = (g^a)^b$ and deciding whether or not $z = g^{ab}$.

On the other hand, the following example shows that there are groups where the discrete logarithm assumption holds but the Decisional Diffie–Hellman Assumption does not.

Example 42 (Efficiently computable bilinear pairings). Let $\mathbb{G}$ be a DL-secure, finite, cyclic group of order r with generator g, and $\mathbb{G}_T$ another group such that there is an efficiently computable pairing map $e(\cdot,\cdot) : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ that is bilinear and non degenerate (4.9).

In a setting like this, it is easy to show that solving DDH cannot be infeasible, since, given some triple (g^a, g^b, z), it is possible to efficiently check whether $z = g^{ab}$ by making use of the following pairing:

$$e(g^a, g^b) \stackrel{?}{=} e(g, z) \tag{4.16}$$

Since the bilinearity properties of $e(\cdot,\cdot)$ imply $e(g^a, g^b) = e(g, g)^{ab} = e(g, g^{ab})$, and $e(g, y) = e(g, y')$ implies $y = y'$ due to the non-degenerate property, the equality means $z = g^{ab}$.

It follows that the DDH assumption is indeed stronger than the discrete log assumption, and groups with efficient pairings cannot be DDH-secure groups.

4.1.6.3 The Computational Diffie–Hellman Assumption

Definition 4.1.6.3. Let $\mathbb{G}$ be a finite cyclic group of order n and let g be a generator of $\mathbb{G}$. The **computational Diffie–Hellman assumption** stipulates that, given randomly and independently chosen elements $a, b \in \mathbb{Z}_r$, it is not possible to compute g^{ab} if only g, g^a and g^b (but not a and b) are known. If this is the case for $\mathbb{G}$, we call $\mathbb{G}$ a **CDH-secure** group.

In general, we don't know if CDH-security is a stronger assumption than DL-security, or if both assumptions are equivalent. We know that DL-security is necessary for CDH-security, but the other direction is currently not well understood. In particular, there are no known DL-secure groups that are not also CDH-secure.

To see why the discrete logarithm assumption is necessary, assume that it does not hold. Then, given a generator g and a group element h, it is easy to compute some element $x \in \mathbb{Z}_p$ with $h = g^x$. In that case, the computational Diffie–Hellman assumption cannot hold, since, given g, g^a and g^b, it is possible to efficiently compute b, meaning that $g^{ab} = (g^a)^b$ can be computed from this data.

The computational Diffie–Hellman assumption is a weaker assumption than the Decisional Diffie–Hellman Assumption. This means that there are groups where CDH holds and DDH does not hold, while there cannot be groups in which DDH holds but CDH does not hold. To see that, assume that it is efficiently possible to compute g^{ab} from g, g^a and g^b. Then, given (g^a, g^b, z) it is easy to decide whether $z = g^{ab}$ holds or not.

Several variations and special cases of CDH exist. For example, the **square Computational Diffie–Hellman Assumption** assumes that, given g and g^x, it is computationally hard to compute g^{x^2}. The **inverse Computational Diffie–Hellman Assumption** assumes that, given g and g^x, it is computationally hard to compute $g^{x^{-1}}$.

4.1.7 Hashing to Groups

4.1.7.1 Hash functions

Generally speaking, a hash function is any function that can be used to map data of arbitrary size to fixed-size values. Since binary strings of arbitrary length are a way to represent data in general, we can understand a **hash function** as the following map where $\{0,1\}^*$ represents the set of all binary strings of arbitrary but finite length and $\{0,1\}^k$ represents the set of all binary strings that have a length of exactly k bits:

$$H : \{0,1\}^* \to \{0,1\}^k \tag{4.17}$$

The **images** of H, that is, the values returned by the hash function H, are called **hash values**, **digests**, or simply **hashes**.

Notation and Symbols 8. In what follows, we call an element $b \in \{0,1\}$ a **bit**. If $s \in \{0,1\}^*$ is a binary string, we write $|s| = k$ for its **length**, that is, for the number of bits in s. We write $<>$ for the empty binary string, and $s =< b_1, b_2, \dots, b_k >$ for a binary string of length k.[6]

check footnote

If two binary strings $s =< b_1, b_2, \dots, b_k >$ and $s' =< b'_1, b'_2, \dots, b'_l >$ are given, then we write $s||s'$ for the **concatenation** that is the string $s||s' =< b_1, b_2, \dots, b_k, b'_1, b'_2, \dots, b'_l >$.

If H is a hash function that maps binary strings of arbitrary length onto binary strings of length k, and $s \in \{0,1\}^*$ is a binary string, we write $H(s)_j$ for the bit at position j in the image $H(s)$.

Example 43 (k-truncation hash). One of the most basic hash functions $H_k : \{0,1\}^* \to \{0,1\}^k$ is given by simply truncating every binary string s of size $|s| > k$ to a string of size k and by filling any string s' of size $|s'| < k$ with zeros. To make this hash function deterministic, we define that both truncation and filling should happen "on the left".

For example, if the parameter k is given by $k = 3$, $s_1 =< 0,0,0,0,1,0,1,0,1,1,1,0 >$ and $s_2 = 1$, then $H_3(x_1) =< 1,1,0 >$ and $H_3(x_2) =< 0,0,1 >$.

should be $H_3(s_1)$ and $H_3(s_2)$?

A desirable property of a hash function is **uniformity**, which means that it should map input values as evenly as possible over its output range. In mathematical terms, every string of length k from $\{0,1\}^k$ should be generated with roughly the same probability.

Of particular interest are so-called **cryptographic** hash functions, which are hash functions that are also **one-way functions**, which essentially means that, given a string y from $\{0,1\}^k$ it is infeasible to find a string $x \in \{0,1\}^*$ such that $H(x) = y$ holds. This property is usually called **preimage-resistance**.

Moreover, if a string $x_1 \in \{0,1\}^*$ is given, then it should be infeasible to find another string $x_2 \in \{0,1\}^*$ with $x_1 \neq x_2$ and $H(x_1) = H_(x_2)$

In addition, it should be infeasible to find two strings $x_1, x_2 \in \{0,1\}^*$ such that $H(x_1) = H(x_2)$, which is called **collision resistance**. It is important to note, though, that collisions always exist, since a function $H : \{0,1\}^* \to \{0,1\}^k$ inevitably maps infinitely many values onto

[6] The difference between the notations $b \in \{0,1\}$ and $s \in \{0,1\}^*$ is the following: $b \in \{0,1\}$ means that b is equal to either 0 or 1, whereas s is a string composed of an arbitrary number of 0s and 1s (and s can also be an empty string).

the same hash. In fact, for any hash function with digests of length k, finding a preimage to a given digest can always be done using a brute force search in 2^k evaluation steps. It should just be practically impossible to compute those values, and statistically very unlikely to generate two of them by chance.

A third property of a cryptographic hash function is that small changes in the input string, like changing a single bit, should generate hash values that look completely different from each other. This is called **diffusion** or the avalanche effect.

Because cryptographic hash functions map tiny changes in input values onto large changes in the output, implementation errors that change the outcome are usually easy to spot by comparing them to expected output values. The definitions of cryptographic hash functions are therefore usually accompanied by some test vectors of common inputs and expected digests. Since the empty string $<>$ is the only string of length 0, a common test vector is the expected digest of the empty string.

Example 44 (k-truncation hash). Consider the k-truncation hash from example 43. Since the empty string has length 0, it follows that the digest of the empty string is the string of length k that only contains 0s:

$$H_k(<>) =< 0,0,\ldots,0,0 > \tag{4.18}$$

It is pretty obvious from the definition of H_k that this simple hash function is not a cryptographic hash function. In particular, every digest is its own preimage, since $H_k(y) = y$ for every string of size exactly k. Finding preimages is therefore easy, so the property of preimage resistance does not hold.

In addition, it is easy to construct collisions, as all strings s of size $|s| > k$ that share the same k-bits “on the right” are mapped to the same hash value. This means that this function is not collision resistant, either.

Finally, this hash function does not have a lot of diffusion, as changing bits that are not part of the k right-most bits won’t change the digest at all.

Computing cryptographically secure hash functions in pen-and-paper style is possible but tedious. Fortunately, Sage can import the **`hashlib`** library, which is intended to provide a reliable and stable base for writing Python programs that require cryptographic functions. The following examples explain how to use `hashlib` in Sage.

Example 45. An example of a hash function that is generally believed to be a cryptographically secure hash function is the so-called **SHA256** hash, which, in our notation, is a function that maps binary strings of arbitrary length onto binary strings of length 256:

$$SHA256 : \{0,1\}^* \to \{0,1\}^{256} \tag{4.19}$$

To evaluate a proper implementation of the $SHA256$ hash function, the digest of the empty string is supposed to be the following:

$$SHA256(<>) = e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855 \tag{4.20}$$

For better human readability, it is common practice to represent the digest of a string not in its binary form, but in a hexadecimal representation. We can use Sage to compute $SHA256$ and freely transit between binary, hexadecimal and decimal representations. To do so, we import `hashlib`’s implementation of SHA256:

```
sage: import hashlib
```

```
sage: test = 'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934
   ca495991b7852b855'
sage: empty_string = ""
sage: binary_string = empty_string.encode()
sage: hasher = hashlib.sha256(binary_string)
sage: result = hasher.hexdigest()
sage: type(result) # Sage represents digests as strings
<class 'str'>
sage: d = ZZ('0x'+ result) # conversion to an integer
sage: d.str(16) == test # hash is equal to test vector
True
sage: d.str(16) # hexadecimal representation
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b8
   55
sage: d.str(2) # binary representation
1110001110110000110001000010100110001111110000011100000101
   0010011010111110111111010011001000100110010110111101110010
   0100100000100111101011100100000111100100011001001001101111001
   0011010011001010010010010101100110010001101101111000010100 1
   01011100001010101
sage: d.str(10) # decimal representation
1029873362495540970295352123225813227897999006481980349933793 9
   7001115665086549
```

4.1.7.2 Hashing to cyclic groups

As we have seen in the previous section, general hash functions map binary strings of arbitrary length onto binary strings of some fixed length. However, it is desirable in various cryptographic primitives to not simply hash to binary strings of fixed length, but to hash into algebraic structures like groups, while keeping (some of) the properties of the hash function, like preimage resistance or collision resistance.

Hash functions like this can be defined for various algebraic structures, but, in a sense, the most fundamental ones are hash functions that map into groups, because they can be easily extended to map into other structures like rings or fields.

To give a more precise definition, let $\mathbb{G}$ be a group and $\{0,1\}^*$ the set of all finite, binary strings, then a **hash-to-group** function is a deterministic map

$$H : \{0,1\}^* \to \mathbb{G} \tag{4.21}$$

As the following example shows, hashing to finite cyclic groups can be trivially achieved for the price of some undesirable properties of the hash function:

Example 46 (Naive cyclic group hash). Let $\mathbb{G}$ be a finite cyclic group of order n. If the task is to implement a hash-to-group function, one immediate approach can be based on the observation that binary strings of size k can be interpreted as integers $z \in \mathbb{Z}$ in the range $0 \leq z < 2^k$ using equation 3.4.

To be more precise, let $H : \{0,1\}^* \to \{0,1\}^k$ be a hash function for some parameter k, g a generator of $\mathbb{G}$, and $s \in \{0,1\}^*$ a binary string. Using equation 3.4 and notation 8, the following expression is a non-negative integer:

$$z_{H(s)} = H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_k \cdot 2^k \tag{4.22}$$

A hash-to-group function for the group $\mathbb{G}$ can then be defined as a composition of the exponential map $g^{(\cdot)}$ of g with the interpretation of $H(s)$ as an integer:

$$H_g : \{0,1\}^* \to \mathbb{G} : \; s \mapsto g^{z_{H(s)}} \tag{4.23}$$

Constructing a hash-to-group function like this is easy for cyclic groups, and it might be good enough in certain applications. It is, however, almost never adequate in cryptographic applications, as a discrete log relation might be constructible between some hash values $H_g(s)$ and $H_g(t)$, regardless of whether or not $\mathbb{G}$ is DL-secure (see section4.1.6.1).

a few examples?

To be more precise, a discrete log relation between the group elements $H_g(s)$ and $H_g(t)$ is any element $x \in \mathbb{Z}_n$ such that $H_g(s) = H_g(t)^x$. To see how such an x can be constructed, assume that $z_{H(s)}$ has a multiplicative inverse in $\mathbb{Z}_n$. In this case, the element $x = z_{H(t)} \cdot z_{H(s)}^{-1}$ from $\mathbb{Z}_n$ is a discrete log relation between $H_g(s)$ and $H_g(t)$:

$$\begin{aligned}
g^{z_{H(t)}} &= g^{z_{H(t)}} && \Leftrightarrow \\
g^{z_{H(t)}} &= g^{z_{H(t)} \cdot z_{H(s)} \cdot z_{H(s)}^{-1}} && \Leftrightarrow \\
g^{z_{H(t)}} &= g^{z_{H(s)} \cdot x} && \Leftrightarrow \\
H_g(t) &= (H_g(s))^x
\end{aligned}$$

Therefore, applications where discrete log relations between hash values are undesirable need different approaches. Many of these approaches start with a way to hash into the set $\mathbb{Z}_r$ of modular r arithmetics.

4.1.7.3 Pedersen Hashes

The so-called **Pedersen Hash Function** [Pedersen, 1992] provides a way to map fixed size tuples of elements from modular arithmetics onto elements of finite cyclic groups in such a way that discrete log relations (see example 46) between different images are avoidable. Compositions of a Pedersen Hash with a general hash function (4.17) then provide hash-to-group functions that map strings of arbitrary length onto group elements.

To be more precise, let j be an integer, $\mathbb{G}$ a finite cyclic group of order n, and $\{g_1, \ldots, g_j\} \subset \mathbb{G}$ a uniform and randomly generated set of generators of $\mathbb{G}$. Then **Pedersen's hash function** is defined as follows:

$$H_{\{g_1,\ldots,g_j\}} : (\mathbb{Z}_r)^j \to \mathbb{G} : \; (x_1, \ldots, x_j) \mapsto \Pi_{i=1}^k g_j^{x_j} \tag{4.24}$$

It can be shown that Pedersen's hash function is collision-resistant under the assumption that $\mathbb{G}$ is DL-secure (see section4.1.6.1). It is important to note though, that the following family of functions does not qualify as a pseudorandom function family.

pseudorandom function family

$$\{H_{\{g_1,\ldots,g_j\}} \mid g_1, \ldots, g_j \in \mathbb{G}\} \tag{4.25}$$

From an implementation perspective, it is important to derive the set of generators $\{g_1, \ldots, g_k\}$ in such a way that they are uniform and random as possible. In particular, any known discrete log relation between two generators, that is, any known $x \in \mathbb{Z}_n$ with $g_h = (g_i)^x$, must be avoided.

Example 47. To compute an actual Pedersen's hash, consider the cyclic group $\mathbb{Z}_5^*$ from example 34. We know from example 38 that the elements 2 and 3 are generators of $\mathbb{Z}_5^*$, and it follows that the following map is a Pedersen's hash function:

$$H_{\{2,3\}} : \mathbb{Z}_4 \times \mathbb{Z}_4 \to \mathbb{Z}_5^* \, ; \, (x,y) \mapsto 2^x \cdot 3^y \tag{4.26}$$

To see how this map can be calculated, we choose the input value $(1,3)$ from $\mathbb{Z}_4 \times \mathbb{Z}_4$. Then, using the multiplication table from (3.24), we calculate $H_{\{2,3\}}(1,3) = 2^1 \cdot 3^3 = 2 \cdot 2 = 4$.

To see how the composition of a hash function with $H_{\{2,3\}}$ defines a hash-to-group function, consider the $SHA256$ hash function from example 45. Given some binary string $s \in \{0,1\}^*$, we can insert the two least significant bits $SHA256(s)_0$ and $SHA256(s)_1$ from the image $SHA256(s)$ into $H_{\{2,3\}}$ to get an element in $\mathbb{F}_5^*$. This defines the following hash-to-group function

$$SHA256\_H_{\{2,3\}} : \{0,1\}^* \to \mathbb{Z}_5^* \, ; \, s \mapsto 2^{SHA256(s)_0} \cdot 3^{SHA256(s)_1}$$

To see how this hash function can be calculated, consider the empty string $<>$. Since we know from the Sage computation in example 45, that $SHA256(<>)_0 = 1$ and that $SHA256(<>)_1 = 0$, we get $SHA\_256H_{\{2,3\}}(<>) = 2^1 \cdot 3^0 = 2$.

Of course, computing $SHA256\_H_{\{2,3\}}$ in a pen-and-paper style is difficult. However, we can easily implement this function in Sage in the following way:

```
sage: import hashlib
sage: def SHA256_H(x):
....:     Z5 = Integers(5) # define the group type
....:     hasher = hashlib.sha256(x) # compute SHA256
....:     digest = hasher.hexdigest()
....:     z = ZZ(digest, 16) # cast into integer
....:     z_bin = z.digits(base=2, padto=256) # cast to 256
  bits
....:     return Z5(2)^z_bin[0] * Z5(3)^z_bin[1]
sage: SHA256_H(b"") # evaluate on empty string
2
sage: SHA256_H(b"SHA") # possible images are {1,2,3}
3
sage: SHA256_H(b"Math")
1
```

Exercise 43. Consider the multiplicative group $\mathbb{Z}_{13}^*$ of modular 13 arithmetic from example 33. Choose a set of 3 generators of $\mathbb{Z}_{13}^*$, define its associated Pedersen Hash Function, and compute the Pedersen Hash of $(3,7,11) \in \mathbb{Z}_{12}$.

Exercise 44. Consider the Pedersen Hash from exercise 43. Compose it with the $SHA256$ hash function from example 45 to define a hash-to-group function. Implement that function in Sage.

4.1.7.4 Pseudorandom Function Families in DDH-secure groups

As noted in 4.24, the family of Pederson's hash functions, parameterized by a set of generators $\{g_1,\ldots,g_j\}$ does not qualify as a family of pseudorandom functions, and should therefore not be instantiated as such. To see an example of a proper family of pseudorandom functions in groups where the decisional Diffie–Hellman assumption (see section4.1.6.2) is assumed to hold true, let $\mathbb{G}$ be a DDH-secure cyclic group of order n with generator g, and let $\{a_0,a_1,\ldots,a_k\} \subset \mathbb{Z}_n^*$ be a uniform randomly generated set of numbers invertible in modular n arithmetics. Then a family of pseudorandom functions, parameterized by the seed $\{a_0, a_1, \ldots, a_k\}$ is given as follows:

seed

$$F_{\{a_0,a_1,\ldots,a_k\}} : \{0,1\}^{k+1} \to \mathbb{G} : \; (b_0, \ldots, b_k) \mapsto g^{b_0 \cdot \Pi_{i=1}^{k} a_i^{b_i}} \tag{4.27}$$

Exercise 45. Consider the multiplicative group $\mathbb{Z}_{13}^*$ of modular 13 arithmetic from example 33 and the parameter $k = 3$. Choose a generator of $\mathbb{Z}_{13}^*$, a seed and instantiate a member of the family given in (4.27) for that seed. Evaluate that member on the binary string $< 1, 0, 1 >$.

instantiate

4.2 Commutative Rings

In the previous section, we have seen that integers are a commutative group with respect to integer addition. However, as we know, there are two arithmetic operations defined on integers: addition and multiplication. However, in contrast to addition, multiplication does not define a group structure, given that integers generally don't have multiplicative inverses. Configurations like these constitute so-called **commutative rings with unit**, and are defined as follows:

Definition 4.2.0.1 (Commutative ring with unit). A **commutative ring with unit** $(R, +, \cdot, 1)$ is a set R with two maps, $+ : R \times R \to R$ and $\cdot : R \times R \to R$, called **addition** and **multiplication**, and an element $1 \in R$, called the **unit**, such that the following conditions hold:

- $(R, +)$ is a commutative group where the neutral element is denoted with 0.
- **Commutativity of multiplication**: $r_1 \cdot r_2 = r_2 \cdot r_1$ for all $r_1, r_2 \in R$.
- **Multiplicative neutral unit** : $1 \cdot g = g$ for all $g \in R$.
- **Associativity**: For every $g_1, g_2, g_3 \in \mathbb{G}$, the equation $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$ holds.
- **Distributivity**: For all $g_1, g_2, g_3 \in R$, the distributive law $g_1 \cdot (g_2 + g_3) = g_1 \cdot g_2 + g_1 \cdot g_3$ holds.

what are the gs here? copy+paste error from 4.1.0.1?

If $(R, +, \cdot, 1)$ is a commutative ring with unit, and $R' \subset R$ is a subset of R such that the restriction of addition and multiplication to R' define a commutative ring with addition $+ : R' \times R' \to R'$, multiplication $\cdot : R' \times R' \to R'$ and unit 1 on R', then $(R', +, \cdot, 1)$ is called a **subring** of $(R, +, \cdot, 1)$.

Notation and Symbols 9. Since we are exclusively concerned with commutative rings in this book, we often just call them rings, keeping the notation of commutativity implicit. A set R with two maps, $+$ and $\cdot$, which satisfies all previously mentioned rules except for the commutativity law of multiplication, is called a non-commutative ring.

If there is no risk of ambiguity (about what the addition and multiplication maps of a ring are), we frequently drop the symbols $+$ and $\cdot$ and simply write R as notation for the ring, keeping those maps implicit. In this case we also say that R is of ring type, indicating that R is not simply a set but a set together with an addition and a multiplication map.[7]

Example 48 (The ring of integers). The set $\mathbb{Z}$ of integers with the usual addition and multiplication is the archetypical example of a commutative ring with unit 1.

```
sage: ZZ
Integer Ring
```

[7]Commutative rings are a large field of research in mathematics, and countless books on the topic exist. For our purposes, an introduction is given in chapter 1, section 2 of Lidl and Niederreiter [1986].

Example 49 (Underlying commutative group of a ring). Every commutative ring with unit $(R,+,\cdot,1)$ gives rise to a group, if we disregard multiplication.

The following example is somewhat unusual, but we encourage you to think through it because it helps to detach the mind from familiar styles of computation, and concentrate on the abstract algebraic explanation.

Example 50. Let $S := \{\bullet,\star,\odot,\otimes\}$ be a set that contains four elements, and let addition and multiplication on S be defined as follows:

$\cup$	$\bullet$	$\star$	$\odot$	$\otimes$
$\bullet$	$\bullet$	$\star$	$\odot$	$\otimes$
$\star$	$\star$	$\odot$	$\otimes$	$\bullet$
$\odot$	$\odot$	$\otimes$	$\bullet$	$\star$
$\otimes$	$\otimes$	$\bullet$	$\star$	$\odot$

$\circ$	$\bullet$	$\star$	$\odot$	$\otimes$
$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$
$\star$	$\bullet$	$\star$	$\odot$	$\otimes$
$\odot$	$\bullet$	$\odot$	$\bullet$	$\odot$
$\otimes$	$\bullet$	$\otimes$	$\odot$	$\star$

(4.28)

Then $(S,\cup,\circ,\star)$ is a ring with unit $\star$ and zero $\bullet$. It therefore makes sense to ask for solutions to equations like the following one:

$$\otimes\circ(x\cup\odot)=\star \tag{4.29}$$

The task here is to find $x\in S$ such that (4.29) holds. To see how such a “moonmath equation” can be solved, we have to keep in mind that rings behave mostly like normal numbers when it comes to bracketing and computation rules. The only differences are the symbols, and the actual way to add and multiply them. With this in mind, we solve the equation for x in the “usual way”: [8]

$$\begin{aligned}
\otimes\circ(x\cup\odot)&=\star && \text{# apply the distributive law}\\
\otimes\circ x\cup\otimes\circ\odot&=\star && \text{#} \otimes\circ\odot=\odot\\
\otimes\circ x\cup\odot&=\star && \text{# concatenate the } \cup \text{ inverse of } \odot \text{ to both sides}\\
\otimes\circ x\cup\odot\cup-\odot&=\star\cup-\odot && \text{#} \odot\cup-\odot=\bullet\\
\otimes\circ x\cup\bullet&=\star\cup-\odot && \text{#} \bullet \text{ is the } \cup \text{ neutral element}\\
\otimes\circ x&=\star\cup-\odot && \text{# for } \cup \text{ we have } -\odot=\odot\\
\otimes\circ x&=\star\cup\odot && \text{#}\star\cup\odot=\otimes\\
\otimes\circ x&=\otimes && \text{# concatenate the } \circ \text{ inverse of } \otimes \text{ to both sides}\\
(\otimes)^{-1}\circ\otimes\circ x&=(\otimes)^{-1}\circ\otimes && \text{# multiply with the multiplicative inverse}\\
\star\circ x&=\star\\
x&=\star
\end{aligned}$$

Even though this equation looked really alien at first glance, we could solve it basically exactly the way we solve “normal” equations containing numbers.

Note, however, that whenever a multiplicative inverse is needed to solve an equation in the usual way in a ring, things can be very different than most of us are used to. For example, the following equation cannot be solved for x in the usual way, since there is no multiplicative inverse for $\odot$ in our ring.

$$\odot\circ x=\otimes \tag{4.30}$$

[8]Note that there are more efficient ways to solve this equation. The point of our computation is to show how the axioms of a ring can be used to solve the equation.

We can confirm this by looking at the multiplication table in (4.28) to see that no such x exits.

As another example, the following equation does not have a single solution but two: $x \in \{\star, \otimes\}$.

$$\odot \circ x = \odot \tag{4.31}$$

Having no solution or two solutions is certainly not something we are used to from types like the rational numbers $\mathbb{Q}$.

Example 51 (Ring of Polynomials). Considering the definition of polynomials from section 3.4 again, we notice that what we have informally called the type R of the coefficients must in fact be a commutative ring with a unit, since we need addition, multiplication, commutativity and the existence of a unit for $R[x]$ to have the properties we expect.

In fact, if we consider R to be a ring and we define addition and multiplication of polynomials as in (3.28), the set $R[x]$ is a commutative ring with a unit, where the polynomial 1 is the multiplicative unit. We call this ring the **ring of polynomials with coefficients in** $\boldsymbol{R}$.

```
sage: ZZ['x']                                        192
Univariate Polynomial Ring in x over Integer Ring    193
```

Example 52 (Ring of modular n arithmetic). Let n be a modulus and $(\mathbb{Z}_n, +, \cdot)$ the set of all remainder classes of integers modulo n, with the projection of integer addition and multiplication as defined in section3.3.4. Then $(\mathbb{Z}_n, +, \cdot)$ is a commutative ring with unit 1.

```
sage: Integers(6)                                    194
Ring of integers modulo 6                            195
```

Example 53 (Binary Representations in Modular Arithmetic). TODO (Non unique)

add example

Example 54 (Polynomial evaluation in the exponent of group generators). As we show in section 6.2.3, a key insight in many zero-knowlege protocols is the ability to encode computations as polynomials and then to hide the information of that computation by evaluating the polynomial "in the exponent" of certain cryptographic groups (section 8.2).

To understand the underlying principle of this idea, consider the exponential map (4.1) again. If $\mathbb{G}$ is a finite cyclic group of order n with generator $g \in \mathbb{G}$, then the ring structure of $(\mathbb{Z}_n, +, \cdot)$ corresponds to the group structure of $\mathbb{G}$ in the following way:

$$g^{x+y} = g^x \cdot g^y \qquad g^{x \cdot y} = (g^x)^y \qquad \text{for all } x, y \in \mathbb{Z}_n \tag{4.32}$$

This correspondense allows polynomials with coefficients in $\mathbb{Z}_n$ to be evaluated "in the exponent" of a group generator. To see what this means, let $p \in \mathbb{Z}_n[x]$ be a polynomial with $p(x) = a_m \cdot x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0$, and let $s \in \mathbb{Z}_n$ be an evaluation point. Then the previously defined exponential laws 4.32 imply the following identity:

$$\begin{aligned} g^{p(s)} &= g^{a_m \cdot s^m + a_{m-1}s^{m-1} + \ldots + a_1 s + a_0} \\ &= \left(g^{s^m}\right)^{a_m} \cdot \left(g^{s^{m-1}}\right)^{a_{m-1}} \cdot \ldots \cdot (g^s)^{a_1} \cdot g^{a_0} \end{aligned} \tag{4.33}$$

Utilizing these identities, it is possible to evaluate any polynomial p of degree $deg(p) \leq m$ at a "secret" evaluation point s in the exponent of g without any knowledge about s, assuming that $\mathbb{G}$ is a DL-group. To see this, assume that the set $\{g, g^s, g^{s^2}, \ldots, g^{s^m}\}$ is given, but s is unknown. Then $g^{p(s)}$ can be computed using (4.33), but it is not feasible to compute s.

Example 55. To give an example of the evaluation of a polynomial in the exponent of a finite cyclic group, consider the exponential map from example 36:

$$3^{(\cdot)} : \mathbb{Z}_4 \to \mathbb{Z}_5^* \; ; \; x \mapsto 3^x \tag{4.34}$$

Choosing the polynomial $p(x) = 2x^2 + 3x + 1$ from $\mathbb{Z}_4[x]$, we first evaluate the polynomial at the point $s = 2$, and then write the result into the exponent 3 as follows:

$$\begin{aligned} 3^{p(2)} &= 3^{2\cdot 2^2 + 3\cdot 2 + 1} \\ &= 3^{2\cdot 0 + 2 + 1} \\ &= 3^3 \\ &= 2 \end{aligned}$$

This was possible because we had access to the evaluation point 2. On the other hand, if we only had access to the set $\{3, 4, 1\}$ and we knew that this set represents the set $\{3, 3^s, 3^{s^2}\}$ for some secret value s, we could evaluate p at the point s in the exponent of 3 as follows:

$$\begin{aligned} 3^{p(s)} &= 1^2 \cdot 4^3 \cdot 3^1 \\ &= 1 \cdot 4 \cdot 3 \\ &= 2 \end{aligned}$$

Both computations agree, since the secret point s was equal to 2 in this example. However the second evaluation was possible without any knowledge about s.

agree

4.2.1 Hashing into Modular Arithmetic

As we have seen in section 4.1.7, various constructions for hashing to groups are known and used in applications. As commutative rings are commutative groups when we disregard the multiplication, hash-to-group constructions can be applied for hashing into commutative rings. We review some frequently used applications below.

put sub-subsection title here

One of the most widely used applications of hash-into-ring constructions are hash functions that map into the ring $\mathbb{Z}_n$ of modular n arithmetics for some modulus n. Different approaches of constructing such a function are known, but probably the most widely used ones are based on the insight that the images of general hash functions can be interpreted as binary representations of integers, as explained in example 46.

It follows from this interpretation that one simple method of hashing into $\mathbb{Z}_n$ is constructed by observing that if n is a modulus with a bit length (3.4) of $k = |n|$, then every binary string $< b_0, b_1, \ldots, b_{k-2} >$ of length $k-1$ defines an integer z in the rage $0 \le z \le 2^{k-1} - 1 < n$:

$$z = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \ldots + b_{k-2} \cdot 2^{k-2} \tag{4.35}$$

Now, since $z < n$, we know that z is guaranteed to be in the set $\{0, 1, \ldots, n-1\}$, and hence it can be interpreted as an element of $\mathbb{Z}_n$. Consequently, if $H : \{0,1\}^* \to \{0,1\}^{k-1}$ is a hash function, then a hash-to-ring function can be constructed as follows:

$$H_{|n|_2 - 1} : \{0,1\}^* \to \mathbb{Z}_r : \; s \mapsto H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k-2} \cdot 2^{k-2} \tag{4.36}$$

A drawback of this hash function is that the distribution of the hash values in $\mathbb{Z}_n$ is not necessarily uniform. In fact, if n is larger than 2^{k-1}, then by design $H_{|n|_2 - 1}$ will never hash onto values $z \geq 2^{k-1}$. Using this hashing method therefore generates approximately uniform hashes only if n is very close to 2^{k-1}. In the worst case, when $n = 2^k - 1$, it misses almost half of all elements from $\mathbb{Z}_n$.

An advantage of this approach is that properties like preimage resistance or collision resistance (see section 4.1.7.1) of the original hash function $H(\cdot)$ are preserved.

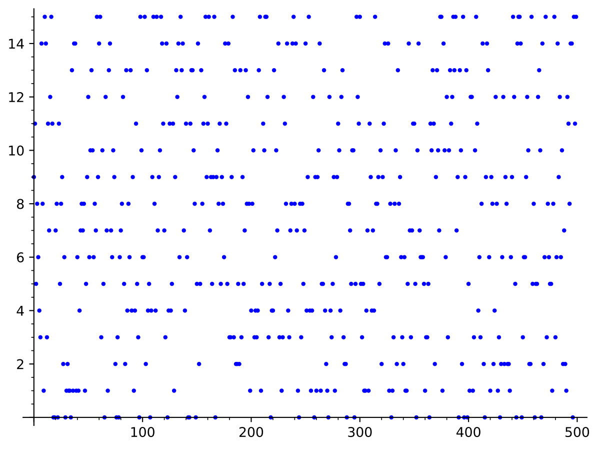
Example 56. To analyze a particular implementation of a $H_{|n|_2-1}$ hash function, we use a 5-bit truncation of the $SHA256$ hash from example 45 and define a hash into $\mathbb{Z}_{16}$ as follows:

$$H_{|16|_2-5} : \{0,1\}^* \to \mathbb{Z}_{16} : \; s \mapsto SHA256(s)_0 \cdot 2^0 + SHAH256(s)_1 \cdot 2^1 + \ldots + SHA256(s)_4 \cdot 2^4$$

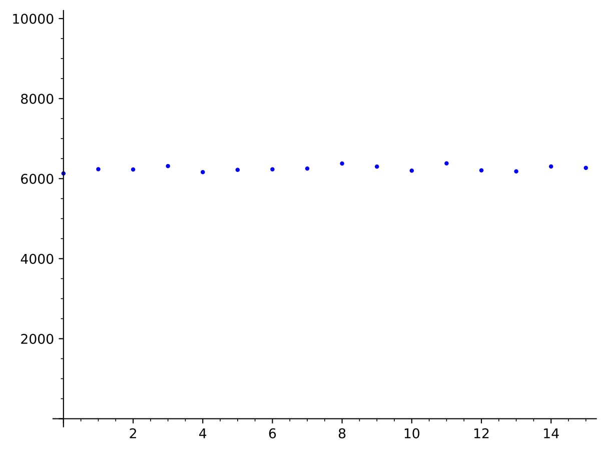
Since $k = |16|_2 = 5$ and $16 - 2^{k-1} = 0$, this hash maps uniformly onto $\mathbb{Z}_{16}$. We can use Sage to implement it:

```
sage: import hashlib                                                      196
sage: def Hash5(x):                                                       197
....:     Z16 = Integers(16)                                              198
....:     hasher = hashlib.sha256(x) # compute SHA56                      199
....:     digest = hasher.hexdigest()                                     200
....:     d = ZZ(digest, base=16) # cast to integer                       201
....:     d = d.str(2)[-4:] # keep 5 least significant bits               202
....:     d = ZZ(d, base=2) # cast to integer                             203
....:     return Z16(d) # cast to Z16                                     204
sage: Hash5(b'')                                                          205
5                                                                         206
```

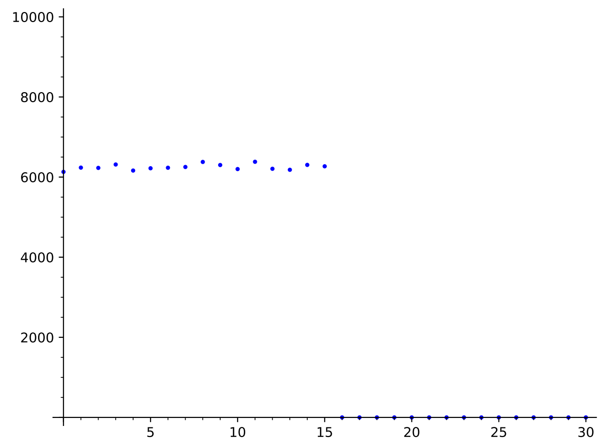
We can then use Sage to apply this function to a large set of input values in order to plot a visualization of the distribution over the set $\{0, \ldots, 15\}$. Executing over 500 input values gives the following plot:



To get an intuition of uniformity, we can count the number of times the hash function $H_{|16|_2-1}$ maps onto each number in the set $\{0, 1, \ldots, 15\}$ in a loop of 100000 hashes, and compare that to the ideal uniform distribution, which would map exactly 6250 samples to each element. This gives the following result:



The lack of uniformity becomes apparent if we want to construct a similar hash function for $\mathbb{Z}_n$ for any other 5 bit integer n in the range $17 \leq n \leq 31$. In this case, the definition of the hash function is exactly the same as for $\mathbb{Z}_{16}$, and hence, the images will not exceed the value 15. So, for example, even in the case of hashing to $\mathbb{Z}_{31}$, the hash function never maps to any value larger than 15, leaving almost half of all numbers out of the image range.



put sub-subsection title here

A second widely used method of hashing into $\mathbb{Z}_n$ is constructed by observing the following: If n is a modulus with a bit-length of $|n|_2 = k_1$, and $H : \{0,1\}^* \to \{0,1\}^{k_2}$ is a hash function that produces digests of size k_2, and $k_2 \geq k_1$, then a hash-to-ring function can be constructed by interpreting the image of H as a binary representation of an integer, and then taking the modulus by n to map into $\mathbb{Z}_n$:.

$$H'_{mod_n} : \{0,1\}^* \to \mathbb{Z}_n : \; s \mapsto \left(H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k_2} \cdot 2^{k_2}\right) \bmod n \tag{4.37}$$

A drawback of this hash function is that computing the modulus requires some computational effort. In addition, the distribution of the hash values in $\mathbb{Z}_n$ might not be uniform, depending on the number $2^{k_2+1} \bmod n$. An advantage of this function is that potential properties of the original hash function $H(\cdot)$ (like preimage resistance or collision resistance) are preserved, and the distribution can be made almost uniform, with only negligible bias depending on what modulus n and images size k_2 are chosen.

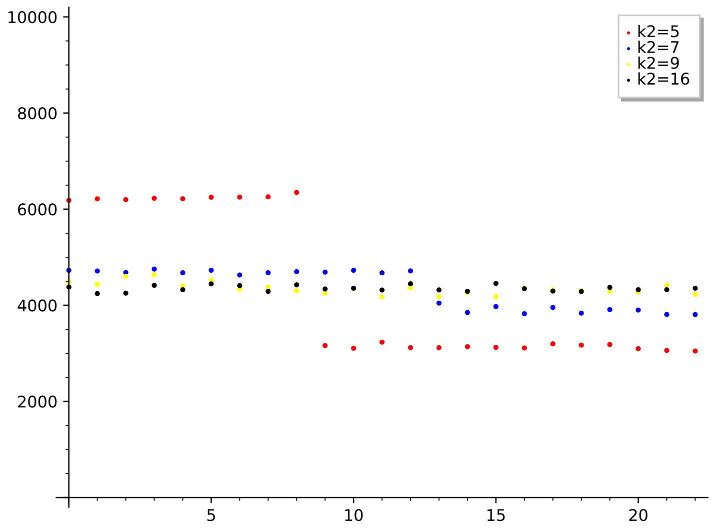
Example 57. To give an implementation of the H_{mod_n} hash function, we use k_2-bit truncation of the $SHA256$ hash from example 45, and define a hash into $\mathbb{Z}_{23}$ as follows:

$$\begin{aligned} &H_{mod_{23},k_2} : \{0,1\}^* \to \mathbb{Z}_{23} : \\ &\qquad s \mapsto \left(SHA256(s)_0 \cdot 2^0 + SHAH256(s)_1 \cdot 2^1 + \ldots + SHA256(s)_{k_2} \cdot 2^{k_2}\right) \bmod 23 \end{aligned}$$

We want to use various instantiations of k_2 to visualize the impact of truncation length on the distribution of the hashes in $\mathbb{Z}_{23}$. We can use Sage to implement it as follows:

```
sage: import hashlib
sage: Z23 = Integers(23)
sage: def Hash_mod23(x, k2):
....:     hasher = hashlib.sha256(x.encode('utf-8')) # Compute
    SHA256
....:     digest = hasher.hexdigest()
....:     d = ZZ(digest, base=16) # cast to integer
....:     d = d.str(2)[-k2:] # keep k2+1 LSB
....:     d = ZZ(d, base=2) # cast to integer
....:     return Z23(d) # cast to Z23
```

We can then use Sage to apply this function to a large set of input values in order to plot visualizations of the distribution over the set $\{0,\ldots,22\}$ for various values of k_2, by counting the number of times it maps onto each number in a loop of 100000 hashes. We get the following plot:



4.2.1.1 The "try-and-increment" method

A third method that can sometimes be found in implementations is the so-called **"try-and-increment" method**. To understand this method, we define an integer $z \in \mathbb{Z}$ from any hash value $H(s)$ as we did in the previous methods:

$$z = H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k-1} \cdot 2^k \tag{4.38}$$

Hashing into $\mathbb{Z}_n$ is then achievable by first computing z, and then trying to see if $z \in \mathbb{Z}_n$. If it is, then the hash is done; if not, the string s is modified in a deterministic way and the process is repeated until a suitable element $z \in \mathbb{Z}_n$ is found. A suitable, deterministic modification could be to concatenate the original string by some bit counter. A "try-and-increment" algorithm would then work like in algorithm 6.

Depending on the parameters, this method can be very efficient. In fact, if k is sufficiently large and n is close to 2^{k+1}, the probability for $z < n$ is very high, and the repeat loop will almost always be executed a single time only. A drawback is, however, that the probability of having to execute the loop multiple times is not zero.

Algorithm 6 Hash-to-$\mathbb{Z}_n$

```
Require: $n \in \mathbb{Z}$ with $|n|_2 = k$ and $s \in \{0,1\}^*$
  procedure TRY-AND-INCREMENT($n, k, s$)
    $c \leftarrow 0$
    repeat
      $s' \leftarrow s||c\_bits()$
      $z \leftarrow H(s')_0 \cdot 2^0 + H(s')_1 \cdot 2^1 + \ldots + H(s')_k \cdot 2^k$
      $c \leftarrow c+1$
    until $z < n$
    return $x$
  end procedure
Ensure: $z \in \mathbb{Z}_n$
```

4.3 Fields

We started this chapter with the definition of a group (section 4.1), which we then expanded into the definition of a commutative ring with a unit (section4.2). These types of rings generalize the behavior of integers. In this section, we look at those special cases of commutative rings where every element other than the neutral element of addition has a multiplicative inverse. Those structures behave very much like the set of rational numbers $\mathbb{Q}$. Rational numbers are, in a sense, an extension of the ring of integers, that is, they are constructed by including newly defined multiplicative inverses (fractions) to the integers. Fields are defined as follows:

Definition 4.3.0.1 (Field). A **field** $(\mathbb{F},+,\cdot)$ is a set $\mathbb{F}$ with two maps $+ : \mathbb{F}\times\mathbb{F} \to \mathbb{F}$ and $\cdot : \mathbb{F}\times\mathbb{F} \to \mathbb{F}$ called **addition** and **multiplication**, such that the following conditions hold:

- $(\mathbb{F},+)$ is a commutative group, where the neutral element is denoted by 0.
- $(\mathbb{F}\setminus\{0\},\cdot)$ is a commutative group, where the neutral element is denoted by 1.
- (Distributivity) The equation $g_1\cdot(g_2+g_3) = g_1\cdot g_2 + g_1\cdot g_3$ holds for all $g_1,g_2,g_3 \in \mathbb{F}$.

If $(\mathbb{F},+,\cdot)$ is a field and $\mathbb{F}' \subset \mathbb{F}$ is a subset of $\mathbb{F}$ such that the restriction of addition and multiplication to $\mathbb{F}'$ define a field with addition $+ : \mathbb{F}'\times\mathbb{F}' \to \mathbb{F}'$ and multiplication $\cdot : \mathbb{F}'\times\mathbb{F}' \to \mathbb{F}'$ on $\mathbb{F}'$, then $(\mathbb{F}',+,\cdot)$ is called a **subfield** of $(\mathbb{F},+,\cdot)$ and $(\mathbb{F},+,\cdot)$ is called an **extension field** of $(\mathbb{F}',+,\cdot)$.

Notation and Symbols 10. If there is no risk of ambiguity (about what the addition and multiplication maps of a field are), we frequently omit the symbols $+$ and $\cdot$, and simply write $\mathbb{F}$ as notation for a field, keeping maps implicit. In this case, we also say that $\mathbb{F}$ is of field type, indicating that $\mathbb{F}$ is not simply a set but a set with an addition and a multiplication map that satisfies the definition of a field (4.3.0.1).[9]

We call $(\mathbb{F},+)$ the **additive group** of the field. We use the notation $\mathbb{F}^* := \mathbb{F}\setminus\{0\}$ for the set of all elements excluding the neutral element of addition, called $(\mathbb{F}^*,\cdot)$ the **multiplicative group** of the field.

The **characteristic** of a field $\mathbb{F}$, represented as $char(\mathbb{F})$, is the smallest natural number $n \geq 1$ for which the n-fold sum of the multiplicative neutral element 1 equals zero, i.e. for which

[9]Since fields are of great importance in cryptography and number theory, many books exists on that topic. For a general introduction, see, for example, chapter 6, section 1 in Mignotte [1992], or chapter 1, section 2 in Lidl and Niederreiter [1986].

$\sum_{i=1}^{n} 1 = 0$. If such an $n > 0$ exists, the field is said to have a **finite characteristic**. If, on the other hand, every finite sum of 1 is such that it is not equal to zero, then the field is defined to have characteristic 0. S: Tried to disambiguate the scope of negation between 1. "It is true of every finite sum of 1 that it is not equal to 0" and 2. "It is not true of every finite sum of 1 that it is equal to 0" From the example below, it looks like 1. is the intended meaning here, correct?

Check change of wording

Example 58 (Field of rational numbers). Probably the best known example of a field is the set of rational numbers $\mathbb{Q}$ together with the usual definition of addition, subtraction, multiplication and division. Since there is no natural number $n \in \mathbb{N}$ such that $\sum_{j=0}^{n} 1 = 0$ in the set of rational numbers, the characteristic of the field $\mathbb{Q}$ is given by $char(\mathbb{Q}) = 0$.

```
sage: QQ
Rational Field
```

Example 59 (Field with two elements). It can be shown that, in any field, the neutral element of addition 0 must be different from the neutral element of multiplication 1, that is, $0 \neq 1$ always holds in a field. This means that the smallest field must contain at least two elements. As the following addition and multiplication tables show, there is indeed a field with two elements, which is usually called $\mathbb{F}_2$:

Let $\mathbb{F}_2 := \{0,1\}$ be a set that contains two elements, and let addition and multiplication on $\mathbb{F}_2$ be defined as follows:

$$\begin{array}{c|cc} + & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} \qquad \begin{array}{c|cc} \cdot & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array} \tag{4.39}$$

Since $1+1=0$ in the field $\mathbb{F}_2$, we know that the characteristic of $\mathbb{F}_2$ given by $char(\mathbb{F}_2) = 2$. The multiplicative subgroup $\mathbb{F}_2^*$ of $\mathbb{F}_2$ is given by the trivial group $\{1\}$.

```
sage: F2 = GF(2)
sage: F2(1) # Get an element from GF(2)
1
sage: F2(1) + F2(1) # Addition
0
sage: F2(1) / F2(1) # Division
1
```

Exercise 46. Consider the ring of modular 5 arithmetics $(\mathbb{Z}_5, +, \cdot)$ from example 14. Show that $(\mathbb{Z}_5, +, \cdot)$ is a field. What is the characteristic of $\mathbb{Z}_5$? Prove that the equation $a \cdot x = b$ has only a single solution $x \in \mathbb{Z}_5$ for any given $a, b \in \mathbb{Z}_5^*$.

Exercise 47. Consider the ring of modular 6 arithmetics $(\mathbb{Z}_6, +, \cdot)$ from example 9. Show that $(\mathbb{Z}_6, +, \cdot)$ is not a field.

4.3.1 Prime fields

As we have seen in many of the examples in previous sections, modular arithmetic behaves similarly to the ordinary arithmetics of integers in many ways. This is due to the fact that remainder class sets $\mathbb{Z}_n$ are commutative rings with units (see example 52).

However, we have also seen in example 36 that, whenever the modulus is a prime number, every remainder class other than the zero class has a modular multiplicative inverse. This is an important observation, since it immediately implies that, in case the modulus is a prime number, the remainder class set $\mathbb{Z}_n$ is not just a ring but actually a **field**. Moreover, since $\sum_{j=0}^{n} 1 = 0$ in $\mathbb{Z}_n$, we know that those fields have the finite characteristic n.

Notation and Symbols 11 (Prime Fields). Let $p \in \mathbb{P}$ be a prime number and $(\mathbb{Z}_p, +, \cdot)$ the ring of modular p arithmetics (see example 52). To distinguish prime fields from arbitrary modular arithmetic rings, we write $(\mathbb{F}_p, +, \cdot)$ for the ring of modular p arithmetics and call it the **prime field** of characteristic p.

Prime fields are the foundation of many of the contemporary algebra-based cryptographic systems, as they have a number of desirable properties. One of these is that any prime field of characteristic p contains exactly p elements, which can be represented on a computer with not more than $log_2(p)$ many bits. On the other hand, fields like rational numbers require a potentially unbounded amount of bits for any full-precision representation.[10]

Since prime fields are special cases of modular arithmetic rings, addition and multiplication can be computed by first doing normal integer addition and multiplication, and then considering the remainder in Euclidean division by p as the result. For any prime field element $x \in \mathbb{F}_p$, its additive inverse (the negative) is given by $-x = p - x \bmod p$. For $x \neq 0$, the multiplicative inverse always exists, and is given by $x^{-1} = x^{p-2}$. Division is then defined by multiplication with the multiplicative inverse, as explained in section 3.3.5. Alternatively, the multiplicative inverse can be computed using the Extended Euclidean Algorithm as explained in (3.23).

Example 60. The smallest possible field is the field $\mathbb{F}_2$ of characteristic 2, as we have seen in example 59. It is the prime field of the prime number 2.

Example 61. The field $\mathbb{F}_5$ from example 14 is a prime field, as defined by its addition and multiplication table (3.24).

Example 62. To summarize the basic aspects of computation in prime fields, let us consider the prime field $\mathbb{F}_5$ (example 14) and simplify the following expression:

$$\left(\frac{2}{3} - 2\right) \cdot 2 \tag{4.40}$$

The first thing to note is that, since $\mathbb{F}_5$ is a field, all rules are identical to the rules we learned in school when we where dealing with rational, real or complex numbers. This means we can use methods like bracketing (distributivity) or addition as usual. For ease of computation, we can consult the addition and multiplication tables in (3.24).

$$\begin{aligned}
\left(\frac{2}{3} - 2\right) \cdot 2 &= \frac{2}{3} \cdot 2 - 2 \cdot 2 && \text{\# distributive law} \\
&= \frac{2 \cdot 2}{3} - 2 \cdot 2 && 4 \bmod 5 = 4 \\
&= \frac{4}{3} - 4 && \text{\# multiplicative inverse of 3 is } 3^{5-2} \bmod 5 = 2 \\
&= 4 \cdot 2 - 4 && \text{\# additive inverse of 4 is } 5 - 4 = 1 \\
&= 4 \cdot 2 + 1 && 8 \bmod 5 = 3 \\
&= 3 + 1 && 4 \bmod 5 = 4 \\
&= 4
\end{aligned}$$

In this example, we computed the multiplicative inverse of 3 using the identity $x^{-1} = x^{p-2}$ in a prime field. This is impractical for large prime numbers. Recall that another way of

[10] For a detailed introduction to the theory of prime fields, see, for example, chapter 2 in Lidl and Niederreiter [1986], or chapter 6 in Mignotte [1992].

computing the multiplicative inverse is the Extended Euclidean Algorithm (see 3.13). To refresh our memory, the algorithm solves the equation $x^{-1} \cdot 3 + t \cdot 5 = 1$, for x^{-1} (even though t is irrelevant in this case). We get the following:

k	r_k	x_k^{-1}	t_k
0	3	1	$\cdot$
1	5	0	$\cdot$
2	3	1	$\cdot$
3	2	-1	$\cdot$
4	1	2	$\cdot$

(4.41)

So the multiplicative inverse of 3 in $\mathbb{Z}_5$ is 2, and, indeed, if we compute the product of 3 with its multiplicative inverse 2, we get the neutral element 1 in $\mathbb{F}_5$.

Exercise 48 (Prime field $\mathbb{F}_3$). Construct the addition and multiplication table of the prime field $\mathbb{F}_3$.

Exercise 49 (Prime field $\mathbb{F}_{13}$). Construct the addition and multiplication table of the prime field $\mathbb{F}_{13}$.

Exercise 50. Consider the prime field $\mathbb{F}_{13}$ from exercise 49. Find the set of all pairs $(x,y) \in \mathbb{F}_{13} \times \mathbb{F}_{13}$ that satisfy the following equation:

$$x^2 + y^2 = 1 + 7 \cdot x^2 \cdot y^2 \tag{4.42}$$

4.3.2 Square Roots

As we know from integer arithmetics, some integers, like 4 or 9, are squares of other integers: for example, $4 = 2^2$ and $9 = 3^2$. However, we also know that not all integers are squares of other integers: for example, there is no integers $x \in \mathbb{Z}$ such that $x^2 = 2$. If an integer a is square of another integer b, then it make sense to define the square root of a to be b.

In the context of prime fields, an element that is a square of another element is also called a **quadratic residue**, and an element that is not a square of another element is called a **quadratic non-residue**. This distinction is of particular importance in our studies on elliptic curves (chapter 5), as only square numbers can actually be points on an elliptic curve.

To make the intuition of quadratic residues and their roots precise, we give the following definition:

Definition 4.3.2.1. let $p \in \mathbb{P}$ be a prime number and $\mathbb{F}_p$ its associated prime field. Then a number $x \in \mathbb{F}_p$ is called a **square root** of another number $y \in \mathbb{F}_p$, if x is a solution to the following equation:

$$x^2 = y \tag{4.43}$$

In this case, y is called a **quadratic residue**. On the other hand, if y is given and the quadratic equation has no solution x , we call y a **quadratic non-residue**.[11]

For any $y \in \mathbb{F}_p$, we denote the set of all square roots of y in the prime field $\mathbb{F}_p$ as follows:

$$\sqrt{y} := \{x \in \mathbb{F}_p \mid x^2 = y\} \tag{4.44}$$

Informally speaking, quadratic residues are numbers that have a square root, while quadratic non-residues are numbers that don't have square roots. The situation therefore parallels the

[11]A more detailed introduction to quadratic residues and their square roots in addition with an introduction to algorithms that compute square roots can be found, for example, in chapter 1, section 1.5 of Cohen [2010].

familiar case of integers, where some integers like 4 or 9 have a square root, and others like 2 or 3 don't (within the set of integers).

If y is a quadratic non-residue, then $\sqrt{y} = \emptyset$ (an empty set), and if $y = 0$, then $\sqrt{y} = \{0\}$.

Moreover if $y \neq 0$ is a quadratic residue, then it has precisely two roots $\sqrt{y} = \{x, p-x\}$ for some $x \in \mathbb{F}_p$. We adopt the convention to call the smaller one (when interpreted as an integer) the **positive square root** and the larger one the **negative square root**.

If $p \in \mathbb{P}_{\geq 3}$ is an odd prime number with associated prime field $\mathbb{F}_p$, then there are precisely $(p+1)/2$ many quardratic residues and $(p-1)/2$ quadratic non-residues.

Example 63 (Quadratic residues and roots in $\mathbb{F}_5$). Let us consider the prime field $\mathbb{F}_5$ from example 14 again. All square numbers can be found on the main diagonal of the multiplication table in (3.24). As you can see, in $\mathbb{F}_5$, only the numbers 0, 1 and 4 have square roots: $\sqrt{0} = \{0\}$, $\sqrt{1} = \{1,4\}$, $\sqrt{2} = \emptyset$, $\sqrt{3} = \emptyset$ and $\sqrt{4} = \{2,3\}$. The numbers 0, 1 and 4 are therefore quadratic residues, while the numbers 2 and 3 are quadratic non-residues.

In order to describe whether an element of a prime field is a square number or not, the so-called **Legendre symbol** can sometimes be found in the literature (e.g. chapter 1, section 1.5. of Cohen [2010]), defined as follows:

Let $p \in \mathbb{P}$ be a prime number and $y \in \mathbb{F}_p$ an element from the associated prime field. Then the *Legendre symbol* of y is defined as follows:

$$\left(\frac{y}{p}\right) := \begin{cases} 1 & \text{if } y \text{ has square roots} \\ -1 & \text{if } y \text{ has no square roots} \\ 0 & \text{if } y = 0 \end{cases} \tag{4.45}$$

Example 64. Looking at the quadratic residues and non-residues in $\mathbb{F}_5$ from example 14 again, we can deduce the following Legendre symbols based on example 63.

$$\left(\tfrac{0}{5}\right) = 0, \quad \left(\tfrac{1}{5}\right) = 1, \quad \left(\tfrac{2}{5}\right) = -1, \quad \left(\tfrac{3}{5}\right) = -1, \quad \left(\tfrac{4}{5}\right) = 1 \,.$$

The Legendre symbol provides a criterion to decide whether or not an element from a prime field has a quadratic root or not. This, however, is not just of theoretical use: the so-called **Euler criterion** provides a compact way to actually compute the Legendre symbol. To see that, let $p \in \mathbb{P}_{\geq 3}$ be an odd prime number and $y \in \mathbb{F}_p$. Then the Legendre symbol can be computed as follows:

$$\left(\frac{y}{p}\right) = y^{\frac{p-1}{2}} \tag{4.46}$$

Example 65. Looking at the quadratic residues and non-residues in $\mathbb{F}_5$ from example 63 again, we can compute the following Legendre symbols using the Euler criterion:

$$\begin{aligned}
\left(\frac{0}{5}\right) &= 0^{\frac{5-1}{2}} = 0^2 = 0 \\
\left(\frac{1}{5}\right) &= 1^{\frac{5-1}{2}} = 1^2 = 1 \\
\left(\frac{2}{5}\right) &= 2^{\frac{5-1}{2}} = 2^2 = 4 = -1 \\
\left(\frac{3}{5}\right) &= 3^{\frac{5-1}{2}} = 3^2 = 4 = -1 \\
\left(\frac{4}{5}\right) &= 4^{\frac{5-1}{2}} = 4^2 = 1
\end{aligned}$$

Exercise 51. Consider the prime field $\mathbb{F}_{13}$ from exercise 49. Compute the Legendre symbol $\left(\frac{x}{13}\right)$ and the set of roots $\sqrt{x}$ for all elements $x \in \mathbb{F}_{13}$.

4.3.2.1 Hashing into prime fields

An important problem in cryptography is the ability to hash to (various subsets) of elliptic curves. As we will see in chapter 5, those curves are often defined over prime fields, and hashing to a curve might start with hashing to the prime field. It is therefore important to understand how to hash into prime fields.

In section 4.2.1, we looked at a few methods of hashing into the modular arithmetic rings $\mathbb{Z}_n$ for arbitrary $n > 1$. As prime fields are just special instances of those rings, all methods for hashing into $\mathbb{Z}_n$ functions can be used for hashing into prime fields, too.

4.3.3 Prime Field Extensions

Prime fields, as defined in the previous section, are basic building blocks of cryptography. However, as we will see in chapter 8, so-called pairing-based SNARK systems are crucially dependent on certain group pairings (4.9) defined on elliptic curves over **prime field extensions**. In this section, we therefore introduce those extensions.[12]

Given some prime number $p \in \mathbb{P}$, a natural number $m \in \mathbb{N}$, and an irreducible polynomial $P \in \mathbb{F}_p[x]$ of degree m with coefficients from the prime field $\mathbb{F}_p$, a prime field extension $(\mathbb{F}_{p^m}, +, \cdot)$ is defined as follows.

The set $\mathbb{F}_{p^m}$ of the prime field extension is given by the set of all polynomials with a degree less than m:

$$\mathbb{F}_{p^m} := \{a_{m-1}x^{m-1} + a_{k-2}x^{k-2} + \ldots + a_1x + a_0 \mid a_i \in \mathbb{F}_p\} \tag{4.47}$$

The addition law of the prime field extension $\mathbb{F}_{p^m}$ is given by the usual addition of polynomials as defined in (3.28):

$$+ : \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \to \mathbb{F}_{p^m}\ , (\textstyle\sum_{j=0}^m a_jx^j, \sum_{j=0}^m b_jx^j) \mapsto \sum_{j=0}^m (a_j + b_j)x^j \tag{4.48}$$

The multiplication law of the prime field extension $\mathbb{F}_{p^m}$ is given by first multiplying the two polynomials as defined in (3.29), then dividing the result by the irreducible polynomial p and keeping the remainder:

$$\cdot : \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \to \mathbb{F}_{p^m}\ ,\ (\textstyle\sum_{j=0}^m a_jx^j, \sum_{j=0}^m b_jx^j) \mapsto \left(\sum_{n=0}^{2m} \sum_{i=0}^n a_ib_{n-i}x^n\right) \bmod P \tag{4.49}$$

The neutral element of the additive group $(\mathbb{F}_{p^m}, +)$ is given by the zero polynomial 0. The additive inverse is given by the polynomial with all negative coefficients. The neutral element of the multiplicative group $(\mathbb{F}^*_{p^m}, \cdot)$ is given by the unit polynomial 1. The multiplicative inverse can be computed by the Extended Euclidean Algorithm (see 3.13). check reference

We can see from the definition of $\mathbb{F}_{p^m}$ that the field is of characteristic p, since the multiplicative neutral element 1 is equivalent to the multiplicative element 1 from the underlying prime field, and hence $\sum_{j=0}^p 1 = 0$. Moreover, $\mathbb{F}_{p^m}$ is finite and contains p^m many elements, since elements are polynomials of degree $< m$, and every coefficient a_j can have p many different values. In addition, we see that the prime field $\mathbb{F}_p$ is a subfield of $\mathbb{F}_{p^m}$ that occurs when we restrict the elements of $\mathbb{F}_{p^m}$ to polynomials of degree zero.

[12] A more detailed introduction can be found for example in chapter 2 of Lidl and Niederreiter [1986].

One key point is that the construction of $\mathbb{F}_{p^m}$ depends on the choice of an irreducible polynomial, and, in fact, different choices will give different multiplication tables, since the remainders from dividing a polynomial product by those polynomials will be different.

It can, however, be shown that the fields for different choices of P are **isomorphic**, which means that there is a one-to-one correspondence between all of them. As a result, from an abstract point of view, they are the same thing. From an implementations point of view, however, some choices are preferable to others because they allow for faster computations.

Remark 3. Similarly to the way prime fields $\mathbb{F}_p$ are generated by starting with the ring of integers and then dividing by a prime number p and keeping the remainder, prime field extensions $\mathbb{F}_{p^m}$ are generated by starting with the ring $\mathbb{F}_p[x]$ of polynomials and then dividing them by an irreducible polynomial of degree m and keeping the remainder.

In fact, it can be shown that $\mathbb{F}_{p^m}$ is the set of all remainders when dividing any polynomial $Q \in \mathbb{F}_p[x]$ by an irreducible polynomial P of degree m. This is analogous to how $\mathbb{F}_p$ is the set of all remainders when dividing integers by p.

Any field $\mathbb{F}_{p^m}$ constructed in the above manner is a field extension of $\mathbb{F}_p$. To be more general, a field $\mathbb{F}_{p^{m_2}}$ is a field extension of a field $\mathbb{F}_{p^{m_1}}$ if and only if m_1 divides m_2. From this, we can deduce that, for any given fixed prime number, there are nested sequences of subfields whenever the power m_j divides the power m_{j+1}:

$$\mathbb{F}_p \subset \mathbb{F}_{p^{m_1}} \subset \cdots \subset \mathbb{F}_{p^{m_k}} \tag{4.50}$$

To get a more intuitive picture of this, we construct an extension field of the prime field $\mathbb{F}_3$ in the following example, and we can see how $\mathbb{F}_3$ sits inside that extension field.

Example 66 (The Extension field $\mathbb{F}_{3^2}$). In exercise 48, we have constructed the prime field $\mathbb{F}_3$. In this example, we apply the definition of a field extension (4.47) to construct the extension field $\mathbb{F}_{3^2}$. We start by choosing an irreducible polynomial of degree 2 with coefficients in $\mathbb{F}_3$. We try $P(t) = t^2 + 1$. Possibly the fastest way to show that P is indeed irreducible is to just insert all elements from $\mathbb{F}_3$ to see if the result is ever zero. We compute as follows:

$$\begin{aligned}
P(0) &= 0^2 + 1 = 1\\
P(1) &= 1^2 + 1 = 2\\
P(2) &= 2^2 + 1 = 1 + 1 = 2
\end{aligned}$$

This implies that P is irreducible, since all factors must be of the form $(t-a)$ for a being a root of P. The set $\mathbb{F}_{3^2}$ contains all polynomials of degrees lower than two with coefficients in $\mathbb{F}_3$, which are precisely as listed below:

$$\mathbb{F}_{3^2} = \{0, 1, 2, t, t+1, t+2, 2t, 2t+1, 2t+2\} \tag{4.51}$$

As expected, our extension field contains 9 elements. Addition is defined as addition of polynomials; for example $(t+2)+(2t+2) = (1+2)t + (2+2) = 1$. Doing this computation for all elements gives the following addition table

+	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
0	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
1	1	2	0	t+1	t+2	t	2t+1	2t+2	2t
2	2	0	1	r+2	t	t+1	2t+2	2t	2t+1
t	t	t+1	t+2	2t	2t+1	2t+2	0	1	2
t+1	t+1	t+2	t	2t+1	2t+2	2t	1	2	0
t+2	t+2	t	t+1	2t+2	2t	2t+1	2	0	1
2t	2t	2t+1	2t+2	0	1	2	t	t+1	t+2
2t+1	2t+1	2t+2	2t	1	2	0	t+1	t+2	t
2t+2	2t+2	2t	2t+1	2	0	1	t+2	t	t+1

(4.52)

As we can see, the group $(\mathbb{F}_3,+)$ is a subgroup of the group $(\mathbb{F}_{3^2},+)$, obtained by only considering the first three rows and columns of this table.

We can use the addition table (4.52) to deduce the additive inverse (the negative) of any element from $\mathbb{F}_{3^2}$. For example, in $\mathbb{F}_{3^2}$ we have $-(2t+1) = t+2$, since $(2t+1)+(t+2) = 0$.

Multiplication needs a bit more computation, as we first have to multiply the polynomials, and whenever the result has a degree ≥ 2, we have to apply a polynomial division algorithm (algorithm 3) to divide the product by the polynomial P and keep the remainder. To see how this works, let us compute the product of $t+2$ and $2t+2$ in $\mathbb{F}_{3^2}$:

$$\begin{aligned}
(t+2)\cdot(2t+2) &= (2t^2+2t+t+1) \bmod (t^2+1) \\
&= (2t^2+1) \bmod (t^2+1) \qquad\qquad \# \, 2t^2+1 : t^2+1 = 2+\frac{2}{t^2+1} \\
&= 2
\end{aligned}$$

This means that the product of $t+2$ and $2t+2$ in $\mathbb{F}_{3^2}$ is 2. Performing this computation for all elements gives the following multiplication table:

·	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
0	0	0	0	0	0	0	0	0	0
1	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
2	0	2	1	2t	2t+2	2t+1	t	t+2	t+1
t	0	t	2t	2	t+2	2t+2	1	t+1	2t+1
t+1	0	t+1	2t+2	t+2	2t	1	2t+1	2	t
t+2	0	t+2	2t+1	2t+2	1	t	t+1	2t	2
2t	0	2t	t	1	2t+1	t+1	2	2t+2	t+2
2t+1	0	2t+1	t+2	t+1	2	2t	2t+2	t	1
2t+2	0	2t+2	t+1	2t+1	t	2	t+2	1	2t

(4.53)

As was the case in previous examples, we can use the table (4.53) to deduce the multiplicative inverse of any non-zero element from $\mathbb{F}_{3^2}$. For example, in $\mathbb{F}_{3^2}$ we have $(2t+1)^{-1} = 2t+2$, since $(2t+1)\cdot(2t+2) = 1$.

Looking at the multiplication table (4.53), we can also see that the only quadratic residues in $\mathbb{F}_{3^2}$ are from the set $\{0,1,2,t,2t\}$, with $\sqrt{0} = \{0\}$, $\sqrt{1} = \{1,2\}$, $\sqrt{2} = \{t,2t\}$, $\sqrt{t} = \{t+2,2t+1\}$ and $\sqrt{2t} = \{t+1,2t+2\}$.

Since $\mathbb{F}_{3^2}$ is a field, we can solve equations as we would for other fields (such as rational numbers). To see that, let us find all $x \in \mathbb{F}_{3^2}$ that solve the quadratic equation $(t+1)(x^2+(2t+$

$2)) = 2$. We compute as follows:

$$\begin{aligned}
(t+1)(x^2+(2t+2)) &= 2 && \text{\# 2 distributive law}\\
(t+1)x^2+(t+1)(2t+2) &= 2 &&\\
(t+1)x^2+(t) &= 2 && \text{\# 2 add the additive inverse of } t\\
(t+1)x^2+(t)+(2t) &= (2)+(2t) &&\\
(t+1)x^2 &= 2t+2 && \text{\# multiply with the multiplicative invers of } t+1\\
(t+2)(t+1)x^2 &= (t+2)(2t+2) && \text{\# multiply with the multiplicative invers of } t+1\\
x^2 &= 2 && \text{\# 2 is quadratic residue. Take the roots.}\\
x &\in \{t, 2t\} &&
\end{aligned}$$

Computations in extension fields are arguably on the edge of what can reasonably be done with pen and paper. Fortunately, Sage provides us with a simple way to do these computations.

```
sage: Z3 = GF(3) # prime field
sage: Z3t.<t> = Z3[] # polynomials over Z3
sage: P = Z3t(t^2+1)
sage: P.is_irreducible()
True
sage: F3_2.<t> = GF(3^2, name='t', modulus=P) # Extension
   field F_3^2
sage: F3_2
Finite Field in t of size 3^2
sage: F3_2(t+2)*F3_2(2*t+2) == F3_2(2)
True
sage: F3_2(2*t+2)^(-1) # multiplicative inverse
2*t + 1
sage: # verify our solution to (t+1)(x^2 + (2t+2)) = 2
sage: F3_2(t+1)*(F3_2(t)**2 + F3_2(2*t+2)) == F3_2(2)
True
sage: F3_2(t+1)*(F3_2(2*t)**2 + F3_2(2*t+2)) == F3_2(2)
True
```

Exercise 52. Consider the extension field $\mathbb{F}_{3^2}$ from the previous example and find all pairs of elements $(x,y) \in \mathbb{F}_{3^2}$, for which the following equation holds:

$$y^2 = x^3 + 4 \tag{4.54}$$

Exercise 53. Show that the polynomial $Q = x^2 + x + 2$ from $\mathbb{F}_3[x]$ is irreducible. Construct the multiplication table of $\mathbb{F}_{3^2}$ with respect to Q and compare it to the multiplication table of $\mathbb{F}_{3^2}$ from example 66.

Exercise 54. Show that the polynomial $P = t^3 + t + 1$ from $\mathbb{F}_5[t]$ is irreducible. Then consider the extension field $\mathbb{F}_{5^3}$ defined relative to P. Compute the multiplicative inverse of $(2t^2+4) \in \mathbb{F}_{5^3}$ using the Extended Euclidean Algorithm. Then find all $x \in \mathbb{F}_{5^3}$ that solve the following equation:

$$(2t^2+4)(x-(t^2+4t+2)) = (2t+3) \tag{4.55}$$

Exercise 55. Consider the prime field $\mathbb{F}_5$. Show that the polynomial $P = x^2 + 2$ from $\mathbb{F}_5[x]$ is irreducible. Implement the finite field $\mathbb{F}_{5^2}$ in Sage.

4.4 Projective Planes

Projective planes are particular geometric constructs defined over a given field. In a sense, projective planes extend the concept of the ordinary Euclidean plane by including "points at infinity."[13]

To understand the idea of constructing of projective planes, note that, in an ordinary Euclidean plane, two lines either intersect in a single point or are parallel. In the latter case, both lines are either the same, that is, they intersect in all points, or do not intersect at all. A projective plane can then be thought of as an ordinary plane, but equipped with an additional "point at infinity" such that two different lines always intersect in a single point. Parallel lines intersect "at infinity".

Such an inclusion of infinity points makes projective planes particularly useful in the description of elliptic curves, as the description of such a curve in an ordinary plane needs an additional symbol for "the point at infinity" to give the set of points on the curve the structure of a group 5.1. Translating the curve into projective geometry includes this "point at infinity" more naturally into the set of all points on a projective plane.

To be more precise, let $\mathbb{F}$ be a field, $\mathbb{F}^3 := \mathbb{F} \times \mathbb{F} \times \mathbb{F}$ the set of all tuples of three elements over $\mathbb{F}$ and $x \in \mathbb{F}^3$ with $x = (X,Y,Z)$. Then there is exactly one *line* L_x in $\mathbb{F}^3$ that intersects both $(0,0,0)$ and x, given by the set $L_x = \{(k \cdot X, k \cdot Y, k \cdot Z) \mid k \in \mathbb{F}\}$. A point in the **projective plane** over $\mathbb{F}$ can then be defined as such a **line** if we exclude the intersection of that line with $(0,0,0)$. This leads to the following definition of a **point** in projective geometry:

$$[X : Y : Z] := \{(k \cdot X, k \cdot Y, k \cdot Z) \mid k \in \mathbb{F}^*\} \tag{4.56}$$

Points in projective geometry are therefore lines in $\mathbb{F}^3$ where the intersection with $(0,0,0)$ is excluded. Given a field $\mathbb{F}$, the **projective plane** of that field is then defined as the set of all points excluding the point $[0 : 0 : 0]$:

$$\mathbb{FP}^2 := \{[X : Y : Z] \mid (X,Y,Z) \in \mathbb{F}^3 \text{ with } (X,Y,Z) \neq (0,0,0)\} \tag{4.57}$$

It can be shown that a projective plane over a finite field $\mathbb{F}_{p^m}$ contains $p^{2m} + p^m + 1$ number of elements.

To understand why the projective point $[X : Y : Z]$ is also a line, consider the situation where the underlying field $\mathbb{F}$ is the set of rational numbers $\mathbb{Q}$. In this case, $\mathbb{Q}^3$ can be seen as the three-dimensional space, and $[X : Y : Z]$ is an ordinary line in this 3-dimensional space that intersects zero and the point with coordinates X, Y and Z such that the intersection with zero is excluded.

The key observation here is that points in the projective plane $\mathbb{FP}^2$ are lines in the 3-dimensional space $\mathbb{F}^3$. However, it should be kept in mind that, for finite fields, the terms **space** and **line** share very little visual similarity with their counterparts over the set of rational numbers.

It follows from this that points $[X : Y : Z] \in \mathbb{FP}^2$ are not simply described by fixed coordinates (X,Y,Z), but by **sets of coordinates**, where two different coordinates (X_1,Y_1,Z_1) and (X_2,Y_2,Z_2) describe the same point if and only if there is some non-zero field element $k \in \mathbb{F}^*$ such that $(X_1,Y_1,Z_1) = (k \cdot X_2, k \cdot Y_2, k \cdot Z_2)$. Points $[X : Y : Z]$ are called **projective coordinates**.

Notation and Symbols 12 (Projective coordinates). Projective coordinates of the form $[X : Y : 1]$ are descriptions of so-called **affine points**. Projective coordinates of the form $[X : Y : 0]$ are descriptions of so-called **points at infinity**. In particular, the projective coordinate $[1 : 0 : 0]$ describes the so-called **line at infinity**.

[13]A detailed explanation of the ideas that lead to the definition of projective planes can be found, for example, in chapter 2 of Ellis and Ellis [1992] or in appendix A of Silverman and Tate [1994].

Example 67. Consider the field $\mathbb{F}_3$ from exercise 48 . As this field only contains three elements, it does not take too much effort to construct its associated projective plane $\mathbb{F}_3\mathbb{P}^2$, which we know only contains 13 elements.

To find $\mathbb{F}_3\mathbb{P}^2$, we have to compute the set of all lines in $\mathbb{F}_3 \times \mathbb{F}_3 \times \mathbb{F}_3$ that intersect $(0,0,0)$, excluding their intersection with $(0,0,0)$. Since those lines are parameterized by tuples (x_1,x_2,x_3), we compute as follows:

$$\begin{aligned}
[0:0:1] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(0,0,1),(0,0,2)\}\\
[0:0:2] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(0,0,2),(0,0,1)\} = [0:0:1]\\
[0:1:0] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(0,1,0),(0,2,0)\}\\
[0:1:1] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(0,1,1),(0,2,2)\}\\
[0:1:2] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(0,1,2),(0,2,1)\}\\
[0:2:0] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(0,2,0),(0,1,0)\} = [0:1:0]\\
[0:2:1] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(0,2,1),(0,1,2)\} = [0:1:2]\\
[0:2:2] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(0,2,2),(0,1,1)\} = [0:1:1]\\
[1:0:0] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(1,0,0),(2,0,0)\}\\
[1:0:1] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(1,0,1),(2,0,2)\}\\
[1:0:2] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(1,0,2),(2,0,1)\}\\
[1:1:0] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(1,1,0),(2,2,0)\}\\
[1:1:1] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(1,1,1),(2,2,2)\}\\
[1:1:2] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(1,1,2),(2,2,1)\}\\
[1:2:0] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(1,2,0),(2,1,0)\}\\
[1:2:1] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(1,2,1),(2,1,2)\}\\
[1:2:2] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(1,2,2),(2,1,1)\}\\
[2:0:0] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(2,0,0),(1,0,0)\} = [1:0:0]\\
[2:0:1] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(2,0,1),(1,0,2)\} = [1:0:2]\\
[2:0:2] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(2,0,2),(1,0,1)\} = [1:0:1]\\
[2:1:0] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(2,1,0),(1,2,0)\} = [1:2:0]\\
[2:1:1] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(2,1,1),(1,2,2)\} = [1:2:2]\\
[2:1:2] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(2,1,2),(1,2,1)\} = [1:2:1]\\
[2:2:0] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(2,2,0),(1,1,0)\} = [1:1:0]\\
[2:2:1] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(2,2,1),(1,1,2)\} = [1:1:2]\\
[2:2:2] &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k\in\mathbb{F}_3^*\} = \{(2,2,2),(1,1,1)\} = [1:1:1]
\end{aligned}$$

These lines define the 13 points in the projective plane $\mathbb{F}_3\mathbb{P}$:

$$\begin{aligned}
\mathbb{F}_3\mathbb{P} = \{&[0:0:1],[0:1:0],[0:1:1],[0:1:2],[1:0:0],[1:0:1],\\
&[1:0:2],[1:1:0],[1:1:1],[1:1:2],[1:2:0],[1:2:1],[1:2:2]\}
\end{aligned}$$

This projective plane contains 9 affine points, three points at infinity and one line at infinity.

To understand the ambiguity in projective coordinates a bit better, let us consider the point $[1:2:2]$. As this point in the projective plane is a line in $\mathbb{F}_3^3\backslash\{(0,0,0)\}$, it has the projective coordinates $(1,2,2)$ as well as $(2,1,1)$, since the former coordinate gives the latter when multiplied in $\mathbb{F}_3$ by the factor 2. In addition, note that, for the same reasons, the points $[1:2:2]$ and $[2:1:1]$ are the same, since their underlying sets are equal.

Exercise 56. Construct the so-called **Fano plane**, that is, the projective plane over the finite field $\mathbb{F}_2$.

Bibliography

Jens Groth. On the size of pairing-based non-interactive arguments. *IACR Cryptol. ePrint Arch.*, 2016:260, 2016. URL `http://eprint.iacr.org/2016/260`.

Hongxi Wu. *Understanding numbers in elementary school mathematics*. American Mathematical Society, Providence, RI, 2011. ISBN 9780821852606.

Maurice Mignotte. *Mathematics for Computer Algebra*. 01 1992. ISBN 978-3-540-97675-2. doi: 10.1007/978-1-4613-9171-5.

G.H. Hardy, E.M. Wright, D.R. Heath-Brown, R. Heath-Brown, J. Silverman, and A. Wiles. *An Introduction to the Theory of Numbers*. Oxford mathematics. OUP Oxford, 2008. ISBN 9780199219865. URL `https://books.google.de/books?id=P6uTBqOa3T4C`.

B. Fine and G. Rosenberger. *Number Theory: An Introduction via the Density of Primes*. Springer International Publishing, 2016. ISBN 9783319438733. URL `https://books.google.de/books?id=-UaWDAEACAAJ`.

P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700.

Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer Publishing Company, Incorporated, 2010. ISBN 3642081428.

Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications / Rudolf Lidl, Harald Niederreiter*. Cambridge University Press Cambridge [Cambridgeshire] ; New York, 1986. ISBN 0521307066.

László Fuchs. *Abelian groups*. Springer Monogr. Math. Cham: Springer, 2015. ISBN 978-3-319-19421-9; 978-3-319-19422-6. doi: 10.1007/978-3-319-19422-6.

Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007. ISBN 978-1-58488-551-1.

Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-46766-3. URL `https://fmouhart.epheme.re/Crypto-1617/TD08.pdf`.

G. Ellis and L.D.M.G. Ellis. *Rings and Fields*. Oxford science publications. Clarendon Press, 1992. ISBN 9780198534556. URL `https://books.google.de/books?id=gDaKGfDMA1wC`.

J.H. Silverman and J.T. Tate. *Rational Points on Elliptic Curves*. Undergraduate Texts in Mathematics. Springer New York, 1994. ISBN 9780387978253. URL `https://books.google.de/books?id=mAJei2-JcE4C`.

J. Hoffstein, J. Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography*. Undergraduate Texts in Mathematics. Springer New York, 2008. ISBN 9780387779942. URL `https://books.google.de/books?id=z2SBIhmqMBMC`.

E. A. Grechnikov. Method for constructing elliptic curves using complex multiplication and its optimizations. 2012. doi: 10.48550/ARXIV.1207.6983. URL `https://arxiv.org/abs/1207.6983`.

David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *J. Cryptol.*, 23(2):224–280, 2010. URL `http://dblp.uni-trier.de/db/journals/joc/joc23.html#FreemanST10`.

R.N. Moll, J. Pustejovsky, M.A. Arbib, and A.J. Kfoury. *An Introduction to Formal Language Theory*. Monographs in Computer Science. Springer New York, 2012. ISBN 9781461395959. URL `https://books.google.de/books?id=tprhBwAAQBAJ`.

Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. Cryptology ePrint Archive, Paper 2013/507, 2013. URL `https://eprint.iacr.org/2013/507`. `https://eprint.iacr.org/2013/507`.